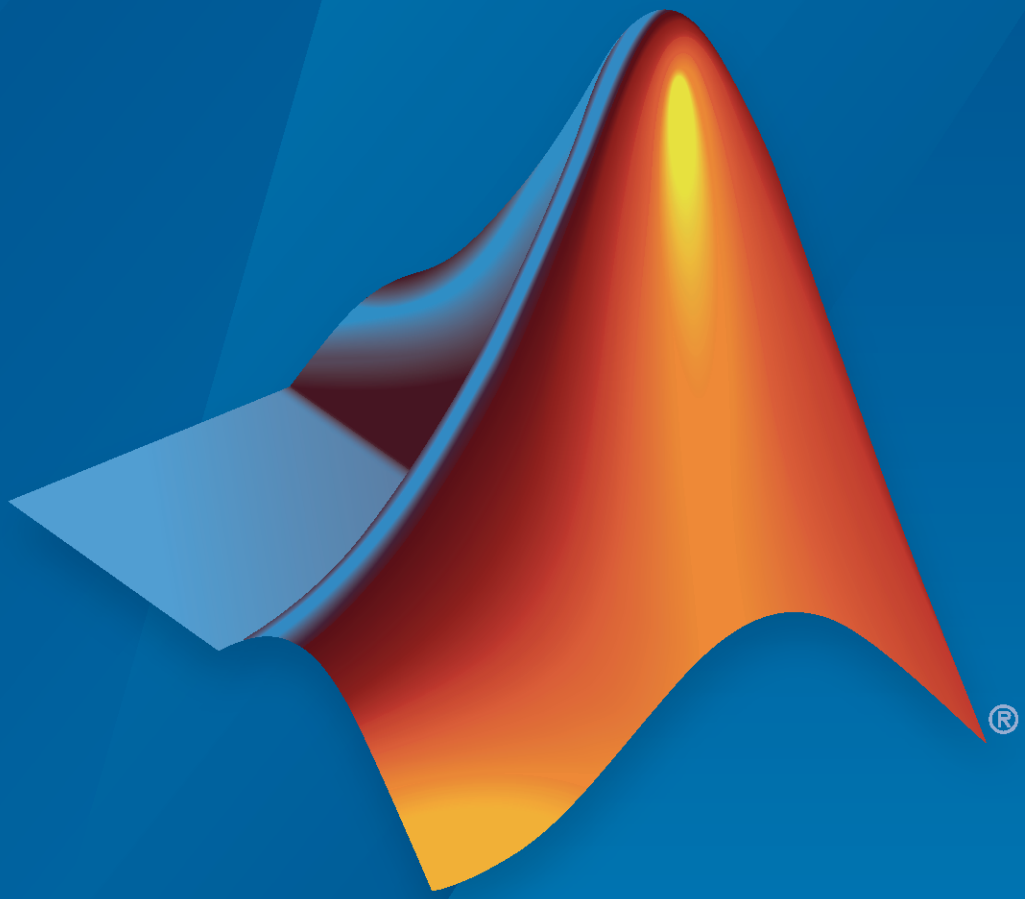
Signal Processing Toolbox™

Reference



MATLAB®

R2022a

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

Signal Processing Toolbox™ Reference

© COPYRIGHT 1988–2022 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

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)
September 2015	Online only	Revised for Version 7.1 (Release 2015b)
March 2016	Online only	Revised for Version 7.2 (Release 2016a)
September 2016	Online only	Revised for Version 7.3 (Release 2016b)
March 2017	Online only	Revised for Version 7.4 (Release 2017a)
September 2017	Online only	Revised for Version 7.5 (Release 2017b)
March 2018	Online only	Revised for Version 8.0 (Release 2018a)
September 2018	Online only	Revised for Version 8.1 (Release 2018b)
March 2019	Online only	Revised for Version 8.2 (Release 2019a)
September 2019	Online only	Revised for Version 8.3 (Release 2019b)
March 2020	Online only	Revised for Version 8.4 (Release 2020a)
September 2020	Online only	Revised for Version 8.5 (Release 2020b)
March 2021	Online only	Revised for Version 8.6 (Release 2021a)
September 2021	Online only	Revised for Version 8.7 (Release 2021b)
March 2022	Online only	Revised for Version 9.0 (Release 2022a)

Contents

Functions

1

Functions

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 = 1×6

    1.0000    0.6147    0.9898    0.0004    0.0034   -0.0077


efinal = 0.1791
```

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`

Introduced before R2006a

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`.

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` | `poly2rc` | `rc2ac`

Introduced before R2006a

alignsignals

Align two signals by delaying earliest signal

Syntax

```
[Xa,Ya] = alignsignals(X,Y)
[Xa,Ya] = alignsignals(X,Y,maxlag)
[Xa,Ya] = alignsignals(X,Y,maxlag,'truncate')
[Xa,Ya,D] = alignsignals( ___ )
```

Description

`[Xa,Ya] = alignsignals(X,Y)` estimates the delay, *D*, between the two input signals, `X` and `Y`, and returns the aligned signals, `Xa` and `Ya`.

- If `Y` is delayed with respect to `X`, then *D* is positive and `X` is delayed by *D* samples.
- If `Y` is advanced with respect to `X`, then *D* is negative and `Y` is delayed by –*D* samples.

Delays in `X` or `Y` can be introduced by prepending zeros.

`[Xa,Ya] = alignsignals(X,Y,maxlag)` uses `maxlag` as the maximum window size to find the estimated delay, *D*, between the two input signals, `X` and `Y`. It returns the aligned signals, `Xa` and `Ya`.

`[Xa,Ya] = alignsignals(X,Y,maxlag,'truncate')` keeps the lengths of the aligned signals, `Xa` and `Ya`, the same as those of the input signals, `X` and `Y`, respectively.

- If the estimated delay, *D*, is positive, then *D* zeros are prepended to `X` and the last *D* samples of `X` are truncated.
- If the estimated delay, *D*, is negative, then –*D* zeros are prepended to `Y` and the last –*D* samples of `Y` are truncated.

Notes `X` and `Y` are row or column vectors of length L_X and L_Y, respectively.

- If $D \geq L_X$, then `Xa` consists of L_X zeros. All samples of `X` are lost.
- If $-D \geq L_Y$, then `Ya` consists of L_Y zeros. All samples of `Y` are lost.

To avoid assigning a specific value to `maxlag` when using the `'truncate'` option, set `maxlag` to `[]`.

`[Xa,Ya,D] = alignsignals( ___ )` returns the estimated delay, `D`. This syntax can include any of the input arguments used in previous syntaxes.

Examples

Align Two Signals Where the First Signal Lags by Three Samples

Align signal `Y` with respect to `X` by delaying it three samples.

Create two signals, X and Y. X is exactly the same as Y, except X has three leading zeros and one additional following zero. Align the two signals.

```
X = [0 0 0 1 2 3 0 0];
Y = [1 2 3 0];

[Xa,Ya] = alignsignals(X,Y)
```

```
Xa = 1×8

     0     0     0     1     2     3     0     0


Ya = 1×7

     0     0     0     1     2     3     0
```

Align Two Signals Where the Second Signal Lags by Two Samples

Align signal X when Y is delayed with respect to X by two samples.

Create two signals, X and Y. Y is exactly the same as X, except Y has two leading zeros. Align the two signals.

```
X = [1 2 3];
Y = [0 0 1 2 3];
maxlag = 2;

[Xa,Ya,D] = alignsignals(X,Y,maxlag)
```

```
Xa = 1×5

     0     0     1     2     3


Ya = 1×5

     0     0     1     2     3


D = 2
```

Align Two Signals Where the Second Signal Is Noisy

Align signal Y with respect to X, despite the fact that Y is a noisy signal.

Create two signals, X and Y. Y is exactly the same as X with some noise added to it. Align the two signals.

```
X = [0 0 1 2 3 0];
Y = [0.02 0.12 1.08 2.21 2.95 -0.09];

[Xa,Ya,D] = alignsignals(X,Y)
```

```
Xa = 1×6

     0     0     1     2     3     0

Ya = 1×6

    0.0200    0.1200    1.0800    2.2100    2.9500   -0.0900

D = 0
```

You do not need to change the input signals to produce the output signals. The delay `D` is zero.

Align Two Signals Using the 'truncate' Option

Invoke the `'truncate'` option when calling the `alignsignals` function.

Create two signals, `X` and `Y`. `Y` is exactly the same as `X`, except `Y` has two leading zeros. Align the two signals, applying the `'truncate'` directive.

```
X = [1 2 3];
Y = [0 0 1 2 3];

[Xa,Ya,D] = alignsignals(X,Y,[],'truncate')
```

```
Xa = 1×3

     0     0     1

Ya = 1×5

     0     0     1     2     3

D = 2
```

Observe that the output signal `Xa` has a length of 3, the same length as input signal `X`.

In the case where using the `'truncate'` option ends up truncating all the original data of `X`, a warning is issued. To make `alignsignals` issue such a warning, run the following example.

```
Y = [0 0 0 0 1 2 3];

[Xa,Ya,D] = alignsignals(X,Y,[],'truncate')
```

```
Warning: All original data in the first input X has been truncated because the length of X is sma

Xa = 1×3

     0     0     0

Ya = 1×7
```

```
     0     0     0     0     1     2     3
```

```
D = 4
```

Align a Signal and a Periodic Repetition of It

Align signal Y with respect to X, despite the fact that Y is a periodic repetition of X. Return the smallest possible delay.

Create two signals, X and Y. Y consists of two copies of the nonzero portion of X separated by zeros. Align the two signals.

```
X = [0 1 2 3];
Y = [1 2 3 0 0 0 0 1 2 3 0 0];

[Xa,Ya,D] = alignsignals(X,Y)
```

```
Xa = 1×4

     0     1     2     3


Ya = 1×13

     0     1     2     3     0     0     0     0     1     2     3     0     0


D = -1
```

Input Arguments

X — First input signal

vector of numeric values

First input signal, specified as a numeric vector of length *LX*.

Example: `[1 2 3]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

Y — Second input signal

vector of numeric values

Second input signal, specified as a numeric vector of length *LY*.

Example: `[0 0 1 2 3]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

`maxlag` — Maximum window size or lag

scalar integer | `[]`

Maximum window size, or lag, specified as an integer-valued scalar. By default, `maxlag` is equal to `max(length(X),length(Y))-1`. If `maxlag` is input as `[]`, it is replaced by the default value. If `maxlag` is negative, it is replaced by its absolute value. If `maxlag` is not integer-valued, or is complex, `Inf`, or `NaN`, then `alignsignals` returns an error.

Example: 2

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`Xa` — Aligned first signal

vector of numeric values

Aligned first signal, returned as a numeric vector that is aligned with the second output argument, `Ya`. If input argument `X` is a row vector, then `Xa` is also a row vector. If input argument `X` is a column vector, then `Xa` is also a column vector. If you specify the `'truncate'` option and the estimated delay *D* is positive, then `Xa` is equivalent to the input signal `X` with *D* zeros prepended to it and its last *D* samples truncated.

`Ya` — Aligned second signal

vector of numeric values

Aligned second signal, returned as a numeric vector that is aligned with the first output argument, `Xa`. If input argument `Y` is a row vector, then `Ya` is also a row vector. If input argument `Y` is a column vector, then `Ya` is also a column vector. If you specify the `'truncate'` option and the estimated delay *D* is negative, then `Ya` is equivalent to the input signal `Y` with –*D* zeros prepended to it and its last –*D* samples truncated.

`D` — Estimated delay between input signals

scalar integer

Estimated delay between input signals, returned as a scalar integer. This integer represents the number of samples by which the two input signals, `X` and `Y` are offset.

- If `Y` is delayed with respect to `X`, then `D` is positive and `X` is delayed by `D` samples.
- If `Y` is advanced with respect to `X`, then `D` is negative and `Y` is delayed by –`D` samples.
- If `X` and `Y` are already aligned, then `D` is zero and neither `X` nor `Y` are delayed.

If you specify a value for the input argument `maxlag`, then `D` must be less than or equal to `maxlag`.

Algorithms

- You can find the theory on delay estimation in the specification of the `finddelay` function (see “Algorithms” on page 1-775).
- The `alignsignals` function uses the estimated delay *D* to delay the earliest signal such that the two signals have the same starting point.
- As specified for the `finddelay` function, the pair of signals need not be exact delayed copies of each other. However, the signals can be successfully aligned only if there is sufficient correlation between them. For more information on estimating covariance and correlation functions, see [1].
- If your signals have features such as pulses or transitions, you can align them more effectively using measurement functions instead of correlation. For an example, see “Align Two Bilevel Waveforms” on page 1-1906.

References

[1] Orfanidis, Sophocles J. *Optimum Signal Processing. An Introduction*. 2nd Ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`dtw` | `edr` | `finddelay` | `findsignal` | `xcorr`

arburg

Autoregressive all-pole model parameters — Burg’s method

Syntax

```
a = 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`.

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

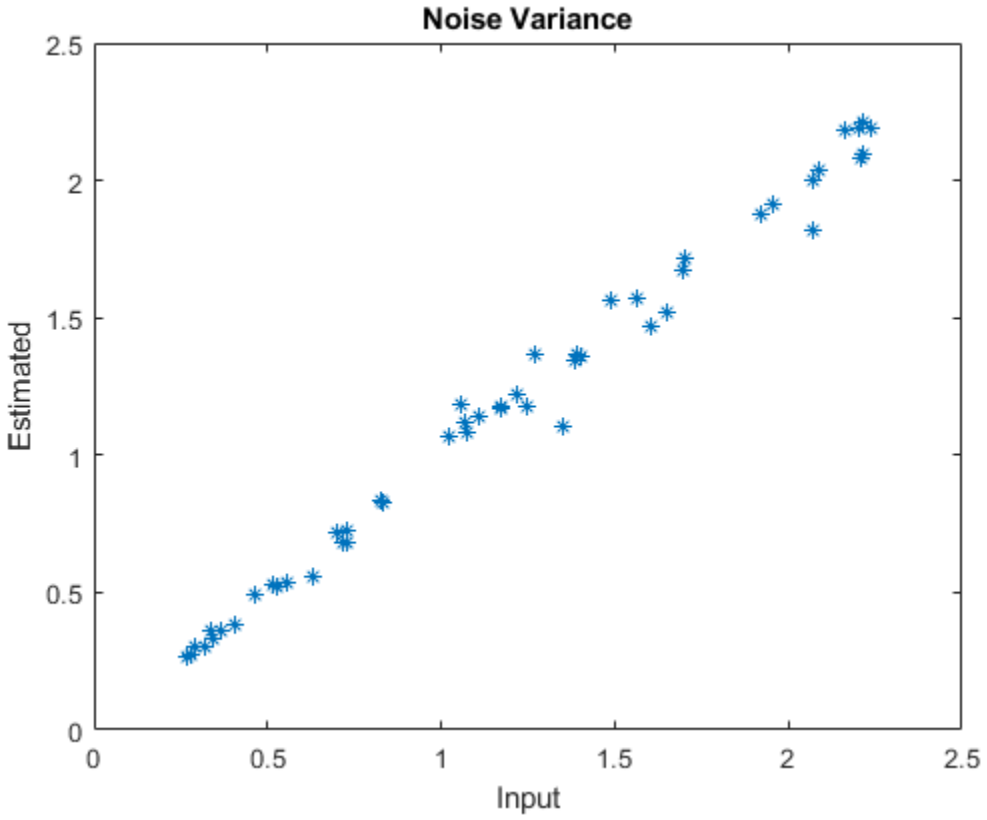
```
rng default

A = [1 -2.7607 3.8106 -2.6535 0.9238];

y = filter(1,A,0.2*randn(1024,1));

arcoeffs = arburg(y,4)
```

```
arcoeffs = 1×5

    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.

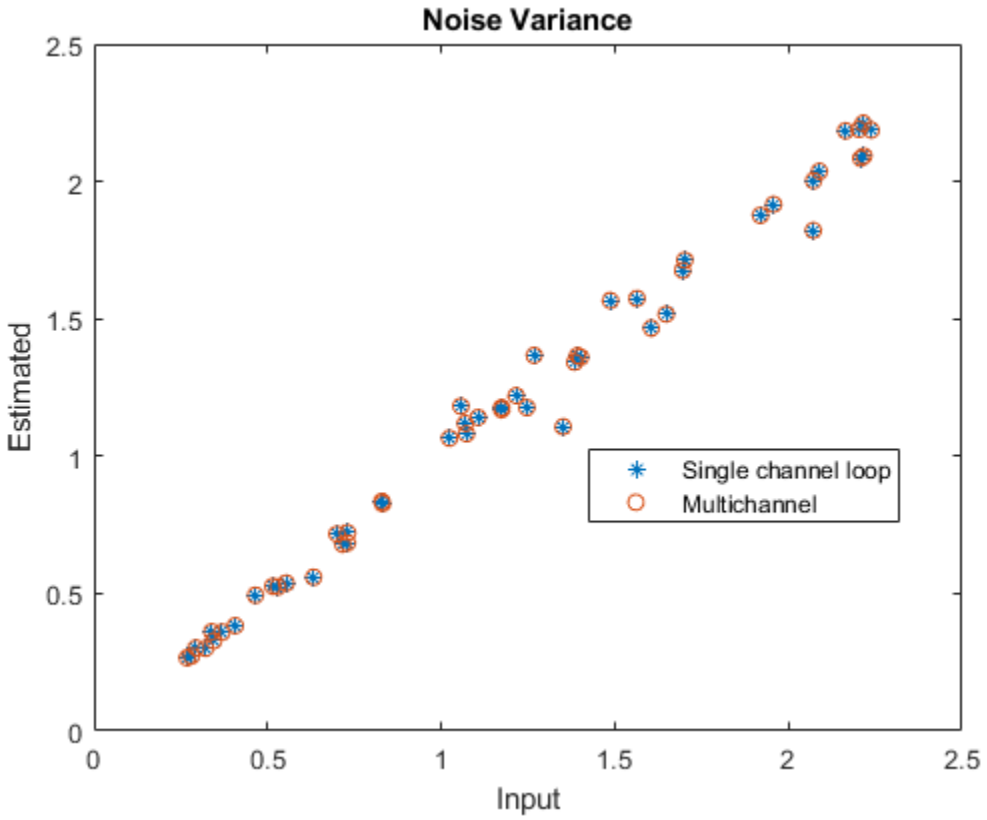
```
nrealiz = 50;

noisestdz = rand(1,nrealiz)+0.5;

randnoise = randn(1024,nrealiz);
noisevar = zeros(1,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 the function's multichannel syntax.

```
Y = filter(1,A,noisestdz.*randnoise);

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

hold on
plot(noisestdz.^2,variances,'o')
hold off
legend('Single channel loop','Multichannel','Location','best')
```



Input Arguments

x — Input array

vector | matrix

Input array, specified as a vector or matrix.

Example: `filter(1,[1 -0.75 0.5],0.2*randn(1024,1))` specifies a second-order autoregressive process.

Data Types: `single` | `double`
Complex Number Support: Yes

p — Model order

positive integer scalar

Model order, specified as a positive integer scalar. `p` must be less than the number of elements or rows of `x`.

Data Types: `single` | `double`

Output Arguments

a — Normalized autoregressive parameters

row vector | matrix

Normalized autoregressive parameters, returned as a vector or matrix. If `x` is a matrix, then each row of `a` corresponds to a column of `x`. `a` has `p` + 1 columns and contains the AR system parameters, $A(z)$, in descending powers of z.

`e` — White noise input variance
scalar | row vector

White noise input variance, returned as a scalar or row vector. If `x` is a matrix, then each element of `e` corresponds to a column of `x`.

`rc` — Reflection coefficients
column vector | matrix

Reflection coefficients, returned as a column vector or matrix. If `x` is a matrix, then each column of `rc` corresponds to a column of `x`. `rc` has `p` rows.

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 squared 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

Burg's 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].

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`arcov` | `armcov` | `aryule` | `levinson` | `lpc`

Topics

“Parametric Modeling”

Introduced before R2006a

arcov

Autoregressive all-pole model parameters — covariance method

Syntax

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

Description

`a = arcov(x,p)` returns the normalized autoregressive (AR) parameters corresponding to a model of order `p` for the input array `x`, where `x` 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.

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

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.

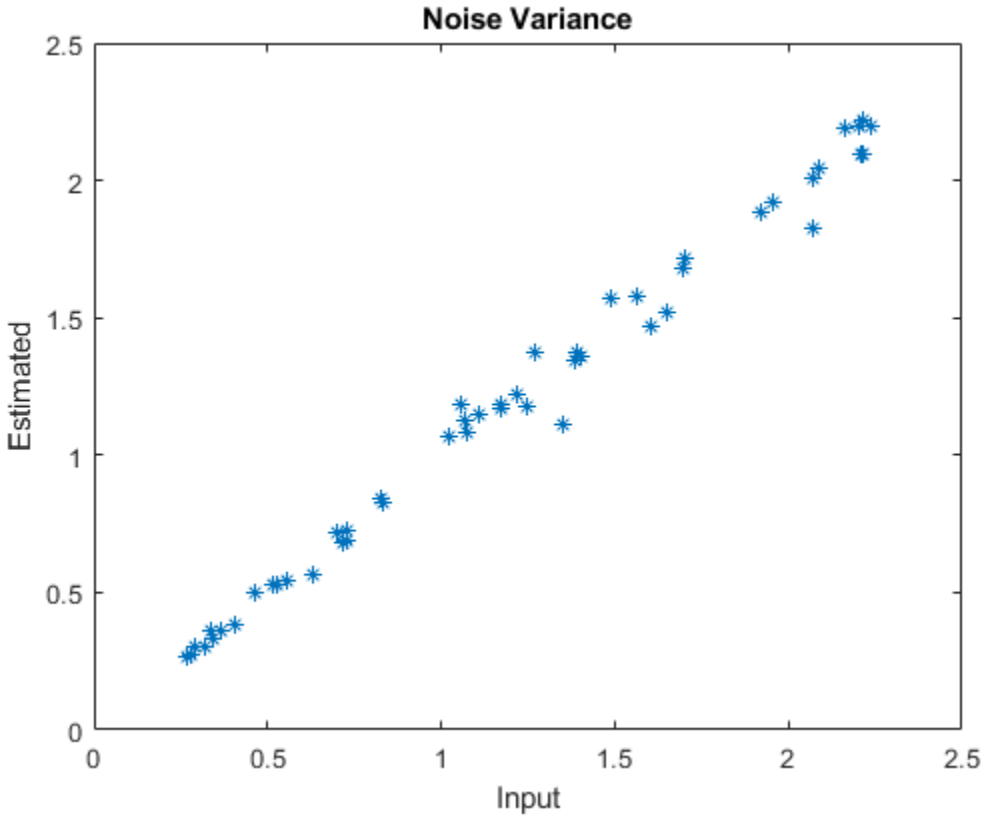
```
rng default

A = [1 -2.7607 3.8106 -2.6535 0.9238];

y = filter(1,A,0.2*randn(1024,1));

arcoeffs = arcov(y,4)
```

```
arcoeffs = 1×5

    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.

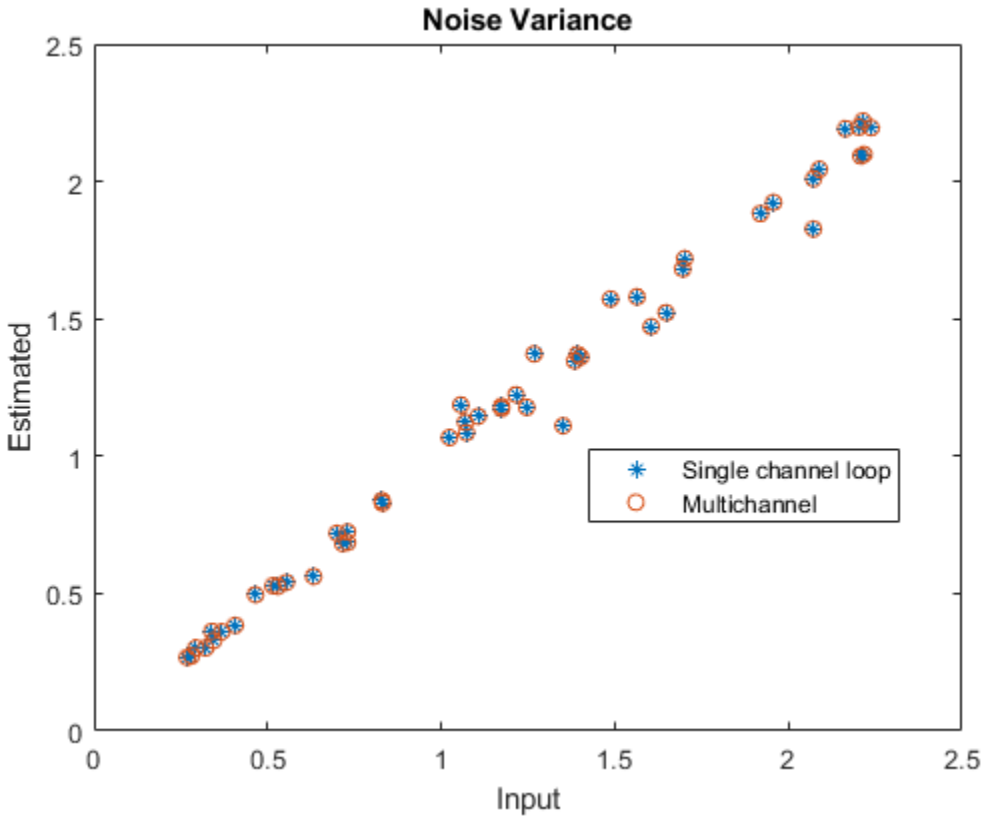
```
nrealiz = 50;

noisestdz = rand(1,nrealiz)+0.5;

randnoise = randn(1024,nrealiz);
noisevar = zeros(1,nrealiz);

for k = 1:nrealiz
    y = filter(1,A,noisestdz(k) * randnoise(:,k));
    [arcoeffs,noisevar(k)] = arcov(y,4);
end
```

```
plot(noisestdz.^2,noisevar,'*')
title('Noise Variance')
xlabel('Input')
ylabel('Estimated')
```



Repeat the procedure using the function's multichannel syntax.

```
Y = filter(1,A,noisestdz.*randnoise);

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

hold on
plot(noisestdz.^2,variances,'o')
hold off
legend('Single channel loop','Multichannel','Location',"best")
```



Input Arguments

x — Input array

vector | matrix

Input array, specified as a vector or matrix.

Example: `filter(1,[1 -0.75 0.5],0.2*randn(1024,1))` specifies a second-order autoregressive process.

Data Types: `single` | `double`
Complex Number Support: Yes

p — Model order

positive integer scalar

Model order, specified as a positive integer scalar. `p` must be less than the number of elements or rows of `x`.

Data Types: `single` | `double`

Output Arguments

a — Normalized autoregressive parameters

row vector | matrix

Normalized autoregressive parameters, returned as a vector or matrix. If `x` is a matrix, then each row of `a` corresponds to a column of `x`. `a` has `p` + 1 columns and contains the AR system parameters, $A(z)$, in descending powers of z.

e — White noise input variance
scalar | row vector

White noise input variance, returned as a scalar or row vector. If `x` is a matrix, then each element of `e` corresponds to a column of `x`.

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 squared prediction error of the autoregression.

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

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

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`arburg` | `armcov` | `aryule` | `lpc` | `pcov` | `prony`

Introduced before R2006a

armcov

Autoregressive all-pole model parameters — modified covariance method

Syntax

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

Description

`a = armcov(x,p)` returns the normalized autoregressive (AR) parameters corresponding to a model of order `p` for the input array `x`. `x` 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

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

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.

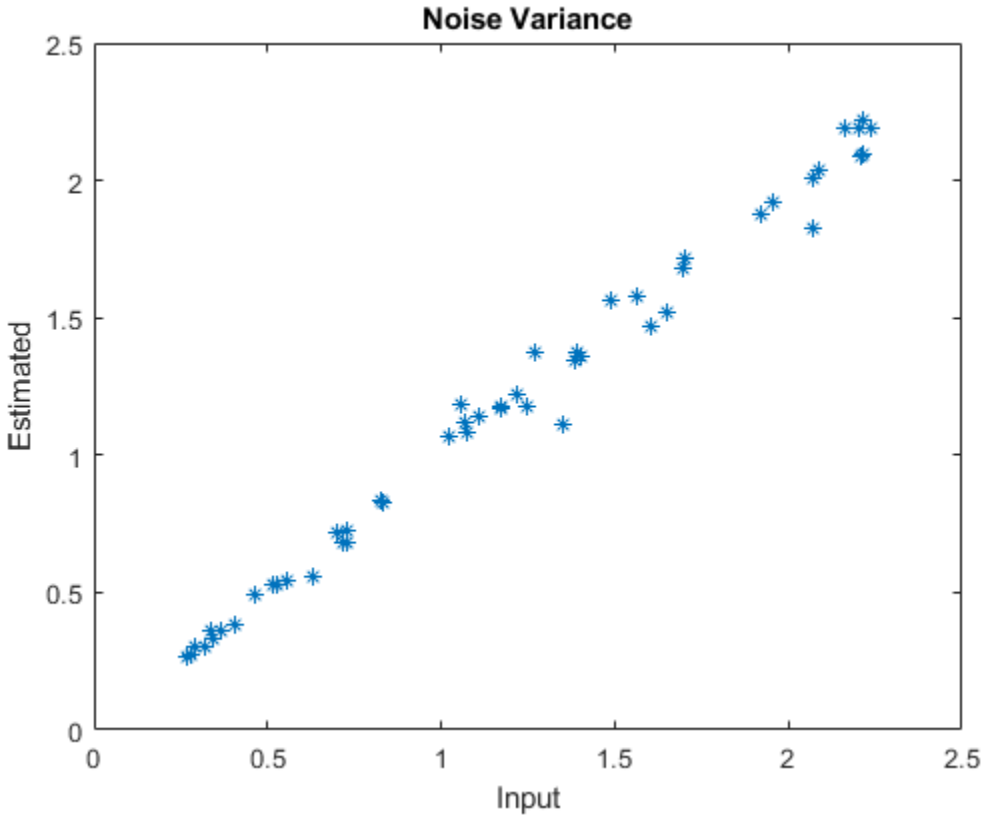
```
rng default

A = [1 -2.7607 3.8106 -2.6535 0.9238];

y = filter(1,A,0.2*randn(1024,1));

arcoeffs = armcov(y,4)
```

```
arcoeffs = 1×5

    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.

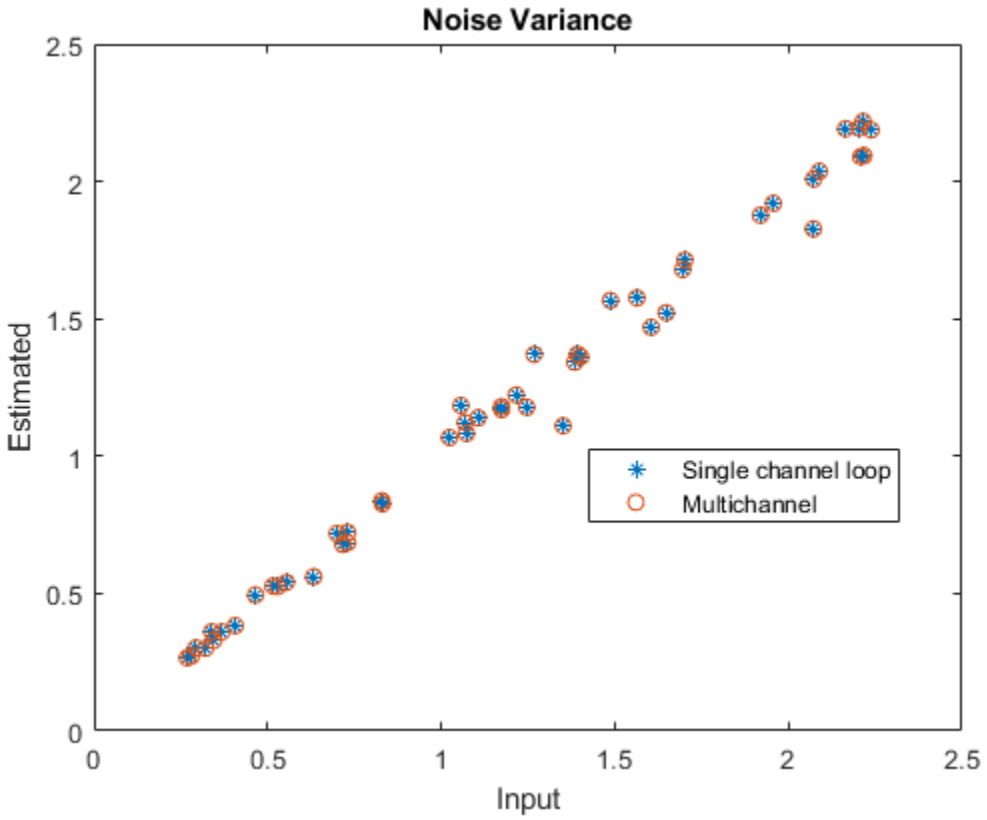
```
nrealiz = 50;

noisestdz = rand(1,nrealiz)+0.5;

randnoise = randn(1024,nrealiz);
noisevar = zeros(1,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 the function's multichannel syntax.

```
Y = filter(1,A,noisestdz.*randnoise);

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

hold on
plot(noisestdz.^2,variances,'o')
hold off
legend('Single channel loop','Multichannel','Location',"best")
```



Input Arguments

x — Input array

vector | matrix

Input array, specified as a vector or matrix.

Example: `filter(1,[1 -0.75 0.5],0.2*randn(1024,1))` specifies a second-order autoregressive process.

Data Types: `single` | `double`
Complex Number Support: Yes

p — Model order

positive integer scalar

Model order, specified as a positive integer scalar. `p` must be less than the number of elements or rows of `x`.

Data Types: `single` | `double`

Output Arguments

a — Normalized autoregressive parameters

row vector | matrix

Normalized autoregressive parameters, returned as a vector or matrix. If `x` is a matrix, then each row of `a` corresponds to a column of `x`. `a` has `p` + 1 columns and contains the AR system parameters, $A(z)$, in descending powers of z.

e — White noise input variance
scalar | row vector

White noise input variance, returned as a scalar or row vector. If `x` is a matrix, then each element of `e` corresponds to a column of `x`.

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 squared prediction error of the autoregression.

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

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

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`arburg` | `arcov` | `aryule` | `lpc` | `pmcov` | `prony`

Introduced before R2006a

aryule

Autoregressive all-pole model parameters — Yule-Walker method

Syntax

```
a = 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`.

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

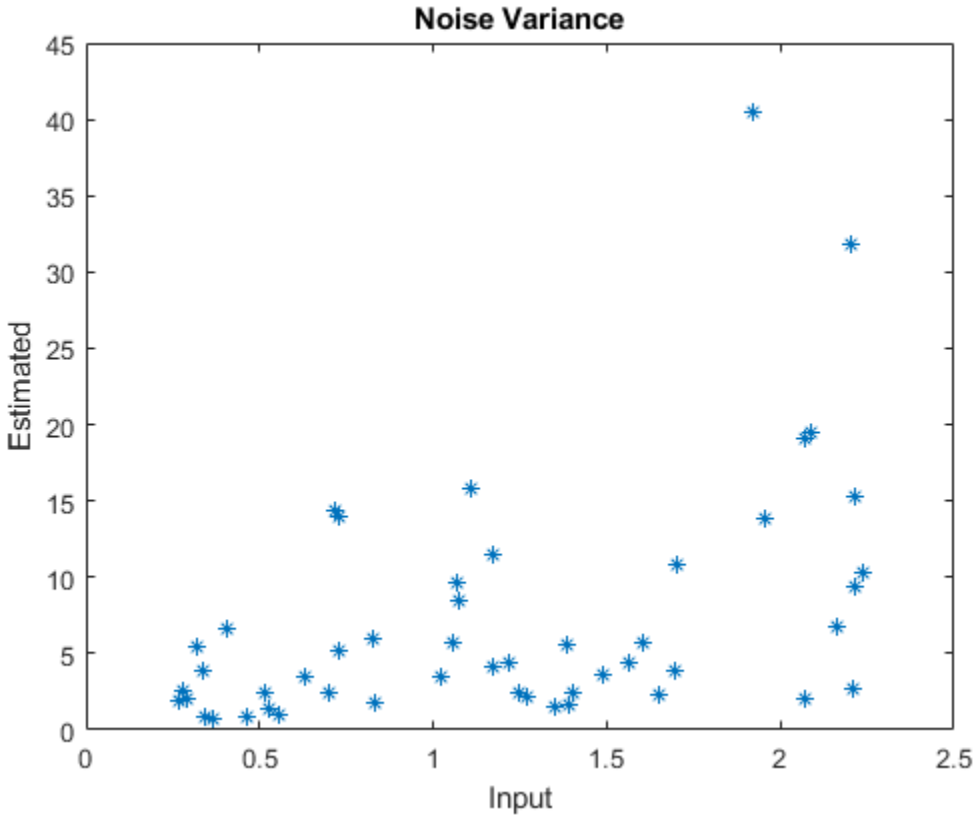
```
rng default

A = [1 -2.7607 3.8106 -2.6535 0.9238];

y = filter(1,A,0.2*randn(1024,1));

arcoeffs = aryule(y,4)
```

```
arcoeffs = 1×5

    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.

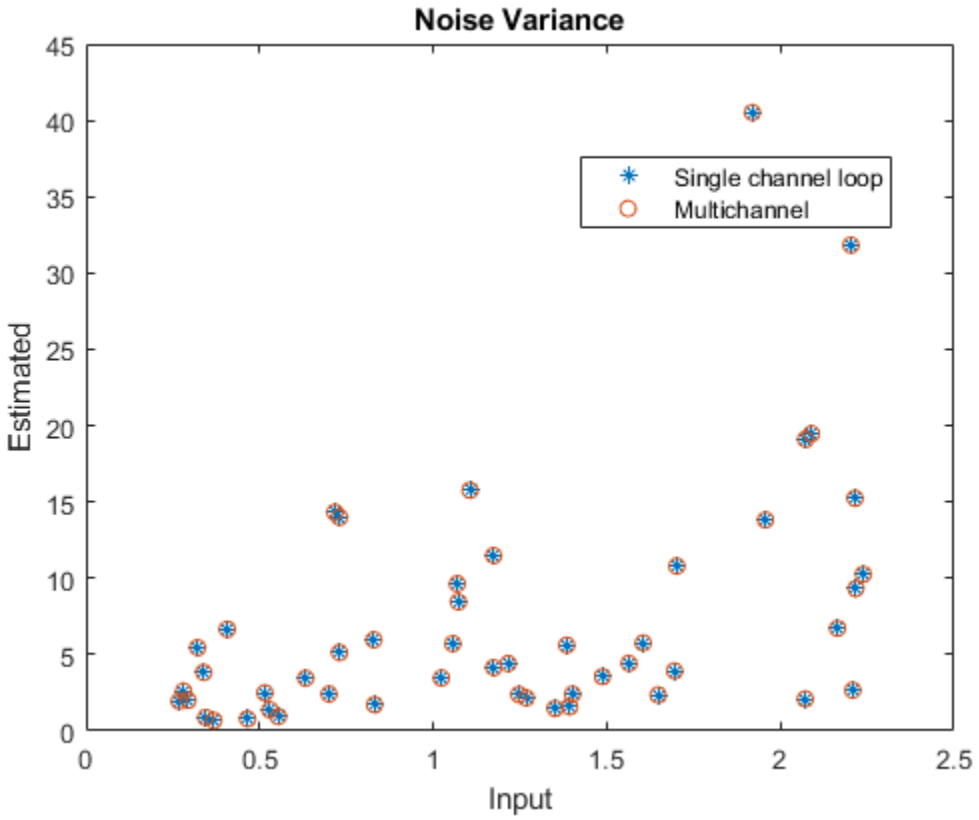
```
nrealiz = 50;

noisestdz = rand(1,nrealiz)+0.5;

randnoise = randn(1024,nrealiz);
noisevar = zeros(1,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 the function's multichannel syntax.

```
Y = filter(1,A,noisestdz.*randnoise);

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

hold on
plot(noisestdz.^2,variances,'o')
hold off
legend('Single channel loop','Multichannel','Location','best')
```



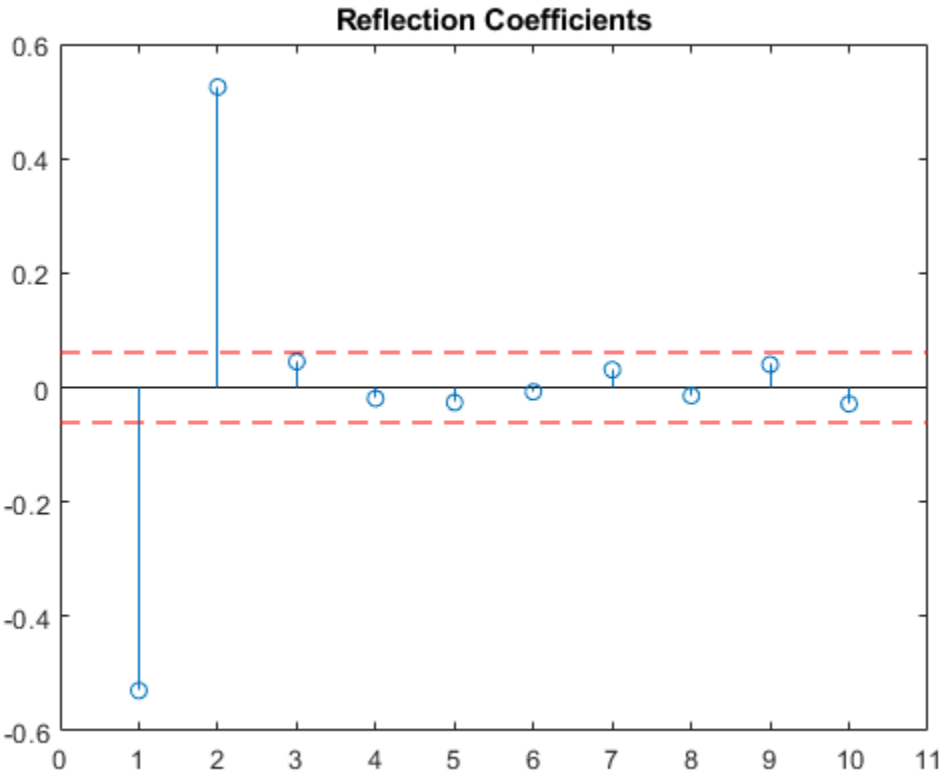
Estimate Model order Using Decay of Reflection Coefficients

Use a vector of polynomial coefficients to generate an AR(2) process by filtering 1024 samples of white noise. Reset the random number generator for reproducible results.

```
rng default

y = filter(1,[1 -0.75 0.5],0.2*randn(1024,1));
```

Use the Yule-Walker method to fit an AR(10) model to the process. Output and plot the reflection coefficients. Only the first two coefficients lie outside the 95% confidence bounds, indicating that an AR(10) model significantly overestimates the time dependence in the data. See “AR Order Selection with Partial Autocorrelation Sequence” for more details.

```
[ar,nvar,rc] = aryule(y,10);

stem(rc)
xlim([0 11])
conf95 = sqrt(2)*erfinv(0.95)/sqrt(1024);
[X,Y] = ndgrid(xlim,conf95*[-1 1]);
hold on
plot(X,Y,'--r')
hold off
title('Reflection Coefficients')
```



Input Arguments

x — Input array

vector | matrix

Input array, specified as a vector or matrix.

Example: `filter(1,[1 -0.75 0.5],0.2*randn(1024,1))` specifies a second-order autoregressive process.

Data Types: `single` | `double`
Complex Number Support: Yes

p — Model order

positive integer scalar

Model order, specified as a positive integer scalar. `p` must be less than the number of elements or rows of `x`.

Data Types: `single` | `double`

Output Arguments

a — Normalized autoregressive parameters

row vector | matrix

Normalized autoregressive parameters, returned as a vector or matrix. If `x` is a matrix, then each row of `a` corresponds to a column of `x`. `a` has `p` + 1 columns and contains the AR system parameters, $A(z)$, in descending powers of z.

`e` — White noise input variance
scalar | row vector

White noise input variance, returned as a scalar or row vector. If `x` is a matrix, then each element of `e` corresponds to a column of `x`.

`rc` — Reflection coefficients
column vector | matrix

Reflection coefficients, returned as a column vector or matrix. If `x` is a matrix, then each column of `rc` corresponds to a column of `x`. `rc` has `p` rows.

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 squared 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.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

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

Topics

“Parametric Modeling”

Introduced before R2006a

bandpass

Bandpass-filter signals

Syntax

```
y = bandpass(x,wpass)
y = bandpass(x,fpass,fs)
y = bandpass(xt,fpass)

y = bandpass( ___ ,Name,Value)

[y,d] = bandpass( ___ )

bandpass( ___ )
```

Description

`y = bandpass(x,wpass)` filters the input signal `x` using a bandpass filter with a passband frequency range specified by the two-element vector `wpass` and expressed in normalized units of π rad/sample. `bandpass` uses a minimum-order filter with a stopband attenuation of 60 dB and compensates for the delay introduced by the filter. If `x` is a matrix, the function filters each column independently.

`y = bandpass(x,fpass,fs)` specifies that `x` has been sampled at a rate of `fs` hertz. The two-element vector `fpass` specifies the passband frequency range of the filter in hertz.

`y = bandpass(xt,fpass)` bandpass-filters the data in timetable `xt` using a filter with a passband frequency range specified in hertz by the two-element vector `fpass`. The function independently filters all variables in the timetable and all columns inside each variable.

`y = bandpass( ___ ,Name,Value)` specifies additional options for any of the previous syntaxes using name-value pair arguments. You can change the stopband attenuation, the “Bandpass Filter Steepness” on page 1-40, and the type of impulse response of the filter.

`[y,d] = bandpass( ___ )` also returns the `digitalFilter` object `d` used to filter the input.

`bandpass( ___ )` with no output arguments plots the input signal and overlays the filtered signal.

Examples

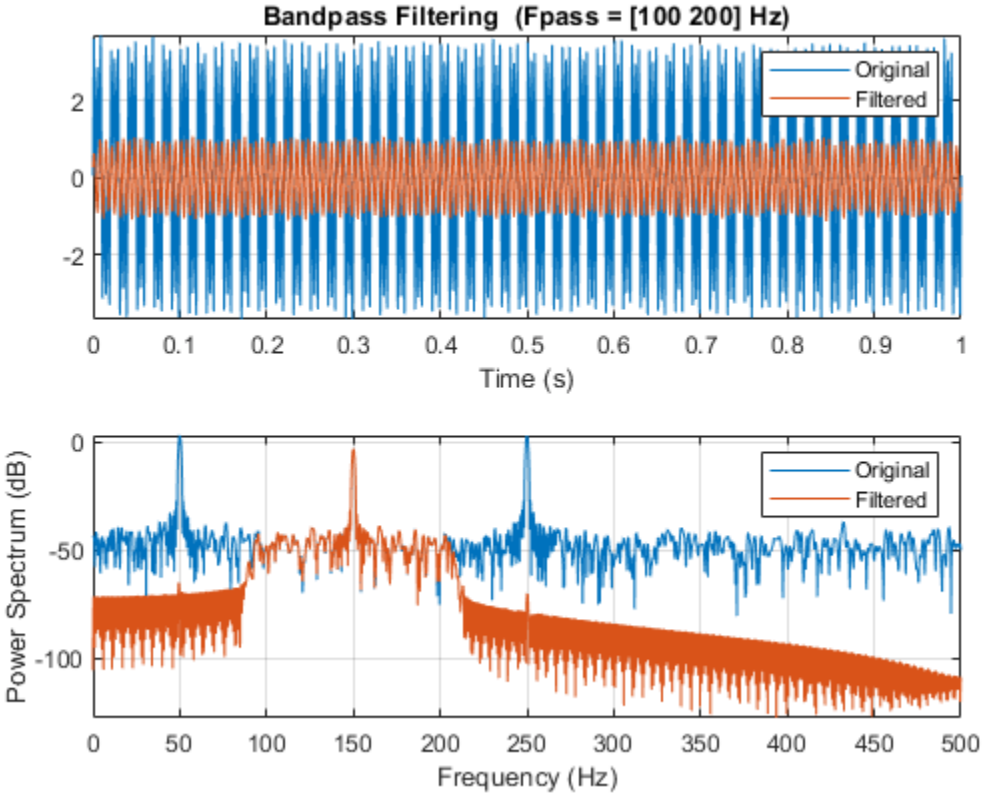
Bandpass Filtering of Tones

Create a signal sampled at 1 kHz for 1 second. The signal contains three tones, one at 50 Hz, another at 150 Hz, and a third at 250 Hz. The high-frequency and low-frequency tones both have twice the amplitude of the intermediate tone. The signal is embedded in Gaussian white noise of variance 1/100.

```
fs = 1e3;
t = 0:1/fs:1;
x = [2 1 2]*sin(2*pi*[50 150 250]'.*t) + randn(size(t))/10;
```

Bandpass-filter the signal to remove the low-frequency and high-frequency tones. Specify passband frequencies of 100 Hz and 200 Hz. Display the original and filtered signals, and also their spectra.

```
bandpass(x,[100 200],fs)
```

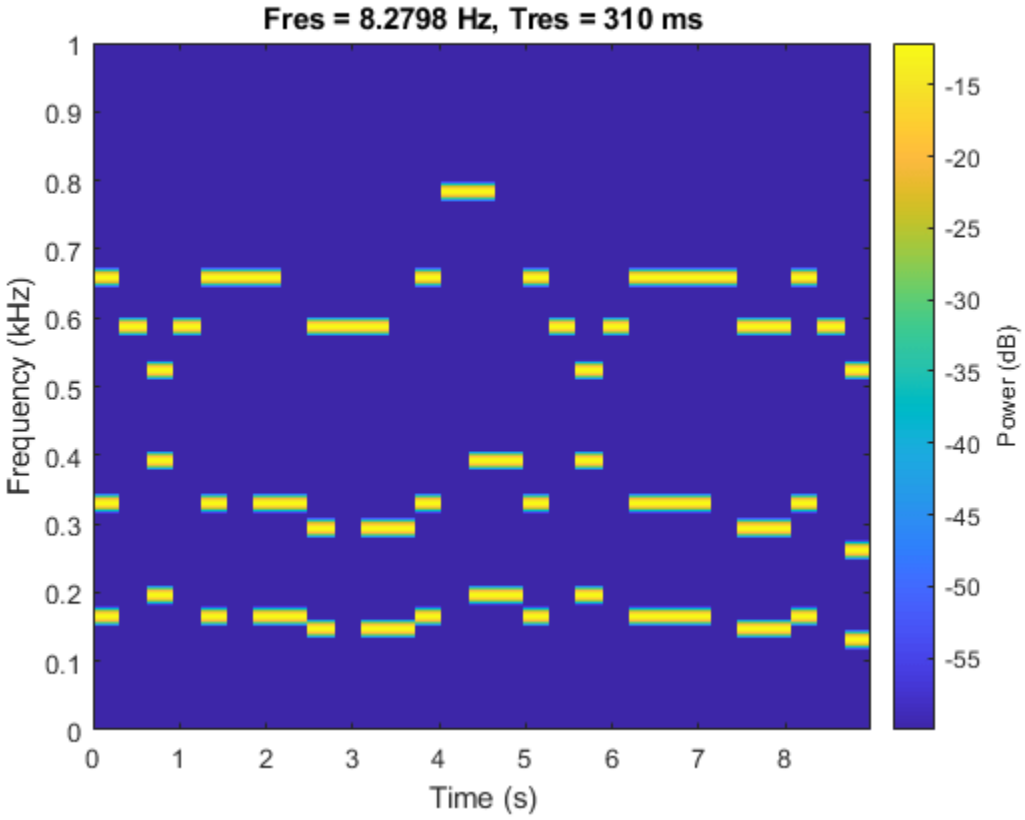


Bandpass Filtering of Musical Signal

Implement a basic digital music synthesizer and use it to play a traditional song. Specify a sample rate of 2 kHz. Plot the spectrogram of the song.

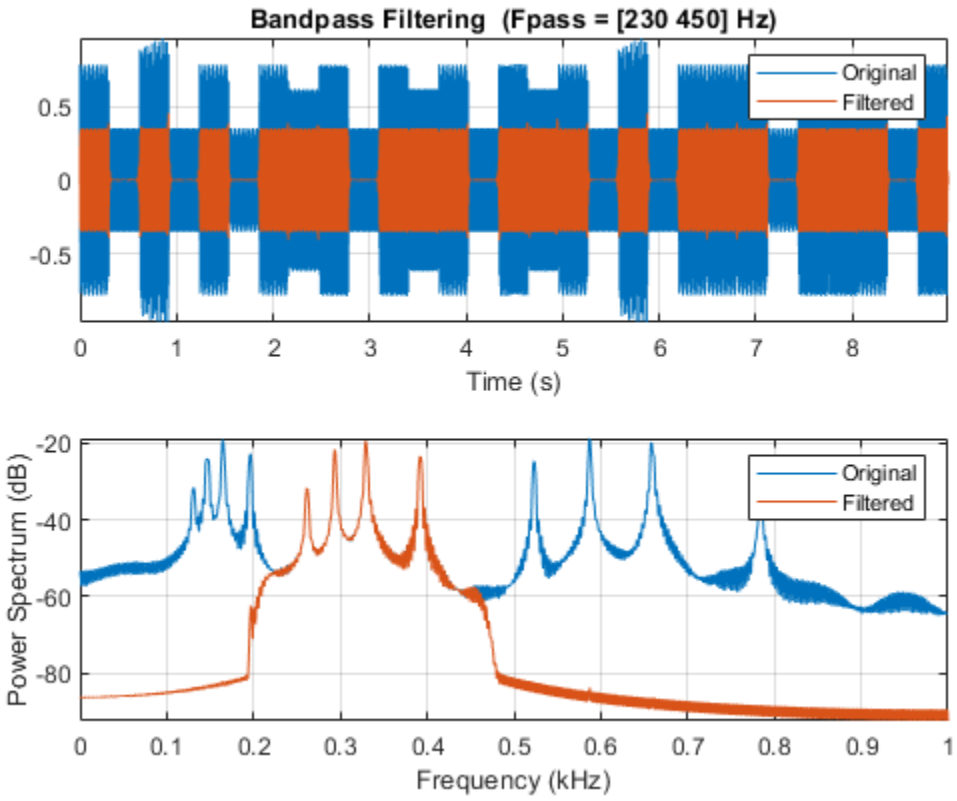
```
fs = 2e3;
t = 0:1/fs:0.3-1/fs;

l = [0 130.81 146.83 164.81 174.61 196.00 220 246.94];
m = [0 261.63 293.66 329.63 349.23 392.00 440 493.88];
h = [0 523.25 587.33 659.25 698.46 783.99 880 987.77];
note = @(f,g) [1 1 1]*sin(2*pi*[l(g) m(g) h(f)]'.*t);

mel = [3 2 1 2 3 3 3 0 2 2 2 0 3 5 5 0 3 2 1 2 3 3 3 3 2 2 3 2 1]+1;
acc = [3 0 5 0 3 0 3 3 2 0 2 2 3 0 5 5 3 0 5 0 3 3 3 0 2 2 3 0 1]+1;

song = [];
for kj = 1:length(mel)
    song = [song note(mel(kj),acc(kj)) zeros(1,0.01*fs)];
end
```

```
song = song/(max(abs(song))+0.1);

% To hear, type sound(song,fs)

pspectrum(song,fs,'spectrogram','TimeResolution',0.31, ...
    'OverlapPercent',0,'MinThreshold',-60)
```



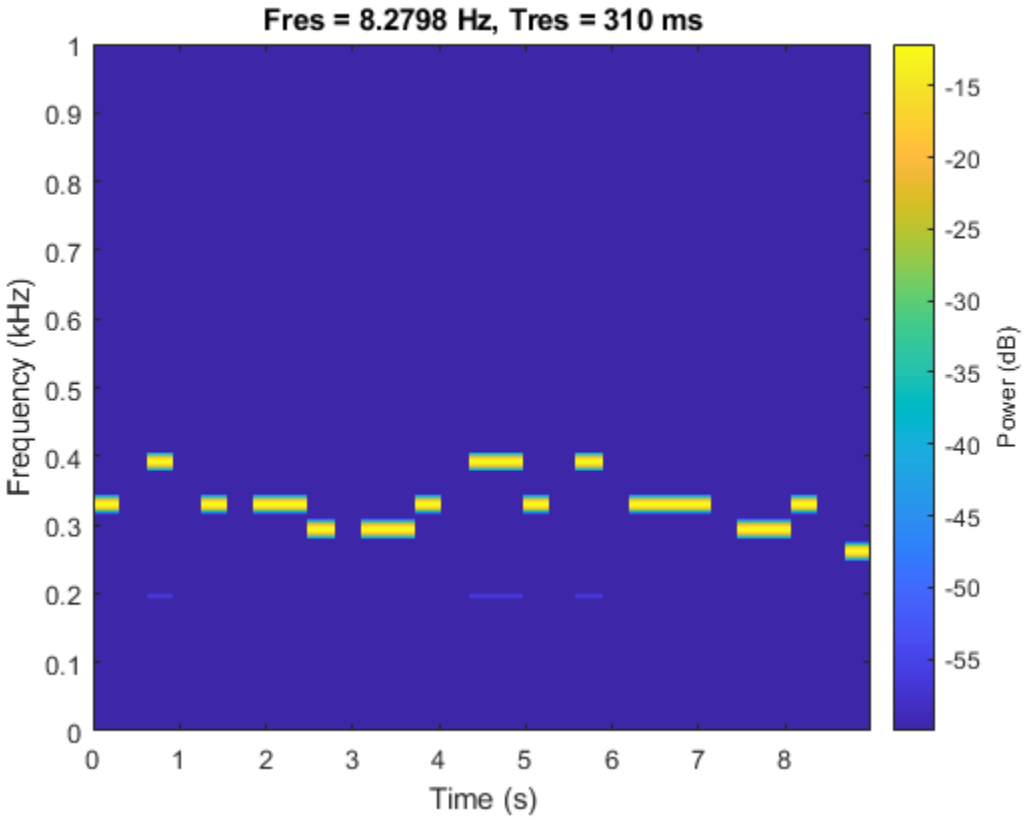
Bandpass-filter the signal to separate the middle register from the other two. Specify passband frequencies of 230 Hz and 450 Hz. Plot the original and filtered signals in the time and frequency domains.

```
pong = bandpass(song,[230 450],fs);

% To hear, type sound(pong,fs)

bandpass(song,[230 450],fs)
```



Plot the spectrogram of the middle register.

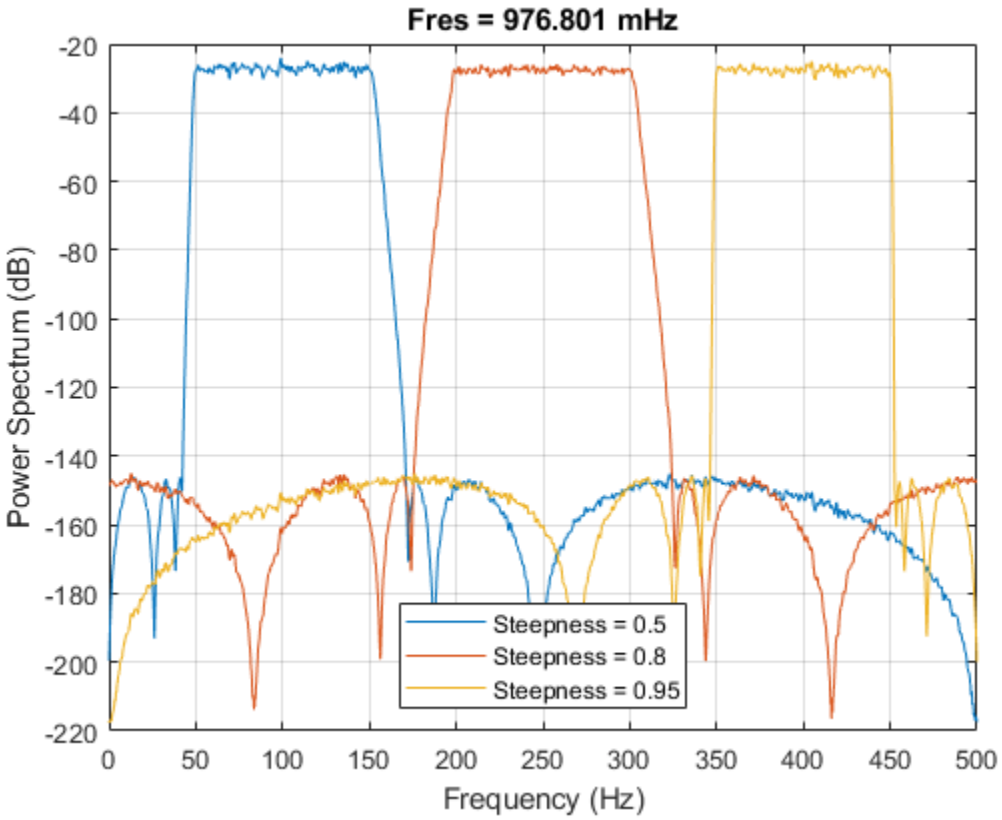
```
figure
pspectrum(pong,fs,'spectrogram','TimeResolution',0.31, ...
    'OverlapPercent',0,'MinThreshold',-60)
```



Bandpass Filter Steepness

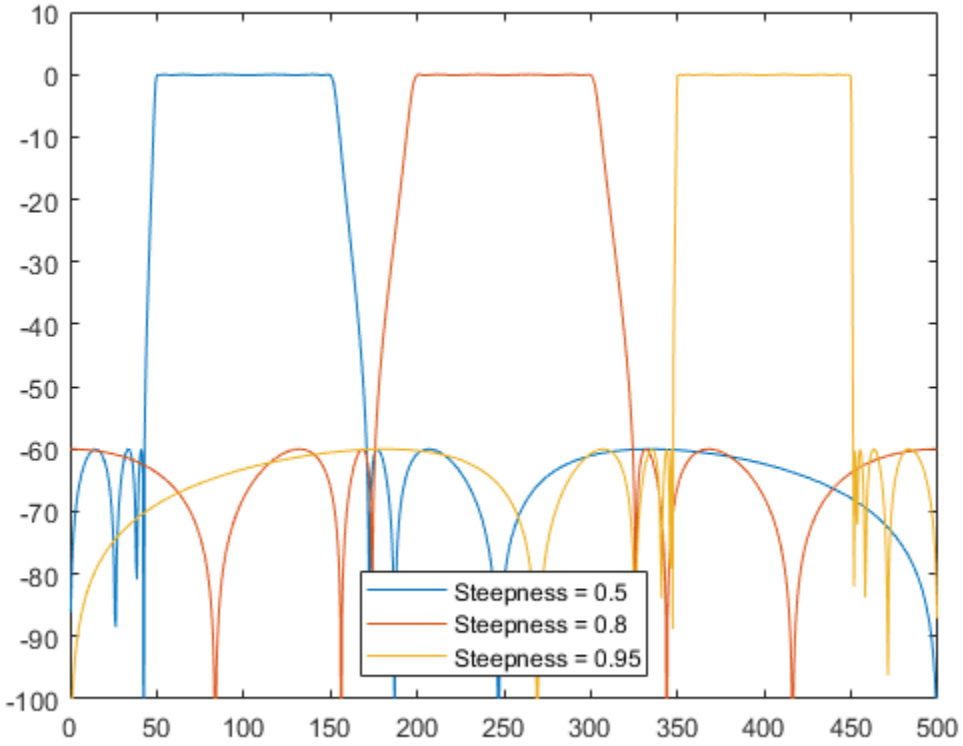
Filter white noise sampled at 1 kHz using an infinite impulse response bandpass filter with a passband width of 100 Hz. Use different steepness values. Plot the spectra of the filtered signals.

```
fs = 1000;
x = randn(20000,1);

[y1,d1] = bandpass(x,[ 50 150],fs,'ImpulseResponse','iir','Steepness',0.5);
[y2,d2] = bandpass(x,[200 300],fs,'ImpulseResponse','iir','Steepness',0.8);
[y3,d3] = bandpass(x,[350 450],fs,'ImpulseResponse','iir','Steepness',0.95);

pspectrum([y1 y2 y3],fs)
legend('Steepness = 0.5','Steepness = 0.8','Steepness = 0.95', ...
    'Location','south')
```



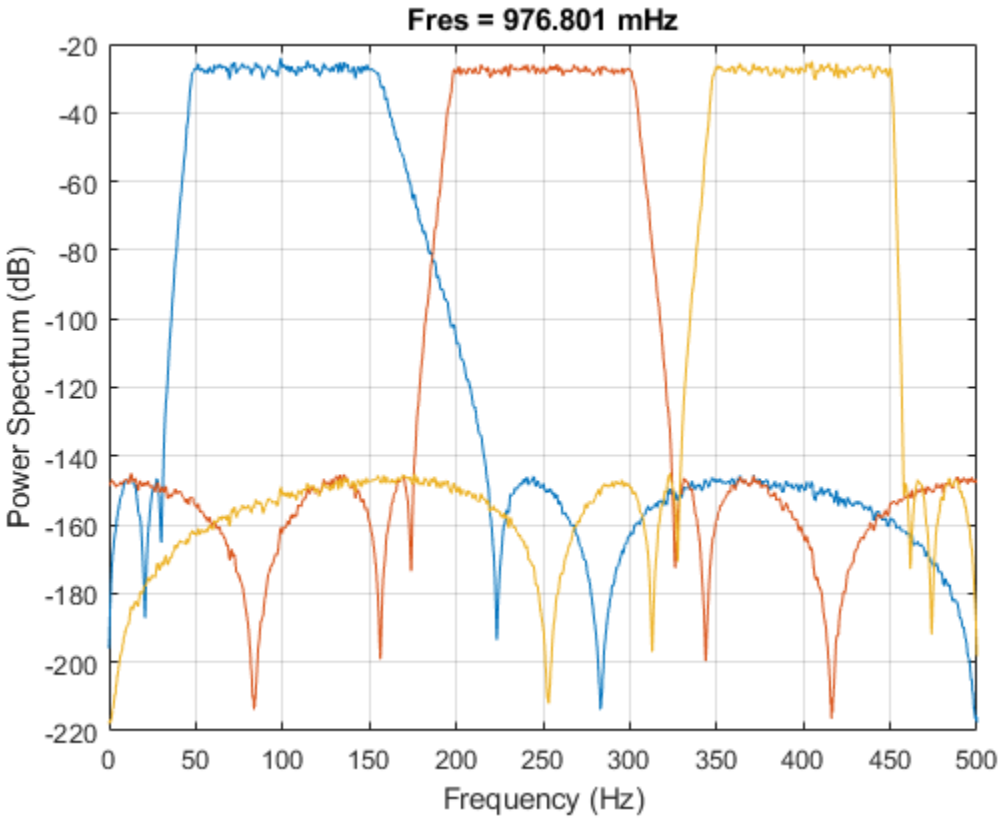
Compute and plot the frequency responses of the filters.

```
[h1,f] = freqz(d1,1024,fs);
[h2,~] = freqz(d2,1024,fs);
[h3,~] = freqz(d3,1024,fs);

plot(f,mag2db(abs([h1 h2 h3])))
legend('Steepness = 0.5','Steepness = 0.8','Steepness = 0.95', ...
    'Location','south')
ylim([-100 10])
```



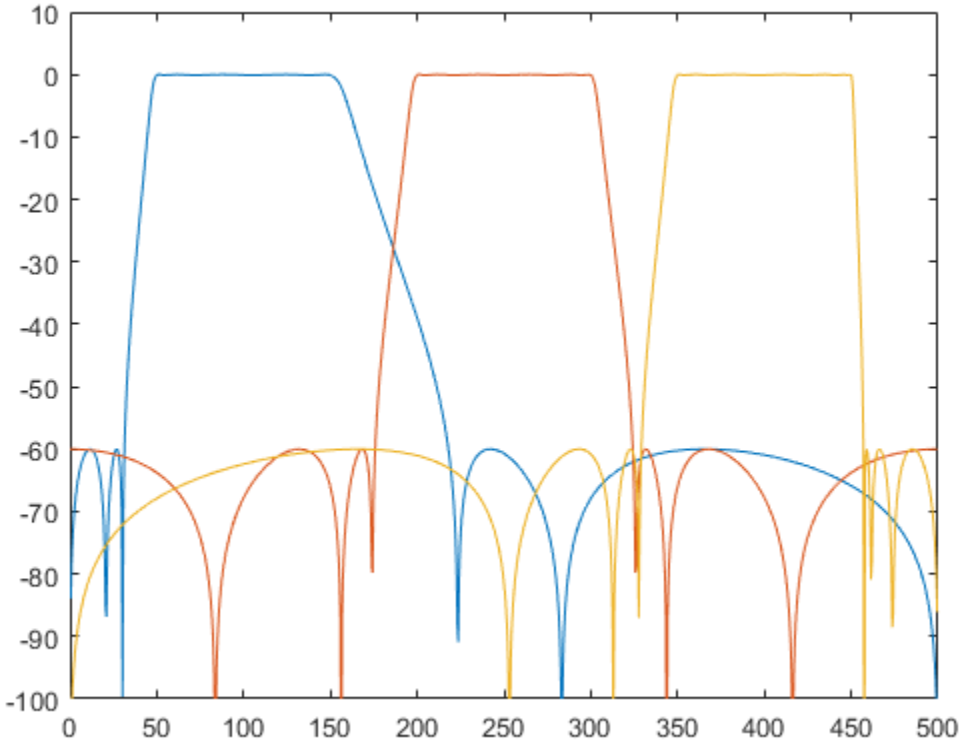
Make the filters asymmetric by specifying different values of steepness at the lower and higher passband frequencies.

```
[y1,d1] = bandpass(x,[ 50 150],fs,'ImpulseResponse','iir','Steepness',[0.5 0.8]);
[y2,d2] = bandpass(x,[200 300],fs,'ImpulseResponse','iir','Steepness',[0.5 0.8]);
[y3,d3] = bandpass(x,[350 450],fs,'ImpulseResponse','iir','Steepness',[0.5 0.8]);

pspectrum([y1 y2 y3],fs)
```



Compute and plot the frequency responses of the filters.

```
[h1,f] = freqz(d1,1024,fs);
[h2,~] = freqz(d2,1024,fs);
[h3,~] = freqz(d3,1024,fs);

plot(f,mag2db(abs([h1 h2 h3])))
ylim([-100 10])
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix.

Example: `sin(2*pi*(0:127)/16)+randn(1,128)/100` specifies a noisy sinusoid

Example: `[2 1].*sin(2*pi*(0:127)'./[16 64])` specifies a two-channel sinusoid.

Data Types: `single` | `double`
Complex Number Support: Yes

wpass — Normalized passband frequency range

two-element vector with elements in (0, 1)

Normalized passband frequency range, specified as a two-element vector with elements in the interval (0, 1).

fpass — Passband frequency range

two-element vector with elements in (0, `fs`/2)

Passband frequency range, specified as a two-element vector with elements in the interval (0, `fs`/2).

`fs` — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar.

`xt` — Input timetable

timetable

Input timetable. `xt` must contain increasing, finite, and equally spaced row times of type `duration` in seconds.

If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

Example: `timetable(seconds(0:4)',randn(5,1),randn(5,2))` contains a single-channel random signal and a two-channel random signal, sampled at 1 Hz for 4 seconds.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ImpulseResponse','iir','StopbandAttenuation',30` filters the input using a minimum-order IIR filter that attenuates by 30 dB the frequencies smaller than `fpass(1)` and the frequencies larger than `fpass(2)`.

`ImpulseResponse` — Type of impulse response

`'auto'` (default) | `'fir'` | `'iir'`

Type of impulse response of the filter, specified as the comma-separated pair consisting of `'ImpulseResponse'` and `'fir'`, `'iir'`, or `'auto'`.

- `'fir'` — The function designs a minimum-order, linear-phase, finite impulse response (FIR) filter. To compensate for the delay, the function appends to the input signal $N/2$ zeros, where N is the filter order. The function then filters the signal and removes the first $N/2$ samples of the output.

  In this case, the input signal must be at least twice as long as the filter that meets the specifications.
- `'iir'` — The function designs a minimum-order infinite impulse response (IIR) filter and uses the `filtfilt` function to perform zero-phase filtering and compensate for the filter delay.

  If the signal is not at least three times as long as the filter that meets the specifications, the function designs a filter with smaller order and thus smaller steepness.
- `'auto'` — The function designs a minimum-order FIR filter if the input signal is long enough, and a minimum-order IIR filter otherwise. Specifically, the function follows these steps:
  - Compute the minimum order that an FIR filter must have to meet the specifications. If the signal is at least twice as long as the required filter order, design and use that filter.
  - If the signal is not long enough, compute the minimum order that an IIR filter must have to meet the specifications. If the signal is at least three times as long as the required filter order, design and use that filter.

- If the signal is not long enough, truncate the order to one-third the signal length and design an IIR filter of that order. The reduction in order comes at the expense of transition band steepness.
- Filter the signal and compensate for the delay.

`Steepness` — Transition band steepness
`0.85` (default) | scalar in the interval [0.5, 1) | two-element vector with elements in the interval [0.5, 1)

Transition band steepness, specified as the comma-separated pair consisting of `'Steepness'` and a scalar or two-element vector with elements in the interval [0.5, 1). As the steepness increases, the filter response approaches the ideal bandpass response, but the resulting filter length and the computational cost of the filtering operation also increase. See “Bandpass Filter Steepness” on page 1-40 for more information.

`StopbandAttenuation` — Filter stopband attenuation
`60` (default) | positive scalar in dB

Filter stopband attenuation, specified as the comma-separated pair consisting of `'StopbandAttenuation'` and a positive scalar in dB.

Output Arguments

`y` — Filtered signal
vector | matrix | timetable

Filtered signal, returned as a vector, a matrix, or a timetable with the same dimensions as the input.

`d` — Bandpass filter
`digitalFilter` object

Bandpass filter used in the filtering operation, returned as a `digitalFilter` object.

- Use `filter(d,x)` to filter a signal `x` using `d`.
- Use **FVTool** to visualize the filter response.
- Use `designfilt` to edit or generate a digital filter based on frequency-response specifications.

More About

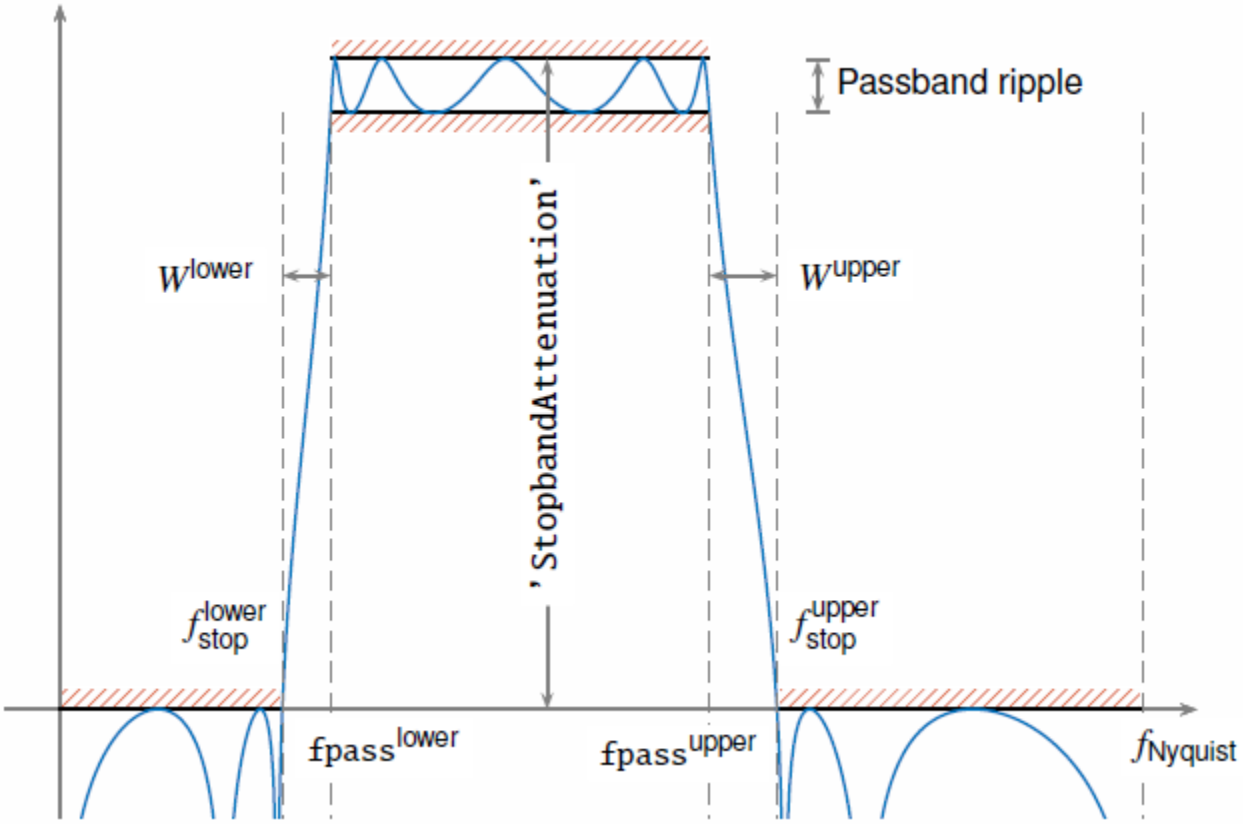
Bandpass Filter Steepness

The `'Steepness'` argument controls the width of a filter's transition regions. The lower the steepness, the wider the transition region. The higher the steepness, the narrower the transition region.

To interpret the filter steepness, consider the following definitions:

- The Nyquist frequency, f_{Nyquist}, is the highest frequency component of a signal that can be sampled at a given rate without aliasing. f_{Nyquist} is 1 ($\times\pi$ rad/sample) when the input signal has no time information, and `fs`/2 hertz when the input signal is a timetable or when you specify a sample rate.

- The lower and upper stopband frequencies of the filter, f_{stop}^{lower} and f_{stop}^{upper}, are the frequencies below which and above which the attenuation is equal to or greater than the value specified using `'StopbandAttenuation'`.
- The lower transition width of the filter, W^{lower}, is $\texttt{fpass}^{lower} - f_{stop}^{lower}$, where the lower passband frequency $\texttt{fpass}^{lower}$ is the first element of the specified `fpass`.
- The upper transition width of the filter, W^{upper}, is $f_{stop}^{upper} - \texttt{fpass}^{upper}$, where the upper passband frequency$\texttt{fpass}^{upper}$ is the second element of `fpass`.
- Most nonideal filters also attenuate the input signal across the passband. The maximum value of this frequency-dependent attenuation is called the passband ripple. Every filter used by `bandpass` has a passband ripple of 0.1 dB.



To control the width of the transition bands, you can specify `'Steepness'` as either a two-element vector, $[s^{lower},s^{upper}]$, or a scalar. When you specify `'Steepness'` as a vector, the function:

- Computes the lower transition width as $W^{lower} = (1 - s^{lower}) \times \texttt{fpass}^{lower}$.
  - When the first element of `'Steepness'` is equal to 0.5, the transition width is 50% of $\texttt{fpass}^{lower}$.
  - As the first element of `'Steepness'` approaches 1, the transition width becomes progressively narrower until it reaches a minimum value of 1% of $\texttt{fpass}^{lower}$.
- Computes the upper transition width as $W^{upper} = (1 - s^{upper}) \times (f_{Nyquist} - \texttt{fpass}^{upper})$.

- When the second element of `'Steepness'` is equal to 0.5, the transition width is 50% of (f_{Nyquist} – `fpass`^{upper}).
- As the second element of `'Steepness'` approaches 1, the transition width becomes progressively narrower until it reaches a minimum value of 1% of (f_{Nyquist} – `fpass`^{upper}).

When you specify `'Steepness'` as a scalar, the function designs a filter with equal lower and upper transition widths. The default value of `'Steepness'` is 0.85.

See Also

Apps
Signal Analyzer

Functions
`bandstop` | `designfilt` | `filter` | `fir1` | `highpass` | `lowpass`

Introduced in R2018a

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 sample rate, `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 `'psd'` option indicates that 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 `'psd'` option 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)
```

```
p = 1.5264
```

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

```
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. Reset the random number generator for reproducible results.

```
rng('default')

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

pband = bandpower(x,1000,[50 150]);
ptot = bandpower(x,1000,[0 500]);
per_power = 100*(pband/ptot)
```

```
per_power = 51.9591
```

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')
```

```
p = 1.5264
```

```
avpow = norm(x,2)^2/numel(x)
```

```
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,[50 150],'psd');
pTot = bandpower(Pxx,F,'psd');
per_power = 100*(pBand/pTot)
```

```
per_power = 49.1798
```

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 ℓ_2 norm.

```
p = bandpower(x)
```

```
p = 1×3

    1.5264    1.5382    1.4717
```

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

```
l2norm = 1×3

    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

column vector | matrix

One- or two-sided PSD estimates, specified as a real-valued column vector or matrix with nonnegative elements.

The power spectral density must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[pxx,f] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` specifies the periodogram PSD estimate of a noisy two-channel sinusoid sampled at 2π Hz and the frequencies at which it is computed.

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`

References

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

[2] Stoica, Petre, and Randolph Moses. *Spectral Analysis of Signals*. Upper Saddle River, NJ: Prentice Hall, 2005.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Input argument `'psd'`, when specified, must be a compile time constant.

Thread-Based Environment

Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

This function fully supports thread-based environments. For more information, see “Run MATLAB Functions in Thread-Based Environment”.

See Also

`periodogram` | `sfdr`

Introduced in R2013a

bandstop

Bandstop-filter signals

Syntax

```
y = bandstop(x,wpass)
y = bandstop(x,fpass,fs)
y = bandstop(xt,fpass)

y = bandstop( ___ ,Name,Value)

[y,d] = bandstop( ___ )

bandstop( ___ )
```

Description

`y = bandstop(x,wpass)` filters the input signal `x` using a bandstop filter with a stopband frequency range specified by the two-element vector `wpass` and expressed in normalized units of π rad/sample. `bandstop` uses a minimum-order filter with a stopband attenuation of 60 dB and compensates for the delay introduced by the filter. If `x` is a matrix, the function filters each column independently.

`y = bandstop(x,fpass,fs)` specifies that `x` has been sampled at a rate of `fs` hertz. The two-element vector `fpass` specifies the stopband frequency range of the filter in hertz.

`y = bandstop(xt,fpass)` bandstop-filters the data in timetable `xt` using a filter with a stopband frequency range specified in hertz by the two-element vector `fpass`. The function independently filters all variables in the timetable and all columns inside each variable.

`y = bandstop( ___ ,Name,Value)` specifies additional options for any of the previous syntaxes using name-value pair arguments. You can change the stopband attenuation, the “Bandstop Filter Steepness” on page 1-58, and the type of impulse response of the filter.

`[y,d] = bandstop( ___ )` also returns the `digitalFilter` object `d` used to filter the input.

`bandstop( ___ )` with no output arguments plots the input signal and overlays the filtered signal.

Examples

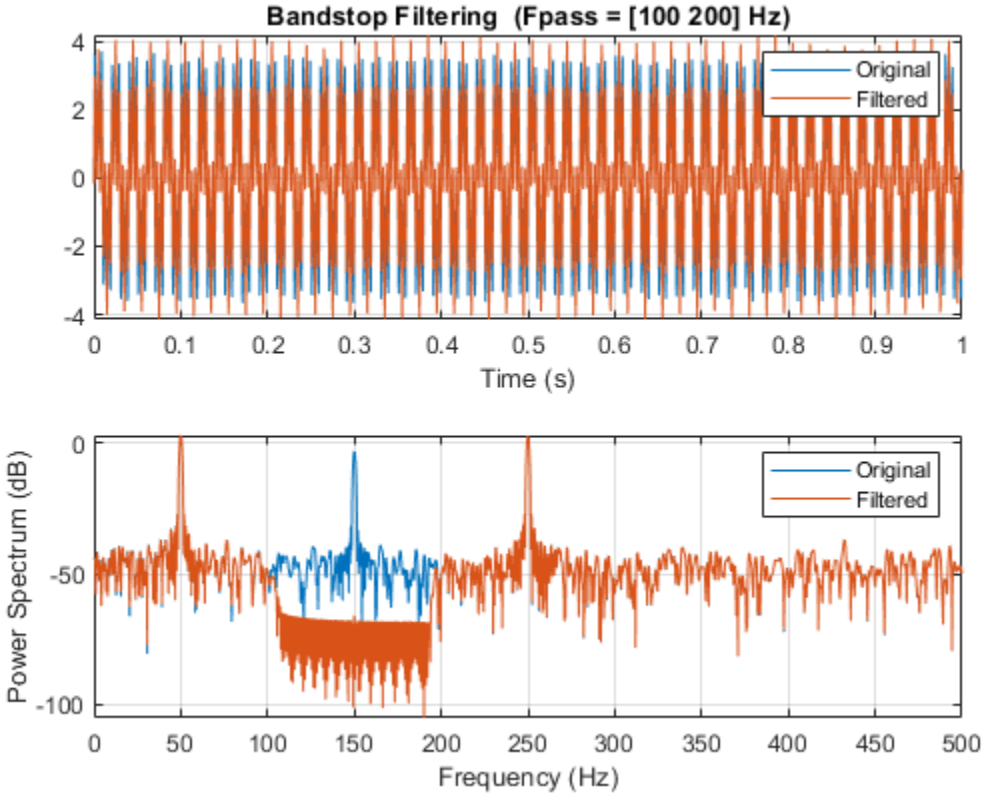
Bandstop Filtering of Tones

Create a signal sampled at 1 kHz for 1 second. The signal contains three tones, one at 50 Hz, another at 150 Hz, and a third at 250 Hz. The high-frequency and low-frequency tones both have twice the amplitude of the intermediate tone. The signal is embedded in Gaussian white noise of variance 1/100.

```
fs = 1e3;
t = 0:1/fs:1;
x = [2 1 2]*sin(2*pi*[50 150 250]'.*t) + randn(size(t))/10;
```

Bandstop-filter the signal to remove the medium-frequency tone. Specify passband frequencies of 100 Hz and 200 Hz. Display the original and filtered signals, and also their spectra.

```
bandstop(x,[100 200],fs)
```

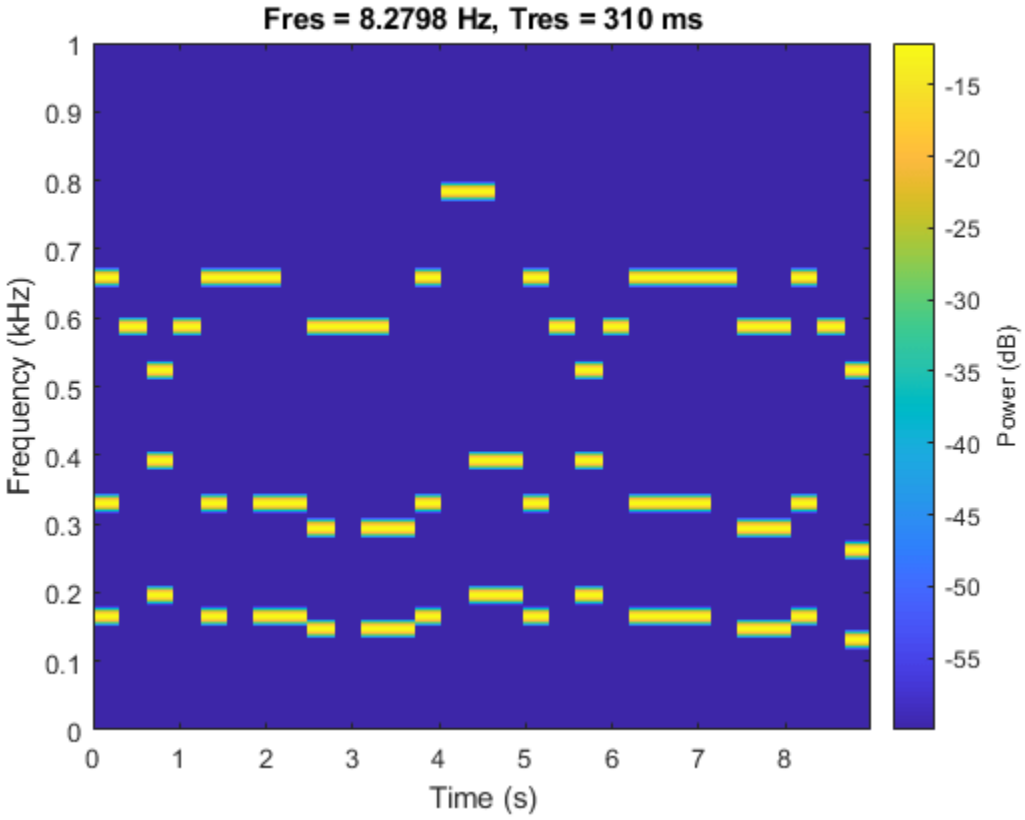


Bandstop Filtering of Musical Signal

Implement a basic digital music synthesizer and use it to play a traditional song. Specify a sample rate of 2 kHz. Plot the spectrogram of the song.

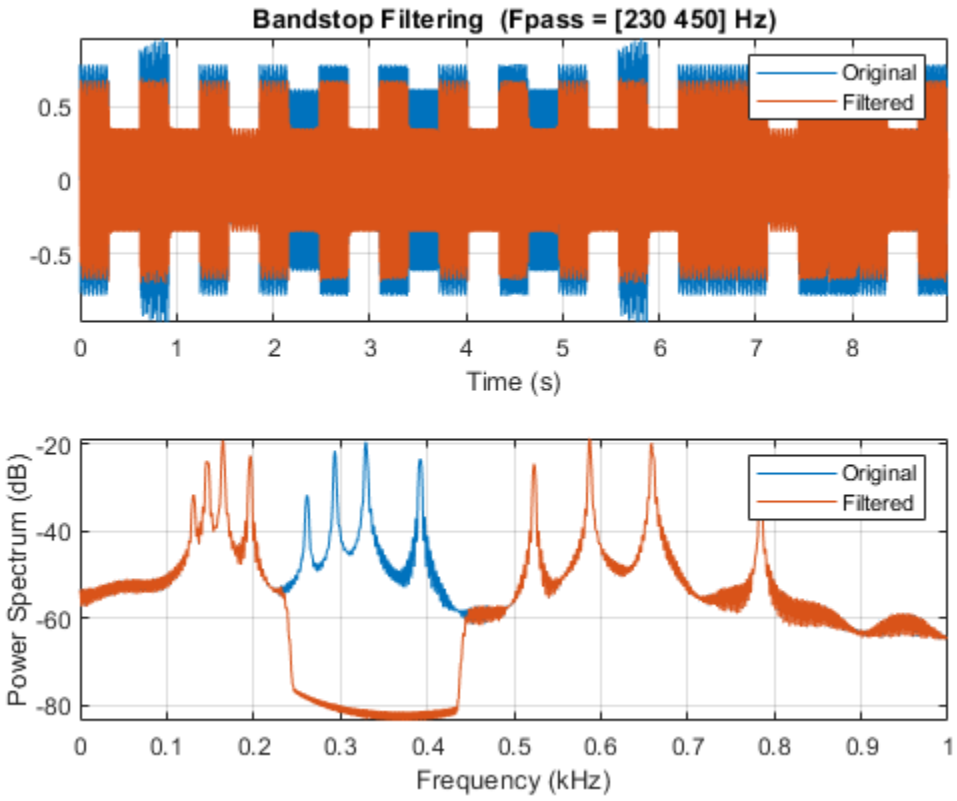
```
fs = 2e3;
t = 0:1/fs:0.3-1/fs;

l = [0 130.81 146.83 164.81 174.61 196.00 220 246.94];
m = [0 261.63 293.66 329.63 349.23 392.00 440 493.88];
h = [0 523.25 587.33 659.25 698.46 783.99 880 987.77];
note = @(f,g) [1 1 1]*sin(2*pi*[l(g) m(g) h(f)]'.*t);

mel = [3 2 1 2 3 3 3 0 2 2 2 0 3 5 5 0 3 2 1 2 3 3 3 3 2 2 3 2 1]+1;
acc = [3 0 5 0 3 0 3 3 2 0 2 2 3 0 5 5 3 0 5 0 3 3 3 0 2 2 3 0 1]+1;

song = [];
for kj = 1:length(mel)
    song = [song note(mel(kj),acc(kj)) zeros(1,0.01*fs)];
end
```

```
song = song/(max(abs(song))+0.1);

% To hear, type sound(song,fs)

pspectrum(song,fs,'spectrogram','TimeResolution',0.31, ...
    'OverlapPercent',0,'MinThreshold',-60)
```



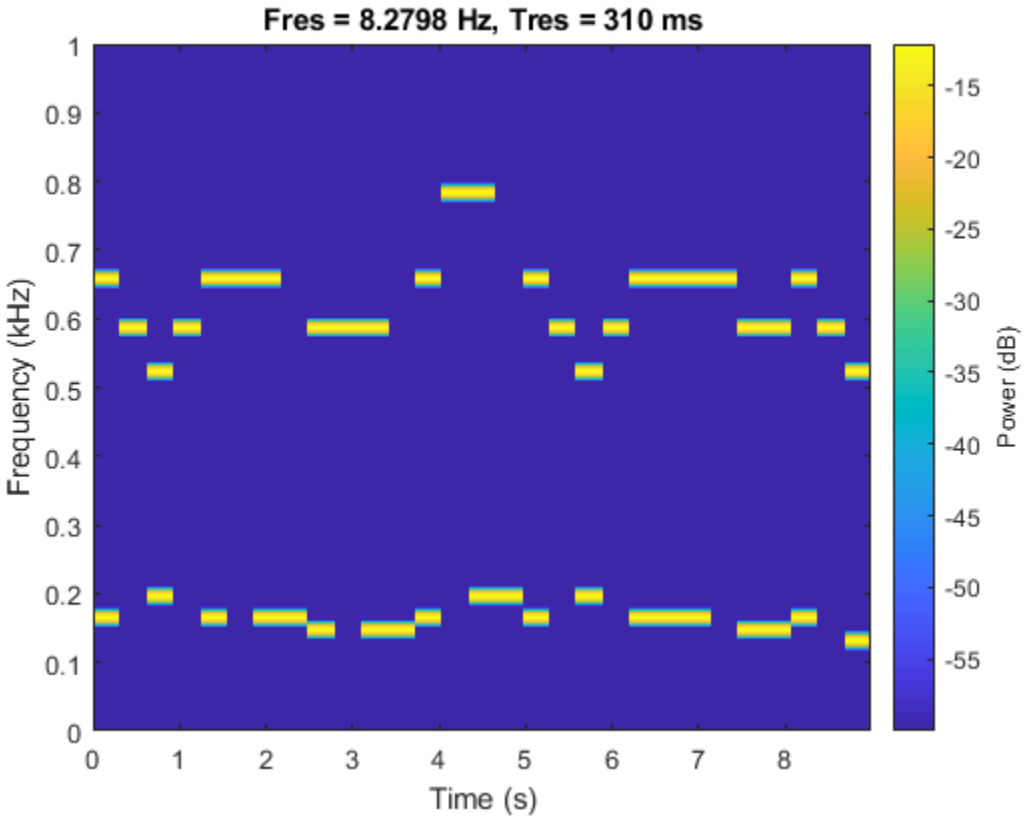
Bandstop-filter the signal to separate the middle register from the other two. Specify passband frequencies of 230 Hz and 450 Hz. Plot the original and filtered signals in the time and frequency domains.

```
bong = bandstop(song,[230 450],fs);

% To hear, type sound(bong,fs)

bandstop(song,[230 450],fs)
```



Plot the spectrogram of the song without the middle register.

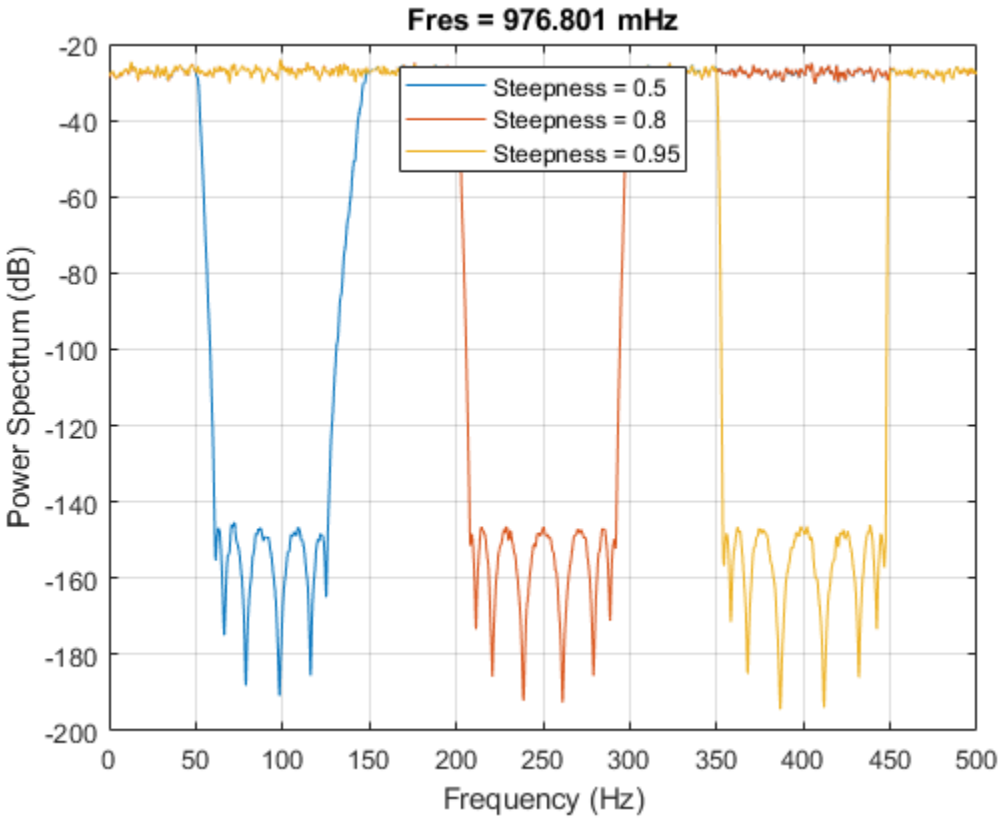
```
figure
pspectrum(bong,fs,'spectrogram','TimeResolution',0.31, ...
    'OverlapPercent',0,'MinThreshold',-60)
```



Bandstop Filter Steepness

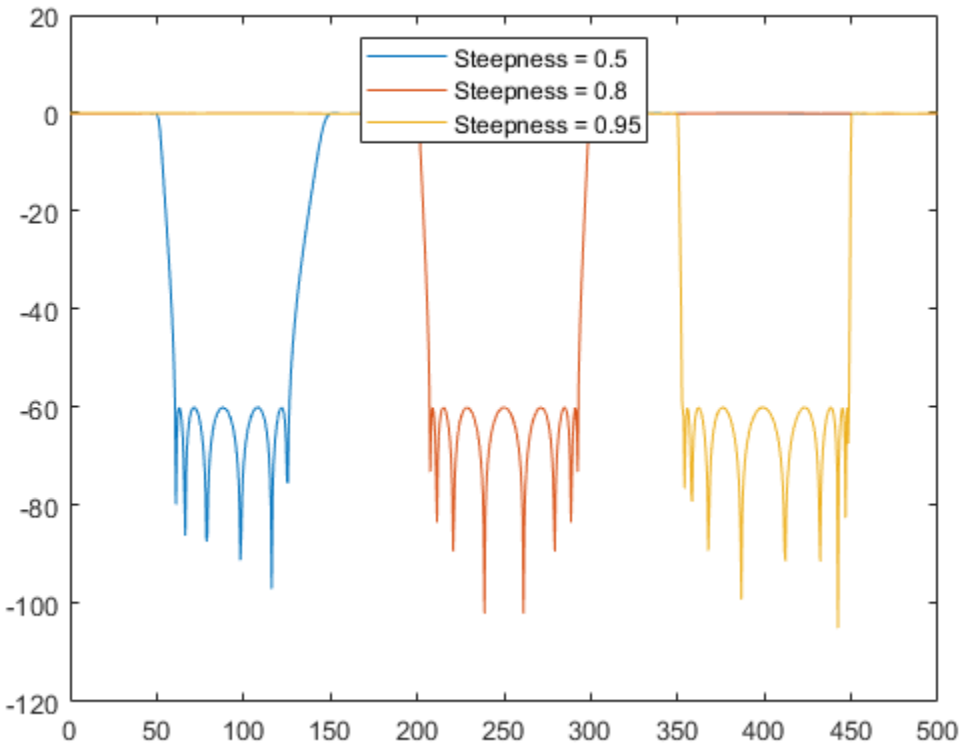
Filter white noise sampled at 1 kHz using an infinite impulse response bandstop filter with a stopband width of 100 Hz. Use different steepness values. Plot the spectra of the filtered signals.

```
fs = 1000;
x = randn(20000,1);

[y1,d1] = bandstop(x,[ 50 150],fs,'ImpulseResponse','iir','Steepness',0.5);
[y2,d2] = bandstop(x,[200 300],fs,'ImpulseResponse','iir','Steepness',0.8);
[y3,d3] = bandstop(x,[350 450],fs,'ImpulseResponse','iir','Steepness',0.95);

pspectrum([y1 y2 y3],fs)
legend('Steepness = 0.5','Steepness = 0.8','Steepness = 0.95', ...
    'Location','north')
```



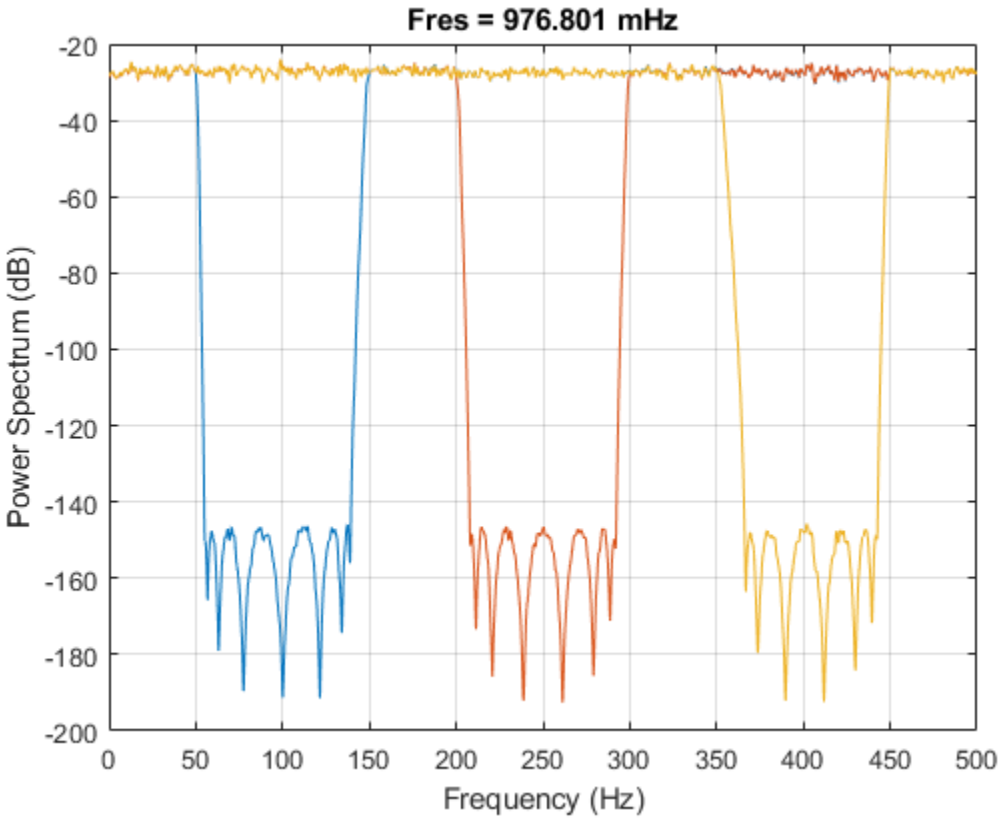
Compute and plot the frequency responses of the filters.

```
[h1,f] = freqz(d1,1024,fs);
[h2,~] = freqz(d2,1024,fs);
[h3,~] = freqz(d3,1024,fs);

plot(f,mag2db(abs([h1 h2 h3])))
legend('Steepness = 0.5','Steepness = 0.8','Steepness = 0.95', ...
    'Location','north')
ylim([-120 20])
```



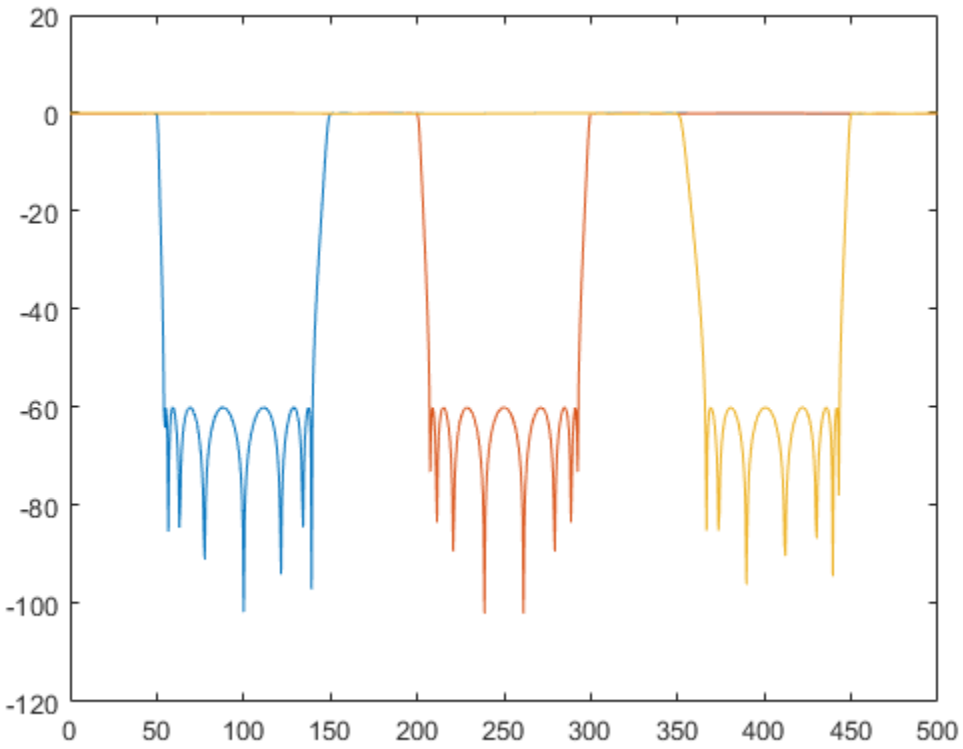
Make the filters asymmetric by specifying different values of steepness at the lower and higher passband frequencies.

```
[y1,d1] = bandstop(x,[ 50 150],fs,'ImpulseResponse','iir','Steepness',[0.5 0.8]);
[y2,d2] = bandstop(x,[200 300],fs,'ImpulseResponse','iir','Steepness',[0.5 0.8]);
[y3,d3] = bandstop(x,[350 450],fs,'ImpulseResponse','iir','Steepness',[0.5 0.8]);

pspectrum([y1 y2 y3],fs)
```



Compute and plot the frequency responses of the filters.

```
[h1,f] = freqz(d1,1024,fs);
[h2,~] = freqz(d2,1024,fs);
[h3,~] = freqz(d3,1024,fs);

plot(f,mag2db(abs([h1 h2 h3])))
ylim([-120 20])
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix.

Example: `sin(2*pi*(0:127)/16)+randn(1,128)/100` specifies a noisy sinusoid

Example: `[2 1].*sin(2*pi*(0:127)'./[16 64])` specifies a two-channel sinusoid.

Data Types: `single` | `double`
Complex Number Support: Yes

wpass — Normalized stopband frequency range

two-element vector with elements in (0, 1)

Normalized stopband frequency range, specified as a two-element vector with elements in the interval (0, 1).

fpass — Stopband frequency range

two-element vector with elements in (0, `fs`/2)

Stopband frequency range, specified as a two-element vector with elements in the interval (0, `fs`/2).

`fs` — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar.

`xt` — Input timetable

timetable

Input timetable. `xt` must contain increasing, finite, and equally spaced row times of type `duration` in seconds.

If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

Example: `timetable(seconds(0:4)',randn(5,1),randn(5,2))` contains a single-channel random signal and a two-channel random signal, sampled at 1 Hz for 4 seconds.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ImpulseResponse','iir','StopbandAttenuation',30` filters the input using a minimum-order IIR filter that attenuates by 30 dB the frequencies from `fpass(1)` to `fpass(2)`.

`ImpulseResponse` — Type of impulse response

`'auto'` (default) | `'fir'` | `'iir'`

Type of impulse response of the filter, specified as the comma-separated pair consisting of `'ImpulseResponse'` and `'fir'`, `'iir'`, or `'auto'`.

- `'fir'` — The function designs a minimum-order, linear-phase, finite impulse response (FIR) filter. To compensate for the delay, the function appends to the input signal $N/2$ zeros, where N is the filter order. The function then filters the signal and removes the first $N/2$ samples of the output.

  In this case, the input signal must be at least twice as long as the filter that meets the specifications.
- `'iir'` — The function designs a minimum-order infinite impulse response (IIR) filter and uses the `filtfilt` function to perform zero-phase filtering and compensate for the filter delay.

  If the signal is not at least three times as long as the filter that meets the specifications, the function designs a filter with smaller order and thus smaller steepness.
- `'auto'` — The function designs a minimum-order FIR filter if the input signal is long enough, and a minimum-order IIR filter otherwise. Specifically, the function follows these steps:
  - Compute the minimum order that an FIR filter must have to meet the specifications. If the signal is at least twice as long as the required filter order, design and use that filter.
  - If the signal is not long enough, compute the minimum order that an IIR filter must have to meet the specifications. If the signal is at least three times as long as the required filter order, design and use that filter.

- If the signal is not long enough, truncate the order to one-third the signal length and design an IIR filter of that order. The reduction in order comes at the expense of transition band steepness.
- Filter the signal and compensate for the delay.

`Steepness` — Transition band steepness
`0.85` (default) | scalar in the interval [0.5, 1) | two-element vector with elements in the interval [0.5, 1)

Transition band steepness, specified as the comma-separated pair consisting of `'Steepness'` and a scalar or two-element vector with elements in the interval [0.5, 1). As the steepness increases, the filter response approaches the ideal bandstop response, but the resulting filter length and the computational cost of the filtering operation also increase. See “Bandstop Filter Steepness” on page 1-58 for more information.

`StopbandAttenuation` — Filter stopband attenuation
`60` (default) | positive scalar in dB

Filter stopband attenuation, specified as the comma-separated pair consisting of `'StopbandAttenuation'` and a positive scalar in dB.

Output Arguments

`y` — Filtered signal
vector | matrix | timetable

Filtered signal, returned as a vector, a matrix, or a timetable with the same dimensions as the input.

`d` — Bandstop filter
`digitalFilter` object

Bandstop filter used in the filtering operation, returned as a `digitalFilter` object.

- Use `filter(d,x)` to filter a signal `x` using `d`.
- Use **FVTool** to visualize the filter response.
- Use `designfilt` to edit or generate a digital filter based on frequency-response specifications.

More About

Bandstop Filter Steepness

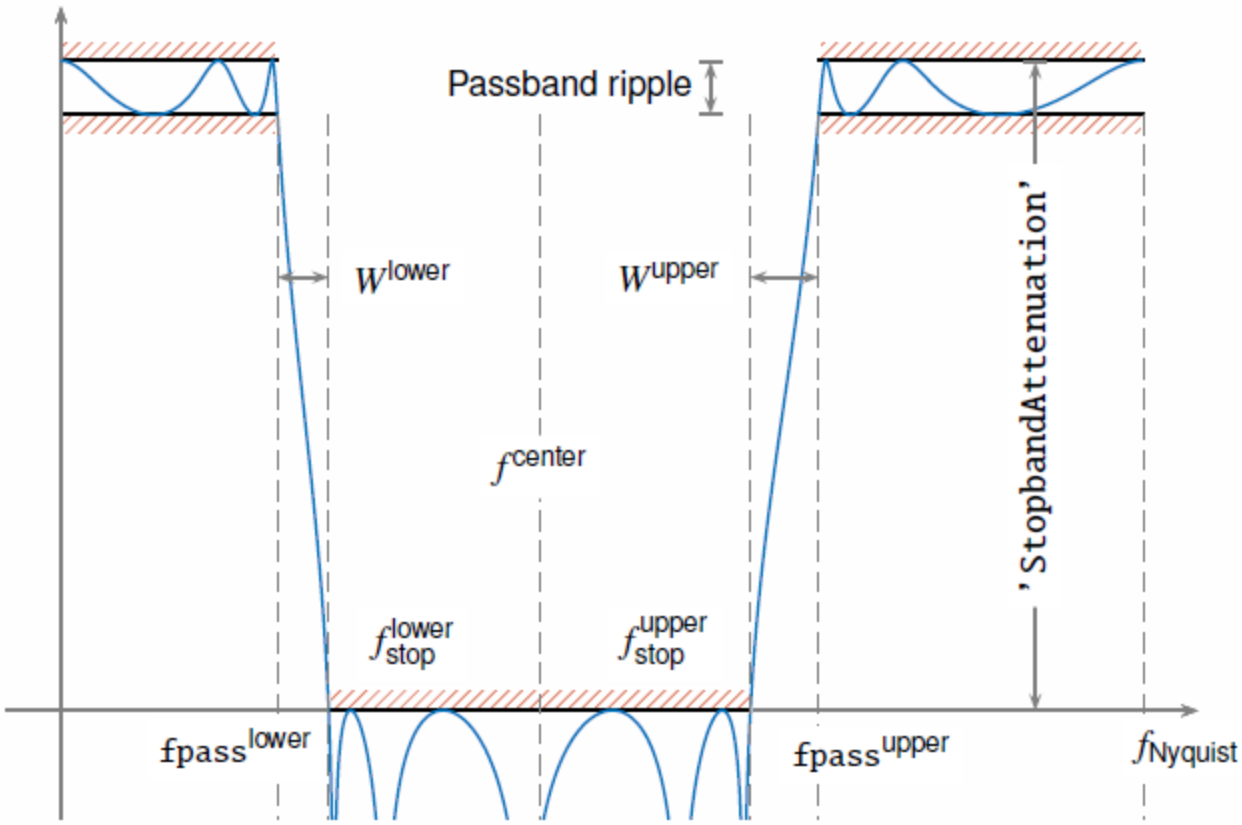
The `'Steepness'` argument controls the width of a filter's transition regions. The lower the steepness, the wider the transition region. The higher the steepness, the narrower the transition region.

To interpret the filter steepness, consider the following definitions:

- The Nyquist frequency, f_{Nyquist}, is the highest frequency component of a signal that can be sampled at a given rate without aliasing. f_{Nyquist} is 1 ($\times\pi$ rad/sample) when the input signal has no time information, and `fs`/2 hertz when the input signal is a timetable or when you specify a sample rate.

- The lower and upper stopband frequencies of the filter, f_{stop}^{lower} and f_{stop}^{upper}, are the frequencies between which the attenuation is equal to or greater than the value specified using `'StopbandAttenuation'`.

  The center of the stopband region is $f^{center} = (f_{stop}^{lower} + f_{stop}^{upper})/2$.
- The lower transition width of the filter, W^{lower}, is f_{stop}^{lower} – $\texttt{fpass}^{lower}$, where the lower bandpass frequency $\texttt{fpass}^{lower}$ is the first element of the specified `fpass`.
- The *upper transition width* of the filter, W^{upper}, is $\texttt{fpass}^{upper}$ – f_{stop}^{upper}, where the upper bandpass frequency $\texttt{fpass}^{upper}$ is the second element of `fpass`.
- Most nonideal filters also attenuate the input signal across the passband. The maximum value of this frequency-dependent attenuation is called the passband ripple. Every filter used by `bandstop` has a passband ripple of 0.1 dB.



To control the width of the transition bands, you can specify `'Steepness'` as either a two-element vector, $[s^{lower},s^{upper}]$, or a scalar. When you specify `'Steepness'` as a vector, the function:

- Computes the lower transition width as
  $W^{lower} = (1 - s^{lower}) \times (f^{center} - \texttt{fpass}^{lower})$.
  - When the first element of `'Steepness'` is equal to 0.5, the transition width is 50% of (f^{center} – $\texttt{fpass}^{lower}$).
  - As the first element of `'Steepness'` approaches 1, the transition width becomes progressively narrower until it reaches a minimum value of 1% of (f^{center} – $\texttt{fpass}^{lower}$).
- Computes the upper transition width as

$W^{upper} = (1 - s^{upper}) \times (\texttt{fpass}^{upper} - f^{center})$.

- When the second element of `'Steepness'` is equal to 0.5, the transition width is 50% of ($\texttt{fpass}^{upper} - f^{center}$).
- As the second element of `'Steepness'` approaches 1, the transition width becomes progressively narrower until it reaches a minimum value of 1% of ($\texttt{fpass}^{upper} - f^{center}$).

When you specify `'Steepness'` as a scalar, the function designs a filter with equal lower and upper transition widths. The default value of `'Steepness'` is 0.85.

See Also

Apps
Signal Analyzer

Functions
`bandpass` | `designfilt` | `filter` | `fir1` | `highpass` | `lowpass`

Introduced in R2018a

barthannwin

Modified Bartlett-Hann window

Syntax

```
w = barthannwin(L)
```

Description

`w = barthannwin(L)` returns an `L`-point modified Bartlett-Hann window.

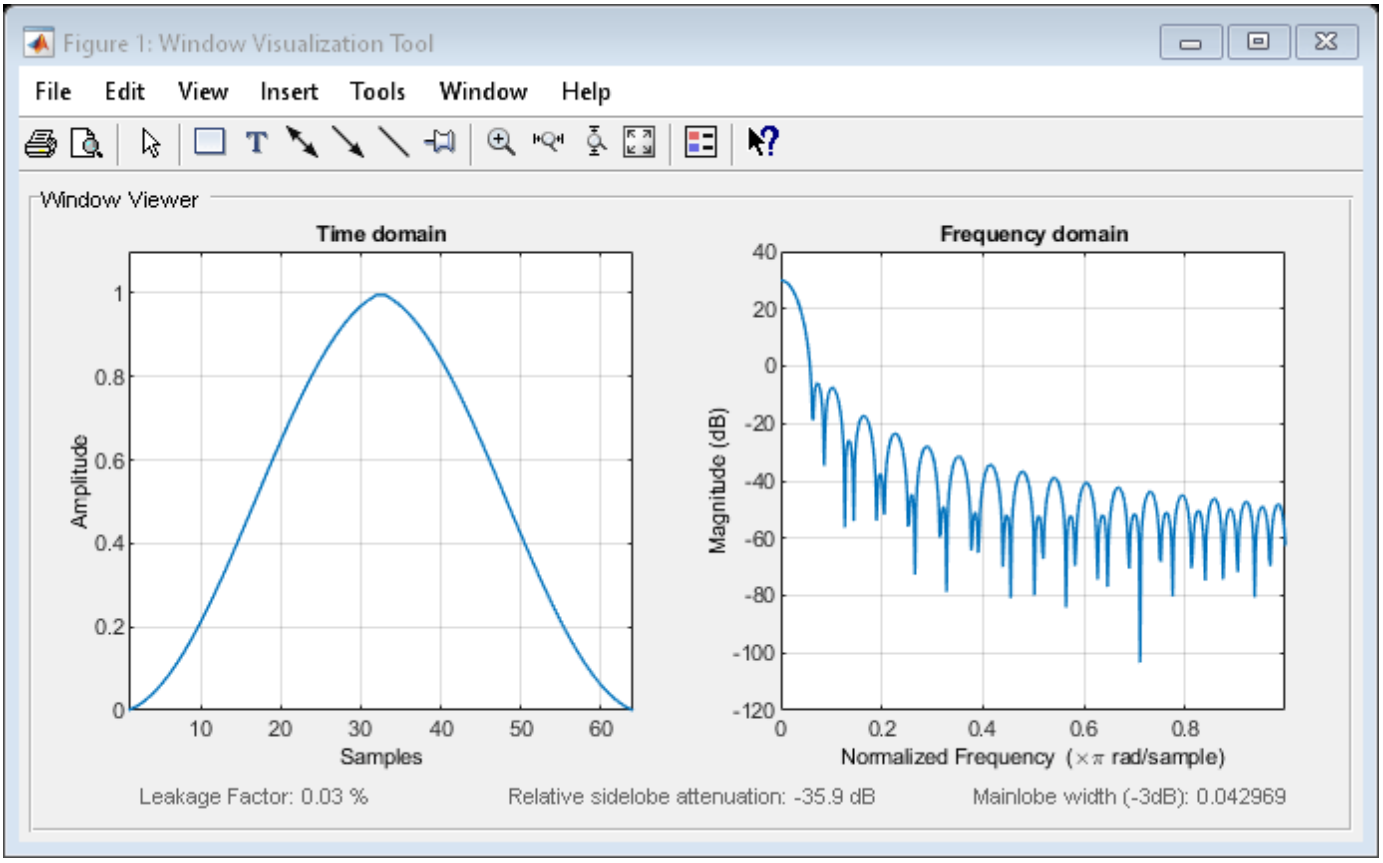
Note The Hann window is also called the Hanning window.

Examples

Bartlett-Hann Window

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

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



Input Arguments

L — Window length

real positive scalar

Window length, specified as a real positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

w — Modified Bartlett-Hann window

column vector

Modified Bartlett-Hann window, returned as a column vector of length L.

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 $0 \le n \le N$ and the window length is $L = N + 1$.

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.

References

[1] Ha, Y. H., and J. A. Pearce. "A New Window and Comparison to Standard Windows." *IEEE® Transactions on Acoustics, Speech, and Signal Processing*. Vol. 37, 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps

Window Designer

Functions

`bartlett` | `blackmanharris` | `bohmanwin` | `nuttallwin` | `parzenwin` | `rectwin` | `triang` | **WVTool**

Introduced before R2006a

bartlett

Bartlett window

Syntax

```
w = bartlett(L)
```

Description

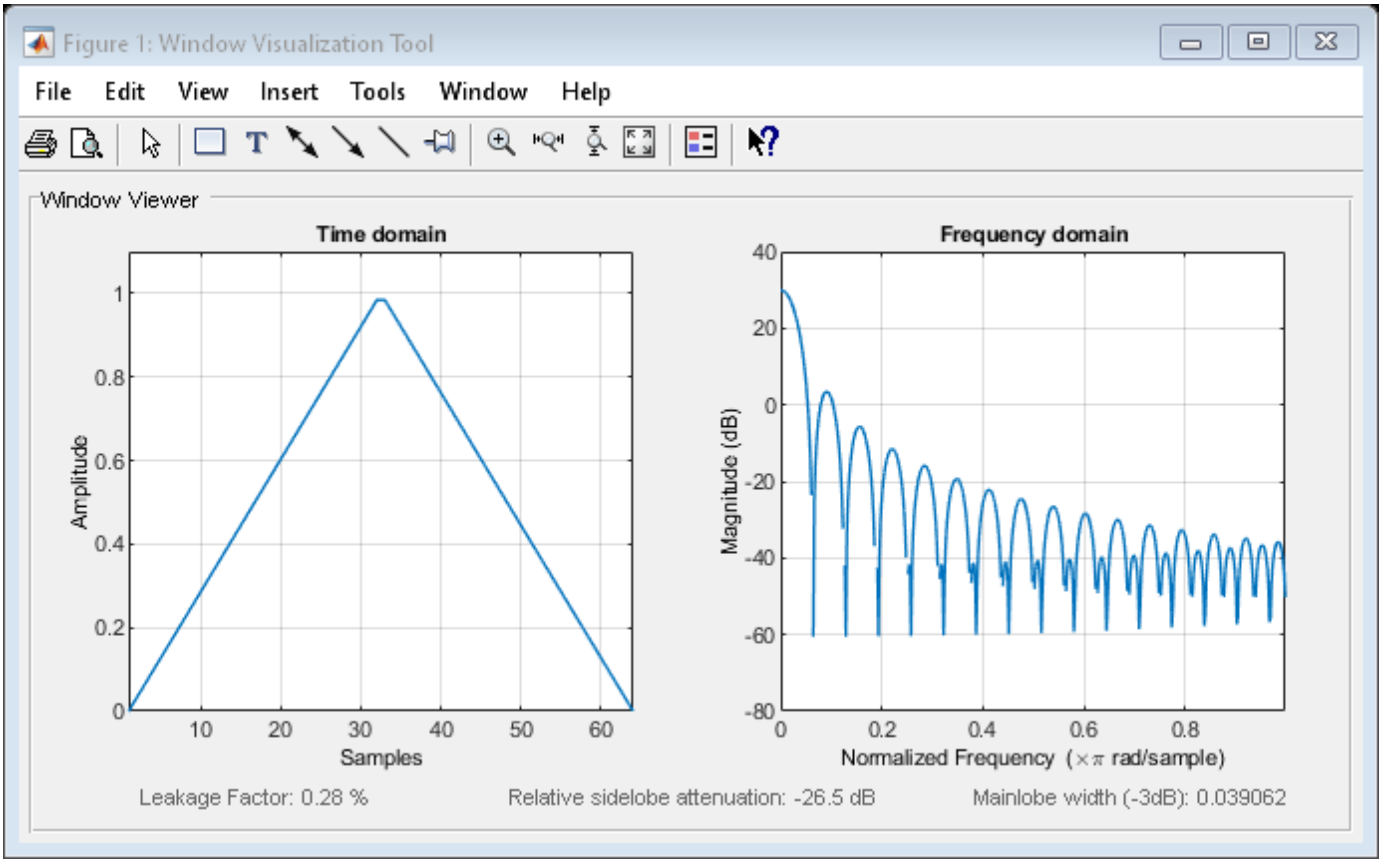
`w = bartlett(L)` returns an L-point symmetric Bartlett window.

Examples

Bartlett Window

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

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



Input Arguments

L — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

Output Arguments

w — Bartlett window

column vector

Bartlett window, returned as a column vector.

Algorithms

The following equation generates the coefficients of a Bartlett window:

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

The window length $L = N + 1$.

The Bartlett window is very similar to a triangular window as returned by the `triang` function. However, the Bartlett window always has zeros at the first and last samples, while the triangular window is nonzero at those points. For odd values of `L`, the center `L-2` points of `bartlett(L)` are equivalent to `triang(L-2)`.

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

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps

Window Designer

Functions
barthannwin | blackmanharris | bohmanwin | nuttallwin | parzenwin | rectwin | triang | **WVTool**

Introduced before R2006a

besselap

Bessel analog lowpass filter prototype

Syntax

```
[z,p,k] = besselap(n)
```

Description

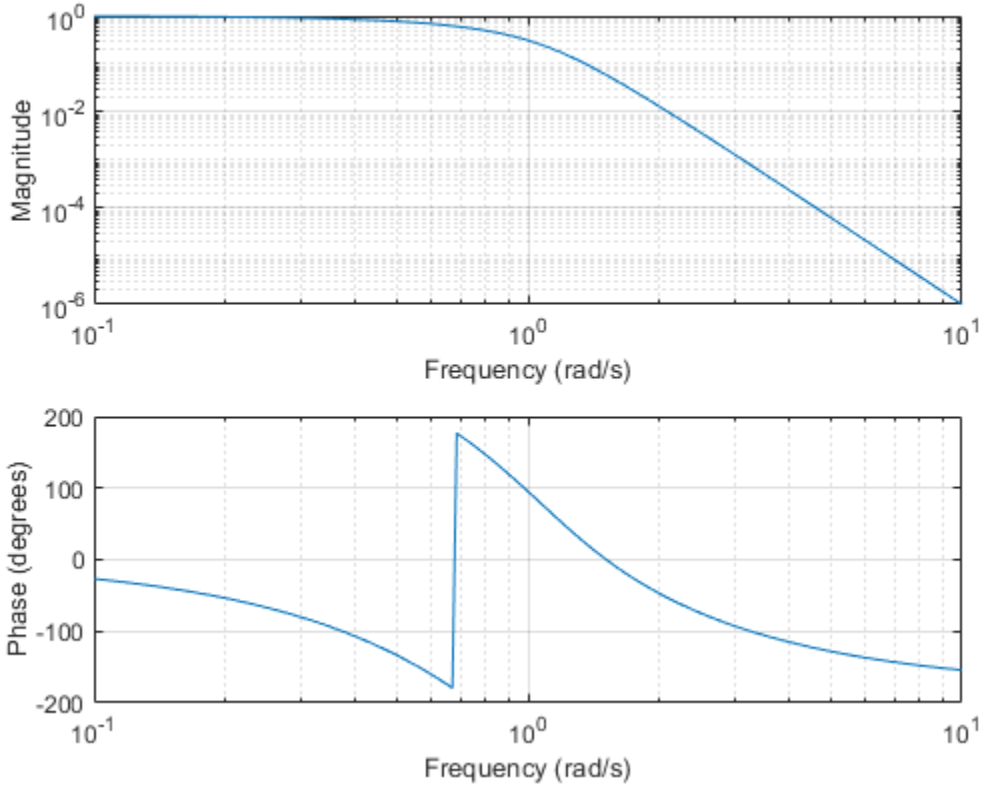
`[z,p,k] = besselap(n)` returns the poles and gain of an order-n Bessel analog lowpass filter prototype.

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
```



Input Arguments

n — Filter order
integer scalar

Filter order, specified as an integer scalar. `n` must be less than or equal to 25.

Data Types: `double`

Output Arguments

z — Zeros
matrix

Zeros of the filter prototype, returned as an empty matrix. `z` is empty because there are no zeros.

p — Poles
column vector

Poles of the filter prototype, returned as a length-`n` column vector.

k — Gain
scalar

Gain of the filter prototype, returned as a scalar.

Algorithms

The `besselap` function finds the filter roots from a lookup table constructed using Symbolic Math Toolbox™ software.

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

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

The `besselap` function 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 $\Omega_c = 1$. The transfer function is expressed in terms of `z`, `p`, and `k` as

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

References

[1] Rabiner, L. R., and B. Gold. *Theory and Application of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1975, pp. 228–230.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Filter order must be a constant. Expressions or variables are allowed if their values do not change.

See Also

`besself` | `buttap` | `cheb1ap` | `cheb2ap` | `ellipap`

Introduced before R2006a

besself

Bessel analog filter design

Syntax

```
[b,a] = besself(n,Wo)
[b,a] = besself(n,Wo,ftype)

[z,p,k] = besself( ___ )
[A,B,C,D] = besself( ___ )
```

Description

`[b,a] = besself(n,Wo)` returns the transfer function coefficients of an `n`th-order lowpass analog Bessel filter, where `Wo` is the angular frequency up to which the filter's group delay is approximately constant. Larger values of `n` produce a group delay that better approximates a constant up to `Wo`. The `besself` function does not support the design of digital Bessel filters.

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

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

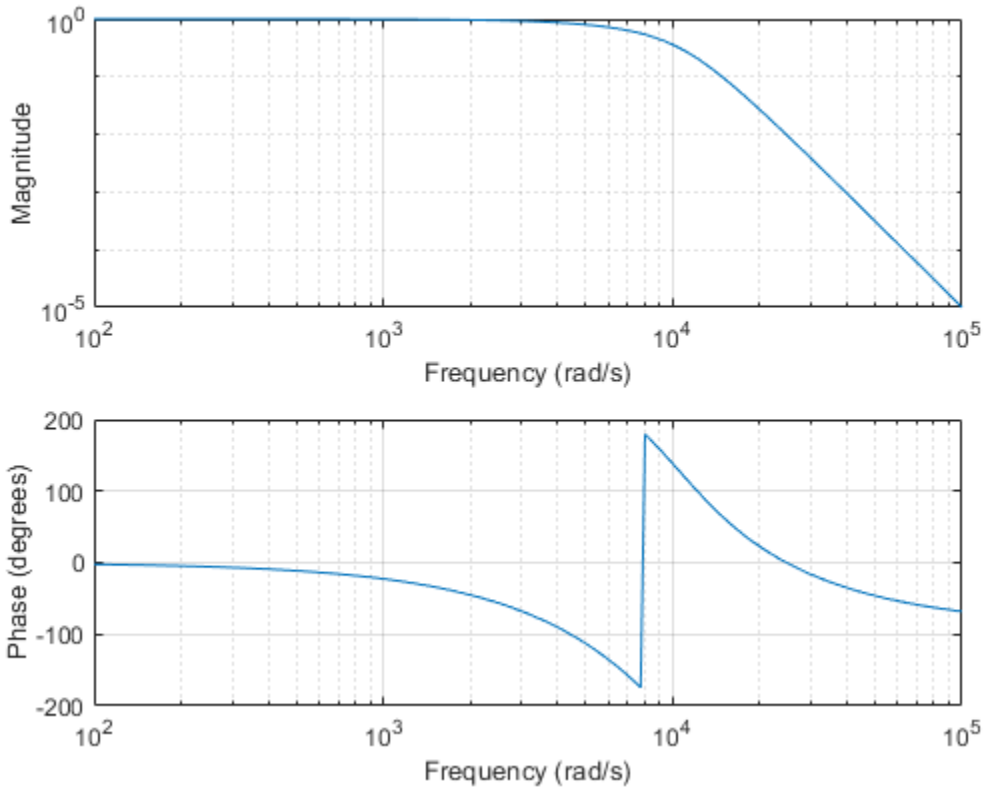
`[A,B,C,D] = besself( ___ )` designs a lowpass, highpass, bandpass, or bandstop analog Bessel filter and returns the matrices that specify its state-space representation.

Examples

Frequency Response of Lowpass Bessel Filter

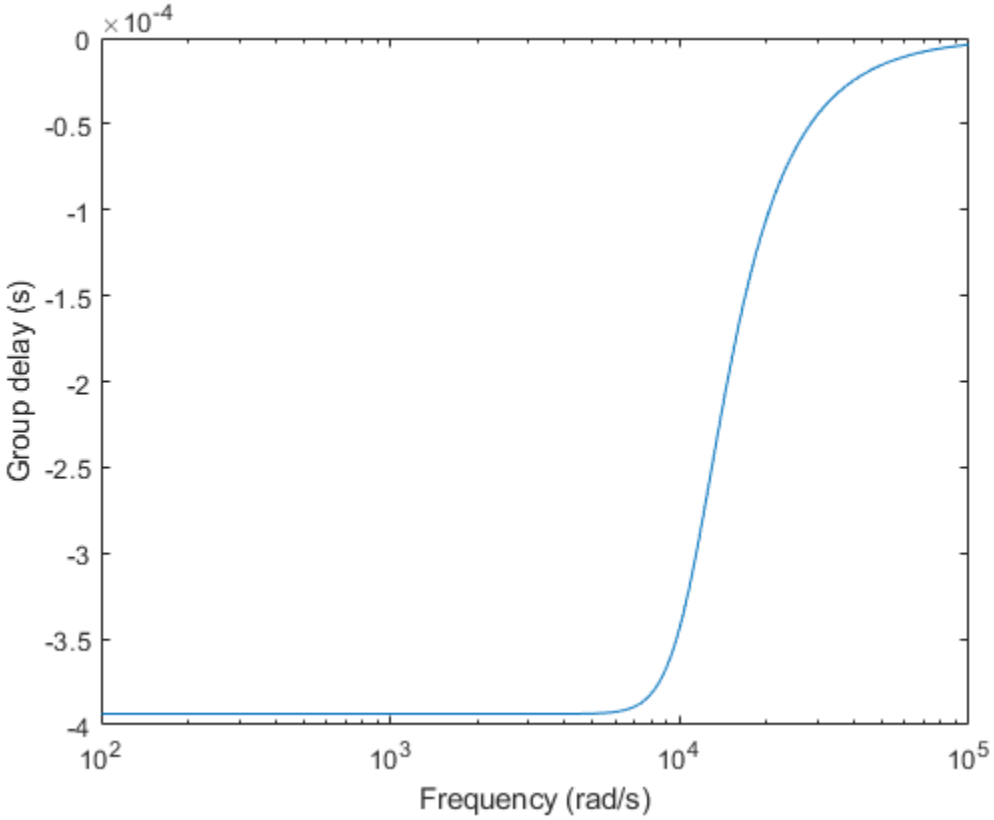
Design a fifth-order analog lowpass Bessel filter with approximately constant group delay up to 10^4 rad/second. Plot the magnitude and phase responses of the filter using `freqs`.

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



Compute the group delay response of the filter as the derivative of the unwrapped phase response. Plot the group delay to verify that it is approximately constant up to the cutoff frequency.

```
[h,w] = freqs(b,a,1000);
grpdel = diff(unwrap(angle(h)))./diff(w);

clf
semilogx(w(2:end),grpdel)
xlabel('Frequency (rad/s)')
ylabel('Group delay (s)')
```



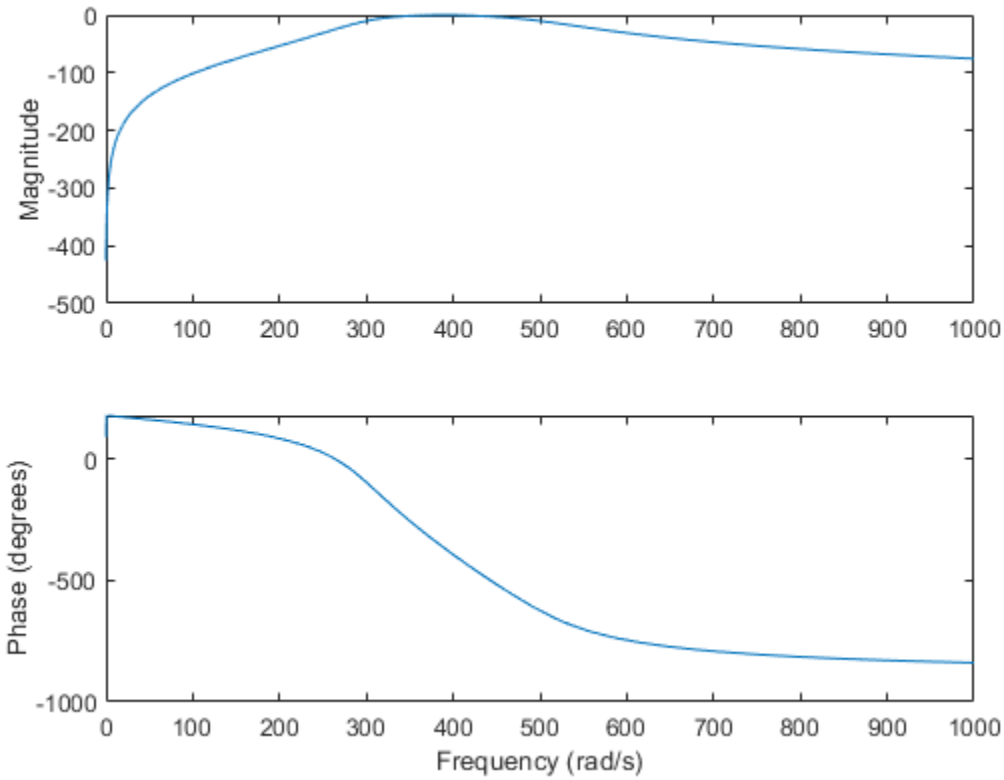
Bandpass Bessel Filter

Design a 12th-order bandpass Bessel filter with the passband ranging from 300 rad/s to 500 rad/s. Compute the frequency response of the filter.

```
[b,a] = besself(6,[300 500],'bandpass');

[h,w] = freqs(b,a);
```

Plot the magnitude and phase responses of the filter. Unwrap the phase response to avoid 180˚ and 360˚ jumps and convert it from radians to degrees. As expected, the phase response is close to linear over the passband.

```
subplot(2,1,1)
plot(w,20*log10(abs(h)))
ylabel('Magnitude')
subplot(2,1,2)
plot(w,180*unwrap(angle(h))/pi)
ylabel('Phase (degrees)')
xlabel('Frequency (rad/s)')
```



Input Arguments

`n` — Filter order

integer scalar

Filter order, specified as an integer scalar. For bandpass and bandstop designs, `n` represents one-half the filter order.

Data Types: `double`

`Wo` — Cutoff frequency

scalar | two-element vector

Cutoff frequency, specified as a scalar or a two-element vector. A cutoff frequency is an upper or lower bound of the frequency range in which the filter's group delay is approximately constant. Cutoff frequencies must be expressed in radians per second and can take on any positive value.

- If `Wo` is scalar, then `besself` designs a lowpass or highpass filter with cutoff frequency `Wo`.
- If `Wo` is a two-element vector `[w1 w2]`, where `w1 < w2`, then `besself` designs a bandpass or bandstop filter with lower cutoff frequency `w1` and higher cutoff frequency `w2`.

Data Types: `double`

`ftype` — Filter type

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

Filter type, specified as:

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

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 2`n` + 1 for bandpass and bandstop filters. The transfer function is expressed in terms of `b` and `a` as

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

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` (2`n` for bandpass and bandstop designs) and a scalar. The transfer function is expressed in terms of `z`, `p`, and `k` as

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

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 = 2`n` for bandpass and bandstop filters, then `A` is $m \times m$, `B` is $m \times 1$, `C` is $1 \times m$, and `D` is 1×1.

The state-space matrices relate the state vector x, the input u, and the output y through

$$\dot{x} = \mathrm{A}\,x + \mathrm{B}\,u$$
$$y = \mathrm{C}\,x + \mathrm{D}\,u.$$

Data Types: `double`

Algorithms

`besself` designs analog Bessel filters, which are characterized by an almost constant group delay across the entire passband, thus preserving the wave shape of filtered signals in the passband.

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.

`besself` uses a four-step algorithm:

1. Find lowpass analog prototype poles, zeros, and gain using the `besselap` function.
2. Convert the poles, zeros, and gain into state-space form.
3. If required, use a state-space transformation to convert the lowpass filter into a bandpass, highpass, or bandstop filter with the desired frequency constraints.
4. Convert the state-space filter back to transfer function or zero-pole-gain form, as required.

References

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

See Also

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

Introduced before R2006a

bilinear

Bilinear transformation method for analog-to-digital filter conversion

Syntax

```
[zd,pd,kd] = bilinear(z,p,k,fs)
[numd,dend] = bilinear(num,den,fs)
[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs)
[___] = bilinear(___,fp)
```

Description

`[zd,pd,kd] = bilinear(z,p,k,fs)` converts the *s*-domain transfer function in pole-zero form specified by `z`, `p`, `k` and sample rate `fs` to a discrete equivalent.

`[numd,dend] = bilinear(num,den,fs)` converts the *s*-domain transfer function specified by numerator `num` and denominator `den` to a discrete equivalent.

`[Ad,Bd,Cd,Dd] = bilinear(A,B,C,D,fs)` converts the continuous-time state-space system in matrices `A`, `B`, `C`, and `D` to a discrete-time system.

`[___] = bilinear(___,fp)` uses parameter `fp` as "match" frequency to specify prewarping.

Examples

Bandpass IIR Filter Design Using Chebyshev Type I Analog Filter

Design the prototype for a 10th-order Chebyshev type I bandpass filter with 3 dB of ripple in the passband. Convert it to state-space form.

```
[z,p,k] = cheb1ap(10,3);
[A,B,C,D] = zp2ss(z,p,k);
```

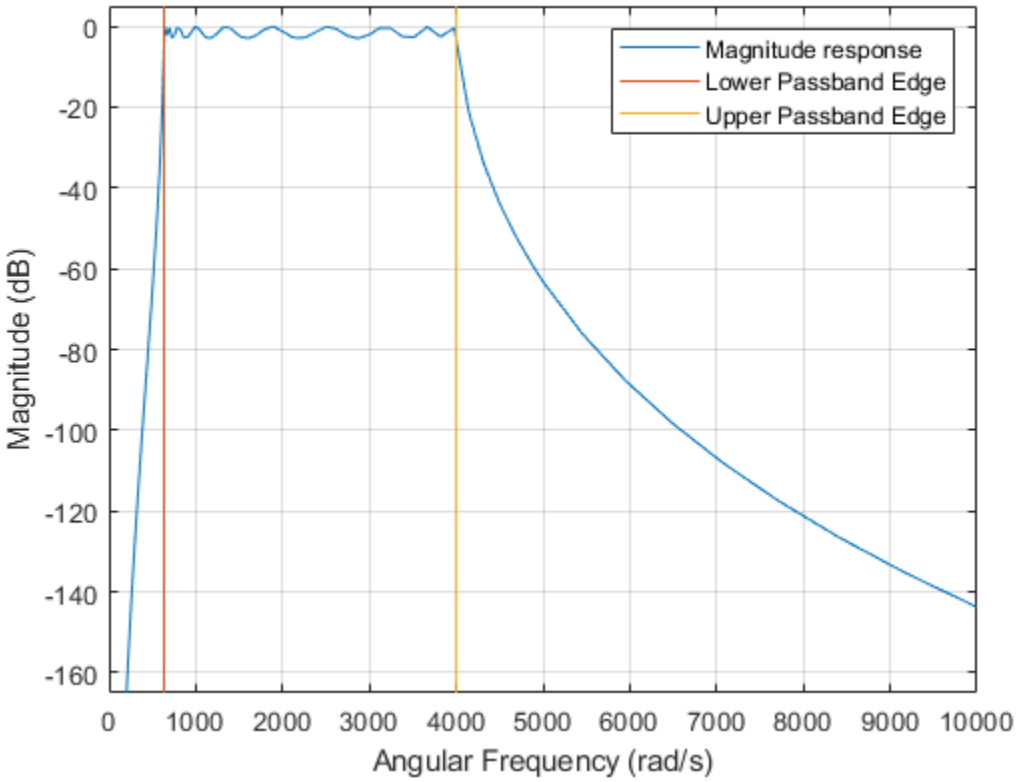
Create an analog filter with sample rate f_s = 2 kHz, prewarped band edges u_1 and u_2 in rad/s, bandwidth $B_w = u_2 - u_1$ and center frequency $W_o = \sqrt{u_1 u_2}$ for use with `lp2bp`. Specify the passband edge frequencies as 100 Hz and 500 Hz.

```
Fs = 2e3;
u1 = 2*Fs*tan(100*(2*pi/Fs)/2);
u2 = 2*Fs*tan(500*(2*pi/Fs)/2);
Bw = u2 - u1;
Wo = sqrt(u1*u2);
[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw);
[b,a] = ss2tf(At,Bt,Ct,Dt);
```

Calculate the frequency response of the analog filter using `freqs`. Plot the magnitude response and the prewarped frequency band edges.

```
[h,w] = freqs(b,a);
plot(w,mag2db(abs(h)))
```

```
hold on
ylim([-165 5])
[U1,U2] = meshgrid([u1 u2],ylim);
plot(U1,U2)
legend('Magnitude response','Lower Passband Edge','Upper Passband Edge')
hold off
xlabel('Angular Frequency (rad/s)')
ylabel('Magnitude (dB)')
grid
```



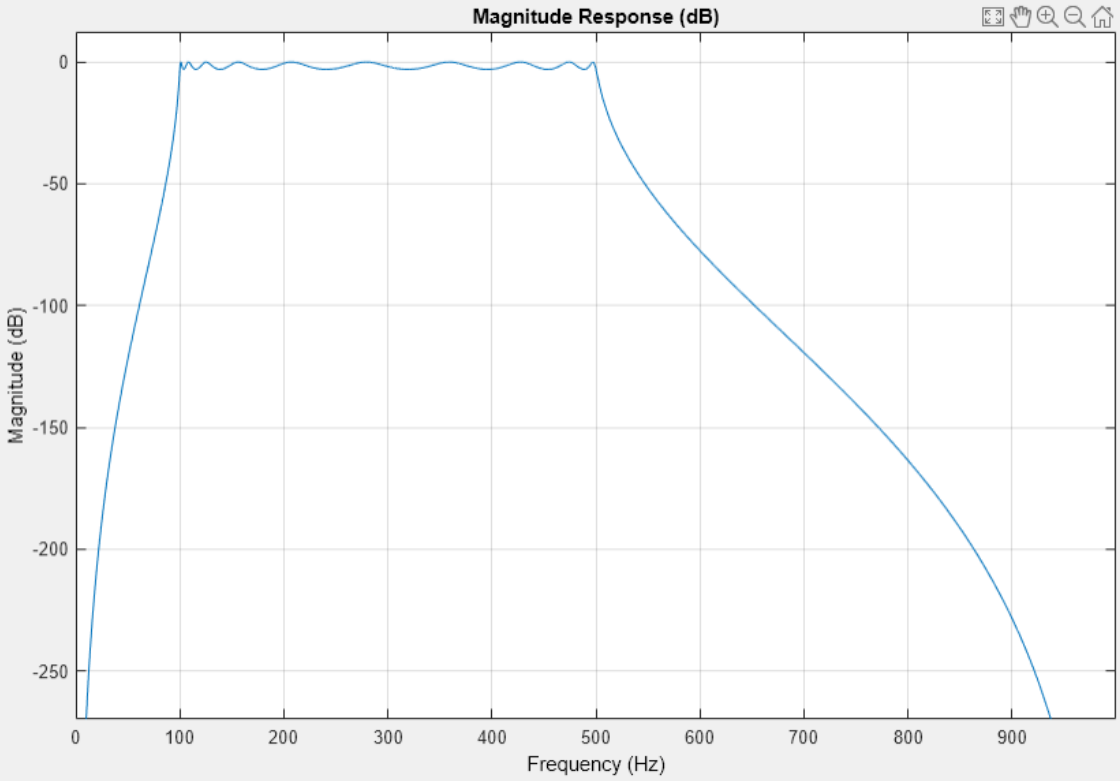
Use `bilinear` to create a digital bandpass filter with sample rate f_s and lower band edge 100 Hz. Convert the digital filter from state-space form to transfer function form using `ss2tf`.

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

```
[bz,az] = ss2tf(Ad,Bd,Cd,Dd);
```

Use `fvtool` to plot the magnitude response of the digital filter.

```
fvtool(bz,az,'Fs',Fs)
```



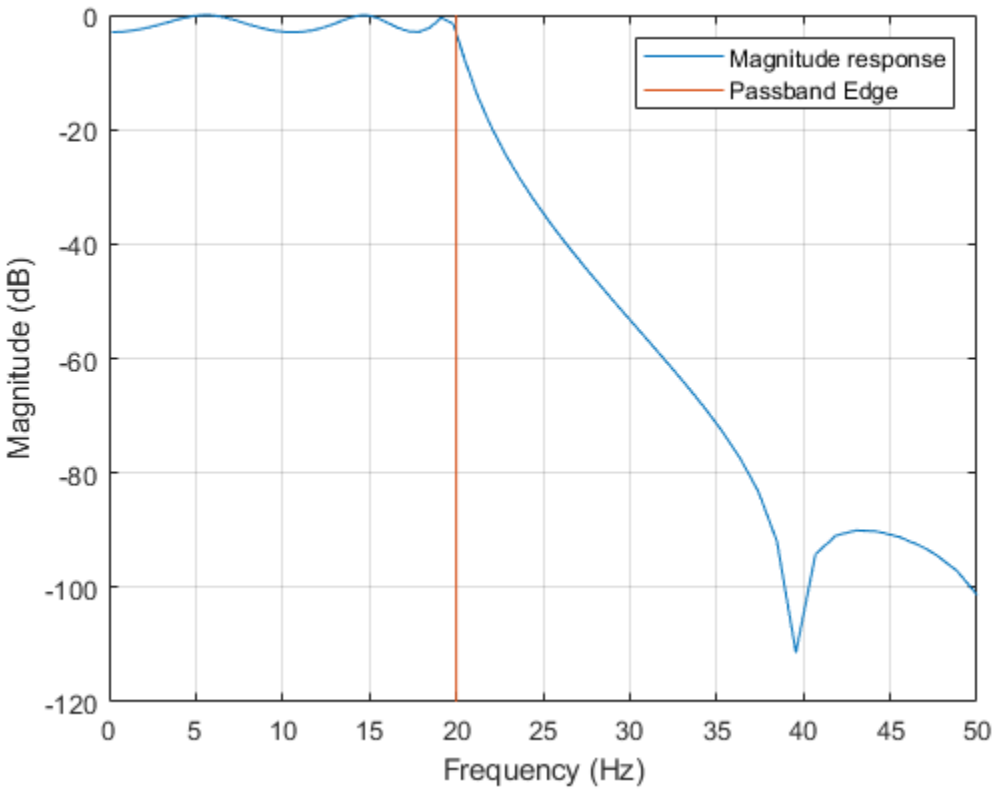
Discrete-Time Representation of an Elliptic Filter

Design a 6th-order elliptic analog lowpass filter with 3 dB of ripple in the passband and a stopband 90 dB down. Set cutoff frequency f_c = 20 Hz and sample rate f_s = 200 Hz.

```
clear
Fc = 20;
Fs = 200;
[z,p,k] = ellip(6,3,90,2*pi*Fc,'s');
[num,den] = zp2tf(z,p,k);
```

Calculate the magnitude response of the analog elliptic filter. Visualize the analog filter.

```
[h,w] = freqs(num,den);
plot(w/(2*pi),mag2db(abs(h)))
hold on
xlim([0 50])
[l1,l2] = meshgrid(Fc,[-120 0]);
plot(l1,l2)
grid
legend('Magnitude response','Passband Edge')
xlabel('Frequency (Hz)')
ylabel('Magnitude (dB)')
```

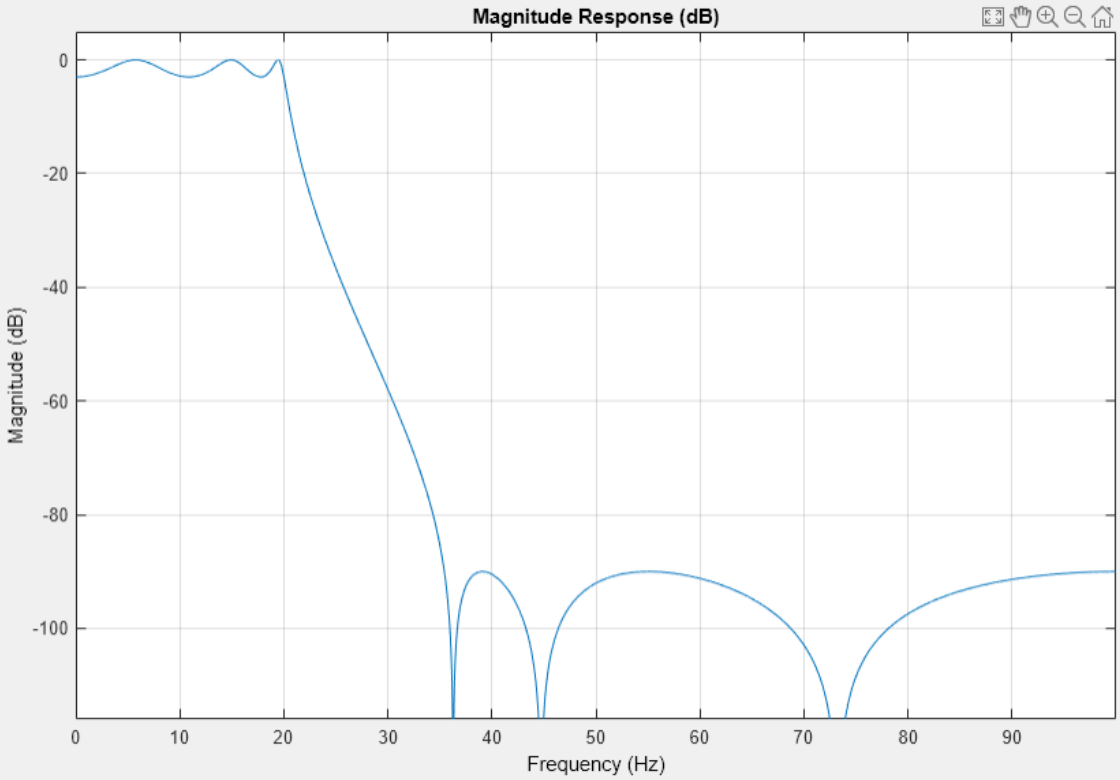


Use `bilinear` to transform it to a discrete-time IIR filter. Set the match frequency as $f_p = 20$ Hz.

```
[numd,dend] = bilinear(num,den,Fs,20);
```

Visualize the filter using `fvtool`.

```
fvtool(numd,dend,'Fs',Fs)
```



Input Arguments

z — Zeros
column vector

Zeros of the *s*-domain transfer function, specified as a column vector.

p — Poles
column vector

Poles of the *s*-domain transfer function, specified as a column vector.

k — Gain
scalar

Gain of the *s*-domain transfer function, specified as a scalar.

`fs` — Sample rate
positive scalar

Sample rate, specified as a positive scalar.

`num` — Numerator coefficients
row vector

Numerator coefficients of the analog transfer function, specified as a row vector.

den — Denominator coefficients
row vector

Denominator coefficients of the analog transfer function, specified as a row vector.

A — State matrix
matrix

State matrix in the *s*-domain, specified as a matrix. If the system has *p* inputs and *q* outputs and is described by *n* state variables, then `A` is *n*-by-*n*.

Data Types: `single` | `double`

B — Input-to-state matrix
matrix

Input-to-state matrix in the *s*-domain, specified as a matrix. If the system has *p* inputs and *q* outputs and is described by *n* state variables, then `B` is *n*-by-*p*.

Data Types: `single` | `double`

C — State-to-output matrix
matrix

State-to-output matrix in the *s*-domain, specified as a matrix. If the system has *p* inputs and *q* outputs and is described by *n* state variables, then `C` is *q*-by-*n*.

Data Types: `single` | `double`

D — Feedthrough matrix
matrix

Feedthrough matrix in the *s*-domain, specified as a matrix. If the system has *p* inputs and *q* outputs and is described by *n* state variables, then `D` is *q*-by-*p*.

Data Types: `single` | `double`

fp — match frequency
positive scalar

Match frequency, specified as a positive scalar.

Output Arguments

zd — Zeros
column vector

Zeros of the *z*-domain transfer function, specified as a column vector.

pd — Poles
column vector

Poles of the *z*-domain transfer function, specified as a column vector.

kd — Gain
scalar

Gain of the *z*-domain transfer function, specified as a scalar.

numd — Numerator coefficients
row vector

Numerator coefficients of the digital transfer function, specified as a row vector.

dend — Denominator coefficients
row vector

Denominator coefficients of the digital transfer function, specified as a row vector.

Ad — State matrix
matrix

State matrix in the *z*-domain, returned as a matrix. If the system is described by *n* state variables, then `Ad` is *n*-by-*n*.

Data Types: `single` | `double`

Bd — Input-to-state matrix
matrix

Input-to-state matrix in the *z*-domain, returned as a matrix. If the system is described by *n* state variables, then `Bd` is *n*-by-1.

Data Types: `single` | `double`

Cd — State-to-output matrix
matrix

State-to-output matrix in the *z*-domain, returned as a matrix. If the system has *q* outputs and is described by *n* state variables, then `Cd` is *q*-by-*n*.

Data Types: `single` | `double`

Dd — Feedthrough matrix
matrix

Feedthrough matrix in the *z*-domain, returned as a matrix. If the system has *q* outputs, then `Dd` is *q*-by-1.

Data Types: `single` | `double`

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.
```

Algorithms

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^{jw}, from $\omega = -\pi$ to π) by

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

`bilinear` can accept an optional parameter `Fp` that specifies prewarping. `fp`, in hertz, indicates a "match" frequency, that is, a frequency for which the frequency responses before and after mapping match exactly. In prewarped mode, the bilinear transformation maps the *s*-plane into the *z*-plane with

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

With the prewarping option, `bilinear` maps the $j\Omega$ axis (from $\Omega = -\infty$ to $+\infty$) repeatedly around the unit circle ($e^{j\omega}$, from $\omega = -\pi$ to π) by

$$\omega = 2\tan^{-1}\left(\frac{\Omega\tan\left(\pi\frac{f_p}{f_s}\right)}{2\pi f_p}\right).$$

In prewarped mode, `bilinear` matches the frequency $2\pi f_p$ (in radians per second) in the *s*-plane to the normalized frequency $2\pi f_p/f_s$ (in radians per second) in the *z*-plane.

The `bilinear` function works with three different linear system representations: zero-pole-gain, transfer function, and state-space form.

`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 ±∞ 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

An analog system in state space form is given by

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

. This system is converted to the discrete form using state-space equations as follows:

$$x[n+1] = A_d x[n] + B_d u[n],$$
$$y[n] \quad = C_d x[n] + D_d u[n]\,.$$

To convert an analog 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 `fp` is not present, let λ=*fs*.

2 Compute `Ad`, `Bd`, `Cd`, and `Dd` in terms of `A`, `B`, `C`, and `D` using

$$A_d = ({}^{I}(I + A\tfrac{1}{2\lambda}),$$
$$B_d = \tfrac{1}{\sqrt{\lambda}}({}^{I}B,$$
$$C_d = \tfrac{1}{\sqrt{\lambda}}C({}^{I},$$
$$D_d = \tfrac{1}{2\lambda}C({}^{I}B + D\,.$$

Transfer Function

For a system in transfer function form, `bilinear` converts 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:

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

`fs` is the sample rate 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.

References

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

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`impinvar` | `lp2bp` | `lp2bs` | `lp2hp` | `lp2lp`

Introduced before R2006a

bitrevorder

Permute data into bit-reversed order

Syntax

```
y = bitrevorder(x)
[y,i] = bitrevorder(x)
```

Description

`y = bitrevorder(x)` returns the input data in bit-reversed order.

`[y,i] = bitrevorder(x)` also returns the bit-reversed indices, `i`, such that `y = x(i)`.

Examples

Vector in Bit-Reversed Order

Create a column vector and obtain its bit-reversed version. Verify by displaying the binary representation explicitly.

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

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

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

```
T=16×4 table
    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
```

Input Arguments

x — Input data

vector | matrix

Input data, specified as a vector or matrix. The length or number of rows 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.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

y — Bit-reversed data

vector | matrix

Bit-reversed data, returned as a vector or matrix. `y` is the same size as `x`.

i — Bit-reversed indices

vector | matrix

Bit-reversed indices, returned as a vector or matrix such that `y = x(i)`. MATLAB® matrices use 1-based indexing, so the first index of `y` is 1, not 0.

More About

Bit-Reversed Ordering

`bitrevorder` is useful for prearranging filter coefficients so that bit-reversed ordering does not have to be performed as part of an `fft` or `ifft` computation.

Bit-reversed ordering 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.

This table shows the numbers 0 through 7, the corresponding bits, and the bit-reversed numbers.

Linear Index	Bits	Bit-Reversed	Bit-Reversed Index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`fft` | `digitrevorder` | `ifft`

Introduced before R2006a

blackman

Blackman window

Syntax

```
w = blackman(L)
w = blackman(L,sflag)
```

Description

`w = blackman(L)` returns an `L`-point symmetric Blackman window.

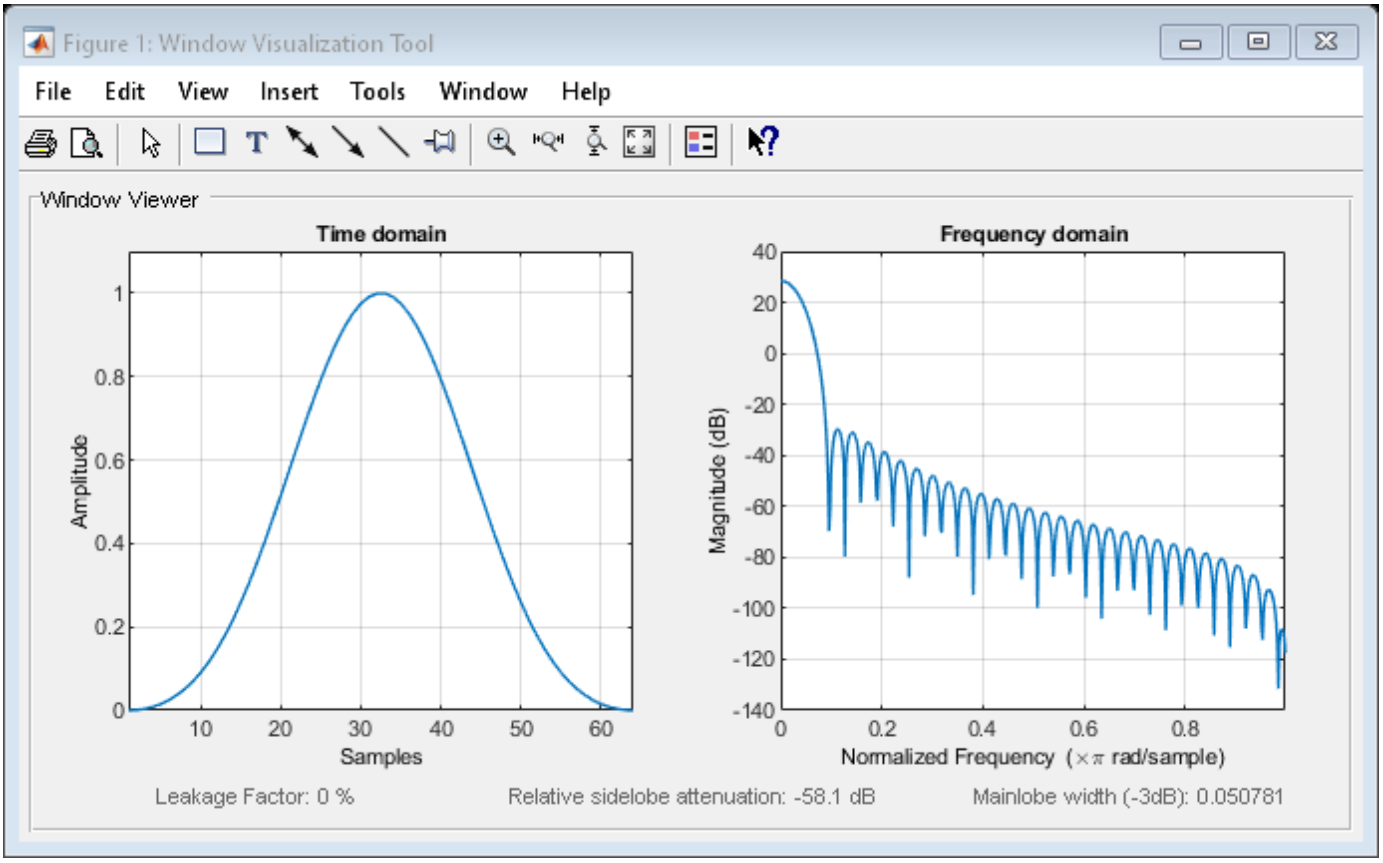
`w = blackman(L,sflag)` returns a Blackman window using the window sampling method specified by `sflag`.

Examples

Blackman Window

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

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



Input Arguments

`L` — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

`sflag` — Window sampling

`'symmetric'` (default) | `'periodic'`

Window sampling method, specified as:

- `'symmetric'` — Use this option when using windows for filter design.
- `'periodic'` — This option is useful for spectral analysis because it enables a windowed signal to have the perfect periodic extension implicit in the discrete Fourier transform. When `'periodic'` is specified, the function computes a window of length `L + 1` and returns the first `L` points.

Output Arguments

`w` — Blackman window

column vector

Blackman window, returned as a column vector.

Algorithms

The following equation defines the Blackman window of length N:

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

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

In the symmetric case, the second half of the Blackman window, $M \le n \le N$ – 1, is obtained by reflecting 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. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1999, pp. 468–471.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps

Window Designer

Functions

`flattopwin` | `hamming` | `hann` | **WVTool**

Introduced before R2006a

blackmanharris

Minimum four-term Blackman-Harris window

Syntax

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

Description

`w = blackmanharris(N)` returns an `N`-point symmetric four-term Blackman-Harris window.

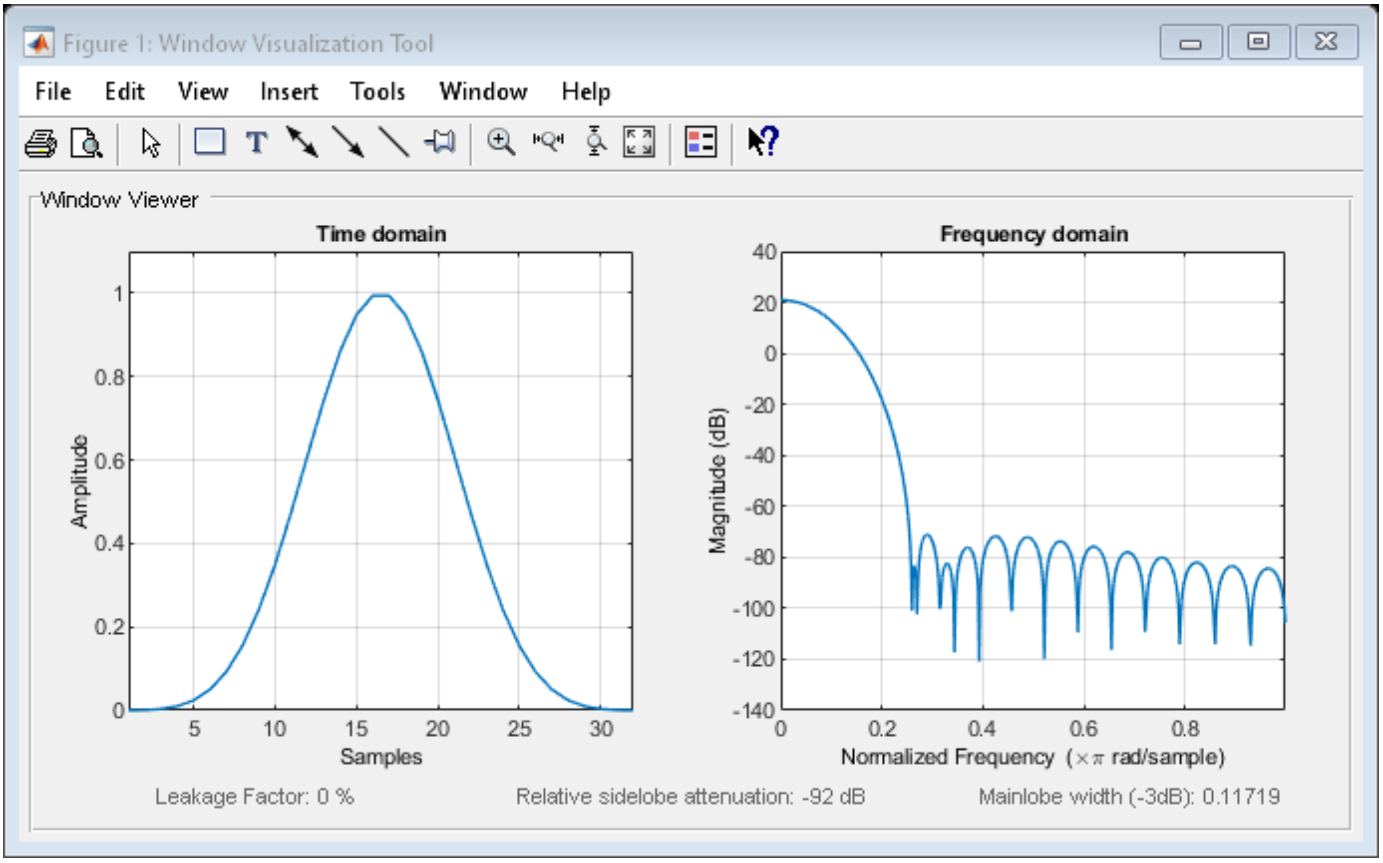
`w = blackmanharris(N,sflag)` returns a Blackman-Harris window using the window sampling method specified by `sflag`.

Examples

Blackman-Harris Window

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

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



Input Arguments

`N` — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

`sflag` — Window sampling

`'symmetric'` (default) | `'periodic'`

Window sampling method, specified as:

- `'symmetric'` — Use this option when using windows for filter design.
- `'periodic'` — This option is useful for spectral analysis because it enables a windowed signal to have the perfect periodic extension implicit in the discrete Fourier transform. When `'periodic'` is specified, the function computes a window of length `L + 1` and returns the first `L` points.

Output Arguments

`w` — Blackman-Harris window

column vector

Blackman-Harris window, returned as a column vector.

Algorithms

The equation for the symmetric four-term Blackman-Harris window of length N is

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

The equation for the periodic four-term Blackman-Harris window of length N is

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

The periodic window is N-periodic.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps

Window Designer

Functions

`barthannwin` | `bartlett` | `bohmanwin` | `nuttallwin` | `parzenwin` | `rectwin` | `triang` | **WVTool**

Introduced before R2006a

bohmanwin

Bohman window

Syntax

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

Description

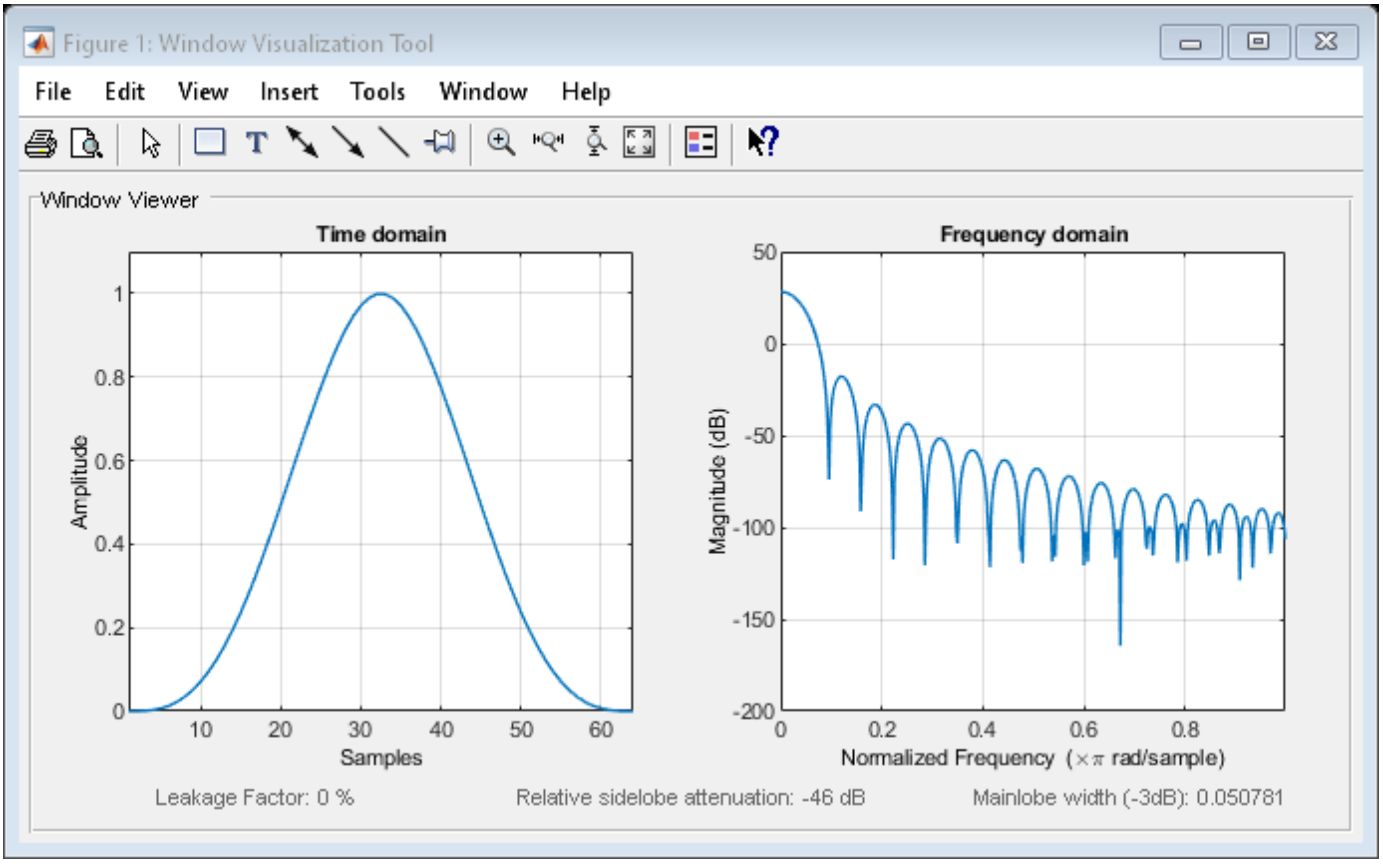
`w = bohmanwin(L)` returns an L-point Bohman window in `w`.

Examples

Bohman Window

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

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



Input Arguments

L — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

Output Arguments

w — Bohman window

column vector

Bohman window, returned as a column vector.

Algorithms

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 $1/w^4$.The equation for computing the coefficients of a Bohman window is

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

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps
Window Designer

Functions
`barthannwin` | `bartlett` | `blackmanharris` | `nuttallwin` | `parzenwin` | `rectwin` | `triang` | **WVTool**

Introduced before R2006a

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`.

`y = buffer(x,n,p)` overlaps or underlaps successive frames in the output matrix by `p` samples.

`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.

`[y,z] = buffer(___)` partitions the length-`L` signal vector `x` into frames of length `n`, and outputs only the full frames in `y`. The vector `z` contains the remaining samples. This syntax can include any combination of input arguments from the previous syntaxes.

`[y,z,opt] = buffer(___)` returns the last `p` samples of an overlapping buffer in output `opt`.

Examples

Continuous Overlapping Buffers

Create a buffer containing 100 frames, each with 11 samples.

```
data = buffer(1:1100,11);
```

Take the frames (columns) in the matrix `data` to be the sequential outputs of a data acquisition board sampling a physical signal: `data(:,1)` is the first A/D 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. 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. Specify a value of –5 for `y(1)`. The carryover vector is empty initially.

```
n = 4;
p = 1;
opt = -5;
z = [];
```

Now repeatedly call `buffer`, each time passing in a new signal frame (column) from `data`. Overflow samples (returned in `z`) are carried over and prepended to the input in the subsequent call to `buffer`.

For the first four iterations, show the input frame `[z;x]'`, the input and output values of `opt`, the output buffer `y`, and the overflow `z`. 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.

```
for i = 1:size(data,2)
    x = data(:,i);
    [y,z,oppt] = buffer([z;x],n,p,opt);
    if i <= 4
        disp(' '), i, ifrm = [z;x]', opts = [opt oppt], y, z, disp(' ')
   end
   opt = oppt;
end
```

```
i = 1

ifrm = 1×13

    10    11     1     2     3     4     5     6     7     8     9    10    11


opts = 1×2

    -5     9


y = 4×3

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


z = 2×1

    10
    11
```

```
i = 2

ifrm = 1×12

    22    12    13    14    15    16    17    18    19    20    21    22


opts = 1×2

     9    21
```

```
y = 4×4

     9    12    15    18
    10    13    16    19
    11    14    17    20
    12    15    18    21


z = 22



i = 3

ifrm = 1×11

    23    24    25    26    27    28    29    30    31    32    33


opts = 1×2

    21    33


y = 4×4

    21    24    27    30
    22    25    28    31
    23    26    29    32
    24    27    30    33


z =

  0x1 empty double column vector



i = 4

ifrm = 1×13

    43    44    34    35    36    37    38    39    40    41    42    43    44


opts = 1×2

    33    42


y = 4×3

    33    36    39
    34    37    40
    35    38    41
    36    39    42
```

```
z = 2×1

    43
    44
```

Continuous Underlapping Buffers

Create a buffer containing 100 frames, each with 11 samples.

```
data = buffer(1:1100,11);
```

Take `data(:,1)` as the first A/D output, containing the first 11 signal samples, `data(:,2)` as 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. Specify a new frame size of 4 and an underlap of –2. Skip the first input element, `x(1)`, by setting `opt` to 1. The carryover vector is empty initially.

```
n = 4;
p = -2;
opt = 1;
z = [];
```

Now repeatedly call `buffer`, each time passing in a new signal frame (column) from `data`. Overflow samples (returned in `z`) are carried over and prepended to the input in the subsequent call to `buffer`.

For the first three iterations, show the input frame `[z';x]'`, the input and output values of `opt`, the output buffer `y`, and the overflow `z`. 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.

```
for i = 1:size(data,2)
   x = data(:,i);
   [y,z,oppt] = buffer([z';x],n,p,opt);
   if i <= 3
       disp(' '), i, ifrm = [z';x]', opts = [opt oppt], y, z, disp(' ')
   end
   opt = oppt;
end
```

```
i = 1

ifrm = 1×11

     1     2     3     4     5     6     7     8     9    10    11
```

```
opts = 1×2

     1     2


y = 4×2

     2     8
     3     9
     4    10
     5    11


z =

  1x0 empty double row vector



i = 2

ifrm = 1×14

    20    21    22    12    13    14    15    16    17    18    19    20    21    22


opts = 1×2

     2     0


y = 4×1

    14
    15
    16
    17


z = 1×3

    20    21    22



i = 3

ifrm = 1×13

    32    33    23    24    25    26    27    28    29    30    31    32    33


opts = 1×2

     0     0
```

```
y = 4×2

    20    26
    21    27
    22    28
    23    29


z = 1×2

    32    33
```

Input Arguments

x — Input signal

vector

Input signal, specified as a vector.

n — Frame length

positive real scalar

Frame length, specified as a positive real scalar.

p — Number of samples

positive real scalar

Number of samples, specified as a real positive scalar.

- 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 `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))`.

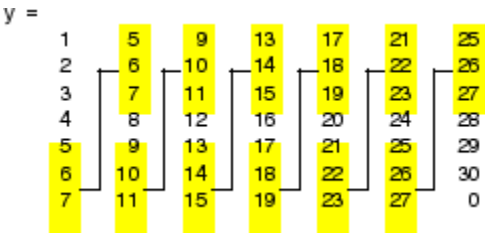
`opt` — Option

`zeros(p,1)` (default) | `'nodelay'` | vector | integer

Option, specified as a vector or integer.

- 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 1-105 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.



- 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 1-105 for an example of how this works in practice.

Output Arguments

`y` — Data frame

matrix

Data frame, returned as a matrix. Each data frame occupies one column of `y`, which has `n` rows and `ceil(L/n)` columns. If `L` is not evenly divisible by `n`, the last column is zero-padded to length `n`.

- 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 = ceil((L-p)/(n-p))` 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)`.

`z` — Remaining samples

vector

Remaining samples, returned as a vector. 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 `L - m*(n-p)` when `length(opt) = p` or `L - ((m-1)*(n-p)+n)` when `opt = 'nodelay'`, and for an underlapping buffer has length `(L-opt) - m*(n-p)`.

- If `y` is an overlapping buffer or a nonoverlapping buffer, then `z` has the same orientation (row or column) as `x`.
- If `y` is an underlapping buffer, then `z` is returned as a row vector.

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.

`opt` — Last `p` samples
vector

Last `p` samples, returned as a vector. 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. 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 1-105 for an example of how this works in practice.

- 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.
- 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`, where `opt` on the right is the input argument to `buffer`, and `opt` on the left is the output argument. `z` is the empty vector. Here `m` is the number of columns in the buffer, with `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.

Diagnostics

Error messages are displayed when `p ≥n` or `length(opt)≠length(p)` in an overlapping buffer case:

```
Frame overlap P must be less than the buffer size N.
Initial conditions must be specified as a length-P vector.
```

More About

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.

For 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 two examples on this page demonstrate how the `opt` parameter should be used.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`reshape`

Introduced before R2006a

buttap

Butterworth filter prototype

Syntax

```
[z,p,k] = buttap(n)
```

Description

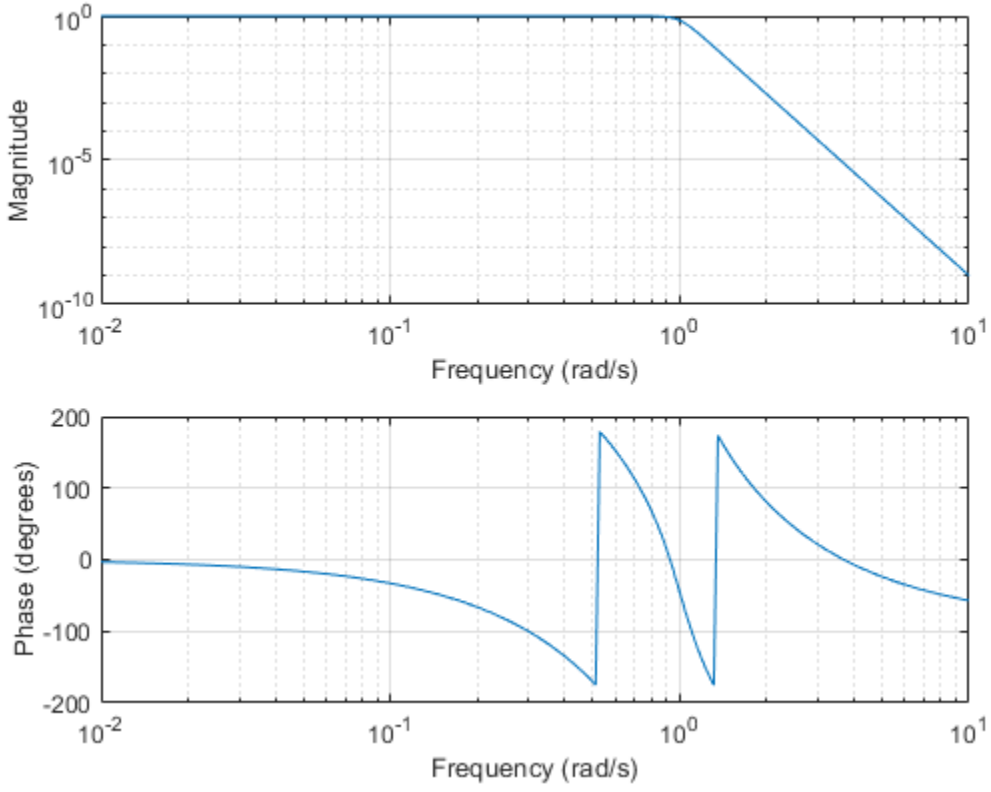
`[z,p,k] = buttap(n)` returns the poles and gain of an order `n` Butterworth analog lowpass filter prototype.

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
```



Input Arguments

n — Order of Butterworth filter

positive integer scalar

Order of Butterworth filter, specified as a positive integer scalar.

Output Arguments

z — Zeros

matrix

Zeros of the system, returned as a matrix. `z` contains the numerator zeros in its columns. `z` is an empty matrix because there are no zeros.

p — Poles

column vector

Poles of the system, returned as a column vector. `p` contains the pole locations of the denominator coefficients of the transfer function.

k — Gains

scalar

Gains of the system, returned as a scalar. `k` contains the gains for each numerator transfer function.

Algorithms

The function `buttap` 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

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

```
z = [];
p = exp(sqrt(-1)*(pi*(1:2:2*n-1)/(2*n)+pi/2)).';
k = real(prod(-p));
```

Note The function `buttap` returns zeros, poles, and gain `(z, p, and k)` in MATLAB. However, the generated C/C++ code for `buttap` returns only poles `p` and gain `k` since zeros `z` is always an empty matrix.

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 ω = 0. The squared magnitude response function is

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

corresponding to a transfer function with poles equally spaced around a circle in the left half plane. The magnitude response at the cutoff angular frequency ω_0 is always $1/\sqrt{2}$ regardless of the filter order. `buttap` sets ω_0 to 1 for a normalized result.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`besselap` | `butter` | `cheb1ap` | `cheb2ap` | `ellipap`

Introduced before R2006a

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 1-116 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`.

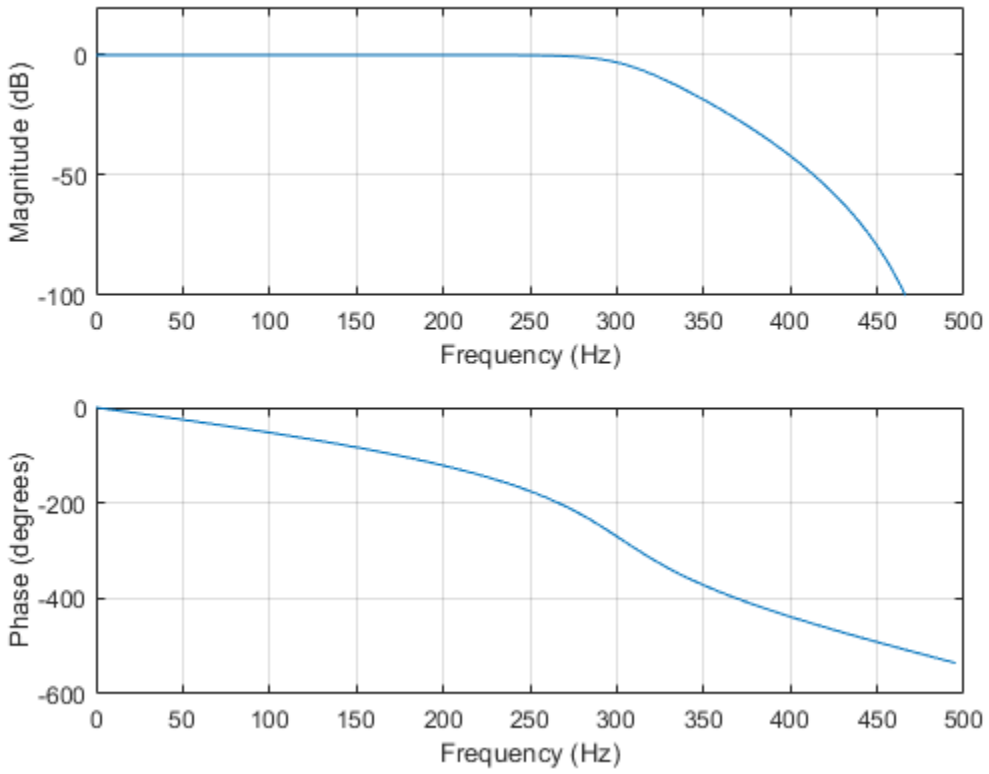
Examples

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 0.6π rad/sample. Plot its magnitude and phase responses. Use it to filter a 1000-sample random signal.

```
fc = 300;
fs = 1000;

[b,a] = butter(6,fc/(fs/2));

freqz(b,a,[],fs)
```

```
subplot(2,1,1)
ylim([-100 20])
```

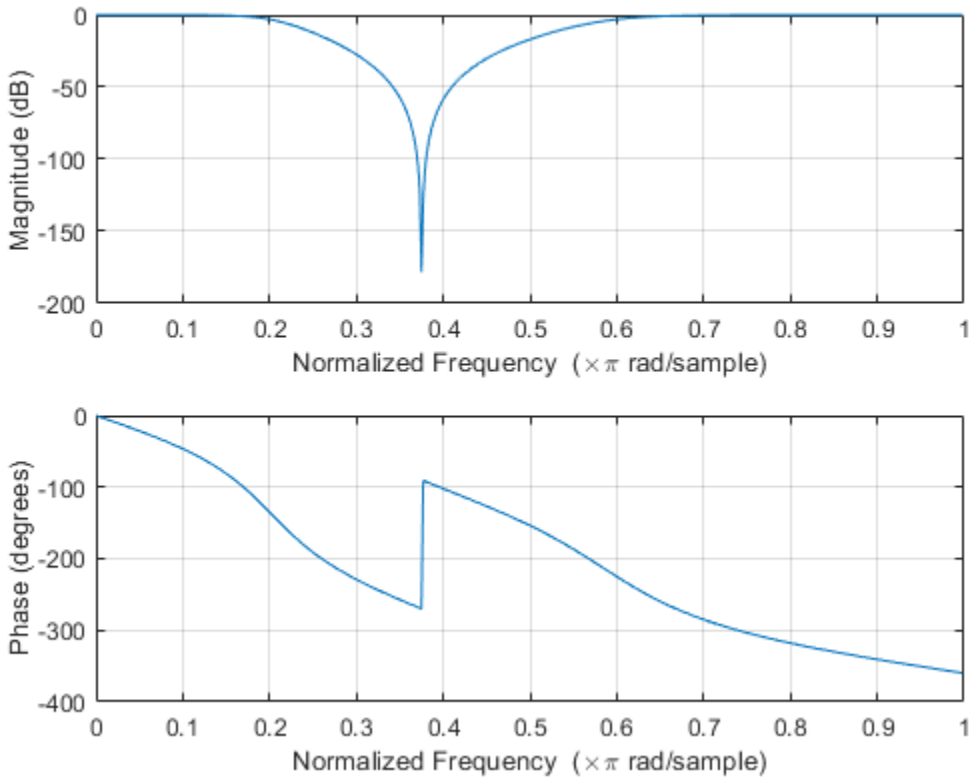



```
dataIn = randn(1000,1);
dataOut = filter(b,a,dataIn);
```

Bandstop Butterworth Filter

Design a 6th-order Butterworth bandstop filter with normalized edge frequencies of 0.2π and 0.6π 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)
```

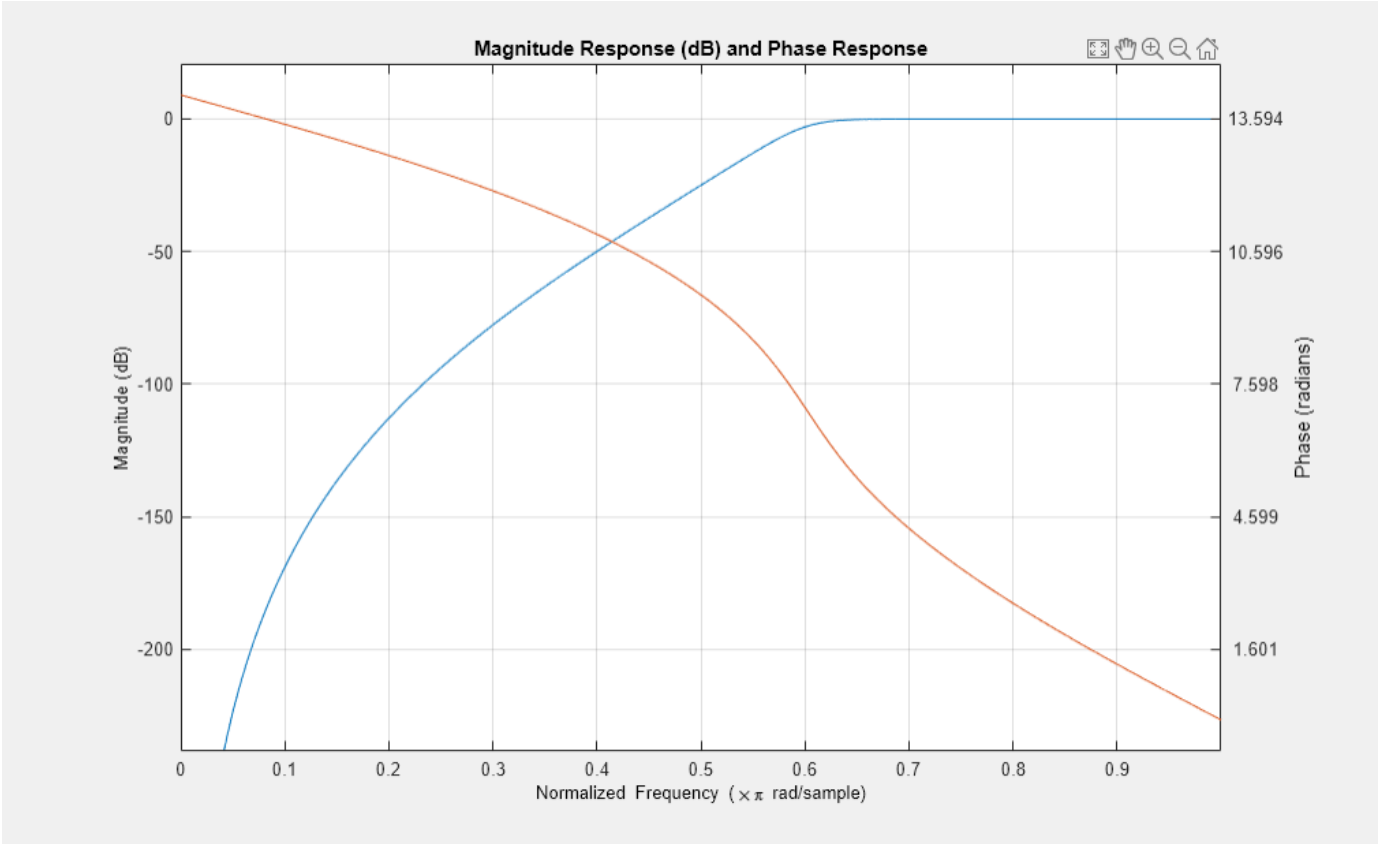



```
dataIn = randn(1000,1);
dataOut = filter(b,a,dataIn);
```

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 0.6π 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')
```



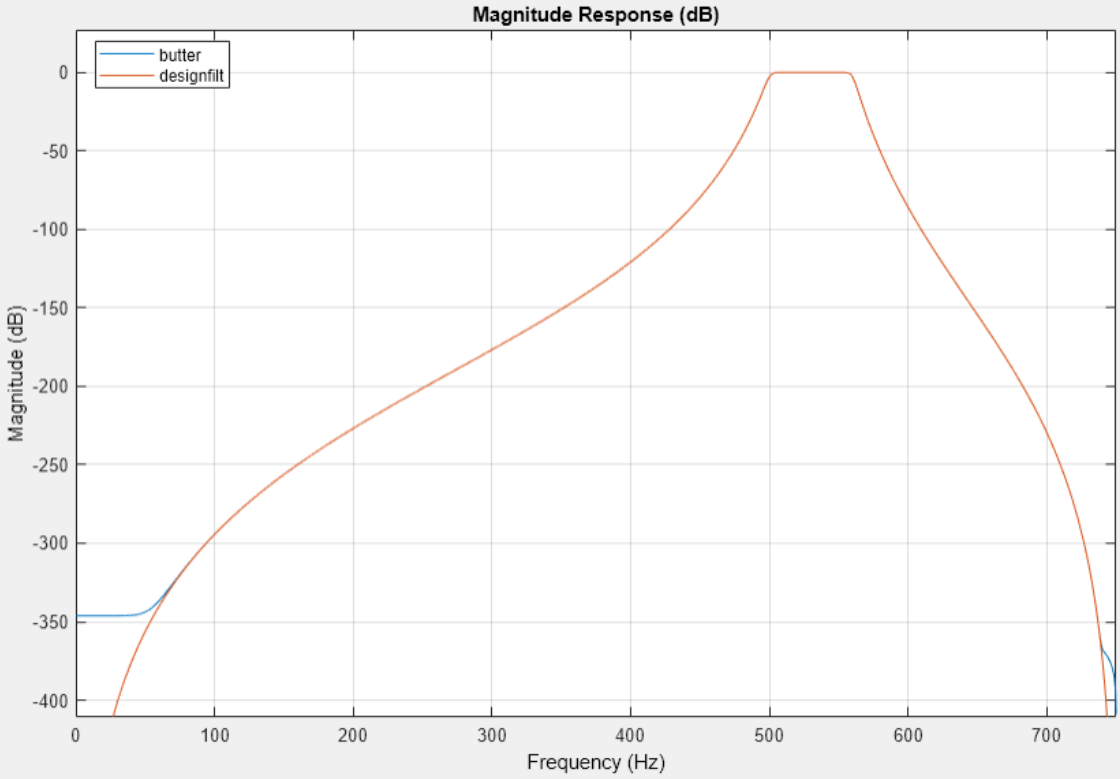
Bandpass Butterworth Filter

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);
fvt = 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 2π 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.

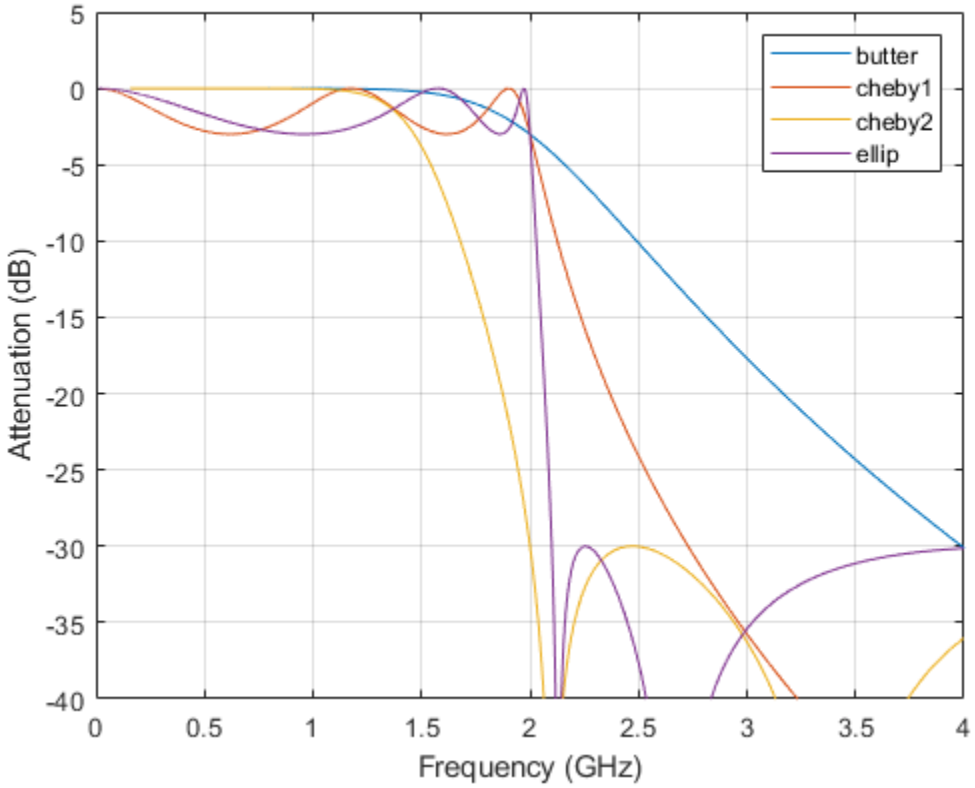
```
[z2,p2,k2] = cheby2(n,30,2*pi*f,'s');
[b2,a2] = zp2tf(z2,p2,k2);
[h2,w2] = freqs(b2,a2,4096);
```

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. For bandpass and bandstop designs, `n` represents one-half the filter order.

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 π 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 one of the following:

- `'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.

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

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

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

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

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

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

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

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

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 1×1.

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

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

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

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

Data Types: `double`

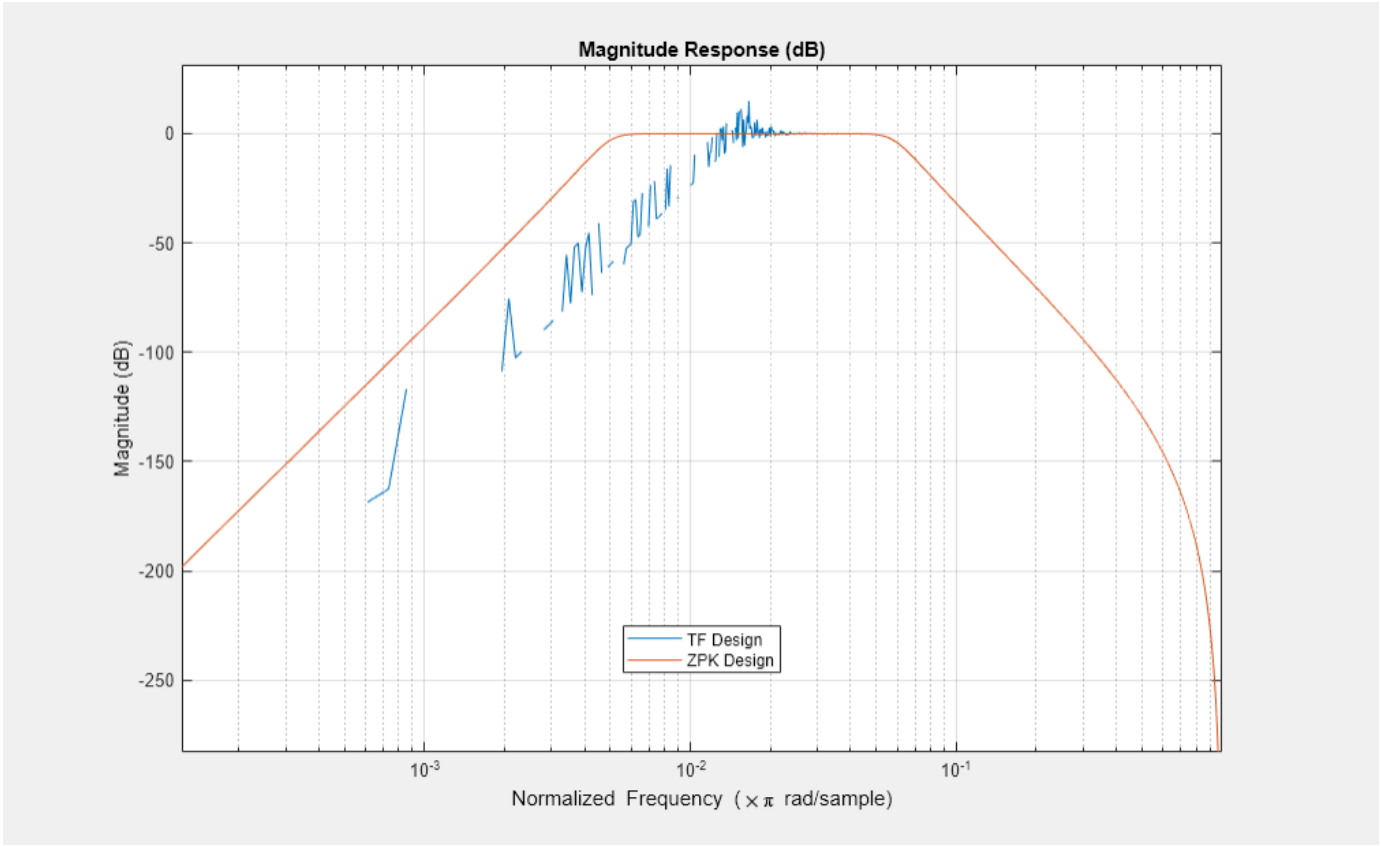
More About

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;
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 `Wn` or at `w1` and `w2`.
5. It converts the state-space filter back to its transfer function or zero-pole-gain form, as required.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

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

Introduced before R2006a

buttord

Butterworth filter order and cutoff frequency

Syntax

```
[n,Wn] = buttord(Wp,Ws,Rp,Rs)

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

Description

`[n,Wn] = buttord(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. `Wp` and `Ws` are respectively the passband and stopband edge frequencies of the filter, normalized from 0 to 1, where 1 corresponds to π rad/sample. The scalar (or vector) of corresponding cutoff frequencies, `Wn`, is also returned. To design a Butterworth filter, use the output arguments `n` and `Wn` as inputs to `butter`.

`[n,Wn] = buttord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Wn` for an analog Butterworth filter. Specify the frequencies `Wp` and `Ws` in radians per second. The passband or the stopband can be infinite.

Examples

Lowpass Butterworth Filter

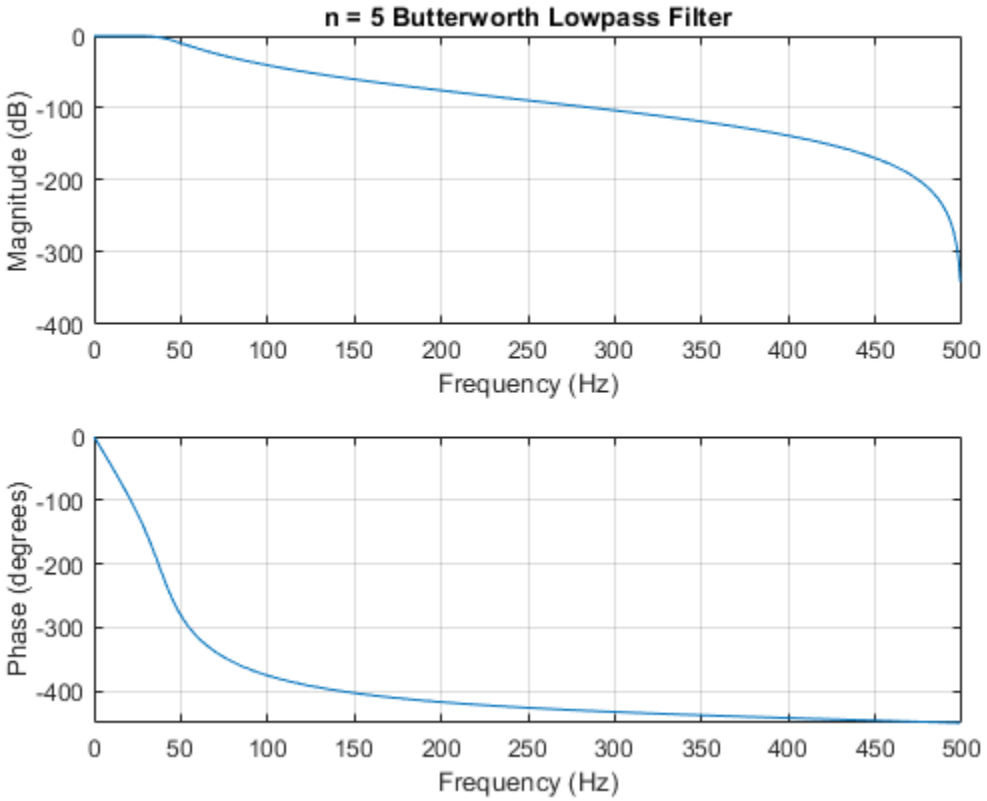
For data sampled at 1000 Hz, design a lowpass filter with no more than 3 dB of ripple in a passband from 0 to 40 Hz, and at least 60 dB of attenuation in the stopband. Find the filter order and cutoff frequency.

```
Wp = 40/500;
Ws = 150/500;

[n,Wn] = buttord(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

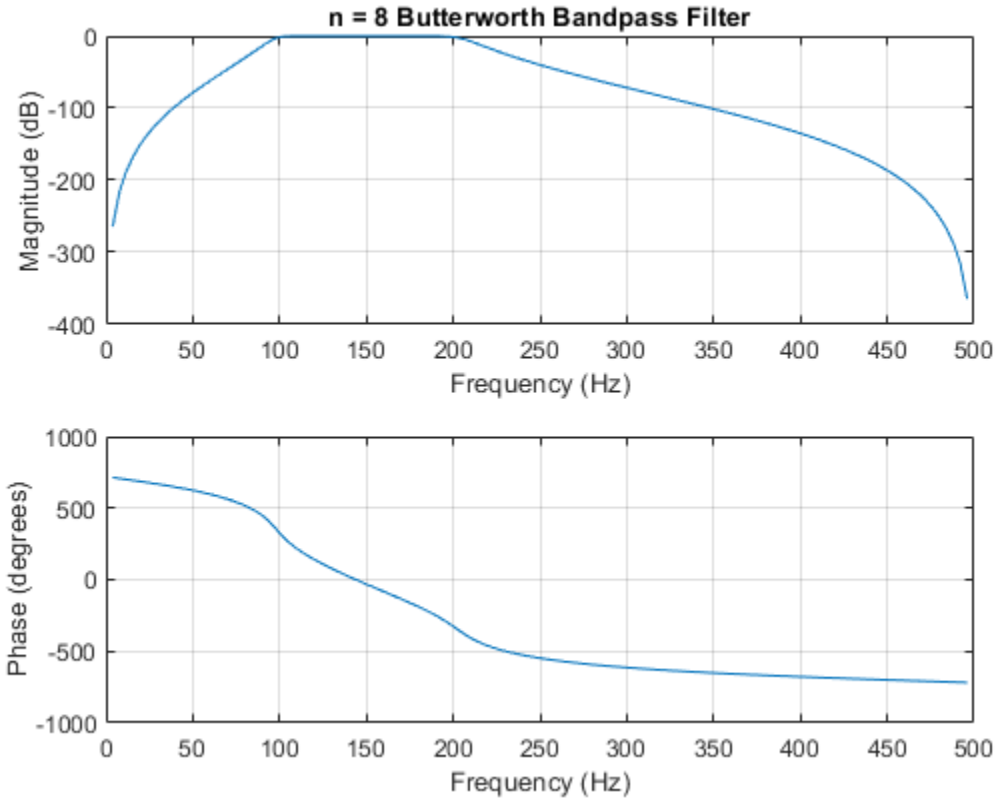
Design a bandpass filter with a passband from 100 to 200 Hz with at most 3 dB of passband ripple and at least 40 dB attenuation in the stopbands. Specify a sample rate of 1 kHz. Set the stopband width to 50 Hz on both sides of the passband. Find the filter order and cutoff frequencies.

```
Wp = [100 200]/500;
Ws = [50 250]/500;
Rp = 3;
Rs = 40;
[n,Wn] = buttord(Wp,Ws,Rp,Rs)
```

```
n = 8

Wn = 1×2

    0.1951    0.4080
```

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))
```



Input Arguments

`Wp` — Passband corner (cutoff) frequency

scalar | two-element vector

Passband corner (cutoff) frequency, specified as a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, π rad/sample.

- If `Wp` and `Ws` are both scalars and `Wp < Ws`, then `buttord` returns the order and cutoff frequency of a lowpass filter. The stopband of the filter ranges from `Ws` to 1 and the passband ranges from 0 to `Wp`.
- If `Wp` and `Ws` are both scalars and `Wp > Ws`, then `buttord` returns the order and cutoff frequency of a highpass filter. The stopband of the filter ranges from 0 to `Ws` and the passband ranges from `Wp` to 1.
- If `Wp` and `Ws` are both vectors and the interval specified by `Ws` contains the one specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`), then `buttord` returns the order and cutoff frequencies of a bandpass filter. The stopband of the filter ranges from 0 to `Ws(1)` and from `Ws(2)` to 1. The passband ranges from `Wp(1)` to `Wp(2)`.
- If `Wp` and `Ws` are both vectors and the interval specified by `Wp` contains the one specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`), then `buttord` returns the order and cutoff frequencies of a bandstop filter. The stopband of the filter ranges from `Ws(1)` to `Ws(2)`. The passband ranges from 0 to `Wp(1)` and from `Wp(2)` to 1.

Data Types: `single` | `double`

Note 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 and cascade the two filters together.

`Ws` — Stopband corner frequency
scalar | two-element vector

Stopband corner frequency, specified as a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, π rad/sample.

Data Types: `single` | `double`

`Rp` — Passband ripple
scalar

Passband ripple, specified as a scalar expressed in dB.

Data Types: `single` | `double`

`Rs` — Stopband attenuation
scalar

Stopband attenuation, specified as a scalar expressed in dB.

Data Types: `single` | `double`

Output Arguments

`n` — Lowest filter order
integer scalar

Lowest filter order, returned as an integer scalar.

`Wn` — Cutoff frequencies
scalar | vector

Cutoff frequencies, returned as a scalar or vector.

Algorithms

`buttord`'s order prediction formula 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. The function 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/second (for lowpass and highpass filters) and to –1 and 1 rad/second (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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`butter` | `cheb1ord` | `cheb2ord` | `ellipord` | `kaiserord`

Introduced before R2006a

cceps

Complex cepstral analysis

Syntax

```
xhat = cceps(x)
[xhat,nd] = cceps(x)
[xhat,nd,xhat1] = cceps(x)
[___] = cceps(x,n)
```

Description

`xhat = cceps(x)` returns the complex cepstrum `xhat` of the real data sequence `x` using the Fourier transform.

Note `cceps` only works on real data.

`[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`.

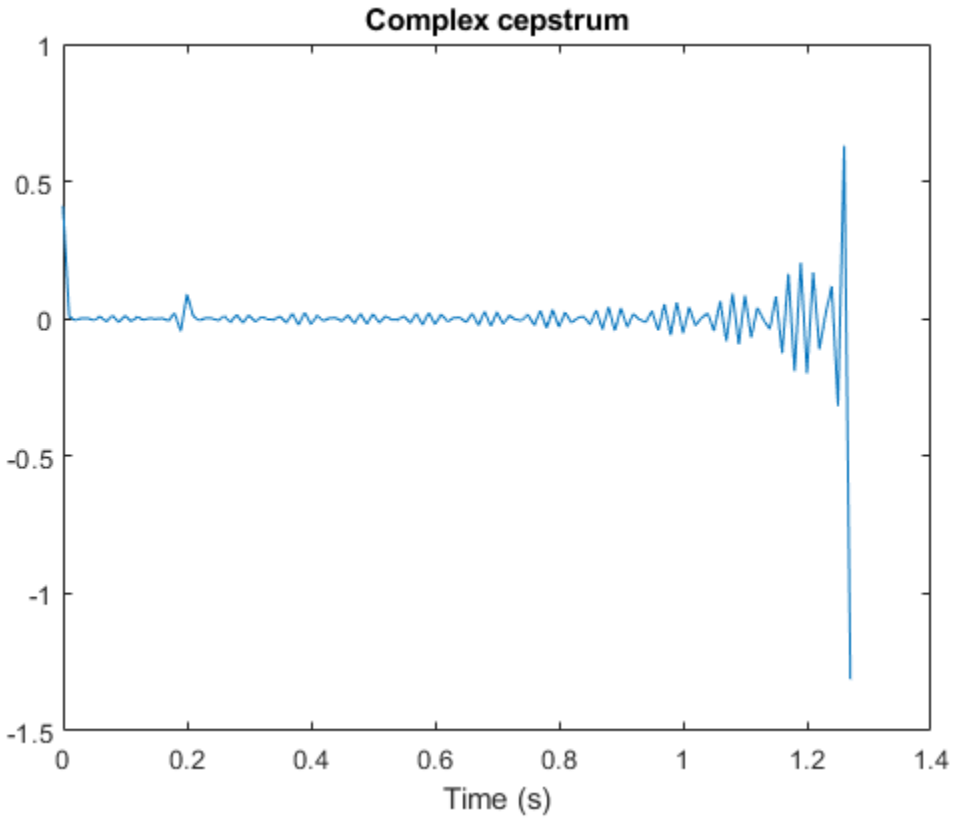
`[___] = cceps(x,n)` zero pads `x` to length `n` and returns the length `n`.

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')
```



Input Arguments

x — Input signal
real vector

Input signal, specified as a real vector. By the application of a linear phase term, the input is altered to have no phase discontinuity at $\pm\pi$ radians. That is, it is circularly shifted (after zero padding) by some samples, if necessary, to have zero phase at π radians.

n — Length of zero-padded signal
positive real integer

Length of zero-padded signal, specified as a positive real integer.

Output Arguments

xhat — Complex cepstrum
vector

Complex cepstrum, returned as a vector.

nd — Number of samples
real positive scalar

Number of samples of circular delay added to `x`, returned as a positive real scalar.

`xhat1` — Second complex cepstrum
vector

Second complex cepstrum, returned as a vector. `xhat1` is computed using an alternative factorization algorithm specified in the references [1] and [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.

Algorithms

Cepstral analysis is a nonlinear signal processing technique that is applied most commonly in speech processing and homomorphic filtering [1]. `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. Schafer, 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

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

Introduced before R2006a

cconv

Modulo-n circular convolution

Syntax

```
c = cconv(a,b)
c = cconv(a,b,n)
```

Description

`c = cconv(a,b)` convolves vectors `a` and `b`.

`c = cconv(a,b,n)` circularly convolves vectors `a` and `b`. `n` is the length of the resulting vector. You can also use `cconv` to compute the circular cross-correlation of two sequences.

Examples

Circular Convolution and Linear Convolution

Generate two signals of different lengths. Compare their circular convolution and their linear convolution. Use the default value for `n`.

```
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 resulting norm is virtually zero, which shows that the two convolutions produce the same result to machine precision.

Circular Convolution

Generate two vectors and compute their modulo-4 circular convolution.

```
a = [2 1 2 1];
b = [1 2 3 4];
c = cconv(a,b,4)
```

```
c = 1×4

    14    16    14    16
```

Circular Cross-Correlation

Generate two complex sequences. Use `cconv` to compute their circular cross-correlation. Flip and conjugate the second operand to comply with the definition of cross-correlation. Specify an output vector length of 7.

```
a = [1 2 2 1]+1i;
b = [1 3 4 1]-2*1i;
c = cconv(a,conj(fliplr(b)),7);
```

Compare the result to the cross-correlation computed using `xcorr`.

```
cref = xcorr(a,b);
dif = norm(c-cref)
```

```
dif = 3.3565e-15
```

Circular Convolution with Varying Output Length

Generate two signals: a five-sample triangular waveform and a first-order FIR filter with response $H(z) = 1 - z^{-1}$.

```
x1 = conv([1 1 1],[1 1 1])
```

```
x1 = 1×5

     1     2     3     2     1
```

```
x2 = [-1 1]
```

```
x2 = 1×2

    -1     1
```

Compute their circular convolution with the default output length. The result is equivalent to the linear convolution of the two signals.

```
ccnv = cconv(x1,x2)
```

```
ccnv = 1×6

   -1.0000   -1.0000   -1.0000    1.0000    1.0000    1.0000
```

```
lcnv = conv(x1,x2)
```

```
lcnv = 1×6

    -1    -1    -1     1     1     1
```

The modulo-2 circular convolution is equivalent to splitting the linear convolution into two-element arrays and summing the arrays.

```
ccn2 = cconv(x1,x2,2)
```

```
ccn2 = 1×2

    -1     1
```

```
nl = numel(lcnv);
mod2 = sum(reshape(lcnv,2,nl/2)')
```

```
mod2 = 1×2

    -1     1
```

Compute the modulo-3 circular convolution and compare it to the aliased linear convolution.

```
ccn3 = cconv(x1,x2,3)
```

```
ccn3 = 1×3

     0     0     0
```

```
mod3 = sum(reshape(lcnv,3,nl/3)')
```

```
mod3 = 1×3

     0     0     0
```

If the output length is smaller than the convolution length and does not divide it exactly, pad the convolution with zeros before adding.

```
c = 5;
z = zeros(c*ceil(nl/c),1);
z(1:nl) = lcnv;

ccnc = cconv(x1,x2,c)
```

```
ccnc = 1×5

    0.0000   -1.0000   -1.0000    1.0000    1.0000
```

```
modc = sum(reshape(z,c,numel(z)/c)')
```

```
modc = 1×5

     0    -1    -1     1     1
```

If the output length is equal to or larger than the convolution length, pad the convolution and do not add.

```
d = 13;
z = zeros(d*ceil(nl/d),1);
z(1:nl) = lcnv;

ccnd = cconv(x1,x2,d)
```

```
ccnd = 1×13

   -1.0000   -1.0000   -1.0000    1.0000    1.0000    1.0000    0.0000   -0.0000    0.0000    0.0
```

```
modd = z'
```

```
modd = 1×13

    -1    -1    -1     1     1     1     0     0     0     0     0     0     0
```

Circular Convolution Using the GPU

The following example requires Parallel Computing Toolbox™ software. Refer to “GPU Support by Release” (Parallel Computing Toolbox) to see what GPUs are supported.

Create two signals consisting of a 1 kHz sine wave in additive white Gaussian noise. The sample 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)
```

```
ans =

   1.4047e-08
```

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

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

Input Arguments

a, b — Input arrays

vector | `gpuArray` object

Input array, specified as vectors or `gpuArray` objects. See “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox) for details on `gpuArray` objects. Using `cconv` with `gpuArray` objects requires Parallel Computing Toolbox™ software. Refer to “GPU Support by Release” (Parallel Computing Toolbox) to see what GPUs are supported.

Example: `sin(2*pi*(0:9)/10) + randn([1 10])/10` specifies a noisy sinusoid as a row vector.

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

n — Convolution length
positive integer

Convolution length, specified as a positive integer. If you do not specify `n`, then the convolution has length `length(a)+length(b)-1`.

Output Arguments

c — Circular convolution
vector | `gpuArray` object

Circular convolution of input vectors, returned as a vector or `gpuArray`.

Tips

For long sequences, circular convolution can be faster than linear convolution.

References

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
`conv` | `xcorr`

Introduced in R2007a

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 `bi` and `ai` are row vectors of at most length 3, and *i* = 1, 2, ..., *L*. The resulting matrix `m` is given by

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

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.

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

```
sos = 3×6

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

See Also

`sos2cell` | `tf2sos`

Introduced before R2006a

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( ___ ,debug)
b = cfirpm( ___ ,lgrid)
b = cfirpm( ___ ,'skip_stage2')
[b,delta] = cfirpm( ___ )
[b,delta,opt] = cfirpm( ___ )
```

Description

`b = cfirpm(n,f,fresp)` returns a length `n`+1 FIR filter with the best approximation to the desired frequency response as returned by the `fresp` function, which is called by its function handle (*`@fresp`*).

`b = cfirpm(n,f,fresp,w)` uses the weights specified by `w` to weight the fit in each frequency band.

`b = cfirpm(n,f,a)` specifies amplitudes `a` at the band edges in `f`. This syntax returns the same result as `b = cfirpm(n,f,{@multiband,a})`.

`b = cfirpm(n,f,a,w)` applies an optional set of positive weights, one per band, for use during optimization. If you do not specify `w`, the function sets the weights to unity.

`b = cfirpm( ___ ,sym)` imposes a symmetry constraint on the impulse response of the design. In addition to specifying `sym`, specify an input combination from any of the previous syntaxes.

`b = cfirpm( ___ ,debug)` displays or hides the intermediate results during the filter design.

`b = cfirpm( ___ ,lgrid)` controls the density of the frequency grid.

`b = cfirpm( ___ ,'skip_stage2')` disables the second-stage optimization algorithm, which executes only when the `cfirpm` function determines that an optimal solution has not been reached by the standard `firpm` error-exchange. Disabling this algorithm can increase the speed of computation but incur a reduction in accuracy. By default, the second-stage optimization is enabled.

`[b,delta] = cfirpm( ___ )` returns the maximum ripple height `delta`.

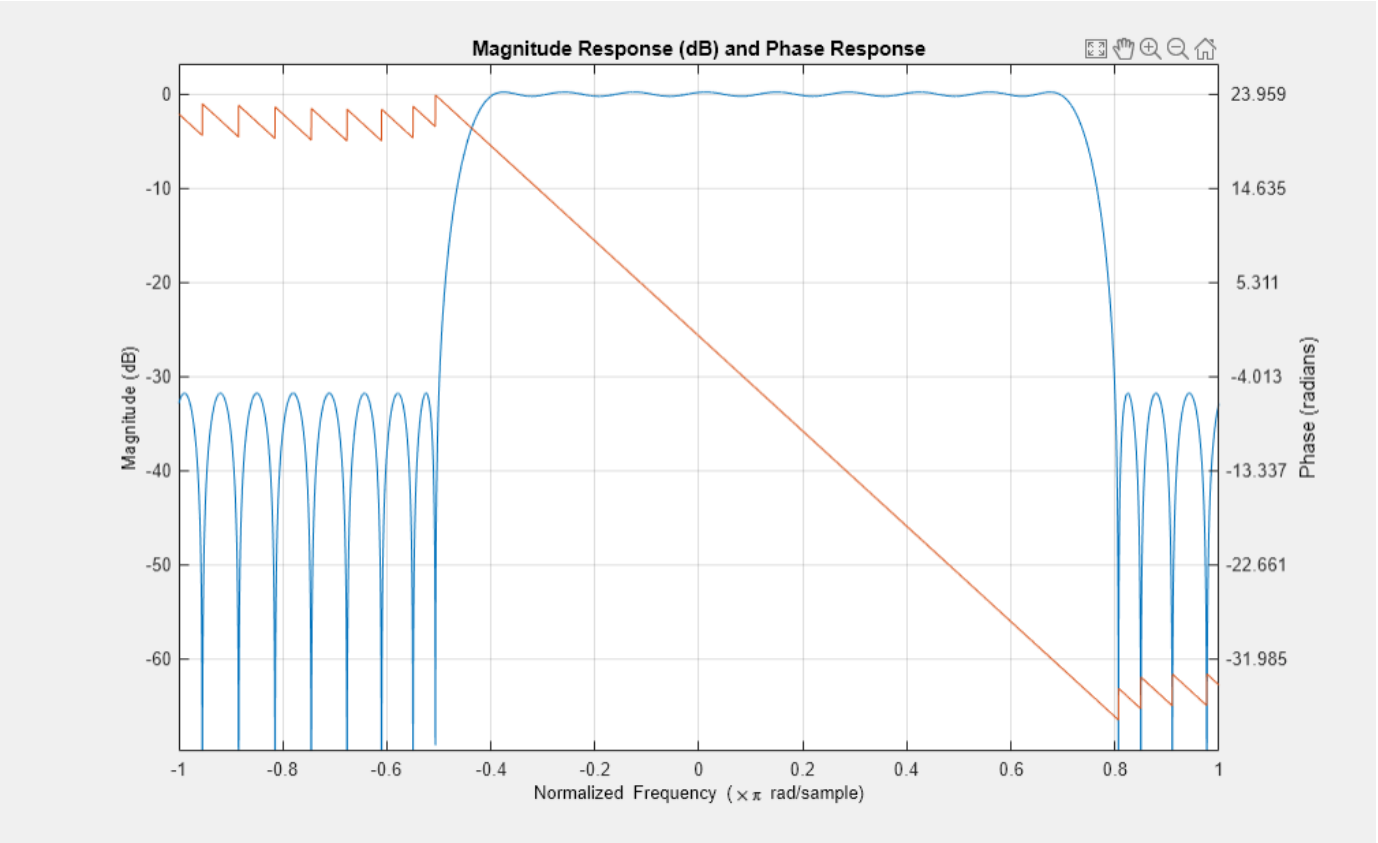
`[b,delta,opt] = cfirpm( ___ )` returns optional results computed by the `cfirpm` function.

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,'OverlayedAnalysis','phase')
```



FIR Approximation to Allpass Response

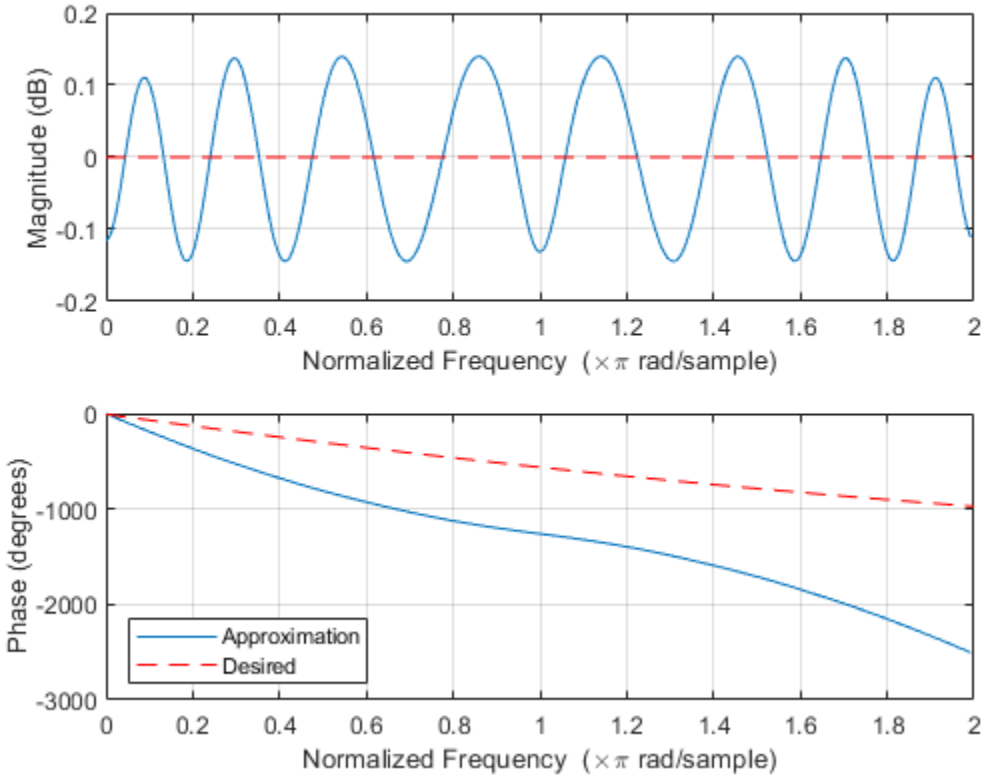
Design a nonlinear-phase allpass FIR filter of order 22 with frequency response given approximately by $\exp(-j\pi fN/2 + j4\pi f|f|)$, where $f \in [-1, 1]$.

```
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')
```



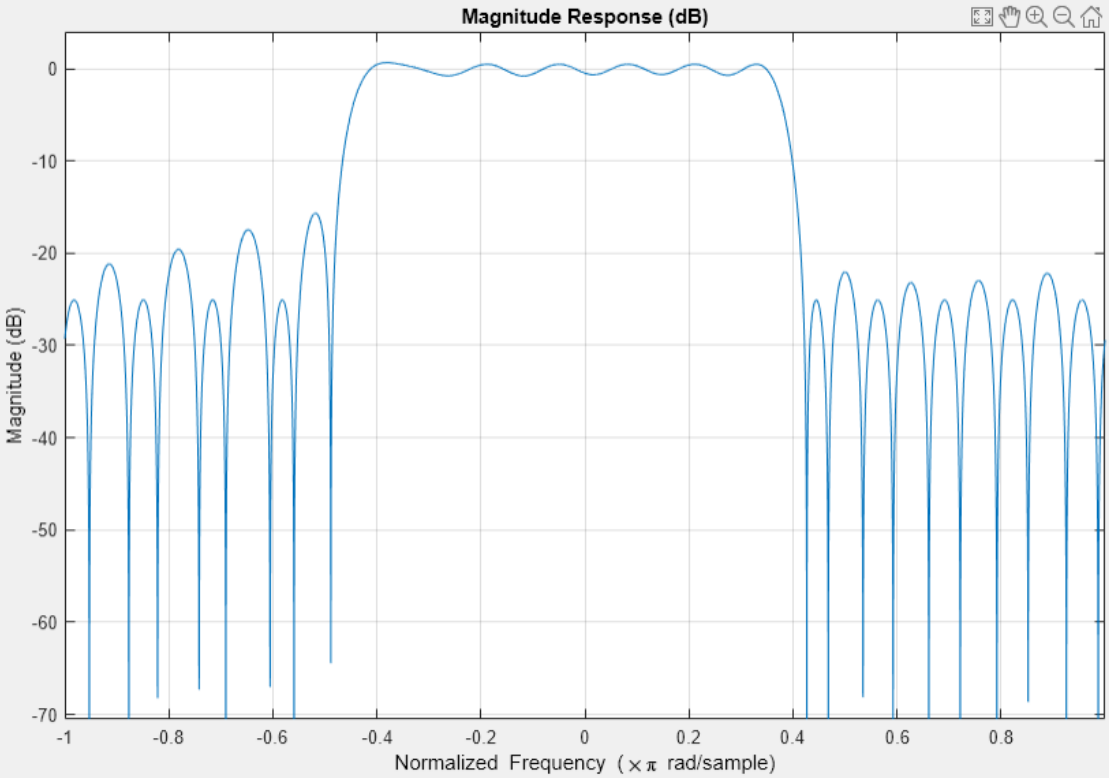
Lowpass Filter using Custom Frequency Response Function

Design a lowpass filter of order 30 using a custom frequency response function `fresp`. The code for the `fresp` function is available at the end of the example.

```
[b,delta]= cfirpm(30,linspace(-1,1,32),@fresp);
```

Use `FVTool` to visualize the magnitude response of the filter.

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



User-Defined `fresp` Function: Design a lowpass filter

The `fresp` function lets you choose to design a lowpass filter, a highpass filter, or a differentiator. The filter order `N` and frequency array `F` must be specified. If the frequency grid `GF` and weights `W` are unspecified, the function determines those values automatically.

```
function [dh,dw] = fresp(N,F,GF,W)

W = [1;1]*(W(:).'); W = W(:);

type = 'lowpass';

mags = zeros(size(W));

switch type
    case 'lowpass'
        mags(10:end-10) = 1;
    case 'highpass'
        mags(1:10) = 1;
        mags(end-10:end) = 1;
    case 'differentiator'
        mags = abs(linspace(-pi,pi,length(mags)));
end

dh = interp1(F(:),mags,GF).*exp(-1j*pi*GF*N/2);
dw = interp1(F(:),W,GF);
```

```
end
```

Input Arguments

`n` — Filter order

real positive scalar

Filter order, specified as a real positive scalar.

`f` — Normalized frequency points

real-valued vector with elements in the range [–1, 1]

Normalized frequency points, specified as a real-valued vector with elements in the range [–1, 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.

`fresp` — Frequency response

function handle

Frequency response, specified as a function handle. For more information, see “Predefined Frequency Response Functions” on page 1-139 and “User-Defined Frequency Response Functions” on page 1-140.

`a` — Desired amplitude

vector

Desired amplitudes at the points specified in `f`, specified as a vector. 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)).

`w` — Weights

real-valued vector

Weights used to adjust the fit in each frequency band, specified as a real-valued vector. The length of `w` is half the length of `f`, so exactly one weight exists per band. If you do not specify `w`, the function sets the weights to unity.

`sym` — Symmetry constraint

`'none'` (default) | `'even'` | `'odd'` | `'real'`

Symmetry constraint imposed on the impulse response of the filter design, specified as one of these values:

- `'none'` — Impose no symmetry constraint. This option is the default if you pass any negative band frequencies to the function or if `fresp` does not supply a default value.
- `'even'` — Impose a real and even impulse response. This option is the default for highpass, lowpass, allpass, bandpass, bandstop, inverse-sinc, and multiband designs.
- `'odd'` — Impose a real and odd impulse response. This option is the default for Hilbert and differentiator designs.
- `'real'` — Impose conjugate symmetry for the frequency response.

If you specify a value other than `'none'`, you must specify the band edges over only positive frequencies (the negative frequency region is filled in from symmetry). If you do not specify `sym`, the function queries `fresp` for a default setting. Any user-supplied `fresp` function must return a valid `sym` option when it is passed `'defaults'` as the filter order `n`.

debug — Display of intermediate results
`'off'` (default) | `'trace'` | `'plots'` | `'both'`

Display of intermediate results during the filter design, specified as `'off'`, `'trace'`, `'plots'`, or `'both'`.

`lgrid` — Density of frequency grid
25 (default) | cell array of an integer

Density of frequency grid, specified as a cell array of an integer. The frequency grid has roughly `2^nextpow2(lgrid*n)` frequency points.

Output Arguments

`b` — Filter coefficients
row vector

Filter coefficients, returned as a row vector of length `n+1`.

`delta` — Maximum ripple height
scalar

Maximum ripple height, returned as a scalar.

`opt` — Optional results
structure

Optional results computed by the `cfirpm` function, returned as a structure containing these fields.

Field	Description
`opt.fgrid`	Frequency grid vector used for the filter design optimization
`opt.des`	Desired frequency response for each point in `opt.fgrid`
`opt.wt`	Weighting for each point in `opt.fgrid`
`opt.H`	Actual frequency response for each point in `opt.fgrid`
`opt.error`	Error at each point in `opt.fgrid`
`opt.iextr`	Vector of indices into `opt.fgrid` for extremal frequencies
`opt.fextr`	Vector of extremal frequencies

More About

Predefined Frequency Response Functions

Predefined `fresp` frequency response functions are included for a number of common filter designs in this section. For more information on how to create a custom `fresp` function, see “Create Function Handle”.

For all of the predefined frequency response functions, the symmetry option `sym` defaults to `'even'` if `f` contains no negative frequencies and `d = 0`. Otherwise `sym` defaults to `'none'`. For details, see `sym`. 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, and positive values create more delay. By default, `d = 0`.

- `@lowpass`, `@highpass`, `@allpass`, `@bandpass`, `@bandstop`

  These functions share a common syntax, exemplified by `@lowpass`.

  `b = cfirpm(n,f,@lowpass,...)` and

  `b = cfirpm(n,f,{@lowpass,d},...)` design a linear-phase (`n/2+d` delay) filter.

  Note For `@bandpass` filters, the first element in the frequency vector must be less than or equal to zero and the last element must be greater than or equal to zero.

- `@multiband` designs a linear-phase frequency response filter with arbitrary band amplitudes.

  `b = cfirpm(n,f,{@multiband,a},...)` and

  `b = cfirpm(n,f,{@multiband,a,d},...)` specify vector `a` containing the desired amplitudes at the band edges in `f`. The desired amplitude at frequencies between pairs of points `f`(*k*) and `f`(*k* +1) for *k* odd is the line segment connecting the points (`f`(*k*),`a`(*k*)) and (`f`(*k*+1),`a`(*k*+1)).

- `@differentiator` designs a linear-phase differentiator. For these designs, zero-frequency must be in a transition band, and band weighting is set to be inversely proportional to frequency.

  `b = cfirpm(n,f,{@differentiator,fs},...)` and

  `b = cfirpm(n,f,{@differentiator,fs,d},...)` specify the sample rate `fs` used to determine the slope of the differentiator response. If omitted, `fs` defaults to 1.

- `@hilbfilt` designs a linear-phase Hilbert transform filter response. For Hilbert designs, zero-frequency must be in a transition band.

  `b = cfirpm(n,f,@hilbfilt,...)` and

  `b = cfirpm(N,F,{@hilbfilt,d},...)` design a linear-phase (`n/2+d` delay) Hilbert transform filter.

- `@invsinc` designs a linear-phase inverse-sinc filter response.

  `b = cfirpm(n,f,{@invsinc,a},...)` and

  `b = cfirpm(n,f,{@invsinc,a,d},...)` specify gain `a` for the sinc function, computed as sinc(`a`**g*), where *g* contains the optimization grid frequencies normalized to the range [–1, 1]. By default, `a = 1`. The group-delay offset is `d` such that the filter response has a group delay of *n*/2+`d` in units of the sample interval, where *n* is the filter order. Negative values create less delay, and positive values create more delay. By default, `d = 0`.

User-Defined Frequency Response Functions

Instead of the predefined frequency response functions for `fresp`, you can use a user-defined function.

The `cfirpm` function calls this user-defined function using this syntax.

```
[dh,dw] = fresp(n,f,gf,w,p1,p2,...)
```

- `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`. The input `gf` determines the frequency grid at which the response function must be evaluated. The `cfirpm` function returns this data 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 you do not specify this input, `cfirpm` sets it to unity weighting before passing it to *`fresp`*.
- `dh` and `dw` are the desired complex frequency response and band weight vectors, respectively, that are evaluated at each frequency in grid `gf`.
- `p1,p2,...` are optional parameters that can be passed to *`fresp`*.

Additionally, the `cfirpm` function makes a preliminary call to *`fresp`* to determine the default symmetry `sym`. `cfirpm` makes this call using this syntax.

```
sym = fresp('defaults',{n,f,[],w,p1,p2,...})
```

The arguments can be used in determining an appropriate symmetry default as necessary. You can use the local function `lowpass` as a template for generating new frequency response functions. To find the `lowpass` function, enter `edit cfirpm` at the command line and search for `lowpass` in the `cfirpm` function code. You can copy the function, modify it, rename it, and save it in your path.

Algorithms

The `cfirpm` function enables you to specify arbitrary frequency-domain constraints for the design of a possibly complex FIR filter. The Chebyshev (or minimax) filter error is optimized, producing equiripple FIR filter designs.

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 the filter does not converge, the algorithm switches to an ascent-descent algorithm that takes over to finish the convergence to the optimal solution. For further details, see the references.

References

[1] Demjanjov, V. F., and V. N. Malozemov. *Introduction to Minimax*. New York: John Wiley & Sons, 1974.

[2] Karam, L.J. *Design of Complex Digital FIR Filters in the Chebyshev Sense*. Ph.D. Thesis, Georgia Institute of Technology, March 1995.

[3] Karam, L.J., and J. H. McClellan. "Complex Chebyshev Approximation for FIR Filter Design." *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 42, no. 3 (March 1995): 207–216.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`fir1` | `fir2` | `firls` | `firpm`

Introduced before R2006a

cheb1ap

Chebyshev Type I analog lowpass filter prototype

Syntax

```
[z,p,k] = cheb1ap(n,Rp)
```

Description

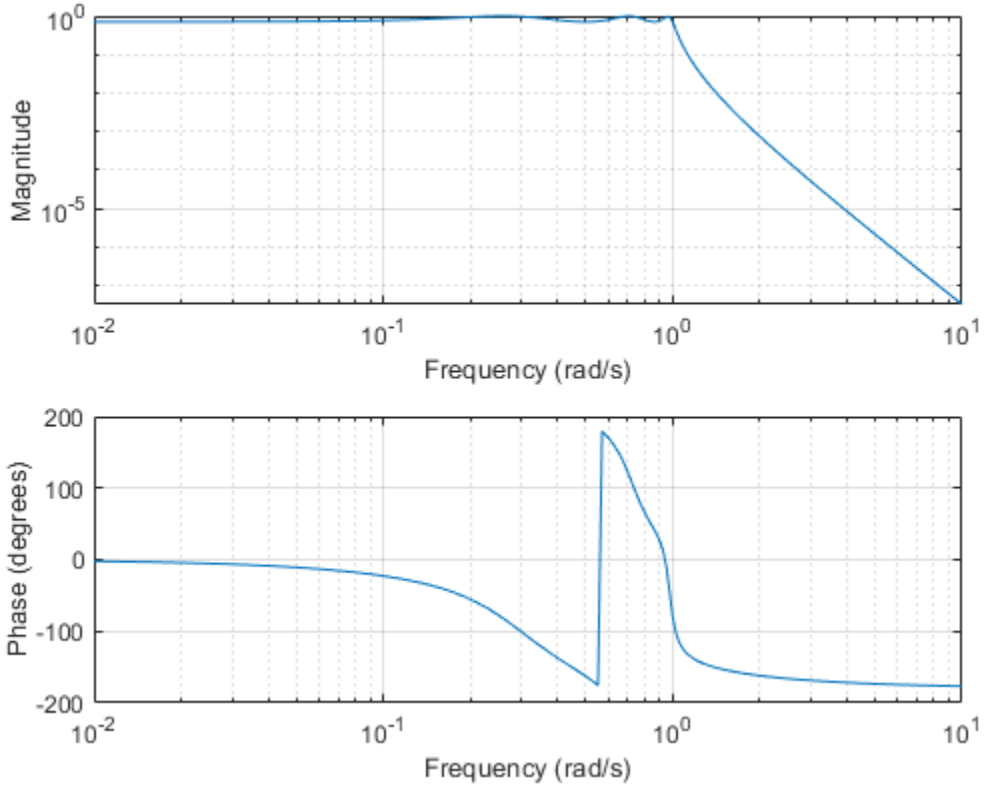
`[z,p,k] = cheb1ap(n,Rp)` returns the poles and gain of an order `n` Chebyshev Type I analog lowpass filter prototype with `Rp` dB of ripple in the passband.

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
```



Input Arguments

n — Filter order

integer

Filter order, specified as an integer.

Data Types: `single` | `double`

Rp — Passband ripple

scalar

Passband ripple, specified as a scalar in decibels.

Data Types: `single` | `double`

Output Arguments

z — Zeros

matrix

Zeros of the filter, returned as a matrix.

p — Poles

n-length column vector

Poles of the filter, returned as an n-length column vector.

k — Gain

scalar

Gain of the filter, returned as a scalar. z is an empty matrix because no zeros exist for this filter design.

Algorithms

Chebyshev Type I filters are equiripple in the passband and monotonic in the stopband. The poles are evenly spaced about an ellipse in the left half plane. The Chebyshev Type I passband edge angular frequency ω_0 is set to 1.0 for a normalized result. This value is the frequency at which the passband ends. The filter has a magnitude response of $10^{-Rp/20}$.

The transfer function is given by

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

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`besselap` | `buttap` | `cheby1` | `cheb2ap` | `ellipap`

Introduced before R2006a

cheb1ord

Chebyshev Type I filter order

Syntax

```
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs,'s')
```

Description

`[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.

`[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs,'s')` designs a lowpass, highpass, bandpass, or bandstop analog Chebyshev Type I filter with cutoff angular frequencies `Wp`.

Examples

Chebyshev Type I Filter Design

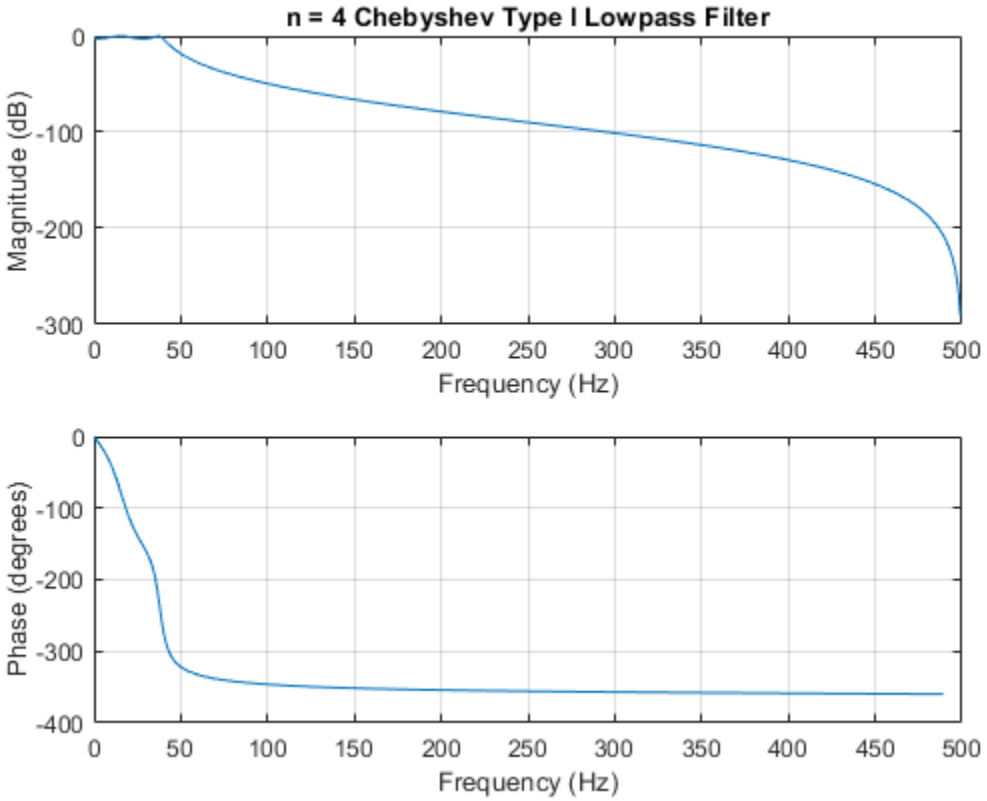
For data sampled at 1000 Hz, design a lowpass filter with less than 3 dB of ripple in the passband defined from 0 to 40 Hz and at least 60 dB of ripple in the stopband defined from 150 Hz to the Nyquist frequency.

```
Wp = 40/500;
Ws = 150/500;
Rp = 3;
Rs = 60;
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)
```

```
n = 4
```

```
Wp = 0.0800
```

```
[b,a] = cheby1(n,Rp,Wp);
freqz(b,a,512,1000)
title('n = 4 Chebyshev Type I Lowpass Filter')
```



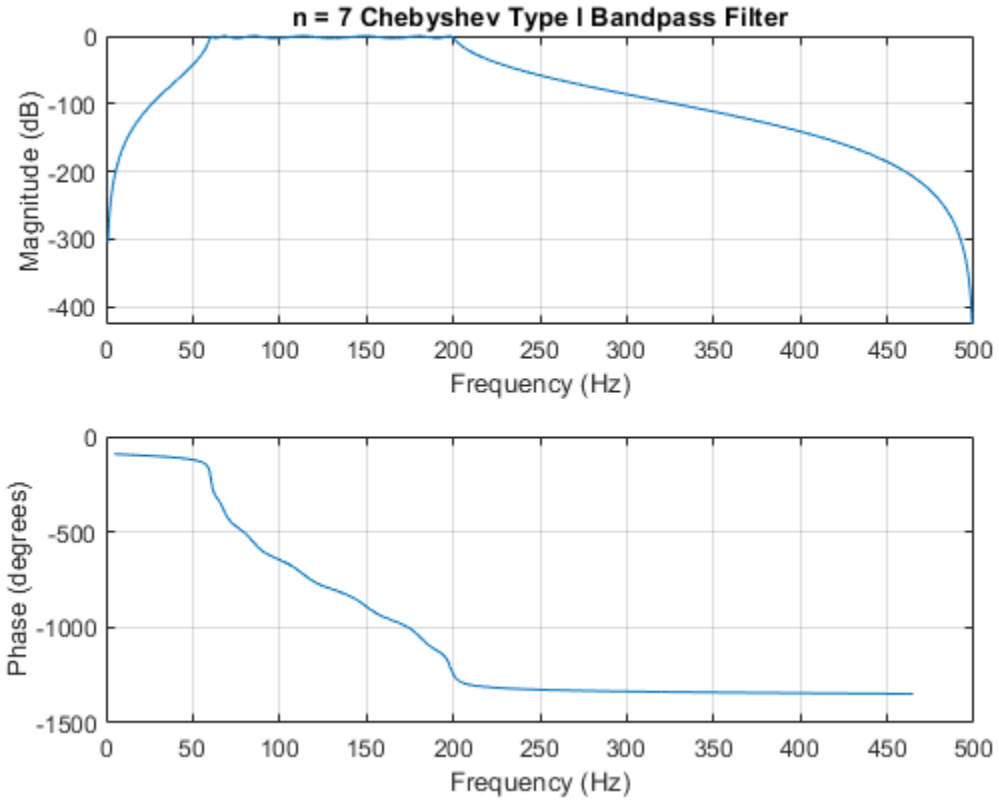
Chebyshev Type I Bandpass Filter Design

Design a bandpass filter with a passband of 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband.

```
Wp = [60 200]/500;
Ws = [50 250]/500;
Rp = 3;
Rs = 40;
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)
```

```
n = 7

Wp = 1×2

    0.1200    0.4000
```

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



Input Arguments

`Wp` — Passband corner (cutoff) frequency

scalar | two-element vector

Passband corner (cutoff) frequency, specified as a scalar or a two-element vector with values between 0 and 1 inclusive, with 1 corresponding to the normalized Nyquist frequency, π rad/sample. For digital filters, the unit of passband corner frequency is in radians per sample. For analog filters, passband corner frequency is in radians per second, and the passband can be infinite. The values of `Wp` and `Ws` determine the type of filter cheb1ord returns:

- If `Wp` and `Ws` are both scalars and `Wp < Ws`, then `cheb1ord` returns the order and cutoff frequency of a lowpass filter. The stopband of the filter ranges from `Ws` to 1, and the passband ranges from 0 to `Wp`.
- If `Wp` and `Ws` are both scalars and `Wp > Ws`, then `cheb1ord` returns the order and cutoff frequency of a highpass filter. The stopband of the filter ranges from 0 to `Ws`, and the passband ranges from `Wp` to 1.
- If `Wp` and `Ws` are both vectors and the interval specified by `Ws` contains the interval specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`), then `cheb1ord` returns the order and cutoff frequencies of a bandpass filter. The stopband of the filter ranges from 0 to `Ws(1)` and from `Ws(2)` to 1. The passband ranges from `Wp(1)` to `Wp(2)`.
- If `Wp` and `Ws` are both vectors and the interval specified by `Wp` contains the interval specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`), then `cheb1ord` returns the order and cutoff frequencies of a

bandstop filter. The stopband of the filter ranges from `Ws(1)` to `Ws(2)`. The passband ranges from 0 to `Wp(1)` and from `Wp(2)` to 1.

Use the following guide to specify filters of different types.

Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	`Wp` < `Ws`, both scalars	`(Ws,1)`	`(0,Wp)`
Highpass	`Wp` > `Ws`, both scalars	`(0,Ws)`	`(Wp,1)`
Bandpass	The interval specified by `Ws` contains the one specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`).	`(0,Ws(1))` and `(Ws(2),1)`	`(Wp(1),Wp(2))`
Bandstop	The interval specified by `Wp` contains the one specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`).	`(0,Wp(1))` and `(Wp(2),1)`	`(Ws(1),Ws(2))`

Data Types: `single` | `double`

Note If your filter specifications call for a bandpass or bandstop filter with unequal ripples in each of the passbands or stopbands, design separate lowpass and highpass filters and cascade the two filters together.

`Ws` — Stopband corner frequency

scalar | two-element vector

Stopband corner frequency, specified as a scalar or a two-element vector with values between 0 and 1 inclusive, with 1 corresponding to the normalized Nyquist frequency.

- For digital filters, stopband corner frequency is in radians per sample.
- For analog filters, stopband corner frequency is in radians per second and the stopband can be infinite.

Note The values of `Wp` and `Ws` determine the filter type.

`Rp` — Passband ripple

scalar

Passband ripple, specified as a scalar in dB.

Data Types: `single` | `double`

`Rs` — Stopband attenuation

scalar

Stopband attenuation, specified as a scalar in dB.

Data Types: `single` | `double`

Output Arguments

n — Lowest filter order
integer scalar

Lowest filter order, returned as an integer scalar.

Wp — Passband corner frequency
scalar | two-element vector

Passband corner frequency, returned as a scalar or a two-element vector. Use the output arguments `n` and `Wp` with the `cheby1` function.

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 lowpass or highpass filters) or to -1 and 1 rad/s (for bandpass or bandstop filters). It then computes the order and natural frequency required for a lowpass filter to match the passband specification exactly when using the values in the `cheby1` function.

References

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

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

Introduced before R2006a

cheb2ap

Chebyshev Type II analog lowpass filter prototype

Syntax

```
[z,p,k] = cheb2ap(n,Rs)
```

Description

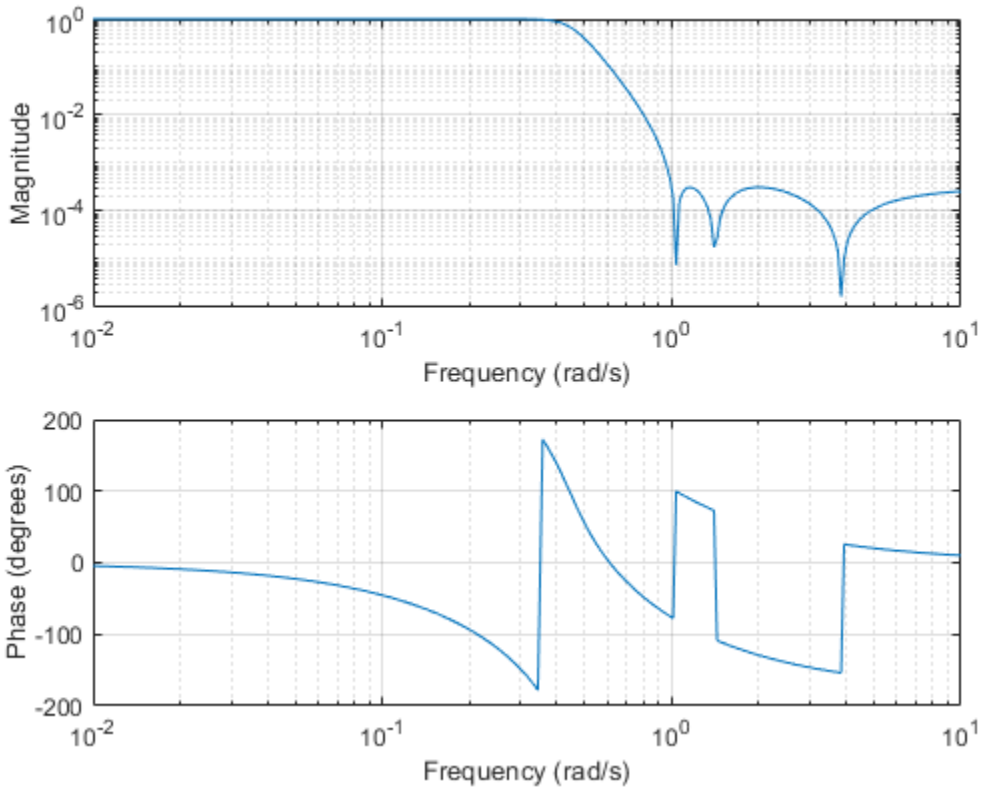
`[z,p,k] = cheb2ap(n,Rs)` returns the zeros, poles, and gain of an order `n` Chebyshev Type II analog lowpass filter prototype with `Rs` dB of ripple down from the passband peak value in the stopband.

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
```



Input Arguments

n — Filter order

integer

Filter order, specified as an integer.

Data Types: `single` | `double`

Rs — Stopband ripple

scalar

Stopband ripple, specified as a scalar in decibels.

Data Types: `single` | `double`

Output Arguments

z — Zeros

n-length column vector

Zeros of the filter, returned as an n-length column vector. If n is odd, z has length n–1.

p — Poles

n-length column vector

Poles of the filter, returned as an `n`-length column vector.

k — Gain
scalar

Gain of the filter, returned as a scalar.

Algorithms

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 the `cheb1ap` function, whose poles are evenly spaced about an ellipse in the left half plane. The Chebyshev Type II stopband edge angular frequency ω_0 is set to 1 for a normalized result. This value is the frequency at which the stopband begins. The filter has a magnitude response of $10^{-Rs/20}$.

Chebyshev Type II filters are sometimes called *inverse Chebyshev* filters because of their relationship to Chebyshev Type I filters. The `cheb2ap` function is a modification of the Chebyshev Type I prototype algorithm:

1. `cheb2ap` replaces the frequency variable ω with $1/\omega$, turning the lowpass filter into a highpass filter while preserving the performance at $\omega = 1$.
2. `cheb2ap` subtracts the filter transfer function from unity.

The transfer function is given by

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

References

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`besselap` | `buttap` | `cheb1ap` | `cheby2` | `ellipap`

Introduced before R2006a

cheb2ord

Chebyshev Type II filter order

Syntax

```
[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)
[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs,'s')
```

Description

`[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.

`[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs,'s')` designs a lowpass, highpass, bandpass, or bandstop analog Chebyshev Type II filter with cutoff angular frequencies `Ws`.

Examples

Chebyshev Type II Filter Design

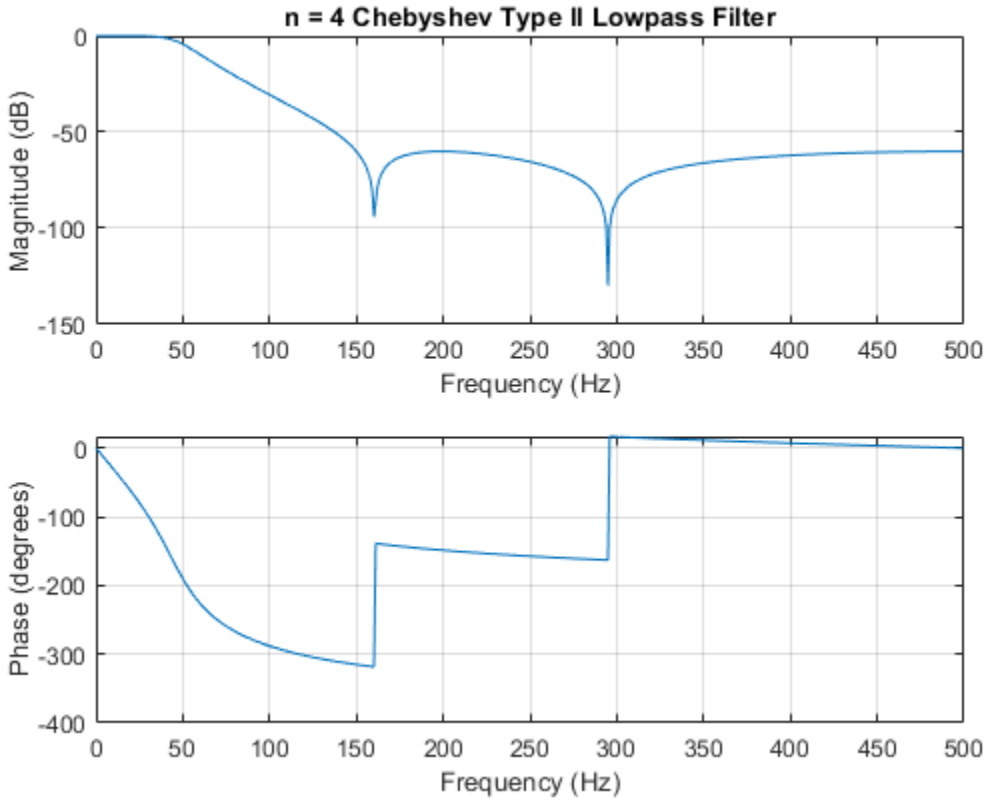
For data sampled at 1000 Hz, design a lowpass filter with less than 3 dB of ripple in the passband defined from 0 to 40 Hz, and at least 60 dB of attenuation in the stopband defined from 150 Hz to the Nyquist frequency.

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

[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)
```

```
n = 4
```

```
Ws = 0.3000
```

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

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



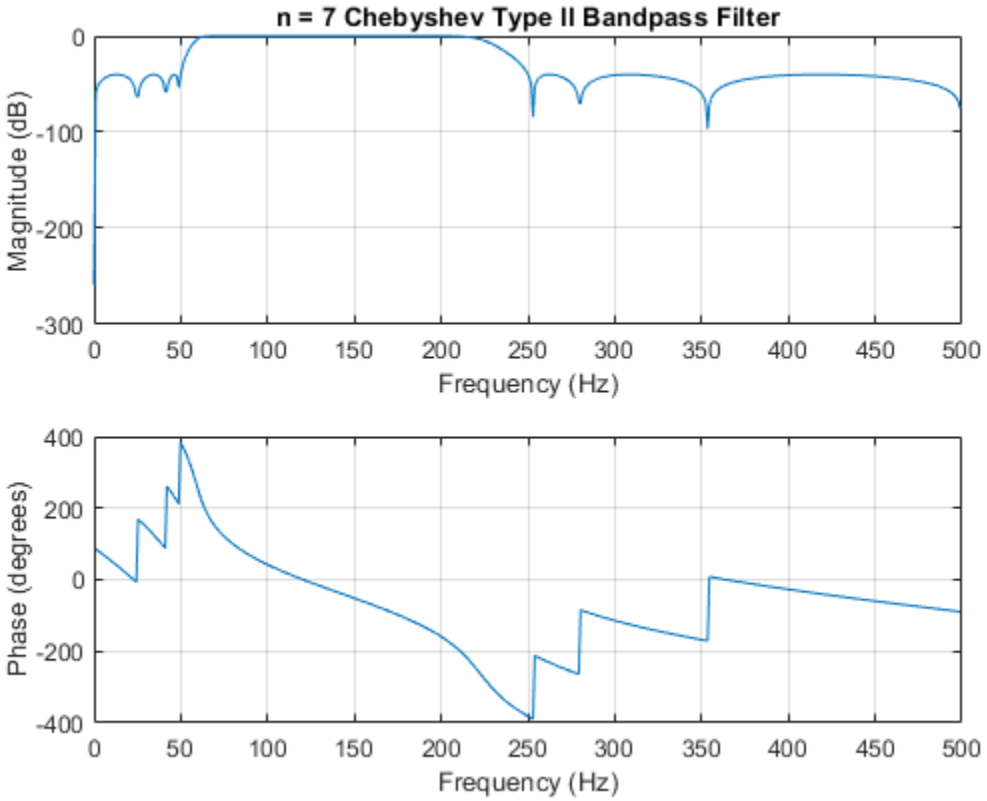
Chebyshev Type II Bandpass Filter Design

Design a bandpass filter with a passband of 60 Hz to 200 Hz, with less than 3 dB of ripple in the passband, and 40 dB attenuation in the stopbands that are 50 Hz wide on both sides of the passband:

```
Wp = [60 200]/500;
Ws = [50 250]/500;
Rp = 3;
Rs = 40;

[n,Ws] = cheb2ord(Wp,Ws,Rp,Rs)
```

```
n = 7

Ws = 1×2

    0.1000    0.5000
```

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

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



Input Arguments

`Wp` — Passband corner (cutoff) frequency

scalar | two-element vector

Passband corner (cutoff) frequency, specified as a scalar or a two-element vector with values between 0 and 1 inclusive, with 1 corresponding to the normalized Nyquist frequency, π rad/sample. For digital filters, the unit of passband corner frequency is in radians per sample. For analog filters, passband corner frequency is in radians per second, and the passband can be infinite. The values of `Wp` and `Ws` determine the type of filter cheb2ord returns:

- If `Wp` and `Ws` are both scalars and `Wp < Ws`, then `cheb2ord` returns the order and cutoff frequency of a lowpass filter. The stopband of the filter ranges from `Ws` to 1, and the passband ranges from 0 to `Wp`.
- If `Wp` and `Ws` are both scalars and `Wp > Ws`, then `cheb2ord` returns the order and cutoff frequency of a highpass filter. The stopband of the filter ranges from 0 to `Ws`, and the passband ranges from `Wp` to 1.
- If `Wp` and `Ws` are both vectors and the interval specified by `Ws` contains the interval specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`), then `cheb2ord` returns the order and cutoff frequencies of a bandpass filter. The stopband of the filter ranges from 0 to `Ws(1)` and from `Ws(2)` to 1. The passband ranges from `Wp(1)` to `Wp(2)`.
- If `Wp` and `Ws` are both vectors and the interval specified by `Wp` contains the interval specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`), then `cheb2ord` returns the order and cutoff frequencies of a

bandstop filter. The stopband of the filter ranges from `Ws(1)` to `Ws(2)`. The passband ranges from 0 to `Wp(1)` and from `Wp(2)` to 1.

Use the following guide to specify filters of different types.

Filter Type Stopband and Passband Specifications

Filter Type	Stopband and Passband Conditions	Stopband	Passband
Lowpass	`Wp` < `Ws`, both scalars	`(Ws,1)`	`(0,Wp)`
Highpass	`Wp` > `Ws`, both scalars	`(0,Ws)`	`(Wp,1)`
Bandpass	The interval specified by `Ws` contains the one specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`).	`(0,Ws(1))` and `(Ws(2),1)`	`(Wp(1),Wp(2))`
Bandstop	The interval specified by `Wp` contains the one specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`).	`(0,Wp(1))` and `(Wp(2),1)`	`(Ws(1),Ws(2))`

Data Types: `single` | `double`

Note 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 and cascade the two filters together.

`Ws` — Stopband corner frequency

scalar | two-element vector

Stopband corner frequency, specified as a scalar or a two-element vector with values between 0 and 1 inclusive, with 1 corresponding to the normalized Nyquist frequency.

- For digital filters, stopband corner frequency is in radians per sample.
- For analog filters, stopband corner frequency is in radians per second and the stopband can be infinite.

Note The values of `Wp` and `Ws` determine the filter type.

`Rp` — Passband ripple

scalar

Passband ripple, specified as a scalar in dB.

Data Types: `single` | `double`

`Rs` — Stopband attenuation

scalar

Stopband attenuation, specified as a scalar in dB.

Data Types: `single` | `double`

Output Arguments

n — Lowest filter order
integer scalar

Lowest filter order, returned as an integer scalar.

Ws — Stopband corner frequency
scalar | two-element vector

Stopband corner frequency, returned as a scalar or a two-element vector. Use the output arguments `n` and `Ws` with the `cheby2` function.

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 and natural frequency required for a lowpass filter to match the stopband specification exactly when using the values in the `cheby2` function.

References

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

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

Introduced before R2006a

chebwin

Chebyshev window

Syntax

```
w = chebwin(L)
w = chebwin(L,r)
```

Description

`w = chebwin(L)` returns an `L`-point Chebyshev window.

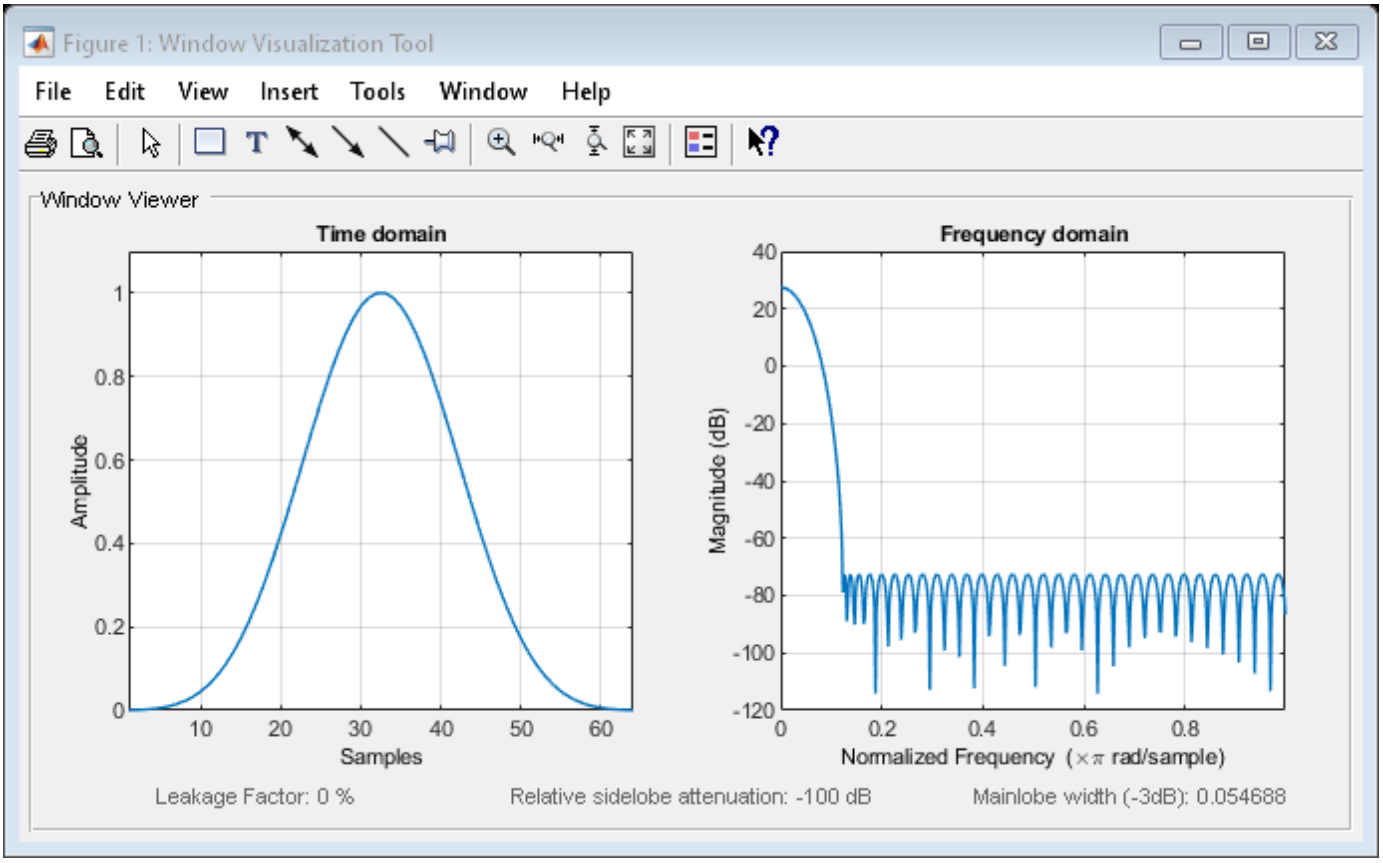
`w = chebwin(L,r)` returns an `L`-point Chebyshev window using sidelobe magnitude factor `r` dB.

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)
```



Input Arguments

`L` — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

`r` — Sidelobe attenuation

`100` dB (default) | positive real scalar

Sidelobe attenuation in dB, specified as a positive integer. The Chebyshev window has a Fourier transform magnitude `r` dB below the mainlobe magnitude.

Data Types: `single` | `double`

Output Arguments

`w` — Chebyshev window

column vector

Chebyshev window, returned as a column vector.

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

More About

An artifact of the equiripple design method used in `chebwin` is the presence of impulses at the endpoints of the time-domain response. The impulses are 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps

Window Designer

Functions

`gausswin` | `kaiser` | `tukeywin` | **WVTool**

Introduced before R2006a

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 `Wp` and `Rp` 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 `Wp`. The resulting bandpass and bandstop designs are of order `2n`.

Note: See “Limitations” on page 1-170 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 `Wp` and `Rp` decibels of passband ripple.

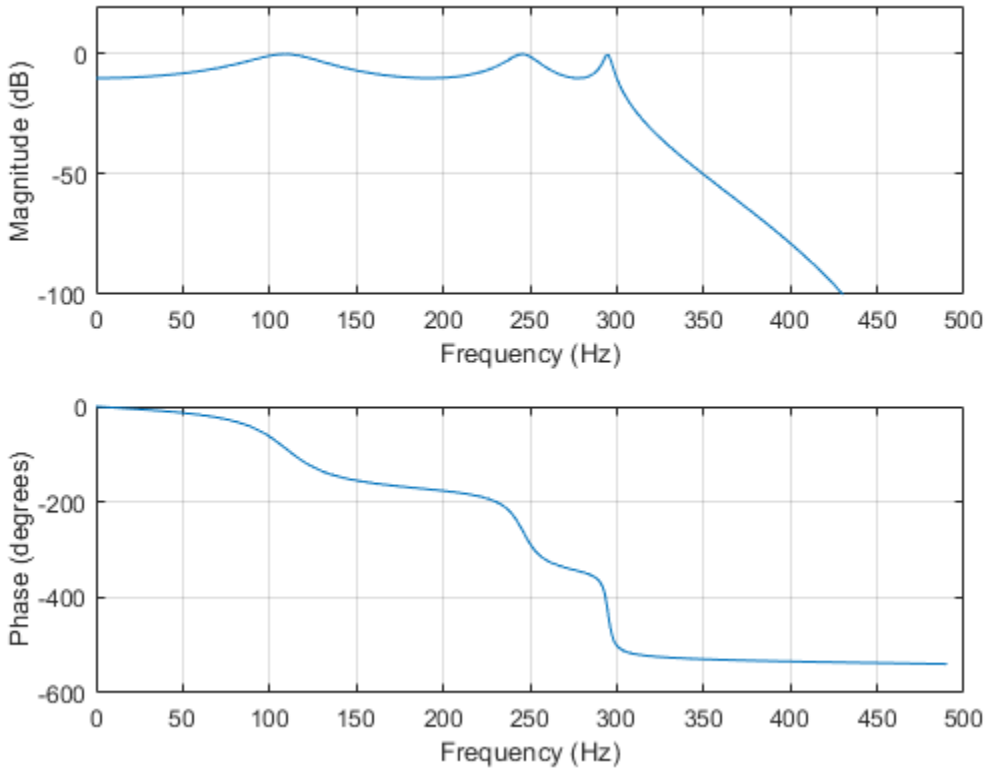
Examples

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 0.6π rad/sample. Plot its magnitude and phase responses. Use it to filter a 1000-sample random signal.

```
fc = 300;
fs = 1000;

[b,a] = cheby1(6,10,fc/(fs/2));
```

```
freqz(b,a,[],fs)

subplot(2,1,1)
ylim([-100 20])
```

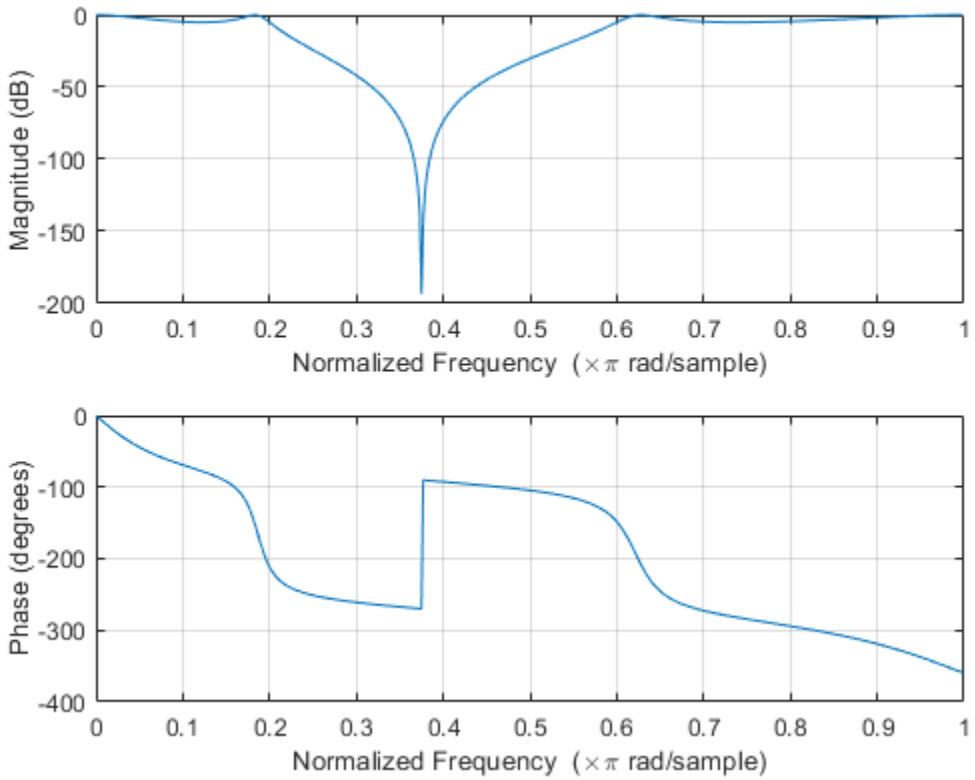



```
dataIn = randn(1000,1);
dataOut = filter(b,a,dataIn);
```

Bandstop Chebyshev Type I Filter

Design a 6th-order Chebyshev Type I bandstop filter with normalized edge frequencies of 0.2π and 0.6π 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)
```

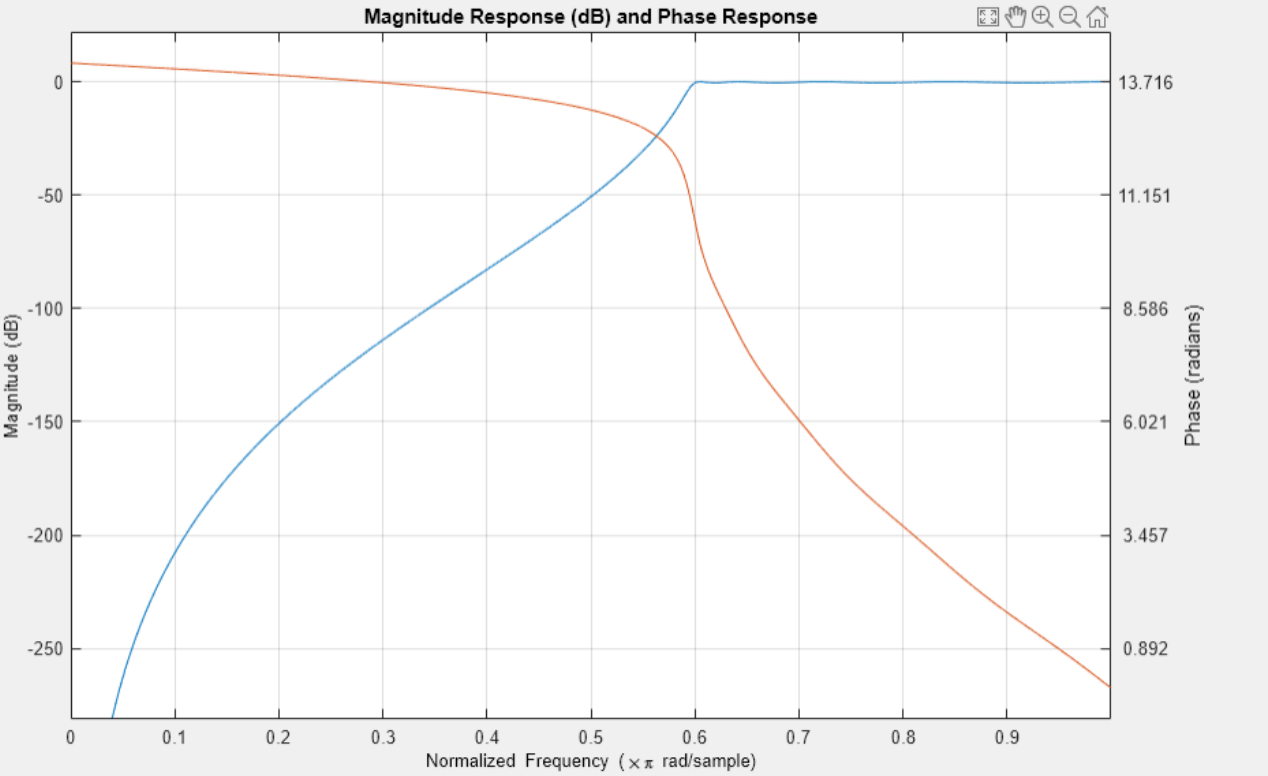



```
dataIn = randn(1000,1);
dataOut = filter(b,a,dataIn);
```

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 0.6π 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')
```



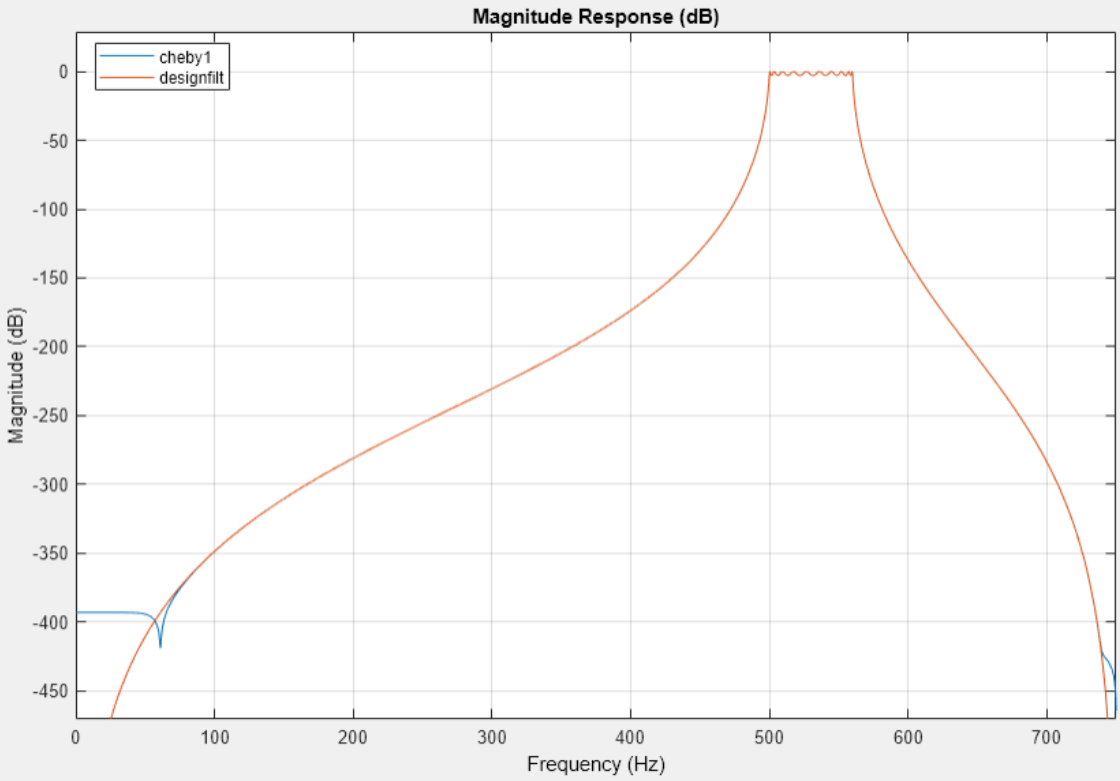
Bandpass Chebyshev Type I Filter

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 2π 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.

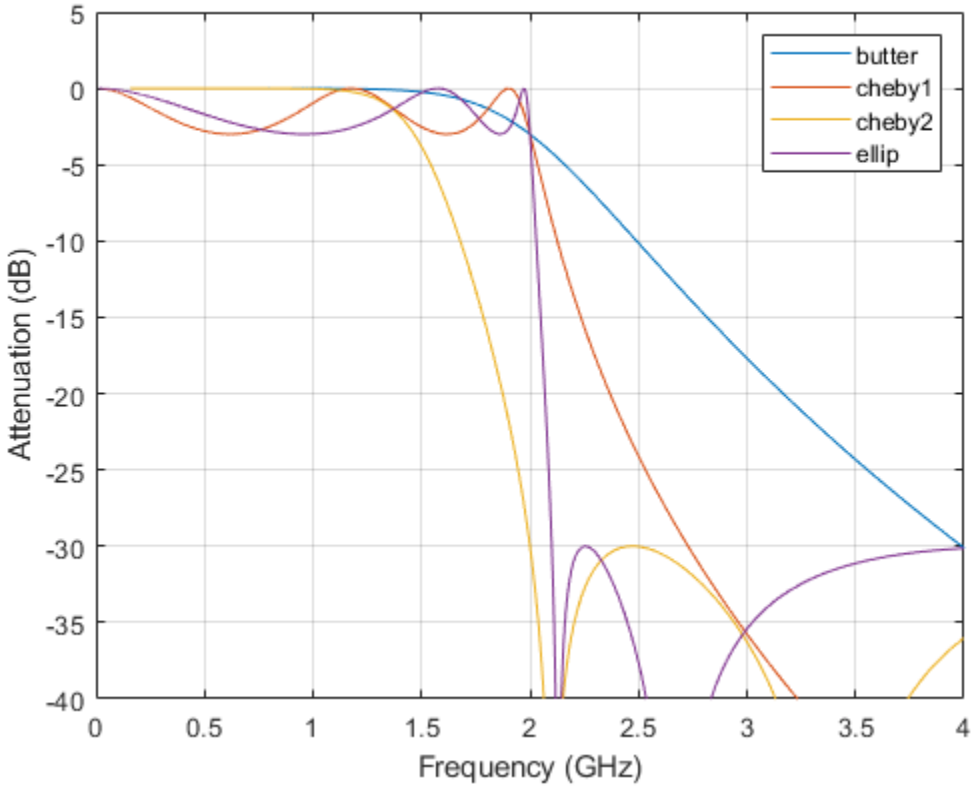
```
[z2,p2,k2] = cheby2(n,30,2*pi*f,'s');
[b2,a2] = zp2tf(z2,p2,k2);
[h2,w2] = freqs(b2,a2,4096);
```

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. For bandpass and bandstop designs, `n` represents one-half the filter order.

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, ℓ, is in linear units, you can convert it to decibels using $Rp = 40 \log_{10}((1+\ell)/(1-\ell))$.

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 –`Rp` decibels. Smaller values of passband ripple, `Rp`, result in wider transition bands.

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

  If `Wp` is the two-element vector `[w1 w2]`, where `w1 < w2`, then `cheby1` designs a bandpass or bandstop filter with lower edge frequency `w1` and higher edge frequency `w2`.
- 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 π 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`

`ftype` — Filter type

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

Filter type, specified as one of the following:

- `'low'` specifies a lowpass filter with passband edge frequency `Wp`. `'low'` is the default for scalar `Wp`.
- `'high'` specifies a highpass filter with passband edge frequency `Wp`.
- `'bandpass'` specifies a bandpass filter of order `2n` if `Wp` is a two-element vector. `'bandpass'` is the default when `Wp` has two elements.
- `'stop'` specifies a bandstop filter of order `2n` if `Wp` is a two-element vector.

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

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

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

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

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

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

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

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

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 1×1.

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

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

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

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

Data Types: `double`

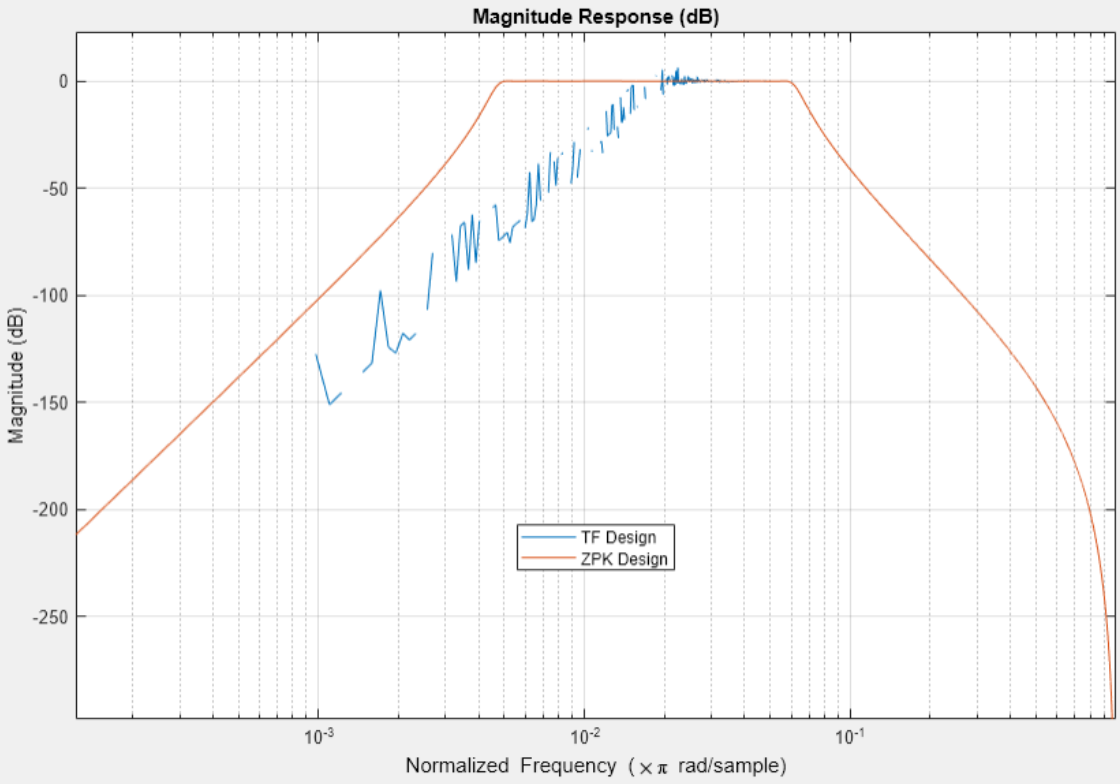
More About

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 `Wp` or `w1` and `w2`.
5. It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`besself` | `butter` | `cheb1ap` | `cheb1ord` | `cheby2` | `designfilt` | `ellip` | `filter` | `sosfilt`

Introduced before R2006a

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 `Ws` and `Rs` 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 `Ws`. The resulting bandpass and bandstop designs are of order `2n`.

Note: See “Limitations” on page 1-181 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 `Ws` and `Rs` decibels of stopband attenuation.

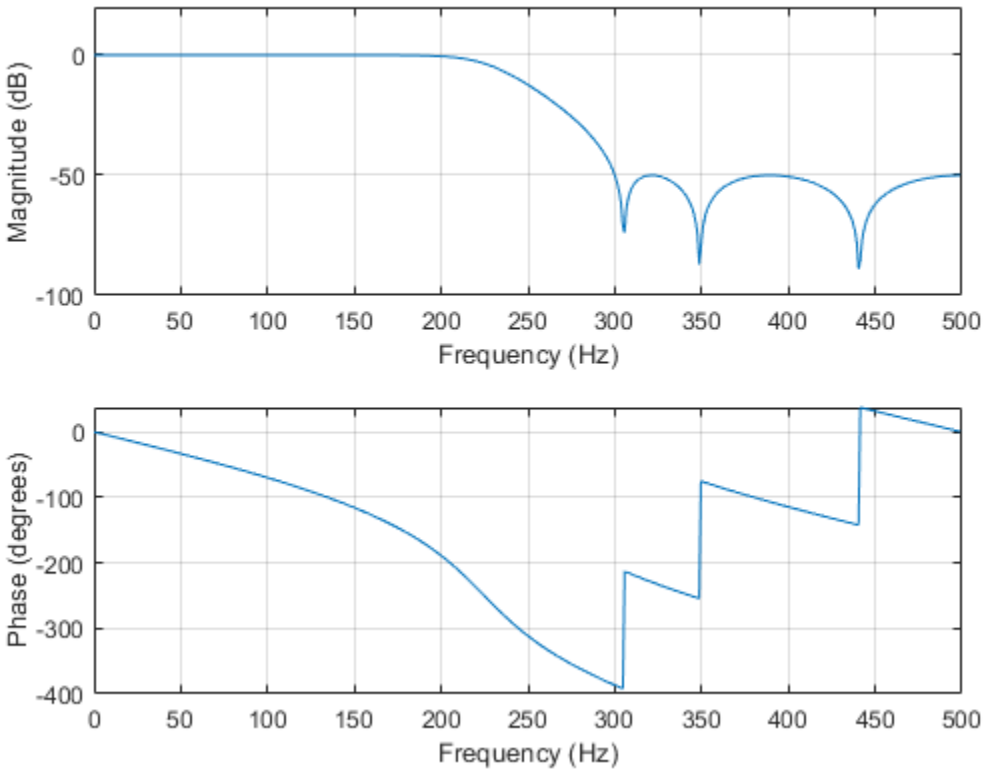
Examples

Lowpass Chebyshev Type II Transfer Function

Design a 6th-order lowpass Chebyshev Type II filter with 50 dB of stopband attenuation and a stopband edge frequency of 300 Hz, which, for data sampled at 1000 Hz, corresponds to 0.6π rad/sample. Plot its magnitude and phase responses. Use it to filter a 1000-sample random signal.

```
fc = 300;
fs = 1000;

[b,a] = cheby2(6,50,fc/(fs/2));
```

```
freqz(b,a,[],fs)

subplot(2,1,1)
ylim([-100 20])
```

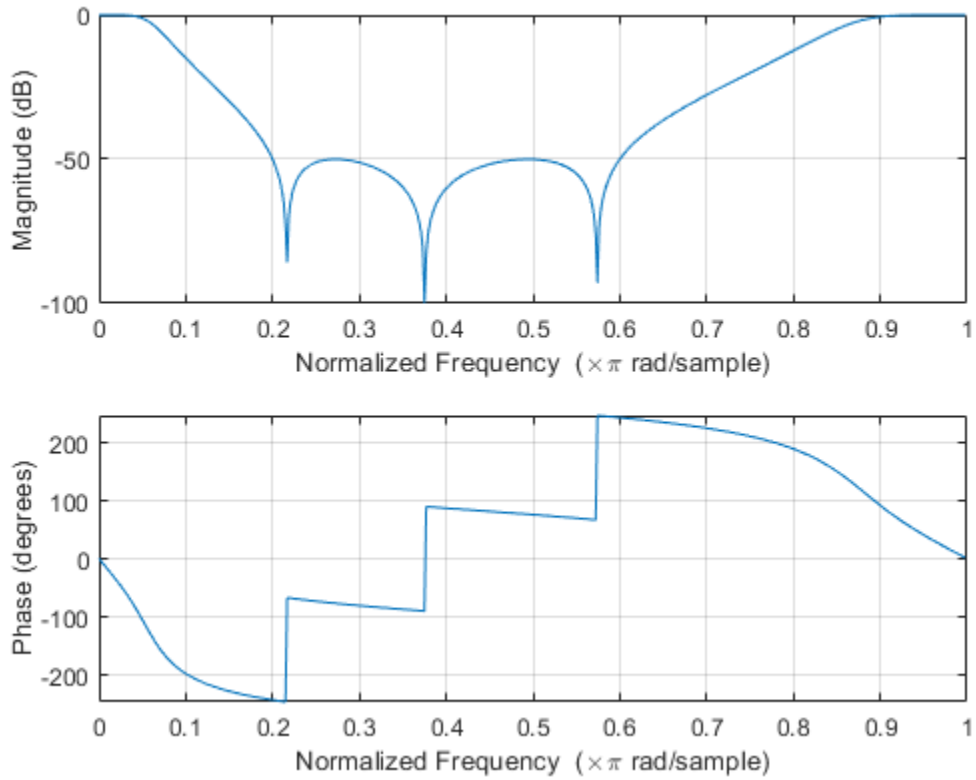



```
dataIn = randn(1000,1);
dataOut = filter(b,a,dataIn);
```

Bandstop Chebyshev Type II Filter

Design a 6th-order Chebyshev Type II bandstop filter with normalized edge frequencies of 0.2π and 0.6π 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)
```

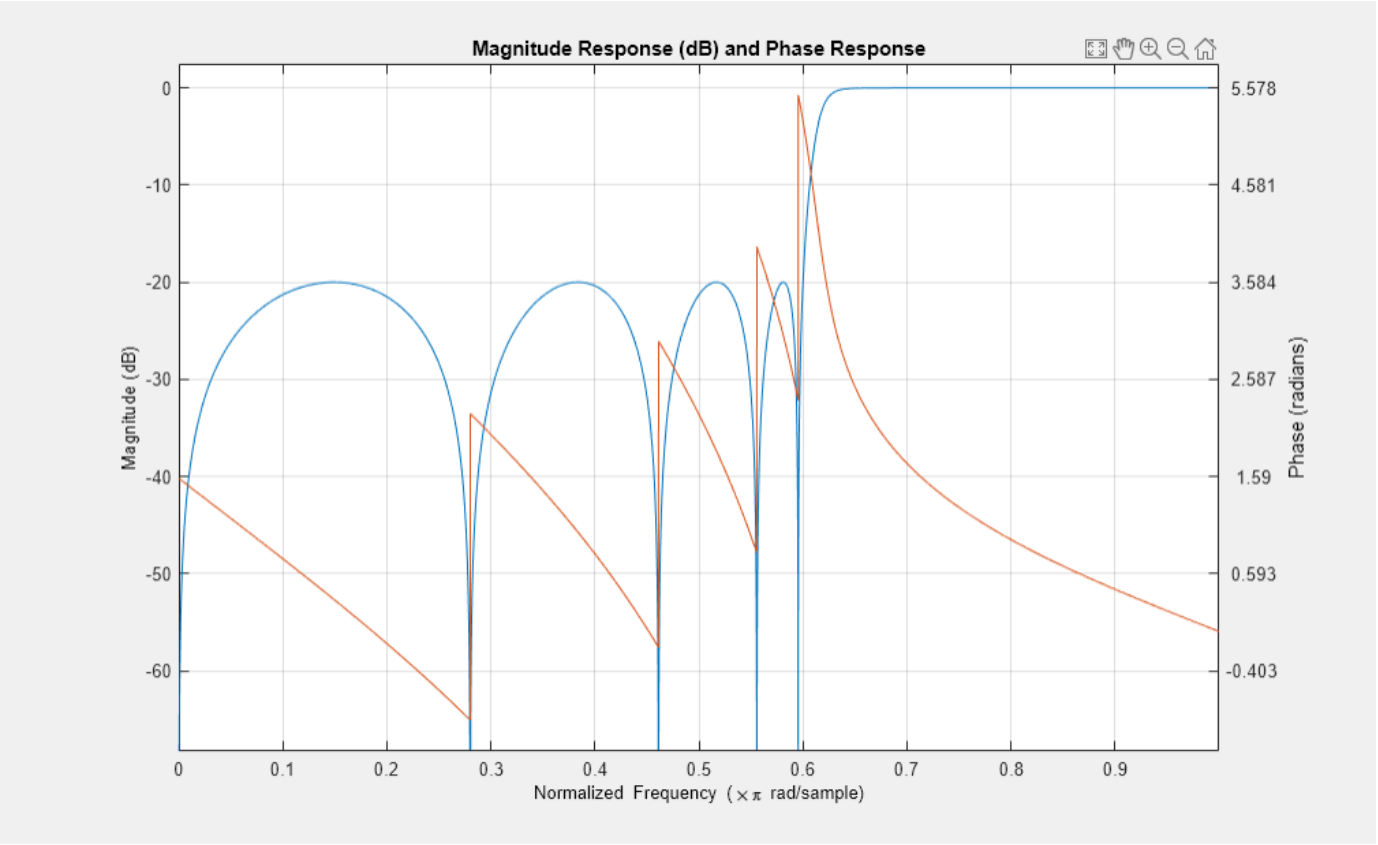



```
dataIn = randn(1000,1);
dataOut = filter(b,a,dataIn);
```

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 0.6π 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')
```



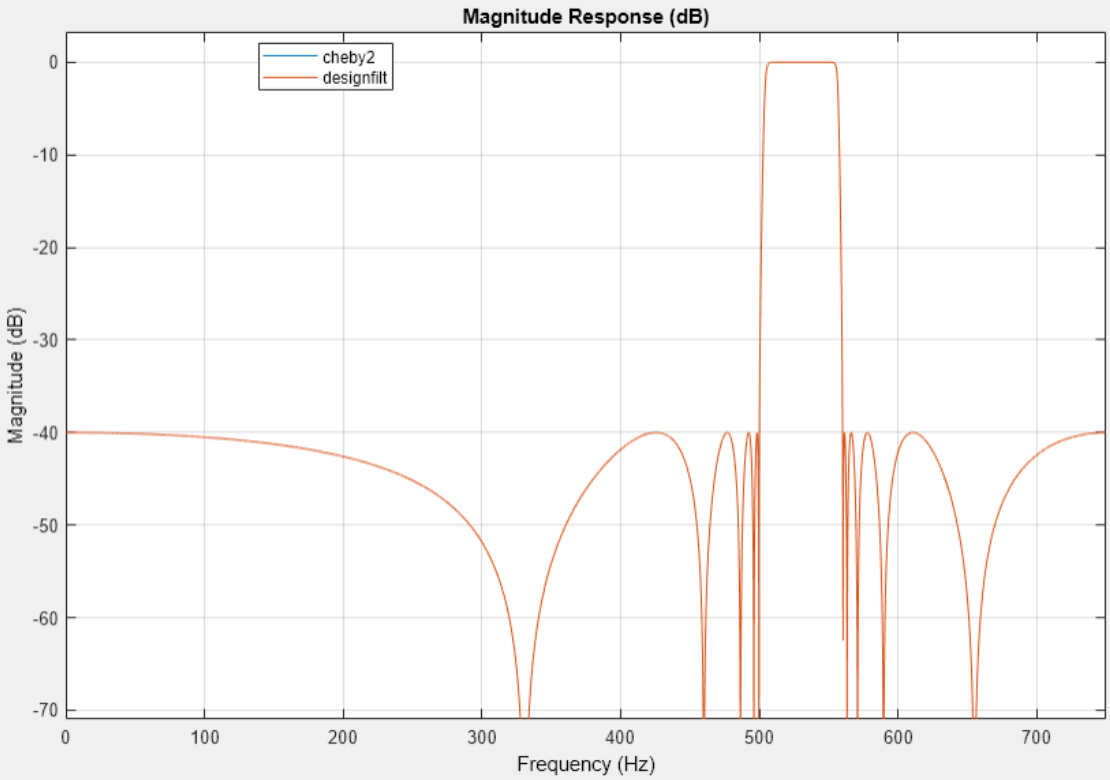
Bandpass Chebyshev Type II Filter

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 `fvtool`.

```
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 2π 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.

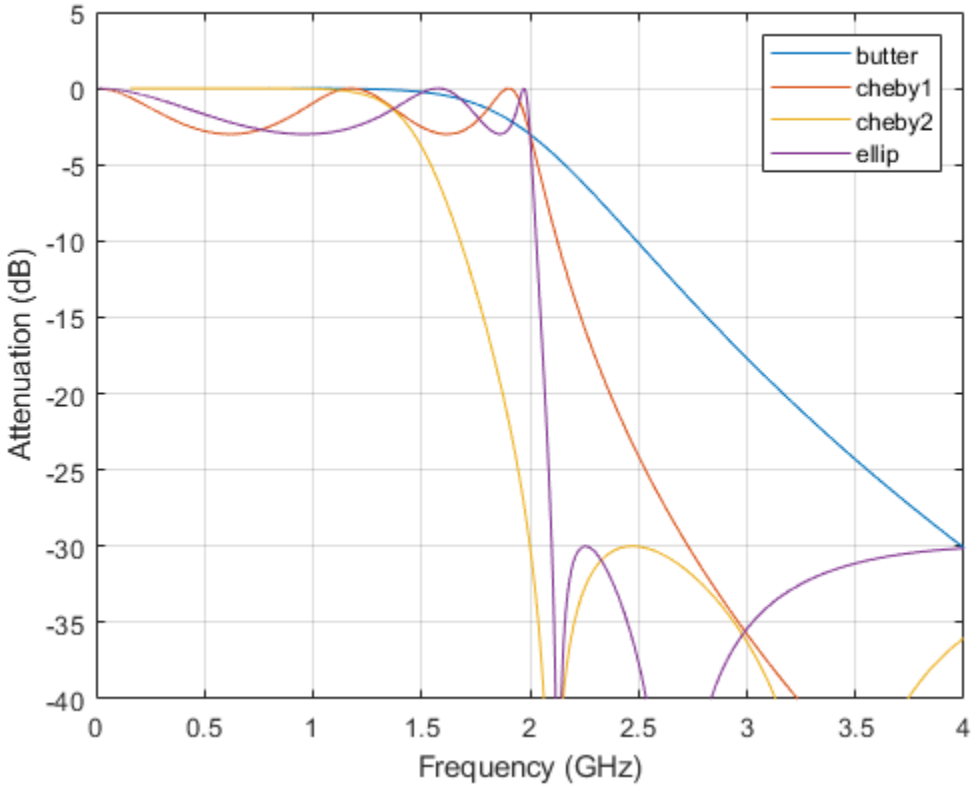
```
[z2,p2,k2] = cheby2(n,30,2*pi*f,'s');
[b2,a2] = zp2tf(z2,p2,k2);
[h2,w2] = freqs(b2,a2,4096);
```

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. For bandpass and bandstop designs, `n` represents one-half the filter order.

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, ℓ, 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 π 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`

`ftype` — Filter type

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

Filter type, specified as one of the following:

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

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

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

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

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

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

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

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

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

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 1×1.

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

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

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

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

Data Types: `double`

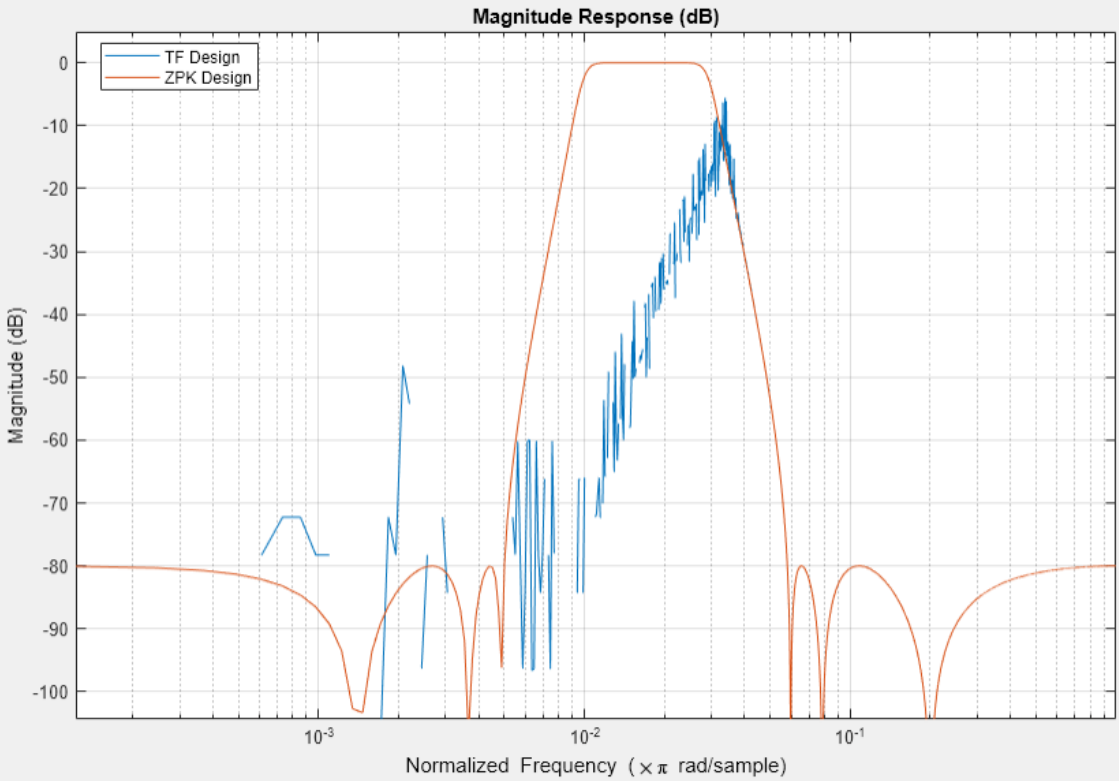
More About

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 `Ws` or `w1` and `w2`.
5. It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

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

Introduced before R2006a

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)

y = chirp(___,cplx)
```

Description

`y = chirp(t,f0,t1,f1)` generates samples of a linear swept-frequency cosine signal at the time instances defined in array `t`. The instantaneous frequency at time 0 is `f0` and the instantaneous frequency at time `t1` is `f1`.

`y = chirp(t,f0,t1,f1,method)` specifies an alternative sweep `method` option.

`y = chirp(t,f0,t1,f1,method,phi)` specifies the initial phase.

`y = chirp(t,f0,t1,f1,'quadratic',phi,shape)` specifies the shape of the spectrogram of a quadratic swept-frequency signal.

`y = chirp(___,cplx)` returns a real chirp if `cplx` is specified as `'real'` and returns a complex chirp if `cplx` is specified as `'complex'`.

Examples

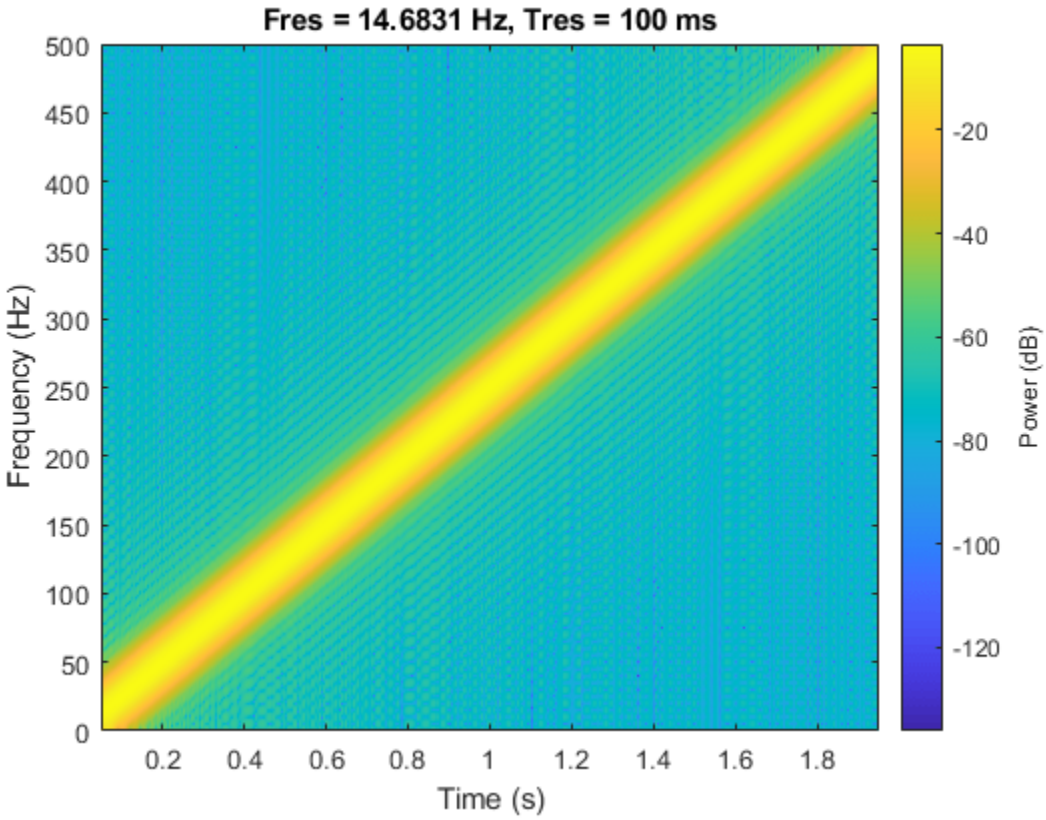
Linear Chirp

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. Divide the signal into segments such that the time resolution is 0.1 second. Specify 99% of overlap between adjoining segments and a spectral leakage of 0.85.

```
pspectrum(y,1e3,'spectrogram','TimeResolution',0.1, ...
    'OverlapPercent',99,'Leakage',0.85)
```



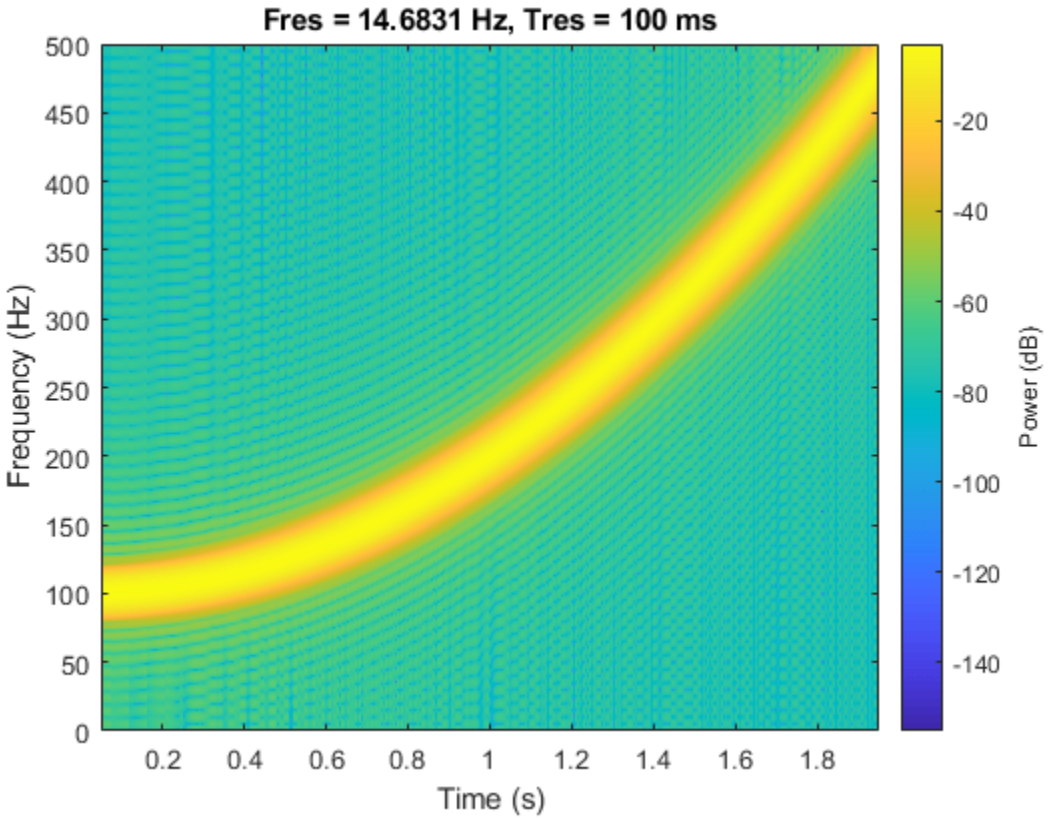
Quadratic Chirp

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. Divide the signal into segments such that the time resolution is 0.1 second. Specify 99% of overlap between adjoining segments and a spectral leakage of 0.85.

```
pspectrum(y,1e3,'spectrogram','TimeResolution',0.1, ...
    'OverlapPercent',99,'Leakage',0.85)
```



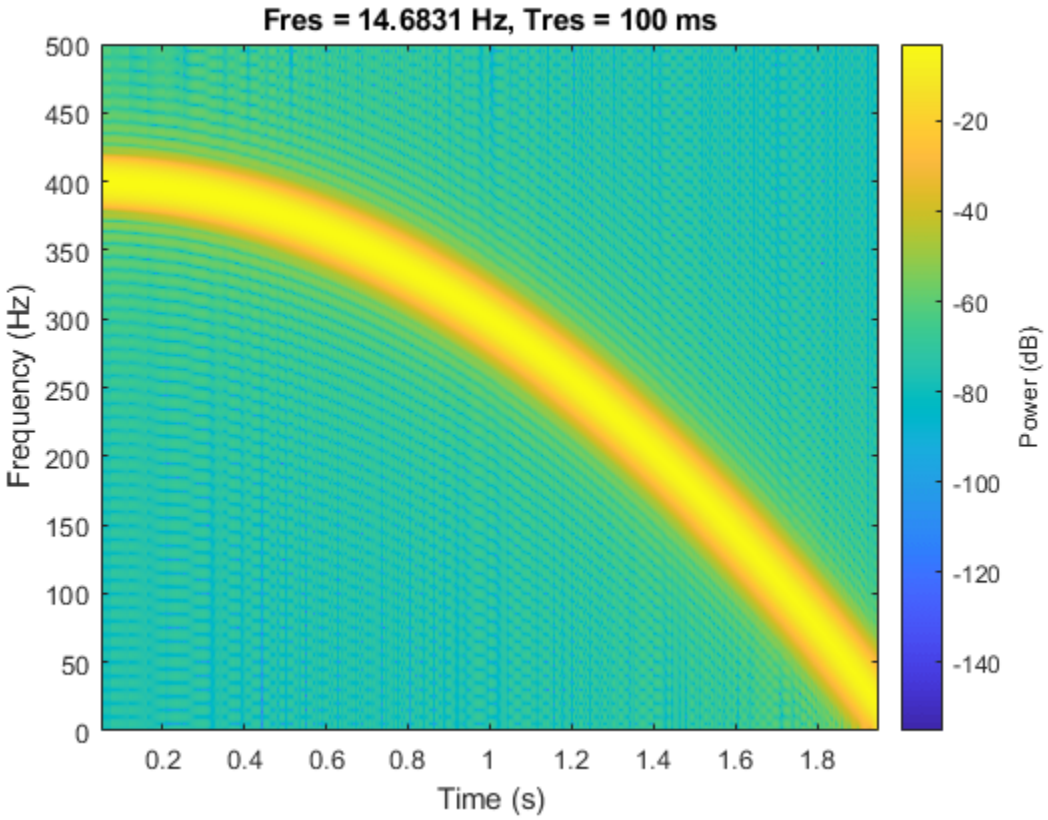
Convex Quadratic Chirp

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. Divide the signal into segments such that the time resolution is 0.1 second. Specify 99% of overlap between adjoining segments and a spectral leakage of 0.85.

```
pspectrum(y,1e3,'spectrogram','TimeResolution',0.1, ...
    'OverlapPercent',99,'Leakage',0.85)
```



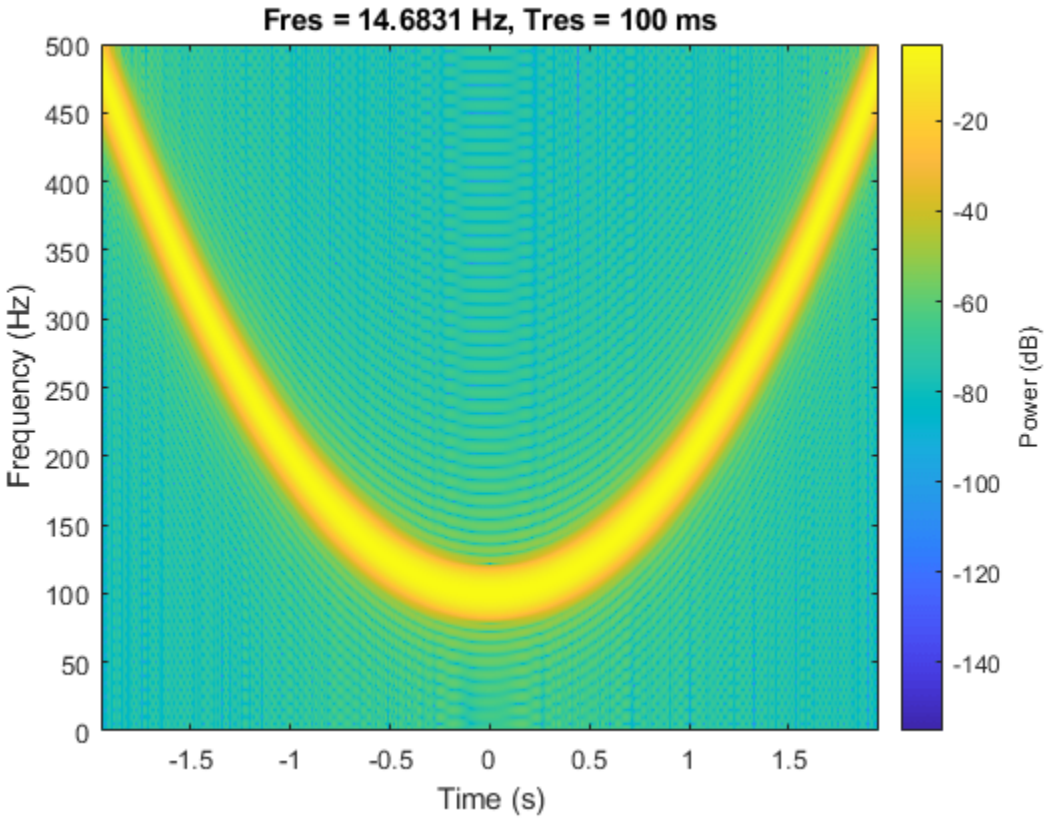
Symmetric Concave Quadratic Chirp

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. Divide the signal into segments such that the time resolution is 0.1 second. Specify 99% of overlap between adjoining segments and a spectral leakage of 0.85.

```
pspectrum(y,t,'spectrogram','TimeResolution',0.1, ...
    'OverlapPercent',99,'Leakage',0.85)
```



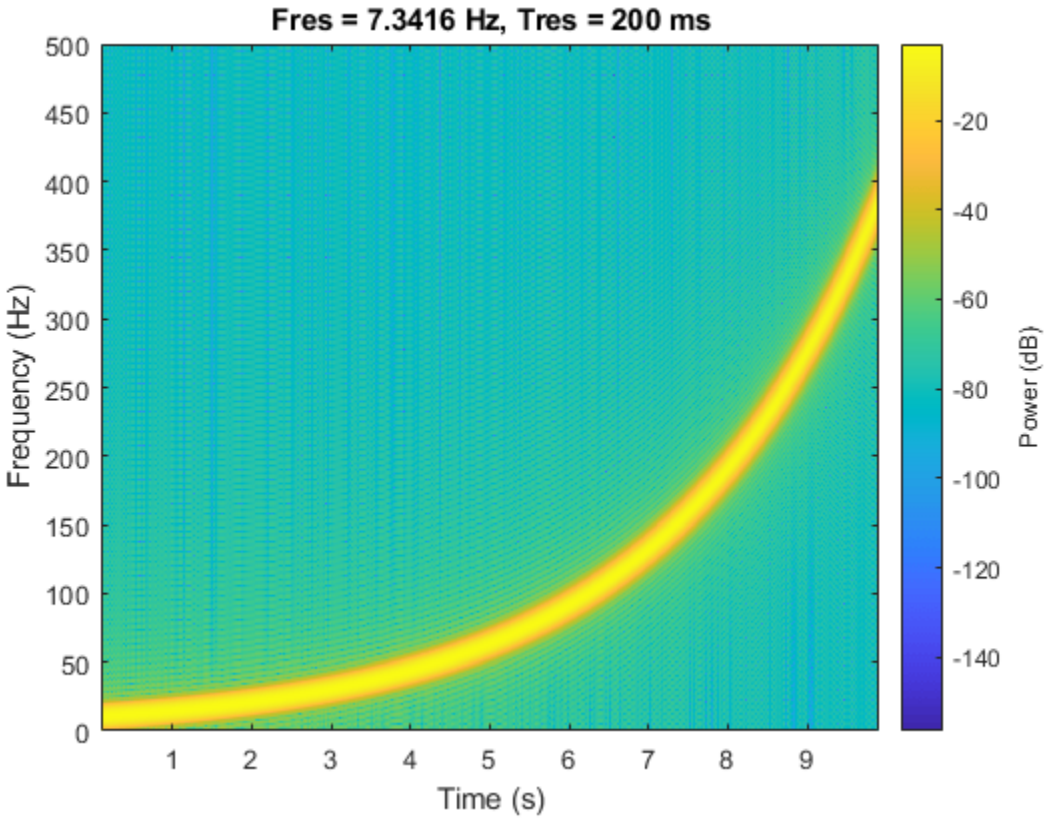
Logarithmic Chirp

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');
```

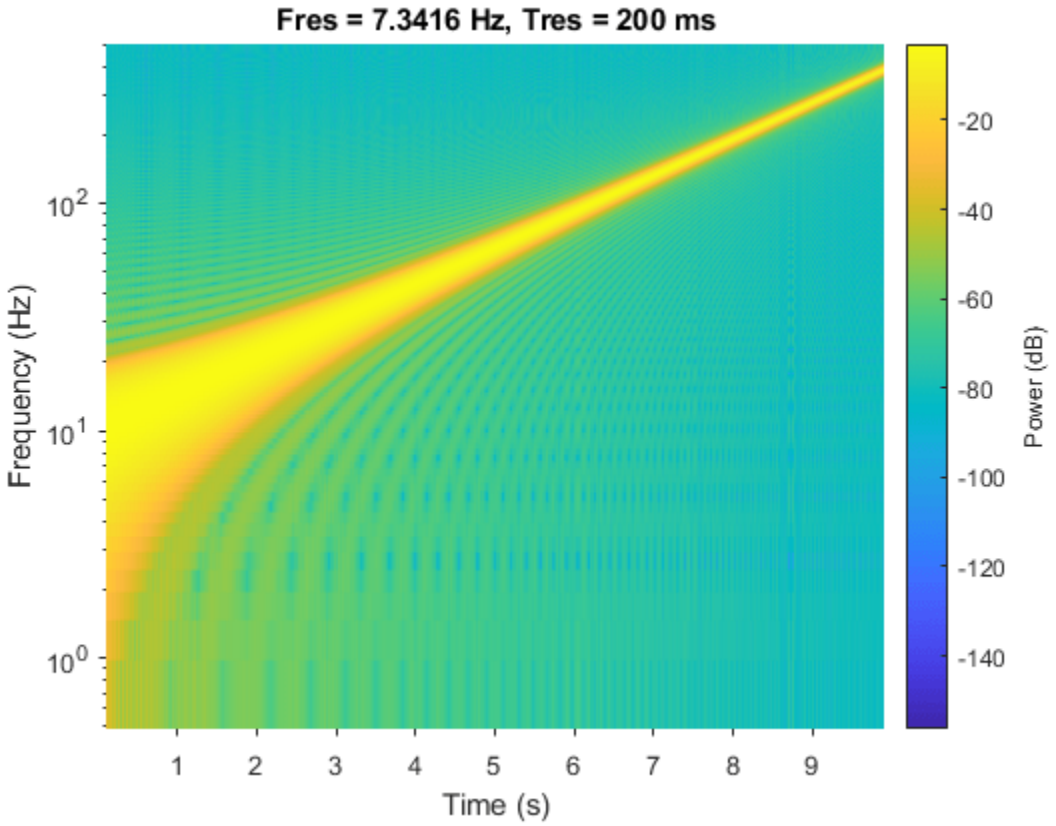
Compute and plot the spectrogram of the chirp. Divide the signal into segments such that the time resolution is 0.2 second. Specify 99% of overlap between adjoining segments and a spectral leakage of 0.85.

```
pspectrum(y,t,'spectrogram','TimeResolution',0.2, ...
    'OverlapPercent',99,'Leakage',0.85)
```



Use a logarithmic scale for the frequency axis. The spectrogram becomes a line, with high uncertainty at low frequencies.

```
ax = gca;
ax.YScale = 'log';
```



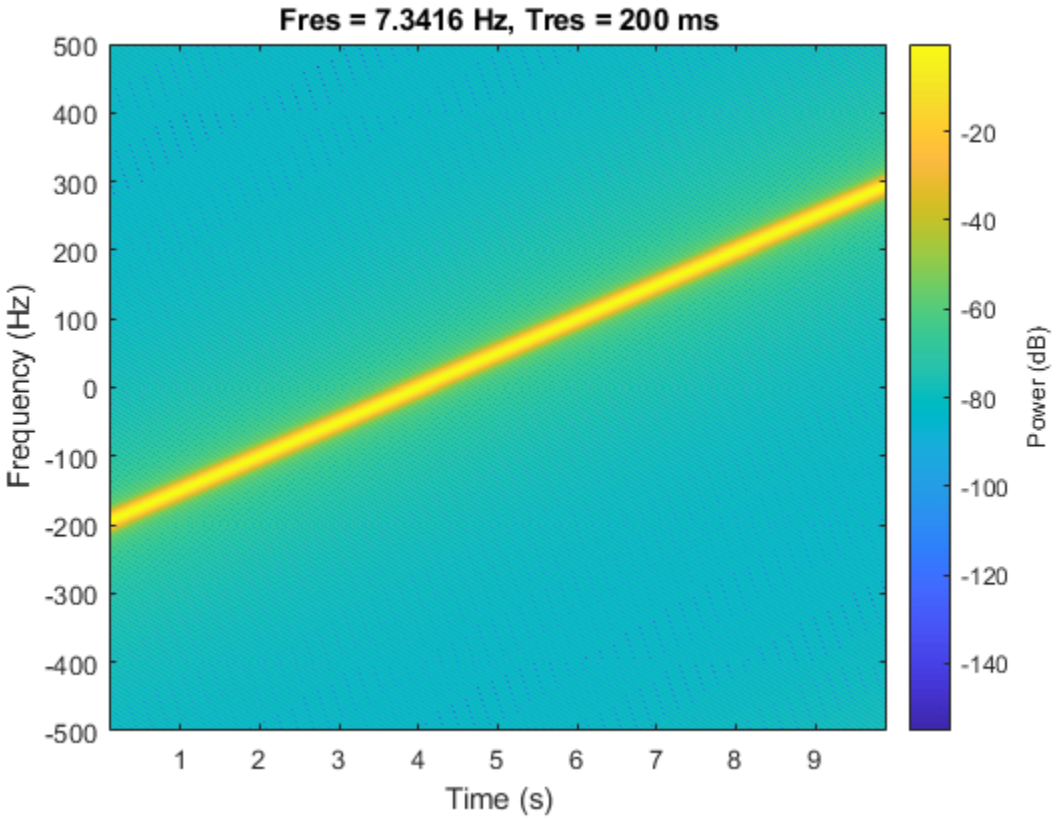
Complex Chirp

Generate a complex linear chirp sampled at 1 kHz for 10 seconds. The instantaneous frequency is –200 Hz initially and 300 Hz at the end. The initial phase is zero.

```
t = 0:1/1e3:10;
fo = -200;
f1 = 300;

y = chirp(t,fo,t(end),f1,'linear',0,'complex');
```

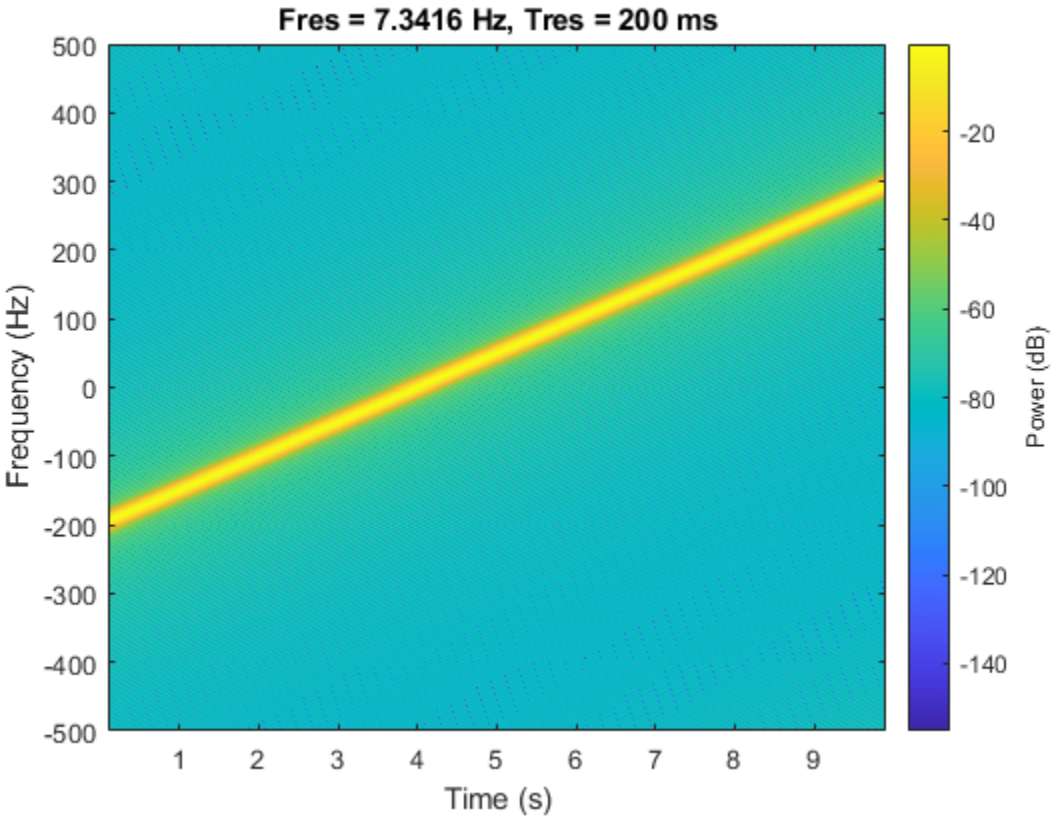
Compute and plot the spectrogram of the chirp. Divide the signal into segments such that the time resolution is 0.2 second. Specify 99% of overlap between adjoining segments and a spectral leakage of 0.85.

```
pspectrum(y,t,'spectrogram','TimeResolution',0.2, ...
    'OverlapPercent',99,'Leakage',0.85)
```



Verify that a complex chirp has real and imaginary parts that are equal but with 90˚ phase difference.

```
x = chirp(t,fo,t(end),f1,'linear',0) + 1j*chirp(t,fo,t(end),f1,'linear',-90);

pspectrum(x,t,'spectrogram','TimeResolution',0.2, ...
    'OverlapPercent',99,'Leakage',0.85)
```



Input Arguments

`t` — Time array

vector

Time array, specified as a vector.

Data Types: `single` | `double`

`f0` — Instantaneous frequency at time 0

`0` (default) | real scalar in Hz

Initial instantaneous frequency at time 0, specified as a real scalar expressed in Hz.

Data Types: `single` | `double`

`t1` — Reference time

`1` (default) | positive scalar in seconds

Reference time, specified as a positive scalar expressed in seconds.

Data Types: `single` | `double`

`f1` — Instantaneous frequency at time `t1`

`100` (default) | real scalar in Hz

Instantaneous frequency at time `t1`, specified as a real scalar expressed in Hz.

Data Types: `single` | `double`

`method` — Sweep method

`'linear'` (default) | `'quadratic'` | `'logarithmic'`

Sweep method, specified as `'linear'`, `'quadratic'`, or `'logarithmic'`.

- `'linear'` — 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. The coefficient β ensures that the desired frequency breakpoint f_1 at time t_1 is maintained.
- `'quadratic'` — 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 10^{-6}.

`phi` — Initial phase

`0` (default) | positive scalar in degrees

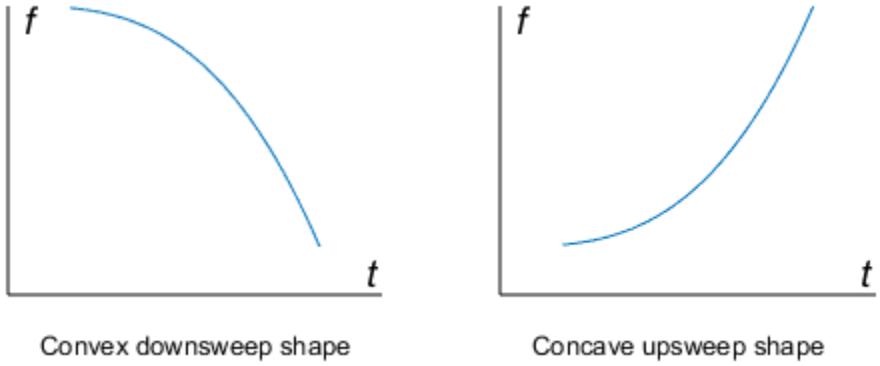
Initial phase, specified as a positive scalar expressed in degrees.

Data Types: `single` | `double`

`shape` — Spectrogram shape of quadratic chirp

`'convex'` | `'concave'`

Spectrogram shape of quadratic chirp, specified as `'convex'` or `'concave'`. `shape` describes the shape of the parabola with respect to the positive frequency axis. If not specified, `shape` is `'convex'` for the downsweep case with $f_0 > f_1$, and `'concave'` for the upsweep case with $f_0 < f_1$.



`cplx` — Output complexity
`'real'` (default) | `'complex'`

Output complexity, specified as `'real'` or `'complex'`.

Output Arguments

`y` — Swept-frequency cosine signal
vector

Swept-frequency cosine signal, returned as a vector.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Thread-Based Environment
Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

This function fully supports thread-based environments. For more information, see “Run MATLAB Functions in Thread-Based Environment”.

See Also
`cos` | `diric` | `gauspuls` | `pulstran` | `rectpuls` | `sawtooth` | `sin` | `sinc` | `square` | `tripuls`

Introduced before R2006a

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 an `n`-element vector, `x`, is the convolution of `h` and `x`.

Examples

Efficient Computation of Convolution

Computing a convolution using `conv` when the signals are vectors is generally more efficient than using `convmtx`. 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 times 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
```

```
t1col = 1×2

    0.0007    0.0144
```

```
t1rat = tcnv\tmtx
```

```
t1rat = 21.4239
```

`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
```

```
tmcol = 1×2

    0.3217    0.0513
```

```
tmrat = tcnv/tmtx
```

```
tmrat = 6.2767
```

`convmtx` is about three times as efficient as `conv`.

Input Arguments

h — Input vector

vector

Input vector, specified as a row or column.

Data Types: `single` | `double`

n — Length of vector to convolve

positive integer

Length of vector to convolve, specified as a positive integer.

- 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`.

Output Arguments

`A` — Convolution matrix

matrix

Convolution matrix of input `h` and the vector `x`, returned as a matrix.

Algorithms

- `convmtx` uses the function `toeplitz` to generate the convolution matrix.
- `convmtx` handles edge conditions by zero padding.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

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

Introduced before R2006a

corrmtx

Data matrix for autocorrelation matrix estimation

Syntax

```
H = corrmtx(x,m)
H = corrmtx(x,m,method)
[H,r] = corrmtx( ___ )
```

Description

`H = corrmtx(x,m)` returns an (*n*+`m`)-by-(`m`+1) rectangular Toeplitz matrix $\mathbf{H}$ = `H` such that $\mathbf{H}^{\dagger}\mathbf{H}$ is a biased estimate of the autocorrelation matrix for the input vector `x`. *n* is the length of `x`, `m` is the prediction model order, and $\mathbf{H}^{\dagger}$ is the conjugate transpose of $\mathbf{H}$.

`H = corrmtx(x,m,method)` computes the matrix `H` according to the method specified by `method`.

`[H,r] = corrmtx( ___ )` also returns the (`m` + 1)-by-(`m` + 1) autocorrelation matrix estimate `r`, computed as $\mathbf{H}^{\dagger}\mathbf{H}$, for any of the previous syntaxes.

Examples

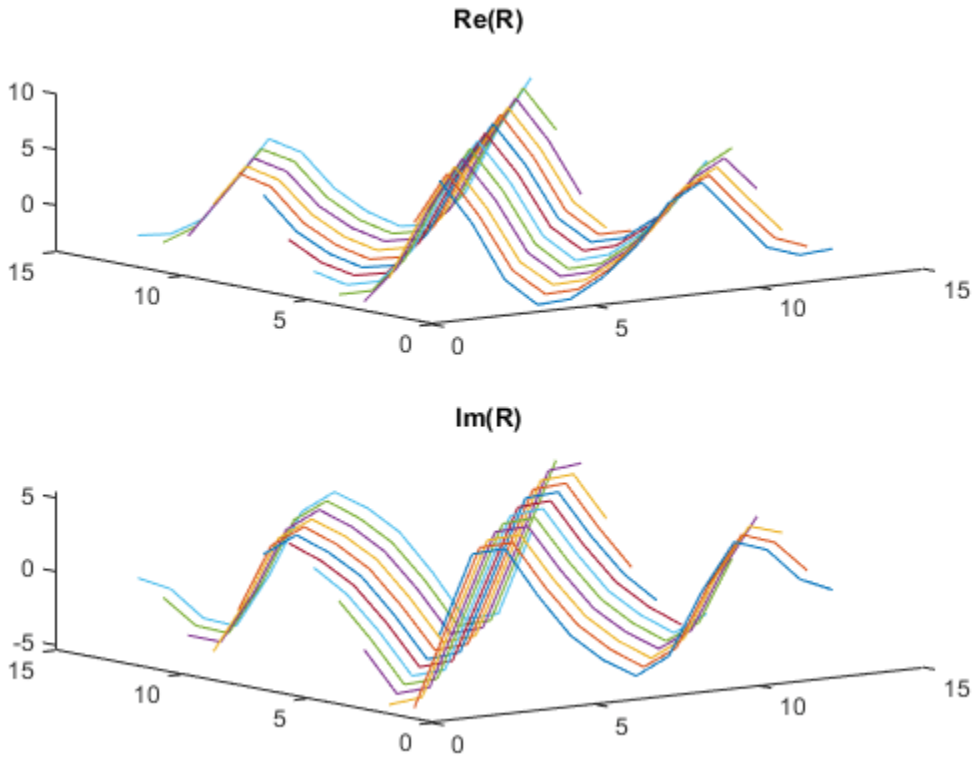
Modified Data and Autocorrelation Matrices

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)')
```



Input Arguments

x — Input data

vector

Input data, specified as a vector.

m — Prediction model order

positive real integer

Prediction model order, specified as a positive real integer.

method — Matrix computation method

`'autocorrelation'` (default) | `'prewindowed'` | `'postwindowed'` | `'covariance'` | `'modified'`

Matrix computation method, specified as `'autocorrelation'`, `'prewindowed'`, `'postwindowed'`, `'covariance'` or `'modified'`.

- `'autocorrelation'`: (default) `H` 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 model. The matrix can be used to perform autoregressive parameter estimation using the Yule-Walker method. For more details, see `aryule`.

- `'prewindowed'`: `H` 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 model.
- `'postwindowed'`: `H` 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 model.
- `'covariance'`: `H` 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 model. The matrix can be used to perform autoregressive parameter estimation using the covariance method. For more details, see `arcov`.
- `'modified'`: `H` 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 model. The matrix can be used to perform autoregressive parameter estimation using the modified covariance method. For more details, see `armcov`.

Output Arguments

`H` — Data matrix

matrix

Data matrix, returned for autocorrelation matrix estimation. The size of `H` depends on the matrix computation method specified in `method`.

`r` — Biased autocorrelation matrix

matrix

Biased autocorrelation matrix, returned as a (`m` + 1)-by-(`m` + 1) rectangular Toeplitz matrix.

Algorithms

The Toeplitz data matrix computed by `corrmtx` depends on the method you select. The matrix determined by the autocorrelation (default) method is:

$$\mathbf{H} = \frac{1}{\sqrt{n}}\begin{bmatrix} x(1) & 0 & \cdots & 0 & 0 \\ x(2) & x(1) & \cdots & 0 & 0 \\ x(3) & x(2) & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x(m) & x(m-1) & \cdots & x(1) & 0 \\ x(m+1) & x(m) & \cdots & x(2) & x(1) \\ x(m+2) & x(m+1) & \cdots & x(3) & x(2) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x(n-1) & x(n-2) & \cdots & x(n-m) & x(n-m-1) \\ x(n) & x(n-1) & \cdots & x(n-m+1) & x(n-m) \\ 0 & x(n) & \cdots & x(n-m+2) & x(n-m+1) \\ \vdots & \vdots & \iddots & \vdots & \vdots \\ 0 & 0 & \cdots & x(n-1) & x(n-2) \\ 0 & 0 & \cdots & x(n) & x(n-1) \\ 0 & 0 & \cdots & 0 & x(n) \end{bmatrix}.$$

In the 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 `H` of `corrmtx` for each method:

- `'autocorrelation'` — (default) `H` = **H**.
- `'prewindowed'` — `H` is the n-by-$(m + 1)$ submatrix of **H** whose first row is $[x(1) \ldots 0]$ and whose last row is $[x(n) \ldots x(n – m)]$.
- `'postwindowed'` — `H` is the n-by-$(m + 1)$ submatrix of **H** whose first row is $[x(m + 1) \ldots x(1)]$ and whose last row is $[0 \ldots x(n)]$.
- `'covariance'` — `H` is the $(n – m)$-by-$(m + 1)$ submatrix of **H** whose first row is $[x(m + 1) \ldots x(1)]$ and whose last row is $[x(n) \ldots x(n – m)]$.
- `'modified'` — `H` is the $2(n – m)$-by-$(m + 1)$ matrix $\mathbf{H}_{\text{mod}}$ defined by

$$\mathbf{H}_{\text{mod}} = \frac{1}{\sqrt{2(n-m)}}\begin{bmatrix} x(m+1) & \cdots & x(1) \\ \vdots & \iddots & \vdots \\ x(n) & \cdots & x(n-m) \\ x^*(1) & \cdots & x^*(m+1) \\ \vdots & \ddots & \vdots \\ x^*(n-m) & \cdots & x^*(n) \end{bmatrix}.$$

References

[1] Marple, S. Lawrence. *Digital Spectral Analysis: With Applications*. Prentice-Hall Signal Processing Series. Englewood Cliffs, N.J: Prentice-Hall, 1987.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

peig | pmusic | rooteig | rootmusic | xcorr

Introduced before R2006a

countlabels

Count number of unique labels

Syntax

```
cnt = countlabels(lblsrc)
cnt = countlabels(lblsrc,Name,Value)
```

Description

Use this function when you are working on a machine or deep learning classification problem and you want to look at the proportions of label values in your dataset.

`cnt = countlabels(lblsrc)` counts the number of unique label category values in `lblsrc` and returns the count in `cnt`.

`cnt = countlabels(lblsrc,Name,Value)` specifies additional input arguments using name-value pairs. For example, `'TableVariable','Color'` reads the labels corresponding to `'Color'`.

Examples

Count Labels in Arrays

Categorical Arrays

Generate a categorical array with the categories `A`, `B`, `C`, and `D`. The array contains samples of each category.

```
lbls = categorical(["B" "C" "A" "D" "B" "A" "A" "B" "C" "A"], ...
    ["A" "B" "C" "D"])
```

```
lbls = 1x10 categorical
     B      C      A      D      B      A      A      B      C      A
```

Count the number of unique label category values in the array.

```
cnt = countlabels(lbls)
```

```
cnt=4×3 table
    Label    Count    Percent
    _____    _____    _______

      A        4        40
      B        3        30
      C        2        20
      D        1        10
```

Generate a second categorical array with the same categories. The array contains samples of each category and one sample with a missing value.

```
mlbls = categorical(["B" "C" "A" "D" "B" "A" missing "B" "C" "A"], ...
    ["A" "B" "C" "D"])

mlbls = 1x10 categorical
  Columns 1 through 9

     B      C      A      D      B      A      <undefined>      B      C

  Column 10

     A
```

Count the number of unique label category values in the array. The sample with a missing value is included in the count as `<undefined>`.

```
mcnt = countlabels(mlbls)

mcnt=5×3 table
       Label       Count    Percent
    ___________    _____    _______

    A                3        30
    B                3        30
    C                2        20
    D                1        10
    <undefined>      1        10
```

Character Arrays

Read William Shakespeare's sonnets with the `fileread` function. Remove all nonalphabetic characters from the text and convert to lowercase.

```
sonnets = fileread("sonnets.txt");
letters = lower(sonnets(regexp(sonnets,"[A-z]")))';
```

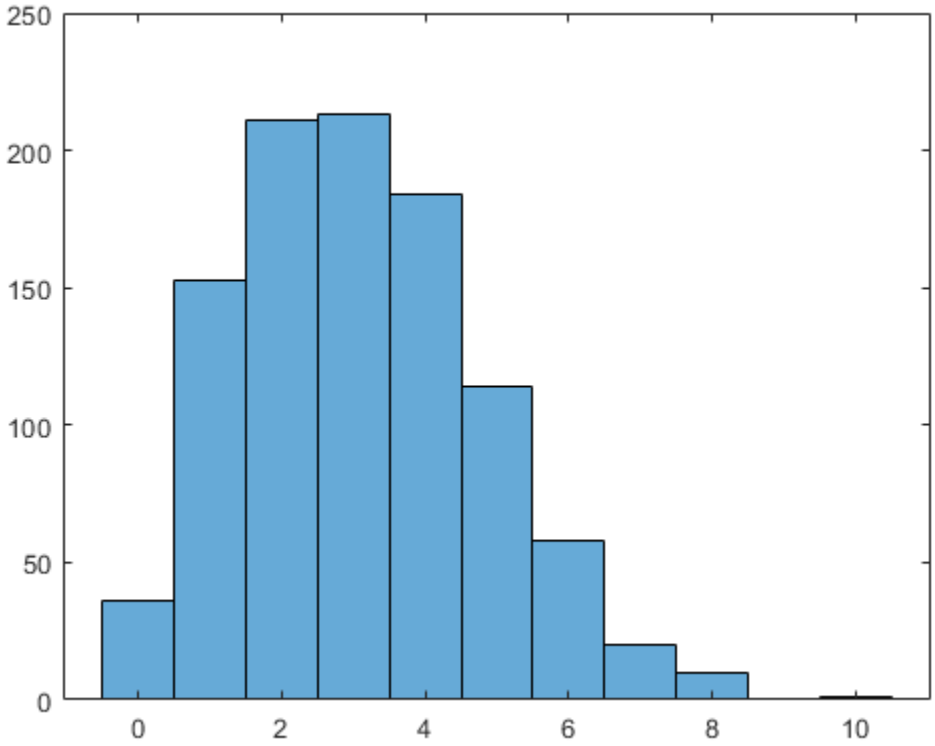
Count how many times each letter appears in the sonnets. List the letters that appear most often.

```
cnt = countlabels(letters);
cnt = sortrows(cnt,"Count","descend");
head(cnt)

ans=8×3 table
    Label    Count    Percent
    _____    _____    _______

      e      9028     12.298
      t      7210     9.8216
      o      5710     7.7782
      h      5064     6.8982
      s      4994     6.8029
      a      4940     6.7293
      i      4895      6.668
      n      4522     6.1599
```

Numeric Arrays

Use the `poisrand` on page 1-0 function to generate an array of 1000 random integers from the Poisson distribution with rate parameter 3. Plot a histogram of the results.

```
N = 1000;
lam = 3;

nums = zeros(N,1);
for jk = 1:N
    nums(jk) = poisrand(lam);
end

histogram(nums)
```



Count the frequencies of the integers represented in the array.

```
mm = countlabels(nums)
```

```
mm=10×3 table
    Label    Count    Percent
    _____    _____    _______

     0         36       3.6
     1        153      15.3
     10         1       0.1
     2        211      21.1
     3        213      21.3
```

```
     4        184      18.4
     5        114      11.4
     6         58       5.8
     7         20         2
     8         10         1
```

```
function num = poisrand(lam)
% Poisson random integer using rejection method
    p = 0;
    num = -1;
    while p <= lam
        p = p - log(rand);
        num = num + 1;
    end
end
```

Count Labels in Tables and Datastores

Create a table of characters with two variables. The first variable `Type1` contains instances of the letters *P*, *Q*, and *R*. The second variable `Type2` contains instances of the letters *A*, *B*, and *D*.

```
tbl = table(["P" "R" "P" "Q" "Q" "Q" "R" "P"]', ...
    ["A" "B" "B" "A" "D" "D" "A" "A"]',...
    'VariableNames',["Type1","Type2"]);
```

Count how many times each letter appears in each of the table variables.

```
cnt = countlabels(tbl,'TableVariable','Type1')
```

```
cnt=3×3 table
    Type1    Count    Percent
    _____    _____    _______

      P        3       37.5
      Q        3       37.5
      R        2         25
```

```
cnt = countlabels(tbl,'TableVariable','Type2')
```

```
cnt=3×3 table
    Type2    Count    Percent
    _____    _____    _______

      A        4        50
      B        2        25
      D        2        25
```

Create an `ArrayDatastore` object containing the table.

```
ads = arrayDatastore(tbl,'OutputType','same');
```

Count how many times each letter appears in each of the table variables.

```
cnt = countlabels(ads,'TableVariable','Type1')
```

```
cnt=3×3 table
    Type1    Count    Percent
    _____    _____    _______

      P        3       37.5
      Q        3       37.5
      R        2         25
```

```
cnt = countlabels(ads,'TableVariable','Type2')
```

```
cnt=3×3 table
    Type2    Count    Percent
    _____    _____    _______

      A        4         50
      B        2         25
      D        2         25
```

Input Arguments

`lblsrc` — Input label source

categorical vector | string vector | logical vector | numeric vector | cell array | table | datastore | `CombinedDatastore` object

Input label source, specified as one of these:

- A categorical vector.
- A string vector or a cell array of character vectors.
- A numeric vector or a cell array of numeric scalars.
- A logical vector or a cell array of logical scalars.
- A table with variables containing any of the previous data types.
- A datastore whose `readall` function returns any of the previous data types.
- A `CombinedDatastore` object containing an underlying datastore whose `readall` function returns any of the previous data types. In this case, you must specify the index of the underlying datastore that has the label values.

`lblsrc` must contain labels that can be converted to a vector with a discrete set of categories.

Example: `lblsrc = categorical(["B" "C" "A" "E" "B" "A" "A" "B" "C" "A"],["A" "B" "C" "D"])` creates the label source as a ten-sample categorical vector with four categories: `A`, `B`, `C`, and `D`.

Example: `lblsrc = [0 7 2 5 11 17 15 7 7 11]` creates the label source as a ten-sample numeric vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `table` | `cell` | `categorical`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'TableVariable','Sex','UnderlyingDatastoreIndex',5` reads the labels corresponding to `'Sex'` only in the fifth underlying datastore of a combined datastore.

`TableVariable` — Table variable to read

first table variable (default) | character vector | string scalar

Table variable to read, specified as a character vector or string scalar. If this argument is not specified, then `countlabels` uses the first table variable.

`UnderlyingDatastoreIndex` — Underlying datastore index

integer scalar

Underlying datastore index, specified as an integer scalar. This argument applies when `lblsrc` is a `CombinedDatastore` object. `countlabels` counts the labels in the datastore obtained using the `UnderlyingDatastores` property of `lblsrc`.

Output Arguments

`cnt` — Unique label counts

table

Unique label counts, returned as a table with these variables:

- `Label` — Unique label category values. If `'TableVariable'` is specified, then the `Label` name is replaced with the table variable name.
- `Count` — Number of instances of each label value.
- `Percent` — Proportion of each label value, expressed as a percentage.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition` | `folders2labels` | `splitlabels`

Introduced in R2021a

cpsd

Cross power spectral density

Syntax

```
pxy = cpsd(x,y)

pxy = cpsd(x,y,window)
pxy = cpsd(x,y,window,noverlap)
pxy = cpsd(x,y,window,noverlap,nfft)

pxy = cpsd( ___ ,'mimo')

[pxy,w] = cpsd( ___ )
[pxy,f] = cpsd( ___ ,fs)

[pxy,w] = cpsd(x,y,window,noverlap,w)
[pxy,f] = cpsd(x,y,window,noverlap,f,fs)

[ ___ ] = cpsd(x,y, ___ ,freqrange)

cpsd( ___ )
```

Description

`pxy = cpsd(x,y)` estimates the cross power spectral density (CPSD) of two discrete-time signals, `x` and `y`, using Welch's averaged, modified periodogram method of spectral estimation.

- If `x` and `y` are both vectors, they must have the same length.
- If one of the signals is a matrix and the other is a vector, then the length of the vector must equal the number of rows in the matrix. The function expands the vector and returns a matrix of column-by-column cross power spectral density estimates.
- If `x` and `y` are matrices with the same number of rows but different numbers of columns, then `cpsd` returns a three-dimensional array, `pxy`, containing cross power spectral density estimates for all combinations of input columns. Each column of `pxy` corresponds to a column of `x`, and each page corresponds to a column of `y`: `pxy(:,m,n) = cpsd(x(:,m),y(:,n))`.
- If `x` and `y` are matrices of equal size, then `cpsd` operates column-wise: `pxy(:,n) = cpsd(x(:,n),y(:,n))`. To obtain a multi-input/multi-output array, append `'mimo'` to the argument list.

For real `x` and `y`, `cpsd` returns a one-sided CPSD. For complex `x` or `y`, `cpsd` returns a two-sided CPSD.

`pxy = cpsd(x,y,window)` uses `window` to divide `x` and `y` into segments and perform windowing.

`pxy = cpsd(x,y,window,noverlap)` uses `noverlap` samples of overlap between adjoining segments.

`pxy = cpsd(x,y,window,noverlap,nfft)` uses `nfft` sampling points to calculate the discrete Fourier transform.

`pxy = cpsd( ___ ,'mimo')` computes a multi-input/multi-output array of cross power spectral density estimates. This syntax can include any combination of input arguments from previous syntaxes.

`[pxy,w] = cpsd( ___ )` returns a vector of normalized frequencies, `w`, at which the cross power spectral density is estimated.

`[pxy,f] = cpsd( ___ ,fs)` returns a vector of frequencies, `f`, expressed in terms of the sample rate, `fs`, at which the cross power spectral density is estimated. `fs` must be the sixth numeric input to `cpsd`. To input a sample rate and still use the default values of the preceding optional arguments, specify these arguments as empty, `[]`.

`[pxy,w] = cpsd(x,y,window,noverlap,w)` returns the cross power spectral density estimates at the normalized frequencies specified in `w`.

`[pxy,f] = cpsd(x,y,window,noverlap,f,fs)` returns the cross power spectral density estimates at the frequencies specified in `f`.

`[ ___ ] = cpsd(x,y, ___ ,freqrange)` returns the cross power spectral density estimate over the frequency range specified by `freqrange`. Valid options for `freqrange` are `'onesided'`, `'twosided'`, and `'centered'`.

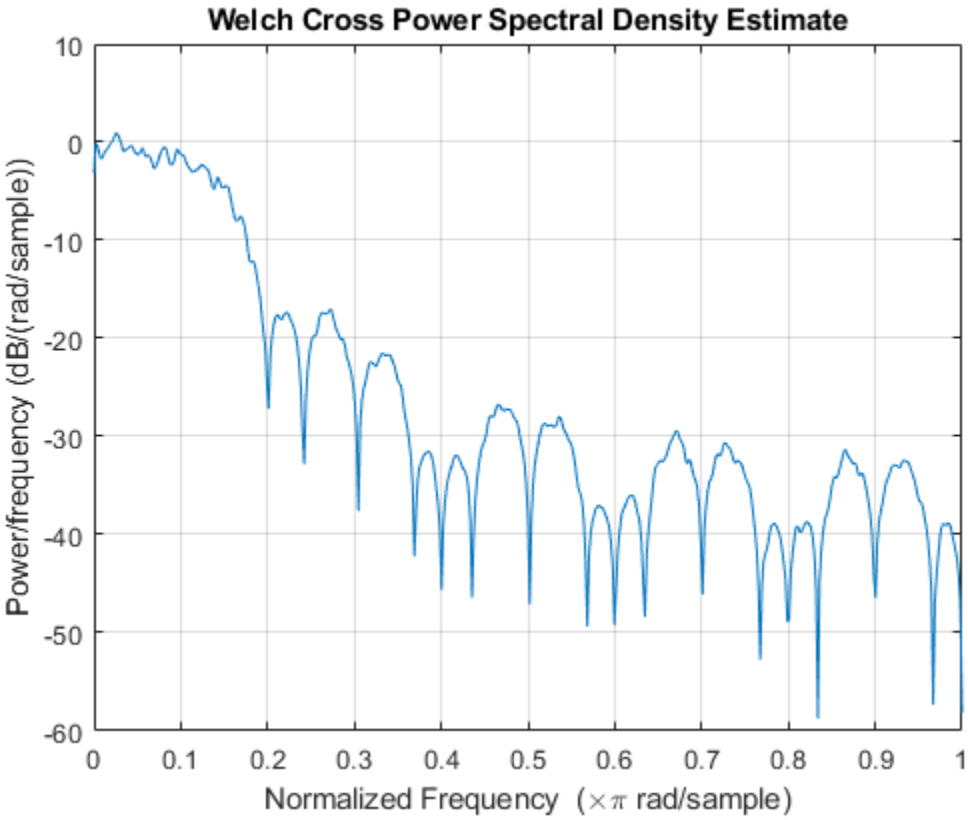
`cpsd( ___ )` with no output arguments plots the cross power spectral density estimate in the current figure window.

Examples

Cross Power Spectral Density of Colored Noise Signals

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

```
r = randn(16384,1);

hx = fir1(30,0.2,rectwin(31));
x = filter(hx,1,r);

hy = ones(1,10)/sqrt(10);
y = filter(hy,1,r);

cpsd(x,y,triang(500),250,1024)
```



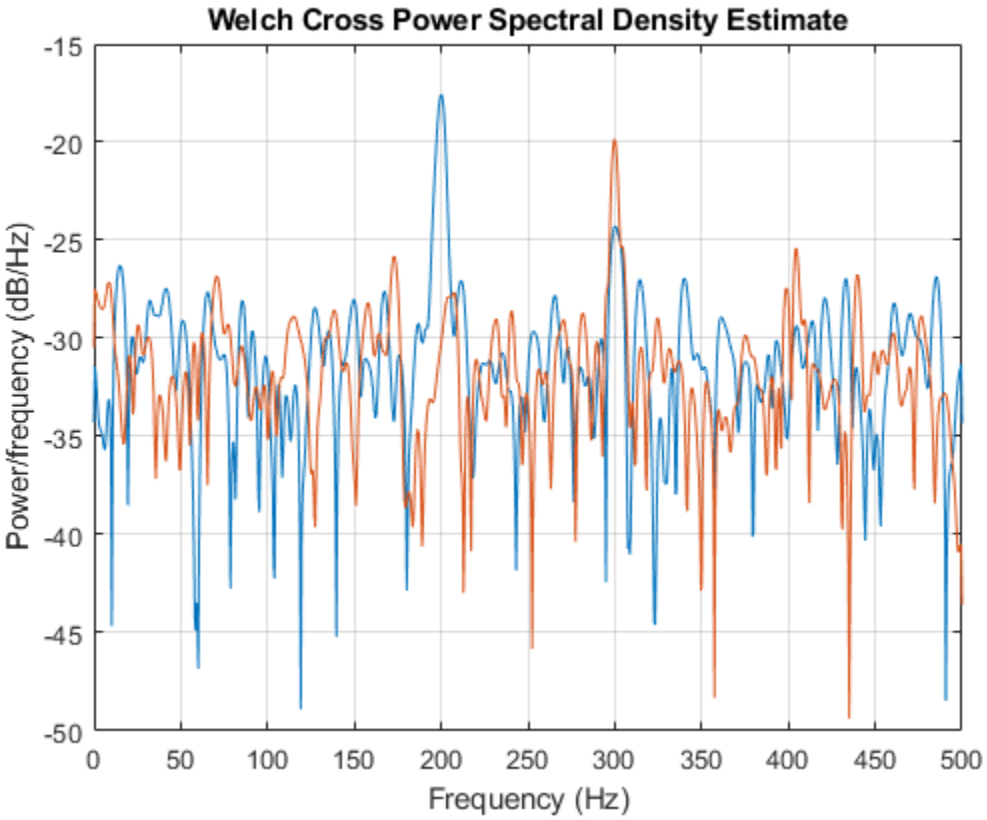
SISO and MIMO Cross Power Spectral Densities

Generate two two-channel sinusoids sampled at 1 kHz for 1 second. The channels of the first signal have frequencies of 200 Hz and 300 Hz. The channels of the second signal have frequencies of 300 Hz and 400 Hz. Both signals are embedded in unit-variance white Gaussian noise.

```
fs = 1e3;
t = (0:1/fs:1-1/fs)';

q = 2*sin(2*pi*[200 300].*t);
q = q+randn(size(q));

r = 2*sin(2*pi*[300 400].*t);
r = r+randn(size(r));
```

Compute the cross power spectral density of the two signals. Use a 256-sample Bartlett window to divide the signals into segments and window the segments. Specify 128 samples of overlap between adjoining segments and 2048 DFT points. Use the built-in functionality of `cpsd` to plot the result.

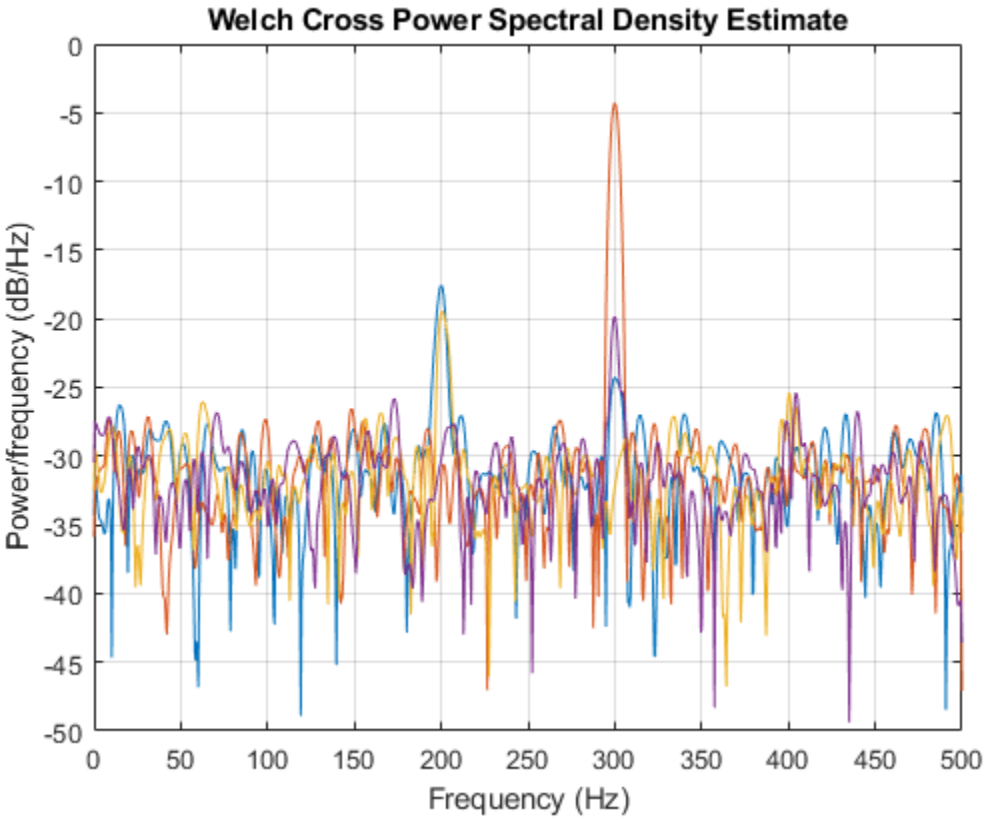
```
cpsd(q,r,bartlett(256),128,2048,fs)
```



By default, `cpsd` works column-by-column on matrix inputs of the same size. Each channel peaks at the frequencies of the original sinusoids.

Repeat the calculation, but now append `'mimo'` to the list of arguments.

```
cpsd(q,r,bartlett(256),128,2048,fs,'mimo')
```



When called with the `'mimo'` option, `cpsd` returns a three-dimensional array containing cross power spectral density estimates for all combinations of input columns. The estimate of the second channel of `q` and the first channel of `r` shows an enhanced peak at the common frequency of 300 Hz.

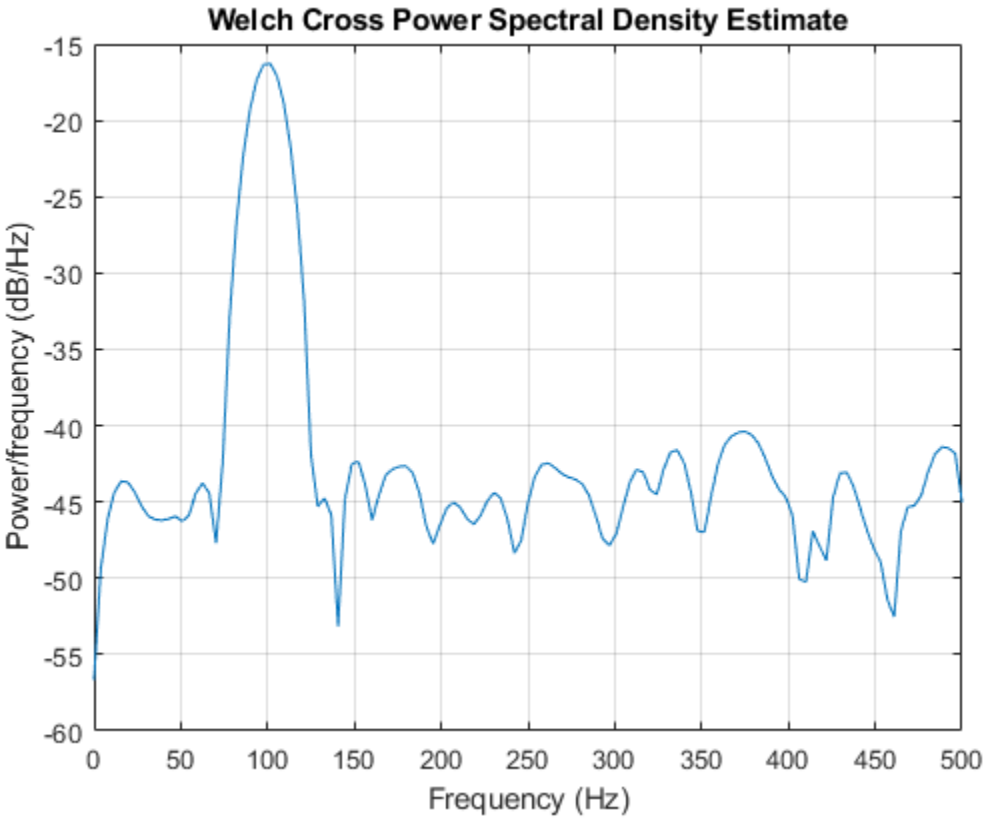
Cross Spectrum Phase of Lagged Sinusoids

Generate two 100 Hz sinusoidal signals sampled at 1 kHz for 296 ms. One of the sinusoids lags the other by 2.5 ms, equivalent to a phase lag of $\pi/2$. Both signals are embedded in white Gaussian noise of variance $1/4^2$.

```
Fs = 1000;
t = 0:1/Fs:0.296;

x = cos(2*pi*t*100)+0.25*randn(size(t));
tau = 1/400;
y = cos(2*pi*100*(t-tau))+0.25*randn(size(t));
```

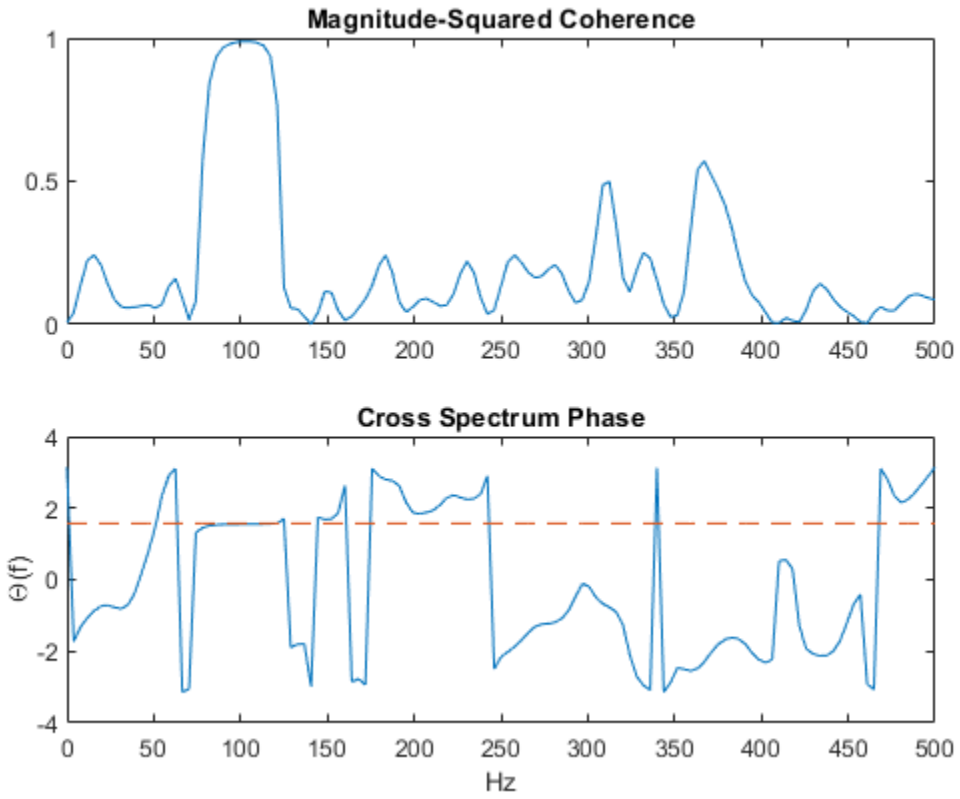
Compute and plot the magnitude of the cross power spectral density. Use the default settings for `cpsd`. The magnitude peaks at the frequency where there is significant coherence between the signals.

```
cpsd(x,y,[],[],[],Fs)
```



Plot magnitude-squared coherence function and the phase of the cross spectrum. The ordinate at the high-coherence frequency corresponds to the phase lag between the sinusoids.

```
[Cxy,F] = mscohere(x,y,[],[],[],Fs);

[Pxy,F] = cpsd(x,y,[],[],[],Fs);

subplot(2,1,1)
plot(F,Cxy)
title('Magnitude-Squared Coherence')

subplot(2,1,2)
plot(F,angle(Pxy))

hold on
plot(F,2*pi*100*tau*ones(size(F)),'--')
hold off

xlabel('Hz')
ylabel('\Theta(f)')
title('Cross Spectrum Phase')
```



Cross Power Spectral Density of Exponential Sequences

Generate two N-sample exponential sequences, $x_a = a^n$ and $x_b = b^n$, with $n \geq 0$. Specify $a = 0.8$, $b = 0.9$, and a small N to see finite-size effects.

```
N = 10;
n = 0:N-1;

a = 0.8;
b = 0.9;

xa = a.^n;
xb = b.^n;
```

Compute and plot the cross power spectral density of the sequences over the complete interval of normalized frequencies, $[-\pi, \pi]$. Specify a rectangular window of length N and no overlap between segments.

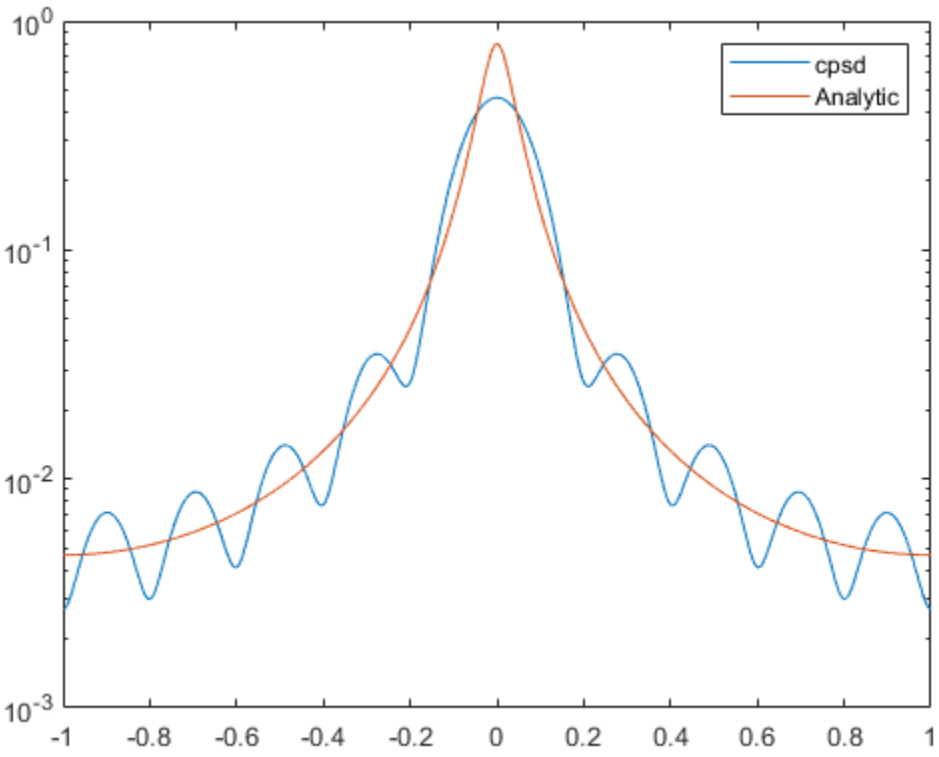
```
w = -pi:1/1000:pi;
wind = rectwin(N);
nove = 0;

[pxx,f] = cpsd(xa,xb,wind,nove,w);
```

The cross power spectrum of the two sequences has an analytic expression for large N:

$$R(\omega) = \frac{1}{1 - ae^{-j\omega}} \frac{1}{1 - be^{j\omega}}.$$

Convert this expression to a cross power spectral density by dividing it by $2\pi N$. Compare the results. The ripple in the `cpsd` result is a consequence of windowing.

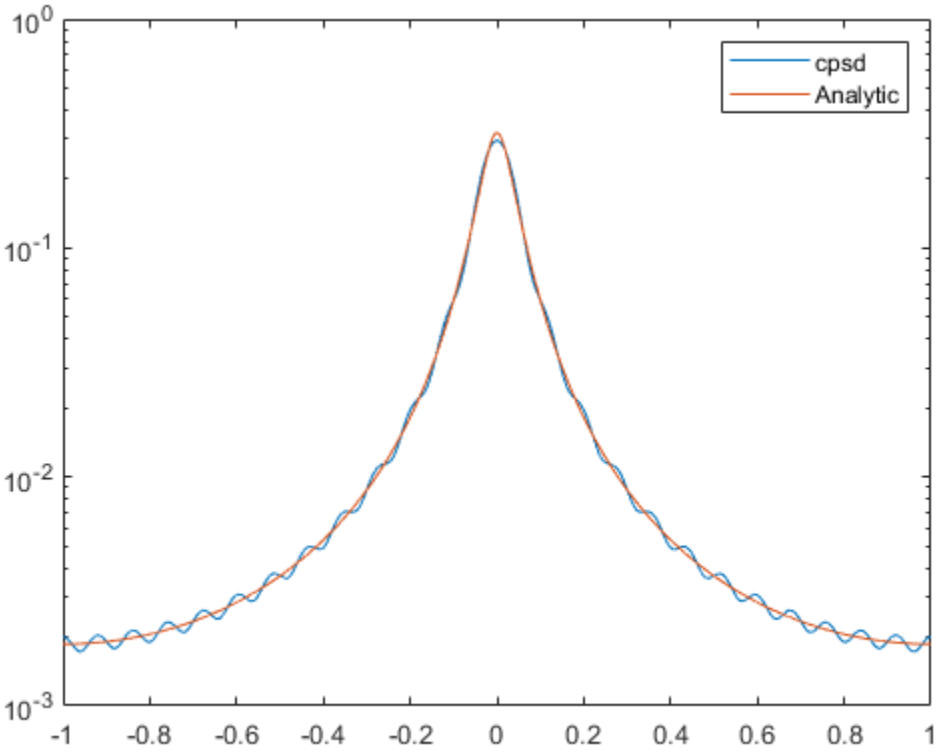
```
nfac = 2*pi*N;

X = 1./(1-a*exp(-1j*w));
Y = 1./(1-b*exp( 1j*w));
R = X.*Y/nfac;

semilogy(f/pi,abs(pxx))
hold on
semilogy(w/pi,abs(R))
hold off
legend('cpsd','Analytic')
```



Repeat the calculation with $N = 25$. The curves agree to six figures for N as small as 100.

```
N = 25;
n = 0:N-1;
xa = a.^n;
xb = b.^n;

wind = rectwin(N);
```

```
[pxx,f] = cpsd(xa,xb,wind,nove,w);
R = X.*Y/(2*pi*N);

semilogy(f/pi,abs(pxx))
hold on
semilogy(w/pi,abs(R))
hold off
legend('cpsd','Analytic')
```



Dial Tone Recognition

Use cross power spectral density to identify a highly corrupted tone.

The sound signals generated when you dial a number or symbol on a digital phone are sums of sinusoids with frequencies taken from two different groups. Each pair of tones contains one frequency of the low group (697 Hz, 770 Hz, 852 Hz, or 941 Hz) and one frequency of the high group (1209 Hz, 1336 Hz, or 1477 Hz).

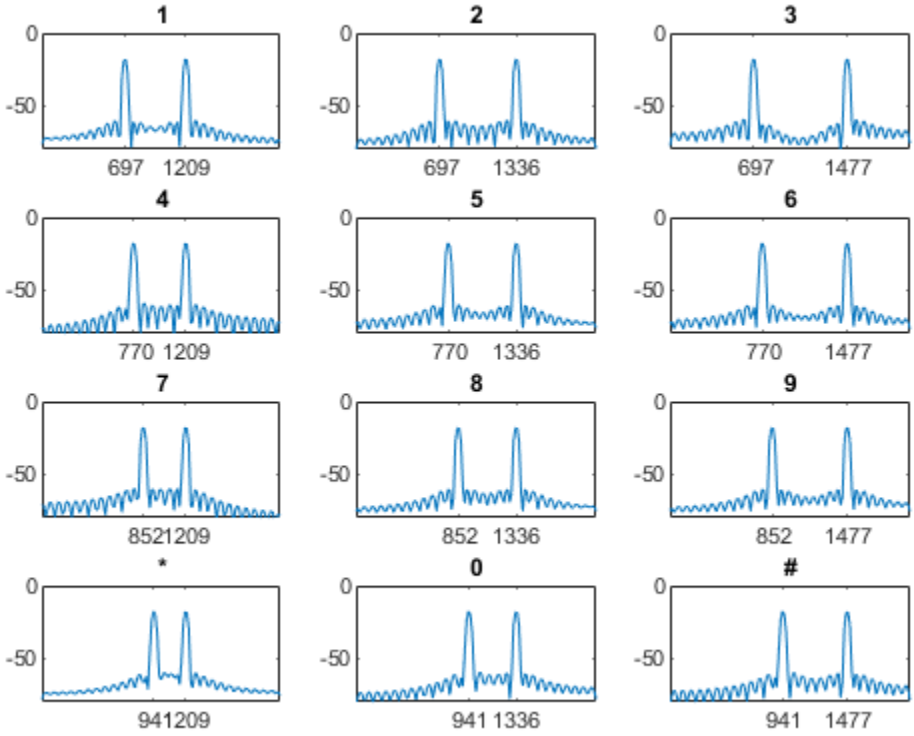
	1209 Hz	1336 Hz	1477 Hz
697 Hz	1	2	3
770 Hz	4	5	6
852 Hz	7	8	9
941 Hz	*	0	#

Generate signals corresponding to all the symbols. Sample each tone at 4 kHz for half a second. Prepare a reference table.

```
fs = 4e3;
t = 0:1/fs:0.5-1/fs;

nms = ['1';'2';'3';'4';'5';'6';'7';'8';'9';'*';'0';'#'];

ver = [697 770 852 941];
hor = [1209 1336 1477];

v = length(ver);
h = length(hor);

for k = 1:v
    for l = 1:h
        idx = h*(k-1)+l;
        tone = sum(sin(2*pi*[ver(k);hor(l)].*t))';
        tones(:,idx) = tone;
    end
end
```

Plot the Welch periodogram of each signal and annotate the component frequencies. Use a 200-sample Hamming window to divide the signals into non-overlapping segments and window the segments.

```
[pxx,f] = pwelch(tones,hamming(200),0,[],fs);

for k = 1:v
    for l = 1:h
        idx = h*(k-1)+l;
        ax = subplot(v,h,idx);
        plot(f,10*log10(pxx(:,idx)))
        ylim([-80 0])
        title(nms(idx))
        tx = [ver(k);hor(l)];
        ax.XTick = tx;
        ax.XTickLabel = int2str(tx);
    end
end
```

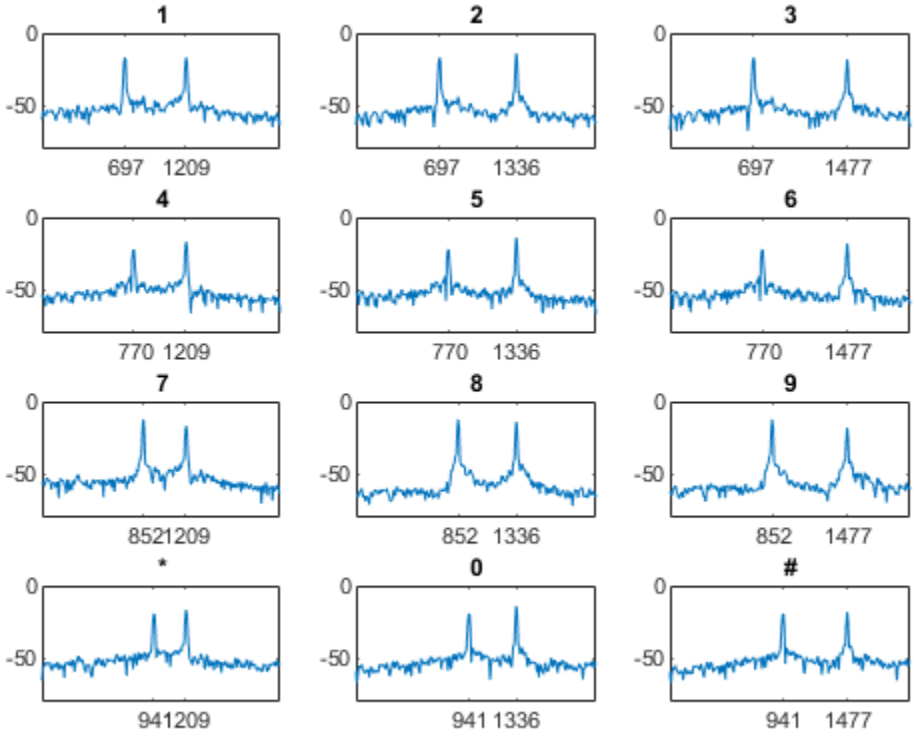


A signal produced by dialing the number 8 is sent through a noisy channel. The received signal is so corrupted that the number cannot be identified by inspection.

```
mys = sum(sin(2*pi*[ver(3);hor(2)].*t))'+5*randn(size(t'));

% To hear, type soundsc(mys,fs)
```

Compute the cross power spectral density of the corrupted signal and the reference signals. Window the signals using a 512-sample Kaiser window with shape factor $\beta = 5$. Plot the magnitude of each spectrum.

```
[pxy,f] = cpsd(mys,tones,kaiser(512,5),100,[],fs);

for k = 1:v
    for l = 1:h
        idx = h*(k-1)+l;
        ax = subplot(v,h,idx);
        plot(f,10*log10(abs(pxy(:,idx))))
        ylim([-80 0])
        title(nms(idx))
        tx = [ver(k);hor(l)];
        ax.XTick = tx;
        ax.XTickLabel = int2str(tx);
    end
end
```



The digit in the corrupted signal has the spectrum with the highest peaks and the highest RMS value.

```
[~,loc] = max(rms(abs(pxy)));

digit = nms(loc)
```

```
digit =
'8'
```

Input Arguments

x, y — Input signals

vectors | matrices

Input signals, specified as vectors or matrices.

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 segments.

- If `window` is an integer, then `cpsd` divides `x` and `y` into segments of length `window` and windows each segment with a Hamming window of that length.
- If `window` is a vector, then `cpsd` divides `x` and `y` into segments of the same length as the vector and windows each segment using `window`.

If the length of `x` and `y` cannot be divided exactly into an integer number of segments with `noverlap` overlapping samples, then the signals are truncated accordingly.

If you specify `window` as empty, then `cpsd` uses a Hamming window such that `x` and `y` are divided into eight segments 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 `cpsd` uses a number that produces 50% overlap between segments. If the segment length is unspecified, the function sets `noverlap` to $\lfloor N/4.5 \rfloor$, where N is the length of the input and output signals.

Data Types: `double` | `single`

`nfft` — Number of DFT points
positive integer | `[]`

Number of DFT points, specified as a positive integer. If you specify `nfft` as empty, then `cpsd` sets the parameter to $\max(256, 2^p)$, where $p = \lceil \log_2 N \rceil$ for input signals of length N.

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 has units of Hz.

`w` — Normalized frequencies
vector

Normalized frequencies, specified as a row or column vector with at least two elements. Normalized frequencies are in rad/sample.

Example: `w = [pi/4 pi/2]`

Data Types: `double`

`f` — Frequencies
vector

Frequencies, specified as a row or column vector with at least two elements. The frequencies are in cycles per unit time. The unit time is specified by the sample rate, `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 cross power spectral density estimate
`'onesided'` | `'twosided'` | `'centered'`

Frequency range for cross power spectral density estimate, specified as `'onesided'`, `'twosided'`, or `'centered'`. The default is `'onesided'` for real-valued signals and `'twosided'` for complex-valued signals.

- `'onesided'` — Returns the one-sided estimate of the cross power spectral density of two real-valued input signals, `x` and `y`. If `nfft` is even, `pxy` has `nfft`/2 + 1 rows and is computed over the interval [0,π] rad/sample. If `nfft` is odd, `pxy` has (`nfft` + 1)/2 rows and the interval is [0,π) rad/sample. If you specify `fs`, the corresponding intervals are [0,`fs`/2] cycles/unit time for even `nfft` and [0,`fs`/2) cycles/unit time for odd `nfft`.
- `'twosided'` — Returns the two-sided estimate of the cross power spectral density of two real-valued or complex-valued input signals, `x` and `y`. In this case, `pxy` has `nfft` rows and is computed over the interval [0,2π) rad/sample. If you specify `fs`, the interval is [0,`fs`) cycles/unit time.
- `'centered'` — Returns the centered two-sided estimate of the cross power spectral density of two real-valued or complex-valued input signals, `x` and `y`. In this case, `pxy` has `nfft` rows and is computed over the interval (–π,π] rad/sample for even `nfft` and (–π,π) rad/sample for odd `nfft`. If you specify `fs`, the corresponding intervals are (–`fs`/2, `fs`/2] cycles/unit time for even `nfft` and (–`fs`/2, `fs`/2) cycles/unit time for odd `nfft`.

Output Arguments

`pxy` — Cross power spectral density
vector | matrix | three-dimensional array

Cross power spectral density, returned as a vector, matrix, or three-dimensional array.

`w` — Normalized frequencies
vector

Normalized frequencies, returned as a real-valued column vector.

- If `pxy` is one-sided, `w` spans the interval [0,π] when `nfft` is even and [0,π) when `nfft` is odd.
- If `pxy` is two-sided, `w` spans the interval [0,2π).
- If `pxy` is DC-centered, `w` spans the interval (–π,π] when `nfft` is even and (–π,π) when `nfft` is odd.

Data Types: `double` | `single`

`f` — Frequencies
vector

Frequencies, returned as a real-valued column vector.

Data Types: `double` | `single`

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, $-\infty < n < \infty$, $-\infty < n < \infty$, and $E\{\cdot\}$ is the expected value operator.

Algorithms

`cpsd` uses Welch's averaged, modified periodogram method of spectral estimation.

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 B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975, pp. 414–419.

[3] 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

See Also

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

Topics

"Cross Spectrum and Magnitude-Squared Coherence"

Introduced before R2006a

cusum

Detect small changes in mean using cumulative sum

Syntax

```
[iupper,ilower] = cusum(x)

[iupper,ilower] = cusum(x,climit,mshift,tmean,tdev)

[iupper,ilower] = cusum( ___ ,'all')
[iupper,ilower,uppersum,lowersum] = cusum( ___ )

cusum( ___ )
```

Description

`[iupper,ilower] = cusum(x)` returns the first index of the upper and lower cumulative sums of `x` that have drifted beyond five standard deviations above and below a target mean. The minimum detectable mean shift is set to one standard deviation. The target mean and standard deviations are estimated from the first 25 samples of `x`.

`[iupper,ilower] = cusum(x,climit,mshift,tmean,tdev)` specifies `climit`, the number of standard deviations that the upper and lower cumulative sums are allowed to drift from the mean. It also specifies the minimum detectable mean shift, the target mean, and the target standard deviation.

`[iupper,ilower] = cusum( ___ ,'all')` returns all the indices at which the upper and lower cumulative sums exceed the control limit.

`[iupper,ilower,uppersum,lowersum] = cusum( ___ )` also returns the upper and lower cumulative sums.

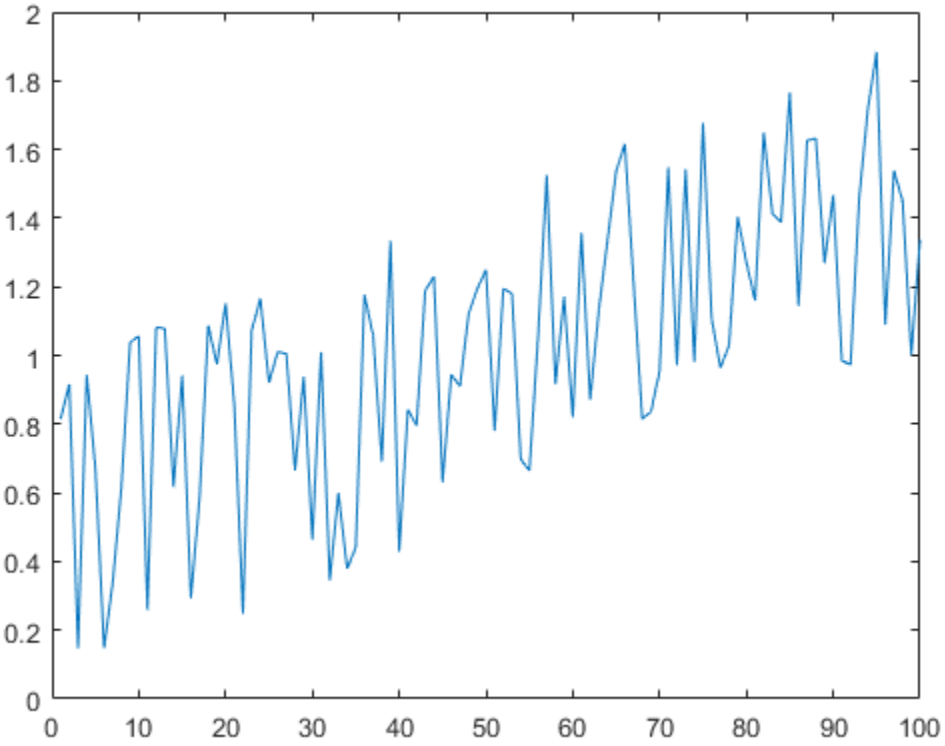
`cusum( ___ )` with no output arguments plots the upper and lower cumulative sums normalized to one standard deviation above and below the target mean.

Examples

`cusum` Default Values

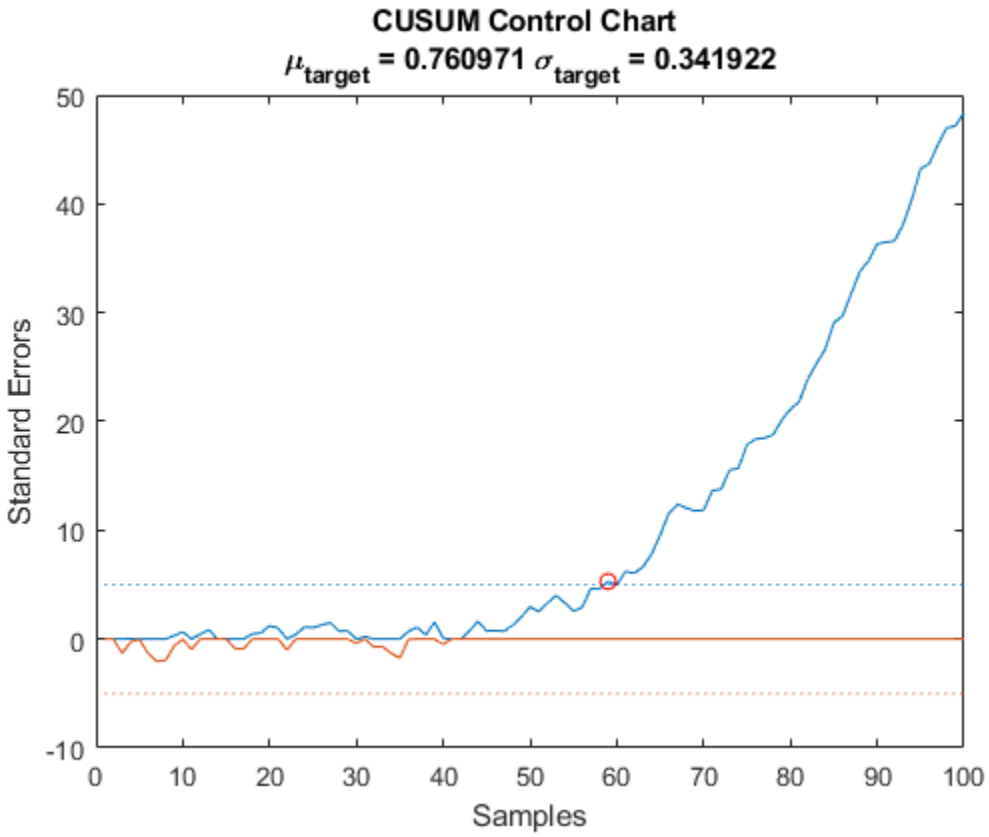
Generate and plot a 100-sample random signal with a linear trend. Reset the random number generator for reproducible results.

```
rng('default')

rnds = rand(1,100);
trnd = linspace(0,1,100);

fnc = rnds + trnd;

plot(fnc)
```



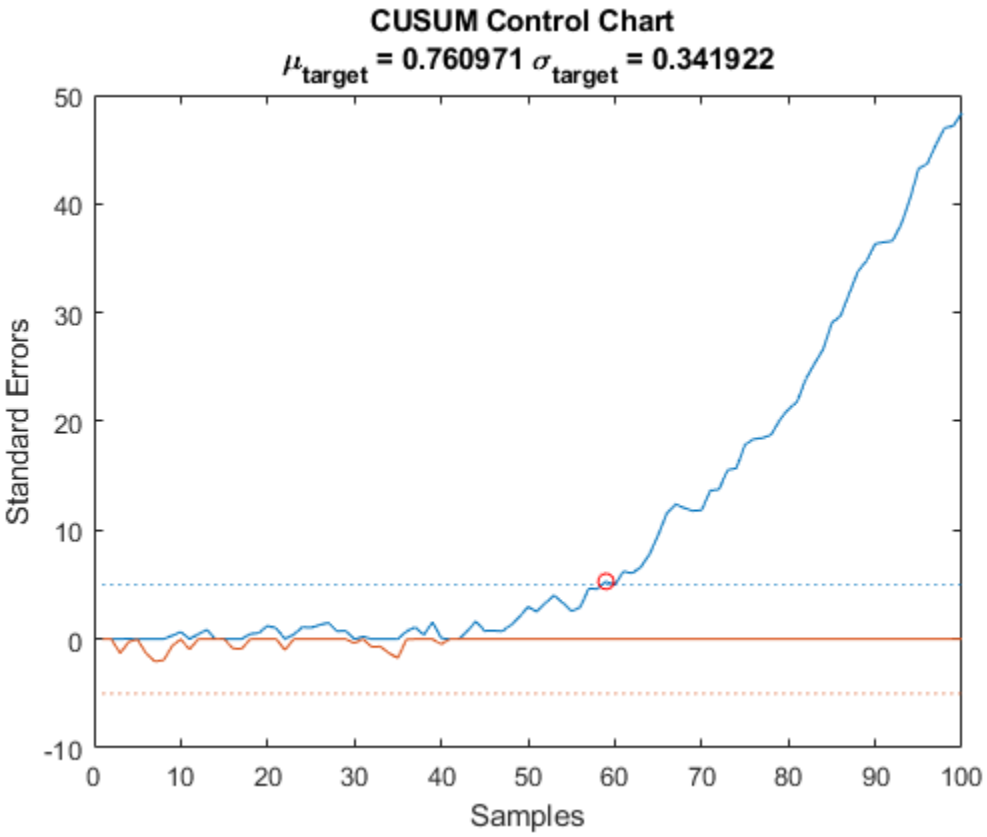
Apply `cusum` to the function using the default values of the input arguments.

```
cusum(fnc)
```



Compute the mean and standard deviation of the first 25 samples. Apply `cusum` using these numbers as the target mean and the target standard deviation. Highlight the point where the cumulative sum drifts more than five standard deviations beyond the target mean. Set the minimum detectable mean shift to one standard deviation.

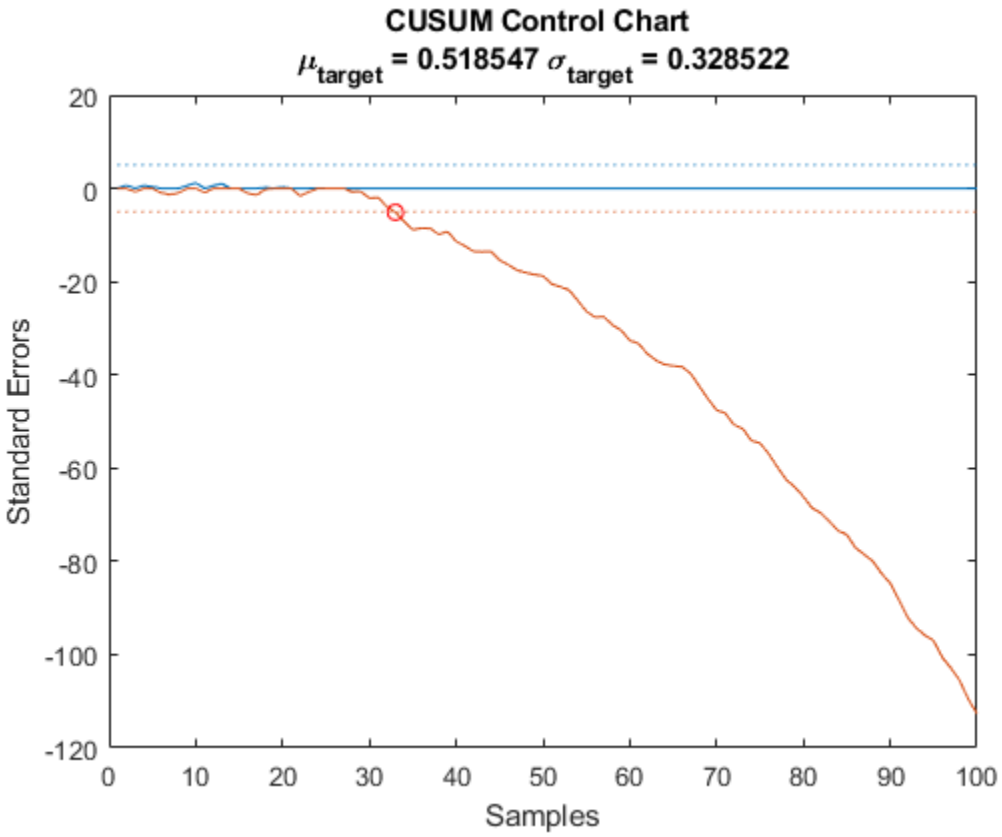
```
mfnc = mean(fnc(1:25));
sfnc = std(fnc(1:25));

cusum(fnc,5,1,mfnc,sfnc)
```



Repeat the calculation using a negative linear trend.

```
nnc = rnds - trnd;
```

```
cusum(nnc)
```



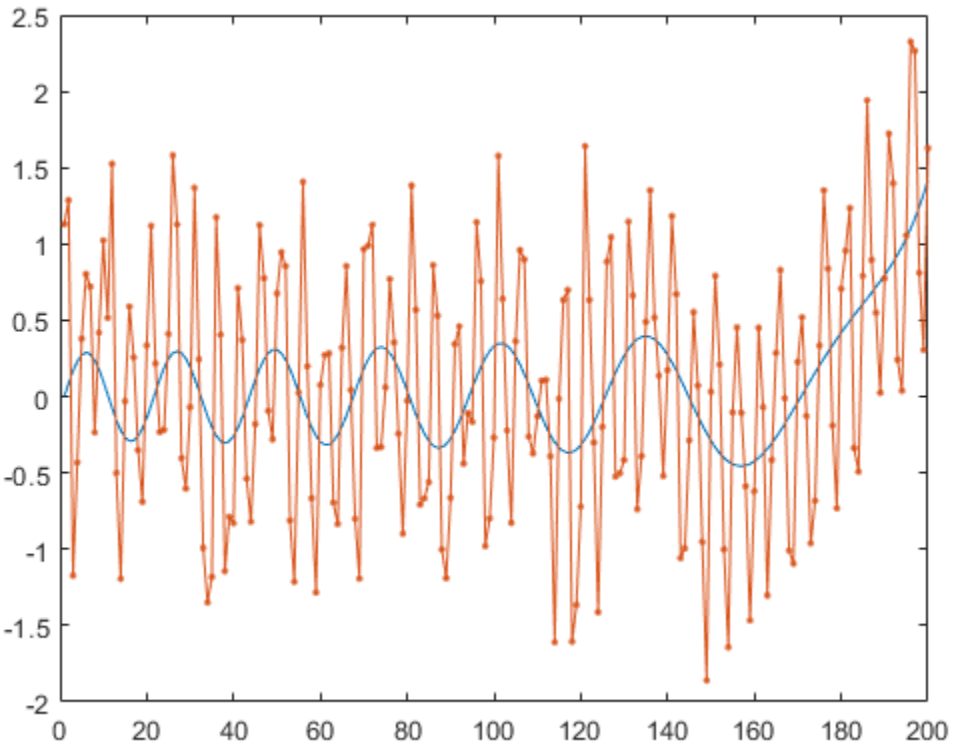
Unstable Motion Detection

Generate a signal resembling motion about an axle that becomes unstable due to wear. Add white Gaussian noise of variance 1/9. Reset the random number generator for reproducible results.

```
rng default

sz = 200;

dr = airy(2,linspace(-14.9371,1.2,sz));
rd = dr + sin(2*pi*(1:sz)/5) + randn(1,sz)/3;
```

Plot the growing background drift and the resulting signal.

```
plot(dr)
hold on
plot(rd,'.-')
hold off
```



Find the mean and standard deviation if the drift is not present and there is no noise. Plot the ideal noiseless signal and its stable background.

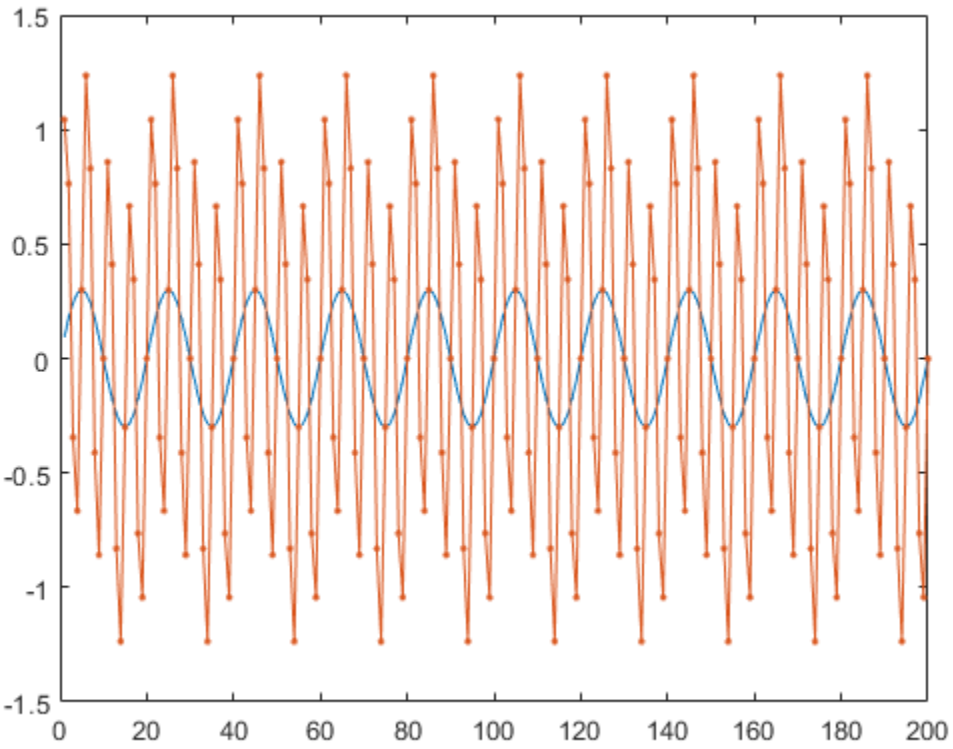
```
id = 0.3*sin(2*pi*(1:sz)/20);
st = id + sin(2*pi*(1:sz)/5);

mf = mean(st)
```

```
mf = -3.8212e-16
```

```
sf = std(st)
```

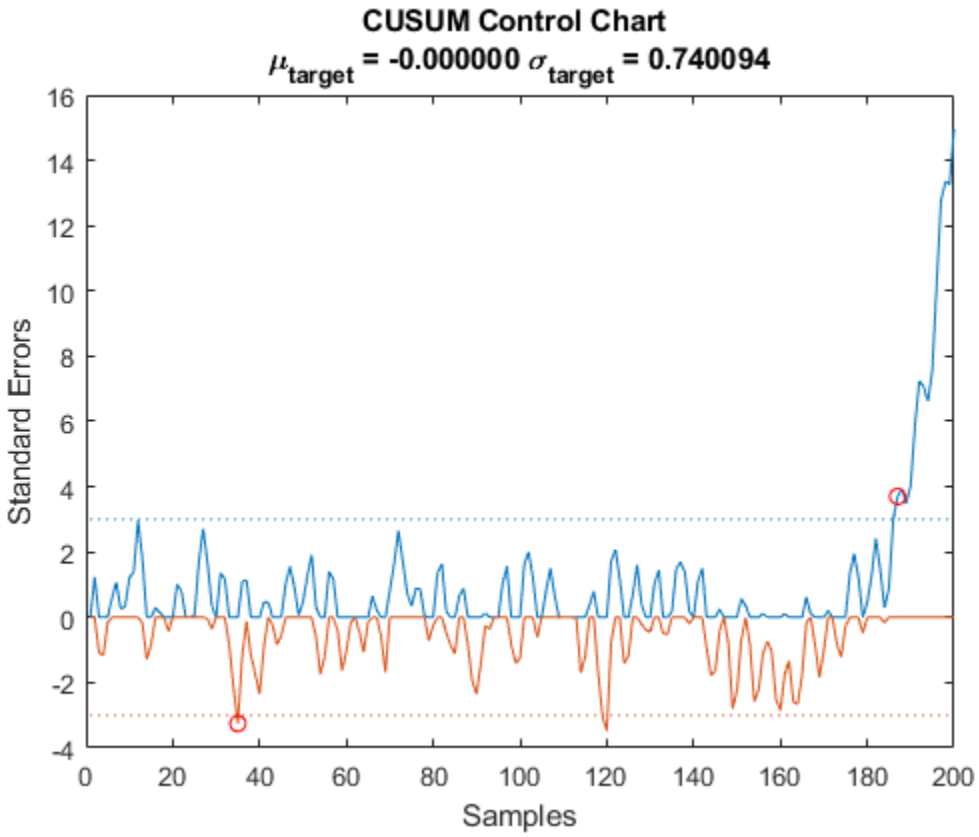
```
sf = 0.7401
```

```
plot(id)
hold on
plot(st,'.-')
hold off
```



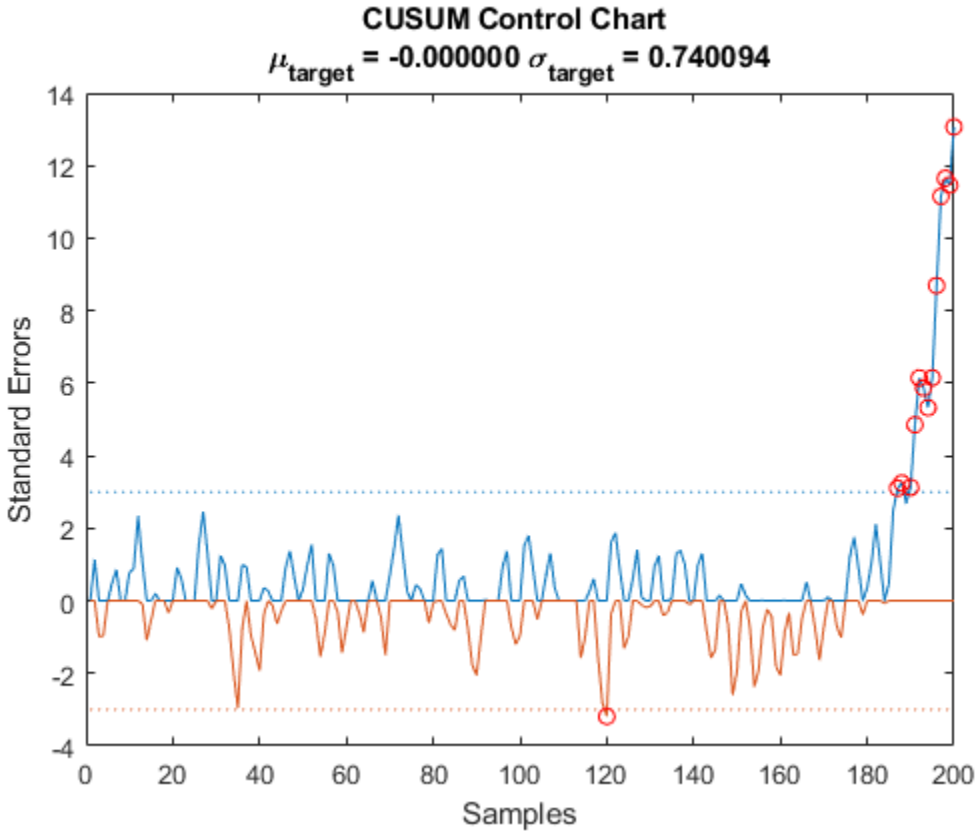
Use the CUSUM control chart to pinpoint the onset of instability. Assume that the system becomes unstable when the signal is three standard deviations beyond its ideal behavior. Specify a minimum detectable shift of one standard deviation.

```
cusum(rd,3,1,mf,sf)
```



Make the violation criterion more strict by increasing the minimum detectable shift. Return all instances of unwanted drift.

```
cusum(rd,3,1.2,mf,sf,'all')
```



Golf Scorecards

Every hole in golf has an associated "par" that indicates the expected number of strokes needed to sink the ball. Skilled players usually complete each hole with a number of strokes very close to par. It is necessary to play several holes and let scores accumulate before a clear winner emerges in a match.

Ben, Jen, and Ken play a full round, which consists of 18 holes. The course has an assortment of par-3, par-4, and par-5 holes. At the end of the game, the players tabulate their scores.

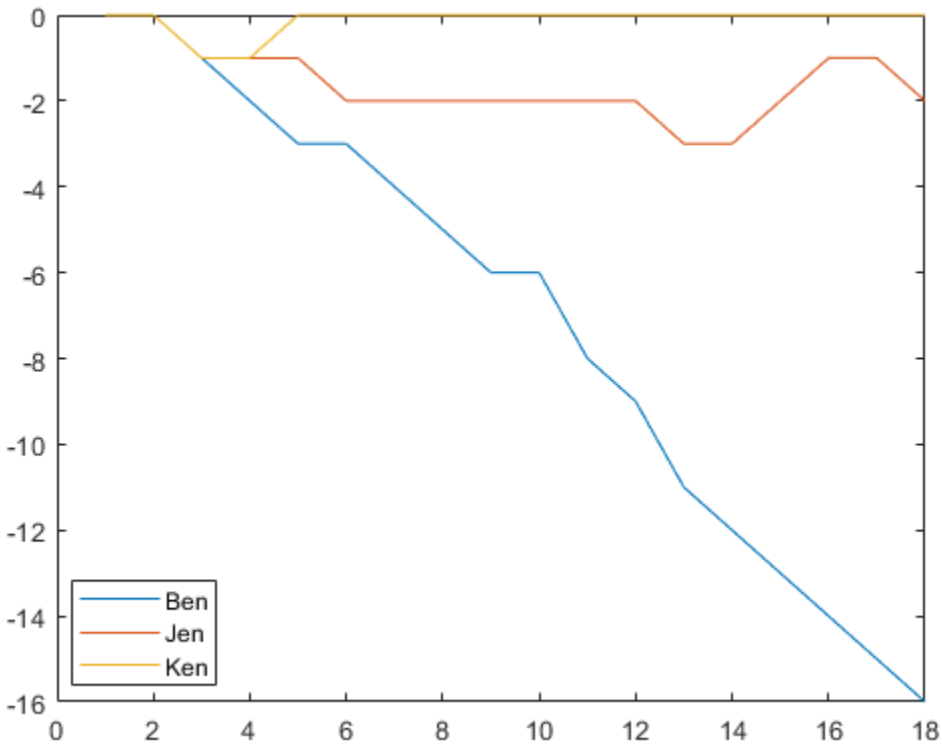
```
hole = 1:18;
par = [4 3 5 3 4 5 3 4 4 4 5 3 5 4 4 4 3 4];

nms = {'Ben';'Jen';'Ken'};

Ben = [4 3 4 2 3 5 2 3 3 4 3 2 3 3 3 3 2 3];
Jen = [4 3 4 3 4 4 3 4 4 4 5 3 4 4 5 5 3 3];
Ken = [4 3 4 3 5 5 4 4 4 4 5 3 5 4 5 4 3 5];

T = table(hole',par',Ben',Jen',Ken', ...
    'VariableNames',['hole';'par';nms])
```

```
T=18×5 table
    hole    par    Ben    Jen    Ken
```

1	4	4	4	4
2	3	3	3	3
3	5	4	4	4
4	3	2	3	3
5	4	3	4	5
6	5	5	4	5
7	3	2	3	4
8	4	3	4	4
9	4	3	4	4
10	4	4	4	4
11	5	3	5	5
12	3	2	3	3
13	5	3	4	5
14	4	3	4	4
15	4	3	5	5
16	4	3	5	4
⋮				

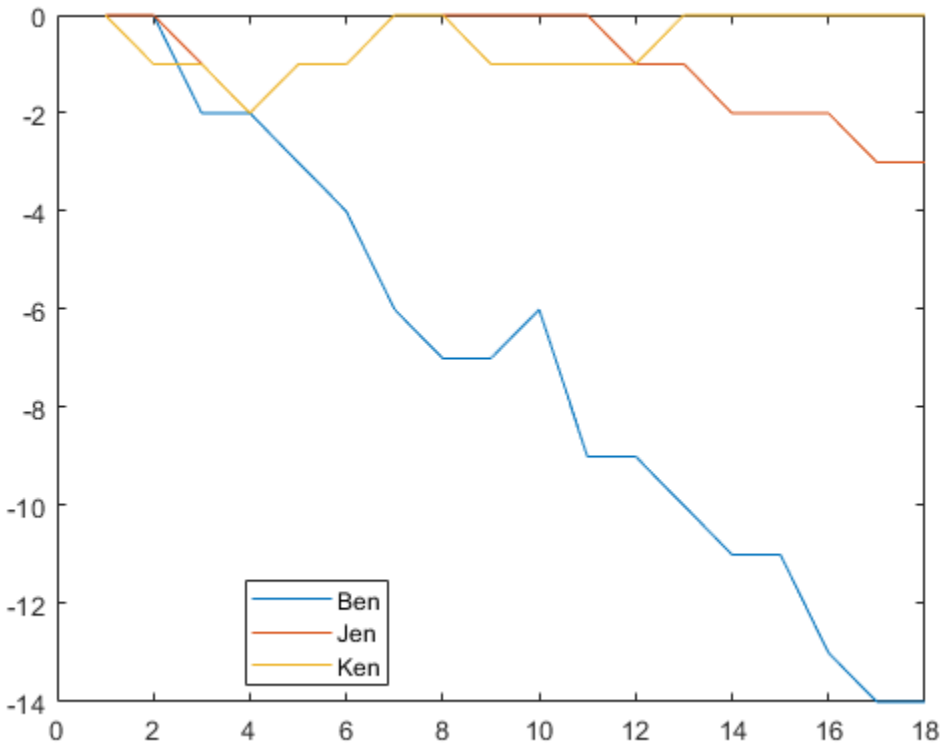
The winner of the round is the player whose lower cumulative sum drifts the most below par at the end. Compute the sums for the three players to determine the winner. Make every shift in mean detectable by setting a small threshold.

```
[~,b,~,Bensum] = cusum(Ben-par,1,1e-4,0);
[~,j,~,Jensum] = cusum(Jen-par,1,1e-4,0);
[~,k,~,Kensum] = cusum(Ken-par,1,1e-4,0);

plot([Bensum;Jensum;Kensum]')
legend(nms,'Location','best')
```



Ben wins the round. Simulate their next game by adding or subtracting a stroke per hole at random.

```
Ben = Ben+randi(3,1,18)-2;
Jen = Jen+randi(3,1,18)-2;
Ken = Ken+randi(3,1,18)-2;

[~,b,~,Bensum] = cusum(Ben-par,1,1e-4,0);
[~,j,~,Jensum] = cusum(Jen-par,1,1e-4,0);
[~,k,~,Kensum] = cusum(Ken-par,1,1e-4,0);

plot([Bensum;Jensum;Kensum]')
legend(nms,'Location','best')
```



Input Arguments

x — Input signal

vector

Input signal, specified as a vector.

Example: `reshape(rand(100,1)*[-1 1],1,200)`

climit — Control limit

5 (default) | real scalar

Control limit, specified as a real scalar expressed in standard deviations.

mshift — Minimum mean shift to detect

1 (default) | real scalar

Minimum mean shift to detect, specified as a real scalar expressed in standard deviations.

tmean — Target mean

`mean(x(1:25))` (default) | real scalar

Target mean, specified as a real scalar. If `tmean` is not specified, then it is estimated as the mean of the first 25 samples of `x`.

`tdev` — Target standard deviation
`std(x(1:25))` (default) | real scalar

Target standard deviation, specified as a real scalar. If `tdev` is not specified, then it is estimated as the standard deviation of the first 25 samples of `x`.

Output Arguments

`iupper,ilower` — Out-of-control point indices
integer scalars | integer vectors

Out-of-control point indices, returned as integer scalars or vectors. If all signal samples are within the specified tolerance, then `cusum` returns empty `iupper` and `ilower` arguments.

`uppersum,lowersum` — Upper and lower cumulative sums
vectors

Upper and lower cumulative sums, returned as vectors.

More About

CUSUM Control Chart

The CUSUM control chart is designed to detect small incremental changes in the mean of a process.

Given a sequence x_1, x_2, x_3, ..., x_n with estimated average m_x and estimated standard deviation σ_x, define upper and lower cumulative process sums using:

- Upper cumulative sum

$$U_i = \begin{cases} 0, & i = 1 \\ \max(0, U_{i-1} + x_i - m_x - \frac{1}{2}n\sigma_x), & i > 1 \end{cases}$$

- Lower sum

$$L_i = \begin{cases} 0, & i = 1 \\ \min(0, L_{i-1} + x_i - m_x + \frac{1}{2}n\sigma_x), & i > 1 \end{cases}$$

The variable n, represented in `cusum` by the `mshift` argument, is the number of standard deviations from the target mean, `tmean`, that make a shift detectable.

A process violates the CUSUM criterion at the sample x_j if it obeys $U_j > c\sigma_x$ or $L_j < -c\sigma_x$. The control limit c is represented in `cusum` by the `climit` argument.

By default, the function returns the first violation it detects. If you specify the `'all'` flag, the function returns every violation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If supplied, input argument `'all'` must be a compile-time constant.

See Also

`findchangepts` | `mean`

Introduced in R2016a

czt

Chirp Z-transform

Syntax

```
y = czt(x,m,w,a)
```

Description

`y = czt(x,m,w,a)` returns the length-`m` chirp Z-transform (CZT) of `x` along the spiral contour on the *z*-plane defined by `w` and `a` through `z = a*w.^-(0:m-1)`.

With the default values of `m`, `w`, and `a`, `czt` returns the Z-transform of `x` at `m` equally spaced points around the unit circle, a result equivalent to the discrete Fourier transform (DFT) of `x` as given by `fft(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);
```

Narrowband Section of 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);
```

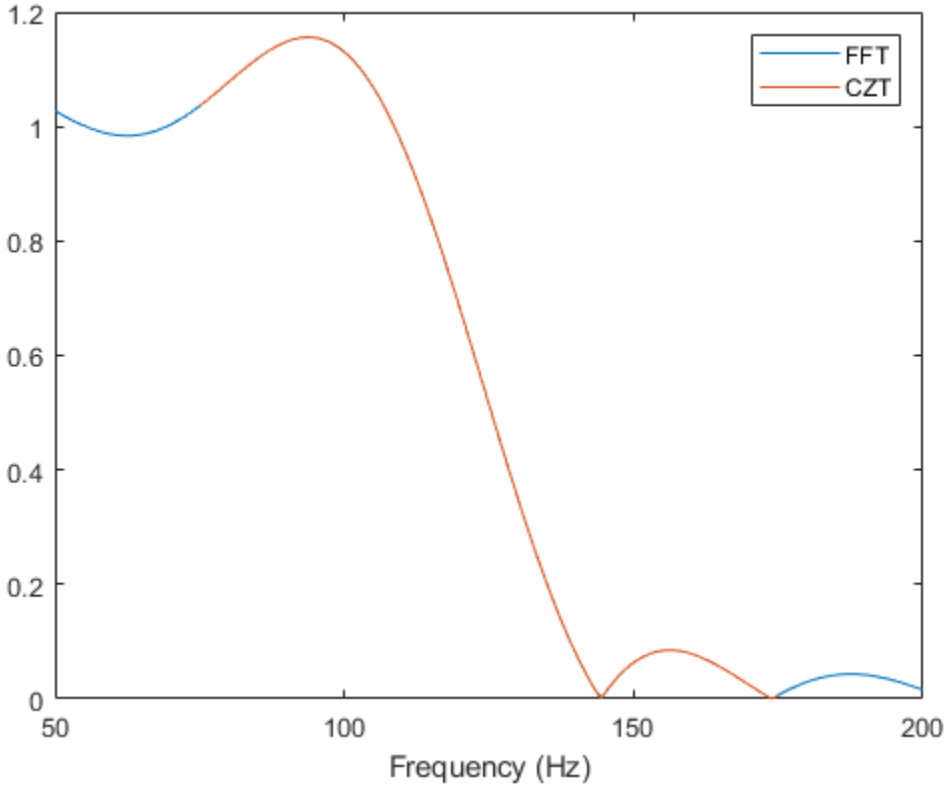
Compute the DFT and the CZT of the filter. Restrict the frequency range of the CZT to the band between 75 and 175 Hz. Generate 1024 samples in each case.

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

f1 = 75;
f2 = 175;
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;

plot(fy,abs(y),fz,abs(z))
xlim([50 200])
legend('FFT','CZT')
xlabel('Frequency (Hz)')
```



Input Arguments

x — Input signal

vector | matrix | 3-D array

Input signal, specified as a vector, a matrix, or a 3-D array. If `x` is a matrix, the function transforms the columns of `x`. If `x` is a 3-D array, the function operates along the first array dimension with size greater than 1.

Example: `sin(pi./[4;2]*(0:159))'` specifies a two-channel sinusoid.

Data Types: `single` | `double`

Complex Number Support: Yes

`m` — Transform length
`length(x)` (default) | positive integer scalar

Transform length, specified as a positive integer scalar.

Data Types: `single` | `double`

`w` — Ratio between spiral contour points
`exp(-2j*pi/m)` (default) | complex scalar

Ratio between spiral contour points, specified as a complex scalar.

Data Types: `single` | `double`
Complex Number Support: Yes

`a` — Spiral contour initial point
`1` (default) | complex scalar

Spiral contour initial point, specified as a complex scalar.

Example: `exp(1j*pi/4)` lies along the unit circle on the *z*-plane and makes an angle of 45 degrees with the real axis.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`y` — Chirp Z-transform
vector | matrix

Chirp Z-transform, returned as a vector or matrix.

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “Automatic dimension restriction” (MATLAB Coder).

- 3-D arrays are not supported for code generation.

Thread-Based Environment
Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

This function fully supports thread-based environments. For more information, see “Run MATLAB Functions in Thread-Based Environment”.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
`fft` | `freqz`

Topics
“Chirp Z-Transform”

Introduced before R2006a

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 `x` to decibels (dB). This syntax assumes that `x` contains voltage measurements across a resistance of 1 Ω.

`dboutput = db(x,SignalType)` specifies the signal type represented by the elements of `x` as either `'voltage'` or `'power'`.

`dboutput = db(x,R)` specifies the resistance, `R`, for voltage measurements.

`dboutput = db(x,'voltage',R)` is equivalent to `db(x,R)`.

Examples

Decibels from Voltage and Power

Express a unit voltage in 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 = 1×2

   -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 = 10×2

   -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
```

Input Arguments

x — Signal measurements

scalar | vector | matrix | *N*-D array

Signal measurements, specified as a scalar, vector, matrix, or *N*-D array.

Data Types: `single` | `double`
Complex Number Support: Yes

SignalType — Type of signal measurements

`'voltage'` (default) | `'power'`

Type of signal measurements, specified as either `'voltage'` or `'power'`. If you specify `SignalType` as `'power'`, then all elements of `x` must be nonnegative.

R — Resistive load

1 Ω (default) | positive scalar

Resistive load, specified as a positive scalar expressed in ohms. This argument is ignored if you specify `SignalType` as `'power'`.

Data Types: `single` | `double`

Output Arguments

dboutput — Energy or power measurements in decibels

scalar | vector | matrix | *N*-D array

Energy or power measurements in decibels, returned as an array with the same dimensions as `x`.

- If `x` contains voltage measurements, then `dboutput` is $10\log_{10}\left(|x|^2/R\right)$.
- If the input `x` contains power measurements, then `dboutput` is $10\log_{10}x$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

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

Introduced in R2011b

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 =
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 =
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`

Introduced in R2008a

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 =
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 =
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`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

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

Introduced in R2007b

dct

Discrete cosine transform

Syntax

```
y = dct(x)
y = dct(x,n)

y = dct(x,n,dim)

y = dct(___,'Type',dcttype)
```

Description

`y = dct(x)` returns the unitary discrete cosine transform of input array `x`. The output `y` has the same size as `x`. If `x` has more than one dimension, then `dct` operates along the first array dimension with size greater than 1.

`y = dct(x,n)` zero-pads or truncates the relevant dimension of `x` to length `n` before transforming.

`y = dct(x,n,dim)` computes the transform along dimension `dim`. To input a dimension and use the default value of `n`, specify the second argument as empty, `[]`.

`y = dct(___,'Type',dcttype)` specifies the type of discrete cosine transform to compute. See “Discrete Cosine Transform” on page 1-256 for details. This option can be combined with any of the previous syntaxes.

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 it to the original signal.

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

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

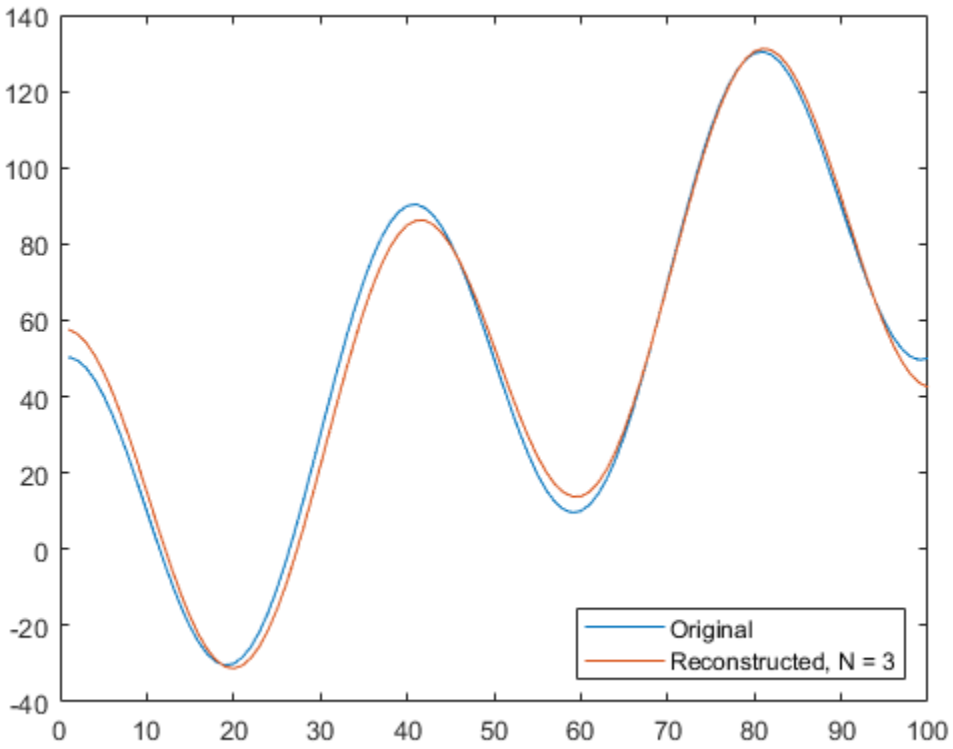


Image Data Compression

Load a file that contains depth measurements of a mold used to mint a United States penny. The data, taken at the National Institute of Standards and Technology, are sampled on a 128-by-128 grid. Display the data.

```
load penny

surf(P)
view(2)
colormap copper
shading interp
axis ij square off
```

Compute the discrete cosine transform of the image data. Operate first along the rows and then along the columns.

```
Q = dct(P,[],1);
R = dct(Q,[],2);
```

Find what fraction of DCT coefficients contain 99.98% of the energy in the image.

```
X = R(:);

[~,ind] = sort(abs(X),'descend');
coeffs = 1;
while norm(X(ind(1:coeffs)))/norm(X) < 0.9998
   coeffs = coeffs + 1;
end
fprintf('%d of %d coefficients are sufficient\n',coeffs,numel(R))
```

```
3572 of 16384 coefficients are sufficient
```

Reconstruct the image using only the necessary coefficients.

```
R(abs(R) < abs(X(ind(coeffs)))) = 0;

S = idct(R,[],2);
T = idct(S,[],1);
```

Display the reconstructed image.

```
surf(T)
view(2)
shading interp
axis ij square off
```

Image Resizing

Load a file that contains depth measurements of a mold used to mint a United States penny. The data, taken at the National Institute of Standards and Technology, are sampled on a 128-by-128 grid. Display the data.

```
load penny

surf(P)
view(2)
colormap copper
shading interp
axis ij square off
```

Compute the discrete cosine transform of the image data using the DCT-1 variant. Operate first along the rows and then along the columns.

```
Q = dct(P,[],1,'Type',1);
R = dct(Q,[],2,'Type',1);
```

Invert the transform. Truncate the inverse so that each dimension of the reconstructed image is one-half the length of the original.

```
S = idct(R,size(P,2)/2,2,'Type',1);
T = idct(S,size(P,1)/2,1,'Type',1);
```

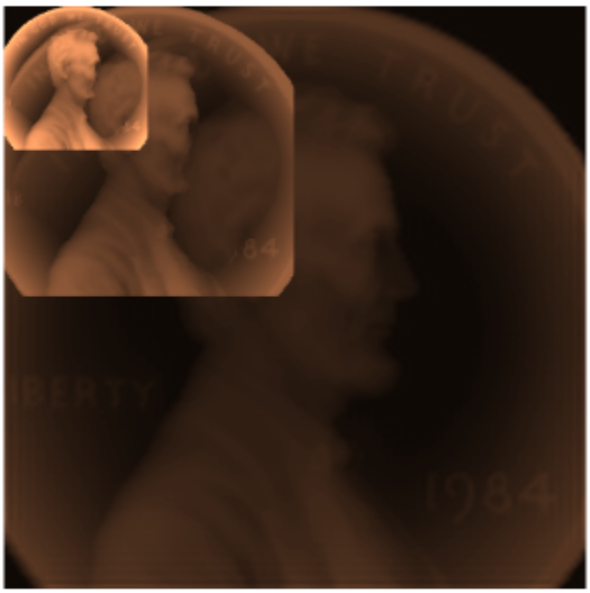
Invert the transform again. Zero-pad the inverse so that each dimension of the reconstructed image is twice the length of the original.

```
U = idct(R,size(P,2)*2,2,'Type',1);
V = idct(U,size(P,1)*2,1,'Type',1);
```

Display the original and reconstructed images.

```
surf(V)
view(2)
shading interp
hold on

surf(P)
view(2)
shading interp
```

```
surf(T)
view(2)
shading interp
hold off
axis ij equal off
```



Input Arguments

x — Input array

vector | matrix | *N*-D array | `gpuArray` object

Input array, specified as a real-valued or complex-valued vector, matrix, *N*-D array, or `gpuArray` object.

See “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox) and “GPU Support by Release” (Parallel Computing Toolbox) for details on `gpuArray` objects.

Example: `sin(2*pi*(0:255)/4)` specifies a sinusoid as a row vector.

Example: `sin(2*pi*[0.1;0.3]*(0:39))'` specifies a two-channel sinusoid.

Data Types: `single` | `double`
Complex Number Support: Yes

n — Transform length

positive integer scalar

Transform length, specified as a positive integer scalar.

Data Types: `single` | `double`

`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar.

Data Types: `single` | `double`

`dcttype` — Discrete cosine transform type
2 (default) | 1 | 3 | 4

Discrete cosine transform type, specified as a positive integer scalar from 1 to 4. See “Discrete Cosine Transform” on page 1-256 for the definitions of the different types of DCT.

Data Types: `single` | `double`

Output Arguments

y — Discrete cosine transform
vector | matrix | *N*-D array | `gpuArray` object

Discrete cosine transform, returned as a real-valued or complex-valued vector, matrix, *N*-D array, or `gpuArray` object.

More About

Discrete Cosine Transform

The discrete cosine transform (DCT) is closely related to the discrete Fourier transform. You can often reconstruct a sequence very accurately from only a few DCT coefficients. This property is useful for applications requiring data reduction.

The DCT has four standard variants. For a signal x of length N, and with $\delta_{k\ell}$ the Kronecker delta, the transforms are defined by:

- DCT-1:

$$y(k) = \sqrt{\frac{2}{N-1}} \sum_{n=1}^{N} x(n) \frac{1}{\sqrt{1+\delta_{n1}+\delta_{nN}}} \frac{1}{\sqrt{1+\delta_{k1}+\delta_{kN}}} \cos\left(\frac{\pi}{N-1}(n-1)(k-1)\right)$$

- DCT-2:

$$y(k) = \sqrt{\frac{2}{N}} \sum_{n=1}^{N} x(n) \frac{1}{\sqrt{1+\delta_{k1}}} \cos\left(\frac{\pi}{2N}(2n-1)(k-1)\right)$$

- DCT-3:

$$y(k) = \sqrt{\frac{2}{N}} \sum_{n=1}^{N} x(n) \frac{1}{\sqrt{1+\delta_{n1}}} \cos\left(\frac{\pi}{2N}(n-1)(2k-1)\right)$$

- DCT-4:

$$y(k) = \sqrt{\frac{2}{N}} \sum_{n=1}^{N} x(n)\cos\left(\frac{\pi}{4N}(2n-1)(2k-1)\right)$$

The series are 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.

All variants of the DCT are *unitary* (or, equivalently, *orthogonal*): To find their inverses, switch k and n in each definition. DCT-1 and DCT-4 are their own inverses. DCT-2 and DCT-3 are inverses of each other.

References

[1] Jain, A. K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

[2] Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

[3] Pennebaker, W. B., and J. L. Mitchell. *JPEG Still Image Data Compression Standard*. New York: Van Nostrand Reinhold, 1993.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- C and C++ code generation for `dct` requires DSP System Toolbox™ software.
- The length of the transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.
- Inputs must be double precision.
- Only DCT-2 is allowed.

Thread-Based Environment

Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

This function fully supports thread-based environments. For more information, see “Run MATLAB Functions in Thread-Based Environment”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- *N*-D input arrays are not supported.
- The `dim` and `dcttype` input arguments are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

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

Topics

“DCT for Speech Signal Compression”

Introduced before R2006a

decimate

Decimation — decrease sample rate by integer factor

Syntax

```
y = decimate(x,r)
y = decimate(x,r,n)
y = decimate(x,r,'fir')
y = decimate(x,r,n,'fir')
```

Description

`y = decimate(x,r)` reduces the sample 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 infinite impulse response (IIR) filter of order 8.

`y = decimate(x,r,n)` uses a Chebyshev filter of order `n`.

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

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

Examples

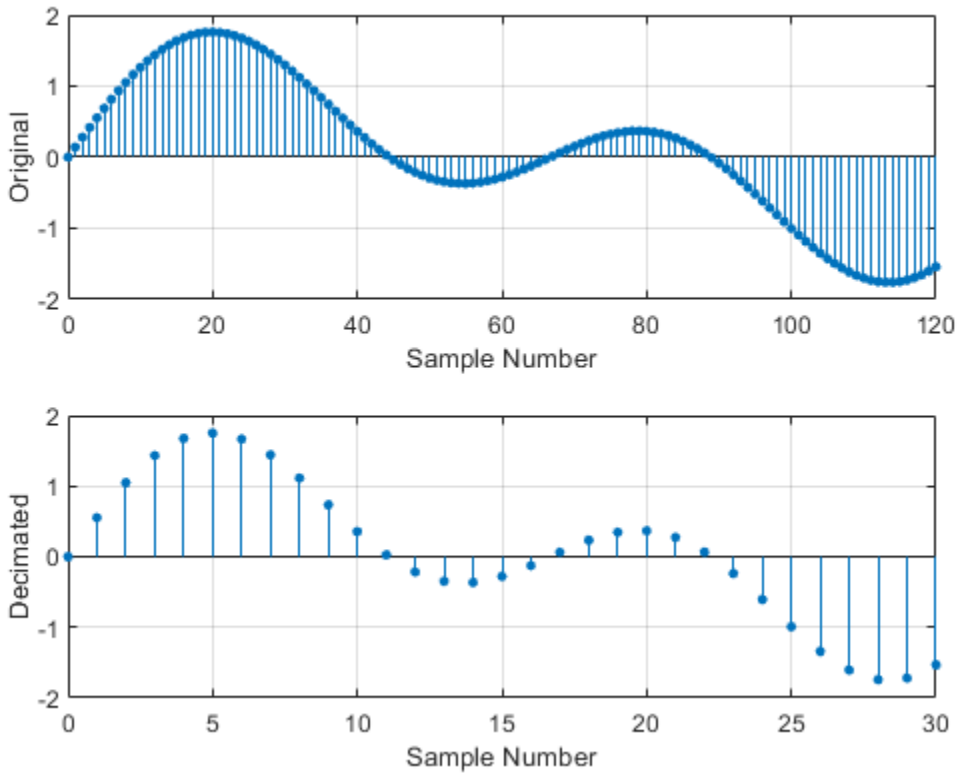
Decimate Signal

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

```
t = 0:1/4e3:1;
x = sin(2*pi*30*t) + sin(2*pi*60*t);
y = decimate(x,4);
```

Plot the original and decimated signals.

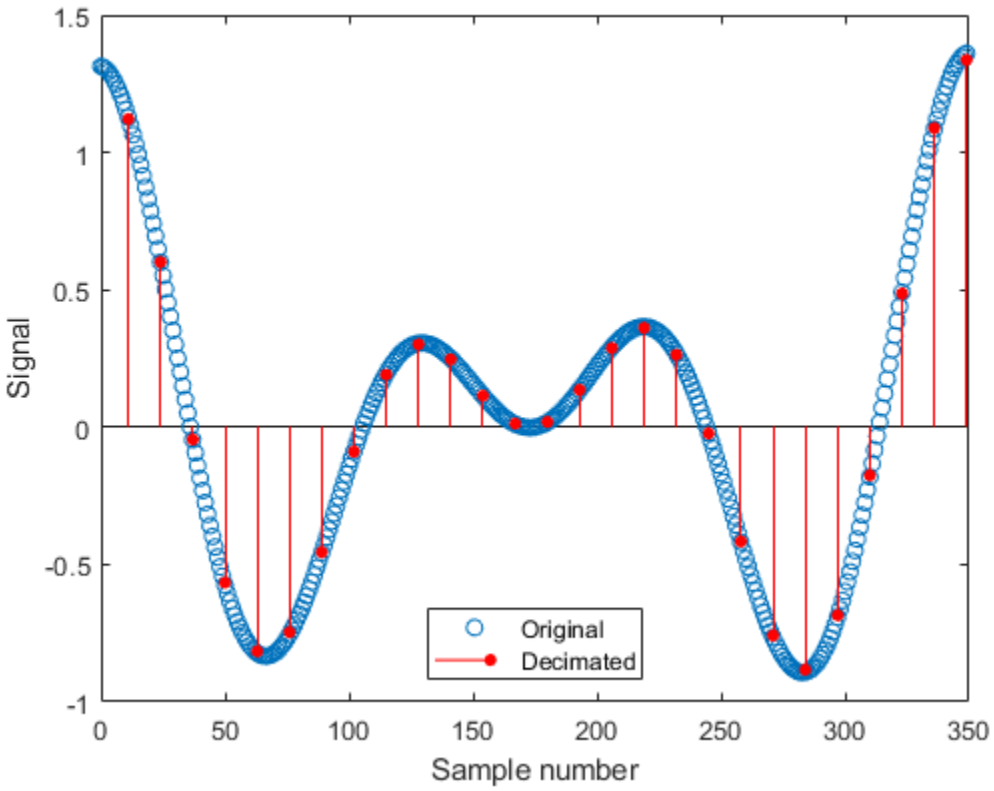
```
subplot(2,1,1)
stem(0:120,x(1:121),'filled','MarkerSize',3)
grid on
xlabel('Sample Number')
ylabel('Original')

subplot(2,1,2)
stem(0:30,y(1:31),'filled','MarkerSize',3)
grid on
xlabel('Sample Number')
ylabel('Decimated')
```



Decimate Signal Using 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,fliplr(y),'ro','filled','markersize',4)

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

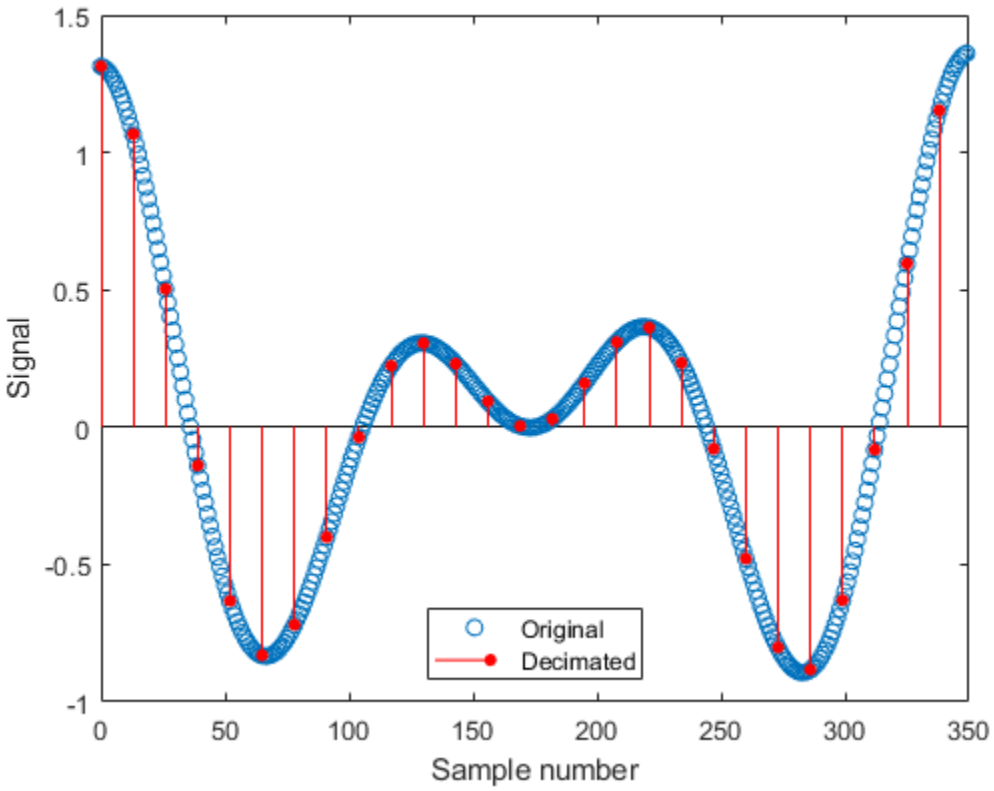


The original and decimated signals have matching *last* elements.

Decimate Signal Using 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.

Input Arguments

x — Input signal

vector

Input signal, specified as a vector.

Data Types: `double`

r — Decimation factor

positive integer

Decimation factor, specified as a positive integer. For better results when `r` is greater than 13, divide `r` into smaller factors and call `decimate` several times.

Data Types: `double`

n — Filter order

positive integer

Filter order, specified as a positive integer. IIR filter orders above 13 are not recommended because of numerical instability. The function displays a warning in those cases.

Data Types: `double`

Output Arguments

y — Decimated signal

vector

Decimated signal, returned as a vector.

Data Types: `double`

Algorithms

Decimation reduces the original sample 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. The function 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 a normalized cutoff frequency of `0.8/r` and a passband ripple of 0.05 dB. Sometimes, the specified filter order produces passband distortion due to round-off errors accumulated from the convolutions needed to create the transfer function. `decimate` automatically reduces the filter order 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 the forward and reverse directions using `filtfilt` to remove phase distortion. In effect, this process 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. In the resampled sequence (`y`), `y(end)` matches `x(end)` when the IIR filter is used, and `y(1)` matches `x(1)` when the FIR filter is used.

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.

See Also

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

Introduced before R2006a

demod

Demodulation for communications simulation

Syntax

```
x = demod(y,fc,fs,method)
x = demod(y,fc,fs,method,opt)
```

Description

`x = demod(y,fc,fs,method)` demodulates the real carrier signal `y` with a carrier frequency `fc` and sample rate `fs` using the method specified in `method`.

`x = demod(y,fc,fs,method,opt)` demodulates the real carrier signal `y` using the additional options specified in `opt`.

Examples

Frequency Modulation and Demodulation

Generate a 150 Hz sinusoid sampled at 8 kHz for 1 second. Embed the modulated signal in white Gaussian noise of variance 0.1^2.

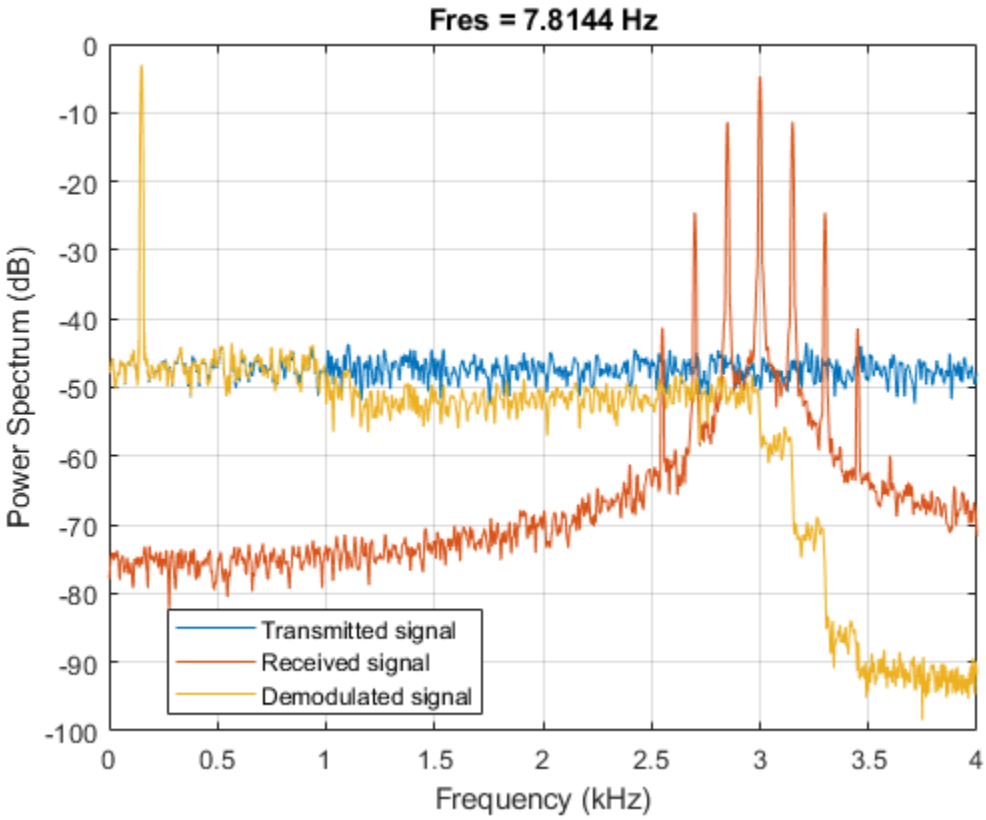
```
fs = 8e3;

t = 0:1/fs:1-1/fs;
s = cos(2*pi*150*t) + randn(size(t))/10;
```

Frequency modulate the signal at a carrier frequency of 3 kHz using a modulation constant of 0.1.

```
fc = 3e3;
rx = modulate(s,fc,fs,'fm',0.1);
```

Frequency demodulate the signal using the same carrier frequency and modulation constant. Compute and plot power spectrum estimates for the transmitted, received, and demodulated signals.

```
y = demod(rx,fc,fs,'fm',0.1);

pspectrum([s;rx;y]',fs,'Leakage',0.85)

legend('Transmitted signal','Received signal','Demodulated signal','Location','best')
```



Input Arguments

y — Modulated signal

real vector | real matrix

Modulated message signal, specified as a real vector or matrix. Except for the methods `pwm` and `ppm`, `y` is the same size as `x`.

fc — Carrier frequency

real positive scalar

Carrier frequency used to modulate the message signal, specified as a real positive scalar.

fs — Sample rate

real positive scalar

Sample rate, specified as a real positive scalar.

method — Method of modulation used

`'am'` (default) | `'amdsb-tc'` | `'amssb'` | `'fm'` | `'pm'` | `'pwm'` | `'ppm'` | `'qam'`

Method of modulation used, specified as one of:

- `am` or `amdsb-sc` — Amplitude demodulation, double sideband, suppressed carrier. Multiplies `y` by a sinusoid of frequency `fc` and applies a fifth-order Butterworth lowpass filter using `filtfilt`.

```
x = y.*cos(2*pi*fc*t);
[b,a] = butter(5,fc*2/fs);
x = filtfilt(b,a,x);
```

- `amdsb-tc` — Amplitude demodulation, double sideband, transmitted carrier. Multiplies `y` by a sinusoid of frequency `fc` and applies a fifth-order Butterworth lowpass filter using `filtfilt`.

  ```
  x = y.*cos(2*pi*fc*t);
  [b,a] = butter(5,fc*2/fs);
  x = filtfilt(b,a,x);
  ```

  If you specify `opt`, `demod` subtracts scalar `opt` from `x`. The default value for `opt` is 0.

- `amssb` — Amplitude demodulation, single sideband. Multiplies `y` by a sinusoid of frequency `fc` and applies a fifth-order Butterworth lowpass filter using `filtfilt`..

  ```
  x = y.*cos(2*pi*fc*t);
  [b,a] = butter(5,fc*2/fs);
  x = filtfilt(b,a,x);
  ```

- `fm` — Frequency demodulation. Demodulates the FM waveform by modulating the Hilbert transform of `y` by a complex exponential of frequency `-fc` Hz and obtains the instantaneous frequency of the result..

  ```
  y=cos(2*pi*fc*t + opt*cumsum(x))
  ```

  `cumsum` is a rectangular approximation of the integral of `x`. `modulate` uses `opt` as the constant of frequency modulation. If you do not specify the `opt` parameter, `modulate` uses a default of `opt = (fc/fs)*2*pi/(max(max(x)))` so the maximum frequency excursion from `fc` is `fc` Hz.

- `pm` — Phase demodulation. Demodulates the PM waveform by modulating the Hilbert transform of `y` by a complex exponential of frequency `-fc` Hz and obtains the instantaneous phase of the result.

  ```
  y=cos(2*pi*fc*t + opt*x)
  ```

  `modulate` uses `opt` as the constant of phase modulation. If you do not specify the `opt` parameter, `modulate` uses a default of `opt = pi/(max(max(x)))` so the maximum phase excursion is π radians.

- `pwm` — Pulse-width demodulation. Finds the pulse widths of a pulse-width modulated signal `y`. `demod` returns in `x` a vector whose elements specify the width of each pulse in fractions of a period. The pulses in `y` should start at the beginning of each carrier period, that is, they should be left justified. `modulate(x,fc,fs,'pwm','centered')` yields pulses centered at the beginning of each period. The length of `y` is `length(x)*fs/fc`.

- `ppm` — Pulse-position demodulation. Finds the pulse positions of a pulse-position modulated signal `y`. For correct demodulation, the pulses cannot overlap. `x` is length `length(t)*fc/fs`.

- `qam`— Quadrature amplitude demodulation. `[x1,x2] = demod(y,fc,fs,'qam')` multiplies `y` by a cosine and a sine of frequency `fc` and applies a fifth-order Butterworth lowpass filter using `filtfilt`.

  ```
  x1 = y.*cos(2*pi*fc*t);
  x2 = y.*sin(2*pi*fc*t);
  [b,a] = butter(5,fc*2/fs);
  x1 = filtfilt(b,a,x1);
  x2 = filtfilt(b,a,x2);
  ```

  The input argument `opt` must be the same size as `y`.

`opt` — Optional input for some methods

real vector

Optional input, specified for some methods. Refer to `method` for more details on how to use `opt`.

Output Arguments

`x` — Demodulated message signal

real vector | real matrix

Demodulated message signal, returned as a real vector or matrix.

See Also

`modulate` | `vco` | `fskdemod` | `genqamdemod` | `mskdemod` | `pamdemod` | `pmdemod` | `qamdemod`

Introduced before R2006a

designfilt

Design digital filters

Note `designfilt` no longer assists in correcting calls to the function within a script or function. For more information, see “Compatibility Considerations”.

Syntax

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

designfilt(d)
```

Description

`d = designfilt(resp,Name,Value)` designs a `digitalFilter` object, `d`, with response type `resp`. Examples of `resp` are `'lowpassfir'` and `'bandstopiir'`. Specify the filter further using a set of “Name-Value Pair Arguments” on page 1-292. The allowed specification sets depend on `resp` and consist of combinations of these:

- “Frequency Constraints” on page 1-0 correspond to the frequencies at which a filter exhibits a desired behavior. Examples include `'PassbandFrequency'` and `'CutoffFrequency'`. You must always specify the frequency constraints.
- “Magnitude Constraints” on page 1-0 describe the filter behavior at particular frequency ranges. Examples include `'PassbandRipple'` and `'StopbandAttenuation'`. `designfilt` provides default values for magnitude constraints left unspecified. In arbitrary-magnitude designs you must always specify the vectors of desired amplitudes.
- “Filter Order” on page 1-0 . 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.
- “Design Method” on page 1-0 is the algorithm used to design the filter. Examples include constrained least squares (`'cls'`) and Kaiser windowing (`'kaiserwin'`). 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 Method Options” on page 1-0 are parameters specific to a given design method. Examples include `'Window'` for the `'window'` method and optimization `'Weights'` for arbitrary-magnitude equiripple designs. `designfilt` provides default values for design options left unspecified.
- “Sample Rate” on page 1-0 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 arguments at the command line, `designfilt` offers to open a “Filter Design Assistant” on page 1-297. 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 1-297. The assistant

helps you design the filter and pastes the corrected MATLAB code on the command line. The designed filter is saved to the workspace.

- Use `filter` in the form `dataOut = filter(d,dataIn)` to filter a signal with a `digitalFilter`, `d`. For IIR filters, the `filter` function uses a direct-form II implementation.
- Use **FVTool** to visualize a `digitalFilter`, `d`.
- Type `d.Coefficients` to obtain the coefficients of a `digitalFilter`, `d`. For IIR filters, the coefficients are expressed as second-order sections.
- See `digitalFilter` for a list of the filtering and analysis functions available for use with `digitalFilter` objects.

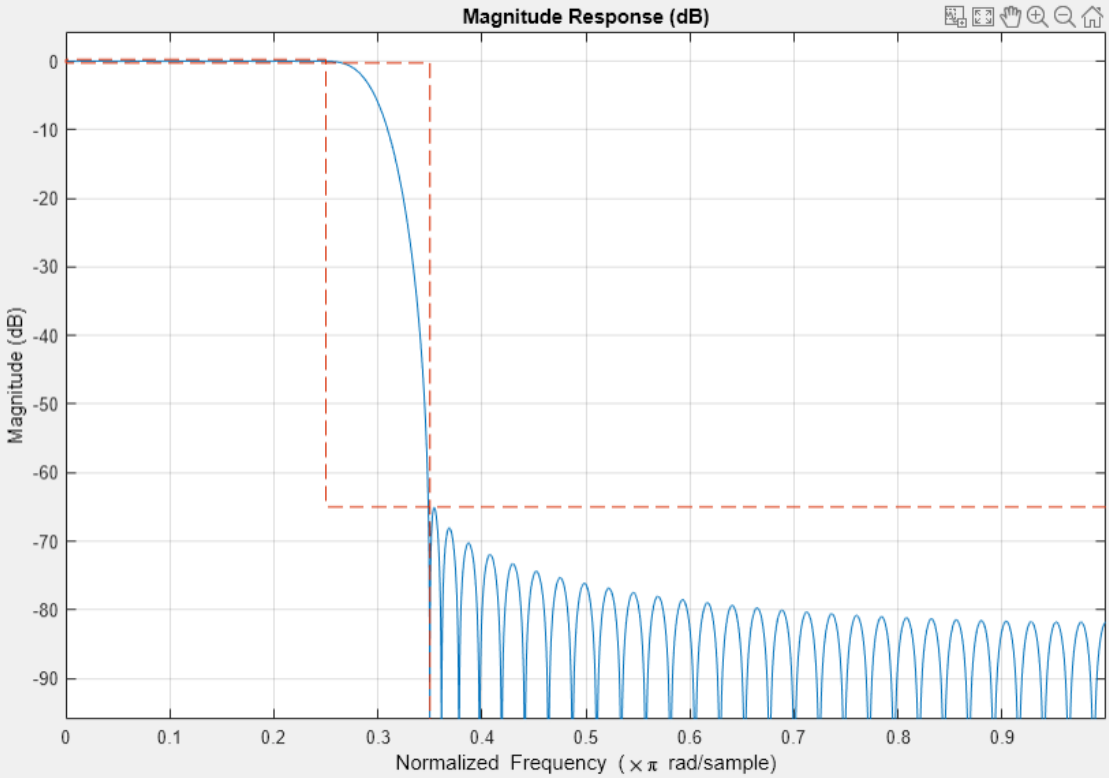
`designfilt(d)` lets you edit an existing digital filter, `d`. It opens a “Filter Design Assistant” on page 1-297 populated with the filter’s specifications, which you can then modify. This is the only way you can edit a `digitalFilter` object. Its properties are otherwise read-only.

Examples

Lowpass FIR Filter

Design a minimum-order lowpass FIR filter with normalized passband frequency 0.25π rad/sample, stopband frequency 0.35π rad/sample, 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)
```

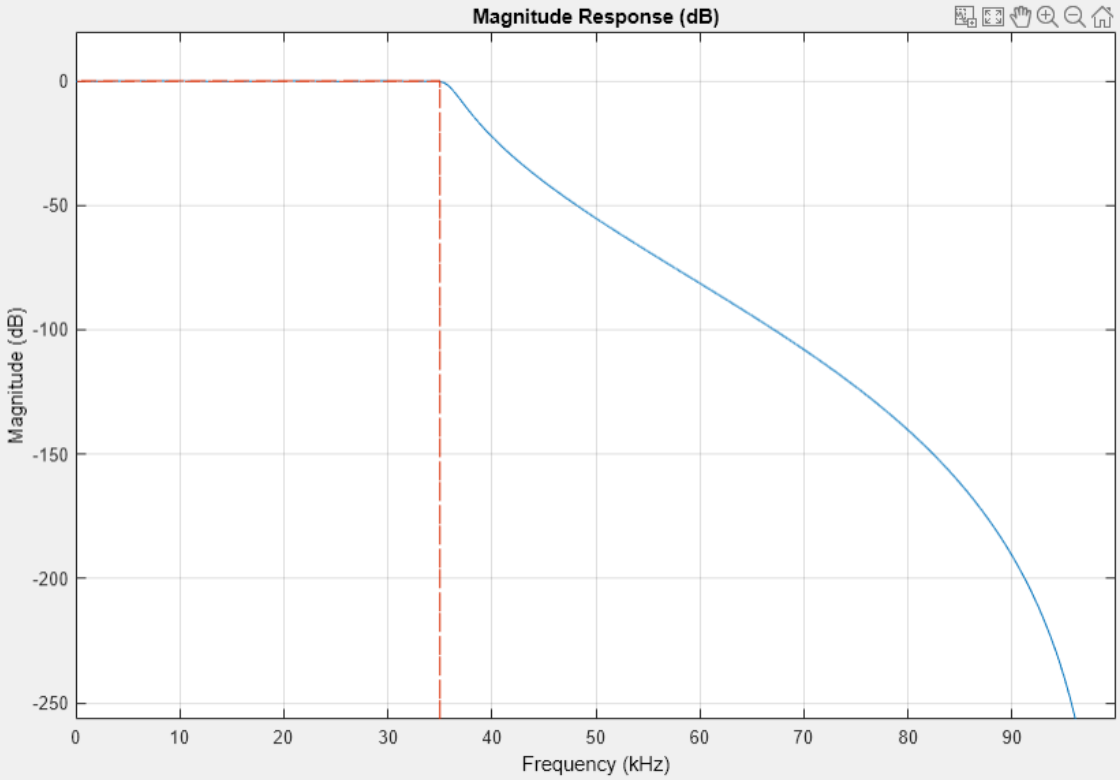



```
dataIn = rand(1000,1);
dataOut = filter(lpFilt,dataIn);
```

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.

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



Use the filter you designed to filter a 1000-sample random signal.

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

Output the filter coefficients, expressed as second-order sections.

```
sos = lpFilt.Coefficients
```

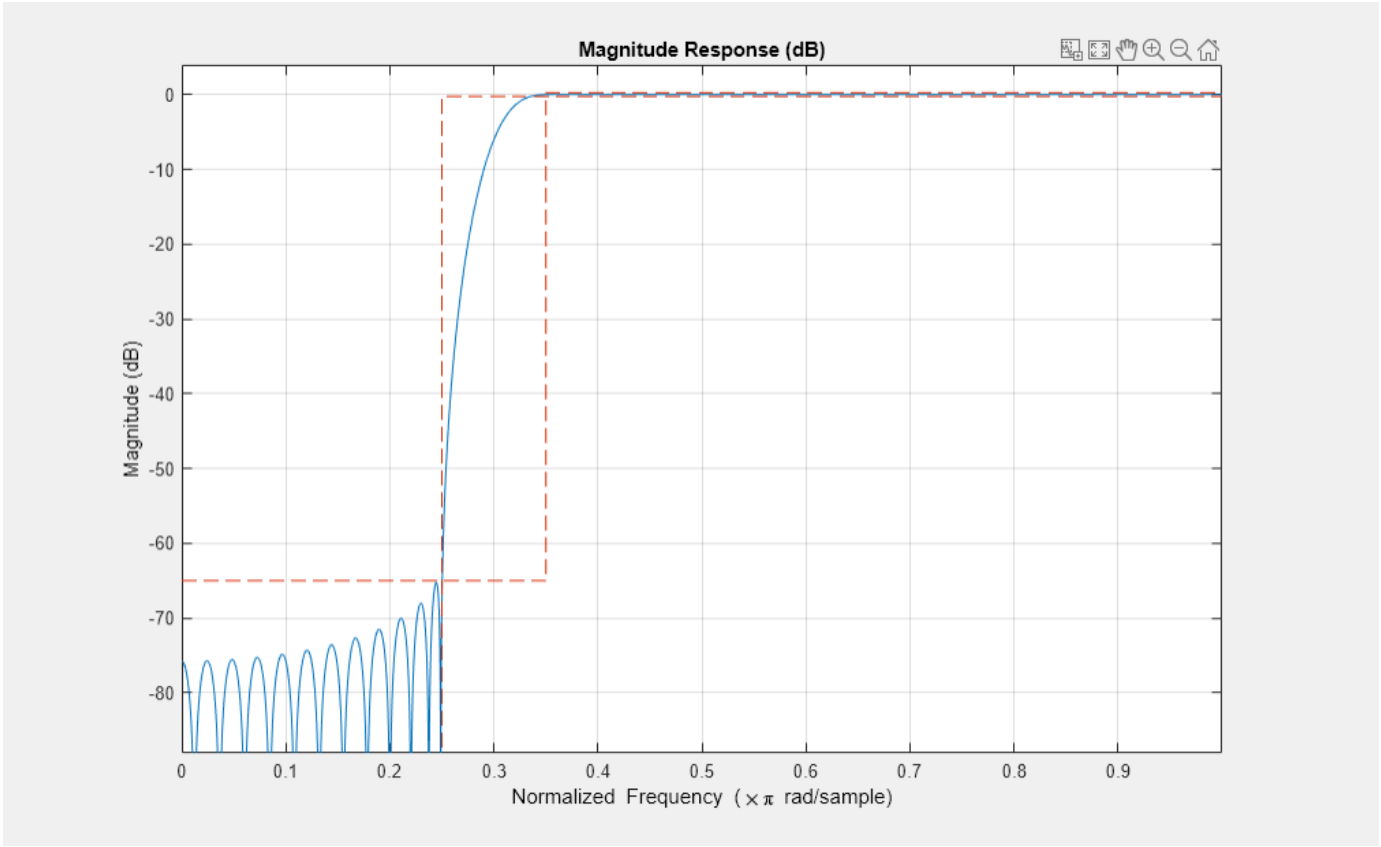
```
sos = 4×6

    0.2666    0.5333    0.2666    1.0000   -0.8346    0.9073
    0.1943    0.3886    0.1943    1.0000   -0.9586    0.7403
    0.1012    0.2023    0.1012    1.0000   -1.1912    0.5983
    0.0318    0.0636    0.0318    1.0000   -1.3810    0.5090
```

Highpass FIR Filter

Design a minimum-order highpass FIR filter with normalized stopband frequency 0.25π rad/sample, passband frequency 0.35π rad/sample, 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)
```

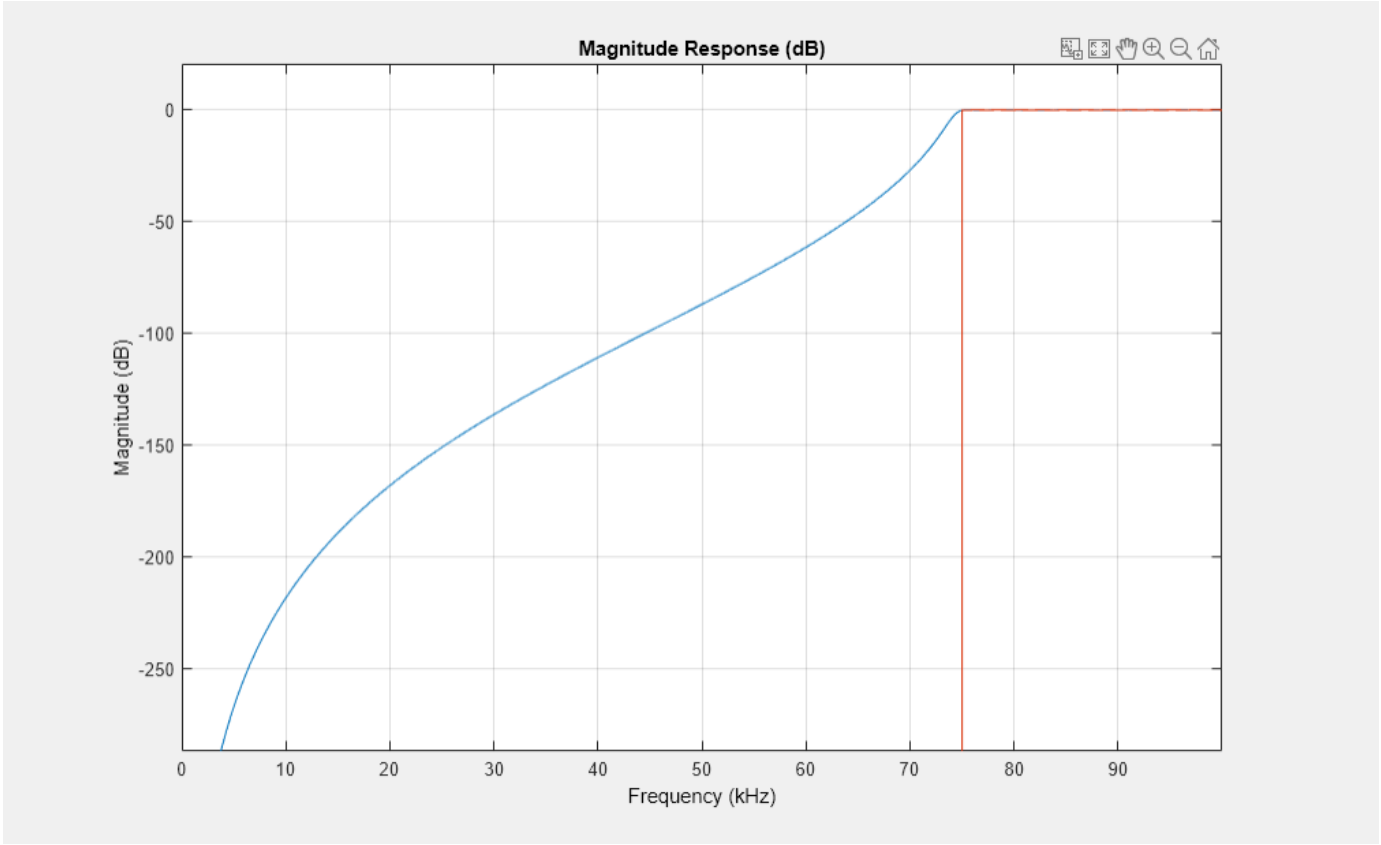



```
dataIn = randn(1000,1);
dataOut = filter(hpFilt,dataIn);
```

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)
```

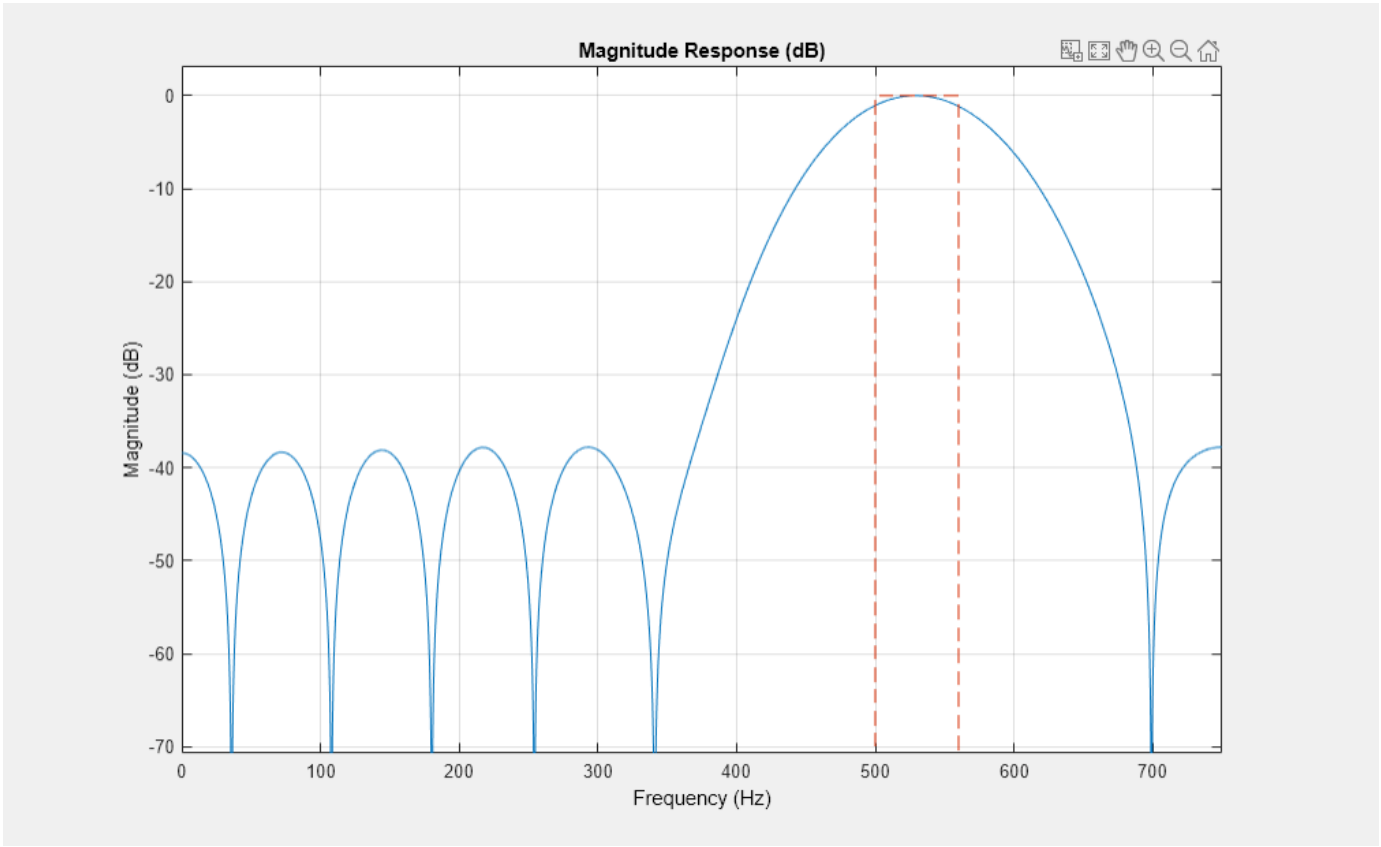



```
dataIn = randn(1000,1);
dataOut = filter(hpFilt,dataIn);
```

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);
```

Output the filter coefficients.

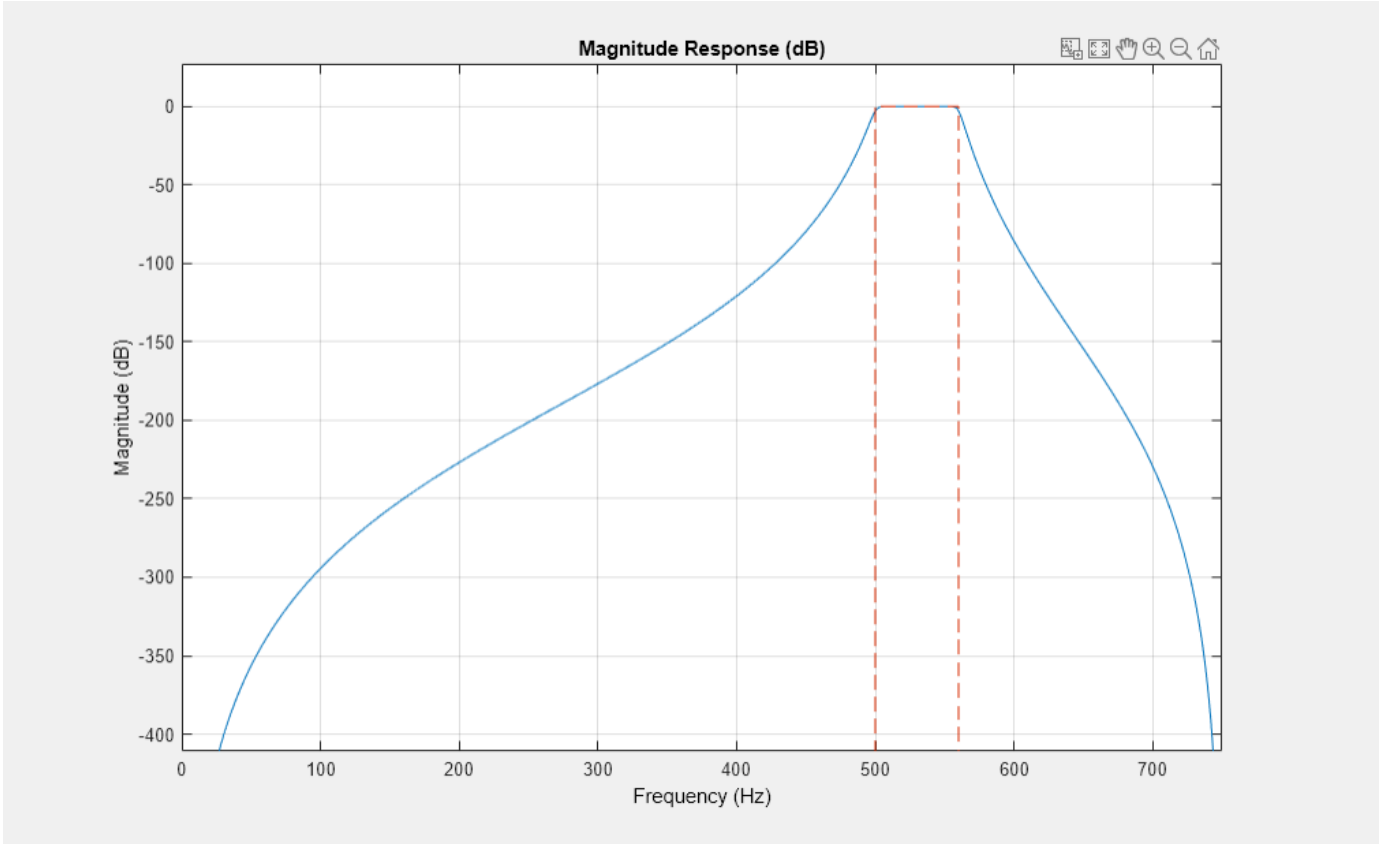
```
b = bpFilt.Coefficients
```

```
b = 1×21

   -0.0113    0.0067    0.0125   -0.0445    0.0504    0.0101   -0.1070    0.1407   -0.0464   -0.1
```

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)
```

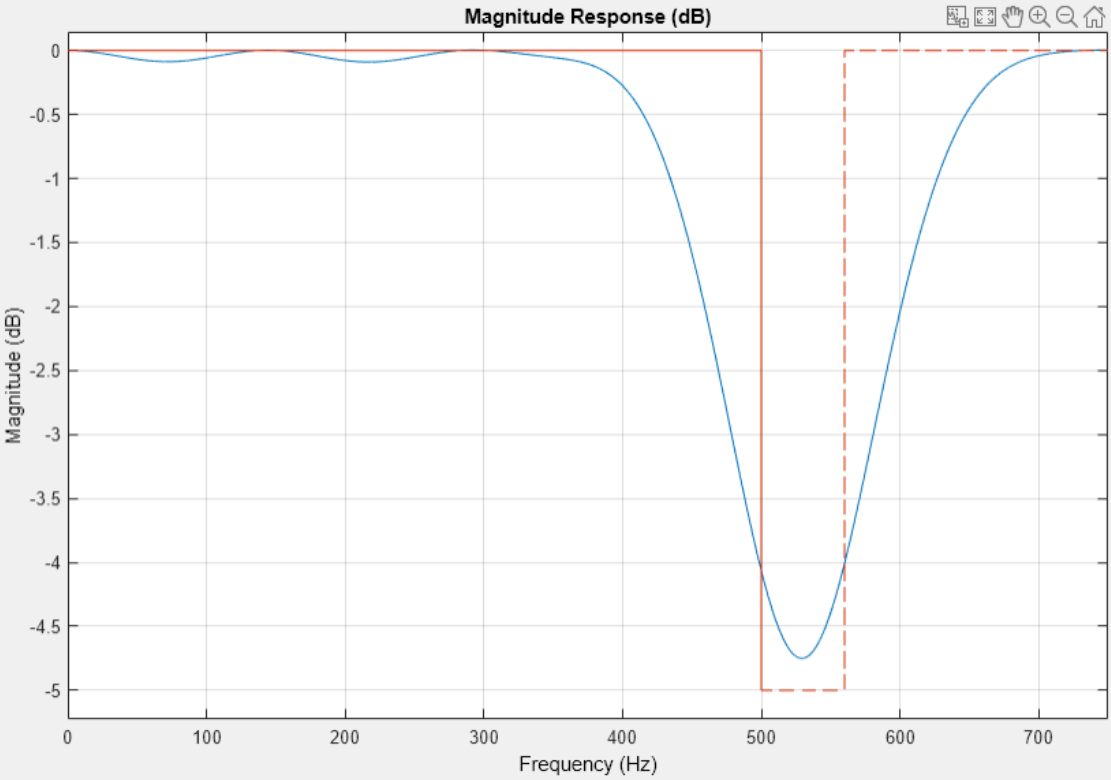



```
dataIn = randn(1000,1);
dataOut = filter(bpFilt,dataIn);
```

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)
```

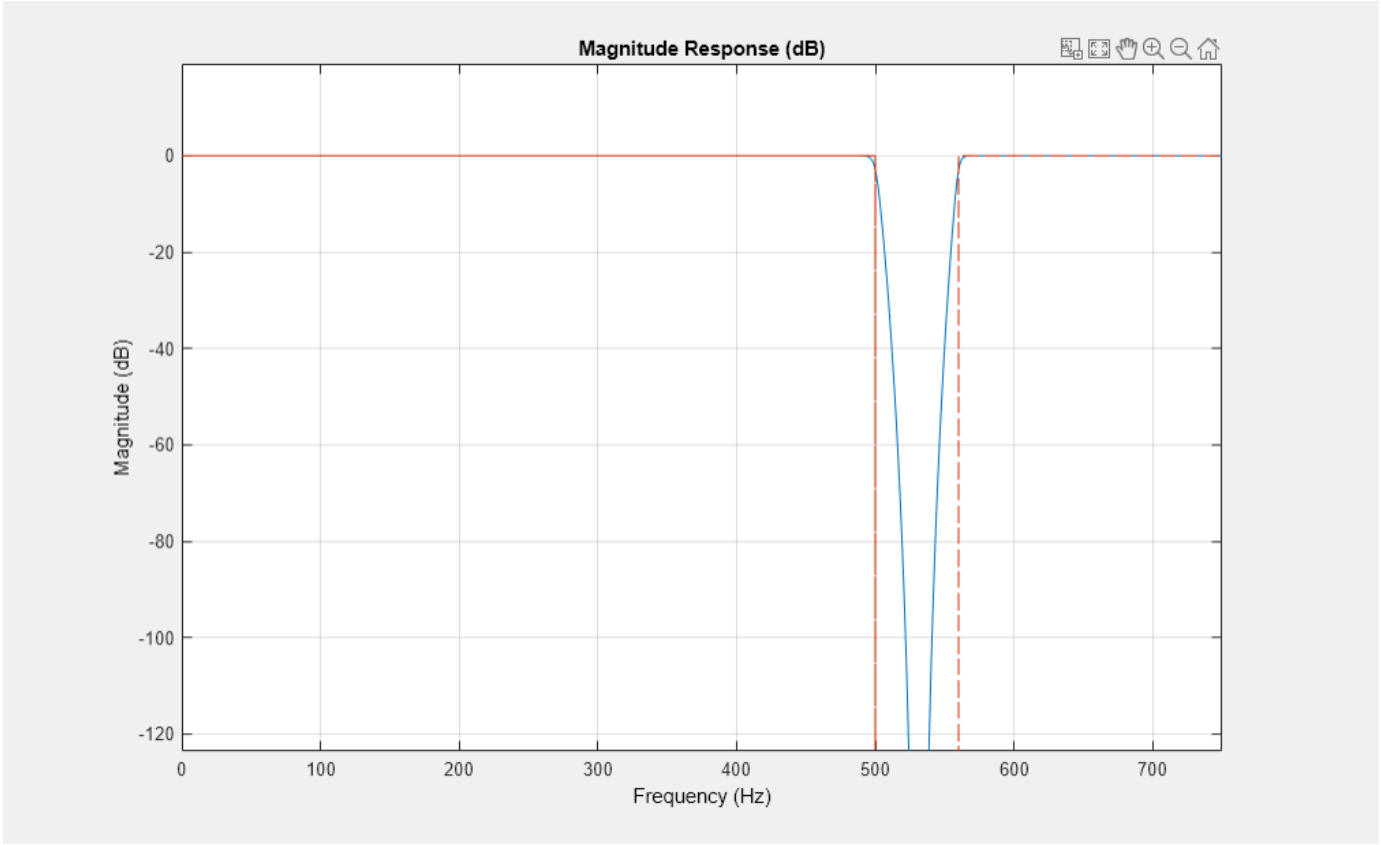



```
dataIn = randn(1000,1);
dataOut = filter(bsFilt,dataIn);
```

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)
```

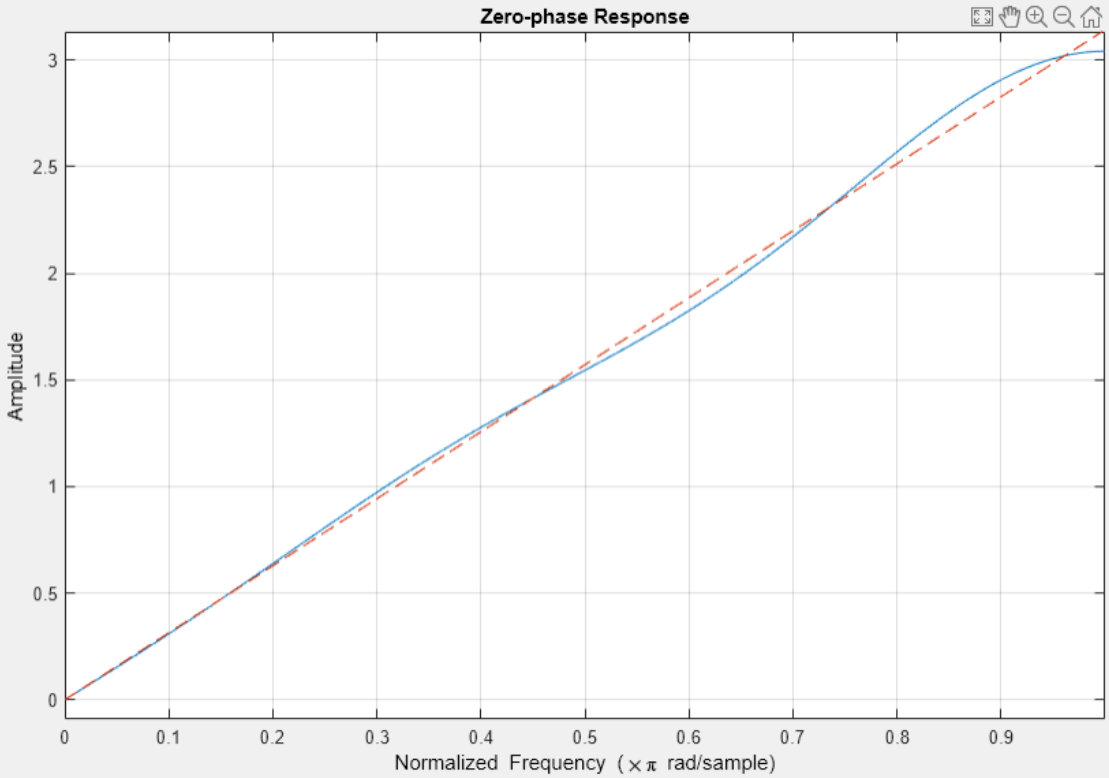



```
dataIn = randn(1000,1);
dataOut = filter(bsFilt,dataIn);
```

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')
```

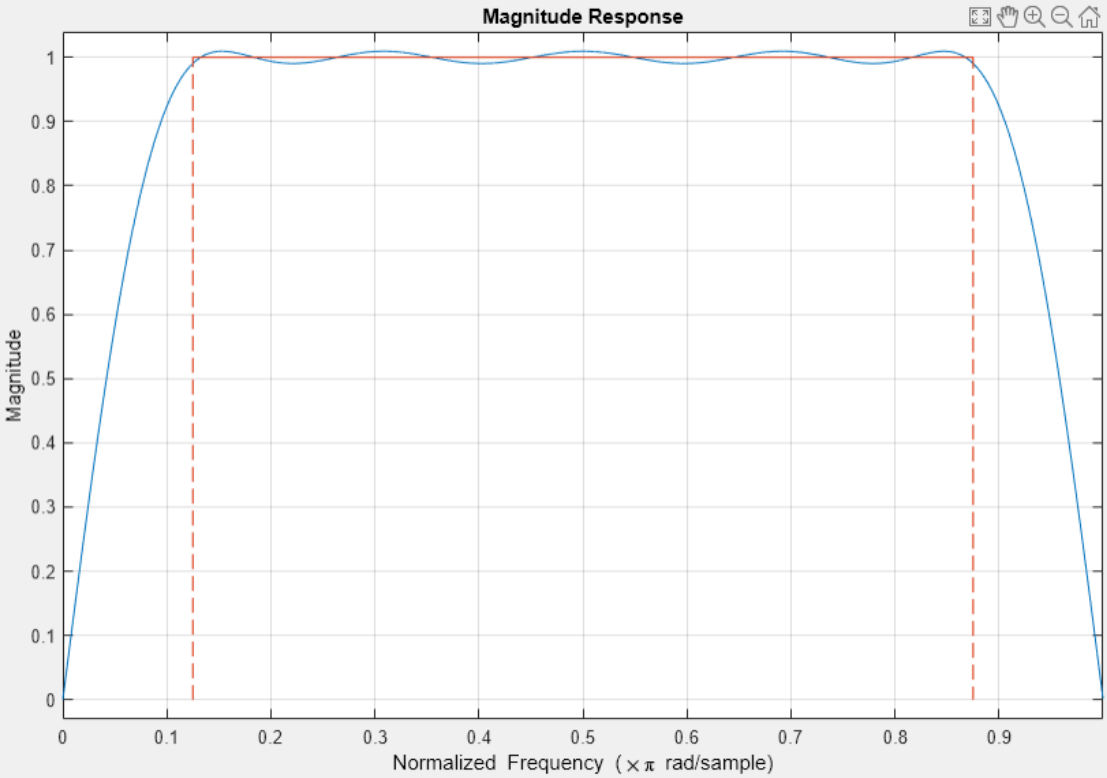



```
dataIn = randn(1000,1);
dataOut = filter(dFilt,dataIn);
```

FIR Hilbert Transformer

Design a Hilbert transformer of order 18. Specify a normalized transition width of 0.25π rad/sample. 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')
```

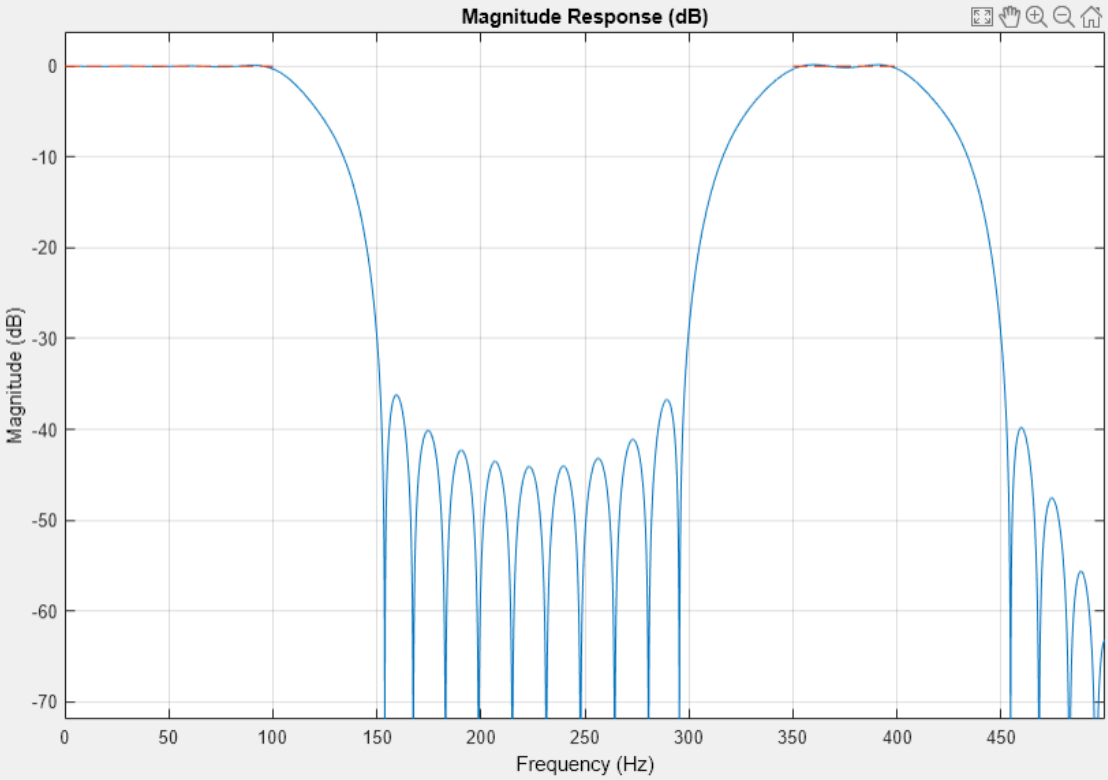



```
dataIn = randn(1000,1);
dataOut = filter(hFilt,dataIn);
```

Arbitrary-Magnitude FIR Filter

You are given a signal sampled at 1 kHz. Design a filter that stops frequencies between 100 Hz and 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], ...
         'SampleRate',1000);
fvtool(mbFilt)
```




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

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 character vector or string scalar.

'lowpassfir' — FIR lowpass filter

response type

Choose this option to design a finite impulse response (FIR) lowpass filter. This example uses the fifth specification set from the 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' 'StopbandFrequency'	'PassbandRipple' 'StopbandAttenuation'	'equiripple' (default)	N/A
			'kaiserwin'	'MinOrder' 'ScalePassband'
'FilterOrder'	'HalfPowerFrequency'	N/A	'maxflat'	N/A
'FilterOrder'	'CutoffFrequency'	N/A	'window'	'Window' 'ScalePassband'
'FilterOrder'	'CutoffFrequency'	'PassbandRipple' 'StopbandAttenuation'	'cls'	'PassbandOffset' 'ZeroPhase'
'FilterOrder'	'PassbandFrequency' 'StopbandFrequency'	N/A	'equiripple' (default)	'PassbandWeight' 'StopbandWeight'
			'ls'	'PassbandWeight' 'StopbandWeight'

`'lowpassiir'` — IIR lowpass filter
response type

Choose this option to design an infinite impulse response (IIR) lowpass filter. This example uses the first specification set from the 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' 'StopbandFrequency'	'PassbandRipple' 'StopbandAttenuation'	'butter' (default)	'MatchExactly'
			'cheby1'	'MatchExactly'
			'cheby2'	'MatchExactly'
			'ellip'	'MatchExactly'
'FilterOrder'	'HalfPowerFrequency'	N/A	'butter'	N/A
'FilterOrder'	'PassbandFrequency'	'PassbandRipple'	'cheby1'	N/A
'FilterOrder'	'PassbandFrequency'	'PassbandRipple' 'StopbandAttenuation'	'ellip'	N/A
'FilterOrder'	'StopbandFrequency'	'StopbandAttenuation'	'cheby2'	N/A
'NumeratorOrder' 'DenominatorOrder'	'HalfPowerFrequency'	N/A	'butter'	N/A

`'highpassfir'` — FIR highpass filter
response type

Choose this option to design a finite impulse response (FIR) highpass filter. This example uses the first specification set from the 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'` `'PassbandFrequency'`	`'StopbandAttenuation'` `'PassbandRipple'`	`'equiripple'` (default)	N/A
			`'kaiserwin'`	`'MinOrder'` `'ScalePassband'`
`'FilterOrder'`	`'CutoffFrequency'`	N/A	`'window'`	`'Window'` `'ScalePassband'`
`'FilterOrder'`	`'CutoffFrequency'`	`'StopbandAttenuation'` `'PassbandRipple'`	`'cls'`	`'PassbandOffset'` `'ZeroPhase'`
`'FilterOrder'`	`'StopbandFrequency'` `'PassbandFrequency'`	N/A	`'equiripple'` (default)	`'PassbandWeight'` `'StopbandWeight'`
			`'ls'`	`'PassbandWeight'` `'StopbandWeight'`

`'highpassiir'` — IIR highpass filter
response type

Choose this option to design an infinite impulse response (IIR) highpass filter. This example uses the first specification set from the 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' 'PassbandFrequency'	'StopbandAttenuation' 'PassbandRipple'	'butter' (default)	'MatchExactly'
			'cheby1'	'MatchExactly'
			'cheby2'	'MatchExactly'
			'ellip'	'MatchExactly'
'FilterOrder'	'HalfPowerFrequency'	N/A	'butter'	N/A
'FilterOrder'	'PassbandFrequency'	'PassbandRipple'	'cheby1'	N/A
'FilterOrder'	'PassbandFrequency'	'StopbandAttenuation' 'PassbandRipple'	'ellip'	N/A
'FilterOrder'	'StopbandFrequency'	'StopbandAttenuation'	'cheby2'	N/A
'NumeratorOrder' 'DenominatorOrder'	'HalfPowerFrequency'	N/A	'butter'	N/A

'bandpassfir' — FIR bandpass filter

response type

Choose this option to design a finite impulse response (FIR) bandpass filter. This example uses the fourth specification set from the 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' 'PassbandFrequency1' 'PassbandFrequency2' 'StopbandFrequency2'	'StopbandAttenuation1' 'PassbandRipple' 'StopbandAttenuation2'	'equiripple' (default)	N/A
			'kaiserwin'	'MinOrder' ScalePassband
'FilterOrder'	'CutoffFrequency1' 'CutoffFrequency2'	N/A	'window'	'Window' 'ScalePassband'
'FilterOrder'	'CutoffFrequency1' 'CutoffFrequency2'	'StopbandAttenuation1' 'PassbandRipple' 'StopbandAttenuation2'	'cls'	'PassbandOffset' 'ZeroPhase'
'FilterOrder'	'StopbandFrequency1' 'PassbandFrequency1' 'PassbandFrequency2' 'StopbandFrequency2'	N/A	'equiripple' (default)	'StopbandWeight1' 'PassbandWeight' 'StopbandWeight2'
			'ls'	'StopbandWeight1' 'PassbandWeight' 'StopbandWeight2'

'bandpassiir' — IIR bandpass filter

response type

Choose this option to design an infinite impulse response (IIR) bandpass filter. This example uses the first specification set from the 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)	'StopbandFrequency1' 'PassbandFrequency1' 'PassbandFrequency2' 'StopbandFrequency2'	'StopbandAttenuation1' 'PassbandRipple' 'StopbandAttenuation2'	'butter' (default)	'MatchExactly'
			'cheby1'	'MatchExactly'
			'cheby2'	'MatchExactly'
			'ellip'	'MatchExactly'
'FilterOrder'	'HalfPowerFrequency1' 'HalfPowerFrequency2'	N/A	'butter'	N/A
'FilterOrder'	'PassbandFrequency1' 'PassbandFrequency2'	'PassbandRipple'	'cheby1'	N/A
'FilterOrder'	'PassbandFrequency1' 'PassbandFrequency2'	'StopbandAttenuation1' 'PassbandRipple' 'StopbandAttenuation2'	'ellip'	N/A
'FilterOrder'	'StopbandFrequency1' 'StopbandFrequency2'	'StopbandAttenuation'	'cheby2'	N/A

`'bandstopfir'` — FIR bandstop filter

response type

Choose this option to design a finite impulse response (FIR) bandstop filter. This example uses the fourth specification set from the 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)	`'PassbandFrequency1'`
	`'StopbandFrequency1'`
	`'StopbandFrequency2'`
	`'PassbandFrequency2'`
		`'PassbandRipple1'`
		`'StopbandAttenuation'`
		`'PassbandRipple2'`
			`'equiripple'` (default)	N/A

			`'kaiserwin'`	`'MinOrder'` `'ScalePassband'`
`'FilterOrder'`	`'CutoffFrequency1'` `'CutoffFrequency2'`	N/A	`'window'`	`'Window'` `'ScalePassband'`

Filter Order Argument Names	Frequency Constraint Argument Names	Magnitude Constraint Argument Names	'DesignMethod' Argument Values	Design Option Argument Names
'FilterOrder'	'CutoffFrequency1' 'CutoffFrequency2'	'PassbandRipple1' 'StopbandAttenuation' 'PassbandRipple2'	'cls'	'PassbandOffset' 'ZeroPhase'
'FilterOrder'	'PassbandFrequency1' 'StopbandFrequency1' 'StopbandFrequency2' 'PassbandFrequency2'	N/A	'equiripple' (default)	'PassbandWeight1' 'StopbandWeight' 'PassbandWeight2'
			'ls'	'PassbandWeight1' 'StopbandWeight' 'PassbandWeight2'

'bandstopiir' — IIR bandstop filter
response type

Choose this option to design an infinite impulse response (IIR) bandstop filter. This example uses the first specification set from the 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)	'PassbandFrequency1' 'StopbandFrequency1' 'StopbandFrequency2' 'PassbandFrequency2'	'PassbandRipple1' 'StopbandAttenuation' 'PassbandRipple2'	'butter' (default)	'MatchExactly'
			'cheby1'	'MatchExactly'
			'cheby2'	'MatchExactly'
			'ellip'	'MatchExactly'
'FilterOrder'	'HalfPowerFrequency1' 'HalfPowerFrequency2'	N/A	'butter'	N/A
'FilterOrder'	'PassbandFrequency1' 'PassbandFrequency2'	'PassbandRipple'	'cheby1'	N/A
'FilterOrder'	'PassbandFrequency1' 'PassbandFrequency2'	'PassbandRipple' 'StopbandAttenuation'	'ellip'	N/A
'FilterOrder'	'StopbandFrequency1' 'StopbandFrequency2'	'StopbandAttenuation'	'cheby2'	N/A

`'differentiatorfir'` — FIR differentiator filter
response type

Choose this option to design a finite impulse response (FIR) differentiator filter. This example uses the second specification set from the 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'` (default)	N/A
			`'ls'`	N/A
`'FilterOrder'`	`'PassbandFrequency'` `'StopbandFrequency'`	N/A	`'equiripple'` (default)	`'PassbandWeight'` `'StopbandWeight'`
			`'ls'`	N/A

`'hilbertfir'` — FIR Hilbert transformer filter
response type

Choose this option to design a finite impulse response (FIR) Hilbert transformer filter. This example uses the specification set from the 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'` — FIR filter of arbitrary magnitude response
response type

Choose this option to design a finite impulse response (FIR) filter of arbitrary magnitude response. This example uses the second specification set from the 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'` (default)	`'Window'`
			`'equiripple'`	`'Weights'`
			`'ls'`	`'Weights'`

`'FilterOrder'`
 	
`'NumBands'`
	`'BandFrequencies1'`
	
	...
	
	`'BandFrequenciesN'`
		`'BandAmplitudes1'`
		
		...
		
		`'BandAmplitudesN'`
			`'equiripple'` (default)	`'BandWeights1'`
				
				...
				
				`'BandWeightsN'`

			`'ls'`	`'BandWeights1'`
				
				...
				
				`'BandWeightsN'`

Data Types: `char` | `string`

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Not all name-value combinations 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 a positive integer scalar.

Data Types: `double`

`NumeratorOrder` — Numerator order
positive integer scalar

Numerator order of an IIR design, specified as a positive integer scalar.

Data Types: `double`

`DenominatorOrder` — Denominator order
positive integer scalar

Denominator order of an IIR design, specified as a positive integer scalar.

Data Types: `double`

Frequency Constraints

`PassbandFrequency`, `PassbandFrequency1`, `PassbandFrequency2` — Passband frequency
positive scalar

Passband frequency, specified as 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 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 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 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 a positive scalar.

Data Types: `double`

`Frequencies` — Response frequencies
vector

Response frequencies, specified as 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 a positive integer scalar not greater than 10.

Data Types: `double`

`BandFrequencies1`, `...`, `BandFrequenciesN` — Multiband response frequencies
vectors

Multiband response frequencies, specified as numeric vectors. `'BandFrequenciesi'`, 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, `'BandAmplitudesi'`. `'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 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 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 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 numeric vectors. `'BandAmplitudesi'`, 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. `'BandAmplitudesi'` must have the same length as `'BandFrequenciesi'`. 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 a character vector or string scalar. 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'` uses a least-squares approximation to compute the filter coefficients and then smooths the impulse response with `'Window'`.

Data Types: `char` | `string`

Design Method Options

`MinOrder` — Minimum order parity

`'any'` (default) | `'even'`

Minimum order parity of a `'kaiserwin'` design, specified as `'any'` or `'even'`. When you set `'MinOrder'` to `'even'`, `designfilt` returns a minimum-order filter with even order. When you set `'MinOrder'` to `'any'`, the returned filter can have even or odd order, whichever is smaller.

Data Types: `char` | `string`

`Window` — Window

numeric vector | window name | function handle | cell array

Window, specified as a vector of length $N + 1$, where N is the filter order. `'Window'` can also be paired with a window name 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` | `string` | `function_handle` | `cell`

`MatchExactly` — Band to match exactly
`'stopband'` | `'passband'` | `'both'`

Band to match exactly, specified as `'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` | `string`

`PassbandOffset` — Passband offset
0 (default) | positive scalar

Passband offset, specified as 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 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 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 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 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 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 sets of positive scalars or of vectors. `'BandWeightsi'`, 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, `'BandWeightsi'` must have the same length as `'BandAmplitudesi'`.

Data Types: `double`

Sample Rate

`SampleRate` — Sample rate
2 (default) | positive scalar

Sample rate, specified as 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.)

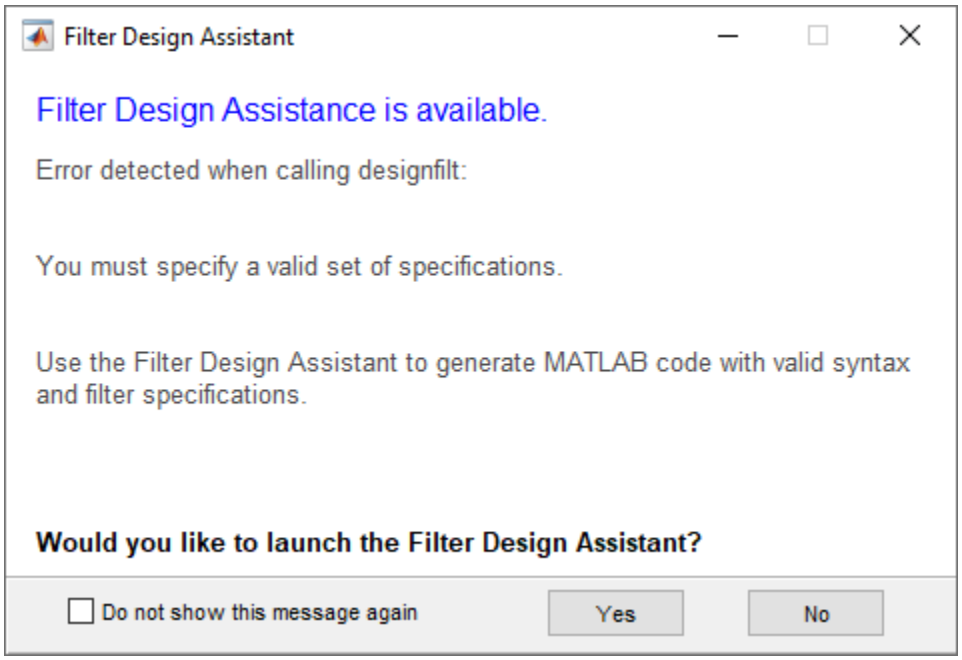
The assistant behaves differently if you call `designfilt` at the command line or within a script or function.

Filter Design Assistant at the Command Line

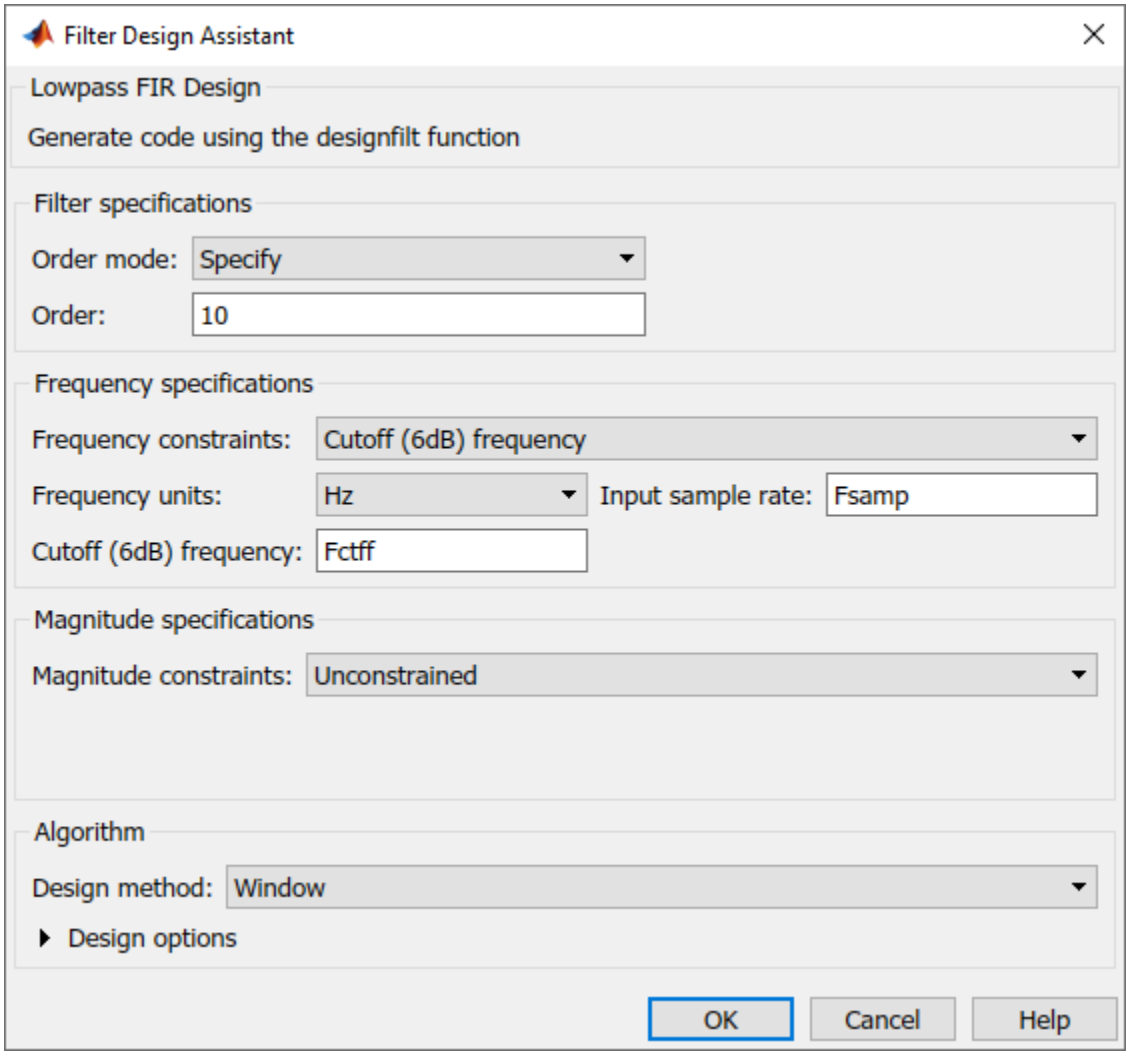
You are given a signal sampled at 2 kHz. You are asked to design a lowpass FIR filter that suppresses frequency components higher than 650 Hz. The “cutoff frequency” sounds like a good candidate for a specification parameter. You type this code at the MATLAB command line.

```
Fsamp = 2e3;
Fctff = 650;
dee = 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 this code on the command line.

```
designfilt('lowpassfir','FilterOrder', 10, ...
    'CutoffFrequency',Fctff,'SampleRate',2000);
```

Typing the name of the filter reiterates the information from the dialog box.

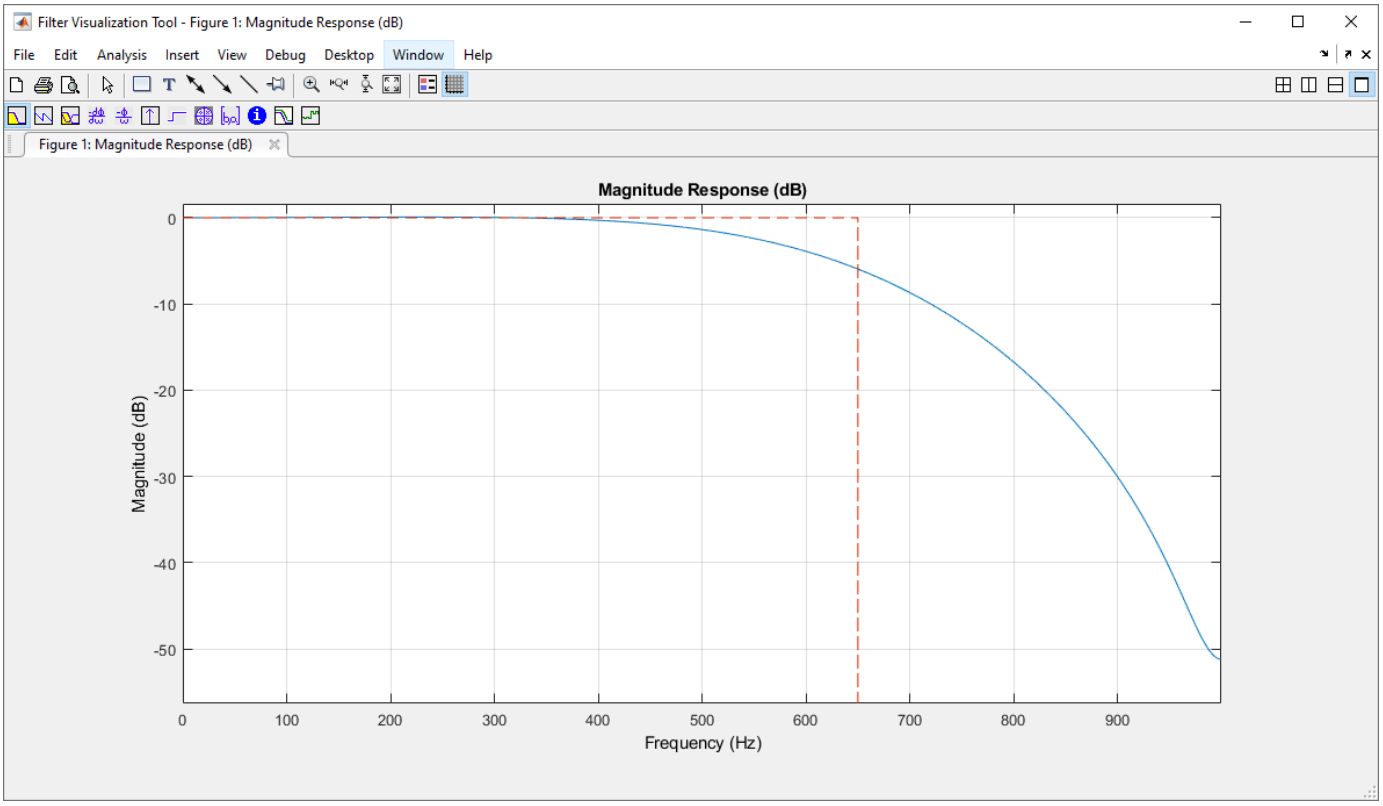
```
dee
```

```
dee =
 digitalFilter with properties:

    Coefficients: [-0.0036 0.0127 -0.0066 -0.0881 0.2595 ...
                   0.6521 0.2595 -0.0881 -0.0066 0.0127 -0.0036]
    Specifications:
    FrequencyResponse: 'lowpass'
      ImpulseResponse: 'fir'
           SampleRate: 2000
      CutoffFrequency: 650
          FilterOrder: 10
```

```
        DesignMethod: 'window'

 Use fvtool to visualize filter
 Use designfilt to edit filter
 Use filter to filter data
```

You invoke **FVTool** and get a plot of `dee`'s frequency response.

```
fvtool(dee)
```

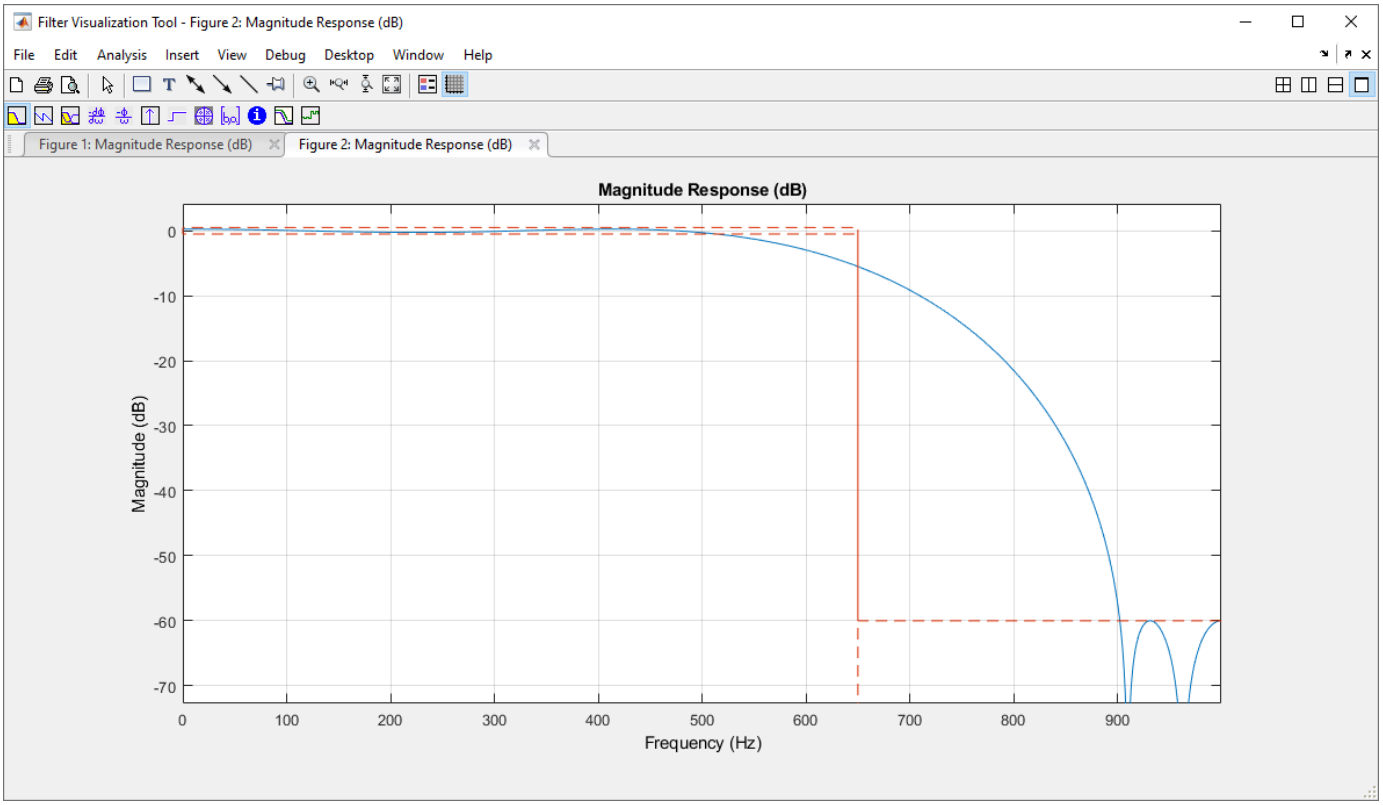


The cutoff does not look particularly sharp. The response is above 40 dB for most frequencies. You remember that the assistant had an option to set up a "magnitude constraint" called the "stopband attenuation". Open the assistant by calling `designfilt` with the filter name as input.

```
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',650,'PassbandRipple',1, ...
    'StopbandAttenuation',60,'SampleRate',2000);
fvtool(dee)
```

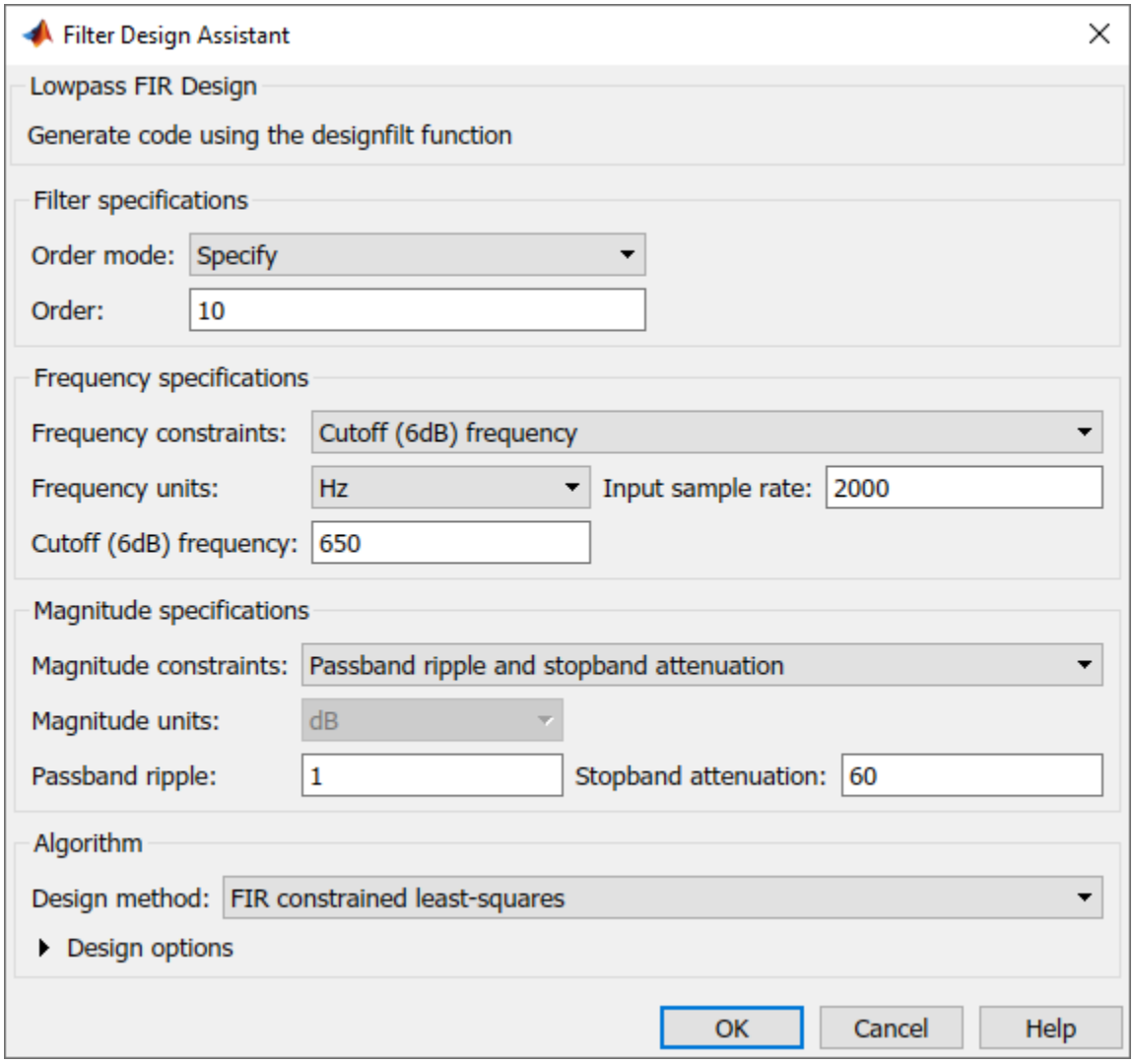


The cutoff still does not look sharp. The attenuation is indeed 60 dB, but for frequencies above 900 Hz.

Again invoke `designfilt` with your filter as input.

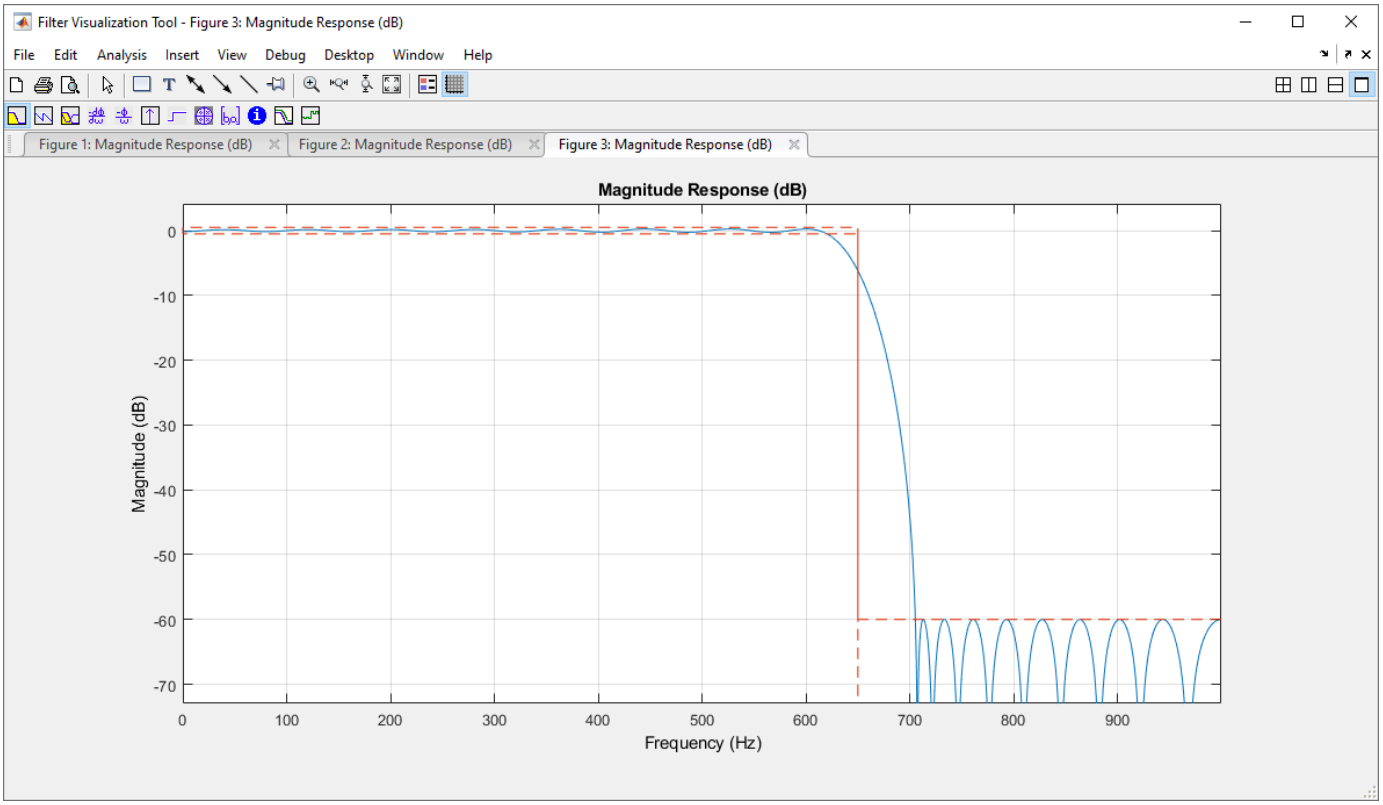
```
designfilt(dee)
```

The assistant reappears.



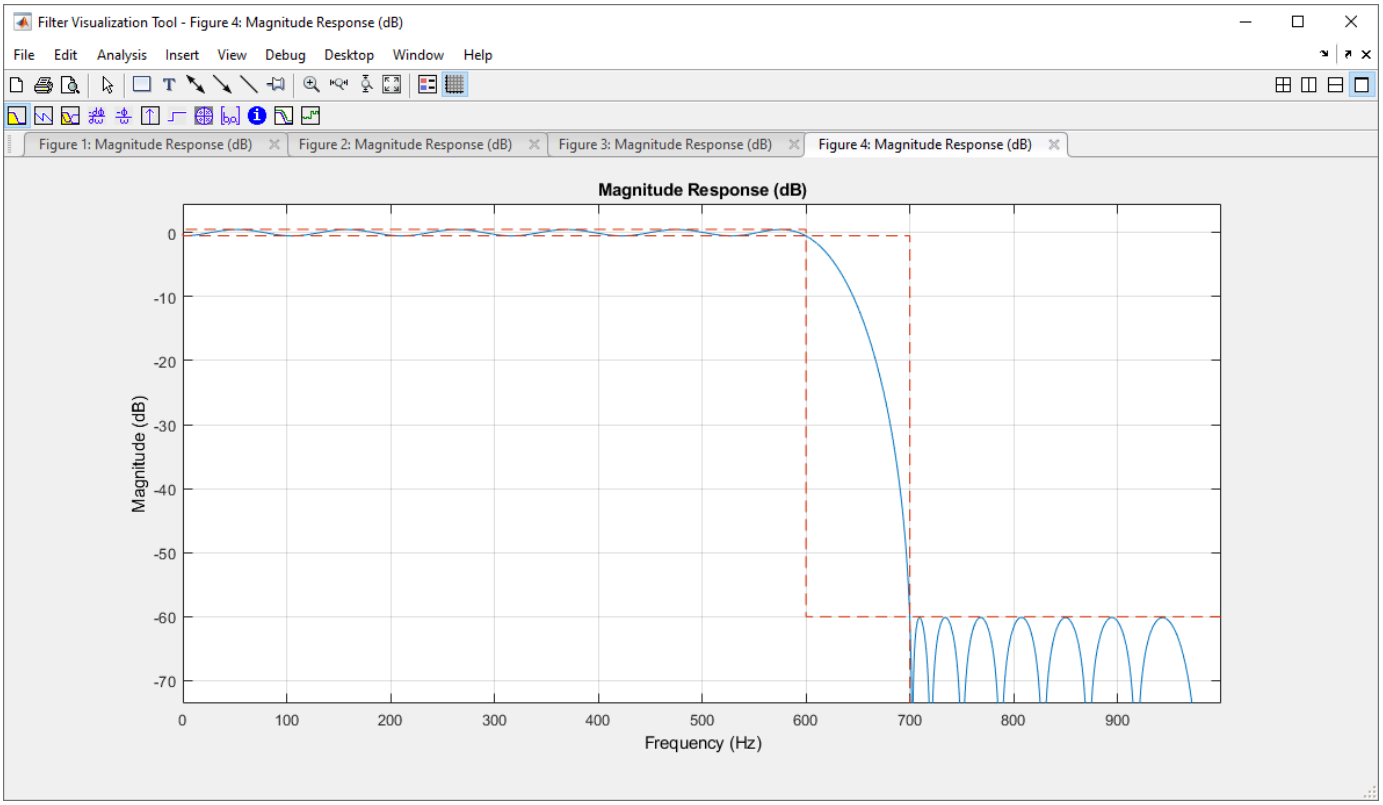
To narrow the distinction between accepted and rejected frequencies, increase the order of the filter or change `Frequency constraints` from `Cutoff (6dB) frequency` to `Passband and stopband frequencies`. If you change the filter order from 10 to 50, you get a sharper filter.

```
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(Order,dataIn)
hpFilter = designfilt('highpassiir','FilterOrder',N);
dataOut = filter(hpFilter,dataIn);
end
```

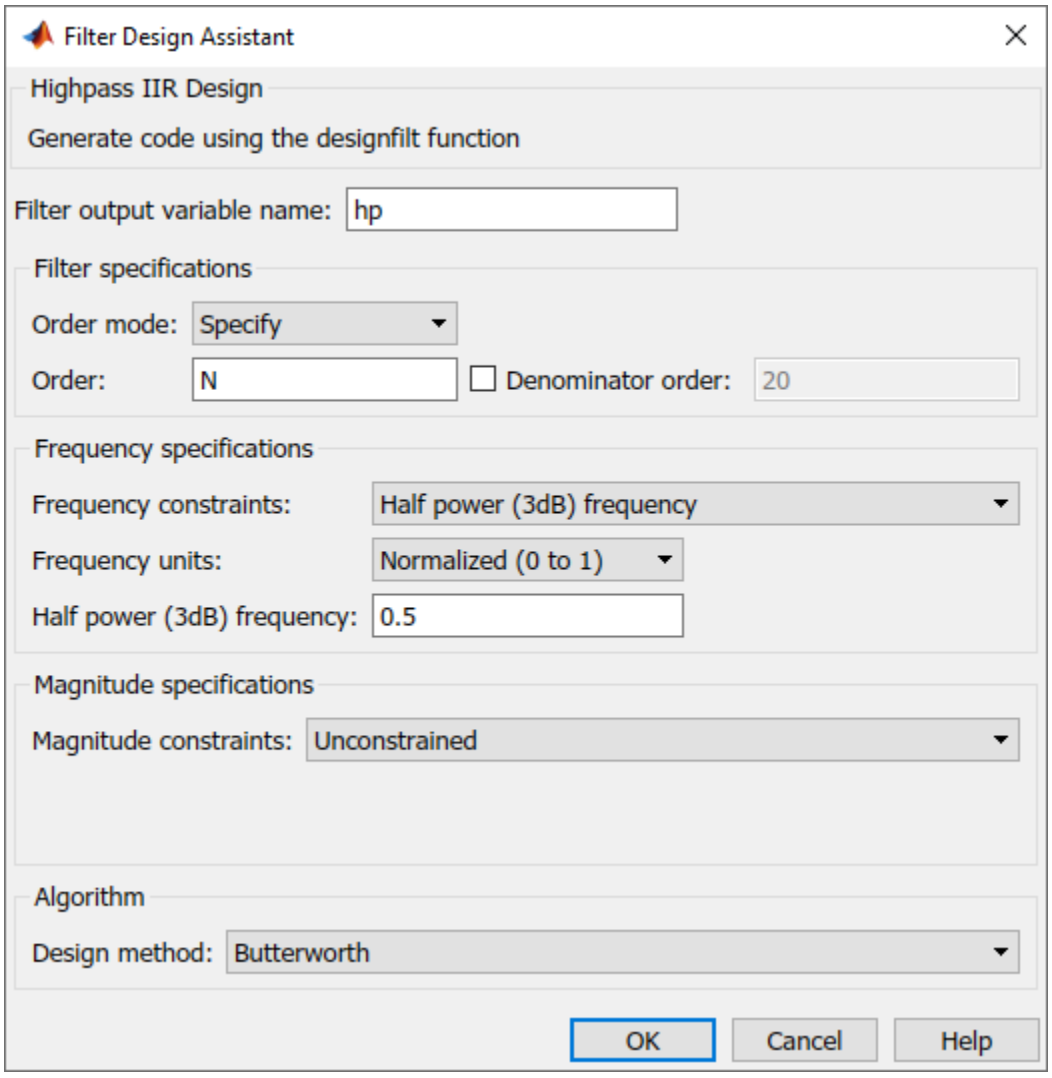
To test your function, create a signal composed of two sinusoids with frequencies 500 and 800 Hz and generate samples for 0.1 s. A 5th-order filter seems reasonable as an initial guess. Create a script called `driveHPfilt.m`.

```
% 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);
N = 5;
sgout = hipassfilt(N,sgin);
```

When you run the script at the command line, you get an error message.

```
Error using designfilt (line 364)

Click here to launch an assistant to help you design your filter.

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);
```

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 this on the command line.

```
hp = designfilt('highpassiir','FilterOrder',N, ...
    'PassbandFrequency',700,'PassbandRipple',1, ...
    'SampleRate',2000);
```

The new `digitalFilter` object `hp` is saved to the workspace. 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 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® software and the MATLAB desktop to run. It is not supported if you run MATLAB with the `-nojvm`, `-nodisplay`, or `-nodesktop` options.

Compatibility Considerations

`designfilt` function no longer assists in correcting calls to `designfilt`

Behavior changed in R2021b

Starting in R2021b, the `designfilt` function no longer assists in correcting calls to `designfilt` within a script or function. In previous releases, the function automatically corrected and executed code on the command line.

You do not need to make any changes to your code. If the call to `designfilt` contains an error, the function issues an error with a link to open the Filter Design Assistant. You can use the assistant to generate a filter and display the corresponding code on the command line. The generated filter object is saved to the workspace.

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`

Topics

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

Introduced in R2014a

Design Filter

Design a digital filter in the Live Editor

Description

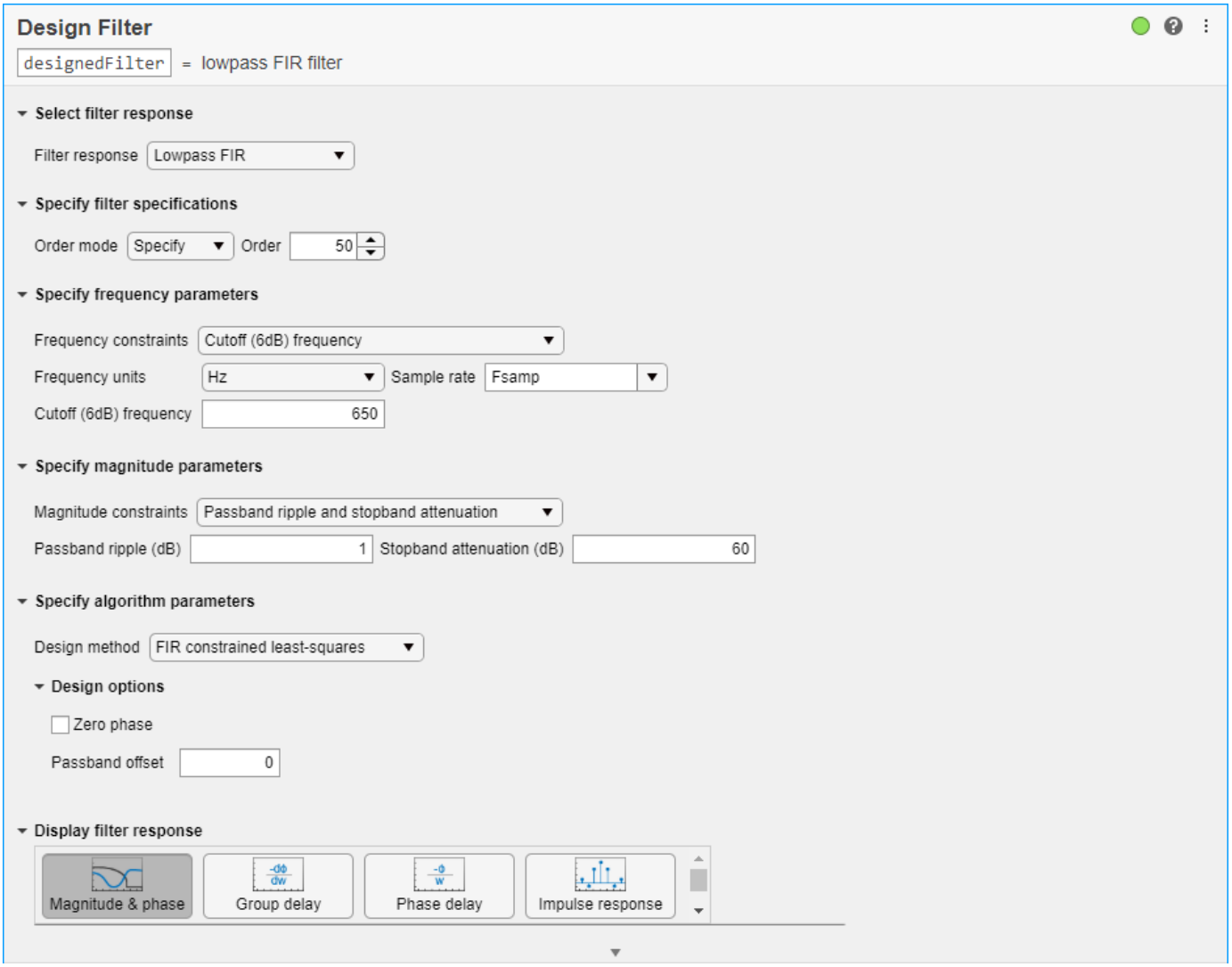
Design Filter helps you design a digital filter interactively. The task automatically generates and runs MATLAB code to design a filter using the `digitalFilter` object.

To get started, select a filter response type. The task offers controls to specify filter parameters that depend on the type of filter response and include:

- Filter order
- Frequency constraints
- Magnitude constraints
- Design method

Choose from a list of display options to visualize the generated filter response and additional filter information. For a detailed description of the filter constraints, design methods, and design method parameters, see the `designfilt` documentation.

For more information about Live Editor tasks, see “Add Interactive Tasks to a Live Script”.



Open the Task

To add the **Design Filter** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Design Filter**.
- In a code block in the script, type a relevant keyword, such as `designfilt`, `filter`, or `lowpass`. Select `Design Filter` from the suggested command completions.

Parameters

`Filter response` — Filter response type

`Lowpass FIR` | `Highpass FIR` | `Bandpass FIR` | `Bandstop FIR` | `Hilbert Transformer FIR` | `Differentiator FIR` | `Lowpass IIR` | `Highpass IIR` | `Bandpass IIR` | `Bandstop IIR`

Choose the filter response type as one of these:

- Lowpass FIR
- Lowpass IIR
- Highpass FIR
- Highpass IIR
- Bandpass FIR
- Bandpass IIR
- Bandstop FIR
- Bandstop IIR
- Hilbert Transformer FIR
- Differentiator FIR

`Filter Order` — Filter order
`Minimum` | `Specify`

Design a minimum order filter or specify a filter order. Some responses might not have a minimum order design available and will require you to specify a filter order value.

`Frequency constraints` — Frequencies at which filter exhibits desired behavior
`Passband and stopband frequencies` | `Cutoff (6dB) frequency` | `Half power (3dB) frequency` | ...

Specify the frequencies at which the designed filter exhibits a desired behavior. Available options depend on filter response type and filter order.

Note You can specify **Frequency units** as `Normalized (0 to 1)` (default) or `Hz`. If you specify frequency units in hertz, you must specify a sample rate.

`Magnitude constraints` — Filter magnitude response behavior at particular frequency ranges
`Passband ripple` | `Stopband attenuation` | ...

Choose the filter magnitude response behavior at the specified frequency ranges. Available options depend on filter response type, filter order, and frequency constraints.

`Design method` — Filter design algorithm
`Butterworth` | `Equiripple` | `FIR least-squares` | ...

Specify the algorithm used to design the filter. Available options depend on filter response type, filter order, and frequency and magnitude constraints. Some design methods have additional options available in the **Design options** section.

Note In some design cases, there are model order restrictions. If an even or odd restriction exists for the selected design method and the specified order is not valid, the task reduces the order by one.

Examples

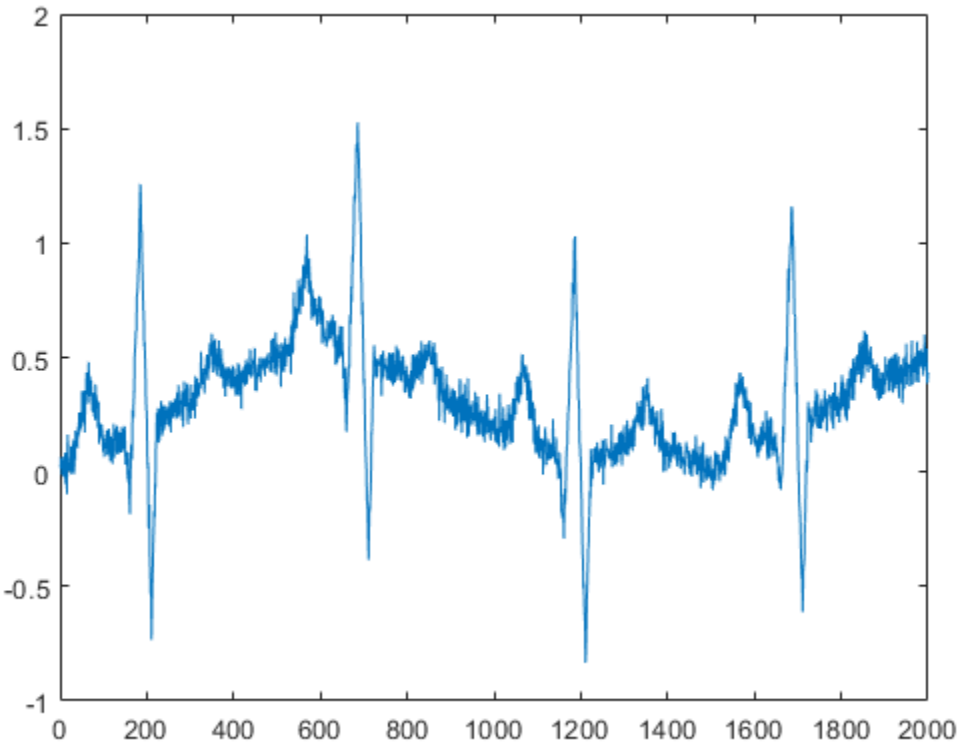
Design a Digital Filter in the Live Editor

This example shows how to use the **Design Filter** task in the Live Editor to generate code for a digital filter. The task helps you interactively design a digital filter, displays the filter response, and generates code.

Create or Load Signal

In the Live Editor, load a noisy electrocardiogram (ECG) signal into the MATLAB® workspace. Plot the data.

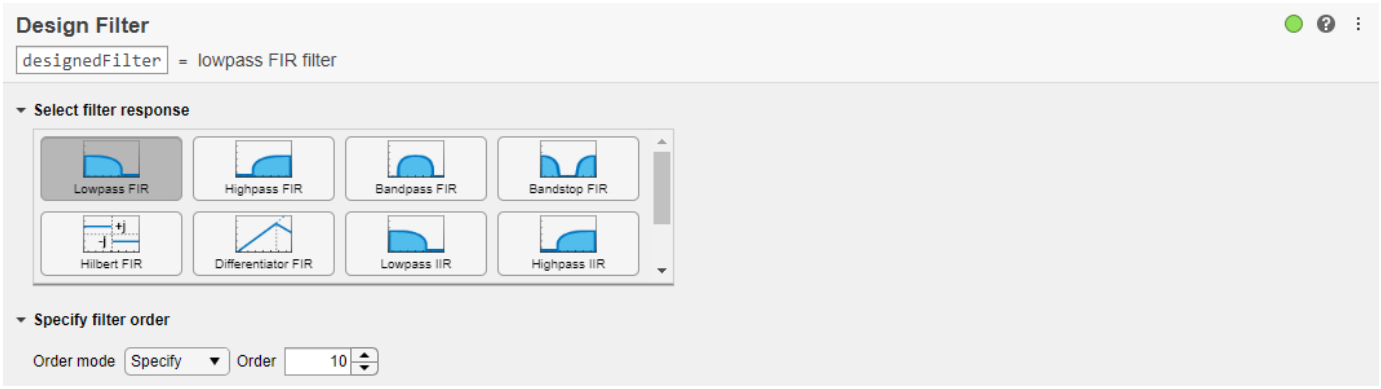
```
load noisyecg
plot(noisyECG_withTrend)
```



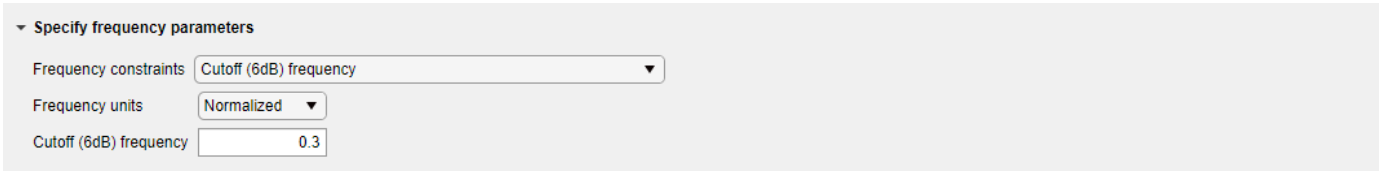
The ECG signal appears noisy. There are several sources of noise that can affect the signal including movement artifacts, high-frequency noise, and power source interference. Interactively design a filter to remove the noise from the signal. In the **Live Editor** tab, expand the **Task** list and select `Design Filter` to open the task.

Design Lowpass FIR Filter Using Kaiser Window

To remove high-frequency noise, first select a **Lowpass FIR** filter and specify the **Order** as `10`. The available options for frequency, magnitude, and algorithm parameters depend on the selected filter response type and filter order.

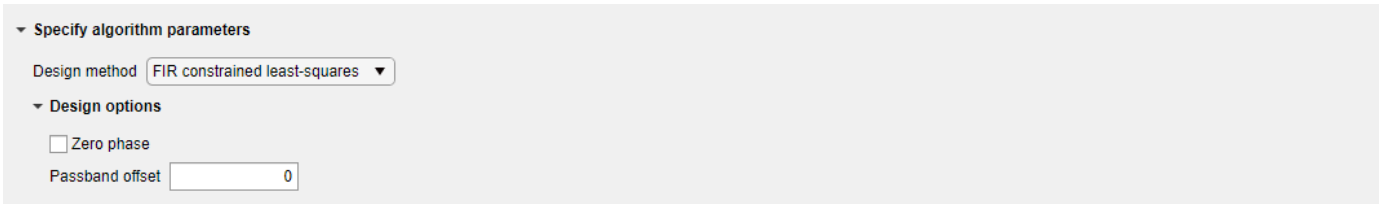


A lowpass filter removes from an input signal the unwanted frequency content above a specified threshold. In the **Specify frequency parameters** section, select `Cutoff (6dB) frequency` from the **Frequency constraints** list. When the sample rate is known, you can select `Hz` from the **Frequency units** list. A **Sample rate** option appears, and you can select a sample rate from the variables in the workspace. In this example, the sample rate is unknown, so specify a normalized cutoff frequency of `0.3` rad/sample.

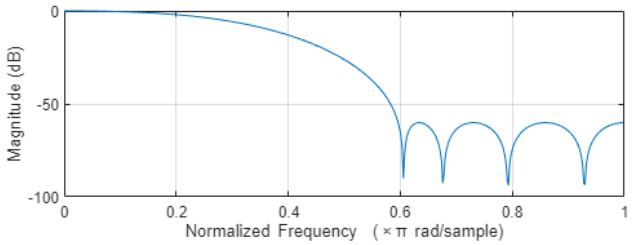


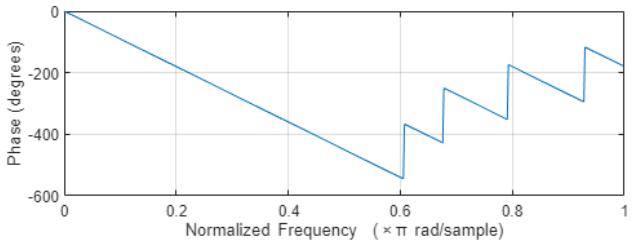
For an FIR lowpass filter, in the **Specify magnitude parameters** section, you can specify constraints to control the amount of passband ripple and stopband attenuation. Select `Passband ripple and stopband attenuation` from the **Magnitude constraints** list. Magnitude constraints and filter order can also affect the transition width of the filter.

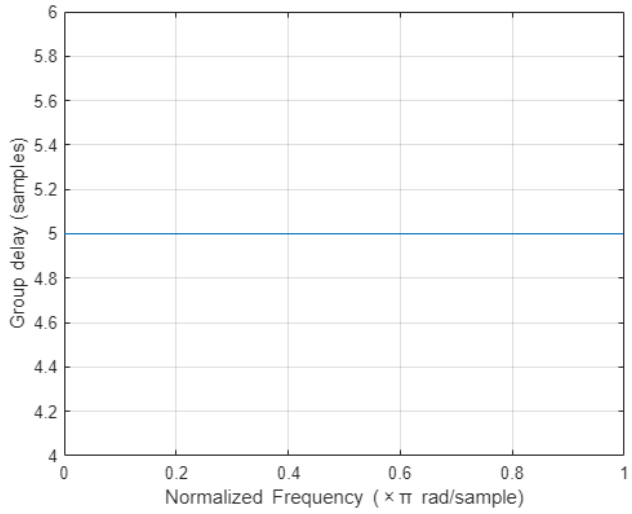
The task chooses an `FIR contrained least-squares` design algorithm by default based on the specified frequency and magnitude parameters. Leave the design options at their default settings.



In the **Display filter response** section, select `Magnitude & phase` and `Group delay` to visualize the designed filter response. In the magnitude plot, you can see the level of attenuation in the stopband is at 60 dB. The group delay plot shows a delay of 5 samples and that the filter is linear phase.





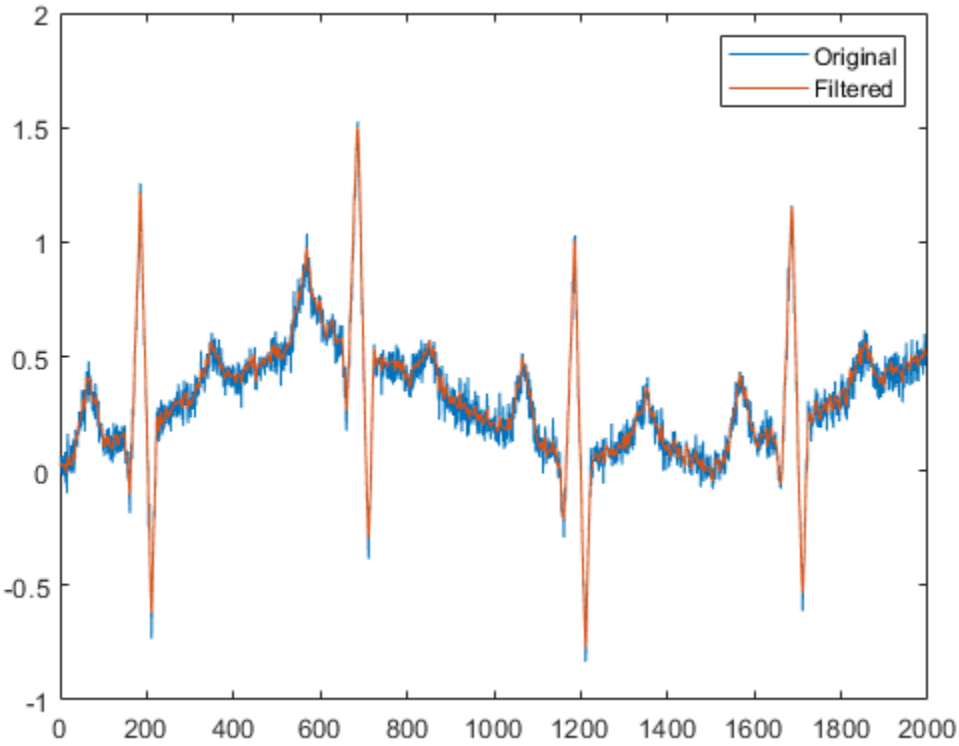


Click the arrow below the **Display filter response** section to show the generated code for the designed filter. You can copy and paste the code on the command line to edit the filter design specifications manually.

```
% Design a digital filter
designedFilter = designfilt('lowpassfir', ...
    'FilterOrder',10,'CutoffFrequency',0.3, ...
    'PassbandRipple',1,'StopbandAttenuation',60);
```

Apply the designed filter to the noisy ECG signal. Account for the delay introduced by the filter and plot the result.

```
load designedFilter
filteredECG = filter(designedFilter,noisyECG_withTrend);
delay = grpdelay(designedFilter);
mdelay = mean(delay);
filteredECG(1:mdelay) = [];

plot(noisyECG_withTrend(1:end-mdelay))
hold on
plot(filteredECG)
legend(["Original","Filtered"])
hold off
```

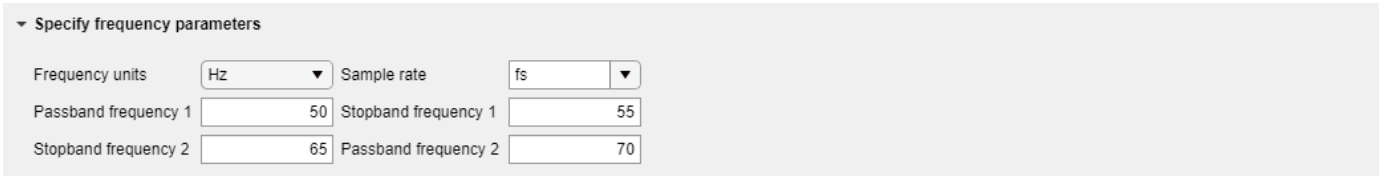


Design Equiripple Bandstop FIR Filter

A medical device like an ECG monitor can be impacted by electromagnetic interference. A power source commonly operates at a frequency of 50 Hz or 60 Hz. For this example, a 60 Hz sinusoid was added as noise to an ECG signal taken from the MIT-BIH Arrhythmia Database [1]. The sample rate is 360 Hz. To remove the noise, open the **Design Filter** task and design a minimum-order bandstop FIR filter. Change the default filter name to `bandstop60Hz`.

Specify the **Frequency units** as `Hz`. To specify a sample rate, enter a value or select a sample rate variable from the list. To appear in the list, a sample rate variable must be saved in the workspace. Create a variable, `fs`, and set it equal to `360` Hz, then select `fs` from the **Sample rate** list. Specify the passband and stopband frequency values to attenuate frequencies between 55–65 Hz for a 10 Hz notch filter centered at 60 Hz.

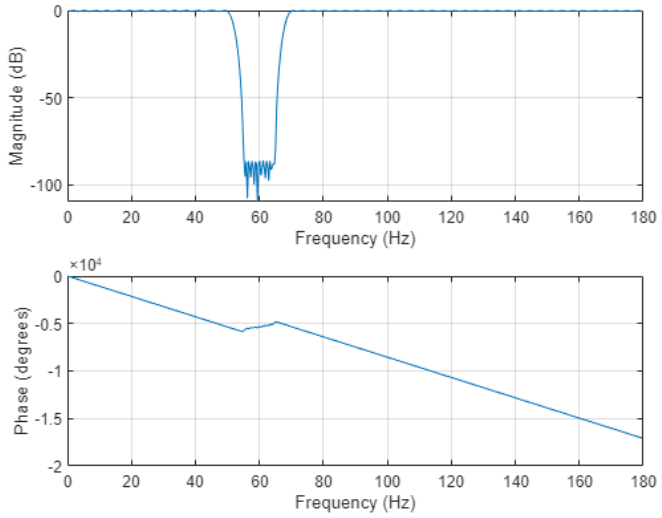
```
fs = 360;
```



Set the **Passband ripple 2 (dB)** to `0.5` and increase the **Stopband attenuation (dB)** to `80`.

The task defaults to an equiripple design method. Display the magnitude and phase responses of the filter.

```
% Visualize magnitude and phase responses
freqz(bandstop60Hz.Coefficients,1,[],fs)
```



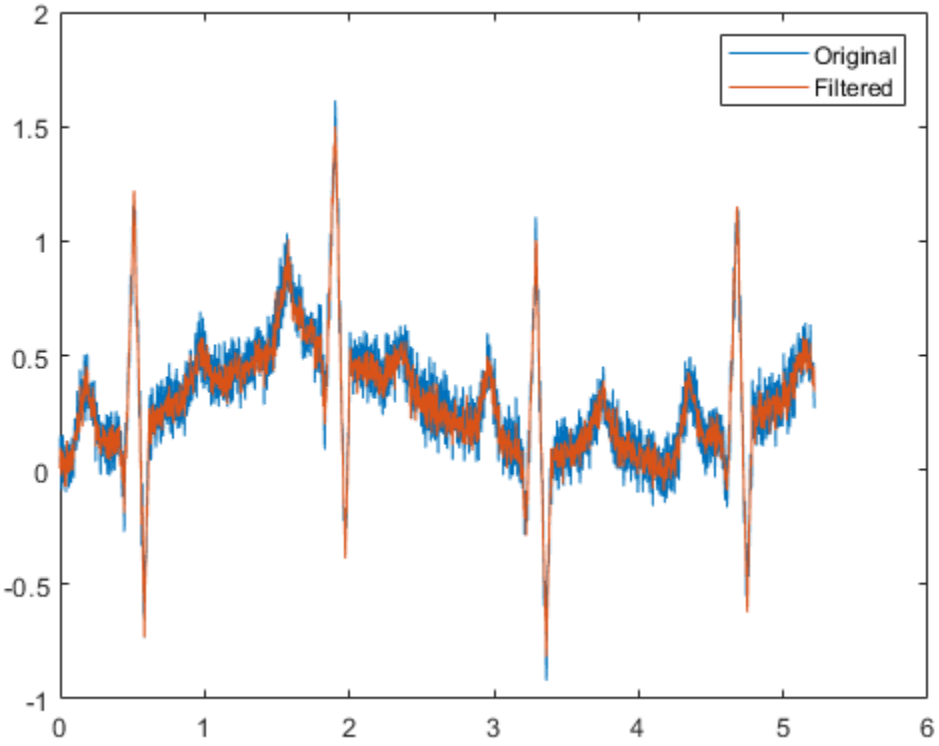
You can also select `Filter information` from the **Display filter response** section to view additional details about the designed filter.

```
% Get filter information
info(bandstop60Hz)
```

```
ans = 20×54 char array
    'FIR Digital Filter (real)                          '
    '-------------------------                          '
    'Filter Length  : 215                               '
    'Stable         : Yes                               '
    'Linear Phase   : Yes (Type 1)                      '
    '                                                   '
    'Design Method Information                          '
    'Design Algorithm : Equiripple                      '
    '                                                   '
    'Design Specifications                              '
    'Sample Rate            : 360 Hz                    '
    'Response               : Bandstop                  '
    'Specification          : Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'
    'Stopband Atten.        : 80 dB                     '
    'Second Passband Ripple : 0.5 dB                    '
    'First Passband Ripple  : 1 dB                      '
    'Second Stopband Edge   : 65 Hz                     '
    'First Stopband Edge    : 55 Hz                     '
    'First Passband Edge    : 50 Hz                     '
    'Second Passband Edge   : 70 Hz                     '
```

Load `ecg60Hz` into the workspace. The MAT-file contains the original ECG signal with added noise (`ecg60`) and the filtered signal (`ecgFilt`). Plot both signals to visualize the filter result.

```
load ecg60Hz
t = 0:1/fs:(length(ecg60)-1)/fs;
plot(t,[ecg60 ecgFilt])
legend(["Original";"Filtered"])
```



Tips

- You can toggle the autorun option by clicking the circle in the top right corner of the task window. If autorun is enabled, the current section including the task runs automatically when a change is made.

References

[1] Moody, G.B., and R.G. Mark. "The Impact of the MIT-BIH Arrhythmia Database". IEEE Eng in Med and Biol 20(3):45-50 (May-June 2001): 45-50.

See Also

Functions

`bandpass` | `bandstop` | `designfilt` | `highpass` | `lowpass`

Introduced in R2021b

dfilt

Discrete-time filter

Syntax

```
Hd = 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.

dfilt.structure	Description	Coefficient Mapping Support in realizemdl	Target Filter Block for block Method
`dfilt.delay`	Delay	Not supported	`Delay` Requires DSP System Toolbox
`dfilt.df1`	Direct-form I	Supported	`Discrete Filter`
`dfilt.df1sos`	Direct-form I, second-order sections	Supported	`Discrete Filter` 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` 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.structure	Description	Coefficient Mapping Support in realizemdl	Target Filter Block for block Method
`dfilt.df2tsos`	Direct-form II transposed, second-order sections	Supported	`Biquad Filter` 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.fftfir`	Overlap-add FIR	Not supported	`Overlap-Add FFT Filter` Requires DSP System Toolbox
`dfilt.latticeallpass`	Lattice allpass	Supported	Not supported
`dfilt.latticear`	Lattice autoregressive (AR)	Supported	`Allpole Filter` Requires DSP System Toolbox
`dfilt.latticearma`	Lattice autoregressive moving- average (ARMA)	Supported	Not supported
`dfilt.latticemamax`	Lattice moving-average (MA) for maximum phase	Supported	Not supported
`dfilt.latticemamin`	Lattice moving-average (MA) for minimum phase	Supported	`Discrete FIR Filter`
`dfilt.statespace`	State-space	Supported.	Not supported
`dfilt.scalar`	Scalar gain object	Supported	`Gain` Requires DSP System Toolbox
`dfilt.cascade`	Filters arranged in series	Supported	Target blocks depend on filter structures in the series
`dfilt.parallel`	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
`addstage`	Adds a stage to a `cascade` or `parallel` object, where a stage is a separate, modular filter. See `dfilt.cascade` and `dfilt.parallel`.
`block`	`block(Hd)` creates a Simulink filter block of the `dfilt` 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 1-318 for a mapping between the target blocks and filter structures. The `block` method can specify these properties/values: `'MapCoeffstoPorts'` indicates whether to map the filter coefficients to constant blocks connected to the generated block. Default value is `'off'`. Setting `'MapCoeffstoPorts'` to `'on'` turns on the mapping and enables the `'CoeffNames'` property, which defines the constant block parameter names. `'CoeffNames'` is a cell array. Default values are `{'Num'}` for Direct form FIR filters, `{'K'}` for lattice filters, `{'Num','Den'}` for IIR filters, and `{Num','Den','g'}` for biquad filters. Variables, defined by `'CoeffNames'`, are created in the MATLAB workspace and have the same data type as the filter's `'Arithmetic'` property. Any existing variable with the same name is overwritten. Note that you can use either `'Link2Obj'` or `'MapCoeffstoPorts'`, but not both simultaneously. `'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.
`cascade`	Returns the series combination of two `dfilt` objects. See `dfilt.cascade`.

Method	Description
`coeffs`	Returns the filter coefficients in a structure containing fields that use the same property names as those in the original `dfilt`.
`convert`	Converts a `dfilt` object from one filter structure to another filter structure.
`fcfwrite`	Writes a filter coefficient ASCII file. The file can contain a single filter or a vector of objects. Default file name is `untitled.fcf`. `fcfwrite(Hd,filename)` writes to a disk file named `filename` in the current working directory. The `.fcf` extension is added automatically. `fcfwrite(...,fmt)` writes the coefficients in the format `fmt`, where `fmt` can be one of the following: `'hex'` for hexadecimal `'dec'` for decimal `'bin'` for binary representation.
`fftcoeffs`	Returns the frequency-domain coefficients used when filtering with a `dfilt.fftfir`.
`filter`	Performs filtering using the `dfilt` object. `y = filter(Hd,x)` filters `x` using the `Hd` filter and returns the filtered data in `y`. See “Using Filter States” on page 1-325 for information on using initial conditions. If `x` is a matrix, each column is filtered as an independent channel. If `x` is a multidimensional array, `filter` operates on the first nonsingleton dimension. `y = filter(Hd,x,dim)` operates along the dimension `dim`. If `x` is a vector or matrix and `dim` is 1, every column of `x` is a channel. If `dim` is 2, every row is a channel.
`firtype`	Returns the type (1-4) of a linear phase FIR filter.
`freqz`	Plots the frequency response in **FVTool**. Note that unlike the `freqz` function, this `dfilt freqz` method has a default length of 8192.
`grpdelay`	Plots the group delay in **FVTool**.
`impz`	Plots the impulse response in **FVTool**.
`impzlength`	Returns the length of the impulse response.
`info`	Displays brief `dfilt` information, such as filter structure, length, stability, linear phase, and, when appropriate, lattice and ladder length. To display detailed information about the design method, options, etc, use `info(Hd, 'long')`. The default display is `'short'`. For multistage filters (`cascade` and `parallel`), use `info(Hd.Stage(x))`, where `x` is the stage number, to see information about that stage.
`isallpass`	Returns a logical `1` (i.e., true) if the `dfilt` object in an allpass filter or a logical `0` (i.e., false) if it is not.

Method	Description
`iscascade`	Returns a logical `1` if the `dfilt` object is cascaded or a logical `0` if it is not.
`isfir`	Returns a logical `1` if the `dfilt` object has finite impulse response (FIR) or a logical `0` if it does not.
`islinphase`	Returns a logical `1` if the `dfilt` object is linear phase or a logical `0` if it is not.
`ismaxphase`	Returns a logical `1` if the `dfilt` object is maximum-phase or a logical `0` if it is not.
`isminphase`	Returns a logical `1` if the `dfilt` object is minimum-phase or a logical `0` if it is not.
`isparallel`	Returns a logical `1` if the `dfilt` object has parallel stages or a logical `0` if it does not.
`isreal`	Returns a logical `1` if the `dfilt` object has real-valued coefficients or a logical 0 if it does not.
`isscalar`	Returns a logical `1` if the `dfilt` object is a scalar or a logical `0` if it is not scalar.
`issos`	Returns a logical `1` if the `dfilt` object has second-order sections or a logical `0` if it does not.
`isstable`	Returns a logical `1` if the `dfilt` object is stable or a logical `0` if it are not.
`nsections`	Returns the number of sections in a second-order sections filter. If a multistage filter contains stages with multiple sections, using `nsections` returns the total number of sections in all the stages (a stage with a single section returns 1).
`nstages`	Returns the number of stages of the filter, where a stage is a separate, modular filter.
`nstates`	Returns the number of states for an object.
`order`	Returns the filter order. If `Hd` is a single-stage filter, the order is given by the number of delays needed for a minimum realization of the filter. If `Hd` has multiple stages, the order is given by the number of delays needed for a minimum realization of the overall filter.
`parallel`	Returns the parallel combination of two `dfilt` filters. See `dfilt.parallel`.
`phasez`	Plots the phase response in **FVTool**.

Method	Description
`realizemdl`	(Available only with Simulink software.) `realizemdl(Hd)` creates a Simulink model containing a subsystem block realization of your `dfilt`. `realizemdl(Hd,p1,v1,p2,v2,...)` creates the block using the properties `p1`, `p2`,... and values `v1`, `v2`,.. specified. The following properties are available: `'Blockname'` specifies the name of the block. The default value is `'Filter'`. `'Destination'` specifies whether to add the block to a current Simulink model, create a new model, or place the block in an existing subsystem in your model. Valid values are `'current'`, `'new'`, or the name of an existing subsystem in your model. Default value is `'current'`. `'OverwriteBlock'` specifies whether to overwrite an existing block that was created by `realizemdl` or create a new block. Valid values are `'on'` and `'off'` and the default is `'off'`. Note that only blocks created by `realizemdl` are overwritten. The following properties optimize the block structure. Specifying `'on'` turns the optimization on and `'off'` creates the block without optimization. The default for each of the following is `'on'`. `'OptimizeZeros'` removes zero-gain blocks. `'OptimizeOnes'` replaces unity-gain blocks with a direct connection. `'OptimizeNegOnes'` replaces negative unity-gain blocks with a sign change at the nearest summation block. `'OptimizeDelayChains'` replaces cascaded chains of delay block with a single integer delay block set to the appropriate delay.
`removestage`	Removes a stage from a cascade or parallel `dfilt`. See `dfilt.cascade` and `dfilt.parallel`.
`setstage`	Overwrites a stage of a cascade or parallel `dfilt`. See `dfilt.cascade` and `dfilt.parallel`.

Method	Description
sos	Converts the dfilt to a second-order sections dfilt. If Hd has a single section, the returned filter has the same class. sos(Hd,flag) specifies the ordering of the second-order sections. If flag='UP', the first row contains the poles closest to the origin, and the last row contains the poles closest to the unit circle. If flag='down', the sections are ordered in the opposite direction. The zeros are always paired with the poles closest to them. sos(Hd,flag,scale) specifies the scaling of the gain and the numerator coefficients of all second-order sections. scale can be 'none', 'inf' (infinity-norm) or 'two' (2-norm). Using infinity-norm scaling with up ordering minimizes the probability of overflow in the realization. Using 2-norm scaling with down ordering minimizes the peak round-off noise.
ss	Converts the dfilt to state-space. To see the separate A,B,C,D matrices for the state-space model, use [A,B,C,D]=ss(Hd).
stepz	Plots the step response in **FVTool**. stepz(Hd,n) computes the first n samples of the step response. stepz(Hd,n,Fs) separates the time samples by T = 1/Fs, where Fs is assumed to be in Hz.
sysobj	Converts the dfilt 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.
tf	Converts the dfilt to a transfer function.
zerophase	Plots the zero-phase response in **FVTool**.
zpk	Converts the dfilt to zeros-pole-gain form.
zplane	Plots a pole-zero plot in **FVTool**.

For more information on each method, use the syntax help 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_name');
```

where `structure_name` is any valid structure name in single quotes. 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

Apps
Signal Analyzer | **Filter Designer**

Functions
`filter` | `freqz` | `grpdelay` | `impz` | `step` | `tf` | `zpk` | `zplane`

Introduced before R2006a

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

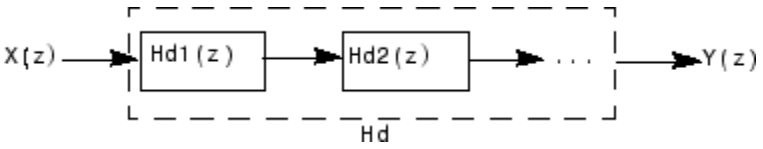
```
addstage(Hd,Hd1)
```

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

Signal Analyzer | `designfilt`

Introduced before R2006a

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

Signal Analyzer | `designfilt`

Introduced before R2006a

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.

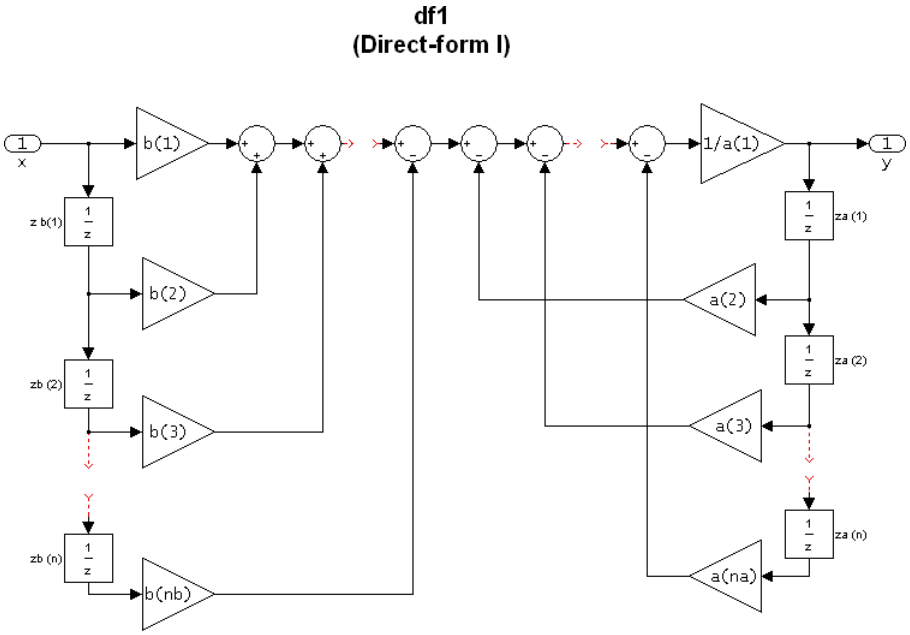


Image of direct form one filter diagram

To display the filter states, use this code to access the `filtstates` object.

```
Hs = Hd.states       % Where Hd is the dfilt.df1 object and
double (Hs)          % Hs is the filtstates object
```

The vector is

$$\begin{bmatrix} zb(1) \\ zb(2) \\ \cdots \\ zb(n) \\ za(1) \\ za(2) \\ \cdots \\ za(n) \end{bmatrix}$$

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

Signal Analyzer | `designfilt`

Introduced before R2006a

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

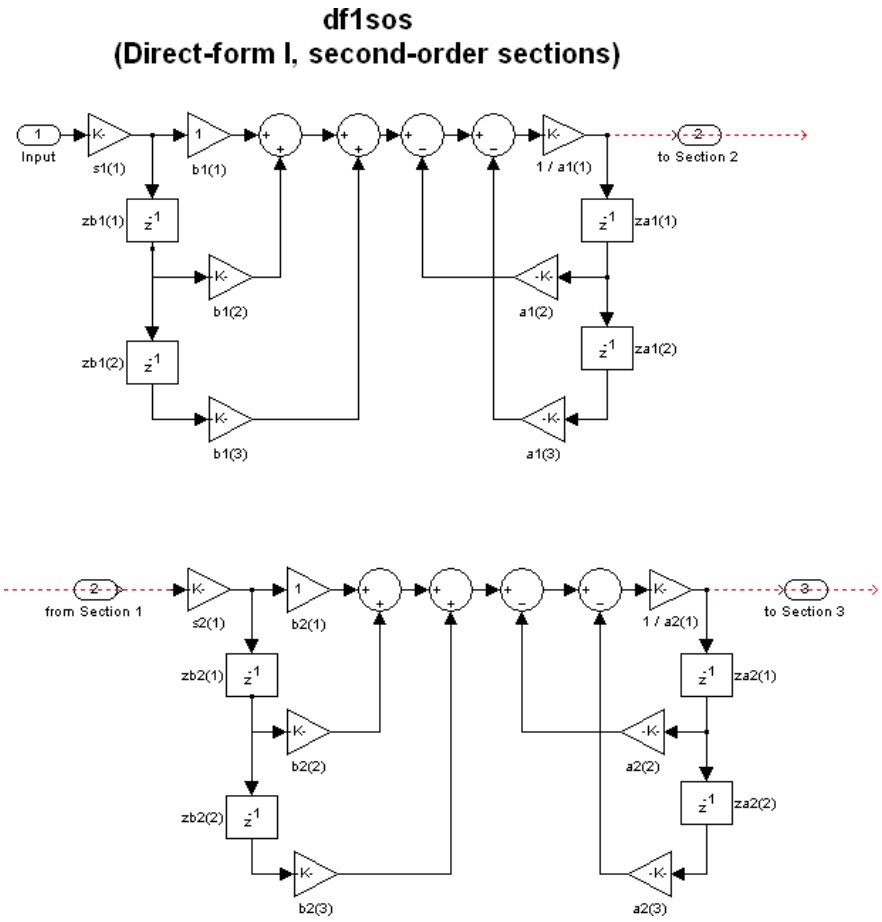
`Hd = dfilt.df1sos(s)` returns a discrete-time, second-order section, direct-form I filter, `Hd`, with coefficients given in the `s` matrix. The filter states for this object are stored in a `filtstates` object.

`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.



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

Signal Analyzer | `designfilt`

Introduced before R2006a

dfilt.df1t

Discrete-time, direct-form I transposed filter

Syntax

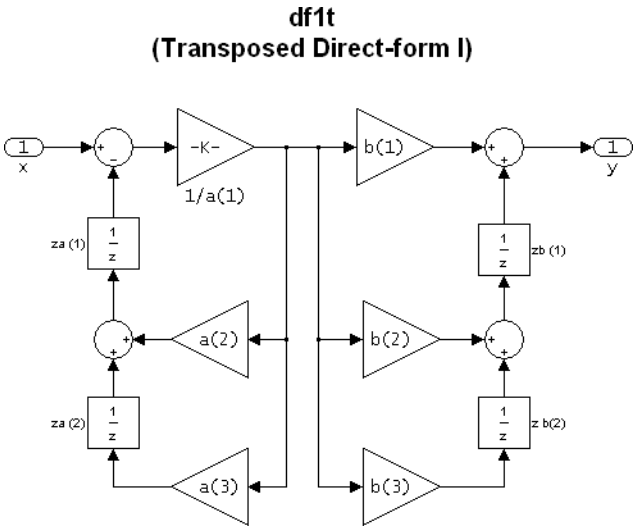
```
Hd = dfilt.df1t(b,a)
Hd = dfilt.df1t
```

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

Signal Analyzer | `designfilt`

Introduced before R2006a

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

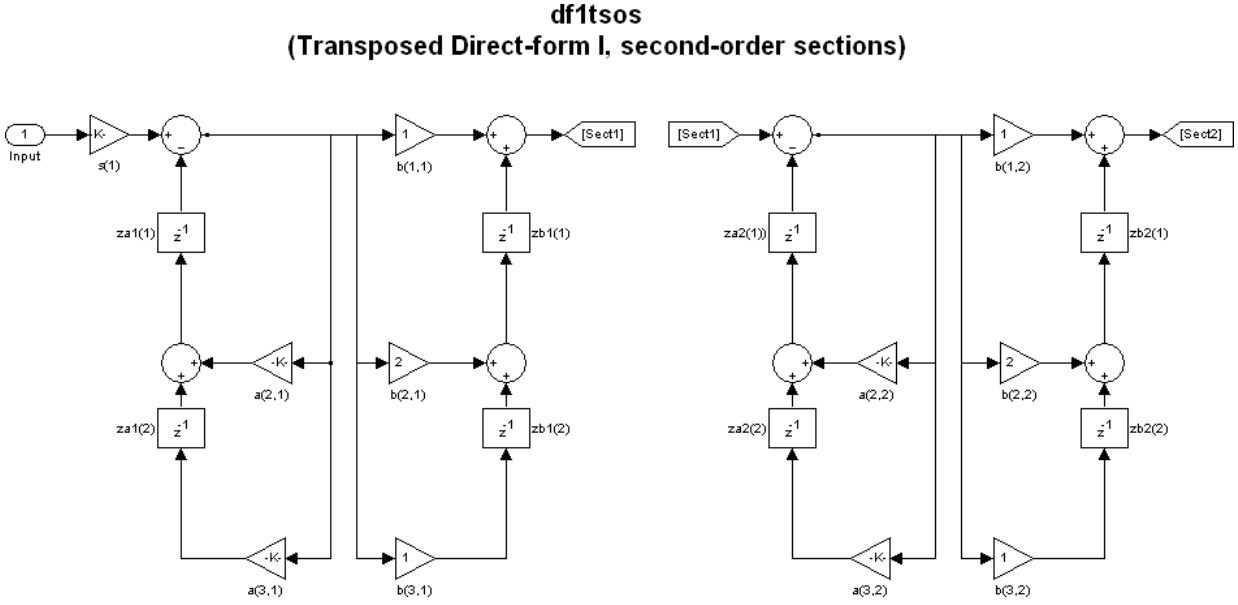
`Hd = dfilt.df1tsos(s)` returns a discrete-time, second-order section, direct-form I, transposed filter, `Hd`, with coefficients given in the `s` matrix. The filter states for this object are stored in a `filtstates` object.

`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.



To display the filter states, use this code to access the `filtstates` object.

```
Hs = Hd.states     % Where Hd is the dfilt.df1 object and
double (Hs)        % Hs is the filtstates object
```

The matrix is

$$\begin{pmatrix} zb1(1) & zb2(1) \\ zb1(2) & zb2(2) \\ za1(1) & za2(1) \\ za1(2) & za2(2) \end{pmatrix}$$

Examples

Specify a second-order sections, direct-form I, transposed discrete-time filter with coefficients from a sixth order, lowpass, elliptical filter using the following code:

```
[z,p,k] = ellip(6,1,60,.4);    % Obtain filter coefficients
[s,g] = zp2sos(z,p,k);         % Convert to SOS
Hd = dfilt.df1tsos(s,g)
```

See Also

Signal Analyzer | `designfilt`

Introduced before R2006a

dfilt.df2

Discrete-time, direct-form II filter

Syntax

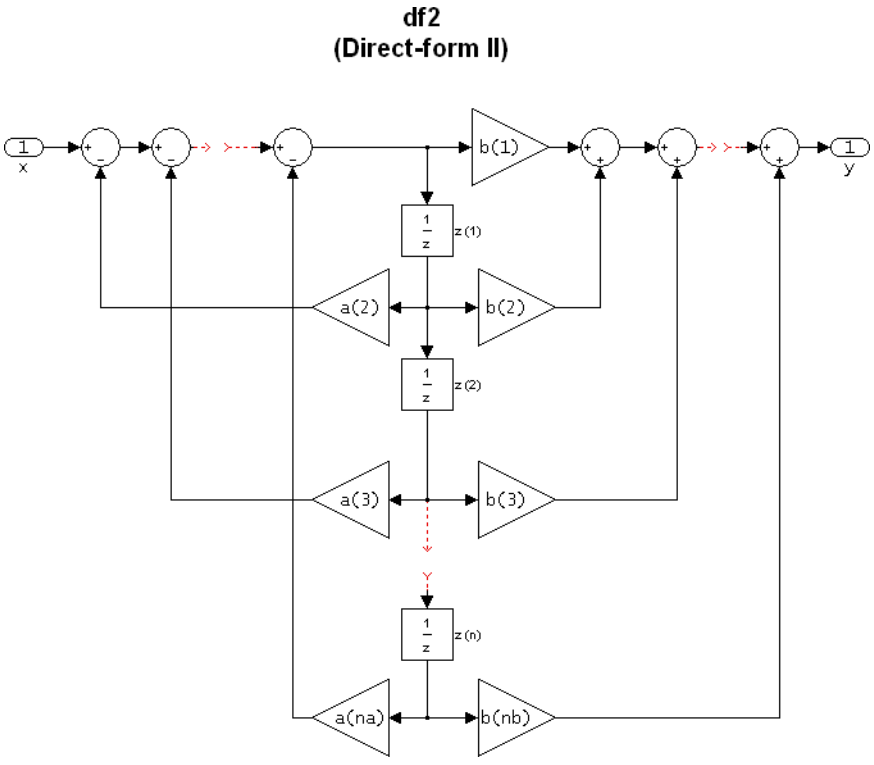
```
Hd = dfilt.df2(b,a)
Hd = dfilt.df2
```

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

Signal Analyzer | `designfilt`

Introduced before R2006a

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

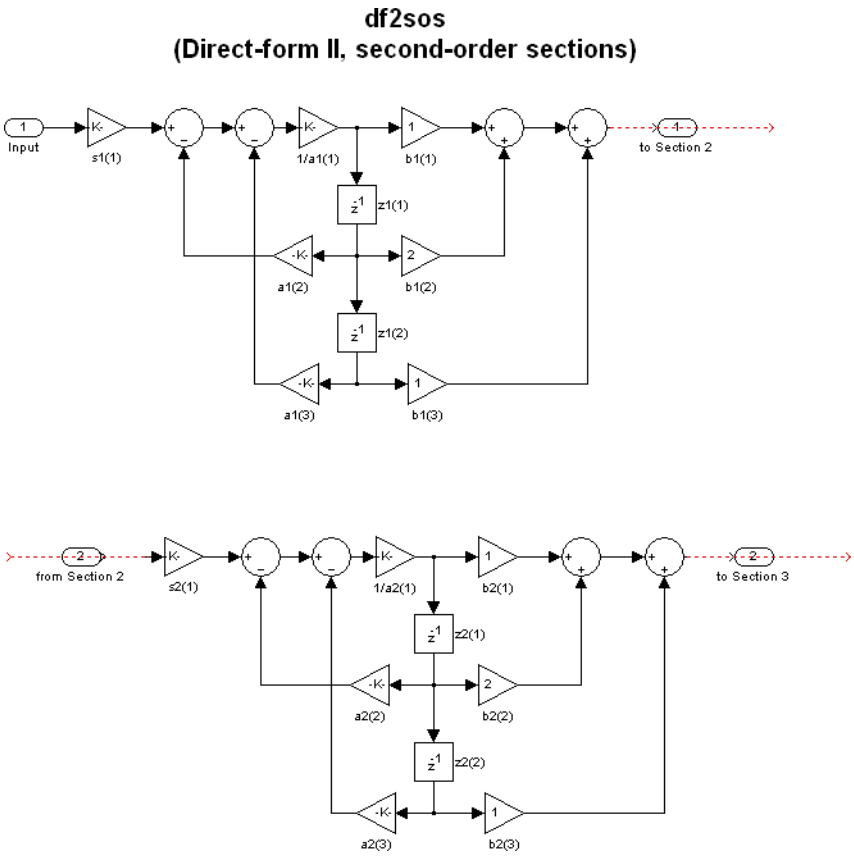
`Hd = dfilt.df2sos(s)` returns a discrete-time, second-order section, direct-form II filter, `Hd`, with coefficients given in the `s` matrix.

`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.



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

Signal Analyzer | `designfilt`

Introduced before R2006a

dfilt.df2t

Discrete-time, direct-form II transposed filter

Syntax

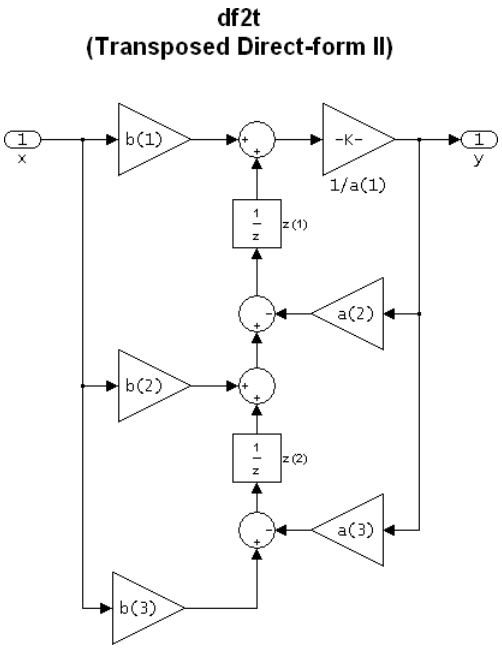
```
Hd = dfilt.df2t(b,a)
Hd = dfilt.df2t
```

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

Signal Analyzer | `designfilt`

Introduced before R2006a

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

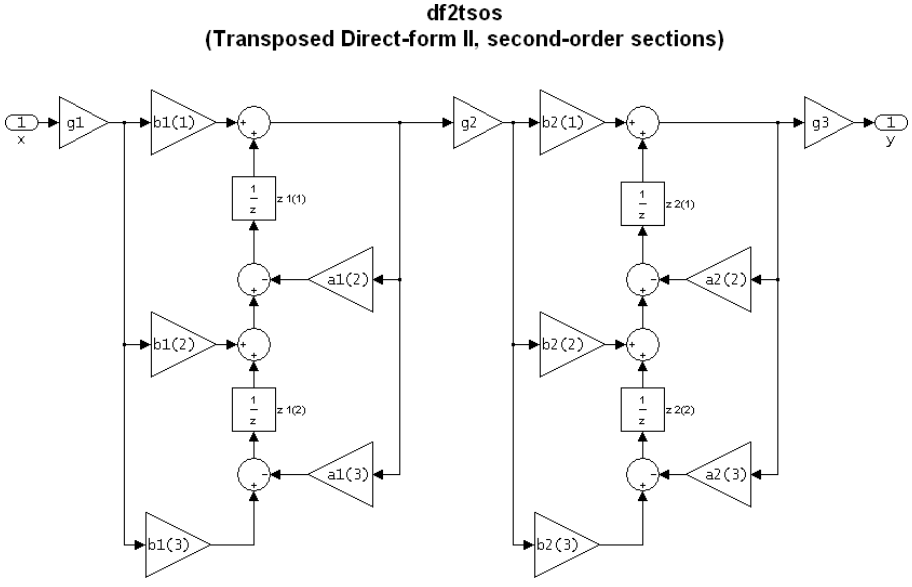
`Hd = dfilt.df2tsos(s)` returns a discrete-time, second-order section, direct-form II, transposed filter, `Hd`, with coefficients given in the `s` matrix.

`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

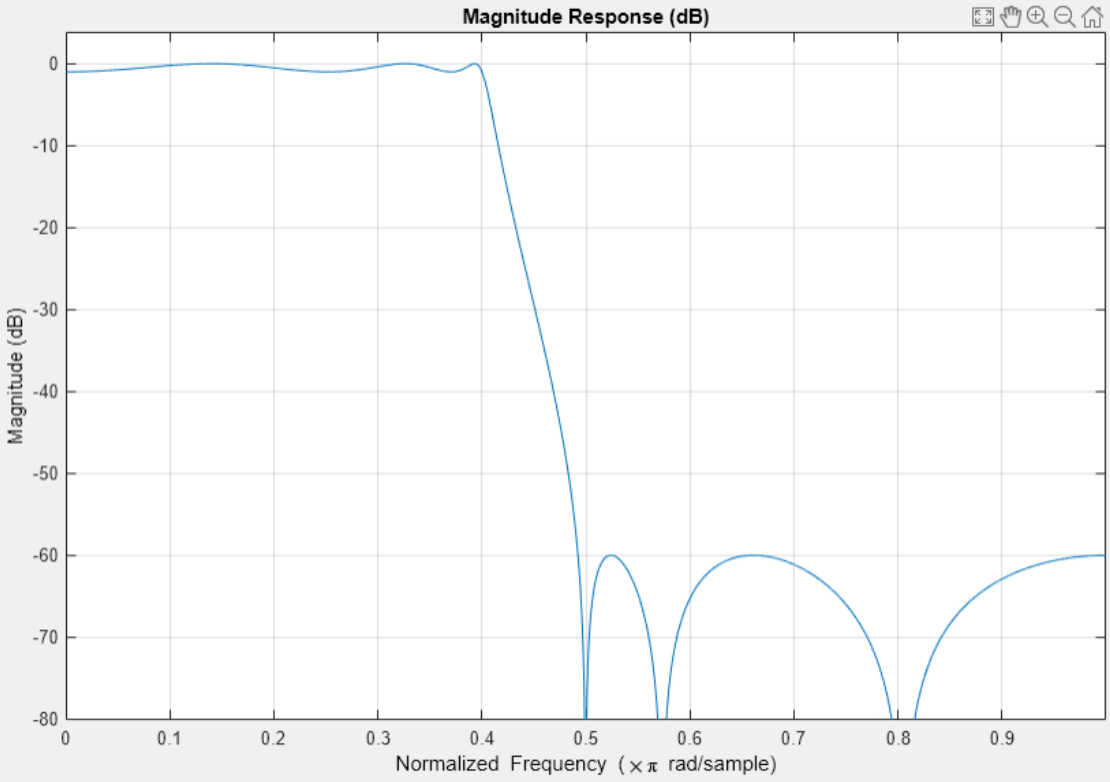
$$\begin{pmatrix} z1(1) & z2(1) \\ z1(2) & z2(2) \end{pmatrix}$$

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 0.4π 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

Signal Analyzer | `designfilt`

Introduced before R2006a

dfilt.dfasymfir

Discrete-time, direct-form antisymmetric FIR filter

Syntax

```
Hd = dfilt.dfasymfir(b)
Hd = dfilt.dfasymfir
```

Description

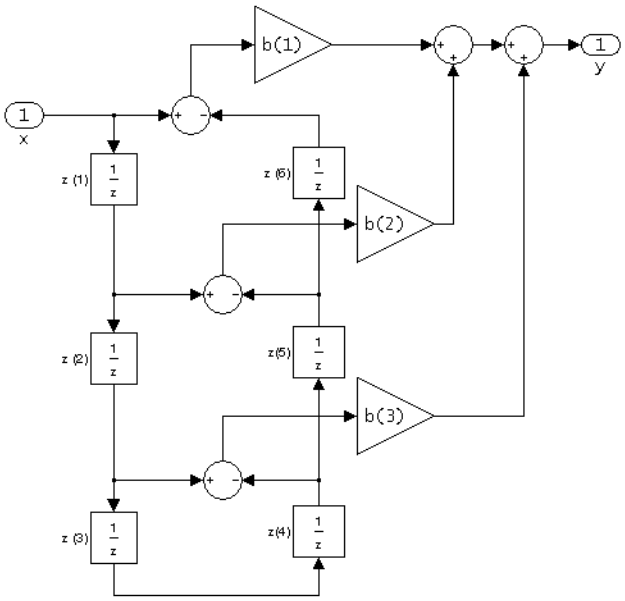
`Hd = dfilt.dfasymfir(b)` returns a discrete-time, direct-form, antisymmetric FIR filter, `Hd`, with numerator coefficients `b`.

`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.

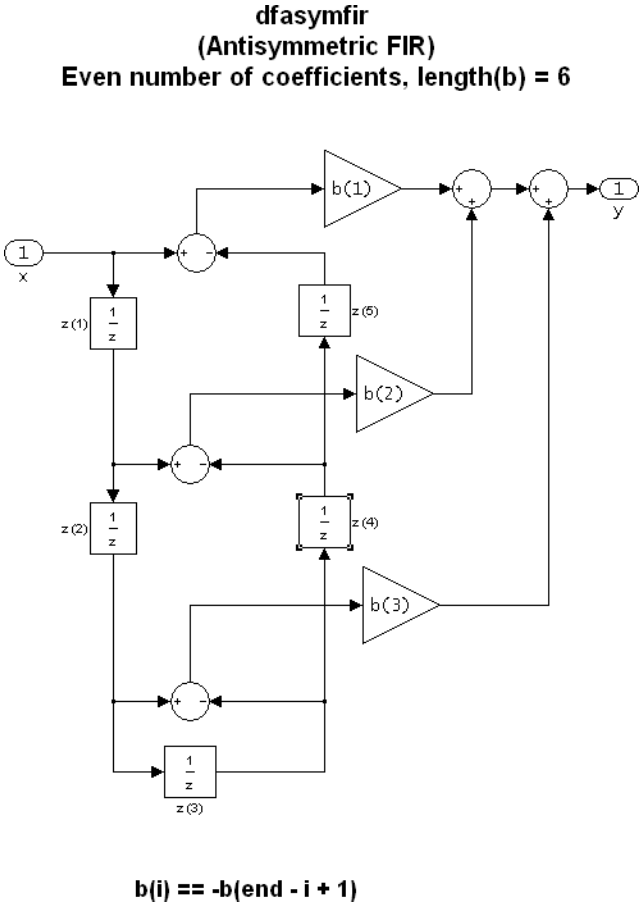
Note Only the first half of vector `b` is used because the second half is assumed to be antisymmetric. In the figure below for an odd number of coefficients, `b(3) = 0`, `b(4) = –b(2)` and `b(5) = –b(1)`, and in the next figure for an even number of coefficients, `b(4) = –b(3)`, `b(5) = –b(2)`, and `b(6) = –b(1)`.

---

dfasymfir
(Antisymmetric FIR)
Even order
Odd number of coefficients, length(b) = 7



Note that antisymmetry is defined as
b(i) == -b(end - i + 1)
so that the middle coefficient is zero for odd length
b((end+1)/2) = 0



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

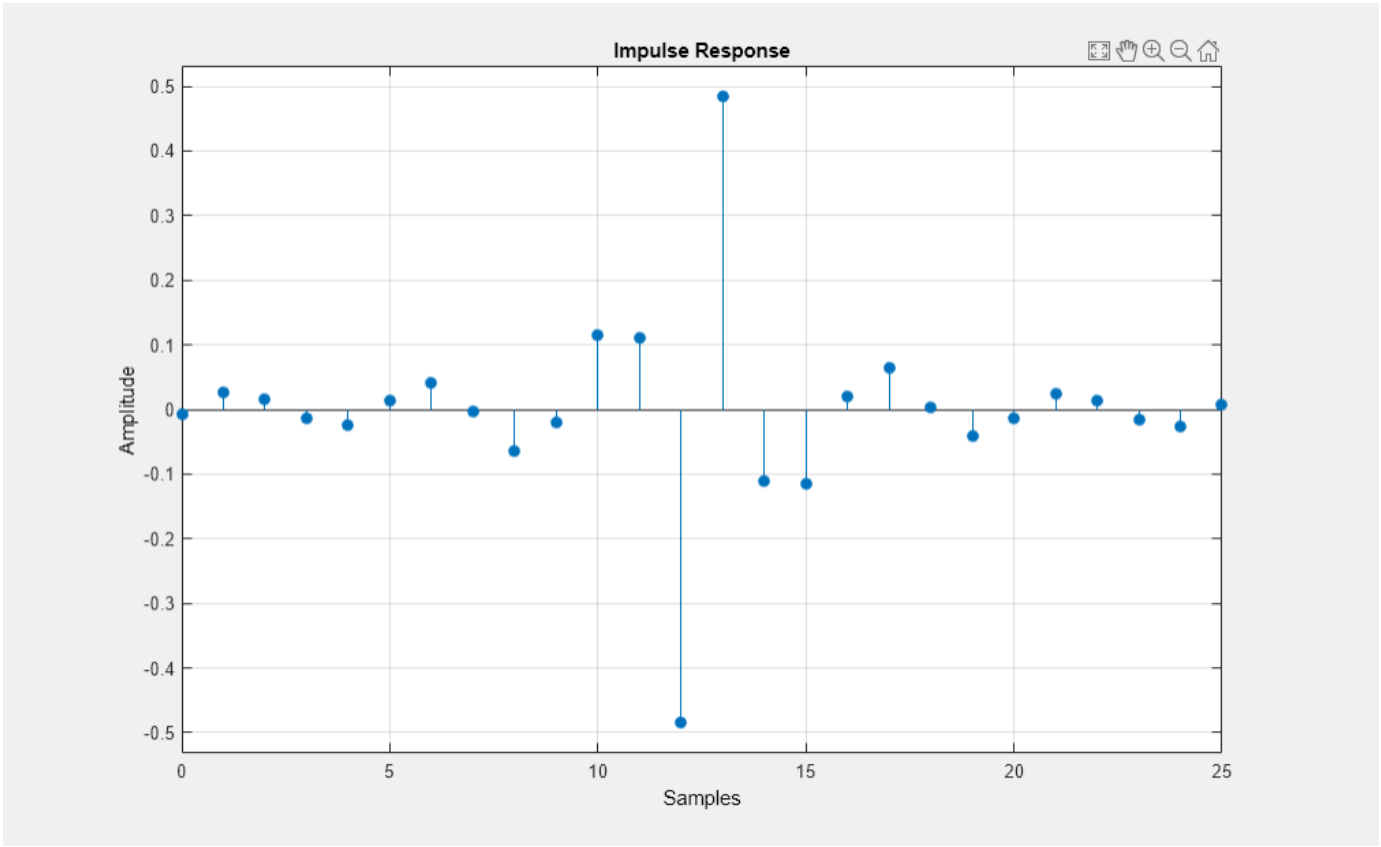
Odd-Order Antisymmetric FIR Filter Structure

Create a Type-4 25th-order highpass direct-form antisymmetric FIR filter structure for a `dfilt` object.

```
Num_coeffs = firpm(25,[0 .4 .5 1],[0 0 1 1],'h');
Hd = dfilt.dfasymfir(Num_coeffs);
```

Display the impulse response of the filter.

```
impz(Hd)
```



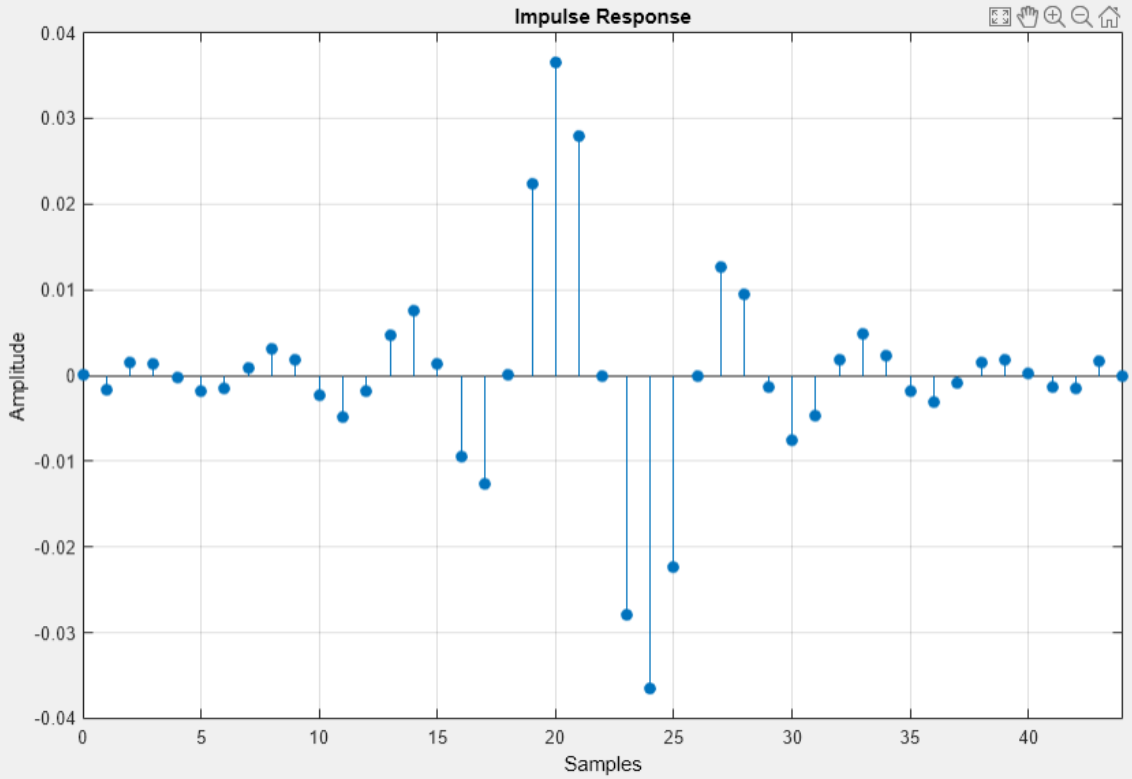
Even-Order Antisymmetric FIR Filter Structure

Create a 44th-order lowpass direct-form antisymmetric FIR differentiator filter structure for a `dfilt` object.

```
Num_coeffs = firpm(44,[0 .3 .4 1],[0 .2 0 0],'differentiator');
Hd = dfilt.dfasymfir(Num_coeffs);
```

Display the impulse response of the filter.

```
impz(Hd)
```



See Also

Signal Analyzer | `designfilt`

Introduced before R2006a

dfilt.dffir

Discrete-time, direct-form, FIR filter

Syntax

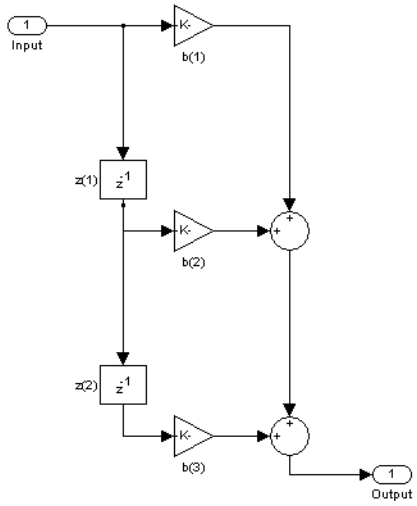
```
Hd = dfilt.dffir(b)
Hd = dfilt.dffir
```

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 30th order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.dffir(b)
```

See Also

Signal Analyzer | `designfilt`

Introduced before R2006a

dfilt.dffirt

Discrete-time, direct-form FIR transposed filter

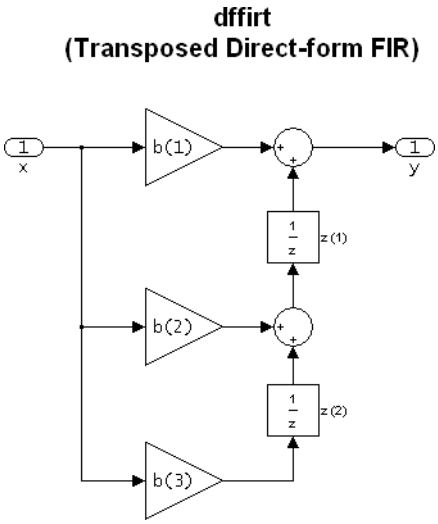
Syntax

```
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 30th order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.dffirt(b)
```

See Also

Signal Analyzer | `designfilt`

Introduced before R2006a

dfilt.dfsymfir

Discrete-time, direct-form symmetric FIR filter

Syntax

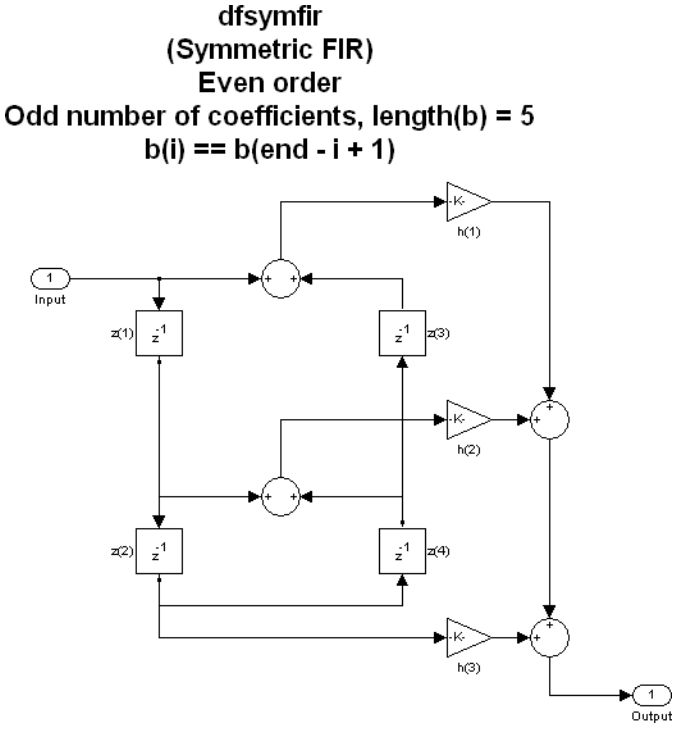
```
Hd = dfilt.dfsymfir(b)
Hd = dfilt.dfsymfir
```

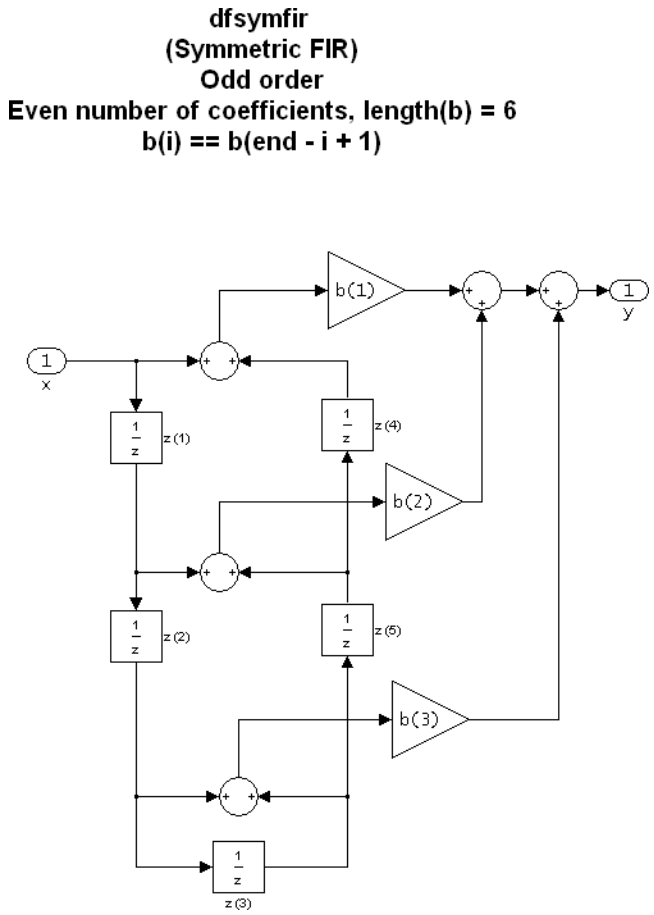
Description

`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)`.





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

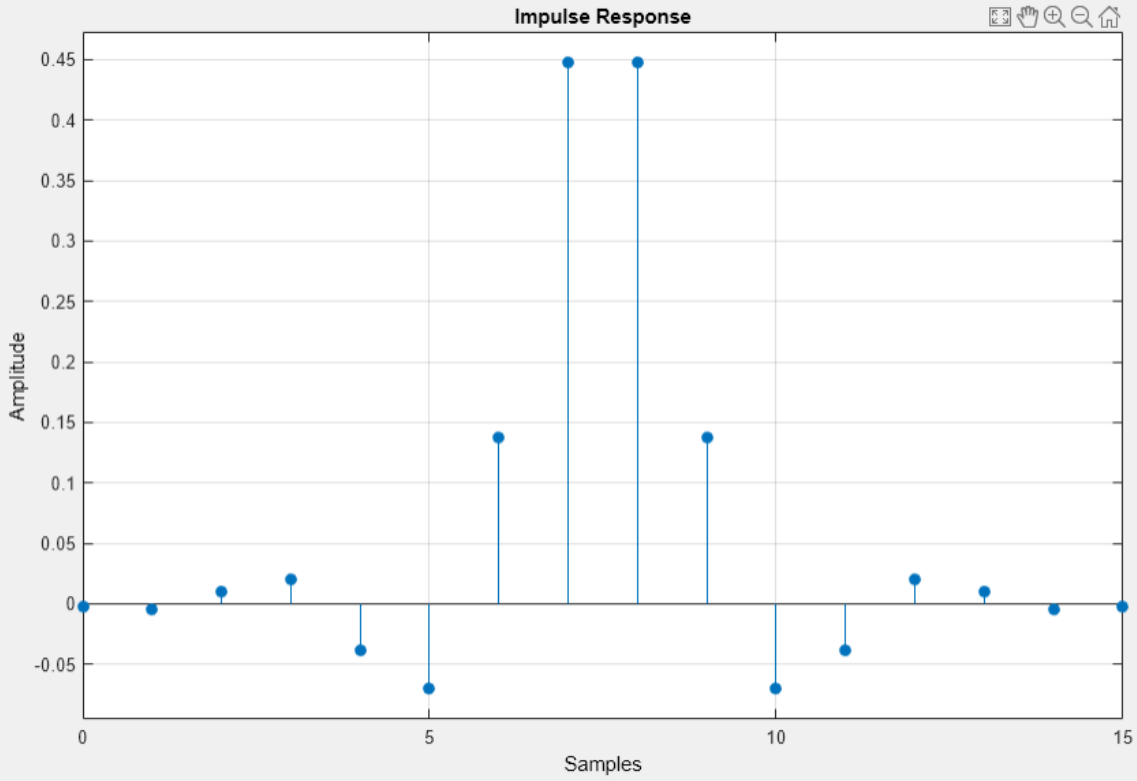
Odd-Order Symmetric FIR Filter Structure

Create a Type-2 15th-order direct-form symmetric FIR filter structure for a `dfilt` object.

```
Num_coeffs = fir1(15,0.5);
Hd = dfilt.dfsymfir(Num_coeffs);
```

Display the impulse response of the filter.

```
impz(Hd)
```



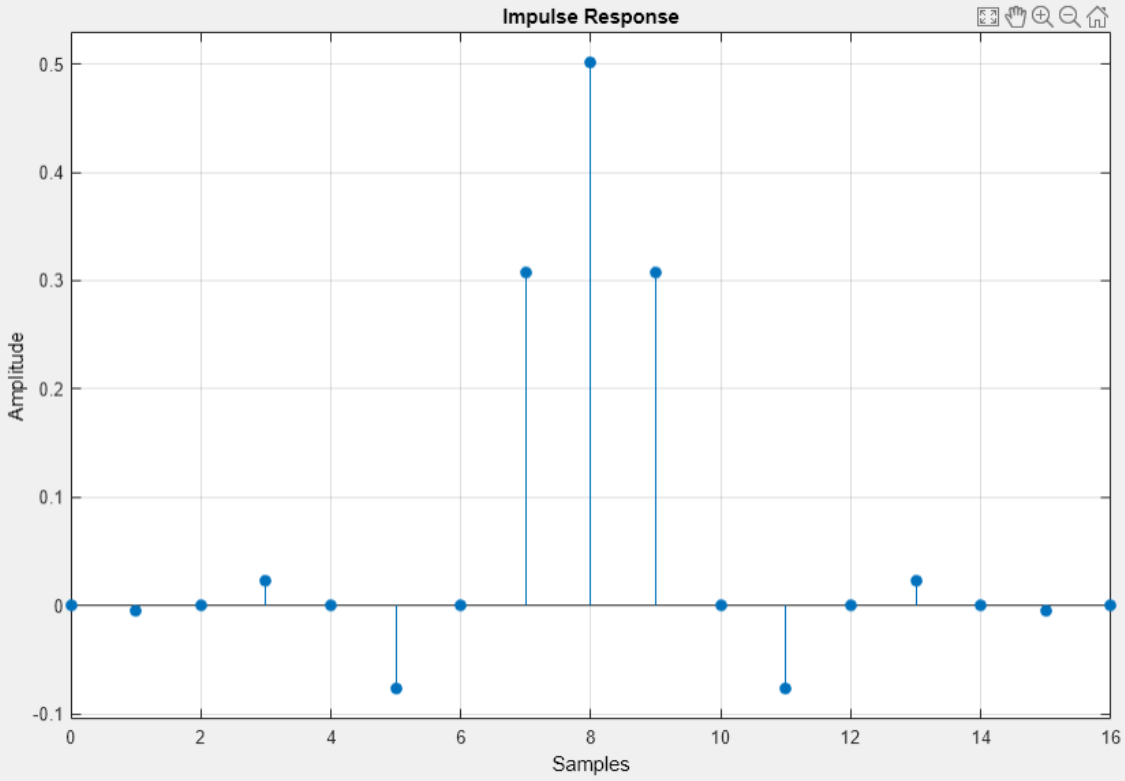
Even-Order Symmetric FIR Filter Structure

Create a Type-1 16th-order direct-form symmetric FIR filter structure for a `dfilt` object.

```
Num_coeffs = fir1(16,0.5);
Hd = dfilt.dfsymfir(Num_coeffs);
```

Display the impulse response of the filter.

```
impz(Hd)
```



See Also

Signal Analyzer | `designfilt`

Introduced before R2006a

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.

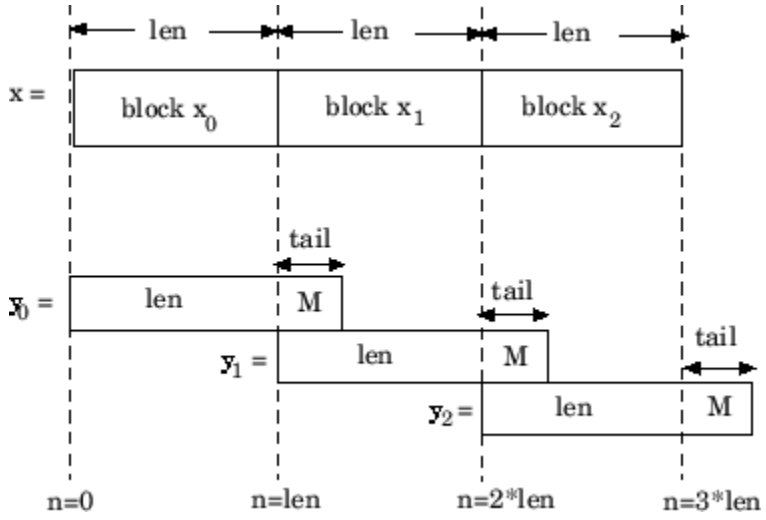
`Hd = dfilt.fftfir(b,len)` returns a discrete-time, FFT, FIR filter, `Hd`, with numerator coefficients, `b` and block length, `len`. The block length is the number of input points to use for each overlap-add computation.

`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.fftfir` are the tails of the final convolution.

Examples

Create an FFT FIR discrete-time filter with coefficients from a 30th order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.fftfir(b)
```

To view the frequency domain coefficients used in the filtering, use the following command.

```
freq_coeffs = fftcoeffs(Hd);
```

See Also

Signal Analyzer | `designfilt`

Introduced before R2006a

dfilt.latticeallpass

Discrete-time, lattice allpass filter

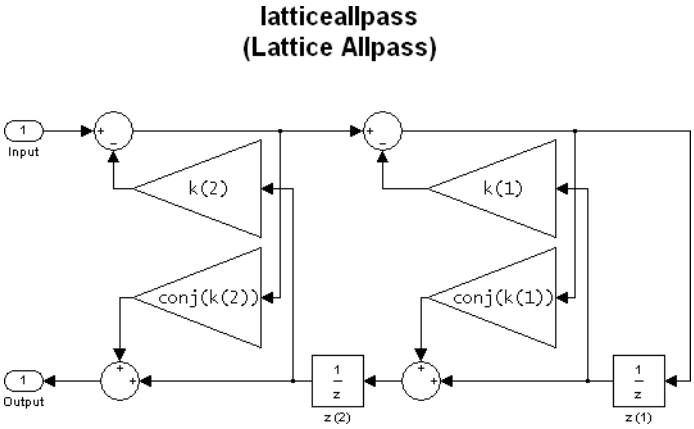
Syntax

```
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

Signal Analyzer | `designfilt`

Introduced before R2006a

dfilt.latticear

Discrete-time, lattice, autoregressive filter

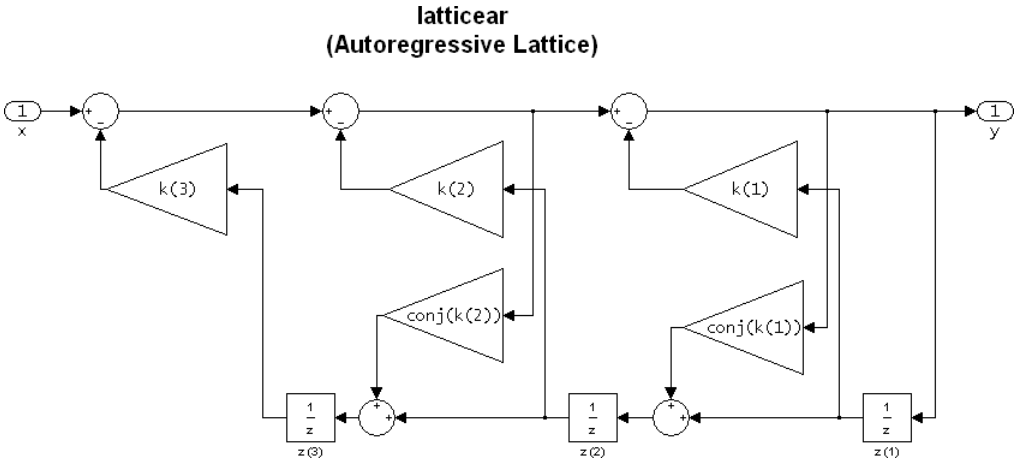
Syntax

```
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

Signal Analyzer | `designfilt`

Introduced before R2006a

dfilt.latticearma

Discrete-time, lattice, autoregressive, moving-average filter

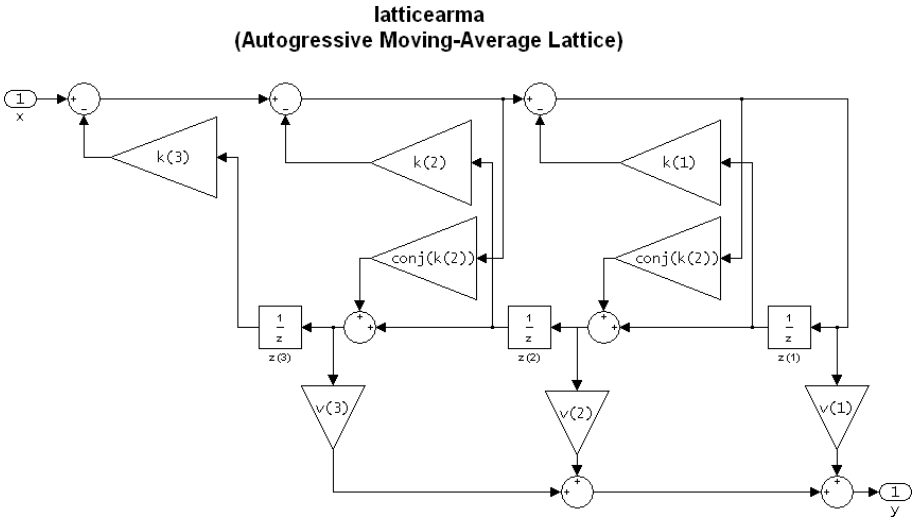
Syntax

```
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

Signal Analyzer | `designfilt`

Introduced before R2006a

dfilt.latticemamax

Discrete-time, lattice, moving-average filter

Syntax

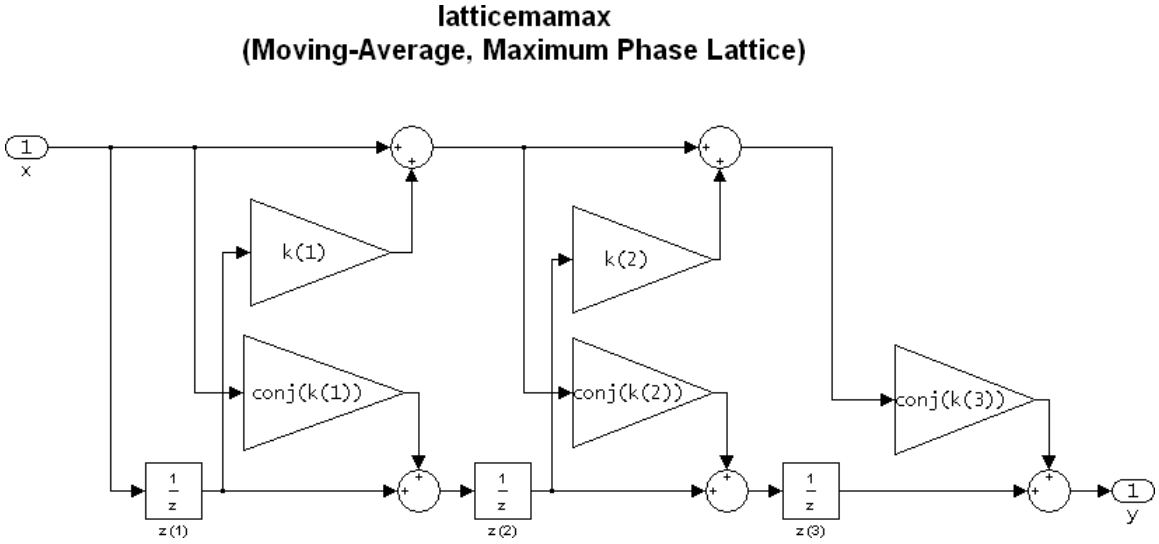
```
Hd = dfilt.latticemamax(k)
Hd = dfilt.latticemamax
```

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.



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)
```

Introduced before R2006a

dfilt.latticemamin

Discrete-time, lattice, moving-average filter

Syntax

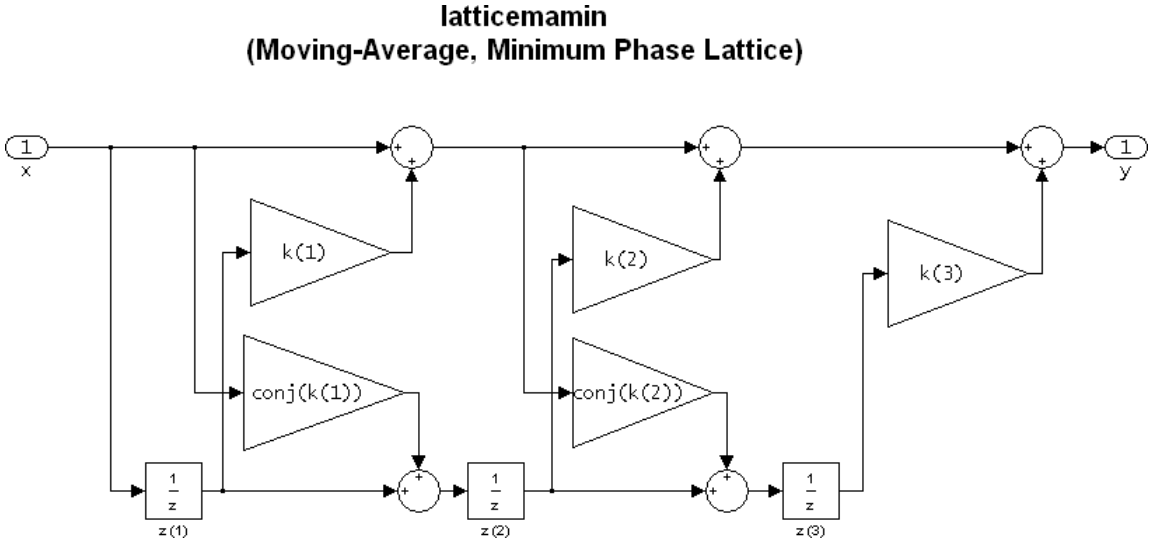
```
Hd = dfilt.latticemamin(k)
Hd = dfilt.latticemamin
```

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.



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

Signal Analyzer | `designfilt`

Introduced before R2006a

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

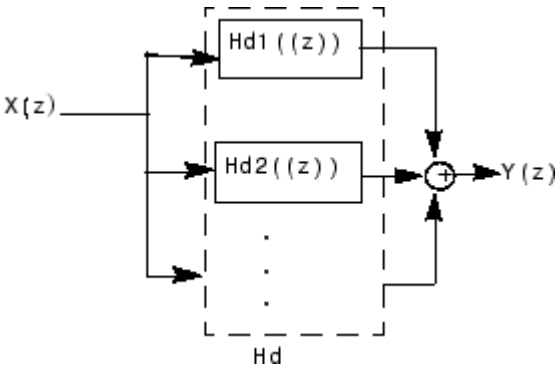
```
Hd.stage(1).states
```

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

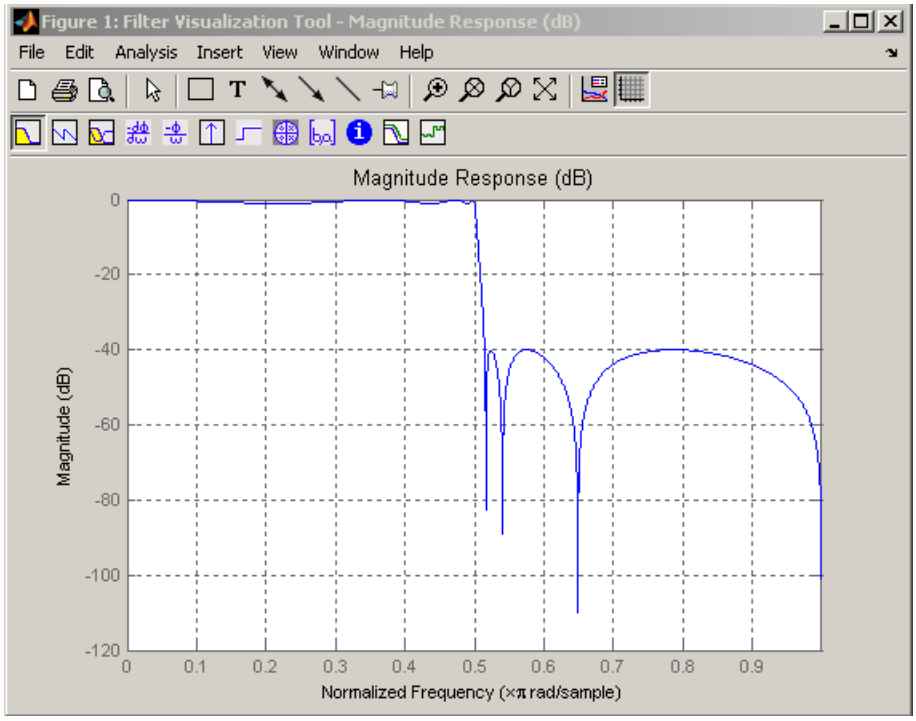
```
info(Hcas.Stage(1))
```

and

```
info(Hcas.Stage(2))
```

To view this filter, use

```
fvtool(Hcas)
```



See Also

Signal Analyzer | `designfilt`

Introduced before R2006a

dfilt.scalar

Discrete-time, scalar filter

Syntax

```
Hd = dfilt.scalar(g)
Hd = dfilt.scalar
```

Description

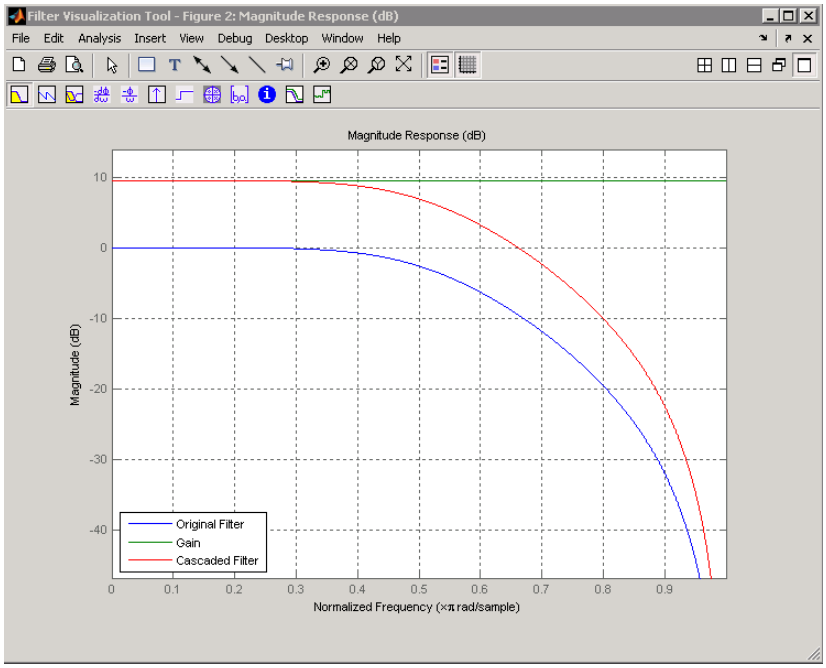
`Hd = dfilt.scalar(g)` returns a discrete-time, scalar filter, `Hd`, with gain `g`, where `g` is a scalar.

`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);
hfvt = fvtool(Hd_filt,Hd_gain,Hd_cascade);
legend(hfvt,'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

Signal Analyzer | `designfilt`

Introduced before R2006a

dfilt.statespace

Discrete-time, state-space filter

Syntax

```
Hd = dfilt.statespace(A,B,C,D)
Hd = dfilt.statespace
```

Description

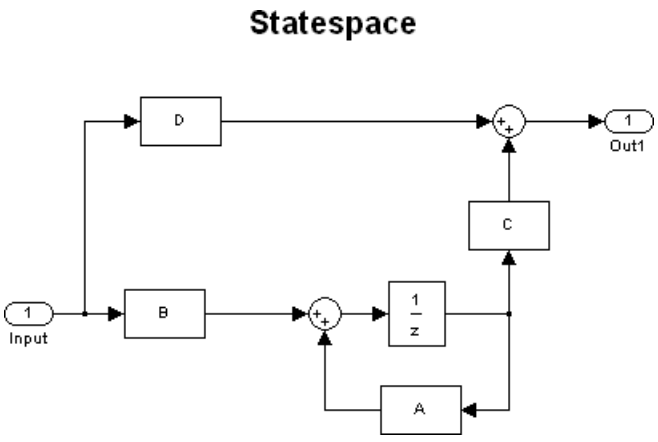
`Hd = dfilt.statespace(A,B,C,D)` returns a discrete-time state-space filter, `Hd`, with rectangular arrays `A`, `B`, `C`, and `D`.

`A`, `B`, `C`, and `D` are from the matrix or state-space form of a filter's difference equations

$$x(n+1) = Ax(n) + Bu(n)$$
$$y(n) = Cx(n) + Du(n)$$

where $x(n)$ is the vector states at time n, $u(n)$ is the input at time n, `y` is the output at time n, `A` is the state-transition matrix, `B` is the input-to-state transmission matrix, `C` is the state-to-output transmission matrix, and `D` is the input-to-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.



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

Signal Analyzer | `designfilt`

Introduced before R2006a

dftmtx

Discrete Fourier transform matrix

Syntax

```
a = dftmtx(n)
```

Description

`a = dftmtx(n)` returns an `n`-by-`n` complex discrete Fourier transform matrix.

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 = 8.0174e-12
```

Input Arguments

n — Discrete Fourier transform length

positive integer

Discrete Fourier transform length, specified as an integer.

Data Types: `single` | `double`

Output Arguments

a — Discrete Fourier transform matrix

matrix

Discrete Fourier transform matrix, returned as a matrix.

More About

Discrete Fourier Transform Matrix

A *discrete Fourier transform matrix* is a complex matrix whose matrix product with a vector computes the discrete Fourier transform of the vector. `dftmtx` takes the FFT of the identity matrix to generate the transform matrix.

For a column vector `x`,

```
y = dftmtx(n)*x
```

is the same as `y = fft(x,n)`. The inverse discrete Fourier transform matrix is

```
ainv = conj(dftmtx(n))/n
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`convmtx` | `fft`

Introduced before R2006a

digitalFilter

Digital filter

Description

Use `designfilt` to design and edit `digitalFilter` objects.

- Use `designfilt` in the form `d = designfilt(resp,Name,Value)` to design a digital filter, `d`, with response type `resp`. Customize the filter further using `Name,Value` pairs.
- Use `designfilt` in the form `designfilt(d)` to edit an existing filter, `d`.

  Note This is the only way to edit an existing `digitalFilter` object. Its properties are otherwise read-only.
- 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`.
- These functions take `digitalFilter` objects as input.

Object Functions

Filtering

Function	Description
`fftfilt`	Filters a signal with a `digitalFilter` using an FFT-based overlap-add method
`filter`	Filters a signal using a `digitalFilter`
`filtfilt`	Performs zero-phase filtering of a signal with a `digitalFilter`

Filter Analysis

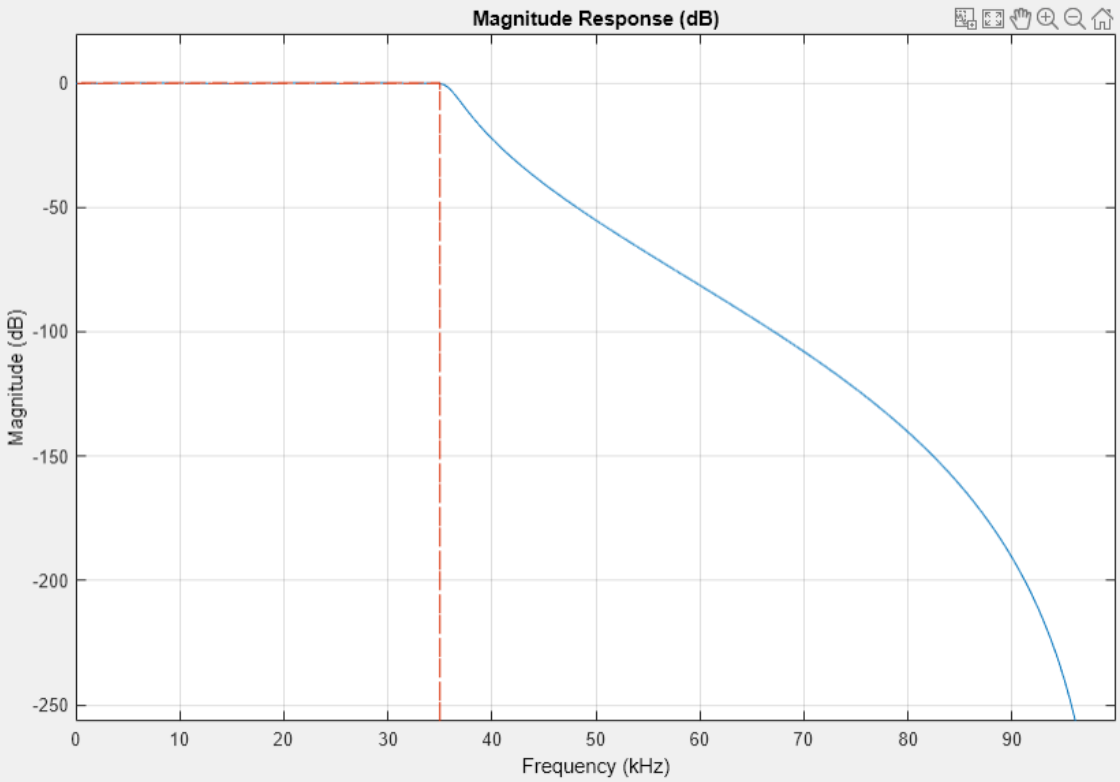
Function	Description
`double`	Casts the coefficients of a `digitalFilter` to double precision
`filt2block`	Generates a Simulink filter block corresponding to a `digitalFilter`
`filtord`	Returns the filter order of a `digitalFilter`
`firtype`	Returns the type (1, 2, 3, or 4) of an FIR `digitalFilter`
`freqz`	Returns or plots the frequency response of a `digitalFilter`
FVTool	Opens the Filter Visualization Tool and displays the magnitude response of a `digitalFilter`
`grpdelay`	Returns or plots the group delay response of a `digitalFilter`
`impz`	Returns or plots the impulse response of a `digitalFilter`
`impzlength`	Returns the length of the impulse response of a `digitalFilter`, whether actual (for FIR filters) or effective (for IIR filters)
`info`	Returns a character array with information about a `digitalFilter`
`isallpass`	Returns `true` if a `digitalFilter` is allpass
`isdouble`	Returns `true` if the coefficients of a `digitalFilter` are double precision
`isfir`	Returns `true` if a `digitalFilter` has a finite impulse response
`islinphase`	Returns `true` if a `digitalFilter` has linear phase
`ismaxphase`	Returns `true` if a `digitalFilter` is maximum phase
`isminphase`	Returns `true` if a `digitalFilter` is minimum phase
`issingle`	Returns `true` if the coefficients of a `digitalFilter` are single precision
`isstable`	Returns `true` if a `digitalFilter` is stable
`phasedelay`	Returns or plots the phase delay response of a `digitalFilter`
`phasez`	Returns or plots the (unwrapped) phase response of a `digitalFilter`
`single`	Casts the coefficients of a `digitalFilter` to single precision
`ss`	Returns the state-space representation of a `digitalFilter`
`stepz`	Returns or plots the step response of a `digitalFilter`
`tf`	Returns the transfer function representation of a `digitalFilter`
`zerophase`	Returns or plots the zero-phase response of a `digitalFilter`
`zpk`	Returns the zero-pole-gain representation of a `digitalFilter`
`zplane`	Displays the poles and zeros of the transfer function represented by a `digitalFilter`

Examples

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.

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



Use the filter you designed to filter a 1000-sample random signal.

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

Output the filter coefficients, expressed as second-order sections.

```
sos = lpFilt.Coefficients
```

```
sos = 4×6

    0.2666    0.5333    0.2666    1.0000   -0.8346    0.9073
    0.1943    0.3886    0.1943    1.0000   -0.9586    0.7403
    0.1012    0.2023    0.1012    1.0000   -1.1912    0.5983
    0.0318    0.0636    0.0318    1.0000   -1.3810    0.5090
```

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

Introduced in R2014a

digitrevorder

Permute input into digit-reversed order

Syntax

```
y = digitrevorder(x,r)
[y,i] = digitrevorder(x,r)
```

Description

`y = digitrevorder(x,r)` returns the input data `x` in digit-reversed order. The digit-reversal is computed using the number system base (radix base) `r`.

`[y,i] = digitrevorder(x,r)` returns the digit-reversed input data and digit-reversed indices, such that `y = x(i)`.

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=9×5 table
    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
```

Input Arguments

x — Input data

vector | matrix

Input data, specified as a vector or matrix. 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.

r — Radix base

integer

Radix base, specified as an integer in the range [2, 36].

Output Arguments

y — Digit-reversed data

vector | matrix

Digit-reversed data, returned as a vector or matrix. `y` is the same size as `x`.

i — Digit-reversed indices

vector | matrix

Digit-reversed indices, returned as a vector or matrix.

Note MATLAB matrices use 1-based indexing, so the first index of `y` will be 1, not 0.

More About

Digit-Reversed Numbers

This table shows the numbers 0 through 15, the corresponding digits and the digit-reversed numbers using radix base-4. The corresponding radix base-2 bits and bit-reversed indices are also shown.

Linear Index	Base-4 Digits	Digit-Reversed	Digit-Reversed Index	Base-2 Bits	Base-2 Reversed (bitrevorder)	Bit- Reversed Index
0	00	00	0	0000	0000	0
1	01	10	4	0001	1000	8
2	02	20	8	0010	0100	4
3	03	30	12	0011	1100	12
4	10	01	1	0100	0010	2
5	11	11	5	0101	1010	10
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

Linear Index	Base-4 Digits	Digit-Reversed	Digit-Reversed Index	Base-2 Bits	Base-2 Reversed (bitrevorder)	Bit- Reversed Index
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

Tips

- The `digitrevorder` function 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.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`bitrevorder` | `fft` | `ifft`

Introduced before R2006a

diric

Dirichlet or periodic sinc function

Syntax

```
y = diric(x,n)
```

Description

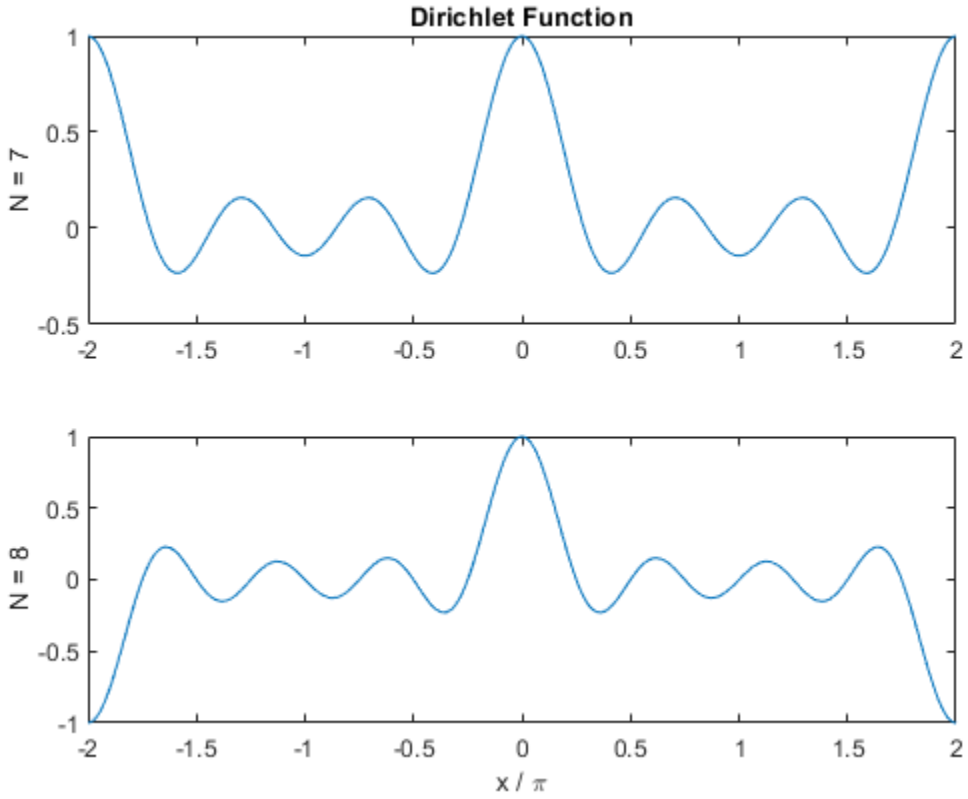
`y = diric(x,n)` returns the “Dirichlet Function” on page 1-391 of degree `n` evaluated at the elements of the input array `x`.

Examples

Dirichlet Function

Compute and plot the Dirichlet function between -2π and 2π for $N = 7$ and $N = 8$. The function has a period of 2π for odd N and 4π for even N.

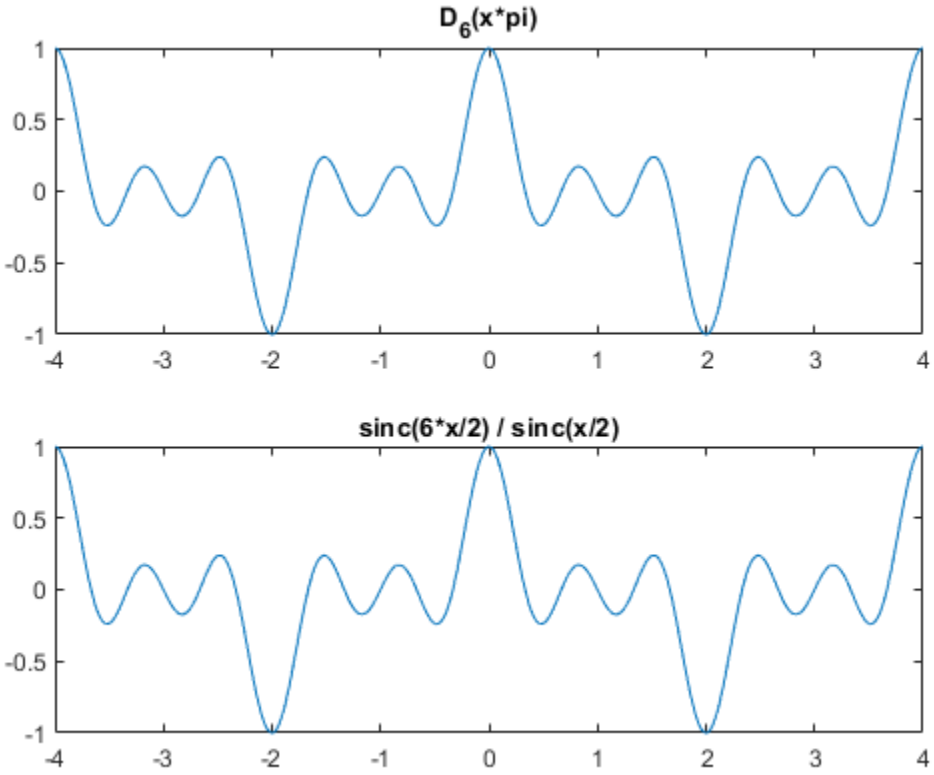
```
x = linspace(-2*pi,2*pi,301);

d7 = diric(x,7);
d8 = diric(x,8);

subplot(2,1,1)
plot(x/pi,d7)
ylabel('N = 7')
title('Dirichlet Function')

subplot(2,1,2)
plot(x/pi,d8)
ylabel('N = 8')
xlabel('x / \pi')
```



Periodic and Aperiodic Sinc Functions

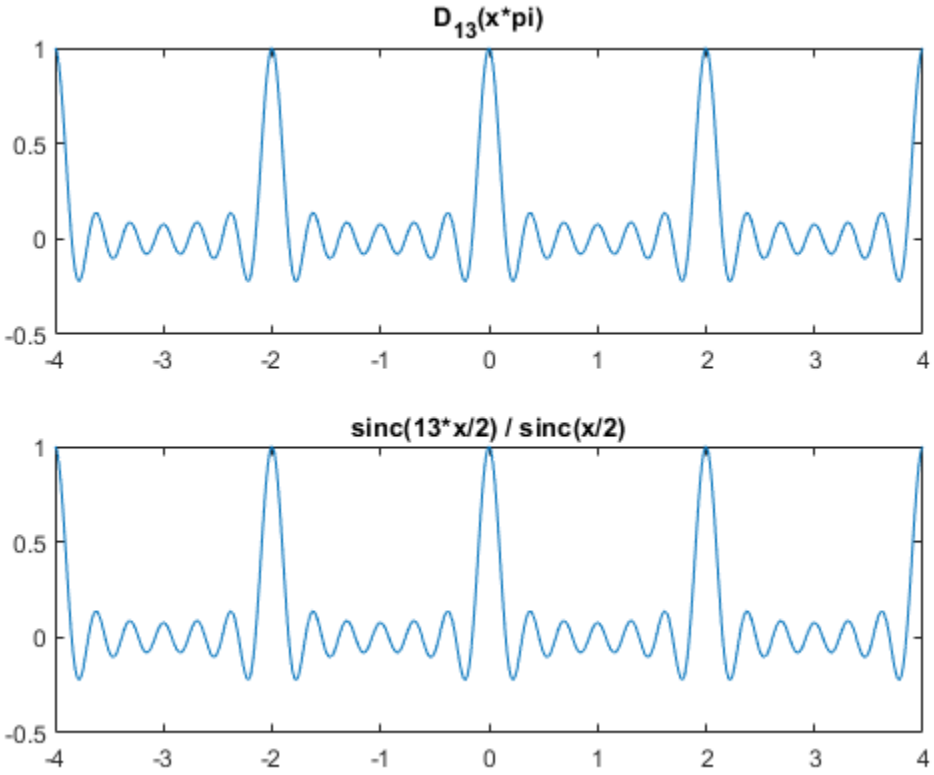
The Dirichlet and sinc functions are related by $D_N(\pi x) = \text{sinc}(Nx/2)/\text{sinc}(x/2)$. Show this relationship for $N = 6$. Avoid indeterminate expressions by specifying that the ratio of sinc functions is $(-1)^{k(N-1)}$ for $x = 2k$, where k is an integer.

```
xmax = 4;
x = linspace(-xmax,xmax,1001)';

N = 6;

yd = diric(x*pi,N);
ys = sinc(N*x/2)./sinc(x/2);
ys(~mod(x,2)) = (-1).^(x(~mod(x,2))/2*(N-1));

subplot(2,1,1)
plot(x,yd)
title('D_6(x*pi)')
subplot(2,1,2)
plot(x,ys)
title('sinc(6*x/2) / sinc(x/2)')
```



Repeat the calculation for $N = 13$.

```
N = 13;

yd = diric(x*pi,N);
ys = sinc(N*x/2)./sinc(x/2);
ys(~mod(x,2)) = (-1).^(x(~mod(x,2))/2*(N-1));

subplot(2,1,1)
plot(x,yd)
title('D_{13}(x*pi)')
subplot(2,1,2)
plot(x,ys)
title('sinc(13*x/2) / sinc(x/2)')
```



Input Arguments

x — Input array

real scalar | real vector | real matrix | real *N*-D array

Input array, specified as a real scalar, vector, matrix, or *N*-D array. When `x` is nonscalar, `diric` is an element-wise operation.

Data Types: `double` | `single`

n — Function degree

positive integer scalar

Function degree, specified as a positive integer scalar.

Data Types: `double` | `single`

Output Arguments

y — Output array

real scalar | real vector | real matrix | real *N*-D array

Output array, returned as a real-valued scalar, vector, matrix, or *N*-D array of the same size as `x`.

More About

Dirichlet Function

The Dirichlet function, or periodic sinc function, is

$$D_N(x) = \begin{cases} \dfrac{\sin(Nx/2)}{N\sin(x/2)} & x \neq 2\pi k, \quad k = 0, \pm 1, \pm 2, \pm 3, \ldots \\ (-1)^{k(N-1)} & x = 2\pi k, \quad k = 0, \pm 1, \pm 2, \pm 3, \ldots \end{cases}$$

for any nonzero integer N.

This function has period 2π for odd N and period 4π for even N. Its maximum value is 1 for all N, and its minimum value is –1 for even N. The magnitude of the function is $1/N$ times the magnitude of the discrete-time Fourier transform of the N-point rectangular window.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

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

Introduced before R2006a

dlstft

Deep learning short-time Fourier transform

Syntax

```
[yr,yi] = dlstft(x)
[yr,yi] = dlstft(x,fs)
[yr,yi] = dlstft(x,ts)

[yr,yi] = dlstft(___,Name,Value)

[yr,yi,f,t] = dlstft(___)
```

Description

`[yr,yi] = dlstft(x)` returns the deep learning "Short-Time Fourier Transform" on page 1-397 (STFT) of `x`. `dlstft` requires Deep Learning Toolbox™.

`[yr,yi] = dlstft(x,fs)` returns the deep learning STFT assuming that `x` was sampled at the rate `fs`.

`[yr,yi] = dlstft(x,ts)` returns the deep learning STFT assuming that `x` was sampled with sample time `ts`.

`[yr,yi] = dlstft(___,Name,Value)` specifies additional options using name-value arguments. Options include the spectral window and the FFT length. These arguments can be added to any of the previous input syntaxes. For example, `'DataFormat','CBT'` specifies the data format of `x` as `CBT`.

`[yr,yi,f,t] = dlstft(___)` returns the frequencies `f` and times `t` at which the deep learning STFT is computed.

Examples

Deep Learning Short-Time Fourier Transform of Chirp

Generate a signal sampled at 600 Hz for 2 seconds. The signal consists of a chirp with sinusoidally varying frequency content.

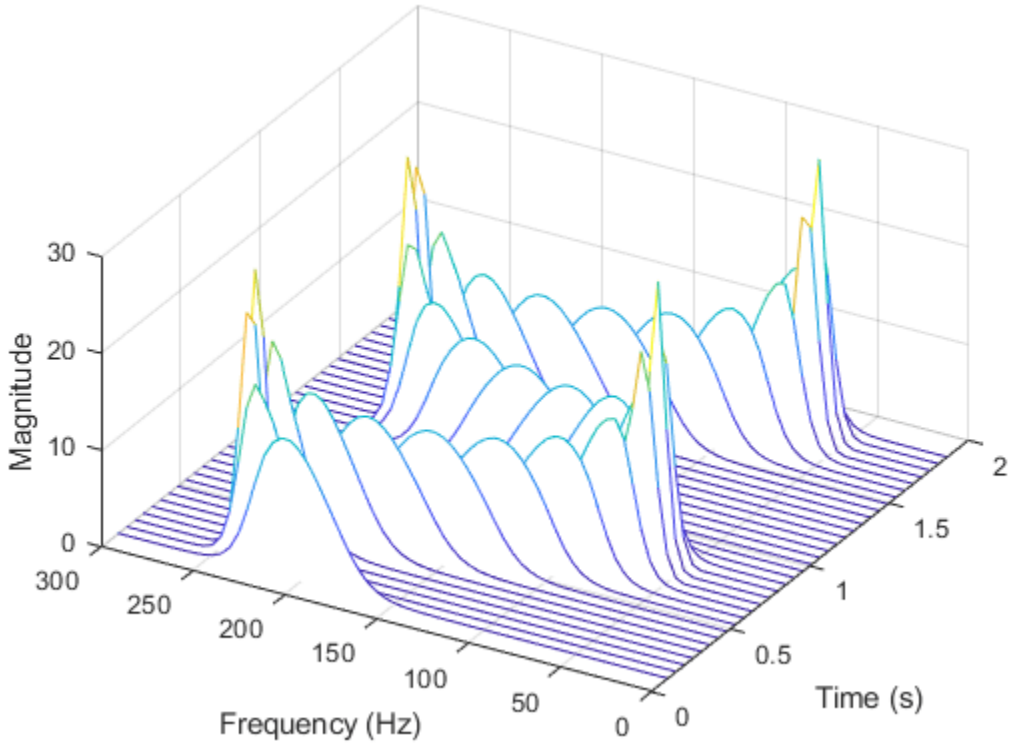
```
fs = 6e2;
t = 0:1/fs:2;
x = vco(sin(2*pi*t),[0.1 0.4]*fs,fs);
```

Store the signal in an unformatted deep learning array. Compute the short-time Fourier transform of the signal. Input the sample time as a `duration` scalar. (Alternatively, input the sample rate as a numeric scalar.) Specify that the input array is in `'CTB'` format.

```
dlx = dlarray(x);

[yr,yi,f,t] = dlstft(dlx,seconds(1/fs),'DataFormat','CTB');
```

Convert the outputs to numeric arrays. Compute the magnitude of the short-time Fourier transform and display it as a waterfall plot.

```
yr = extractdata(yr);
yi = extractdata(yi);

f = extractdata(f);
t = seconds(t);

waterfall(f,t,squeeze(hypot(yr,yi))')

ax = gca;
ax.XDir = 'reverse';
view(30,45)

ylabel('Time (s)')
xlabel('Frequency (Hz)')
zlabel('Magnitude')
```



Deep Learning Short-Time Fourier Transform of Sinusoid

Generate a 3-by-160(-by-1) array containing one batch of a three-channel, 160-sample sinusoidal signal. The normalized sinusoid frequencies are $\pi/4$ rad/sample, $\pi/2$ rad/sample, and $3\pi/4$ rad/sample. Save the signal as a `dlarray`, specifying the dimensions in order. `dlarray` permutes the

array dimensions to the 'CBT' shape expected by a deep learning network. Display the array dimension sizes.

```
x = dlarray(cos(pi.*(1:3)'/4*(0:159)),'CTB');
[nchan,nbtch,nsamp] = size(x)
```

```
nchan = 3
```

```
nbtch = 1
```

```
nsamp = 160
```

Compute the deep learning short-time Fourier transform of the signal. Specify a 64-sample rectangular window and an FFT length of 1024.

```
[re,im,f,t] = dlstft(x,'Window',rectwin(64),'FFTLength',1024);
```

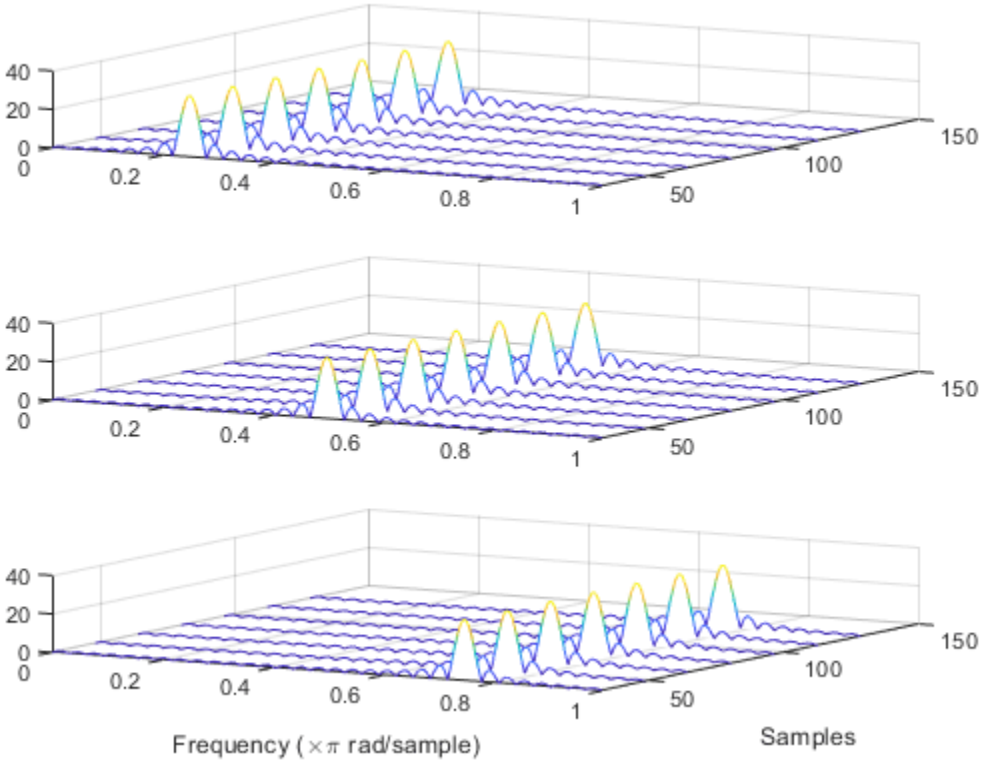
`dlstft` computes the transform along the 'T' dimension. The output arrays are in 'SCBT' format. The 'S' dimension corresponds to frequency in the short-time Fourier transform.

Extract the data from the deep learning arrays.

```
re = squeeze(extractdata(re));
im = squeeze(extractdata(im));

f = extractdata(f);
t = extractdata(t);
```

Compute the magnitude of the short-time Fourier transform. Plot the magnitude separately for each channel in a waterfall plot.

```
z = abs(re + 1j*im);

for kj = 1:nchan
    subplot(nchan,1,kj)
    waterfall(f/pi,t,squeeze(z(:,kj,:))')
    view(30,45)
end

xlabel('Frequency (\times\pi rad/sample)')
ylabel('Samples')
```



Input Arguments

x — Input array

`dlarray` object | numeric array

Input array, specified as an unformatted `dlarray`, a formatted `dlarray` in `'CBT'` format, or a numeric array. If `x` is an unformatted `dlarray` or a numeric array, you must specify the `'DataFormat'` as some permutation of `'CBT'`.

Example: `dlarray(cos(pi./[4;2]*(0:159)),'CTB')` and `dlarray(cos(pi./[4;2]*(0:159))','TCB')` both specify one batch observation of a two-channel sinusoid in the `'CBT'` format.

fs — Sample rate

2π (default) | positive numeric scalar

Sample rate, specified as a positive numeric scalar.

ts — Sample time

`duration` scalar

Sample time, specified as `duration` scalar. Specifying `ts` is equivalent to setting a sample rate f_s = 1/`ts`.

Example: `seconds(1)` is a `duration` scalar representing a 1-second time difference between consecutive signal samples.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Window',hamming(100),'OverlapLength',50,'FFTLength',128` windows the data using a 100-sample Hamming window, with 50 samples of overlap between adjoining segments and a 128-point FFT.

`DataFormat` — Data format of input

character vector | string scalar

Data format of input, specified as a character vector or string scalar. This argument is valid only if `x` is unformatted.

Each character in this argument must be one of these labels:

- `C` — Channel
- `B` — Batch observations
- `T` — Time

The `dlstft` function accepts any permutation of `'CBT'`. You can specify at most one of each of the `C`, `B`, and `T` labels.

Each element of the argument labels the matching dimension of `x`. If the argument is not in the listed order (`'C'` followed by `'B'` and so on), then `dlstft` implicitly permutes both the argument and the data to match the order, but without changing how the data is stored.

Example: `'CBT'`

`Window` — Spectral window

`hann(128,'periodic')` (default) | vector

Spectral window, specified as a vector. If you do not specify the window or specify it as empty, the function uses a Hann window of length 128. The length of `'Window'` must be greater than or equal to 2.

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: `double` | `single`

`OverlapLength` — Number of overlapped samples

75% of window length (default) | nonnegative integer

Number of overlapped samples, specified as a nonnegative integer smaller than the length of `'Window'`. If you omit `'OverlapLength'` or specify it as empty, it is set to the largest integer less than 75% of the window length, which is 96 samples for the default Hann window.

Data Types: `double` | `single`

`FFTLength` — Number of discrete Fourier transform (DFT) points

128 (default) | positive integer

Number of DFT points, specified as a positive integer. The value must be greater than or equal to the window length. If the length of the input signal is less than the DFT length, the data is padded with zeros.

Data Types: `double` | `single`

Output Arguments

`yr`, `yi` — Short-time Fourier transform

formatted `dlarray` objects | unformatted `dlarray` objects

Short-time Fourier transform, returned as two formatted `dlarray` objects. `yr` contains the real part of the transform. `yi` contains the imaginary part of the transform.

- If `x` is a formatted `dlarray`, `yr` and `yi` are `'SCBT'` formatted `dlarray` objects. The `'S'` dimension corresponds to frequency in the short-time Fourier transform.
- If `x` is an unformatted `dlarray` or a numeric array, `yr` and `yi` are unformatted `dlarray` objects. The dimension order in `yr` and `yi` is `'SCBT'`.

If no time information is specified, then the STFT is computed over the Nyquist range $[0, \pi]$ if `'FFTLength'` is even and over $[0, \pi)$ if `'FFTLength'` is odd. If you specify time information, then the intervals are $[0, f_s/2]$ and $[0, f_s/2)$, respectively, where f_s is the effective sample rate.

`f` — Frequencies

`dlarray` object

Frequencies at which the deep learning STFT is computed, returned as a `dlarray` object.

- If the input array does not contain time information, then the frequencies are in normalized units of rad/sample.
- If the input array contains time information, then `f` contains frequencies expressed in Hz.

`t` — Times

`dlarray` object | `duration` array

Times at which the deep learning STFT is computed, returned as a `dlarray` object or a `duration` array.

- If you do not specify time information, then `t` contains sample numbers.
- If you specify a sample rate, then `t` contains time values in seconds.
- If you specify a sample time, then `t` is a `duration` array with the same time format as `x`.

More About

Short-Time Fourier Transform

The short-time Fourier transform (STFT) is used to analyze how the frequency content of a nonstationary signal changes over time.

The STFT of a signal is calculated by sliding an analysis window of length M over the signal and calculating the discrete Fourier transform of the windowed data. The window hops over the original

signal at intervals of R samples. Most window functions taper off at the edges to avoid spectral ringing. If a nonzero overlap length L is specified, overlap-adding the windowed segments compensates for the signal attenuation at the window edges. The DFT of each windowed segment is added to a matrix that contains the magnitude and phase for each point in time and frequency. The number of columns in the STFT matrix is given by

$$k = \left\lfloor \frac{N_x - L}{M - L} \right\rfloor,$$

where N_x is the length of the original signal $x(n)$ and the $\lfloor \rfloor$ symbols denote the floor function. The number of rows in the matrix equals N_{DFT}, the number of DFT points, for centered and two-sided transforms and $\lfloor N_{\mathrm{DFT}}/2 \rfloor + 1$ for one-sided transforms.

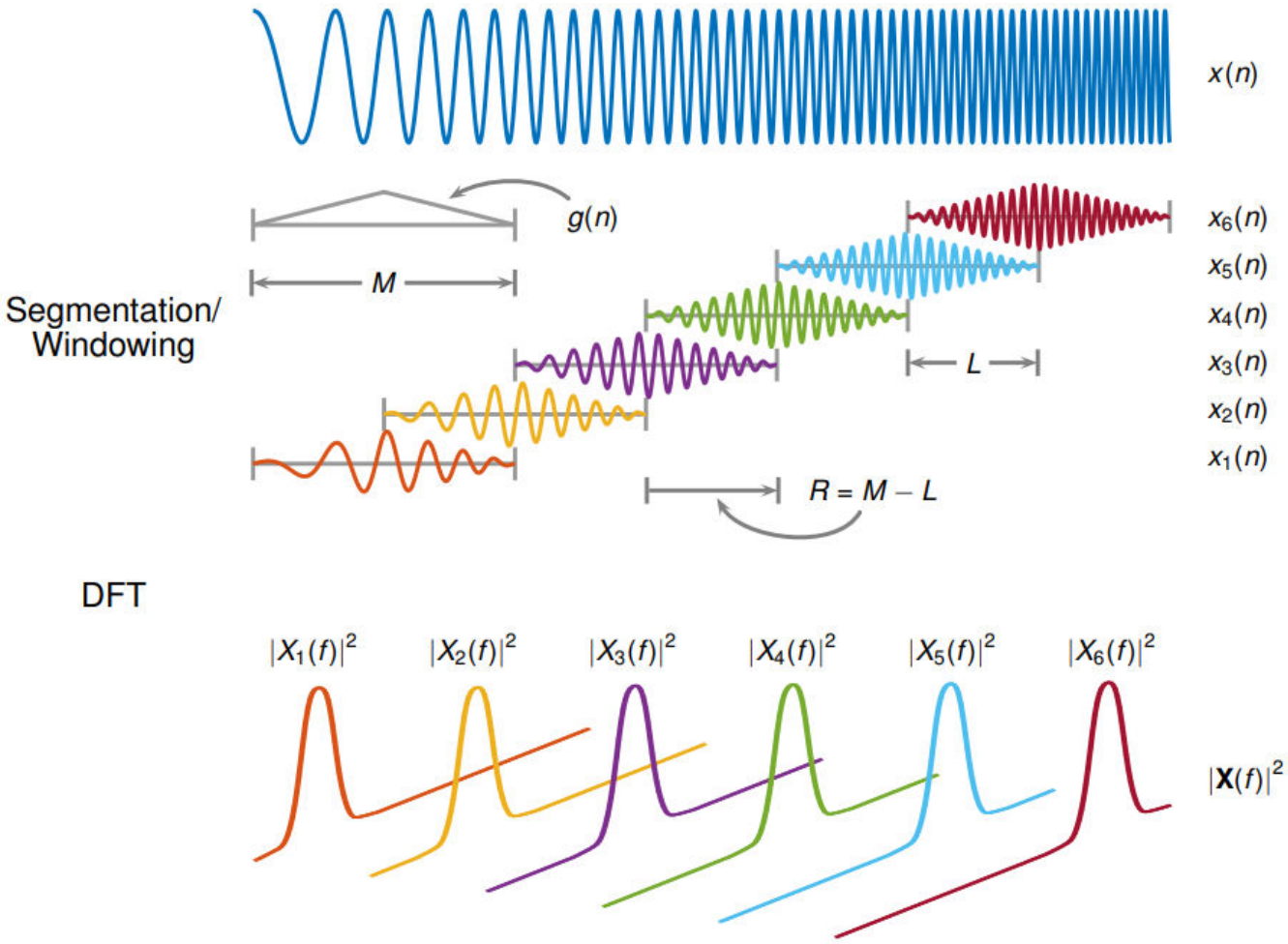
The STFT matrix is given by $\mathbf{X}(f) = \left[X_1(f)\ X_2(f)\ X_3(f)\ \cdots\ X_k(f) \right]$ such that the mth element of this matrix is

$$X_m(f) = \sum_{n=-\infty}^{\infty} x(n) g(n - mR) e^{-j2\pi f n},$$

where

- $g(n)$ — Window function of length M.
- $X_m(f)$ — DFT of windowed data centered about time mR.
- R — Hop size between successive DFTs. The hop size is the difference between the window length $M$and the overlap length L.

The magnitude squared of the STFT yields the `spectrogram` representation of the power spectral density of the function.



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dlarray` | `stft` | `istft` | `stftmag2sig` | `stftLayer`

Topics

“Spoken Digit Recognition with Custom Log Spectrogram Layer and Deep Learning”

Introduced in R2021a

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 0.2π rad/sample and a normalized stopband frequency of 0.55π 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)

coed =
   0.003947882145754


coee = e.Coefficients(1)

coee = single
   0.0039479


coef = f.Coefficients(1)

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 0.5π 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`

Introduced in R2014a

downsample

Decrease sample rate by integer factor

Syntax

```
y = downsample(x,n)
y = downsample(x,n,phase)
```

Description

`y = downsample(x,n)` decreases the sample rate of `x` by keeping the first sample and then every `n`th sample after the first. If `x` is a matrix, the function treats each column as a separate sequence.

`y = downsample(x,n,phase)` specifies the number of samples by which to offset the downsampled sequence.

Examples

Decrease Sample Rates

Decrease the sample rate of a sequence by a factor of 3.

```
x = [1 2 3 4 5 6 7 8 9 10];
y = downsample(x,3)
```

```
y = 1×4

     1     4     7    10
```

Decrease the sample rate of the sequence by a factor of 3 and add a phase offset of 2.

```
y = downsample(x,3,2)
```

```
y = 1×3

     3     6     9
```

Decrease the sample rate of a matrix by a factor of 3.

```
x = [1  2  3;
     4  5  6;
     7  8  9;
    10 11 12];
y = downsample(x,3)
```

```
y = 2×3

     1     2     3
    10    11    12
```

Input Arguments

x — Input array

vector | matrix

Input array, specified as a vector or matrix. If `x` is a matrix, the function treats the columns as independent channels.

Example: `cos(pi/4*(0:159)) + randn(1,160)` specifies a sinusoid embedded in white Gaussian noise.

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

n — Downsampling factor

positive integer

Downsampling factor, specified as a positive integer.

Data Types: `single` | `double`

phase — Offset

`0` (default) | positive integer

Offset, specified as a positive integer from 0 to `n` – 1.

Data Types: `single` | `double`

Output Arguments

y — Downsampled array

vector | matrix

Downsampled array, returned as a vector or matrix.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`decimate` | `interp` | `interp1` | `resample` | `spline` | `upfirdn` | `upsample`

Introduced before R2006a

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 `round(2*time_halfbandwidth)` discrete prolate spheroidal (DPSS), or Slepian sequences of length `seq_length`. `dps_seq` is a matrix with `seq_length` rows and `round(2*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 sample rate. `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.

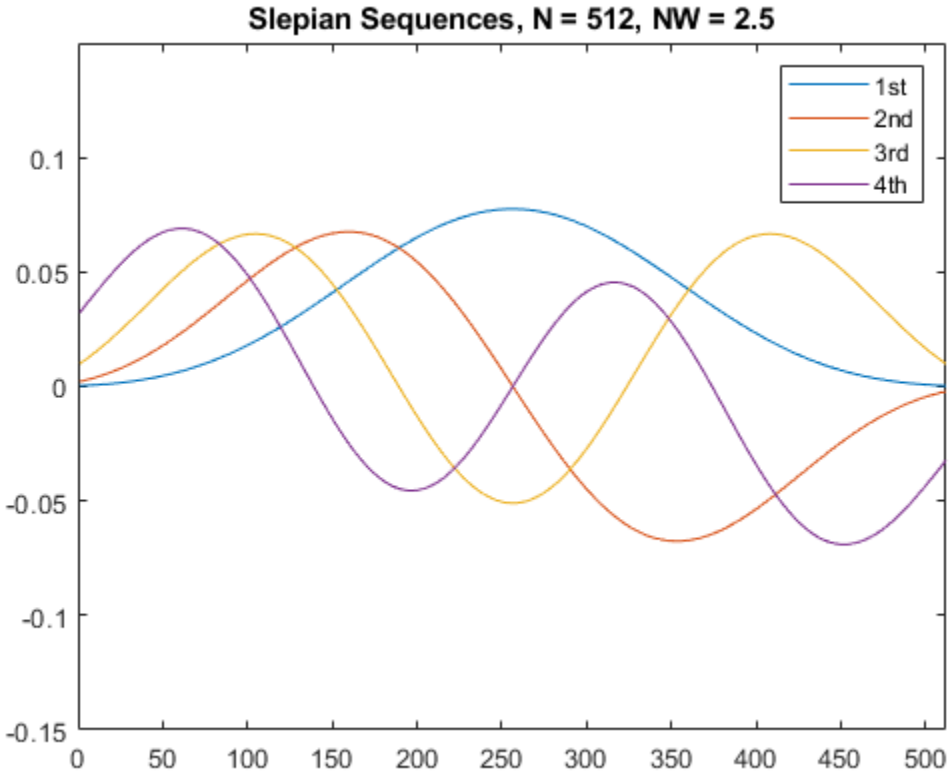
`[...] = dpss(...,'trace')` prints the method used to compute the DPSSs in the command window. Possible methods include: direct, spline interpolation, and linear interpolation.

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 = 1×4

    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:

$$\lambda = \frac{\int_{-W}^{W} |X(f)|^2 df}{\int_{-Fs/2}^{Fs/2} |X(f)|^2 df}$$

where F_s is the sample rate and $|W| < Fs/2$. 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 2*W*. Both the sequence length and bandwidth affect how many Slepian sequences have concentration ratios near one. As a rule, there are 2*NW* – 1 Slepian sequences with energy concentration ratios approximately equal to one. Beyond 2*NW* – 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 sample rate.

References

Percival, D. B., and A. T. Walden. *Spectral Analysis for Physical Applications.* Cambridge, UK: Cambridge University Press, 1993.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`dpssclear` | `dpssload` | `dpsssave` | `pmtm`

Topics

“Nonparametric Methods”

Introduced before R2006a

dpssclear

Remove discrete prolate spheroidal sequences from database

Syntax

```
dpssclear(n,nw)
```

Description

`dpssclear(n,nw)` removes sequences with length `n` and time-bandwidth product `nw` from the DPSS MAT-file database `dpss.mat`.

See Also

`dpss` | `dpssdir` | `dpssload` | `dpsssave`

Introduced before R2006a

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` | `dpssclear` | `dpssload` | `dpsssave`

Introduced before R2006a

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` | `dpssclear` | `dpssdir` | `dpsssave`

Introduced before R2006a

dpsssave

Discrete prolate spheroidal or Slepian sequence database

Syntax

```
dpsssave(time_halfbandwith,dps_seq,lambda)
status = dpsssave(time_halfbandwith,dps_seq,lambda)
```

Description

`dpsssave(time_halfbandwith,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_halfbandwith` is a real-valued scalar determining the frequency concentration of the Slepian sequences in `dps_seq`. `dps_seq` is a *NxK* matrix of Slepian sequences where *N* is the length of the sequences. `lambda` is a *1xK* 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_halfbandwith,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] = dpss(seq_length,time_halfbandwidth);
status = dpsssave(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 sample rate and $|W| < Fs/2$. 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 2*W*. Both the sequence length and bandwidth affect how many Slepian sequences have concentration ratios near one. As a rule, there are 2*NW* – 1 Slepian sequences with energy concentration ratios approximately equal to one. Beyond 2*NW* – 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 sample rate.

References

Percival, D. B., and A. T. Walden. *Spectral Analysis for Physical Applications.* Cambridge, UK: Cambridge University Press, 1993.

See Also

`dpss` | `dpssclear` | `dpssdir` | `dpssload`

Introduced before R2006a

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.

dspdata.dataobj	Description	Corresponding Functions
`dspdata.msspectrum`	Mean-square spectrum data (power)	`periodogram` `pwelch`

`dspdata.psd`	Power spectral density data (power/frequency)	`pburg`
		`pcov`
		`periodogram`
		`pmcov`
		`pmtm`
		`pwelch`
		`pyulear`

`dspdata.pseudospectrum`	Pseudospectrum data (power)	`peig` `pmusic`

For more information on each *`dataobj`* type, use the syntax `help dspdata.`*`dataobj`* at the MATLAB prompt or refer to its reference page.

Methods

Methods provide ways of performing functions directly on your `dspdata` object. You can apply these methods directly on the variable you assigned to your `dspdata` object.

Method	Description
`avgpower`	This method applies only to `dspdata.psd` objects. `avgpower(Hs)` computes the average power of a signal, `Hs`, 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 `SpectrumType` 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 [0,pi] if the number of frequency points is even and [0,pi) if it is odd. For a two-sided spectrum, the range is [0,2pi). `avgpower(Hs,freqrange)` specifies the frequency range over which to calculate the average power. `freqrange` is a two-element vector containing the lower and upper bounds of the frequency range. If a frequency value does not match exactly the frequency in `Hs`, the next closest value is used. The first frequency value in `freqrange` is included in the calculation and the second value is excluded.
`centerdc`	`centerdc(Hs)` or `centerdc(Hs,true)` shifts the data and frequency values so that the DC component is at the center of the spectrum. If the `SpectrumType` property is `'onesided'`, it is changed to `'twosided'` and then the DC component is centered. `centerdc(Hs,'false')` shifts the data and frequency values so that the DC component is at the left edge of the spectrum.

Method	Description
`findpeaks`	`findpeaks(Hs)` finds local maxima or peaks. If no peaks are found, `findpeaks` returns an empty vector. `[pks,frqs] = findpeaks(x)` returns the peaks' values, `pks`, and the frequencies, `frqs`, at which they occur. `findpeaks(x,'minpeakheight',mph)` returns only peaks greater than the minimum peak height `mph`, where `mph` is a real scalar. The default is `-Inf`. `findpeaks(x,'minpeakdistance',mpd)` returns only peaks separated by the minimum frequency units distance `mpd`, which is a positive integer. Setting the minimum peak distance ignores smaller peaks that may occur close to larger local peaks. The default is 1. `findpeaks(x,'threshold',th)` returns only peaks greater than their neighbors by at least the threshold, `th`, which is a real, scalar value greater than or equal to 0. The default is 0. `findpeaks(x,'npeaks',np)` returns a maximum of `np` number of peaks. When `np` peaks are found, the search stops. The default is to return all peaks. `findpeaks(x,'sortstr',str)` specifies the sorting order, where `str` is `'ascend'`, `'descend'`, or `'none'`. When `str` is set to `'ascend'`, the peaks are sorted from smallest to largest. When `str` is set to `'descend'` the peaks are sorted in descending order. When `str` is set to `'none'`, the peaks are returned in the order in which they occur.
`halfrange`	`halfrange(Hs)` converts the spectrum of `Hs` to a spectrum calculated over half the Nyquist interval. All associated properties affected by the new frequency range are adjusted automatically. This method is used for `dspdata.pseudospectrum` objects. The spectrum is assumed to be from a real signal. That is, `halfrange` uses half the data points regardless of whether the data is symmetric.
`normalizefreq`	`normalizefreq(Hs)` or `normalizefreq(Hs,true)` normalizes the frequency specifications in the `Hs` object to `Fs` so the frequencies are between 0 and 1. It also sets the `NormalizedFrequency` property to `true`. `normalizefreq(Hs,false)` converts the frequencies to linear frequencies. `normalizefreq(Hs,false,Fs)` sets a new sampling frequency, `Fs`. This can be used only with `false`.

Method	Description
`onesided`	`onesided(Hs)` converts the spectrum of `Hs` to a spectrum calculated over half the Nyquist interval and containing the total signal power. All associated properties affected by the new frequency range are adjusted automatically. This method is used for `dspdata.psd` and `dspdata.msspectrum` objects. The spectrum is assumed to be from a real signal. That is, `onesided` uses half the data points regardless of whether the data is symmetric.
`plot`	Displays the data graphically in the current figure window. For a `dspdata.psd` object, it displays the power spectral density in dB/Hz. For a `dspdata.msspectrum` object, it displays the mean–square in dB. For a `dspdata.pseudospectrum` object, it displays the pseudospectrum in dB.
`sfdr`	This method applies only to `dspdata.msspectrum` objects. `sfdr(Hs)` computes the spurious-free dynamic range (SFDR) in dB of a mean square spectrum object `Hs`. SFDR is the usable range before spurious noise interferes with the signal. `[sfd,spur,frq] = sfdr(Hs)` returns the magnitude of the highest `spur` and the frequency `frq` at which it occurs. `sfdr(Hs,'minspurlevel',msl)` ignores spurs below the minimum spur level `msl`, which is a real scalar in dB. `sfdr(Hs,'minspurdistance',msd)` includes spurs only if they are separated by at least the minimum spur distance `msd`, which is a real, positive scalar in frequency units.
`twosided`	`twosided(Hs)` converts the `Hs` spectrum to a spectrum calculated over the whole Nyquist interval. All associated properties affected by the new frequency range are adjusted automatically. This method is used for `dspdata.psd` and `dspdata.msspectrum` objects. If your data is nonuniformly sampled, converting from `onesided` to `twosided` may produce incorrect results.
`wholerange`	`wholerange(Hs)` converts the `Hs` spectrum to a spectrum calculated over the whole Nyquist interval. All associated properties affected by the new frequency range are adjusted automatically. This method is used for `dspdata.pseudospectrum` objects. If your data is nonuniformly sampled, converting from `half` to `wholerange` may produce incorrect results.

For more information on each method, use the syntax `help dspdata/`*`method`* at the MATLAB prompt.

Plotting a dspdata Object

The plot method displays the `dspdata` object spectrum in a separate figure window.

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`

Introduced before R2006a

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 character vector

Property	Default Value	Description
`Frequencies`	`[]` type `double`	Vector of frequencies at which the spectrum is evaluated. The range of this vector depends on the `SpectrumType` value. For a one-sided spectrum, the default range is [0, π) or [0, `Fs`/2) for odd length, and [0, π] or [0, `Fs`/2] for even length, if `Fs` is specified. For a two-sided spectrum, it is [0, 2π) or [0, `Fs`). The length of the `Frequencies` vector must match the length of the columns of `Data`. If you do not specify `Frequencies`, a default vector is created. If one-sided is selected, then the whole number of FFT points (nFFT) for this vector is assumed to be even. If `onesided` is selected and you specify `Frequencies`, the last frequency point is compared to the next-to-last point and to pi (or `Fs`/2, if `Fs` is specified). If the last point is closer to pi (or `Fs`/2) than it is to the previous point, nFFT is assumed to be even. If it is closer to the previous point, nFFT is assumed to be odd.
`Fs`	`'Normalized'`	Sampling frequency, which is `'Normalized'` if `NormalizedFrequency` is `true`. If `NormalizedFrequency` is `false` `Fs` defaults to `1` Hz.
`SpectrumType`	`'Onesided'`	Nyquist interval over which the spectral density is calculated. Valid values are `'Onesided'` and `'Twosided'`. See the `onesided` and `twosided` methods in `dspdata` for information on changing this property. The interval for `Onesided` is [0 π) or [0 π] depending on the number of FFT points, and for `Twosided` the interval is [0 2π).
`NormalizedFrequency`	`true`	Whether the frequency is normalized (`true`) or not (`false`). This property is set automatically at construction time based on `Fs`. If `Fs` is specified, `NormalizedFrequency` is set to `false`. See the `normalizefreq` method in `dspdata` for information on changing this property.

`Hmss = dspdata.msspectrum(Data,Frequencies)` uses the mean–square spectrum data contained in `Data` and `Frequencies` vectors.

`Hmss = dspdata.msspectrum(...,'Fs',Fs)` uses the sampling frequency `Fs`. Specifying `Fs` uses a default set of linear frequencies (in `Hz`) based on `Fs` and sets `NormalizedFrequency` to `false`.

`Hmss = dspdata.msspectrum(...,'SpectrumType',SpectrumType)` uses `SpectrumType` to specify the interval over which the mean-square spectrum was calculated. For data that ranges from [0 π) or [0 π], set the `SpectrumType` to `onesided`; for data that ranges from [0 2π), set the `SpectrumType` to `twosided`.

`Hmss = dspdata.msspectrum(...,'CenterDC',flag)` uses the value of `flag` to indicate whether the zero-frequency (DC) component is centered. If `flag` is `true`, it indicates that the DC component is in the center of the two-sided spectrum. Set the `flag` to `false` if the DC component is on the left edge of the spectrum.

Methods

Methods provide ways of performing functions directly on your `dspdata` object without having to specify the parameters again. You can apply a method directly on the variable you assigned to your `dspdata.msspectrum` object. You can use the following methods with a `dspdata.msspectrum` object.

- `centerdc`
- `normalizefreq`
- `onesided`
- `plot`
- `sfdr`
- `twosided`

For example, to normalize the frequency and set the `NormalizedFrequency` parameter to true, use

```
Hmss = normalizefreq(Hs)
```

For detailed information on using the methods and plotting the spectrum, see the `dspdata` reference page.

Examples

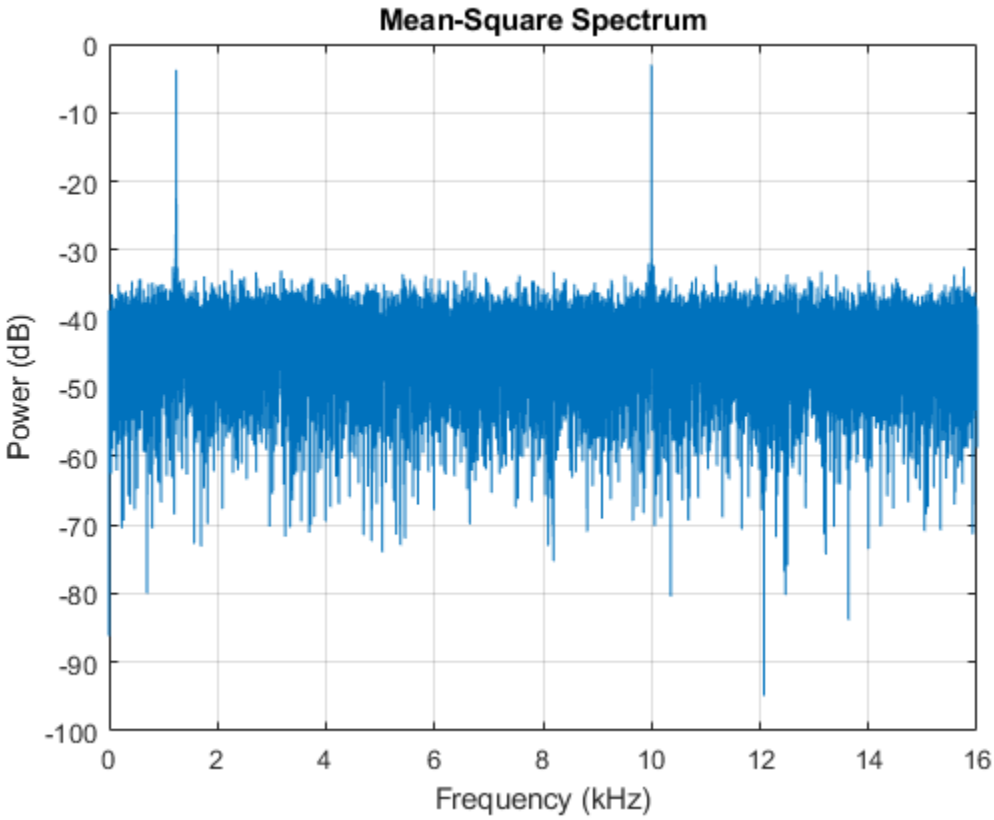
Mean-Square Spectrum of Sinusoids

Create a signal consisting 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));
```

Compute the one-sided PSD estimate of the signal. Use the result to construct a `dspdata` object. Plot the mean-square spectrum.

```
P = periodogram(x,[],[],Fs);

Hmss = dspdata.msspectrum(P,'Fs',Fs,'spectrumtype','onesided');

plot(Hmss)
```



See Also

`periodogram` | `pwelch`

Introduced before R2006a

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`, `pmcov`, `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 character vector

Property	Default Value	Description
`Frequencies`	`[]` type `double`	Vector of frequencies at which the power spectral density is evaluated. The range of this vector depends on the `SpectrumType` value. For one-sided, the default range is [0, π) or [0, `Fs`/2) for odd length, and [0, π] or [0, `Fs`/2] for even length, if `Fs` is specified. For two-sided, it is [0, 2pi) or [0, `Fs`). If you do not specify `Frequencies`, a default vector is created. If one-sided is selected, then the whole number of FFT points (nFFT) for this vector is assumed to be even. If `onesided` is selected and you specify `Frequencies`, the last frequency point is compared to the next-to-last point and to π (or `Fs`/2, if `Fs` is specified). If the last point is closer to π (or `Fs`/2) than it is to the previous point, nFFT is assumed to be even. If it is closer to the previous point, nFFT is assumed to be odd. The length of the `Frequencies` vector must match the length of the columns of `Data`.
`Fs`	`'Normalized'`	Sampling frequency, which is `'Normalized'` if `NormalizedFrequency` is `true`. If `NormalizedFrequency` is `false` `Fs` defaults to `1`.
`SpectrumType`	`'Onesided'`	Nyquist interval over which the power spectral density is calculated. Valid values are `'Onesided'` and `'Twosided'`. A one-sided PSD contains the total signal power in half the Nyquist interval. See the `onesided` and `twosided` methods in `dspdata` for information on changing this property. The range for half the Nyquist interval is [0 pi) or [0 pi] depending on the number of FFT points. For the whole Nyquist interval, the range is [0 2pi).
`NormalizedFrequency`	`true`	Whether the frequency is normalized (`true`) or not (`false`). This property is set automatically at construction time based on `Fs`. If `Fs` is specified, `NormalizedFrequency` is set to `false`. See the `normalizefreq` method in `dspdata` for information on changing this property.

`Hpsd = dspdata.psd(Data,Frequencies)` uses the power spectral density estimation data contained in `Data` and `Frequencies` vectors.

`Hpsd = dspdata.psd(...,'Fs',Fs)` uses the sampling frequency `Fs`. Specifying `Fs` uses a default set of linear frequencies (in `Hz`) based on `Fs` and sets `NormalizedFrequency` to `false`.

`Hpsd = dspdata.psd(...,'SpectrumType',SpectrumType)` specifies the interval over which the power spectral density is calculated. For data that ranges from [0 π) or [0 π], set the `SpectrumType` to `onesided`; for data that ranges from [0 2π), 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

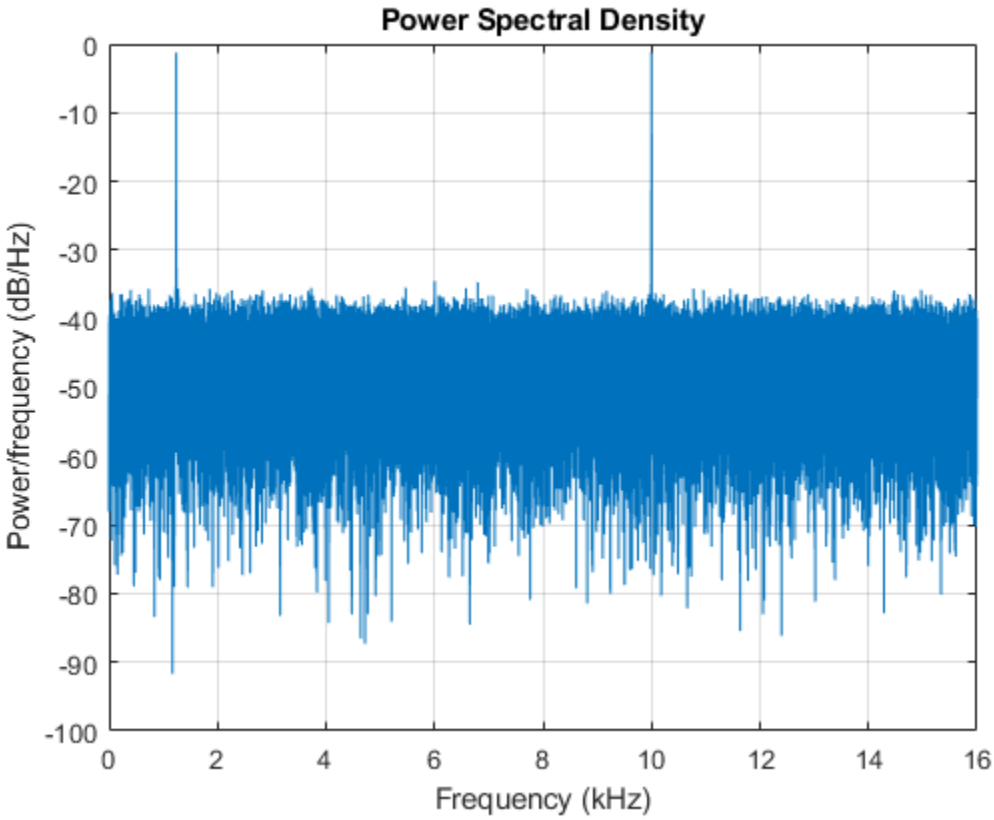
Resolve Signal Components

Estimate the one-sided power spectral density of a noisy sinusoidal signal with two frequency components.

```
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;
```

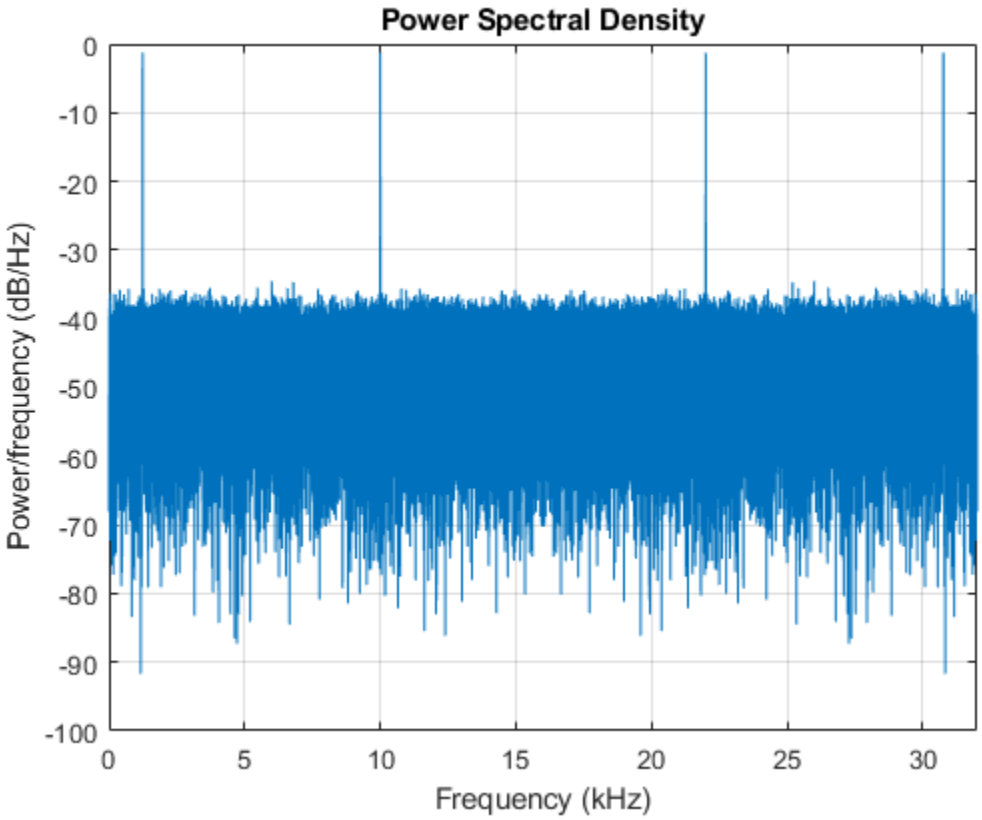
Store the spectrum in a PSD data object and plot the result.

```
Hpsd = dspdata.psd(Pxx(1:length(Pxx)/2),'Fs',Fs);
plot(Hpsd)
```



Create a two-sided spectrum and plot it.

```
Hpsd = dspdata.psd(Pxx,'Fs',Fs,'SpectrumType','twosided');
plot(Hpsd)
```



See Also

`pburg` | `pcov` | `periodogram` | `pmcov` | `pmtm` | `pwelch` | `pyulear`

Introduced before R2006a

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 character vector
`Frequencies`	`[]` type `double`	Vector of frequencies at which the power spectral density is evaluated. The range of this vector depends on the `SpectrumRange` value. For half, the default range is [0, π) or [0, `Fs`/2) for odd length, and [0, π] or [0, `Fs`/2] for even length, if `Fs` is specified. For whole, it is [0, 2π) or [0, `Fs`). If you do not specify `Frequencies`, a default vector is created. If half the Nyquist range is selected, then the whole number of FFT points (nFFT) for this vector is assumed to be even. If `half` the Nyquist range is selected and you specify `Frequencies`, the last frequency point is compared to the next-to-last point and to π (or `Fs`/2, if `Fs` is specified). If the last point is closer to π (or `Fs`/2) than it is to the previous point, nFFT is assumed to be even. If it is closer to the previous point, nFFT is assumed to be odd. The length of the `Frequencies` vector must match the length of the columns of `Data`.

Property	Default Value	Description
`Fs`	`'Normalized'`	Sampling frequency, which is `'Normalized'` if `NormalizedFrequency` is `true`. If `NormalizedFrequency` is `false` `Fs` defaults to `1`.
`SpectrumRange`	`'Half'`	Nyquist interval over which the pseudospectrum is calculated. Valid values are `'Half'` and `'Whole'`. See the `half` and `whole` methods in `dspdata` for information on changing this property. The interval for `Half` is [0 π) or [0 π] depending on the number of FFT points, and for `Whole` the interval is [0 2π).
`NormalizedFrequency`	`true`	Whether the frequency is normalized (`true`) or not (`false`). This property is set automatically at construction time based on `Fs`. If `Fs` is specified, `NormalizedFrequency` is set to `false`. See the `normalizefreq` method in `dspdata` for information on changing this property.

`Hps = dspdata.pseudospectrum(Data,Frequencies)` uses the pseudospectrum estimation data contained in the `Data` and `Frequencies` vectors.

`Hps = dspdata.pseudospectrum(...,'Fs',Fs)` uses the sampling frequency `Fs`. Specifying `Fs` uses a default set of linear frequencies (in `Hz`) based on `Fs` and sets `NormalizedFrequency` to `false`.

`Hps = dspdata.pseudospectrum(...,'SpectrumRange',SpectrumRange)` uses the `SpectrumRange` argument to specify the interval over which the pseudospectrum was calculated. For data that ranges from [0 π) or [0 π], set the `SpectrumRange` to `half`; for data that ranges from [0 2π), set the `SpectrumRange` to `whole`.

`Hps = dspdata.pseudospectrum(...,'CenterDC',flag)` uses the value of `flag` to indicate whether the zero-frequency (DC) component is centered. If `flag` is `true`, it indicates that the DC component is in the center of the whole Nyquist range spectrum. Set the `flag` to `false` if the DC component is on the left edge of the spectrum.

Methods

Methods provide ways of performing functions directly on your `dspdata` object. You can apply a method directly on the variable you assigned to your `dspdata.pseudospectrum` object. You can use the following methods with a `dspdata.pseudospectrum` object.

- `centerdc`
- `halfrange`
- `normalizefreq`
- `plot`
- `wholerange`

For example, to normalize the frequency and set the `NormalizedFrequency` parameter to true, use

```
Hps = normalizefreq(Hps)
```

For detailed information on using the methods and plotting the pseudospectrum, see the `dspdata` reference page.

Examples

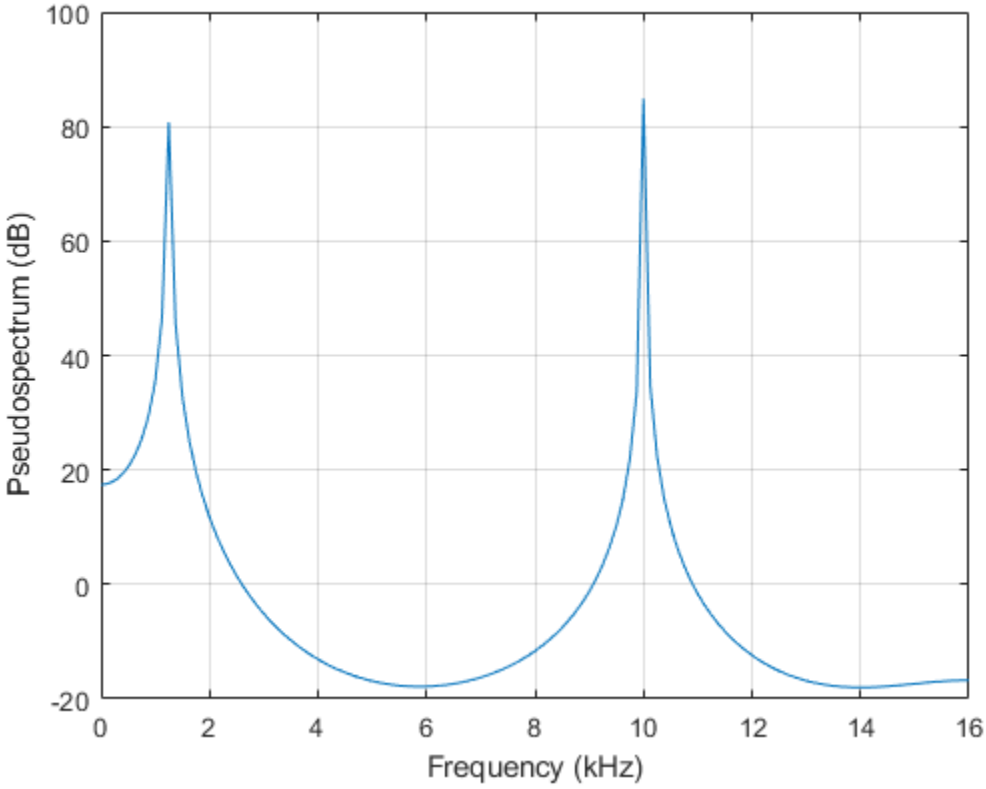
Store and Plot Pseudospectrum Data

Use eigenanalysis to estimate the pseudospectrum of a noisy sinusoidal signal with two frequency components.

```
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 a `pseudospectrum` data object to store the results. Plot the pseudospectrum.

```
hps = dspdata.pseudospectrum(P,'Fs',Fs);

plot(hps)
```



See Also

`peig` | `pmusic`

Introduced before R2006a

dspfwiz

Create Simulink filter block using Realize Model panel

Syntax

```
dspfwiz
```

Description

Note You must have the Simulink product installed to use this function.

`dspfwiz` opens **Filter Designer** with the Realize Model panel displayed.

Use other panels in **Filter Designer** to design your filter and then use the Realize Model panel to create your filter as a subsystem block, which is a combination of Add, 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

Apps
Filter Designer

Introduced before R2006a

dtw

Distance between signals using dynamic time warping

Syntax

```
dist = dtw(x,y)
[dist,ix,iy] = dtw(x,y)

[ ___ ] = dtw(x,y,maxsamp)

[ ___ ] = dtw( ___ ,metric)

dtw( ___ )
```

Description

`dist = dtw(x,y)` stretches two vectors, `x` and `y`, onto a common set of instants such that `dist`, the sum of the Euclidean distances between corresponding points, is smallest. To stretch the inputs, `dtw` repeats each element of `x` and `y` as many times as necessary. If `x` and `y` are matrices, then `dist` stretches them by repeating their columns. In that case, `x` and `y` must have the same number of rows.

`[dist,ix,iy] = dtw(x,y)` returns the common set of instants, or *warping path*, such that `x(ix)` and `y(iy)` have the smallest possible `dist` between them.

The vectors `ix` and `iy` have the same length. Each contains a monotonically increasing sequence in which the indices to the elements of the corresponding signal, `x` or `y`, are repeated the necessary number of times.

When `x` and `y` are matrices, `ix` and `iy` are such that `x(:,ix)` and `y(:,iy)` are minimally separated.

`[ ___ ] = dtw(x,y,maxsamp)` restricts the warping path to be within `maxsamp` samples of a straight-line fit between `x` and `y`. This syntax returns any of the output arguments of previous syntaxes.

`[ ___ ] = dtw( ___ ,metric)` specifies the distance metric to use in addition to any of the input arguments in previous syntaxes.

`dtw( ___ )` without output arguments plots the original and aligned signals.

- If the signals are real vectors, the function displays the two original signals on a subplot and the aligned signals in a subplot below the first one.
- If the signals are complex vectors, the function displays the original and aligned signals in three-dimensional plots.
- If the signals are real matrices, the function uses `imagesc` to display the original and aligned signals.
- If the signals are complex matrices, the function plots their real and imaginary parts in the top and bottom half of each image.

Examples

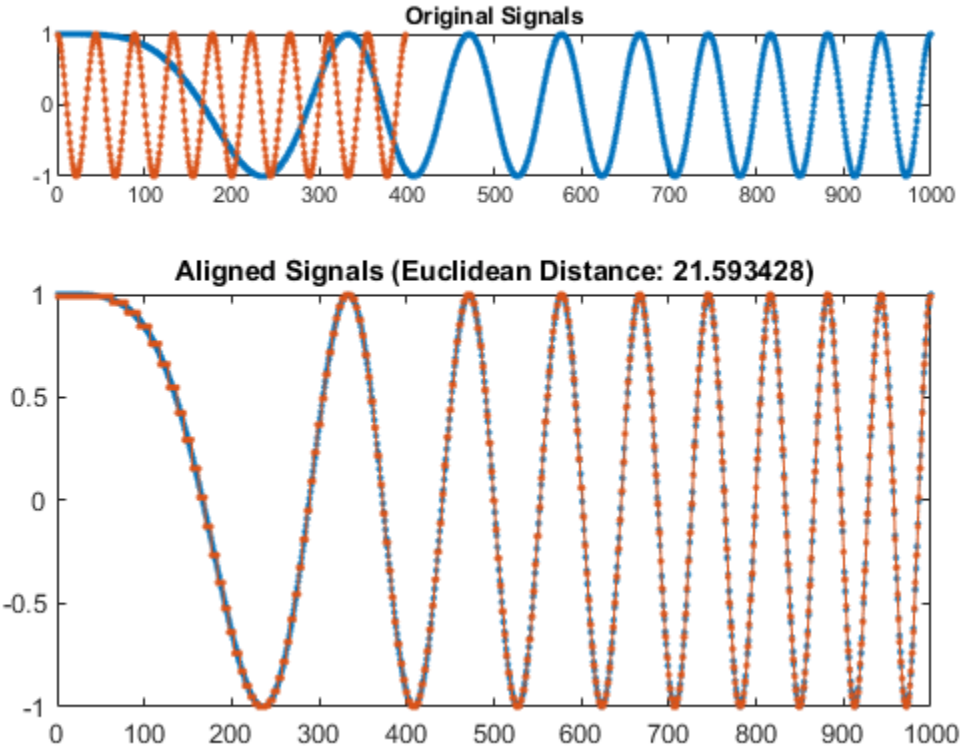
Dynamic Time Warping of Chirp and Sinusoid

Generate two real signals: a chirp and a sinusoid.

```
x = cos(2*pi*(3*(1:1000)/1000).^2);
y = cos(2*pi*9*(1:399)/400);
```

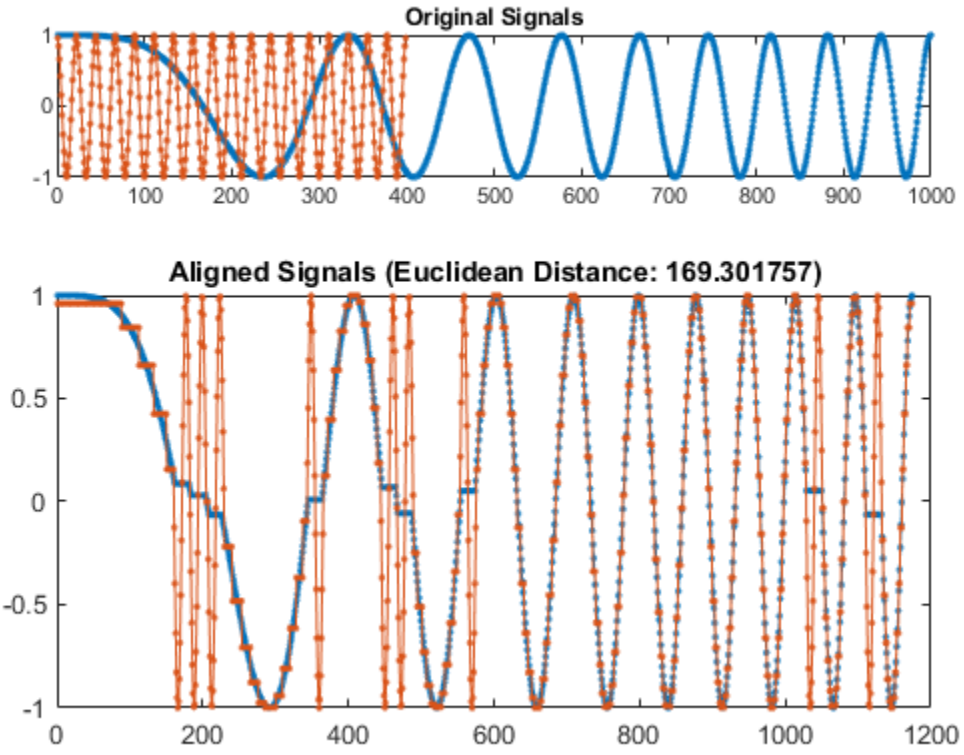
Use dynamic time warping to align the signals such that the sum of the Euclidean distances between their points is smallest. Display the aligned signals and the distance.

```
dtw(x,y);
```



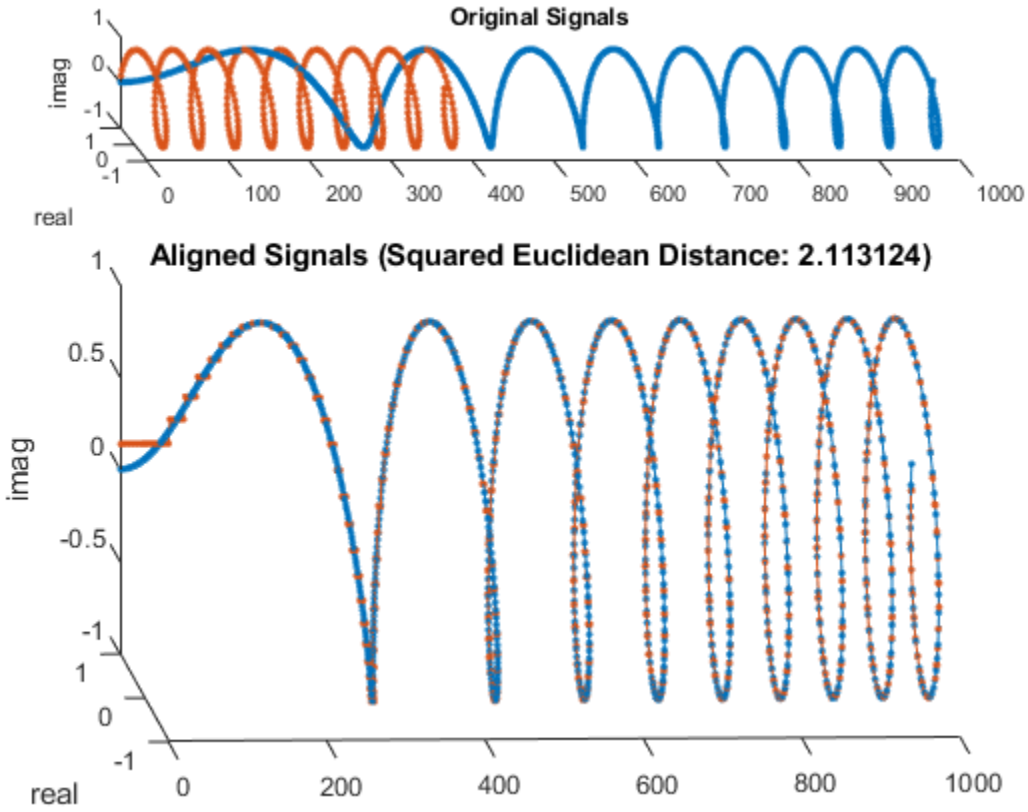
Change the sinusoid frequency to twice its initial value. Repeat the computation.

```
y = cos(2*pi*18*(1:399)/400);

dtw(x,y);
```



Add an imaginary part to each signal. Restore the initial sinusoid frequency. Use dynamic time warping to align the signals by minimizing the sum of squared Euclidean distances.

```
x = exp(2i*pi*(3*(1:1000)/1000).^2);
y = exp(2i*pi*9*(1:399)/400);

dtw(x,y,'squared');
```



Align Writing Samples

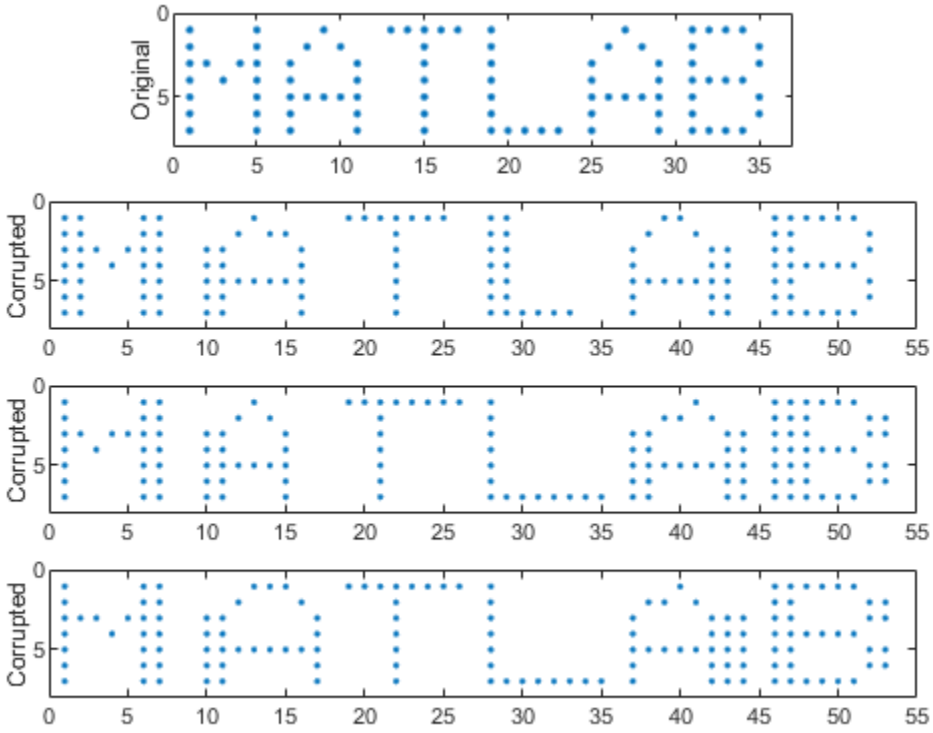
Devise a typeface that resembles the output of early computers. Use it to write the word MATLAB®.

```
chr = @(x)dec2bin(x')-48;

M = chr([34 34 54 42 34 34 34]);
A = chr([08 20 34 34 62 34 34]);
T = chr([62 08 08 08 08 08 08]);
L = chr([32 32 32 32 32 32 62]);
B = chr([60 34 34 60 34 34 60]);

MATLAB = [M A T L A B];
```

Corrupt the word by repeating random columns of the letters and varying the spacing. Show the original word and three corrupted versions. Reset the random number generator for reproducible results.

```
rng('default')

c = @(x)x(:,sort([1:6 randi(6,1,3)]));

subplot(4,1,1,'XLim',[0 60])
spy(MATLAB)
xlabel('')
```

```
ylabel('Original')

for kj = 2:4
    subplot(4,1,kj,'XLim',[0 60])
    spy([c(M) c(A) c(T) c(L) c(A) c(B)])
    xlabel('')
    ylabel('Corrupted')
end
```



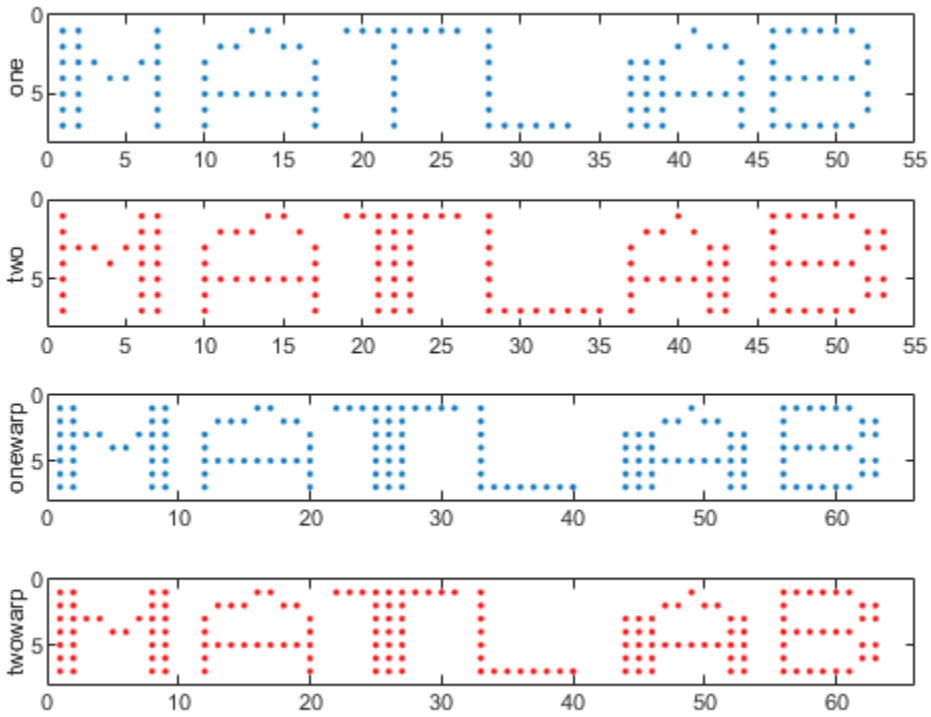
Generate two more corrupted versions of the word. Align them using dynamic time warping.

```
one = [c(M) c(A) c(T) c(L) c(A) c(B)];
two = [c(M) c(A) c(T) c(L) c(A) c(B)];

[ds,ix,iy] = dtw(one,two);

onewarp = one(:,ix);
twowarp = two(:,iy);
```

Display the unaligned and aligned words.

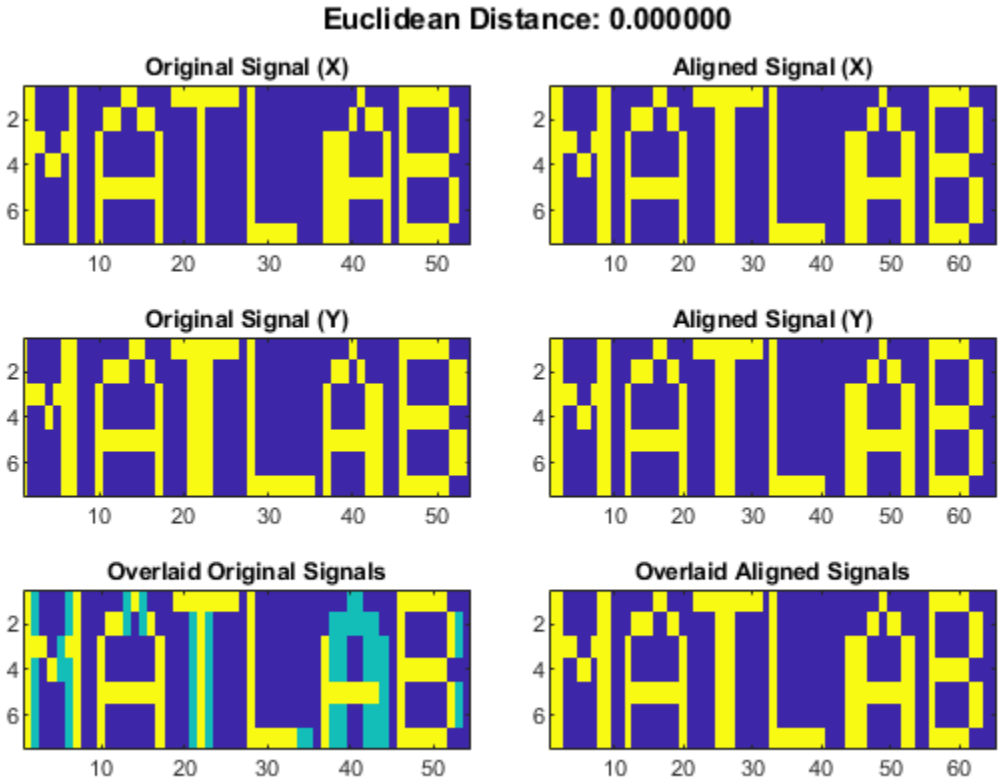
```
figure

subplot(4,1,1)
spy(one)
xlabel('')
ylabel('one')
```

```
subplot(4,1,2)
spy(two,'r')
xlabel('')
ylabel('two')

subplot(4,1,3)
spy(onewarp)
xlabel('')
ylabel('onewarp')

subplot(4,1,4)
spy(twowarp,'r')
xlabel('')
ylabel('twowarp')
```



Repeat the computation using the built-in functionality of `dtw`.

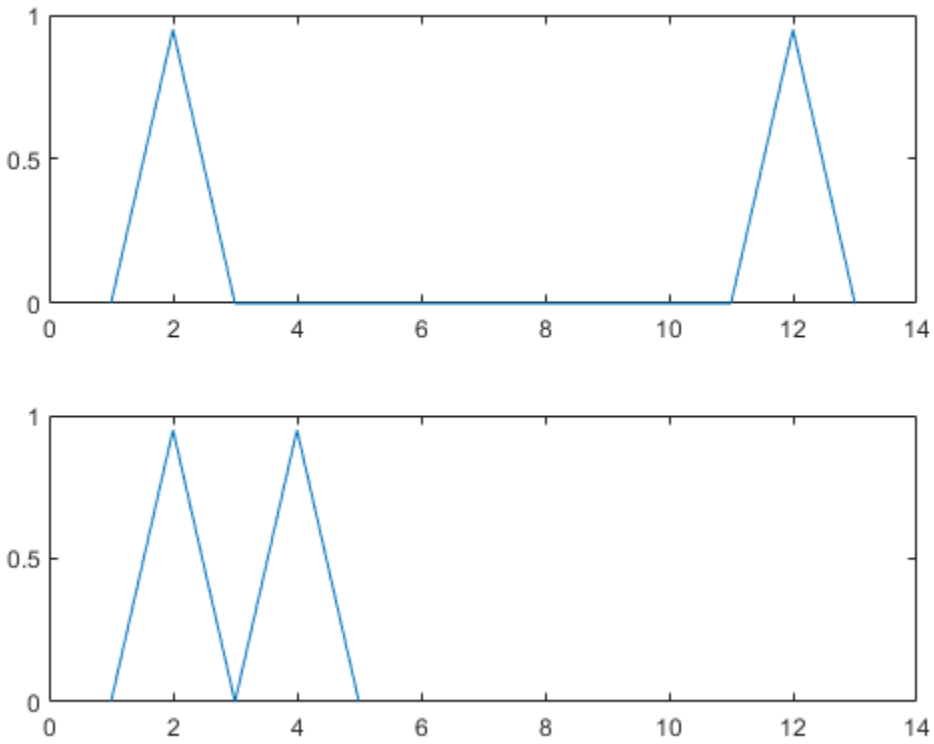
```
dtw(one,two);
```



Constrained Warping Path

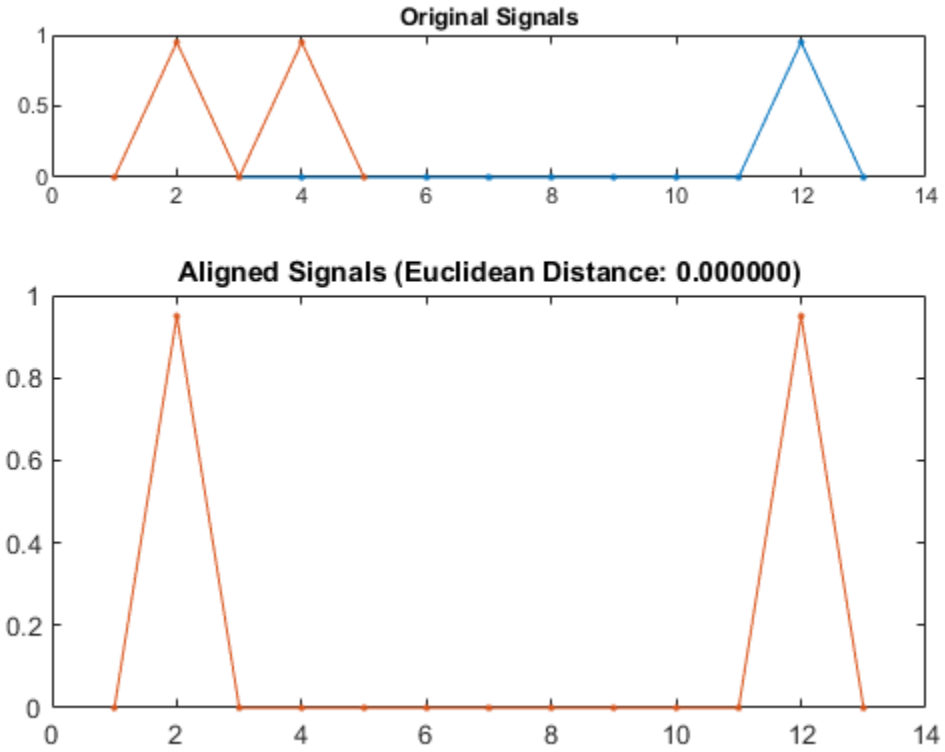
Generate two signals consisting of two distinct peaks separated by valleys of different lengths. Plot the signals.

```
x1 = [0 1 0 0 0 0 0 0 0 0 0 1 0]*.95;
x2 = [0 1 0 1 0]*.95;

subplot(2,1,1)
plot(x1)
xl = xlim;
subplot(2,1,2)
plot(x2)
xlim(xl)
```



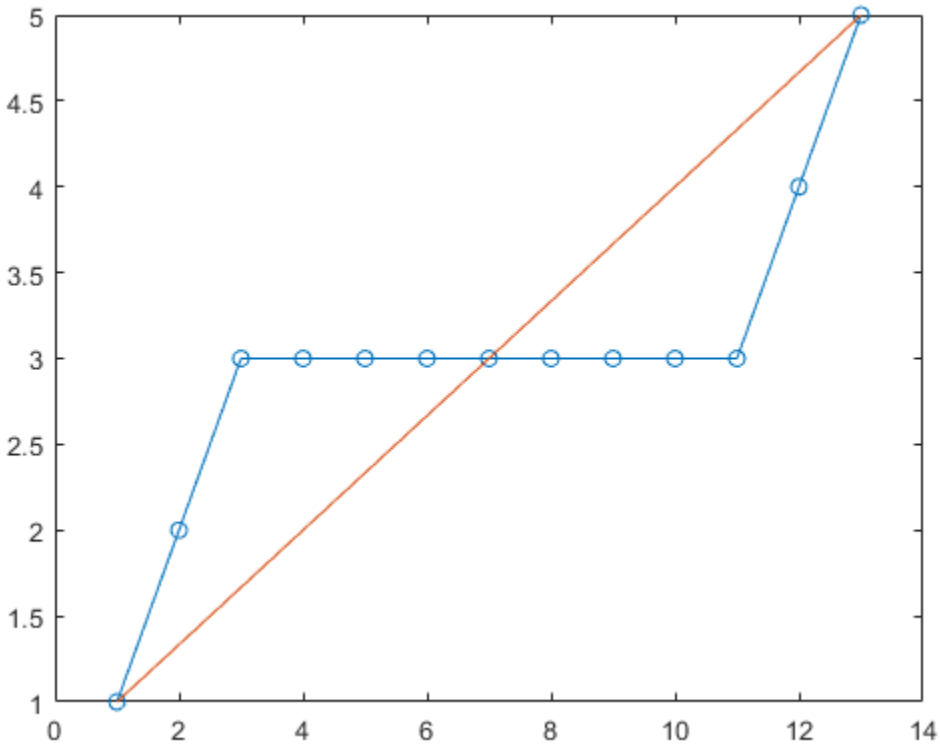
Align the signals with no restriction on the warping path. To produce perfect alignment, the function needs to repeat only one sample of the shorter signal.

```
figure
dtw(x1,x2);
```



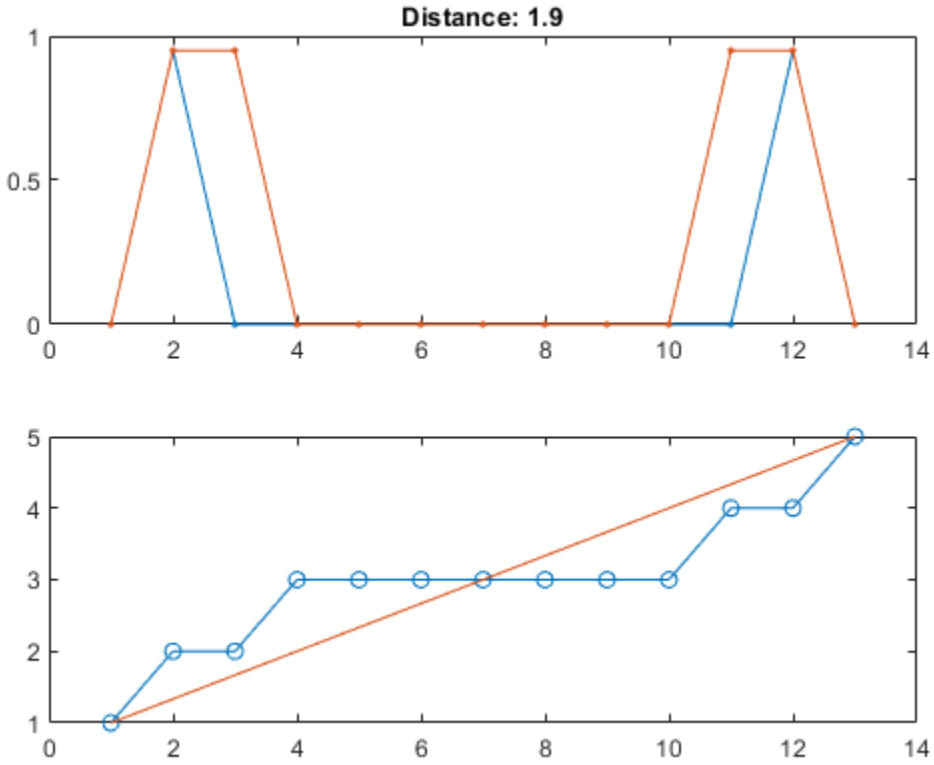
Plot the warping path and the straight-line fit between the two signals. To achieve alignment, the function expands the trough between the peaks generously.

```
[d,i1,i2] = dtw(x1,x2);

figure
plot(i1,i2,'o-',[i1(1) i1(end)],[i2(1) i2(end)])
```



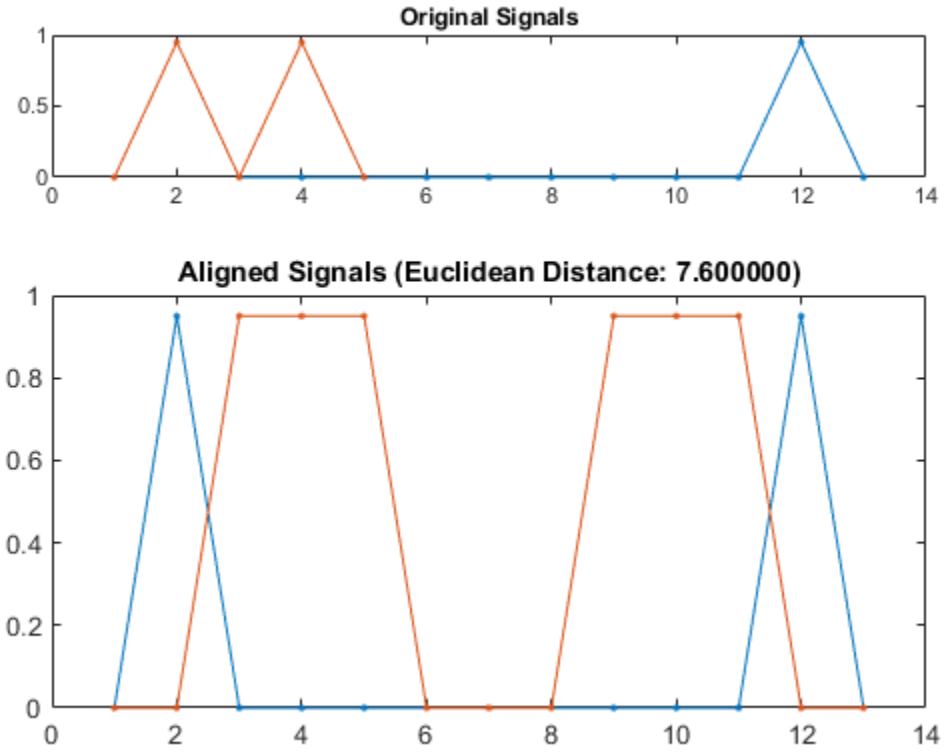
Repeat the computation, but now constrain the warping path to deviate at most three elements from the straight-line fit. Plot the stretched signals and the warping path.

```
[dc,i1c,i2c] = dtw(x1,x2,3);

subplot(2,1,1)
plot([x1(i1c);x2(i2c)]','.-')
title(['Distance: ' num2str(dc)])
subplot(2,1,2)
plot(i1c,i2c,'o-',[i1(1) i1(end)],[i2(1) i2(end)])
```



The constraint precludes the warping from concentrating too much on a small subset of samples, at the expense of alignment quality. Repeat the calculation with a one-sample constraint.

```
dtw(x1,x2,1);
```



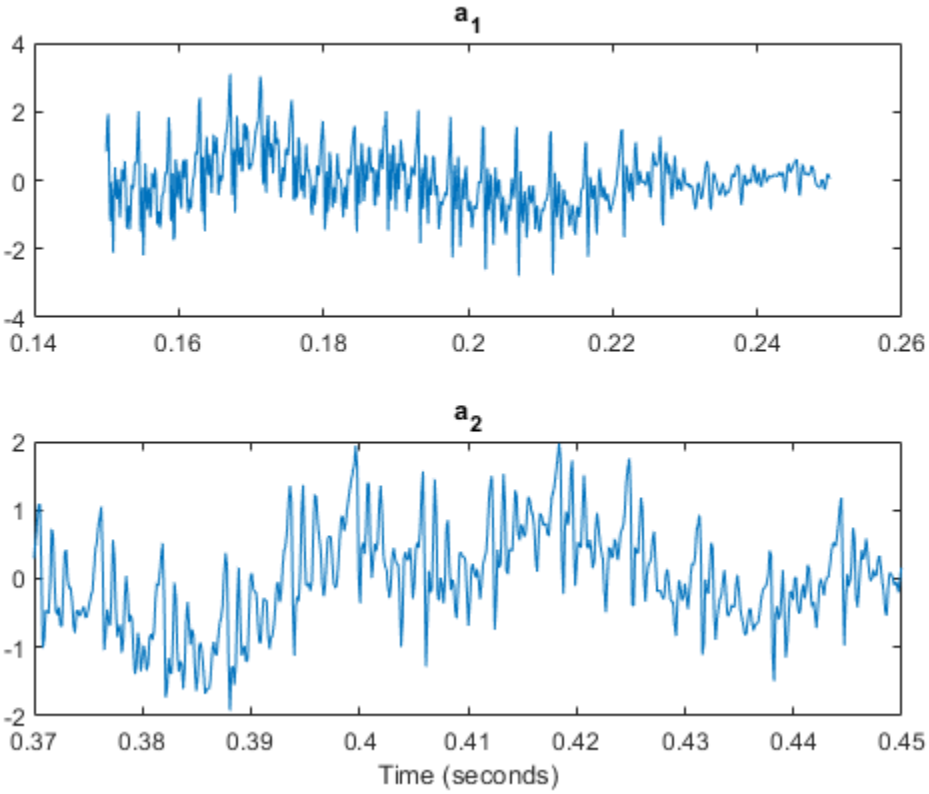
Dynamic Time Warping of Speech Signals

Load a speech signal sampled at F_s = 7418 Hz. The file contains a recording of a female voice saying the word "MATLAB®."

```
load mtlb

% To hear, type soundsc(mtlb,Fs)
```

Extract the two segments that correspond to the two instances of the /æ/ phoneme. The first one occurs roughly between 150 ms and 250 ms, and the second one between 370 ms and 450 ms. Plot the two waveforms.

```
a1 = mtlb(round(0.15*Fs):round(0.25*Fs));
a2 = mtlb(round(0.37*Fs):round(0.45*Fs));

subplot(2,1,1)
plot((0:numel(a1)-1)/Fs+0.15,a1)
title('a_1')
subplot(2,1,2)
plot((0:numel(a2)-1)/Fs+0.37,a2)
title('a_2')
xlabel('Time (seconds)')
```



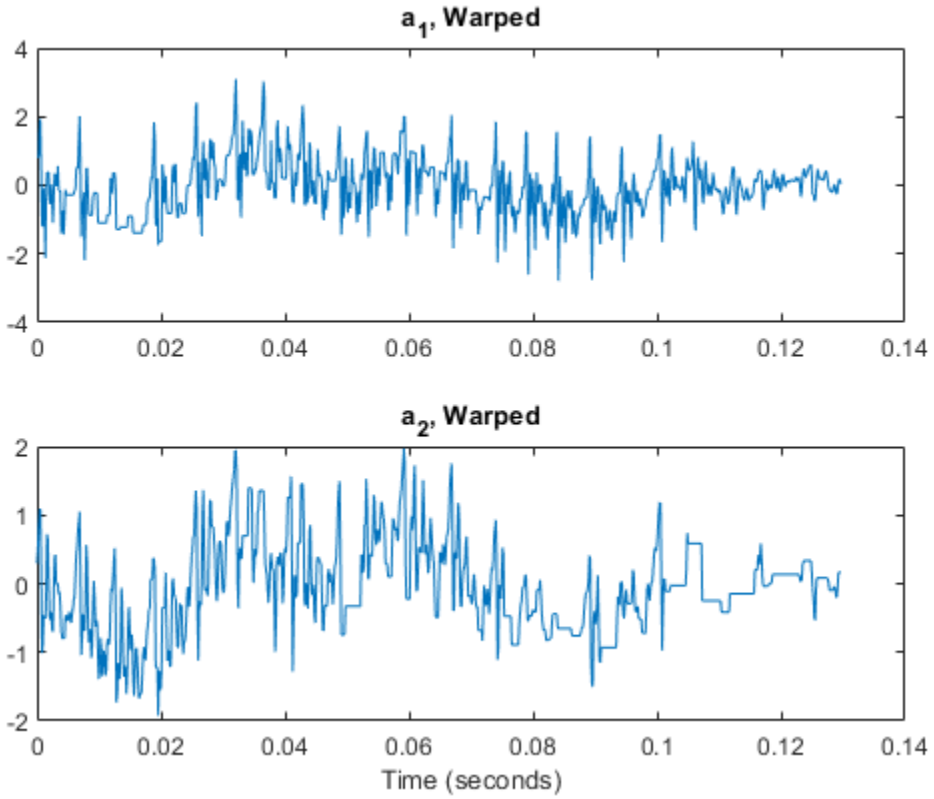

```
% To hear, type soundsc(a1,Fs), pause(1), soundsc(a2,Fs)
```

Warp the time axes so that the Euclidean distance between the signals is minimized. Compute the shared "duration" of the warped signals and plot them.

```
[d,i1,i2] = dtw(a1,a2);

a1w = a1(i1);
a2w = a2(i2);

t = (0:numel(i1)-1)/Fs;
duration = t(end)
```

```
duration = 0.1297
```

```
subplot(2,1,1)
plot(t,a1w)
title('a_1, Warped')
subplot(2,1,2)
plot(t,a2w)
title('a_2, Warped')
xlabel('Time (seconds)')
```



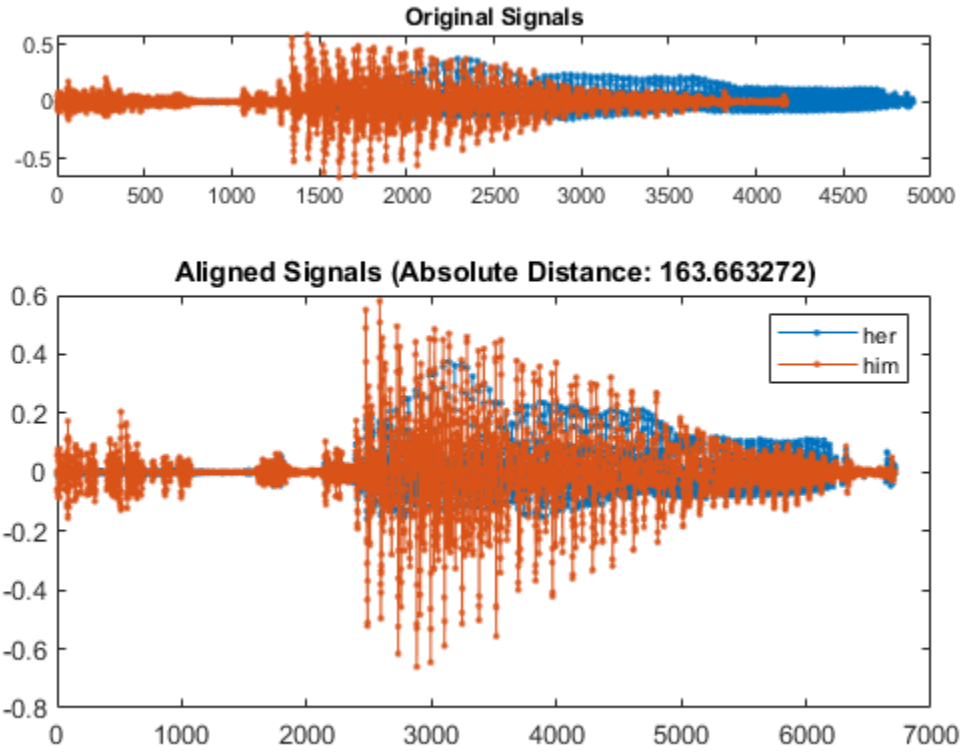

```
% To hear, type soundsc(a1w,Fs), pause(1), sound(a2w,Fs)
```

Repeat the experiment with a complete word. Load a file containing the word "strong," spoken by a woman and by a man. The signals are sampled at 8 kHz.

```
load('strong.mat')

% To hear, type soundsc(her,fs), pause(2), soundsc(him,fs)
```

Warp the time axes so that the absolute distance between the signals is minimized. Plot the original and transformed signals. Compute their shared warped "duration."

```
dtw(her,him,'absolute');
legend('her','him')
```




```
[d,iher,ihim] = dtw(her,him,'absolute');
duration = numel(iher)/fs
```

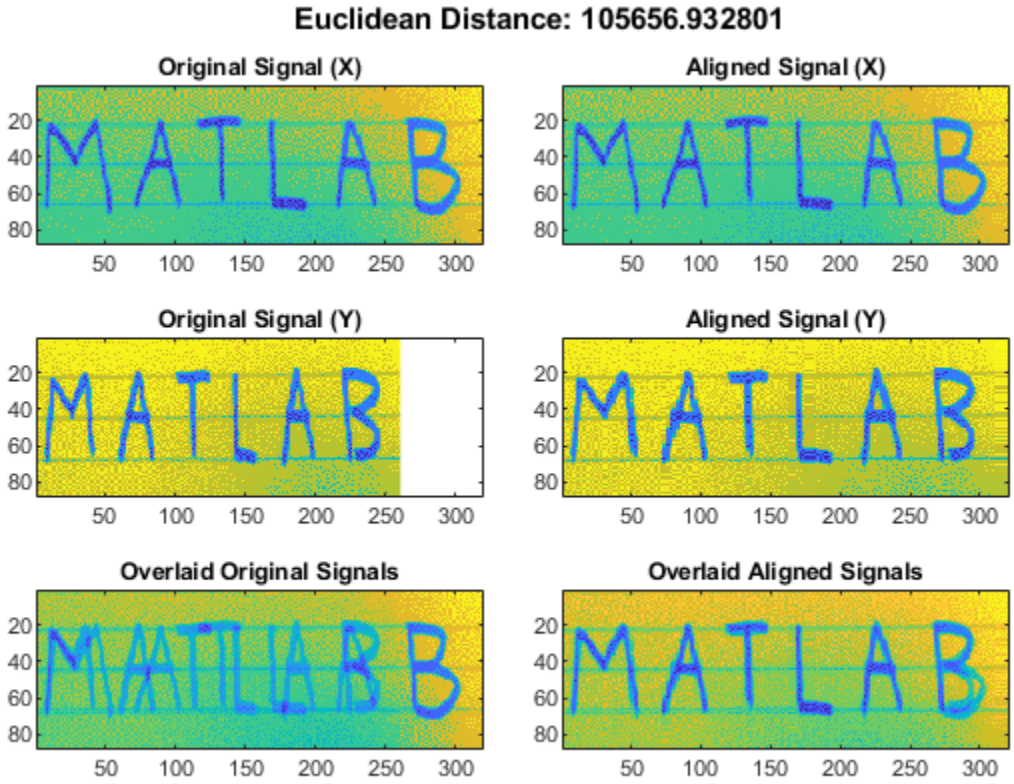
```
duration = 0.8394
```

```
% To hear, type soundsc(her(iher),fs), pause(2), soundsc(him(ihim),fs)
```

Dynamic Time Warping for Handwriting Alignment

The files `MATLAB1.gif` and `MATLAB2.gif` contain two handwritten samples of the word "MATLAB®." Load the files and align them along the *x*-axis using dynamic time warping.

```
samp1 = 'MATLAB1.gif';
samp2 = 'MATLAB2.gif';

x = double(imread(samp1));
y = double(imread(samp2));

dtw(x,y);
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a real or complex vector or matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

y — Input signal

vector | matrix

Input signal, specified as a real or complex vector or matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

maxsamp — Width of adjustment window

`Inf` (default) | positive integer

Width of adjustment window, specified as a positive integer.

Data Types: `single` | `double`

`metric` — Distance metric

`'euclidean'` (default) | `'absolute'` | `'squared'` | `'symmkl'`

Distance metric, specified as `'euclidean'`, `'absolute'`, `'squared'`, or `'symmkl'`. If **X** and **Y** are both *K*-dimensional signals, then `metric` prescribes $d_{mn}(\mathbf{X},\mathbf{Y})$, the distance between the *m*th sample of **X** and the *n*th sample of **Y**. See “Dynamic Time Warping” on page 1-447 for more information about $d_{mn}(\mathbf{X},\mathbf{Y})$.

- `'euclidean'` — Root sum of squared differences, also known as the Euclidean or ℓ_2 metric:

$$d_{mn}(\mathbf{X},\mathbf{Y}) = \sqrt{\sum_{k=1}^{K} (x_{k,m} - y_{k,n})^{*}(x_{k,m} - y_{k,n})}$$

- `'absolute'` — Sum of absolute differences, also known as the Manhattan, city block, taxicab, or ℓ_1 metric:

$$d_{mn}(\mathbf{X},\mathbf{Y}) = \sum_{k=1}^{K} \left|x_{k,m} - y_{k,n}\right| = \sum_{k=1}^{K} \sqrt{(x_{k,m} - y_{k,n})^{*}(x_{k,m} - y_{k,n})}$$

- `'squared'` — Square of the Euclidean metric, consisting of the sum of squared differences:

$$d_{mn}(\mathbf{X},\mathbf{Y}) = \sum_{k=1}^{K} (x_{k,m} - y_{k,n})^{*}(x_{k,m} - y_{k,n})$$

- `'symmkl'` — Symmetric Kullback-Leibler metric. This metric is valid only for real and positive **X** and **Y**:

$$d_{mn}(\mathbf{X},\mathbf{Y}) = \sum_{k=1}^{K} (x_{k,m} - y_{k,n})(\log x_{k,m} - \log y_{k,n})$$

Output Arguments

`dist` — Minimum distance

positive real scalar

Minimum distance between signals, returned as a positive real scalar.

`ix` — Warping path for first signal

vector of indices | matrix of indices

Warping path for first signal, returned as a vector or matrix of indices.

`iy` — Warping path for second signal

vector of indices | matrix of indices

Warping path for second signal, returned as a vector or matrix of indices.

More About

Dynamic Time Warping

Two signals with equivalent features arranged in the same order can appear very different due to differences in the durations of their sections. Dynamic time warping distorts these durations so that

the corresponding features appear at the same location on a common time axis, thus highlighting the similarities between the signals.

Consider the two K-dimensional signals

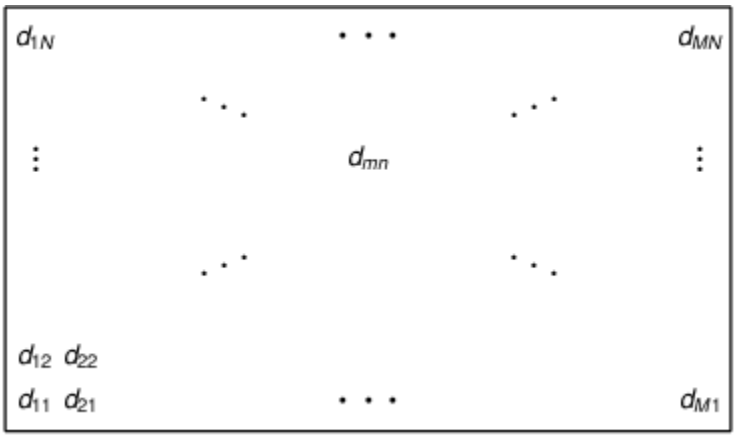
$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,M} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{K,1} & x_{K,2} & \cdots & x_{K,M} \end{bmatrix}$$

and

$$\mathbf{Y} = \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,N} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ y_{K,1} & y_{K,2} & \cdots & y_{K,N} \end{bmatrix},$$

which have M and N samples, respectively. Given $d_{mn}(\mathbf{X},\mathbf{Y})$, the distance between the mth sample of $\mathbf{X}$ and the nth sample of $\mathbf{Y}$ specified in `metric`, `dist` stretches $\mathbf{X}$ and $\mathbf{Y}$ onto a common set of instants such that a global signal-to-signal distance measure is smallest.

Initially, the function arranges all possible values of $d_{mn}(\mathbf{X},\mathbf{Y})$ into a lattice of the form



Then `dist` looks for a path through the lattice—parameterized by two sequences of the same length, `ix` and `iy`—such that

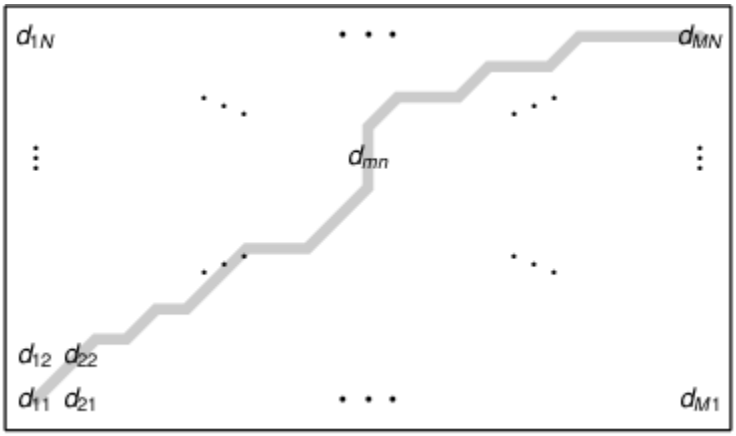
$$d = \sum_{\substack{m \in \text{ix} \\ n \in \text{iy}}} d_{mn}(\mathbf{X},\mathbf{Y})$$

is minimum. Acceptable `dist` paths start at $d_{11}(\mathbf{X},\mathbf{Y})$, end at $d_{MN}(\mathbf{X},\mathbf{Y})$, and are combinations of “chess king” moves:

- Vertical moves: $(m,n) \rightarrow (m+1,n)$
- Horizontal moves: $(m,n) \rightarrow (m,n+1)$
- Diagonal moves: $(m,n) \rightarrow (m+1,n+1)$

This structure ensures that any acceptable path aligns the complete signals, does not skip samples, and does not repeat signal features. Additionally, a desirable path runs close to the diagonal line extended between $d_{11}(\mathbf{X},\mathbf{Y})$ and $d_{MN}(\mathbf{X},\mathbf{Y})$. This extra constraint, adjusted by the `maxsamp` argument, ensures that the warping compares sections of similar length and does not overfit outlier features.

This is a possible path through the lattice:



References

[1] Paliwal, K. K., Anant Agarwal, and Sarvajit S. Sinha. "A Modification over Sakoe and Chiba's Dynamic Time Warping Algorithm for Isolated Word Recognition." *Signal Processing*. Vol. 4, 1982, pp. 329–333.

[2] Sakoe, Hiroaki, and Seibi Chiba. "Dynamic Programming Algorithm Optimization for Spoken Word Recognition." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. ASSP-26, No. 1, 1978, pp. 43–49.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`alignsignals` | `edr` | `finddelay` | `findsignal` | `xcorr`

Introduced in R2016a

dutycycle

Duty cycle of pulse waveform

Syntax

```
d = dutycycle(x)
d = dutycycle(x,fs)
d = dutycycle(x,t)

[d,initcross,finalcross,nextcross,midlev] = dutycycle( ___ )

[ ___ ] = dutycycle( ___ ,Name,Value)

dutycycle( ___ )

d = dutycycle(tau,prf)
```

Description

`d = dutycycle(x)` returns the ratio of pulse width to pulse period for each positive-polarity pulse. The function identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state. To determine the transitions that define each pulse, `dutycycle` estimates the state levels of `x` by a histogram method. The low-state and high-state boundaries are expressed as the state level plus or minus a scalar multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-457 for more details.

`d = dutycycle(x,fs)` specifies the sample rate at which `x` is sampled. The first sample instant of `x` corresponds to $t = 0$.

`d = dutycycle(x,t)` specifies the instants, `t`, at which `x` is sampled.

`[d,initcross,finalcross,nextcross,midlev] = dutycycle( ___ )` with any input arguments from previous syntaxes also 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`.
- 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`.
- A vector, `nextcross`, whose elements correspond to the mid-crossings (mid-reference level instants) of the next detected transition for each pulse.
- A scalar, `midlev`, that corresponds to the mid-reference level.

`[ ___ ] = dutycycle( ___ ,Name,Value)` returns the ratio of pulse width to pulse period with additional options specified by one or more `Name,Value` pair arguments.

`dutycycle( ___ )` plots the waveform, the location of the mid-reference level instants, the associated reference levels, the state levels, and the associated lower and upper state boundaries.

`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`.

Examples

Duty Cycle of Bilevel Waveform

Determine the duty cycle of a bilevel waveform. Use the vector indices as the sample instants.

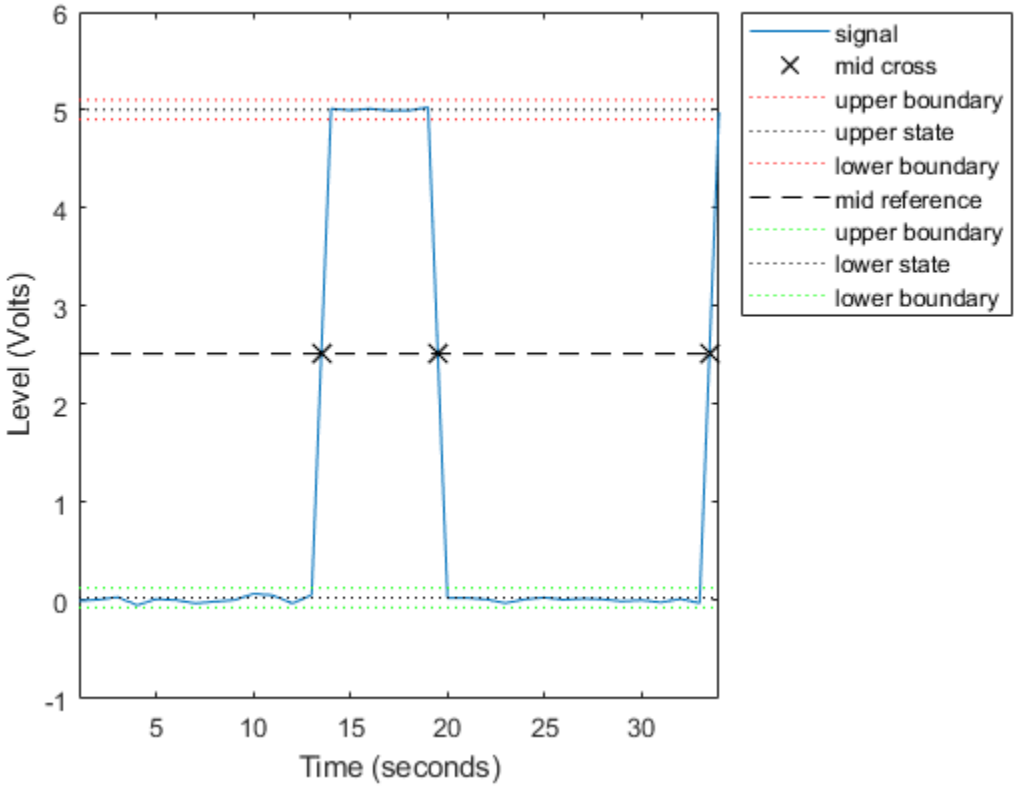
```
load('pulseex.mat','x')
```

```
d = dutycycle(x)
```

```
d = 0.3001
```

Annotate the result on a plot of the waveform.

```
dutycycle(x);
```



Duty Cycle of Bilevel Waveform with Sample Rate

Determine the duty cycle of a bilevel waveform. The sample rate is 4 MHz.

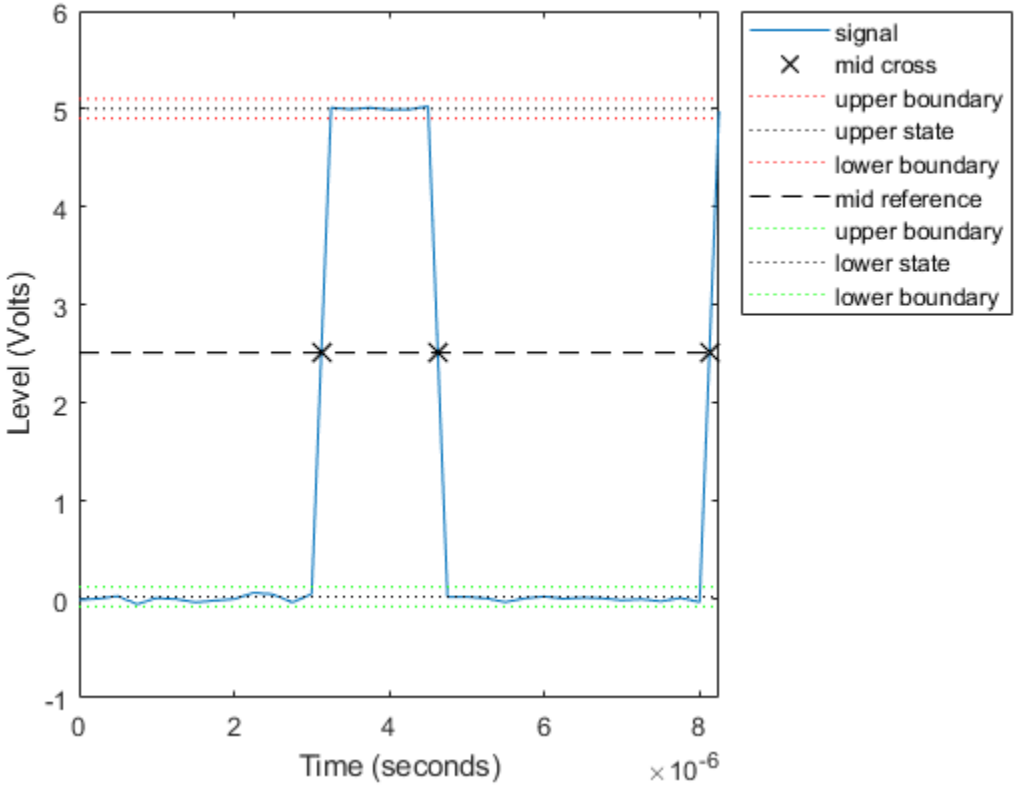
```
load('pulseex.mat','x','t')
fs = 1/(t(2)-t(1));
```

```
d = dutycycle(x,fs)
```

```
d = 0.3001
```

Annotate the result on a plot of the waveform.

```
dutycycle(x,fs);
```



Duty Cycle of Bilevel Waveform with Three Pulses

Create a pulse waveform with three pulses. The sample rate is 4 MHz. Determine the initial and final mid-reference level instants. Plot the result.

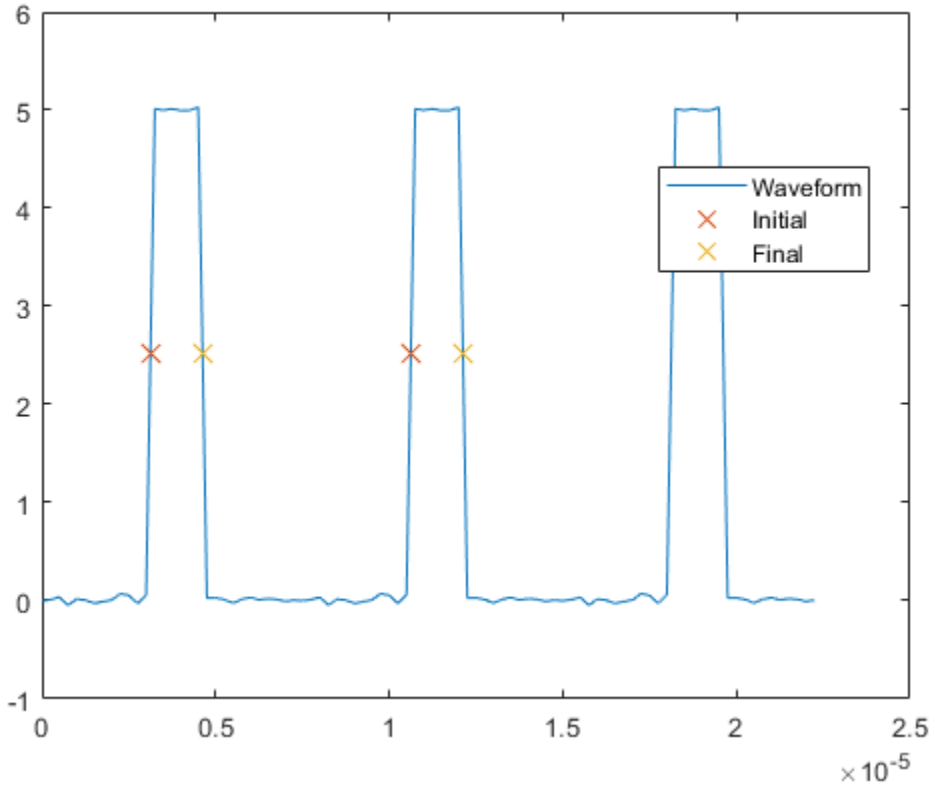
```
load('pulseex.mat','x')
fs = 4e6;

pulse = x(1:30);
wavef = [pulse;pulse;pulse];
t = (0:length(wavef)-1)/fs;

[~,initcross,finalcross,~,midlev] = dutycycle(wavef,t)
```

```
initcross = 2×1
10^-4 ×

    0.0312
    0.1062
```

```
finalcross = 2×1
10^-4 ×

    0.0463
    0.1213


midlev = 2.5177
```

Even though there are three pulses, only two pulses have corresponding subsequent transitions. Plot the result.

```
plot(t,wavef)
hold on
plot([initcross finalcross],midlev*ones(2),'x','MarkerSize',10)
hold off
legend('Waveform','Initial','Final','Location','best')
```



Input Arguments

x — Bilevel waveform

real-valued vector

Bilevel waveform, specified as a real-valued vector.

Example: `pulstran(0:0.1:10,1:2:9,@rectpuls)` specifies a bilevel waveform containing five one-second pulses.

Data Types: `double`

`fs` — Sample rate
1 Hz (default) | positive scalar

Sample rate, specified as a positive scalar expressed in Hz.

Data Types: `double`

`t` — Sample instants
vector

Sample instants, specified as a vector of the same length as `x`.

Data Types: `double`

`tau`, `prf` — Pulse width and repetition frequency
scalars

Pulse width and repetition frequency, specified as scalars. Express the pulse width in seconds and the repetition frequency in pulses per second. The product of `tau` and `prf` must be less than or equal to one.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MidPercentReferenceLevel',90,'Tolerance',0.5` specifies that the mid-reference level is 90% of the waveform amplitude and the tolerance around the lower- and upper-state boundaries is 0.5%.

`MidPercentReferenceLevel` — Mid-reference level
50 (default) | positive scalar

Mid-reference level, specified as the comma-separated pair consisting of `'MidPercentReferenceLevel'` and a positive scalar expressed as a percentage of the waveform amplitude.

Data Types: `double`

`Polarity` — Pulse polarity
`'positive'` (default) | `'negative'`

Pulse polarity, specified as the comma-separated pair consisting of `'Polarity'` and either `'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 1-456 for examples of positive and negative-polarity pulses.

Data Types: `char`

`StateLevels` — Low- and high-state levels
1-by-2 real-valued vector

Low- and high-state levels, specified as the comma-separated pair consisting of `'StateLevels'` and 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 a histogram method.

Data Types: `double`

`Tolerance` — Tolerance levels
2 (default) | positive scalar

Tolerance levels (lower- and upper-state boundaries), specified as the comma-separated pair consisting of `'Tolerance'` and a positive scalar expressed as a percentage. See “State-Level Tolerances” on page 1-457 for more information.

Data Types: `double`

Output Arguments

`d` — Duty cycle
vector | scalar

Duty cycle, returned as a vector or scalar. The elements of `d` correspond to the ratio of pulse width to pulse period for each pulse in `x`. `d` obeys $0 \le d \le 1$ because the pulse width cannot exceed the pulse period. `d` has length equal to the number of pulse periods in `x`. If you specify `tau` and `prf` as arguments, `d` is a scalar.

`initcross` — Mid-reference level instant of initial transition
vector

Mid-reference level instants of initial transitions, returned as a vector. The elements of `initcross` correspond to the mid-crossings (mid-reference level instants) of the initial transition of each pulse with a corresponding `nextcross`.

`finalcross` — Mid-reference level instant of final transition
vector

Mid-reference level instants of final transitions, returned as a vector. The elements of `finalcross` correspond to the mid-crossings (mid-reference level instants) of the final transition of each pulse with a corresponding `nextcross`.

`nextcross` — Next transition mid-crossing
vector

Next transition mid-crossing, returned as a vector. The elements of `nextcross` correspond to the mid-crossings (mid-reference level instants) of the next detected transition for each pulse.

`midlev` — Mid-reference level
scalar

Mid-reference level waveform value, returned as a scalar. `midlevel` is a scalar because in a bilevel pulse waveform the state levels are constant.

More About

Duty Cycle

The duty cycle of a bilevel pulse is the ratio of average power to peak power.

The energy in a bilevel, or rectangular, pulse is equal to the product of the peak power, P_t, and the pulse width, τ. 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.

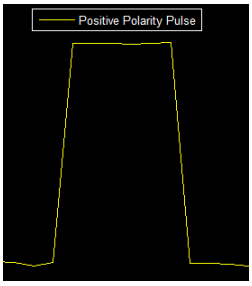
The ratio of average power to peak power is the duty cycle:

$$D = \frac{P_t \tau / T}{P_t}$$

Pulse Polarity

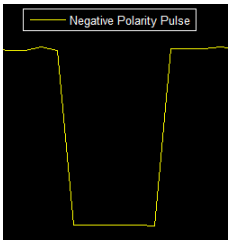
The polarity of a pulse is defined by the direction of its initial transition.

If the pulse has a positive-going initial transition, the pulse has positive polarity. This 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. This figure shows a negative-polarity pulse:



Equivalently, a negative-polarity (negative-going) pulse has an originating state more positive than the terminating state.

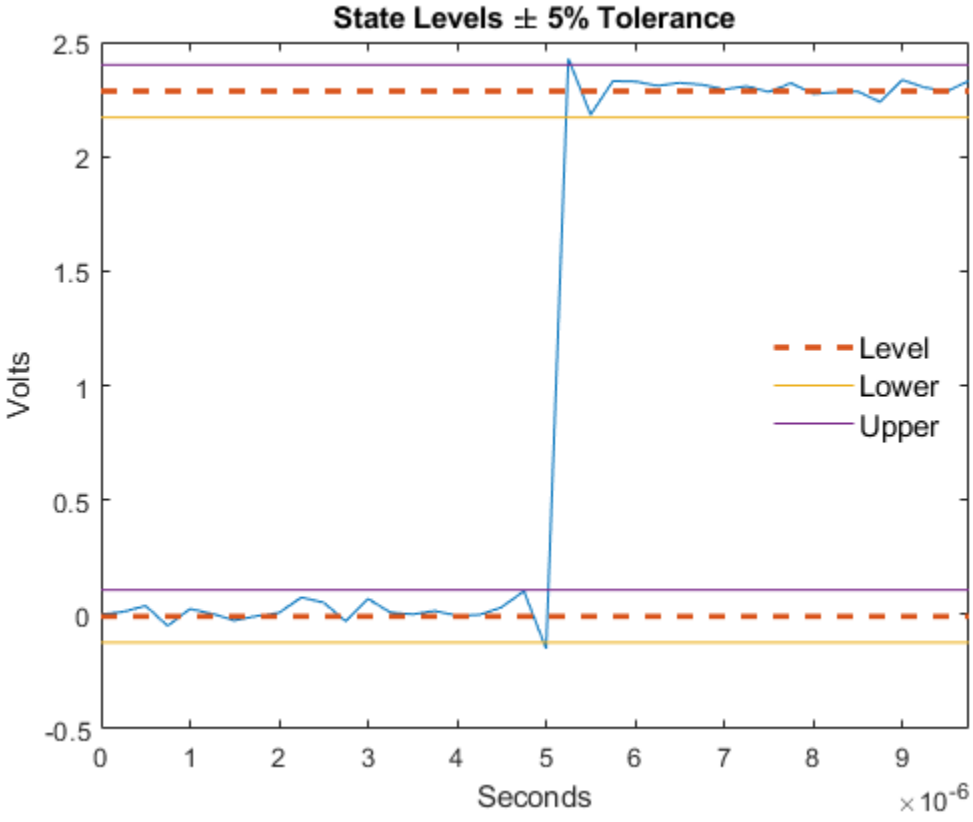
State-Level Tolerances

You can specify lower- and upper-state boundaries for each state level. Define the boundaries as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, specify the scalar as a small number such as 2/100 or 3/100. In general, the $\alpha\%$ 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.

This figure shows lower and upper 5% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The thick dashed lines indicate the estimated state levels.



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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the first input to `dutycycle` is a variable-sized vector during code generation, then it must not reduce to a scalar at runtime.

Thread-Based Environment

Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

Usage notes and limitations:

- The syntax with no output arguments is not supported.

For more information, see “Run MATLAB Functions in Thread-Based Environment”.

See Also

`midcross` | `pulseperiod` | `pulsesep` | `pulsewidth`

Introduced in R2012a

edfinfo

Get information about EDF/EDF+ file

Description

Create an `edfinfo` object to get information about a European Data Format (EDF) or EDF+ file. `edfinfo` objects contain information such as file size, number of data records, number of signals, and number of samples.

Creation

Syntax

```
info = edfinfo(filename)
```

Description

`info = edfinfo(filename)` returns an `edfinfo` object for the EDF or EDF+ file specified by `filename`.

Input Arguments

`filename` — Name of EDF or EDF+ file

character vector | string scalar

Name of EDF or EDF+ file, specified as a character vector or string scalar.

Depending on the location of the file, `filename` can take one of these forms.

Location	Form
Current folder or folder on the MATLAB path	Specify the name of the file in `filename`. **Example:** `'data.edf'`
File in a folder	If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name. **Example:** `'C:\myFolder\data.edf'` **Example:** `'myDir\myFile.ext'`

Note `edfinfo` does not support EyeLink® EDF files.

Data Types: `char` | `string`

Properties

File Properties

`Filename` — File name
string scalar

This property is read-only.

File name, returned as a string scalar.

Example: `"ecg_20200411_120.edf"`

Data Types: `string`

`FileModDate` — Date last modified
string scalar

This property is read-only.

Date last modified, returned as a string scalar with the date and time the file was last modified.

Example: `"11-Apr-2020 15:38:37"`

`FileSize` — File size in bytes
integer scalar

This property is read-only.

File size in bytes, returned as an integer scalar.

Example: `4040992`

Data Types: `double`

Header Properties

`Version` — Data format version
`"0"`

This property is read-only.

Data format version, returned as `"0"`.

Data Types: `string`

`Patient` — Patient identification details
string scalar

This property is read-only.

Patient identification details, returned as a string scalar. Patient identification details can include Patient ID, sex or gender, birth date in `'dd-MMM-yyyy'` format, and name.

Example: `"X F X 120 04-JUL-1982"`

Data Types: `string`

`Recording` — Recording identification details
string scalar

This property is read-only.

Recording identification details, returned as a string scalar. Recording identification details may include its start date and time, the ID of the technician that made the recording, and the ID of the equipment that made the recording.

Example: `"Startdate 04-JUL-1982 X X X"`

Data Types: `string`

`StartDate` — Recording start date
string scalar

This property is read-only.

Recording start date, returned as a string scalar in `'dd.MM.yy'` format.

Example: `"04.07.82"`

Data Types: `string`

`StartTime` — Recording start time
string scalar

This property is read-only.

Recording start time, returned as a string scalar in `'HH.mm.ss'` format.

Example: `"17.16.37"`

Data Types: `string`

`HeaderBytes` — Header size in bytes
integer scalar

This property is read-only.

Header size in bytes, returned as an integer scalar. `HeaderBytes` is given by (256 + `NumSignals` × 256) bytes. The first 256 bytes correspond to a static header and are required for all EDF and EDF+ files. The other bytes depend on the number of signals present in the data records.

Example: `2048`

Data Types: `double`

`Reserved` — EDF+ interruption information
`"EDF+C"` | `"EDF+D"` | `""`

This property is read-only.

EDF+ interruption information, returned as `"EDF+C"` or `"EDF+D"` for EDF+ compliant files.

- `"EDF+C"` — The recording is continuous: There are no interruptions and all data records are contiguous, such that the start time of each data record coincides with the start time of the previous record plus its duration.

- `"EDF+D"` — The recording is discontinuous with interruptions between consecutive data records.

For files that are not EDF+ compliant, this property is an empty string (`""`).

Data Types: `string`

`NumDataRecords` — Number of data records in file
integer scalar

This property is read-only.

Number of data records in file, returned as an integer scalar.

Note If `filename` is not EDF compliant, `NumDataRecords` can be set to `-1` when the number of data records is unknown. If `filename` is EDF compliant, `NumDataRecords` must be set to a positive integer. If `filename` has `Reserved` set to a nonempty string and `NumDataRecords` set to `-1`, `edfinfo` throws an error.

Data Types: `double`

`DataRecordDuration` — Duration of each data record
duration scalar

This property is read-only.

Duration of each data record, returned as a duration scalar.

Data Types: `duration`

`NumSignals` — Number of signals in file
integer scalar

This property is read-only.

Number of signals in file, returned as an integer scalar.

Data Types: `double`

Signal Record Properties

`SignalLabels` — Signal names
string vector

This property is read-only.

Signal names, returned as a string vector of length `NumSignals`.

```
["Thorax 1";"Abdomen 3"]
```

Data Types: `string`

`TransducerTypes` — Transducer details
string vector

This property is read-only.

Transducer details, returned as a string vector of length `NumSignals`. Each element of `TransducerTypes` contains details about the transducer used to obtain the corresponding signal in `SignalLabels`.

Example: `["AgAgCl electrodes";"thermistor"]`

Data Types: `string`

`PhysicalDimensions` — Signal data units
string vector

This property is read-only.

Signal data units, returned as a string vector of length `NumSignals`. Each element of `PhysicalDimensions` contains the measurement units used to express the values of the corresponding signal in `SignalLabels`.

Example: `["uV";"mV"]`

Data Types: `string`

`PhysicalMin` — Signal minimum physical value
numeric vector

This property is read-only.

Signal minimum physical value, returned as a numeric vector of length `NumSignals`. Each element of `PhysicalMin` contains the minimum physical value of the corresponding signal in `SignalLabels`.

Data Types: `double`

`PhysicalMax` — Signal maximum physical value
numeric vector

This property is read-only.

Signal maximum physical value, returned as a numeric vector of length `NumSignals`. Each element of `PhysicalMax` contains the maximum physical value of the corresponding signal in `SignalLabels`.

Data Types: `double`

`DigitalMin` — Signal minimum digital value
numeric vector

This property is read-only.

Signal minimum digital value, returned as a numeric vector of length `NumSignals`. Each element of `DigitalMin` contains the minimum digital value of the corresponding signal in `SignalLabels`.

Data Types: `double`

`DigitalMax` — Signal maximum digital value
numeric vector

This property is read-only.

Signal maximum digital value, returned as a numeric vector of length `NumSignals`. Each element of `DigitalMax` contains the maximum digital value of the corresponding signal in `SignalLabels`.

Data Types: `double`

`NumSamples` — Number of samples in signal
numeric vector

This property is read-only.

Number of samples in signal, returned as a numeric vector of length `NumSignals`. Each element of `NumSamples` contains the number of samples in the corresponding signal in `SignalLabels`.

Data Types: `double`

`Prefilter` — Signal data units
string vector

This property is read-only.

Signal data units, returned as a string vector of length `NumSignals`. Each element of `Prefilter` contains details about the filters, if any, used to preprocess the corresponding signal in `SignalLabels`.

Example: `["HP:10Hz LP:80Hz N:60Hz";"HP:0.1Hz LP:90Hz N:60Hz"]`

Data Types: `string`

`SignalReserved` — Additional signal information
string vector

This property is read-only.

Additional signal information, returned as a string vector of length `NumSignals`. Each element of `SignalReserved` contains additional information, if any, about the corresponding signal in `SignalLabels`.

Data Types: `string`

`Annotations` — Annotations present in signal records
timetable

This property is read-only.

Annotations present in signal records, returned as a timetable containing these variables:

- `Onset` — Time at which the annotation occurred, expressed as a duration indicating the number of seconds elapsed since the start time of the file.
- `Annotations` — A string that contains the annotation text.
- `Duration` — A duration scalar that indicates the duration of the event described by the annotation. If the file does not specify an annotation duration, this variable is returned as `NaN`.

Data Types: `table`

Examples

Get Information About EDF File

Use the `edfinfo` function to create an `edfinfo` object containing information about the file `example.edf`.

```
info = edfinfo('example.edf')
```

```
info =
  edfinfo with properties:

              Filename: "example.edf"
          FileModDate: "30-Oct-2020 12:27:26"
              FileSize: 31488
               Version: "0"
               Patient: "Patient 7"
             Recording: "Startdate not recorded"
             StartDate: "10.10.20"
             StartTime: "12.02.18"
           HeaderBytes: 768
              Reserved: ""
       NumDataRecords: 6
   DataRecordDuration: 10 sec
           NumSignals: 2
         SignalLabels: [2x1 string]
      TransducerTypes: [2x1 string]
   PhysicalDimensions: [2x1 string]
          PhysicalMin: [2x1 double]
          PhysicalMax: [2x1 double]
           DigitalMin: [2x1 double]
           DigitalMax: [2x1 double]
            Prefilter: [2x1 string]
           NumSamples: [2x1 double]
       SignalReserved: [2x1 string]
          Annotations: [0x2 timetable]
```

Display this information about the second signal in the file:

- Its name
- The physical units in which the data is expressed
- The minimum and maximum physical values of the data
- The number of samples it contains

```
nsig = 2;

disp([info.SignalLabels(nsig) info.PhysicalDimensions(nsig) ...
    info.PhysicalMin(nsig) info.PhysicalMax(nsig) info.NumSamples(nsig)])
```

```
    "ECG2"    "mV"    "-229.048"    "229.041"    "1280"
```

Tips

You can convert an `edfinfo` object to a MATLAB structure using the `get` function. For example:

```
info = edf('example.edf');
strc = get(info)
```

References

[1] Kemp, Bob, Alpo Värri, Agostinho C. Rosa, Kim D. Nielsen, and John Gade. “A Simple Format for Exchange of Digitized Polygraphic Recordings.” *Electroencephalography and Clinical Neurophysiology* 82, no. 5 (May 1992): 391–93. https://doi.org/10.1016/0013-4694(92)90009-7.

[2] Kemp, Bob, and Jesus Olivan. "European Data Format 'plus' (EDF+), an EDF Alike Standard Format for the Exchange of Physiological Data." *Clinical Neurophysiology* 114, no. 9 (2003): 1755–1761. https://doi.org/10.1016/S1388-2457(03)00123-8.

See Also

`datetime` | `duration` | `edfread` | `get` | `timetable`

External Websites

European Data Format

Introduced in R2020b

edfwrite

Create or modify EDF or EDF+ file

Description

Create an `edfwrite` object to write or modify a European Data Format (EDF) or EDF+ file.

Creation

Syntax

```
edfw = edfwrite(filename)
edfw = edfwrite(filename,hdr,sigdata)
edfw = edfwrite(filename,hdr,annotationslist)
edfw = edfwrite(filename,hdr,sigdata,annotationslist)
edfw = edfwrite( ___ ,Name,Value)
```

Description

`edfw = edfwrite(filename)` creates an `edfwrite` object for an existing EDF or EDF+ file specified by `filename`.

`edfw = edfwrite(filename,hdr,sigdata)` creates an `edfwrite` object and a new EDF or EDF + file with signal data, `sigdata`. File properties are specified in the header structure, `hdr`.

`edfw = edfwrite(filename,hdr,annotationslist)` creates an `edfwrite` object and a new EDF or EDF+ file with annotations, `annotationslist`.

`edfw = edfwrite(filename,hdr,sigdata,annotationslist)` creates an `edfwrite` object and a new EDF or EDF+ file with signal data and annotations.

`edfw = edfwrite( ___ ,Name,Value)` sets “Properties” on page 1-468 using name-value arguments. You can specify `DataRecordTimes`, `AnnotationsEncoding`, and `InputSampleType`.

Input Arguments

`filename` — Name of EDF or EDF+ file

character vector | string scalar

Name of EDF or EDF+ file, specified as a character vector or string scalar.

Depending on the location of the file, `filename` can take one of these forms.

Location	Form
Current folder or folder on the MATLAB path	Specify the name of the file in `filename`. **Example:** `'data.edf'`

Location	Form
File in a folder	If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name. **Example:** `'C:\myFolder\data.edf'` **Example:** `'myDir\myFile.ext'`

Note `edfwrite` does not support EyeLink EDF files.

Data Types: `char` | `string`

`hdr` — Header
structure

Header details, specified as a structure. See `edfheader` for more information.

Data Types: `struct`

`sigdata` — Signal data
matrix | cell array

Signal data, specified as a numeric matrix with one or more columns or a cell array of numeric vectors.

Data Types: `double` | `cell`

`annotationslist` — Annotations
timetable

Annotations, specified as a timetable containing these variables:

- `Onset` — Time at which the annotation occurred, expressed as a duration indicating the number of seconds elapsed since the start time of the file. Use `Onset` to specify the `rowTimes` in the timetable.
- `Annotations` — A string that contains the annotation text.
- `Duration` — A duration scalar that indicates the duration of the event described by the annotation. If the file does not specify an annotation duration, this variable is returned as `NaN`.

Data Types: `table`

Properties

File Properties

`Filename` — File name
string scalar

This property is read-only.

File name, returned as a string scalar.

Example: `"ecg_20200411_120.edf"`

Data Types: `string`

`FileType` — File type
`"EDF"` | `"EDF+"`

File type, returned as `"EDF"` or `"EDF+"`.

Data Types: `string`

`FileModDate` — Date last modified
string scalar

This property is read-only.

Date last modified, returned as a string scalar with the date and time the file was last modified.

Example: `"11-Apr-2020 15:38:37"`

`FileSize` — File size in bytes
integer scalar

This property is read-only.

File size in bytes, returned as an integer scalar.

Example: `4040992`

Data Types: `double`

Header Properties

`Version` — Data format version
`"0"`

This property is read-only.

Data format version, returned as `"0"`.

Data Types: `string`

`Patient` — Patient identification details
string scalar

This property is read-only.

Patient identification details, returned as a string scalar. Patient identification details can include Patient ID, sex or gender, birth date in `'dd-MMM-yyyy'` format, and name.

Example: `"X F X 120 04-JUL-1982"`

Data Types: `string`

`Recording` — Recording identification details
string scalar

This property is read-only.

Recording identification details, returned as a string scalar. Recording identification details may include its start date and time, the ID of the technician that made the recording, and the ID of the equipment that made the recording.

Example: `"Startdate 04-JUL-1982 X X X"`

Data Types: `string`

`StartDate` — Recording start date

string scalar

This property is read-only.

Recording start date, returned as a string scalar in `'dd.MM.yy'` format.

Example: `"04.07.82"`

Data Types: `string`

`StartTime` — Recording start time

string scalar

This property is read-only.

Recording start time, returned as a string scalar in `'HH.mm.ss'` format.

Example: `"17.16.37"`

Data Types: `string`

`HeaderBytes` — Header size in bytes

integer scalar

This property is read-only.

Header size in bytes, returned as an integer scalar. `HeaderBytes` is given by (256 + `NumSignals` × 256) bytes. The first 256 bytes correspond to a static header and are required for all EDF and EDF+ files. The other bytes depend on the number of signals present in the data records.

Example: `2048`

Data Types: `double`

`Reserved` — EDF+ interruption information

`"EDF+C"` | `"EDF+D"` | `""`

This property is read-only.

EDF+ interruption information, returned as `"EDF+C"` or `"EDF+D"` for EDF+ compliant files.

- `"EDF+C"` — The recording is continuous: There are no interruptions and all data records are contiguous, such that the start time of each data record coincides with the start time of the previous record plus its duration.
- `"EDF+D"` — The recording is discontinuous with interruptions between consecutive data records.

For files that are not EDF+ compliant, this property is an empty string (`""`).

Data Types: `string`

`NumDataRecords` — Number of data records in file
integer scalar

This property is read-only.

Number of data records in file, returned as an integer scalar.

Note If `filename` is not EDF compliant, `NumDataRecords` can be set to `-1` when the number of data records is unknown. If `filename` is EDF compliant, `NumDataRecords` must be set to a positive integer. If `filename` has `Reserved` set to a nonempty string and `NumDataRecords` set to `-1`, `edfinfo` throws an error.

Data Types: `double`

`DataRecordDuration` — Duration of each data record
duration scalar

This property is read-only.

Duration of each data record, returned as a duration scalar.

Data Types: `duration`

`NumSignals` — Number of signals in file
integer scalar

This property is read-only.

Number of signals in file, returned as an integer scalar.

Data Types: `double`

`DataRecordTimes` — Start time of each data record
duration vector

Start time of each data record, returned as a duration vector. `DataRecordTimes` must be specified for an EDF+ file with discontinuous record start times. The vector must be equal in length to `NumDataRecords`.

Data Types: `duration`

Signal Properties

`SignalLabels` — Signal names
string vector

This property is read-only.

Signal names, returned as a string vector of length `NumSignals`.

```
["Thorax 1";"Abdomen 3"]
```

Data Types: `string`

`TransducerTypes` — Transducer details
string vector

This property is read-only.

Transducer details, returned as a string vector of length `NumSignals`. Each element of `TransducerTypes` contains details about the transducer used to obtain the corresponding signal in `SignalLabels`.

Example: `["AgAgCl electrodes";"thermistor"]`

Data Types: `string`

`PhysicalDimensions` — Signal data units
string vector

This property is read-only.

Signal data units, returned as a string vector of length `NumSignals`. Each element of `PhysicalDimensions` contains the measurement units used to express the values of the corresponding signal in `SignalLabels`.

Example: `["uV";"mV"]`

Data Types: `string`

`PhysicalMin` — Signal minimum physical value
numeric vector

This property is read-only.

Signal minimum physical value, returned as a numeric vector of length `NumSignals`. Each element of `PhysicalMin` contains the minimum physical value of the corresponding signal in `SignalLabels`.

Data Types: `double`

`PhysicalMax` — Signal maximum physical value
numeric vector

This property is read-only.

Signal maximum physical value, returned as a numeric vector of length `NumSignals`. Each element of `PhysicalMax` contains the maximum physical value of the corresponding signal in `SignalLabels`.

Data Types: `double`

`DigitalMin` — Signal minimum digital value
numeric vector

This property is read-only.

Signal minimum digital value, returned as a numeric vector of length `NumSignals`. Each element of `DigitalMin` contains the minimum digital value of the corresponding signal in `SignalLabels`.

Data Types: `double`

`DigitalMax` — Signal maximum digital value
numeric vector

This property is read-only.

Signal maximum digital value, returned as a numeric vector of length `NumSignals`. Each element of `DigitalMax` contains the maximum digital value of the corresponding signal in `SignalLabels`.

Data Types: `double`

`Prefilter` — Signal data units

string vector

This property is read-only.

Signal data units, returned as a string vector of length `NumSignals`. Each element of `Prefilter` contains details about the filters, if any, used to preprocess the corresponding signal in `SignalLabels`.

Example: `["HP:10Hz LP:80Hz N:60Hz";"HP:0.1Hz LP:90Hz N:60Hz"]`

Data Types: `string`

`NumSamples` — Number of samples in signal

numeric vector

This property is read-only.

Number of samples in signal, returned as a numeric vector of length `NumSignals`. Each element of `NumSamples` contains the number of samples in the corresponding signal in `SignalLabels`.

Data Types: `double`

`SignalReserved` — Additional signal information

string vector

This property is read-only.

Additional signal information, returned as a string vector of length `NumSignals`. Each element of `SignalReserved` contains additional information, if any, about the corresponding signal in `SignalLabels`.

Data Types: `string`

`InputSampleType` — Input sample type of signal data

`"digital"` (default) | `"physical"`

Input sample type of signal data, returned as `"digital"` or `"physical"`. The function defaults to `"digital"` and writes the signal data into the file with no digital scaling. If `'InputSampleType'` is set to `"physical"`, then `edfwrite` applies digital scaling to the signal data.

Data Types: `string`

Annotation Properties

`Annotations` — Annotations present in signal records

timetable

This property is read-only.

Annotations present in signal records, returned as a timetable containing these variables:

- `Onset` — Time at which the annotation occurred, expressed as a duration indicating the number of seconds elapsed since the start time of the file.
- `Annotations` — A string that contains the annotation text.
- `Duration` — A duration scalar that indicates the duration of the event described by the annotation. If the file does not specify an annotation duration, this variable is returned as `NaN`.

Data Types: `table`

`AnnotationsEncoding` — Encoding format
`"US-ASCII"` (default) | `"UTF-8"` | `"LATIN1"`

Encoding format used to write annotations into the file, returned as `"US-ASCII"`, `"UTF-8"`, or `"LATIN1"`.

Data Types: `string`

Object Functions

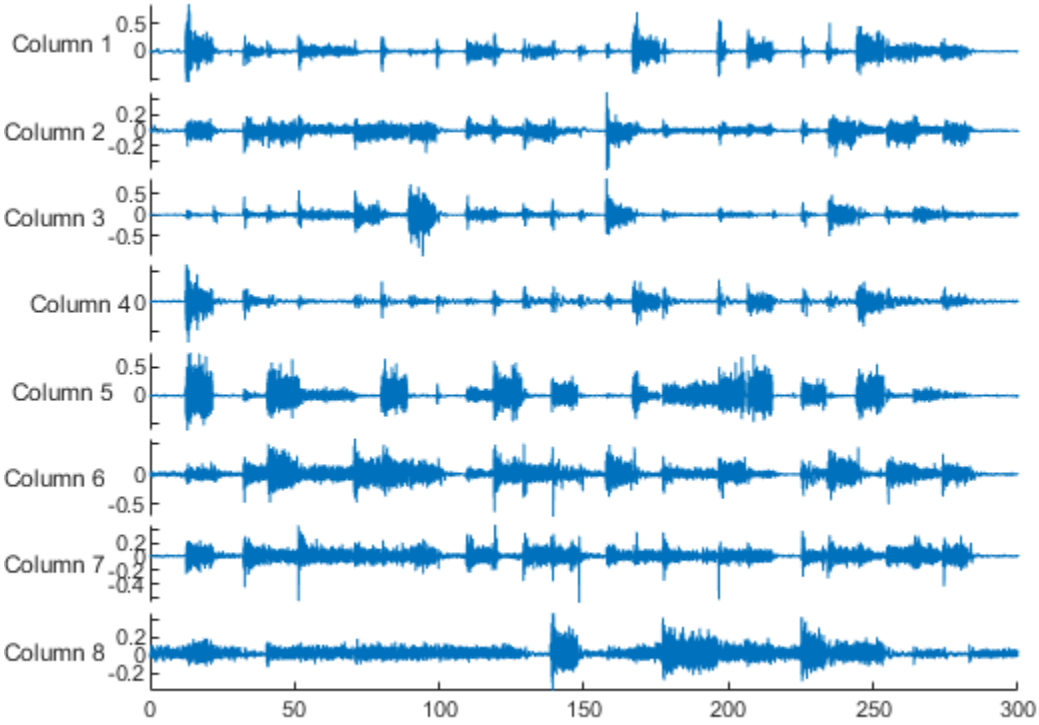
addAnnotations	Add annotations to EDF or EDF+ file
addSignals	Add new signals to EDF or EDF+ file
deleteAnnotations	Delete annotations from EDF or EDF+ file
deleteSignals	Delete signals from EDF or EDF+ file
modifyAnnotations	Modify annotations in EDF or EDF+ file
modifyHeader	Modify header details of EDF or EDF+ file
modifySignals	Modify signals in EDF or EDF+ file

Examples

Create EDF+ File with Annotations

Load `EMGdata.mat` into the workspace. The file contains eight channels of surface electromyography (EMG) data [1] recorded from eight arm muscles. The data is available at www.sce.carleton.ca/faculty/chan/index.php?page=matlab. The sample rate is 1000 Hz. Plot the signals.

```
load EMGdata

fs = 1000;
t = 0:1/fs:(size(data,1)-1)/fs;
stackedplot(t,data)
```



The bursts of increased signal amplitude correspond to different forearm motions that last 3 seconds each. `EMGindex.mat` contains the type of motion and the start index (sample) of each motion in two variables: `motion` and `start_index`. The motion types are:

1. Hand open
2. Hand close
3. Wrist flexion
4. Wrist extension
5. Supination
6. Pronation
7. Rest

Load the data into the workspace.

```
load EMGindex
```

Create a timetable of annotations.

1. Use `Onset` to specify the row times. `Onset` contains the start index of each motion in seconds.
2. `Annotations` specifies the types of motion as a string array.
3. `Duration` specifies the duration of each motion in seconds.

```
Onset = seconds(start_index./fs);
Annotations = string(motion);
```

```
Duration = seconds(ones(length(Onset),1)*3);

annotationslist = timetable(Onset,Annotations,Duration);
```

Use `edfheader` to create a header structure for the EDF+ file and set the properties. See `edfheader` for more information.

```
hdr = edfheader("EDF+");
hdr.NumDataRecords = 1;
hdr.DataRecordDuration = seconds(length(data(:,1))/fs);
hdr.NumSignals = 8;
hdr.SignalLabels = ["F1" "F2" "F3" "F4" "F5" "F6" "F7" "B1"];
hdr.PhysicalDimensions = repelem("mV",8);
hdr.PhysicalMin = min(data);
hdr.PhysicalMax = max(data);
hdr.DigitalMin = [-32768 -32768 -32768 -32768 -32768 -32768 -32768 -32768];
hdr.DigitalMax = [32767 32767 32767 32767 32767 32767 32767 32767];
```

Write an EDF+ file containing the header structure, signal data, and annotations. Specify the input sample type as `physical`. The file is saved in the current working directory.

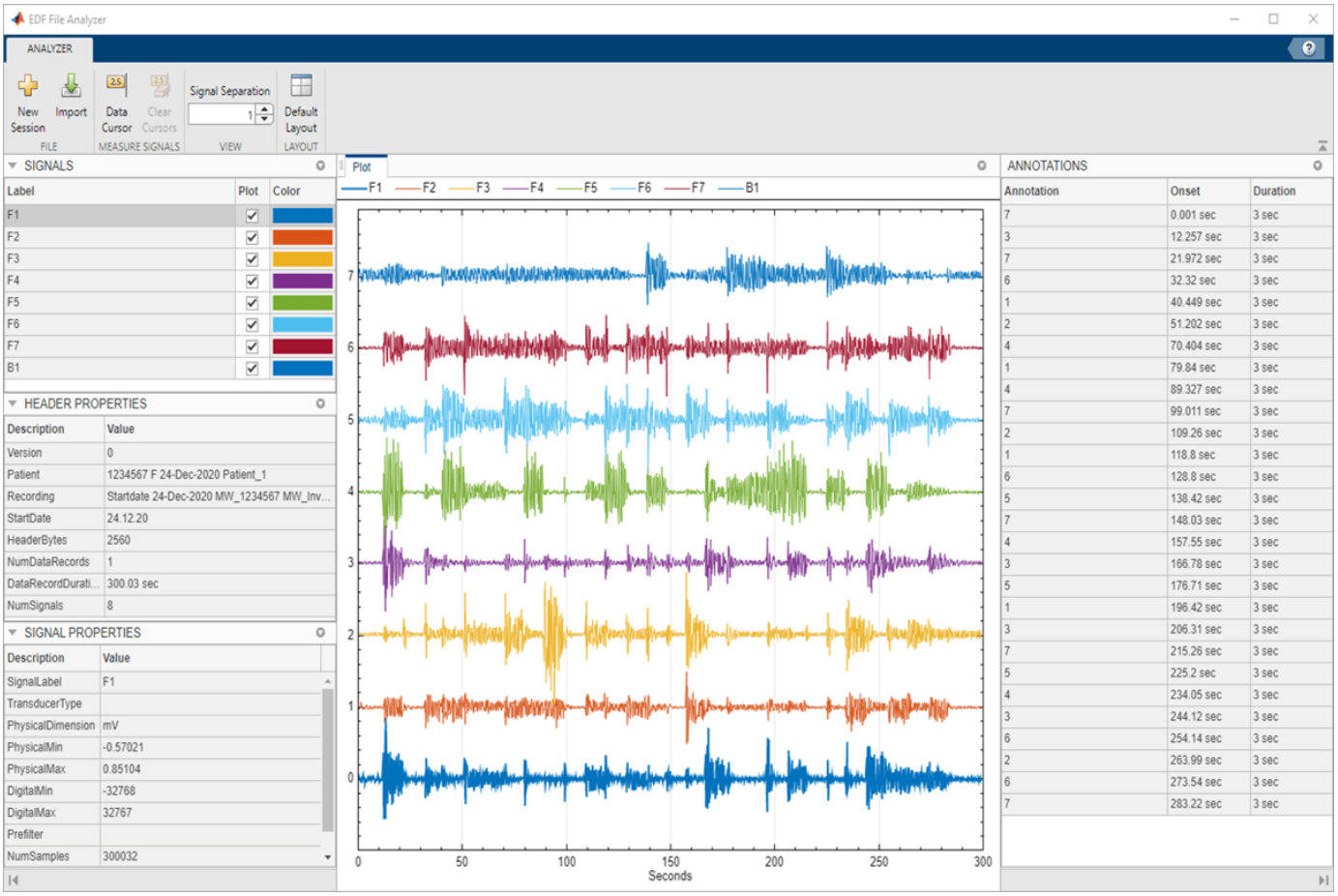
```
edfw = edfwrite("armEMG.edf",hdr,data,annotationslist,'InputSampleType',"physical");
```

Display information about the file.

```
edfinfo("armEMG.edf")
```

```
ans =
  edfinfo with properties:

              Filename: "armEMG.edf"
          FileModDate: "26-Feb-2022 20:49:23"
             FileSize: 4803836
              Version: "0"
              Patient: "1234567 F 26-Feb-2022 Patient_1"
            Recording: "Startdate 26-Feb-2022 MW_1234567 MW_Inv_01 MW_Eq_01"
            StartDate: "26.02.22"
            StartTime: "20.49.22"
          HeaderBytes: 2560
             Reserved: "EDF+C"
       NumDataRecords: 1
   DataRecordDuration: 300.03 sec
           NumSignals: 8
         SignalLabels: [8x1 string]
      TransducerTypes: [8x1 string]
   PhysicalDimensions: [8x1 string]
          PhysicalMin: [8x1 double]
          PhysicalMax: [8x1 double]
           DigitalMin: [8x1 double]
           DigitalMax: [8x1 double]
            Prefilter: [8x1 string]
           NumSamples: [8x1 double]
       SignalReserved: [8x1 string]
          Annotations: [28x2 timetable]
```

You can use EDF File Analyzer to view the signals and annotations stored in the file. Use the `Signal Separation` option to separate the signals for better visualization.



Delete the EDF+ file. Comment out this code if you want to keep the file.

```
delete armEMG.edf
```

Tips

- To create an EDF+ file containing only annotations, specify `NumDataRecords` and `NumSignals` as `0`, `DataRecordDuration` as a duration scalar with value `0`, and all signal properties as empty.
- Launch the **EDF File Analyzer** app to visualize the signals in your EDF or EDF+ file.

References

[1] Chan, Adrian D.C., and Geoffrey C. Green. 2007. "Myoelectric Control Development Toolbox". Paper presented at *30th Conference of the Canadian Medical & Biological Engineering Society, Toronto, Canada, 2007*.

[2] Kemp, Bob, Alpo Värri, Agostinho C. Rosa, Kim D. Nielsen, and John Gade. “A Simple Format for Exchange of Digitized Polygraphic Recordings.” *Electroencephalography and Clinical Neurophysiology* 82, no. 5 (May 1992): 391–93. https://doi.org/10.1016/0013-4694(92)90009-7.

[3] Kemp, Bob, and Jesus Olivan. "European Data Format 'plus' (EDF+), an EDF Alike Standard Format for the Exchange of Physiological Data." *Clinical Neurophysiology* 114, no. 9 (2003): 1755–1761. https://doi.org/10.1016/S1388-2457(03)00123-8.

See Also

Apps
EDF File Analyzer

Objects
`edfinfo`

Functions
`edfheader` | `edfread`

External Websites
European Data Format

Introduced in R2021a

addAnnotations

Add annotations to EDF or EDF+ file

Syntax

```
edfw = addAnnotations(edfw,tsal)
```

Description

`edfw = addAnnotations(edfw,tsal)` adds the annotations in `tsal` to the European Data Format (EDF) or EDF+ file.

Examples

Add Annotations to EDF File

Create a new EDF file that contains a header and a random 10-sample signal.

```
sig = randn(10,1);
hdr = edfheader("EDF");
hdr.NumSignals = 1;
hdr.NumDataRecords = 1;
hdr.PhysicalMin = min(sig);
hdr.PhysicalMax = max(sig);
hdr.DigitalMin = -32768;
hdr.DigitalMax = 32768;

edfw = edfwrite("random.edf",hdr,sig,"InputSampleType","physical");
```

Create a timetable that contains three annotations which occur at 2, 3, and 7 seconds. Specify the annotation strings as "Two", "Three", and "Seven". Each annotation duration is 1 second.

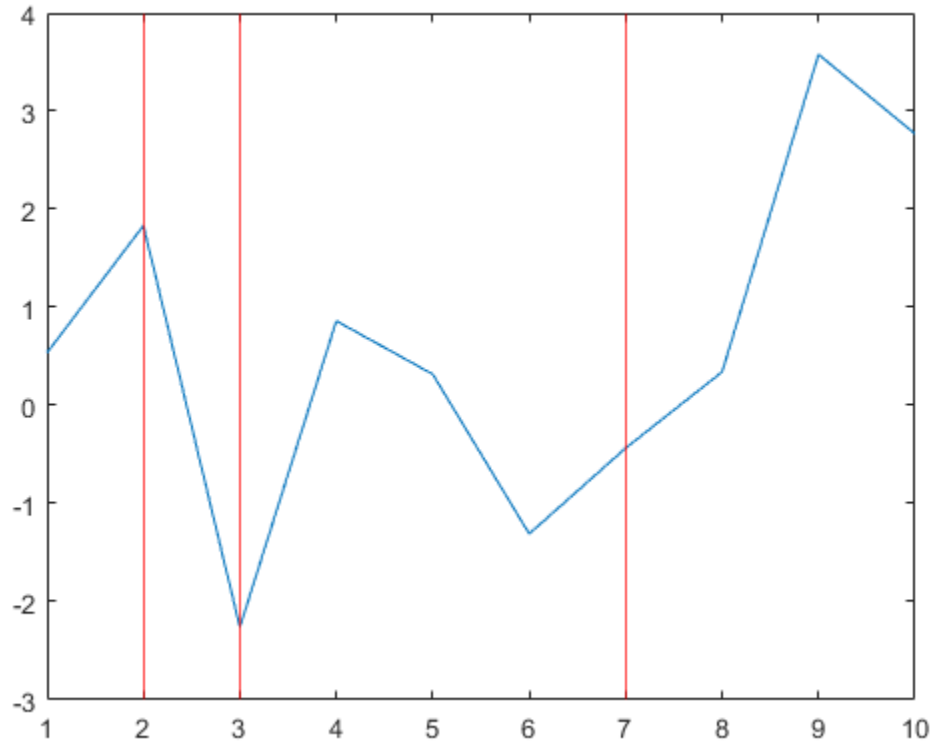
```
Onset = seconds([2;3;7]);
Annotations = ["Two" "Three" "Seven"]';
Duration = seconds(ones(3,1));

tsal = timetable(Onset,Annotations,Duration)
```

```
tsal=3×2 timetable
    Onset    Annotations    Duration
    _____    ___________    ________

    2 sec      "Two"          1 sec
    3 sec      "Three"        1 sec
    7 sec      "Seven"        1 sec
```

Add the annotations to `edfw`. Use `edfread` to read the data and annotations present in the file. Plot the data and add red vertical lines at each annotation onset.

```
edfw = addAnnotations(edfw,tsal);
[data,anns] = edfread("random.edf");
```

```
plot(data.Signal_1{1})
xline(seconds(anns.Onset),'r')
```



Input Arguments

`edfw` — EDF or EDF+ file

`edfwrite` object

EDF or EDF+ file, specified as an `edfwrite` object.

`tsal` — List of timestamped annotations

timetable

List of timestamped annotations, specified as a timetable containing these variables:

- `Onset` — Time at which the annotation occurred, expressed as a duration indicating the number of seconds elapsed since the start time of the file. Use `Onset` to specify the `RowTimes` in the timetable.
- `Annotations` — A string that contains the annotation text.
- `Duration` — A duration scalar that indicates the duration of the event described by the annotation. If the file does not specify an annotation duration, this variable is returned as `NaN`.

Output Arguments

`edfw` — EDF or EDF+ file
`edfwrite` object

EDF or EDF+ file, returned as an `edfwrite` object.

References

[1] Kemp, Bob, Alpo Värri, Agostinho C. Rosa, Kim D. Nielsen, and John Gade. "A Simple Format for Exchange of Digitized Polygraphic Recordings." *Electroencephalography and Clinical Neurophysiology* 82, no. 5 (May 1992): 391–93. https://doi.org/10.1016/0013-4694(92)90009-7.

[2] Kemp, Bob, and Jesus Olivan. "European Data Format 'plus' (EDF+), an EDF Alike Standard Format for the Exchange of Physiological Data." *Clinical Neurophysiology* 114, no. 9 (2003): 1755–1761. https://doi.org/10.1016/S1388-2457(03)00123-8.

See Also

Apps
EDF File Analyzer

Objects
`edfwrite` | `edfinfo`

Functions
`edfheader` | `edfread`

External Websites
European Data Format

Introduced in R2021a

addSignals

Add new signals to EDF or EDF+ file

Syntax

```
edfw = addSignals(edfw,signallabels,signaldata)
edfw = addSignals( ___ ,Name,Value)
```

Description

`edfw = addSignals(edfw,signallabels,signaldata)` adds new signals to the European Data Format (EDF) or EDF+ file with labels in `signallabels` and data in `signaldata`.

`edfw = addSignals( ___ ,Name,Value)` specifies additional options using name-value pairs. For example, `'DigitalMin',-2048,'DigitalMax',2048` specifies the digital minimum and digital maximum values of `sigdata`.

Examples

Add Signal to EDF+ File

Load a `labeledSignalSet` into the workspace. `heartrates` contains two electrocardiogram (ECG) signals from the MIT-BIH Arrhythmia Database [1]. The sample rate is 250 Hz.

```
load HeartRates
```

Create an EDF+ file that contains a header and the first signal in the labeled signal set (`y200`).

```
sig1 = getSignal(heartrates,1);
sig1 = sig1.y200;

hdr = edfheader("EDF+");
hdr.SignalLabels = "y200";
hdr.NumDataRecords = 1;
hdr.PhysicalMin = min(sig1);
hdr.PhysicalMax = max(sig1);

edfw = edfwrite("ECG.edf",hdr,sig1,"InputSampleType","physical");
```

Retrieve the second signal from `heartrates` and add it to the EDF+ file with signal label `y203`. Specify the physical minimum and maximum values of the second signal.

```
sig2 = getSignal(heartrates,2);
sig2 = sig2.y203;
sig2Label = "y203";

edfw = addSignals(edfw,sig2Label,sig2,'PhysicalMin',min(sig2),'PhysicalMax',max(sig2));
```

Use `edfinfo` to view the file properties. The number of signals in the header record is 2.

```
edfinfo("ECG.edf")
```

```
ans =
  edfinfo with properties:

             Filename: "ECG.edf"
         FileModDate: "26-Feb-2022 20:29:29"
            FileSize: 28814
             Version: "0"
             Patient: "1234567 F 26-Feb-2022 Patient_1"
           Recording: "Startdate 26-Feb-2022 MW_1234567 MW_Inv_01 MW_Eq_01"
           StartDate: "26.02.22"
           StartTime: "20.29.28"
         HeaderBytes: 1024
            Reserved: "EDF+C"
       NumDataRecords: 1
   DataRecordDuration: 1 sec
           NumSignals: 2
         SignalLabels: [2x1 string]
      TransducerTypes: [2x1 string]
   PhysicalDimensions: [2x1 string]
          PhysicalMin: [2x1 double]
          PhysicalMax: [2x1 double]
           DigitalMin: [2x1 double]
           DigitalMax: [2x1 double]
            Prefilter: [2x1 string]
           NumSamples: [2x1 double]
       SignalReserved: [2x1 string]
          Annotations: [0x2 timetable]
```

Input Arguments

`edfw` — EDF or EDF+ file

`edfwrite` object

EDF or EDF+ file, specified as an `edfwrite` object.

`signallabels` — Signal names

string vector | cell array of character vectors

Signal names, specified as a string vector or cell array of character vectors. The number of signal names must equal the number of input signals in `signaldata`.

Data Types: `string`

`signaldata` — Input signal data

numeric matrix | cell array of numeric vectors

Input signal data, specified as a numeric matrix or cell array of numeric vectors. The number of samples in each signal must be a multiple of the number of data records in `NumDataRecords`. Specify `signaldata` as a numeric matrix when all input signals have the same sample rate. If input signals have different sample rates or lengths, specify `signaldata` as a cell array of numeric vectors.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'PhysicalMin',-5,'PhysicalMax',5` specifies the physical minimum and physical maximum values of the input signal as -5 and 5, respectively.

`PhysicalMin` — Signal minimum physical value
numeric vector

Signal minimum physical value, specified as a numeric vector of length `NumSignals`. The signal physical minimum value must be less than the corresponding signal physical maximum value. `PhysicalMin` must be specified when `InputSampleType` is set to `'digital'`. If the input sample type is `'physical'` and `PhysicalMin` is not specified, then the function uses the minimum value of each signal as the physical minimum value.

Data Types: `double`

`PhysicalMax` — Signal maximum physical value
numeric vector

Signal maximum physical value, specified as a numeric vector of length `NumSignals`. The signal physical maximum value must be greater than the corresponding signal physical minimum value. `PhysicalMax` must be specified when `InputSampleType` is set to `'digital'`. If the input sample type is `'physical'` and `PhysicalMax` is not specified, then the function uses the maximum value of each signal as the physical maximum value.

Data Types: `double`

`DigitalMin` — Signal digital minimum value
numeric vector

Signal digital minimum value, specified as a numeric vector of length `NumSignals`. The signal digital minimum value must be less than the corresponding signal digital maximum value. `DigitalMin` values are based on the analog-to-digital converter used to generate `signaldata`. If not specified, the signal digital minimum value defaults to `-32768`.

Data Types: `double`

`DigitalMax` — Signal digital maximum value
numeric vector

Signal digital maximum value, specified as a numeric vector of length `NumSignals`. The signal digital maximum value must be greater than the corresponding signal digital minimum value. `DigitalMax` values are based on the analog-to-digital converter used to generate `signaldata`. If not specified, the signal digital maximum value defaults to `32767`.

Data Types: `double`

Output Arguments

`edfw` — EDF or EDF+ file
`edfwrite` object

EDF or EDF+ file, returned as an `edfwrite` object.

References

[1] Moody, G.B., and R.G. Mark. “The Impact of the MIT-BIH Arrhythmia Database.” *IEEE Engineering in Medicine and Biology Magazine* 20, no. 3 (June 2001): 45–50. https://doi.org/10.1109/51.932724.

[2] Kemp, Bob, Alpo Värri, Agostinho C. Rosa, Kim D. Nielsen, and John Gade. “A Simple Format for Exchange of Digitized Polygraphic Recordings.” *Electroencephalography and Clinical Neurophysiology* 82, no. 5 (May 1992): 391–93. https://doi.org/10.1016/0013-4694(92)90009-7.

[3] Kemp, Bob, and Jesus Olivan. "European Data Format 'plus' (EDF+), an EDF Alike Standard Format for the Exchange of Physiological Data." *Clinical Neurophysiology* 114, no. 9 (2003): 1755–1761. https://doi.org/10.1016/S1388-2457(03)00123-8.

See Also

Apps
EDF File Analyzer

Objects
`edfwrite` | `edfinfo`

Functions
`edfheader` | `edfread`

External Websites
European Data Format

Introduced in R2021a

deleteAnnotations

Delete annotations from EDF or EDF+ file

Syntax

```
edfw = deleteAnnotations(edfw,annotationindices)
edfw = deleteAnnotations(edfw)
```

Description

`edfw = deleteAnnotations(edfw,annotationindices)` deletes the annotations at the indices specified in `annotationindices` from the European Data Format (EDF) or EDF+ file.

`edfw = deleteAnnotations(edfw)` deletes all the annotations present in `edfw`.

Examples

Delete Annotations from EDF+ File

Load an `edfwrite` object into the workspace that contains a timetable with 28 annotations. Each annotation corresponds to the onset of one of six arm motions or a rest period:

- Hand open – "1"
- Hand close – "2"
- Wrist flexion – "3"
- Wrist extension – "4"
- Supination – "5"
- Pronation – "6"
- Rest – "7"

```
load edfw
```

Delete the rest periods ("7") from `edfw` and view the annotations timetable. There are 22 annotations remaining and no instances of rest.

```
idx = find(edfw.Annotations.Annotations == "7");
edfw = deleteAnnotations(edfw,idx);
edfw.Annotations
```

```
ans=22×2 timetable
      Onset       Annotations    Duration
    __________    ___________    ________

    12.257 sec        "3"         3 sec
    32.32 sec         "6"         3 sec
    40.449 sec        "1"         3 sec
    51.202 sec        "2"         3 sec
    70.404 sec        "4"         3 sec
```

```
    79.84 sec         "1"         3 sec
    89.327 sec        "4"         3 sec
    109.26 sec        "2"         3 sec
    118.8 sec         "1"         3 sec
    128.8 sec         "6"         3 sec
    138.42 sec        "5"         3 sec
    157.55 sec        "4"         3 sec
    166.78 sec        "3"         3 sec
    176.71 sec        "5"         3 sec
    196.42 sec        "1"         3 sec
    206.31 sec        "3"         3 sec
      ⋮
```

Create a region-of-interest (ROI) table that contains the remaining annotations. Convert the duration arrays to double arrays.

```
anns = edfw.Annotations;
region = seconds([anns.Onset anns.Onset+anns.Duration]);
label = anns.Annotations;
roi = table(region,label)
```

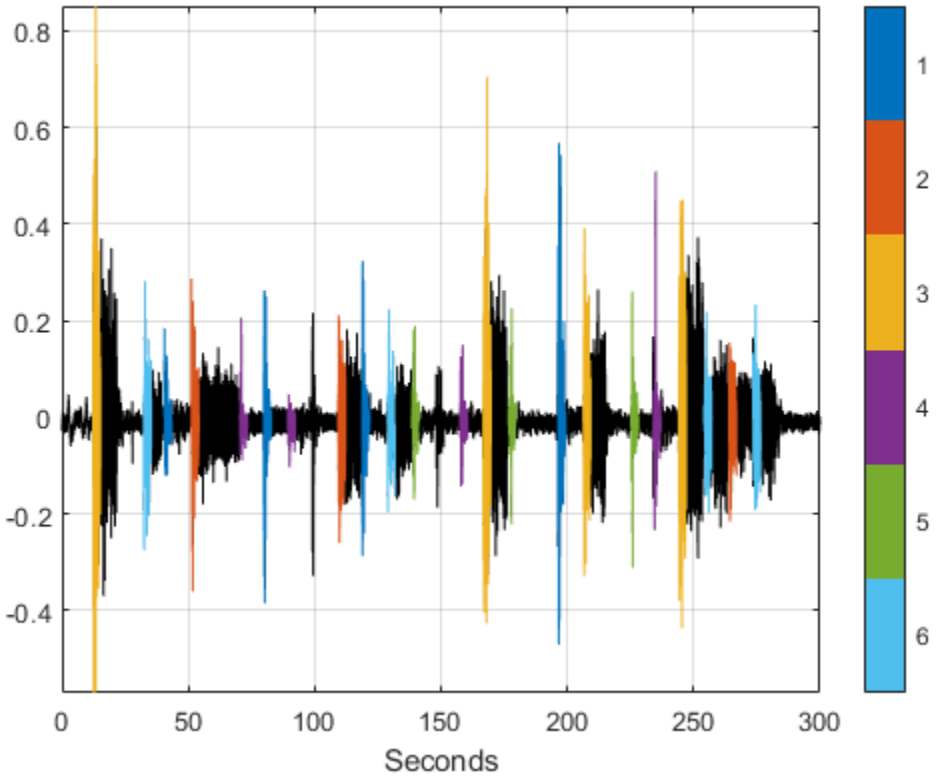
```
roi=22×2 table
         region          label
    ________________    _____

    12.257    15.257     "3"
     32.32     35.32     "6"
    40.449    43.449     "1"
    51.202    54.202     "2"
    70.404    73.404     "4"
     79.84     82.84     "1"
    89.327    92.327     "4"
    109.25    112.25     "2"
    118.81    121.81     "1"
     128.8     131.8     "6"
    138.42    141.42     "5"
    157.55    160.55     "4"
    166.78    169.78     "3"
    176.71    179.71     "5"
    196.43    199.43     "1"
    206.31    209.31     "3"
      ⋮
```

Load the electromyography (EMG) data [1] related to the annotations. The data is available at www.sce.carleton.ca/faculty/chan/index.php?page=matlab. The sample rate is 1000 Hz. Create a signal variable that contains only the first channel of data.

```
load EMGdata
fs = 1000;
x = data(:,1);
```

Create a signal mask for the regions of interest and motion labels. Plot the EMG signal along with the annotation regions.

```
msk = signalMask(roi,"SampleRate",fs);
plotsigroi(msk,x)
```



Input Arguments

`edfw` — EDF or EDF+ file
`edfwrite` object

EDF or EDF+ file, specified as an `edfwrite` object.

`annotationindices` — Annotation indices
numeric vector

Annotation indices, specified as a numeric vector. The values in `annotationindices` must be row indices of the `Annotations` property.

Data Types: `double`

Output Arguments

`edfw` — EDF or EDF+ file
`edfwrite` object

EDF or EDF+ file, returned as an `edfwrite` object.

References

[1] Chan, Adrian D.C., and Geoffrey C. Green. 2007. "Myoelectric Control Development Toolbox". Paper presented at *30th Conference of the Canadian Medical & Biological Engineering Society, Toronto, Canada, 2007*.

[2] Kemp, Bob, Alpo Värri, Agostinho C. Rosa, Kim D. Nielsen, and John Gade. "A Simple Format for Exchange of Digitized Polygraphic Recordings." *Electroencephalography and Clinical Neurophysiology* 82, no. 5 (May 1992): 391–93. https://doi.org/10.1016/0013-4694(92)90009-7.

[3] Kemp, Bob, and Jesus Olivan. "European Data Format 'plus' (EDF+), an EDF Alike Standard Format for the Exchange of Physiological Data." *Clinical Neurophysiology* 114, no. 9 (2003): 1755–1761. https://doi.org/10.1016/S1388-2457(03)00123-8.

See Also

Apps
EDF File Analyzer

Objects
`edfwrite` | `edfinfo`

Functions
`edfheader` | `edfread`

External Websites
European Data Format

Introduced in R2021a

deleteSignals

Delete signals from EDF or EDF+ file

Syntax

```
edfw = deleteSignals(edfw,signallabels)
edfw = deleteSignals(edfw)
```

Description

`edfw = deleteSignals(edfw,signallabels)` deletes the signals specified in `signallabels` from the European Data Format (EDF) or EDF+ file.

`edfw = deleteSignals(edfw)` deletes all the signals from the EDF or EDF+ file.

Examples

Delete Signal from EDF+ File

Load `edfw.mat` into the workspace. The `edfwrite` object contains electromyography (EMG) data [1] from eight different arm muscles. The data is available at www.sce.carleton.ca/faculty/chan/index.php?page=matlab. Display the number of signals and the signal labels.

```
load edfw
edfw.NumSignals
```

```
ans = 8
```

```
edfw.SignalLabels
```

```
ans = 8x1 string
    "F1"
    "F2"
    "F3"
    "F4"
    "F5"
    "F6"
    "F7"
    "B1"
```

Delete signal `B1` from `edfw`. Display the number of signals and the signal labels.

```
edfw = deleteSignals(edfw,"B1");
edfw.NumSignals
```

```
ans = 7
```

```
edfw.SignalLabels
```

```
ans = 7x1 string
    "F1"
```

```
"F2"
"F3"
"F4"
"F5"
"F6"
"F7"
```

Input Arguments

`edfw` — EDF or EDF+ file
`edfwrite` object

EDF or EDF+ file, specified as an `edfwrite` object.

`signallabels` — Signal names
string vector

Signal names, specified as a string vector.

Data Types: `string`

Output Arguments

`edfw` — EDF or EDF+ file
`edfwrite` object

EDF or EDF+ file, returned as an `edfwrite` object.

Note When a signal is deleted, the recording becomes discontinuous and the `Reserved` property of the EDF or EDF+ file is converted to `EDF+D`.

References

[1] Chan, Adrian D.C., and Geoffrey C. Green. 2007. "Myoelectric Control Development Toolbox". Paper presented at *30th Conference of the Canadian Medical & Biological Engineering Society, Toronto, Canada, 2007*.

[2] Kemp, Bob, Alpo Värri, Agostinho C. Rosa, Kim D. Nielsen, and John Gade. "A Simple Format for Exchange of Digitized Polygraphic Recordings." *Electroencephalography and Clinical Neurophysiology* 82, no. 5 (May 1992): 391–93. https://doi.org/10.1016/0013-4694(92)90009-7.

[3] Kemp, Bob, and Jesus Olivan. "European Data Format 'plus' (EDF+), an EDF Alike Standard Format for the Exchange of Physiological Data." *Clinical Neurophysiology* 114, no. 9 (2003): 1755–1761. https://doi.org/10.1016/S1388-2457(03)00123-8.

See Also

Apps
EDF File Analyzer

Objects
`edfwrite` | `edfinfo`

Functions
`edfheader` | `edfread`

External Websites
European Data Format

Introduced in R2021a

modifyAnnotations

Modify annotations in EDF or EDF+ file

Syntax

```
edfw = modifyAnnotations(edfw,annotationindices,tsal)
```

Description

`edfw = modifyAnnotations(edfw,annotationindices,tsal)` modifies annotations in `edfw` with the annotations in `tsal` at the indices specified in `annotationindices`.

Examples

Modify Annotations for Heart Rate Signal

Load a `labeledSignalSet` into the workspace. `heartrates` contains two electrocardiogram (ECG) signals from the MIT-BIH Arrhythmia Database [1] and a labels table for each signal. The sample rate is 250 Hz.

```
load HeartRates
```

Create an annotations timetable using the label values for the first signal in the labeled signal set.

```
labels = getLabelValues(heartrates,1,'QRSregions')
```

```
labels=41×2 table
       ROILimits       Value
    _______________    _____

    0.556     0.724     QRS
    1.304       1.4     QRS
     1.82     1.976     QRS
    2.628     2.724     QRS
    3.092     3.292     QRS
    3.916      4.02     QRS
    4.384     4.564     QRS
    5.184     5.272     QRS
    5.668     5.844     QRS
    6.428     6.516     QRS
    6.876     7.048     QRS
    7.688     7.784     QRS
    8.192     8.416     QRS
    9.092     9.208     QRS
    9.592     9.768     QRS
    10.34    10.428     QRS
      ⋮
```

```
Onset = seconds(labels.ROILimits(:,1));
Annotations = string(labels.Value);
```

```
Duration = seconds(labels.ROILimits(:,2) - labels.ROILimits(:,1));

tsal = timetable(Onset,Annotations,Duration);
```

Create a new EDF+ file that contains a header, the first ECG signal in the labeled signal set, and the annotations.

```
sig = getSignal(heartrates,1);
sig = sig.y200;

hdr = edfheader("EDF+");
hdr.SignalLabels = "y200";
hdr.NumDataRecords = 1;
hdr.PhysicalMin = min(sig);
hdr.PhysicalMax = max(sig);

edfw = edfwrite("heartrate1.edf",hdr,sig,tsal,"InputSampleType","physical");
```

Modify the annotation at every odd index to "`skip`". Display the modified annotations in `edfw`.

```
mtsal = tsal(1:2:end,:);
mtsal.Annotations = repelem("skip",21)';

idx = 1:2:numel(tsal(:,1));
edfw = modifyAnnotations(edfw,idx,mtsal);
edfw.Annotations
```

```
ans=41×2 timetable
      Onset      Annotations    Duration
    _________    ___________    _________

    0.556 sec      "skip"       0.168 sec
    1.304 sec      "QRS"        0.096 sec
    1.82 sec       "skip"       0.156 sec
    2.628 sec      "QRS"        0.096 sec
    3.092 sec      "skip"         0.2 sec
    3.916 sec      "QRS"        0.104 sec
    4.384 sec      "skip"        0.18 sec
    5.184 sec      "QRS"        0.088 sec
    5.668 sec      "skip"       0.176 sec
    6.428 sec      "QRS"        0.088 sec
    6.876 sec      "skip"       0.172 sec
    7.688 sec      "QRS"        0.096 sec
    8.192 sec      "skip"       0.224 sec
    9.092 sec      "QRS"        0.116 sec
    9.592 sec      "skip"       0.176 sec
    10.34 sec      "QRS"        0.088 sec
      ⋮
```

Input Arguments

`edfw` — EDF or EDF+ file

`edfwrite` object

EDF or EDF+ file, specified as an `edfwrite` object.

`annotationindices` — Annotation indices
numeric vector

Annotation indices, specified as a numeric vector. The values in `annotationindices` must be row indices of the `Annotations` property.

Data Types: `double`

`tsal` — List of timestamped annotations
timetable

List of timestamped annotations, specified as a timetable containing these variables:

- `Onset` — Time at which the annotation occurred, expressed as a duration indicating the number of seconds elapsed since the start time of the file. Use `Onset` to specify the `RowTimes` in the timetable.
- `Annotations` — A string that contains the annotation text.
- `Duration` — A duration scalar that indicates the duration of the event described by the annotation. If the file does not specify an annotation duration, this variable is returned as `NaN`.

The number of rows in `tsal` must be equal to the number of elements in `annotationindices`.

Output Arguments

`edfw` — EDF or EDF+ file
`edfwrite` object

EDF or EDF+ file, returned as an `edfwrite` object.

References

[1] Moody, G.B., and R.G. Mark. “The Impact of the MIT-BIH Arrhythmia Database.” *IEEE Engineering in Medicine and Biology Magazine* 20, no. 3 (June 2001): 45–50. https://doi.org/10.1109/51.932724.

[2] Kemp, Bob, Alpo Värri, Agostinho C. Rosa, Kim D. Nielsen, and John Gade. “A Simple Format for Exchange of Digitized Polygraphic Recordings.” *Electroencephalography and Clinical Neurophysiology* 82, no. 5 (May 1992): 391–93. https://doi.org/10.1016/0013-4694(92)90009-7.

[3] Kemp, Bob, and Jesus Olivan. "European Data Format 'plus' (EDF+), an EDF Alike Standard Format for the Exchange of Physiological Data." *Clinical Neurophysiology* 114, no. 9 (2003): 1755–1761. https://doi.org/10.1016/S1388-2457(03)00123-8.

See Also

Apps
EDF File Analyzer

Objects
`edfwrite` | `edfinfo`

Functions
`edfheader` | `edfread`

External Websites
European Data Format

Introduced in R2021a

modifyHeader

Modify header details of EDF or EDF+ file

Syntax

```
edfw = modifyHeader(edfw,hdr)
```

Description

`edfw = modifyHeader(edfw,hdr)` modifies header fields in `edfw` using the header specified in the structure `hdr`.

Examples

Modify Header Record

Create a header record for an EDF file and specify the patient identification, recording information, and start time.

```
hdr = edfheader("EDF");
hdr.Patient = "001 M 2020";
hdr.Recording = "This is the original header record";
hdr.StartTime = "00.00.01"
```

```
hdr = struct with fields:
              Patient: "001 M 2020"
            Recording: "This is the original header record"
            StartDate: "26.02.22"
            StartTime: "00.00.01"
             Reserved: ""
       NumDataRecords: -1
   DataRecordDuration: 1 sec
           NumSignals: []
         SignalLabels: [0x0 string]
      TransducerTypes: [0x0 string]
   PhysicalDimensions: [0x0 string]
          PhysicalMin: []
          PhysicalMax: []
           DigitalMin: []
           DigitalMax: []
            Prefilter: [0x0 string]
       SignalReserved: [0x0 string]
```

Create a new EDF file that contains the header record and a random 10-sample signal. Specify in `hdr` the number of signals and the signal physical minimum and maximum values. Set the input sample type as `physical`.

```
sig = randn(10,1);
hdr.NumSignals = 1;
hdr.PhysicalMin = min(sig);
```

```
hdr.PhysicalMax = max(sig);
edfw = edfwrite("file.edf",hdr,sig,"InputSampleType","physical");
```

Create a new header structure with modified patient, recording, and start time information.

```
newhdr.Patient = "002 F 2020";
newhdr.Recording = "This is a test";
newhdr.StartTime = "11.11.10";
```

Modify the original header record in `file` with the new information in `newhdr`. Display the file properties.

```
edfw = modifyHeader(edfw,newhdr);
edfinfo("file.edf")
```

```
ans =
  edfinfo with properties:

             Filename: "file.edf"
          FileModDate: "26-Feb-2022 21:20:30"
             FileSize: 532
              Version: "0"
              Patient: "002 F 2020"
            Recording: "This is a test"
            StartDate: "26.02.22"
            StartTime: "11.11.10"
          HeaderBytes: 512
             Reserved: ""
       NumDataRecords: -1
   DataRecordDuration: 1 sec
           NumSignals: 1
         SignalLabels: "Signal_1"
      TransducerTypes: ""
   PhysicalDimensions: ""
          PhysicalMin: -2.2588
          PhysicalMax: 3.5784
           DigitalMin: 0
           DigitalMax: 0
            Prefilter: ""
           NumSamples: 10
       SignalReserved: ""
          Annotations: [0x2 timetable]
```

Input Arguments

`edfw` — EDF or EDF+ file

`edfwrite` object

EDF or EDF+ file, specified as an `edfwrite` object.

`hdr` — Header

structure

Header, specified as a structure. `hdr` can contain one or more of these fields:

- `Patient`
- `Recording`
- `StartDate`
- `StartTime`
- `SignalLabels`
- `TransducerTypes`
- `PhysicalDimensions`
- `PhysicalMin`
- `PhysicalMax`
- `DigitalMin`
- `DigitalMax`
- `Prefilter`
- `SignalReserved`

See `edfheader` for more information about the possible fields in the header structure.

Data Types: `struct`

Output Arguments

`edfw` — EDF or EDF+ file
`edfwrite` object

EDF or EDF+ file, returned as an `edfwrite` object.

References

[1] Kemp, Bob, Alpo Värri, Agostinho C. Rosa, Kim D. Nielsen, and John Gade. “A Simple Format for Exchange of Digitized Polygraphic Recordings.” *Electroencephalography and Clinical Neurophysiology* 82, no. 5 (May 1992): 391–93. https://doi.org/10.1016/0013-4694(92)90009-7.

[2] Kemp, Bob, and Jesus Olivan. "European Data Format 'plus' (EDF+), an EDF Alike Standard Format for the Exchange of Physiological Data." *Clinical Neurophysiology* 114, no. 9 (2003): 1755–1761. https://doi.org/10.1016/S1388-2457(03)00123-8.

See Also

Apps
EDF File Analyzer

Objects
`edfwrite` | `edfinfo`

Functions
`edfheader` | `edfread`

External Websites
European Data Format

Introduced in R2021a

modifySignals

Modify signals in EDF or EDF+ file

Syntax

```
edfw = modifySignals(edfw,signaldata)
edfw = modifySignals( ___ ,Name,Value)
```

Description

`edfw = modifySignals(edfw,signaldata)` modifies all signals present in the European Data Format (EDF) or EDF+ file with the new signals in `signaldata`.

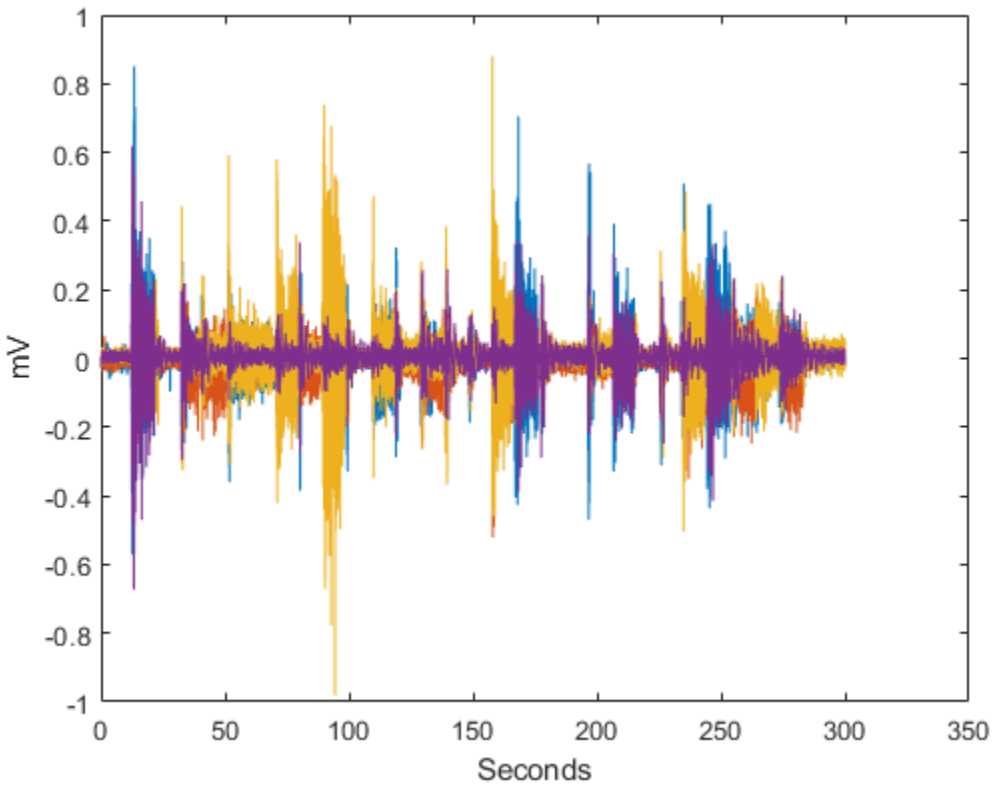
`edfw = modifySignals( ___ ,Name,Value)` specifies additional options using name-value arguments. For example, `'PhysicalMin',-3,'PhysicalMax',6` specifies the signal physical minimum value as –3 and the physical maximum value as 6.

Examples

Modify Signals in EDF+ File

Load `EMGdata.mat` into the workspace. The file contains eight channels of electromyography (EMG) data [1] recorded from eight arm muscles. The data is available at www.sce.carleton.ca/faculty/chan/index.php?page=matlab. The sample rate is 1000 Hz. Plot the first four channels in `data`.

```
load EMGdata
fs = 1000;
t = 0:1/fs:(size(data,1)-1)/fs;
plot(t,data(:,1:4))
xlabel('Seconds')
ylabel('mV')
```

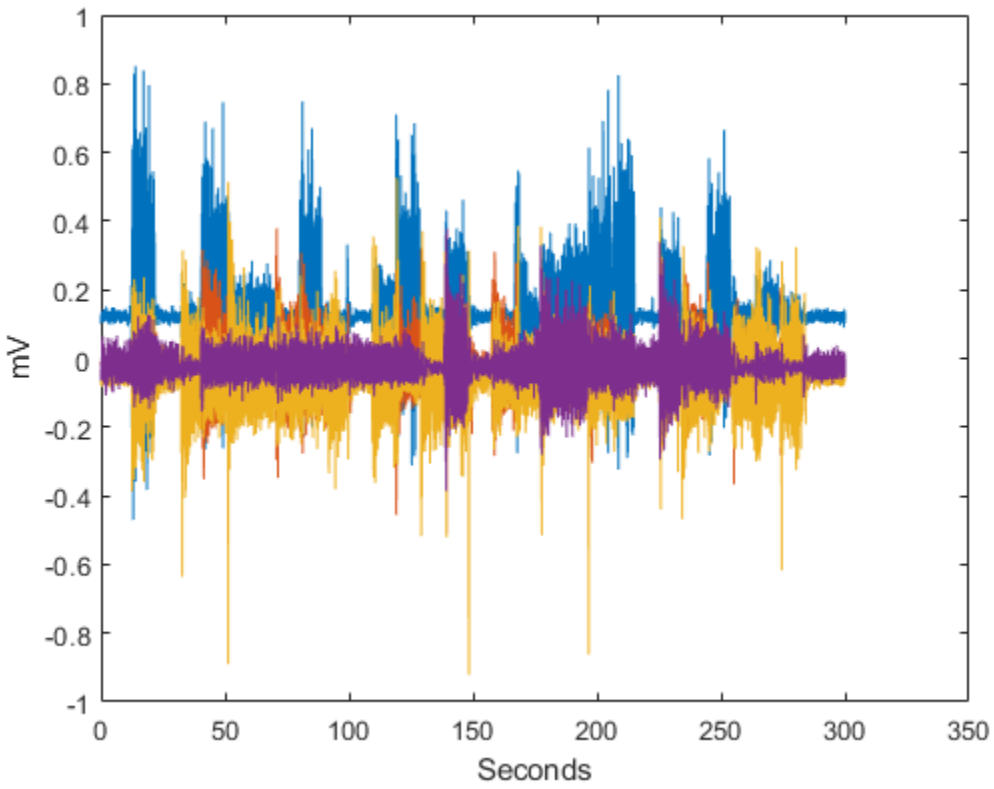


Create an EDF+ file that contains a header and signal data from channels 1-4 in `data`. See `edfheader` for more information about creating a header structure.

```
sig = data(:,1:4);
hdr = edfheader("EDF+");
hdr.NumSignals = 4;
hdr.NumDataRecords = 1;
hdr.PhysicalMin = min(sig);
hdr.PhysicalMax = max(sig);
hdr.DigitalMin = repelem(-32768,4);
hdr.DigitalMax = repelem(32767,4);

edfw = edfwrite("EMG.edf",hdr,sig,"InputSampleType","physical");
```

Modify the signals in `edfw` with the data from channels 5-8 in `data`. Use `edfread` to read the data in `EMG.edf` and plot the signals.

```
modsig = data(:,5:8);
edfw = modifySignals(edfw,modsig);

x = edfread("EMG.edf");
for i = 1:4
    y = x.(i){1};
    plot(t,y)
    xlabel('Seconds')
    ylabel('mV')
    hold on
end
```



Input Arguments

`edfw` — EDF or EDF+ file

`edfwrite` object

EDF or EDF+ file, specified as an `edfwrite` object.

`signaldata` — Input signal data

numeric matrix | cell array of numeric vectors

Input signal data, specified as a numeric matrix or cell array of numeric vectors. The number of samples in each signal must be a multiple of the number of data records in `NumDataRecords`.

Note Specify `signaldata` as a numeric matrix when all input signals have the same sample rate. If input signals have different sample rates or lengths, specify `signaldata` as a cell array of numeric vectors.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'SelectedSignals',["F1" "B1"],'SelectedDataRecords',[1 2]` instructs `modifySignals` to modify the first and second data records of signals `F1` and `B1`.

`SelectedSignals` — Signal names
string vector | cell array of character vectors

Signal names, specified as a string vector or cell array of character vectors. `modifySignals` modifies the data of the signals with names specified by `SelectedSignals`.

Data Types: `char` | `string`

`SelectedDataRecords` — Data records
numeric vector

Data records, specified as a numeric vector. `modifySignals` modifies all the signals in the data records specified by the record indices in `SelectedDataRecords`.

Data Types: `double`

`PhysicalMin` — Signal minimum physical value
numeric vector

Signal minimum physical value, specified as a numeric vector of length `NumSignals`. The signal physical minimum value must be less than the corresponding signal physical maximum value. Specify `PhysicalMin` only when the entire signal is modified. If not specified, the function uses the existing physical minimum value of the corresponding signal.

Data Types: `double`

`PhysicalMax` — Signal maximum physical value
numeric vector

Signal maximum physical value, specified as a numeric vector of length `NumSignals`. The signal physical maximum value must be greater than the corresponding signal physical minimum value. Specify `PhysicalMax` only when the entire signal is modified. If not specified, the function uses the existing physical maximum value of the corresponding signal.

Data Types: `double`

`DigitalMin` — Signal digital minimum value
numeric vector

Signal digital minimum value, specified as a numeric vector of length `NumSignals`. The signal digital minimum value must be less than the corresponding signal digital maximum value. `DigitalMin` values are based on the analog-to-digital converter used to generate `signaldata`. If not specified, the signal digital minimum value defaults to `-32768`.

Data Types: `double`

`DigitalMax` — Signal digital maximum value
numeric vector

Signal digital maximum value, specified as a numeric vector of length `NumSignals`. The signal digital maximum value must be greater than the corresponding signal digital minimum value. `DigitalMax` values are based on the analog-to-digital converter used to generate `signaldata`. If not specified, the signal digital maximum value defaults to `32767`.

Data Types: `double`

Output Arguments

`edfw` — EDF or EDF+ file
`edfwrite` object

EDF or EDF+ file, returned as an `edfwrite` object.

References

[1] Chan, Adrian D.C., and Geoffrey C. Green. 2007. "Myoelectric Control Development Toolbox". Paper presented at *30th Conference of the Canadian Medical & Biological Engineering Society, Toronto, Canada, 2007*.

[2] Kemp, Bob, Alpo Värri, Agostinho C. Rosa, Kim D. Nielsen, and John Gade. "A Simple Format for Exchange of Digitized Polygraphic Recordings." *Electroencephalography and Clinical Neurophysiology* 82, no. 5 (May 1992): 391–93. https://doi.org/10.1016/0013-4694(92)90009-7.

[3] Kemp, Bob, and Jesus Olivan. "European Data Format 'plus' (EDF+), an EDF Alike Standard Format for the Exchange of Physiological Data." *Clinical Neurophysiology* 114, no. 9 (2003): 1755–1761. https://doi.org/10.1016/S1388-2457(03)00123-8.

See Also

Apps
EDF File Analyzer

Objects
`edfwrite` | `edfinfo`

Functions
`edfheader` | `edfread`

External Websites
European Data Format

Introduced in R2021a

edfheader

Create header structure for EDF or EDF+ file

Syntax

```
hdr = edfheader(filetype)
```

Description

`hdr = edfheader(filetype)` creates a header structure that can be used to create European Data Format (EDF) or EDF+ files with `edfwrite`.

Examples

Create Header Record

Create a header record for an EDF+ file and specify the recording information. Display the header properties.

```
hdr = edfheader("EDF+");
hdr.Patient = "P42Dory F";
hdr.Recording = "AJMS Device2";
hdr.StartDate = "27.12.1993";
hdr.StartTime = "04.22.24";
hdr.Reserved = "EDF+C";
hdr.NumDataRecords = 1;
hdr.DataRecordDuration = seconds(4.22)
```

```
hdr = struct with fields:
                Patient: "P42Dory F"
              Recording: "AJMS Device2"
              StartDate: "27.12.1993"
              StartTime: "04.22.24"
               Reserved: "EDF+C"
         NumDataRecords: 1
    DataRecordDuration: 4.22 sec
            NumSignals: []
          SignalLabels: [0x0 string]
       TransducerTypes: [0x0 string]
    PhysicalDimensions: [0x0 string]
           PhysicalMin: []
           PhysicalMax: []
            DigitalMin: []
            DigitalMax: []
             Prefilter: [0x0 string]
        SignalReserved: [0x0 string]
```

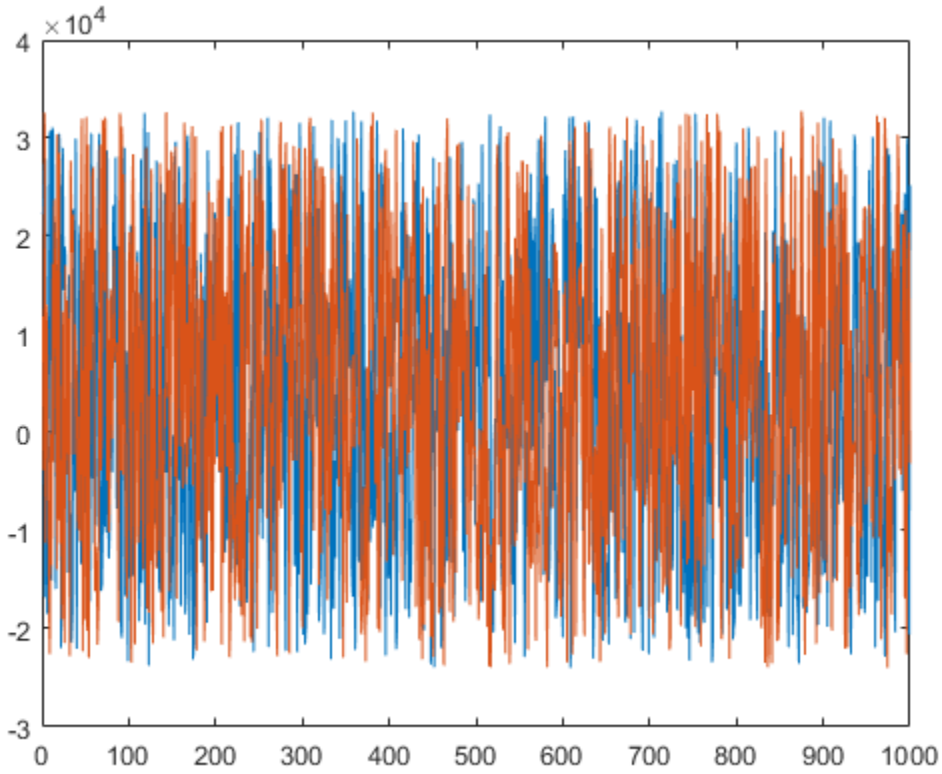
Create Header and Write EDF File with Signal Data

Create a header record for a new EDF file.

```
hdr = edfheader("EDF");
```

Generate two random 1000-sample signals containing integers in the range [–24000, 32767] and add random noise to the second signal. Plot both signals.

```
sigdata = randi([-24000 32767],1000,2);
sigdata(:,2) = sigdata(:,2) + 0.7*randn(1000,1);
plot(sigdata)
```



Specify header properties based on the two digital signals you created. The digital minimum and maximum values correspond to the extreme values that can occur, so specify these values as –32768 and 32767.

```
hdr.NumSignals = 2;
hdr.NumDataRecords = 1;
hdr.PhysicalMin = [-3200 -3200];
hdr.PhysicalMax = [3200 3200];
hdr.DigitalMin = [-32768 -32768];
hdr.DigitalMax = [32767 32767];
```

Write a new EDF file with the header structure and the random data. View the file properties.

```
edfw = edfwrite("rand.edf",hdr,sigdata);
edfinfo("rand.edf")
```

```
ans =
  edfinfo with properties:

             Filename: "rand.edf"
         FileModDate: "26-Feb-2022 20:49:33"
            FileSize: 4768
             Version: "0"
             Patient: "1234567 F 26-Feb-2022 Patient_1"
           Recording: "Startdate 26-Feb-2022 MW_1234567 MW_Inv_01 MW_Eq_01"
           StartDate: "26.02.22"
           StartTime: "20.49.33"
         HeaderBytes: 768
            Reserved: ""
       NumDataRecords: 1
   DataRecordDuration: 1 sec
           NumSignals: 2
         SignalLabels: [2x1 string]
      TransducerTypes: [2x1 string]
   PhysicalDimensions: [2x1 string]
          PhysicalMin: [2x1 double]
          PhysicalMax: [2x1 double]
           DigitalMin: [2x1 double]
           DigitalMax: [2x1 double]
            Prefilter: [2x1 string]
           NumSamples: [2x1 double]
       SignalReserved: [2x1 string]
          Annotations: [0x2 timetable]
```

Specify a new patient identification record, change the recording start time to `21:12:00`, and specify a label for each signal. Display the header structure to see the modified properties.

```
hdr.Patient = "20210410 F 27-JUL-2017";
hdr.SignalLabels = ["sig1" "sig2"];
hdr.StartTime = "21.12.00"
```

```
hdr = struct with fields:
             Patient: "20210410 F 27-JUL-2017"
           Recording: "Startdate 26-Feb-2022 MW_1234567 MW_Inv_01 MW_Eq_01"
           StartDate: "26.02.22"
           StartTime: "21.12.00"
            Reserved: ""
       NumDataRecords: 1
   DataRecordDuration: 1 sec
           NumSignals: 2
         SignalLabels: ["sig1"    "sig2"]
      TransducerTypes: [0x0 string]
   PhysicalDimensions: [0x0 string]
          PhysicalMin: [-3200 -3200]
          PhysicalMax: [3200 3200]
           DigitalMin: [-32768 -32768]
           DigitalMax: [32767 32767]
            Prefilter: [0x0 string]
       SignalReserved: [0x0 string]
```

Input Arguments

`filetype` — File type

`"EDF"` | `"EDF+"`

File type, specified as `"EDF"` or `"EDF+"`.

Data Types: `string`

Output Arguments

`hdr` — Header

structure

Header record, returned as a structure with these fields:

Field	Description
`Patient`	Patient identification details, returned as a string scalar. Patient identification details can include Patient ID, sex or gender, birth date in `'dd-MMM-yyyy'` format, and name.
`Recording`	Recording identification details, returned as a string scalar. Recording identification details may include its start date and time, the ID of the technician that made the recording, and the ID of the equipment that made the recording.
`StartDate`	Recording start date, returned as a string scalar in `'dd.MM.yy'` format.
`StartTime`	Recording start time, returned as a string scalar in `'HH.mm.ss'` format.
`Reserved`	EDF+ interruption information, returned as `"EDF+C"` or `"EDF+D"` for EDF + compliant files. • `"EDF+C"` — The recording is continuous. There are no interruptions and all data records are contiguous, such that the start time of each data record coincides with the start time of the previous record plus its duration. • `"EDF+D"` — The recording is discontinuous with interruptions between consecutive data records. For files that are not EDF+ compliant, this property is an empty string (`""`).
`NumDataRecords`	Number of data records in file, returned as an integer scalar. **Note** If `filename` is not EDF compliant, `NumDataRecords` can be set to `-1` when the number of data records is unknown. If `filename` is EDF compliant, `NumDataRecords` must be set to a positive integer. If `filename` has `Reserved` set to a nonempty string and `NumDataRecords` set to `-1`, `edfinfo` throws an error.
`DataRecordDuration`	Duration of each data record, returned as a duration scalar.
`NumSignals`	Number of signals in file, returned as an integer scalar.

Field	Description
`SignalLabels`	Signal names, returned as a string vector of length `NumSignals`. **Note** If `SignalLabels` is not specified, `edfwrite` uses the default label `"Signal_i"` for the *i*th signal.
`TransducerTypes`	Transducer details, returned as a string vector of length `NumSignals`. Each element of `TransducerTypes` contains details about the transducer used to obtain the corresponding signal in `SignalLabels`.
`PhysicalDimensions`	Signal data units, returned as a string vector of length `NumSignals`. Each element of `PhysicalDimensions` contains the measurement units used to express the values of the corresponding signal in `SignalLabels`.
`PhysicalMin`	Signal minimum physical value, returned as a numeric vector of length `NumSignals`. Each element of `PhysicalMin` contains the minimum physical value of the corresponding signal in `SignalLabels`.
`PhysicalMax`	Signal maximum physical value, returned as a numeric vector of length `NumSignals`. Each element of `PhysicalMax` contains the maximum physical value of the corresponding signal in `SignalLabels`.
`DigitalMin`	Signal minimum digital value, returned as a numeric vector of length `NumSignals`. Each element of `DigitalMin` contains the minimum digital value of the corresponding signal in `SignalLabels`.
`DigitalMax`	Signal maximum digital value, returned as a numeric vector of length `NumSignals`. Each element of `DigitalMax` contains the maximum digital value of the corresponding signal in `SignalLabels`.
`Prefilter`	Signal data units, returned as a string vector of length `NumSignals`. Each element of `Prefilter` contains details about the filters, if any, used to preprocess the corresponding signal in `SignalLabels`.
`SignalReserved`	Additional signal information, returned as a string vector of length `NumSignals`. Each element of `SignalReserved` contains additional information, if any, about the corresponding signal in `SignalLabels`.

References

[1] Kemp, Bob, Alpo Värri, Agostinho C. Rosa, Kim D. Nielsen, and John Gade. “A Simple Format for Exchange of Digitized Polygraphic Recordings.” *Electroencephalography and Clinical Neurophysiology* 82, no. 5 (May 1992): 391–93. https://doi.org/10.1016/0013-4694(92)90009-7.

[2] Kemp, Bob, and Jesus Olivan. "European Data Format 'plus' (EDF+), an EDF Alike Standard Format for the Exchange of Physiological Data." *Clinical Neurophysiology* 114, no. 9 (2003): 1755–1761. https://doi.org/10.1016/S1388-2457(03)00123-8.

See Also

Apps
EDF File Analyzer

Objects
`edfwrite` | `edfinfo`

Functions
`edfread`

External Websites
European Data Format

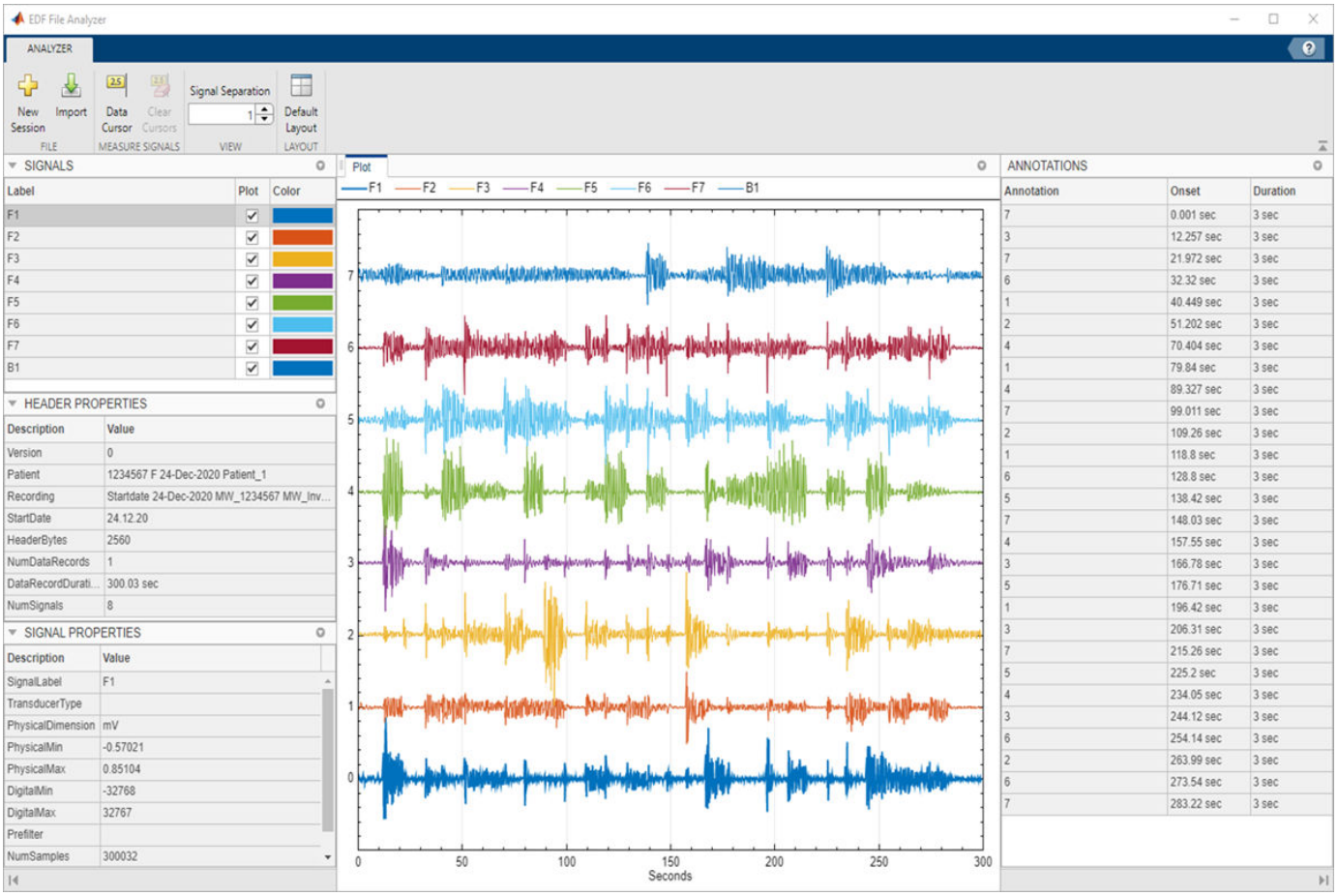
Introduced in R2021a

EDF File Analyzer

View EDF or EDF+ files

Description

The **EDF File Analyzer** app is an interactive tool for visualizing and analyzing data stored in a European Data Format (EDF) or EDF+ file. In the app, you can import an EDF or EDF+ file, plot signals, and view properties and annotations.



Open the EDF File Analyzer App

- MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- MATLAB command prompt: Enter `edfFileAnalyzer`.

Examples

- “Create EDF+ File with Annotations” on page 1-474

Programmatic Use

`edfFileAnalyzer` opens the **EDF File Analyzer** app.

See Also

Objects
`edfinfo` | `edfwrite`

Functions
`edfheader` | `edfread`

Topics
“Create EDF+ File with Annotations” on page 1-474

Introduced in R2021a

edfread

Read data from EDF/EDF+ file

Syntax

```
data = edfread(filename)
data = edfread(filename,Name,Value)

[data,annotations] = edfread( ___ )
```

Description

`data = edfread(filename)` reads the European Data Format (EDF) or EDF+ file specified in `filename` into a timetable, `data`.

`data = edfread(filename,Name,Value)` reads the file into a timetable with additional options specified by one or more name-value pair arguments.

`[data,annotations] = edfread( ___ )` also returns the annotations present in the data records.

Examples

Read EDF File into Timetable

Read data from the EDF file `example.edf` into a timetable. The file contains two signals, `ECG` and `ECG2`. Each signal contains six data records, and each data record has a duration of 10 seconds.

```
tt = edfread('example.edf')
```

```
tt=6×2 timetable
    Record Time          ECG               ECG2
    ___________    _______________    _______________

    0 sec          {1280x1 double}    {1280x1 double}
    10 sec         {1280x1 double}    {1280x1 double}
    20 sec         {1280x1 double}    {1280x1 double}
    30 sec         {1280x1 double}    {1280x1 double}
    40 sec         {1280x1 double}    {1280x1 double}
    50 sec         {1280x1 double}    {1280x1 double}
```

Create an `edfinfo` object containing information about `example.edf`. Verify that the signals have the expected names. Extract the sample rates of the signals using the “DataRecordDuration” on page 1-0 and “NumSamples” on page 1-0 properties of the object.

```
info = edfinfo('example.edf');

info.SignalLabels
```

```
ans = 2x1 string
    "ECG"
```

```
    "ECG2"
```

```
fs = info.NumSamples/seconds(info.DataRecordDuration)
```

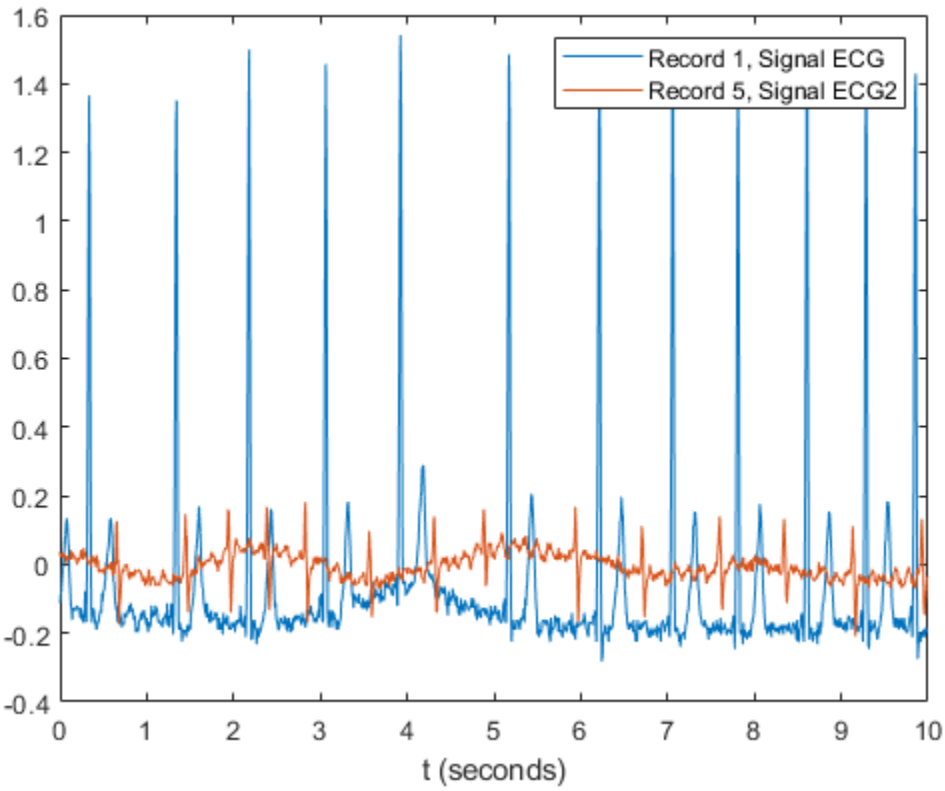
```
fs = 2×1

   128
   128
```

Plot the first record of the first signal. For more information about accessing data in tables, see "Access Data in Tables".

```
recnum = 1;
signum = 1;
t = (0:info.NumSamples(signum)-1)/fs(signum);
y = tt.(signum){recnum};

plot(t,y)
legend(strcat("Record ",int2str(recnum),", Signal ",info.SignalLabels(signum)))
hold on
```

Extract and plot the fifth record of the second signal.

```
recnum = 5;
signum = 2;
t = (0:info.NumSamples(signum)-1)/fs(signum);
y = tt.(signum){recnum};

plot(t,y, ...
    'DisplayName',strcat("Record ",int2str(recnum),", Signal ",info.SignalLabels(signum)))
hold off
xlabel('t (seconds)')
```



Read Subset of EDF File

Create an `edfinfo` object to obtain information about the EDF file `example.edf`. Extract the number of records and the names of the variables contained in the file.

```
info = edfinfo('example.edf');

nrec = info.NumDataRecords
```

```
nrec = 6
```

```
vars = info.SignalLabels
```

```
vars = 2x1 string
    "ECG"
    "ECG2"
```

Read the second and fifth records corresponding to the variable `ECG2`. Return the signals as timetables with row times corresponding to signal sample times. Express the time information as `datetime` arrays.

```
data = edfread('example.edf', ...
    'SelectedDataRecords',[2 5],'SelectedSignals',"ECG2", ...
    'DataRecordOutputType','timetable','TimeOutputType','datetime')
```

```
data=2×1 timetable
        Record Time                ECG2
    ____________________    ____________________

    10-Oct-2020 12:02:28    {1280x1 timetable}
    10-Oct-2020 12:02:58    {1280x1 timetable}
```

Change the name of the row times to `"Date and Time"` and the name of the variable to `"Electrocardiogram"`.

```
data.Properties.DimensionNames = ["Date and Time" "Variables"];
data.Properties.VariableNames = "Electrocardiogram";

data
```

```
data=2×1 timetable
       Date and Time        Electrocardiogram
    ____________________    ____________________

    10-Oct-2020 12:02:28    {1280x1 timetable}
    10-Oct-2020 12:02:58    {1280x1 timetable}
```

Input Arguments

`filename` — Name of EDF or EDF+ file

character vector | string scalar

Name of EDF or EDF+ file, specified as a character vector or string scalar.

Depending on the location of the file, `filename` can take one of these forms.

Location	Form
Current folder or folder on the MATLAB path	Specify the name of the file in `filename`. **Example:** `'data.edf'`
File in a folder	If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name. **Example:** `'C:\myFolder\data.edf'` **Example:** `'myDir\myFile.ext'`

Note `edfread` does not support EyeLink EDF files.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'SelectedSignals',["Thorax" "Abdomen"],'SelectedDataRecords',[2 7],'TimeOutputType','datetime'` instructs `edfread` to read the second and seventh data records corresponding to the `Thorax` and `Abdomen` signals and return the time information as `datetime` arrays.

`SelectedSignals` — Names of signals to read
string vector | cell array of character vectors

Names of signals to read, specified as the comma-separated pair consisting of `'SelectedSignals'` and a string vector or a cell array of character vectors.

- `'SelectedSignals'` must be a subset of the signal names contained in the file. To get the names of all the signals in the file, create an `edfinfo` object and use the `SignalLabels` property.
- If this argument is not specified, `edfread` reads all the signals in the file.

Example: Both `["Thorax 1" "Abdomen 3"]` and `{'Thorax 1' 'Abdomen 3'}` specify `Thorax 1` and `Abdomen 3` as the signals to read from a file.

Data Types: `char` | `string`

`SelectedDataRecords` — Indices of records to read
`1:height(edfread(filename))` (default) | vector of positive integers

Indices of records to read, specified as the comma-separated pair consisting of `'SelectedDataRecords'` and a vector of positive integers. The integers in the vector must be unique and strictly increasing.

- `'SelectedDataRecords'` must be a subset of the data records contained in the file. To see how many records are in the file, create an `edfinfo` object and use the `NumDataRecords` property. Alternatively, read the whole file and use the MATLAB function `height`.
- If this argument is not specified, `edfread` reads all the data records in the file.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`DataRecordOutputType` — Data output type
`'vector'` (default) | `'timetable'`

Data output type, specified as the comma-separated pair consisting of `'DataRecordOutputType'` and either `'vector'` or `'timetable'`.

- `'vector'` — Return the signals in `data` as vectors.
- `'timetable'` — Return the signals in `data` as timetables with row times corresponding to signal sample times.

Data Types: `char` | `string`

`TimeOutputType` — Time output type
`'duration'` (default) | `'datetime'`

Time output type, specified as the comma-separated pair consisting of `'TimeOutputType'` and either `'duration'` or `'datetime'`.

- `'duration'` — Return the time information in `data` as `duration` arrays.

- `'datetime'` — Return the time information in `data` as `datetime` arrays.

Data Types: `char` | `string`

Output Arguments

`data` — Output data

timetable

Output data, returned as a timetable. Each row of `data` corresponds to a record, and each variable of `data` corresponds to a signal.

- If `'DataRecordOutputType'` is specified as `'vector'`, the signal segment for each data record is returned as a vector.
- If `'DataRecordOutputType'` is specified as `'timetable'`, the signal segment for each data record is returned as a timetable with row times corresponding to signal sample times.

Each row time of `data` contains the start time of the corresponding data record.

- If `'TimeOutputType'` is set to `'duration'`, the start time of each record is relative to the start time of the file recording.
- If `'TimeOutputType'` is set to `'datetime'`, the start time of each record is the absolute start time.

`annotations` — Record annotations

timetable

Record annotations, returned as a timetable. The timetable contains these variables:

- `Onset` — Time at which the annotation occurred. The data type of `Onset` depends on the value specified for `'TimeOutputType'`.
- `Annotations` — A string that contains the annotation text.
- `Duration` — A duration scalar that indicates the duration of the event described by the annotation. If the file does not specify an annotation duration, this variable is returned as `NaN`.

References

[1] Kemp, Bob, Alpo Värri, Agostinho C. Rosa, Kim D. Nielsen, and John Gade. "A Simple Format for Exchange of Digitized Polygraphic Recordings." *Electroencephalography and Clinical Neurophysiology* 82, no. 5 (May 1992): 391–93. https://doi.org/10.1016/0013-4694(92)90009-7.

[2] Kemp, Bob, and Jesus Olivan. "European Data Format 'plus' (EDF+), an EDF Alike Standard Format for the Exchange of Physiological Data." *Clinical Neurophysiology* 114, no. 9 (2003): 1755–1761. https://doi.org/10.1016/S1388-2457(03)00123-8.

See Also

`datetime` | `duration` | `edfinfo` | `get` | `timetable`

External Websites

European Data Format

Introduced in R2020b

edr

Edit distance on real signals

Syntax

```
dist = edr(x,y,tol)
[dist,ix,iy] = edr(x,y,tol)

[___] = edr(x,y,maxsamp)

[___] = edr(___,metric)

edr(___)
```

Description

`dist = edr(x,y,tol)` returns the “Edit Distance on Real Signals” on page 1-532 between sequences `x` and `y`. `edr` returns the minimum number of elements that must be removed from `x`, `y`, or both `x` and `y`, so that the sum of Euclidean distances between the remaining signal elements lies within the specified tolerance, `tol`.

`[dist,ix,iy] = edr(x,y,tol)` returns the warping path such that `x(ix)` and `y(iy)` have the smallest possible `dist` between them. When `x` and `y` are matrices, `ix` and `iy` are such that `x(:,ix)` and `y(:,iy)` are minimally separated.

`[___] = edr(x,y,maxsamp)` restricts the insertion operations so that the warping path remains within `maxsamp` samples of a straight-line fit between `x` and `y`. This syntax returns any of the output arguments of previous syntaxes.

`[___] = edr(___,metric)` specifies the distance metric to use in addition to any of the input arguments in previous syntaxes. `metric` can be one of `'euclidean'`, `'absolute'`, `'squared'`, or `'symmkl'`.

`edr(___)` without output arguments plots the original and aligned signals.

- If the signals are real vectors, the function displays the two original signals on a subplot and the aligned signals in a subplot below the first one.
- If the signals are complex vectors, the function displays the original and aligned signals in three-dimensional plots.
- If the signals are real matrices, the function uses `imagesc` to display the original and aligned signals.
- If the signals are complex matrices, the function plots their real and imaginary parts in the top and bottom half of each image.

Examples

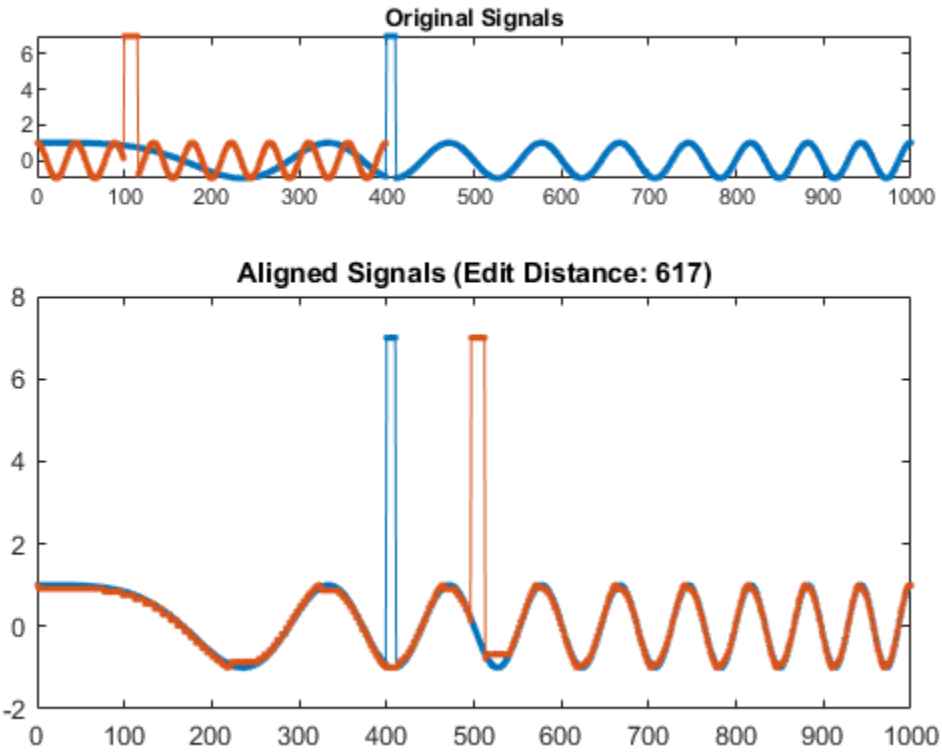
Edit Distance Between Chirp and Sinusoid with Outliers

Generate two real signals: a chirp and a sinusoid. Add a clearly outlying section to each signal.

```
x = cos(2*pi*(3*(1:1000)/1000).^2);
y = cos(2*pi*9*(1:399)/400);

x(400:410) = 7;
y(100:115) = 7;
```

Warp the signals so that the edit distance between them is smallest. Specify a tolerance of 0.1. Plot the aligned signals, both before and after the warping, and output the distance between them.

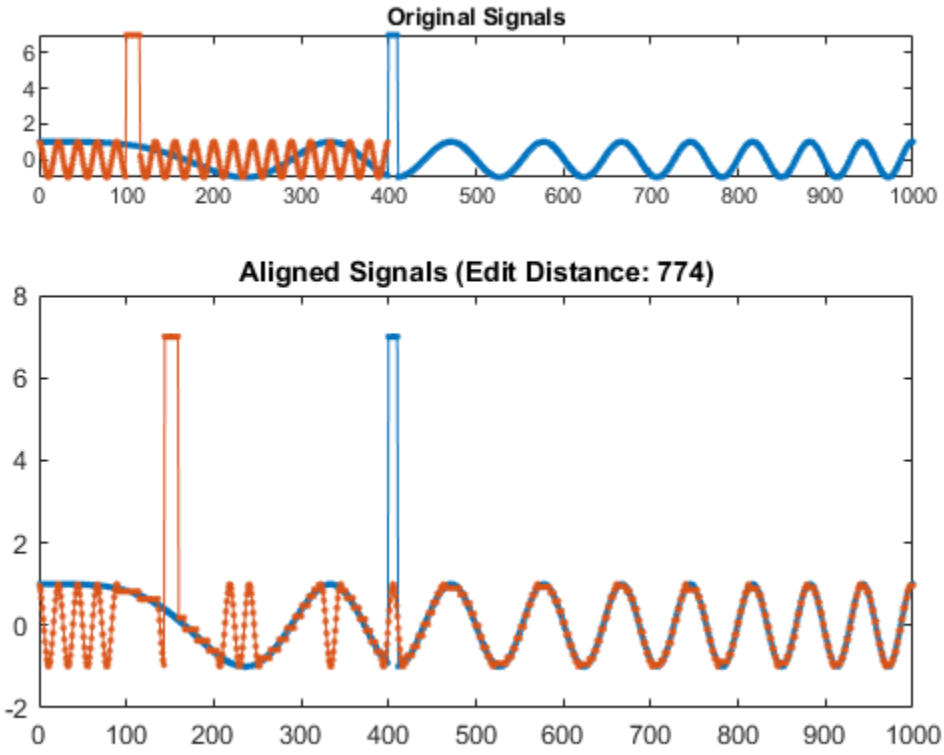
```
tol = 0.1;
edr(x,y,tol)
```




```
ans = 617
```

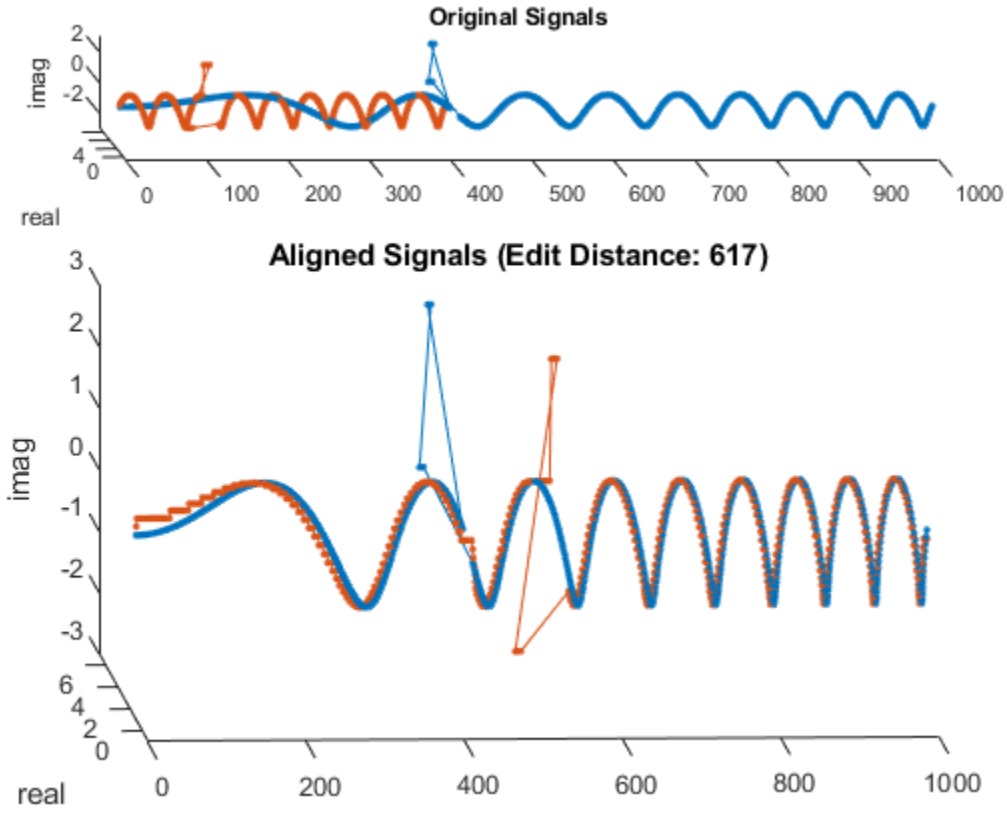
Change the sinusoid frequency to twice its initial value. Repeat the computation.

```
y = cos(2*pi*18*(1:399)/400);
y(100:115) = 7;

edr(x,y,tol);
```



Add an imaginary part to each signal. Restore the initial sinusoid frequency. Align the signals by minimizing the sum of squared Euclidean distances.

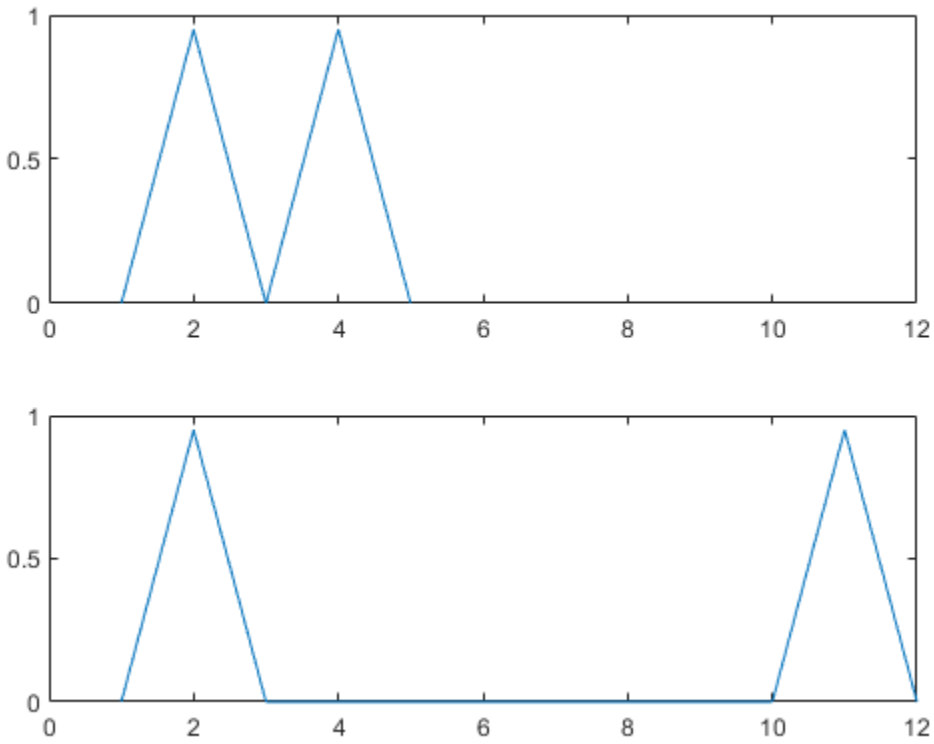
```
x = exp(2i*pi*(3*(1:1000)/1000).^2);
y = exp(2i*pi*9*(1:399)/400);

x(400:405) = 5+3j;
x(405:410) = 7;

y(100:107) = 3j;
y(108:115) = 7-3j;

edr(x,y,tol,'squared');
```



Edit Distance and Warping Path

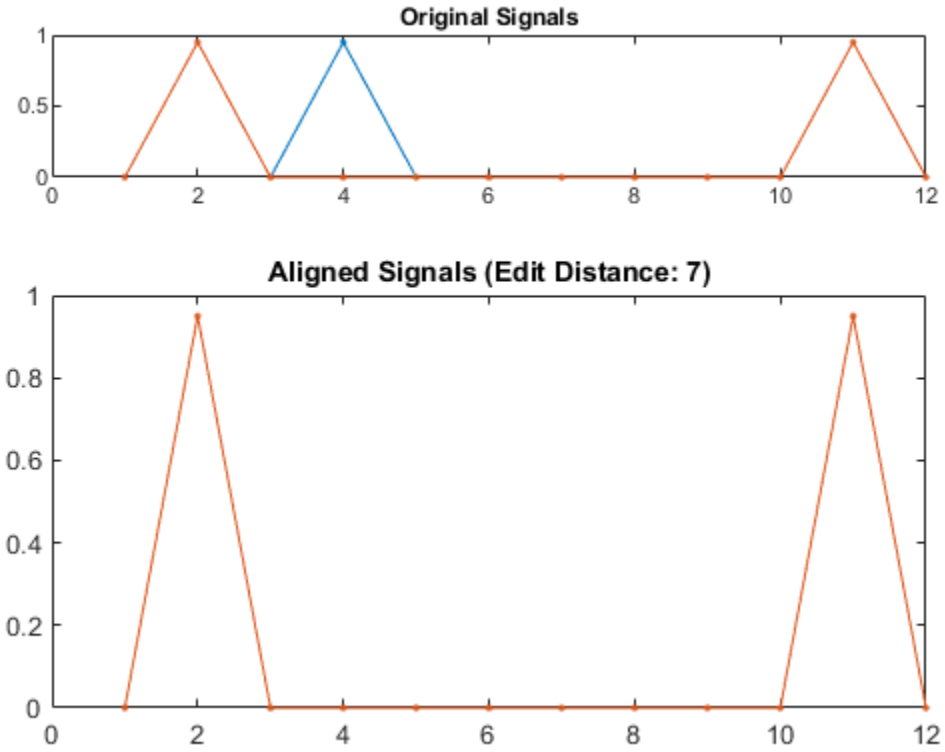
Generate two signals consisting of two distinct peaks separated by valleys of different lengths. Plot the signals.

```
x1 = [0 1 0 1 0]*.95;
x2 = [0 1 0 0 0 0 0 0 0 0 1 0]*.95;

subplot(2,1,1)
plot(x1)
xlim([0 12])
subplot(2,1,2)
plot(x2)
xlim([0 12])
```



Compute the edit distance between the signals. Set a small tolerance so that the only matches are between equal samples.

```
tol = 0.1;

figure
edr(x1,x2,tol);
```



The distance between the signals is 7. To align them, it is necessary to remove the seven central zeros of `x2` or add seven zeros to `x1`.

Compute the **D** matrix, whose bottom-right element corresponds to the edit distance. For the definition of **D**, see “Edit Distance on Real Signals” on page 1-532.

```
cnd = (abs(x1'-x2))>tol;
D = zeros(length(x1)+1,length(x2)+1);
D(1,2:end) = 1:length(x2);
D(2:end,1) = 1:length(x1);

for h = 2:length(x1)+1
    for k = 2:length(x2)+1
        D(h,k) = min([D(h-1,k)+1 ...
            D(h,k-1)+1 ...
            D(h-1,k-1)+cnd(h-1,k-1)]);
    end
end

D
```

```
D = 6×13

     0     1     2     3     4     5     6     7     8     9    10    11    12
     1     0     1     2     3     4     5     6     7     8     9    10    11
     2     1     0     1     2     3     4     5     6     7     8     9    10
     3     2     1     0     1     2     3     4     5     6     7     8     9
```

```
     4     3     2     1     1     2     3     4     5     6     7     7     8
     5     4     3     2     1     1     2     3     4     5     6     7     7
```

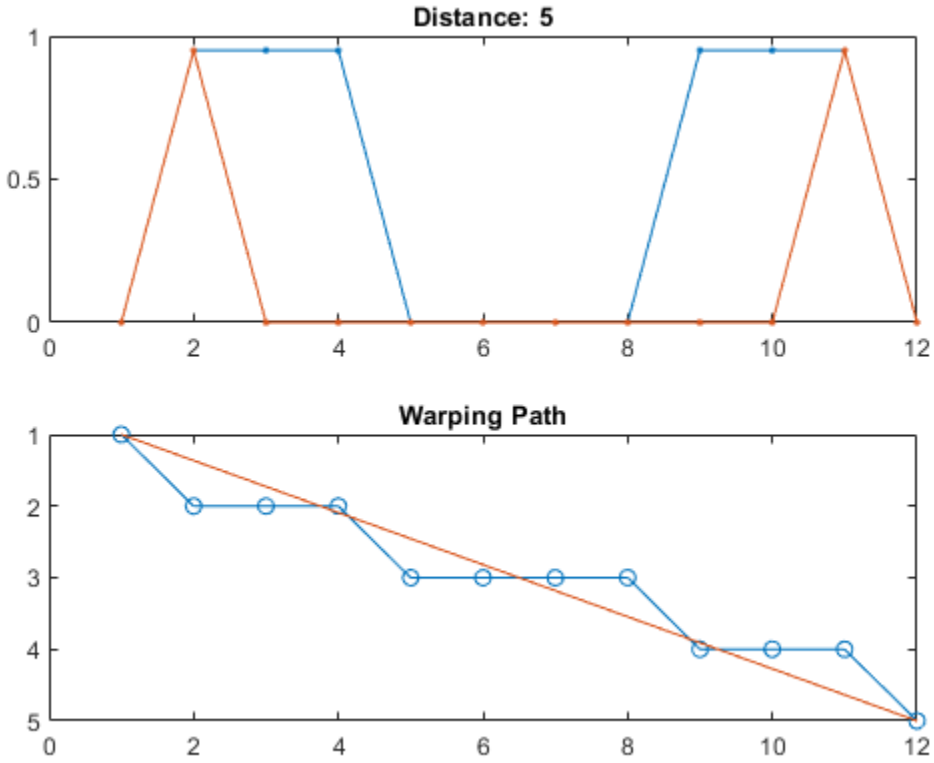
Compute and display the warping path that aligns the signals.

```
[d,i1,i2] = edr(x1,x2,tol);

E = zeros(length(x1),length(x2));

for k = 1:length(i1)
    E(i1(k),i2(k)) = NaN;
end

E
```

```
E = 5×12

   NaN     0     0     0     0     0     0     0     0     0     0     0
     0   NaN     0     0     0     0     0     0     0     0     0     0
     0     0   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN     0     0
     0     0     0     0     0     0     0     0     0     0   NaN     0
     0     0     0     0     0     0     0     0     0     0     0   NaN
```

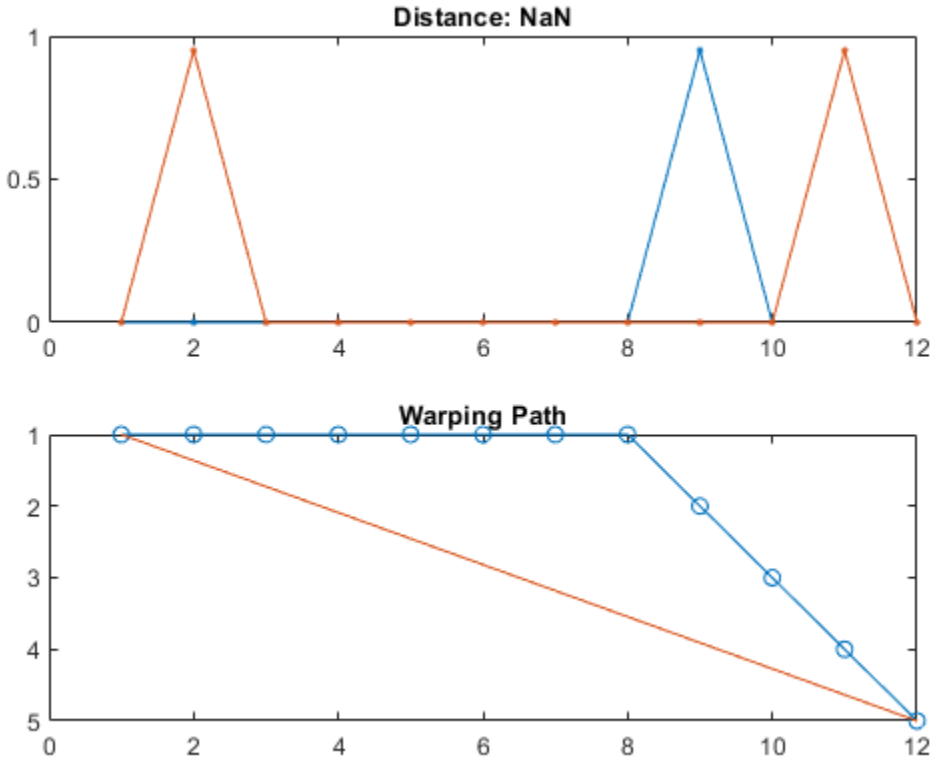
Repeat the computation, but now constrain the warping path to deviate at most two elements from the diagonal. Plot the stretched signals and the warping path. In the second plot, set the matrix columns to run along the *x*-axis.

```
[dc,i1c,i2c] = edr(x1,x2,tol,2);

subplot(2,1,1)
plot([x1(i1c);x2(i2c)]','.-')
title(['Distance: ' num2str(dc)])
subplot(2,1,2)
plot(i2c,i1c,'o-',[i2(1) i2(end)],[i1(1) i1(end)])
axis ij
title('Warping Path')
```



The constraint results in a smaller edit distance but distorts the signals. If the constraint cannot be met, then `edr` returns `NaN` for the distance. See this by forcing the warping path to deviate at most one element from the diagonal.

```
[dc,i1c,i2c] = edr(x1,x2,tol,1);

subplot(2,1,1)
plot([x1(i1c);x2(i2c)]','.-')
title(['Distance: ' num2str(dc)])
subplot(2,1,2)
plot(i2c,i1c,'o-',[i2(1) i2(end)],[i1(1) i1(end)])
axis ij
title('Warping Path')
```



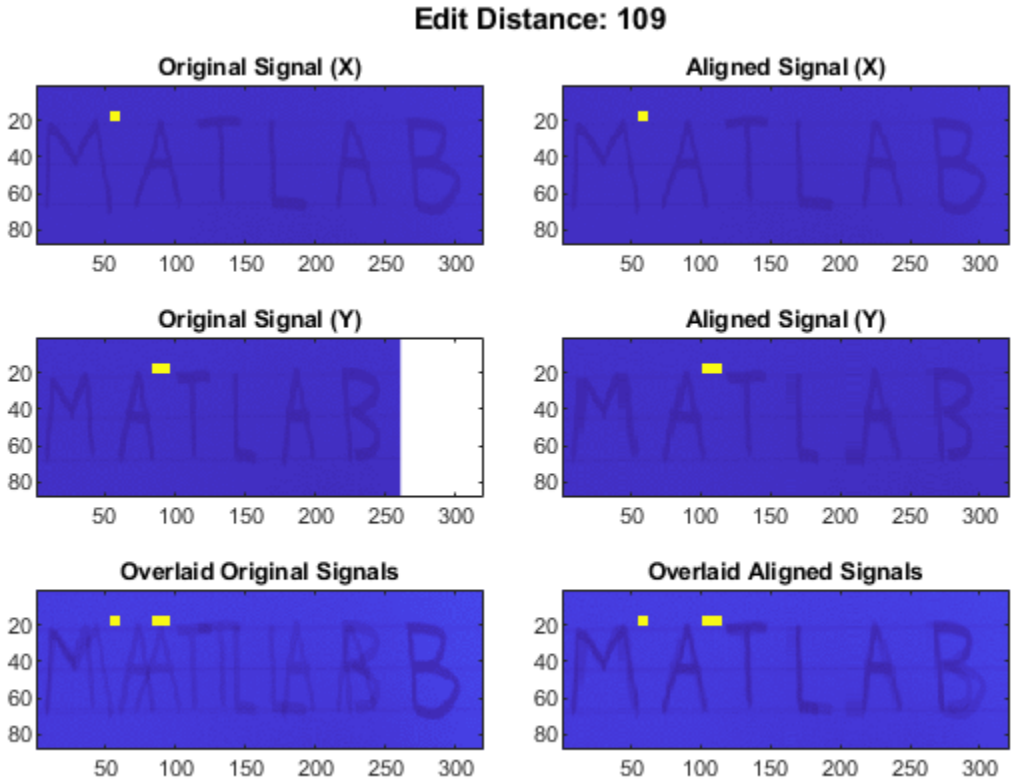
Align Blotched Handwriting Samples

The files `MATLAB1.gif` and `MATLAB2.gif` contain two handwritten samples of the word "MATLAB®." Load the files. Add outliers by blotching the data.

```
samp1 = 'MATLAB1.gif';
samp2 = 'MATLAB2.gif';

x = double(imread(samp1));
y = double(imread(samp2));

x(15:20,54:60) = 4000;
y(15:20,84:96) = 4000;
```

Align the handwriting samples along the *x*-axis using the edit distance. Specify a tolerance of 450.

```
edr(x,y,450);
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a real or complex vector or matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

y — Input signal

vector | matrix

Input signal, specified as a real or complex vector or matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

`tol` — Tolerance

positive scalar

Tolerance, specified as a positive scalar.

Data Types: `single` | `double`

`maxsamp` — Width of adjustment window
`Inf` (default) | positive integer

Width of adjustment window, specified as a positive integer.

Data Types: `single` | `double`

`metric` — Distance metric
`'euclidean'` (default) | `'absolute'` | `'squared'` | `'symmkl'`

Distance metric, specified as `'euclidean'`, `'absolute'`, `'squared'`, or `'symmkl'`. If **X** and **Y** are both *K*-dimensional signals, then `metric` prescribes $d_{mn}(\mathbf{X},\mathbf{Y})$, the distance between the *m*th sample of **X** and the *n*th sample of **Y**.

- `'euclidean'` — Root sum of squared differences, also known as the Euclidean or ℓ_2 metric:

$$d_{mn}(\mathbf{X},\mathbf{Y}) = \sqrt{\sum_{k=1}^{K}(x_{k,m}-y_{k,n})^{*}(x_{k,m}-y_{k,n})}$$

- `'absolute'` — Sum of absolute differences, also known as the Manhattan, city block, taxicab, or ℓ_1 metric:

$$d_{mn}(\mathbf{X},\mathbf{Y}) = \sum_{k=1}^{K}\left|x_{k,m}-y_{k,n}\right| = \sum_{k=1}^{K}\sqrt{(x_{k,m}-y_{k,n})^{*}(x_{k,m}-y_{k,n})}$$

- `'squared'` — Square of the Euclidean metric, consisting of the sum of squared differences:

$$d_{mn}(\mathbf{X},\mathbf{Y}) = \sum_{k=1}^{K}(x_{k,m}-y_{k,n})^{*}(x_{k,m}-y_{k,n})$$

- `'symmkl'` — Symmetric Kullback-Leibler metric. This metric is valid only for real and positive **X** and **Y**:

$$d_{mn}(\mathbf{X},\mathbf{Y}) = \sum_{k=1}^{K}(x_{k,m}-y_{k,n})(\log x_{k,m}-\log y_{k,n})$$

Output Arguments

`dist` — Minimum distance
positive real scalar

Minimum distance between signals, returned as a positive real scalar.

`ix,iy` — Warping path
vectors of indices

Warping path, returned as vectors of indices. `ix` and `iy` have the same length. Each vector contains a monotonically increasing sequence in which the indices to the elements of the corresponding signal, `x` or `y`, are repeated the necessary number of times.

More About

Edit Distance on Real Signals

Two signals with equivalent features arranged in the same order can appear very different due to differences in the durations of their sections. `edr` distorts these durations so that the corresponding features appear at the same location on a common time axis, thus highlighting the similarities between the signals. The criterion used to perform the distortion is designed to be robust to outliers.

Consider the two K-dimensional signals

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,M} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{K,1} & x_{K,2} & \cdots & x_{K,M} \end{bmatrix}$$

and

$$\mathbf{Y} = \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,N} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ y_{K,1} & y_{K,2} & \cdots & y_{K,N} \end{bmatrix},$$

which have M and N samples, respectively. Given $d_{mn}(\mathbf{X},\mathbf{Y})$, the distance between the mth sample of $\mathbf{X}$ and the nth sample of $\mathbf{Y}$ specified in `metric`, the `edr` function stretches $\mathbf{X}$ and $\mathbf{Y}$ onto a common set of instants such that the *edit distance* between the signals is smallest.

Given ε, a real number that is the tolerance specified in `tol`, declare that the mth sample of $\mathbf{X}$ and the nth sample of $\mathbf{Y}$ *match* if $d_{mn}(\mathbf{X},\mathbf{Y}) < \varepsilon$. If two samples, m and n, do not match, you can make them match in any of three ways:

1. Remove m from the first signal, such as when the next sample does match n. This removal is equivalent to adding m to the second signal and obtaining two consecutive matches.
2. Lengthen the first signal by adding in position a sample that matches n and displacing the rest of the samples by one location. This addition is equivalent to removing the unmatched n from the second signal.
3. Substitute m with n in the first signal, or, equivalently, remove both m and n.

The edit distance is the total number of these operations that are needed to make the two signals match. This number is not unique. To compute the smallest possible edit distance between $\mathbf{X}$ and $\mathbf{Y}$, start from these facts:

1. Two empty signals have zero distance between them.
2. The distance between an empty signal and a signal with L samples is L, because that is the number of samples that must be added to the empty signal to recover the other one. Equivalently, L is the number of samples that must be removed from an L-sample signal to empty it.

Create an $(M + 1)$-by-$(N + 1)$ matrix, $\mathbf{D}$, such that:

1. $D_{1,1} = 0$.
2. $D_{m,1} = m - 1$ for $m = 2, \ldots, M + 1$.

3 $D_{1,n} = n - 1$ for $n = 2, \ldots, N + 1$.

4 For $m, n > 1$,

$$D_{m,n} = \min\left\{\begin{array}{c} D_{m-1,n} + 1 \\ D_{m,n-1} + 1 \\ D_{m-1,n-1} + \begin{cases} 0 \Leftarrow d_{m,n}(\mathbf{X},\mathbf{Y}) \le \varepsilon \\ 1 \Leftarrow d_{m,n}(\mathbf{X},\mathbf{Y}) > \varepsilon \end{cases} \end{array}\right\}.$$

The smallest edit distance between **X** and **Y** is then $D_{M+1,N+1}$.

The *warping path* through **D** that results in this smallest edit distance is parameterized by two sequences of the same length, `ix` and `iy`, and is a combination of “chess king” moves:

- Vertical moves: $(m,n) \rightarrow (m + 1,n)$ corresponds to removing a sample from **X** or adding a sample to **Y**. Each move increases the edit distance by 1.
- Horizontal moves: $(m,n) \rightarrow (m,n + 1)$ corresponds to removing a sample from **Y** or adding a sample to **X**. Each move increases the edit distance by 1.
- Diagonal moves: $(m,n) \rightarrow (m + 1,n + 1)$ corresponds to a match if $d_{m,n}(\mathbf{X},\mathbf{Y}) \le \varepsilon$ or corresponds to removing one sample from each signal if $d_{m,n}(\mathbf{X},\mathbf{Y}) > \varepsilon$. Matches do not increase the distance. Removals increase it by 1.

This structure ensures that any acceptable path aligns the complete signals, does not skip samples, and does not repeat signal features. Additionally, a desirable path runs close to the diagonal line extended between $d_{1,1}(\mathbf{X},\mathbf{Y})$ and $d_{M,N}(\mathbf{X},\mathbf{Y})$. This extra constraint, adjusted by the `maxsamp` argument, ensures that the warping compares sections of similar length.

The penalty for making two samples match is independent of the difference in value between the samples. Two samples that differ by a little more than the tolerance incur the same penalty as two samples that are markedly different. For that reason, the edit distance is not affected by outliers. Conversely, repeating samples to align two signals has a cost, which is not the case with dynamic time warping.

References

[1] Chen, Lei, M. Tamer Özsu, and Vincent Oria. "Robust and Fast Similarity Search for Moving Object Trajectories." *Proceedings of 24th ACM International Conference on Management of Data (SIGMOD ‘05)*. 2005, pp. 491–502.

[2] Paliwal, K. K., Anant Agarwal, and Sarvajit S. Sinha. "A Modification over Sakoe and Chiba’s Dynamic Time Warping Algorithm for Isolated Word Recognition." *Signal Processing*. Vol. 4, 1982, pp. 329–333.

[3] Sakoe, Hiroaki, and Seibi Chiba. "Dynamic Programming Algorithm Optimization for Spoken Word Recognition." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. ASSP-26, No. 1, 1978, pp. 43–49.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`alignsignals` | `dtw` | `finddelay` | `findsignal` | `xcorr`

Introduced in R2016b

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 `Wp`. The resulting filter has `Rp` decibels of peak-to-peak passband ripple and `Rs` 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 `Wp`. The resulting bandpass and bandstop designs are of order `2n`.

Note: See “Limitations” on page 1-543 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 `Wp`, `Rp` decibels of passband ripple, and `Rs` decibels of stopband attenuation.

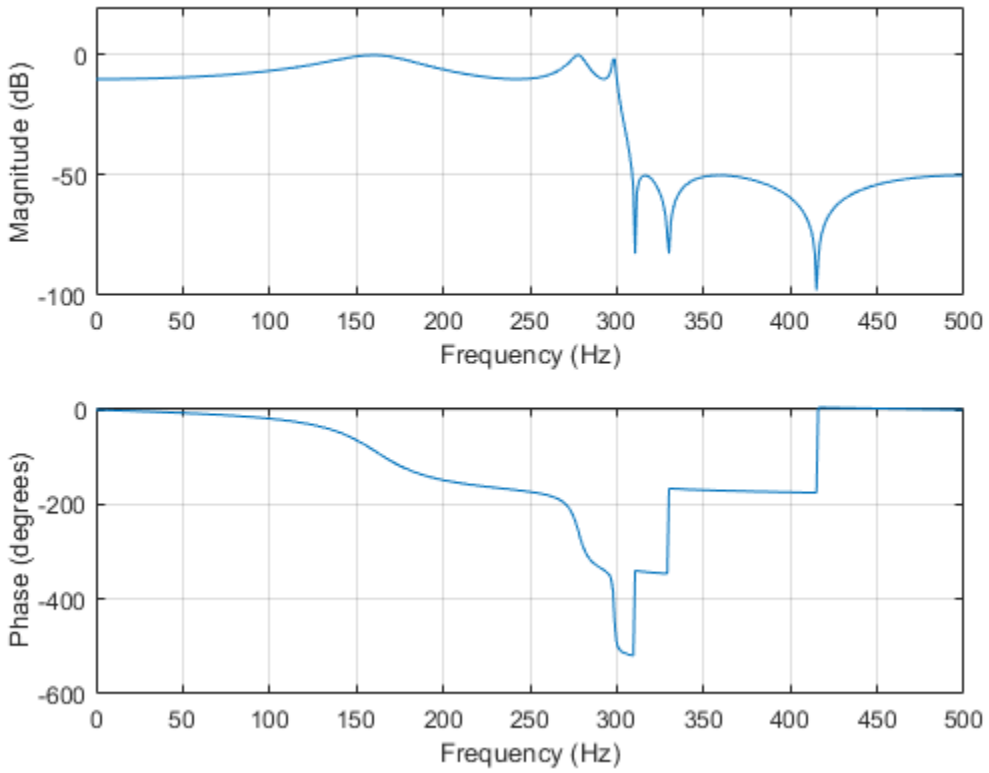
Examples

Lowpass Elliptic Transfer Function

Design a 6th-order lowpass elliptic filter with 10 dB of passband ripple, 50 dB of stopband attenuation, and a passband edge frequency of 300 Hz, which, for data sampled at 1000 Hz, corresponds to 0.6π rad/sample. Plot its magnitude and phase responses. Use it to filter a 1000-sample random signal.

```
fc = 300;
fs = 1000;
```

```
[b,a] = ellip(6,10,50,fc/(fs/2));

freqz(b,a,[],fs)

subplot(2,1,1)
ylim([-100 20])
```

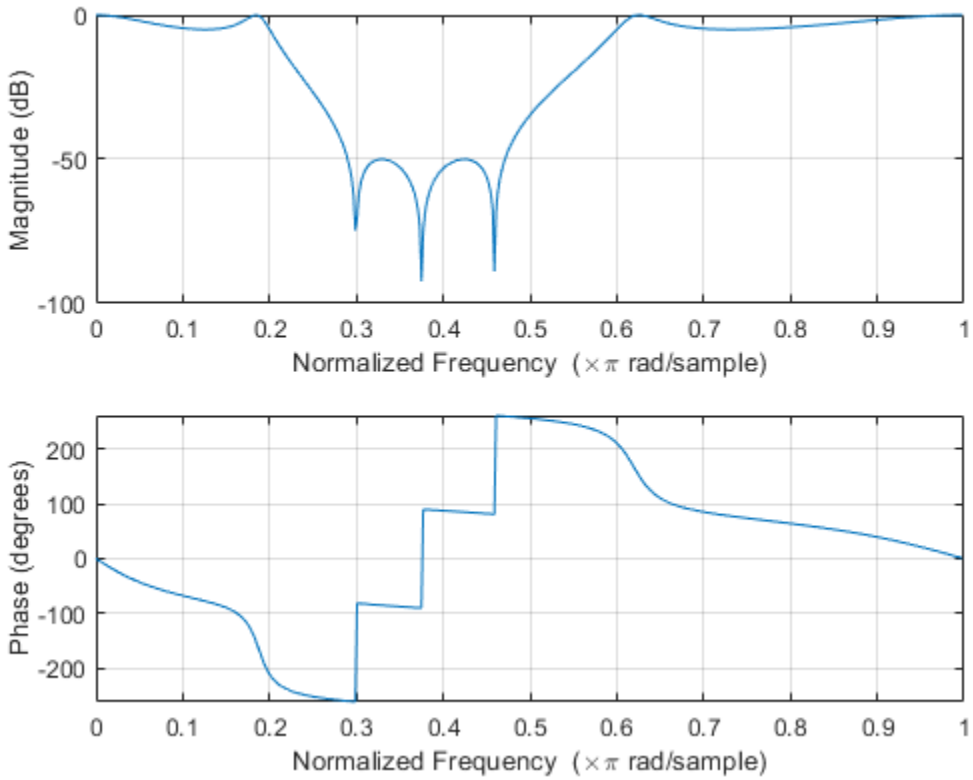



```
dataIn = randn(1000,1);
dataOut = filter(b,a,dataIn);
```

Bandstop Elliptic Filter

Design a 6th-order elliptic bandstop filter with normalized edge frequencies of 0.2π and 0.6π 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)
```

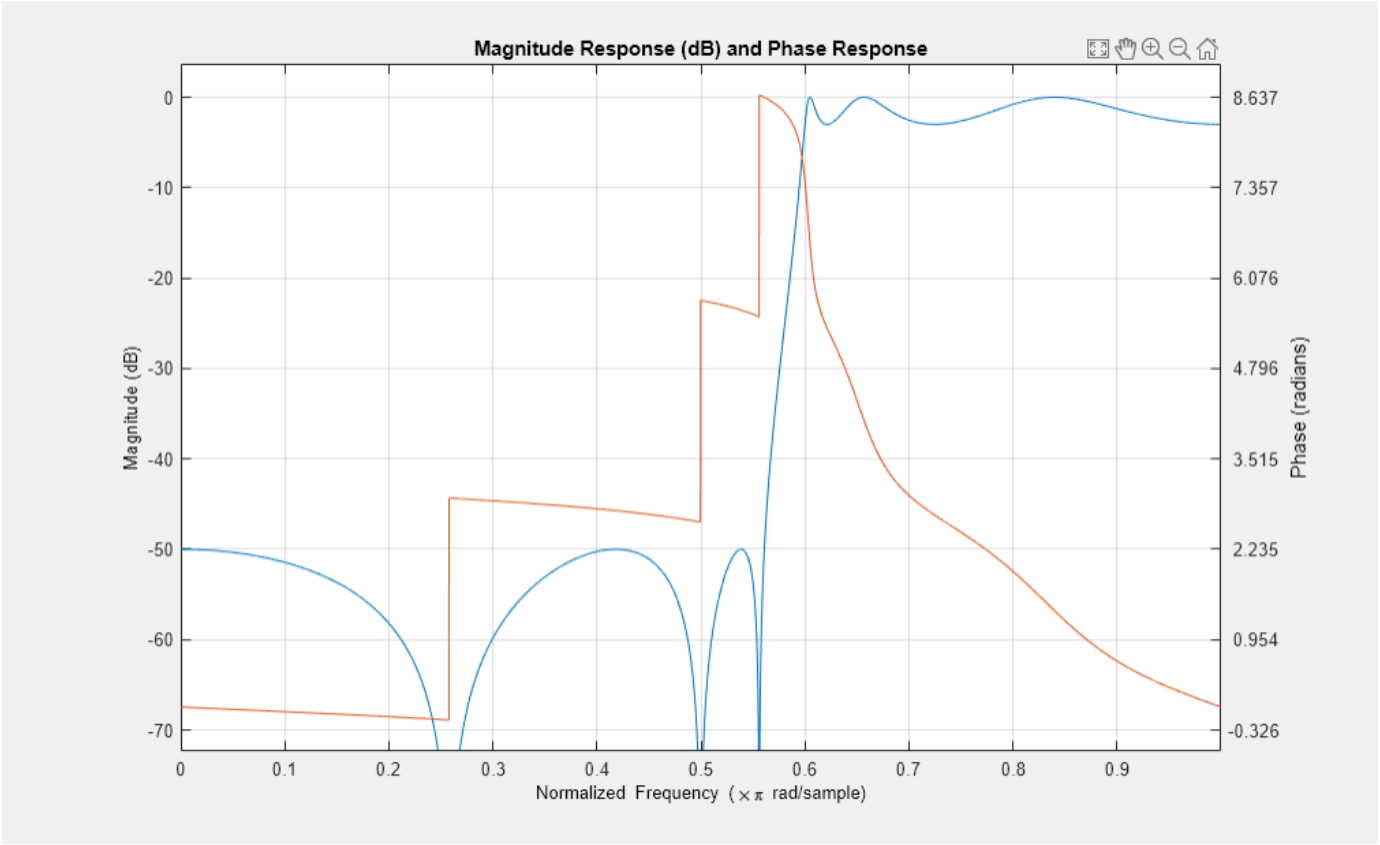



```
dataIn = randn(1000,1);
dataOut = filter(b,a,dataIn);
```

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 0.6π 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')
```



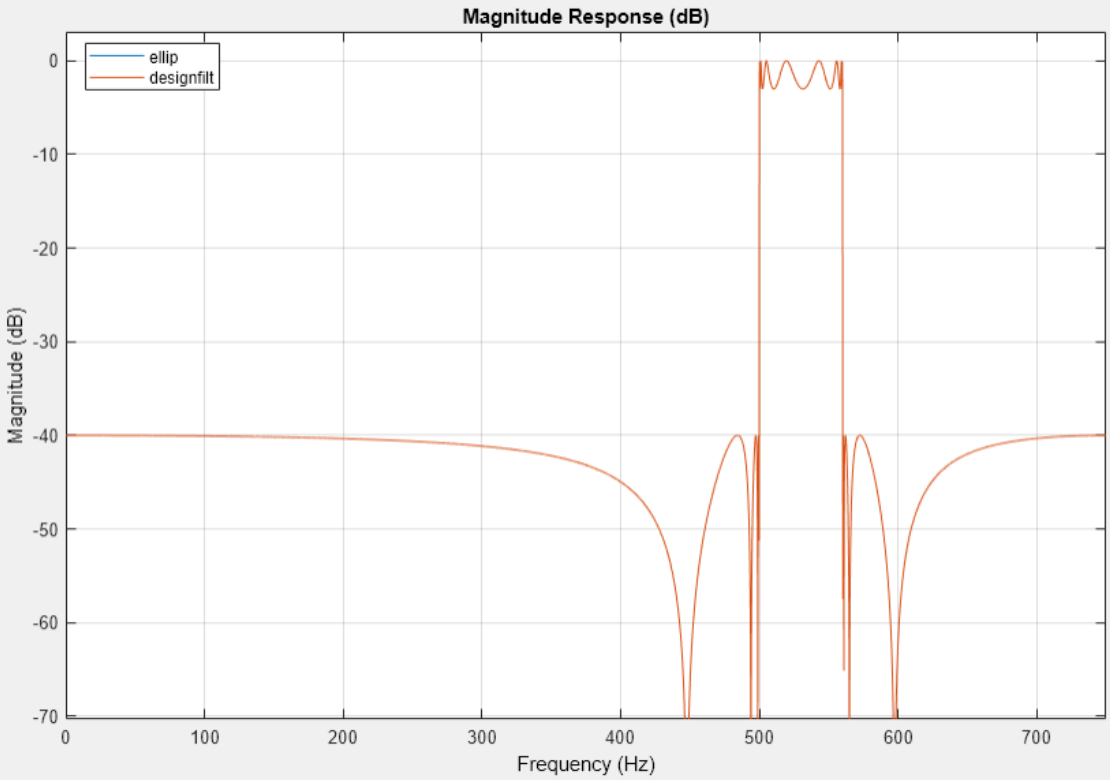
Bandpass Elliptic Filter

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 2π 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.

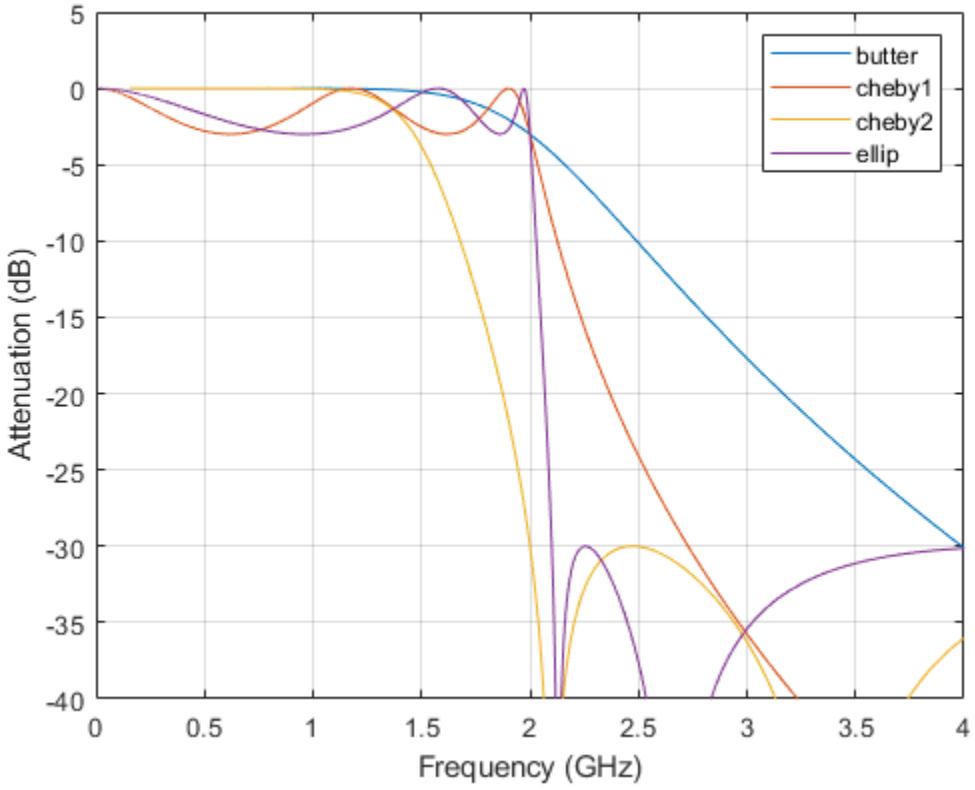
```
[z2,p2,k2] = cheby2(n,30,2*pi*f,'s');
[b2,a2] = zp2tf(z2,p2,k2);
[h2,w2] = freqs(b2,a2,4096);
```

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. For bandpass and bandstop designs, `n` represents one-half the filter order.

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, ℓ, is in linear units, you can convert it to decibels using $Rp = 40 \log_{10}((1+\ell)/(1-\ell))$.

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

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 –`Rp` decibels. Smaller values of passband ripple, `Rp`, and larger values of stopband attenuation, `Rs`, both result in wider transition bands.

- If `Wp` is a scalar, then `ellip` designs a lowpass or highpass filter with edge frequency `Wp`.

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

`ftype` — Filter type

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

Filter type, specified as one of the following:

- `'low'` specifies a lowpass filter with passband edge frequency `Wp`. `'low'` is the default for scalar `Wp`.
- `'high'` specifies a highpass filter with passband edge frequency `Wp`.
- `'bandpass'` specifies a bandpass filter of order 2`n` if `Wp` is a two-element vector. `'bandpass'` is the default when `Wp` has two elements.
- `'stop'` specifies a bandstop filter of order 2`n` if `Wp` is a two-element vector.

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 2`n` + 1 for bandpass and bandstop filters.

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

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

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

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

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` (2`n` for bandpass and bandstop designs) and a scalar.

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

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

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

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

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 = 2`n` for bandpass and bandstop filters, then `A` is $m \times m$, `B` is $m \times 1$, `C` is $1 \times m$, and `D` is 1×1.

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

$$x(k+1) = \mathrm{A}\,x(k) + \mathrm{B}\,u(k)$$
$$y(k) = \mathrm{C}\,x(k) + \mathrm{D}\,u(k)\,.$$

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

$$\dot{x} = \mathrm{A}\,x + \mathrm{B}\,u$$
$$y = \mathrm{C}\,x + \mathrm{D}\,u\,.$$

Data Types: `double`

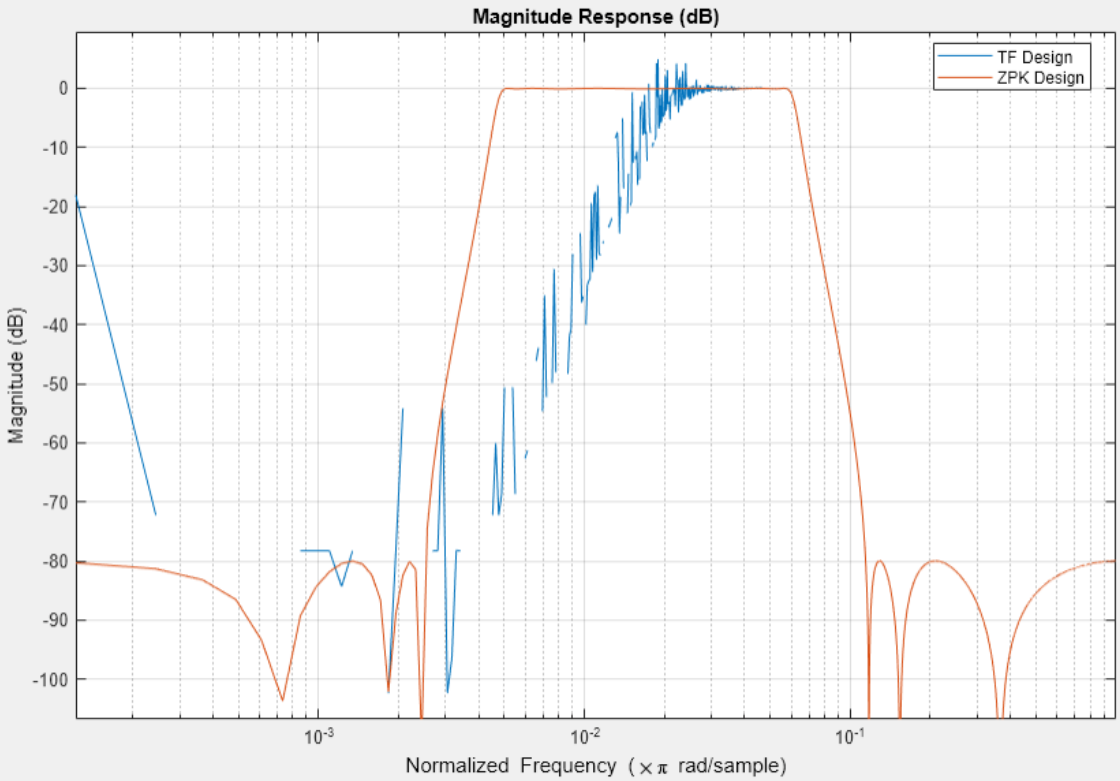
More About

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 `Wp` or `w1` and `w2`.
5. It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`besself` | `butter` | `cheby1` | `cheby2` | `designfilt` | `ellipap` | `ellipord` | `filter` | `sosfilt`

Introduced before R2006a

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 `Rp` dB of ripple in the passband, and a stopband `Rs` dB down from the peak value in the passband. The zeros and poles are returned in length `n` column vectors `z` and `p` and the gain in scalar `k`. If `n` is odd, `z` is length `n - 1`. The transfer function in factored zero-pole form is

$$H(s) = \frac{z(s)}{p(s)} = k\frac{(s-z_1)(s-z_2)...(s-z_N)}{(s-p_1)(s-p_2)...(s-p_M)}$$

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 ω_0 of the elliptic filter to 1 for a normalized result. The *passband edge angular frequency* is the frequency at which the passband ends and the filter has a magnitude response of $10^{-Rp/20}$.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`besselap` | `buttap` | `cheb1ap` | `cheb2ap` | `ellip`

Introduced before R2006a

ellipord

Minimum order for elliptic filters

Syntax

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

Description

`[n,Wn] = ellipord(Wp,Ws,Rp,Rs)` returns the lowest order, `n`, of the digital elliptic filter with no more than `Rp` dB of passband ripple and at least `Rs` dB of attenuation in the stopband. `Wp` and `Ws`, are respectively, the passband and stopband edge frequencies of the filter, normalized from 0 to 1, where 1 corresponds to π rad/sample. The scalar (or vector) of corresponding cutoff frequencies, `Wn`, is also returned. To design an elliptic filter, use the output arguments `n` and `Wn` as inputs to `ellip`.

`[n,Wn] = ellipord(Wp,Ws,Rp,Rs,'s')` finds the minimum order `n` and cutoff frequencies `Wn` for an analog elliptic filter. Specify the frequencies `Wp` and `Ws` in radians per second. The passband or the stopband can be infinite.

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.

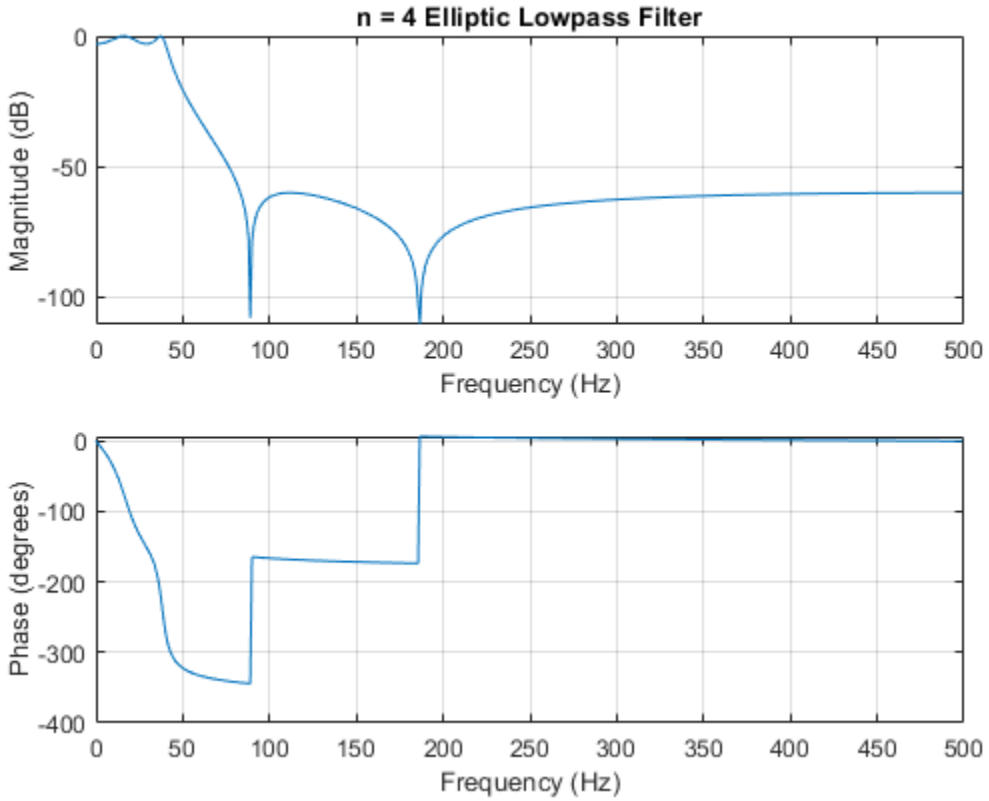
```
Wp = 40/500;
Ws = 150/500;
Rp = 3;
Rs = 60;
[n,Wp] = ellipord(Wp,Ws,Rp,Rs)
```

```
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.

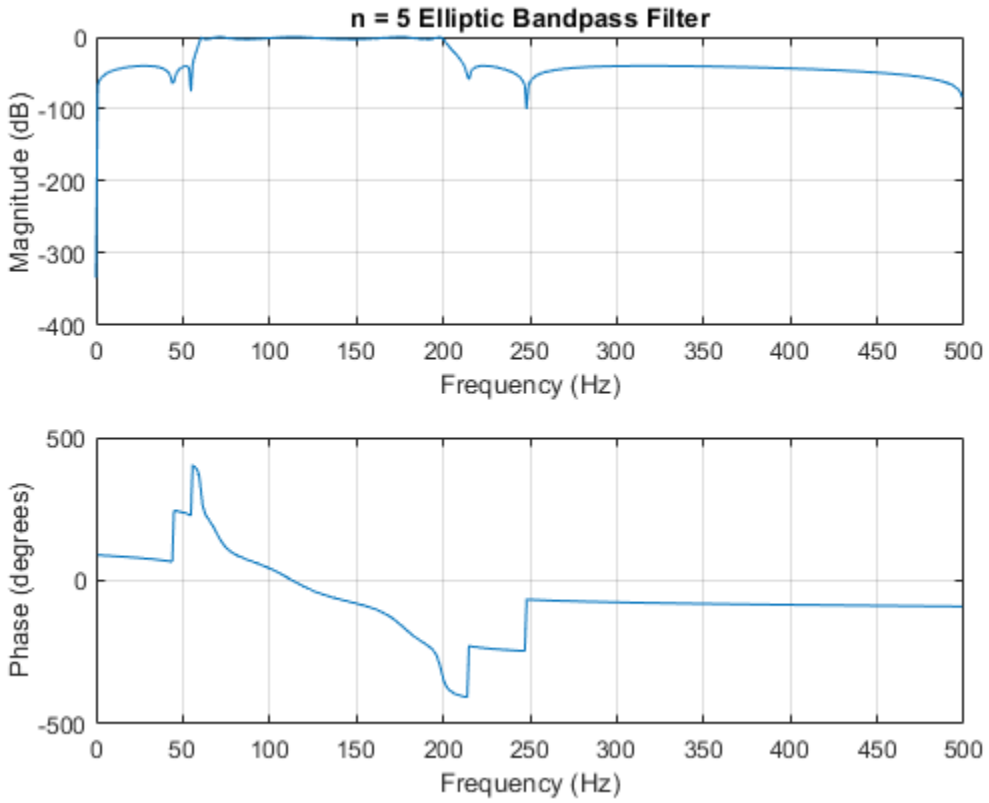
```
Wp = [60 200]/500;
Ws = [50 250]/500;
Rp = 3;
Rs = 40;

[n,Wp] = ellipord(Wp,Ws,Rp,Rs)
```

```
n = 5

Wp = 1×2

    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))
```



Input Arguments

Wp — Passband corner (cutoff) frequency

scalar | two-element vector

Passband corner (cutoff) frequency, specified as a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, π rad/sample.

- If `Wp` and `Ws` are both scalars and `Wp < Ws`, then `ellipord` returns the order and cutoff frequency of a lowpass filter. The stopband of the filter ranges from `Ws` to 1 and the passband ranges from 0 to `Wp`.
- If `Wp` and `Ws` are both scalars and `Wp > Ws`, then `ellipord` returns the order and cutoff frequency of a highpass filter. The stopband of the filter ranges from 0 to `Ws` and the passband ranges from `Wp` to 1.
- If `Wp` and `Ws` are both vectors and the interval specified by `Ws` contains the one specified by `Wp` (`Ws(1) < Wp(1) < Wp(2) < Ws(2)`), then `ellipord` returns the order and cutoff frequencies of a bandpass filter. The stopband of the filter ranges from 0 to `Ws(1)` and from `Ws(2)` to 1. The passband ranges from `Wp(1)` to `Wp(2)`.
- If `Wp` and `Ws` are both vectors and the interval specified by `Wp` contains the one specified by `Ws` (`Wp(1) < Ws(1) < Ws(2) < Wp(2)`), then `ellipord` returns the order and cutoff frequencies of a

bandstop filter. The stopband of the filter ranges from `Ws(1)` to `Ws(2)`. The passband ranges from 0 to `Wp(1)` and from `Wp(2)` to 1.

Data Types: `single` | `double`

Note 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 and cascade the two filters together.

`Ws` — Stopband corner frequency

scalar | two-element vector

Stopband corner frequency, specified as a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, π rad/sample.

Data Types: `single` | `double`

`Rp` — Passband ripple

scalar

Passband ripple, specified as a scalar expressed in dB.

Data Types: `single` | `double`

`Rs` — Stopband attenuation

scalar

Stopband attenuation, specified as a scalar expressed in dB.

Data Types: `single` | `double`

Output Arguments

`n` — Lowest filter order

integer scalar

Lowest filter order, returned as an integer scalar.

`Wn` — Cutoff frequencies

scalar | vector

Cutoff frequencies, returned as a scalar or vector.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`buttord` | `cheb1ord` | `cheb2ord` | `ellip`

Introduced before R2006a

emd

Empirical mode decomposition

Syntax

```
[imf,residual] = emd(x)
[imf,residual,info] = emd(x)
[___] = emd(___,Name,Value)

emd(___)
```

Description

`[imf,residual] = emd(x)` returns intrinsic mode functions `imf` and residual signal `residual` corresponding to the empirical mode decomposition of `x`. Use `emd` to decompose and simplify complicated signals into a finite number of intrinsic mode functions required to perform Hilbert spectral analysis.

`[imf,residual,info] = emd(x)` returns additional information `info` on IMFs and residual signal for diagnostic purposes.

`[___] = emd(___,Name,Value)` performs the empirical mode decomposition with additional options specified by one or more `Name,Value` pair arguments.

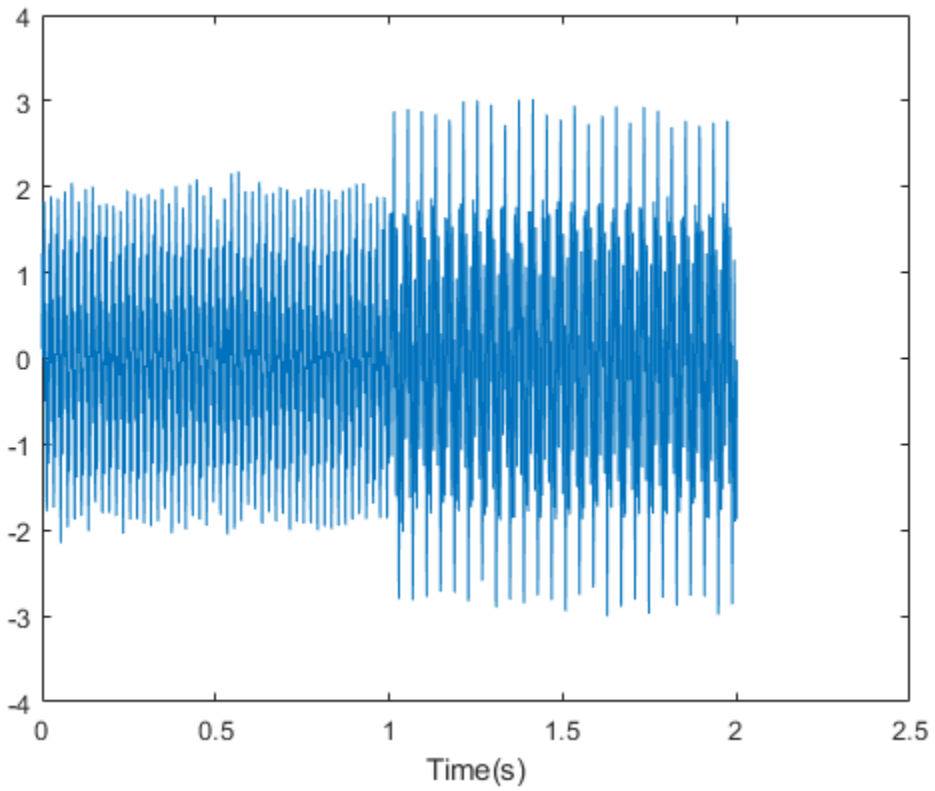
`emd(___)` plots the original signal, IMFs, and residual signal as subplots in the same figure.

Examples

Perform Empirical Mode Decomposition and Visualize Hilbert Spectrum of Signal

Load and visualize a nonstationary continuous signal composed of sinusoidal waves with a distinct change in frequency. The vibration of a jackhammer and the sound of fireworks are examples of nonstationary continuous signals. The signal is sampled at a rate `fs`.

```
load('sinusoidalSignalExampleData.mat','X','fs')
t = (0:length(X)-1)/fs;
plot(t,X)
xlabel('Time(s)')
```



The mixed signal contains sinusoidal waves with different amplitude and frequency values.

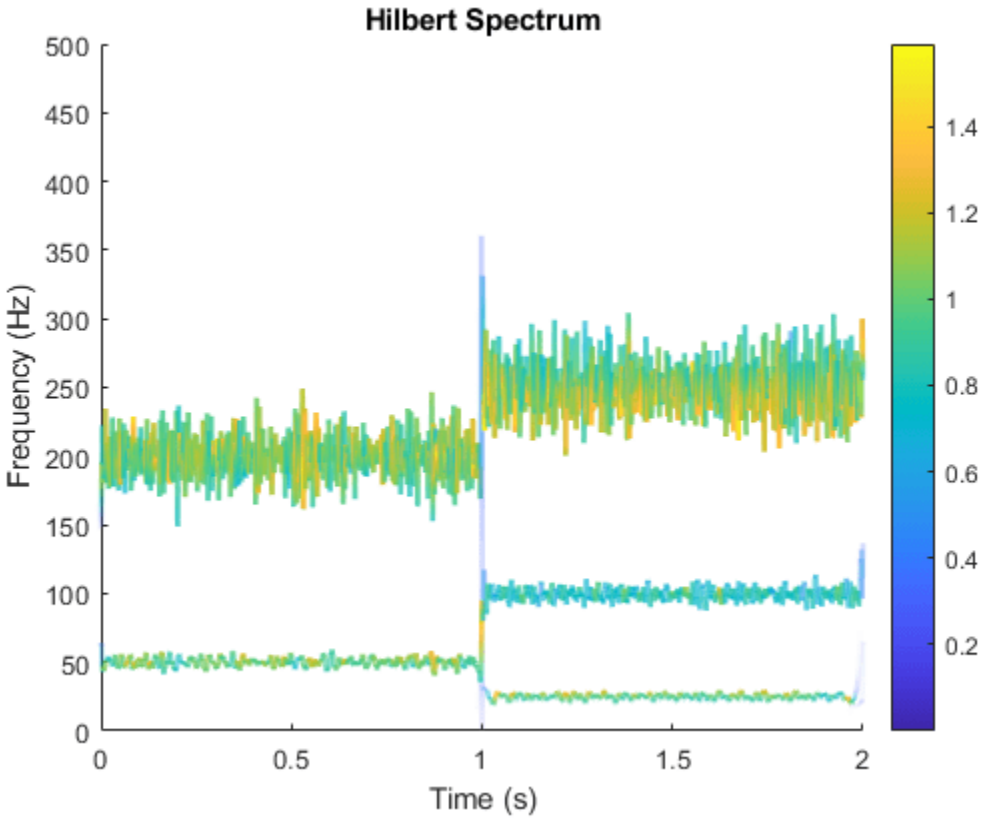
To create the Hilbert spectrum plot, you need the intrinsic mode functions (IMFs) of the signal. Perform empirical mode decomposition to compute the IMFs and residuals of the signal. Since the signal is not smooth, specify '`pchip`' as the interpolation method.

```
[imf,residual,info] = emd(X,'Interpolation','pchip');
```

The table generated in the command window indicates the number of sift iterations, the relative tolerance, and the sift stop criterion for each generated IMF. This information is also contained in `info`. You can hide the table by adding the `'Display',0` name value pair.

Create the Hilbert spectrum plot using the `imf` components obtained using empirical mode decomposition.

```
hht(imf,fs)
```



The frequency versus time plot is a sparse plot with a vertical color bar indicating the instantaneous energy at each point in the IMF. The plot represents the instantaneous frequency spectrum of each component decomposed from the original mixed signal. Three IMFs appear in the plot with a distinct change in frequency at 1 second.

Zero Crossings and Extrema in Intrinsic Mode Function of Sinusoid

This trigonometric identity presents two different views of the same physical signal:

$$\frac{5}{2}\cos 2\pi f_1 t + \frac{1}{4}\left(\cos 2\pi (f_1 + f_2)t + \cos 2\pi (f_1 - f_2)t\right) = \left(2 + \cos^2 \pi f_2 t\right)\cos 2\pi f_1 t.$$

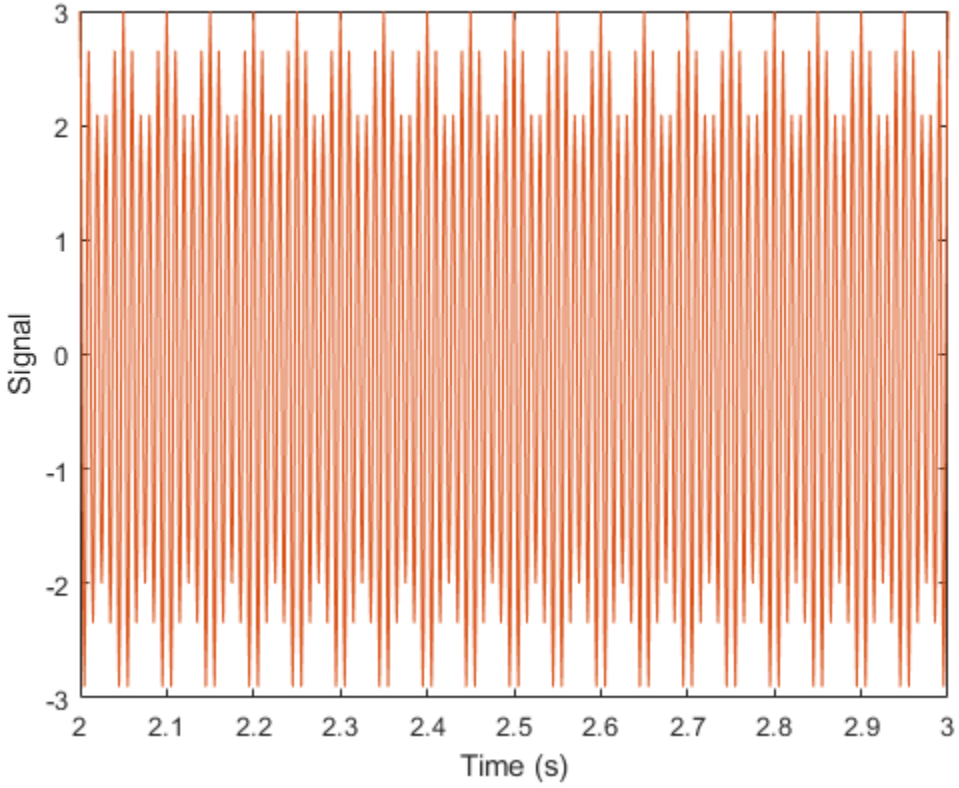
Generate two sinusoids, `s` and `z`, such that `s` is the sum of three sine waves and `z` is a single sine wave with a modulated amplitude. Verify that the two signals are equal by calculating the infinity norm of their difference.

```
t = 0:1e-3:10;
omega1 = 2*pi*100;
omega2 = 2*pi*20;
s = 0.25*cos((omega1-omega2)*t) + 2.5*cos(omega1*t) + 0.25*cos((omega1+omega2)*t);
z = (2+cos(omega2/2*t).^2).*cos(omega1*t);

norm(s-z,Inf)
```

```
ans = 3.2729e-13
```

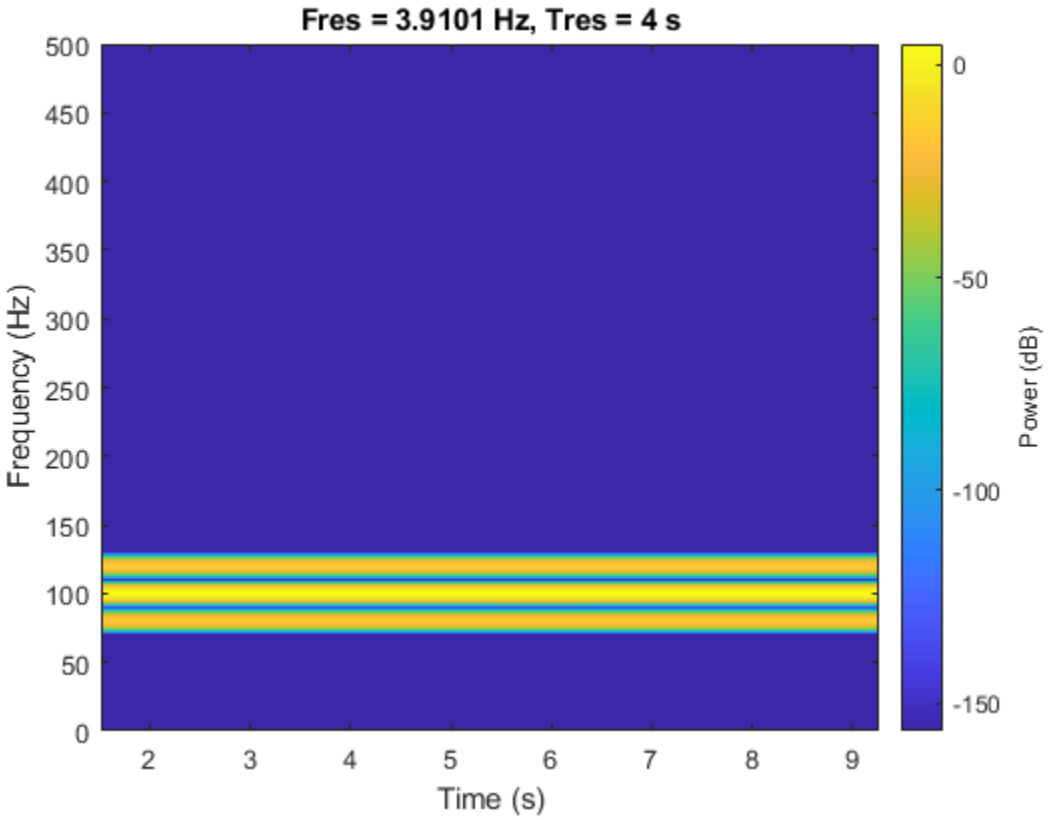
Plot the sinusoids and select a 1-second interval starting at 2 seconds.

```
plot(t,[s' z'])
xlim([2 3])
xlabel('Time (s)')
ylabel('Signal')
```



Obtain the spectrogram of the signal. The spectrogram shows three distinct sinusoidal components. Fourier analysis sees the signals as a superposition of sine waves.

```
pspectrum(s,1000,'spectrogram','TimeResolution',4)
```



Use `emd` to compute the intrinsic mode functions (IMFs) of the signal and additional diagnostic information. The function by default outputs a table that indicates the number of sifting iterations, the relative tolerance, and the sifting stop criterion for each IMF. Empirical mode decomposition sees the signal as `z`.

```
[imf,~,info] = emd(s);
```

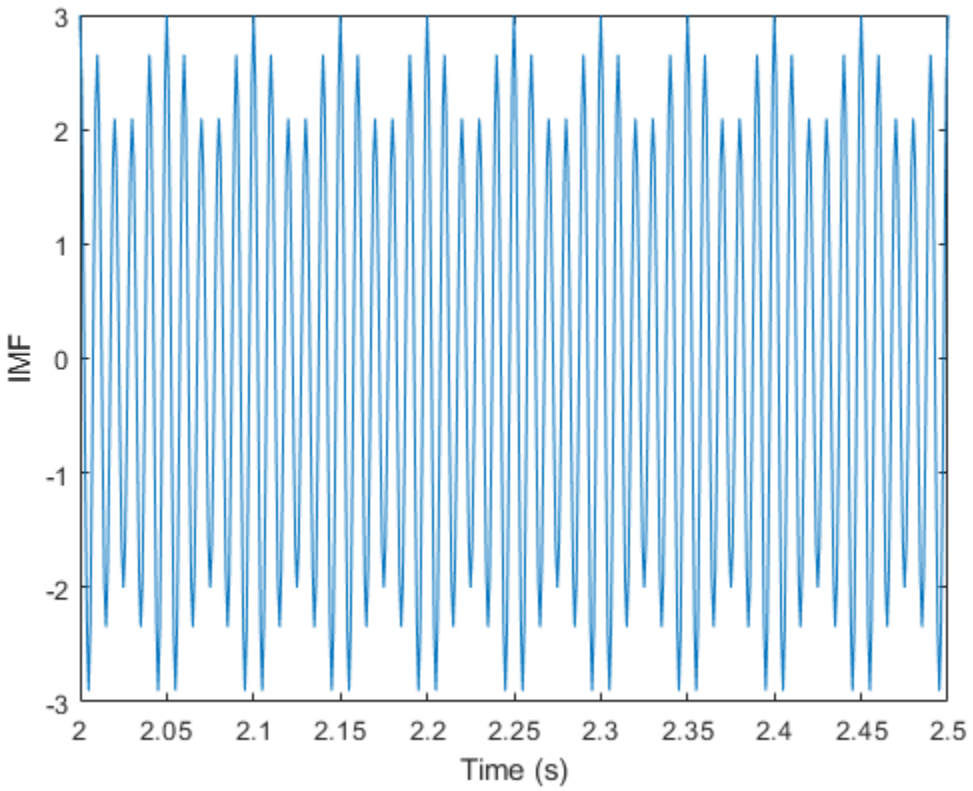
The number of zero crossings and local extrema differ by at most one. This satisfies the necessary condition for the signal to be an IMF.

```
info.NumZerocrossing - info.NumExtrema
```

```
ans = 1
```

Plot the IMF and select a 0.5-second interval starting at 2 seconds. The IMF is an AM signal because `emd` views the signal as amplitude modulated.

```
plot(t,imf)
xlim([2 2.5])
xlabel('Time (s)')
ylabel('IMF')
```

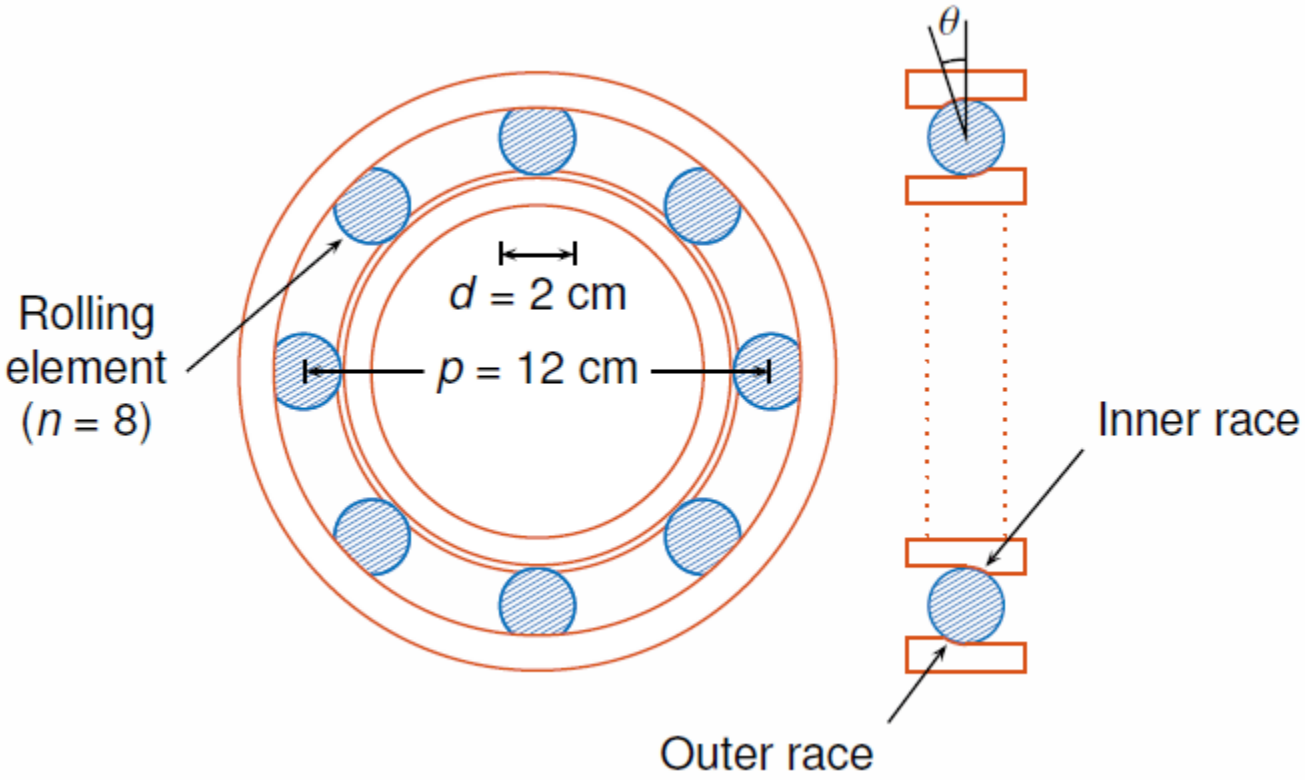


Compute Intrinsic Mode Functions of Vibration Signal

Simulate a vibration signal from a damaged bearing. Perform empirical mode decomposition to visualize the IMFs of the signal and look for defects.

A bearing with a pitch diameter of 12 cm has eight rolling elements. Each rolling element has a diameter of 2 cm. The outer race remains stationary as the inner race is driven at 25 cycles per second. An accelerometer samples the bearing vibrations at 10 kHz.

```
fs = 10000;
f0 = 25;
n = 8;
d = 0.02;
p = 0.12;
```



The vibration signal from the healthy bearing includes several orders of the driving frequency.

```
t = 0:1/fs:10-1/fs;
yHealthy = [1 0.5 0.2 0.1 0.05]*sin(2*pi*f0*[1 2 3 4 5]'.*t)/5;
```

A resonance is excited in the bearing vibration halfway through the measurement process.

```
yHealthy = (1+1./(1+linspace(-10,10,length(yHealthy)).^4)).*yHealthy;
```

The resonance introduces a defect in the outer race of the bearing that results in progressive wear. The defect causes a series of impacts that recur at the ball pass frequency outer race (BPFO) of the bearing:

$$\text{BPFO} = \frac{1}{2} n f_0 \left[1 - \frac{d}{p}\cos\theta\right],$$

where f_0 is the driving rate, n is the number of rolling elements, d is the diameter of the rolling elements, p is the pitch diameter of the bearing, and θ is the bearing contact angle. Assume a contact angle of 15° and compute the BPFO.

```
ca = 15;
bpfo = n*f0/2*(1-d/p*cosd(ca));
```

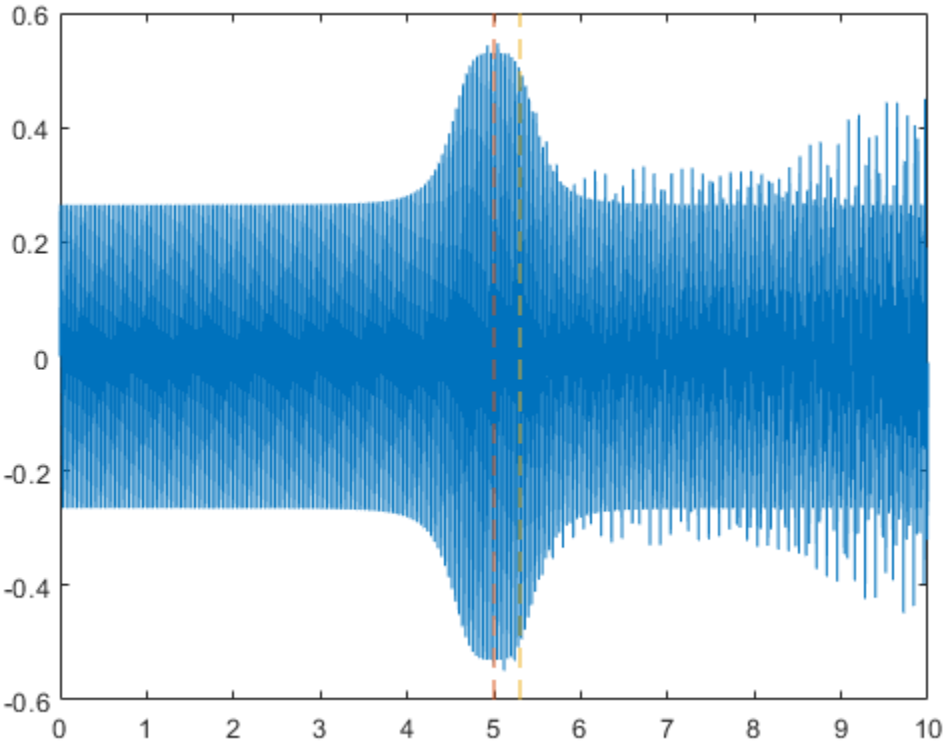
Use the `pulstran` function to model the impacts as a periodic train of 5-millisecond sinusoids. Each 3 kHz sinusoid is windowed by a flat top window. Use a power law to introduce progressive wear in the bearing vibration signal.

```
fImpact = 3000;
tImpact = 0:1/fs:5e-3-1/fs;
```

```
wImpact = flattopwin(length(tImpact))'/10;
xImpact = sin(2*pi*fImpact*tImpact).*wImpact;

tx = 0:1/bpfo:t(end);
tx = [tx; 1.3.^tx-2];

nWear = 49000;
nSamples = 100000;
yImpact = pulstran(t,tx',xImpact,fs)/5;
yImpact = [zeros(1,nWear) yImpact(1,(nWear+1):nSamples)];
```

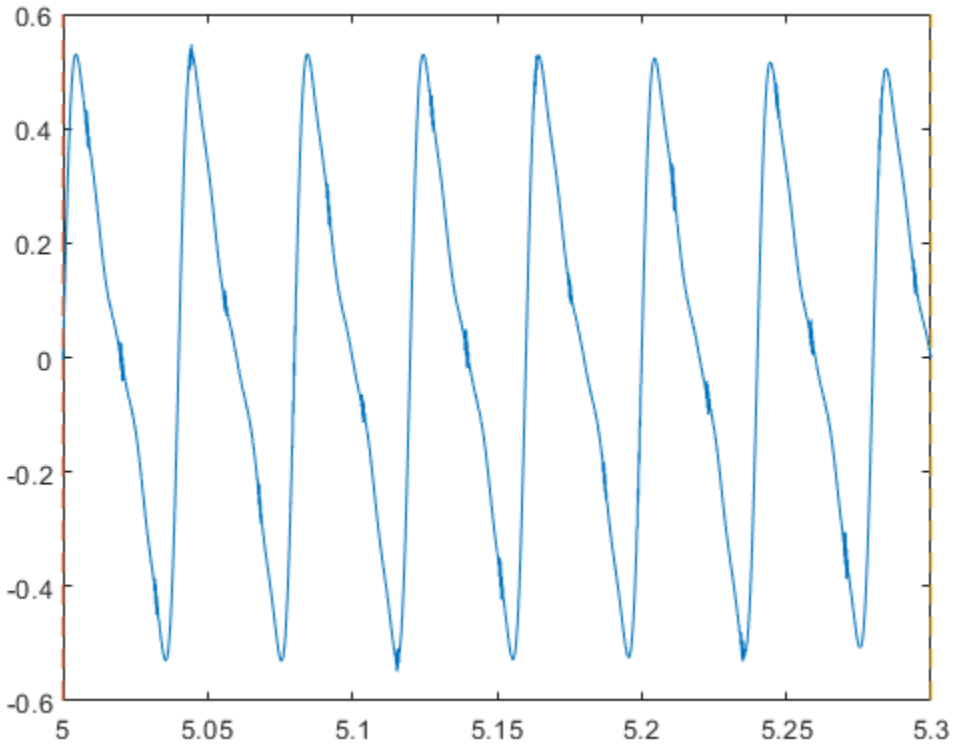
Generate the BPFO vibration signal by adding the impacts to the healthy signal. Plot the signal and select a 0.3-second interval starting at 5.0 seconds.

```
yBPFO = yImpact + yHealthy;

xLimLeft = 5.0;
xLimRight = 5.3;
yMin = -0.6;
yMax = 0.6;

plot(t,yBPFO)

hold on
[limLeft,limRight] = meshgrid([xLimLeft xLimRight],[yMin yMax]);
plot(limLeft,limRight,'--')
hold off
```



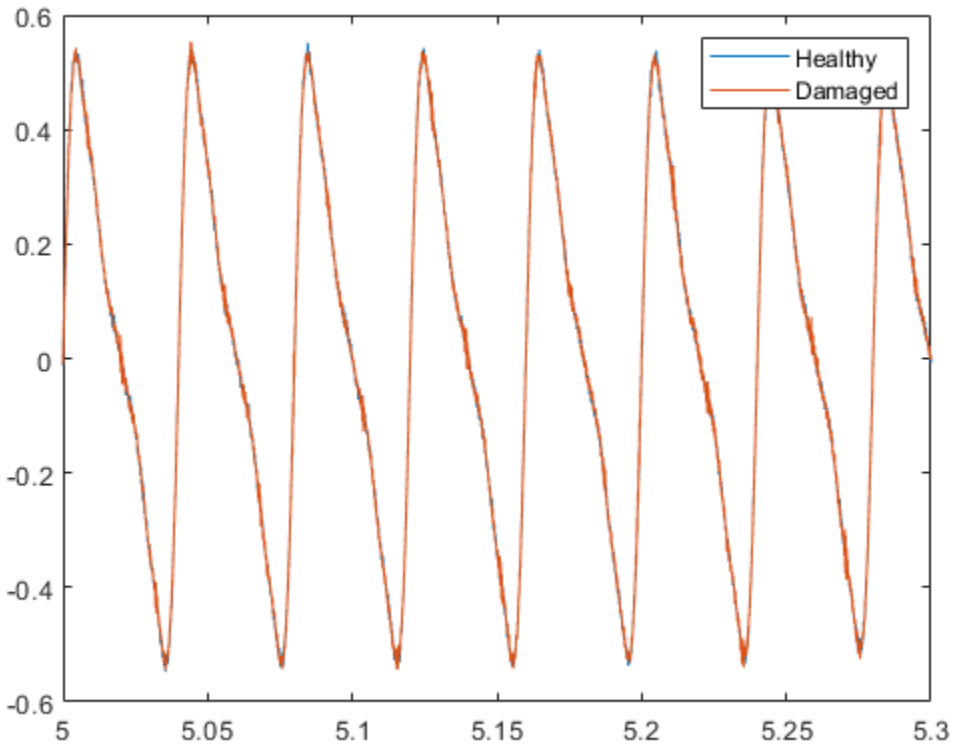
Zoom in on the selected interval to visualize the effect of the impacts.

```
xlim([xLimLeft xLimRight])
```



Add white Gaussian noise to the signals. Specify a noise variance of $1/150^2$.

```
rn = 150;
yGood = yHealthy + randn(size(yHealthy))/rn;
yBad = yBPFO + randn(size(yHealthy))/rn;

plot(t,yGood,t,yBad)
xlim([xLimLeft xLimRight])
legend('Healthy','Damaged')
```



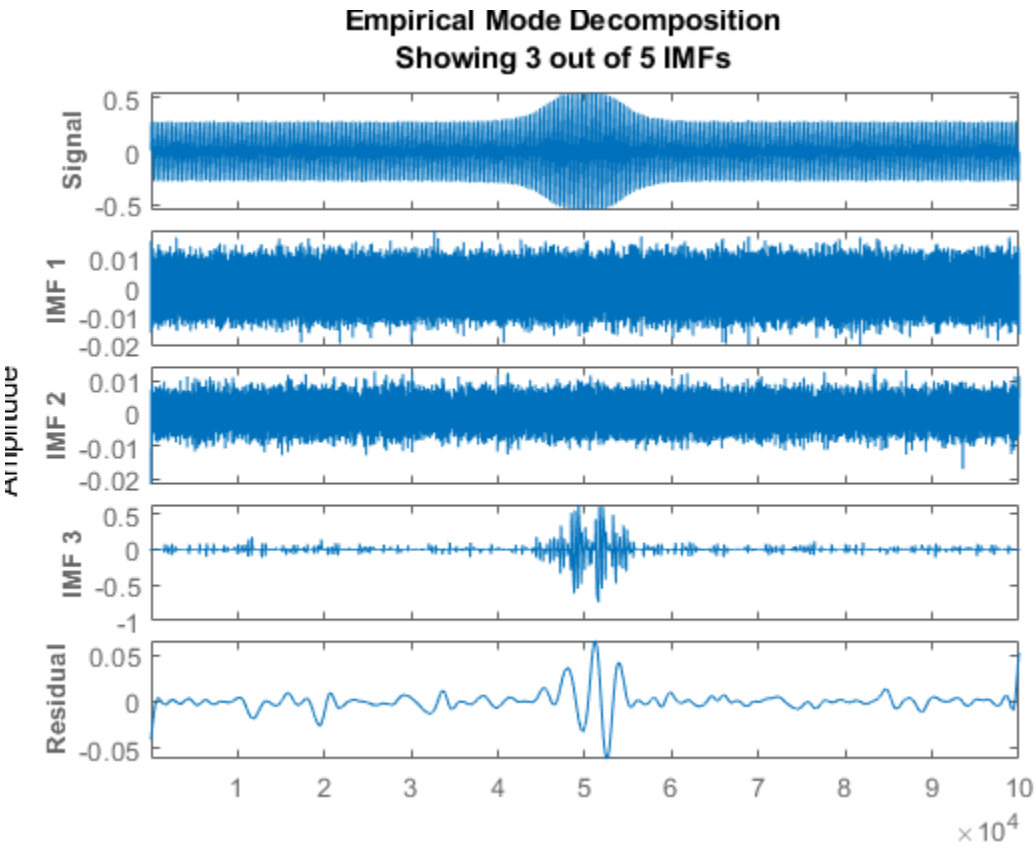
Use `emd` to perform an empirical mode decomposition of the healthy bearing signal. Compute the first five intrinsic mode functions (IMFs). Use the `'Display'` name-value pair to show a table with the number of sifting iterations, the relative tolerance, and the sifting stop criterion for each IMF.

```
imfGood = emd(yGood,'MaxNumIMF',5,'Display',1);
```

```
Current IMF  |  #Sift Iter  |  Relative Tol  |  Stop Criterion Hit
      1      |        3     |     0.017132   |  SiftMaxRelativeTolerance
      2      |        3     |      0.12694   |  SiftMaxRelativeTolerance
      3      |        6     |      0.14582   |  SiftMaxRelativeTolerance
      4      |        1     |     0.011082   |  SiftMaxRelativeTolerance
      5      |        2     |      0.03463   |  SiftMaxRelativeTolerance
Decomposition stopped because maximum number of intrinsic mode functions was extracted.
```

Use `emd` without output arguments to visualize the first three modes and the residual.

```
emd(yGood,'MaxNumIMF',5)
```

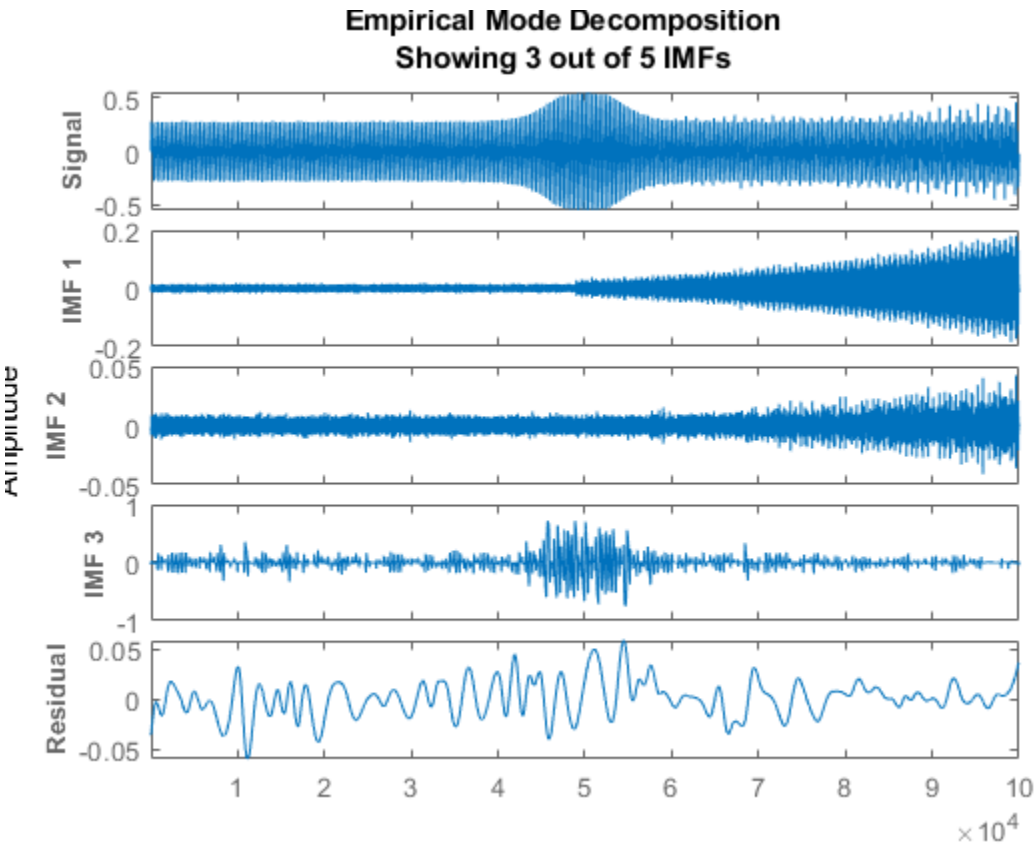


Compute and visualize the IMFs of the defective bearing signal. The first empirical mode reveals the high-frequency impacts. This high-frequency mode increases in energy as the wear progresses. The third mode shows the resonance in the vibration signal.

```
imfBad = emd(yBad,'MaxNumIMF',5,'Display',1);
```

```
Current IMF  |  #Sift Iter  |  Relative Tol  |  Stop Criterion Hit
      1      |        2     |     0.041274   |  SiftMaxRelativeTolerance
      2      |        3     |      0.16695   |  SiftMaxRelativeTolerance
      3      |        3     |      0.18428   |  SiftMaxRelativeTolerance
      4      |        1     |     0.037177   |  SiftMaxRelativeTolerance
      5      |        2     |     0.095861   |  SiftMaxRelativeTolerance
Decomposition stopped because maximum number of intrinsic mode functions was extracted.
```

```
emd(yBad,'MaxNumIMF',5)
```

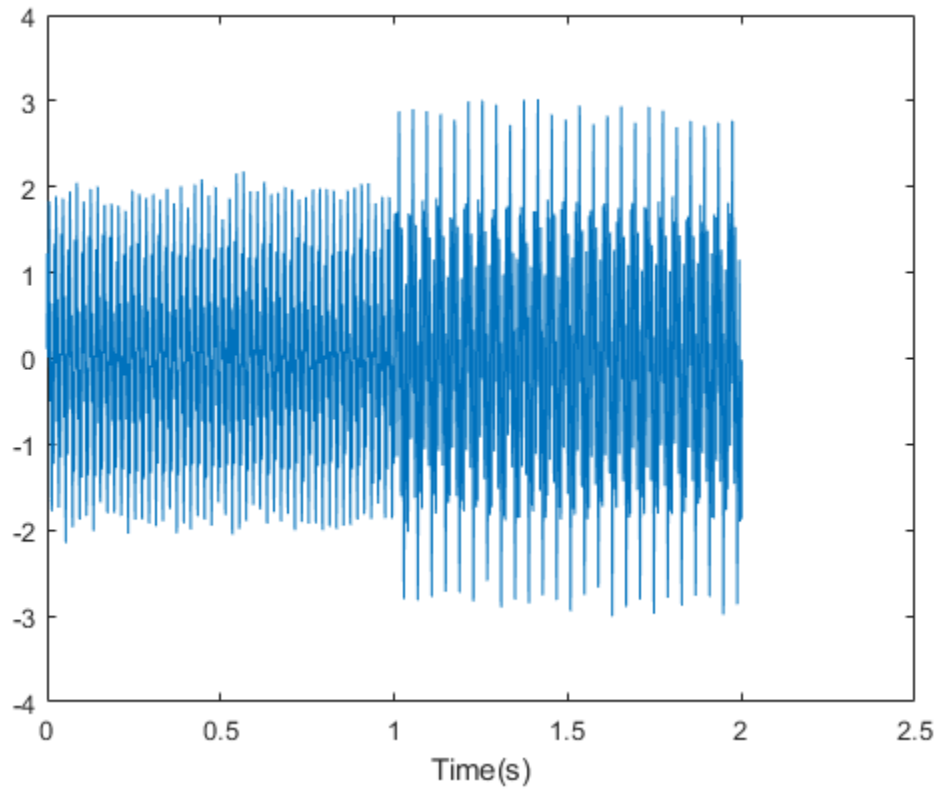


The next step in the analysis is to compute the Hilbert spectrum of the extracted IMFs. For more details, see the “Compute Hilbert Spectrum of Vibration Signal” on page 1-1003 example.

Visualize Residual and Intrinsic Mode Functions of Signal

Load and visualize a nonstationary continuous signal composed of sinusoidal waves with a distinct change in frequency. The vibration of a jackhammer and the sound of fireworks are examples of nonstationary continuous signals. The signal is sampled at a rate `fs`.

```
load('sinusoidalSignalExampleData.mat','X','fs')
t = (0:length(X)-1)/fs;
plot(t,X)
xlabel('Time(s)')
```

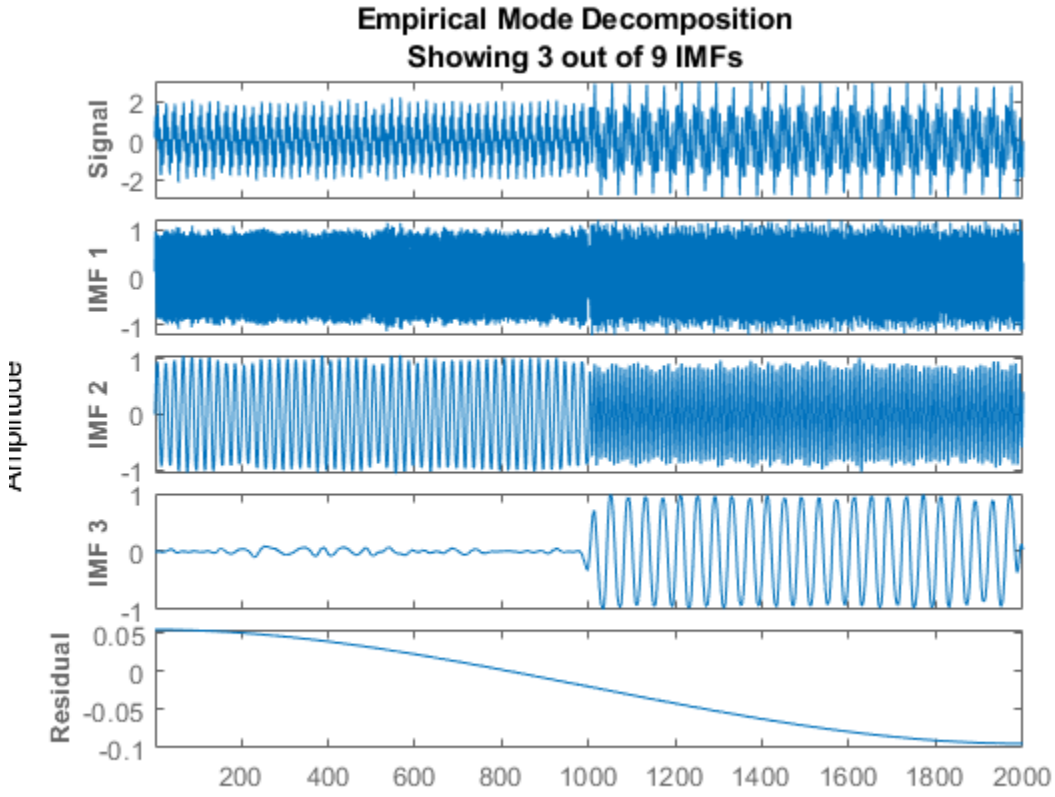


The mixed signal contains sinusoidal waves with different amplitude and frequency values.

Perform empirical mode decomposition to plot the intrinsic mode functions and residual of the signal. Since the signal is not smooth, specify '`pchip`' as the interpolation method.

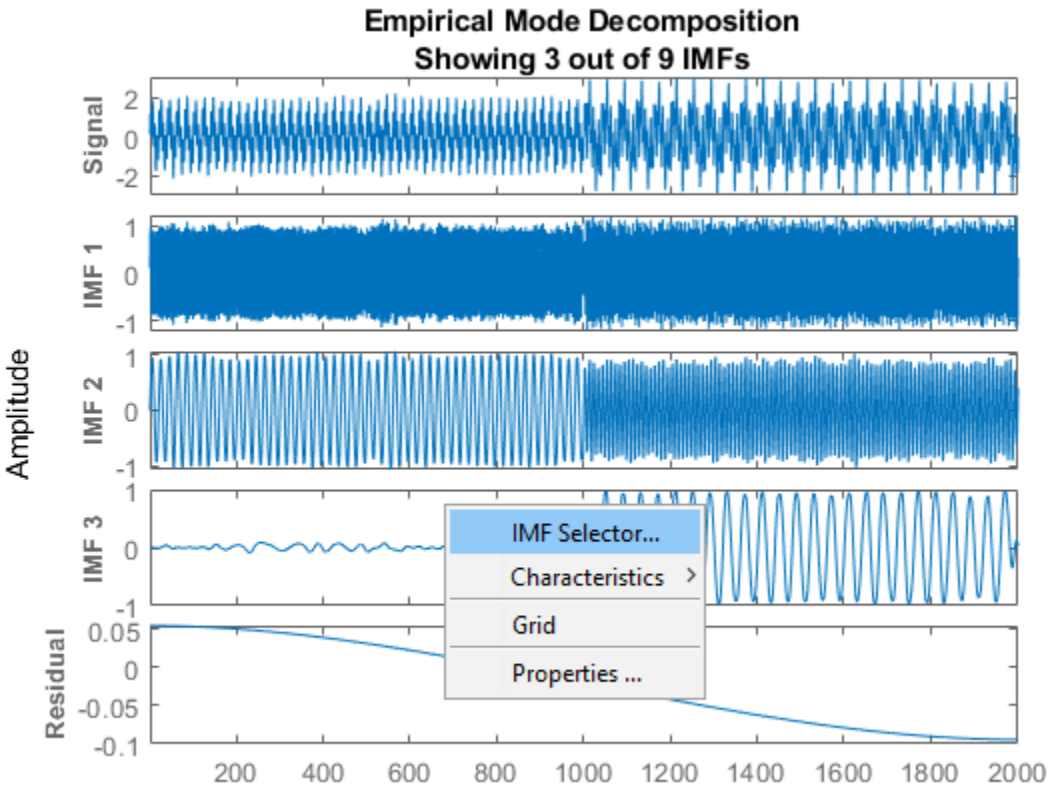
```
emd(X,'Interpolation','pchip','Display',1)
```

```
Current IMF  |  #Sift Iter  |  Relative Tol  |  Stop Criterion Hit
      1      |        2     |     0.026352   |  SiftMaxRelativeTolerance
      2      |        2     |    0.0039573   |  SiftMaxRelativeTolerance
      3      |        1     |     0.024838   |  SiftMaxRelativeTolerance
      4      |        2     |      0.05929   |  SiftMaxRelativeTolerance
      5      |        2     |      0.11317   |  SiftMaxRelativeTolerance
      6      |        2     |      0.12599   |  SiftMaxRelativeTolerance
      7      |        2     |      0.13802   |  SiftMaxRelativeTolerance
      8      |        3     |      0.15937   |  SiftMaxRelativeTolerance
      9      |        2     |      0.15923   |  SiftMaxRelativeTolerance
Decomposition stopped because the number of extrema in the residual signal is less than the 'MaxI
```

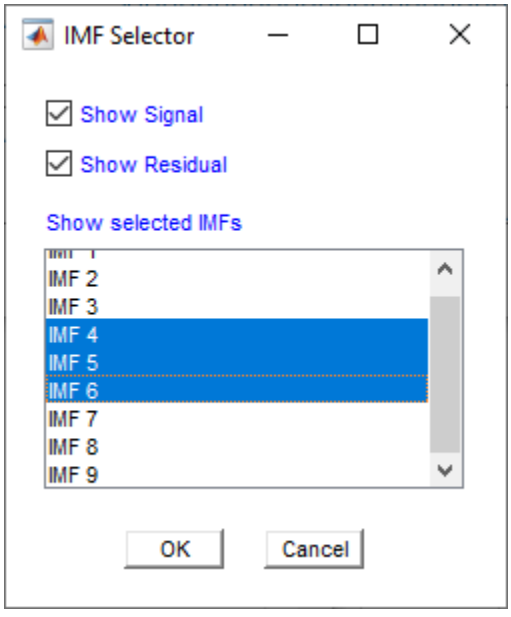


`emd` generates an interactive plot with the original signal, the first 3 IMFs, and the residual. The table generated in the command window indicates the number of sift iterations, the relative tolerance, and the sift stop criterion for each generated IMF. You can hide the table by removing the `'Display'` name-value pair or specifying it as `0`.

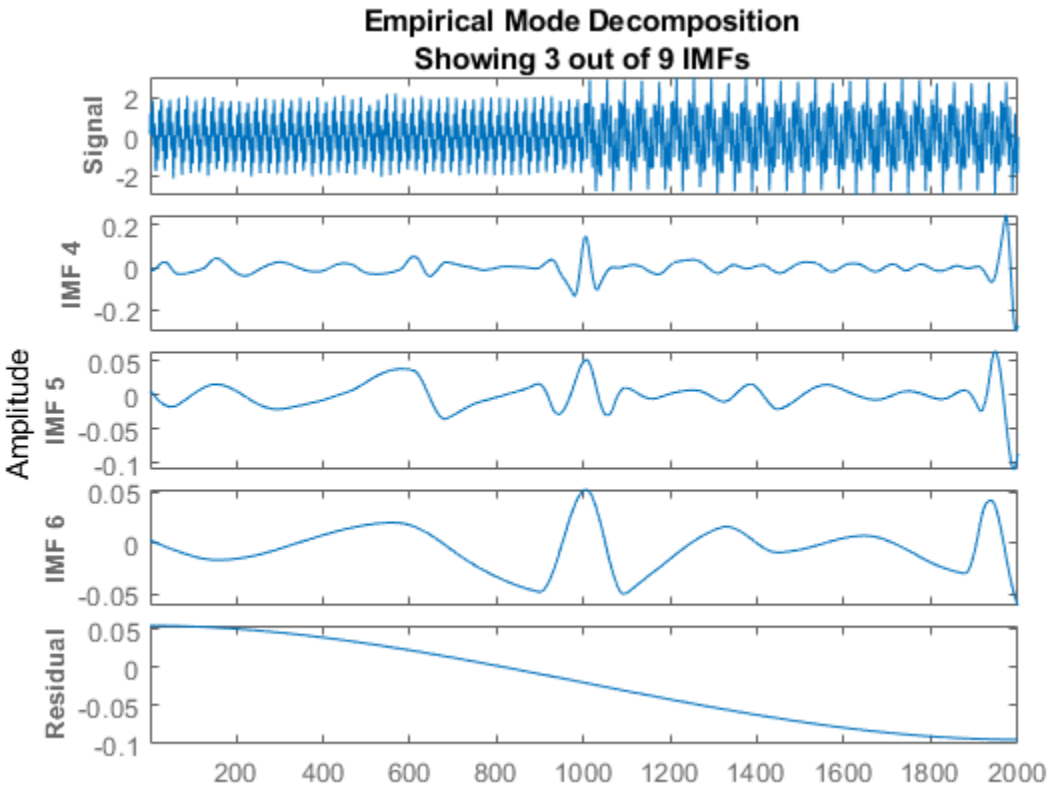
Right-click on the white space in the plot to open the **IMF selector** window. Use **IMF selector** to selectively view the generated IMFs, the original signal, and the residual.



Select the IMFs to be displayed from the list. Choose whether to display the original signal and residual on the plot.



The selected IMFs are now displayed on the plot.



Use the plot to visualize individual components decomposed from the original signal along with the residual. Note that the residual is computed for the total number of IMFs, and does not change based on the IMFs selected in the **IMF selector** window.

Input Arguments

x — Time-domain signal

vector | timetable

Time-domain signal, specified as a real-valued vector, or a single-variable timetable with a single column. If `x` is a timetable, `x` must contain increasing, finite row times.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MaxNumIMF',5`

SiftRelativeTolerance — Cauchy-type convergence criterion

`0.2` (default) | positive scalar

Cauchy-type convergence criterion, specified as the comma-separated pair consisting of `'SiftRelativeTolerance'` and a positive scalar. `SiftRelativeTolerance` is one of the sifting stop criteria, that is, sifting stops when the current relative tolerance is less than `SiftRelativeTolerance`. For more information, see “Sift Relative Tolerance” on page 1-572.

`SiftMaxIterations` — Maximum number of sifting iterations
`100` (default) | positive scalar integer

Maximum number of sifting iterations, specified as the comma-separated pair consisting of `'SiftMaxIterations'` and a positive scalar integer. `SiftMaxIterations` is one of the sifting stop criteria, that is, sifting stops when the current number of iterations is larger than `SiftMaxIterations`.

`SiftMaxIterations` can be specified using only positive whole numbers.

`MaxNumIMF` — Maximum number of IMFs extracted
`10` (default) | positive scalar integer

Maximum number of IMFs extracted, specified as the comma-separated pair consisting of `'MaxNumIMF'` and a positive scalar integer. `MaxNumIMF` is one of the decomposition stop criteria, that is, decomposition stops when number of IMFs generated is equal to `MaxNumIMF`.

`MaxNumIMF` can be specified using only positive whole numbers.

`MaxNumExtrema` — Maximum number of extrema in the residual signal
`1` (default) | positive scalar integer

Maximum number of extrema in the residual signal, specified as the comma-separated pair consisting of `'MaxNumExtrema'` and a positive scalar integer. `MaxNumExtrema` is one of the decomposition stop criteria, that is, decomposition stops when number of extrema is less than `MaxNumExtrema`.

`MaxNumExtrema` can be specified using only positive whole numbers.

`MaxEnergyRatio` — Signal to residual energy ratio
`20` (default) | scalar

Signal to residual energy ratio, specified as the comma-separated pair consisting of `'MaxEnergyRatio'` and a scalar. `MaxEnergyRatio` is the ratio of the energy of the signal at the beginning of sifting and the average envelope energy. `MaxEnergyRatio` is one of the decomposition stop criteria, that is, decomposition stops when current energy ratio is larger than `MaxEnergyRatio`. For more information, see “Energy Ratio” on page 1-573.

`Interpolation` — Interpolation method for envelope construction
`'spline'` (default) | `'pchip'`

Interpolation method for envelope construction, specified as the comma-separated pair consisting of `'Interpolation'` and either `'spline'` or `'pchip'`.

Specify `Interpolation` as:

- `'spline'`, if `x` is a smooth signal
- `'pchip'`, if `x` is a nonsmooth signal

`'spline'` interpolation method uses cubic splines, while `'pchip'` uses piecewise-cubic Hermite interpolating polynomials.

`Display` — Toggle information display in the command window
0 (default) | 1

Toggle information display in the command window, specified as the comma-separated pair consisting of `'Display'` and either 0 or 1. The table generated in the command window indicates the number of sift iterations, the relative tolerance, and the sift stop criterion for each generated IMF. Specify `Display` as 1 to show the table or 0 to hide the table.

Output Arguments

`imf` — Intrinsic mode function
matrix | timetable

Intrinsic mode function (IMF), returned as a matrix or timetable. Each IMF is an amplitude and frequency modulated signal with positive and slowly varying envelopes. To perform spectral analysis of a signal, you can apply the Hilbert-Huang transform to its IMFs. See `hht` and “Intrinsic Mode Functions” on page 1-572.

`imf` is returned as:

- A matrix whose each column is an `imf`, when `x` is a vector
- A timetable, when `x` is a single data column timetable

`residual` — Residual of the signal
column vector | single data column timetable

Residual of the signal, returned as a column vector or a single data column timetable. `residual` represents the portion of the original signal `x` not decomposed by `emd`.

`residual` is returned as:

- A column vector, when `x` is a vector.
- A single data column timetable, when `x` is a single data column timetable.

`info` — Additional information for diagnostics
structure

Additional information for diagnostics, returned as a structure with the following fields:

- `NumIMF` — Number of IMFs extracted

  `NumIMF` is a vector from 1 to *N*, where *N* is the number of IMFs. If no IMFs are extracted, `NumIMF` is empty.
- `NumExtrema` — Number of extrema in each IMF

  `NumExtrema` is a vector equal in length to the number of IMFs. The *k*th element of `NumExtrema` is the number of extrema found in the *k*th IMF. If no IMFs are extracted, `NumExtrema` is empty.
- `NumZerocrossing` — Number of zero crossings in each IMF

  Number of zero crossings in each IMF. `NumZerocrossing` is a vector equal in length to the number of IMFs. The *k*th element of `NumZerocrossing` is the number of zero crossings in the *k*th IMF. If no IMFs are extracted, `NumZerocrossing` is empty.

- `NumSifting` — Number of sifting iterations used to extract each IMF

  `NumSifting` is a vector equal in length to the number of IMFs. The *k*th element of `NumSifting` is the number of sifting iterations used in the extraction of the *k*th IMF. If no IMFs are extracted, `NumSifting` is empty.

- `MeanEnvelopeEnergy` — Energy of the mean of the upper and lower envelopes obtained for each IMF

  If `UE` is the upper envelope and `LE` is the lower envelope, `MeanEnvelopeEnergy` is `mean(((LE+UL)/2).^2)`. `MeanEnvelopeEnergy` is a vector equal in length to the number of IMFs. The *k*th element of `MeanEnvelopeEnergy` is the mean envelope energy for the *k*th IMF. If no IMFs are extracted, `MeanEnvelopeEnergy` is empty.

- `RelativeTolerance` — Final relative tolerance of the residual for each IMF

  The relative tolerance is defined as the ratio of the squared 2-norm of the difference between the residual from the previous sifting step and the residual from the current sifting step to the squared 2-norm of the residual from the *i*th sifting step. The sifting process stops when `RelativeTolerance` is less than `SiftRelativeTolerance`. For additional information, see “Sift Relative Tolerance” on page 1-572. `RelativeTolerance` is a vector equal in length to the number of IMFs. The *k*th element of `RelativeTolerance` is the final relative tolerance obtained for the *k*th IMF. If no IMFs are extracted, `RelativeTolerance` is empty.

More About

Empirical Mode Decomposition

The empirical mode decomposition (EMD) algorithm decomposes a signal $x(t)$ into intrinsic mode functions (IMFs) and a residual in an iterative process. The core component of the algorithm involves *sifting* a function $x(t)$ to obtain a new function $Y(t)$:

- First find the local minima and maxima of $x(t)$.
- Then use the local extrema to construct lower and upper envelopes $s_-(t)$ and $s_+(t)$, respectively, of $x(t)$. Form the mean of the envelopes, $m(t)$.
- Subtract the mean from $x(t)$ to obtain the residual: $Y(t) = x(t) - m(t)$.

An overview of the decomposition is as follows:

1. To begin, let $r_0(t) = x(t)$, where $x(t)$ is the initial signal, and let $i = 0$.
2. Before sifting, check $r_i(t)$:
   a. Find the total number (TN) of local extrema of $r_i(t)$.
   b. Find the energy ratio (ER) of $r_i(t)$ (see “Energy Ratio” on page 1-573).
3. If (ER > `MaxEnergyRatio`) or (TN < `MaxNumExtrema`) or (number of IMFs > `MaxNumIMF`) then stop the decomposition.
4. Let $r_{i,\mathrm{Prev}}(t) = r_i(t)$.
5. Sift $r_{i,\mathrm{Prev}}(t)$ to obtain $r_{i,\mathrm{Cur}}(t)$.
6. Check $r_{i,\mathrm{Cur}}(t)$
   a. Find the relative tolerance (RT) of $r_{i,\mathrm{Cur}}(t)$ (see “Sift Relative Tolerance” on page 1-572).

b Get current sift iteration number (IN).

7 If (RT < `SiftRelativeTolerance`) or (IN > `SiftMaxIterations`) then stop sifting. An IMF has been found: $\mathrm{IMF}_i(t) = r_{i,\mathrm{Cur}}(t)$. Otherwise, let $r_{i,\mathrm{Prev}}(t) = r_{i,\mathrm{Cur}}(t)$ and go to Step 5.

8 Let $r_{i+1}(t) = r_i(t) - r_{i,\mathrm{Cur}}(t)$.

9 Let $i = i + 1$. Return to Step 2.

For additional information, see [1] and [3].

Intrinsic Mode Functions

The EMD algorithm decomposes, via an iterative sifting process, a signal $x(t)$ into IMFs $imf_i(t)$ and a residual $r_N(t)$:

$$X(t) = \sum_{i=1}^{N} \mathrm{IMF}_i(t) + r_N(t)$$

When first introduced by Huang et al. [1], an IMF was defined to be a function with two characteristics:

- The number of local extrema — the total number of local minima and local maxima — and the number of zero crossings differ by at most one.
- The mean value of the upper and lower envelopes constructed from the local extrema is zero.

However, as noted in [4], sifting until a strict IMF is obtained can result in IMFs that have no physical significance. Specifically, sifting until the number of zero crossings and local extrema differ by at most one can result in pure-tone like IMFs, in other words, functions very similar to what would be obtained by projection on the Fourier basis. This situation is precisely what EMD strives to avoid, preferring AM-FM modulated components for their physical significance.

Reference [4] proposes options to obtain physically meaningful results. The `emd` function relaxes the original IMF definition by using “Sift Relative Tolerance” on page 1-572, a Cauchy-type stop criterion. The `emd` function iterates to extract natural AM-FM modes. The IMFs generated may fail to satisfy the local extrema-zero crossings criteria. See “Zero Crossings and Extrema in Intrinsic Mode Function of Sinusoid” on page 1-555.

Sift Relative Tolerance

Sift Relative Tolerance is a Cauchy-type stop criterion proposed in [4]. Sifting stops when current relative tolerance is less than `SiftRelativeTolerance`. The current relative tolerance is defined as

$$\text{Relative Tolerance} \triangleq \frac{\left\| r_{\mathrm{prev}}(t) - r_{\mathrm{cur}}(t) \right\|_2^2}{\left\| r_{\mathrm{prev}}(t) \right\|_2^2}.$$

Because the Cauchy criterion does not directly count the number of zero crossings and local extrema, it is possible that the IMFs returned by the decomposition do not satisfy the strict definition of an intrinsic mode function. In those cases, you can try reducing the value of the `SiftRelativeTolerance` from its default value. See [4] for a detailed discussion of stopping criteria. The reference also discusses the advantages and disadvantages of insisting on strictly defined IMFs in empirical mode decomposition.

Energy Ratio

Energy ratio is the ratio of the energy of the signal at the beginning of sifting and the average envelope energy [2]. Decomposition stops when current energy ratio is larger than `MaxEnergyRatio`. For the *i*th IMF, the energy ratio is defined as

$$\text{Energy Ratio} \triangleq 10\log_{10}\left(\frac{\|X(t)\|_2}{\|r_i(t)\|_2}\right).$$

References

[1] Huang, Norden E., Zheng Shen, Steven R. Long, Manli C. Wu, Hsing H. Shih, Quanan Zheng, Nai-Chyuan Yen, Chi Chao Tung, and Henry H. Liu. “The Empirical Mode Decomposition and the Hilbert Spectrum for Nonlinear and Non-Stationary Time Series Analysis.” *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 454, no. 1971 (March 8, 1998): 903–95. https://doi.org/10.1098/rspa.1998.0193.

[2] Rato, R.T., M.D. Ortigueira, and A.G. Batista. “On the HHT, Its Problems, and Some Solutions.” *Mechanical Systems and Signal Processing* 22, no. 6 (August 2008): 1374–94. https://doi.org/10.1016/j.ymssp.2007.11.028.

[3] Rilling, Gabriel, Patrick Flandrin, and Paulo Gonçalves. "On Empirical Mode Decomposition and Its Algorithms." *IEEE-EURASIP Workshop on Nonlinear Signal and Image Processing* 2003. NSIP-03. Grado, Italy. 8–11.

[4] Wang, Gang, Xian-Yao Chen, Fang-Li Qiao, Zhaohua Wu, and Norden E. Huang. “On Intrinsic Mode Function.” *Advances in Adaptive Data Analysis* 02, no. 03 (July 2010): 277–93. https://doi.org/10.1142/S1793536910000549.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Timetables are not supported for code generation.
- If supplied, the interpolation method specified using the `'Interpolation'` name-value pair must be a compile-time constant.

See Also

Apps

Signal Multiresolution Analyzer

Functions

`hht` | `vmd`

Topics

“Time-Frequency Gallery”

Introduced in R2018a

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 1000 samples in length.

```
bw = enbw(hamming(1000))
```

```
bw = 1.3638
```

Equivalent Noise Bandwidth of Flat Top Window

Determine the equivalent noise bandwidth in Hz of a flat top window 10000 samples in length. The sample rate is 44.1 kHz.

```
bw = enbw(flattopwin(10000),44.1e3)
```

```
bw = 16.6285
```

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));
```

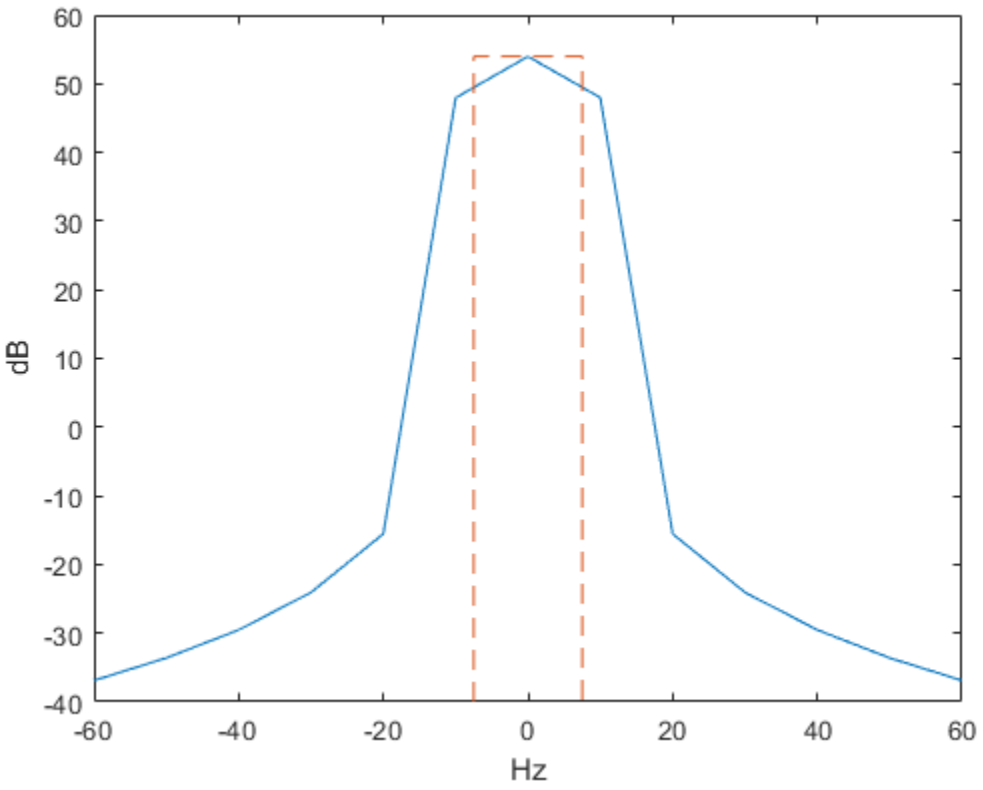
Obtain the equivalent (rectangular) noise bandwidth of the Von Hann window.

```
bw = enbw(hann(1000),Fs)
```

```
bw = 15.0150
```

Plot the squared-magnitude DFT of the window and use the equivalent noise bandwidth to overlay the equivalent rectangle. The two-sided bandwidth is split evenly across the spectrum.

```
freq = -(Fs/2):Fs/length(win):Fs/2-(Fs/length(win));
maxgain = 20*log10(abs(windft(length(win)/2+1)));

plot(freq,20*log10(abs(windft)))
hold on
plot(bw/2*[-1 -1 1 1],[-40 maxgain maxgain -40],'--')
hold off

xlabel('Hz')
ylabel('dB')
axis([-60 60 -40 60])
```



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 1-575 for an example that plots the equivalent rectangular bandwidth over the magnitude spectrum of a von Hann window.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`bandpower` | `sfdr`

Introduced in R2013a

envelope

Signal envelope

Syntax

```
[yupper,ylower] = envelope(x)

[yupper,ylower] = envelope(x,fl,'analytic')
[yupper,ylower] = envelope(x,wl,'rms')
[yupper,ylower] = envelope(x,np,'peak')

envelope( ___ )
```

Description

`[yupper,ylower] = envelope(x)` returns the upper and lower envelopes of the input sequence, `x`, as the magnitude of its analytic signal. The analytic signal of `x` is found using the discrete Fourier transform as implemented in `hilbert`. The function initially removes the mean of `x` and adds it back after computing the envelopes. If `x` is a matrix, then `envelope` operates independently over each column of `x`.

`[yupper,ylower] = envelope(x,fl,'analytic')` returns the envelopes of `x` determined using the magnitude of its analytic signal. The analytic signal is computed by filtering `x` with a Hilbert FIR filter of length `fl`. This syntax is used if you specify only two arguments.

`[yupper,ylower] = envelope(x,wl,'rms')` returns the upper and lower root-mean-square envelopes of `x`. The envelopes are determined using a sliding window of length `wl` samples.

`[yupper,ylower] = envelope(x,np,'peak')` returns the upper and lower peak envelopes of `x`. The envelopes are determined using spline interpolation over local maxima separated by at least `np` samples.

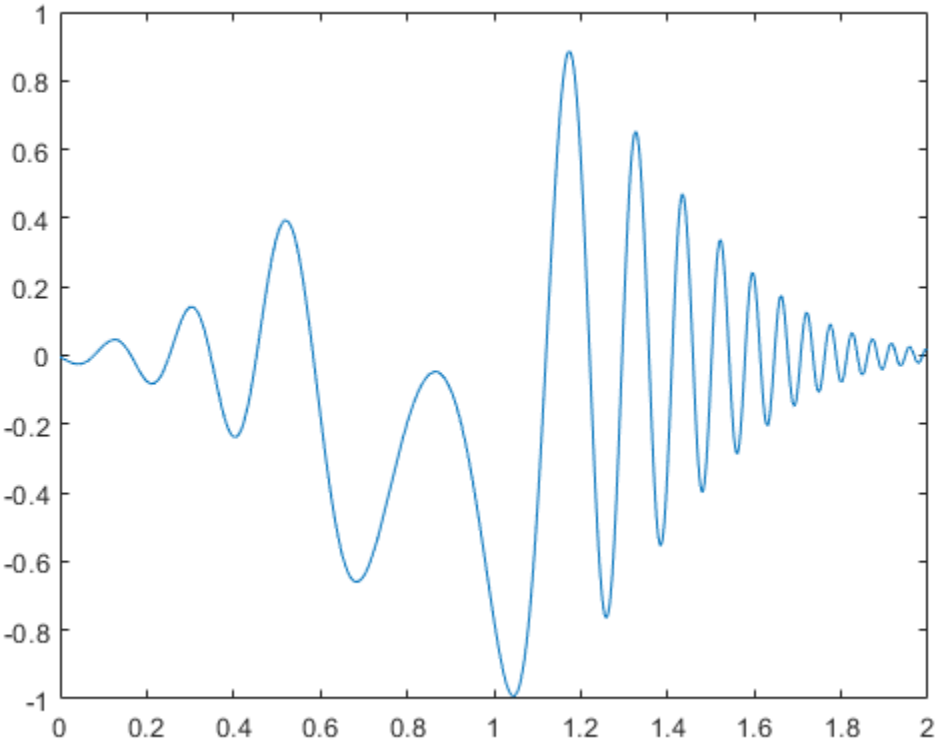
`envelope( ___ )` with no output arguments plots the signal and its upper and lower envelopes. This syntax accepts any of the input arguments from previous syntaxes.

Examples

Analytic Envelopes of Chirp

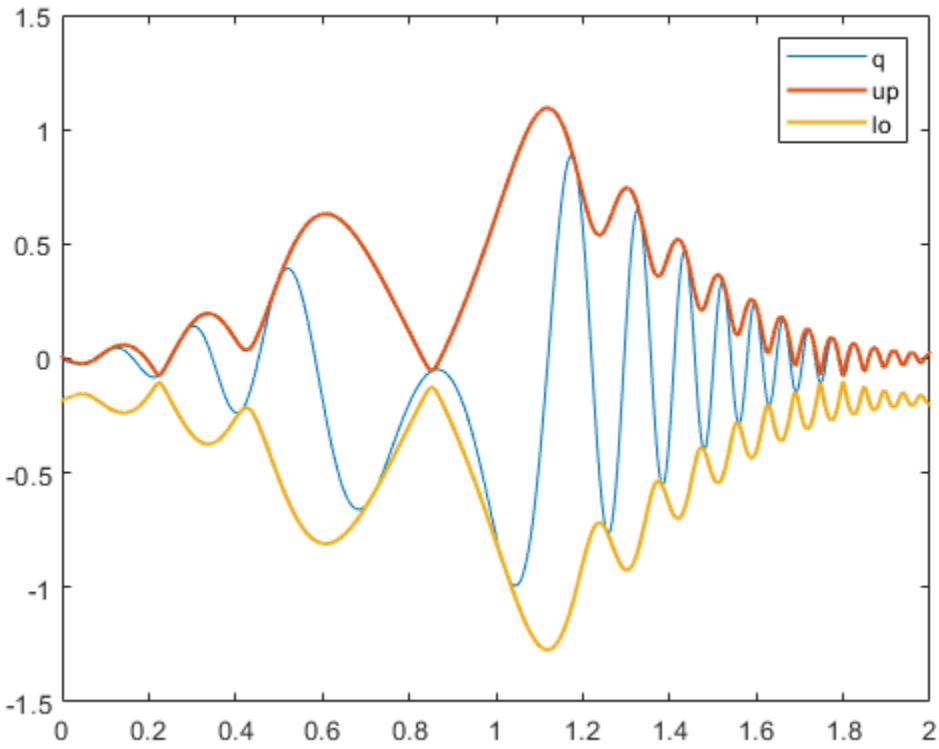
Generate a quadratic chirp modulated by a Gaussian. Specify a sample rate of 2 kHz and a signal duration of 2 seconds.

```
t = 0:1/2000:2-1/2000;
q = chirp(t-2,4,1/2,6,'quadratic',100,'convex').*exp(-4*(t-1).^2);
plot(t,q)
```



Compute the upper and lower envelopes of the chirp using the analytic signal.

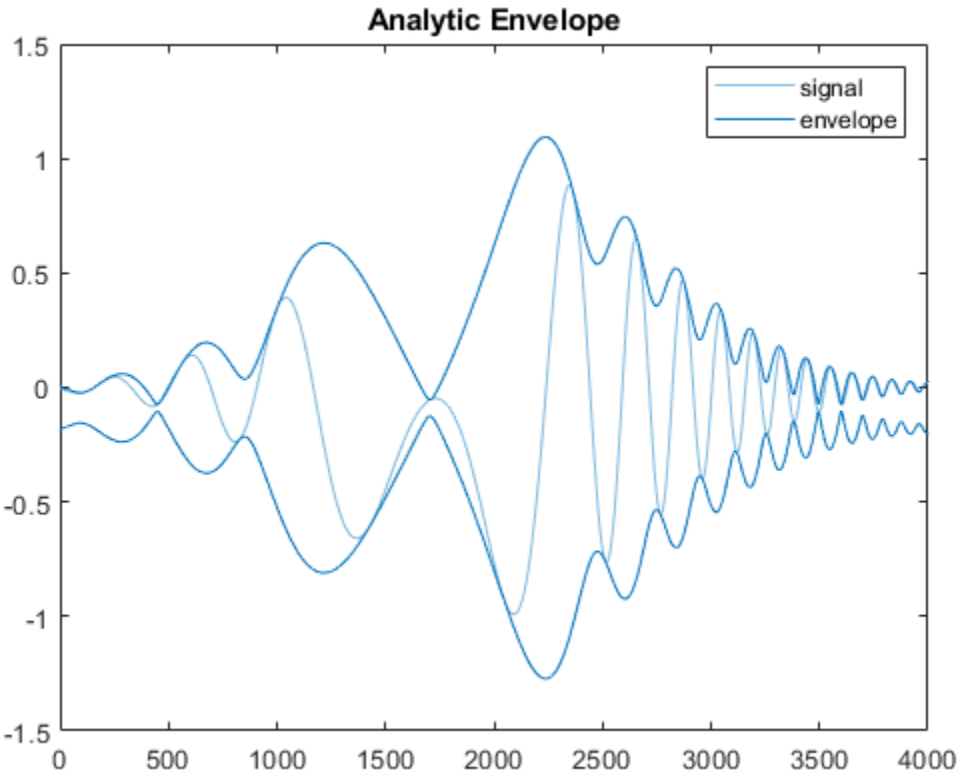
```
[up,lo] = envelope(q);
hold on
plot(t,up,t,lo,'linewidth',1.5)
legend('q','up','lo')
hold off
```



The signal is asymmetric due to the nonzero mean.

Use `envelope` without output arguments to plot the signal and envelopes as a function of sample number.

```
envelope(q)
```



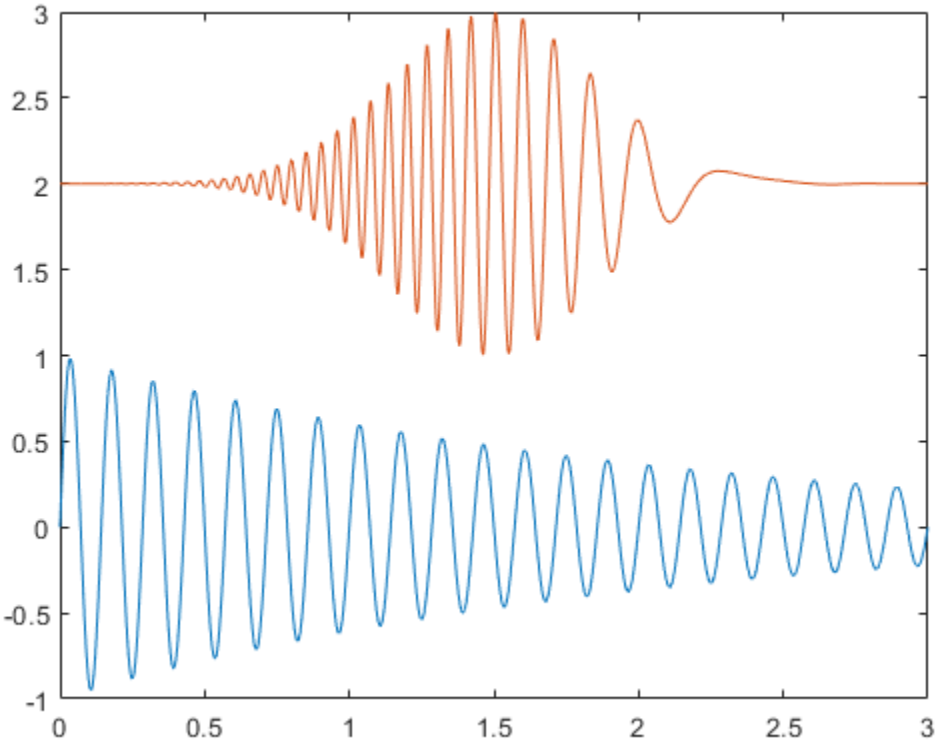
Analytic Envelopes of Multichannel Signal Using Filter

Create a two-channel signal sampled at 1 kHz for 3 seconds:

- One channel is an exponentially decaying sinusoid. Specify a frequency of 7 Hz and a time constant of 2 seconds.
- The other channel is a time-displaced Gaussian-modulated chirp with a DC value of 2. Specify an initial chirp frequency of 30 Hz that decays to 5 Hz after 2 seconds.

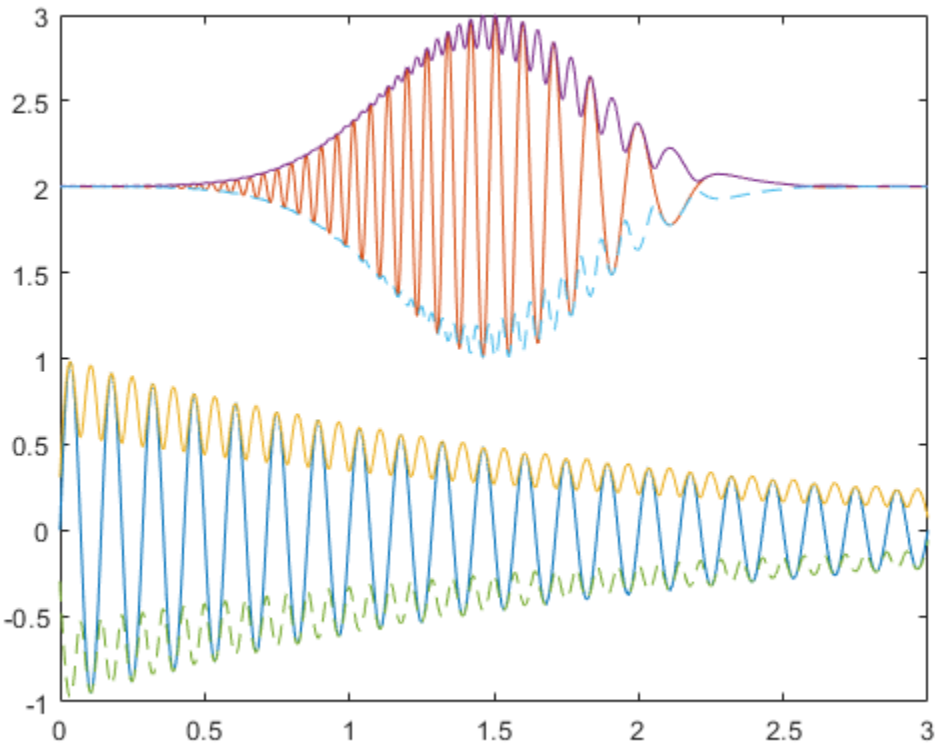
Plot the signal.

```
t = 0:1/1000:3;
q1 = sin(2*pi*7*t).*exp(-t/2);
q2 = chirp(t,30,2,5).*exp(-(2*t-3).^2)+2;
q = [q1;q2]';

plot(t,q)
```



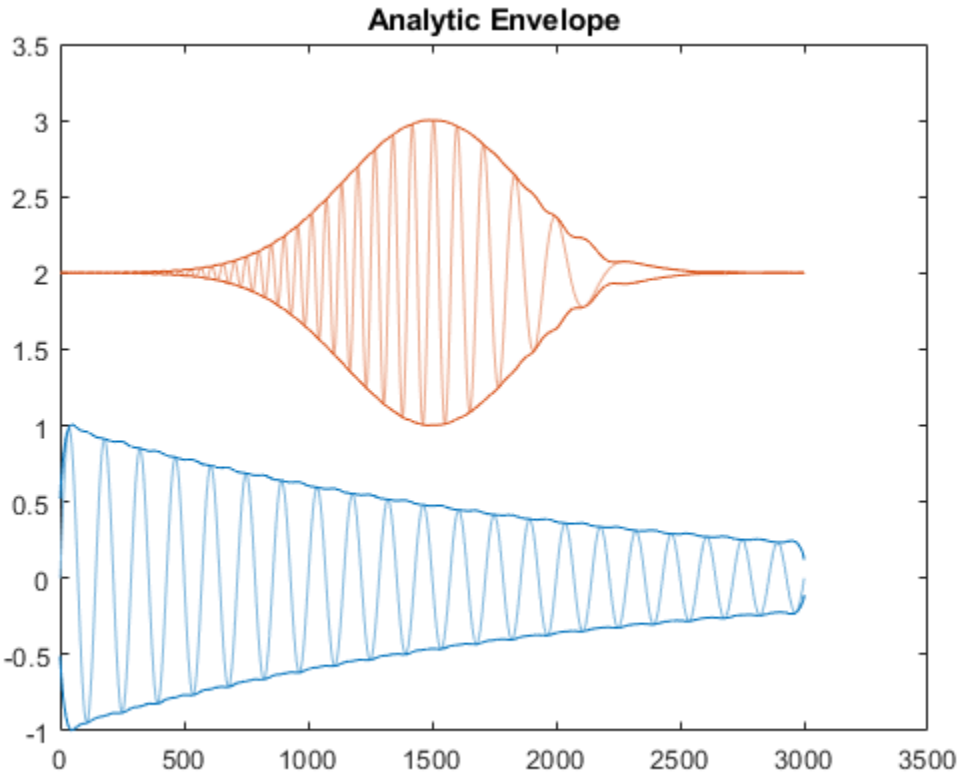
Compute the upper and lower envelopes of the signal. Use a Hilbert filter with a length of 100. Plot the channels and the envelopes. Use solid lines for the upper envelopes and dashed lines for the lower envelopes.

```
[up,lo] = envelope(q,100,'analytic');
hold on
plot(t,up,'-',t,lo,'--')
hold off
```



Call `envelope` without output arguments to produce a plot of the signal and its envelopes as a function of sample number. Increase the filter length to 300 to obtain a smoother shape. The `'analytic'` flag is the default when you specify two input arguments.

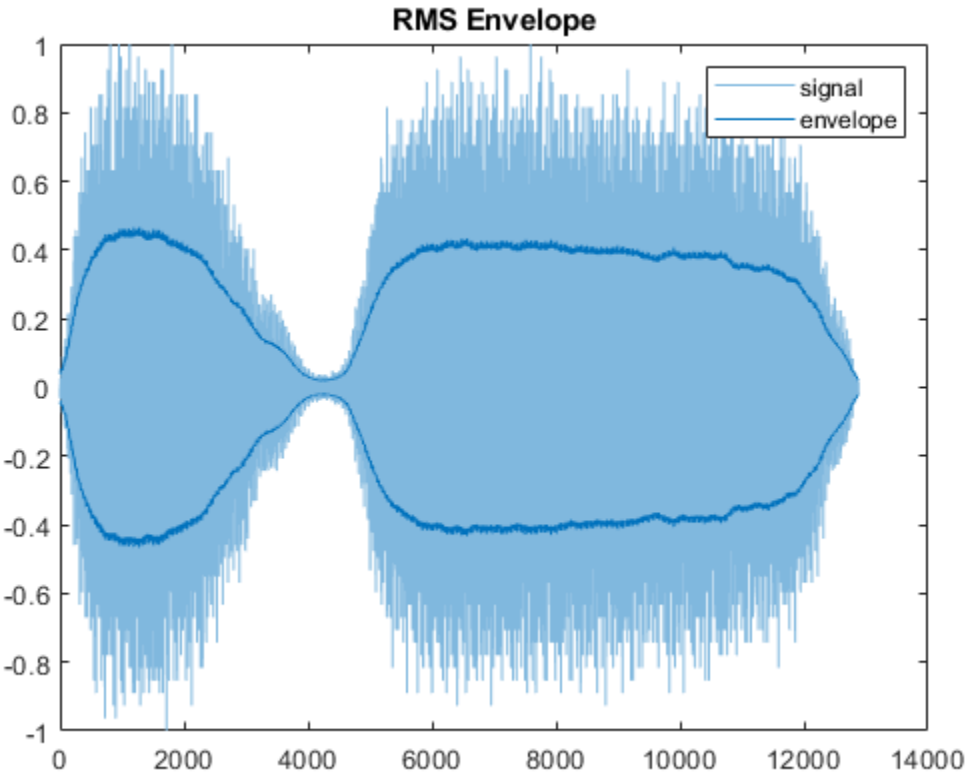
```
envelope(q,300)
```



Moving RMS Envelopes of Audio Recording

Compute and plot the moving RMS envelopes of a recording of a train whistle. Use a window with a length of 150 samples.

```
load('train')

envelope(y,150,'rms')
```

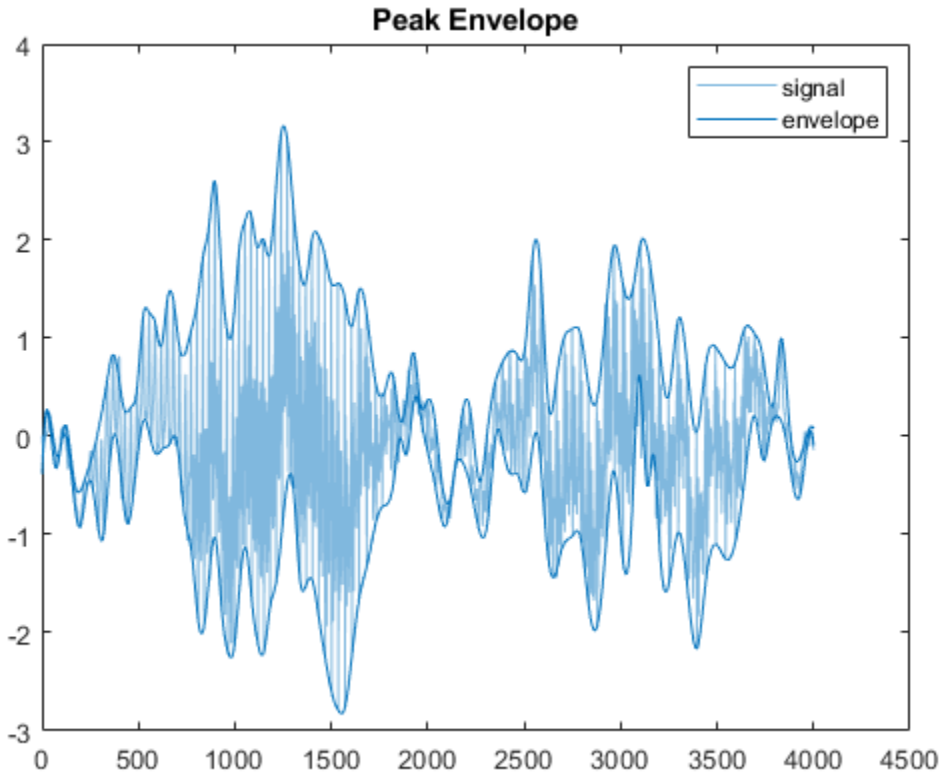


Peak Envelopes of Speech Signal

Plot the upper and lower peak envelopes of a speech signal smoothed over 30-sample intervals.

```
load('mtlb')
```

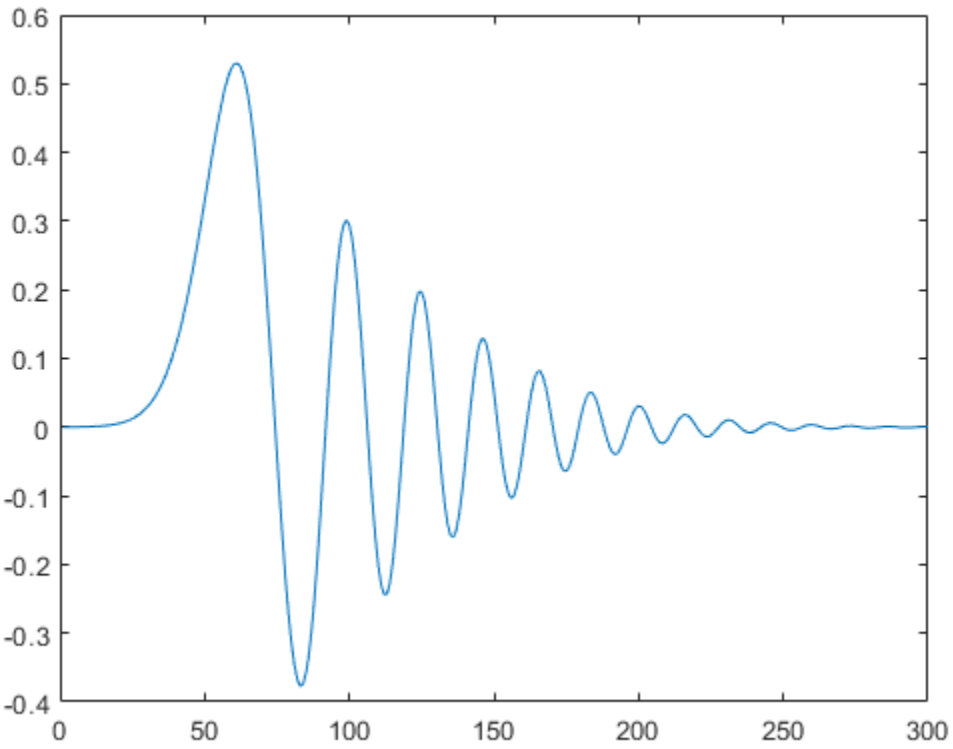
```
envelope(mtlb,30,'peak')
```



Envelope of Asymmetric Sequence

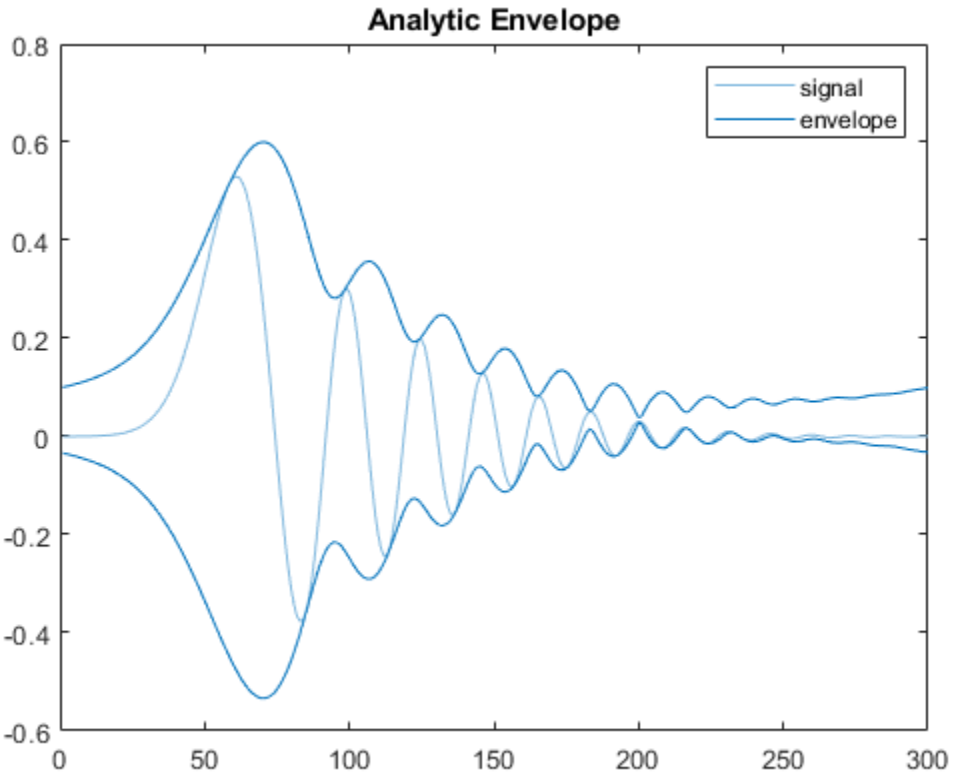
Create and plot a signal that resembles the initial detection of a light pulse propagating through a dispersive medium.

```
t = 0.5:-1/100:-2.49;
z = airy(t*10).*exp(-t.^2);

plot(z)
```



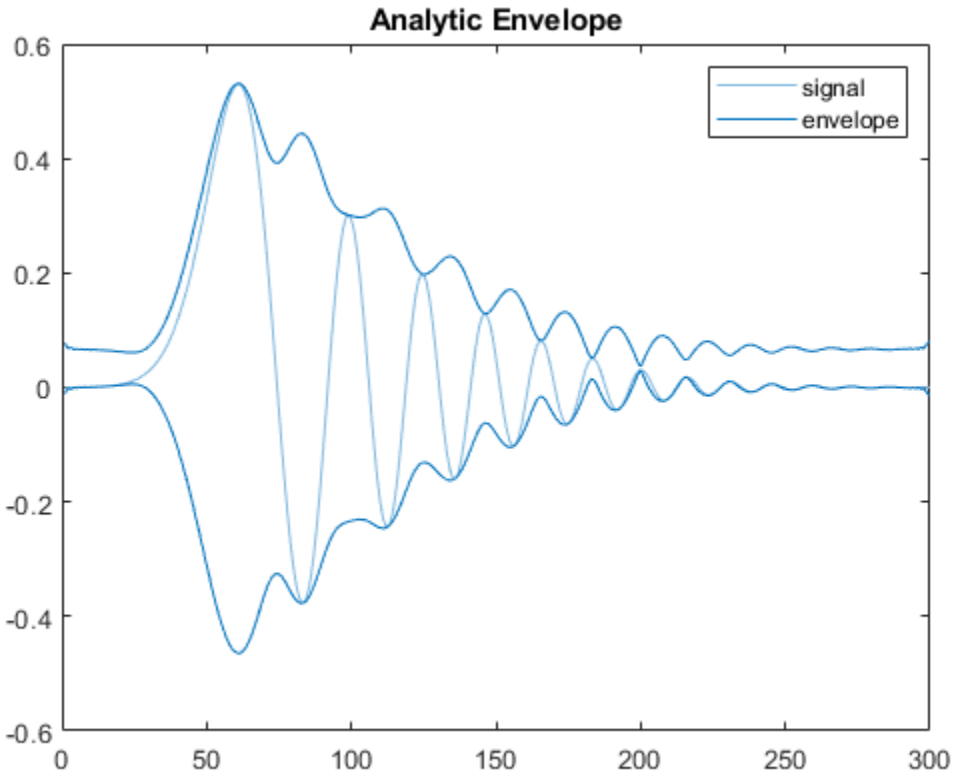
Determine the envelopes of the sequence using the magnitude of its analytic signal. Plot the envelopes.

```
envelope(z)
```



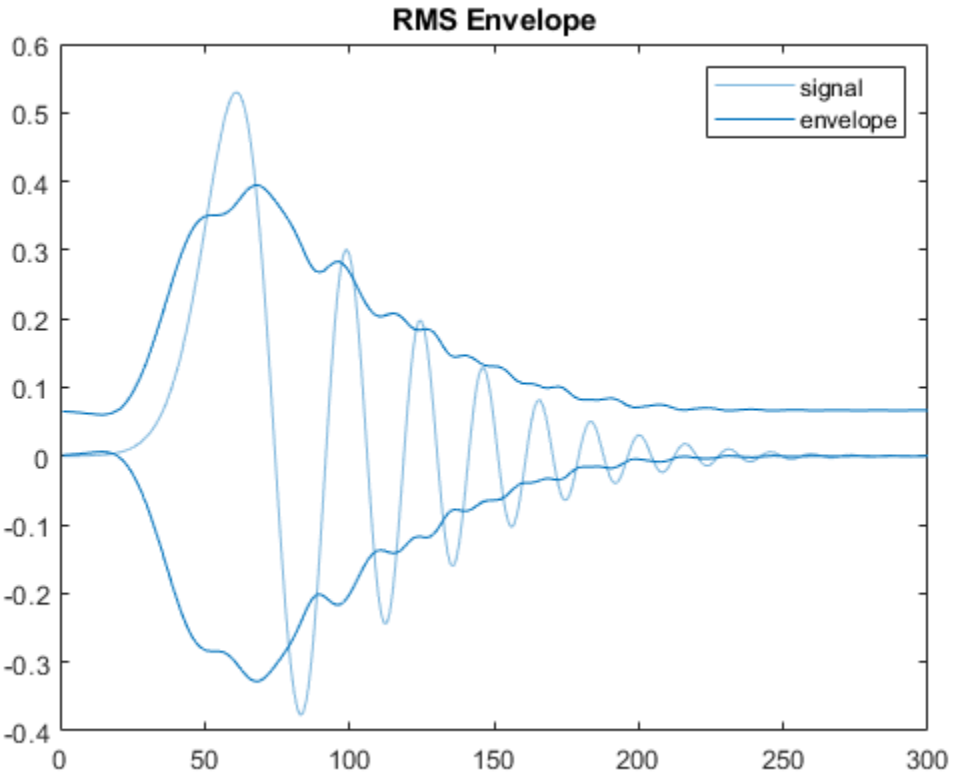
Compute the analytic envelope of the signal using a 50-tap Hilbert filter.

```
envelope(z,50,'analytic')
```



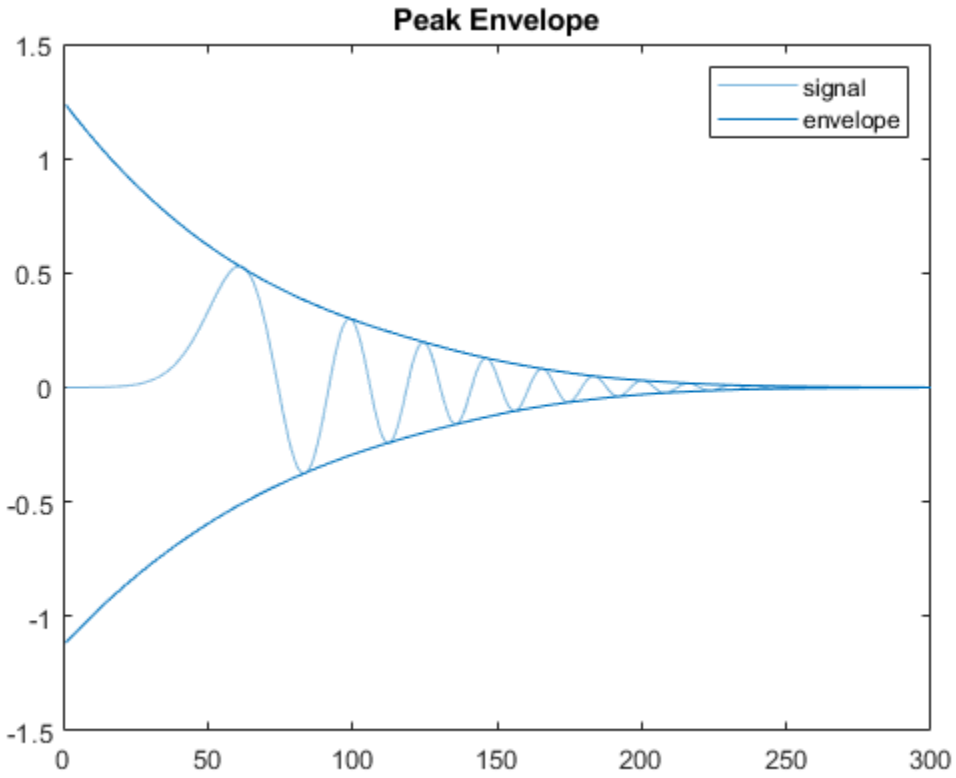
Compute the RMS envelope of the signal using a 40-sample moving window. Plot the result.

```
envelope(z,40,'rms')
```



Determine the peak envelopes. Use spline interpolation with not-a-knot conditions over local maxima separated by at least 10 samples.

```
envelope(z,10,'peak')
```



Input Arguments

x — Input sequence

vector | matrix

Input sequence, specified as a vector or matrix. If `x` is a vector, it is treated as a single channel. If `x` is a matrix, then `envelope` computes the envelope estimates independently for each column. All elements of `x` must be finite.

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`

fl — Hilbert filter length

positive integer scalar

Hilbert filter length, specified as a positive integer scalar. The filter is created by windowing an ideal brick-wall filter with a Kaiser window of length `fl` and shape parameter $\beta = 8$.

Data Types: `single` | `double`

wl — Window length

positive integer scalar

Window length, specified as a positive integer scalar.

Data Types: `single` | `double`

`np` — Peak separation
positive integer scalar

Peak separation, specified as a positive integer scalar.

Data Types: `single` | `double`

Output Arguments

`yupper,ylower` — Upper and lower signal envelopes
vectors | matrices

Upper and lower signal envelopes, returned as vectors or matrices.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Plotting is supported for simulation only. It is not supported in standalone code.

See Also
`findpeaks` | `hilbert` | `movmax` | `movmean` | `movmin` | `rms`

Topics
“Envelope Extraction”

Introduced in R2015b

envspectrum

Envelope spectrum for machinery diagnosis

Syntax

```
es = envspectrum(x,fs)
es = envspectrum(xt)

es = envspectrum( ___ ,Name,Value)

[es,f,env,t] = envspectrum( ___ )

envspectrum( ___ )
```

Description

`es = envspectrum(x,fs)` returns the envelope spectrum of a signal `x` sampled at a rate `fs`. If `x` is a matrix, then the function computes the envelope spectrum independently for each column and returns the result in the corresponding column of `es`.

`es = envspectrum(xt)` returns the envelope spectrum of a signal stored in the MATLAB timetable `xt`.

`es = envspectrum( ___ ,Name,Value)` specifies additional options for any of the previous syntaxes using name-value pair arguments. Options include the algorithm used to compute the envelope signal and the frequency band over which to estimate the spectrum.

`[es,f,env,t] = envspectrum( ___ )` returns `f`, a vector of frequencies at which `es` is computed; `env`, the envelope signal; and `t`, the times at which `env` is computed.

`envspectrum( ___ )` with no output arguments plots the envelope signal and the envelope spectrum in the current figure.

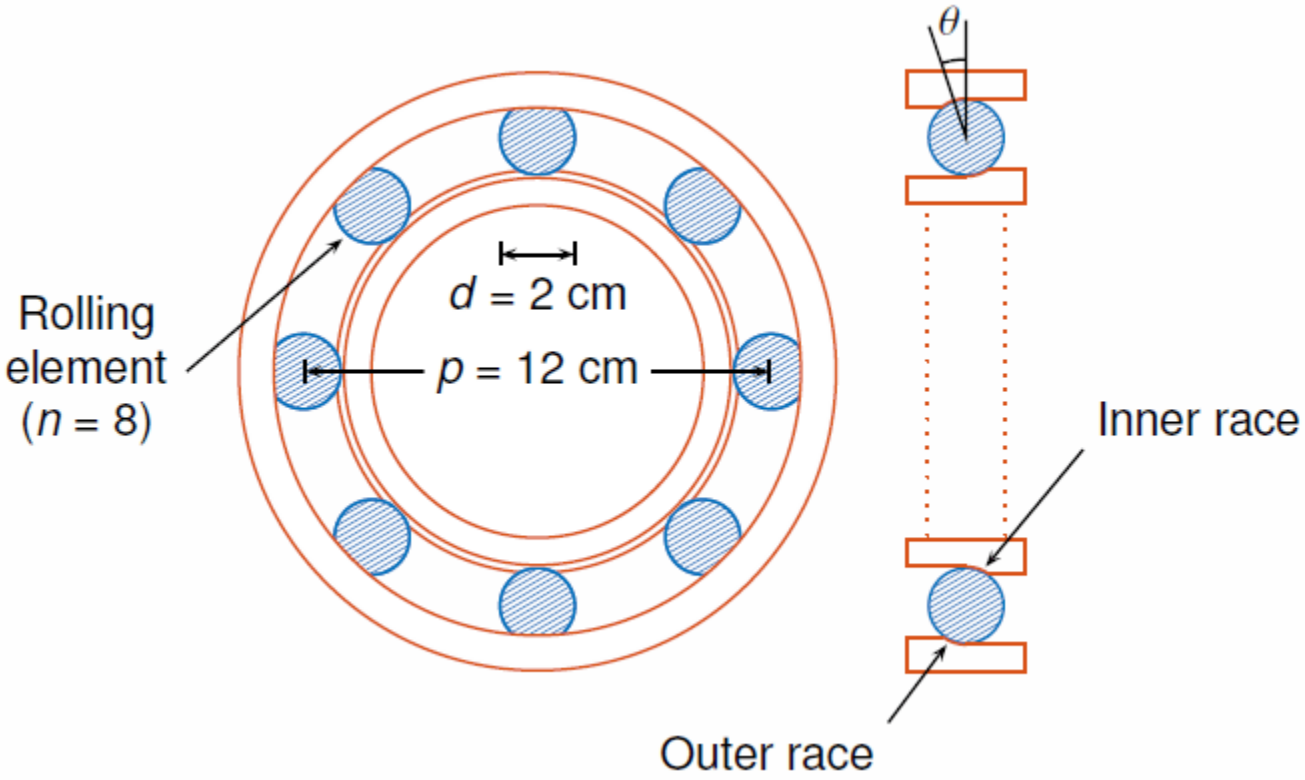
Examples

Envelope Spectrum of Vibration Signals

Simulate two vibration signals, one from a healthy bearing and one from a damaged bearing. Compute and compare their envelope spectra.

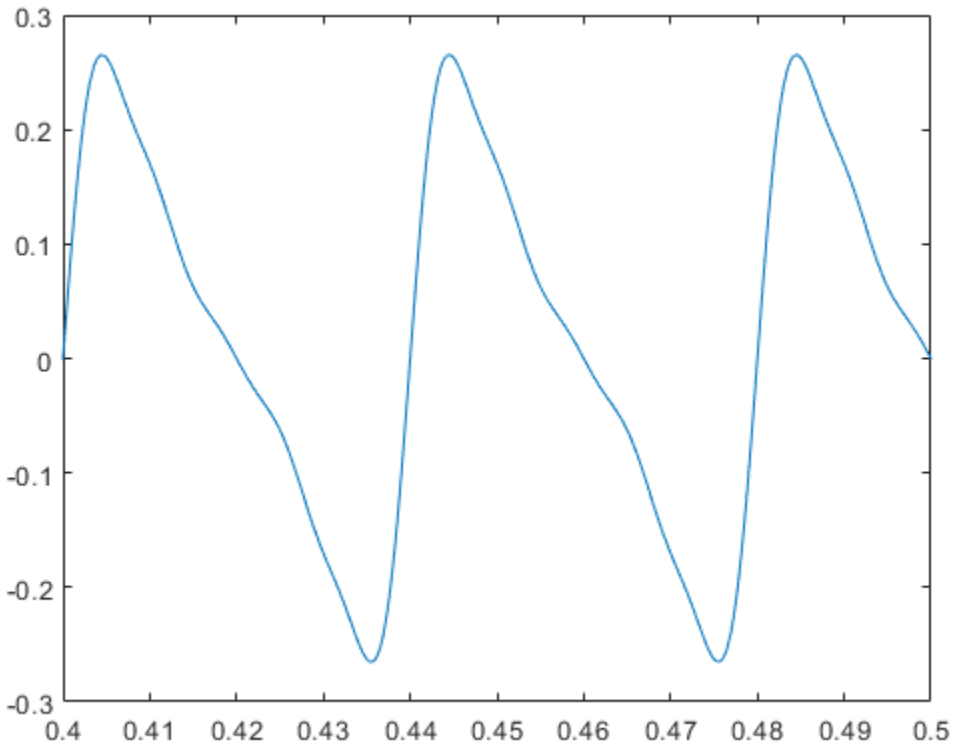
A bearing with a pitch diameter of 12 cm has eight rolling elements. Each rolling element has a diameter of 2 cm. The outer race remains stationary as the inner race is driven at 25 cycles per second. An accelerometer samples the bearing vibrations at 10 kHz.

```
fs = 10000;
f0 = 25;
n = 8;
d = 0.02;
p = 0.12;
```



The vibration signal from the healthy bearing includes several orders of the driving frequency. Plot 0.1 second of data.

```
t = 0:1/fs:1-1/fs;
z = [1 0.5 0.2 0.1 0.05]*sin(2*pi*f0*[1 2 3 4 5]'.*t)/5;

plot(t,z)
xlim([0.4 0.5])
```



A defect in the outer race of the bearing causes a series of 5 millisecond impacts on the bearing. Eventually, those impacts result in bearing wear. The impacts occur at the ball pass frequency outer race (BPFO) of the bearing,

$$\text{BPFO} = \frac{1}{2} n f_0 \left[1 - \frac{d}{p} \cos\theta \right],$$

where f_0 is the driving rate, n is the number of rolling elements, d is the diameter of the rolling elements, p is the pitch diameter of the bearing, and θ is the bearing contact angle. Assume a contact angle of zero and compute the BPFO.

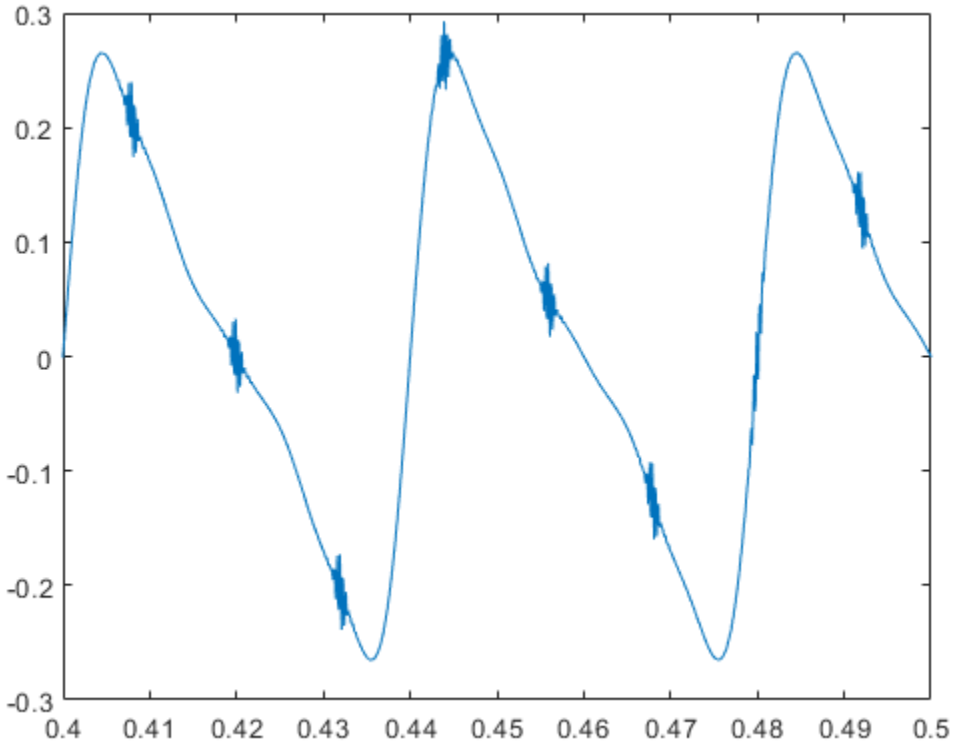
```
ca = 0;
bpfo = n*f0/2*(1-d/p*cos(ca))
```

```
bpfo = 83.3333
```

Model each impact as a 3 kHz sinusoid windowed by a flat top window. Make the impact periodic by convolving it with a comb function. Plot 0.1 second of data.

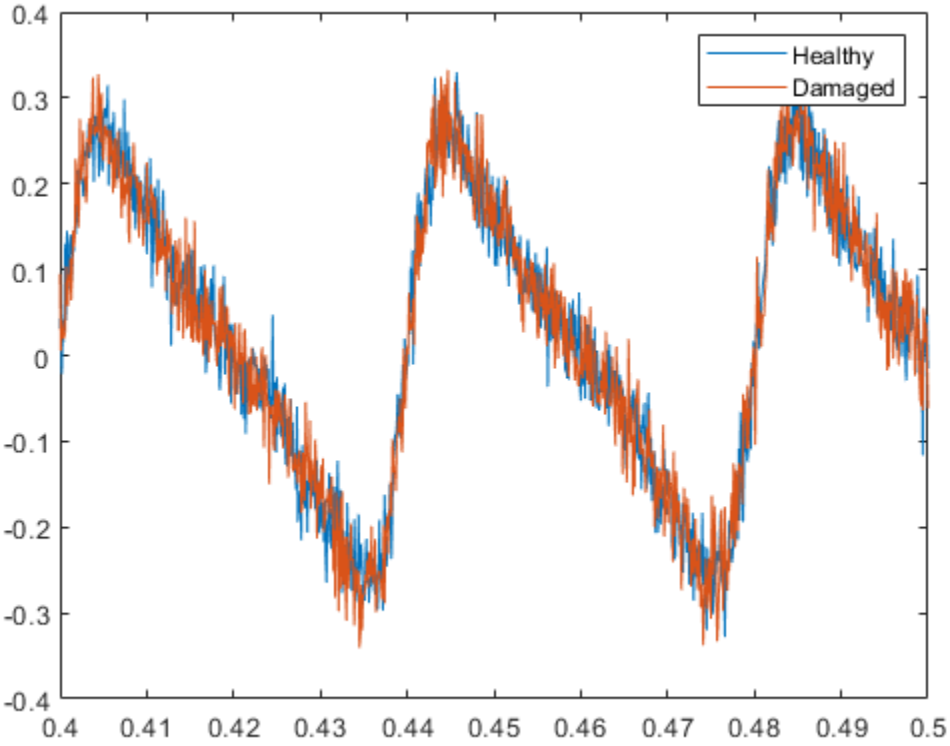
```
fImpact = 3000;
tImpact = 0:1/fs:5e-3-1/fs;
xImpact = sin(2*pi*fImpact*tImpact).*flattopwin(length(tImpact))'/10;

xComb = zeros(size(t));
xComb(1:fs/bpfo:end) = 1;

x = conv(xComb,xImpact,'same')/3;
```

```
plot(t,x+z)
xlim([0.4 0.5])
```



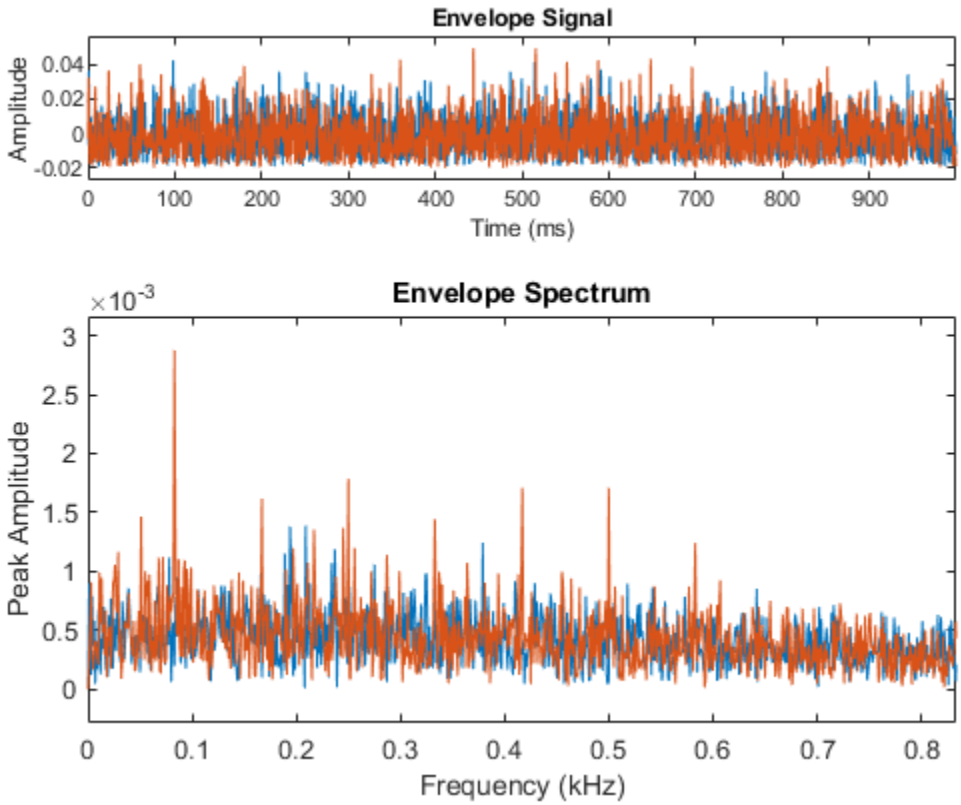
Add white Gaussian noise to the signals. Specify a noise variance of $1/30^2$. Plot 0.1 second of data.

```
yGood = z + randn(size(z))/30;
yBad = x+z + randn(size(z))/30;
plot(t,yGood,t,yBad)
xlim([0.4 0.5])
legend('Healthy','Damaged')
```



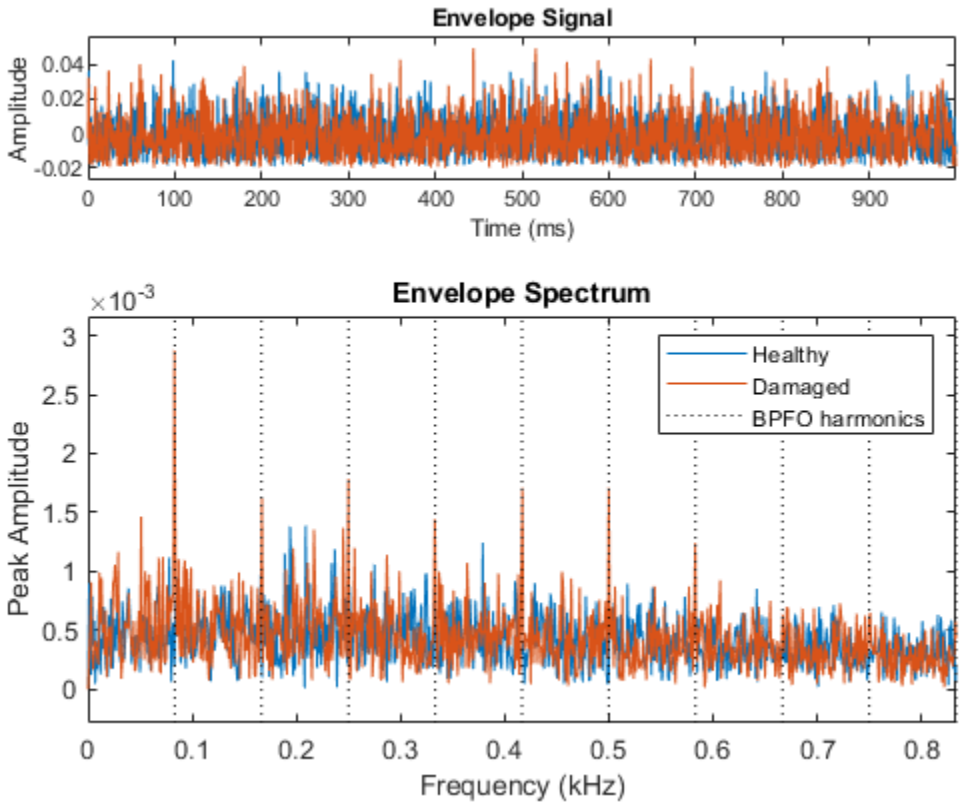
Compute and plot the envelope signals and spectra.

```
envspectrum([yGood' yBad'],fs)
xlim([0 10*bpfo]/1000)
```



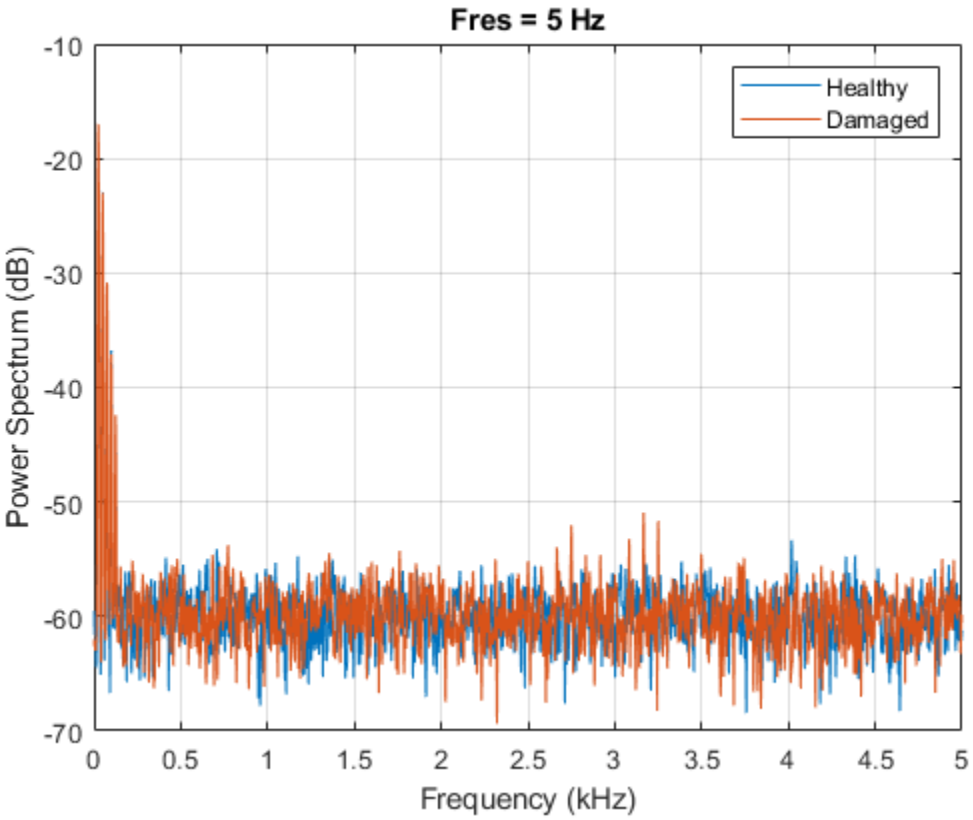
Compare the peak locations to the frequencies of harmonics of the BPFO. The BPFO harmonics in the envelope spectrum are a sign of bearing wear.

```
harmImpact = (1:10)*bpfo;
[X,Y] = meshgrid(harmImpact,ylim);

hold on
plot(X/1000,Y,':k')
legend('Healthy','Damaged','BPFO harmonics')
hold off
```



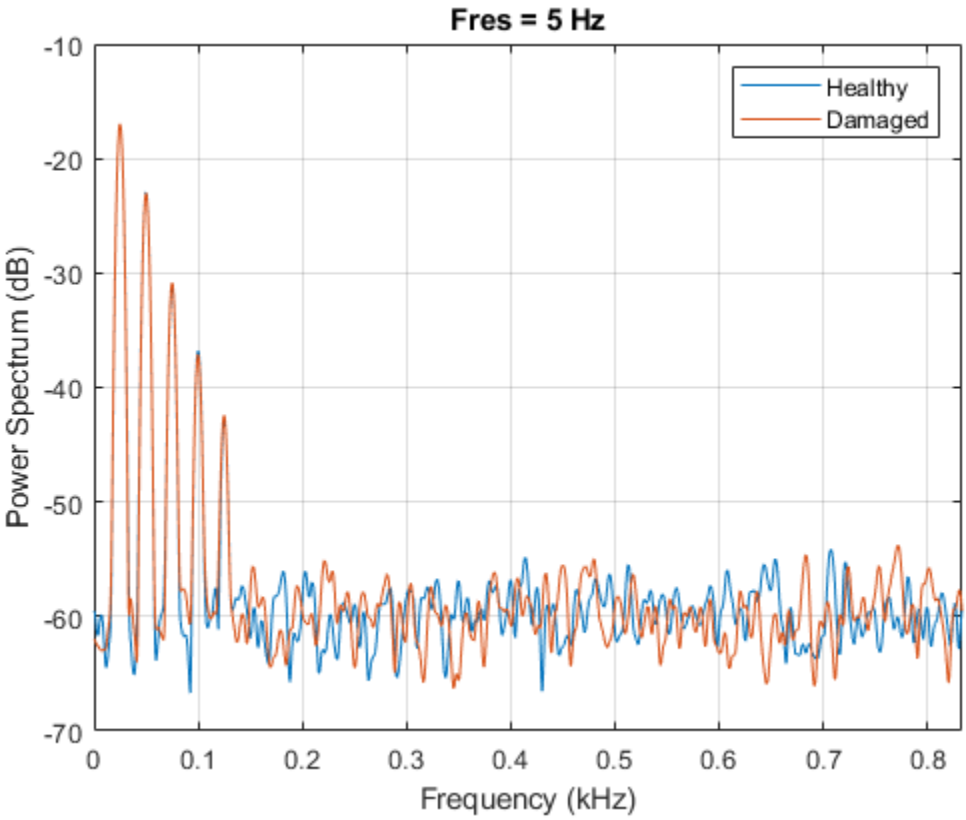
Compute the Welch spectra of the signals. Specify a frequency resolution of 5 Hz.

```
figure
pspectrum([yGood' yBad'],fs,'FrequencyResolution',5)
legend('Healthy','Damaged')
```



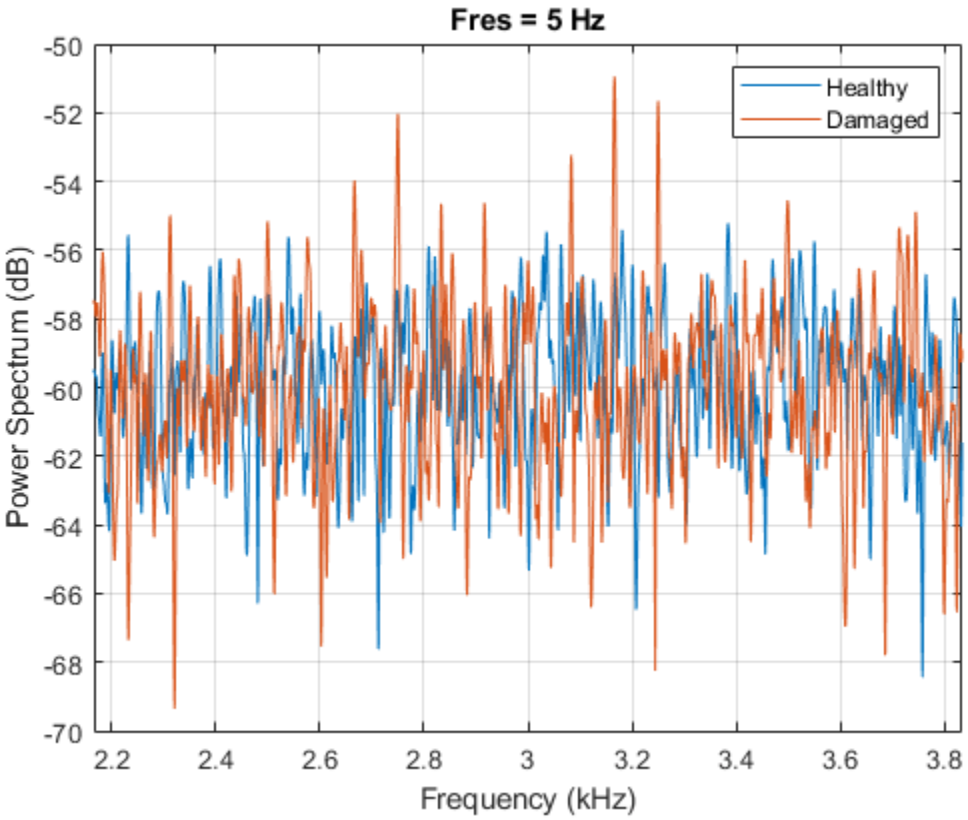
At the low end of the spectrum, the driving frequency and its orders obscure other features. The spectrum of the healthy bearing and the spectrum of the damaged bearing are indistinguishable.

```
xlim([0 10*bpfo]/1000)
```



The spectrum of the faulty bearing shows BPFO harmonics modulated by the impact frequency.

```
xlim((bpfo*[-10 10]+fImpact)/1000)
```



Envelope Spectrum of Timetable

Generate a two-channel signal that resembles the vibration signals from a bearing that completes a rotation every 10 milliseconds. The signal is sampled at 10 kHz for 0.2 seconds, which corresponds to 20 bearing rotations.

```
fs = 10000;
tmax = 20;
mlt = 0.01;
t = 0:1/fs:mlt-1/fs;
```

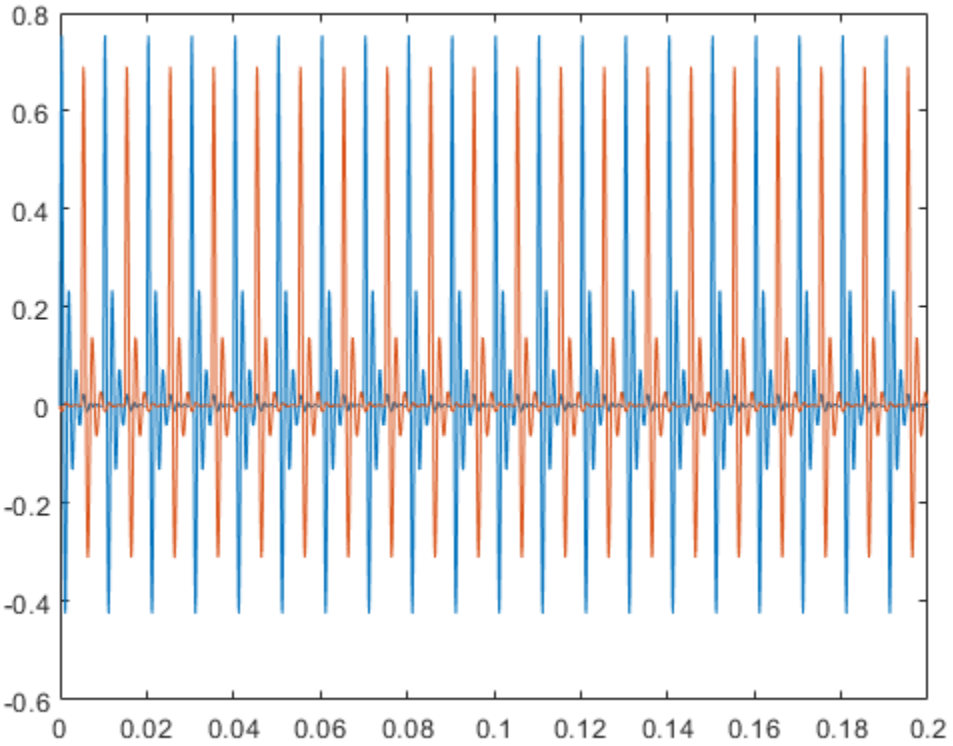
During each 10-millisecond interval:

- The first channel is a damped sinusoid with damping constant 700 and sinusoid frequency 600 Hz.
- The second channel is another damped sinusoid with damping constant 800 and sinusoid frequency 500 Hz. The second channel lags the first channel by 5 milliseconds.

Plot the signal.

```
y1 = sin(2*pi*600*t).*exp(-700*t);
y2 = sin(2*pi*500*t).*exp(-800*t);
y2 = [y2(51:100) y2(1:50)];

T = (0:1/fs:mlt*tmax-1/fs)';
```

```
Y = repmat([y1;y2],1,tmax)';

plot(T,Y)
```

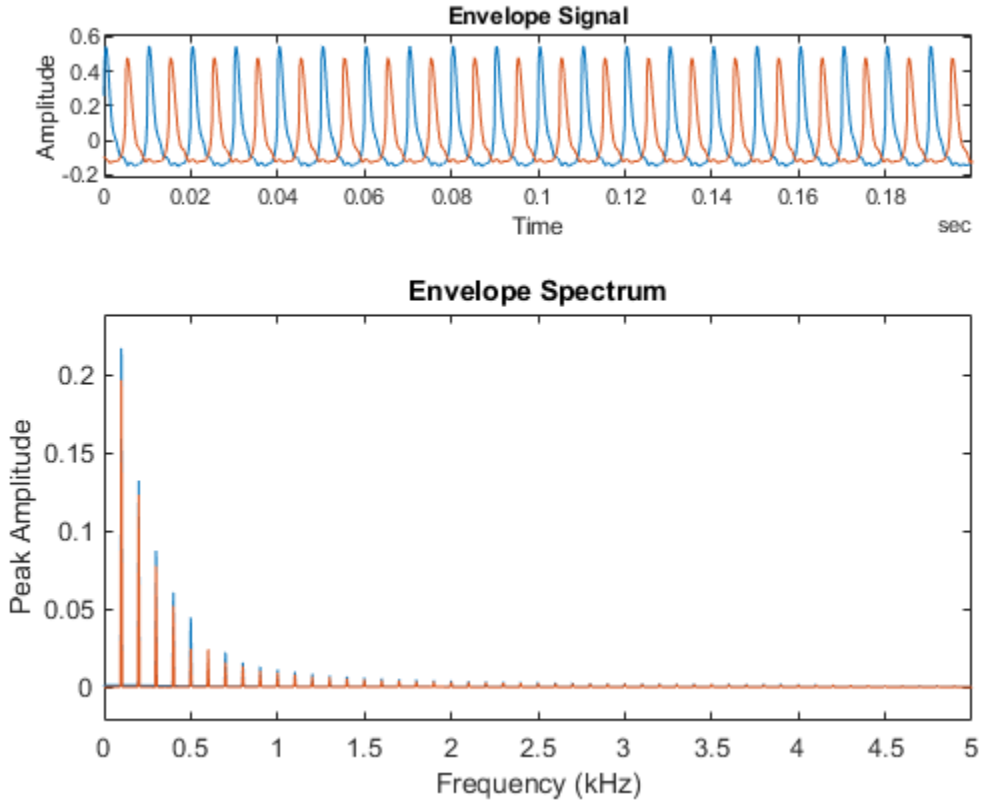


Create a duration array using the time interval T. Construct a timetable with the duration array and the two-channel signal.

```
dt = seconds(T);
ttb = timetable(dt,Y);
```

Use `envspectrum` with no output arguments to display the envelope signal and envelope spectrum of the two channels. Compute the spectrum on the whole Nyquist interval, excluding 100 Hz intervals at the ends.

```
envspectrum(ttb,'Band',[100 4900])
```



The envelope spectra of the signals have peaks at integer multiples of the repetition rate of 1/0.01 = 0.1 kHz. This is just as expected. `envspectrum` removes the high-frequency sinusoidal components and focuses on the lower-frequency repetition behavior. This is why the envelope spectrum is a useful tool for the analysis of rotational machinery.

Compute the envelope signal and the times at which it is computed. Check the types of the output variables.

```
[~,~,ttbenv,ttbt] = envspectrum(ttb,'Band',[100 4900]);
whos ttb*
```

```
  Name           Size            Bytes  Class        Attributes

  ttb         2000x1             48977  timetable
  ttbenv      2000x1             48985  timetable
  ttbt        2000x1             16002  duration
```

The time vector is of `duration` type, like the time values of the input timetable. The output timetable has the same size as the input timetable.

Store each channel of the input timetable as a separate variable. Compute the envelope signal and the time vector. Check the output types.

```
btb = timetable(dt,Y(:,1),Y(:,2));

[~,~,btbenv,btbt] = envspectrum(btb,'Band',[100 4900]);
whos btb*
```

```
  Name           Size            Bytes  Class        Attributes

  btb         2000x2             49199  timetable
  btbenv      2000x2             49219  timetable
  btbt        2000x1             16002  duration
```

The output timetable has the same size as the input timetable.

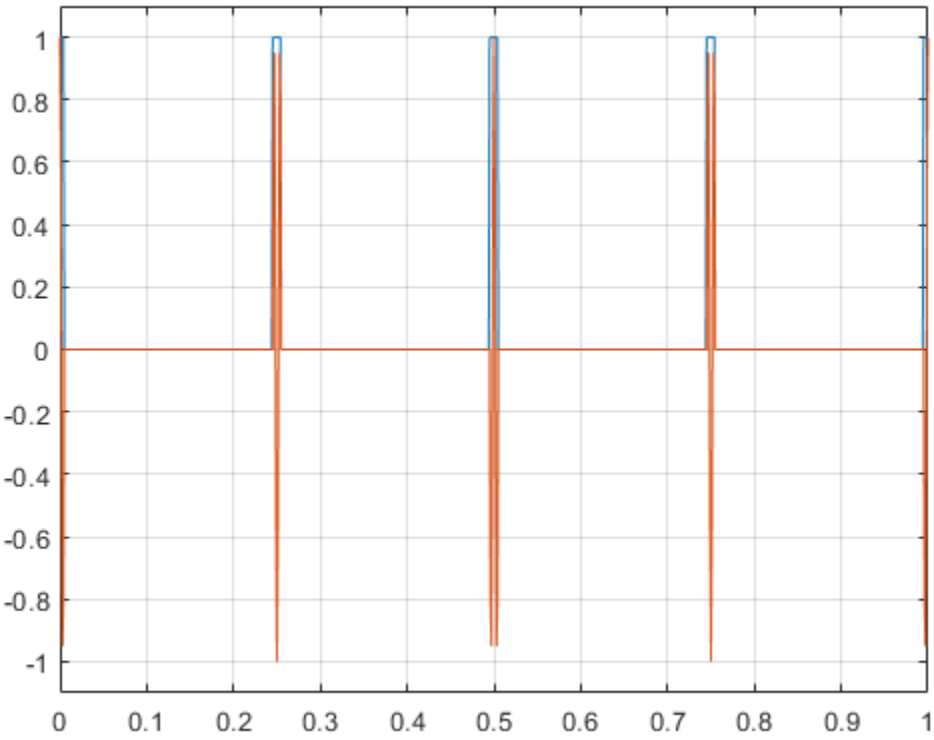
Envelope Spectrum of Modulated Pulses

Generate a signal sampled at 1 kHz for 5 seconds. The signal consists of 0.01-second rectangular pulses that repeat every $T = 0.25$ second. Amplitude modulate the signal onto a sinusoid of carrier frequency 150 Hz.

```
fs = 1e3;
tmax = 5;

t = 0:1/fs:tmax;
y = pulstran(t,0:0.25:tmax,'rectpuls',0.01);

fc = 150;
z = modulate(y,fc,fs);
```

Plot the original and modulated signals. Show only the first few cycles.

```
plot(t,y,t,z,'-')
grid on
axis([0 1 -1.1 1.1])
```

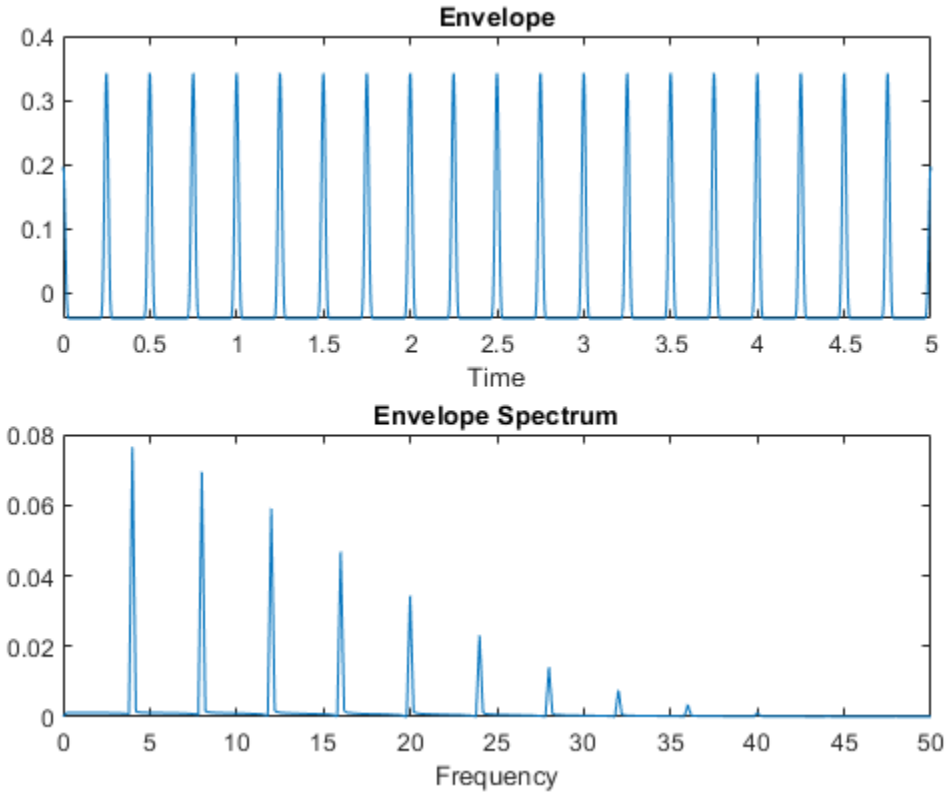


Compute the envelope and envelope spectrum of the signal. Determine the signal envelope using complex demodulation. Compute the envelope spectrum on a 20 Hz interval centered at the carrier frequency.

```
[q,f,e,te] = envspectrum(z,fs,'Method','demod','Band',[fc-10 fc+10]);
```

Plot the envelope signal and the envelope spectrum. Zoom in on the interval from 0 to 50 Hz.

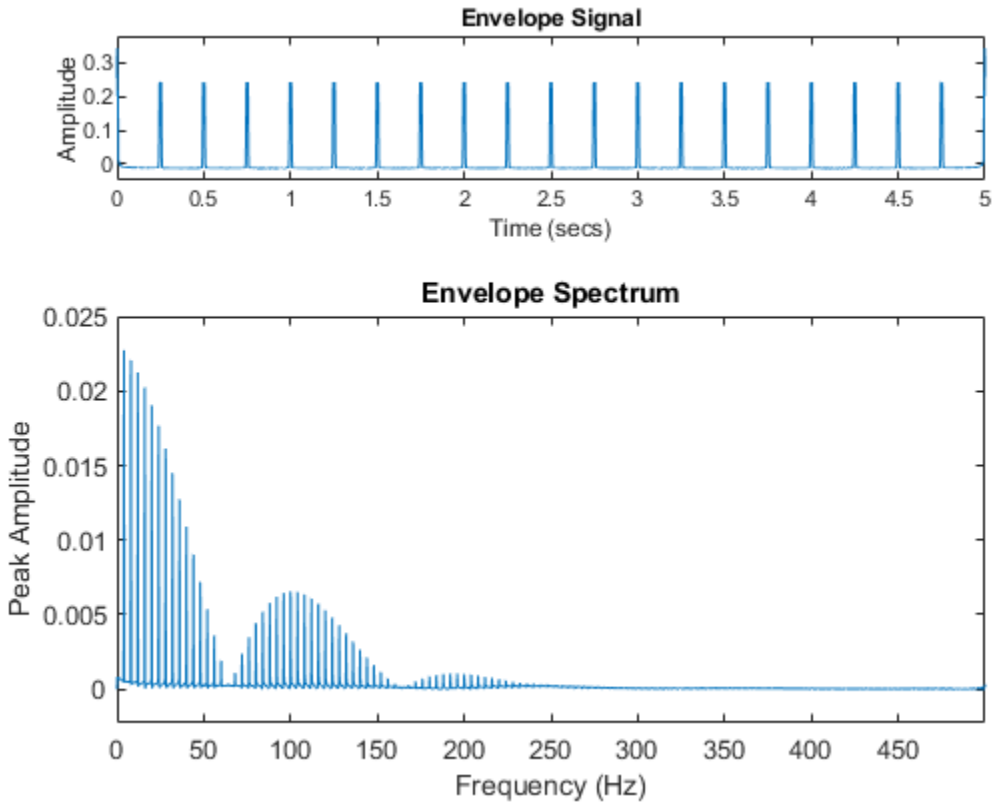
```
subplot(2,1,1)
plot(te,e)
xlabel('Time')
title('Envelope')

subplot(2,1,2)
plot(f,q)
xlim([0 50])
xlabel('Frequency')
title('Envelope Spectrum')
```



The envelope signal has the same period in time, $T = 0.25$ second, as the original signal. The envelope spectrum has pulses at $1 / T = 4$ Hz.

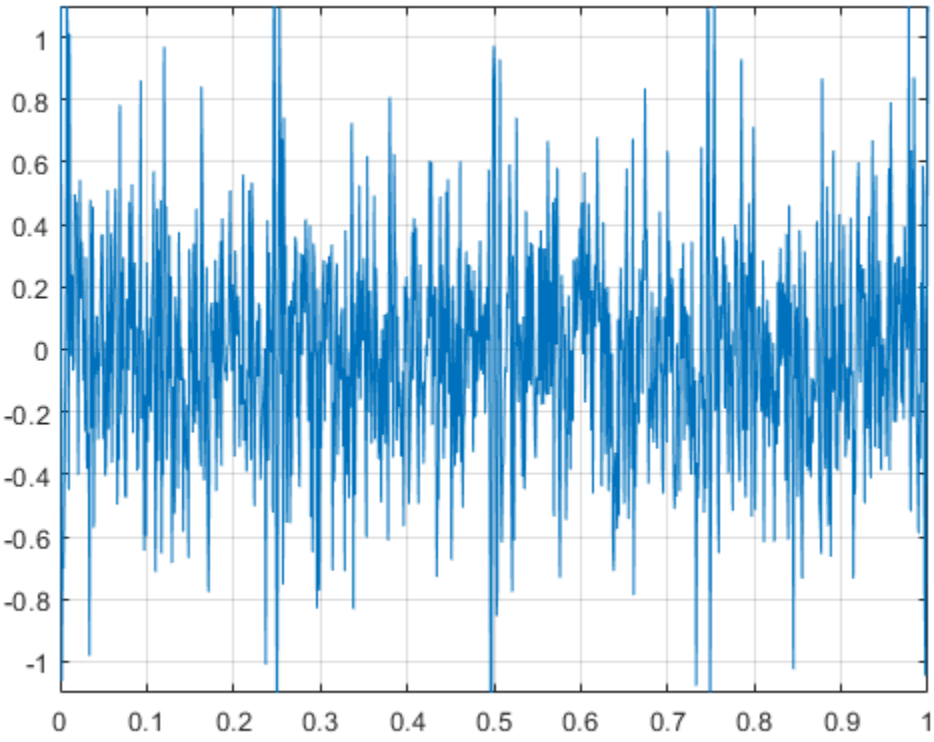
Repeat the computation, but now use the `hilbert` function to compute the envelope. Bandpass-filter the signal using a 10th-order finite impulse response (FIR) filter. Plot the envelope signal and envelope spectrum using the built-in functionality of `envspectrum`.

```
envspectrum(z,fs,'Method','hilbert','FilterOrder',10)
```



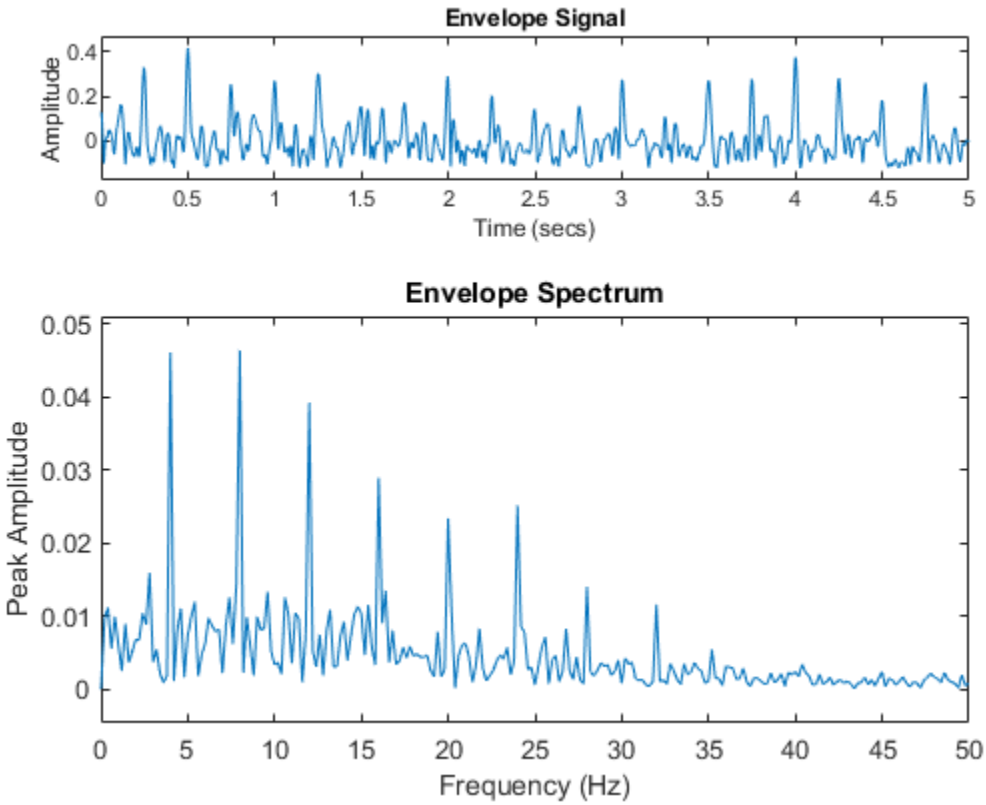
Embed the signal in white Gaussian noise of variance 1/3. Plot the result.

```
zn = z + randn(size(z))/3;

plot(t,zn,'-')
grid on
axis([0 1 -1.1 1.1])
```



Compute and display the envelope signal and envelope spectrum. Compute the envelope spectrum using complex demodulation on a 10 Hz interval centered at the carrier frequency. Zoom in on the interval from 0 to 50 Hz.

```
envspectrum(zn,fs,'Band',[fc-5 fc+5])
xlim([0 50])
```



Input Arguments

`x` — Input signal

vector | matrix

Input signal, specified as a vector or a matrix. If `x` is a vector, it is treated as a single channel. If `x` is a matrix, then `envspectrum` computes the envelope spectrum independently for each column and returns the result in the corresponding column of `es`.

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

`fs` — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar.

Data Types: `single` | `double`

`xt` — Input timetable

timetable

Input timetable. `xt` must contain increasing finite row times. If `xt` represents a multichannel signal, then it must have either a single variable containing a matrix or multiple variables consisting of vectors.

If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

Example: `timetable(seconds(0:4)',randn(5,2))` specifies a two-channel, random variable sampled at 1 Hz for 4 seconds.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Method','hilbert','FilterOrder',30,'Band',[0 fs/4]` computes the envelope spectrum between 0 and one-half the Nyquist frequency using a 30th-order bandpass filter and computing the envelope of the analytic signal.

`Method` — Algorithm for computing the envelope signal

`'demod'` (default) | `'hilbert'`

Algorithm for computing the envelope signal, specified as the comma-separated pair consisting of `'Method'` and either `'hilbert'` or `'demod'`. See “Algorithms” on page 1-613 for more information.

`Band` — Frequency band to compute envelope spectrum

`[fs/4 fs*3/8]` (default) | two-element vector

Frequency band to compute envelope spectrum, specified as the comma-separated pair consisting of `'Band'` and a two-element vector of strictly increasing values between 0 and the Nyquist frequency.

Data Types: `single` | `double`
Complex Number Support: Yes

`FilterOrder` — FIR filter order

`50` (default) | positive integer scalar

FIR filter order, specified as the comma-separated pair consisting of `'FilterOrder'` and a positive integer scalar.

- If `'Method'` is `'hilbert'`, then this argument specifies the order of an FIR bandpass filter.
- If `'Method'` is `'demod'`, then this argument specifies the order of an FIR lowpass filter.

Data Types: `single` | `double`

Output Arguments

`es` — Envelope spectrum

vector | matrix

Envelope spectrum, returned as a vector or matrix.

`f` — Frequencies
vector

Frequencies at which the envelope spectrum is computed, returned as a vector.

`env` — Envelope signal
vector | matrix | timetable

Envelope signal, returned as a vector, matrix, or timetable.

If the input to `envspectrum` is a timetable, then `env` is also a timetable. The time values of `env` have the same format as the time values of the input timetable.

- If the input is a timetable with a single variable containing a matrix, then `env` has a single variable containing a matrix.
- If the input is a timetable with multiple variables consisting of vectors, then `env` has multiple variables consisting of vectors.

`t` — Time values
vector

Time values at which the envelope signal is computed, returned as a vector.

If the input to `envspectrum` is a timetable, then `t` has the same format as the time values of the input timetable.

Algorithms

`envspectrum` initially removes the DC bias from the input signal, `x`, and then computes the envelope signal.

- If `'Method'` is set to `'hilbert'`, the function:

  1. Bandpass-filters the signal. The FIR filter has an order specified by `'FilterOrder'` and cutoff frequencies at `ba(1)` and `ba(2)`, where `ba` is a frequency band specified using `'Band'`.
  2. Computes the analytic signal using the `hilbert` function.
  3. Computes the envelope signal as the absolute value of the analytic signal.

- If `'Method'` is set to `'demod'`, the function:

  1. Performs complex demodulation of the signal. The signal is multiplied by $\exp(j2\pi f_0 t)$, where f_0 = (`ba(1)` + `ba(2)`)/2.
  2. Lowpass-filters the demodulated signal to compute the analytic signal. The FIR filter has an order specified by `'FilterOrder'` and a cutoff frequency of (`ba(2)` – `ba(1)`)/2.
  3. Computes the envelope signal as twice the absolute value of the analytic signal.

After computing the envelope signal, the function removes the DC bias from the envelope and computes the envelope spectrum using the FFT.

References

[1] Randall, Robert Bond. *Vibration-Based Condition Monitoring*. Chichester, UK: John Wiley & Sons, 2011.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Timetables are not supported for code generation.

See Also

`envelope` | `hilbert` | `orderspectrum`

Topics

“Vibration Analysis of Rotating Machinery”
“Rolling Element Bearing Fault Diagnosis” (Predictive Maintenance Toolbox)

Introduced in R2017b

equiripple

Equiripple single-rate FIR filter from specification object

Syntax

```
hd = design(d,'equiripple')
hd = design(d,'equiripple',Name,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 = design(d,'equiripple',Name,Value)` returns an equiripple FIR filter where you specify design options as `Name,Value` pairs.

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.

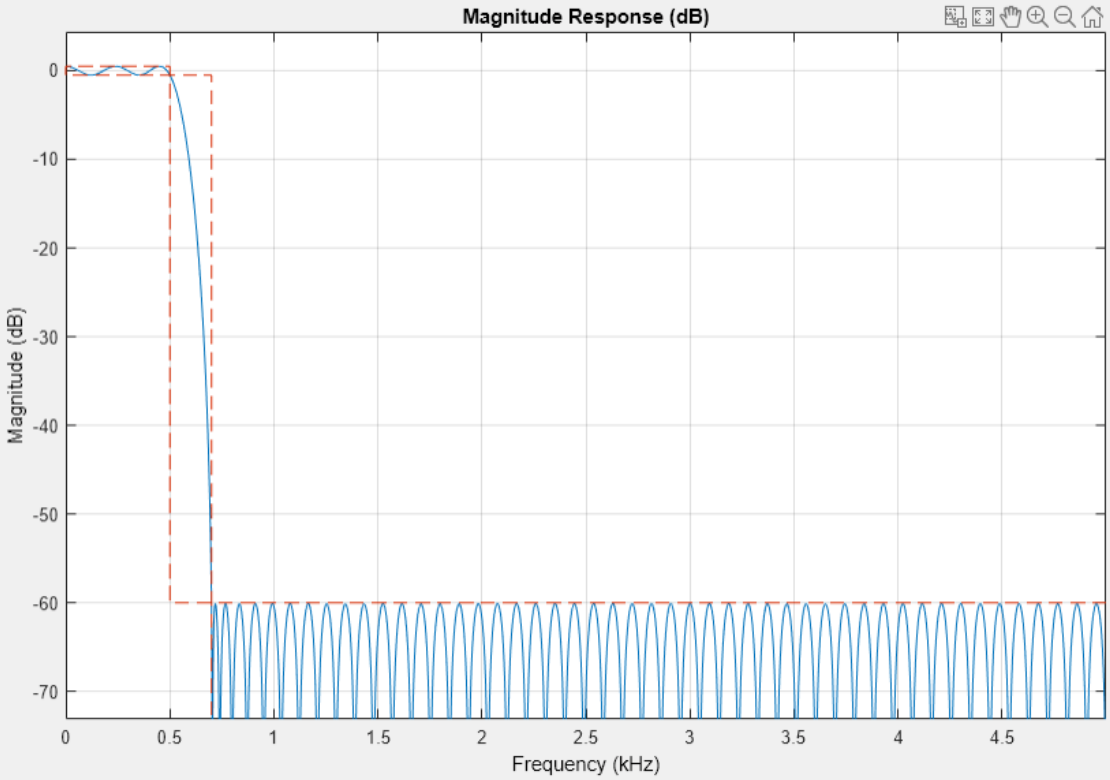
```
help(d,'equiripple')
```

Examples

Lowpass Equiripple Filter

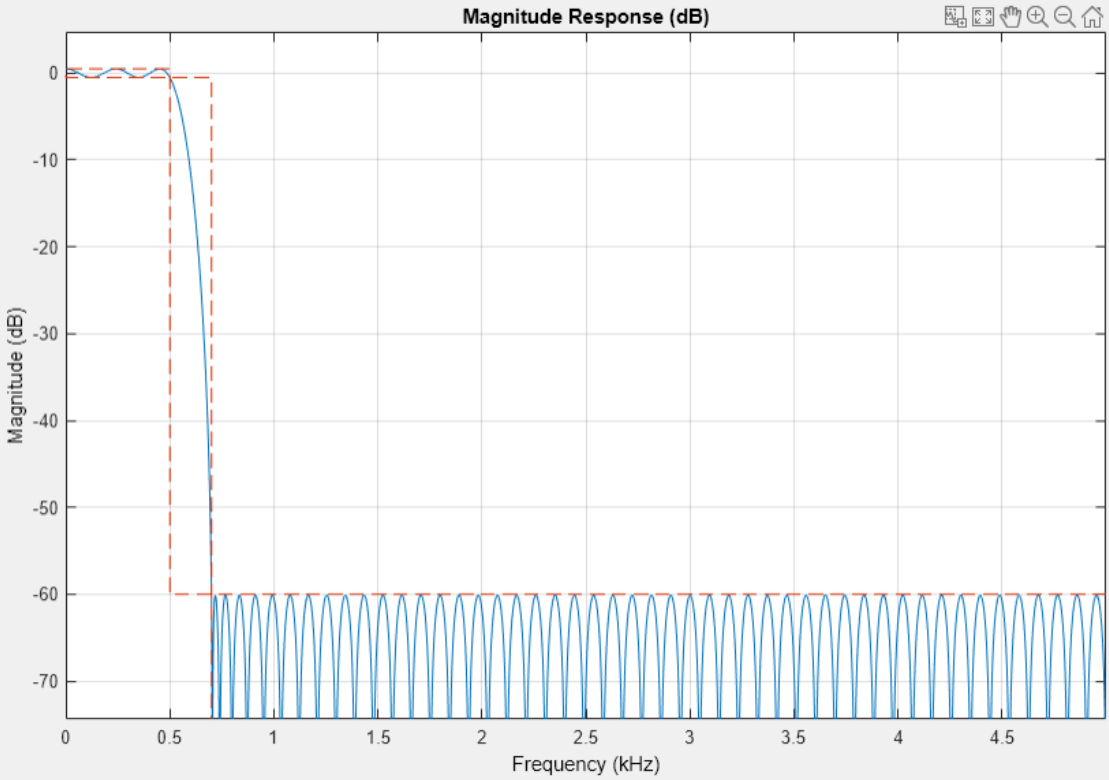
Create a lowpass equiripple filter. Assume the data is sampled at 10 kHz. The passband frequency is 500 Hz and the stopband frequency of 700 Hz. The desired passband ripple is 1 dB with 60 dB of stopband attenuation. Use FVTool to display the magnitude response of the filter.

```
Fs = 10000;

Hd = fdesign.lowpass('Fp,Fst,Ap,Ast',500,700,1,60,Fs);
d = design(Hd,'equiripple');

fvtool(d)
```



Design a lowpass equiripple filter with direct-form transposed structure and a density factor of 20.

```
df = design(Hd,'equiripple','FilterStructure','dffirt','DensityFactor',20);
fvtool(df)
```



See Also

Apps
Filter Designer

Functions
`designfilt` | `fdesign`

Introduced in R2009a

eqtflength

Equalize lengths of transfer function 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` or the vector `den` so that the resulting output vectors `b` and `a` have the same length. `b` and `a` represent the same discrete-time transfer function as `num` and `den`, but are of equal length.

`[b,a,n,m] = eqtflength(num,den)` returns the numerator order `n` and the denominator order `m`, not including any trailing zeros.

Examples

Equalize Transfer Function Numerator and Denominator Lengths

Consider the following discrete-time SISO transfer function model:

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

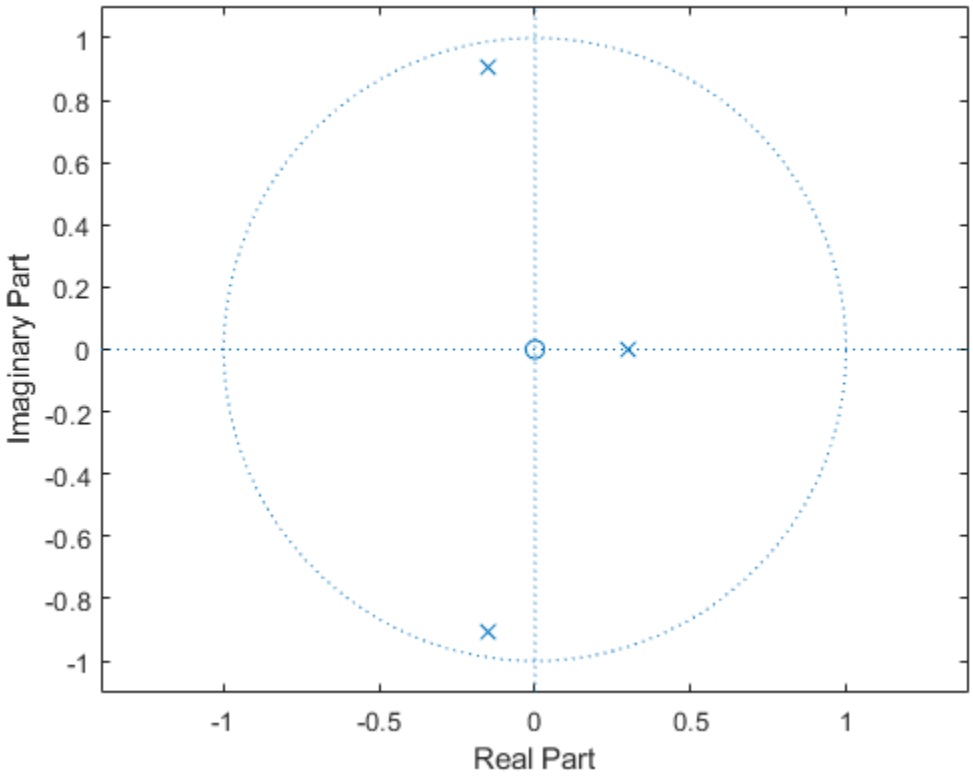
Equalize the numerator and denominator polynomial lengths. Determine the polynomial orders.

```
num = [0 0 2];
den = [4 0 3 -1];

[b,a,n,m] = eqtflength(num,den)
```

```
b = 1×4

     0     0     2     0


a = 1×4

     4     0     3    -1


n = 2

m = 3
```

Visualize the poles and zeros of the transfer function.

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



Convert the transfer function with equalized numerator and denominator to state-space form. `b` and `a` must have equal lengths to find the state-space representation of the discrete-time transfer function.

```
[A,B,C,D] = tf2ss(b,a)
```

```
A = 3×3

         0   -0.7500    0.2500
    1.0000         0         0
         0    1.0000         0


B = 3×1

     1
     0
     0


C = 1×3

         0    0.5000         0


D = 0
```

Input Arguments

num — Numerator coefficients

vector

Numerator polynomial coefficients of discrete-time transfer function, specified as a vector.

Data Types: double
Complex Number Support: Yes

den — Denominator coefficients

vector

Denominator polynomial coefficients of discrete-time transfer function, specified as a vector.

Data Types: double
Complex Number Support: Yes

Output Arguments

b — Numerator coefficients

row vector

Numerator polynomial coefficients of discrete-time transfer function, returned as a row vector. b has the same length as a.

a — Denominator coefficients

row vector

Denominator polynomial coefficients of discrete-time transfer function, returned as a row vector. a has the same length as b.

n — Numerator order

integer

Numerator order, returned as an integer. Any trailing zeros in b are excluded when computing n.

m — Denominator order

integer

Denominator order, returned as an integer. Any trailing zeros in a are excluded when computing m.

Tips

- 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.

Algorithms

eqtflength(num,den) appends zeros to either num or den as necessary. eqtflength removes any trailing zeros that num and den have in common.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`tf2ss` | `tf2zp`

Introduced before R2006a

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% reference level to the 10% reference level (See “Percent Reference Levels” on page 1-628). To determine the transitions, `falltime` estimates the state levels of the input waveform using a histogram method. `falltime` identifies all regions that 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 1-628). 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 sample rate in hertz. The sample rate 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. You can use this output syntax with any of the previous input syntaxes.

`[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.

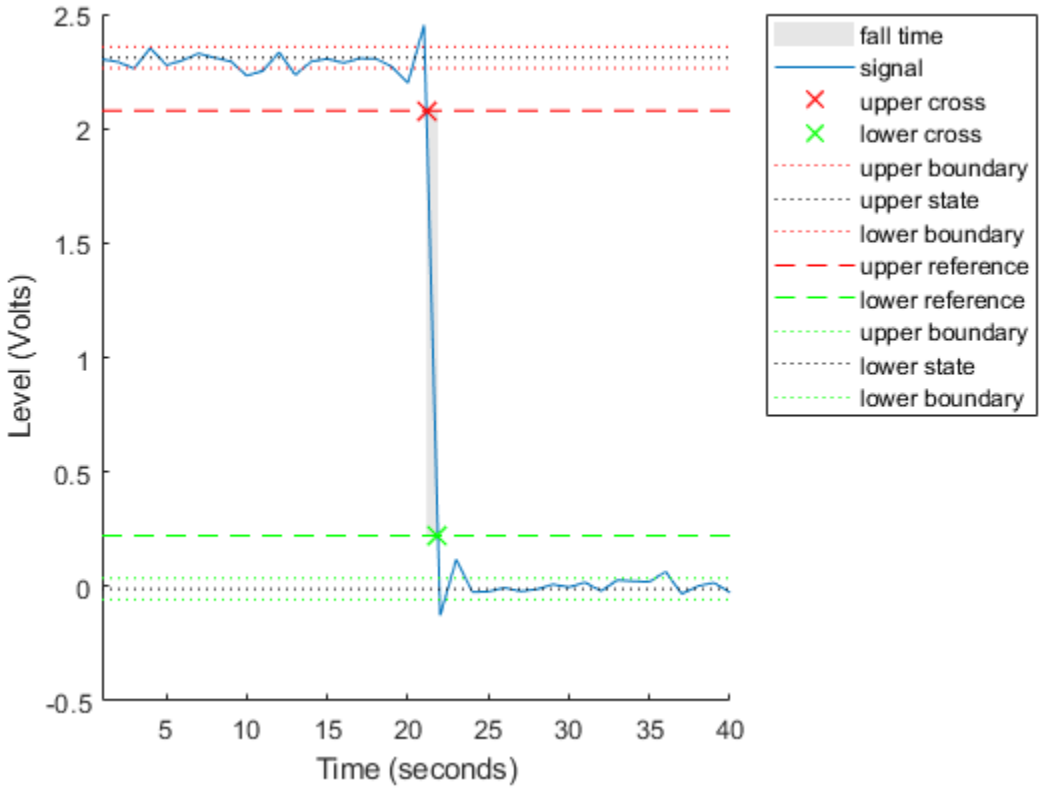
Examples

Fall Time of 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% and 90% percent reference levels. Plot the waveform and annotate the fall time.

```
load('negtransitionex.mat','x')

falltime(x)
```



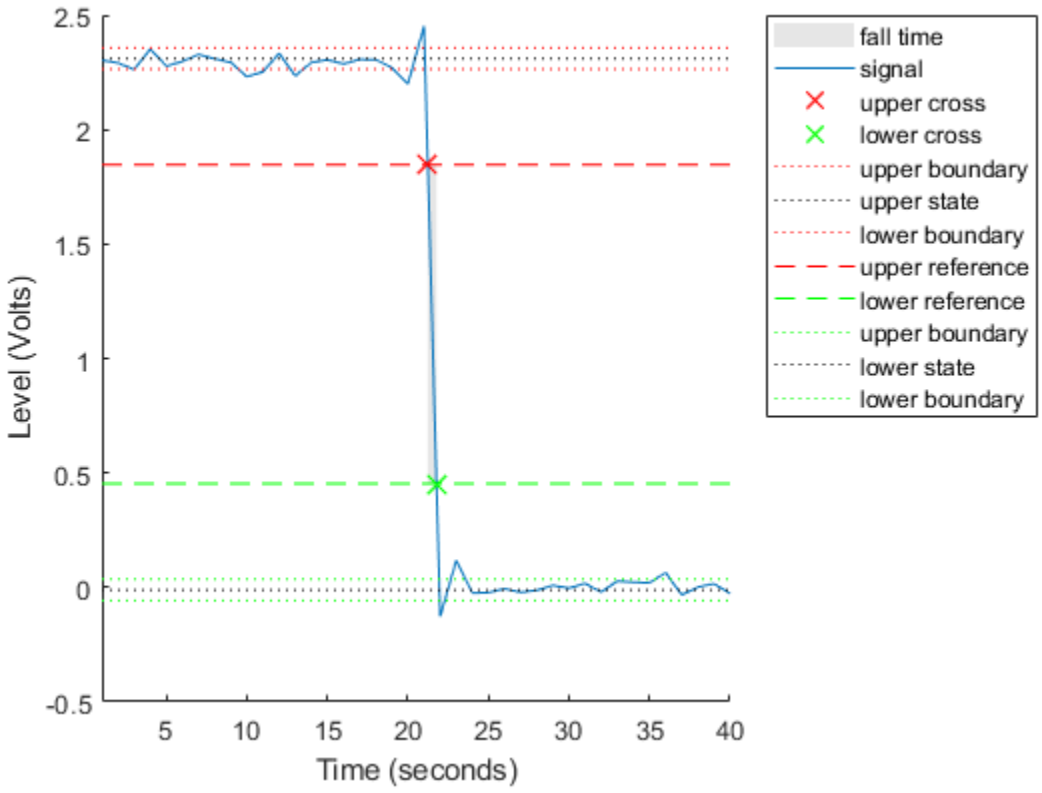

```
ans = 0.7200
```

Fall Time 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. Determine the fall time using 20% and 80% reference levels. Plot the waveform and annotate the fall time.

```
load('negtransitionex.mat','x','t')

falltime(x,'PercentReferenceLevels',[20 80])
```




```
ans = 0.5400
```

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')
```

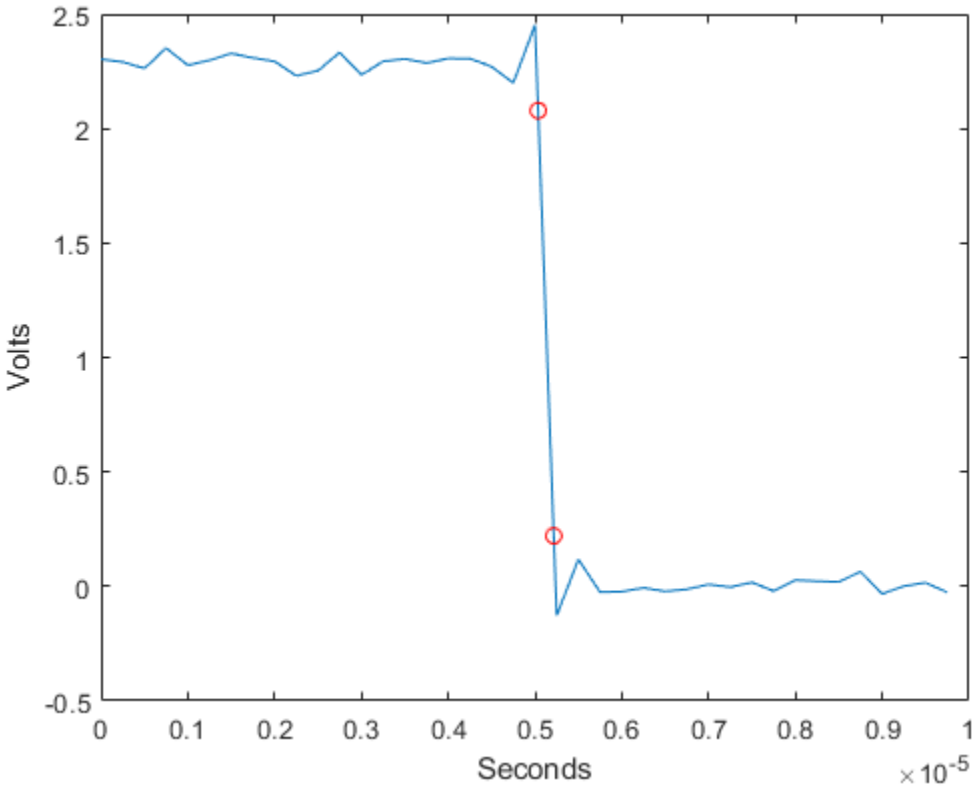
Determine the fall time, reference-level instants, and reference levels.

```
[f,lt,ut,ll,ul] = falltime(x,t);
```

Plot the waveform 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,x)

xlabel('Seconds')
ylabel('Volts')

hold on
```

```
plot([lt ut],[ll ul],'ro')
hold off
```




```
fprintf('Rise time is %g seconds.',lt-ut)
```

```
Rise time is 1.8e-07 seconds.
```

Input Arguments

x — Bilevel waveform

real vector

Bilevel waveform, specified as a real-valued vector.

fs — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar in hertz.

t — sample instants

real vector

Sample instants, specified as a vector. The length of `t` must equal the length of the bilevel waveform `x`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

`PercentReferenceLevels` — Reference levels as a percentage of the waveform amplitude
`[10 90]` (default) | two-element positive row vector

Reference levels as a percentage of the waveform amplitude, specified as the comma-separated pair consisting of `'PercentReferenceLevels'`and a two-element positive row vector. The elements of the row vector correspond to the lower and upper percent reference levels. The high state level is defined to be 100 percent, and the low state level is defined to be 0 percent. See “Percent Reference Levels” on page 1-628 for more details.

`StateLevels` — Low and high state levels
two-element positive row vector

Low and high state levels, specified as the comma-separated pair consisting of `'StateLevels'` and a two-element positive row vector. The first and second elements of the vector correspond to the low and high state levels.

`Tolerance` — Lower- and upper- state boundaries
2 (default) | real positive scalar

Lower- and upper-state boundaries, specified as the comma-separated pair consisting of `'Tolerance'` and a real positive scalar as a percentage value. See “State-Level Tolerances” on page 1-628 for more information on this name-value pair.

Output Arguments

`f` — Duration of negative-going transition
vector

Duration of negative-going transition, returned as a vector. If you specify the sample rate `fs` or the sample instants `t` fall times are in seconds. If you do not specify a sample rate or sample instants, fall times are in samples.

`lt` — Lower reference-level crossing instants
vector

Lower reference-level crossing instants, returned as a vector. The vector `lt` contains the time instants when the 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` — Upper reference-level crossing instants
vector

Upper reference-level crossing instants, returned as a vector. The vector `ut` contains the time instants when the 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
real numeric scalar

Lower reference level in waveform amplitude units, returned as a real numeric scalar. `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.

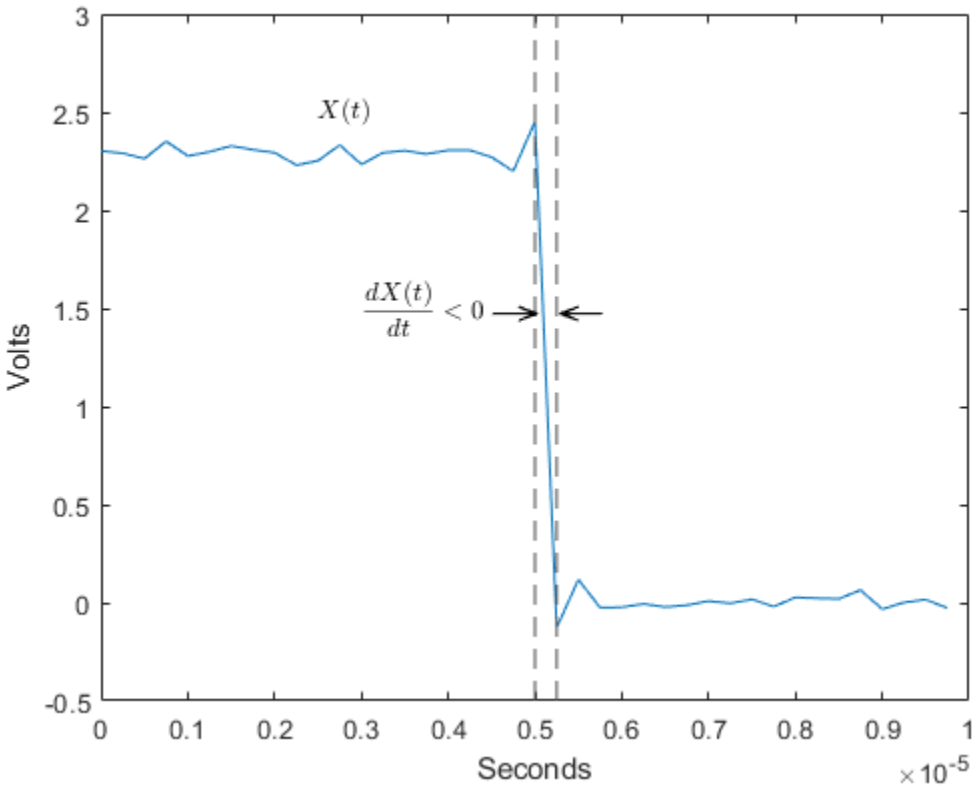
`ul` — Upper reference level
real numeric scalar

Upper reference level in waveform amplitude units, returned as a real numeric scalar. `ul` 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.

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. This 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

$$S_1 + \frac{L}{100}(S_2 - S_1)\,.$$

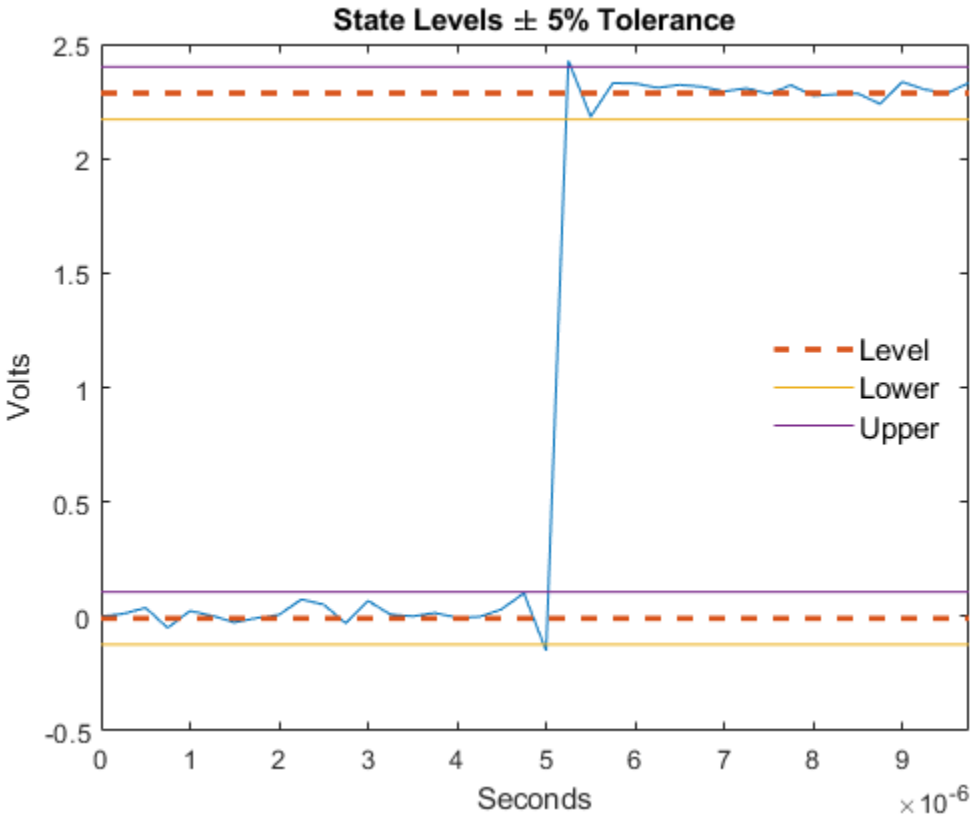
State-Level Tolerances

You can specify lower- and upper-state boundaries for each state level. Define the boundaries as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, specify the scalar as a small number such as 2/100 or 3/100. In general, the $\alpha\%$ 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.

This figure shows lower and upper 5% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The thick dashed lines indicate the estimated state levels.



References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`risetime` | `slewrate` | `statelevels`

Introduced in R2012a

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

`y = fftfilt(b,x)` filters the data in vector `x` with the filter described by coefficient vector `b`.

`y = fftfilt(b,x,n)` uses `n` to determine the length of the FFT.

`y = fftfilt(d,x)` filters the data in vector `x` with a `digitalFilter` object `d`.

`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` stored in `gpuArrayb`.

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. Repeat the experiment 100 times to improve the statistics.

```
rng default

N = 100;

shrt = 20;
long = 2000;

tfs = 0;
tls = 0;
tfl = 0;
tll = 0;

for kj = 1:N

    x = rand(1,1e6);

    bshrt = rand(1,shrt);
```

```
    tic
    sfs = fftfilt(bshrt,x);
    tfs = tfs+toc/N;

    tic
    sls = filter(bshrt,1,x);
    tls = tls+toc/N;

    blong = rand(1,long);

    tic
    sfl = fftfilt(blong,x);
    tfl = tfl+toc/N;

    tic
    sll = filter(blong,1,x);
    tll = tll+toc/N;

end
```

Compare and display the average times.

```
fprintf('%4d-tap filter averages: fftfilt: %f s; filter: %f s\n',shrt,tfs,tls)
```

```
  20-tap filter averages: fftfilt: 0.218688 s; filter: 0.006076 s
```

```
fprintf('%4d-tap filter averages: fftfilt: %f s; filter: %f s\n',long,tfl,tll)
```

```
2000-tap filter averages: fftfilt: 0.056758 s; filter: 0.083478 s
```

Overlap-Add Filtering on the GPU

This example requires Parallel Computing Toolbox™ software. Refer to “GPU Support by Release” (Parallel Computing Toolbox) for a list of supported GPUs.

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.

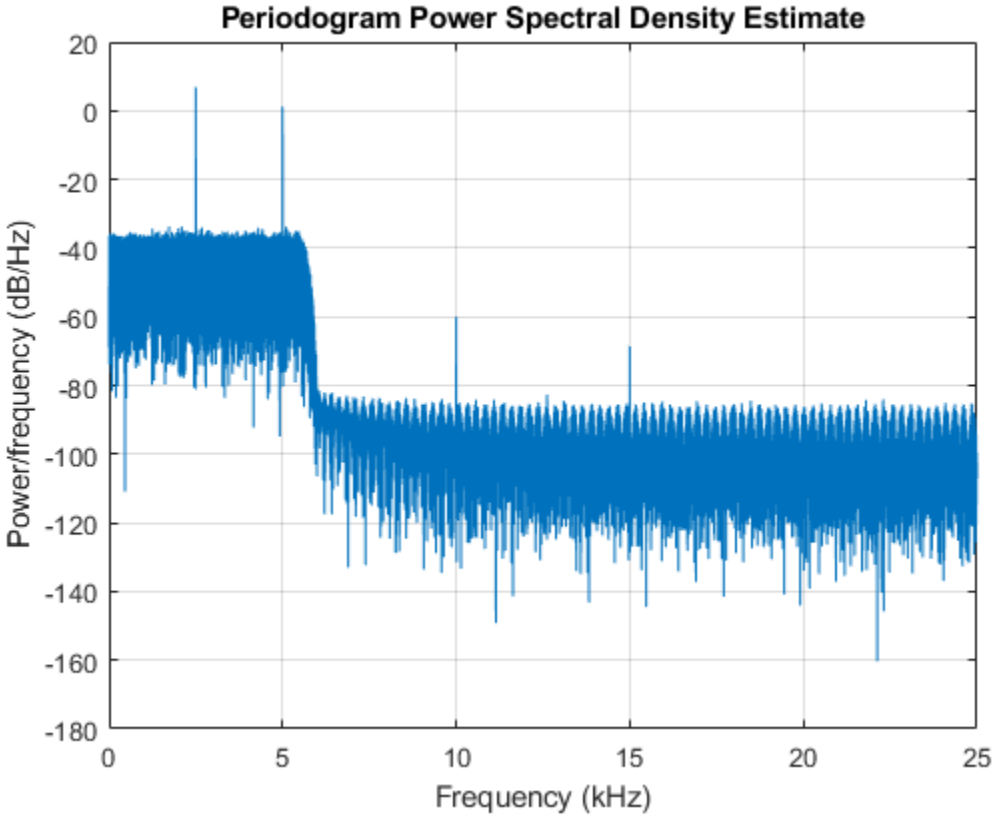
```
Fs = 50e3;
t = 0:1/Fs:10-(1/Fs);
x = cos(2*pi*2500*t) + 0.5*sin(2*pi*5000*t) + 0.25*cos(2*pi*10000*t)+ ...
    0.125*sin(2*pi*15000*t) + randn(size(t));
```

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)
```



Input Arguments

b — Filter coefficients

vector | matrix

Filter coefficients, specified as a vector. If `b` is a matrix, `fftfilt` applies the filter in each column of `b` to the signal vector `x`.

x — Input data

vector | matrix

Input data, specified as a vector. If `x` is a matrix, `fftfilt` filters its columns. 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`. `fftfilt` works for both real and complex inputs.

n — FFT length

positive integer

FFT length, specified as a positive integer. By default, `fftfilt` chooses an FFT length and a data block length that guarantee efficient execution time.

d — Digital filter
`digitalFilter` object

Digital filter, specified as a `digitalFilter` object. Use `designfilt` to generate `d` based on frequency-response specifications.

gpuArrayb, gpuArrayX — GPU arrays
`gpuArray`

GPU arrays, specified as a `gpuArray` object. `gpuArrayb` contains the filter coefficients, and `gpuArrayX` is the input data. See “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox) for details on `gpuArray` objects. Using `fftfilt` with `gpuArray` objects requires Parallel Computing Toolbox software. Refer to “GPU Support by Release” (Parallel Computing Toolbox) for a list of supported GPUs. The filtered data, `y`, is a `gpuArray` object. See “Overlap-Add Filtering on the GPU” on page 1-631 for an example of overlap-add filtering on the GPU.

Output Arguments

y — Output data
vector | matrix | `gpuArray`

Output data, returned as a vector, matrix or `gpuArray` object.

More About

Comparison to `filter` function

When the input signal is relatively large, `fftfilt` is faster than `filter`.

`filter` 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 $\frac{1}{2}L\log_2 L$ where L is the block length. It then performs L point-wise multiplications for a total cost of $L + L\log_2 L = L(1 + \log_2 L)$ multiplications. The cost ratio is therefore $L(1 + \log_2 L)/(NL) = (1 + \log_2 L)/N$ which is approximately $\log_2 L$ / N.

Therefore, `fftfilt` is faster when $\log_2 L$ is less than N.

Algorithms

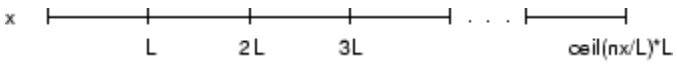
`fftfilt` filters data using the efficient FFT-based method of *overlap-add* [1], a frequency domain filtering technique that works only for FIR filters by combining successive frequency domain filtered blocks of an input sequence. The operation performed by `fftfilt` is described in the time domain by the difference equation:

$$y(n) = b(1)x(n) + b(2)x(n-1) + \cdots + b(nb+1)x(n-nb)$$

An equivalent representation is the Z-transform or frequency domain description:

$$Y(z) = \left(b(1) + b(2)z^{-1} + \cdots + b(nb+1)z^{-nb}\right)X(z)$$

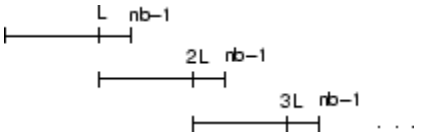
`fftfilt` uses `fft` to implement the overlap-add method. `fftfilt` breaks an input sequence `x` into length L data blocks, where L must be greater than the filter length N.



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` for the filter and signal. If you do not specify a value for `n` (which determines FFT length), `fftfilt` chooses these key parameters automatically:

- If `length(x)` is greater than `length(b)`, `fftfilt` chooses values that minimize the number of blocks times the number of flops per FFT.
- If `length(b)` is greater than or equal to `length(x)`, `fftfilt` uses a single FFT of length

  ```
  2^nextpow2(length(b) + length(x) - 1)
  ```

  This computes

  ```
  y = ifft(fft(B,nfft).*fft(X,nfft))
  ```

If you supply a value for `n`, `fftfilt` chooses an FFT length, `nfft`, of `2^nextpow2(n)` and a data block length of `nfft - length(b) + 1`. If `n` is less than `length(b)`, `fftfilt` sets `n` to `length(b)`.

References

[1] Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

`digitalFilter` objects are not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

`digitalFilter` objects are not supported for code generation.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`conv` | `designfilt` | `digitalFilter` | `filter` | `filtfilt`

Introduced before R2006a

fillgaps

Fill gaps using autoregressive modeling

Syntax

```
y = fillgaps(x)
y = fillgaps(x,maxlen)
y = fillgaps(x,maxlen,order)

fillgaps( ___ )
```

Description

`y = fillgaps(x)` replaces any `NaN`s present in a signal `x` with estimates extrapolated from forward and reverse autoregressive fits of the remaining samples. If `x` is a matrix, then the function treats each column as an independent channel.

`y = fillgaps(x,maxlen)` specifies the maximum number of samples to use in the estimation. Use this argument when your signal is not well characterized throughout its range by a single autoregressive process.

`y = fillgaps(x,maxlen,order)` specifies the order of the autoregressive model used to reconstruct the gaps.

`fillgaps( ___ )` with no output arguments plots the original samples and the reconstructed signal. This syntax accepts any input arguments from previous syntaxes.

Examples

Fill Gaps in Audio File

Load a speech signal sampled at F_s = 7418 Hz. The file contains a recording of a female voice saying the word "MATLAB®." Play the sound.

```
load mtlb

% To hear, type soundsc(mtlb,Fs)
```

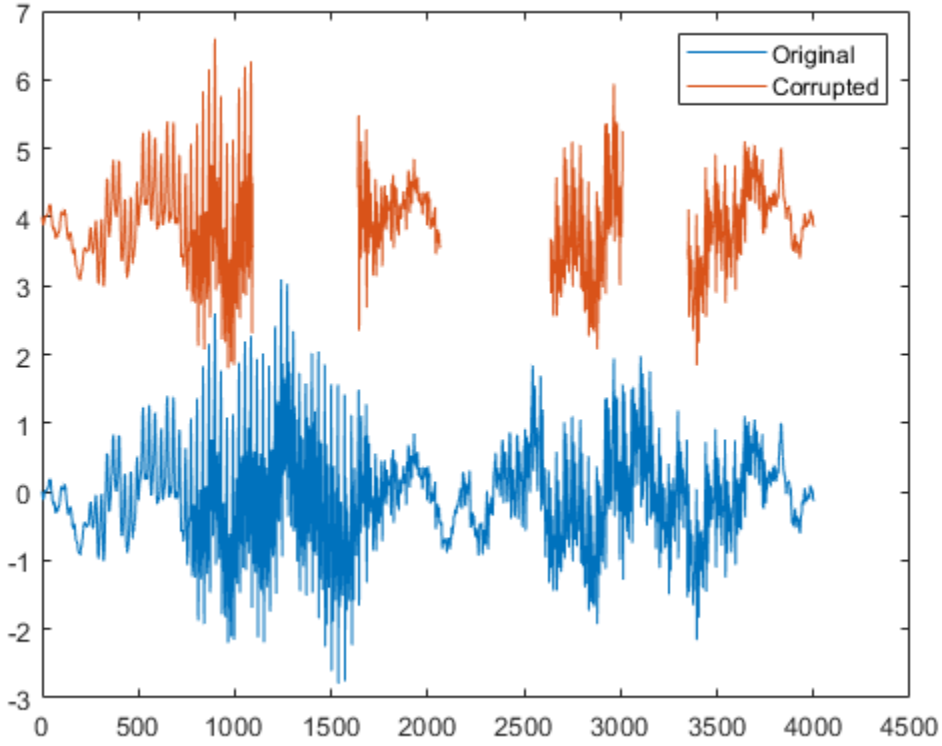
Simulate a situation in which a noisy transmission channel corrupts parts of the signal irretrievably. Introduce gaps of random length roughly every 500 samples. Reset the random number generator for reproducible results.

```
rng default

gn = 3;

mt = mtlb;

gl = randi([300 600],gn,1);
```

```
for kj = 1:gn
    mt(kj*1000+randi(100)+(1:gl(kj))) = NaN;
end
```

Plot the original and corrupted signals. Offset the corrupted signal for ease of display. Play the signal with the gaps.

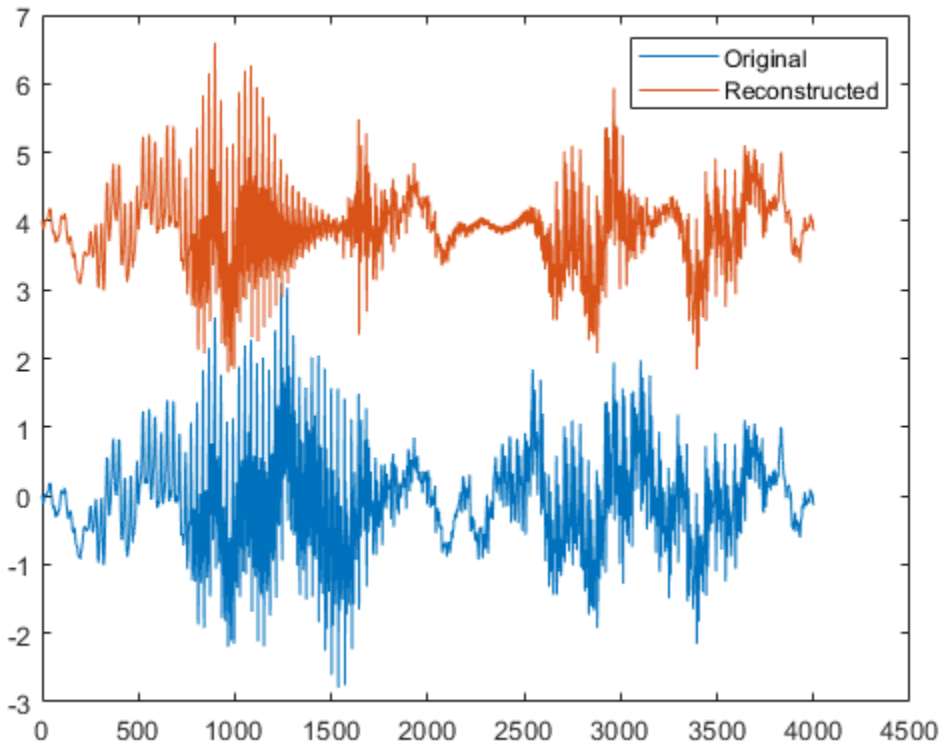
```
plot([mtlb mt+4])
legend('Original','Corrupted')
```




```
% To hear, type soundsc(mt,Fs)
```

Reconstruct the signal using an autoregressive process. Use `fillgaps` with the default settings. Plot the original and reconstructed signals, again using an offset. Play the reconstructed signal.

```
lb = fillgaps(mt);

plot([mtlb lb+4])
legend('Original','Reconstructed')
```




```
% To hear, type soundsc(lb,Fs)
```

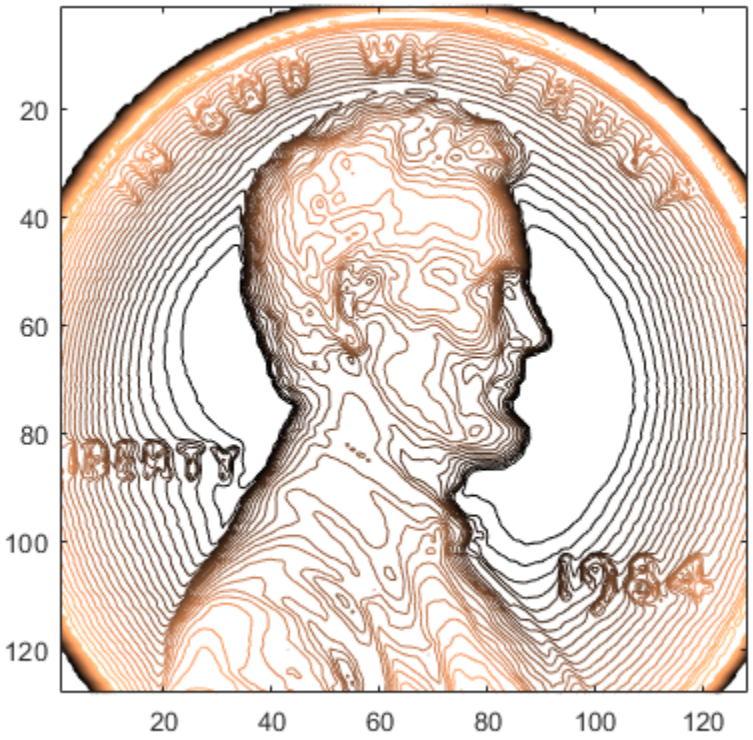
Fill Gaps in Two-Dimensional Data

Load a file that contains depth measurements of a mold used to mint a United States penny. The data, taken at the National Institute of Standards and Technology, are sampled on a 128-by-128 grid.

```
load penny
```

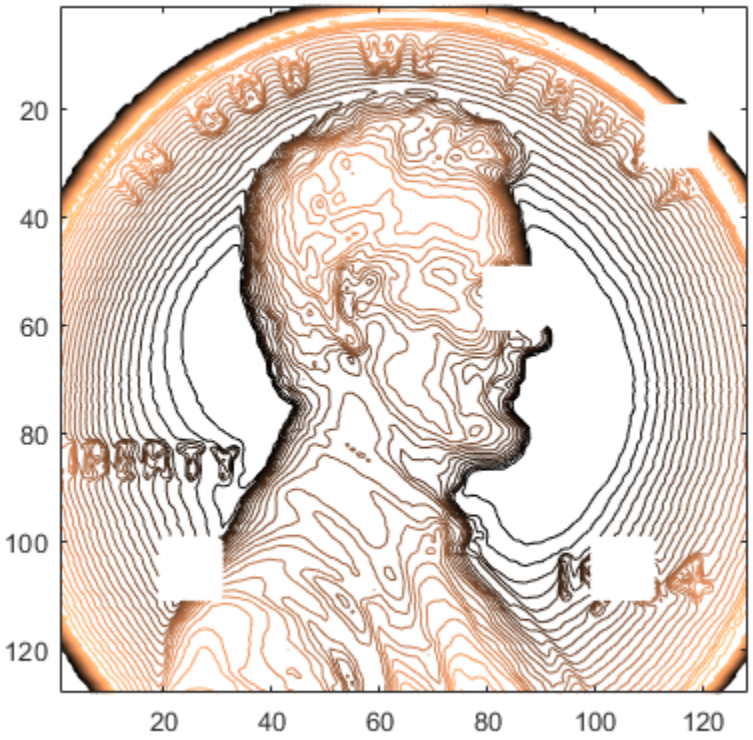
Draw a contour plot with 25 copper-colored contour lines.

```
nc = 25;

contour(P,nc)
colormap copper
axis ij square
```



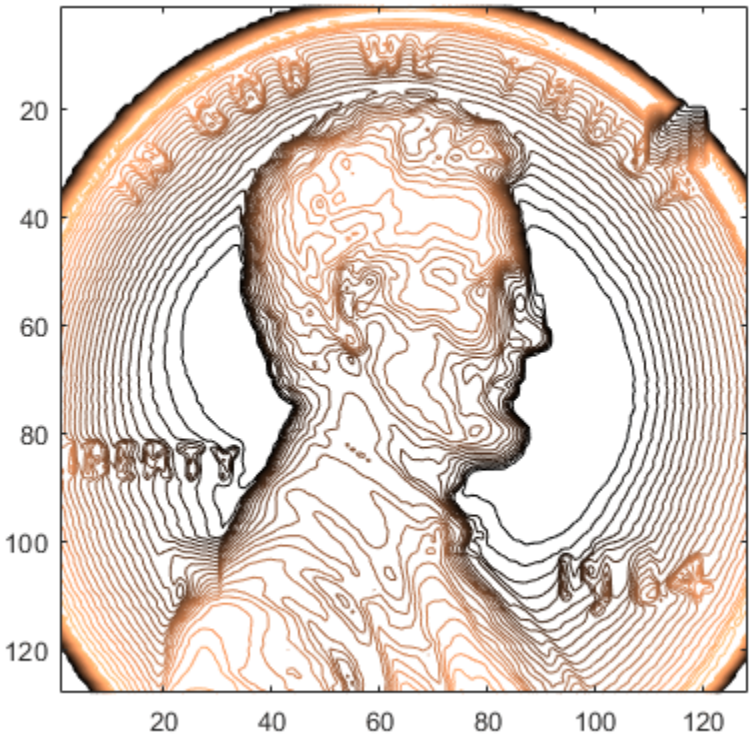
Introduce four 10-by-10 gaps into the data. Draw a contour plot of the corrupted signal.

```
P(50:60,80:90) = NaN;
P(100:110,20:30) = NaN;
P(100:110,100:110) = NaN;
P(20:30,110:120) = NaN;

contour(P,nc)
colormap copper
axis ij square
```



Use `fillgaps` to reconstruct the data, treating each column as an independent channel. Specify an 8th-order autoregressive model extrapolated from 30 samples at each end. Draw a contour plot of the reconstruction.

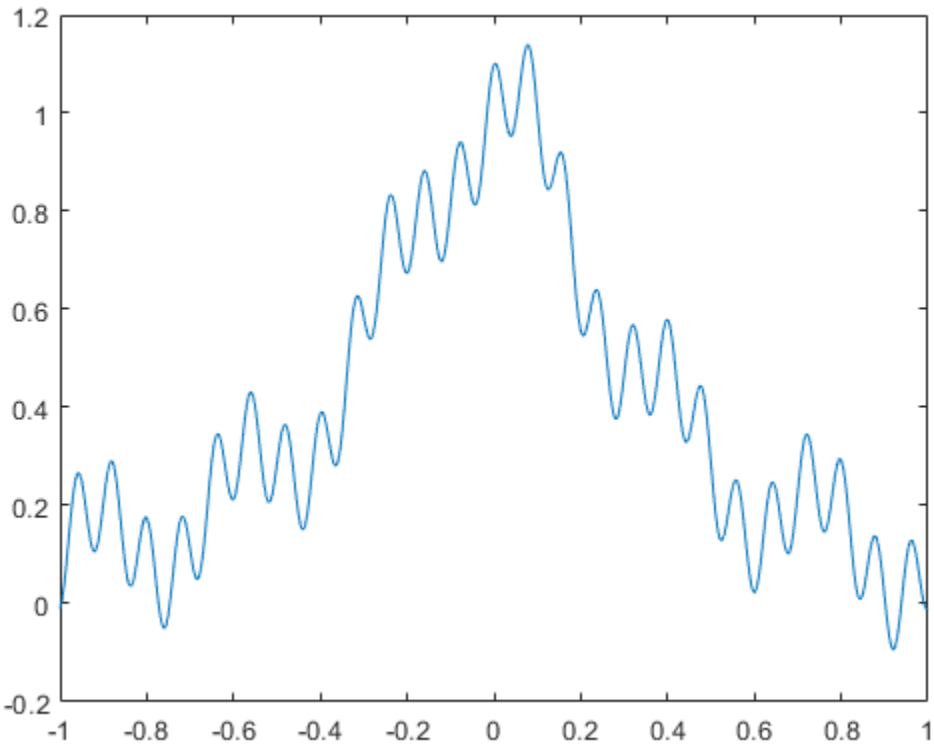
```
q = fillgaps(P,30,8);

contour(q,nc)
colormap copper
axis ij square
```



Fill Gaps in Function

Generate a function that consists of the sum of two sinusoids and a Lorentzian curve. The function is sampled at 200 Hz for 2 seconds. Plot the result.

```
x = -1:0.005:1;

f = 1./(1+10*x.^2)+sin(2*pi*3*x)/10+cos(25*pi*x)/10;

plot(x,f)
```

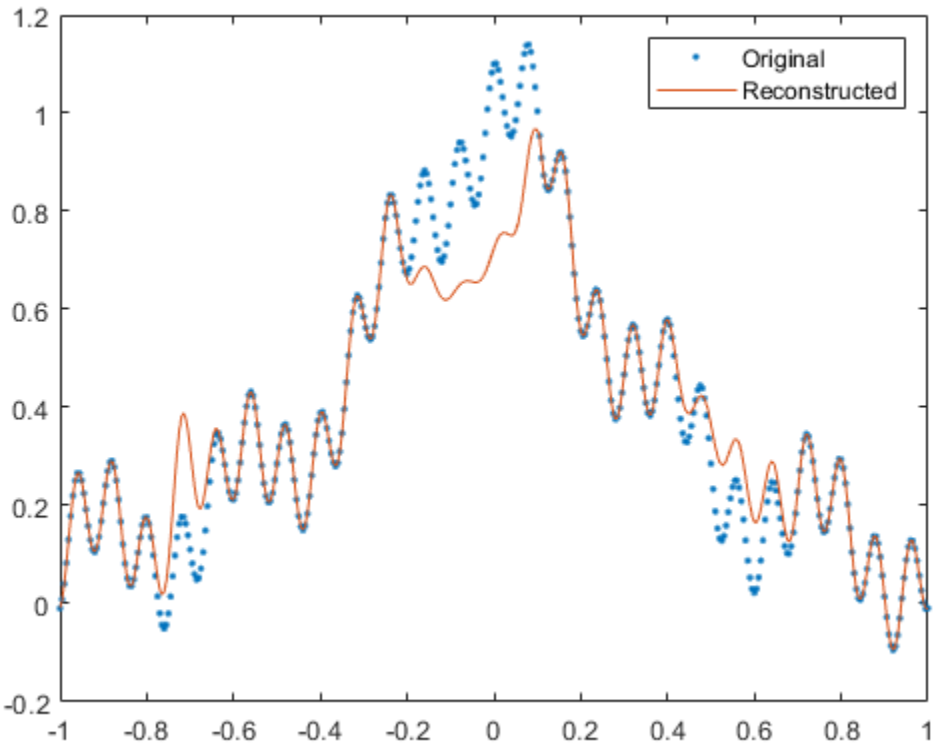


Insert gaps at intervals (-0.8,-0.6), (-0.2,0.1), and (0.4,0.7).

```
h = f;

h(x>-0.8 & x<-0.6) = NaN;
h(x>-0.2 & x< 0.1) = NaN;
h(x> 0.4 & x< 0.7) = NaN;
```

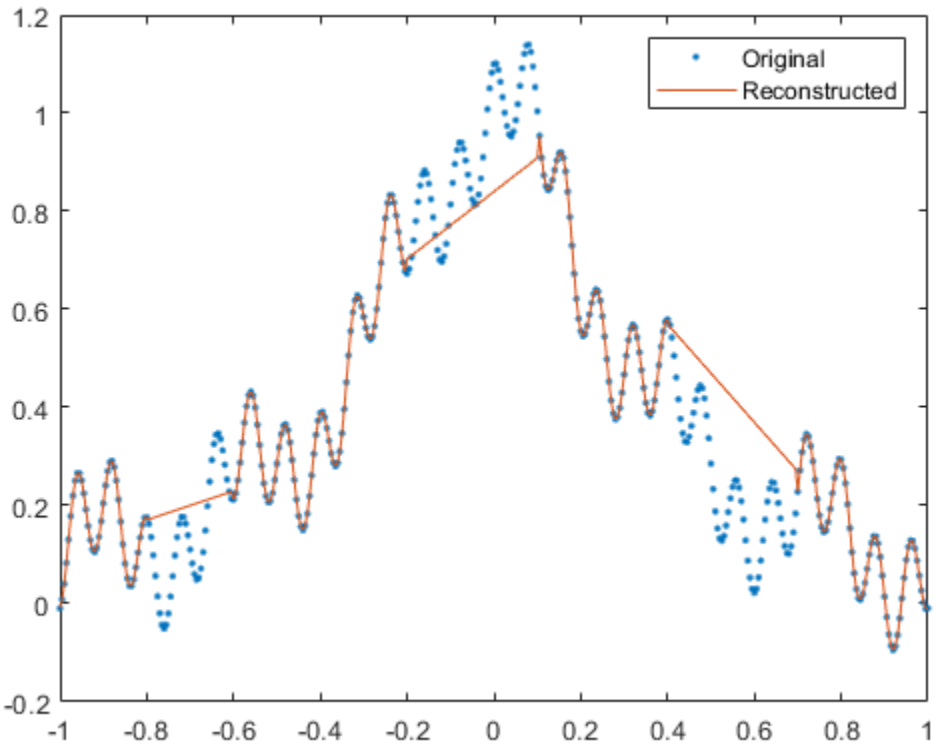
Fill the gaps using the default settings of `fillgaps`. Plot the original and reconstructed functions.

```
y = fillgaps(h);

plot(x,f,'.',x,y)
legend('Original','Reconstructed')
```



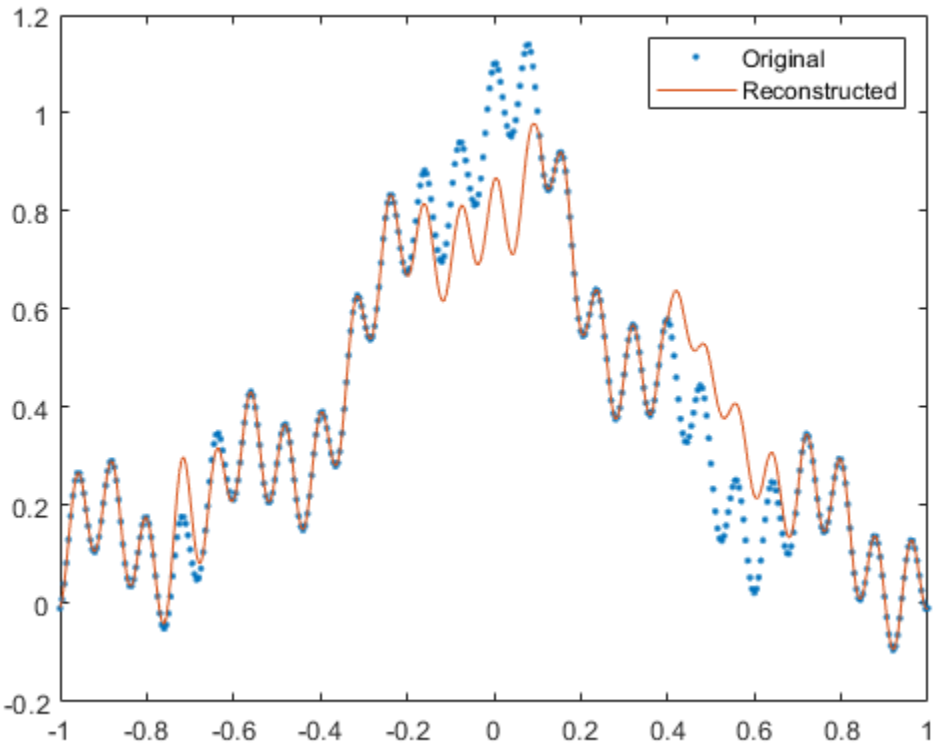
Repeat the computation, but now specify a maximum prediction-sequence length of 3 samples and a model order of 1. Plot the original and reconstructed functions. At its simplest, `fillgaps` performs a linear fit.

```
y = fillgaps(h,3,1);

plot(x,f,'.',x,y)
legend('Original','Reconstructed')
```



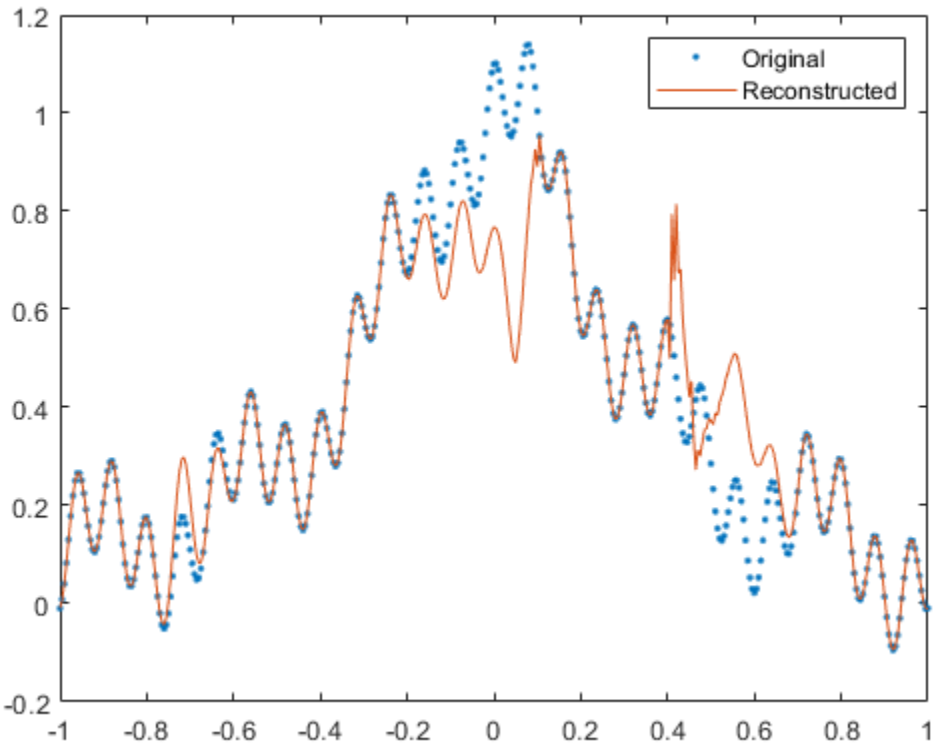
Specify a maximum prediction-sequence length of 80 samples and a model order of 40. Plot the original and reconstructed functions.

```
y = fillgaps(h,80,40);

plot(x,f,'.',x,y)
legend('Original','Reconstructed')
```



Change the model order to 70. Plot the original and reconstructed functions.

```
y = fillgaps(h,80,70);

plot(x,f,'.',x,y)
legend('Original','Reconstructed')
```



The reconstruction is imperfect because very high model orders often have problems with finite precision.

Fill Gaps in Chirp

Generate a multichannel signal consisting of two instances of a chirp sampled at 1 kHz for 1 second. The frequency of the chirp is zero at 0.3 seconds and increases linearly to reach a final value of 40 Hz. Each instance has a different DC value.

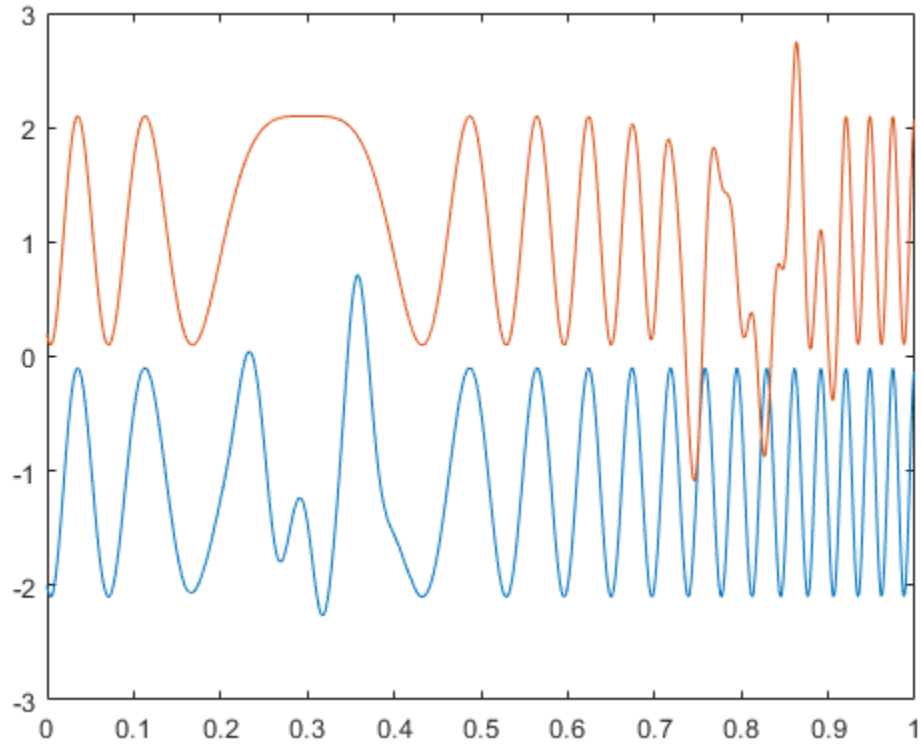
```
Fs = 1000;
t = 0:1/Fs:1-1/Fs;
r = chirp(t-0.3,0,0.7,40);
f = 1.1;
q = [r-f;r+f]';
```

Introduce gaps to the signal. One of the gaps covers the low-frequency region, and the other covers the high-frequency region.

```
gap = (460:720);
q(gap-300,1) = NaN;
q(gap+200,2) = NaN;
```

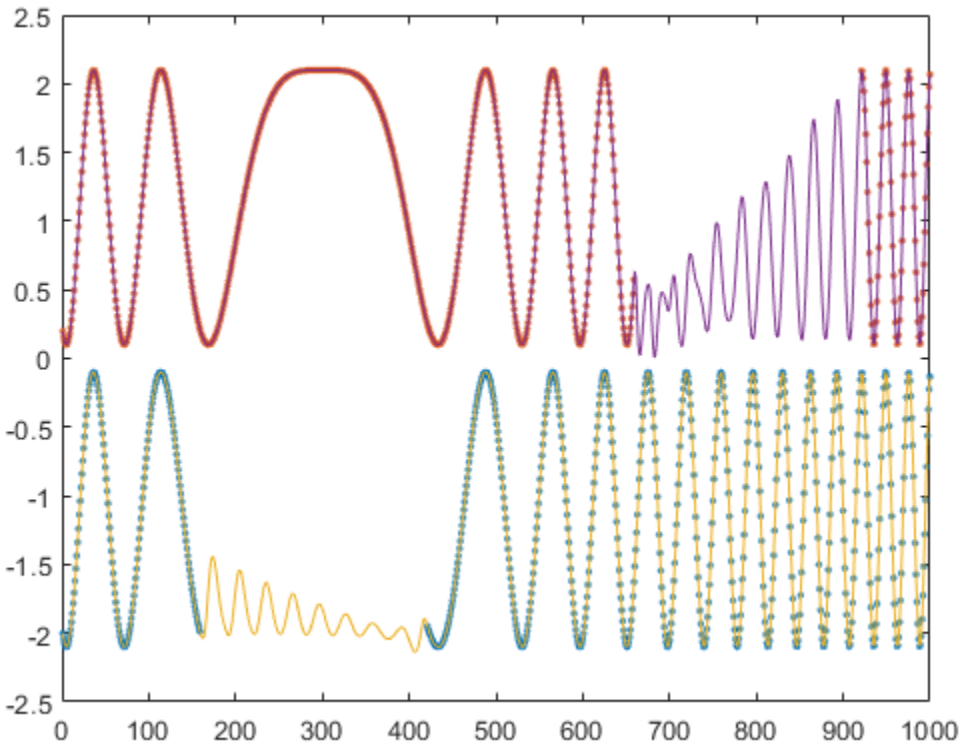
Fill the gaps using the default parameters. Plot the reconstructed signals.

```
y = fillgaps(q);

plot(t,y)
```



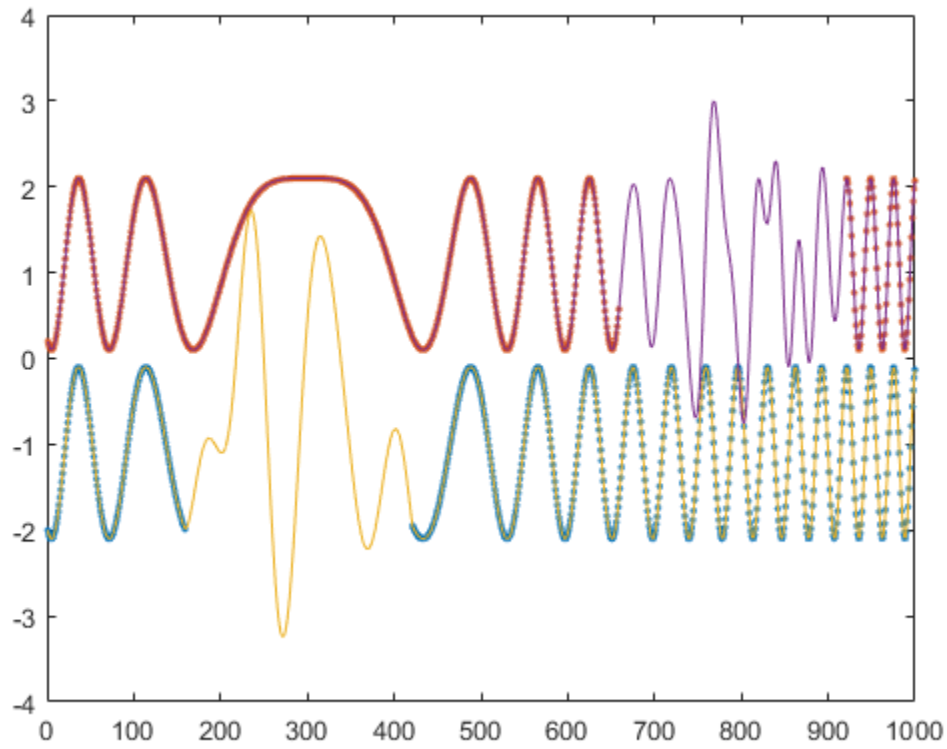
Fill the gaps by fitting 14th-order autoregressive models to the signal. Limit the models to incorporate 15 samples on the end of each gap. Use the functionality of `fillgaps` to plot the reconstructions.

```
fillgaps(q,15,14)
```



Increase the number of samples to use in the estimation to 150. Increase the model order to 140.

```
fillgaps(q,150,140)
```



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` contains `NaN`s to represent missing samples.

Example: `cos(pi/4*(0:159))+reshape(ones(32,1)*[0 NaN 0 NaN 0],1,160)` is a single-channel row-vector signal missing 40% of its samples.

Example: `cos(pi./[4;2]*(0:159))'+reshape(ones(64,1)*[0 NaN 0 NaN 0],160,2)` is a two-channel signal with large gaps.

Data Types: `single` | `double`

maxlen — Maximum length of prediction sequences

positive integer

Maximum length of prediction sequences, specified as a positive integer. If you leave `maxlen` unspecified, then `fillgaps` iteratively fits autoregressive models using all previous points for forward estimation and all future points for backward estimation.

Data Types: `single` | `double`

order — Autoregressive model order

`'aic'` (default) | positive integer

Autoregressive model order, specified as `'aic'` or a positive integer. The order is truncated when `order` is infinite or when there are not enough available samples. If you specify `order` as `'aic'`, or leave it unspecified, then `fillgaps` selects the order that minimizes the Akaike information criterion.

Data Types: `single` | `double` | `char` | `string`

Output Arguments

y — Reconstructed signal

vector | matrix

Reconstructed signal, returned as a vector or matrix.

References

[1] Akaike, Hirotugu. "Fitting Autoregressive Models for Prediction." *Annals of the Institute of Statistical Mathematics*. Vol. 21, 1969, pp. 243–247.

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

[3] Orfanidis, Sophocles J. *Optimum Signal Processing: An Introduction*. 2nd Edition. New York: McGraw-Hill, 1996.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The size of `order` must be a compile-time constant.

See Also

`arburg` | `resample`

Introduced in R2016a

filterBuilder

Interactive filter design

Syntax

```
filterBuilder(h)
filterBuilder('response')
```

Description

`filterBuilder` starts a interactive 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.

For more information on how to use `filterBuilder`, see “Filter Builder 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 using `fdesign`.

  Note You must have the DSP System Toolbox software to create and import filter System objects.
- Enter `filterBuilder('response')` to replace *`response`* with a response method from the following table. MATLAB opens a filter design dialog that corresponds to the specified response.

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 Method	Description of Resulting Filter Design	Filter Object
arbgrpdelay on page 1-662	Arbitrary group delay filter design	`fdesign.arbgrpdelay`

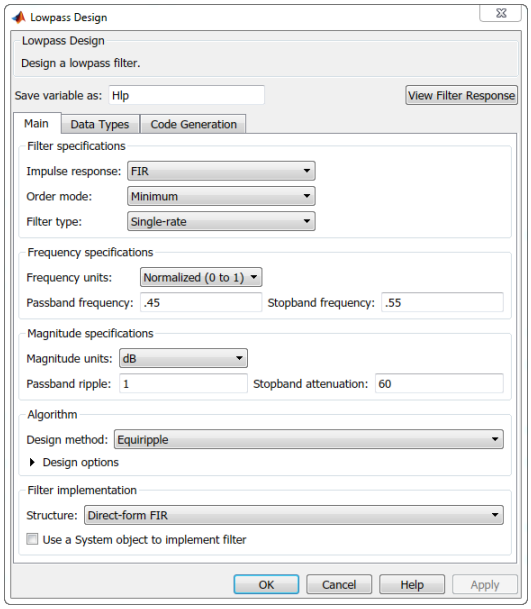
Response Method	Description of Resulting Filter Design	Filter Object
arbmag on page 1-662	Arbitrary magnitude filter design	`fdesign.arbmag`
arbmagnphase on page 1-662	Arbitrary response filter (magnitude and phase)	`fdesign.arbmagnphase`
audioweighting on page 1-665	Audio weighting filter	`fdesign.audioweighting`
bandpass on page 1-666 or `bp`	Bandpass filter	`fdesign.bandpass`
bandstop on page 1-670 or `bs`	Bandstop filter	`fdesign.bandstop`
cic on page 1-675	CIC filter	`fdesign.decimator(M,'cic',...)` or `fdesign.interpolator(L,'cic',...)` See `fdesign.decimator` and `fdesign.interpolator`
ciccomp on page 1-676	CIC compensator	`fdesign.ciccomp`
comb on page 1-679	Comb filter	`fdesign.comb`
diff on page 1-681	Differentiator filter	`fdesign.differentiator`
fracdelay on page 1-684	Fractional delay filter	`fdesign.fracdelay`
halfband on page 1-684 or `hb`	Halfband filter	`fdesign.halfband`
highpass on page 1-687 or `hp`	Highpass filter	`fdesign.highpass`
hilb on page 1-691	Hilbert filter	`fdesign.hilbert`
isinc on page 1-694, isinclp on page 1-694, or isinchp on page 1-694	Inverse sinc lowpass or highpass filter	`fdesign.isinclp` and `fdesign.isinchp`
lowpass on page 1-698 or `lp`	Lowpass filter (default)	`fdesign.lowpass`
notch on page 1-702	Notch filter	`fdesign.notch`
nyquist on page 1-702	Nyquist filter	`fdesign.nyquist`
octave on page 1-706	Octave filter	`fdesign.octave`
parameq on page 1-707	Parametric equalizer filter	`fdesign.parameq`
peak on page 1-710	Peak filter	`fdesign.peak`

Note Because they do not change the filter structure, the magnitude specifications and design method are tunable when using `filterBuilder`.

Filter Builder Design Panes

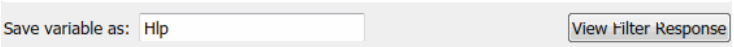
Main Design Pane

The main pane of Filter Builder 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.



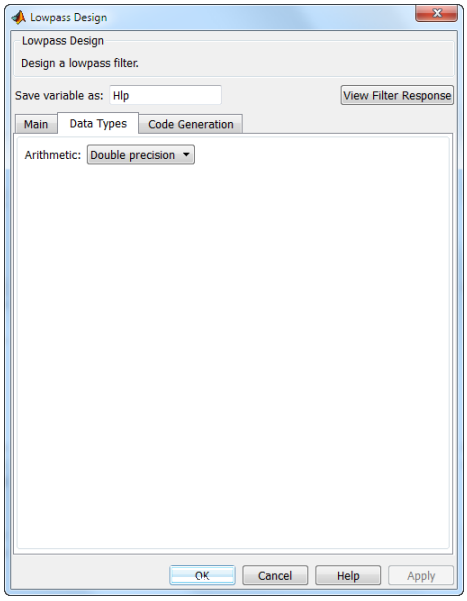
- **Save variable as** — When you click **Apply** to apply your changes or **OK** to close this dialog box, `filterBuilder` saves the current filter to your MATLAB workspace as a filter object with the name you enter.
- **View Filter Response** — Displays the magnitude response for the current filter specifications and design method by opening the Filter Visualization Tool (**FVTool**). If you specify the input sample rate, then **FVTool** plots the magnitude response as a function of frequency in Hz. If you do not specify the input sample rate, then **FVTool** plots the magnitude response as a function of normalized frequency in rad/sample.

Note The `filterBuilder` dialog box includes an **Apply** option. Each time you click **Apply**, `filterBuilder` writes the modified filter to your MATLAB workspace. This modified filter has the variable name you assign in **Save variable as**. To apply changes without overwriting the variable in you workspace, change the variable name in **Save variable as** before you click **Apply**.

There are three tabs in the Filter Builder 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

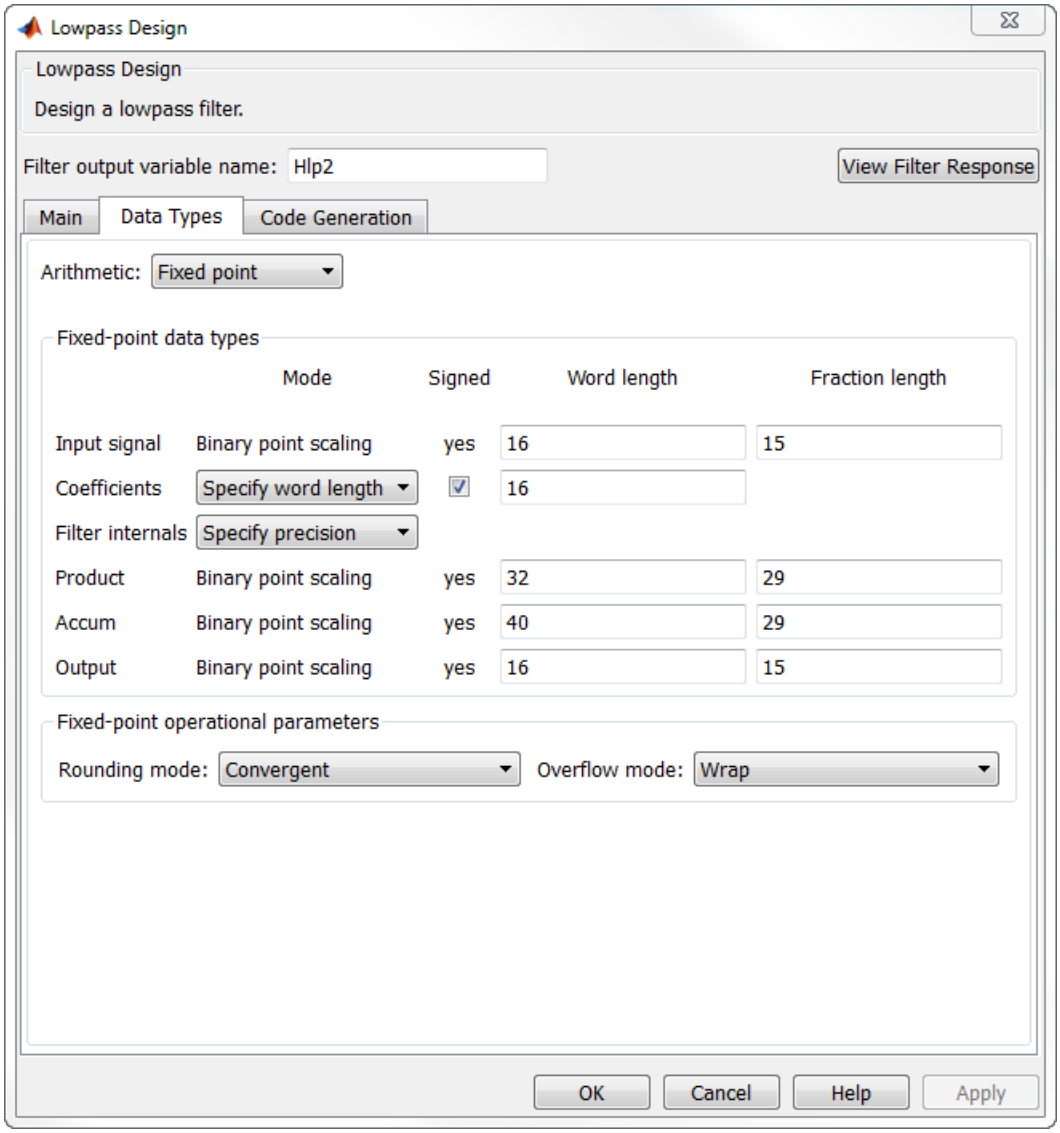
The second tab in the Filter Builder dialog box is shown in the following figure.



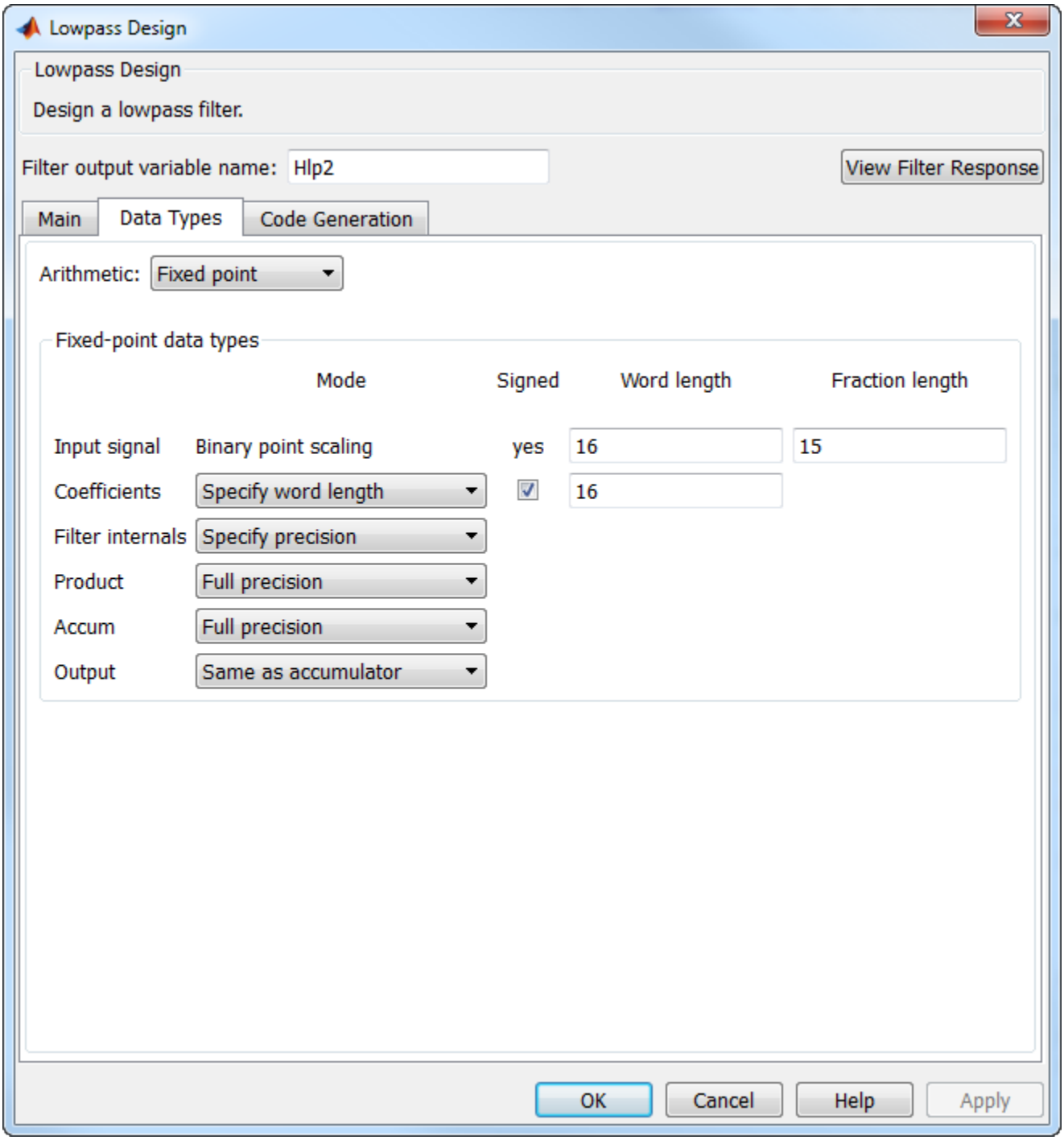
The **Arithmetic** drop down box allows the choice of `Double precision`, `Single precision`, or `Fixed point`. Some of these options may be unavailable depending on the filter parameters. The following table describes these options.

Arithmetic List Entry	Effect on the Filter
`Double precision`	All filtering operations and coefficients use double-precision, floating-point representations and math. When you use `filterBuilder` to create a filter, `double precision` is the default value for the Arithmetic property.
`Single precision`	All filtering operations and coefficients use single-precision floating-point representations and math.
`Fixed point`	This entry 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.



When you select **Use a System object to implement filter** check box in the **Main** pane, the **Data Types** pane appears as below:



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.

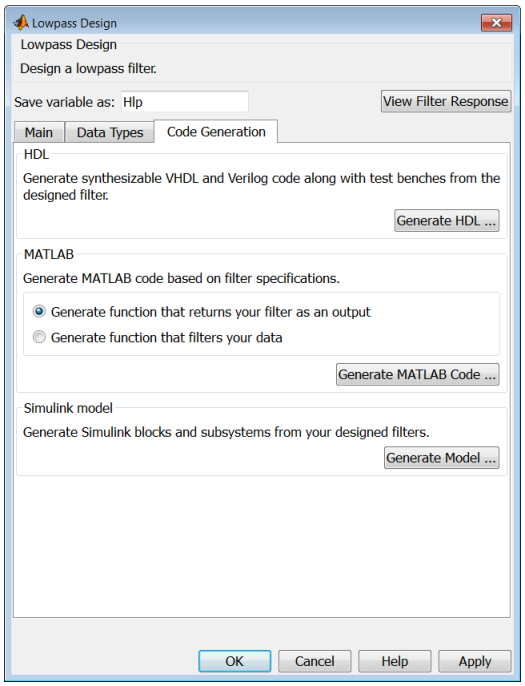
The effect of clearing or selecting **Cast before sum** is as follows:

- 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 UI from the Filter Builder” (Filter Design HDL Coder).

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 a filter object using `fdesign`.

- **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. The data type of the filter output is set according to the data type settings in the **Data Types** pane.

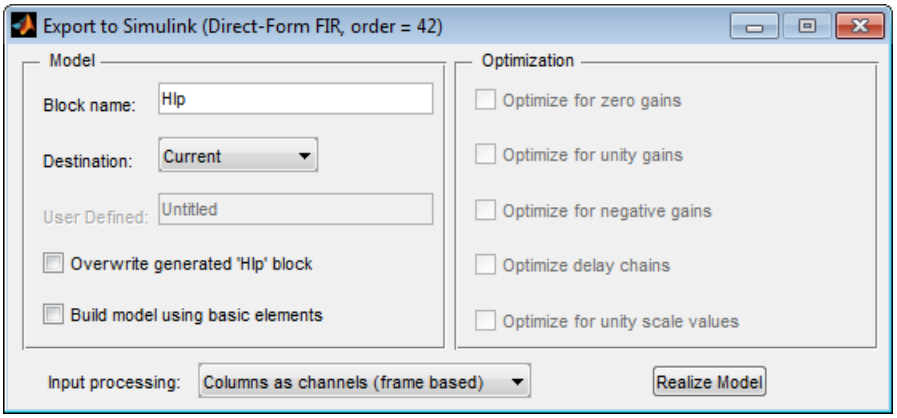
Clicking on the **Generate MATLAB code** button, brings up a Save File dialog. Specify the file name and location, and save. The filter is now contained in an editable file.

Simulink Model

Generate Simulink blocks and subsystems from your designed filters

When you click **Generate Model**, the filter builder generates Simulink blocks and subsystems from your designed filters.

Clicking on the **Generate Model** button opens the Export to Simulink 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 at the location specified in `User Defined`.
- **Overwrite generated 'Filter' block** — Overwrites an existing block with the name specified in **Block Name**. Clear this check box to create a new block with the same name.
- **Build model using basic elements** — Builds the model using only basic blocks.
- **Optimize for zero gains** — Removes all zero-gain blocks from the model.
- **Optimize for unity gains** — Replaces all unity gains with direct connections.
- **Optimize for negative gains** — Removes all negative unity gain blocks, and changes sign at the nearest summation block.
- **Optimize delay chains** — Replaces delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for unity scale values** — 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)` — The block treats each column of the input as a separate channel.
  - `Elements as channels (sample based)` — The block treats each element of the input as a separate channel.

  For more information about sample-based and frame-based processing, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).
- **Realize Model** — Builds the model with the set parameters.

When the **Use a System object to implement filter** check box is selected in the **Main** pane, the **Generate Model** button in the **Simulink model** panel is disabled under the following conditions:

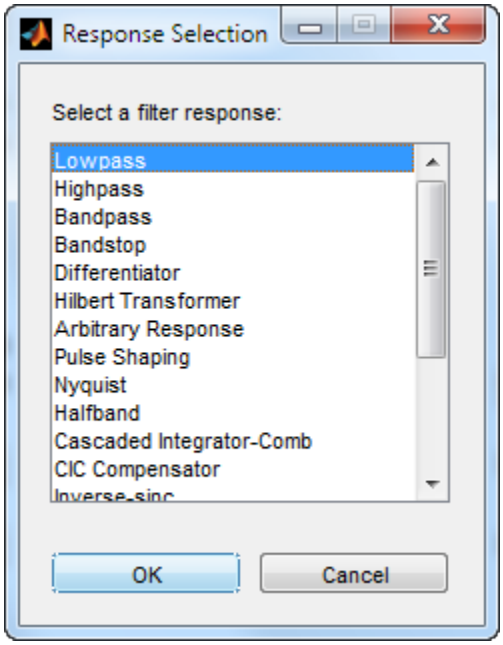
- Select **Filter response** as `Comb` and **Arithmetic** on the **Data Types** pane as `Fixed point`.
- Select **Filter response** as `Arbitrary Response`, **Impulse response** as `IIR`, set **Specify response as** to either `Magnitudes and phases` or `Frequency response`, and **Arithmetic** on the **Data Types** pane as `Fixed point`.

These settings design a `dsp.IIRFilter` System object™ with fixed point arithmetic. Generating a Simulink model for fixed point `dsp.IIRFilter` object is not supported.

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 Filter Design — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

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 sample rate

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

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Audio Weighting Filter Design — 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 sample rate** — The sampling frequency in **Frequency units**. For example, if **Frequency units** is set to `kHz`, setting **Input sample rate** 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, this check box is cleared. When the current design method or structure is not supported by a System object filter, then this check box is disabled.

Bandpass Filter Design — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select `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

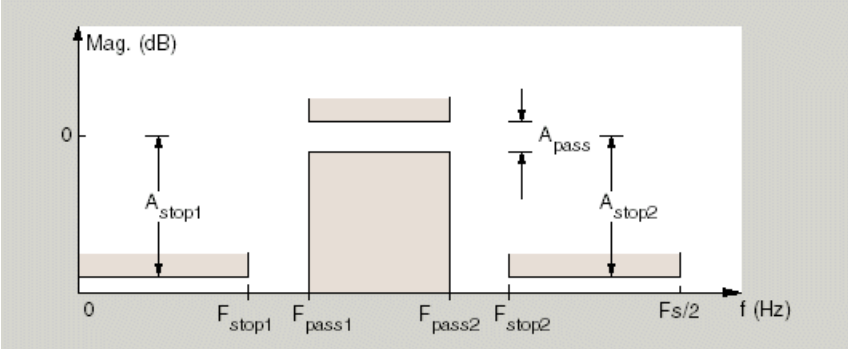
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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 Stopband frequency 1 (Fstop1) and Passband frequency 1 (Fpass1) represent transition regions where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. This dialog applies only when **Order mode** is **Specify**.

- `Passband and stopband frequencies` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband frequency` — Define the filter by specifying frequencies for the edges of the passband.
- `Stopband frequency` — Define the filter by specifying frequencies for the edges of the stopbands.
- `Half power (3dB) frequency` — 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.
- `Half power (3dB) frequencies 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)

- `Half power (3dB) frequencies and stopband width` — Define the filter by specifying frequencies for the 3-dB points in the filter response and the width of the stopband. (IIR filters)
- `Cutoff (6dB) frequency` — Define the filter response by specifying the locations of the 6-dB points. The 6-dB point is the frequency for the point 6 dB 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 to 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 sample rate** parameter.

Input sample rate

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.

Stopband frequency 1

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**.

Passband frequency 1

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**.

Passband frequency 2

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**.

Stopband frequency 2

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: **Stopband attenuation 1**, **Stopband attenuation 2**, and **Passband ripple**. 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.

Stopband attenuation 1

Enter the filter attenuation in the first stopband in the units you choose for **Magnitude units**, either linear or decibels.

Passband ripple

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Stopband attenuation 2

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

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Bandstop Filter Design — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select `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

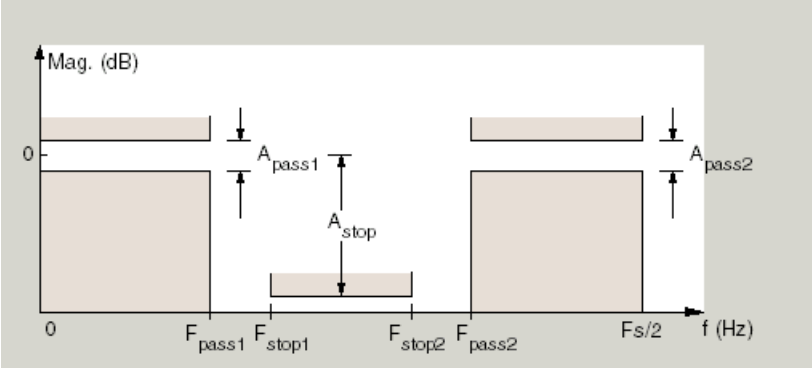
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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 frequencies` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband frequency` — Define the filter by specifying frequencies for the edges of the passband.
- `Stopband frequency` — Define the filter by specifying frequencies for the edges of the stopbands.
- `Half power (3dB) frequency` — 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 point below the passband value.
- `Half power (3dB) frequencies 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).
- `Half power (3dB) frequencies and stopband width` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband (IIR filters).
- `Cutoff (6dB) frequency` — 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 to 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 sample rate** parameter.

Input sample rate

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 sample rate

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`.

Passband frequency 1

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**.

Stopband frequency 1

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**.

Stopband frequency 2

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**.

Passband frequency 2

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: **Passband ripple 1**, **Passband ripple 2**, and **Stopband attenuation**. 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.

Passband ripple 1

Enter the filter ripple allowed in the first passband in the units you choose for **Magnitude units**, either linear or decibels.

Stopband attenuation

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels

Passband ripple 2

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

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

CIC Filter Design — Main Pane

Filter specifications

Parameters in this group enable you to specify your CIC filter format, such as the filter type and the differential delay.

Filter type

Select whether your filter will be a `decimator` or an `interpolator`. Your choice determines the type of filter and the design methods and structures that are available to implement your filter. Selecting `decimator` or `interpolator` activates the **Factor** option. When you design an interpolator, you enable the **Output sample rate** 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 to 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 sample rate** parameter.

Input sample rate

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 sample rate

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.

Passband frequency

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.

Stopband attenuation

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

CIC Compensator Design — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the filter order mode and the filter type.

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 to 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 sample rate** parameter.

Input sample rate

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 sample rate

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.

Passband frequency

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**.

Stopband frequency

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.

Passband ripple

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

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Comb Filter Design —Main Pane

Filter specifications

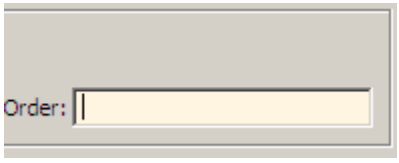
Parameters in this group enable you to specify the type of comb filter and the number of peaks or notches.

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.

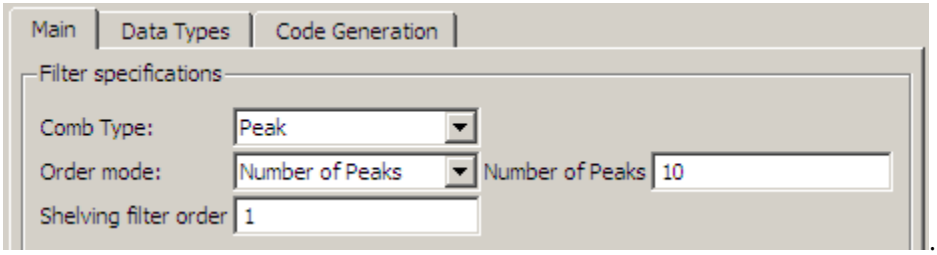
Order mode

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/`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 sample rate** 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.

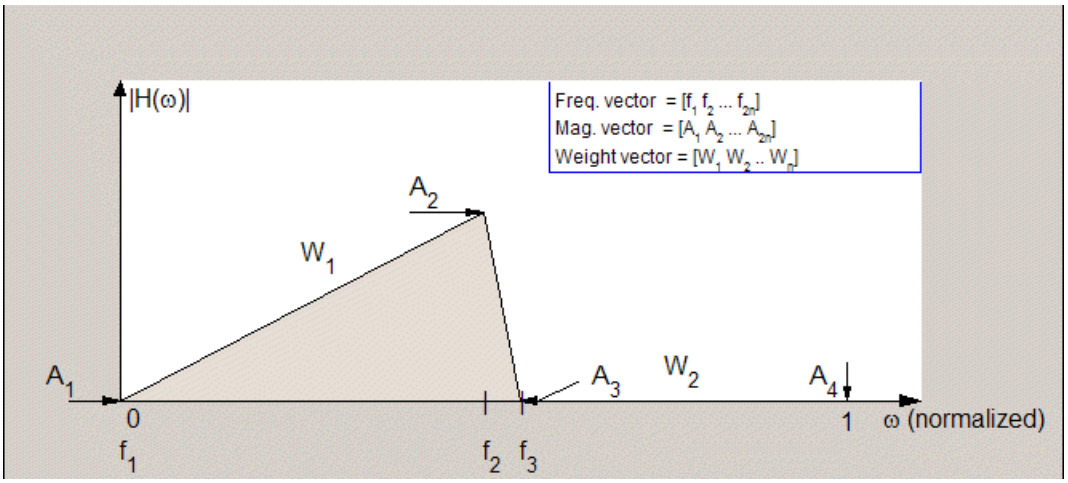
Use a System object to implement filter

Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

Differentiator Filter Design — 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 **Passband frequency** (f_1) and **Stopband frequency** (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 to 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 sample rate** parameter.

Input sample rate

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.

Passband frequency

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**.

Stopband frequency

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.

Passband ripple

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Stopband attenuation 2

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

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Fractional Delay Design — 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 to 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 sample rate** parameter.

Input sample rate

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 — Main Pane

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.

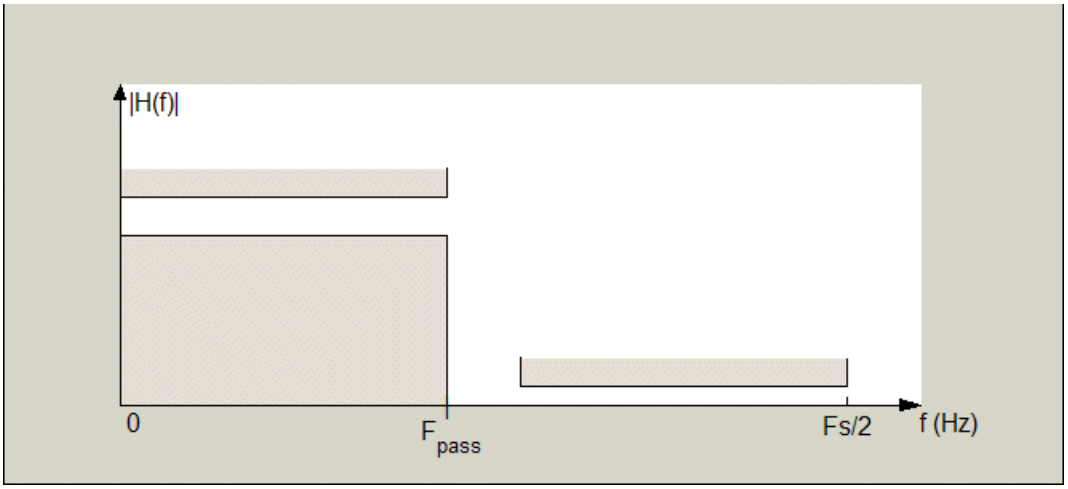
When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that decimates or interpolates your input by a factor of two.

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 to 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 sample rate** parameter.

Input sample rate

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).

Stopband attenuation

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

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to either `Interpolator` or `Decimator`. The filter builder always implements the filter as a System object.

Highpass Filter Design — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select `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

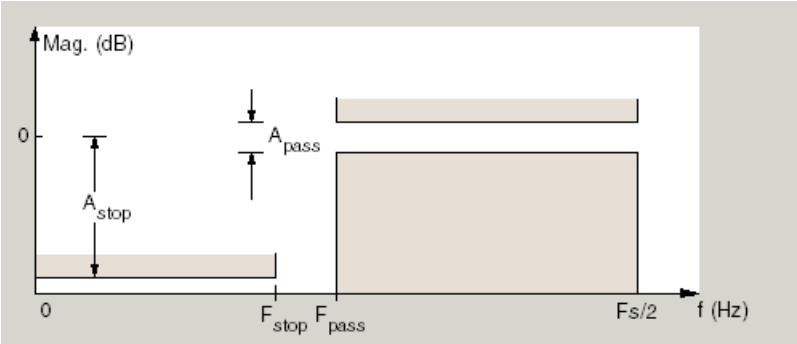
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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 Stopband frequency (Fstop) and Passband frequency (Fpass) represents the transition region where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Stopband edge and passband edge` — Define the filter by specifying the frequencies for the edges for the stopband and passband.
- `Passband frequency` — Define the filter by specifying the frequency for the edge of the passband.
- `Stopband frequency` — Define the filter by specifying the frequency for the edges of the stopband.

- `Stopband and half power (3dB) frequencies` — Define the filter by specifying the stopband edge frequency and the 3-dB down point (IIR designs).
- `Half power (3dB) and passband frequencies` — Define the filter by specifying the 3-dB down point and passband edge frequency (IIR designs).
- `Half power (3dB) frequency` — Define the filter by specifying the frequency for the 3-dB point (IIR designs or maxflat FIR).
- `Cutoff (6dB) frequency` — 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 to 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 sample rate** parameter.

Input sample rate

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.

Passband frequency

Enter the frequency at the of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Stopband frequency

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.

Stopband attenuation

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Passband ripple

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 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.

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

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Hilbert Filter Design — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select `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

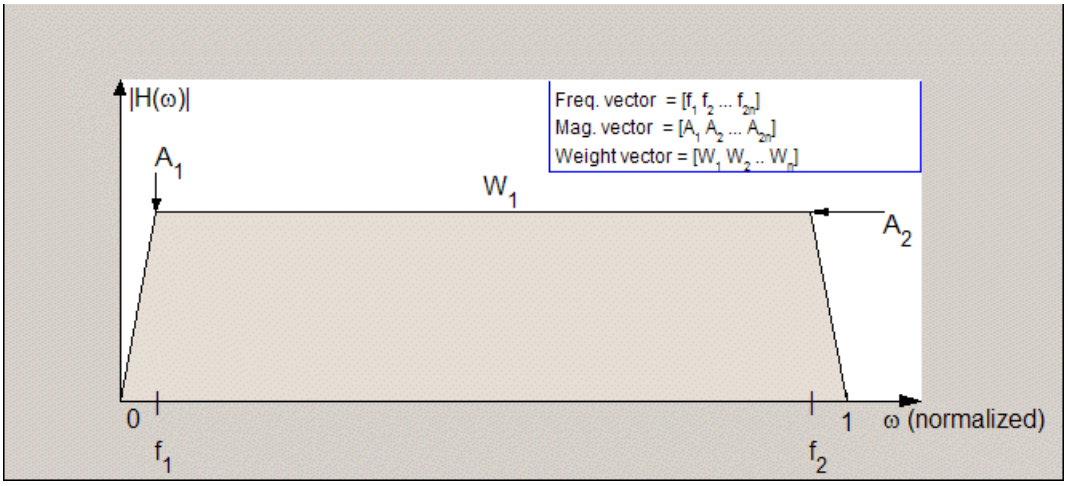
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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 to 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 sample rate** parameter.

Input sample rate

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.

Passband ripple

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

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Inverse Sinc Filter Design — 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

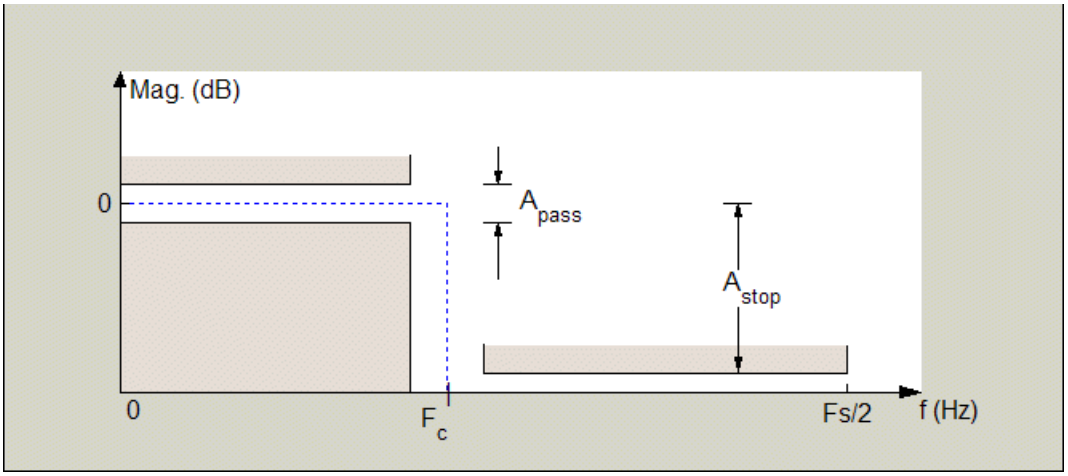
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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 Passband frequency (Fpass) and Stopband frequency (Fstop) 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 frequencies` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband frequency` — Define the filter by specifying frequencies for the edges of the passband.
- `Stopband frequency` — Define the filter by specifying frequencies for the edges of the stopbands.
- `Cutoff (6dB) frequency` — 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 to 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 sample rate** parameter.

Input sample rate

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.

Passband frequency

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**.

Stopband frequency

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.

Passband ripple

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Stopband attenuation

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 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.

Sinc frequency factor

A frequency dilation factor. The sinc frequency factor, C , parameterizes the passband magnitude response for a lowpass design through $H(\omega) = \text{sinc}(C\omega)\text{^}(-P)$ and for a highpass design through $H(\omega) = \text{sinc}(C(1-\omega))\text{^}(-P)$.

Sinc power

Negative power of passband magnitude response. The sinc power, P, parameterizes the passband magnitude response for a lowpass design through $H(\omega) = \text{sinc}(C\omega)\text{^}(-P)$ and for a highpass design through $H(\omega) = \text{sinc}(C(1-\omega))\text{^}(-P)$.

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

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Lowpass Filter Design — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select `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

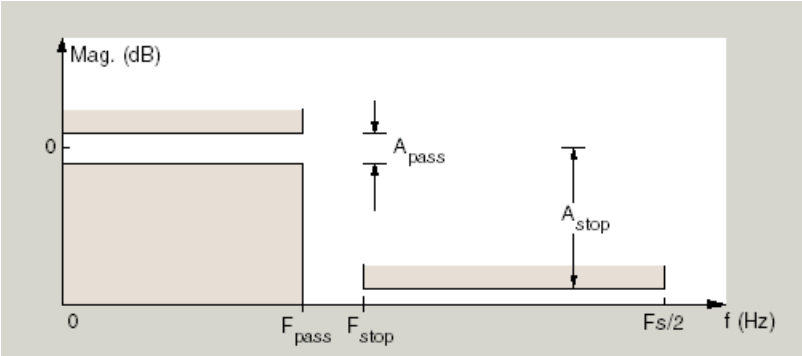
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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 Passband frequency (F_{pass}) and Stopband frequency (F_{stop}) represent transition regions where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband frequencies` — Define the filter by specifying the frequencies for the edge of the stopband and passband.
- `Passband frequency` — Define the filter by specifying the frequency for the edge of the passband.
- `Stopband frequency` — 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).
- `Half power (3dB) and stopband frequencies` — Define the filter by specifying the 3-dB down point and stopband edge frequency (IIR designs).
- `Half power (3dB) frequency` — Define the filter by specifying the frequency for the 3-dB point (IIR designs or maxflat FIR).
- `Cutoff (6dB) frequency` — 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 to 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 sample rate** parameter.

Input sample rate

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.

Passband frequency

Enter the frequency at the of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Stopband frequency

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.

Passband ripple

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Stopband attenuation

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 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.

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

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Notch

See “Peak/Notch Filter Design — Main Pane” on page 1-710.

Nyquist Filter Design — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Band

Specifies the location of the center of the transition region between the passband and the stopband. The center of the transition region, bw, is calculated using the value for Band:

bw = Fs/(2 × Band).

Impulse response

Select `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

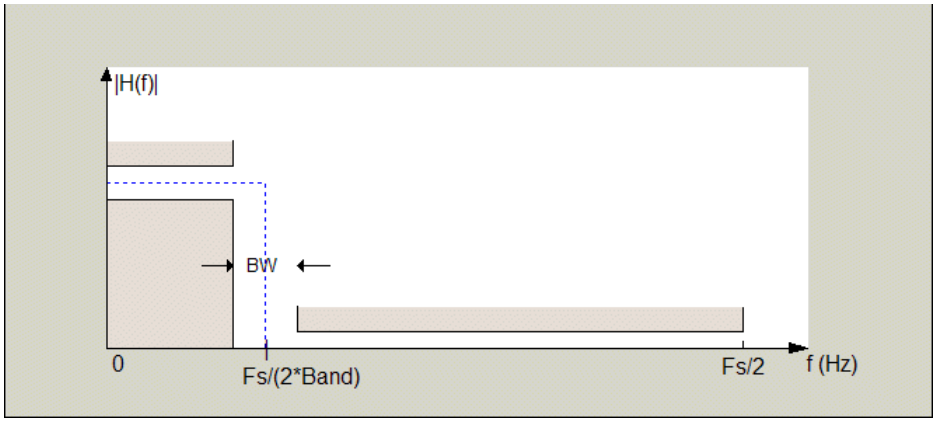
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

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.

- `Transition width` — Define the filter using transition width and stopband attenuation or transition width and order.
- `Unconstrained` — Define the filter by specifying the filter order and having no constraints on the transition width and stopband attenuation. You can add constraints on the magnitude by specifying the stopband attenuation.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 to 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 sample rate** parameter.

Input sample rate

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.

Stopband attenuation

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

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Octave Filter Design — 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 sample rate

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 cleared. 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 — 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 sample rate** box is disabled for input.

Input sample rate

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 `db(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 cleared. 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 — Main Pane

Filter specifications

In this area you can specify whether you want to design a peaking filter or a notching filter, as well as the order of the filter.

Response

Select `Peak` or `Notch` from the drop-down box.

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 sample rate

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`.

Passband ripple

This input box is enabled if the magnitude constraints selected are `Passband ripple` or `Passband ripple and stopband attenuation`. Enter the passband ripple.

Stopband attenuation

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 cleared. 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 —Main Pane

Filter specifications

Parameters in this group enable you to specify the shape and length of the filter.

Pulse shape

Select the shape of the impulse response from the following options:

- `Raised Cosine`
- `Square Root Raised Cosine`
- `Gaussian`

Order mode

This specification is only available for raised cosine and square root raised cosine filters. For these filters, select one of the following options:

- `Minimum`— This option will result in the minimum-length filter satisfying the user-specified **Frequency specifications**.
- `Specify order`—This option allows the user to construct a raised cosine or square root cosine filter of a specified order by entering an even number in the **Order** input box. The length of the impulse response will be `Order+1` .
- `Specify symbols`—This option enables the user to specify the length of the impulse response in an alternative manner. If `Specify symbols` is chosen, the **Order** input box changes to the **Number of symbols** input box.

Samples per symbol

Specify the oversampling factor. Increasing the oversampling factor guards against aliasing and improves the FIR filter approximation to the ideal frequency response. If **Order** is specified in **Number of symbols**, the filter length will be **Number of symbols*Samples per symbol**+1. The product **Number of symbols*Samples per symbol** must be an even number.

If a Gaussian filter is specified, the filter length must be specified in **Number of symbols** and **Samples per symbol**. The product **Number of symbols*Samples per symbol** must be an even number. The filter length will be **Number of symbols*Samples per symbol**+1.

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

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Introduced in R2009a

Filter Designer

Design filters starting with algorithm selection

Description

The **Filter Designer** app enables you to design and analyze digital filters. You can also import and modify existing filter designs.

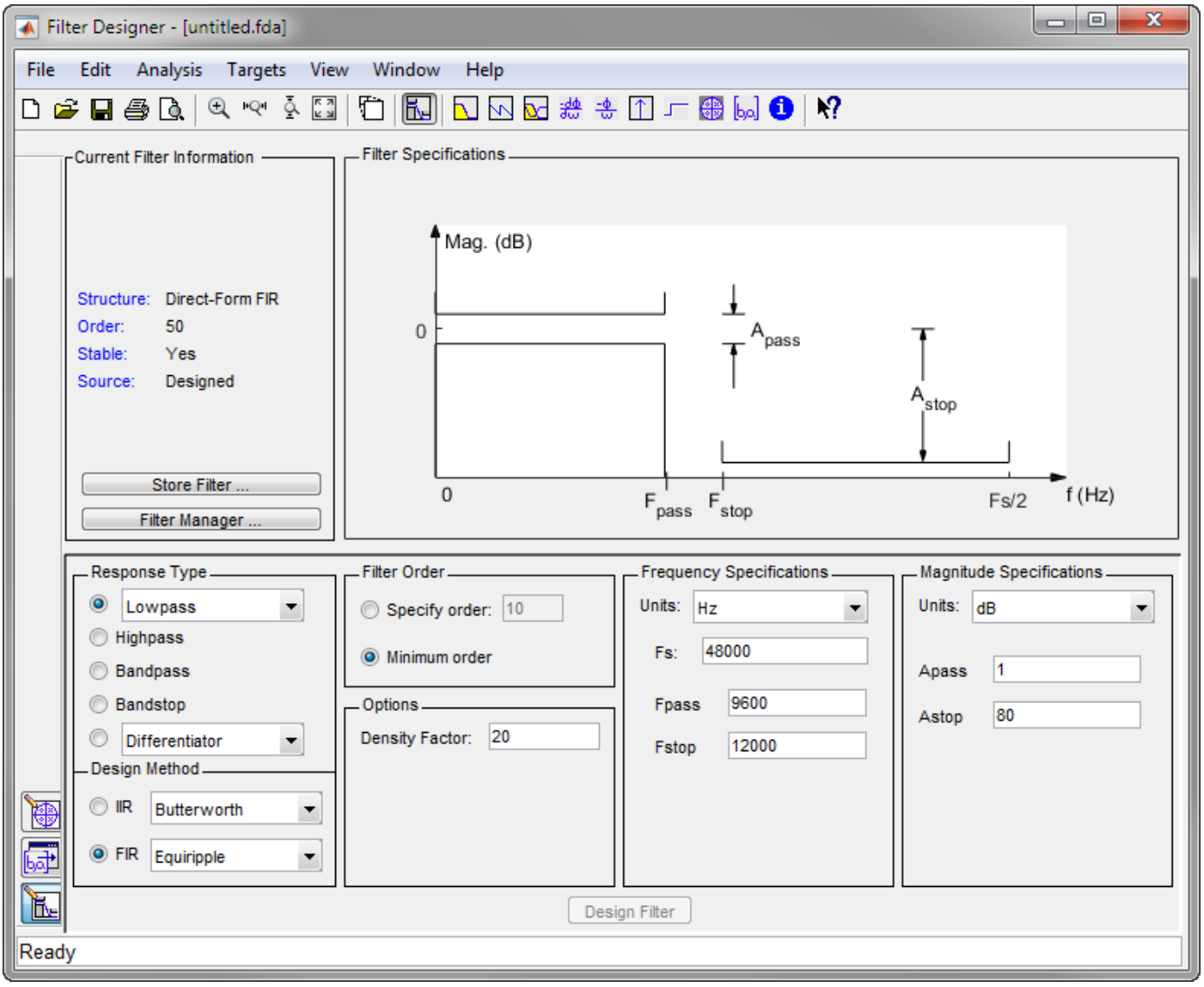
Using the app, you can:

- Choose a response type and filter design method
- Set filter design specifications
- Analyze, edit, and optimize a filter design
- Export a filter design or generate MATLAB code

For more information, see “Introduction to Filter Designer”.

If the DSP System Toolbox product is installed, **Filter Designer** integrates advanced filter design methods and the ability to quantize filters. For more information, see `filterDesigner` (DSP System Toolbox).

Note This app requires a screen resolution greater than 640 × 480.



Open the Filter Designer App

- MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- Enter `filterDesigner` in the MATLAB command prompt.

Examples

FIR Bandpass Filter with Asymmetric Attenuation

Use the **Filter Designer** app to create a 50th-order equiripple FIR bandpass filter to be used with signals sampled at 1 kHz.

```
N = 50;
Fs = 1e3;
```

Specify that the passband spans frequencies of 200–300 Hz and that the transition region on either side has a width of 50 Hz.

```
Fstop1 = 150;
Fpass1 = 200;
Fpass2 = 300;
Fstop2 = 350;
```

Specify weights for the optimization fit:

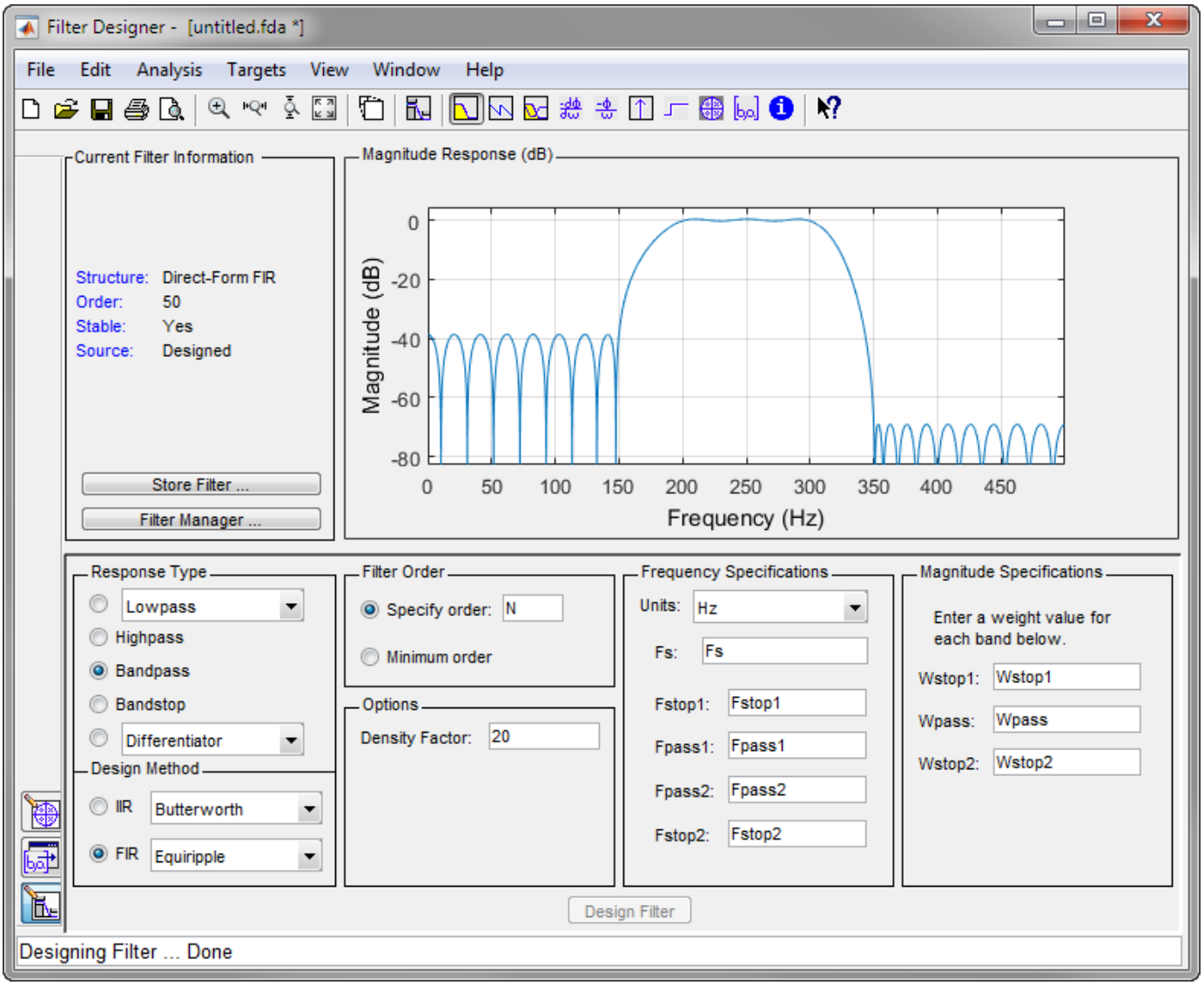
- 3 for the low-frequency stopband
- 1 for the passband
- 100 for the high-frequency stopband

Open the **Filter Designer** app.

```
Wstop1 = 3;
Wpass = 1;
Wstop2 = 100;

filterDesigner
```

Use the app to design the rest of the filter. To specify the frequency constraints and magnitude specifications, use the variables you created.

1. Set **Response Type** to `Bandpass`.
2. Set **Design Method** to `FIR`. From the drop-down list, select `Equiripple`.
3. Under **Filter Order**, specify the order as `N`.
4. Under **Frequency Specifications**, specify **Fs** as `Fs`.
5. Click **Design Filter**.



Arbitrary Magnitude Filter

Design an FIR filter with the following piecewise frequency response:

- A sinusoid between 0 and 0.19π rad/sample.

  ```
  F1 = 0:0.01:0.19;
  A1 = 0.5+sin(2*pi*7.5*F1)/4;
  ```

- A piecewise linear section between 0.2π rad/sample and 0.78π 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 0.79π rad/sample and the Nyquist frequency.

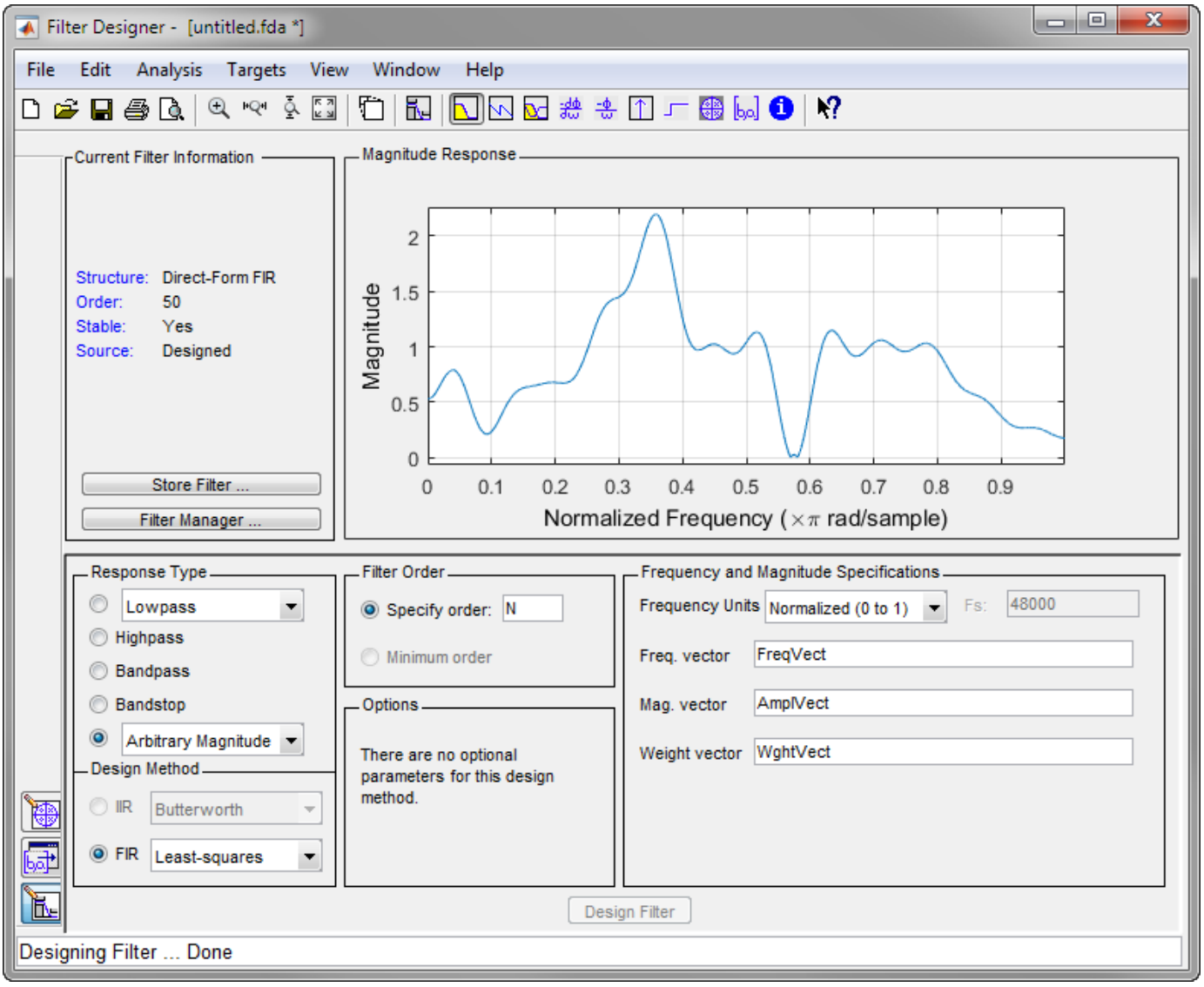
  ```
  F3 = 0.79:0.01:1;
  A3 = 0.2+18*(1-F3).^2;
  ```

Specify a filter order of 50. Consolidate the frequency and amplitude vectors. To give all bands equal weights during the optimization fit, specify a weight vector of all ones. Open the **Filter Designer** app.

```
N = 50;

FreqVect = [F1 F2 F3];
AmplVect = [A1 A2 A3];
WghtVect = ones(1,N/2);

filterDesigner
```

Use the app to design the filter.

1. Under **Response Type**, select the button next to `Differentiator`. From the drop-down list, choose `Arbitrary Magnitude`.
2. Set **Design Method** to `FIR`. From the drop-down list, select `Least-squares`.
3. Under **Filter Order**, specify the order as the variable `N`.
4. Under **Frequency and Magnitude Specifications**, specify the variables you created:
   - **Freq. vector** — `FreqVect`.
   - **Mag. vector** — `AmplVect`.
   - **Weight vector** — `WghtVect`.
5. Click **Design Filter**.
6. Right-click the *y*-axis of the plot and select **Magnitude** to express the magnitude response in linear units.



- “Introduction to Filter Designer”
- “Getting Started with Filter Designer”

See Also

Apps
Signal Analyzer | **Window Designer**

Functions
`designfilt` | **FVTool** | **WVTool**

Topics
“Introduction to Filter Designer”
“Getting Started with Filter Designer”

Introduced before R2006a

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 0.5π 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 0.2π rad/s.

```
b = firpm(30,[.1 .9],[1 1],'Hilbert');
Linf = filternorm(b,1,inf)
```

```
Linf = 1.0028
```

Algorithms

Given a filter with frequency response $H(e^{j\omega})$, the L_p-norm for $1 \le p < \infty$ is given by

$$\left\|H(e^{j\omega})\right\|_p = \left(\frac{1}{2\pi}\int_{-\pi}^{\pi}\left|H(e^{j\omega})\right|^p d\omega\right)^{1/p}.$$

For the case $p \to \infty$, the L_∞-norm is

$$\left\|H(e^{j\omega})\right\|_\infty = \max_{-\pi \le \omega \le \pi}\left|H(e^{j\omega})\right|.$$

For the case $p = 2$, Parseval's theorem states that

$$\left\|H(e^{j\omega})\right\|_2 = \left(\frac{1}{2\pi}\int_{-\pi}^{\pi}\left|H(e^{j\omega})\right|^2 d\omega\right)^{1/2} = \left(\sum_n |h(n)|^2\right)^{1/2},$$

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`zp2sos` | `norm`

Introduced before R2006a

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. After filtering the data in the forward direction, the function reverses the filtered sequence and runs it back through the filter. The result has these characteristics:

- Zero phase distortion.
- A filter transfer function equal to 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. Do not use `filtfilt` with differentiator and Hilbert FIR filters, because the operation of these filters depends heavily on their phase response.

`y = filtfilt(sos,g,x)` zero-phase filters the input data `x` using the second-order section (biquad) filter represented by the matrix `sos` and the scale values `g`.

`y = filtfilt(d,x)` zero-phase filters the input data `x` using a digital filter `d`. Use `designfilt` to generate `d` based on frequency-response specifications.

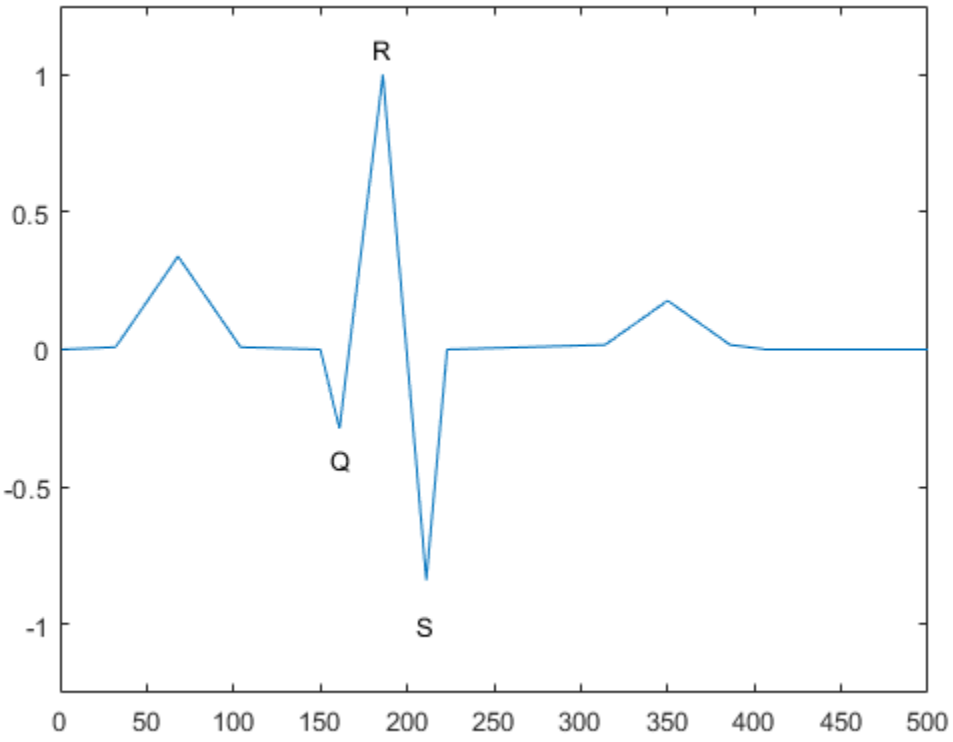
Examples

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.

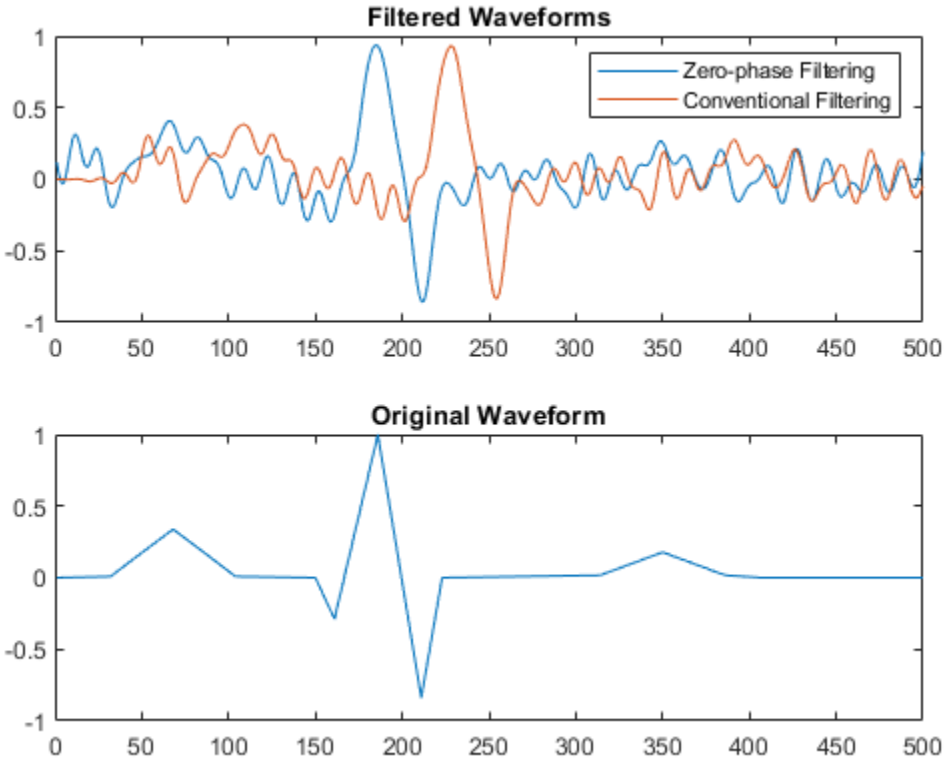
Zero-phase filter a synthetic electrocardiogram (ECG) waveform. The function that generates the waveform is at the end of the example. The QRS complex is an important feature in the ECG. Here it begins around time point 160.

```
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")
```



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.

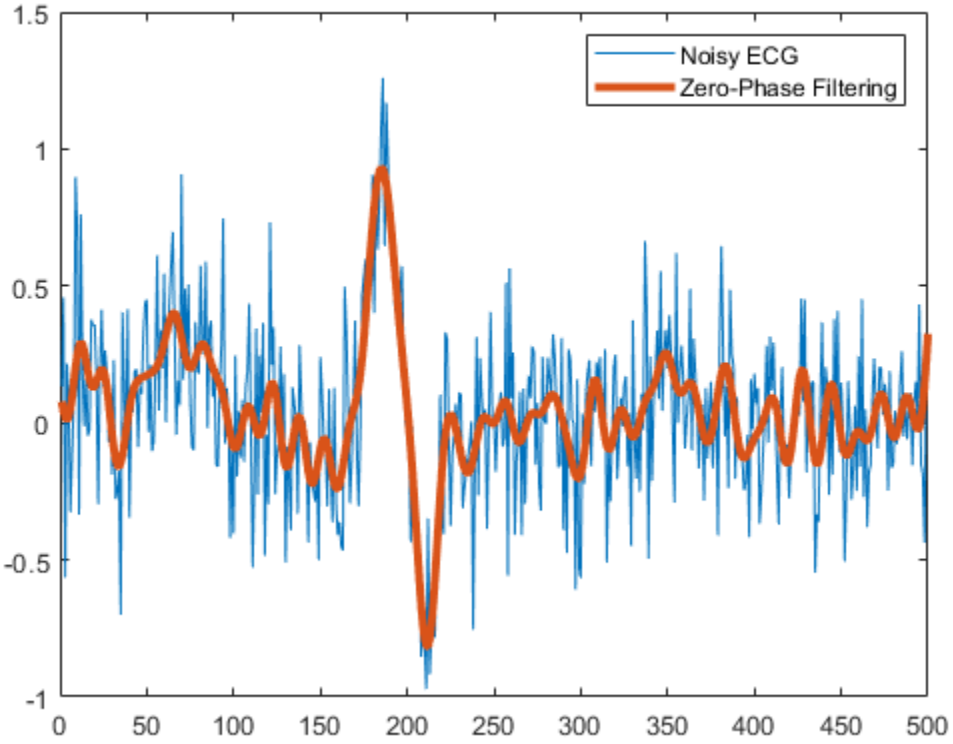
```
rng default

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 filtering 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")
```



This function generates the ECG waveform.

```
function x = ecg(L)
%ECG Electrocardiogram (ECG) signal generator.
%   ECG(L) generates a piecewise linear ECG signal of length L.
%
%   EXAMPLE:
%   x = ecg(500).';
%   y = sgolayfilt(x,0,3); % Typical values are: d=0 and F=3,5,9, etc.
%   y5 = sgolayfilt(x,0,5);
%   y15 = sgolayfilt(x,0,15);
%   plot(1:length(x),[x y y5 y15]);

%   Copyright 1988-2002 The MathWorks, Inc.

a0 = [0,1,40,1,0,-34,118,-99,0,2,21,2,0,0,0]; % Template
d0 = [0,27,59,91,131,141,163,185,195,275,307,339,357,390,440];
a = a0/max(a0);
d = round(d0*L/d0(15)); % Scale them to fit in length L
d(15)=L;

for i=1:14
       m = d(i):d(i+1)-1;
       slope = (a(i+1)-a(i))/(d(i+1)-d(i));
       x(m+1) = a(i)+slope*(m-d(i));
end
```

```
end
```

Input Arguments

b, a — Transfer function coefficients

vectors

Transfer function coefficients, specified as vectors. If you use an all-pole filter, enter `1` for `b`. If you use an all-zero (FIR) filter, enter `1` for `a`.

Example: `b = [1 3 3 1]/6` and `a = [3 0 1 0]/3` specify a third-order Butterworth filter with a normalized 3-dB frequency of 0.5π rad/sample.

Data Types: `double`

x — Input signal

vector | matrix | *N*-D array

Input signal, specified as a real-valued or complex-valued vector, matrix, or *N*-D array. `x` must be finite-valued. The length of `x` must be greater than three times the filter order, defined as max(length(B)-1, length(A)-1). The function operates along the first array dimension of `x` unless `x` is a row vector. If `x` is a row vector, then the function operates along the second dimension.

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: `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, then the function treats the input as 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 a normalized 3-dB frequency of 0.5π rad/sample.

Data Types: `double`

g — Scale factors

vector

Scale factors, specified as a vector.

Data Types: `double`

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 a normalized 3-dB frequency of 0.5π rad/sample.

Data Types: `double`

Output Arguments

y — Filtered signal

vector | matrix | *N*-D array

Filtered signal, returned as a vector, matrix, or *N*-D array.

References

[1] Gustafsson, F. “Determining the initial states in forward-backward filtering.” *IEEE Transactions on Signal Processing*. Vol. 44, April 1996, pp. 988–992.

[2] Mitra, Sanjit K. *Digital Signal Processing*. 2nd Ed. New York: McGraw-Hill, 2001.

[3] Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

`digitalFilter` objects are not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

`digitalFilter` objects are not supported for code generation.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`designfilt` | `digitalFilter` | `fftfilt` | `filter` | `filter2`

Topics

“Remove the 60 Hz Hum from a Signal”
“Practical Introduction to Digital Filtering”
“Anti-Causal, Zero-Phase Filter Implementation”

Introduced before R2006a

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.

`z = filtic(b,a,y)` assumes that the input `x` is 0 in the past.

Examples

Zero Input Response for Past Input and Output

Determine the zero input response of the following system:
$y(n) + 1.12y(n-1) = 0.1x(n) + 0.2x(n-1)$ with initial condition $y(-1) = 1$. Set the numerator and denominator coefficients and the initial conditions for the output.

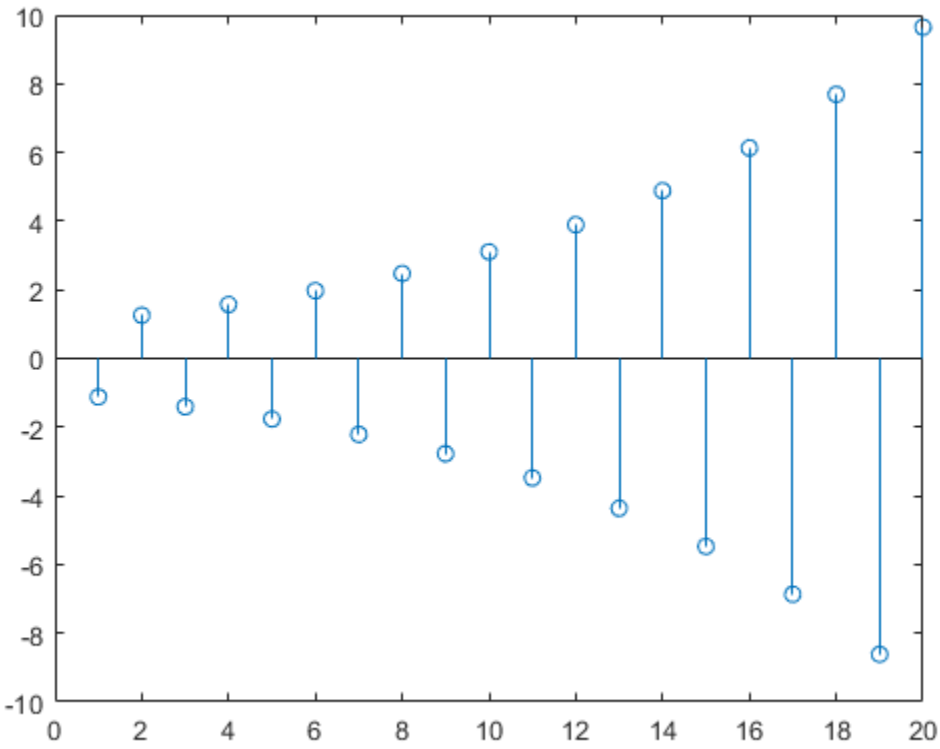
```
b = [0.1 0.2];
a = [1 1.12];
Y = 1;
```

Calculate the zero input initial conditions for the system.

```
xic = filtic(b,a,Y);
```

Compute the zero input response.

```
yzi = filter(b,a,zeros(1,20),xic);
stem(yzi)
```



Input Arguments

b, a — Transfer function coefficients

vectors

Transfer function coefficients, specified as vectors.

Example: `b = [1 3 3 1]/6` and `a = [3 0 1 0]/3` specify a third-order Butterworth filter with a normalized 3-dB frequency of 0.5π rad/sample.

Data Types: `double`

y — Past output

vector

Past output, specified as a vector. The vector `y` contains the most recent output first, and oldest output last as in

$$y = [y(-1), y(-2), y(-3), \ldots, y(-m)]$$

where `m` is `length(a)-1` (the denominator order). If `length(y)` is less than `m`, the function pads it with zeros to length `m`.

Data Types: `double`

x — Past input

vector

Past input, specified as a vector. The vector `x` contains the most recent input first, and oldest input last as in

$$x = [x(-1), x(-2), x(-3), \ldots, x(-n)]$$

where `n` is `length(b)-1` (the numerator order). If `length(x)` is less than `n`, the function pads it with zeros to length `n`.

Data Types: `double`

Output Arguments

z — Initial conditions

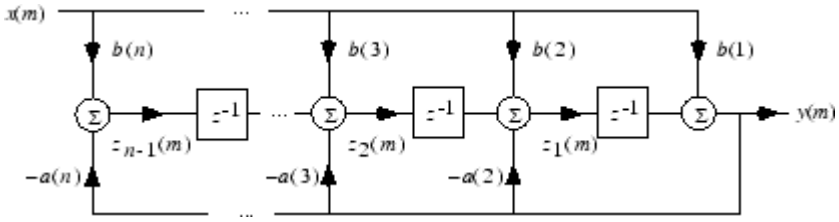
column vector

Initial conditions, returned as a column vector. The length of `z` is equal to the larger of *n* and *m*. `z` describes the state of the delays given past inputs `x` and past outputs `y`.

Algorithms

The `filtic` function performs a reverse difference equation to obtain the delay states `z`. Elements of `x` beyond `x(n-1)` and elements of `y` beyond `y(m-1)` are unnecessary so `filtic` ignores them.

The transposed direct-form II structure is shown in this illustration, where *n* – 1 is the filter order.



References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 296, 301-302.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`filter` | `filtfilt`

Introduced before R2006a

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 0.5π 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)

FIR = 114

IIR = filtord(iir)

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 Kth 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 Kth row of the matrix, `sos`, are ordered as follows.

$$\left[B_k(1)\ B_k(2)\ B_k(3)\ \ A_k(1)\ A_k(2)\ A_k(3)\right].$$

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 0.5π rad/sample.

Output Arguments

n — Filter order
integer

Filter order, specified as an integer.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`designfilt` | `digitalFilter` | `isallpass` | `isminphase` | `ismaxphase` | `isstable`

Introduced in R2013a

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
```

Structures

Structures for `filtstates` specify the type of filter structure. Available types of structures for `filtstates` are shown below.

filtstates.structure	Description
`filtstates.dfiir`	filtstates for IIR direct-form I filters (`dfilt.df1`, `dfilt.df1t`, `dfilt.df1sos`, and `dfilt.df1tsos`)
`filtstates.cic`	filtstates for cascaded integrator comb filters. (Available only with 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`

Introduced before R2006a

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
`double`	Converts a `filtstates` object to a double-precision vector containing the values of the numerator and denominator states. The numerator states are listed first in this vector, followed by the denominator states.
`single`	Converts a `filtstates` object to a single-precision vector containing the values of the numerator and denominator states. (This method is used with the 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`

Introduced before R2006a

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`.

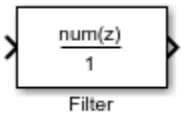
`filt2block(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Generate Block from FIR Filter

Design a 30th-order FIR filter using the window method. Specify a cutoff frequency of $\pi/4$ rad/sample. Create a Simulink® block.

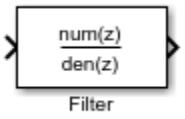
```
b = fir1(30,0.25);
filt2block(b)
```



Generate Block from IIR Filter

Design a 30th-order IIR Butterworth filter. Specify a cutoff frequency of $\pi/4$ rad/sample. Create a Simulink® block.

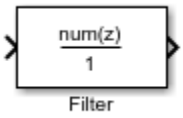
```
[b,a] = butter(30,0.25);
filt2block(b,a)
```



Generate FIR Filter with Direct Form I Transposed Structure

Design a 30th-order FIR filter using the window method. Specify a cutoff frequency of $\pi/4$ rad/sample. Create a Simulink® block with a direct form I transposed structure.

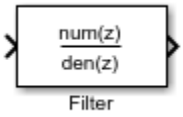
```
b = fir1(30,0.25);
filt2block(b,'FilterStructure','directFormTransposed')
```



Generate IIR Filter with Direct Form I Structure

Design a 30th-order IIR Butterworth filter. Specify a cutoff frequency of $\pi/4$ rad/sample. Create a Simulink® block with a direct form I structure.

```
[b,a] = butter(30,0.25);
filt2block(b,a,'FilterStructure','directForm1')
```



Generate Subsystem Block from Second-Order Section Matrix

Design a 5-th order Butterworth filter with a cutoff frequency of $\pi/5$ rad/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 40Design 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:

$$\left[B_k(1)\ B_k(2)\ B_k(3)\ A_k(1)\ A_k(2)\ A_k(3)\right]$$

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('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)` specifies a third-order Butterworth filter with normalized 3 dB frequency 0.5π rad/sample.

structure — Filter structure

character vector | string scalar

Filter structure, specified as a character vector or string scalar. 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'`

Input	Filter Structures
d	• For FIR filters: 'directForm' (default), 'directFormTransposed', 'directFormSymmetric', 'directFormAntiSymmetric', 'overlapAdd'. The 'overlapAdd' structure is only available when you omit 'subsystem' and requires a DSP System Toolbox software license. • For IIR filters: 'directForm2Transposed' (default), 'directForm1', 'directForm1Transposed', 'directForm2'

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: filt2block(...,'subsystem','BlockName','Lowpass FIR','FrameBasedProcessing',false)

Destination — Destination for Simulink filter block
'current' (default) | 'new' | character vector | string scalar

Destination for the Simulink filter block, specified as a character vector or string scalar. 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 an existing model.

Example: filt2block([1 2 1],'Destination','MyModel','BlockName','New block')

Data Types: char | string

BlockName — Block name
character vector | string scalar

Block name, specified as a character vector or string scalar.

Data Types: char | 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 character vectors | string array

Coefficient variable names, specified as a cell array of character vectors or a string 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` | `string`

`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

`designfilt` | `digitalFilter`

Introduced in R2013a

findchangepts

Find abrupt changes in signal

Syntax

```
ipt = findchangepts(x)
ipt = findchangepts(x,Name,Value)

[ipt,residual] = findchangepts( ___ )

findchangepts( ___ )
```

Description

`ipt = findchangepts(x)` returns the index at which the mean of `x` changes most significantly.

- If `x` is a vector with *N* elements, then `findchangepts` partitions `x` into two regions, `x(1:ipt-1)` and `x(ipt:N)`, that minimize the sum of the residual (squared) error of each region from its local mean.
- If `x` is an *M*-by-*N* matrix, then `findchangepts` partitions `x` into two regions, `x(1:M,1:ipt-1)` and `x(1:M,ipt:N)`, returning the column index that minimizes the sum of the residual error of each region from its local *M*-dimensional mean.

`ipt = findchangepts(x,Name,Value)` specifies additional options using name-value arguments. Options include the number of changepoints to report and the statistic to measure instead of the mean. See “Changepoint Detection” on page 1-767 for more information.

`[ipt,residual] = findchangepts( ___ )` also returns the residual error of the signal against the modeled changes, incorporating any of the previous specifications.

`findchangepts( ___ )` without output arguments plots the signal and any detected changepoints. For more information, see “Statistic” on page 1-0 .

Note Before plotting, the `findchangepts` function clears (`clf`) the current figure. To plot the signal and detected changepoints in a subplot, use a plotting function. See “Audio File Segmentation” on page 1-752.

Examples

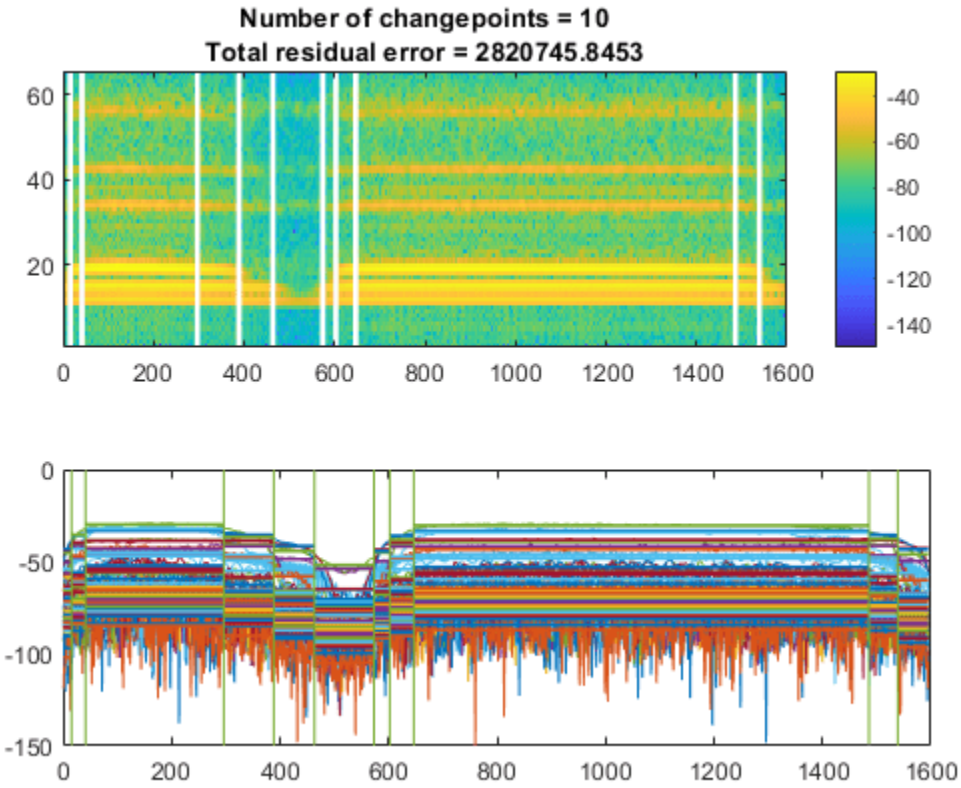
Changepoints in One and Two Dimensions

Load a data file containing a recording of a train whistle sampled at 8192 Hz. Find the 10 points at which the root-mean-square level of the signal changes most significantly.

```
load train

findchangepts(y,'MaxNumChanges',10,'Statistic','rms')
```

Compute the short-time power spectral density of the signal. Divide the signal into 128-sample segments and window each segment with a Hamming window. Specify 120 samples of overlap between adjoining segments and 128 DFT points. Find the 10 points at which the mean of the power spectral density changes the most significantly.

```
[s,f,t,pxx] = spectrogram(y,128,120,128,Fs);

findchangepts(pow2db(pxx),'MaxNumChanges',10)
```



Changepoint Search Options

Reset the random number generator for reproducible results. Generate a random signal where:

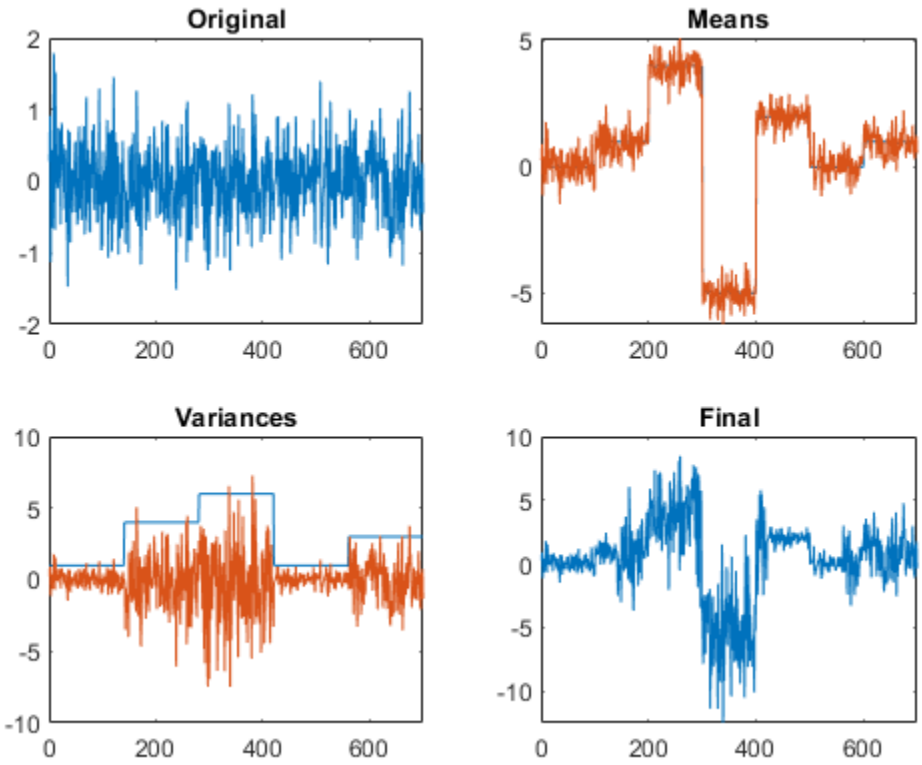
- The mean is constant in each of seven regions and changes abruptly from region to region.
- The variance is constant in each of five regions and changes abruptly from region to region.

```
rng('default')

lr = 20;

mns = [0 1 4 -5 2 0 1];
nm = length(mns);

vrs = [1 4 6 1 3];
```

```
nv = length(vrs);

v = randn(1,lr*nm*nv)/2;

f = reshape(repmat(mns,lr*nv,1),1,lr*nm*nv);
y = reshape(repmat(vrs,lr*nm,1),1,lr*nm*nv);

t = v.*y+f;
```

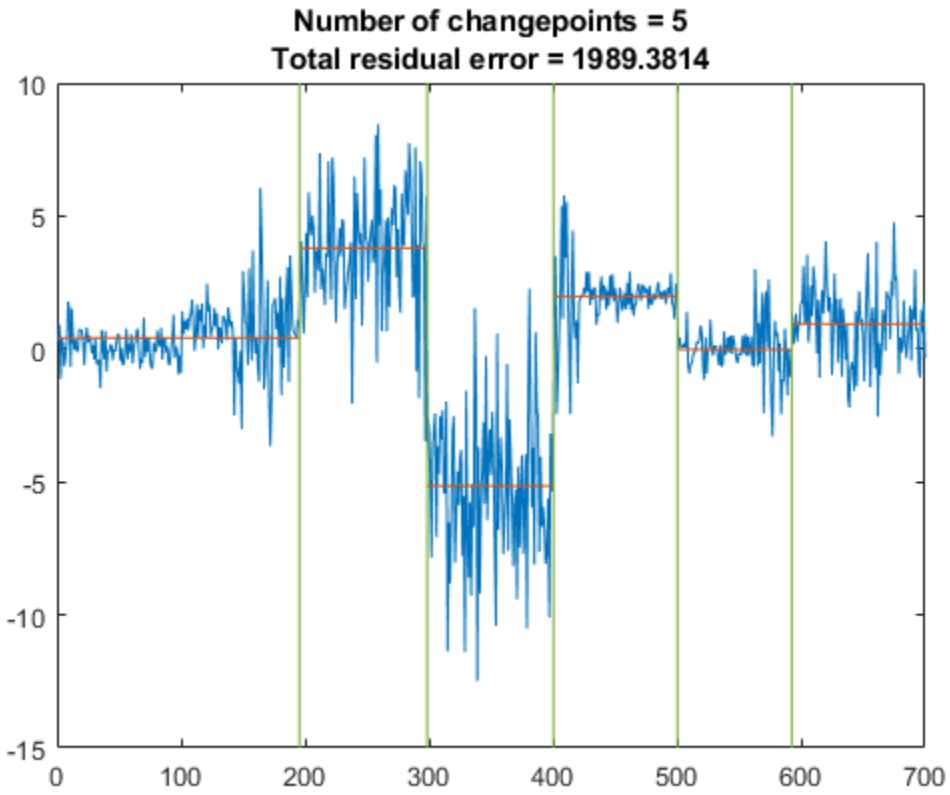
Plot the signal, highlighting the steps of its construction.

```
subplot(2,2,1)
plot(v)
title('Original')
xlim([0 700])

subplot(2,2,2)
plot([f;v+f]')
title('Means')
xlim([0 700])

subplot(2,2,3)
plot([y;v.*y]')
title('Variances')
xlim([0 700])

subplot(2,2,4)
plot(t)
title('Final')
xlim([0 700])
```



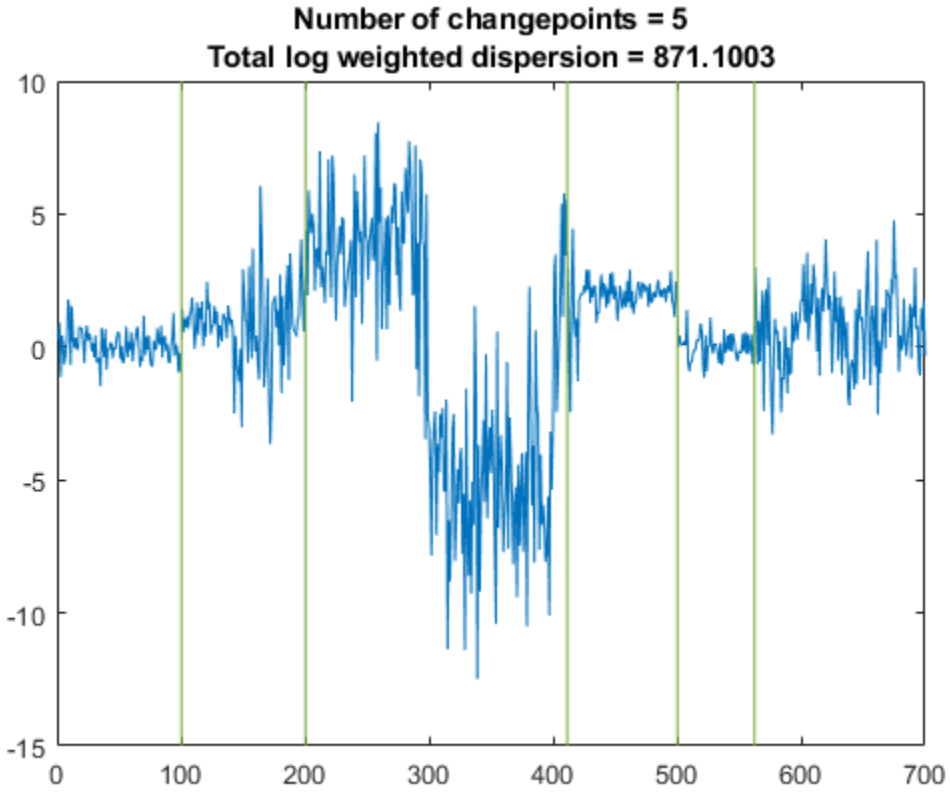
Find the five points where the mean of the signal changes most significantly.

```
figure
findchangepts(t,'MaxNumChanges',5)
```



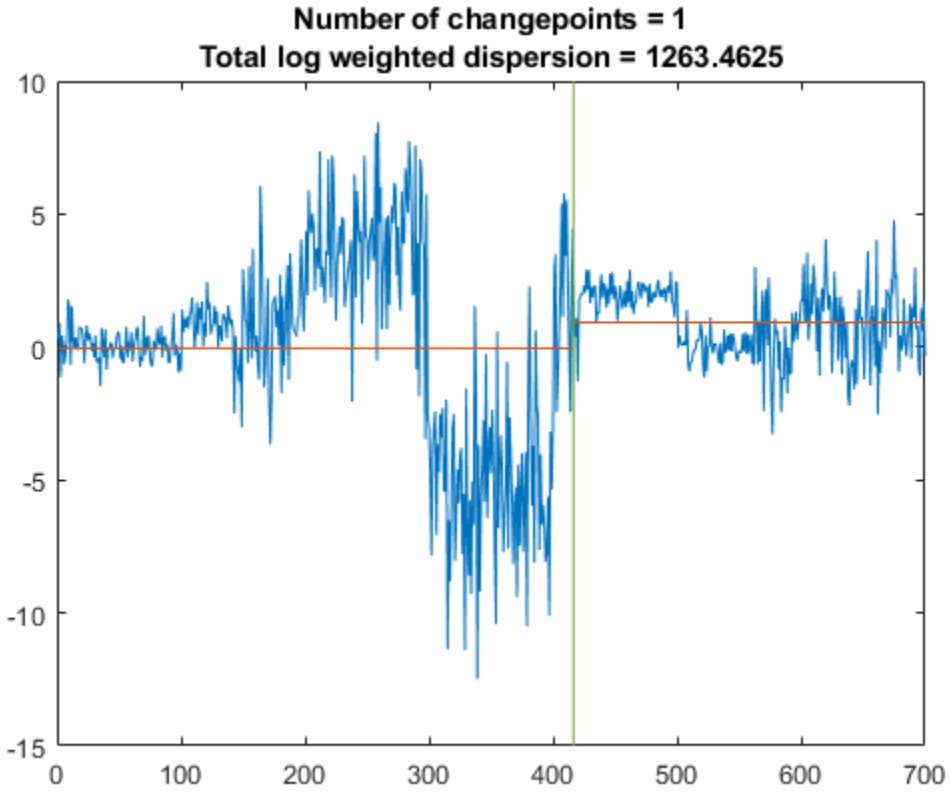
Find the five points where the root-mean-square level of the signal changes most significantly.

```
findchangepts(t,'MaxNumChanges',5,'Statistic','rms')
```



Find the point where the mean and standard deviation of the signal change the most.

```
findchangepts(t,'Statistic','std')
```



Audio File Segmentation

Load a speech signal sampled at F_s = 7418 Hz. The file contains a recording of a female voice saying the word "MATLAB®."

```
load mtlb
```

Discern the vowels and consonants in the word by finding the points at which the variance of the signal changes significantly. Limit the number of changepoints to five.

```
numc = 5;

[q,r] = findchangepts(mtlb,Statistic="rms",MaxNumChanges=numc);
```

Create a signal mask for the speech signal based on the changepoint indices. See `signalMask` for more information about using a signal mask.

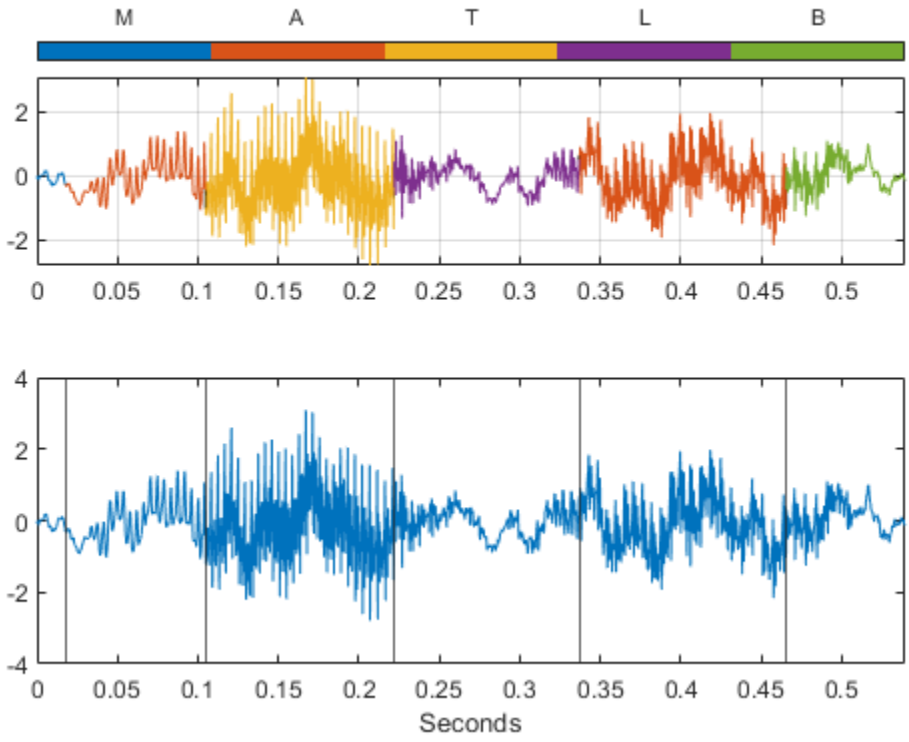
```
t = (0:length(mtlb)-1)/Fs;
roitable = ([[1;q] [q;length(mtlb)]]);

x = ["M" "A" "T" "L" "A" "B"]';
c = categorical(x,unique(x,"stable"));

msk = signalMask(table(t(roitable),c),SampleRate=Fs,RightShortening=1);
roimask(msk)
```

```
ans=6×2 table
           Var1            c
    ___________________    _

          0    0.017525    M
    0.01766     0.10461    A
    0.10475     0.22162    T
    0.22176     0.33675    L
    0.33688     0.46535    A
    0.46549     0.53909    B
```

Plot the speech signal and detected changepoints in a subplot along with the regions of interest from the signal mask:

- In the upper subplot, use the `plotsigroi` function to visualize the signal mask regions. Adjust the settings to make the colorbar appear at the top..
- In the lower subplot, plot the original speech signal and add the detected changepoints as vertical lines.

```
subplot(2,1,1)
plotsigroi(msk,mtlb)
colorbar("off")
nc = numel(c)-1;
colormap(gca,lines(nc));
colorbar(TickLabels=categories(c),Ticks=1/2/nc:1/nc:1, ...
    TickLength=0,Location="northoutside")
xlabel("")

subplot(2,1,2)
plot(t,mtlb)
hold on
xline(q/Fs)
hold off
xlim([0 t(end)])
xlabel("Seconds")
```



To play the sound with a pause after each of the segments, uncomment these lines.

```
% for k = 1:length(roitable)
%     intv = roitable(k,1):roitable(k,2);
%     soundsc(mtlb(intv).*hann(length(intv)),Fs)
%     pause(.5)
% end
```

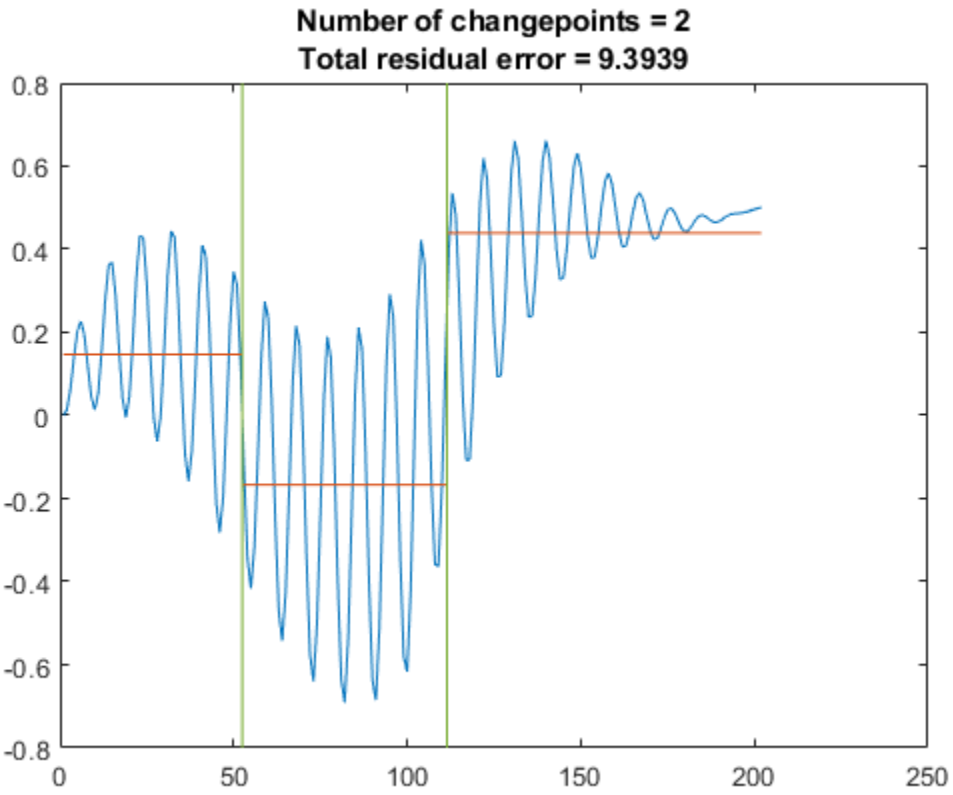
Change of Mean, RMS Level, Standard Deviation, and Slope

Create a signal that consists of two sinusoids with varying amplitude and a linear trend.

```
vc = sin(2*pi*(0:201)/17).*sin(2*pi*(0:201)/19).* ...
    [sqrt(0:0.01:1) (1:-0.01:0).^2]+(0:201)/401;
```

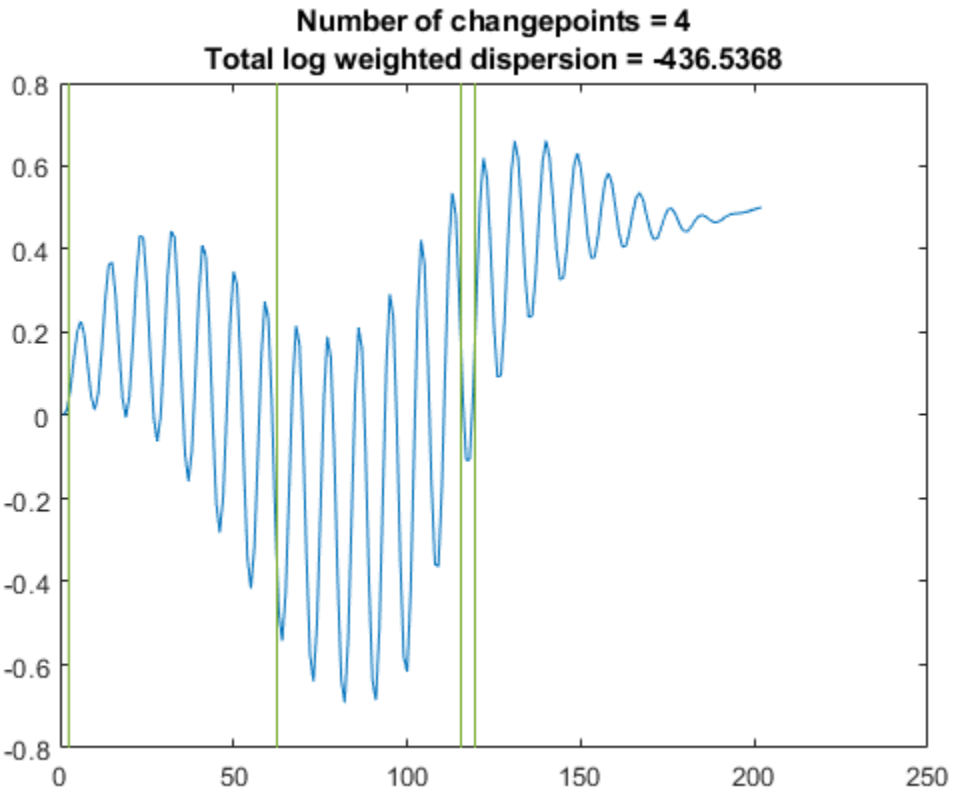
Find the points where the signal mean changes most significantly. The `'Statistic'` name-value argument is optional in this case. Specify a minimum residual error improvement of 1.

```
findchangepts(vc,'Statistic','mean','MinThreshold',1)
```



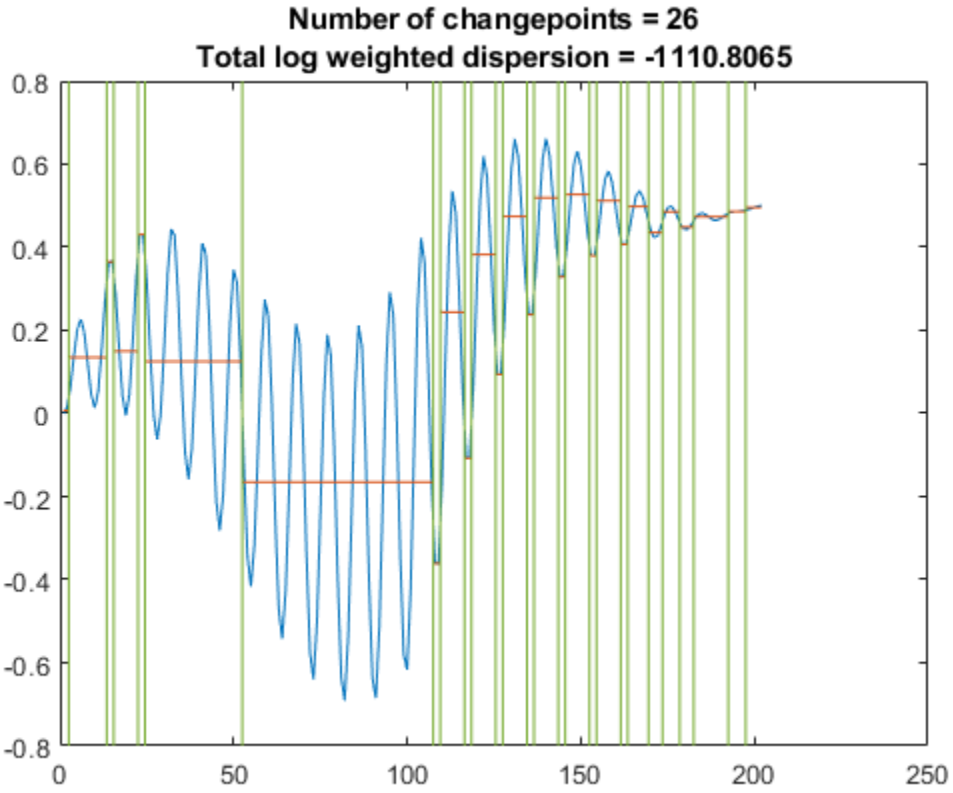
Find the points where the root-mean-square level of the signal changes the most. Specify a minimum residual error improvement of 6.

```
findchangepts(vc,'Statistic','rms','MinThreshold',6)
```



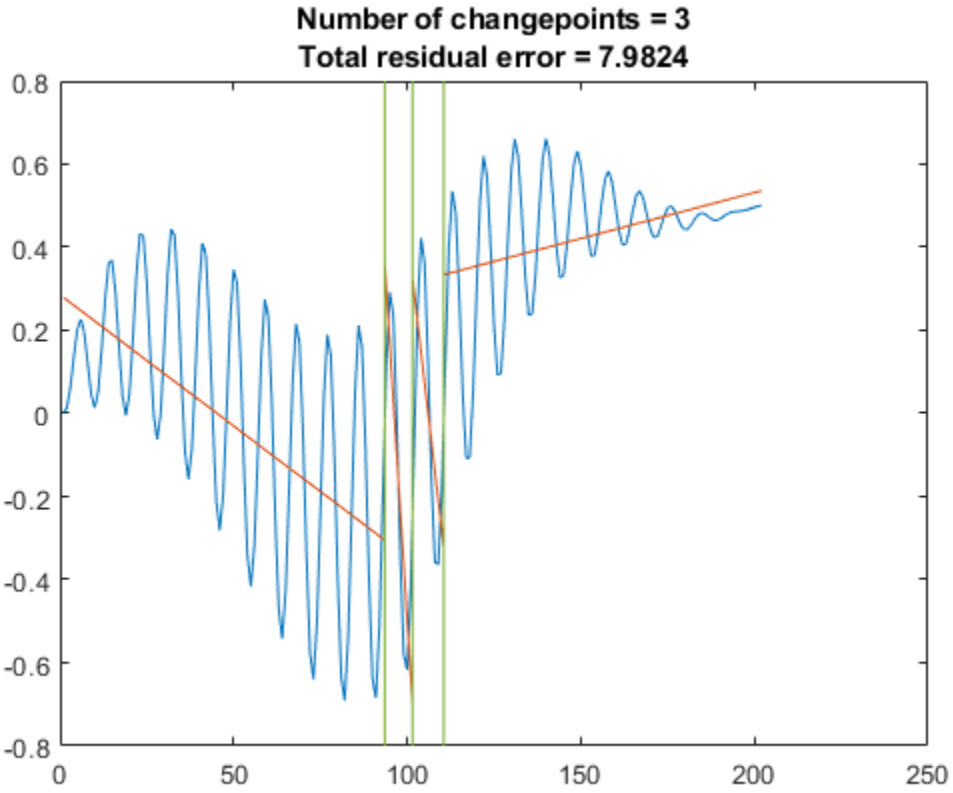
Find the points where the standard deviation of the signal changes most significantly. Specify a minimum residual error improvement of 10.

```
findchangepts(vc,'Statistic','std','MinThreshold',10)
```



Find the points where the mean and the slope of the signal change most abruptly. Specify a minimum residual error improvement of 0.6.

```
findchangepts(vc,'Statistic','linear','MinThreshold',0.6)
```



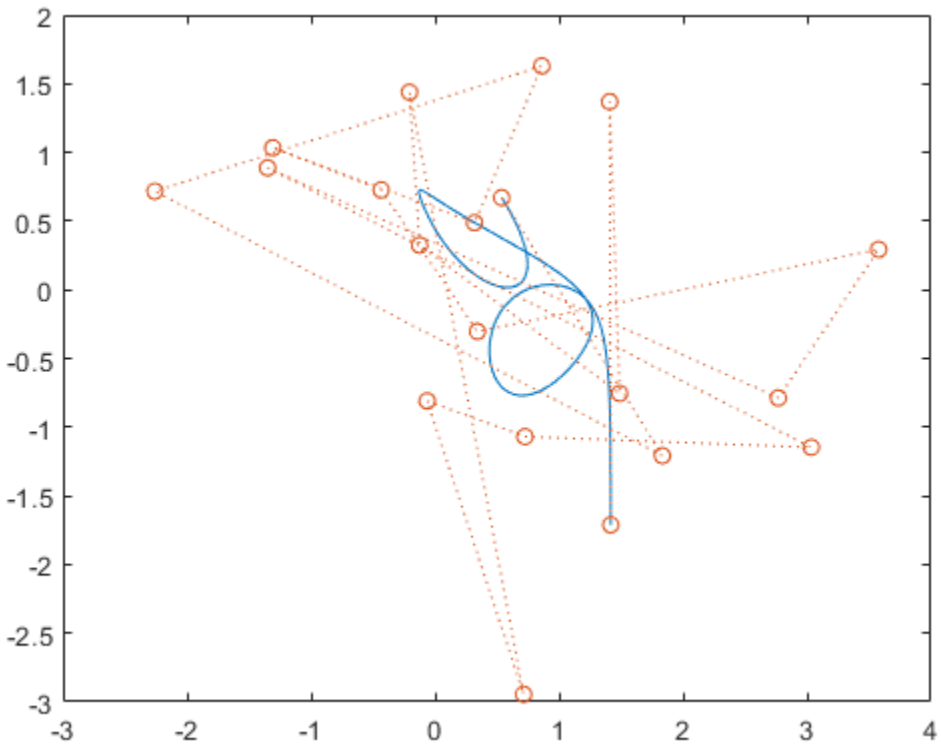
Changepoints of 2-D and 3-D Bézier Curves

Generate a two-dimensional, 1000-sample Bézier curve with 20 random control points. A Bézier curve is defined by:

$$C(t) = \sum_{k=0}^{m} \binom{m}{k} t^k (1-t)^{m-k} P_k,$$

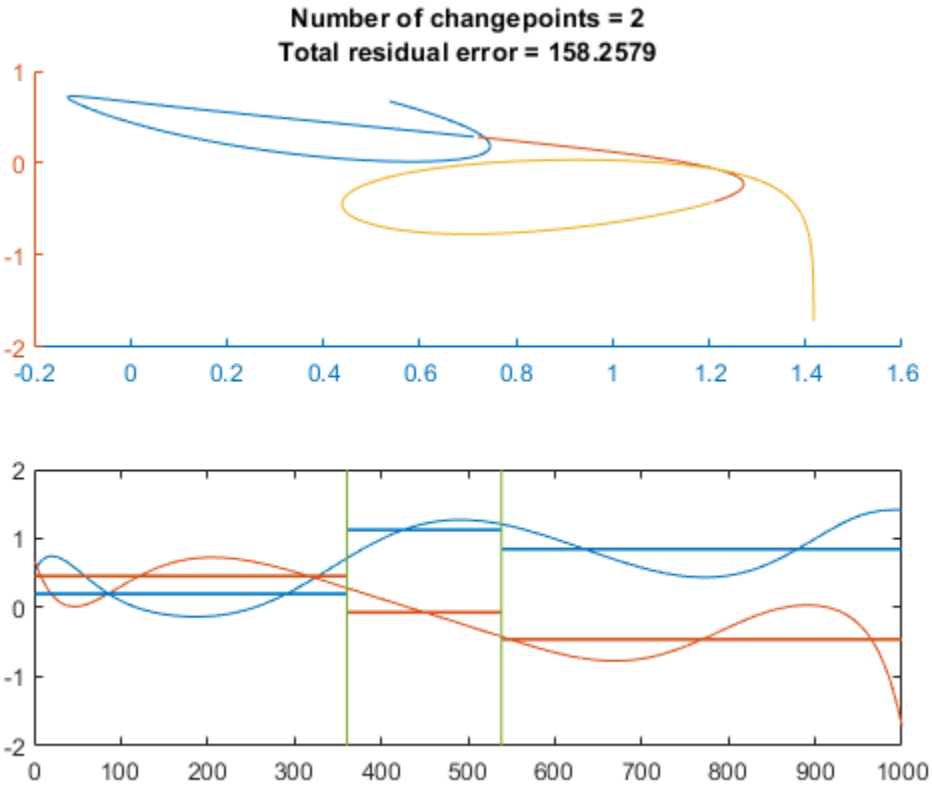
where P_k is the kth of m control points, t ranges from 0 to 1, and $\binom{m}{k}$ is a binomial coefficient. Plot the curve and the control points.

```
m = 20;
P = randn(m,2);
t = linspace(0,1,1000)';

pol = t.^(0:m-1).*(1-t).^(m-1:-1:0);
bin = gamma(m)./gamma(1:m)./gamma(m:-1:1);
crv = bin.*pol*P;

plot(crv(:,1),crv(:,2),P(:,1),P(:,2),'o:')
```



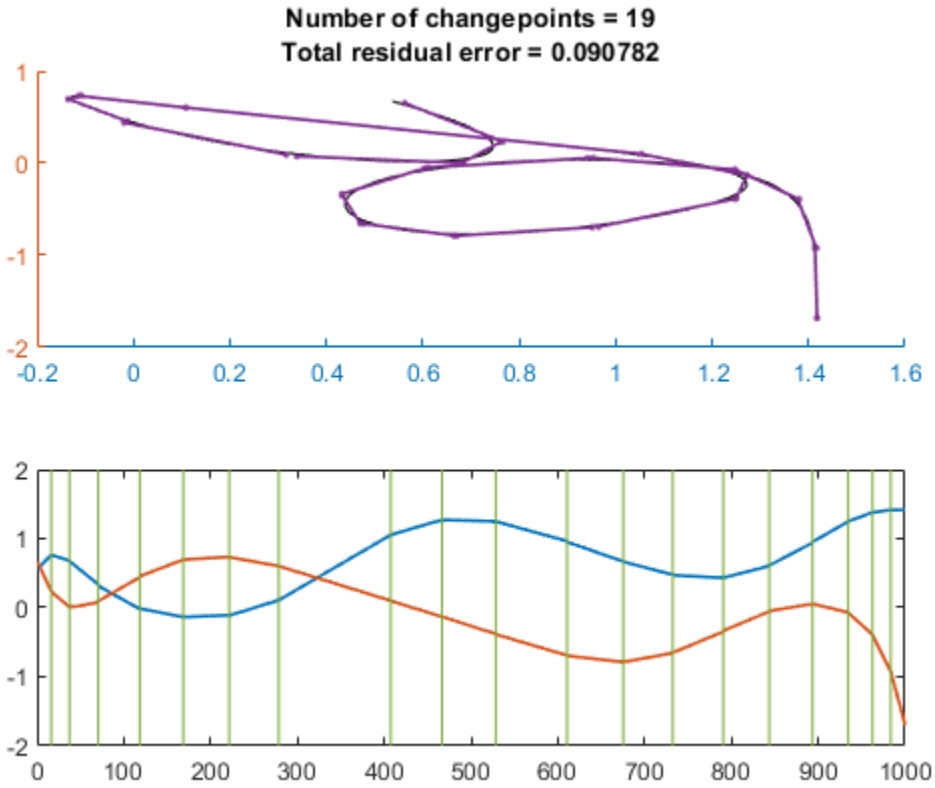
Partition the curve into three segments, such that the points in each segment are at a minimum distance from the segment mean.

```
findchangepts(crv','MaxNumChanges',3)
```



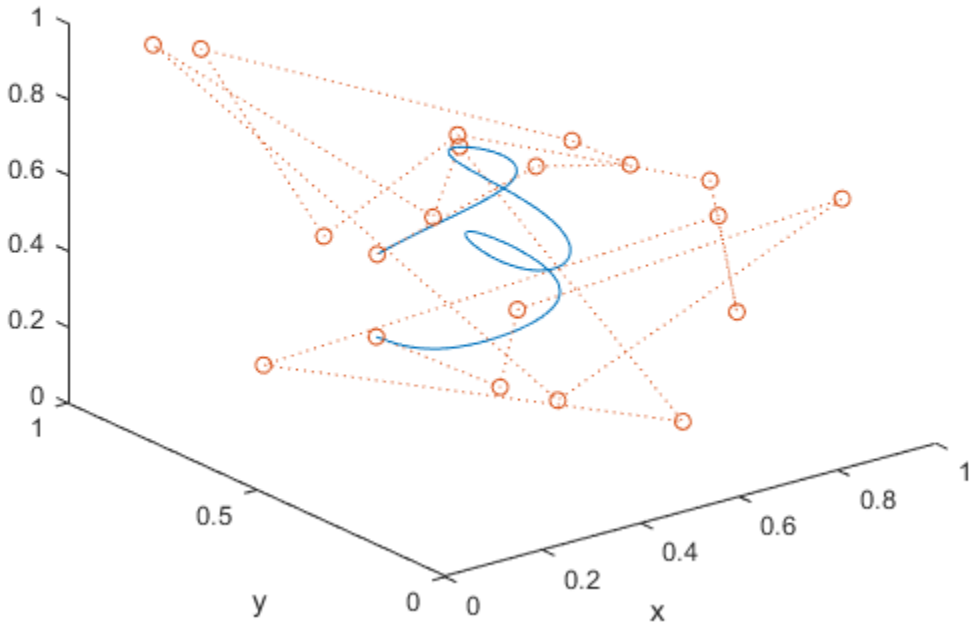
Partition the curve into 20 segments that are best fit by straight lines.

```
findchangepts(crv','Statistic','linear','MaxNumChanges',19)
```



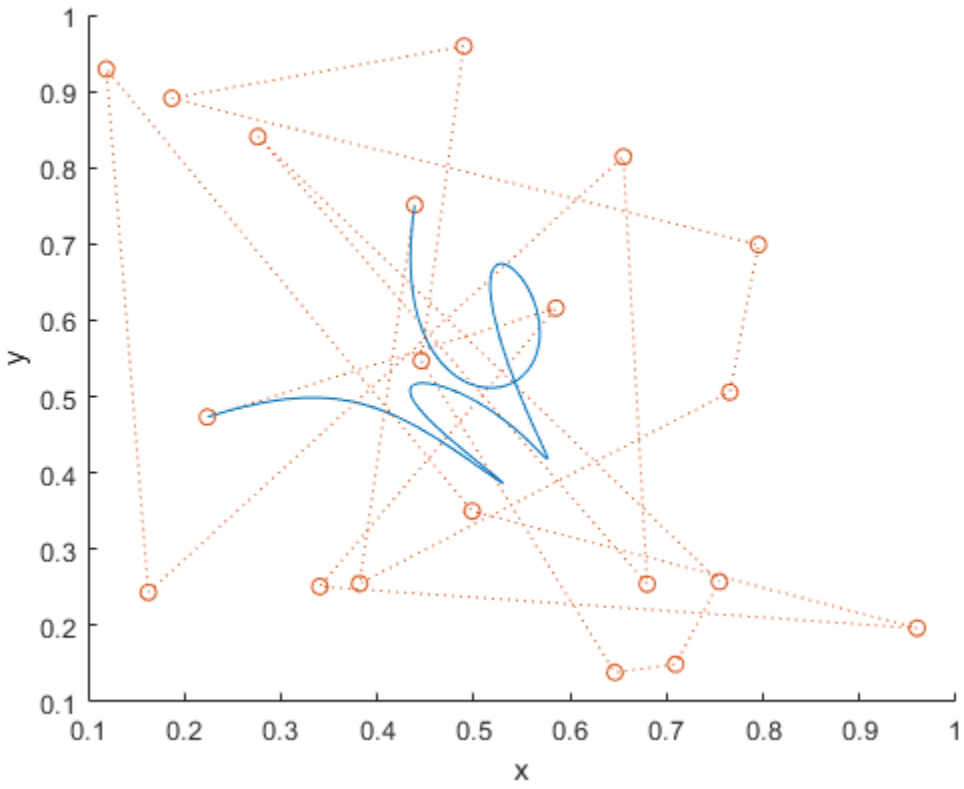
Generate and plot a three-dimensional Bézier curve with 20 random control points.

```
P = rand(m,3);
crv = bin.*pol*P;

plot3(crv(:,1),crv(:,2),crv(:,3),P(:,1),P(:,2),P(:,3),'o:')
xlabel('x')
ylabel('y')
```



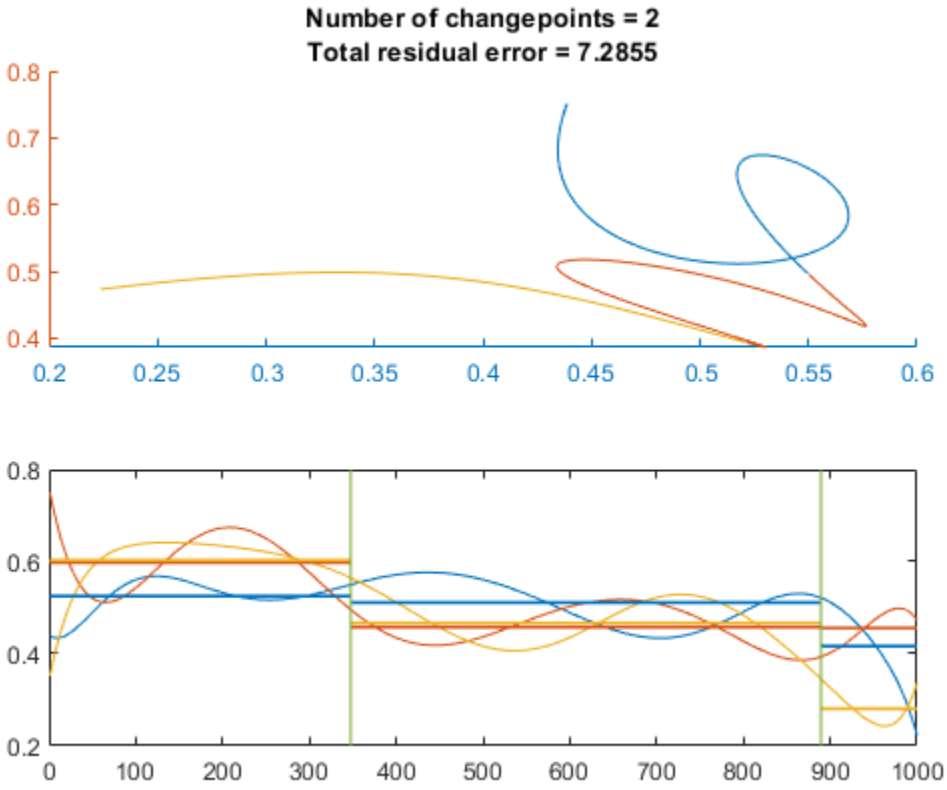
Visualize the curve from above.

```
view([0 0 1])
```



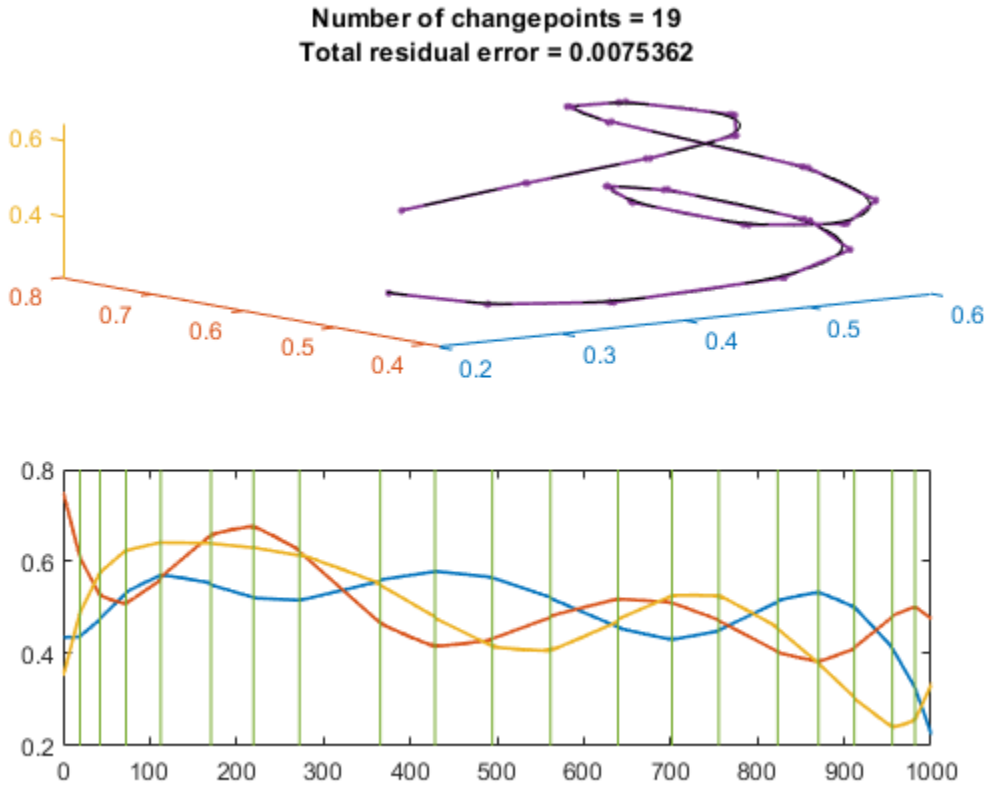
Partition the curve into three segments, such that the points in each segment are at a minimum distance from the segment mean.

```
findchangepts(crv','MaxNumChanges',3)
```



Partition the curve into 20 segments that are best fit by straight lines.

```
findchangepts(crv','Statistic','linear','MaxNumChanges',19)
```



Input Arguments

x — Input signal

real vector

Input signal, specified as a real vector.

Example: `reshape(randn(100,3)+[-3 0 3],1,300)` is a random signal with two abrupt changes in mean.

Example: `reshape(randn(100,3).*[1 20 5],1,300)` is a random signal with two abrupt changes in root-mean-square level.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MaxNumChanges',3,'Statistic',"rms",'MinDistance',20` finds up to three points where the changes in root-mean-square level are most significant and where the points are separated by at least 20 samples.

`MaxNumChanges` — Maximum number of significant changes to return
1 (default) | integer scalar

Maximum number of significant changes to return, specified as an integer scalar. After finding the point with the most significant change, `findchangepts` gradually loosens its search criterion to include more changepoints without exceeding the specified maximum. If any search setting returns more than the maximum, then the function returns nothing. If `MaxNumChanges` is not specified, then the function returns the point with the most significant change. You cannot specify `MinThreshold` and `MaxNumChanges` simultaneously.

Example: `findchangepts([0 1 0])` returns the index of the second sample.

Example: `findchangepts([0 1 0],'MaxNumChanges',1)` returns an empty matrix.

Example: `findchangepts([0 1 0],'MaxNumChanges',2)` returns the indices of the second and third points.

Data Types: `single` | `double`

`Statistic` — Type of change to detect
`"mean"` (default) | `"rms"` | `"std"` | `"linear"`

Type of change to detect, specified as one of these values:

- `"mean"` — Detect changes in mean. If you call `findchangepts` with no output arguments, the function plots the signal, the changepoints, and the mean value of each segment enclosed by consecutive changepoints.
- `"rms"` — Detect changes in root-mean-square level. If you call `findchangepts` with no output arguments, the function plots the signal and the changepoints.
- `"std"` — Detect changes in standard deviation, using Gaussian log-likelihood. If you call `findchangepts` with no output arguments, the function plots the signal, the changepoints, and the mean value of each segment enclosed by consecutive changepoints.
- `"linear"` — Detect changes in mean and slope. If you call `findchangepts` with no output arguments, the function plots the signal, the changepoints, and the line that best fits each portion of the signal enclosed by consecutive changepoints.

Example: `findchangepts([0 1 2 1],'Statistic',"mean")` returns the index of the second sample.

Example: `findchangepts([0 1 2 1],'Statistic',"rms")` returns the index of the third sample.

`MinDistance` — Minimum number of samples between changepoints
integer scalar

Minimum number of samples between changepoints, specified as an integer scalar. If you do not specify this number, then the default is 1 for changes in mean and 2 for other changes.

Example: `findchangepts(sin(2*pi*(0:10)/5),'MaxNumChanges',5,'MinDistance',1)` returns five indices.

Example: `findchangepts(sin(2*pi*(0:10)/5),'MaxNumChanges',5,'MinDistance',3)` returns two indices.

Example: `findchangepts(sin(2*pi*(0:10)/5),'MaxNumChanges',5,'MinDistance',5)` returns no indices.

Data Types: `single` | `double`

`MinThreshold` — Minimum improvement in total residual error

real scalar

Minimum improvement in total residual error for each changepoint, specified as a real scalar that represents a penalty. This option acts to limit the number of returned significant changes by applying the additional penalty to each prospective changepoint. You cannot specify `MinThreshold` and `MaxNumChanges` simultaneously.

Example: `findchangepts([0 1 2],'MinThreshold',0)` returns two indices.

Example: `findchangepts([0 1 2],'MinThreshold',1)` returns one index.

Example: `findchangepts([0 1 2],'MinThreshold',2)` returns no indices.

Data Types: `single` | `double`

Output Arguments

`ipt` — Changepoint locations

vector

Changepoint locations, returned as a vector of integer indices.

`residual` — Residual error

vector

Residual error of the signal against the modeled changes, returned as a vector.

More About

Changepoint Detection

A changepoint is a sample or time instant at which some statistical property of a signal changes abruptly. The property in question can be the mean of the signal, its variance, or a spectral characteristic, among others.

To find a signal changepoint, `findchangepts` employs a parametric global method. The function:

1. Chooses a point and divides the signal into two sections.
2. Computes an empirical estimate of the desired statistical property for each section.
3. At each point within a section, measures how much the property deviates from the empirical estimate. Adds the deviations for all points.
4. Adds the deviations section-to-section to find the total residual error.
5. Varies the location of the division point until the total residual error attains a minimum.

The procedure is clearest when the chosen statistic is the mean. In that case, `findchangepts` minimizes the total residual error from the "best" horizontal level for each section. Given a signal x_1, x_2, ..., x_N, and the subsequence mean and variance

$$\text{mean}([x_m \cdots x_n]) = \frac{1}{n-m+1}\sum_{r=m}^{n} x_r,$$

$$\text{var}([x_m \cdots x_n]) = \frac{1}{n-m+1}\sum_{r=m}^{n}\left(x_r - \text{mean}([x_m \cdots x_n])\right)^2 \equiv \frac{S_{xx}|_m^n}{n-m+1},$$

where the sum of squares

$$S_{xy}|_m^n \equiv \sum_{r=m}^{n}\left(x_r - \text{mean}([x_m \cdots x_n])\right)\left(y_r - \text{mean}([y_m \cdots y_n])\right),$$

`findchangepts` finds k such that

$$\begin{aligned} J &= \sum_{i=1}^{k-1}\left(x_i - \text{mean}([x_1 \cdots x_{k-1}])\right)^2 + \sum_{i=k}^{N}\left(x_i - \text{mean}([x_k \cdots x_N])\right)^2 \\ &= (k-1)\text{var}([x_1 \cdots x_{k-1}]) + (N-k+1)\text{var}([x_k \cdots x_N]) \end{aligned}$$

is smallest. This result can be generalized to incorporate other statistics. `findchangepts` finds k such that

$$J(k) = \sum_{i=1}^{k-1}\Delta\left(x_i; \chi([x_1 \cdots x_{k-1}])\right) + \sum_{i=k}^{N}\Delta\left(x_i; \chi([x_k \cdots x_N])\right)$$

is smallest, given the section empirical estimate χ and the deviation measurement Δ.

Minimizing the residual error is equivalent to maximizing the log likelihood. Given a normal distribution with mean μ and variance σ^2, the log-likelihood for N independent observations is

$$\log\prod_{i=1}^{N}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-(x_i-\mu)^2/2\sigma^2} = -\frac{N}{2}\left(\log 2\pi + \log\sigma^2\right) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(x_i-\mu)^2.$$

- If `Statistic` is specified as `"mean"`, the variance is fixed and the function uses

  $$\begin{aligned} \sum_{i=m}^{n}\Delta\left(x_i; \chi([x_m \cdots x_n])\right) &= \sum_{i=m}^{n}\left(x_i - \text{mean}([x_m \cdots x_n])\right)^2 \\ &= (n-m+1)\text{var}([x_m \cdots x_n]), \end{aligned}$$

  as obtained previously.
- If `Statistic` is specified as `"std"`, the mean is fixed and the function uses

  $$\begin{aligned} \sum_{i=m}^{n}\Delta\left(x_i; \chi([x_m \cdots x_n])\right) &= (n-m+1)\log\sum_{i=m}^{n}\sigma^2([x_m \cdots x_n]) \\ &= (n-m+1)\log\left(\frac{1}{n-m+1}\sum_{i=m}^{n}\left(x_i - \text{mean}([x_m \cdots x_n])\right)^2\right) \\ &= (n-m+1)\log\text{var}([x_m \cdots x_n]). \end{aligned}$$
- If `Statistic` is specified as `"rms"`, the total deviation is the same as for `"std"` but with the mean set to zero:

$$\sum_{i=m}^{n} \Delta\left(x_i; \chi([x_m \ \cdots \ x_n])\right) = (n-m+1)\log\left(\frac{1}{n-m+1}\sum_{r=m}^{n} x_r^2\right).$$

- If `Statistic` is specified as `"linear"`, the function uses as total deviation the sum of squared differences between the signal values and the predictions of the least-squares linear fit through the values. This quantity is also known as the error sum of squares, or SSE. The best-fit line through x_m, x_{m+1}, ..., x_n is

  $$\widehat{x}(t) = \frac{S_{xt}|_m^n}{S_{tt}|_m^n}(t - \text{mean}([t_m \ \cdots \ t_n])) + \text{mean}([x_m \ \cdots \ x_n])$$

  and the SSE is

  $$\sum_{i=m}^{n} \Delta\left(x_i; \chi([x_m \ \cdots \ x_n])\right) = \sum_{i=m}^{n} \left(x_i - \widehat{x}(t_i)\right)^2$$

  $$= S_{xx}|_m^n - \frac{S_{xt}^2|_m^n}{S_{tt}|_m^n}$$

  $$= (n-m+1)\text{var}([x_m \ \cdots \ x_n])$$

  $$- \frac{\left(\sum_{i=m}^{n} (x_i - \text{mean}([x_m \ \cdots \ x_n]))(i - \text{mean}([m \ m+1 \ \cdots \ n]))\right)^2}{(n-m+1)\text{var}([m \ m+1 \ \cdots \ n])}.$$

Signals of interest often have more than one changepoint. Generalizing the procedure is straightforward when the number of changepoints is known. When the number is unknown, you must add a penalty term to the residual error, since adding changepoints always decreases the residual error and results in overfitting. In the extreme case, every point becomes a changepoint and the residual error vanishes. `findchangepts` uses a penalty term that grows linearly with the number of changepoints. If there are K changepoints to be found, then the function minimizes

$$J(K) = \sum_{r=0}^{K-1} \sum_{i=k_r}^{k_{r+1}-1} \Delta\left(x_i; \chi\left(\left[x_{k_r} \ \cdots \ x_{k_{r+1}-1}\right]\right)\right) + \beta K,$$

where k_0 and k_K are respectively the first and the last sample of the signal.

- The proportionality constant, denoted by β and specified in `MinThreshold`, corresponds to a fixed penalty added for each changepoint. `findchangepts` rejects adding additional changepoints if the decrease in residual error does not meet the threshold. Set `MinThreshold` to zero to return all possible changes.
- If you do not know what threshold to use or have a rough idea of the number of changepoints in the signal, specify `MaxNumChanges` instead. This option gradually increases the threshold until the function finds fewer changes than the specified value.

To perform the minimization itself, `findchangepts` uses an exhaustive algorithm based on dynamic programming with early abandonment.

References

[1] Killick, Rebecca, Paul Fearnhead, and Idris A. Eckley. “Optimal detection of changepoints with a linear computational cost.” *Journal of the American Statistical Association*. Vol. 107, No. 500, 2012, pp. 1590–1598.

[2] Lavielle, Marc. “Using penalized contrasts for the change-point problem.” *Signal Processing*. Vol. 85, August 2005, pp. 1501–1510.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you specify input argument `Statistic`, then it must be a compile-time constant.

See Also

`cusum`

Introduced in R2016a

finddelay

Estimate delay(s) between signals

Syntax

```
d = finddelay(x,y)
d = finddelay(x,y,maxlag)
```

Description

`d = finddelay(x,y)` returns an estimate of the delay `d` between input signals `x` and `y`. Delays in `x` and `y` can be introduced by prepending zeros.

`d = finddelay(x,y,maxlag)` uses `maxlag` to find the estimated delay(s) between `x` and `y`.

Examples

X and Y Are Vectors, and maxlag Is Not Specified

The following shows `Y` being delayed with respect to `X` by two samples.

```
X = [1 2 3];
Y = [0 0 1 2 3];
D = finddelay(X,Y)
```

```
D = 2
```

Here is a case of `Y` advanced with respect to `X` by three samples.

```
X = [0 0 0 1 2 3 0 0]';
Y = [1 2 3 0]';
D = finddelay(X,Y)
```

```
D = -3
```

The following illustrates a case where `Y` is aligned with `X` but is noisy.

```
X = [0 0 1 2 3 0];
Y = [0.02 0.12 1.08 2.21 2.95 -0.09];
D = finddelay(X,Y)
```

```
D = 0
```

If `Y` is a periodic version of `X`, the smallest possible delay is returned.

```
X = [0 1 2 3];
Y = [1 2 3 0 0 0 0 1 2 3 0 0];
D = finddelay(X,Y)
```

```
D = -1
```

X Is a Vector, Y Is a Matrix, and maxlag Is a Scalar

`maxlag` is specified as a scalar (same maximum window sizes).

```
X = [0 1 2];
Y = [0 1 0 0;
     1 2 0 0;
     2 0 1 0;
     0 0 2 1];
maxlag = 3;
D = finddelay(X,Y,maxlag)
```

```
D = 1×4

     0    -1     1     1
```

X and Y Are Matrices, and maxlag Is Not Specified

Specify `X` and `Y` of the same size. `finddelay` works column-by-column.

```
X = [0 1 0 0;
     1 2 0 0;
     2 0 1 0;
     1 0 2 1;
     0 0 0 2];
Y = [0 0 1 0;
     1 1 2 0;
     2 2 0 1;
     1 0 0 2;
     0 0 0 0];
D = finddelay(X,Y)
```

```
D = 1×4

     0     1    -2    -1
```

Repeat the computation, but now add an extra row of zeros as the second row of `Y`.

```
Y = [0 0 1 0;
     0 0 0 0;
     1 1 2 0;
     2 2 0 1;
     1 0 0 2;
     0 0 0 0];
D = finddelay(X,Y)
```

```
D = 1×4

     1     2    -1     0
```

X and Y Are Matrices, and maxlag Is Specified

Create two multichannel signals, X and Y, such that each channel of Y has a delayed identical copy of each channel of X.

```
X = [1 3 2 0 0 0 0 0;
     0 0 0 0 0 1 3 2]';

Y = [0 0 0 1 3 2;
     1 3 2 0 0 0]';
```

Compute the column-by-column delays. Set a maximum correlation window size of 8 for each channel.

```
maxlag = [8 8];
D = finddelay(X,Y,maxlag)
```

```
D = 1×2

     3    -5
```

Decrease the correlation window size to 3 for the first channel and 5 for the second.

```
maxlag = [3 5];
D = finddelay(X,Y,maxlag)
```

```
D = 1×2

     3    -5
```

Increase the correlation window size to 5 for the first channel and decrease it to 3 for the second.

```
maxlag = [5 3];
D = finddelay(X,Y,maxlag)
```

```
D = 1×2

     3    -3
```

Input Arguments

x — Reference input

vector | matrix

Reference input, specified as a vector or a matrix.

y — Input signal

vector | matrix

Input signal, specified as a vector or a matrix.

maxlag — Maximum correlation window size

`max(length(x),length(y)) − 1` | `max(size(x,1),size(y,1)) − 1` | `max(length(x),size(y,1)) − 1` | `max(size(x,1),length(y)) − 1` | integer scalar | integer vector

Maximum correlation window size, specified as an integer scalar or vector. If any element of `maxlag` is negative, it is replaced by its absolute value. If any element of `maxlag` is not integer-valued, or is complex, `Inf`, or `NaN`, then `finddelay` returns an error.

Output Arguments

`d` — delay

integer scalar | integer value

Delay between input signals, returned as an integer scalar or vector. If `y` is delayed with respect to `x`, then `d` is positive. If `y` is advanced with respect to `x`, then `d` is negative. If several delays are possible, as in the case of periodic signals, the delay with the smallest absolute value is returned. In the case that both a positive and a negative delay with the same absolute value are possible, the positive delay is returned.

If `x` is a matrix of size M_X-by-N_X ($M_X > 1$ and $N_X > 1$) and `y` is a matrix of size M_Y-by-N_Y ($M_Y > 1$ and $N_Y > 1$ and $N_Y > 1$), returns a row vector `d` of estimated delays between each column of `x` and the corresponding column of `y`. With this usage the number of columns of `x` must be equal to the number of columns of `y` (i.e., $N_X = N_Y$).

Tips

- `x` and `y` need not be exact delayed copies of each other, as `finddelay(x,y)` returns an estimate of the delay via cross-correlation. However this estimated delay has a useful meaning only if there is sufficient correlation between delayed versions of `x` and `y`.
- The calculation of the vector of estimated delays, `d`, depends on `x`, `y`, and `maxlag` as shown in the following table.

`maxlag`	`X`	`Y`	`D` is calculated by...
Integer-valued scalar	Row or column vector or matrix	Row or column vector or matrix	Cross-correlating the columns of `X` and `Y` over a range of lags –`maxlag`:`maxlag`.
Integer-valued row or column vector	Row or column vector of length $L_X \geq 1$	Matrix of size M_Y-by-N_Y ($M_Y > 1$, $N_Y > 1$)	Cross-correlating `X` and column j of `Y` over a range of lags –`maxlag`(j):`maxlag`(j), for $j = 1{:}N_Y$.
Integer-valued row or column vector	Matrix of size M_X-by-N_X ($M_X > 1$, $N_X > 1$)	Row or column vector of length $L_Y \geq 1$	Cross-correlating column j of `X` and `Y` over a range of lags –`maxlag`(j):`maxlag`(j), for $j = 1{:}N_X$.
Integer-valued row or column vector	Matrix of size M_X-by-N_X ($M_X > 1$, $N_X > 1$)	Matrix of size M_Y-by-N_Y ($M_Y > 1$, $N_Y = N_X > 1$)	Cross-correlating column j of `X` and column j of `Y` over a range of lags –`maxlag`(j):`maxlag`(j), for $j = 1{:}N_Y$.

- If you wish to treat a row vector `x` of length L_X as comprising one sample from L_X different channels, you need to append one or more rows of zeros to `x` so that it appears as a matrix. Then each column of `x` will be considered a channel.

  For example, `x = [1 1 1 1]` is considered a single channel comprising four samples. To treat it as four different channels, each channel comprising one sample, define a new matrix `xm`:

  Each column of `xm` corresponds to a single channel, each one containing the samples `1` and `0`.

```
xm = [1 1 1 1;
      0 0 0 0];
```

Algorithms

The `finddelay` function uses the `xcorr` function to determine the cross-correlation between each pair of signals at all possible lags specified by the user. The normalized cross-correlation between each pair of signals is then calculated. The estimated delay is given by the negative of the lag for which the normalized cross-correlation has the largest absolute value.

If more than one lag leads to the largest absolute value of the cross-correlation, such as in the case of periodic signals, the delay is chosen as the negative of the smallest (in absolute value) of such lags.

Pairs of signals need not be exact delayed copies of each other. However, the estimated delay has a useful meaning only if there is sufficient correlation between at least one pair of the delayed signals.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`alignsignals` | `dtw` | `edr` | `findsignal` | `xcorr`

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`. The peaks are output in order of occurrence. 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 arguments in addition to any of the input arguments in previous syntaxes.

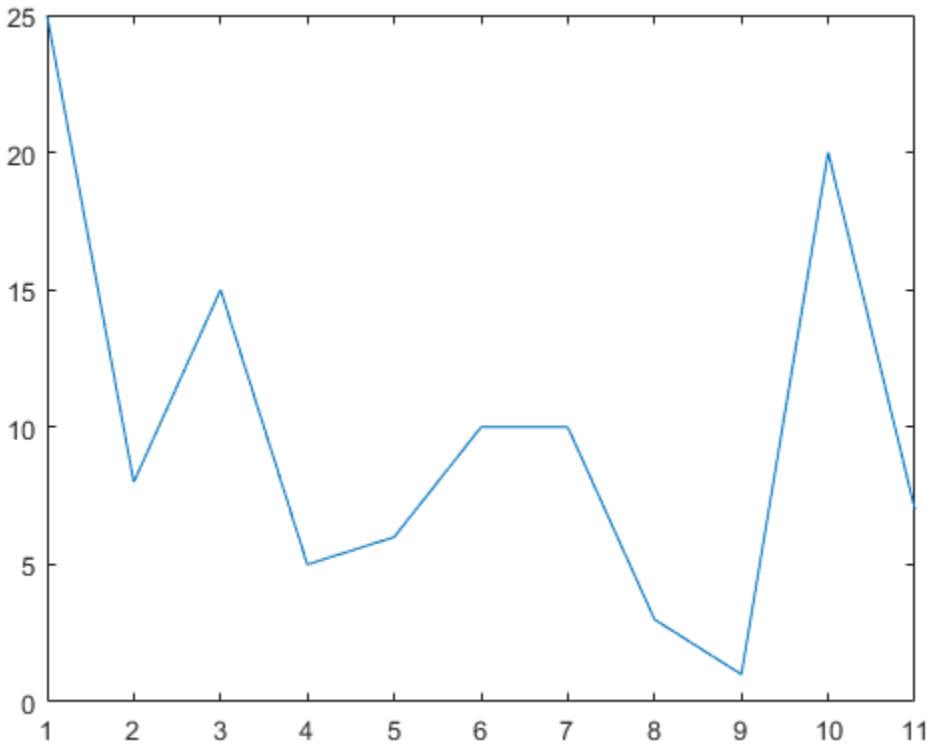
`findpeaks(___)` without output arguments plots the signal and overlays the peak values.

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)
```



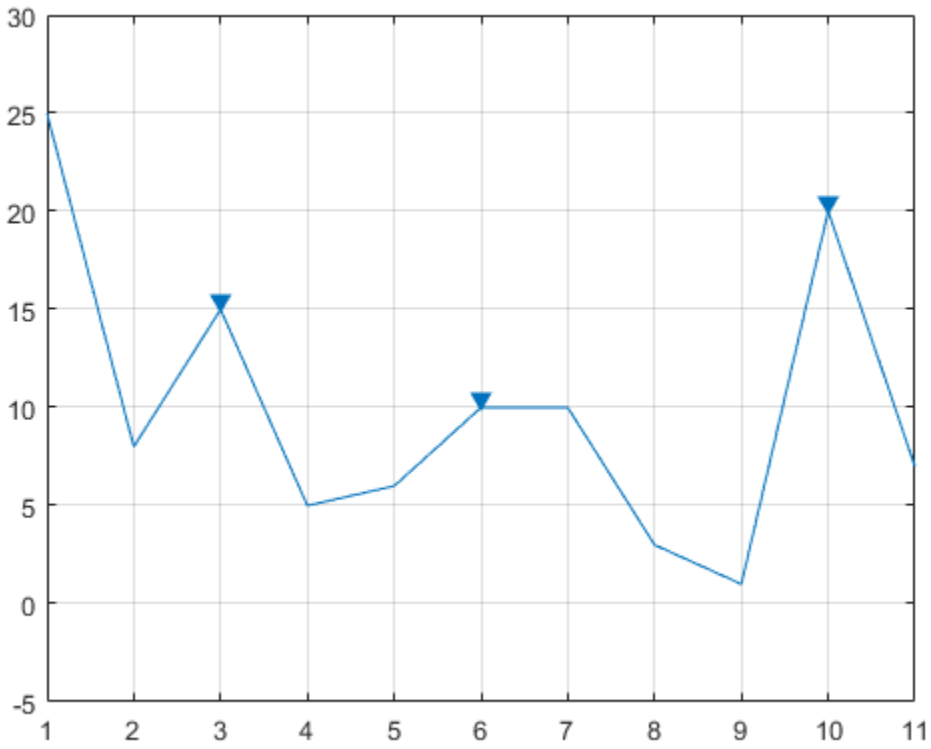
Find the local maxima. The peaks are output in order of occurrence. The first sample is not included despite being the maximum. For the flat peak, the function returns only the point with lowest index.

```
pks = findpeaks(data)
```

```
pks = 1×3

    15    10    20
```

Use `findpeaks` without output arguments to display the peaks.

```
findpeaks(data)
```



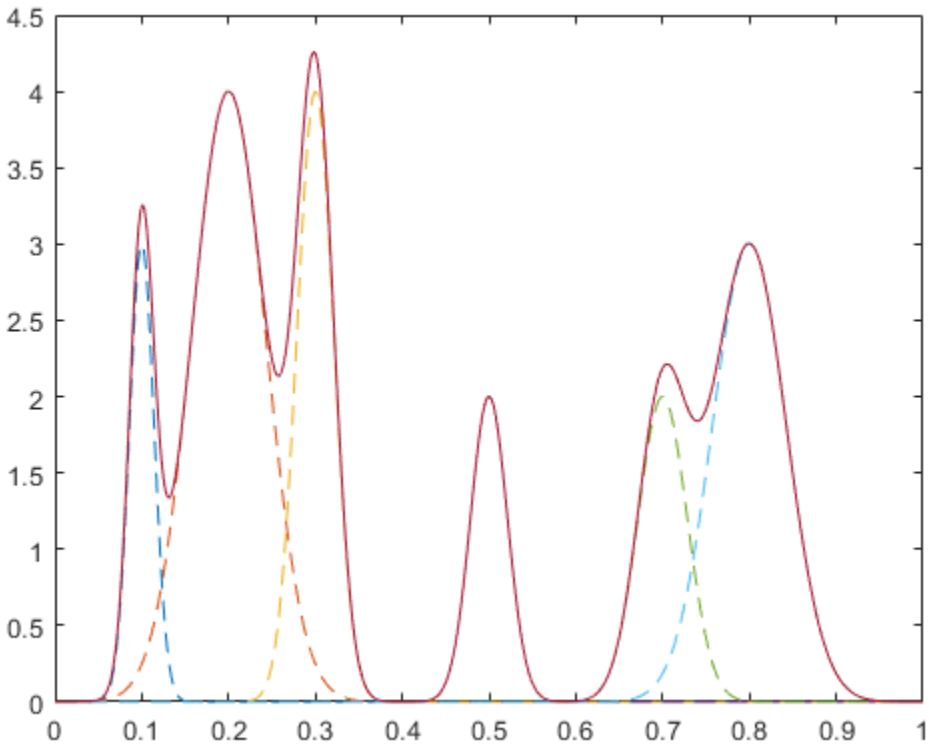
Find Peaks and Their Locations

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 = [3 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)
```



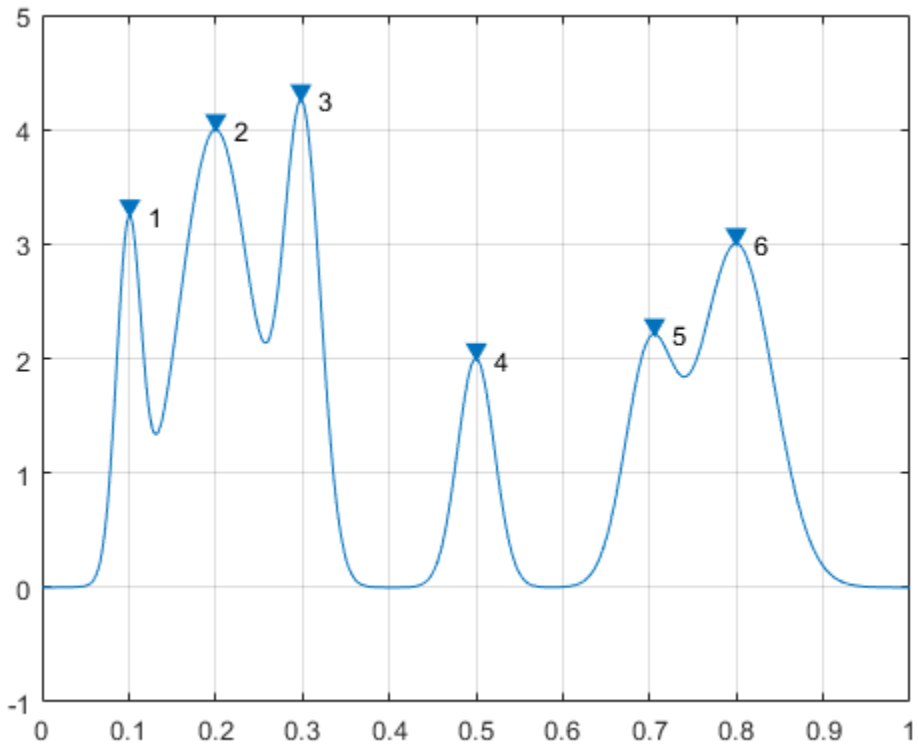
Use `findpeaks` with default settings to find the peaks of the signal and their locations.

```
[pks,locs] = findpeaks(PeakSig,x);
```

Plot the peaks using `findpeaks` and label them.

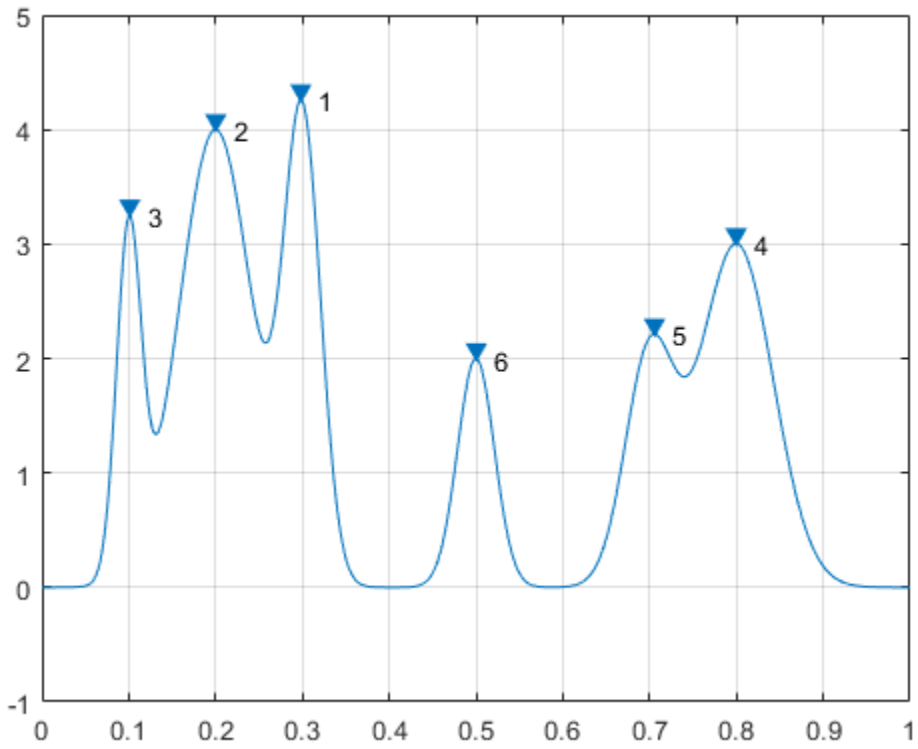
```
findpeaks(PeakSig,x)
```

```
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))'))
```



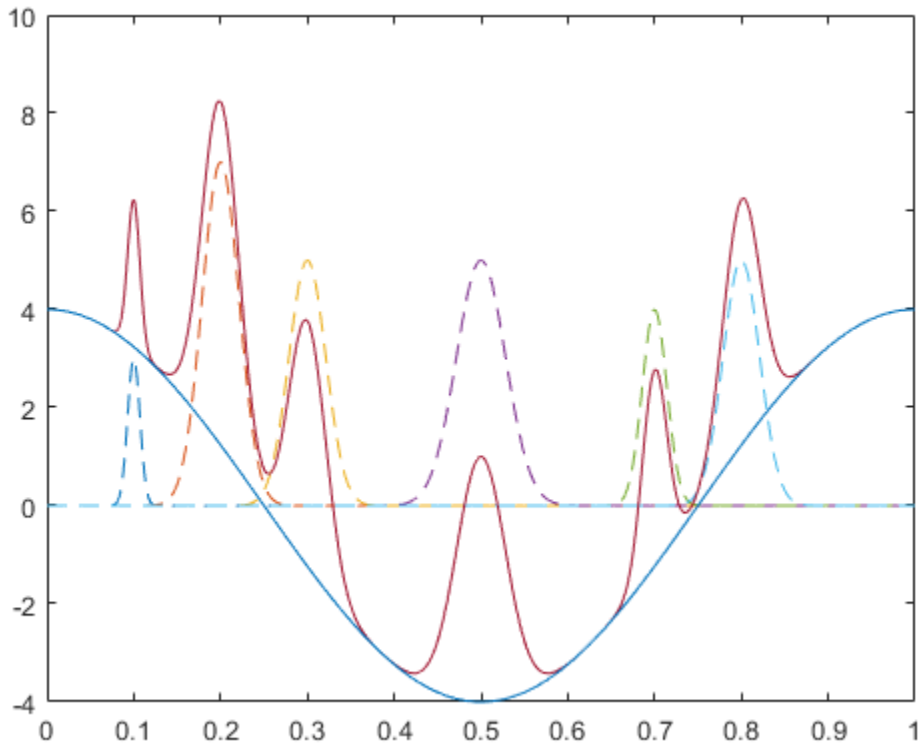
Peak Prominences

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;
```

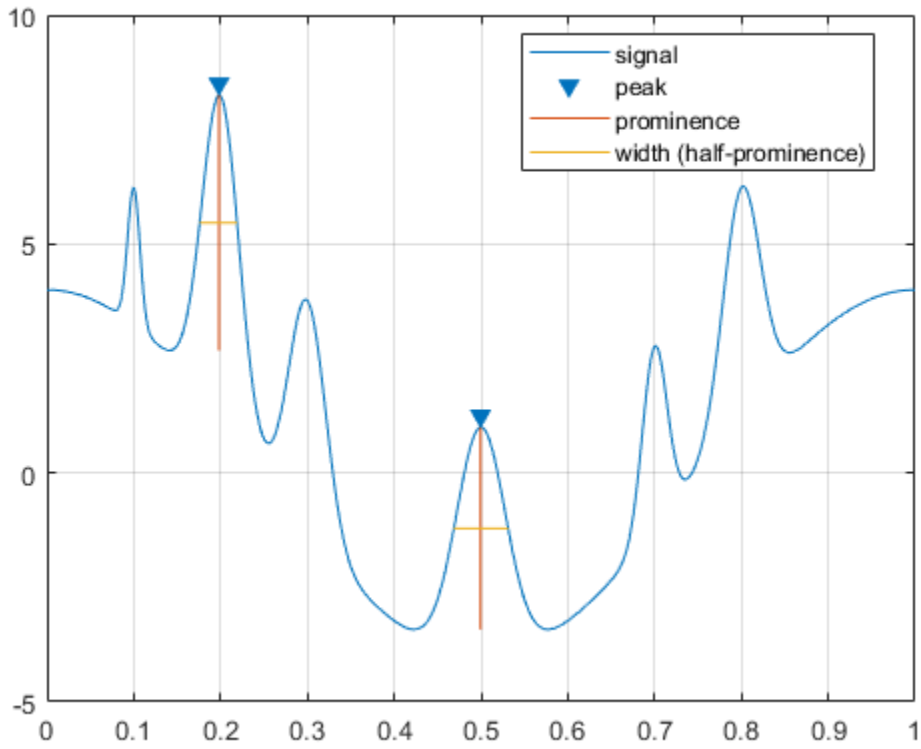
Plot the individual curves and their sum.

```
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
```

```
widths = 1×6

    0.0154    0.0431    0.0377    0.0625    0.0274    0.0409
```

```
proms
```

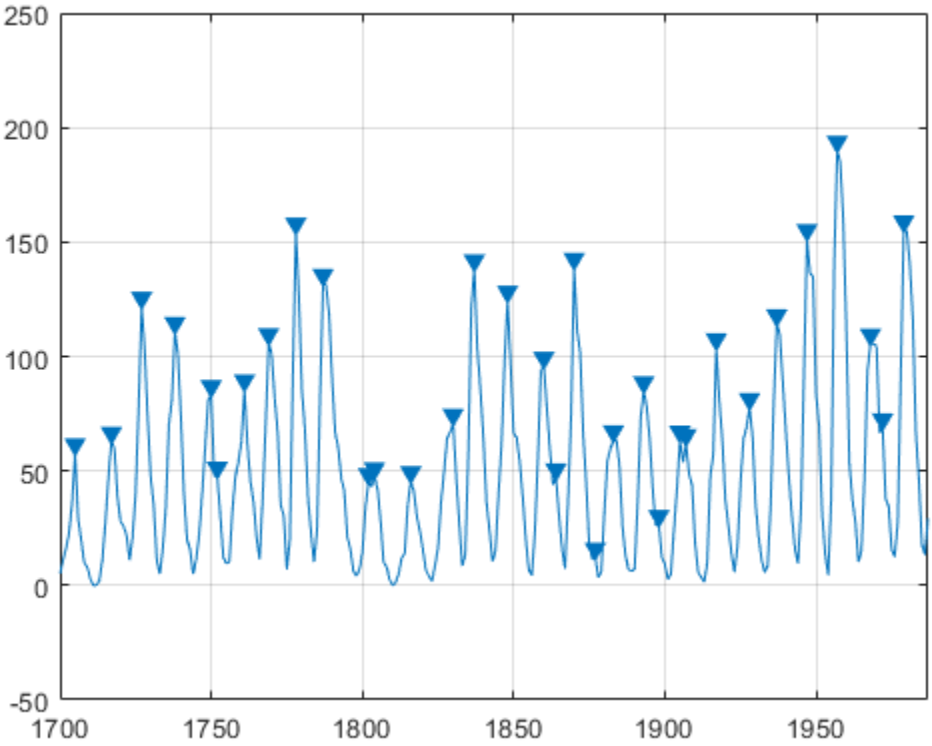
```
proms = 1×6

    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.

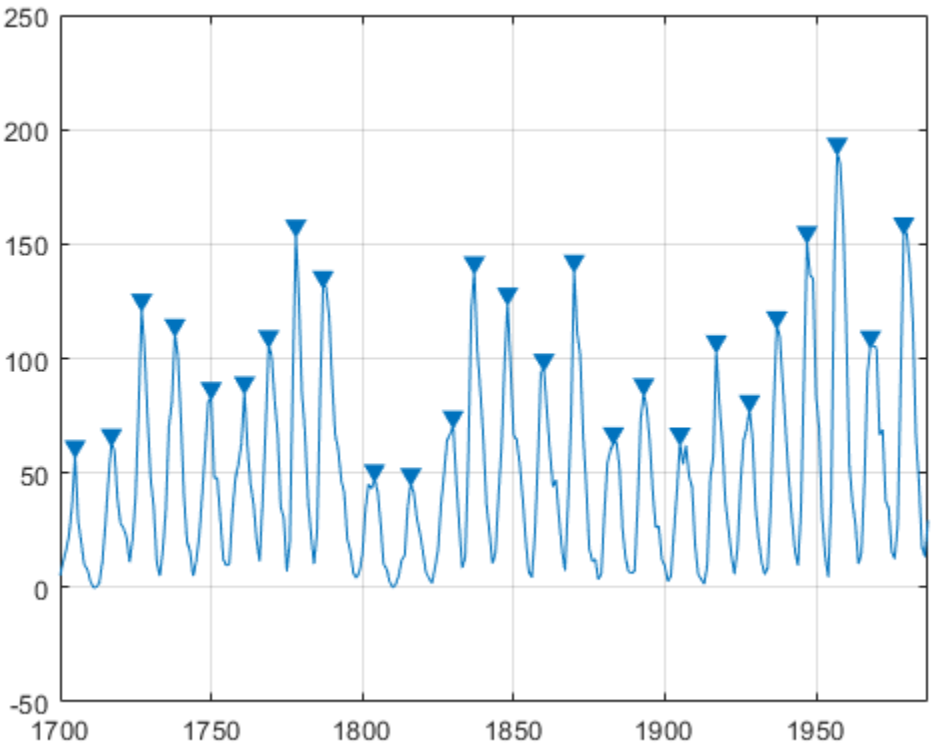
Load the file `sunspot.dat`, which contains the average number of sunspots observed every year from 1700 to 1987. Find and plot the maxima.

```
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))
```

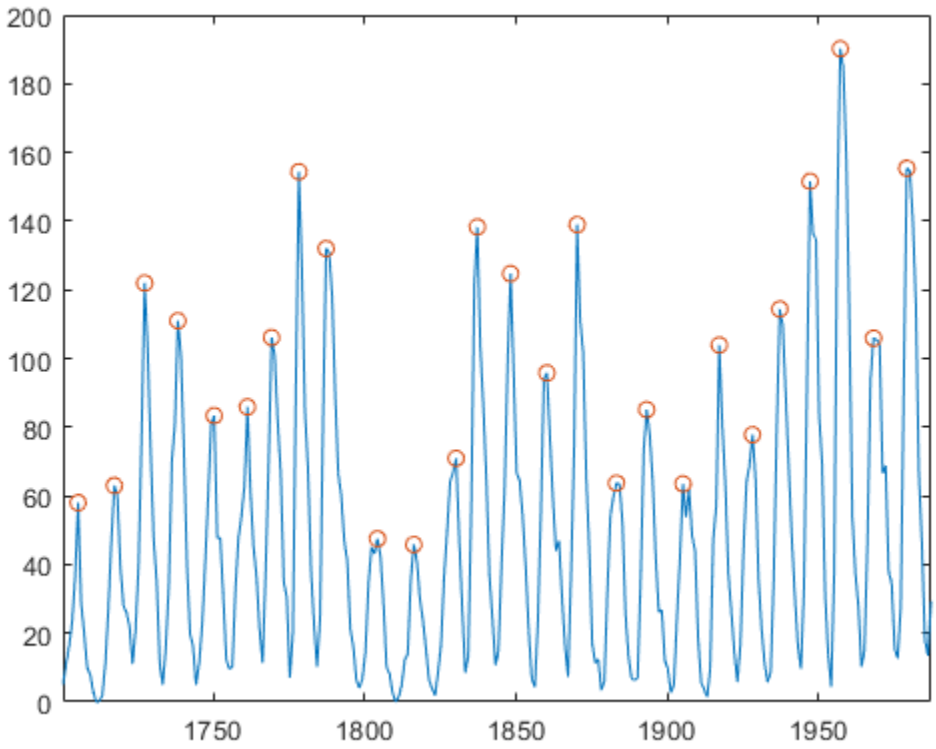
```
meanCycle = 10.9600
```

Create a `datetime` array using the year data. Assume the sunspots were counted every year on March 20th, close to the vernal equinox. Find the peak sunspot years. Use the `years` function to specify the minimum peak separation as a `duration`.

```
ty = datetime(year,3,20);
```

```
[pk,lk] = findpeaks(avSpots,ty,'MinPeakDistance',years(6));
```

```
plot(ty,avSpots,lk,pk,'o')
```



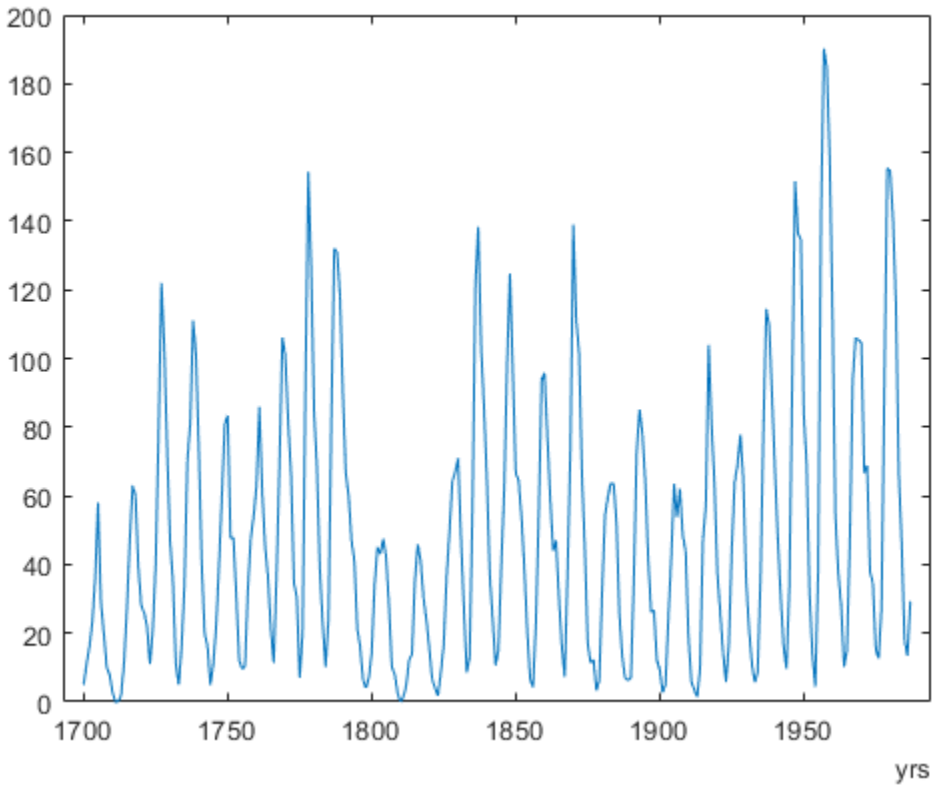
Compute the mean sunspot cycle using `datetime` functionality.

```
dttmCycle = years(mean(diff(lk)))
```

```
dttmCycle = 10.9600
```

Create a timetable with the data. Specify the time variable in years. Plot the data. Show the last five entries of the timetable.

```
TT = timetable(years(year),avSpots);
plot(TT.Time,TT.Variables)
```




```
entries = TT(end-4:end,:)
```

```
entries=5×1 timetable
      Time      avSpots
    ________    _______

    1983 yrs     66.6
    1984 yrs     45.9
    1985 yrs     17.9
    1986 yrs     13.4
    1987 yrs     29.3
```

Constrain Peak Features

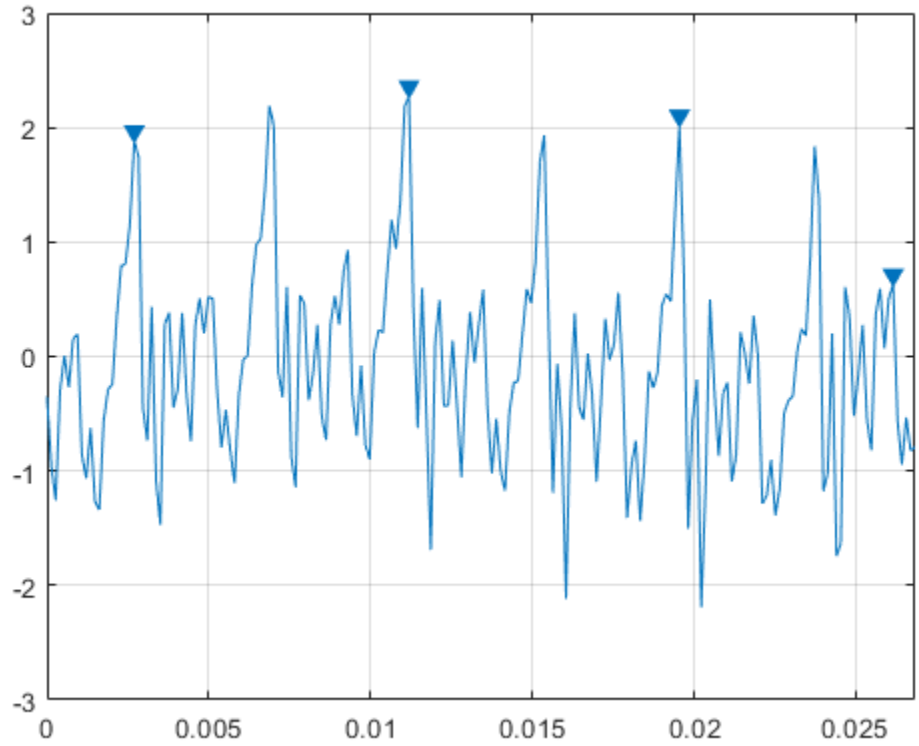
Load an audio signal sampled at 7418 Hz. Select 200 samples.

```
load mtlb
select = mtlb(1001:1200);
```

Find the peaks that are separated by at least 5 ms.

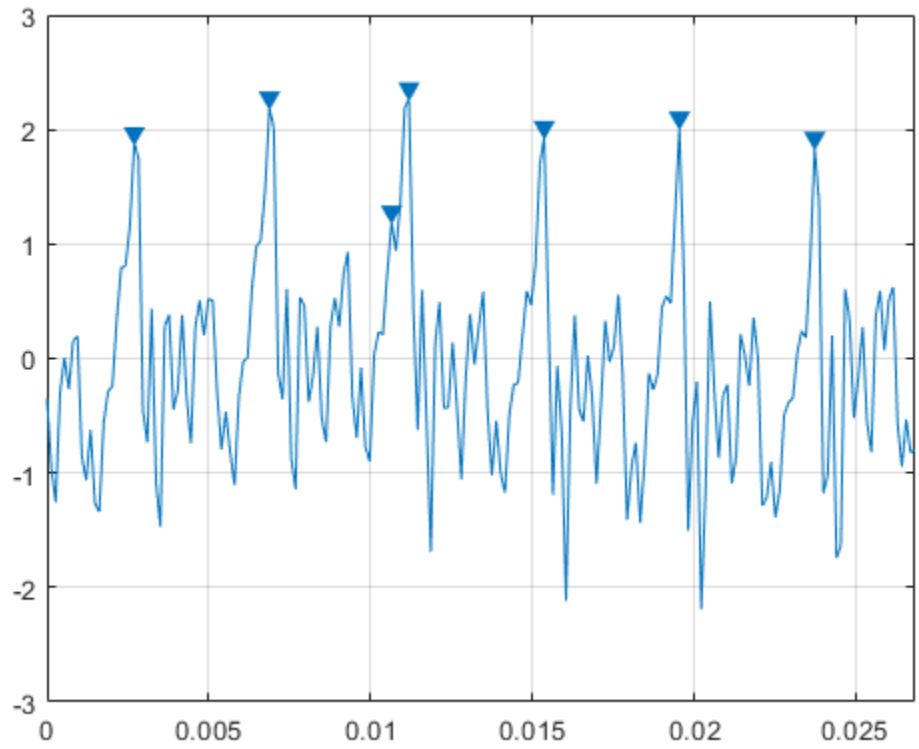
To apply this constraint, `findpeaks` chooses the tallest peak in the signal and eliminates all peaks within 5 ms of it. The function then repeats the procedure for the tallest remaining peak and iterates until it runs out of peaks to consider.

```
findpeaks(select,Fs,'MinPeakDistance',0.005)
```



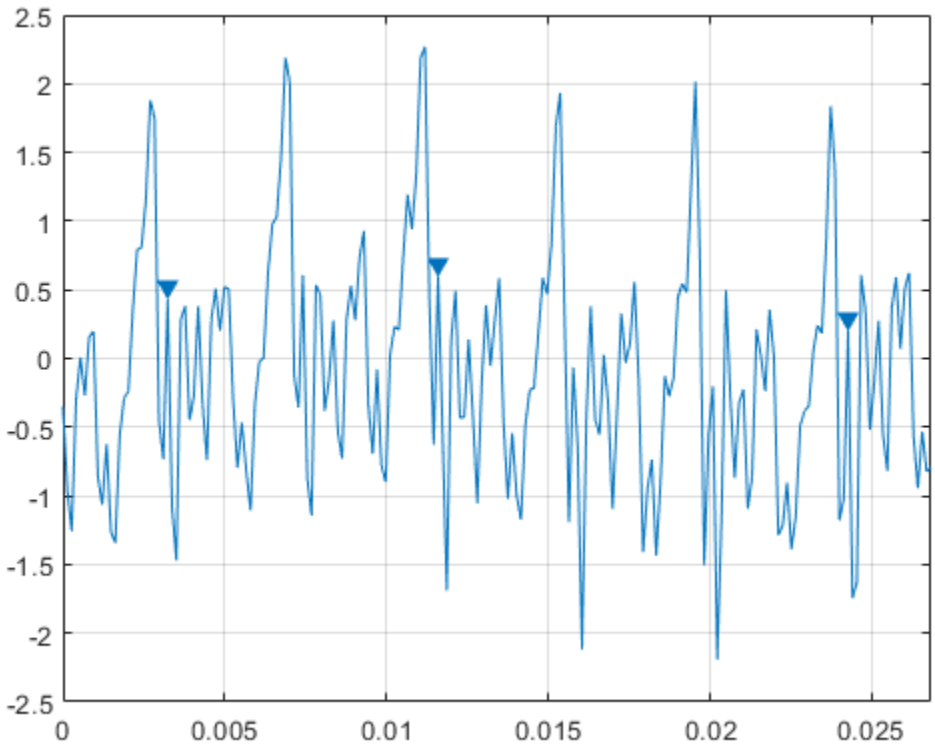
Find the peaks that have an amplitude of at least 1 V.

```
findpeaks(select,Fs,'MinPeakHeight',1)
```



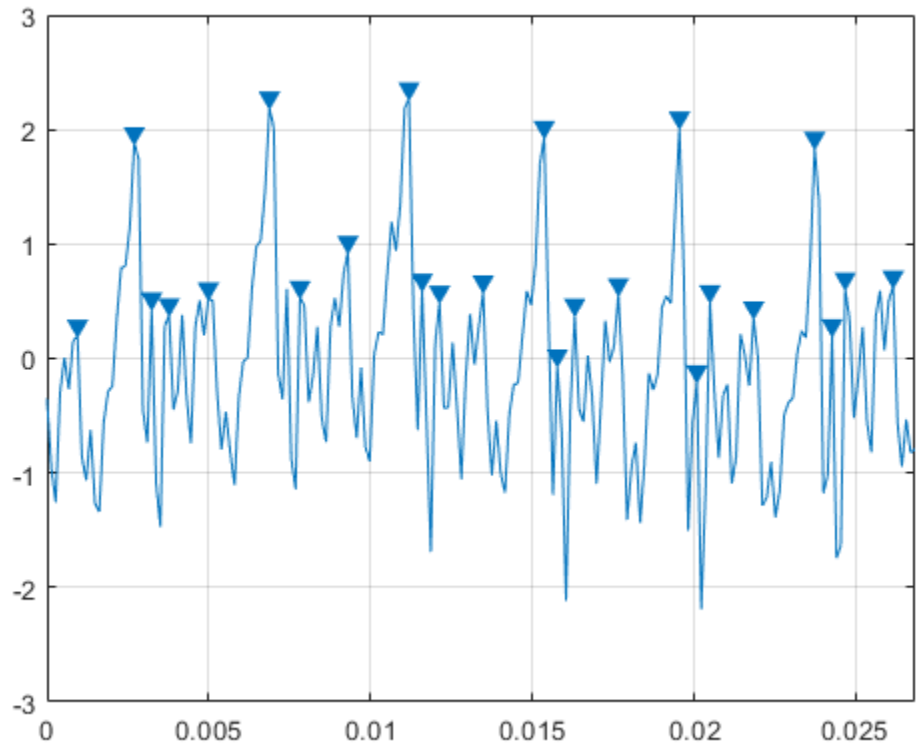
Find the peaks that are at least 1 V higher than their neighboring samples.

```
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

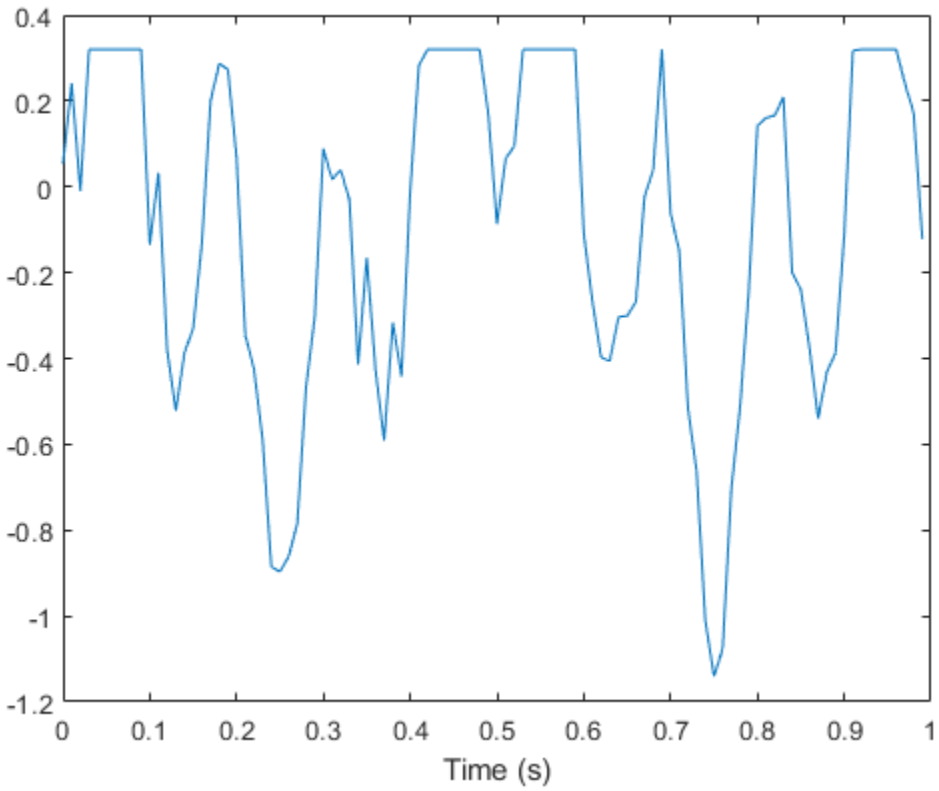
Sensors can return clipped readings if the data are larger than a given saturation point. You can choose to disregard these peaks as meaningless or incorporate them to your analysis.

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. The signal is sampled for one second at a rate of 100 Hz. 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;
```

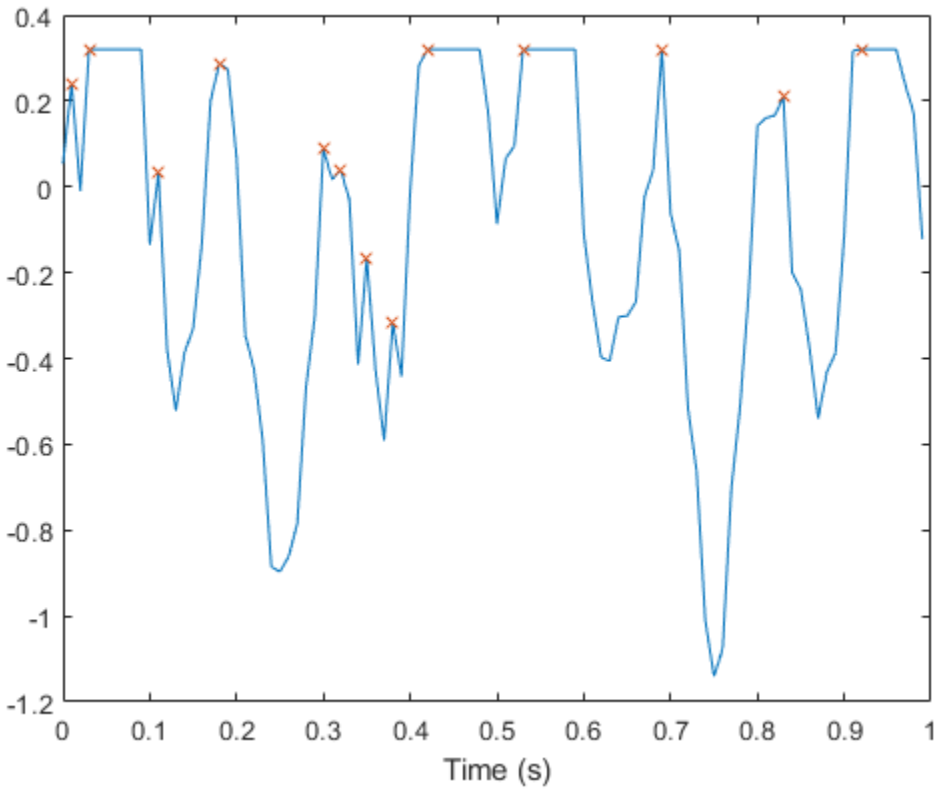
Simulate a saturated measurement by truncating every reading that is greater than a specified bound of 0.32. Plot the saturated signal.

```
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.

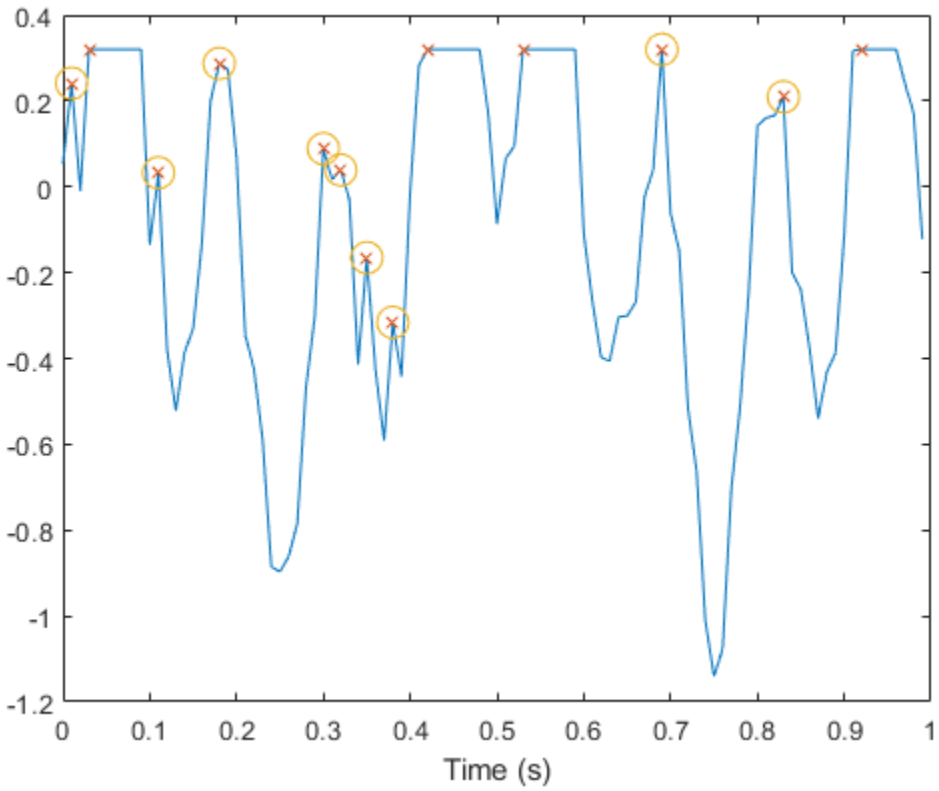
```
[pk,lc] = findpeaks(s,t);

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)
```



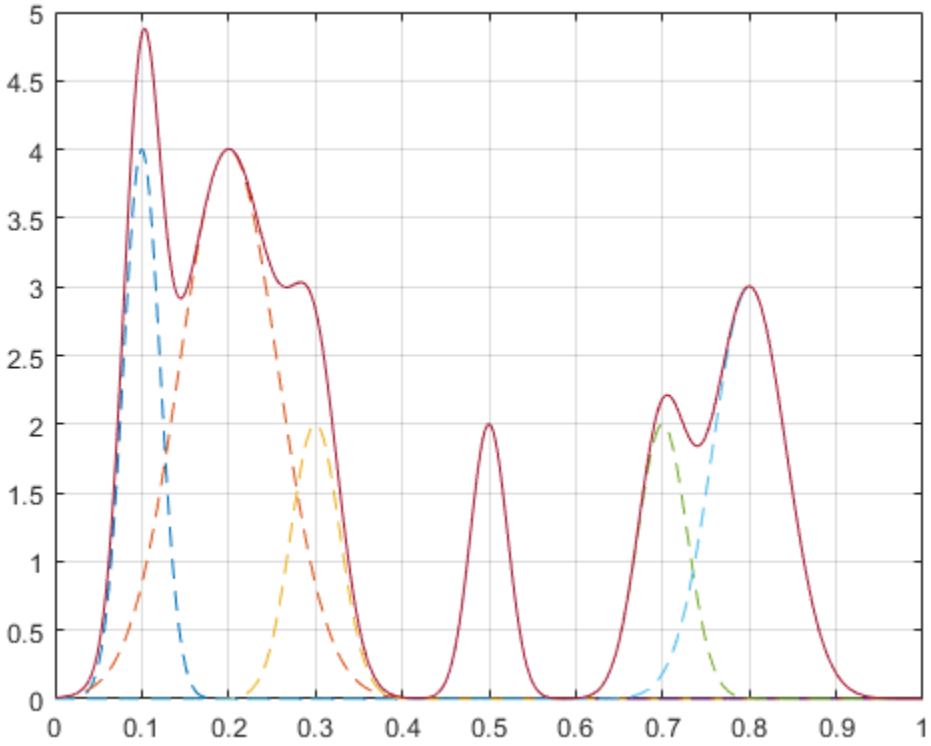
Determine Peak Widths

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);
```

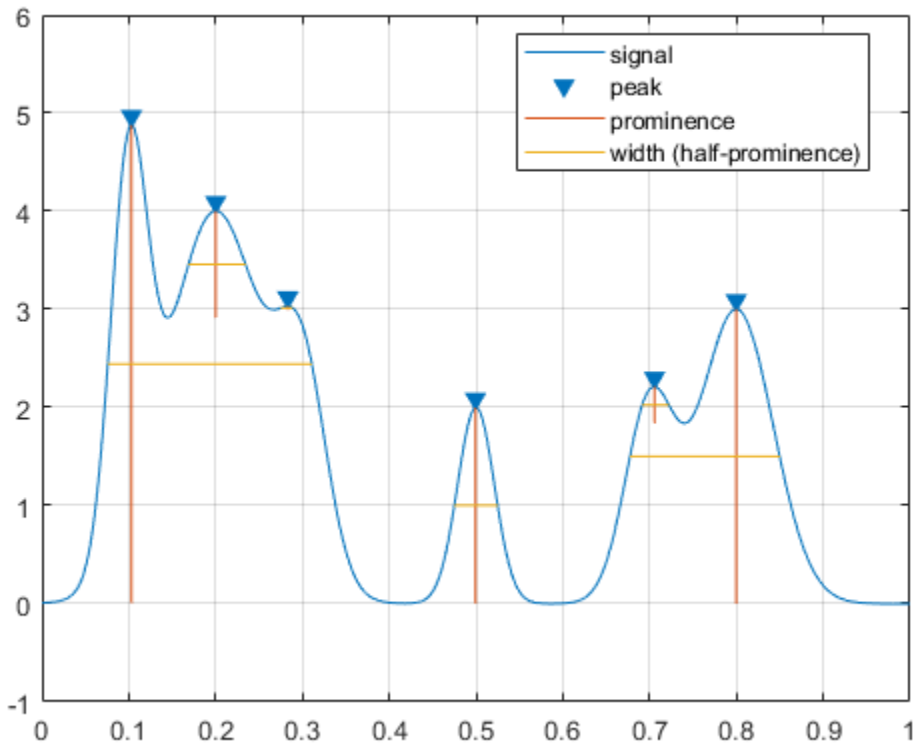
Plot the individual curves and their sum.

```
plot(x,Gauss,'--',x,PeakSig)
grid
```



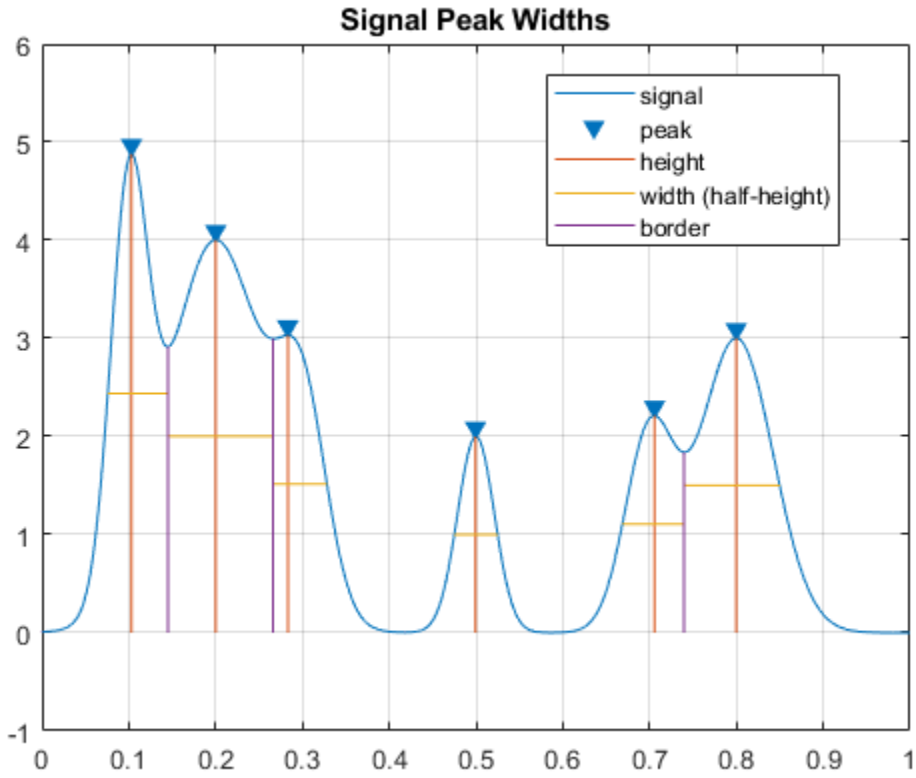
Measure the widths of the peaks using the half prominence as reference.

```
findpeaks(PeakSig,x,'Annotate','extents')
```



Measure the widths again, this time using the half height as reference.

```
findpeaks(PeakSig,x,'Annotate','extents','WidthReference','halfheight')
title('Signal Peak Widths')
```



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: `double` | `single`

`x` — Locations

vector | `datetime` array

Locations, specified as a vector or a `datetime` array. `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: `double` | `single` | `datetime`

`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: `double` | `single`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 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: `double` | `single`

`SortStr` — Peak sorting

`'none'` (default) | `'ascend'` | `'descend'`

Peak sorting, specified as 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.

`MinPeakHeight` — Minimum peak height

`-Inf` (default) | real scalar

Minimum peak height, specified as 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: `double` | `single`

`MinPeakProminence` — Minimum peak prominence

0 (default) | real scalar

Minimum peak prominence, specified as a real scalar. Use this argument to have `findpeaks` return only those peaks that have a relative importance of at least `'MinPeakProminence'`. For more information, see “Prominence” on page 1-801.

Data Types: `double` | `single`

`Threshold` — Minimum height difference

0 (default) | nonnegative real scalar

Minimum height difference between a peak and its neighbors, specified as 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: `double` | `single`

`MinPeakDistance` — Minimum peak separation
0 (default) | positive real scalar

Minimum peak separation, specified as 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 `x` is a `datetime` array, then specify `'MinPeakDistance'` as a `duration` scalar or as a numeric scalar expressed in days.
- 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: `double` | `single` | `duration`

`WidthReference` — Reference height for width measurements
`'halfprom'` (default) | `'halfheight'`

Reference height for width measurements, specified as 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 1-801 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.

`MinPeakWidth` — Minimum peak width
0 (default) | positive real scalar

Minimum peak width, specified as 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 `x` is a `datetime` array, then specify `'MinPeakWidth'` as a `duration` scalar or as a numeric scalar expressed in days.
- If you specify a sample rate, `Fs`, then `'MinPeakWidth'` must be expressed in units of time.
- If you specify neither `x` nor `Fs`, then `'MinPeakWidth'` must be expressed in units of samples.

Data Types: `double` | `single` | `duration`

`MaxPeakWidth` — Maximum peak width
`Inf` (default) | positive real scalar

Maximum peak width, specified as 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 `x` is a `datetime` array, then specify `'MaxPeakWidth'` as a `duration` scalar or as a numeric scalar expressed in days.
- If you specify a sample rate, `Fs`, then `'MaxPeakWidth'` must be expressed in units of time.
- If you specify neither `x` nor `Fs`, then `'MaxPeakWidth'` must be expressed in units of samples.

Data Types: `double` | `single` | `duration`

`Annotate` — Plot style
`'peaks'` (default) | `'extents'`

Plot style, specified as 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.

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.

`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 numeric vector of time instants with a time difference of 1/`Fs` between consecutive samples.
- If you specify neither `x` nor `Fs`, then `locs` is a vector of integer indices.

`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.

`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 1-801 for more information.

More About

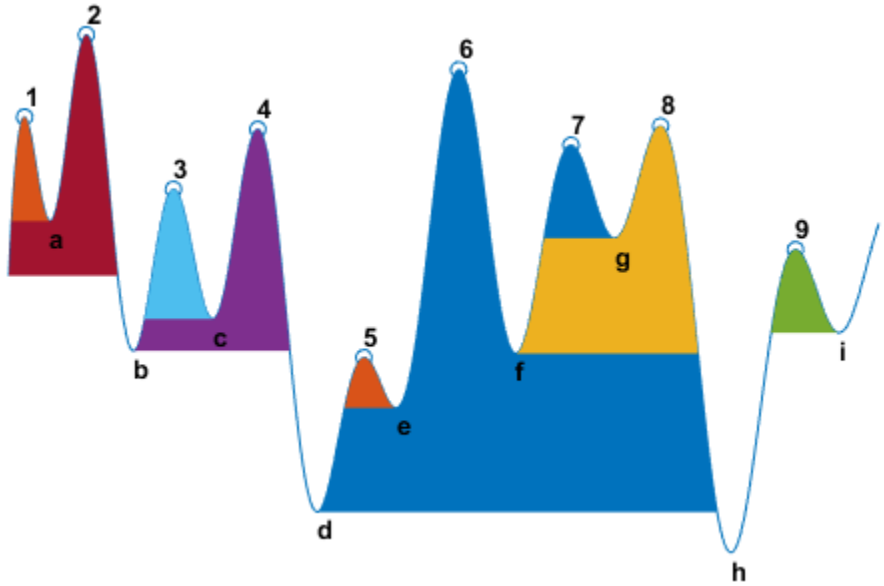
Prominence

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**
8	Crossing due to peak **6**	Right end	**f**	**h**	**f**
9	Crossing due to peak **8**	Crossing due to right endpoint	**h**	**i**	**i**

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`fminbnd` | `fminsearch` | `fzero` | `islocalmax` | `islocalmin` | `max`

Topics

“Peak Analysis”
“Find Peaks in Data”

Introduced in R2007b

findsignal

Find signal location using similarity search

Syntax

```
[istart,istop,dist] = findsignal(data,signal)
[istart,istop,dist] = findsignal(data,signal,Name,Value)

findsignal( ___ )
```

Description

`[istart,istop,dist] = findsignal(data,signal)` returns the start and stop indices of a segment of the data array, `data`, that best matches the search array, `signal`. The best-matching segment is such that `dist`, the squared Euclidean distance between the segment and the search array, is smallest. If `data` and `signal` are matrices, then `findsignal` finds the start and end columns of the region of `data` that best matches `signal`. In that case, `data` and `signal` must have the same number of rows.

`[istart,istop,dist] = findsignal(data,signal,Name,Value)` specifies additional options using name-value pair arguments. Options include the normalization to apply, the number of segments to report, and the distance metric to use.

`findsignal( ___ )` without output arguments plots `data` and highlights any identified instances of `signal`.

- If the arrays are real vectors, the function displays `data` as a function of sample number.
- If the arrays are complex vectors, the function displays `data` on an Argand diagram.
- If the arrays are real matrices, the function uses `imagesc` to display `signal` on a subplot and `data` with the highlighted regions on another subplot.
- If the arrays are complex matrices, the function plots their real and imaginary parts in the top and bottom half of each image.

Examples

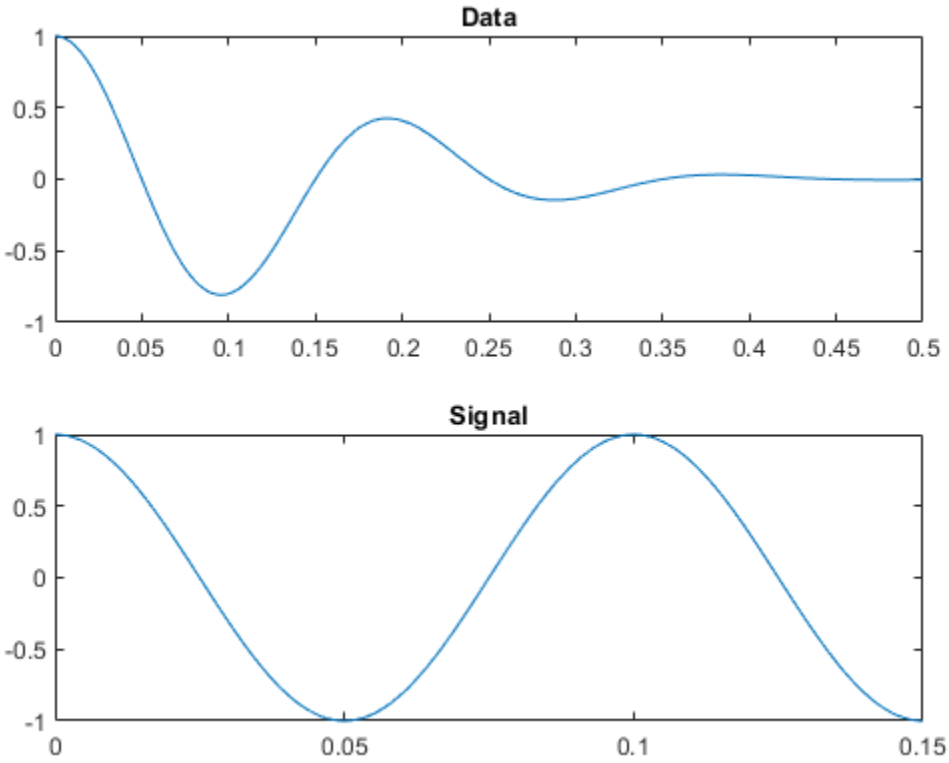
Locate Signal in Data

Generate a data set consisting of a 5 Hz Gaussian pulse with 50% bandwidth, sampled for half a second at a rate of 1 kHz.

```
fs = 1e3;

t = 0:1/fs:0.5;
data = gauspuls(t,5,0.5);
```

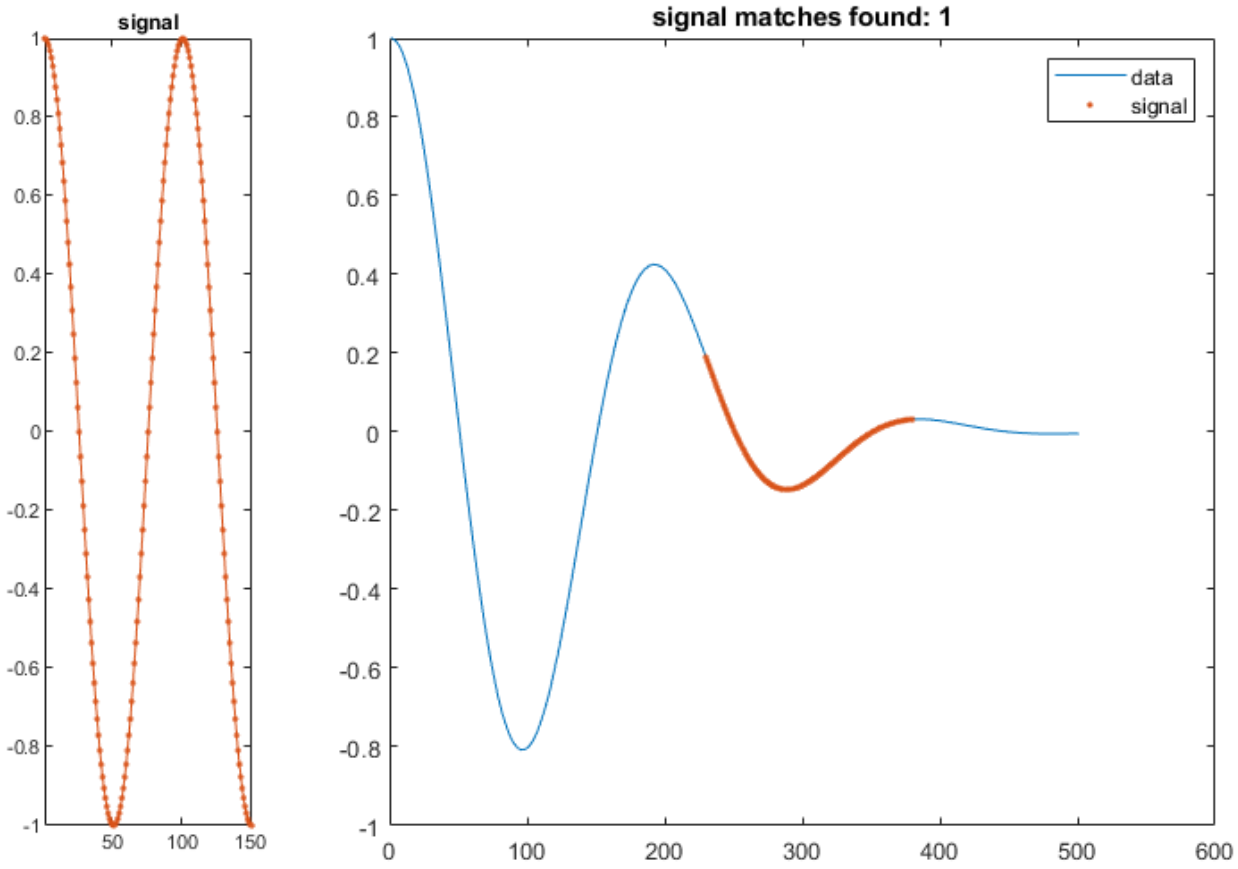
Create a signal consisting of one-and-a-half cycles of a 10 Hz sinusoid. Plot the data set and the signal.

```
ts = 0:1/fs:0.15;
signal = cos(2*pi*10*ts);

subplot(2,1,1)
plot(t,data)
title('Data')
subplot(2,1,2)
plot(ts,signal)
title('Signal')
```



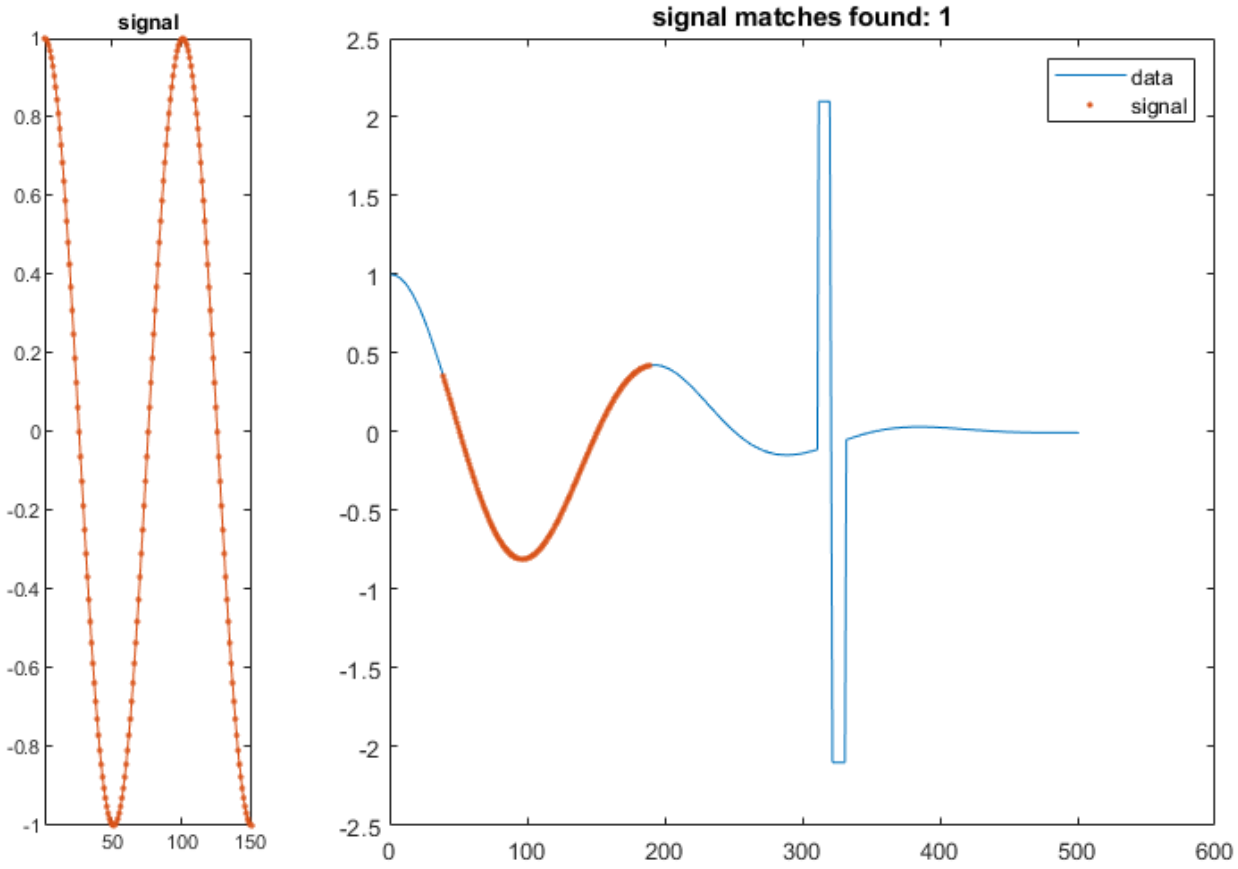
Find the segment of the data that has the smallest squared Euclidean distance to the signal. Plot the data and highlight the segment.

```
figure
findsignal(data,signal)
```



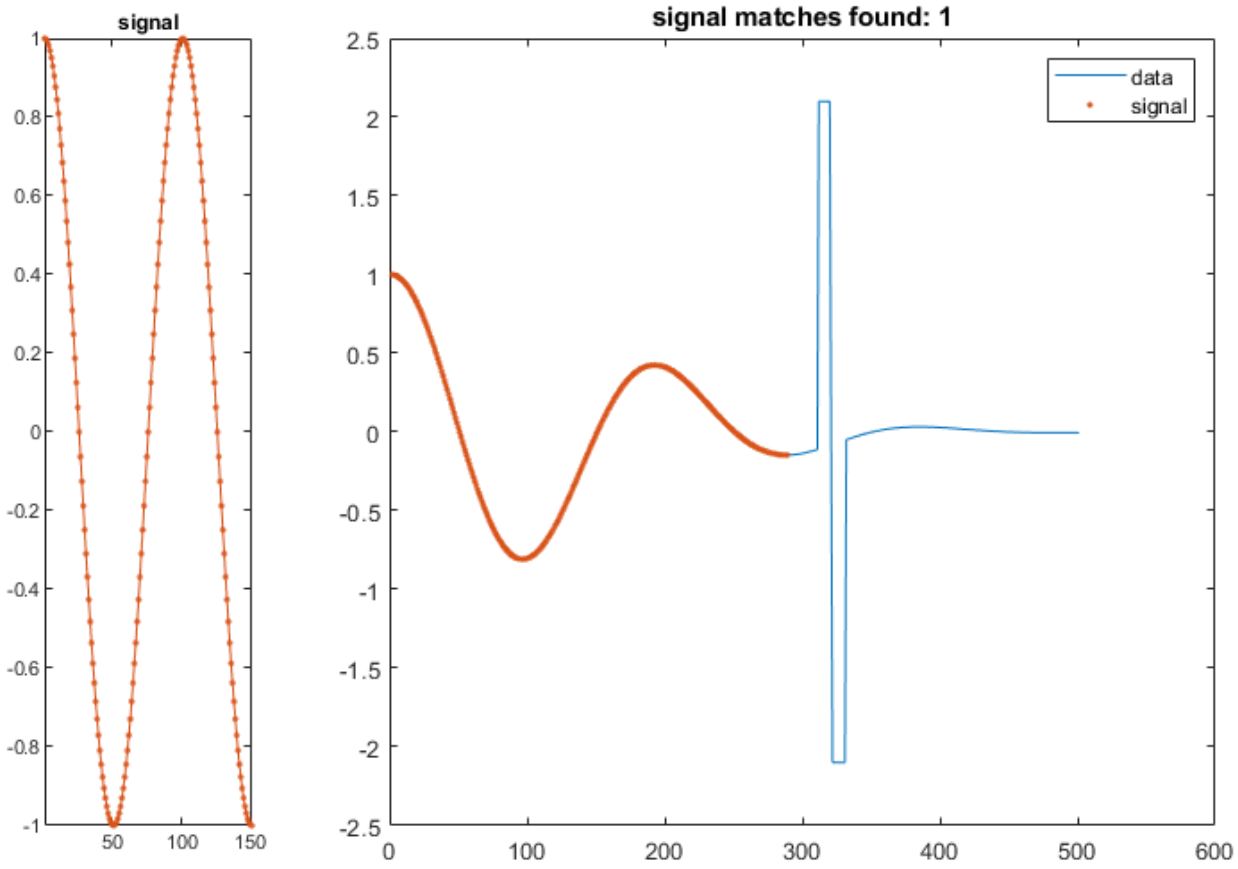
Add two clearly outlying sections to the data set. Find the segment that is closest to the signal in the sense of having the smallest absolute distance.

```
dt = data;
dt(t>0.31&t<0.32) = 2.1;
dt(t>0.32&t<0.33) = -2.1;

findsignal(dt,signal,'Metric','absolute')
```



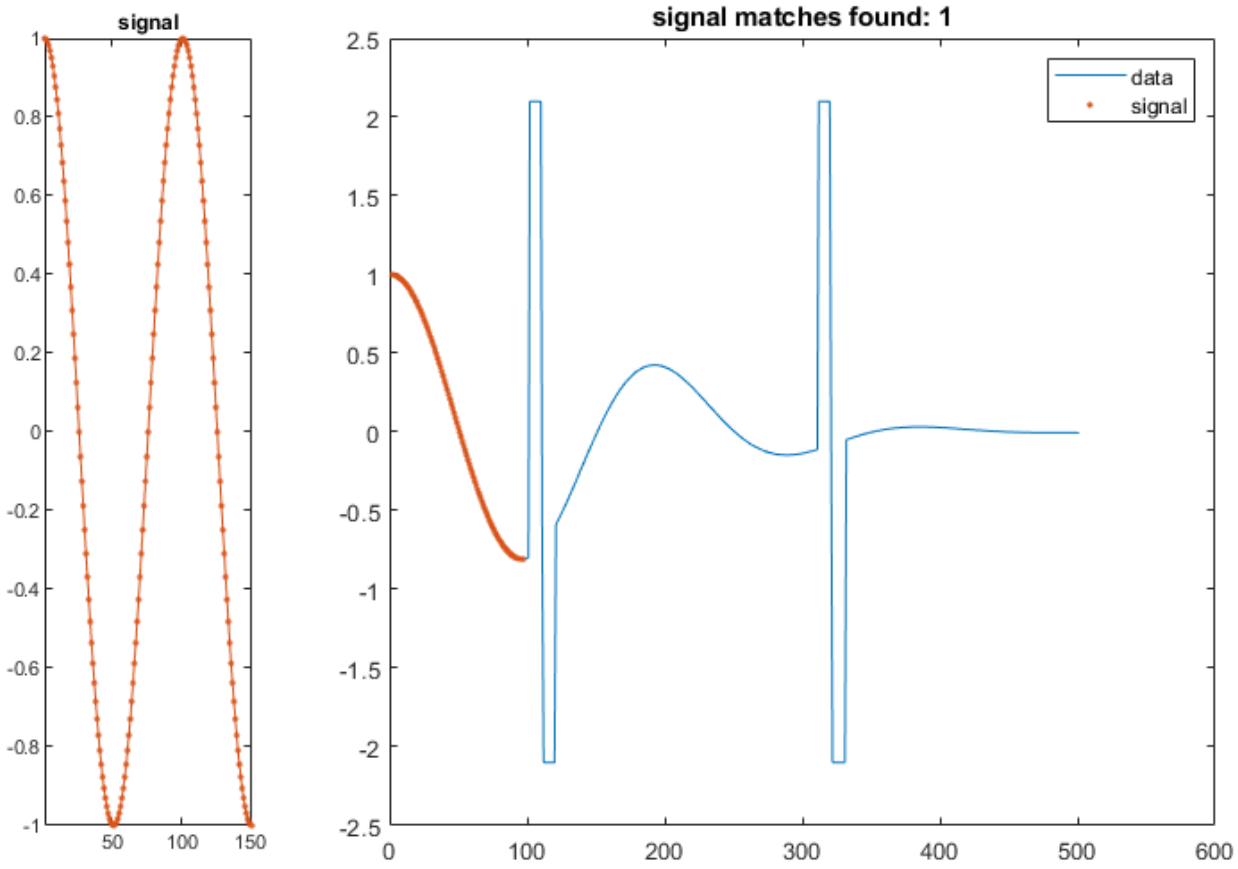
Let the *x*-axes stretch if the stretching results in a smaller absolute distance between the closest data segment and the signal.

```
findsignal(dt,signal,'TimeAlignment','dtw','Metric','absolute')
```



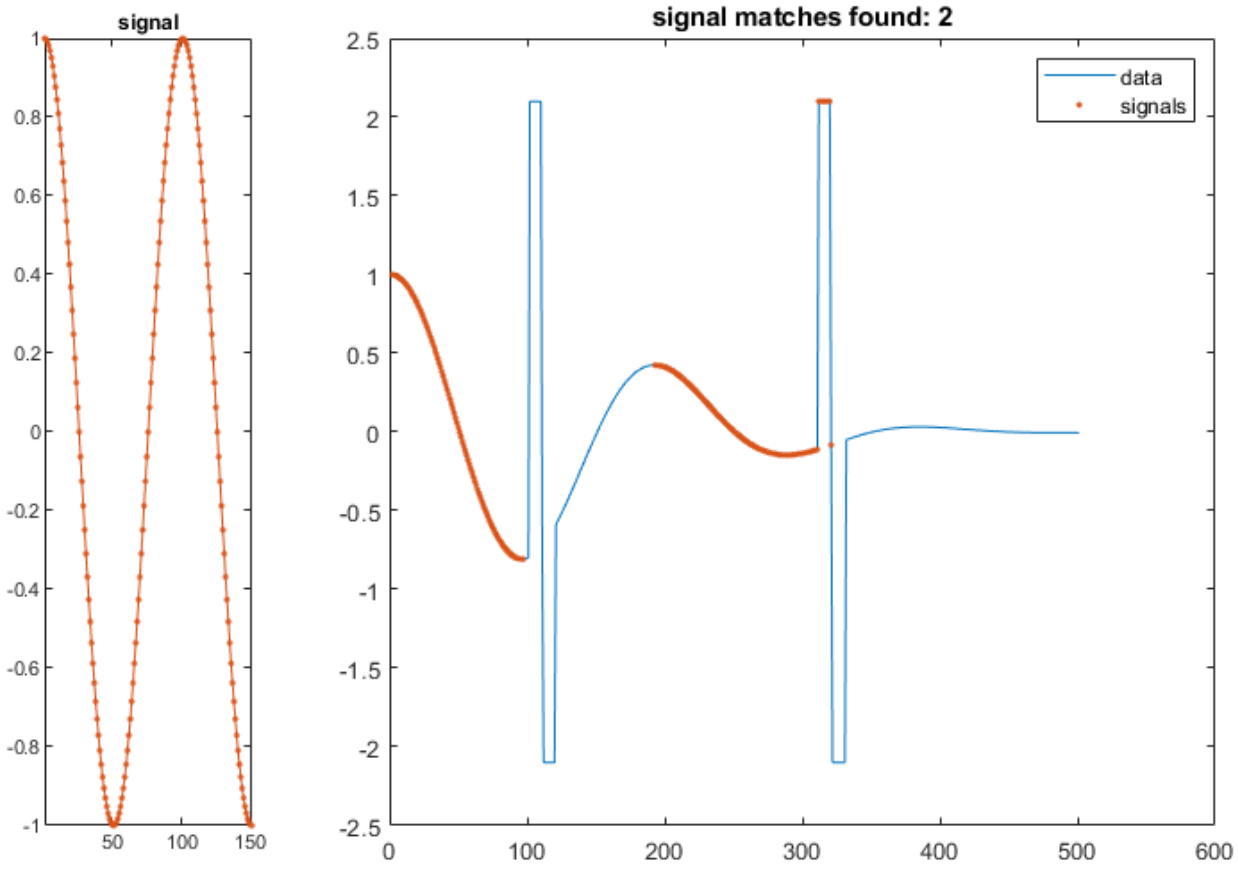
Add two more outlying sections to the data set.

```
dt(t>0.1&t<0.11) = 2.1;
dt(t>0.11&t<0.12) = -2.1;

findsignal(dt,signal,'TimeAlignment','dtw','Metric','absolute')
```



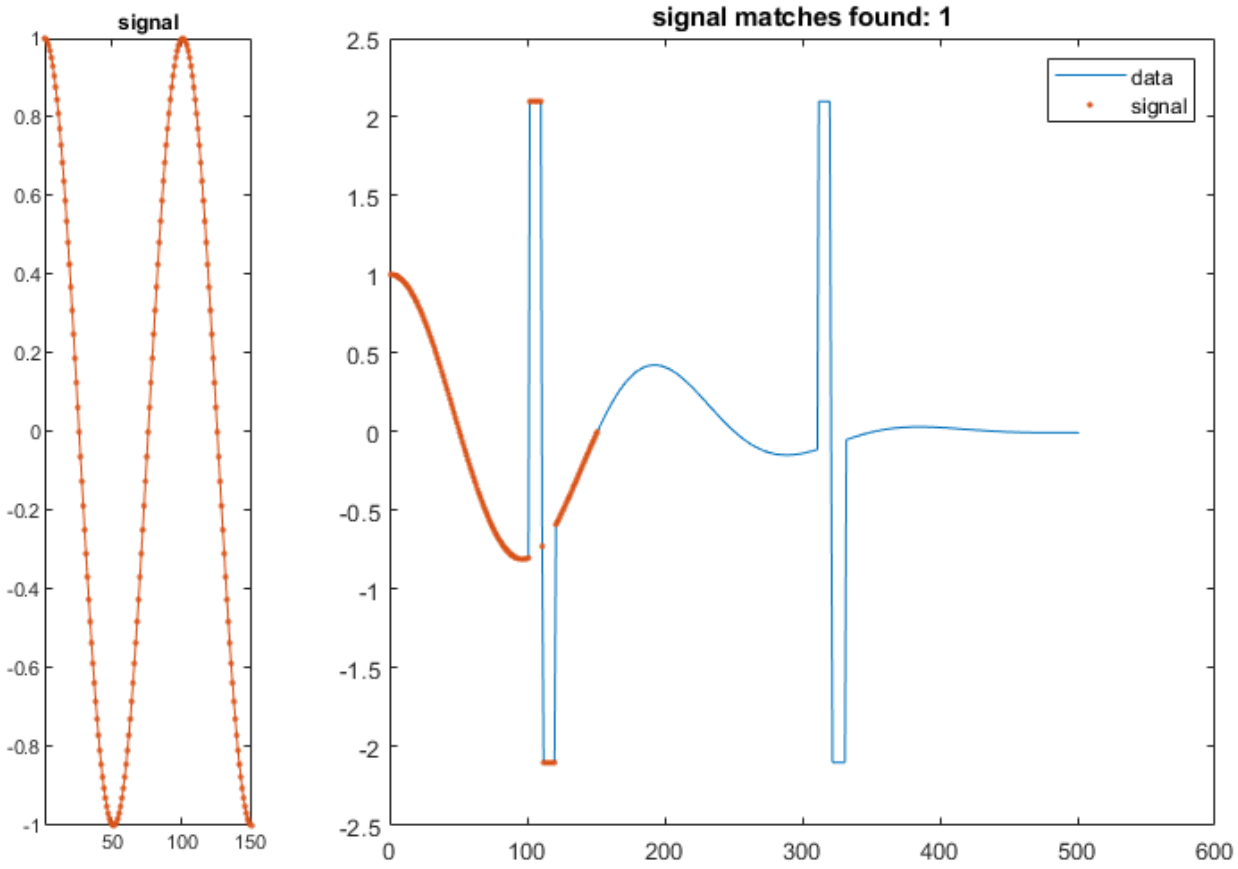
Find the two data segments closest to the signal.

```
findsignal(dt,signal,'TimeAlignment','dtw','Metric','absolute', ...
    'MaxNumSegments',2)
```



Go back to finding one segment. Choose `'edr'` as the *x*-axis stretching criterion. Select an edit distance tolerance of 3. The edit distance between nonmatching samples is independent of the actual separation, making `'edr'` robust to outliers.

```
findsignal(dt,signal,'TimeAlignment','edr','EDRTolerance',3, ...
    'Metric','absolute')
```

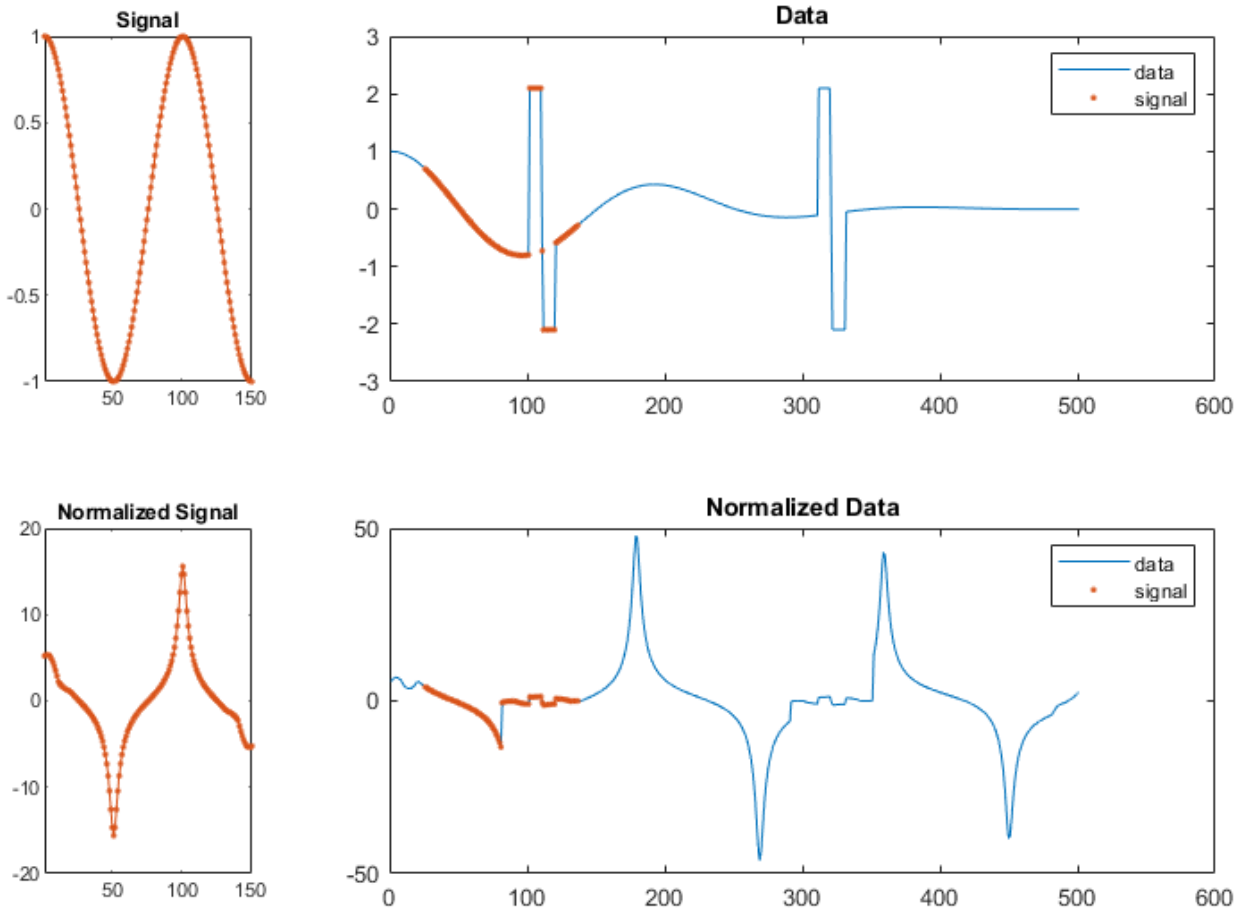


Repeat the calculation, but now normalize the data and the signal.

- Define a moving window with 10 samples to either side of each data and signal point.
- Subtract the mean of the data in the window and divide by the local standard deviation.

Find the normalized data segment that has the smallest absolute distance to the normalized signal. Display the unnormalized and normalized versions of the data and the signal.

```
findsignal(dt,signal,'TimeAlignment','edr','EDRTolerance',3, ...
    'Normalization','zscore','NormalizationLength',21, ...
    'Metric','absolute','Annotate','all')
```



Find Signal in Data with Abrupt Changes

Generate a random data array where:

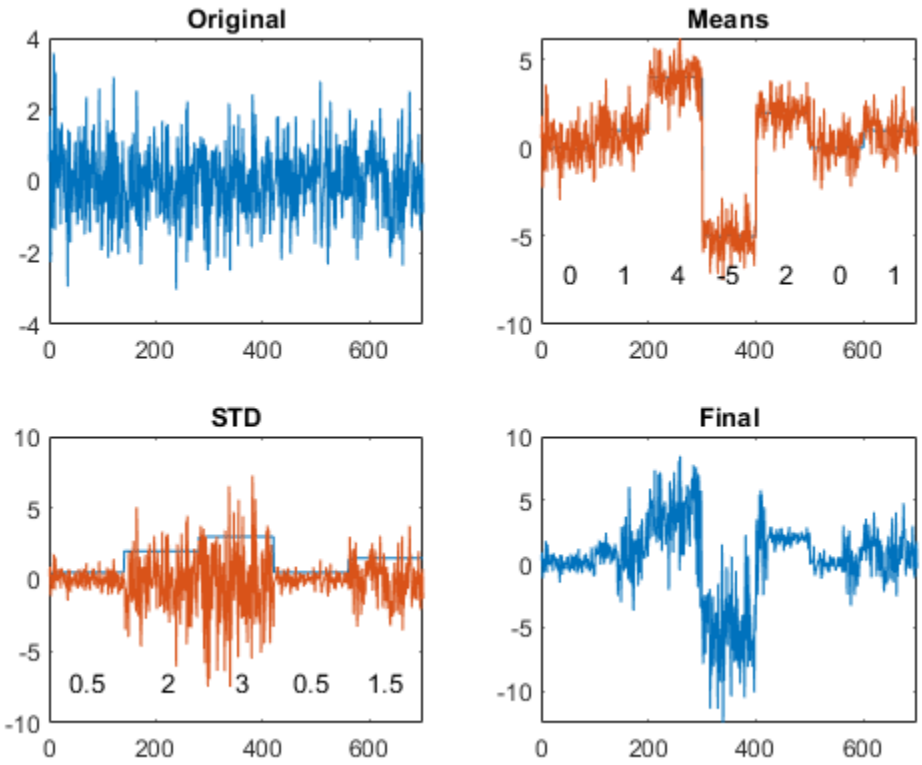
- The mean is constant in each of seven regions and changes abruptly from region to region.
- The standard deviation is constant in each of five regions and changes abruptly from region to region.

```
lr = 20;

mns = [0 1 4 -5 2 0 1];
nm = length(mns);

vrs = [1 4 6 1 3]/2;
nv = length(vrs);

v = randn(1,lr*nm*nv);

f = reshape(repmat(mns,lr*nv,1),1,lr*nm*nv);
```

```
y = reshape(repmat(vrs,lr*nm,1),1,lr*nm*nv);

t = v.*y+f;
```

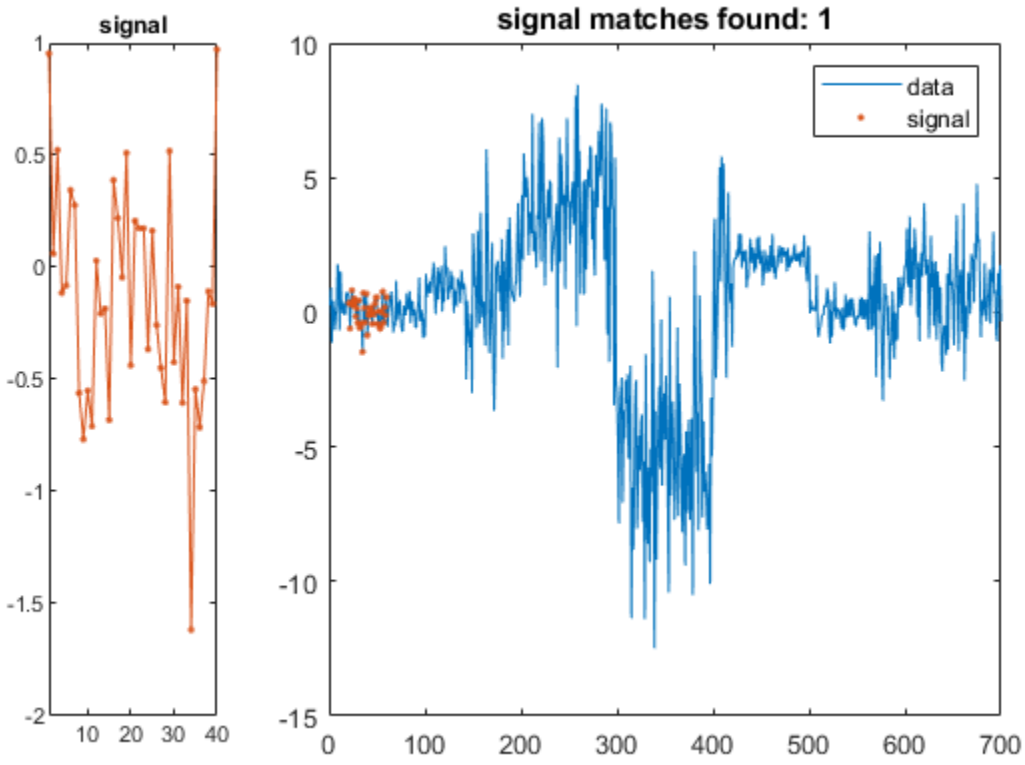
Plot the data, highlighting the steps of its construction. Display the mean and standard deviation of each region.

```
subplot(2,2,1)
plot(v)
title('Original')
xlim([0 700])

subplot(2,2,2)
plot([f;v+f]')
title('Means')
xlim([0 700])
text(lr*nv*nm*((0:1/nm:1-1/nm)+1/(2*nm)),-7*ones(1,nm),num2str(mns'), ...
    'HorizontalAlignment',"center")

subplot(2,2,3)
plot([y;v.*y]')
title('STD')
xlim([0 700])
text(lr*nv*nm*((0:1/nv:1-1/nv)+1/(2*nv)),-7*ones(1,nv),num2str(vrs'), ...
    'HorizontalAlignment',"center")

subplot(2,2,4)
plot(t)
title('Final')
xlim([0 700])
```



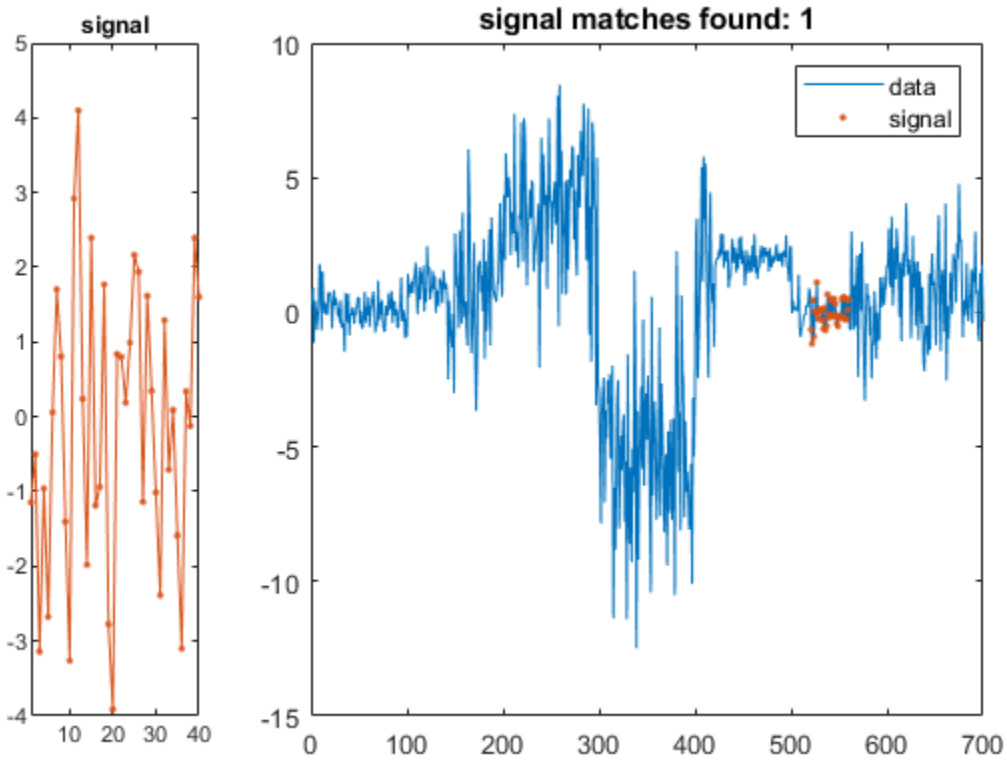
Create a random signal with a mean of zero and a standard deviation of 1/2. Find and display the segment of the data array that best matches the signal.

```
sg = randn(1,2*lr)/2;

findsignal(t,sg)
```



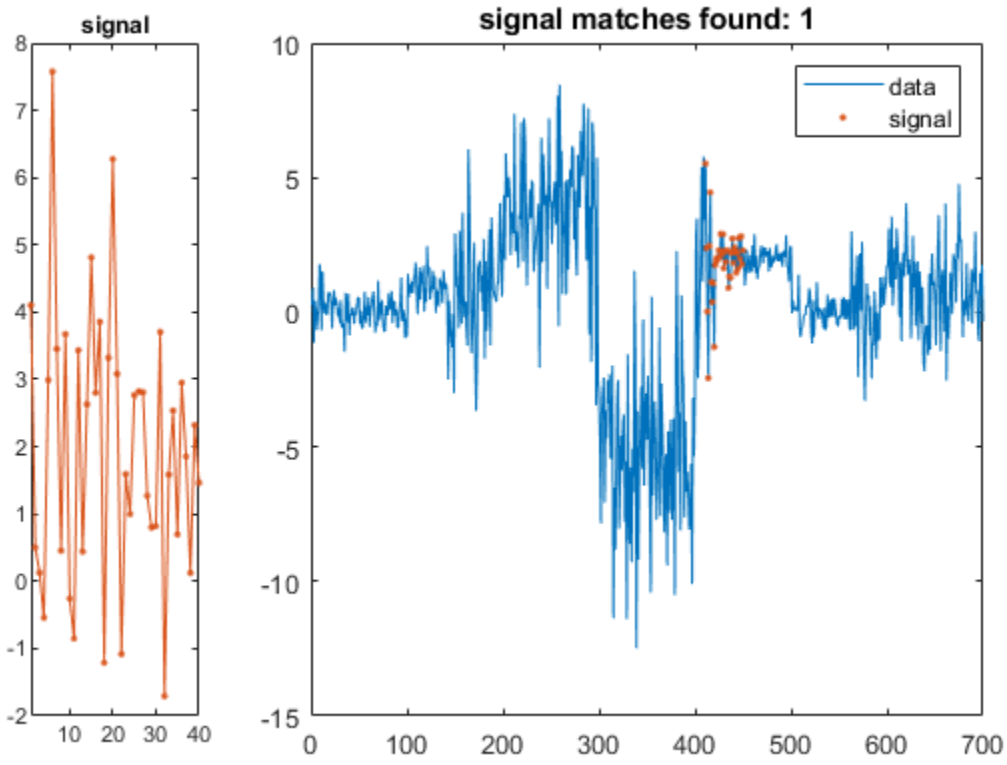
Create a random signal with a mean of zero and a standard deviation of 2. Find and display the segment of the data array that best matches the signal.

```
sg = randn(1,2*lr)*2;

findsignal(t,sg)
```



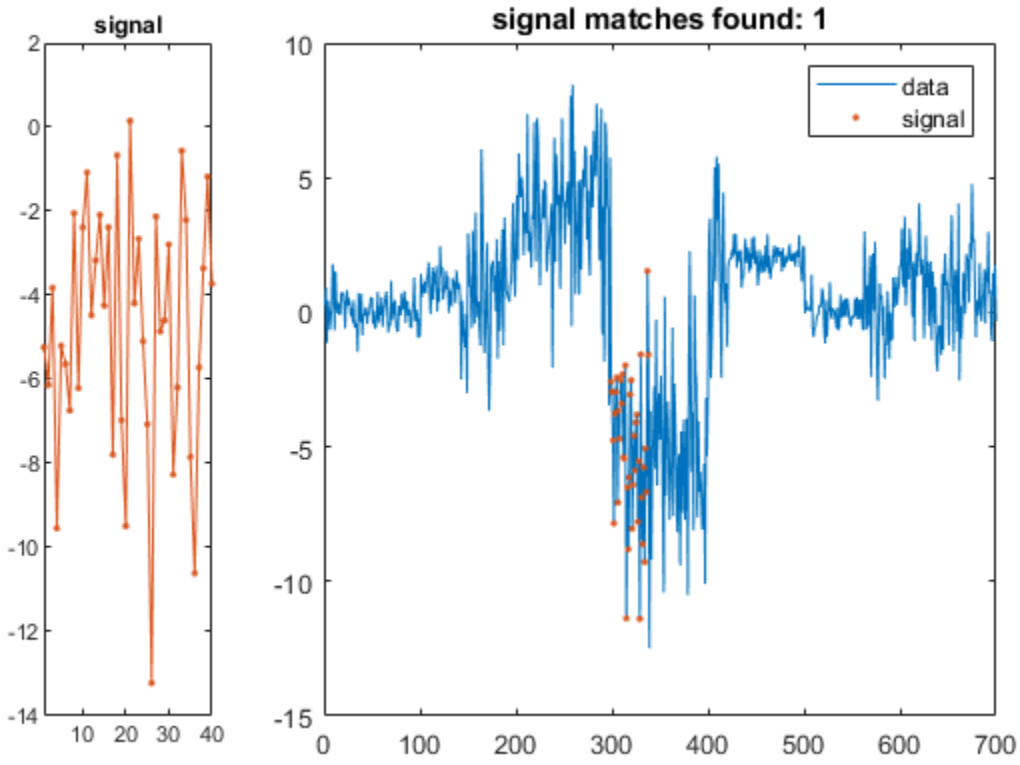
Create a random signal with a mean of 2 and a standard deviation of 2. Find and display the segment of the data array that best matches the signal.

```
sg = randn(1,2*lr)*2+2;

findsignal(t,sg)
```



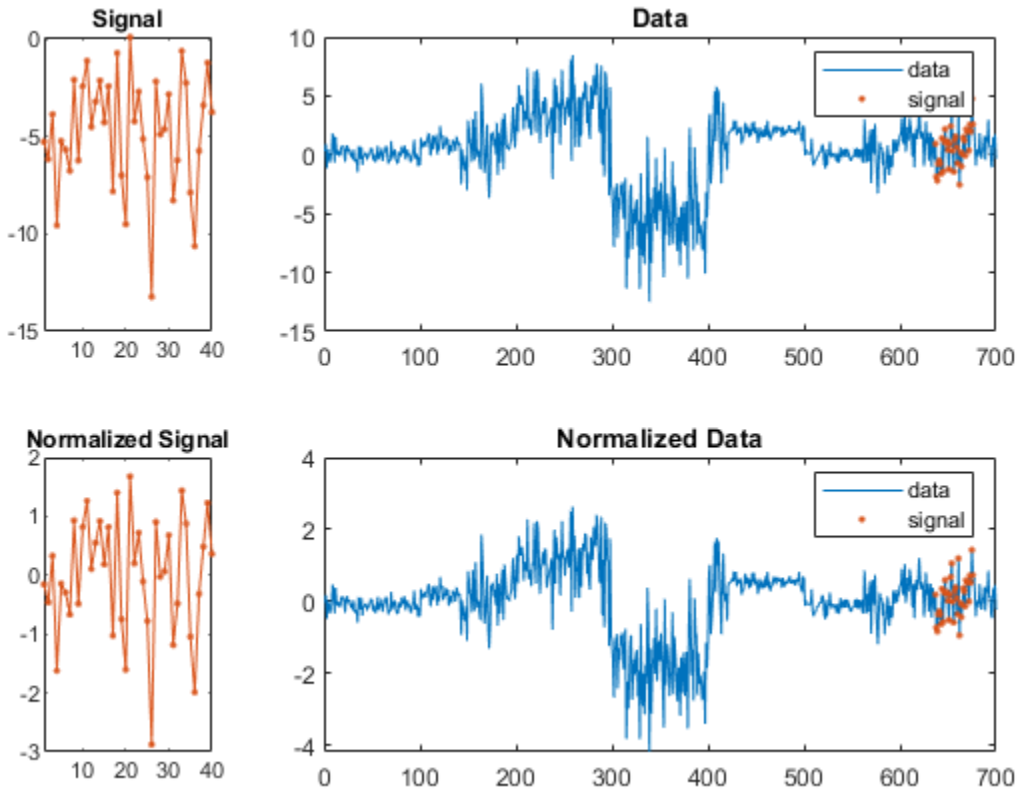
Create a random signal with a mean of -4 and a standard deviation of 3. Find and display the segment of the data array that best matches the signal.

```
sg = randn(1,2*lr)*3-4;

findsignal(t,sg)
```



Repeat the calculation, but this time subtract the mean from both the signal and the data.

```
findsignal(t,sg,'Normalization','zscore','Annotate','all')
```



Find Letter in Writing Sample

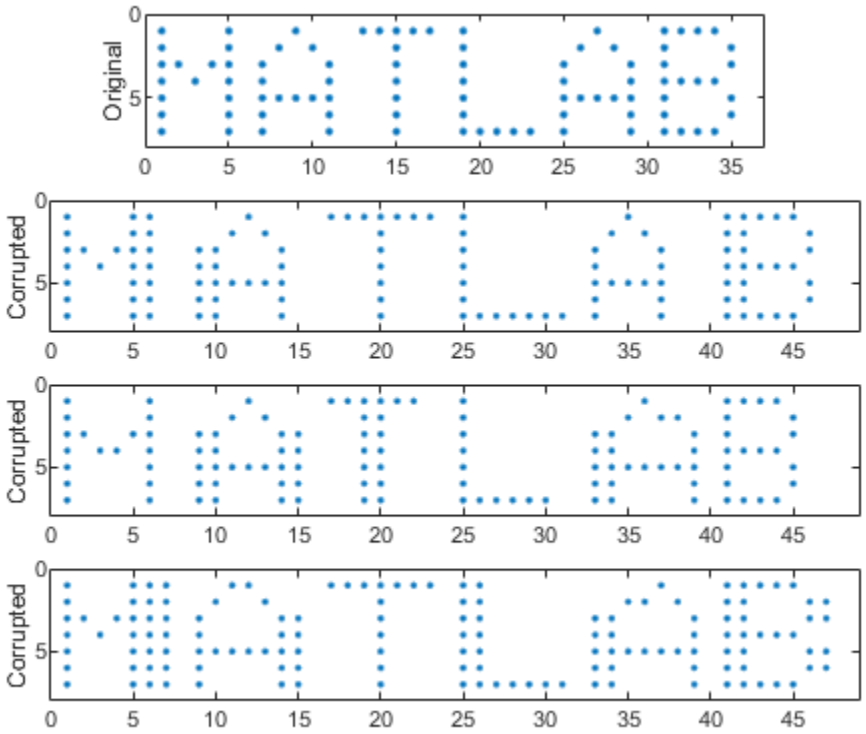
Devise a typeface that resembles the output of early computers. Use it to write the word MATLAB®.

```
rng default

chr = @(x)dec2bin(x')-48;

M = chr([34 34 54 42 34 34 34]);
A = chr([08 20 34 34 62 34 34]);
T = chr([62 08 08 08 08 08 08]);
L = chr([32 32 32 32 32 32 62]);
B = chr([60 34 34 60 34 34 60]);

MATLAB = [M A T L A B];
```

Corrupt the word by repeating random columns of the letters and varying the spacing. Show the original word and three corrupted versions.

```
c = @(x)x(:,sort([1:6 randi(6,1,2)]));

subplot(4,1,1,'XLim',[0 60])
spy(MATLAB)
xlabel('')
ylabel('Original')
```

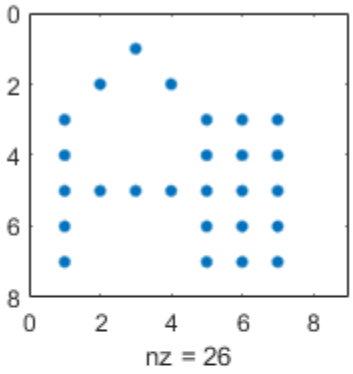
```
for kj = 2:4
    subplot(4,1,kj,'XLim',[0 60])
    spy([c(M) c(A) c(T) c(L) c(A) c(B)])
    xlabel('')
    ylabel('Corrupted')
end
```

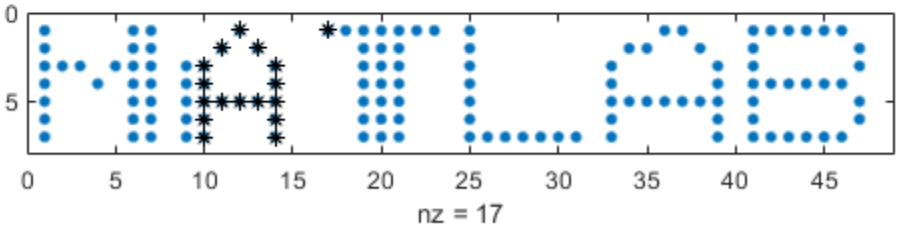


Generate one more corrupted version of the word. Search for a noisy version of the letter "A." Display the distance between the search array and the data segment closest to it. The segment spills into the "T" because the horizontal axes are rigid.

```
corr = [c(M) c(A) c(T) c(L) c(A) c(B)];

sgn = c(A);

[ist,ind,dst] = findsignal(corr,sgn);

clf
subplot(2,1,1)
spy(sgn)
subplot(2,1,2)
spy(corr)
chk = zeros(size(corr));
chk(:,ist:ind) = corr(:,ist:ind);
hold on
```

```
spy(chk,'*k')
hold off
```



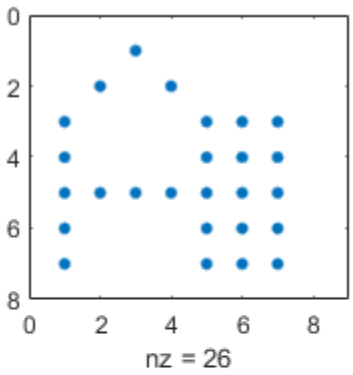


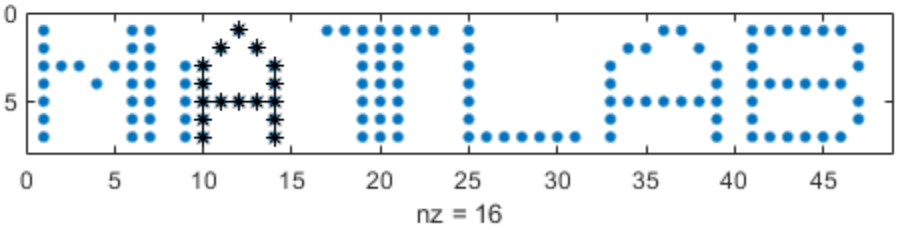

```
dst
```

```
dst = 11
```

Allow the horizontal axes to stretch. The closest segment is the intersection of the search array and the first instance of "A." The distance between the segment and the array is zero.

```
[ist,ind,dst] = findsignal(corr,sgn,'TimeAlignment','dtw');

subplot(2,1,1)
spy(sgn)
subplot(2,1,2)
spy(corr)
chk = zeros(size(corr));
chk(:,ist:ind) = corr(:,ist:ind);
hold on
spy(chk,'*k')
hold off
```



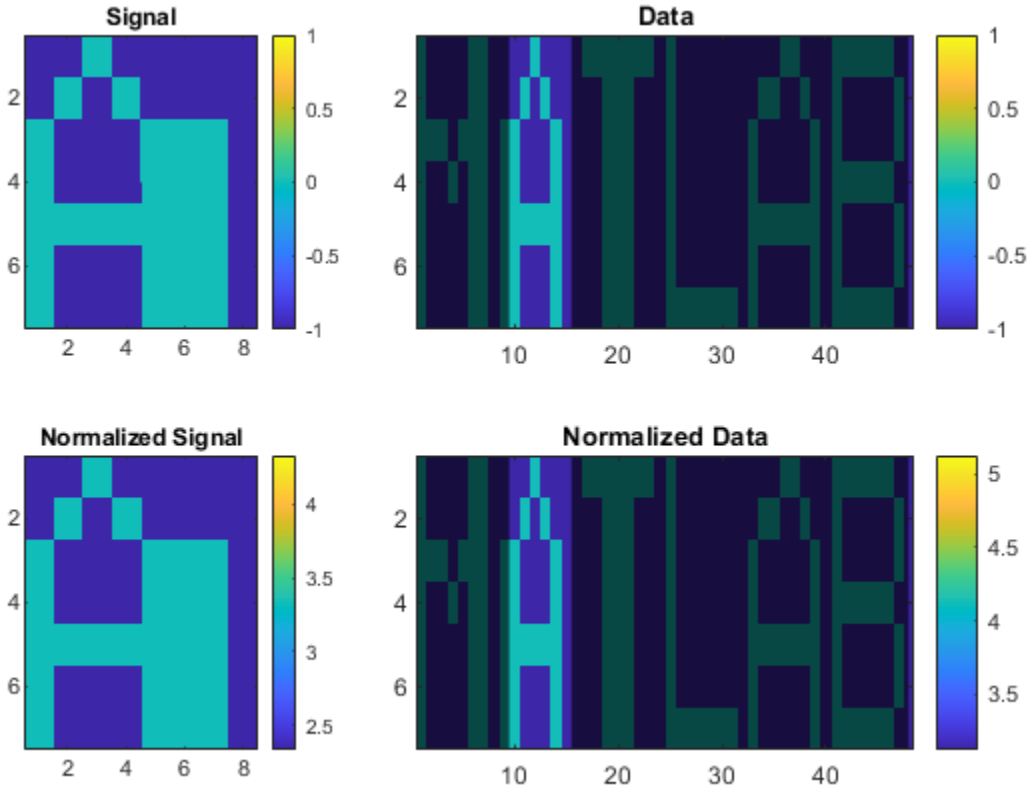



```
dst
```

```
dst = 0
```

Repeat the computation using the built-in functionality of `findsignal`. Divide by the local mean to normalize the data and the signal. Use the symmetric Kullback-Leibler metric.

```
findsignal(corr,sgn,'TimeAlignment','dtw', ...
    'Normalization','power','Metric','symmkl','Annotate','all')
```



Input Arguments

`data` — Data array

vector | matrix

Data array, specified as a vector or matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

`signal` — Search array

vector | matrix

Search array, specified as a vector or matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MaxNumSegments',2,'Metric','squared','Normalization','center','NormalizationLength',11` finds the two segments of the data array that have the smallest squared Euclidean distances to the search signal. Both the data and the signal are normalized by subtracting the mean of a sliding window. The window has five samples to either side of each point, for a total length of 5 + 5 + 1 = 11 samples.

`Normalization` — Normalization statistic
`'none'` (default) | `'center'` | `'power'` | `'zscore'`

Normalization statistic, specified as the comma-separated pair consisting of `'Normalization'` and one of these values:

- `'none'` — Do not normalize.
- `'center'` — Subtract local mean.
- `'power'` — Divide by local mean.
- `'zscore'` — Subtract local mean and divide by local standard deviation.

`NormalizationLength` — Normalization length
`length(data)` (default) | integer scalar

Normalization length, specified as the comma-separated pair consisting of `'NormalizationLength'` and an integer scalar. This value represents the minimum number of samples over which to normalize each sample in both the data and the signal. If the signal is a matrix, then `'NormalizationLength'` represents a number of columns.

Data Types: `single` | `double`

`MaxDistance` — Maximum segment distance
`Inf` (default) | positive real scalar

Maximum segment distance, specified as the comma-separated pair consisting of `'MaxDistance'` and a positive real scalar. If you specify `'MaxDistance'`, then `findsignal` returns the start and stop indices of all segments of `data` whose distances from `signal` are both local minima and smaller than `'MaxDistance'`.

Data Types: `single` | `double`

`MaxNumSegments` — Maximum number of segments to return
`1` (default) | positive integer scalar

Maximum number of segments to return, specified as the comma-separated pair consisting of `'MaxNumSegments'` and a positive integer scalar. If you specify `'MaxNumSegments'`, then `findsignal` locates all segments of `data` whose distances from the signal are local minima and returns up to `'MaxNumSegments'` segments with smallest distances.

Data Types: `single` | `double`

`TimeAlignment` — Time alignment technique
`'fixed'` (default) | `'dtw'` | `'edr'`

Time alignment technique, specified as the comma-separated pair consisting of `'TimeAlignment'` and one of these values:

- `'fixed'` — Do not stretch or repeat samples to minimize the distance.
- `'dtw'` — Attempt to reduce the distance by stretching the time axis and repeating samples in either the data or the signal. See `dtw` for more information.
- `'edr'` — Minimize the number of edits so that the distance between each remaining sample of the data segment and its signal counterpart lies within a given tolerance. An edit consists of removing a sample from the data, the signal, or both. Specify the tolerance using the `'EDRTolerance'` argument. Use this option when any of the input arrays has outliers. See `edr` for more information.

`EDRTolerance` — Edit distance tolerance
real scalar

Edit distance tolerance, specified as the comma-separated pair consisting of `'EDRTolerance'` and a real scalar. Use this argument to find the signal when the `'TimeAlignment'` name-value pair argument is set to `'edr'`.

Data Types: `single` | `double`

`Metric` — Distance metric
`'squared'` (default) | `'absolute'` | `'euclidean'` | `'symmkl'`

Distance metric, specified as the comma-separated pair consisting of `'Metric'` and one of `'squared'`, `'absolute'`, `'euclidean'`, or `'symmkl'`. If $\mathbf{X}$ and $\mathbf{Y}$ are both K-dimensional signals, then `Metric` prescribes $d_{mn}(\mathbf{X},\mathbf{Y})$, the distance between the mth sample of $\mathbf{X}$ and the nth sample of $\mathbf{Y}$. See “Dynamic Time Warping” on page 1-447 for more information about $d_{mn}(\mathbf{X},\mathbf{Y})$.

- `'squared'` — Square of the Euclidean metric, consisting of the sum of squared differences:

$$d_{mn}(\mathbf{X},\mathbf{Y}) = \sum_{k=1}^{K} (x_{k,m} - y_{k,n})^{*}(x_{k,m} - y_{k,n})$$

- `'euclidean'` — Root sum of squared differences, also known as the Euclidean or ℓ_2 metric:

$$d_{mn}(\mathbf{X},\mathbf{Y}) = \sqrt{\sum_{k=1}^{K} (x_{k,m} - y_{k,n})^{*}(x_{k,m} - y_{k,n})}$$

- `'absolute'` — Sum of absolute differences, also known as the Manhattan, city block, taxicab, or ℓ_1 metric:

$$d_{mn}(\mathbf{X},\mathbf{Y}) = \sum_{k=1}^{K} \left|x_{k,m} - y_{k,n}\right| = \sum_{k=1}^{K} \sqrt{(x_{k,m} - y_{k,n})^{*}(x_{k,m} - y_{k,n})}$$

- `'symmkl'` — Symmetric Kullback-Leibler metric. This metric is valid only for real and positive $\mathbf{X}$ and $\mathbf{Y}$:

$$d_{mn}(\mathbf{X},\mathbf{Y}) = \sum_{k=1}^{K} (x_{k,m} - y_{k,n})(\log x_{k,m} - \log y_{k,n})$$

`Annotate` — Plot style
`'signal'` (default) | `'data'` | `'all'`

Plot style, specified as the comma-separated pair consisting of `'Annotate'` and one of these values:

- `'data'` plots the data and highlights the regions that best match the signal.

- `'signal'` plots the signal in a separate subplot.
- `'all'` plots the signal, the data, the normalized signal, and the normalized data in separate subplots.

This argument is ignored if you call `findsignal` with output arguments.

Output Arguments

`istart,istop` — Segment start and end indices

integer scalars | vectors

Segment start and end indices, returned as integer scalars or vectors.

`dist` — Minimum data-signal distance

scalar | vector

Minimum data-signal distance, returned as a scalar or a vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`alignsignals` | `dtw` | `edr` | `findpeaks` | `finddelay` | `strfind` | `xcorr`

Topics

“Finding a Signal in Data”
“Find a Signal in a Measurement”

Introduced in R2016b

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.

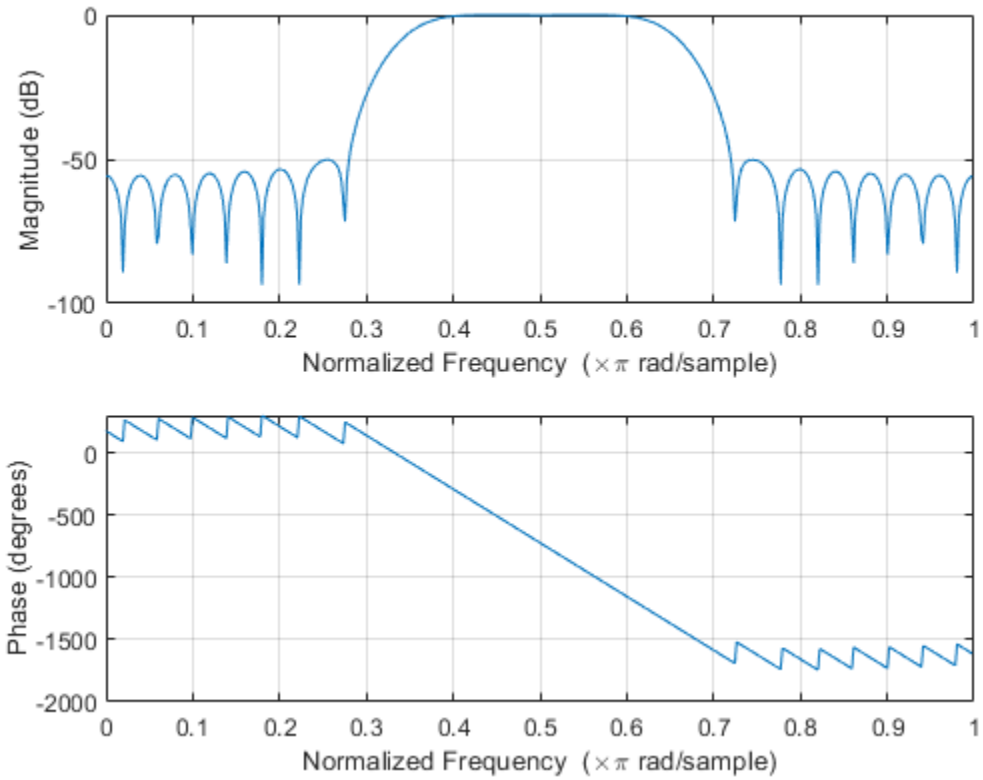
Note: Use `fir2` for windowed filters with arbitrary frequency response.

Examples

FIR Bandpass Filter

Design a 48th-order FIR bandpass filter with passband $0.35\pi \le \omega \le 0.65\pi$ rad/sample. Visualize its magnitude and phase responses.

```
b = fir1(48,[0.35 0.65]);
freqz(b,1,512)
```

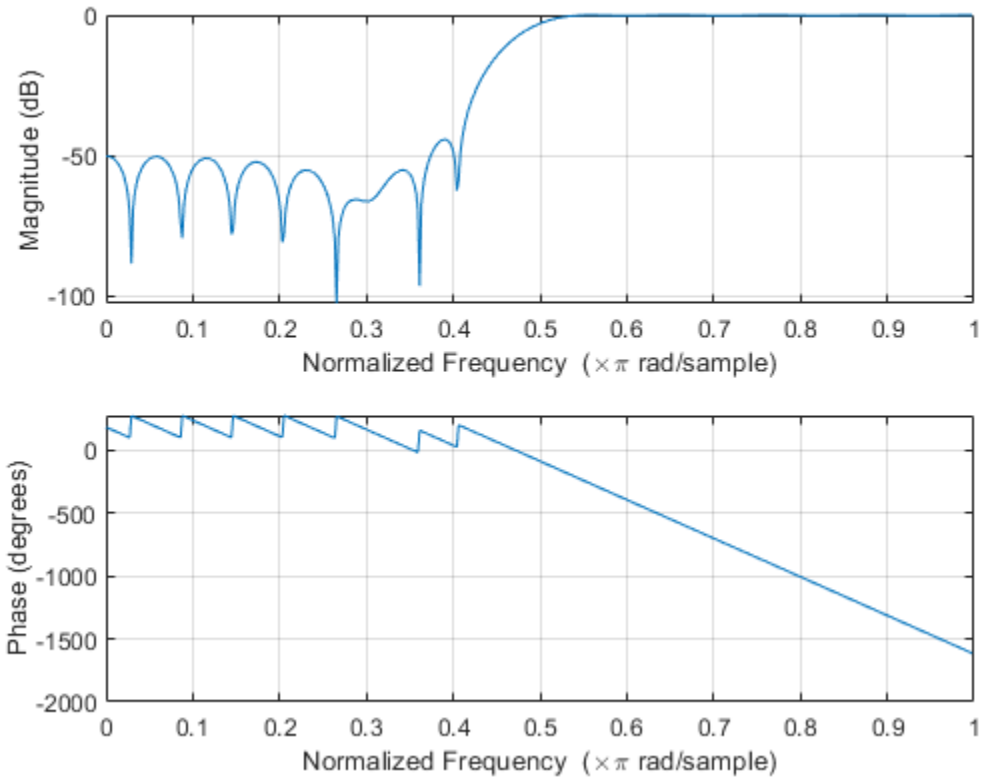


FIR Highpass Filter

Load `chirp.mat`. The file contains a signal, `y`, that has most of its power above `Fs/4`, or half the Nyquist frequency. The sample rate is 8192 Hz.

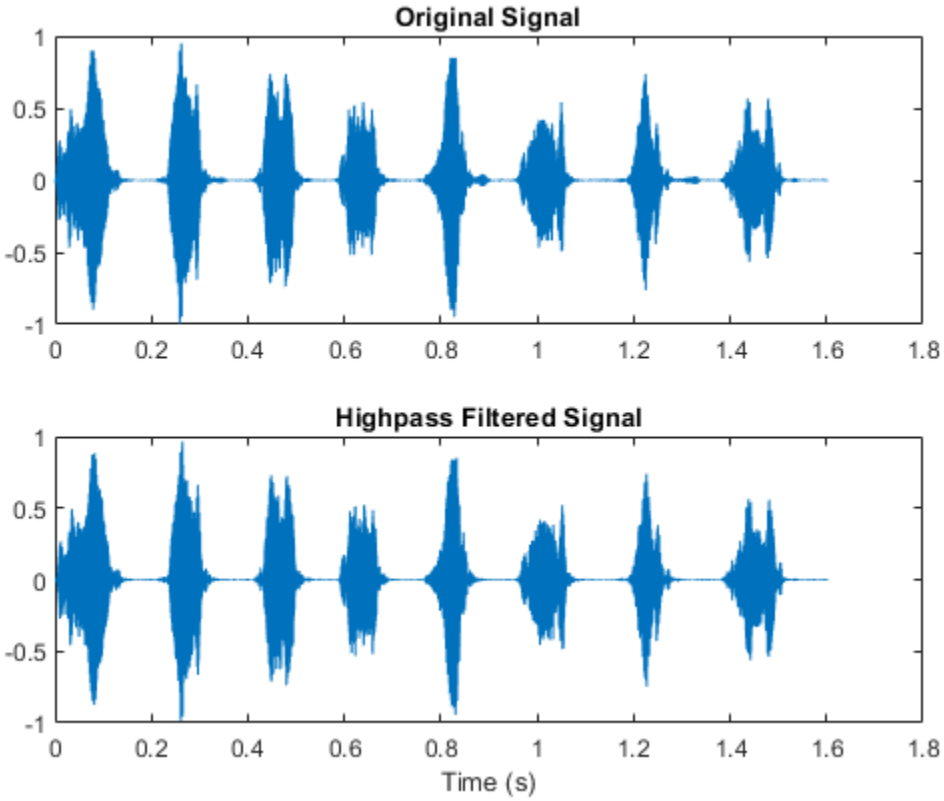
Design a 34th-order FIR highpass filter to attenuate the components of the signal below `Fs/4`. Use a cutoff frequency of 0.48 and a Chebyshev window with 30 dB of ripple.

```
load chirp

t = (0:length(y)-1)/Fs;

bhi = fir1(34,0.48,'high',chebwin(35,30));
freqz(bhi,1)
```



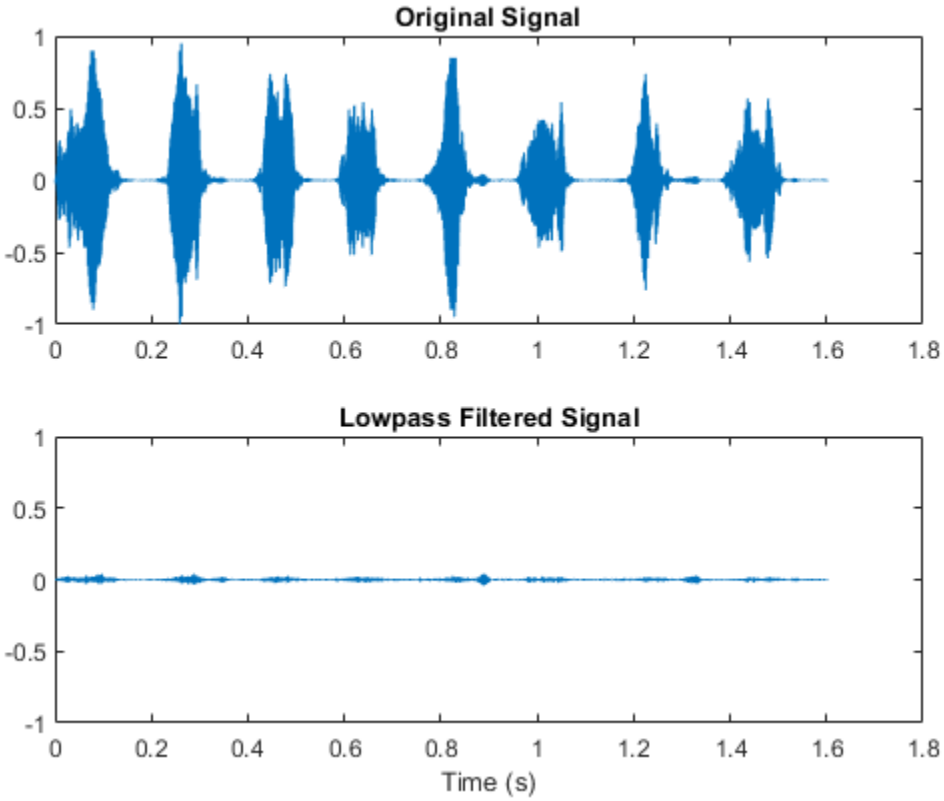
Filter the signal. Display the original and highpass-filtered signals. Use the same *y*-axis scale for both plots.

```
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)
```



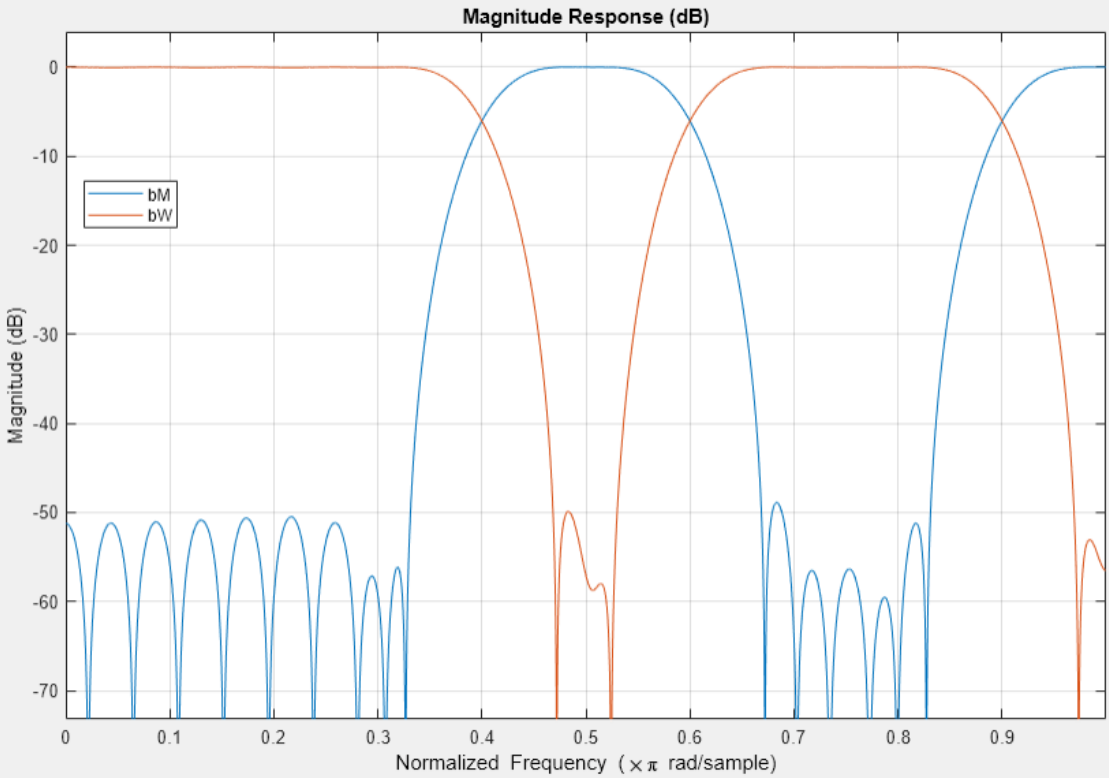
Multiband FIR Filter

Design a 46th-order FIR filter that attenuates normalized frequencies below 0.4π rad/sample and between 0.6π and 0.9π rad/sample. Call it `bM`.

```
ord = 46;

low = 0.4;
bnd = [0.6 0.9];

bM = fir1(ord,[low bnd]);
```

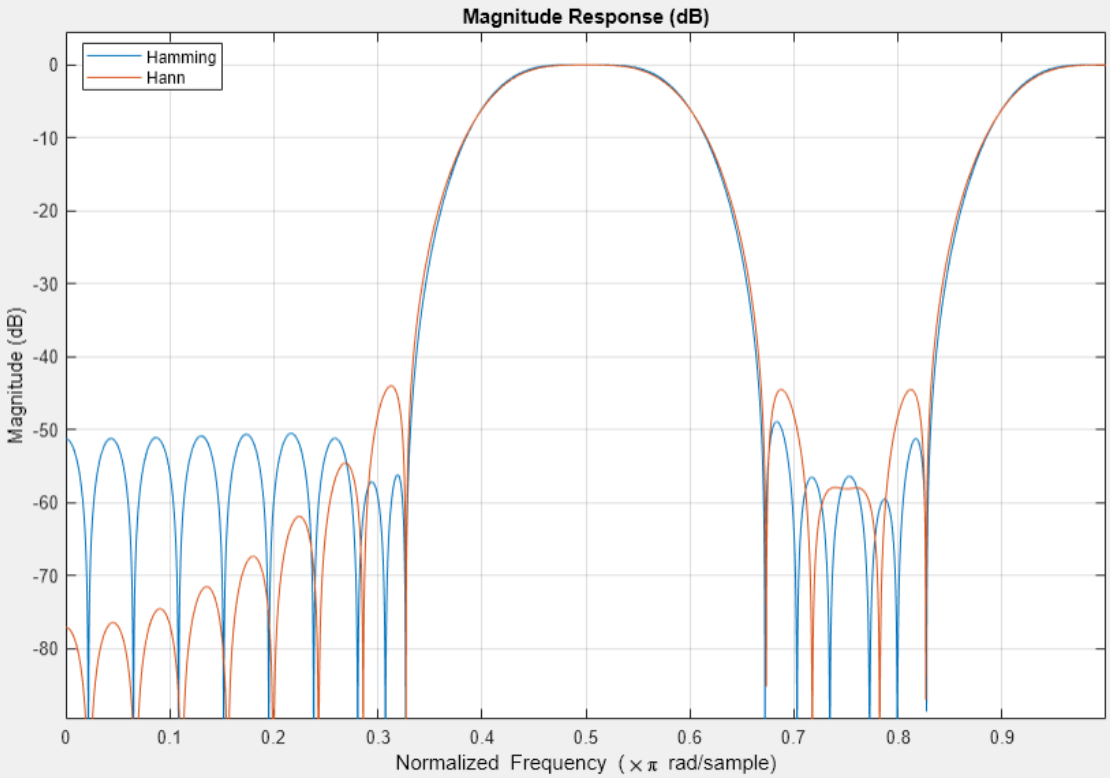
Redesign `bM` so that it passes the bands it was attenuating and stops the other frequencies. Call the new filter `bW`. Use `fvtool` to display the frequency responses of the filters.

```
bW = fir1(ord,[low bnd],'DC-1');

hfvt = fvtool(bM,1,bW,1);
legend(hfvt,'bM','bW')
```



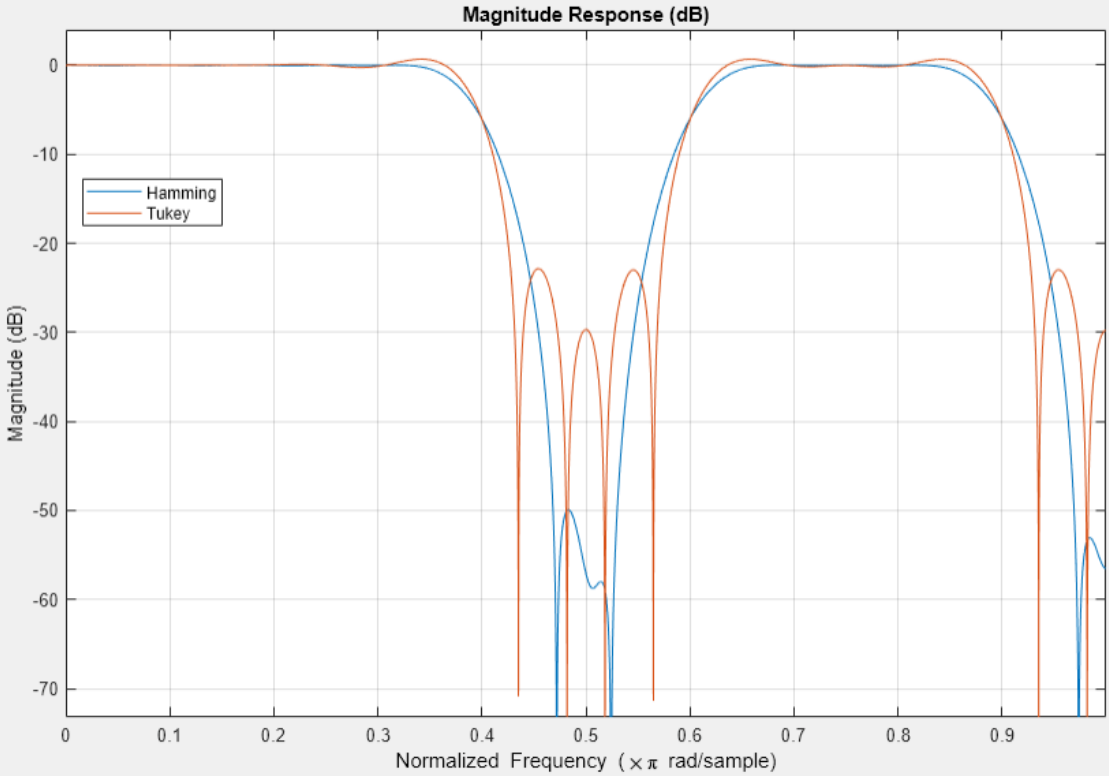
Redesign `bM` using a Hann window. (The `'DC-0'` is optional.) Compare the magnitude responses of the Hamming and Hann designs.

```
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 `Wn` must be strictly greater than 0 and strictly smaller than 1, where 1 corresponds to the Nyquist frequency: $0 < Wn < 1$. The Nyquist frequency is half the sample rate or π rad/sample.

- If `Wn` is a scalar, then `fir1` designs a lowpass or highpass filter with cutoff frequency `Wn`. The cutoff frequency is the frequency at which the normalized gain of the filter is –6 dB.
- If `Wn` is the two-element vector `[w1 w2]`, where $w1 < w2$, then `fir1` designs a bandpass or bandstop filter with lower cutoff frequency `w1` and higher cutoff frequency `w2`.

- If `Wn` is the multi-element vector `[w1 w2 ... wn]`, where `w1` < `w2` < ... < `wn`, then `fir1` returns an `n`th-order multiband filter with bands 0 < ω < `w1`, `w1` < ω < `w2`, ..., `wn` < ω < 1.

Data Types: `double`

`ftype` — Filter type
`'low'` | `'bandpass'` | `'high'` | `'stop'` | `'DC-0'` | `'DC-1'`

Filter type, specified as one of the following:

- `'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 if `Wn` is a two-element vector. `'bandpass'` is the default when `Wn` has two elements.
- `'stop'` specifies a bandstop filter if `Wn` 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 `Wn` has more than two elements.
- `'DC-1'` specifies that the first band of a multiband filter is a passband.

`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 either `'scale'` or `'noscale'`.

- `'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.

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:

$B(z) = b(1) + b(2)z + ... + b(n+1)z^{-n}$.

Algorithms

`fir1` uses a least-squares approximation to compute the filter coefficients and then smooths the impulse response with `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, Algorithm 5.2.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`cfirpm` | `designfilt` | `filter` | `fir2` | `fircls` | `fircls1` | `firls` | `firpm` | `freqz` | `hamming` | `kaiserord`

Introduced before R2006a

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

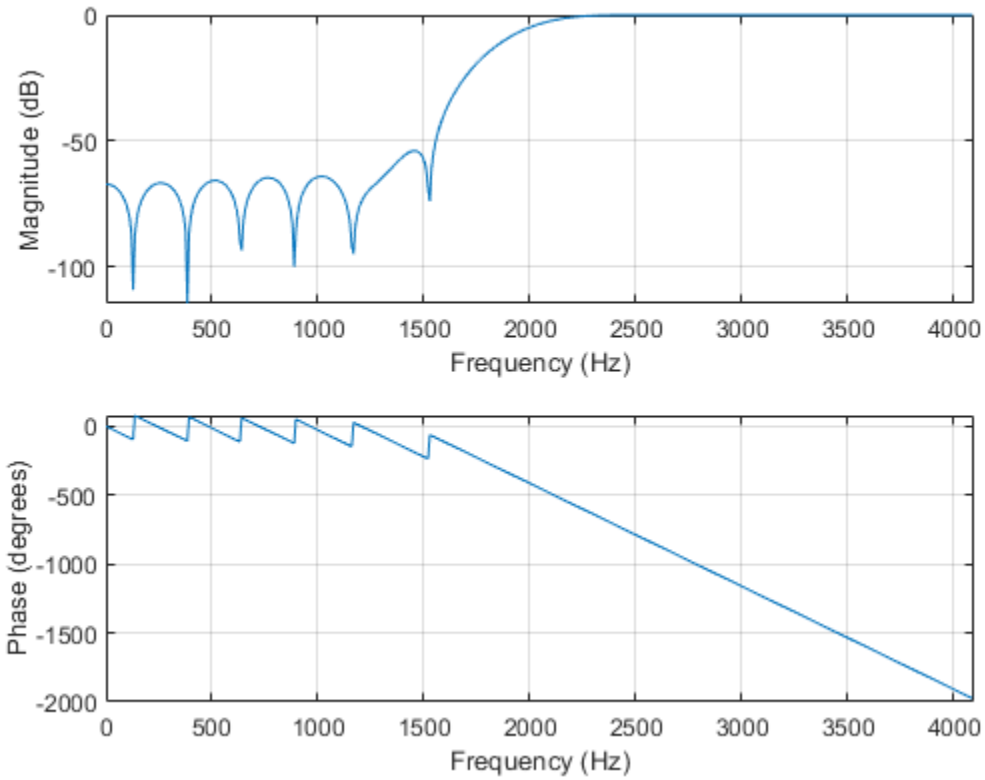
Attenuation of Low Frequencies

Load the MAT-file `chirp`. The file contains a signal, `y`, sampled at a frequency `Fs` = 8192 Hz. The signal has most of its power above `Fs`/4 = 2048 Hz, or half the Nyquist frequency. Add random noise to the signal.

```
load chirp
y = y + randn(size(y))/25;
t = (0:length(y)-1)/Fs;
```

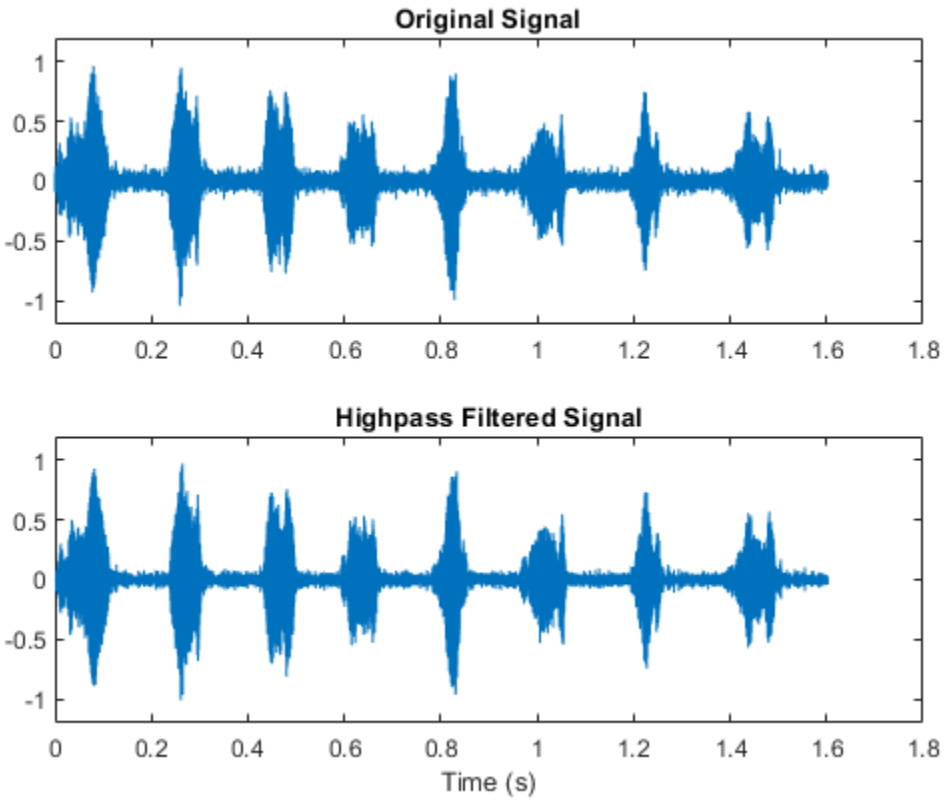
Design a 34th-order FIR highpass filter to attenuate the components of the signal below `Fs`/4. Specify a normalized cutoff frequency of 0.48, which corresponds to about 1966 Hz. Visualize the frequency response of the filter.

```
f = [0 0.48 0.48 1];
mhi = [0 0 1 1];
bhi = fir2(34,f,mhi);

freqz(bhi,1,[],Fs)
```



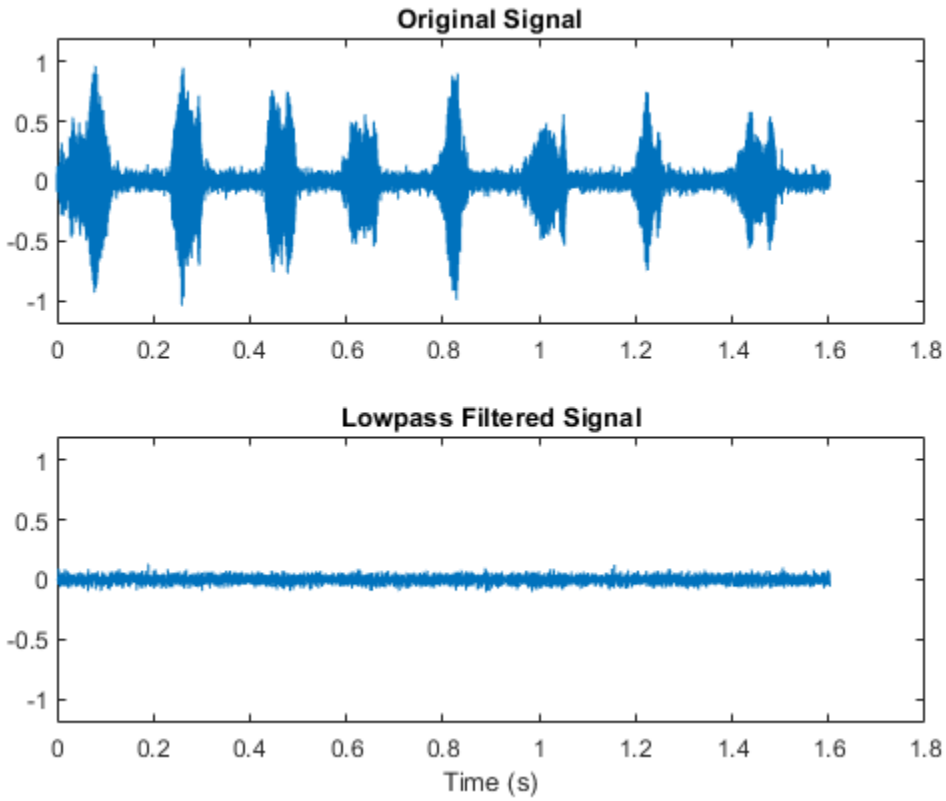
Filter the chirp signal. Plot the signal before and after filtering.

```
outhi = filter(bhi,1,y);

figure
subplot(2,1,1)
plot(t,y)
title('Original Signal')
ylim([-1.2 1.2])

subplot(2,1,2)
plot(t,outhi)
title('Highpass 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.

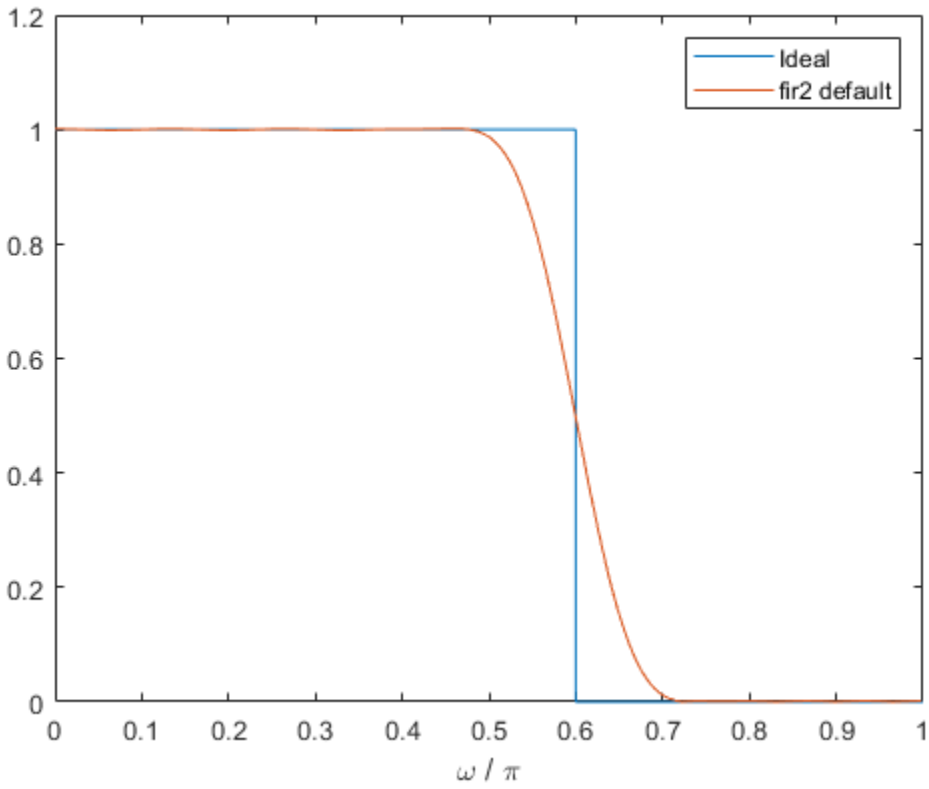
```
mlo = [1 1 0 0];
blo = fir2(34,f,mlo);
outlo = filter(blo,1,y);

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

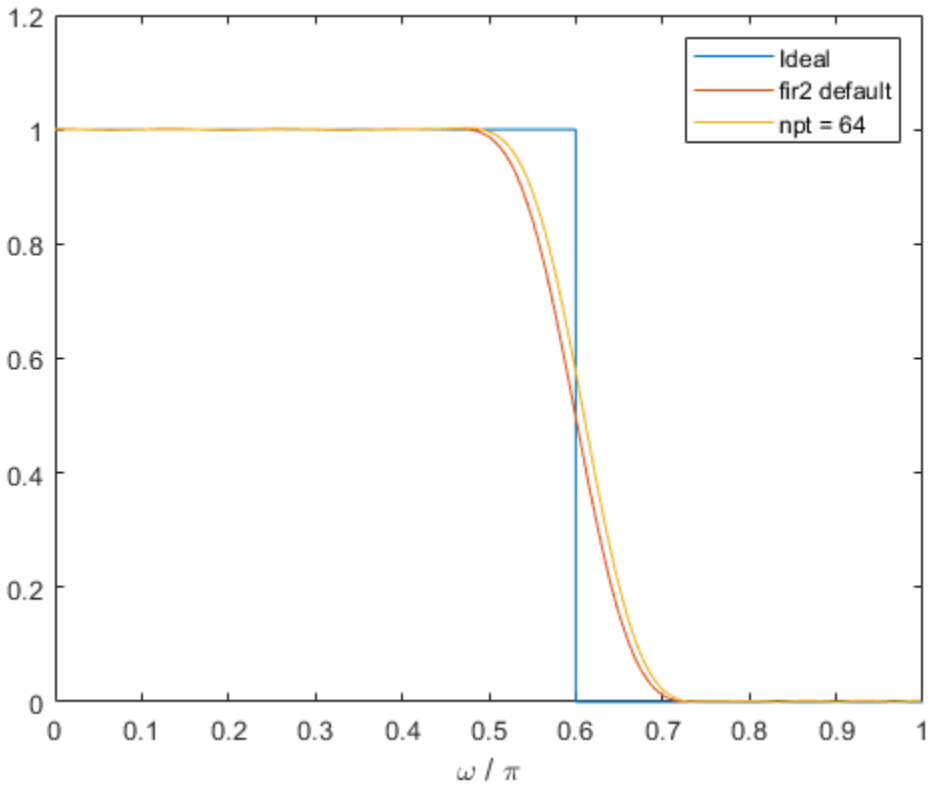
Design a 30th-order lowpass filter with a normalized cutoff frequency of 0.6π rad/sample. Plot the ideal frequency response overlaid with the actual frequency response.

```
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)
```



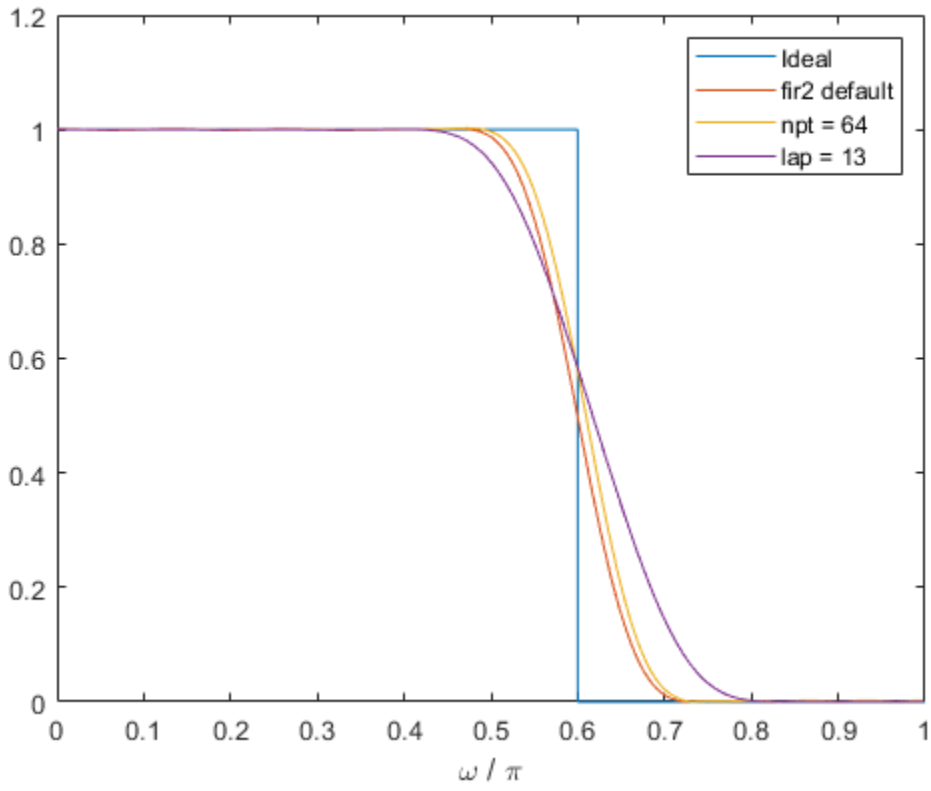
Redesign the filter using a 64-point interpolation grid.

```
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 0.18π rad/sample.

```
F1 = 0:0.01:0.18;
A1 = 0.5+sin(2*pi*7.5*F1)/4;
```

- A piecewise linear section between 0.2π rad/sample and 0.78π 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 0.79π 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.

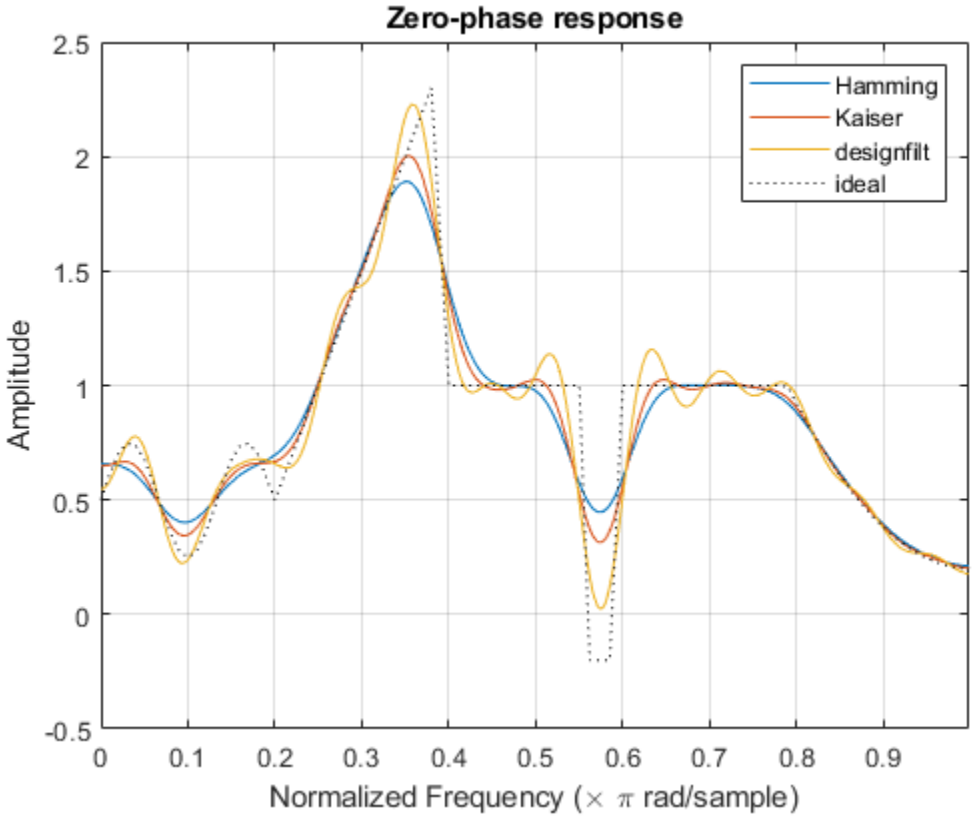
```
kai = fir2(N,FreqVect,AmplVect,kaiser(N+1,3));
```

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:

$B(z) = b(1) + b(2)z + \ldots + b(n+1)z^{-n}$.

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] Jackson, L. B. *Digital Filters and Signal Processing*. 3rd Ed. Boston: Kluwer Academic Publishers, 1996.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`butter` | `cheby1` | `cheby2` | `designfilt` | `ellip` | `filter` | `fir1` | `hamming` | `maxflat` | `firpm` | `yulewalk`

Introduced before R2006a

fircls

Constrained-least-squares 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. The frequency-magnitude characteristics of this filter match those given by vectors `f` and `amp`.

`fircls(n,f,amp,up,lo,"design_flag")` enables you to specify visual display options for the filter design.

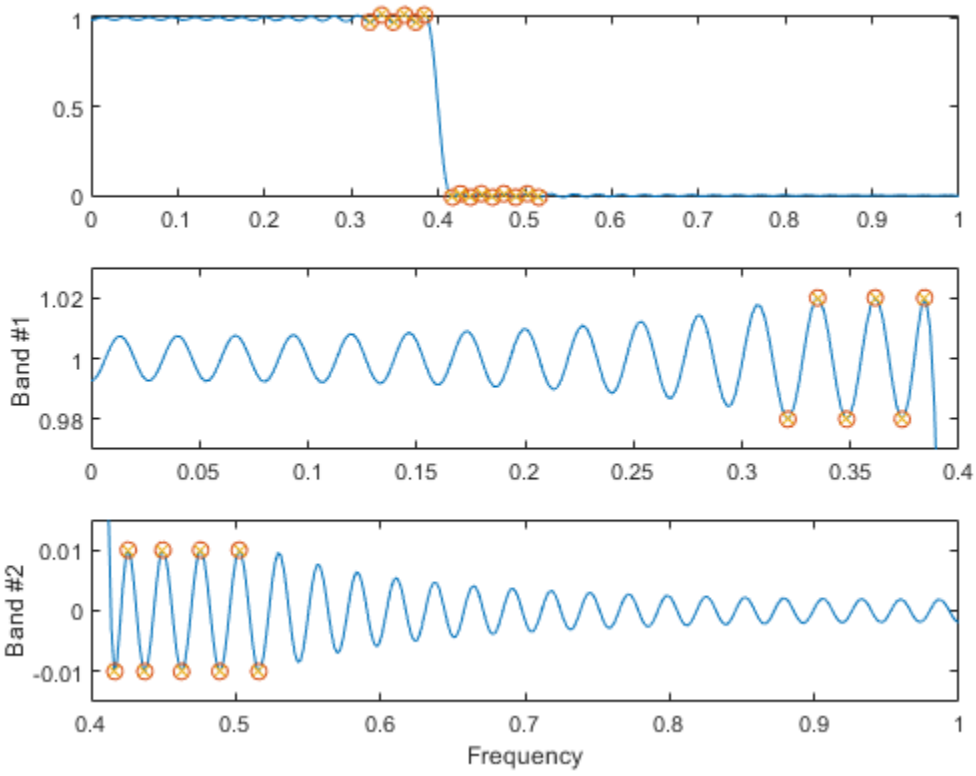
Examples

Constrained Least-Squares Lowpass Filter

Design a 150th-order lowpass filter with a normalized cutoff frequency of 0.4π rad/sample. Specify a maximum absolute error of 0.02 in the passband and 0.01 in the stopband. Display the design error and magnitude responses of the filter.

```
n = 150;
f = [0 0.4 1];
a = [1 0];
up = [1.02 0.01];
lo = [0.98 -0.01];
b = fircls(n,f,a,up,lo,"both");
```

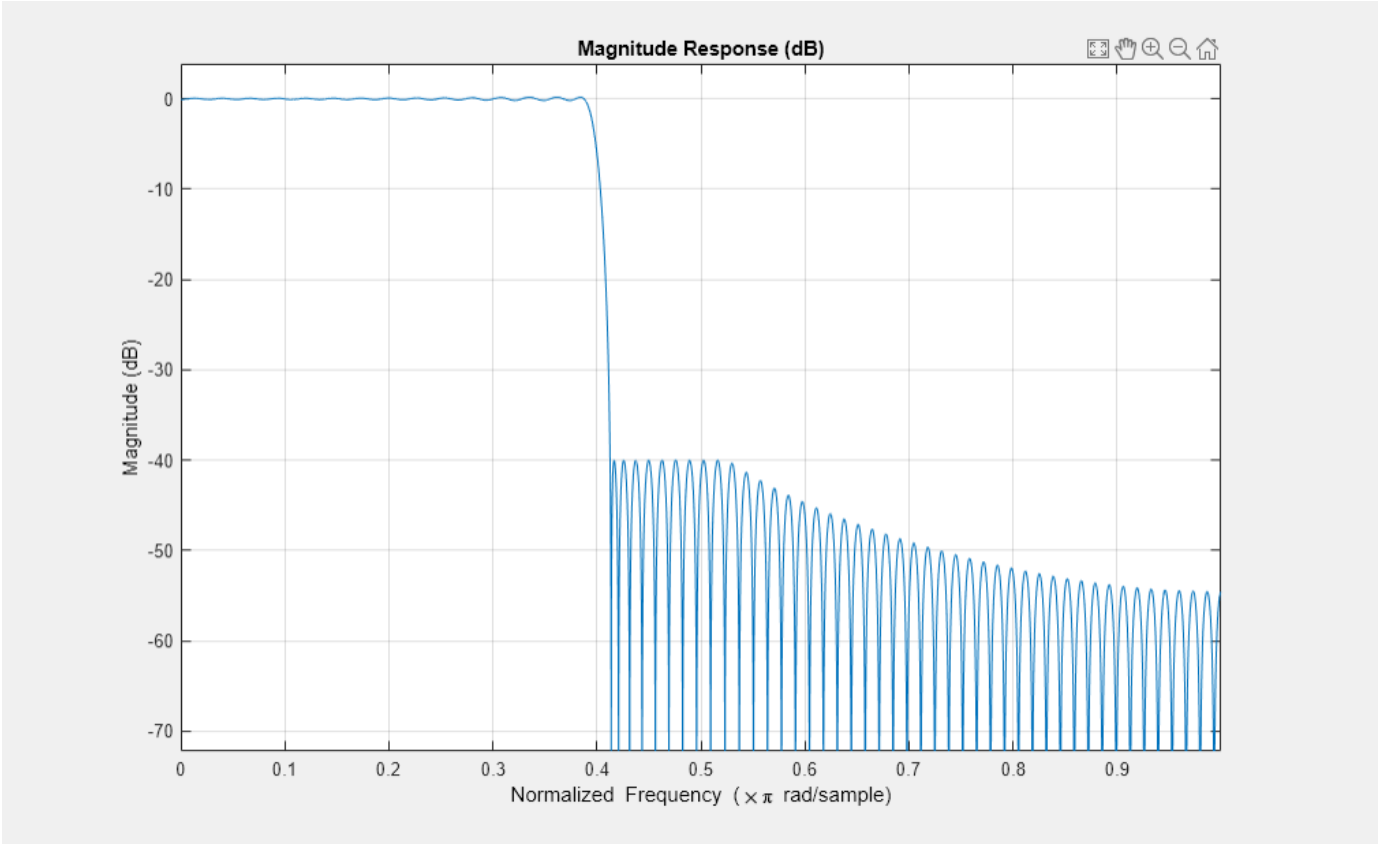
```
  Bound Violation = 0.0788344298966
  Bound Violation = 0.0096137744998
  Bound Violation = 0.0005681345753
  Bound Violation = 0.0000051519942
  Bound Violation = 0.0000000348656
```




```
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)
```



Input Arguments

`n` — Filter order

real positive scalar

Filter order, specified as a real positive scalar.

Note The `fircls` function 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.

`f` — Normalized frequency points

real-valued vector

Normalized frequency points, specified as a real-valued vector. The transition frequencies are in the range [0, 1], where 1 corresponds to the Nyquist frequency. The first point of `f` must be `0` and the last point must be `1`. The frequencies must be in increasing order.

`amp` — Piecewise-constant desired amplitude

real-valued vector

Piecewise-constant desired amplitude of the frequency response, specified as a real-valued vector. The length of `amp` is equal to the number of bands in the response, `length(f)-1`.

`up` — Upper bounds
real-valued vector

Upper bounds for the frequency response in each band, specified as a real-valued vector with the same length as `amp`.

`lo` — Lower bounds
real-valued vector

Lower bounds for the frequency response in each band, specified as a real-valued vector with the same length as `amp`.

Note Normally, the lower value in the stopband is specified as negative. By setting `lo` equal to `0` in the stopbands, a nonnegative frequency response amplitude is obtained. Such filters are spectrally factored to obtain minimum phase filters.

`"design_flag"` — Filter design display
`"trace"` | `"plots"` | `"both"`

Filter design display, specified as one of these:

- `"trace"` — View a textual display of the design error at each iteration step.
- `"plots"` — View a collection of plots showing the full-band magnitude response of the filter 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"` — View both a textual display and plots.

Output Arguments

`b` — Filter coefficients
row vector

Filter coefficients, returned as a row vector of length `n + 1`.

Algorithms

The `fircls` function 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`fircls1` | `firls` | `firpm`

Introduced before R2006a

fircls1

Constrained-least-squares linear-phase FIR lowpass and highpass 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 < wo < wt < 1`: the amplitude of the filter is within `ds` of 0 over the frequency range $wt < \omega < 1$.
- Highpass:
  - `0 < wt < wo < 1`: the amplitude of the filter is within `ds` of 0 over the frequency range $0 < \omega < wt$.
  - `0 < wo < wt < 1`: the amplitude of the filter is within `dp` of 1 over the frequency range $wt < \omega < 1$.

`b = fircls1(n,wo,dp,ds,wp,ws,k)` generates a lowpass FIR filter `b` with a weighted function, where `n+1` is the filter length, `wo` is the normalized cutoff frequency, `dp` is the maximum passband deviation from 1 (passband ripple), and `ds` is the maximum stopband deviation from 0 (stopband ripple). `wp` is the passband edge of the L2 weight function and `ws` is the stopband edge of the L2 weight function, where `wp < wo < ws`. `k` is the ratio (passband L2 error)/(stopband L2 error)

$$k = \frac{\int_0^{w_p} |A(\omega) - D(\omega)|^2 d\omega}{\int_{w_z}^{\pi} |A(\omega) - D(\omega)|^2 d\omega}$$

`b = fircls1(n,wo,dp,ds,wp,ws,k,'high')` generates a highpass FIR filter `b` with a weighted function, where `ws < wo < wp`.

`b = fircls1(n,wo,dp,ds,...,'design_flag')` enables you to monitor the filter design, where `'design_flag'` can be

- `'trace'`, for a textual display of the design table used in the design
- `'plots'`, for plots of the filter's magnitude, group delay, and zeros and poles. All plots are updated at each iteration step. The O's on the plot are the estimated extremals of the new iteration and the X's are the estimated extremals of the previous iteration, where the extremals are the peaks (maximum and minimum) of the filter ripples. Only ripples that have a corresponding O and X are made equal.
- `'both'`, for both the textual display and plots

Note In the design of very narrow band filters with small `dp` and `ds`, there may not exist a filter of the given length that meets the specifications.

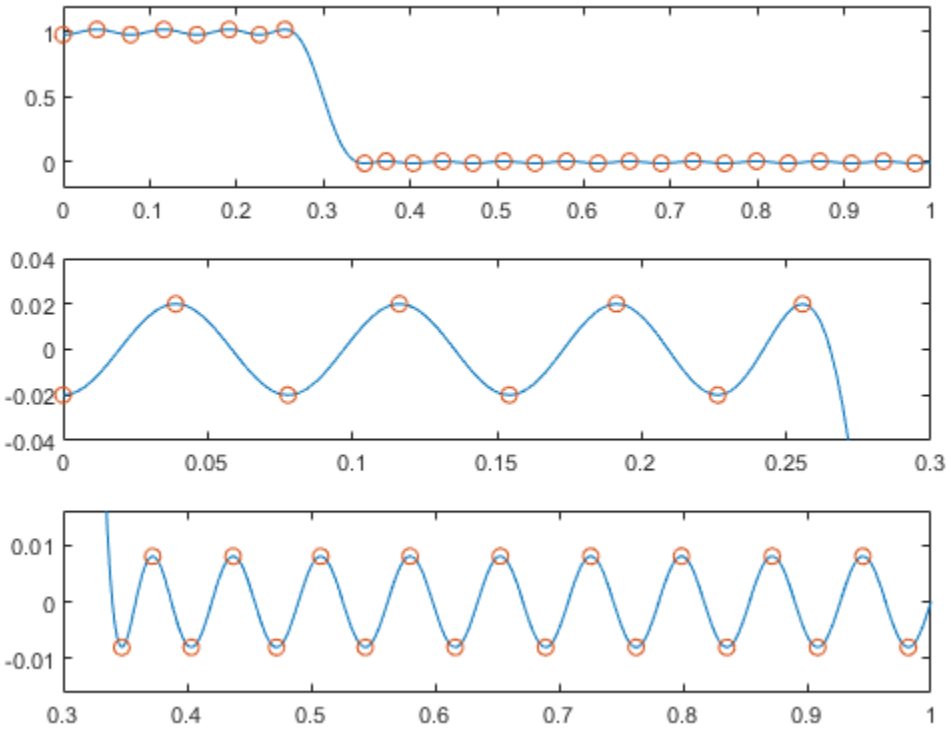
Examples

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');
```

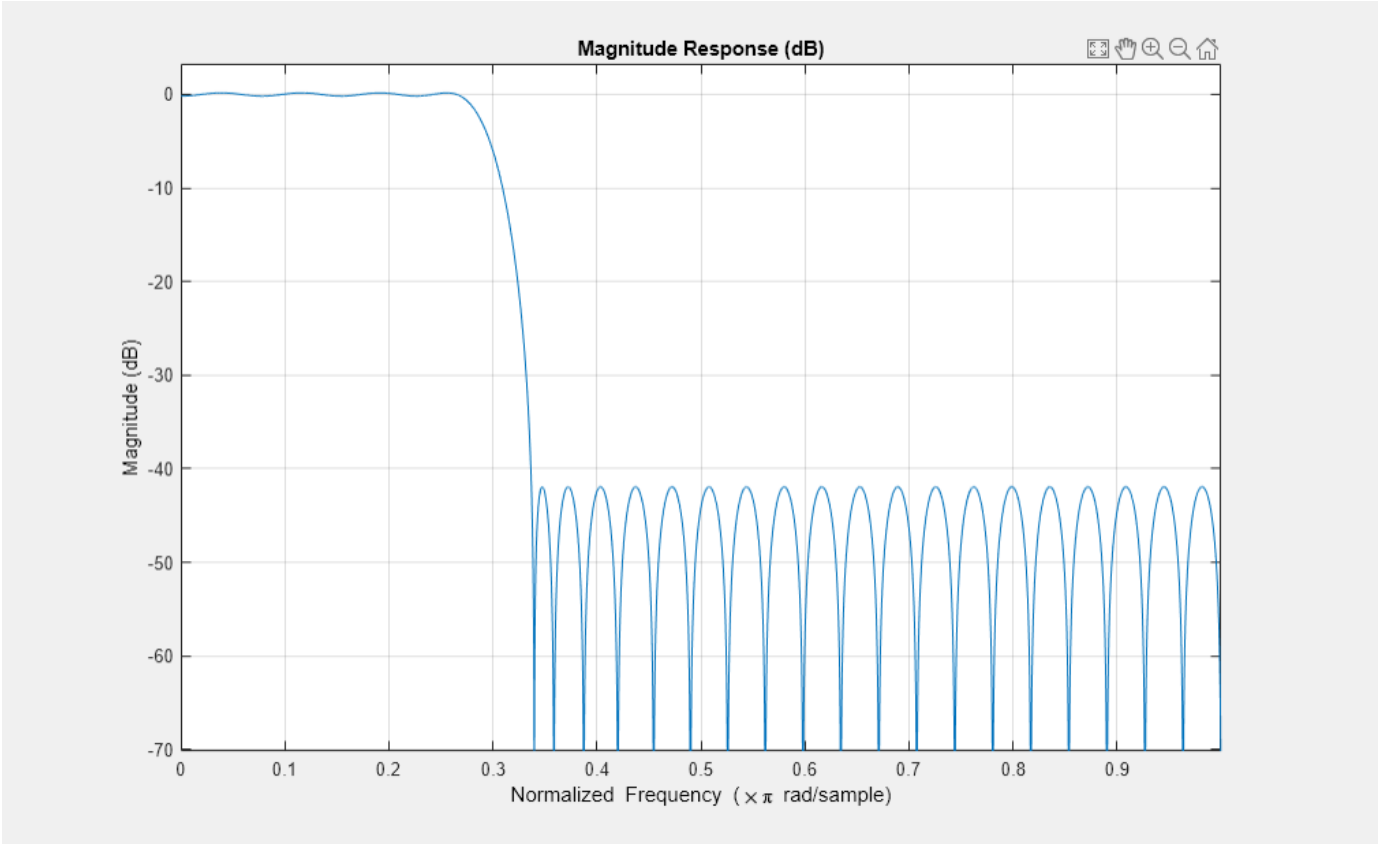
```
    Bound Violation = 0.0870385343920
    Bound Violation = 0.0149343456540
    Bound Violation = 0.0056513587932
    Bound Violation = 0.0001056264205
    Bound Violation = 0.0000967624352
    Bound Violation = 0.0000000226538
```




```
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)
```



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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`fircls` | `firls` | `firpm`

Introduced before R2006a

firls

Least-squares linear-phase FIR filter design

Syntax

```
b = firls(n,f,a)
b = firls(n,f,a,w)
b = firls( ___ ,ftype)
```

Description

`b = firls(n,f,a)` returns row vector `b` containing the `n+1` coefficients of an order-`n` FIR filter. The frequency and amplitude characteristics of the resulting filter match those given by vectors `f` and `a`.

`b = firls(n,f,a,w)` uses `w` to weigh the frequency bins.

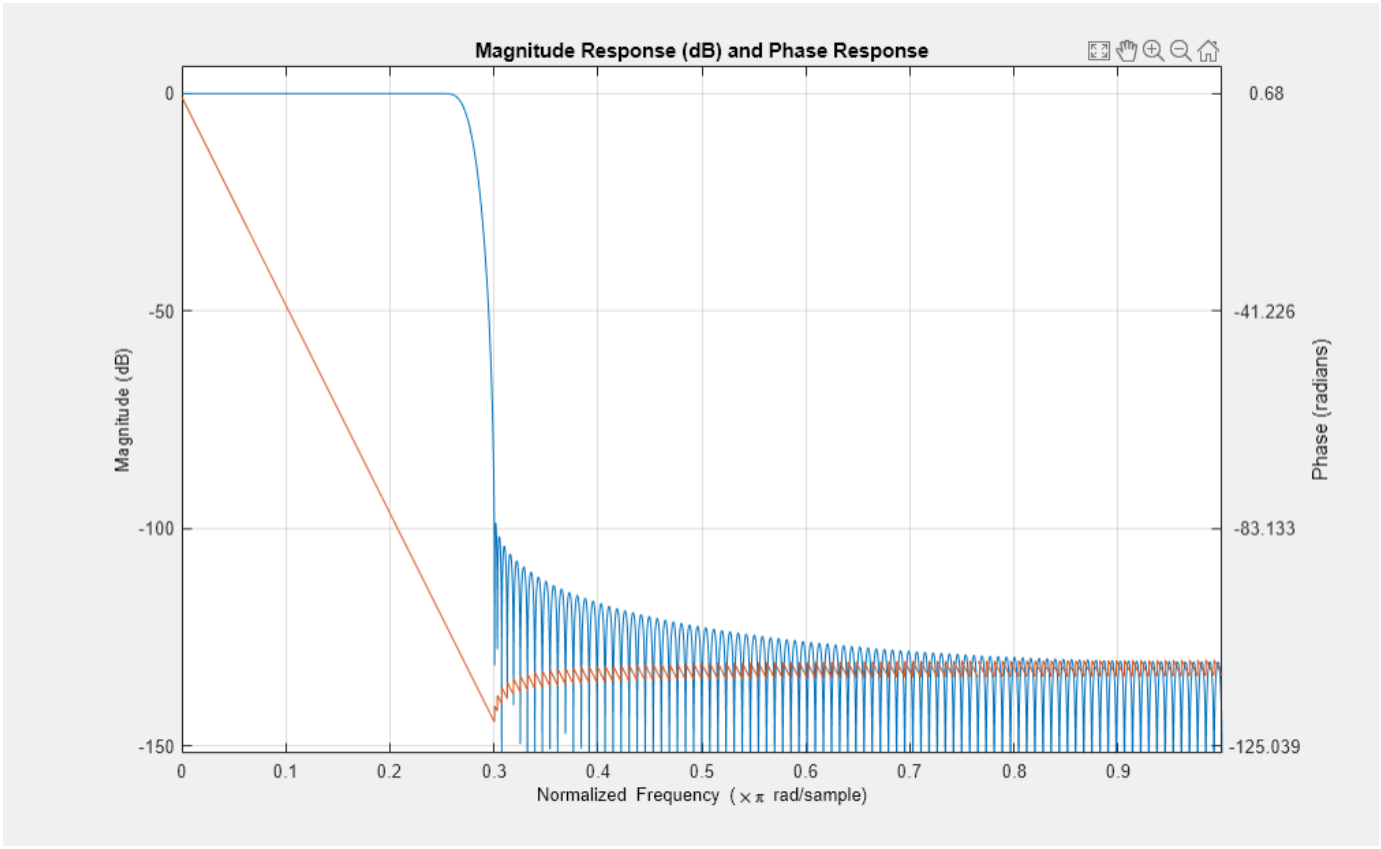
`b = firls( ___ ,ftype)` designs antisymmetric (odd) filters, where `ftype` specifies the filter as a differentiator or Hilbert transformer. You can use `ftype` with any of the previous input syntaxes.

Examples

Filter with Transition Band

Design an FIR lowpass filter of order 255 with a transition region between 0.25π and 0.3π. Use `fvtool` to display the magnitude and phase responses of the filter.

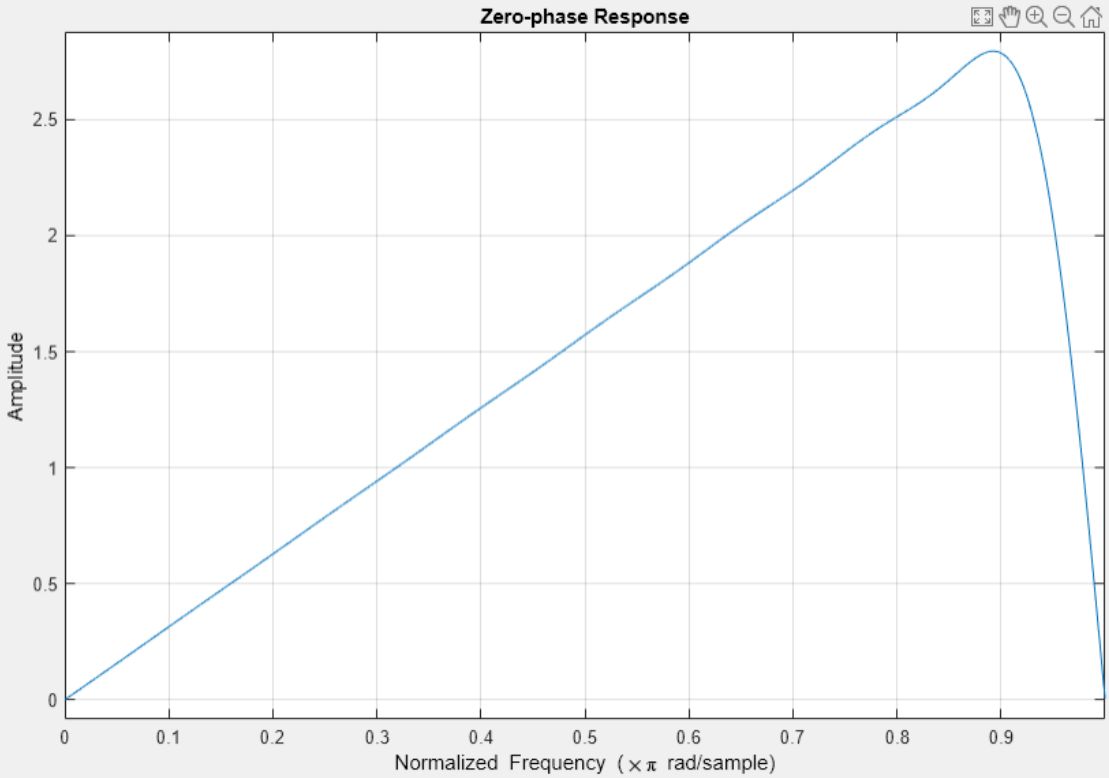
```
b = firls(255,[0 0.25 0.3 1],[1 1 0 0]);
fvtool(b,1,'OverlayedAnalysis','phase')
```



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 0.9π. Include a factor of π in the amplitude because the frequencies are normalized by π. 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')
```



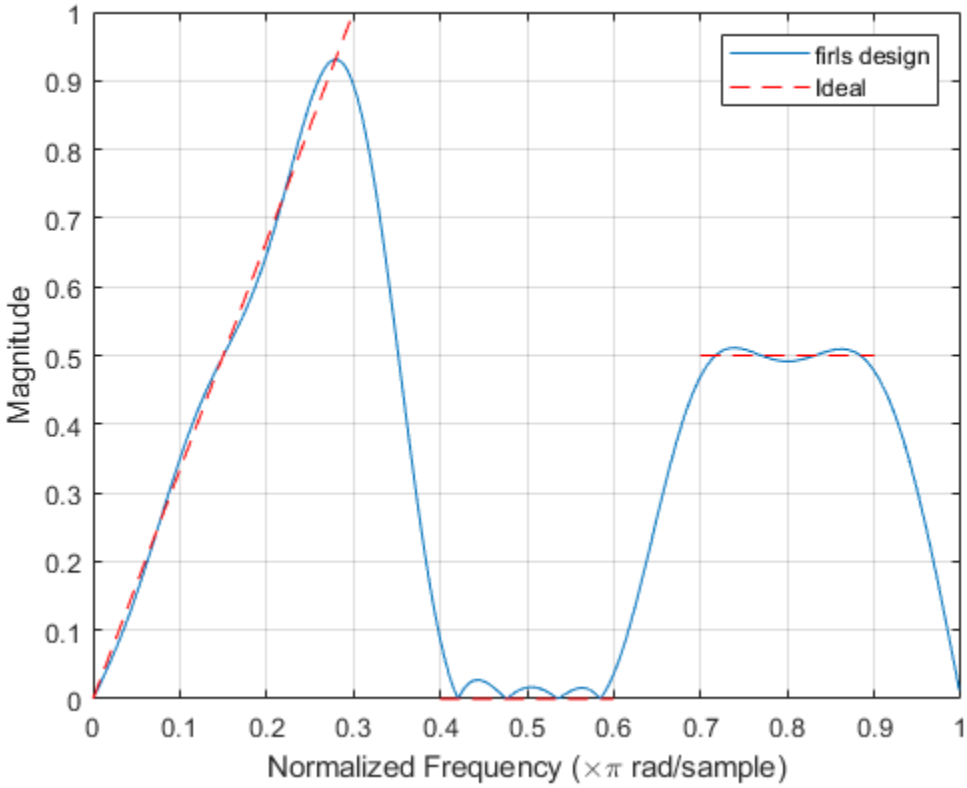
Filter with Piecewise Linear Passbands

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')
```



Lowpass Filter Design with Weighted Fit

Design an FIR lowpass filter. The passband ranges from DC to 0.45π rad/sample. The stopband ranges from 0.55π rad/sample to the Nyquist frequency. Produce three different designs, changing the weights of the bands in the least-squares fit.

In the first design, make the stopband weight higher than the passband weight by a factor of 100. Use this specification when it is critical that the magnitude response in the stopband is flat and close to 0. The passband ripple is about 100 times higher than the stopband ripple.

```
bhi = firls(18,[0 0.45 0.55 1],[1 1 0 0],[1 100]);
```

In the second design, reverse the weights so that the passband weight is 100 times the stopband weight. Use this specification when it is critical that the magnitude response in the passband is flat and close to 1. The stopband ripple is about 100 times higher than the passband ripple.

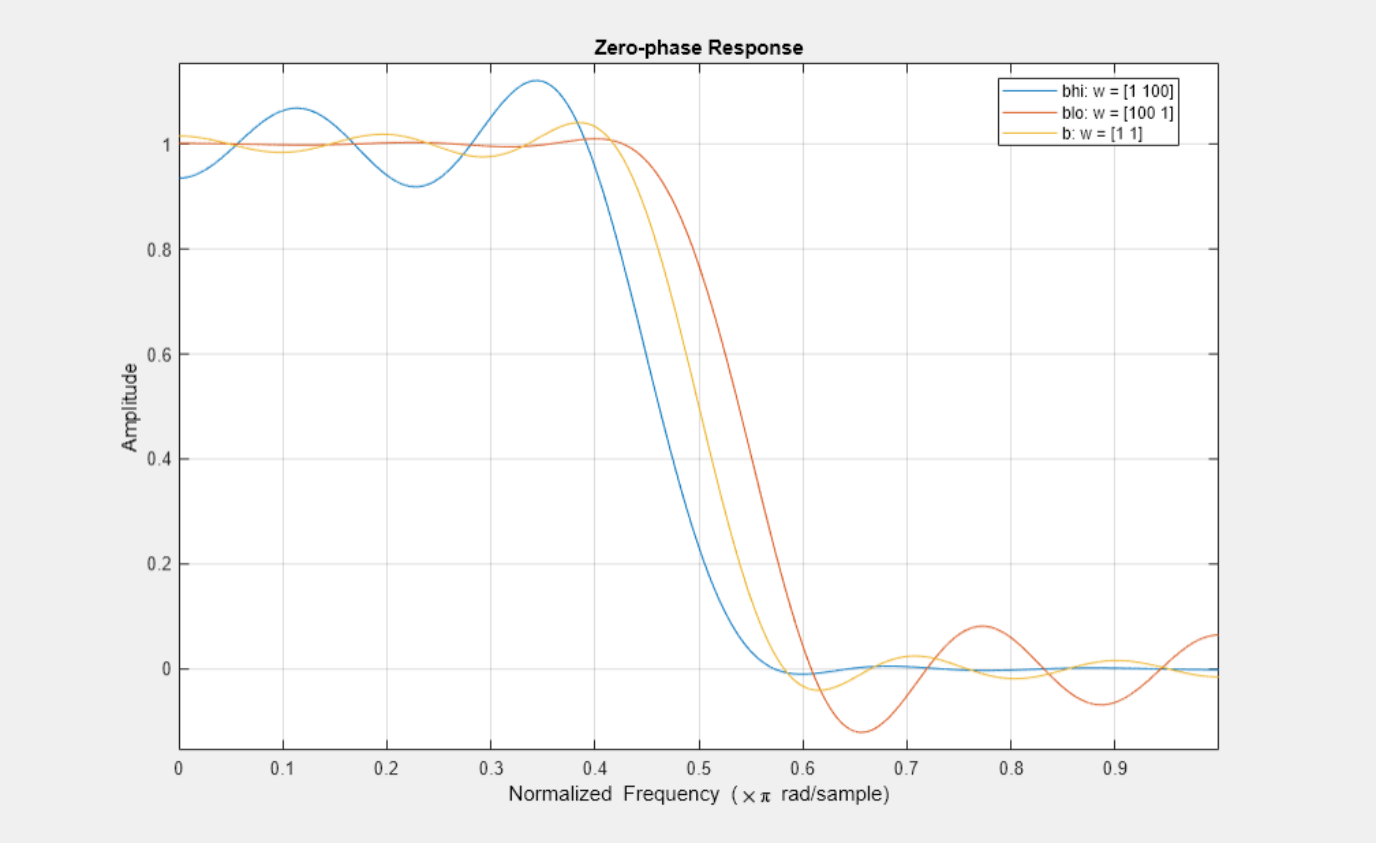
```
blo = firls(18,[0 0.45 0.55 1],[1 1 0 0],[100 1]);
```

In the third design, give the same weight to both bands. The result is a filter with similar ripple in the passband and the stopband.

```
b = firls(18,[0 0.45 0.55 1],[1 1 0 0],[1 1]);
```

Visualize the magnitude responses of the three filters.

```
hfvt = fvtool(bhi,1,blo,1,b,1,'MagnitudeDisplay','Zero-phase');
legend(hfvt,'bhi: w = [1 100]','blo: w = [100 1]','b: w = [1 1]')
```



Input Arguments

n — Filter order

real positive scalar

Filter order, specified as a real positive scalar.

f — Normalized frequency points

real-valued vector

Normalized frequency points, specified as a real-valued vector. The argument must be in the range [0, 1] , where 1 corresponds to the Nyquist frequency. The number of elements in the vector is always a multiple of 2. The frequencies must be in nondecreasing order.

a — Desired amplitude

vector

Desired amplitudes at the points specified in `f`, specified as a vector. `f` and `a` must be the same length. The length must be an even number.

- 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 regions or regions that are not important for a particular application.

`w` — Weights
real-valued vector

Weights used to adjust the fit in each frequency band, specified as a real-valued vector. The length of `w` is half the length of `f` and `a`, so there is exactly one weight per band.

`ftype` — Filter type
`'hilbert'` | `'differentiator'`

Filter type for linear-phase filters with odd symmetry (type III and type IV), specified as either `'hilbert'` or `'differentiator'`:

- `'hilbert'` — The output coefficients in `b` obey the relation $b(k) = -b(n + 2 - k)$, $k = 1, ..., n + 1$. This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.
- `'differentiator'` — For nonzero amplitude bands, the filter weighs the error by a factor 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, these filters minimize the maximum relative error (the maximum of the ratio of the error to the desired amplitude).

Output Arguments

`b` — Filter coefficients
row vector

Filter coefficients, returned as a row vector of length `n + 1`. The coefficients are in increasing order.

More About

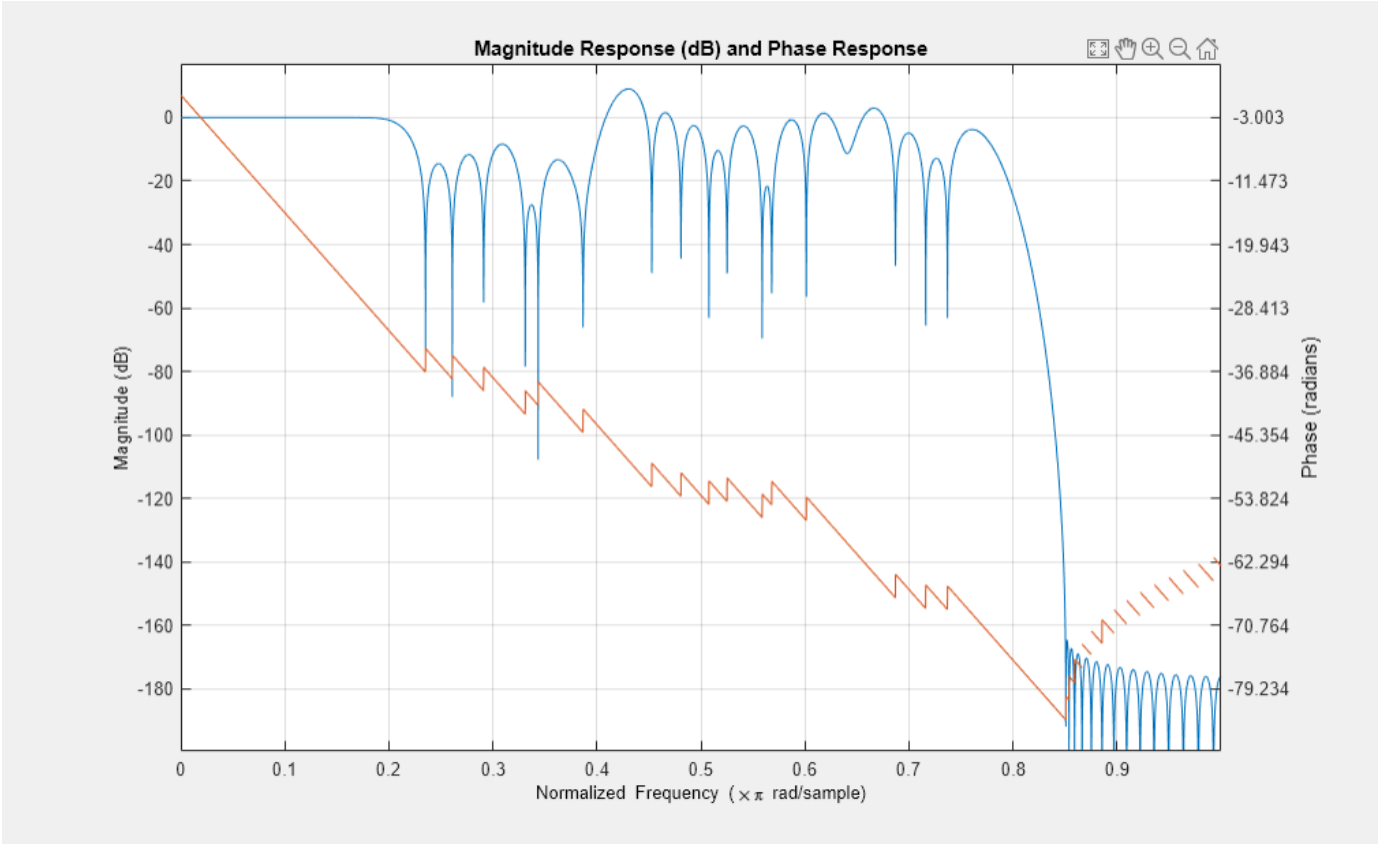
Filter Length and Transition Width Incompatibility

If you design a filter such that the product of the filter length and the transition width is large, you might get this warning message: `Matrix is close to singular or badly scaled`. The following example illustrates this limitation.

```
b = firls(100,[0 0.15 0.85 1],[1 1 0 0]);
```

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  2.6024
```

```
fvtool(b,1,'OverlayedAnalysis','phase')
```



In this case, the filter coefficients `b` might not represent the desired filter. You can check the filter by looking at its frequency response.

Algorithms

`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.

Reference [2] describes the theoretical approach behind `firls`. The function solves a system of linear equations involving an inner product matrix of roughly the size `n\2` using the MATLAB `\` operator.

These are type I (`n` is odd) and type II (`n` is even) linear-phase filters. Vectors `f` and `a` specify the frequency-amplitude characteristics of the filter:

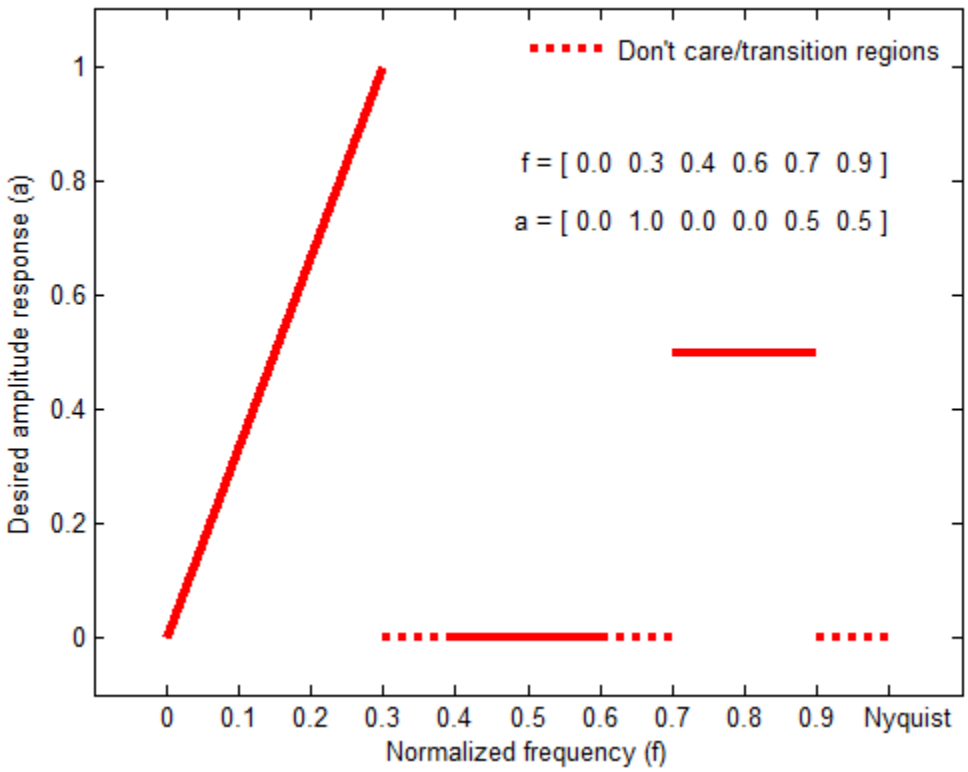
- `f` is a vector of pairs of frequency points, specified in the range 0 to 1, where 1 corresponds to the Nyquist frequency. The frequencies must be in increasing order. Duplicate frequency points are allowed and, in fact, can be used to design a filter that is exactly the same as the filters returned by the `fir1` and `fir2` functions with a rectangular (`rectwin`) window.
- `a` is a vector containing the desired amplitudes 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 ("don't care") regions.

- `f` and `a` are the same length. This length must be an even number.

This figure illustrates the relationship between the `f` and `a` vectors in defining a desired amplitude response.



This function designs type I, II, III, and IV linear-phase filters. Type I and II are the default filters when n is even and odd, respectively, while the `'hilbert'` and `'differentiator'` flags produce type III (n is even) and IV (n is odd) filters. The various filter types have different symmetries and constraints on their frequency responses (see [1] 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	$b(k) = b(n+2-k), \quad k = 1, \ldots, n+1$	No restriction	No restriction
Type II	Odd	$b(k) = b(n+2-k), \quad k = 1, \ldots, n+1$	No restriction	$H(1) = 0$
Type III	Even	$b(k) = -b(n+2-k), \quad k = 1, \ldots, n+1$	$H(0) = 0$	$H(1) = 0$
Type IV	Odd	$b(k) = -b(n+2-k), \quad k = 1, \ldots, n+1$	$H(0) = 0$	No restriction

References

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

[2] Parks, Thomas W., and C. Sidney Burrus. *Digital Filter Design*. Hoboken, NJ: John Wiley & Sons, 1987, pp. 54–83.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`fir1` | `fir2` | `firpm` | `rcosdesign`

Introduced before R2006a

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,lgrid)
[b,err] = firpm( ___ )
[b,err,res] = firpm( ___ )
b = firpm(n,f,fresp,w)
b = firpm(n,f,fresp,w,ftype)
```

Description

`b = firpm(n,f,a)` returns row vector `b` containing the `n+1` coefficients of an order-`n` FIR filter. The frequency and amplitude characteristics of the resulting filter match those given by vectors `f` and `a`.

`b = firpm(n,f,a,w)` uses `w` to weigh the frequency bins.

`b = firpm(n,f,a,ftype)` uses a filter type specified by `'ftype'`.

`b = firpm(n,f,a,lgrid)` uses the integer `lgrid` to control the density of the frequency grid.

`[b,err] = firpm( ___ )` returns the maximum ripple height in `err`. You can use this with any of the previous input syntaxes.

`[b,err,res] = firpm( ___ )` returns the frequency response characteristics as a structure `res`.

`b = firpm(n,f,fresp,w)` returns an FIR filter whose frequency-amplitude characteristics best approximate the response returned by function handle `fresp`.

`b = firpm(n,f,fresp,w,ftype)` designs antisymmetric (odd) filters, where `ftype` specifies the filter as a differentiator or Hilbert transformer. If you do not specify an `ftype`, a call is made to `fresp` to determine the default symmetry property.

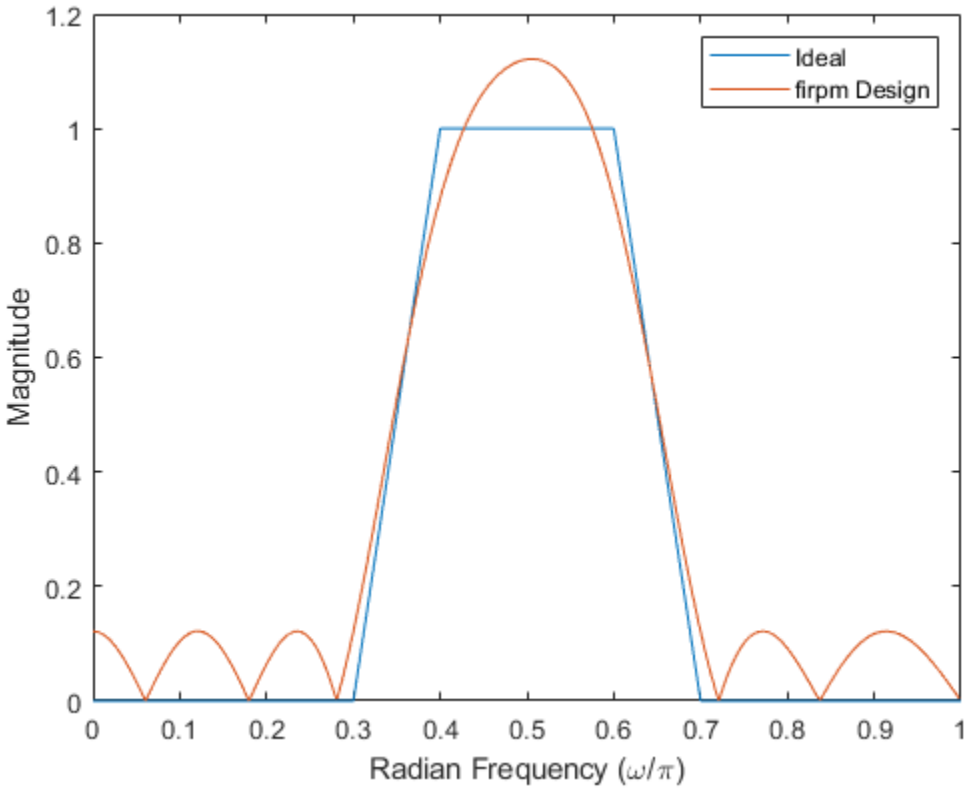
Examples

Parks-McClellan Bandpass Filter

Use the Parks-McClellan algorithm to design an FIR bandpass filter of order 17. Specify normalized stopband frequencies of 0.3π and 0.7π rad/sample and normalized passband frequencies of 0.4π and 0.6π rad/sample. Plot the ideal and actual magnitude responses.

```
f = [0 0.3 0.4 0.6 0.7 1];
a = [0 0 1 1 0 0];
b = firpm(17,f,a);
```

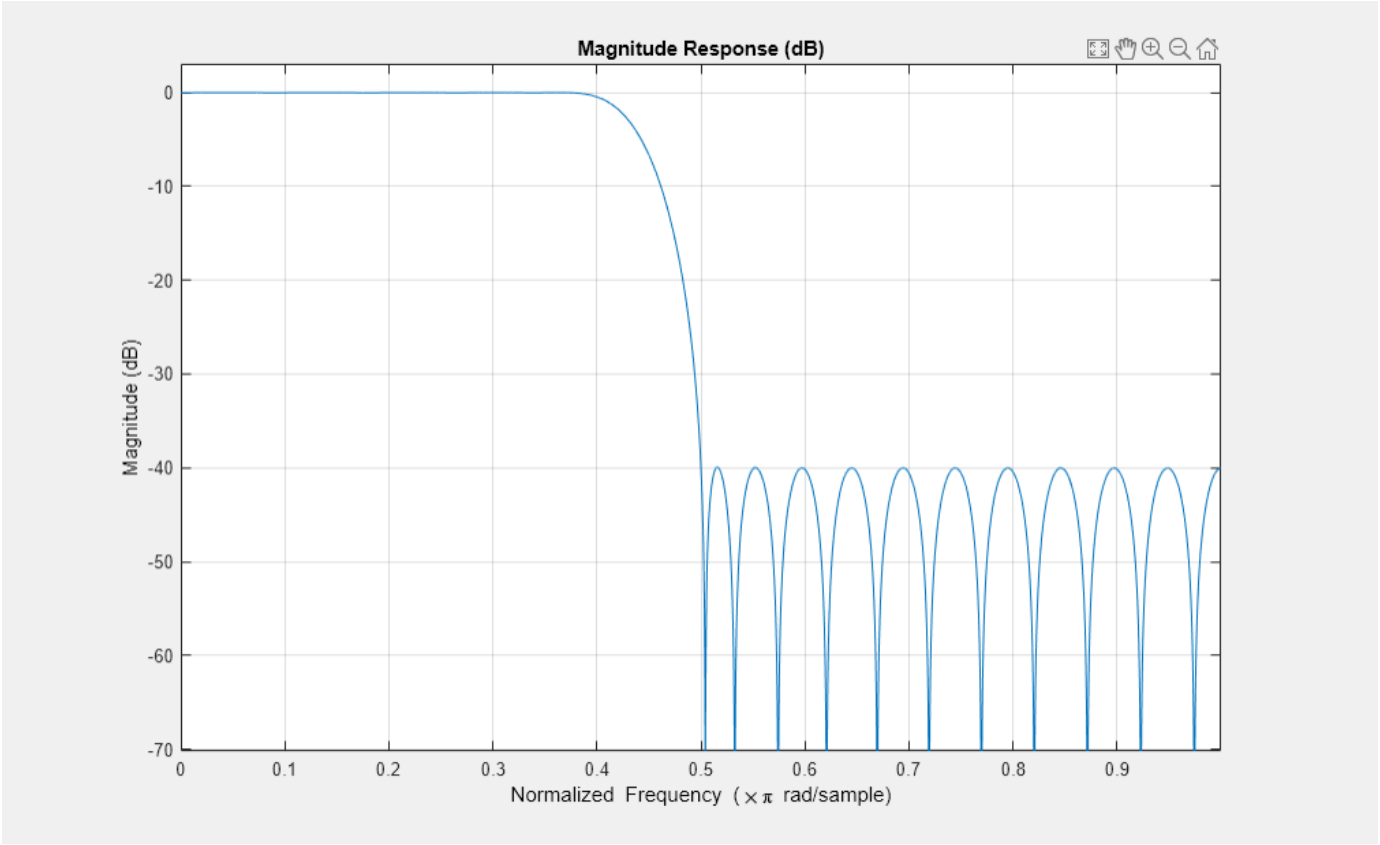
```
[h,w] = freqz(b,1,512);
plot(f,a,w/pi,abs(h))
legend('Ideal','firpm Design')
xlabel 'Radian Frequency (\omega/\pi)', ylabel 'Magnitude'
```



Parks-McClellan 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 0.01 and a maximum passband error (ripple) of 0.001. Obtain the required filter order, normalized frequency band edges, frequency band amplitudes, and weights using `firpmord`.

```
[n,fo,ao,w] = firpmord([1500 2000],[1 0],[0.001 0.01],8000);
b = firpm(n,fo,ao,w);
fvtool(b,1)
```



FIR Bandpass Filter with Asymmetric Attenuation

Use the Parks-McClellan algorithm to create a 50th-order equiripple FIR bandpass filter to be used with signals sampled at 1 kHz.

```
N = 50;
Fs = 1e3;
```

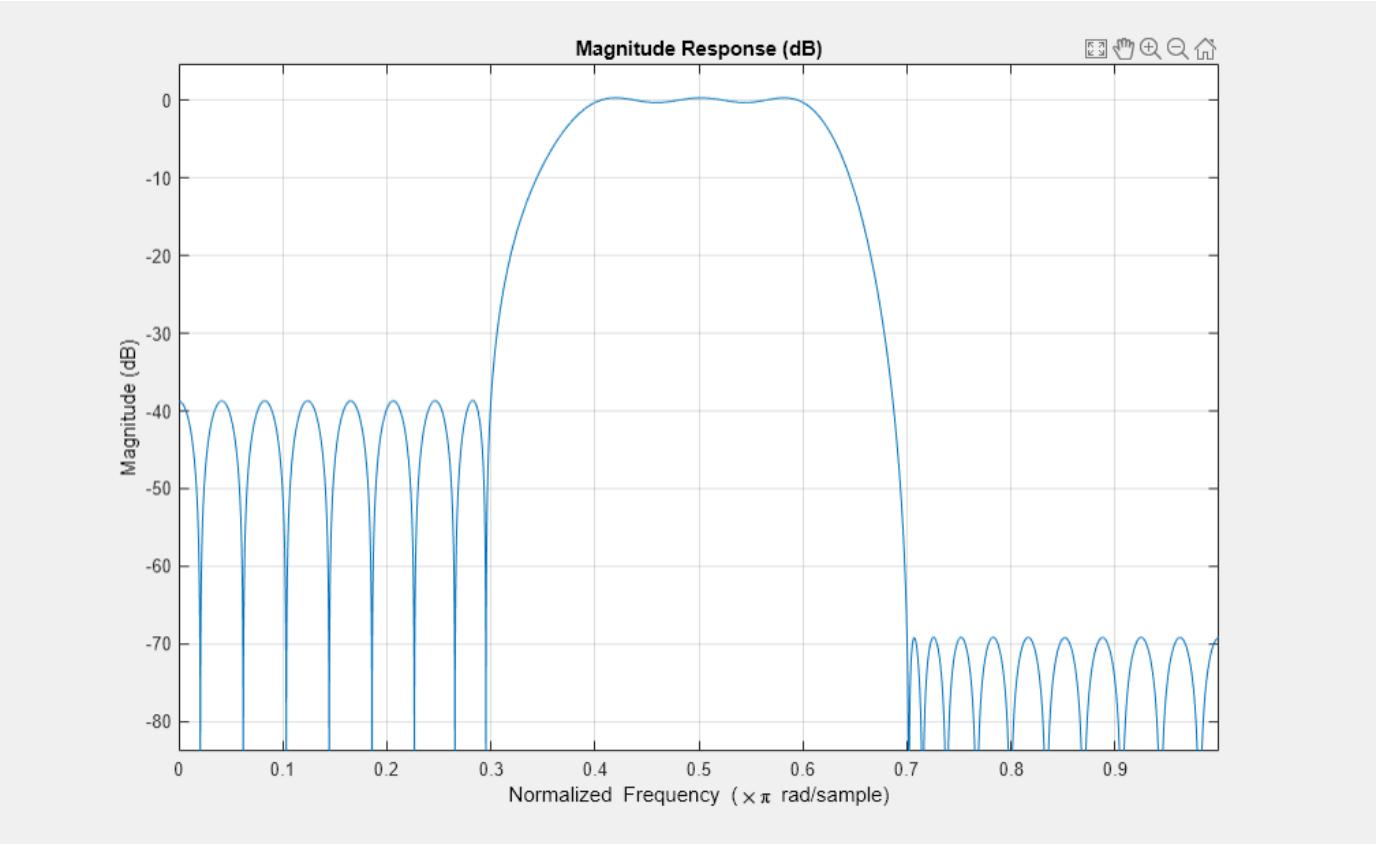
Specify that the passband spans the frequencies between 200 Hz and 300 Hz and that the transition region on either side of the passband has a width of 50 Hz.

```
Fstop1 = 150;
Fpass1 = 200;
Fpass2 = 300;
Fstop2 = 350;
```

Design the filter so that the optimization fit weights the low-frequency stopband with a weight of 3, the passband with a weight of 1, and the high-frequency stopband with a weight of 100. Display the magnitude response of the filter.

```
Wstop1 = 3;
Wpass = 1;
Wstop2 = 100;
```

```
b = firpm(N,[0 Fstop1 Fpass1 Fpass2 Fstop2 Fs/2]/(Fs/2), ...
    [0 0 1 1 0 0],[Wstop1 Wpass Wstop2]);

fvtool(b,1)
```



Input Arguments

n — Filter order

real positive scalar

Filter order, specified as a real positive scalar.

f — Normalized frequency points

real-valued vector

Normalized frequency points, specified as a real-valued vector. The argument must be in the range [0, 1] , where 1 corresponds to the Nyquist frequency. The number of elements in the vector is always a multiple of 2. The frequencies must be in increasing order.

a — Desired amplitude

vector

Desired amplitudes at the points specified in `f`, specified as a vector. `f` and `a` must be the same length. The length must be an even number.

- 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 regions or regions that are not important for a particular application.

`w` — Weights

real-valued vector

Weights used to adjust the fit in each frequency band, specified as a real-valued vector. The length of `w` is half the length of `f` and `a`, so there is exactly one weight per band.

`ftype` — Filter type

`'hilbert'` | `'differentiator'`

Filter type for linear-phase filters with odd symmetry (type III and type IV), specified as either `'hilbert'` or `'differentiator'`:

- `'hilbert'` — The output coefficients in `b` obey the relation $b(k) = -b(n + 2 - k)$, $k = 1, ..., n + 1$. 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 nonzero amplitude bands, the filter weighs 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).

`lgrid` — Density of frequency grid

16 (default) | 1-by-1 cell array with integer value

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`.

`fresp` — Frequency response

function handle

Frequency response, specified as a function handle. The function is called from within `firpm` with this 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`.

Output Arguments

`b` — Filter coefficients

row vector

Filter coefficients, returned as a row vector of length `n + 1`. The coefficients are in increasing order.

`err` — Maximum ripple height

scalar

Maximum ripple height, returned as a scalar.

`res` — Frequency response characteristics

structure

Frequency response characteristics, returned as a structure. The structure `res` has the following fields:

`res.fgrid`	Frequency grid vector used for the filter design optimization
`res.des`	Desired frequency response for each point in `res.fgrid`
`res.wt`	Weighting for each point in `opt.fgrid`
`res.H`	Actual frequency response for each point in `res.fgrid`
`res.error`	Error at each point in `res.fgrid` (`res.des-res.H`)
`res.iextr`	Vector of indices into `res.fgrid` for extremal frequencies
`res.fextr`	Vector of extremal frequencies

Tips

If your filter design fails to converge, the filter design might not be 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.

Algorithms

`firpm` designs a linear-phase FIR filter using the Parks-McClellan algorithm [2]. 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.

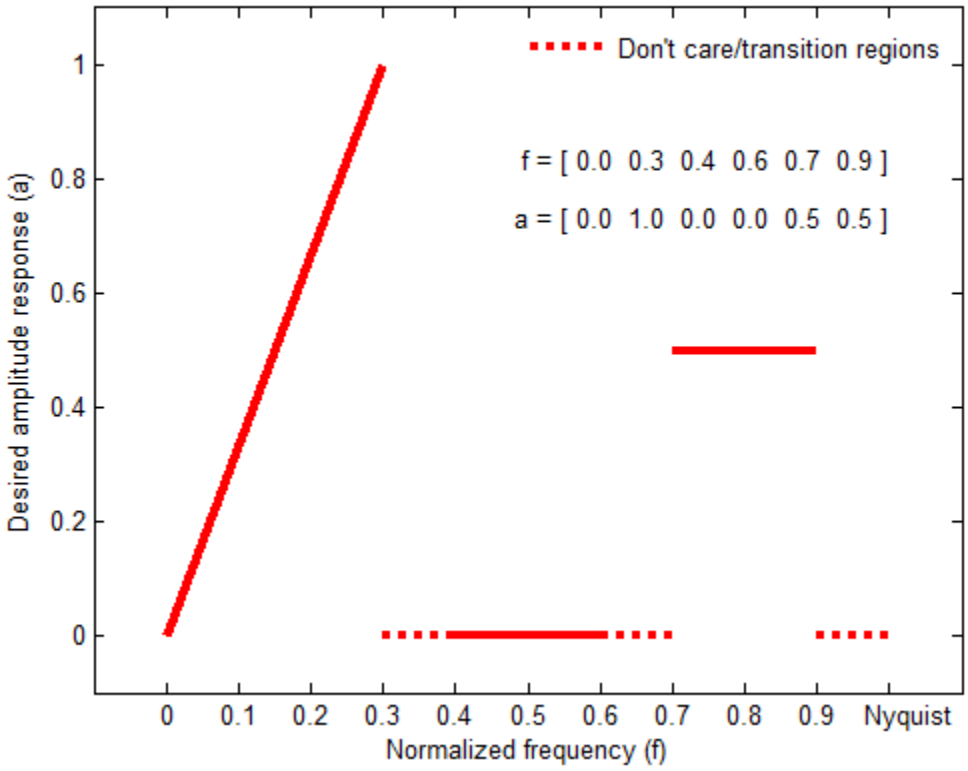
These are type I (`n` odd) and type II (`n` even) linear-phase filters. Vectors `f` and `a` specify the frequency-amplitude characteristics of the filter:

- `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.

The figure below illustrates the relationship between the `f` and `a` vectors in defining a desired amplitude response.



`firpm` always uses an even filter order for configurations with even symmetry and a nonzero passband at the Nyquist frequency. The reason for the even filter order is that 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.

`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 specified with `'hilbert'` or `'differentiator'`, respectively, using the `ftype` argument. The different types of filters have different symmetries and certain constraints on their frequency responses. (See [3] for more details.)

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

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

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.

Note `b = firpm(n,f,a,w)` is equivalent to `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` and see “Create Function Handle” for more information.

References

[1] 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.

[2] 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 5.1.

[3] Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1999, p. 486.

[4] Parks, Thomas W., and C. Sidney Burrus. *Digital Filter Design*. New York: John Wiley & Sons, 1987, p. 83.

[5] 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, Number 4, 1975, pp. 595–610.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`butter` | `cheby1` | `cheby2` | `cfirpm` | `ellip` | `fir1` | `fir2` | `fircls` | `fircls1` | `firls` | `firpmord` | `rcosdesign` | `yulewalk`

Introduced before R2006a

firpmord

Parks-McClellan optimal FIR filter order estimation

Syntax

```
[n,fo,ao,w] = firpmord(f,a,dev)
[___] = firpmord(___,fs)
c = firpmord(___,'cell')
```

Description

`[n,fo,ao,w] = firpmord(f,a,dev)` returns the approximate order `n`, normalized frequency band edges `fo`, frequency band amplitudes `ao`, and weights `w` that meet input specifications `f`, `a`, and `dev`.

`[___] = firpmord(___,fs)` specifies a sampling frequency `fs`. `fs` defaults to 2 Hz, implying a Nyquist frequency of 1 Hz. You can specify band edges scaled to a particular application's sample rate. You can use this with any of the previous input syntaxes.

`c = firpmord(___,'cell')` returns a cell array `c` whose elements are the parameters to `firpm`.

Examples

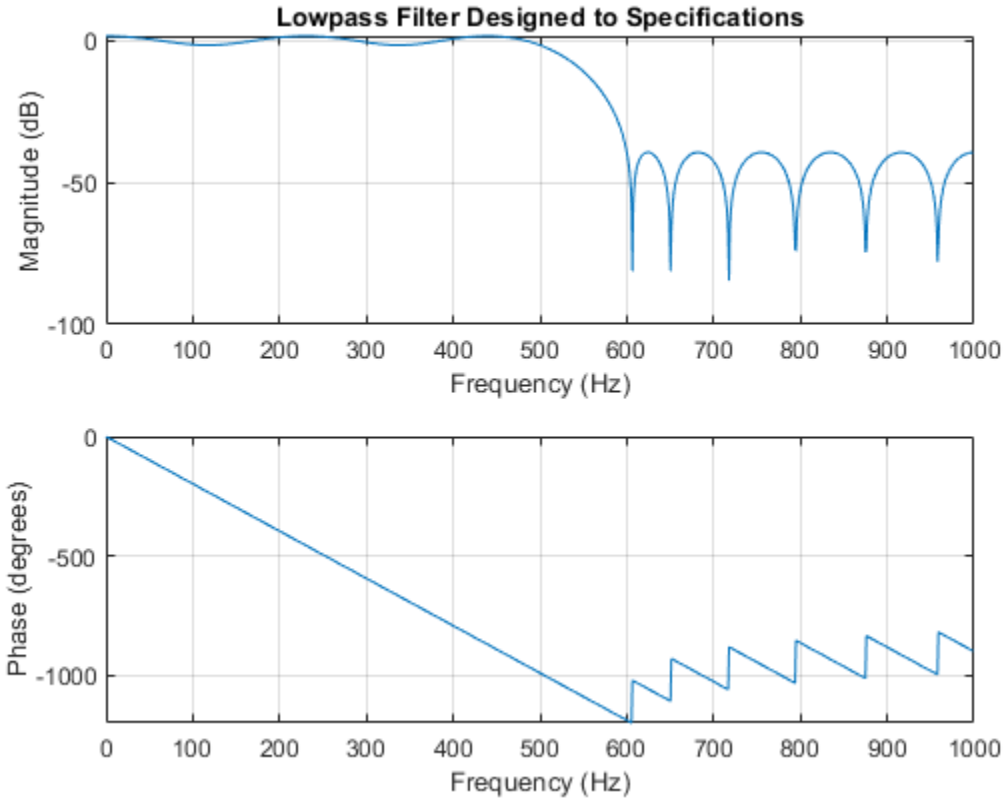
Minimum-Order Lowpass Filter

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 in dB
rs = 40;          % Stopband ripple in dB
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')
```



The filter falls slightly short of meeting the stopband attenuation and passband ripple specifications. Using `n+1` instead of `n` in the call to `firpm` achieves the desired amplitude characteristics.

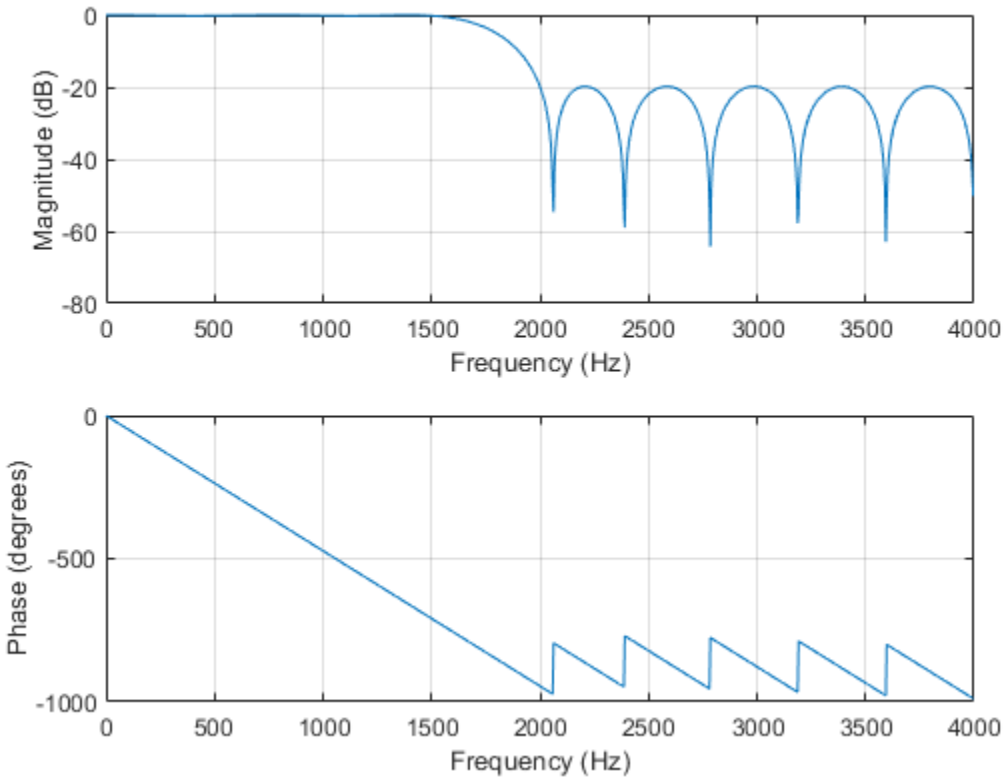
Parks-McClellan Order of Lowpass Filter

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)
```



Input Arguments

`f` — Frequency band edges

real-valued vector

Frequency band edges, specified as a real-valued vector. The argument must be in the range [0, F_s/2], where F_s is the Nyquist frequency. The number of elements in the vector is always a multiple of 2. The frequencies must be in increasing order.

`a` — Desired amplitude

vector

Desired amplitudes at the points contained in `f`, specified as a vector. `f` and `a` must satisfy the condition length(f) = 2length(a)–2. The desired function is piecewise constant.

`dev` — Maximum allowable deviation

vector

Maximum allowable deviation, specified as a vector. `dev` has the same size as `a`. It specifies the maximum allowable deviation or ripples between the frequency response and the desired amplitude of the output filter for each band.

`fs` — Sample rate

2 Hz (default) | real scalar

Sample rate, specified as a real scalar.

Output Arguments

`n` — Filter order
positive integer

Filter order, returned as a positive integer.

`fo` — Normalized frequency points
real-valued vector

Normalized frequency points, specified as a real-valued vector. The argument must be in the range [0, 1] , where 1 corresponds to the Nyquist frequency. The number of elements in the vector is always a multiple of 2. The frequencies must be in increasing order.

`ao` — Amplitude response
real-valued vector

Amplitude response, returned as a real-valued vector.

`w` — weights
real-valued vector

Weights used to adjust the fit in each frequency band, specified as a real-valued vector. The length of `w` is half the length of `f` and `a`, so there is exactly one weight per band.

`c` — FIR filter parameters
cell array

FIR filter parameters, returned as a cell array.

Algorithms

`firpmord` uses the algorithm suggested in [1]. This function produces inaccurate results for band edges close to either 0 or the Nyquist frequency, `fs/2`.

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`.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`buttord` | `cheb1ord` | `cheb2ord` | `ellipord` | `kaiserord` | `firpm`

Introduced before R2006a

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.

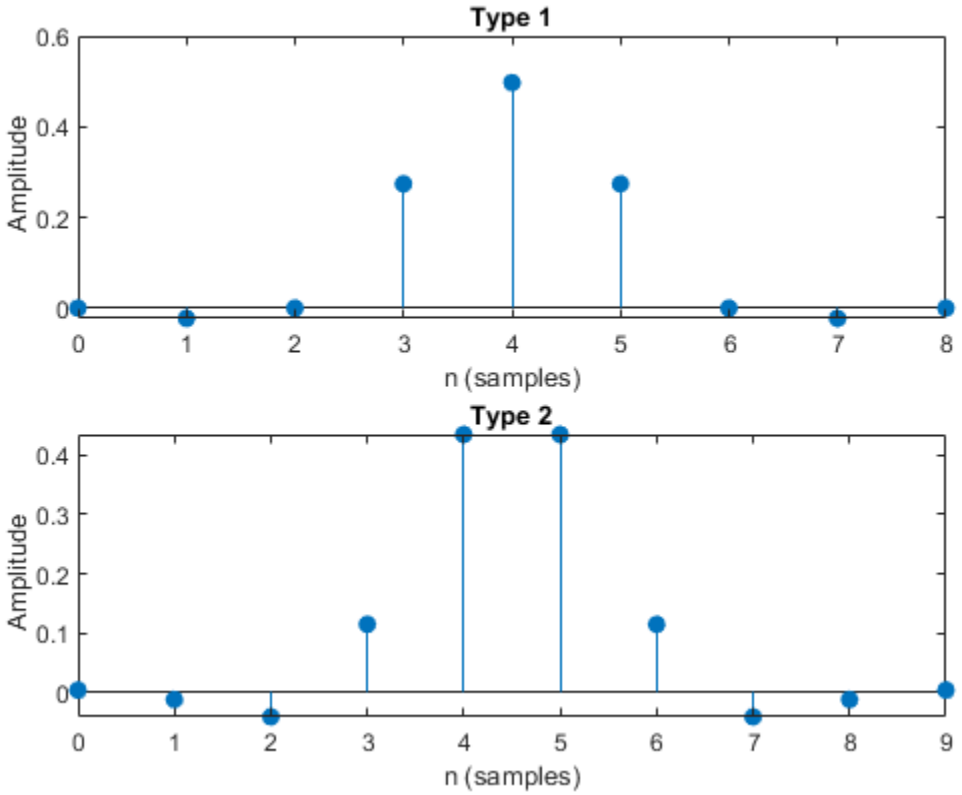
`t = firtype(d)` determines the type, `t`, of an FIR filter, `d`. `t` can be 1, 2, 3, or 4. The filter must be real and have linear phase.

Examples

Types of Linear Phase Filters

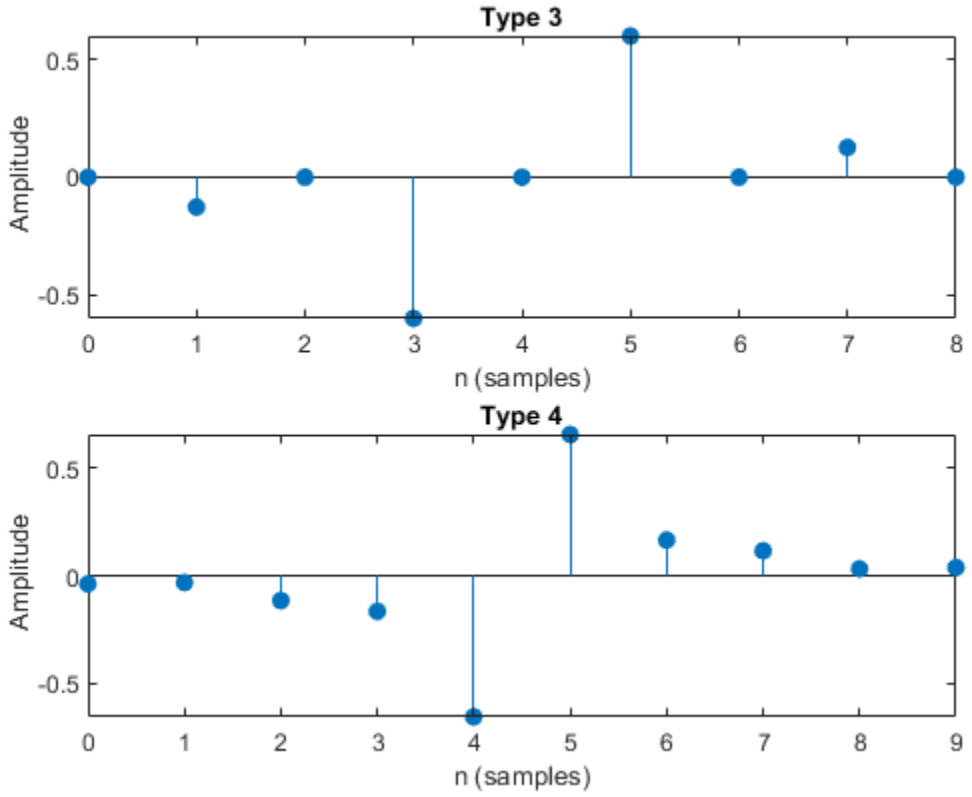
Design two FIR filters using the window method, one of even order and the other of odd order. Determine their types and plot their impulse responses.

```
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))])
```

```
d1 is of type 1
```

```
d2 = designfilt('lowpassfir','DesignMethod','window', ...
                'FilterOrder',9,'CutoffFrequency',0.5);
disp(['d2 is of type ' int2str(firtype(d2))])
```

```
d2 is of type 2
```

```
d3 = designfilt('hilbertfir','DesignMethod','equiripple', ...
                'FilterOrder',8,'TransitionWidth',0.4);
disp(['d3 is of type ' int2str(firtype(d3))])
```

```
d3 is of type 3
```

```
d4 = designfilt('hilbertfir','DesignMethod','equiripple', ...
                'FilterOrder',9,'TransitionWidth',0.4);
disp(['d4 is of type ' int2str(firtype(d4))])
```

```
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

FIR filter, specified as a `digitalFilter` object. Use `designfilt` to generate a digital filter based on frequency-response specifications.

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`

Introduced in R2013a

flattopwin

Flat top weighted window

Syntax

```
w = flattopwin(L)
w = flattopwin(L,sflag)
```

Description

`w = flattopwin(L)` returns an `L`-point symmetric flat top window

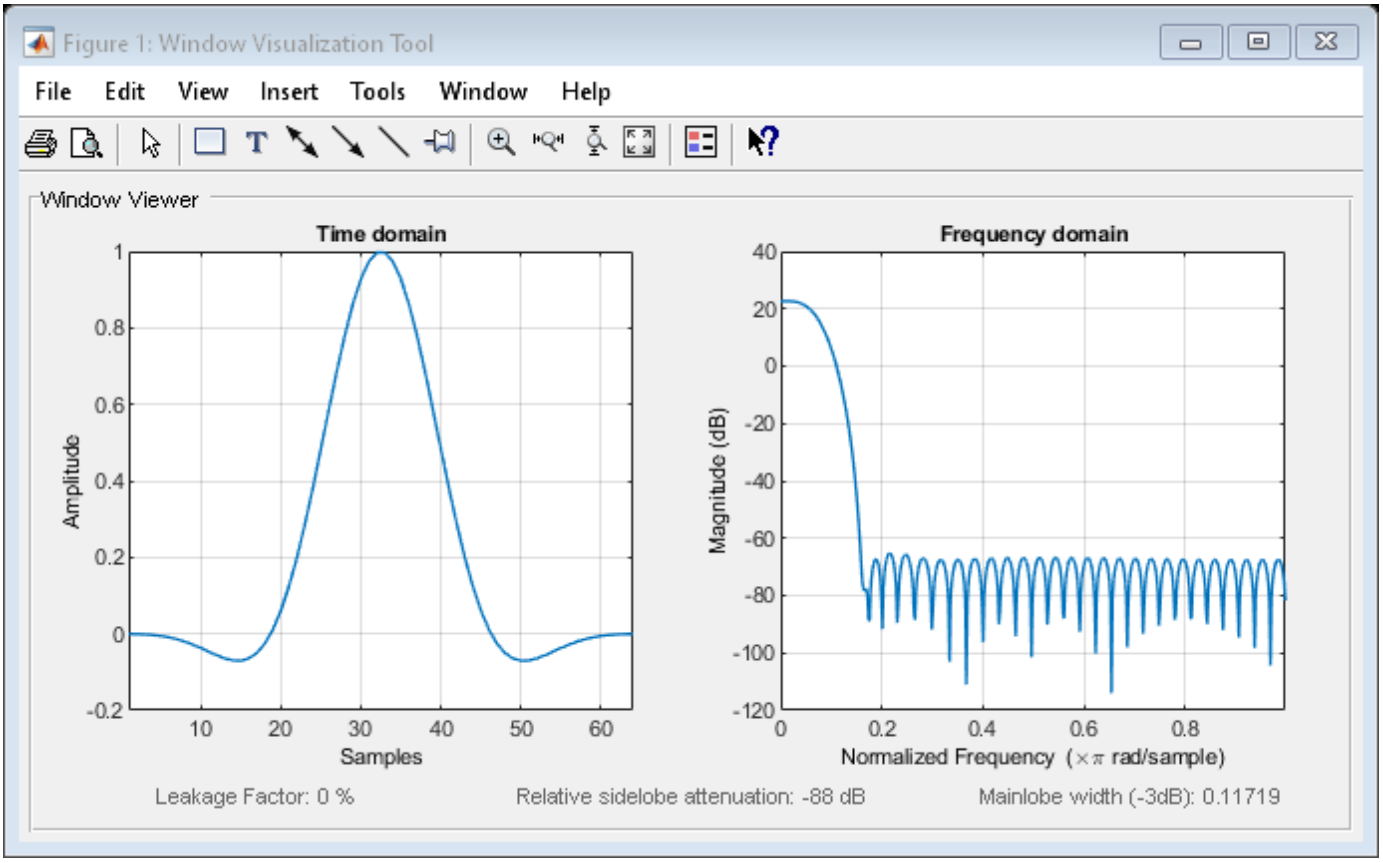
`w = flattopwin(L,sflag)` returns an `L`-point symmetric flat top window using the window sampling method specified by `sflag`.

Examples

Flat Top Window

Create a 64-point symmetric flat top window. View the result using `wvtool`.

```
N = 64;
w = flattopwin(N);
wvtool(w)
```



Input Arguments

L — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

sflag — Window sampling

`'symmetric'` (default) | `'periodic'`

Window sampling method, specified as:

- `'symmetric'` — Use this option when using windows for filter design.
- `'periodic'` — This option is useful for spectral analysis because it enables a windowed signal to have the perfect periodic extension implicit in the discrete Fourier transform. When `'periodic'` is specified, the function computes a window of length `L + 1` and returns the first `L` points.

Output Arguments

w — Flat top window

column vector

Flat top window, returned as a column vector.

Algorithms

Flat top windows are summations of cosines. The coefficients of a flat top window are computed from the following equation:

$$w(n) = a_0 - a_1\cos\left(\frac{2\pi n}{L-1}\right) + a_2\cos\left(\frac{4\pi n}{L-1}\right) - a_3\cos\left(\frac{6\pi n}{L-1}\right) + a_4\cos\left(\frac{8\pi n}{L-1}\right),$$

where $0 \le n \le L-1$. The coefficient values are:

Coefficient	Value
a_0	0.21557895
a_1	0.41663158
a_2	0.277263158
a_3	0.083578947
a_4	0.006947368

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.

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*. Vol. x, Number 3, 1987, pp. 1–28.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps

Window Designer

Functions

`blackman` | `hamming` | `hann` | **WVTool**

Introduced before R2006a

folders2labels

Get list of labels from folder names

Syntax

```
lbls = folders2labels(loc)
lbls = folders2labels(loc,Name,Value)

lbls = folders2labels(ds)

[lbls,files] = folders2labels( ___ )
```

Description

Use this function when you are working on a machine or deep learning classification problem and your labeled data is stored in folders that have the corresponding label names.

`lbls = folders2labels(loc)` creates a list of labels based on the folder names specified by the location `loc`.

`lbls = folders2labels(loc,Name,Value)` specifies additional input arguments using name-value pairs. For example, `'FileExtensions','.mat'` includes only .mat files in the scan for labels.

`lbls = folders2labels(ds)` creates a list of labels based on the files contained in `ds`. `ds` can be a datastore, a `matlab.io.datastore.FileSet` object, or a `matlab.io.datastore.BlockedFileSet` object.

`[lbls,files] = folders2labels( ___ )` additionally returns a list of files. The `i`th element of `lbls` corresponds to the label of the `i`th file in `files`.

Examples

Labels from Folder Names

Create a folder called `Files` in the current folder containing three subfolders, `Files_1`, `Files_2`, and `Files_3`. Add to each subfolder a random number of files, each containing a random signal of random size.

```
mkdir Files

for kj = 1:3
    fname = "Files_" + kj;
    mkdir(fname)
    for jk = 1:randi(4)
        sname = "sig_" + kj + "_" + jk;
        sgn = randn(randi([30 50]),randi(2));
        save(sname,"sgn")
        movefile(sname + ".mat",fname)
    end
```

```
    movefile(fname,"Files")
end
```

List the contents of the folders.

```
dir("*/*/*")
```

```
Files Found in: Files\Files_1

.            sig_1_1.mat  sig_1_3.mat
..           sig_1_2.mat  sig_1_4.mat

Files Found in: Files\Files_2

.            ..           sig_2_1.mat  sig_2_2.mat

Files Found in: Files\Files_3

.            ..           sig_3_1.mat  sig_3_2.mat  sig_3_3.mat
```

Create a list of labels based on the folder names.

```
lbls = folders2labels("Files")
```

```
lbls = 9x1 categorical
     Files_1
     Files_1
     Files_1
     Files_1
     Files_2
     Files_2
     Files_3
     Files_3
     Files_3
```

List the names of the files associated with the labels.

```
[~,files] = folders2labels("Files");
[~,fnames] = fileparts(files)
```

```
fnames = 9x1 string
    "sig_1_1"
    "sig_1_2"
    "sig_1_3"
    "sig_1_4"
    "sig_2_1"
    "sig_2_2"
    "sig_3_1"
    "sig_3_2"
    "sig_3_3"
```

Remove the `Files` directory you created at the beginning of the example.

```
rmdir Files s
```

Input Arguments

`loc` — Files or folders to scan for labels

character vector | cell array of character vectors | string scalar | string array

Files or folders to scan for labels, specified as a character vector, a cell array of character vectors, a string scalar, or a string array, containing the location of files or folders that are local or remote.

- Local files or folders — Specify `loc` as a local path to files or folders. If the files are not in the current folder, then the local path must specify full or relative paths. Files within subfolders of the specified folder are included by default. You can use the wildcard character (*) when specifying the local path. This character specifies that the file search include all matching files or all files in the matching folders.
- A remote location specified using an internationalized resource identifier (IRI).
- Remote files or folders — Specify `loc` to be the full paths of the files or folders as a uniform resource locator (URL) of the form `hdfs:///path_to_file`. For more information, see “Work with Remote Data”.

`folders2labels` looks for all file formats. To specify a custom list of file extensions to scan, use the `FileExtensions` argument.

Example: `'whale.mat'`

Example: `'../dir/data/signal.mat'`

Example: `"../dir/data/"`

Example: `{'dataFiles/Files_1/' 'dataFiles/Files_2/'}`

Example: `["dataFiles/Files_1/" "dataFiles/Files_2/"]`

Data Types: `char` | `string` | `cell`

`ds` — Data repository

datastore | `matlab.io.datastore.FileSet` object | `matlab.io.datastore.BlockedFileSet` object

Data repository, specified as a datastore, a `matlab.io.datastore.FileSet` object, or a `matlab.io.datastore.BlockedFileSet` object.

- If `ds` is a datastore, it must contain a `Files` property from which label names are parsed.
- If `ds` is a `matlab.io.datastore.FileSet` object, `folders2labels` obtains the label names from the file names listed in the `FileInfo` property of `ds`.
- If `ds` is a `matlab.io.datastore.BlockedFileSet` object, `folders2labels` obtains the label names from the file names listed in the `BlockInfo` property of `ds`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `folders2labels('C:\dir\signaldata','FileExtensions','.csv')` specifies a local path and includes only CSV files in the scan for labels.

`IncludeSubfolders` — Subfolder inclusion flag
`false` or `0` (default) | `true` or `1`

Subfolder inclusion flag, specified as `true` or `false`. Specify `true` to include all files and subfolders within each folder or `false` to include only the files within each folder.

Example: `'IncludeSubfolders',true`

Data Types: `logical` | `double`

`FileExtensions` — Signal file extensions
character vector | cell array of character vectors | string scalar | string array

Signal file extensions, specified as a string scalar, string array, character vector, or cell array of character vectors.

Example: `'FileExtensions','.csv'`

Data Types: `string` | `char` | `cell`

Output Arguments

`lbls` — List of labels
categorical vector

List of labels, returned as a categorical vector.

`files` — List of files
string vector

List of files, returned as a string vector. The `i`th element of `lbls` corresponds to a label for the `i`th file in `files`.

See Also

Signal Labeler | `countlabels` | `splitlabels`

Introduced in R2021a

freqs

Frequency response of analog filters

Syntax

```
h = freqs(b,a,w)
[h,wout] = freqs(b,a,n)
freqs( ___ )
```

Description

`h = freqs(b,a,w)` returns the complex frequency response of the analog filter specified by the coefficient vectors `b` and `a`, evaluated at the angular frequencies `w`.

`[h,wout] = freqs(b,a,n)` uses `n` frequency points to compute `h` and returns the corresponding angular frequencies in `wout`.

`freqs( ___ )` with no output arguments plots the magnitude and phase responses as functions of angular frequency in the current figure window. You can use this syntax with either of the previous input syntaxes.

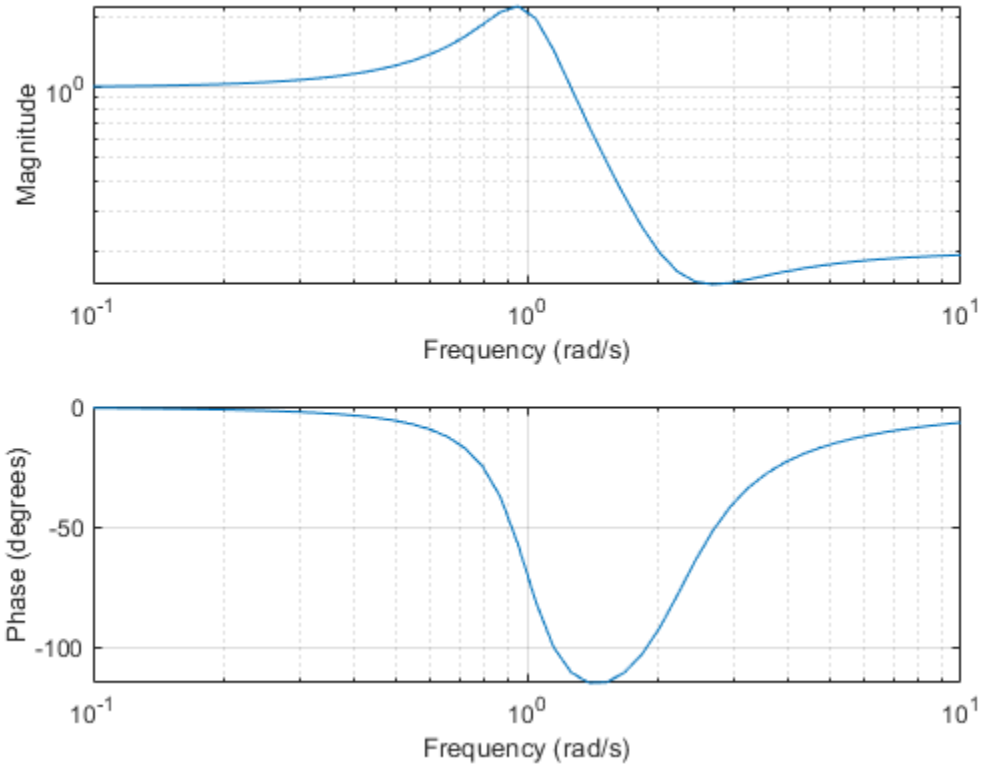
Examples

Frequency Response from Transfer Function

Find and graph the frequency response of the transfer function

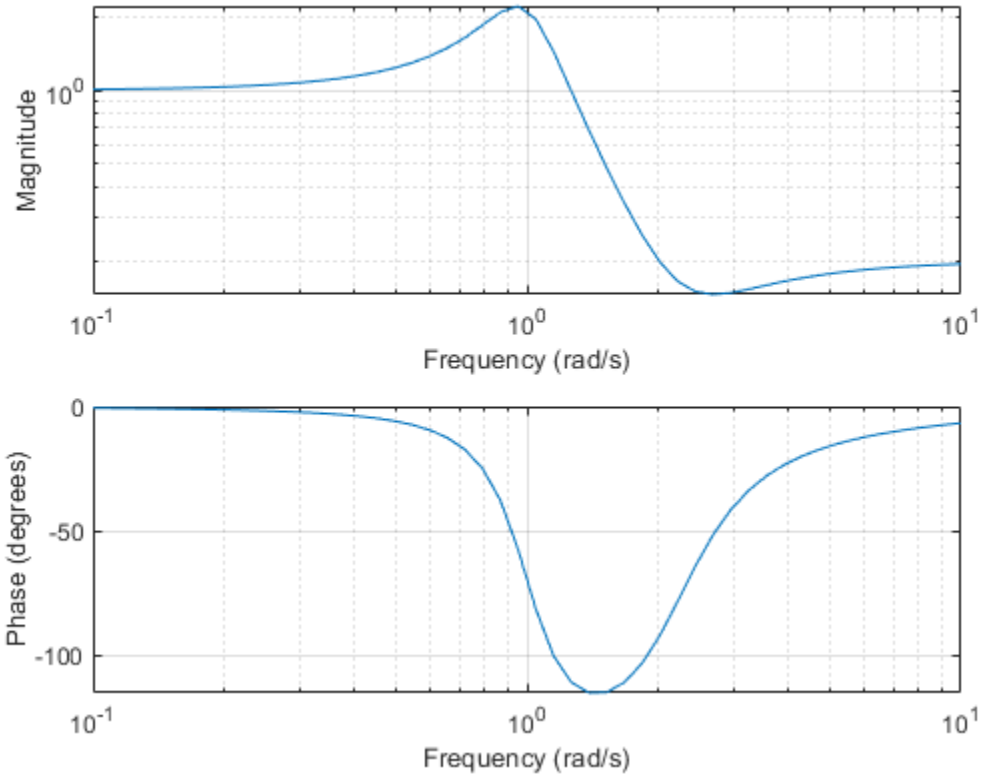
$$H(s) = \frac{0.2s^2 + 0.3s + 1}{s^2 + 0.4s + 1}.$$

```
a = [1 0.4 1];
b = [0.2 0.3 1];
w = logspace(-1,1);

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)')
```



You can also generate the plots by calling `freqs` with no output arguments.

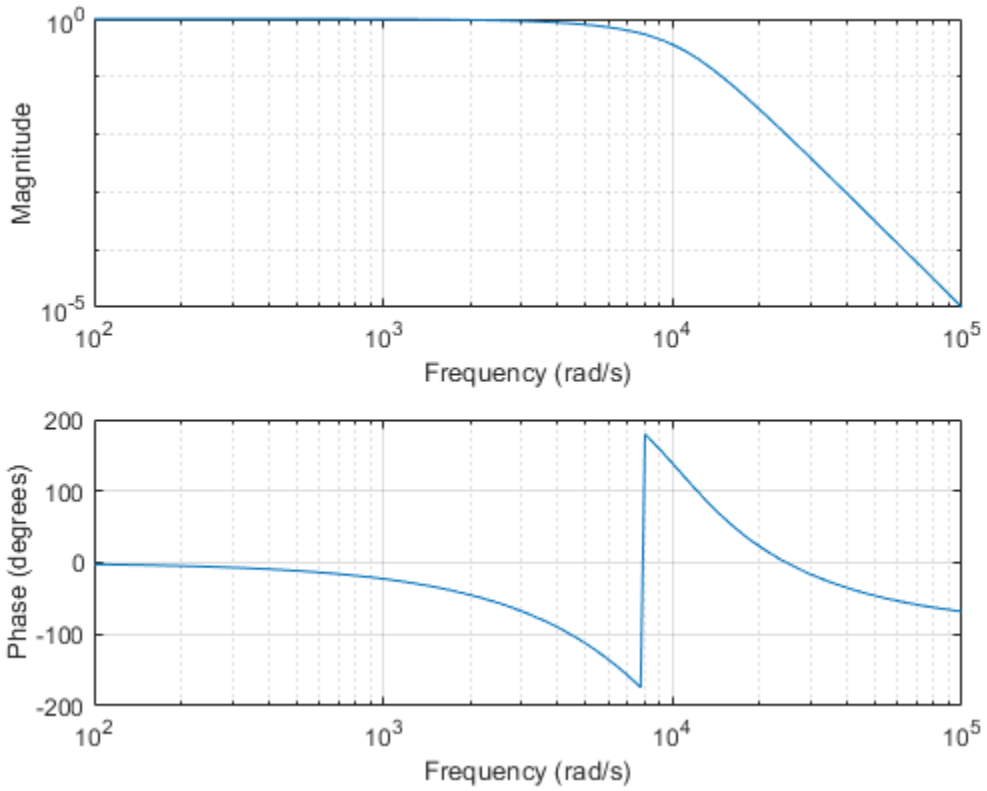
```
figure
freqs(b,a,w)
```



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
```



Input Arguments

b, a — Transfer function coefficients

vectors

Transfer function coefficients, specified as vectors.

Example: `[b,a] = butter(5,50,'s')` specifies a fifth-order Butterworth filter with a cutoff frequency of 50 rad/second.

Data Types: `single` | `double`

w — Angular frequencies

positive real vector

Angular frequencies, specified as a positive real vector expressed in rad/second.

Example: `2*pi*logspace(6,9)` specifies 50 logarithmically spaced angular frequencies from 1 MHz ($2\pi \times 10^6$ rad/second) and 1 GHz ($2\pi \times 10^9$ rad/second).

Data Types: `single` | `double`

n — Number of evaluation points

200 (default) | positive integer scalar

Number of evaluation points, specified as a positive integer scalar.

Data Types: `single` | `double`

Output Arguments

`h` — Frequency response

vector

Frequency response, returned as a vector.

`wout` — Angular frequencies

vector

Angular frequencies at which `h` is computed, returned as a vector.

Algorithms

`freqs` returns the complex frequency response of an analog filter specified by `b` and `a`. The function evaluates the ratio of Laplace transform polynomials

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

along the imaginary axis at the frequency points $s = j\omega$:

```
s = 1j*w;
h = polyval(b,s)./polyval(a,s);
```

See Also

`abs` | `angle` | `freqz` | `invfreqs` | `logspace` | `polyval`

Introduced before R2006a

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	Description of Resulting Filter
`dffir`	Direct-form FIR filter
`dffirt`	Transposed direct-form FIR filter
`dfsymfir`	Symmetrical direct-form FIR filter
`dfasymfir`	Asymmetrical direct-form FIR filter
`fftfir`	Fast Fourier transform FIR filter

`hd = design(...,'Window',window)` designs filters using the window specified in `window`. Provide the input argument `window` as

- The window name in single quotes. For example, use `'bartlett'`, or `'hamming'`. See `window` for the full list of available windows.
- 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

Filters with Arbitrary Magnitude Response

Design FIR filters that have arbitrary magnitude responses.

Initially, generate a real filter whose response has three distinct sections:

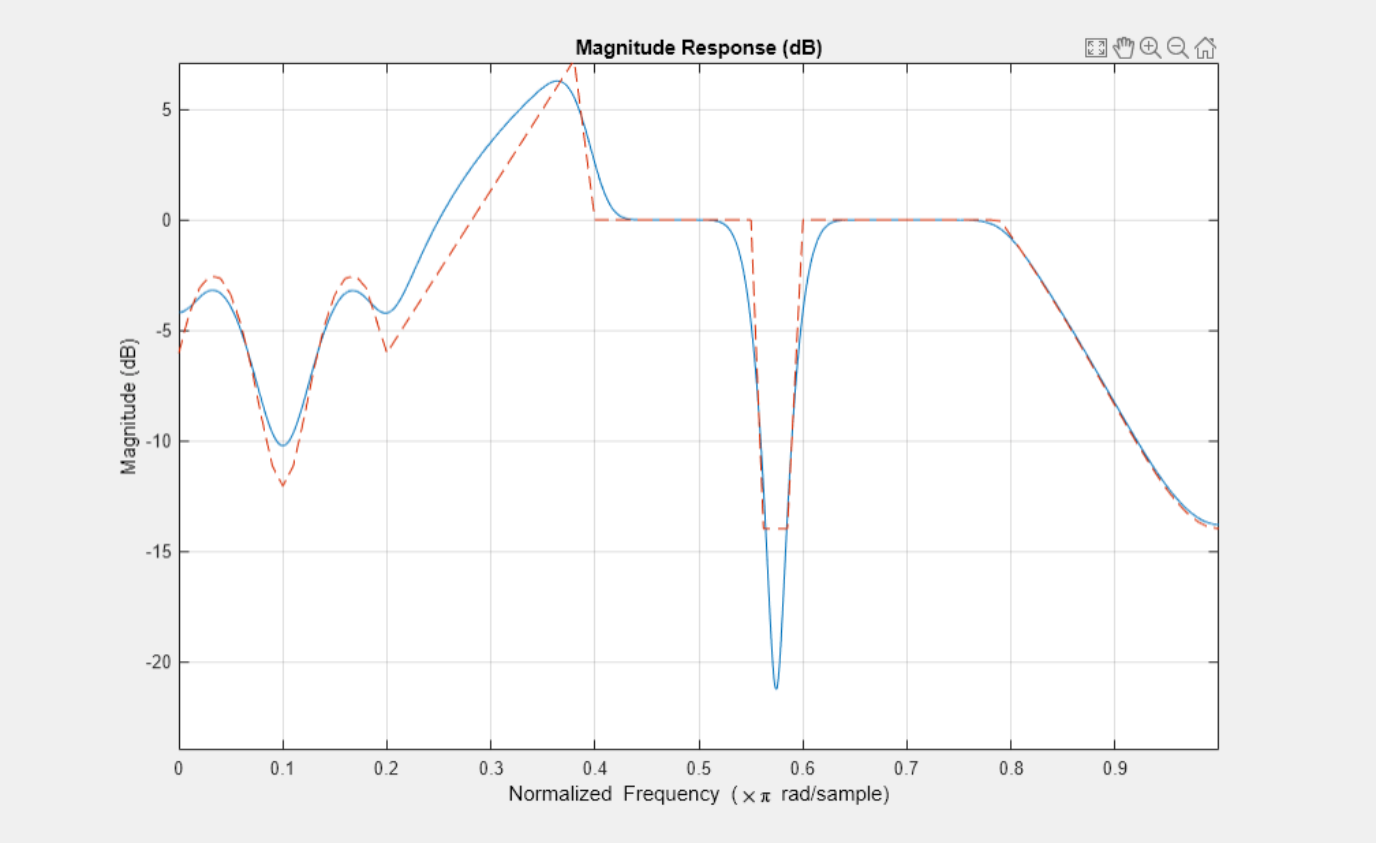
- A sinusoidal response section for lower frequencies
- A piecewise linear response section for intermediate frequencies

- A quadratic response section for higher frequencies.

```
b1 = 0:0.01:0.18;
a1 = 0.5+sin(2*pi*7.5*b1)/4;

b2 = [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];

b3 = [0.79:0.01:1];
a3 = 0.2+18*(1-b3).^2;

f = [b1 b2 b3];
a = [a1 a2 a3];
n = 300;
```

Design the filter using a Kaiser window with $\beta = 50$;

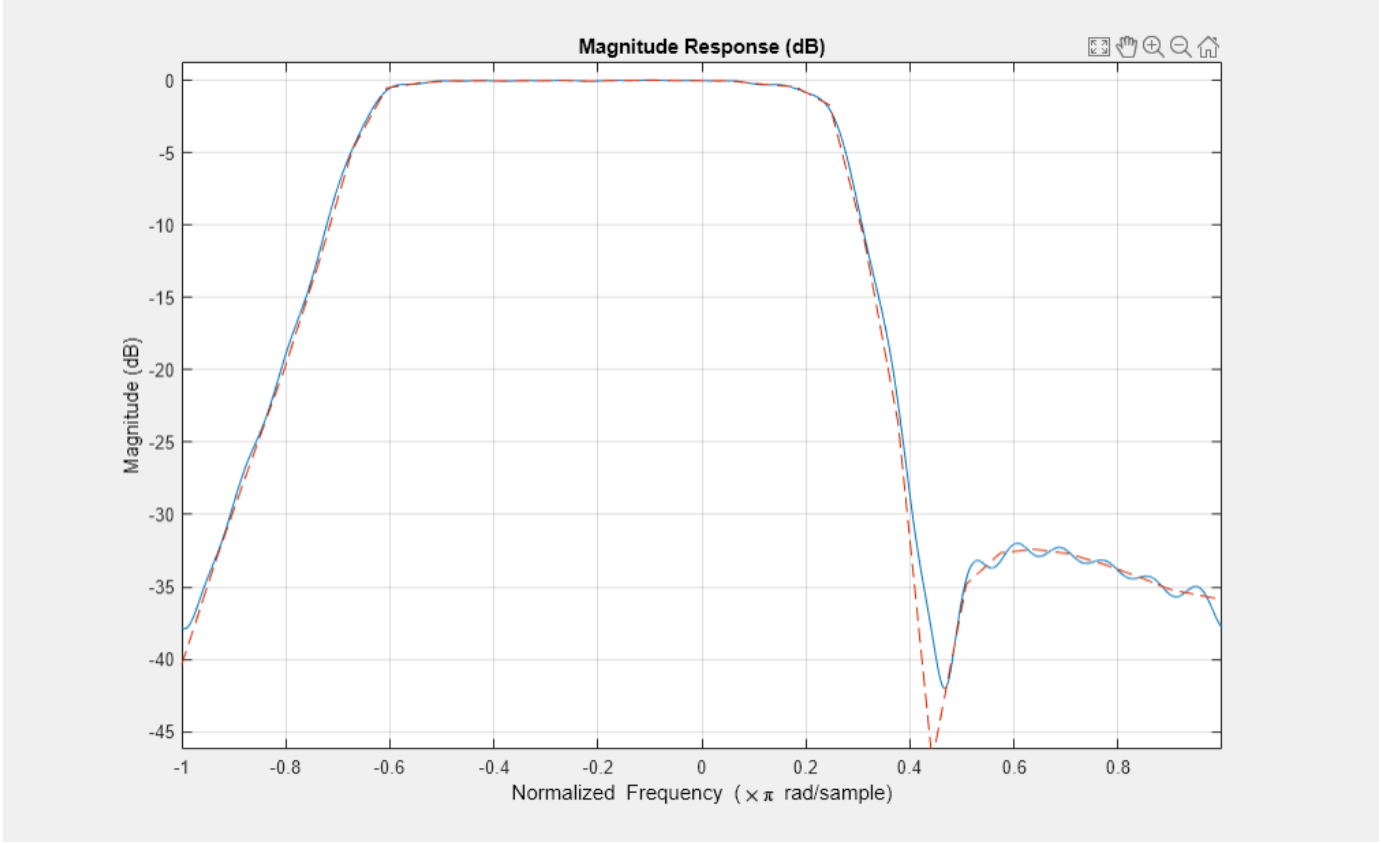
```
d = fdesign.arbmag('n,f,a',n,f,a);
hd = design(d,'freqsamp','Window',{@kaiser,50});
fvtool(hd)
```



Design an arbitrary-magnitude complex FIR filter. The vector `f` contains frequency locations. The vector `a` contains the desired filter response values at the locations specified in `f`. Use a rectangular window for the design.

```
f = [-1 -0.93443 -0.86885 -0.80328 -0.7377 -0.67213 -0.60656 -0.54098 ...
    -0.47541 -0.40984 -0.34426 -0.27869 -0.21311 -0.14754 -0.081967 ...
    -0.016393 0.04918 0.11475 0.18033 0.2459 0.31148 0.37705 0.44262 ...
```

```
    0.5082 0.57377 0.63934 0.70492 0.77049 0.83607 0.90164 1];

a = [0.0095848 0.021972 0.047249 0.099869 0.23119 0.57569 0.94032 ...
    0.98084 0.99707 0.99565 0.9958 0.99899 0.99402 0.99978 ...
    0.99995 0.99733 0.99731 0.96979 0.94936 0.8196 0.28502 ...
    0.065469 0.0044517 0.018164 0.023305 0.02397 0.023141 0.021341 ...
    0.019364 0.017379 0.016061];
n = 48;

d = fdesign.arbmag('n,f,a',n,f,a);
hdc = design(d,'freqsamp','Window','rectwin');
fvtool(hdc)
```



FVTool shows 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` returns a complex-valued filter.

See Also

Apps
Filter Designer

Functions
`designfilt` | `fdesign`

Introduced in R2009a

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 transfer function coefficients stored in `b` and `a`.

`[h,w] = freqz(sos,n)` returns the `n`-point complex frequency response corresponding to the second-order sections matrix `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 a digital filter designed to filter signals sampled at a rate `fs`.

`[h,f] = freqz(___,n,'whole',fs)` returns the frequency vector at `n` points ranging between 0 and `fs`.

`h = freqz(___,w)` returns the frequency response vector `h` evaluated at the normalized frequencies supplied in `w`.

`h = freqz(___,f,fs)` returns the frequency response vector `h` evaluated at the physical frequencies supplied in `f`.

`freqz(___)` with no output arguments plots the frequency response of the filter.

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:

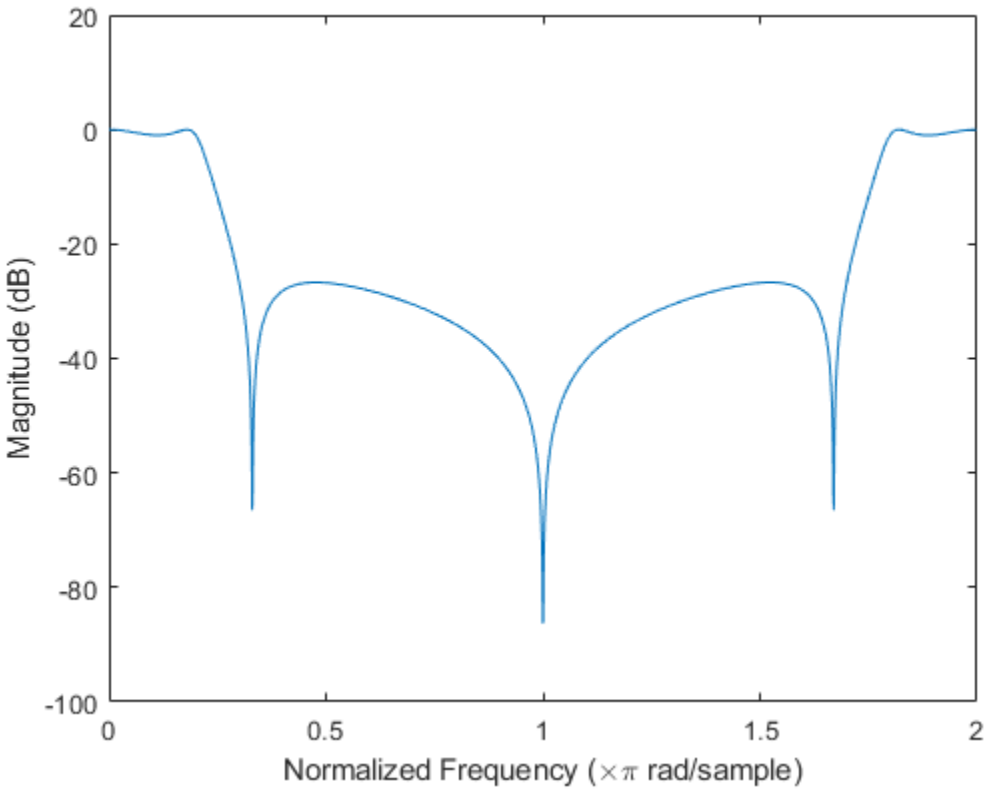
$$H(z) = \frac{0.05634(1+z^{-1})(1-1.0166z^{-1}+z^{-2})}{(1-0.683z^{-1})(1-1.4461z^{-1}+0.7957z^{-2})}.$$

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:

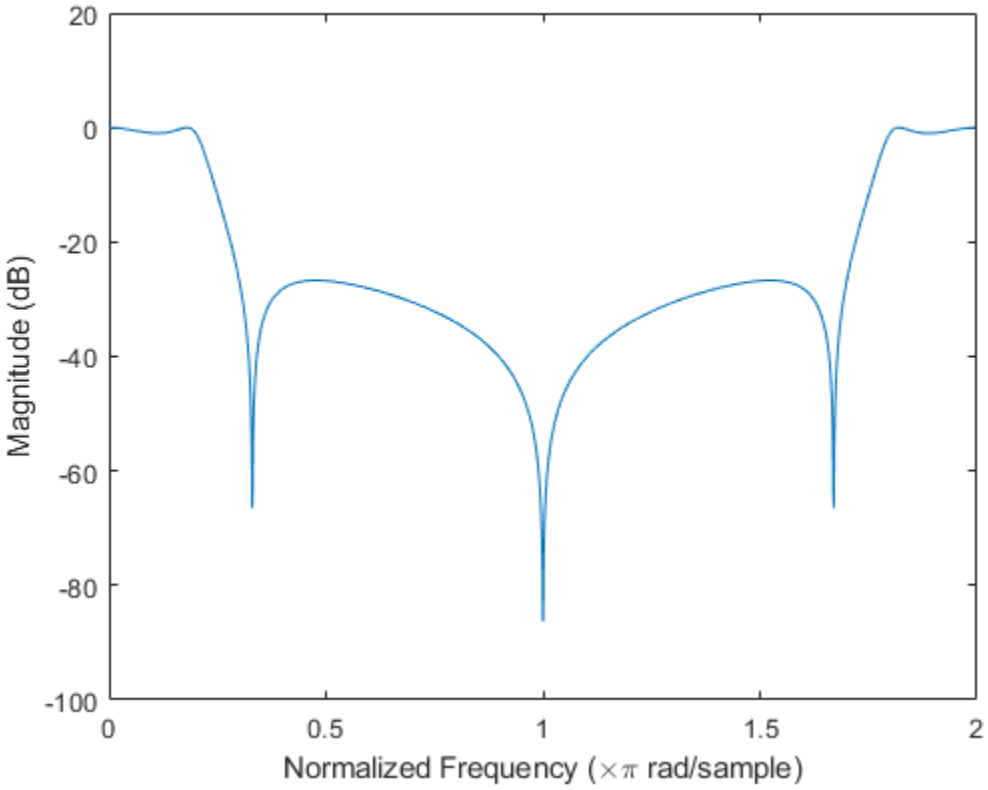
$$H(z) = \frac{0.05634(1+z^{-1})(1-1.0166z^{-1}+z^{-2})}{(1-0.683z^{-1})(1-1.4461z^{-1}+0.7957z^{-2})}.$$

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.

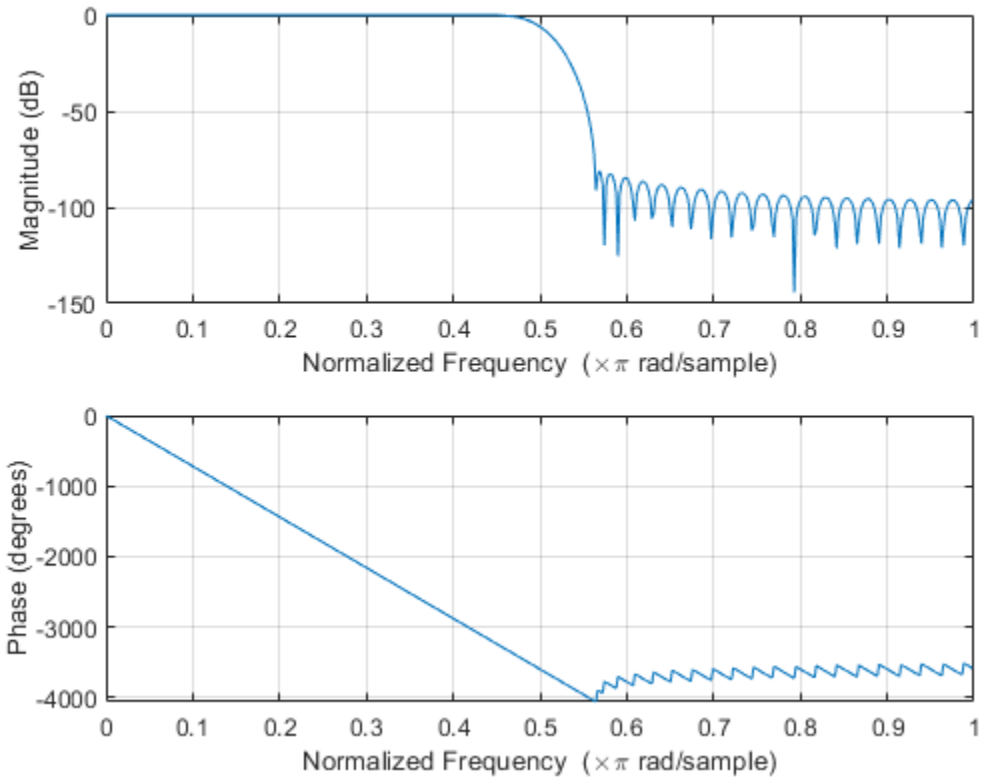
```
plot(w/pi,20*log10(abs(h)))
ax = gca;
ax.YLim = [-100 20];
ax.XTick = 0:.5:2;
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Magnitude (dB)')
```



Frequency Response of an FIR filter

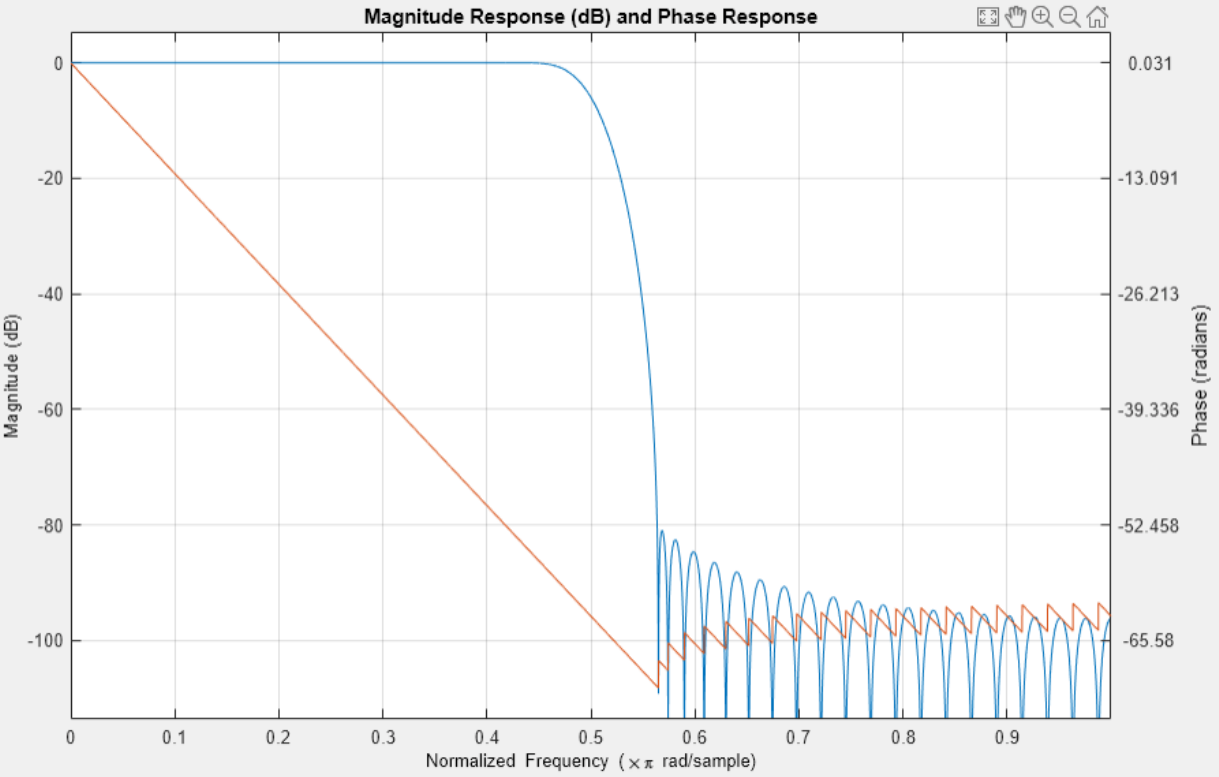
Design an FIR lowpass filter of order 80 using a Kaiser window with $\beta = 8$. Specify a normalized cutoff frequency of 0.5π rad/sample. Display the magnitude and phase responses of the filter.

```
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)
```

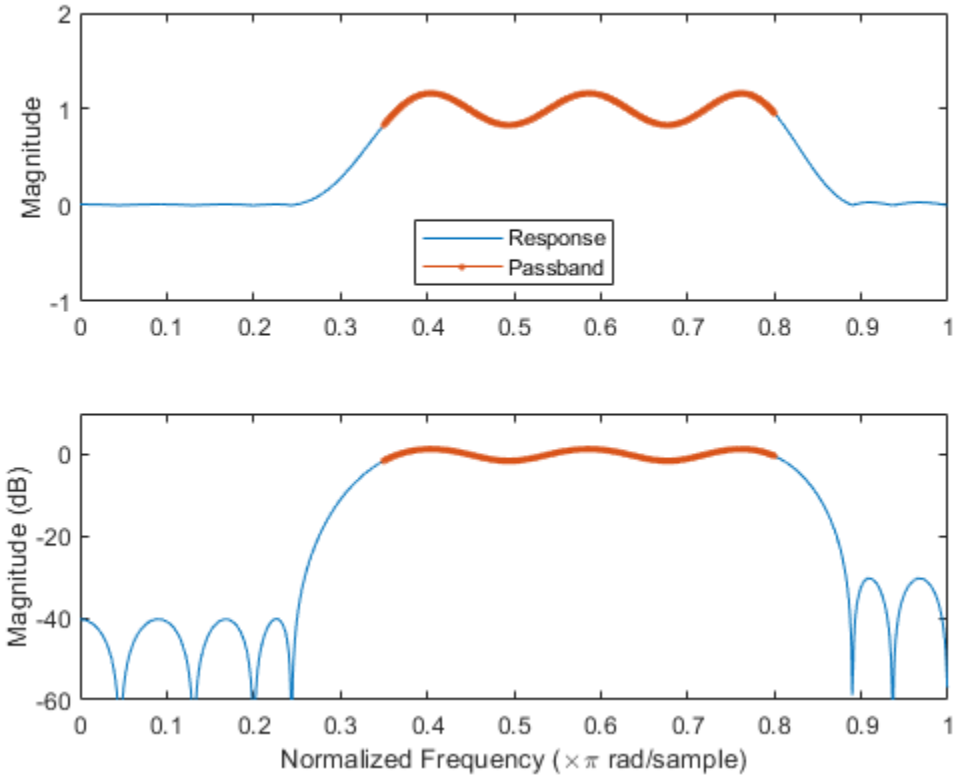


Frequency Response of an FIR Bandpass Filter

Design an FIR bandpass filter with passband between 0.35π and 0.8π rad/sample and 3 dB of ripple. The first stopband goes from 0 to 0.1π rad/sample and has an attenuation of 40 dB. The second stopband goes from 0.9π 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

$$H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})} = \frac{\mathrm{b(1)}+\mathrm{b(2)}\,e^{-j\omega}+\mathrm{b(3)}\,e^{-j2\omega}+\cdots+\mathrm{b(M)}\,e^{-j(M-1)\omega}}{\mathrm{a(1)}+\mathrm{a(2)}\,e^{-j\omega}+\mathrm{a(3)}\,e^{-j2\omega}+\cdots+\mathrm{a(N)}\,e^{-j(N-1)\omega}}.$$

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 0.5π 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.

`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, the function treats the input as 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 0.5π 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 0.5π rad/sample.

`fs` — Sample rate

positive scalar

Sample 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 rad/sample. `w` must have at least two elements, because otherwise the function interprets it as `n`. `w` = *π* corresponds to the Nyquist frequency.

`f` — Frequencies

vector

Frequencies, specified as a vector. `f` must have at least two elements, because otherwise the function interprets it as `n`. 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`, then `h` has length `n`. If you do not specify `n`, or specify `n` as the empty vector, then `h` has length 512.

If the input to `freqz` is single precision, the function computes the frequency response using single-precision arithmetic. The output `h` is single precision.

w — Angular frequencies

vector

Angular frequencies, returned as a vector. `w` has values ranging from 0 to π. If you specify `'whole'` in your input, the values in `w` range from 0 to 2π. 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 `fs`/2 Hz. If you specify `'whole'` in your input, the values in `f` range from 0 to `fs` 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.

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 do not 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 input, `freqz` evaluates the polynomials at each frequency point and divides the numerator response by the denominator response. To evaluate the polynomials, the function uses Horner's method.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the first input to `freqz` is a variable-size matrix at compile time, then it must not become a vector or an empty array at runtime.
- If the input `n` is variable-size at compile time, then it must not become a scalar or an empty array at runtime.

See Also

`abs` | `angle` | `designfilt` | `digitalFilter` | `fft` | `filter` | `freqs` | `impz` | `invfreqs` | `logspace`

Introduced before R2006a

fsst

Fourier synchrosqueezed transform

Syntax

```
s = fsst(x)
[s,w,n] = fsst(x)

[s,f,t] = fsst(x,fs)
[s,f,t] = fsst(x,ts)

[___] = fsst(___,window)

fsst(___)
fsst(___,freqloc)
```

Description

`s = fsst(x)` returns the Fourier synchrosqueezed transform of the input signal, `x`. Each column of `s` contains the synchrosqueezed spectrum of a windowed segment of `x`.

`[s,w,n] = fsst(x)` returns a vector of normalized frequencies, `w`, and a vector of sample numbers, `n`, at which the Fourier synchrosqueezed transform is computed. `w` corresponds to the columns of `s` and `n` corresponds to the rows of `s`.

`[s,f,t] = fsst(x,fs)` returns a vector of cyclical frequencies, `f`, and a vector of time instants, `t`, expressed in terms of the sample rate, `fs`.

`[s,f,t] = fsst(x,ts)` specifies the sample time, `ts`, as a `duration` scalar. `t` is in the same units as `ts`. The units of `f` are reciprocal to the units of `ts`.

`[___] = fsst(___,window)` uses `window` to divide the signal into segments and perform windowing. You can use any combination of input arguments from previous syntaxes to obtain the corresponding output arguments.

`fsst(___)` with no output arguments plots the synchrosqueezed transform in the current figure window.

`fsst(___,freqloc)` specifies the axis on which to plot the frequency.

Examples

Fourier Synchrosqueezed Transform of Sinusoidal Signal

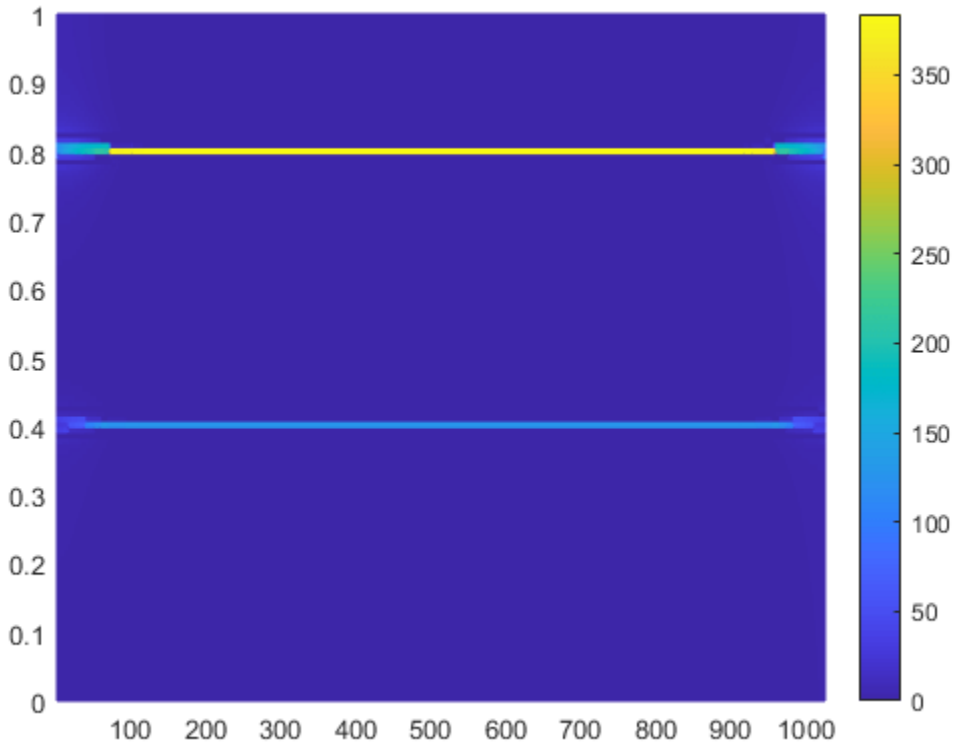
Generate 1024 samples of a signal that consists of a sum of sinusoids embedded in white Gaussian noise. The normalized frequencies of the sinusoids are $2\pi/5$ rad/sample and $4\pi/5$ rad/sample. The higher frequency sinusoid has 3 times the amplitude of the other sinusoid.

```
N = 1024;
n = 0:N-1;
```

```
w0 = 2*pi/5;
x = sin(w0*n)+3*sin(2*w0*n);
```

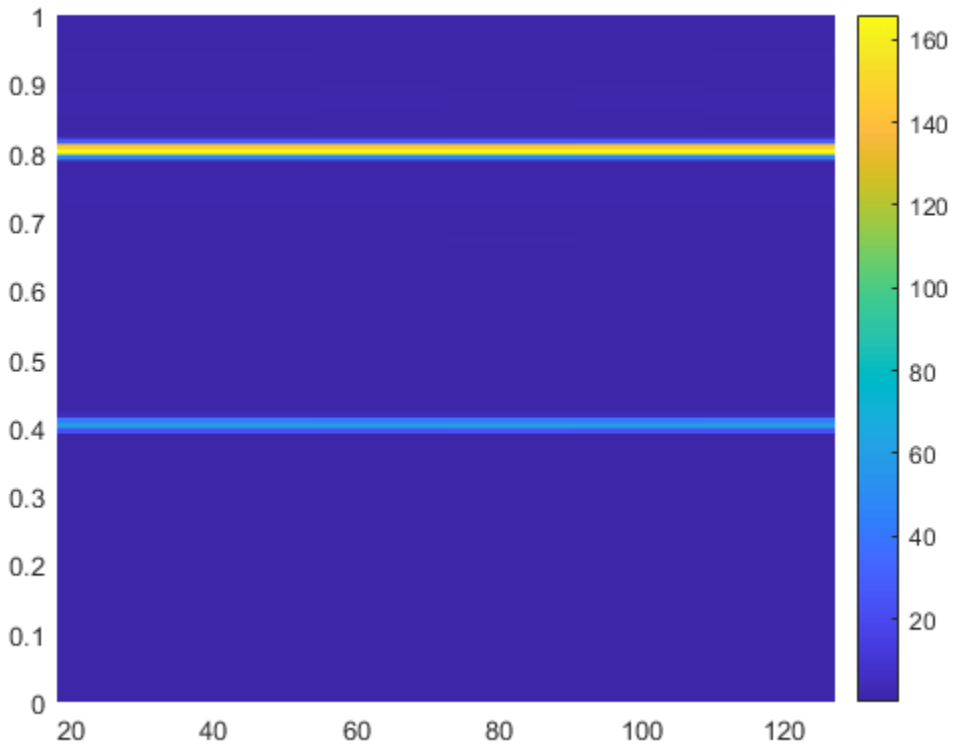
Compute the Fourier synchrosqueezed transform of the signal. Plot the result.

```
[s,w,n] = fsst(x);

mesh(n,w/pi,abs(s))

axis tight
view(2)
colorbar
```



Compute the conventional short-time Fourier transform of the signal for comparison. Use the default values of `spectrogram`. Plot the result.

```
[s,w,n] = spectrogram(x);

surf(n,w/pi,abs(s),'EdgeColor','none')

axis tight
view(2)
colorbar
```



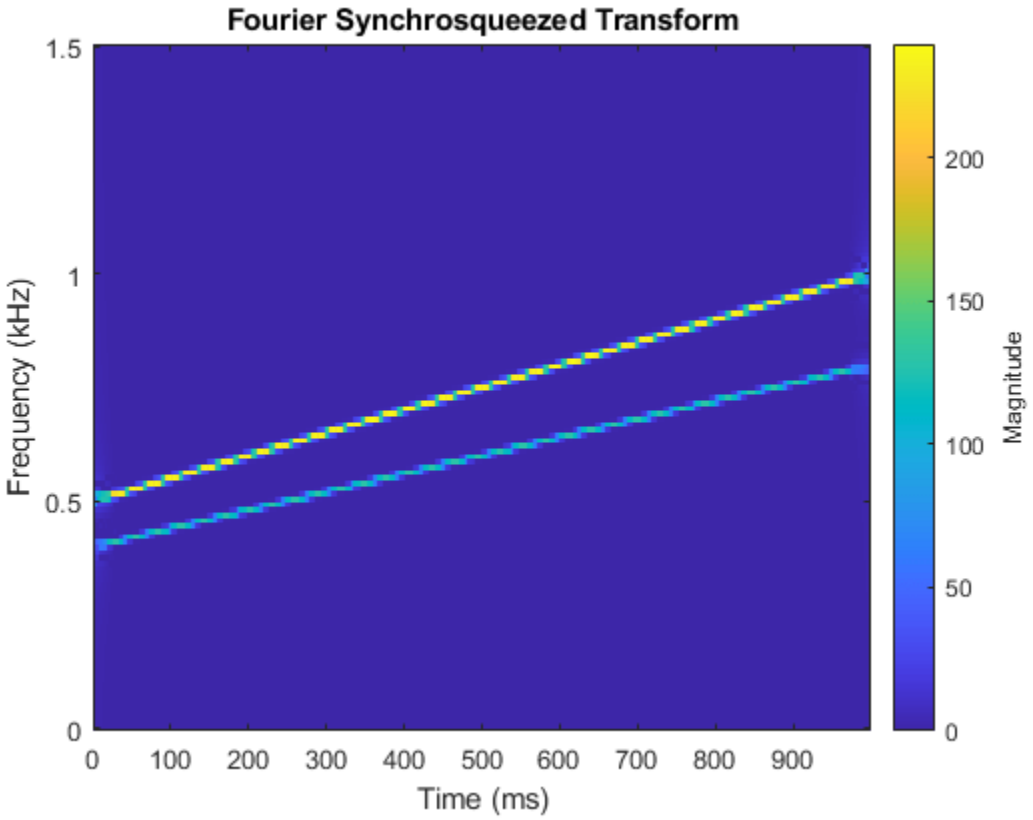
Synchrosqueezed Transform of Linear Chirps

Generate a signal that consists of two chirps. The signal is sampled at 3 kHz for one second. The first chirp has an initial frequency of 400 Hz and reaches 800 Hz at the end of the sampling. The second chirp starts at 500 Hz and reaches 1000 Hz at the end. The second chirp has twice the amplitude of the first chirp.

```
fs = 3000;
t = 0:1/fs:1-1/fs;

x1 = chirp(t,400,t(end),800);
x2 = 2*chirp(t,500,t(end),1000);
```

Compute and plot the Fourier synchrosqueezed transform of the signal.

```
fsst(x1+x2,fs,'yaxis')
```



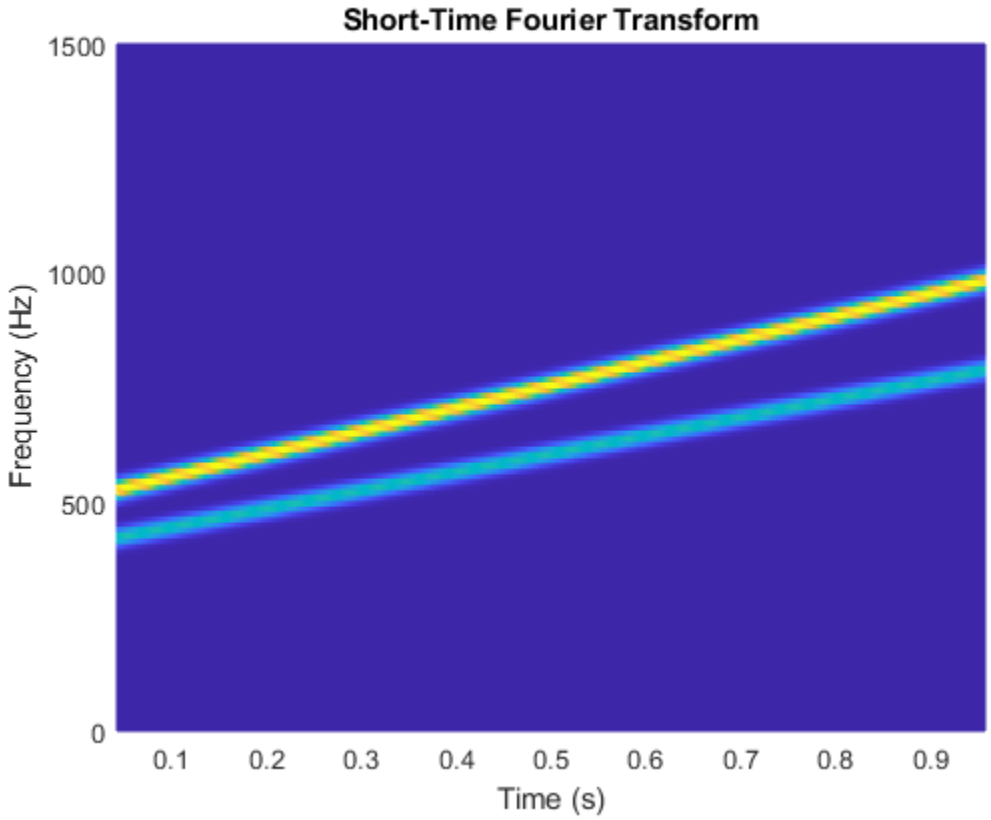
Compare the synchrosqueezed transform with the short-time Fourier transform (STFT). Compute the STFT using the `spectrogram` function. Specify the default parameters used by `fsst`:

- A 256-point Kaiser window with $\beta = 10$ to window the signal
- An overlap of 255 samples between adjoining windowed segments
- An FFT length of 256

```
[stft,f,t] = spectrogram(x1+x2,kaiser(256,10),255,256,fs);
```

Plot the absolute value of the STFT.

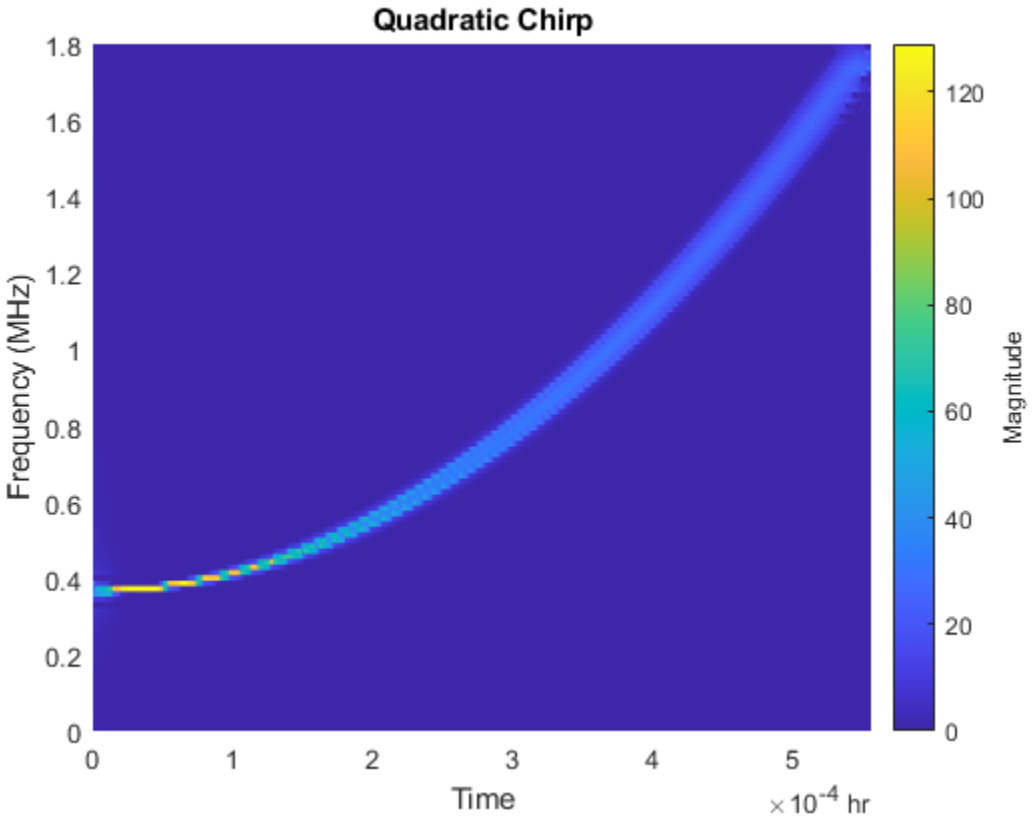
```
mesh(t,f,abs(stft))

xlabel('Time (s)')
ylabel('Frequency (Hz)')
title('Short-Time Fourier Transform')
axis tight
view(2)
```



Fourier Synchrosqueezed Transform of Chirps

Compute and display the Fourier synchrosqueezed transform of a quadratic chirp that starts at 100 Hz and crosses 200 Hz at `t` = 1 s. Specify a sample rate of 1 kHz. Express the sample time as a `duration` scalar.

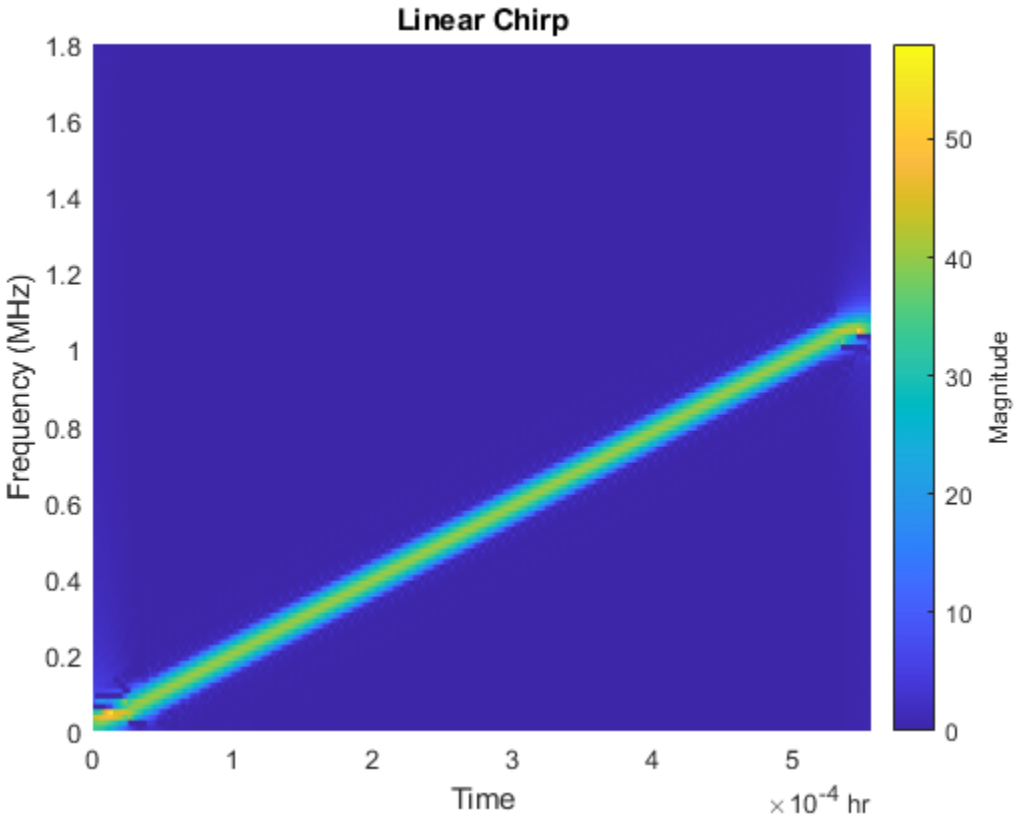
```
fs = 1000;
t = 0:1/fs:2;
ts = duration(0,0,1/fs);

x = chirp(t,100,1,200,'quadratic');

fsst(x,ts,'yaxis')

title('Quadratic Chirp')
```



The synchrosqueezing algorithm works under the assumption that the frequency of the signal varies slowly. Thus the spectrum is better concentrated at early times, where the rate of change of frequency is smaller.

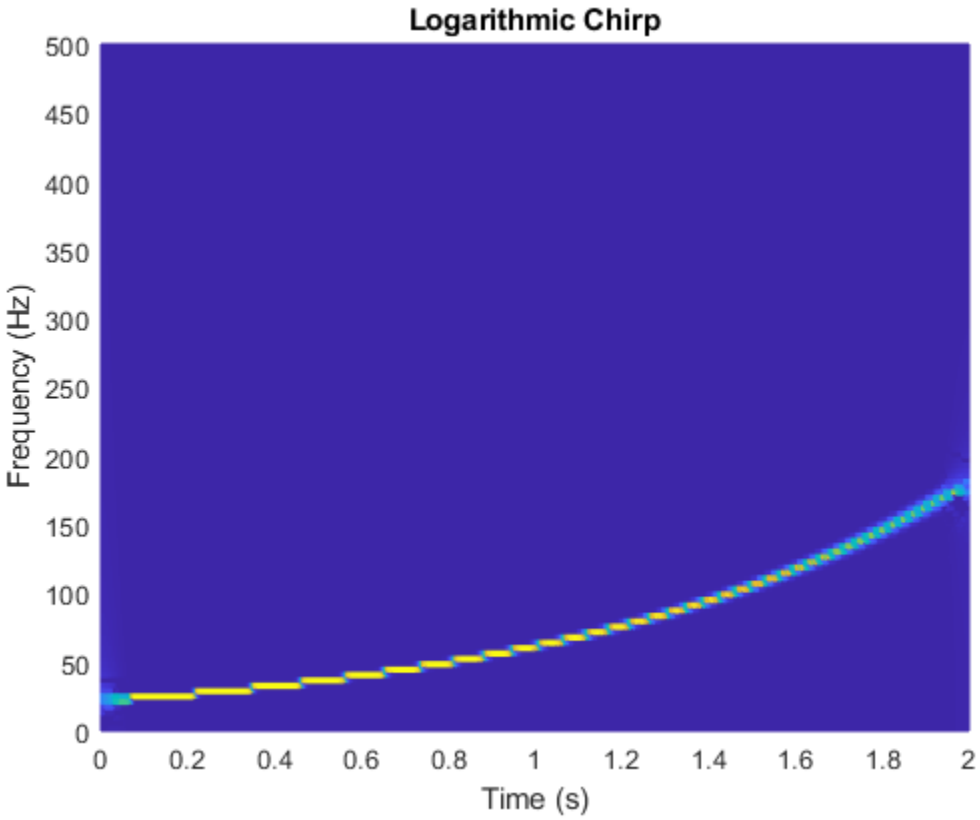
Compute and display the Fourier synchrosqueezed transform of a linear chirp that starts at DC and crosses 150 Hz at `t` = 1 s. Use a 256-sample Hamming window.

```
x = chirp(t,0,1,150);

fsst(x,ts,hamming(256),'yaxis')

title('Linear Chirp')
```



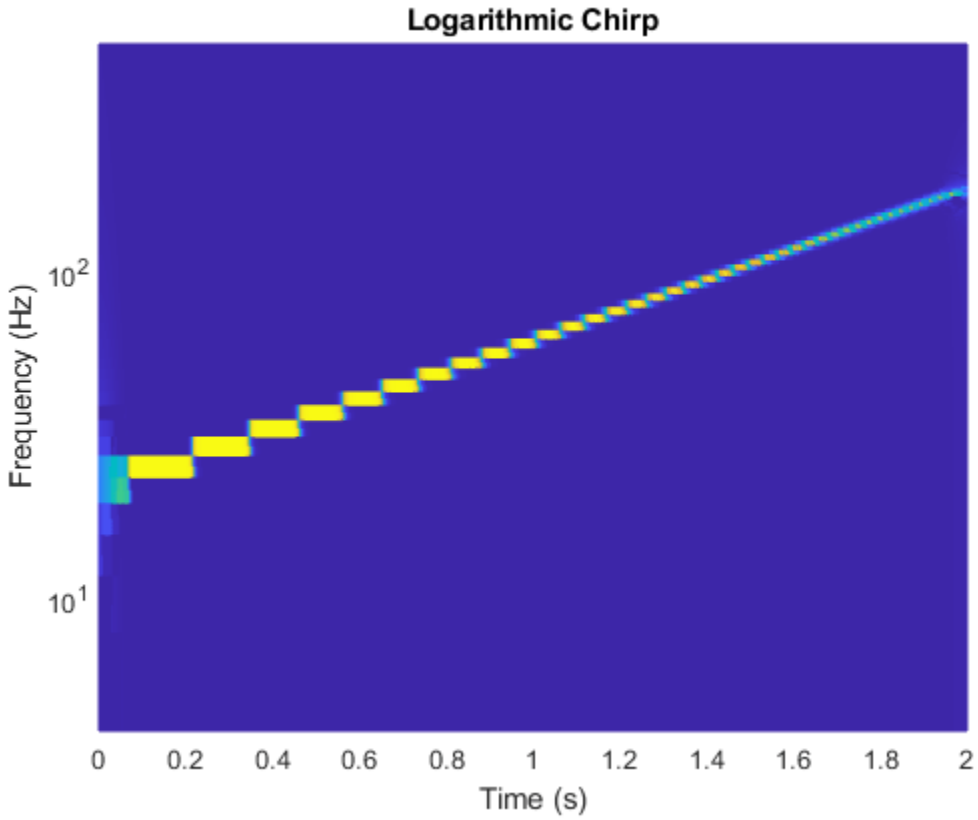
Compute and display the Fourier synchrosqueezed transform of a logarithmic chirp. The chirp is sampled at 1 kHz, starts at 20 Hz, and crosses 60 Hz at `t` = 1 s. Use a 256-sample Kaiser window with $\beta = 20$.

```
x = chirp(t,20,1,60,'logarithmic');

[s,f,t] = fsst(x,fs,kaiser(256,20));

clf
mesh(t,f,(abs(s)))

title('Logarithmic Chirp')
xlabel('Time (s)')
ylabel('Frequency (Hz)')
view(2)
```



Use a logarithmic scale for the frequency axis. The transform becomes a straight line.

```
ax = gca;
ax.YScale = 'log';
axis tight
```



Fourier Synchrosqueezed Transform of Speech Signal

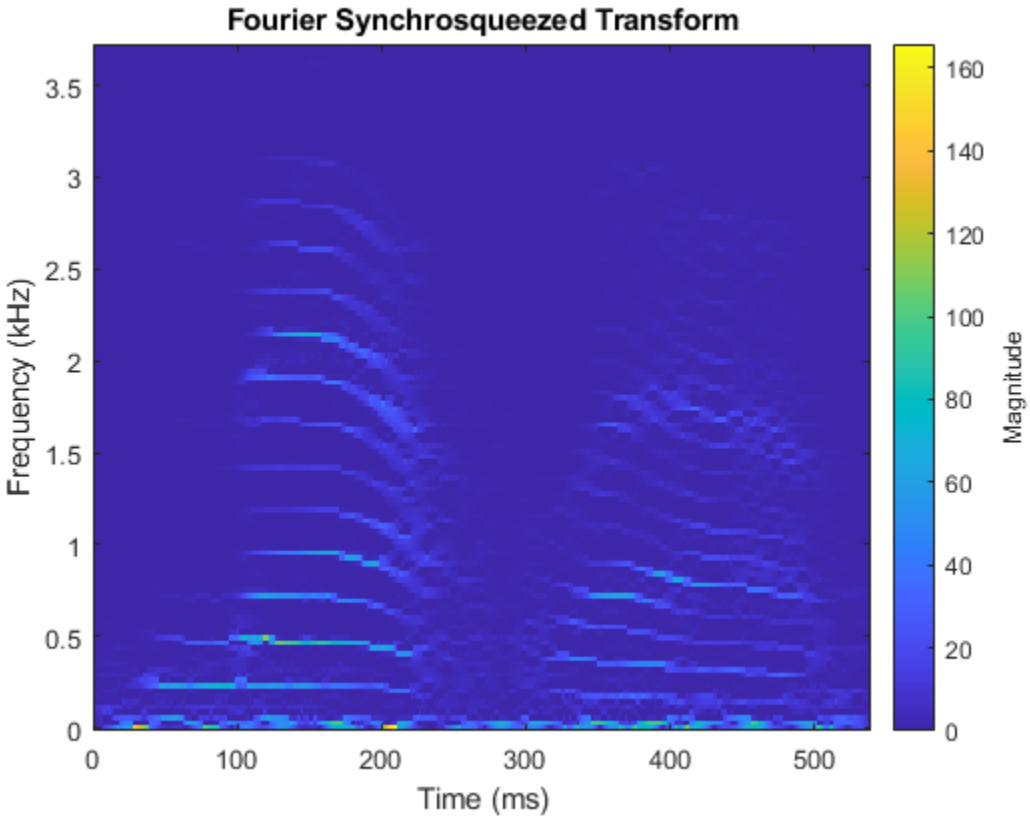
Load a speech signal sampled at F_s = 7418 Hz. The file contains a recording of a female voice saying the word "MATLAB®."

```
load mtlb
```

```
% To hear, type sound(mtlb,Fs)
```

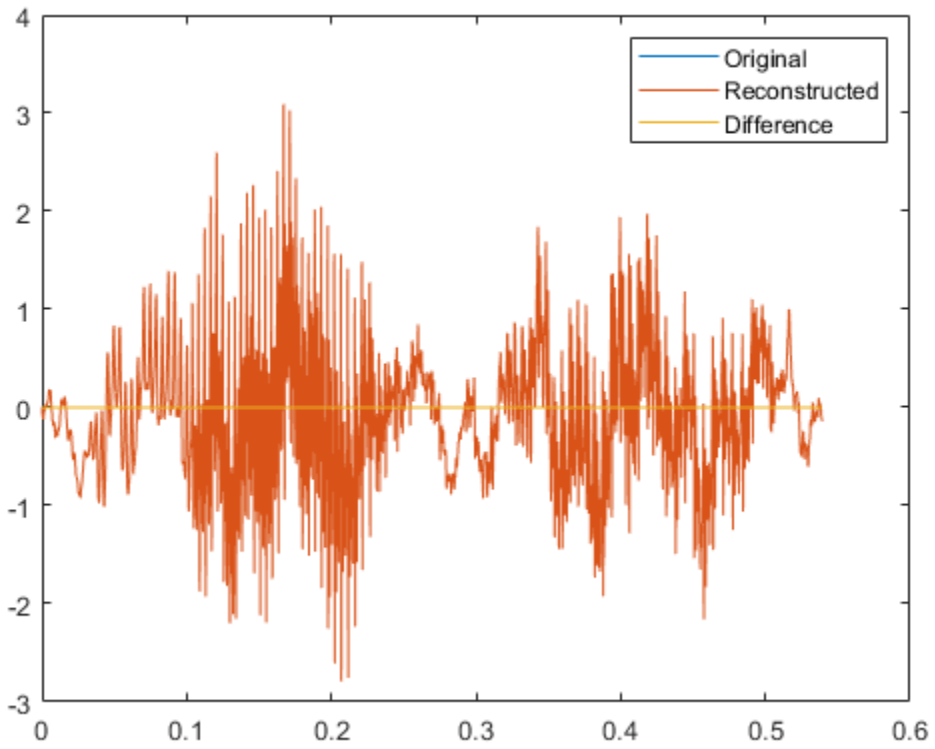
Compute the synchrosqueezed transform of the signal. Use a Hann window of length 256. Display the time on the *x*-axis and the frequency on the *y*-axis.

```
fsst(mtlb,Fs,hann(256),'yaxis')
```



Use `ifsst` to invert the transform. Compare the original and reconstructed signals.

```
sst = fsst(mtlb,Fs,hann(256));

xrc = ifsst(sst,hann(256));

plot((0:length(mtlb)-1)/Fs,[mtlb xrc xrc-mtlb])
legend('Original','Reconstructed','Difference')
```




```
% To hear, type sound(xrc-mtlb,Fs)
```

Input Arguments

x — Input signal

vector

Input signal, specified as a 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

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, then the sample rate is in Hz.

Data Types: `double` | `single`

ts — Sample time

`duration` scalar

Sample time, specified as a `duration` scalar. The sample time is the time elapsed between consecutive samples of `x`.

Data Types: `duration`

`window` — Window used to divide the signal into segments
`kaiser(256,10)` (default) | integer | vector | `[]`

Window used to divide the signal into segments, specified as an integer or as a row or column vector.

- If `window` is an integer, then `fsst` divides `x` into segments of length `window` and windows each segment with a Kaiser window of that length and $\beta = 10$. The overlap between adjacent segments is `window` – 1.
- If `window` is a vector, then `fsst` divides `x` into segments of the same length as the vector and windows each segment using `window`. The overlap between adjacent segments is `length(window)` – 1.
- If `window` is not specified, then `fsst` divides `x` into segments of length 256 and windows each segment with a 256-sample Kaiser window with $\beta = 10$. The overlap between adjacent segments is 255. If `x` has fewer than 256 samples, then the function uses a single Kaiser window with the same length as `x` and $\beta = 10$.

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: `double` | `single`

`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 `fsst` with output arguments.

Output Arguments

`s` — Fourier synchrosqueezed transform
matrix

Fourier synchrosqueezed transform, returned as a matrix. The number of columns of `s` equals the length of `x`. Time increases across the columns of `s` and frequency increases down the rows of `s`, starting from zero. If `x` is real, then its synchrosqueezed spectrum is one-sided. If `x` is complex, then its synchrosqueezed spectrum is two-sided and centered.

`w` — Normalized frequencies
vector

Normalized frequencies, returned as a vector. The length of `w` equals the number of rows in `s`.

`n` — Sample numbers
vector

Sample numbers, returned as a vector. The length of `n` equals the number of columns in `s`. Each sample number in `n` is the midpoint of a windowed segment of `x`.

`f` — Cyclical frequencies
vector

Cyclical frequencies, returned as a vector. The length of `f` equals the number of rows in `s`.

`t` — Time instants
vector

Time instants, returned as a vector. The length of `t` equals the number of columns in `s`. Each time value in `t` is the midpoint of a windowed segment of `x`.

More About

Fourier Synchrosqueezed Transform

Many real-world signals such as speech waveforms, machine vibrations, and physiologic signals can be expressed as a superposition of amplitude-modulated and frequency-modulated modes. For time-frequency analysis, it is convenient to express such signals as sums of analytic signals through

$$f(t) = \sum_{k=1}^{K} f_k(t) = \sum_{k=1}^{K} A_k(t) e^{j2\pi\phi_k(t)}.$$

The phases $\phi_k(t)$ have time derivatives $d\phi_k(t)/dt$ that correspond to instantaneous frequencies. When the exact phases are unknown, you can use the Fourier synchrosqueezed transform to estimate them.

The Fourier synchrosqueezed transform is based on the short-time Fourier transform implemented in the `spectrogram` function. For certain kinds of nonstationary signals, the synchrosqueezed transform resembles the reassigned spectrogram because it generates sharper time-frequency estimates than the conventional transform. The `fsst` function determines the short-time Fourier transform of a function, f, using a spectral window, g, and computing

$$V_g f(t,\eta) = \int_{\infty}^{\infty} f(x) g(x-t) e^{-j2\pi\eta(x-t)}\, dx.$$

Unlike the conventional definition, this definition has an extra factor of $e^{j2\pi\eta t}$. The transform values are then “squeezed” so that they concentrate around curves of instantaneous frequency in the time-frequency plane. The resulting synchrosqueezed transform is of the form

$$T_g f(t,\omega) = \int_{\infty}^{\infty} V_g f(t,\eta)\, \delta(\omega - \Omega_g f(t,\eta))\, d\eta,$$

where the instantaneous frequencies are estimated with the “phase transform”

$$\Omega_g f(t,\eta) = \frac{1}{j2\pi} \frac{\frac{\partial}{\partial t} V_g f(t,\eta)}{V_g f(t,\eta)} = \eta - \frac{1}{j2\pi} \frac{V_{\partial g/\partial t} f(t,\eta)}{V_g f(t,\eta)}.$$

The transform in the denominator decreases the influence of the window. To see a simple example, refer to “Detect Closely Spaced Sinusoids”. The definition of $T_g f(t,\omega)$ differs by a factor of $1/g(0)$ from other expressions found in the literature. `fsst` incorporates the factor in the mode-reconstruction step.

Unlike the reassigned spectrogram, the synchrosqueezed transform is invertible and thus can reconstruct the individual modes that compose the signal. Invertibility imposes some constraints on the computation of the short-time Fourier transform:

- The number of DFT points is equal to the length of the specified window.
- The overlap between adjoining windowed segments is one less than the window length.
- The reassignment is performed only in frequency.

To find the modes, integrate the synchrosqueezed transform over a small frequency interval around $\Omega_g f(t,\eta)$:

$$f_k(t) \approx \frac{1}{g(0)} \int_{\omega - \Omega_k(t)| < \varepsilon} T_g f(t,\omega)\, d\omega,$$

where ε is a small number.

The synchrosqueezed transform produces narrow ridges compared to the windowed short-time Fourier transform. However, the width of the short-time transform still affects the ability of the synchrosqueezed transform to separate modes. To be resolvable, the modes must obey these conditions:

1. For each mode, the frequency must be strictly greater than the rate of change of the amplitude: $\frac{d\phi_k(t)}{dt} > \frac{dA_k(t)}{dt}$ for all k.
2. Distinct modes must be separated by at least the frequency bandwidth of the window. If the support of the window is the interval [–Δ,Δ], then $\left|\frac{d\phi_k(t)}{dt} - \frac{d\phi_m(t)}{dt}\right| > 2\Delta$ for $k \neq m$.

For an illustration, refer to “Detect Closely Spaced Sinusoids”.

References

[1] Auger, François, Patrick Flandrin, Yu-Ting Lin, Stephen McLaughlin, Sylvain Meignen, Thomas Oberlin, and Hau-Tieng Wu. "Time-Frequency Reassignment and Synchrosqueezing: An Overview." *IEEE Signal Processing Magazine*. Vol. 30, November 2013, pp. 32–41.

[2] Oberlin, Thomas, Sylvain Meignen, and Valérie Perrier. "The Fourier-based Synchrosqueezing Transform." *Proceedings of the 2014 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 315–319.

[3] Thakur, Gaurav, and Hau-Tieng Wu. "Synchrosqueezing-based Recovery of Instantaneous Frequency from Nonuniform Samples." *SIAM Journal of Mathematical Analysis*. Vol. 43, 2011, pp. 2078–2095.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The window must be double precision.

- Duration arrays are not supported for code generation.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- The length of the window must be smaller than or equal to the length of the input signal.
- The syntax with no output arguments is not supported.

Thread-Based Environment
Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

Usage notes and limitations:

- The syntax with no output arguments is not supported.

For more information, see “Run MATLAB Functions in Thread-Based Environment”.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Apps
Signal Analyzer

Functions
`ifsst` | `pspectrum` | `spectrogram` | `tfridge`

Topics
“Hilbert Transform and Instantaneous Frequency”
“Practical Introduction to Time-Frequency Analysis”
“Time-Frequency Gallery”

Introduced in R2016b

FVTool

Filter Visualization Tool

Description

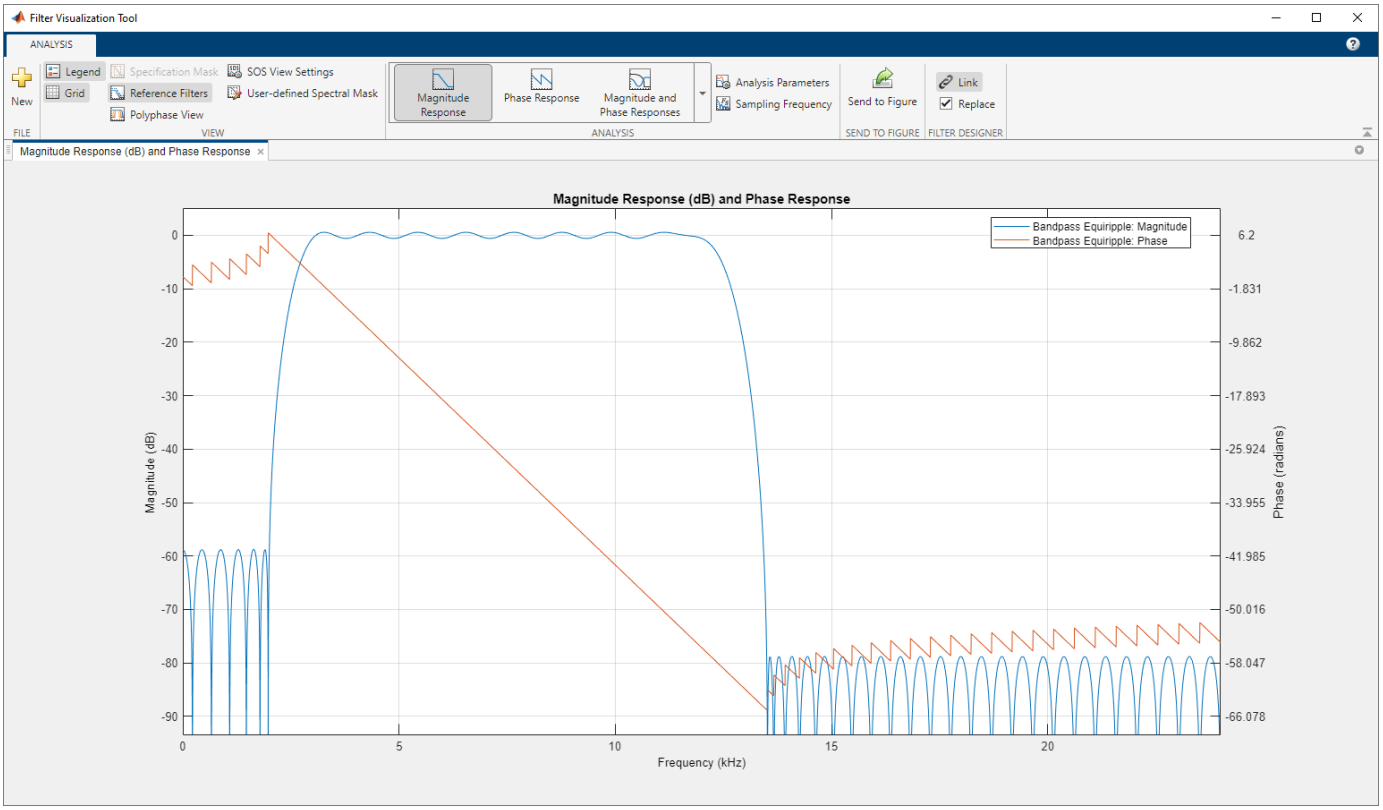
Filter Visualization Tool is an interactive app that enables you to display and analyze the responses, coefficients, and other information of a filter. You can also synchronize **FVTool** and **Filter Designer** to immediately visualize any changes made to a filter design.

In the app, you can view:

- Magnitude response
- Phase response
- Group delay
- Phase delay
- Impulse response
- Step response
- Pole-zero plot
- Filter coefficients
- Filter information

For more information, see “Analysis Types” on page 1-936.

If you have installed the DSP System Toolbox, **FVTool** can also visualize the frequency response of a filter System object. If you need to filter streaming data in real time, using System objects is the recommended approach. For more information, see `fvtool` (DSP System Toolbox).



Open the FVTool

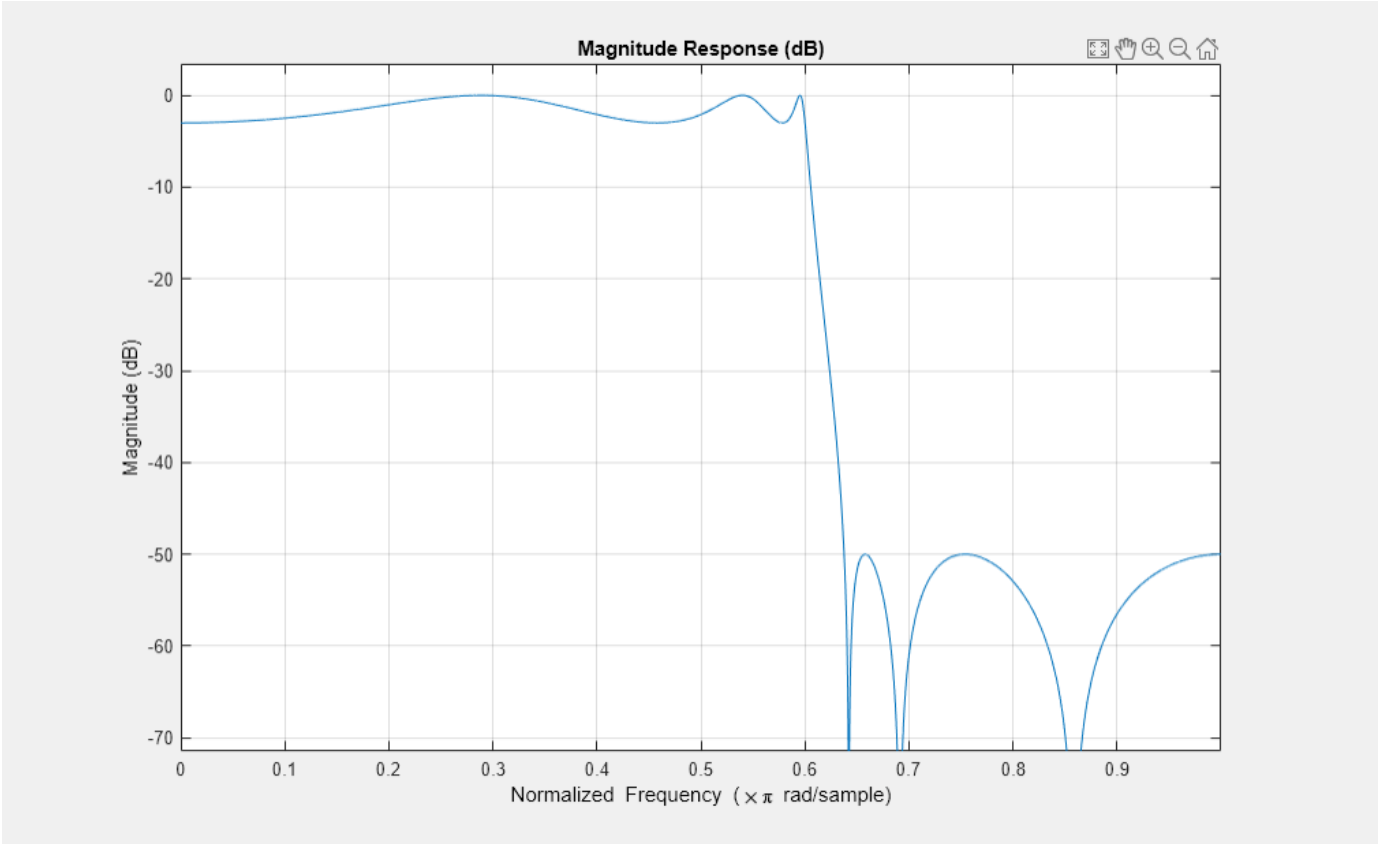
FVTool can be opened programmatically using one of the methods described in “Programmatic Use” on page 1-935.

Examples

Magnitude Response of Elliptic Filter

Consider a 6th-order elliptic filter with a passband ripple of 3 dB, a stopband attenuation of 50 dB, a sample rate of 1 kHz, and a normalized passband edge of 300 Hz. Display the magnitude response of the filter.

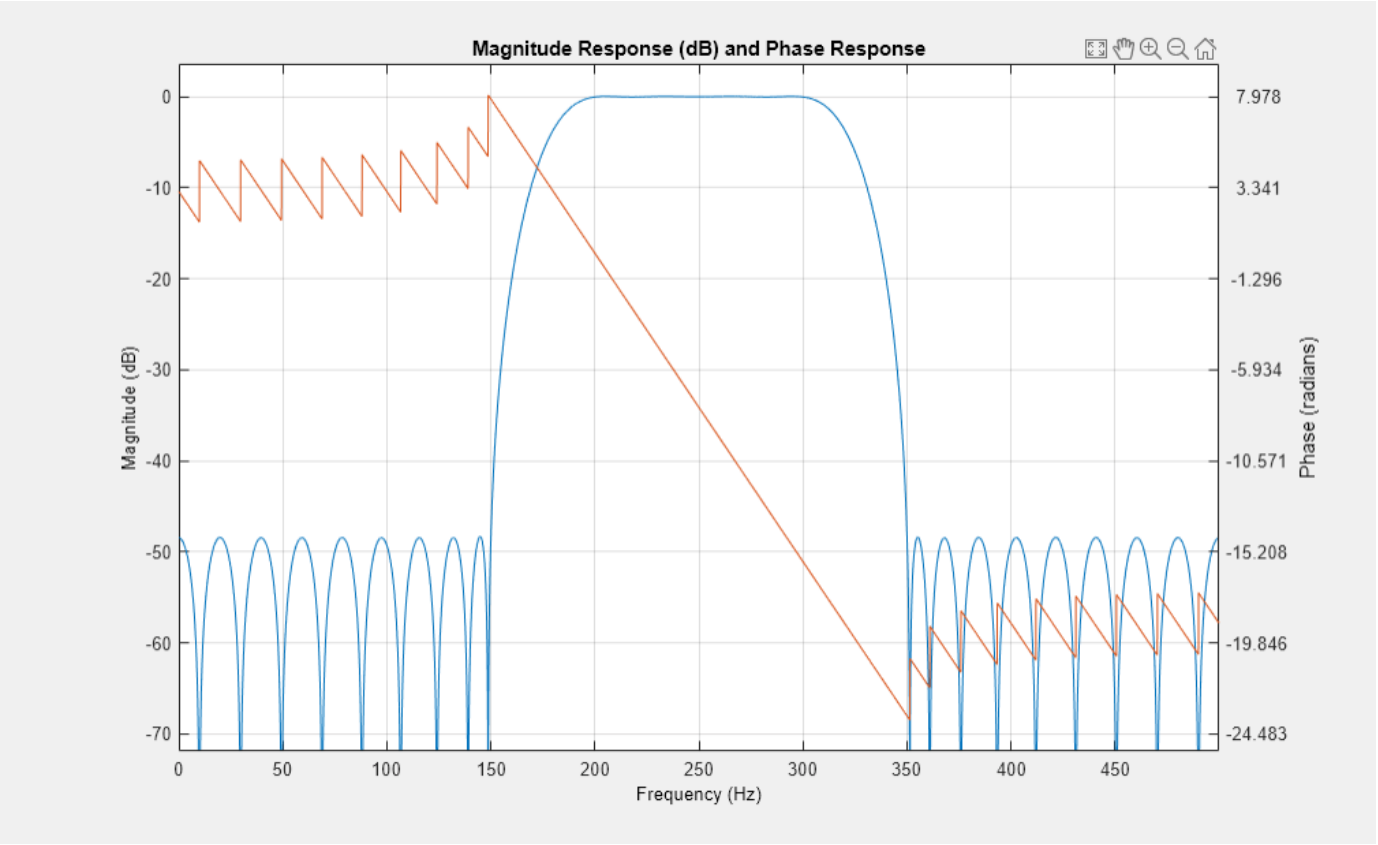
```
[b,a] = ellip(6,3,50,300/500);
fvtool(b,a)
```



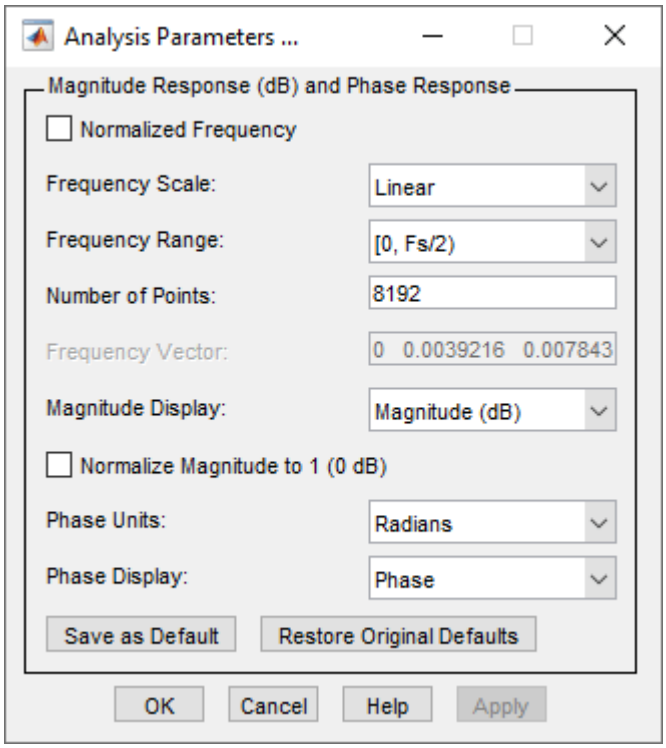
Magnitude and Phase Response of Bandpass FIR Filter

Design a 50th-order bandpass FIR filter with stopband frequencies of 150 Hz and 350 Hz and passband frequencies of 200 Hz and 300 Hz. The sample rate is 1000 Hz. Visualize the magnitude and phase response of the filter.

```
N = 50;
Fstop1 = 150;
Fstop2 = 350;

Fpass1 = 200;
Fpass2 = 300;

Fs = 1e3;

bpFilt = designfilt("bandpassfir",FilterOrder=N, ...
StopbandFrequency1=Fstop1,StopbandFrequency2=Fstop2,...
PassbandFrequency1=Fpass1,PassbandFrequency2=Fpass2,...
SampleRate=Fs);

fvtool(bpFilt,Analysis="freq")
```



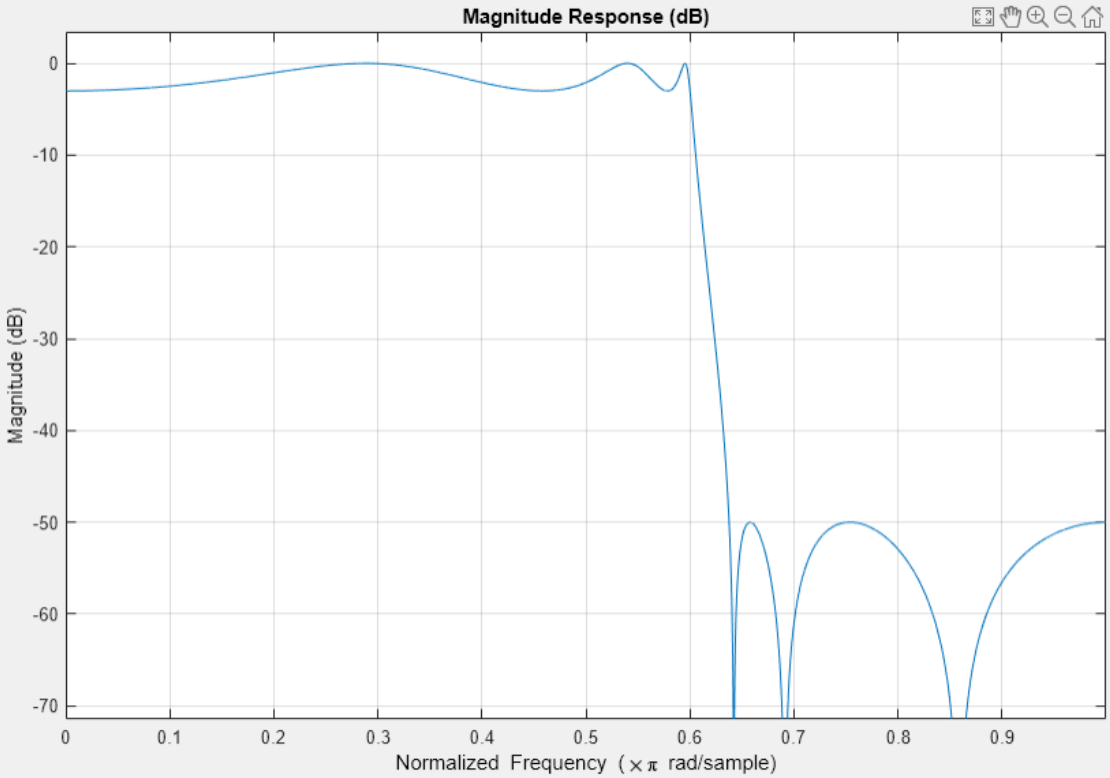
Select the **Analysis Parameters** button in the toolstrip. An Analysis Parameters window appears that shows the parameters associated with the plot.



FVTool Figure Handle Commands

Display the magnitude response of a 6th-order elliptic filter. Specify a passband ripple of 3 dB, a stopband attenuation of 50 dB, a sample rate of 1 kHz, and a normalized passband edge of 300 Hz. Obtain the handle for FVTool.

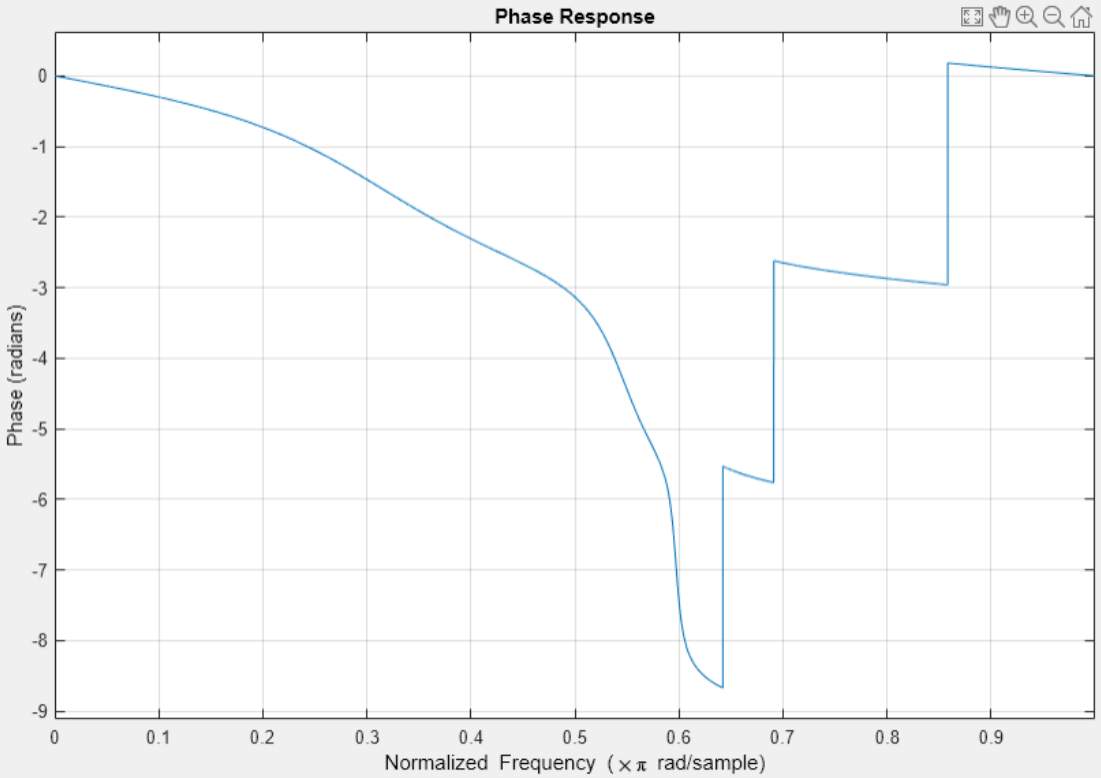
```
[b,a] = ellip(6,3,50,300/500);
h = fvtool(b,a)
```




```
h =
  Figure (filtervisualizationtool) with properties:

      Number: 1
        Name: 'Magnitude Response (dB)'
       Color: [0.9400 0.9400 0.9400]
    Position: [360 502 560 420]
       Units: 'pixels'

  Use get to show all properties
```

Use the FVTool handle to display the phase response of the filter.

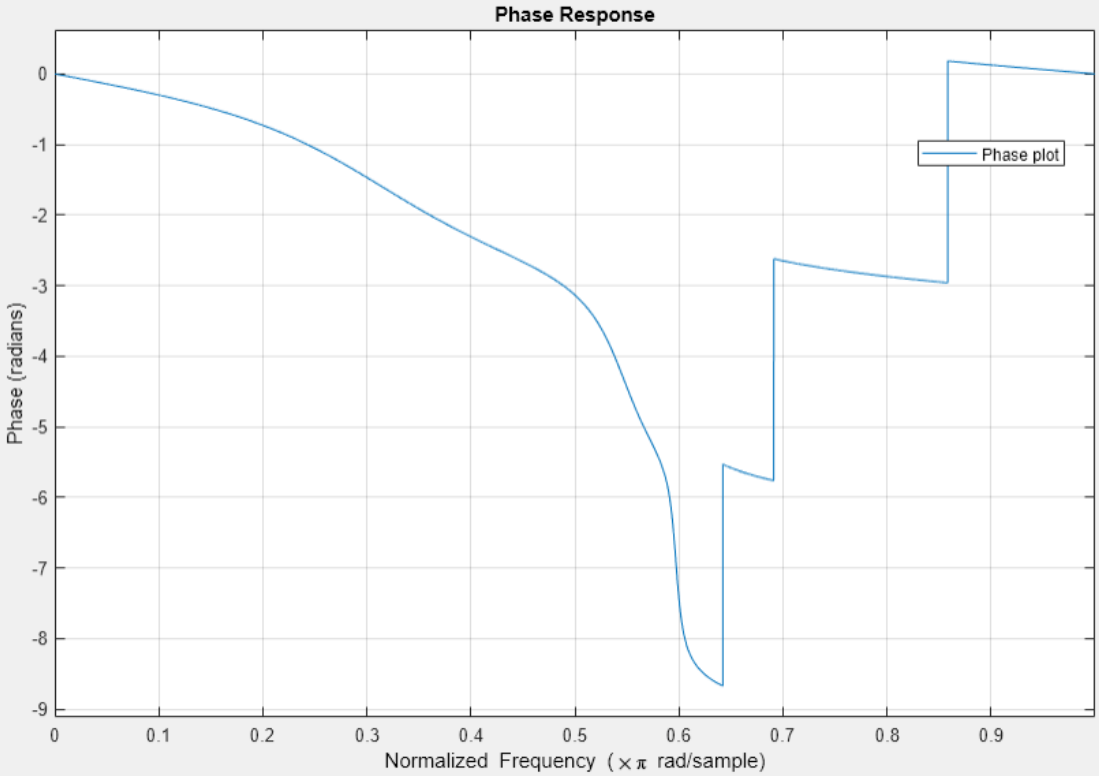
```
h.Analysis = "phase"
```




```
h =
  Figure (filtervisualizationtool) with properties:

      Number: 1
        Name: 'Figure 1: Phase Response'
       Color: [0.9400 0.9400 0.9400]
    Position: [1 1 1008 618]
       Units: 'pixels'

  Use get to show all properties
```

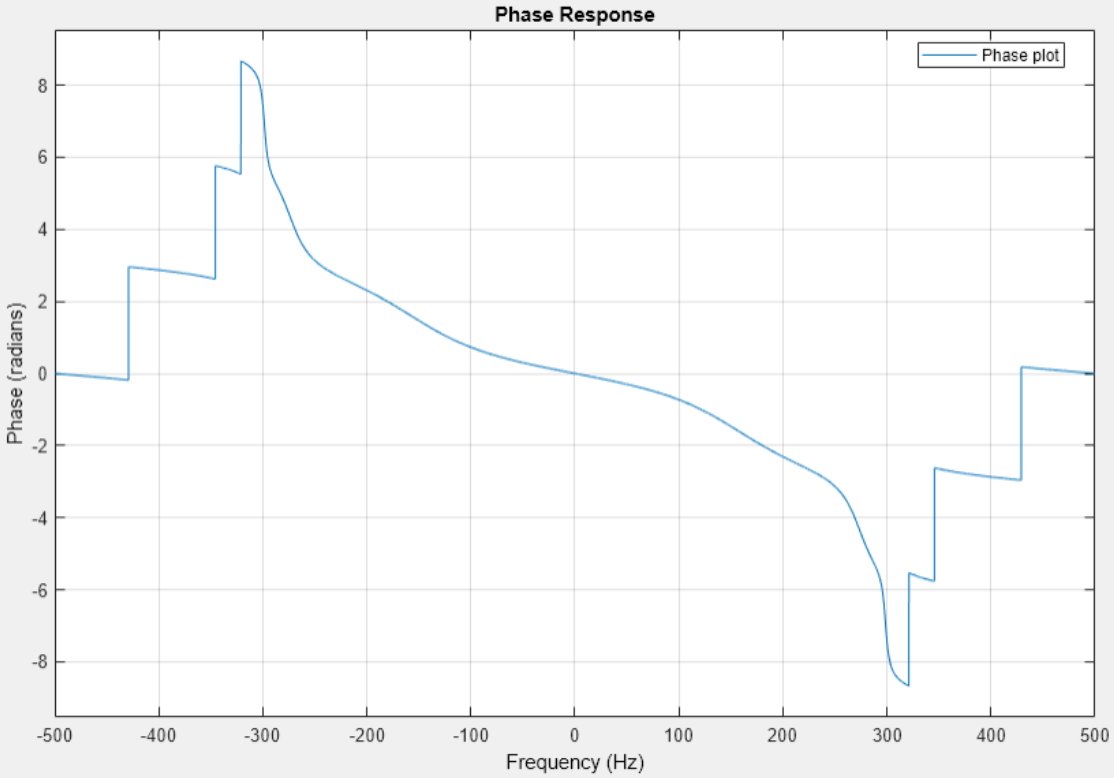
Turn on the plot legend and add text.

```
legend(h,"Phase plot")
```



Specify a sample rate of 1 kHz. Display the two-sided centered response.

```
h.Fs = 1000;
h.FrequencyRange = "[-Fs/2, Fs/2)"
```




```
h =
  Figure (filtervisualizationtool) with properties:

      Number: 1
        Name: 'Phase Response'
       Color: [0.9400 0.9400 0.9400]
    Position: [1 1 1008 618]
       Units: 'pixels'

  Use get to show all properties
```

View the all the properties of the plot. The properties specific to FVTool are at the end of the list.

```
get(h)
```

```
                    Grid: on
                  Legend: 'on'
              DesignMask: 'off'
          SOSViewSettings: [1x1 dspopts.sosview]
                      Fs: 1000
                Alphamap: [0 0.0159 0.0317 0.0476 0.0635 0.0794 0.0952 ... ]
          CloseRequestFcn: 'closereq'
                    Color: [0.9400 0.9400 0.9400]
                 Colormap: [256x3 double]
              ContextMenu: [0x0 GraphicsPlaceholder]
              CurrentAxes: [1x1 Axes]
         CurrentCharacter: ''
```

```
          CurrentObject: [0x0 GraphicsPlaceholder]
           CurrentPoint: [0 0]
           DockControls: on
               FileName: ''
           IntegerHandle: on
          InvertHardcopy: on
             KeyPressFcn: ''
           KeyReleaseFcn: ''
                    Name: 'Phase Response'
                NextPlot: 'new'
             NumberTitle: off
              PaperUnits: 'inches'
       PaperOrientation: 'portrait'
          PaperPosition: [-1 2.2812 10.5000 6.4375]
     PaperPositionMode: 'auto'
             PaperSize: [8.5000 11]
             PaperType: 'usletter'
               Pointer: 'arrow'
     PointerShapeCData: [16x16 double]
   PointerShapeHotSpot: [1 1]
               Position: [1 1 1008 618]
               Renderer: 'opengl'
           RendererMode: 'auto'
                 Resize: on
              ResizeFcn: @(~,~)fix_listbox_position(this,hFVT)
           SelectionType: 'normal'
                 ToolBar: 'none'
                    Type: 'figure'
                   Units: 'pixels'
  WindowButtonDownFcn: ''
WindowButtonMotionFcn: ''
    WindowButtonUpFcn: ''
    WindowKeyPressFcn: ''
  WindowKeyReleaseFcn: ''
 WindowScrollWheelFcn: ''
           WindowStyle: 'docked'
          BeingDeleted: off
          ButtonDownFcn: ''
               Children: [6x1 Graphics]
               Clipping: on
              CreateFcn: ''
              DeleteFcn: ''
            BusyAction: 'queue'
     HandleVisibility: 'on'
                HitTest: on
          Interruptible: on
                 Parent: [1x1 Root]
               Selected: off
    SelectionHighlight: on
                   Tag: 'filtervisualizationtool'
              UserData: []
               Visible: on
          ShowReference: 'on'
   NormalizedFrequency: 'off'
          PolyphaseView: 'off'
     OverlayedAnalysis: ''
        NumberofPoints: 8192
            PhaseUnits: 'Radians'
```

```
       Analysis: 'phase'
FrequencyVector: [0 0.0039 0.0078 0.0118 0.0157 0.0196 0.0235 ... ]
   PhaseDisplay: 'Phase'
 FrequencyRange: '[-Fs/2, Fs/2)'
 FrequencyScale: 'Linear'
```

- “Filter Analysis Using FVTool”

Programmatic Use

`fvtool(b,a)` opens FVTool and displays the magnitude response of the digital filter defined with numerator `b` and denominator `a`. Specify `b` and `a` coefficients in ascending order of power z^{-1}.

`fvtool(sos)` opens FVTool and displays the magnitude response of the digital filter defined by the L-by-6 matrix of second order sections:

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

The rows of `sos` contain the numerator and denominator coefficients b_{ik} and a_{ik} of the cascade of 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 of sections L 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.

`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`.

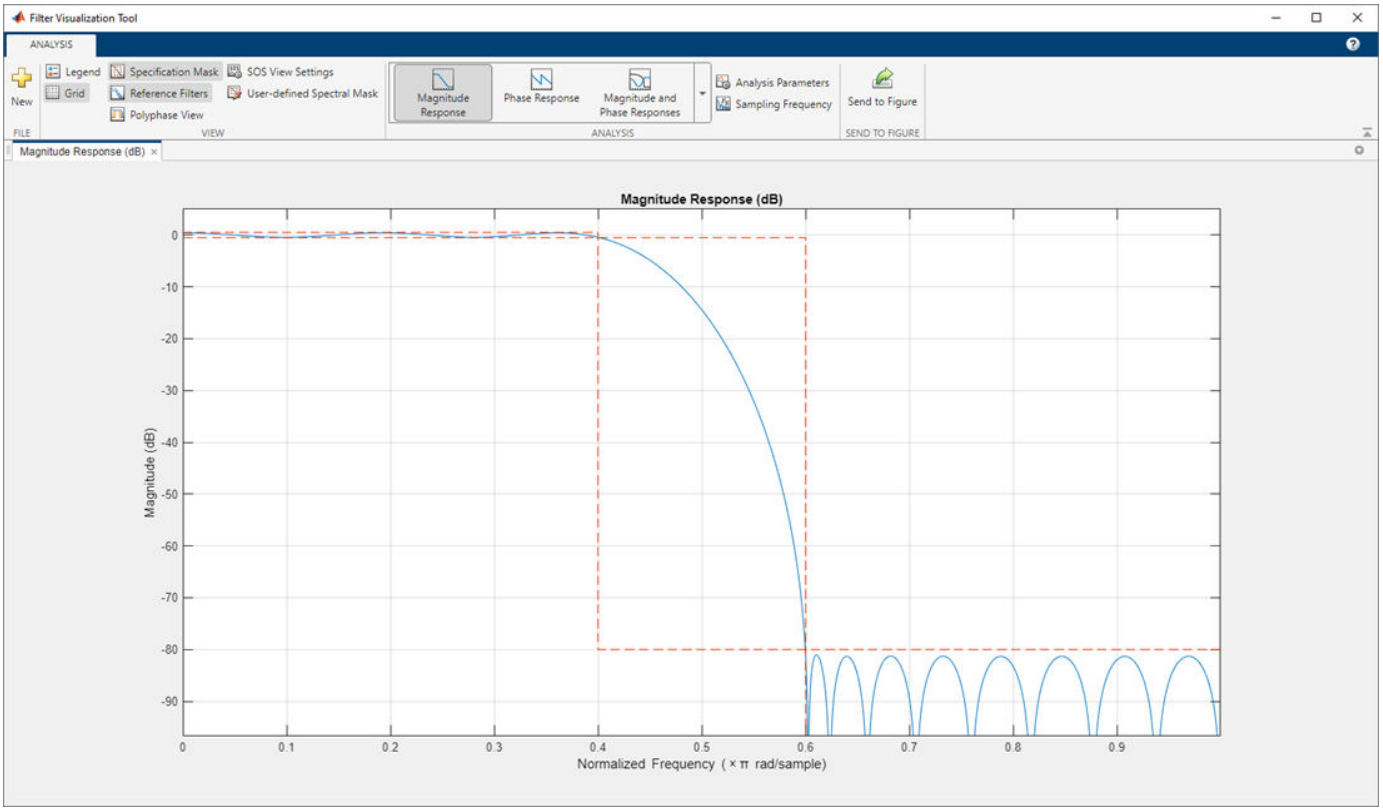
`h = fvtool( ___ )` returns a figure handle `h`. You can use this handle to interact with FVTool from the command line. For more information, see “Controlling FVTool from the MATLAB Command Line”.

More About

Controlling FVTool from the GUI

Display and analyze responses of one or multiple filters using controls on the toolstrip.

- By default, the app displays the magnitude response of a filter. To change the display, select an option from the **Analysis** list in the **Analysis** section of the toolstrip.
- To overlay a second response on the plot, select an available response from the **Overlay Analysis** list in the **Analysis** section of the toolstrip. The app adds a second *y*-axis to the right side of the response plot. The Analysis Parameters dialog box shows parameters for the *x*-axis and both *y*-axes.
- To adjust view settings, analysis parameters, or specify sampling frequency, use the corresponding buttons on the toolstrip. You can also toggle the plot legend and grid.
- To edit a plot, first click **Send to Figure**. In the new figure window, use the plot editing toolbar.



Analysis Types

FVTool has these analysis types:

Analysis	Description
Magnitude response	• To see the zero-phase response, right-click the *y*-axis label on the plot and select **Zero-phase** from the context menu. • For more information, see `freqz` and `zerophase`.

Analysis	Description
Phase response	• For more information, see `phasez`.
Magnitude response and phase response	• The responses are superimposed on one another. • For more information, see `freqz`.
Group delay	• Group delay is the average delay of the filter as a function of frequency. • For more information, see `grpdelay`.
Phase delay	• Phase delay is the time delay the filter imposes on each component of the input signal. • For more information, see `phasedelay`.
Impulse response	• The impulse response is the response of the filter to an impulse input. • For more information, see `impz`.
Step response	• The step response is the response of the filter to a step input. • For more information, see `stepz`.
Pole-zero plot	• The pole-zero plot shows the pole and zero locations of the filter on the *z*-plane. • For more information, see `zplane`.
Filter coefficients	• The coefficients depend on the filter structure (direct-form or lattice). • The app displays filter coefficients in a text box. For SOS filters, the app displays each section as a separate filter.
Filter information	• The filter information includes the structure, phase, stability, and implementation cost of the filter.

Linking to Filter Designer

When the **Filter Designer** app displays the analysis for a filter, in the app, select **View > Filter Visualization Tool** or the **Full View Analysis** toolbar button to open **FVTool** for the filter. In **FVTool**, use the **Link** button to link to **Filter Designer**. FVTool updates the current display with any changes made to the filter in **Filter Designer**. By default, the app retains the current filter and adds the new filter to the display. To remove the current filter and insert the new filter, select the **Replace** check box in the **Filter Designer** section of the toolstrip.

Compatibility Considerations

Controls are on the toolstrip

Behavior changed in R2022a

The behavior of **fvtool** has changed. In previous releases, the app had plot editing and analysis toolbars. Starting this release, access filter visualization controls on the toolstrip. To edit a plot, first export the plot to a figure and then use the figure controls.

See Also

Apps

Signal Analyzer | **Filter Designer**

Functions
`designfilt` | `digitalFilter`

Topics
“Filter Analysis Using FVTool”
“Modifying the Axes”
“Modifying the Plot”
“Controlling FVTool from the MATLAB Command Line”

Introduced before R2006a

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`.

`y = fwht(x,n)` returns the `n`-point discrete Walsh-Hadamard transform.

`y = fwht(x,n,ordering)` specifies the ordering to use for the returned Walsh-Hadamard transform coefficients.

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 = 1×8

     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 = 1×8

    19    -1    11    -9    -7    13   -15     5
```

Obtain the same result by extracting the nonzero values and Walsh functions programmatically.

```
ww = fwht(eye(length(y)))*length(y)
```

```
ww = 8×8
```

```
     1     1     1     1     1     1     1     1
     1     1     1     1    -1    -1    -1    -1
     1     1    -1    -1    -1    -1     1     1
     1     1    -1    -1     1     1    -1    -1
     1    -1    -1     1     1    -1    -1     1
     1    -1    -1     1    -1     1     1    -1
     1    -1     1    -1    -1     1    -1     1
     1    -1     1    -1     1    -1     1    -1
```

```
nz = find(y);
w = sum(y(nz)'.*ww(nz,:))
```

```
w = 1×8

    19    -1    11    -9    -7    13   -15     5
```

Input Arguments

x — Input signal

matrix | vector

Input signal, specified as a matrix or a vector. If `x` is a matrix, the Fast Walsh-Hadamard transform is calculated on each column of `x`. `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.

n — Points in discrete Walsh Hadamard transform

positive even integer scalar

Points in discrete Walsh Hadamard transform, specified as a positive even integer scalar.`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.

ordering — Order of Walsh Hadamard transform coefficients

`'sequency'` | `'hadamard'` | `'dyadic'`

Order of Walsh Hadamard transform coefficients, specified as `'sequency'`, `'hadamard'` or `'dyadic'`. 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:

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”.

Output Arguments

y — Discrete Walsh-Hadamard transform

matrix | vector

Discrete Walsh-Hadamard transform, returned as a matrix or a vector.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`ifwht` | `dct` | `idct` | `fft` | `ifft`

Introduced in R2008b

gauspuls

Gaussian-modulated sinusoidal RF 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

`yi = gauspuls(t,fc,bw)` returns a unit-amplitude Gaussian-modulated sinusoidal RF pulse at the times indicated in array `t`, with a center frequency `fc` in hertz and a fractional bandwidth `bw`

`yi = gauspuls(t,fc,bw,bwr)` returns a unit-amplitude inphase Gaussian RF pulse with a fractional bandwidth of `bw` as measured at a level of `bwr` dB with respect to the normalized signal peak.

`[yi,yq] = gauspuls( ___ )` also returns the quadrature pulse. This syntax can include any combination of input arguments from previous syntaxes.

`[yi,yq,ye] = gauspuls( ___ )` returns the RF signal envelope.

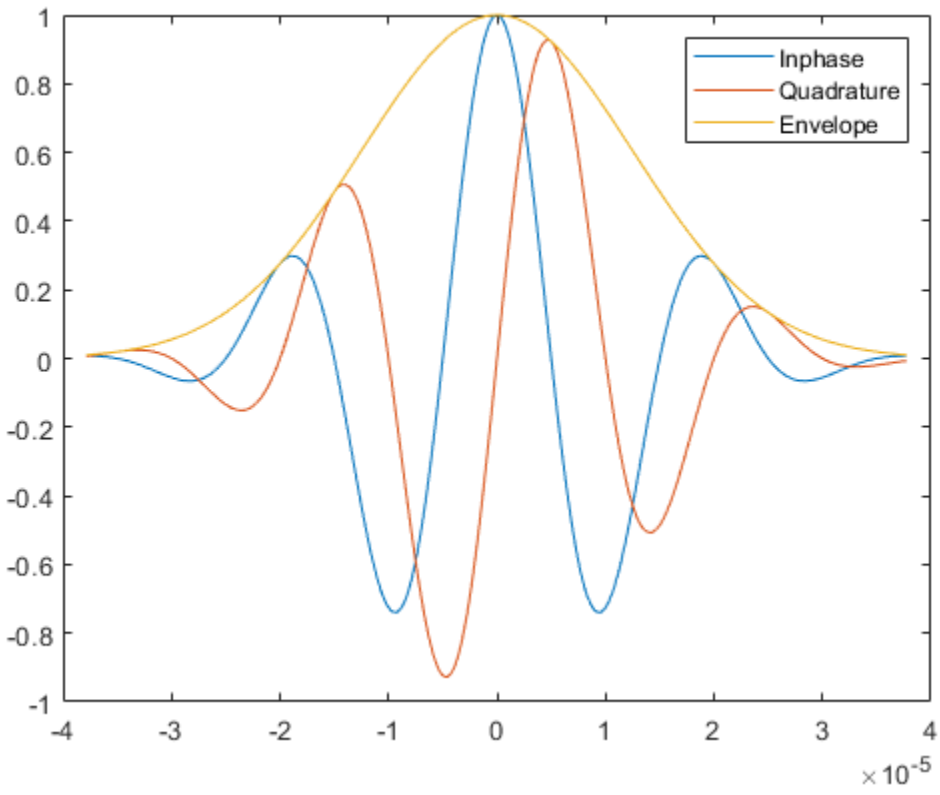
`tc = gauspuls('cutoff',fc,bw,bwr,tpe)` returns the cutoff time `tc` at which the trailing pulse envelope falls below `tpe` dB with respect to the peak envelope amplitude.

Examples

Generate Gaussian RF Pulse

Plot a 50 kHz Gaussian RF pulse with 60% bandwidth, sampled at a rate of 10 MHz. Truncate the pulse where the envelope falls 40 dB below the peak. Also plot the quadrature pulse and the RF signal envelope.

```
tc = gauspuls('cutoff',50e3,0.6,[],-40);
t = -tc : 1e-7 : tc;
[yi,yq,ye] = gauspuls(t,50e3,0.6);

plot(t,yi,t,yq,t,ye)
legend('Inphase','Quadrature','Envelope')
```



Input Arguments

`t` — Vector of time values

vector

Vector of time values at which the unit-amplitude Gaussian RF pulse is calculated.

Data Types: `single` | `double`

`fc` — Center frequency

`1000` (default) | real positive scalar

Center frequency of the Gaussian-modulated sinusoidal pulses, specified as a real positive scalar expressed in Hz.

`bw` — Fractional bandwidth

`0.5` (default) | real positive scalar

Fractional bandwidth of the Gaussian-modulated sinusoidal pulses,specified as a real positive scalar.

`bwr` — Fractional bandwidth reference level

–6 (default) | real negative scalar

Fractional bandwidth reference level of the Gaussian-modulated sinusoidal pulses, specified as a real negative scalar. `bwr` indicates a reference level less than peak (unit) envelope amplitude. The

fractional bandwidth is specified in terms of power ratios. This corresponds to the `-3 dB` point expressed in magnitude ratios.

`tpe` — Trailing pulse envelope level
`–60` (default) | real negative scalar

Trailing pulse envelope level, specified as a real negative scalar in dB. The `tpe` indicates a reference level less than peak (unit) envelope amplitude.

Output Arguments

`yi` — Inphase Gaussian pulse
vector

Inphase Gaussian-modulated sinusoidal pulse, returned as a vector of unit amplitude at the times indicated by the time vector `t`.

`yq` — Quadrature Gaussian pulse
vector

Quadrature Gaussian-modulated sinusoidal pulse, returned as a vector of unit amplitude at the times indicated by the time vector `t`.

`ye` — RF signal envelope
vector

RF signal envelope of unit amplitude at the times indicated by the time vector `t`.

`tc` — Cutoff time
positive real scalar

The cutoff time in seconds at which the trailing pulse envelope falls below `tpe` dB with respect to the peak envelope amplitude.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`chirp` | `cos` | `diric` | `pulstran` | `rectpuls` | `sawtooth` | `sin` | `sinc` | `square` | `tripuls`

Introduced before R2006a

gaussdesign

Gaussian FIR pulse-shaping filter design

Syntax

```
h = gaussdesign(bt,span,sps)
```

Description

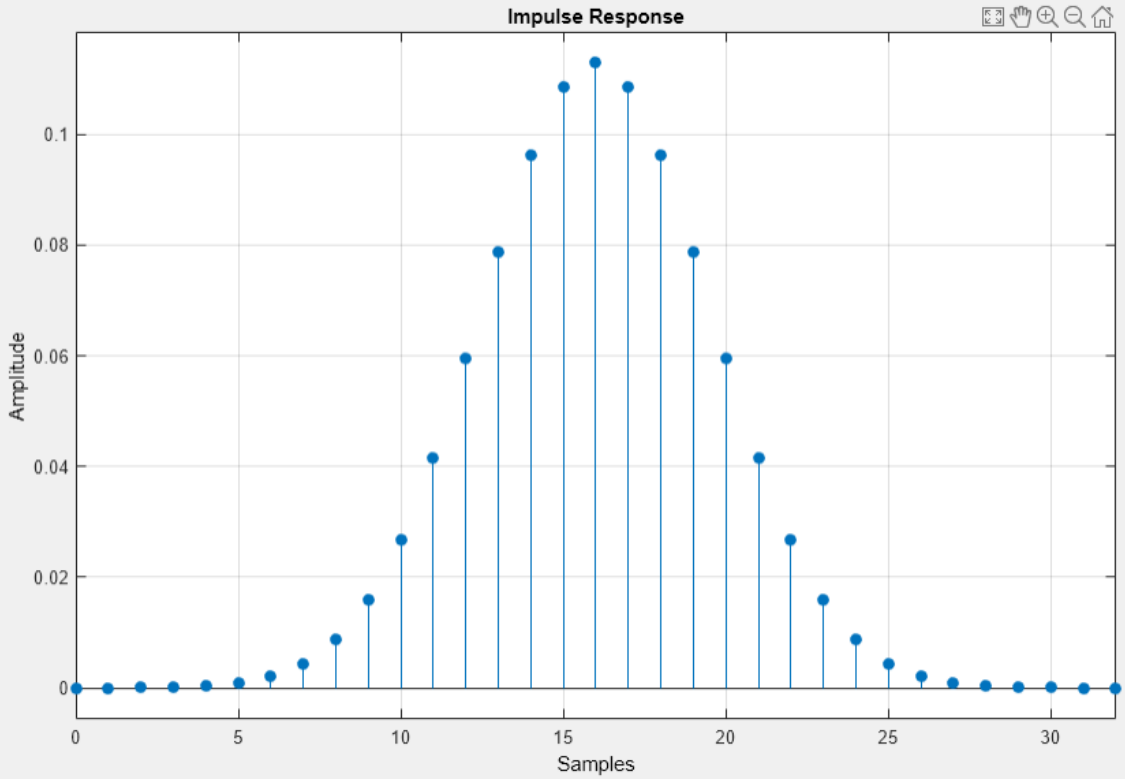
`h = gaussdesign(bt,span,sps)` designs a lowpass FIR Gaussian pulse-shaping filter and returns a vector, `h`, of filter coefficients. The filter is truncated to `span` symbols, and each symbol period contains `sps` samples. The order of the filter, `sps*span`, must be even.

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`

Algorithms

The impulse response of the Gaussian filter is given by

$$h(t) = \frac{\exp\left(\frac{-t^2}{2\delta^2}\right)}{\sqrt{2\pi}\cdot\delta}$$

where

$$\delta = \frac{\sqrt{\log 2}}{2\pi BT}.$$

BT is the bandwidth-symbol time product specified in `bt`, where *B* is the 3-dB bandwidth of the filter and *T* is the symbol time. The number of symbols between the start and end of the impulse (`span`) and the number of samples per symbol (`sps`) determine the length of the impulse response: $span \times sps + 1$.

For more information, see “FIR Gaussian Pulse-Shaping Filter Design”.

References

[1] 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.

[2] Rappaport, Theodore S. *Wireless Communications: Principles and Practice*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 2002.

See Also

`rcosdesign`

Introduced in R2013b

gausswin

Gaussian window

Syntax

```
w = gausswin(L)
w = gausswin(L,alpha)
```

Description

`w = gausswin(L)` returns an `L`-point Gaussian window.

`w = gausswin(L,alpha)` returns an `L`-point Gaussian window with width factor `alpha`.

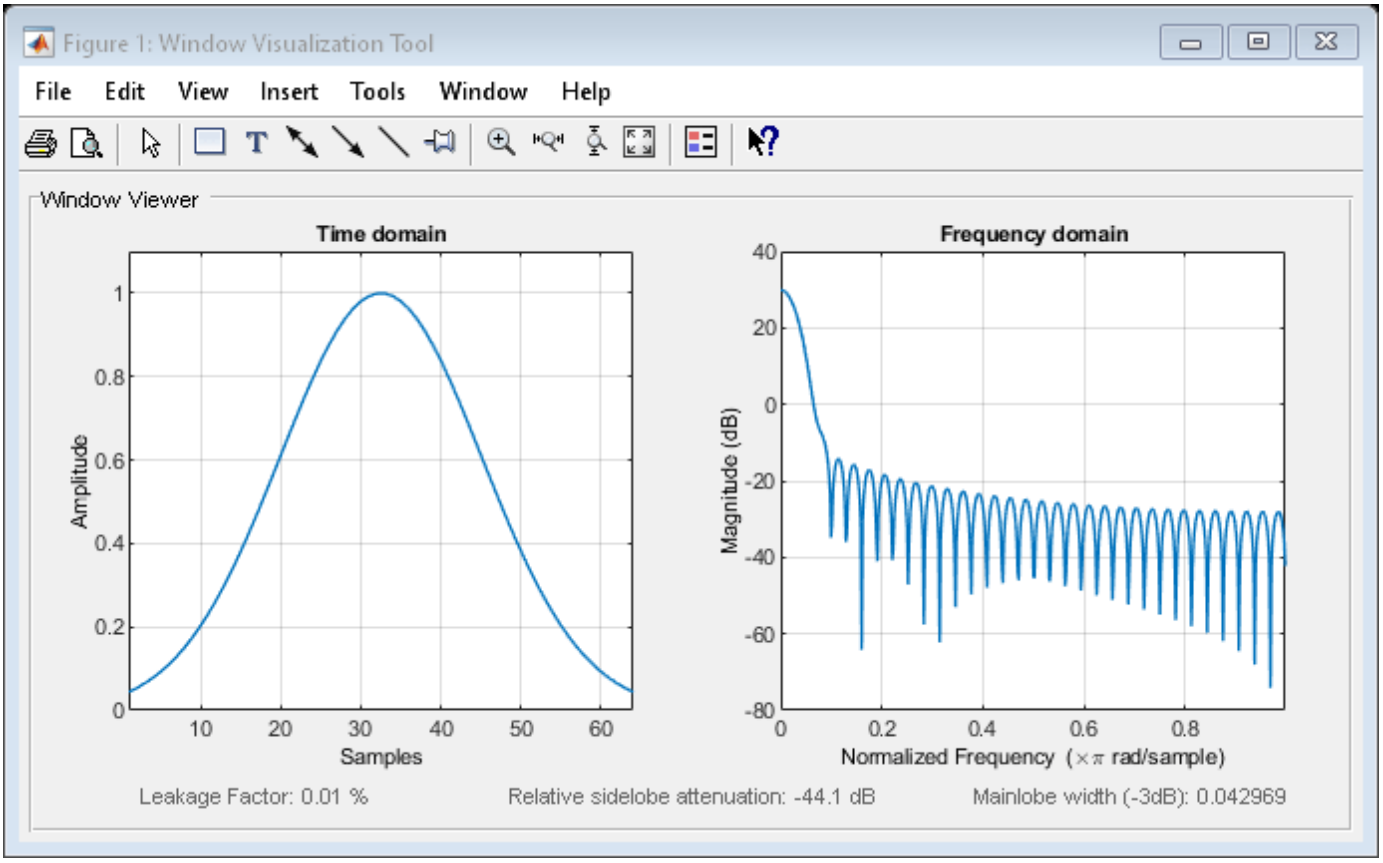
Note If the window appears to be clipped, increase `L`, the number of points.

Examples

Gaussian Window

Create a 64-point Gaussian window. Display the result in `wvtool`.

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



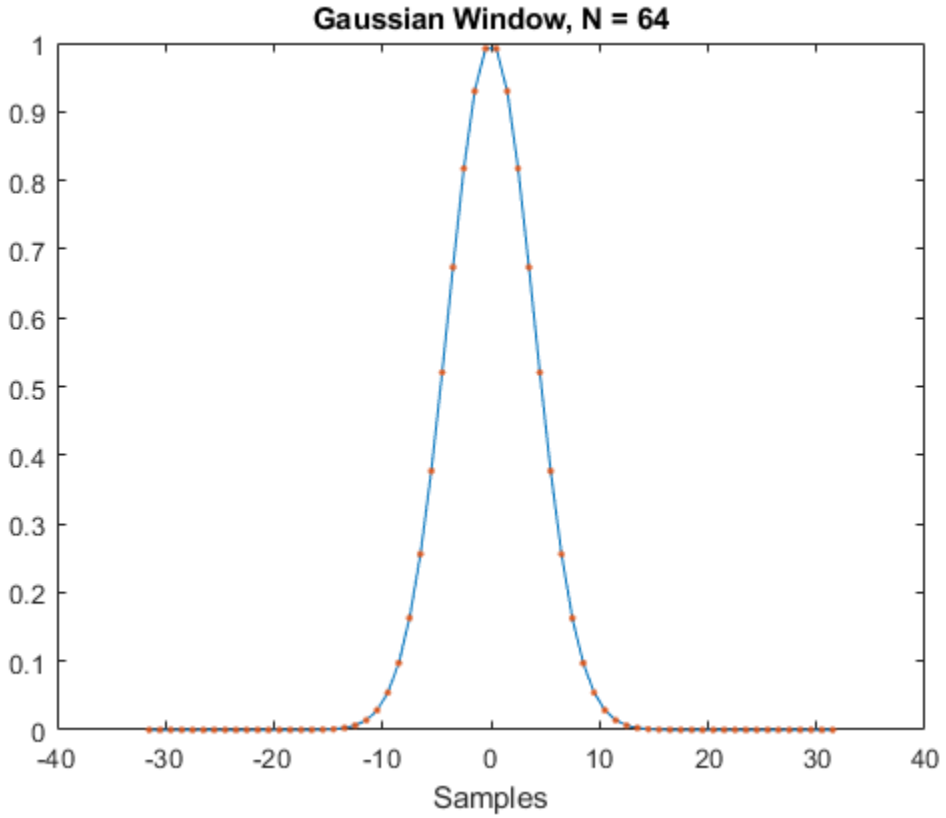
Gaussian Window and the Fourier Transform

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')
```

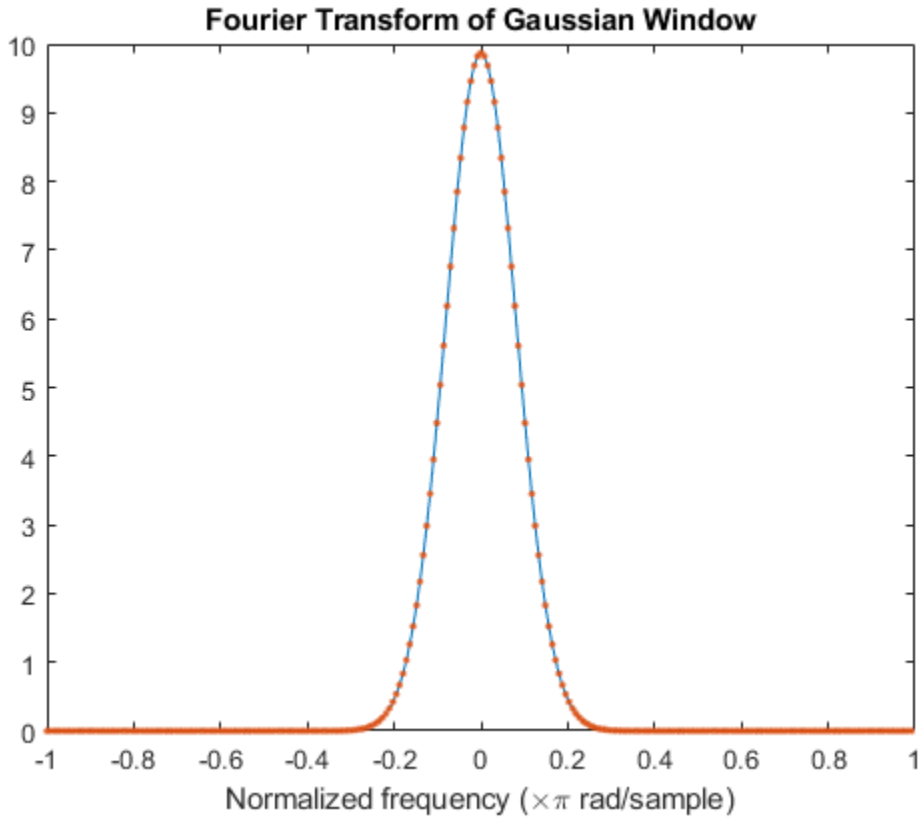


Obtain the Fourier transform of the Gaussian window at 256 points. Use `fftshift` to center the Fourier transform at zero frequency (DC).

```
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.

```
ydft = exp(-1/2*(freq/(1/stdev)).^2)*(stdev*sqrt(2*pi));

plot(freq/pi,abs(wdft))
hold on
plot(freq/pi,abs(ydft),'.')
hold off

xlabel('Normalized frequency (\times\pi rad/sample)')
title('Fourier Transform of Gaussian Window')
```



Input Arguments

L — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

alpha — Width factor

`2.5` (default) | positive real scalar

Width factor, specified as a positive real scalar. `alpha` is inversely proportional to the width of the window.

Data Types: `single` | `double`

Output Arguments

w — Gaussian window

column vector

Gaussian window, returned as a column vector.

Algorithms

The coefficients of a Gaussian window are computed from the following equation:

$$w(n) = e^{-\frac{1}{2}\left(\alpha\frac{n}{(L-1)/2}\right)^2} = e^{-n^2/2\sigma^2},$$

where $-(L-1)/2 \le n \le (L-1)/2$, and α is inversely proportional to the standard deviation, σ, of a Gaussian random variable. The exact correspondence with the standard deviation of a Gaussian probability density function is $\sigma = (L-1)/(2\alpha)$.

References

[1] Hansen, Eric W. *Fourier Transforms: Principles and Applications*. New York: John Wiley & Sons, 2014.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps

Window Designer

Functions

WVTool | `chebwin` | `kaiser` | `tukeywin`

Introduced before R2006a

gmonopuls

Gaussian monopulse

Syntax

```
y = gmonopuls(t,fc)
tc = gmonopuls('cutoff',fc)
```

Description

`y = gmonopuls(t,fc)` returns samples of the unit-amplitude Gaussian monopulse with center frequency `fc` at the times indicated in array `t`.

`tc = gmonopuls('cutoff',fc)` returns the time duration between the maximum and minimum amplitudes of the pulse.

Examples

Gaussian Monopulse

Consider a Gaussian monopulse with center frequency f_c = 2 GHz and sampled at a rate of 100 GHz. Determine the cutoff time t_c using the `'cutoff'` option and compute the monopulse between $-2t_c$ and $2t_c$.

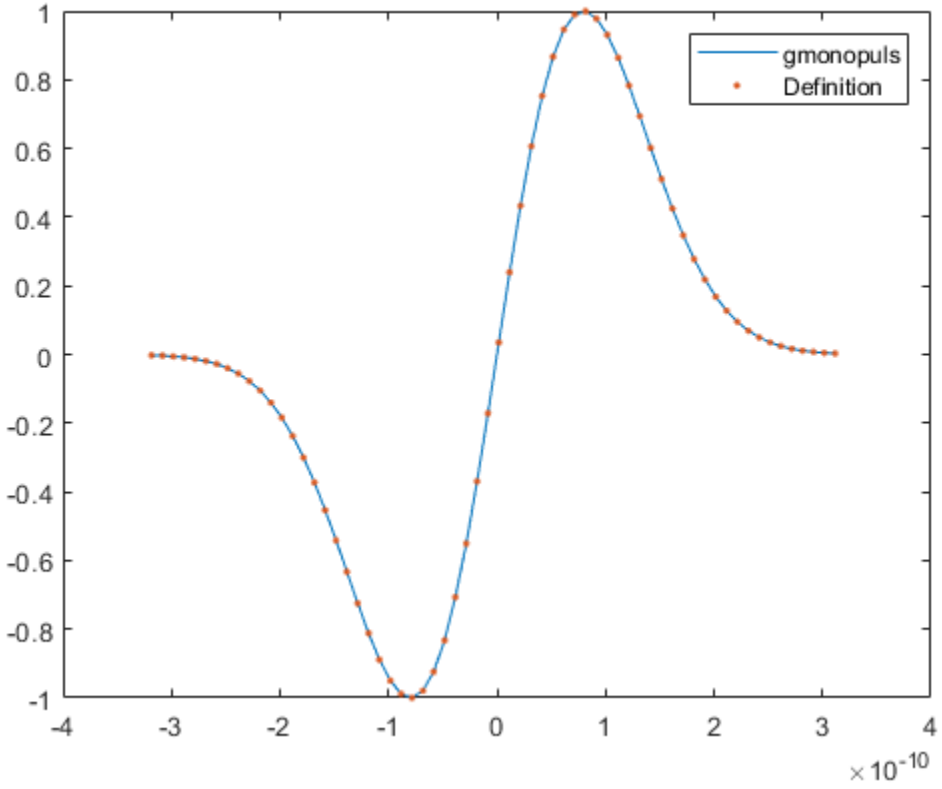
```
fc = 2e9;
fs = 100e9;

tc = gmonopuls('cutoff',fc);
t  = -2*tc:1/fs:2*tc;

y = gmonopuls(t,fc);
```

The monopulse is defined by the equation

$$y(t) = e^{1/2}(t/\sigma)\exp(-(t/\sigma)^2/2),$$

where $\sigma = t_c/2 = 1/(2\pi f_c)$ and the exponential factor is such that $y(\sigma) = 1$. Plot the two curves and verify that they match.

```
sg = 1/(2*pi*fc);

ys = exp(1/2)*t/sg.*exp(-(t/sg).^2/2);

plot(t,y,t,ys,'.')
legend('gmonopuls','Definition')
```



Gaussian Monopulse Pulse Train

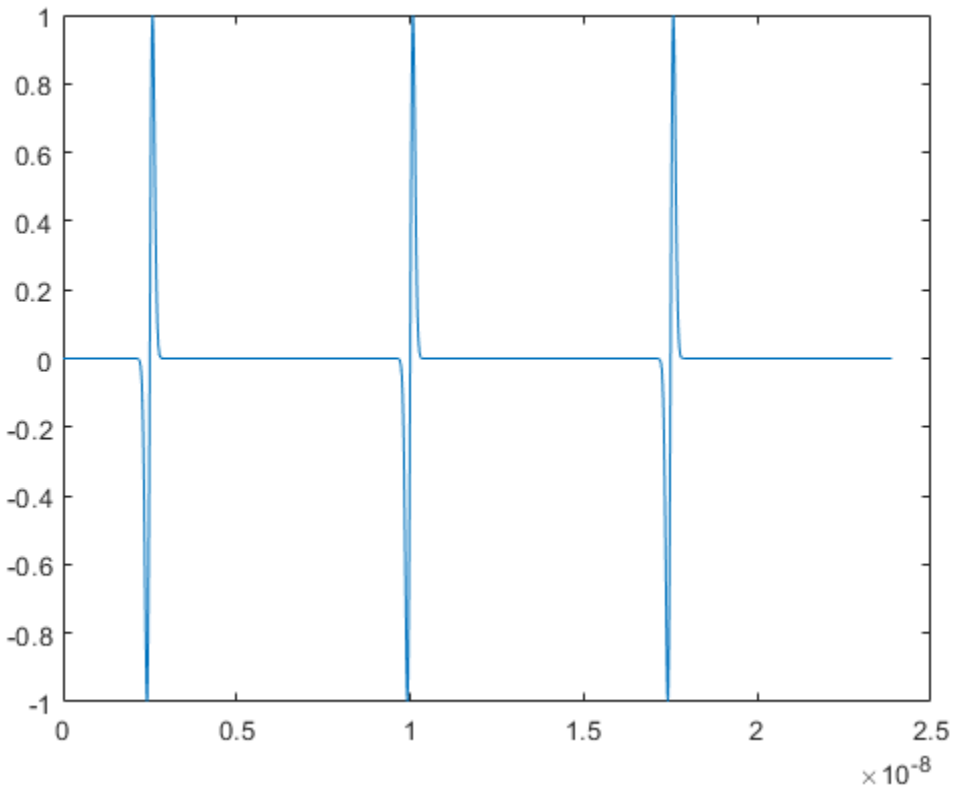
Consider a Gaussian monopulse with center frequency f_c = 2 GHz and sampled at a rate of 100 GHz. Use the monopulse to construct a pulse train with a spacing of 7.5 ns.

Determine the width t_c of each pulse using the `'cutoff'` option. Set the delay times to be integer multiples of the spacing.

```
fc = 2e9;
fs = 100e9;

tc = gmonopuls('cutoff',fc);
D = ((0:2)*7.5+2.5)*1e-9;
```

Generate the pulse train such that the total duration is $150t_c$. Plot the result.

```
t  = 0:1/fs:150*tc;
yp = pulstran(t,D,'gmonopuls',fc);

plot(t,yp)
```



Input Arguments

`t` — Time values

vector

Time values at which the unit-amplitude Gaussian monopulse is calculated, specified as a vector.

`fc` — Center frequency

1000 (default) | real positive scalar

Center frequency, specified as a real positive scalar expressed in hertz. By default, `fc = 1000` Hz.

Output Arguments

`y` — Monopulse

vector

Monopulse of unit amplitude, returned as a vector.

`tc` — Time duration

scalar

Time duration between the maximum and minimum amplitudes of the pulse, returned as a scalar.

Tips

Default values are substituted for empty or omitted trailing input arguments.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`chirp` | `gauspuls` | `pulstran` | `rectpuls` | `tripuls`

Introduced before R2006a

goertzel

Discrete Fourier transform with second-order Goertzel algorithm

Syntax

```
dft = goertzel(data)
dft = goertzel(data,findx)
dft = goertzel(data,findx,dim)
```

Description

`dft = goertzel(data)` returns the discrete Fourier transform (DFT) of the input array `data` using a second-order Goertzel algorithm. If `data` has more than one dimension, then `goertzel` operates along the first array dimension with size greater than 1.

`dft = goertzel(data,findx)` returns the DFT for the frequency indices specified in `findx`.

`dft = goertzel(data,findx,dim)` computes the DFT along dimension `dim`. To input a dimension and use the default value of `findx`, specify the second argument as empty, `[]`.

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.

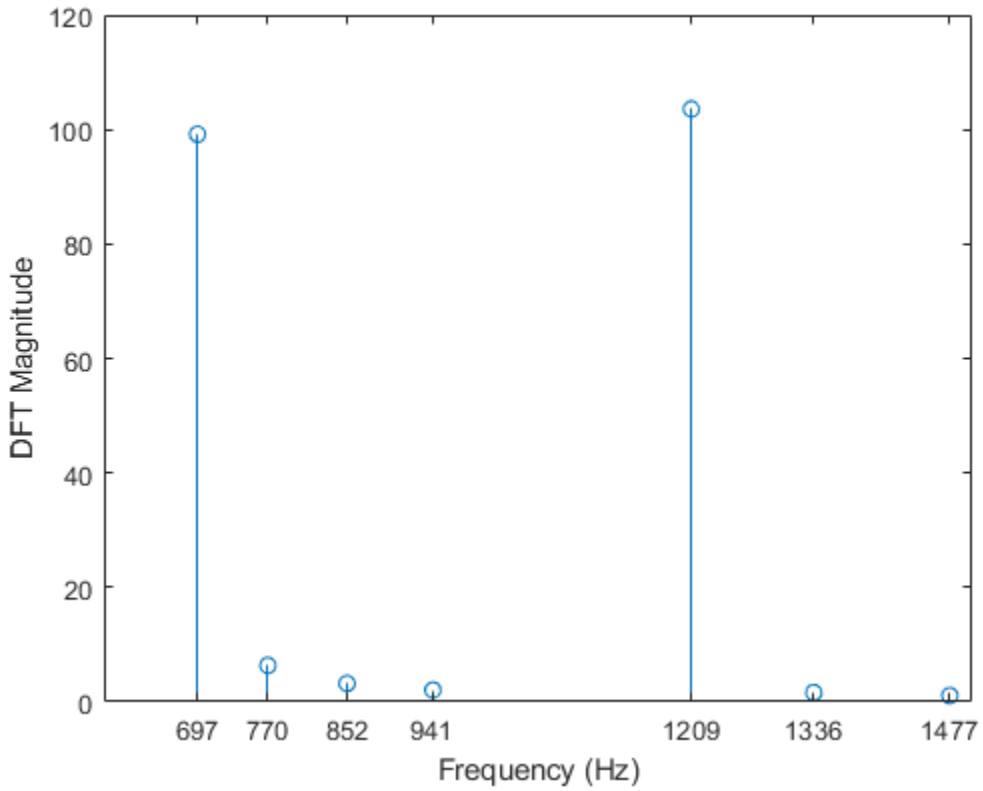
```
Fs = 8000;
N = 205;
lo = sin(2*pi*697*(0:N-1)/Fs);
hi = sin(2*pi*1209*(0:N-1)/Fs);
data = lo + hi;
```

Use the Goertzel algorithm to compute the discrete Fourier transform (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)')
ylabel('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));
```

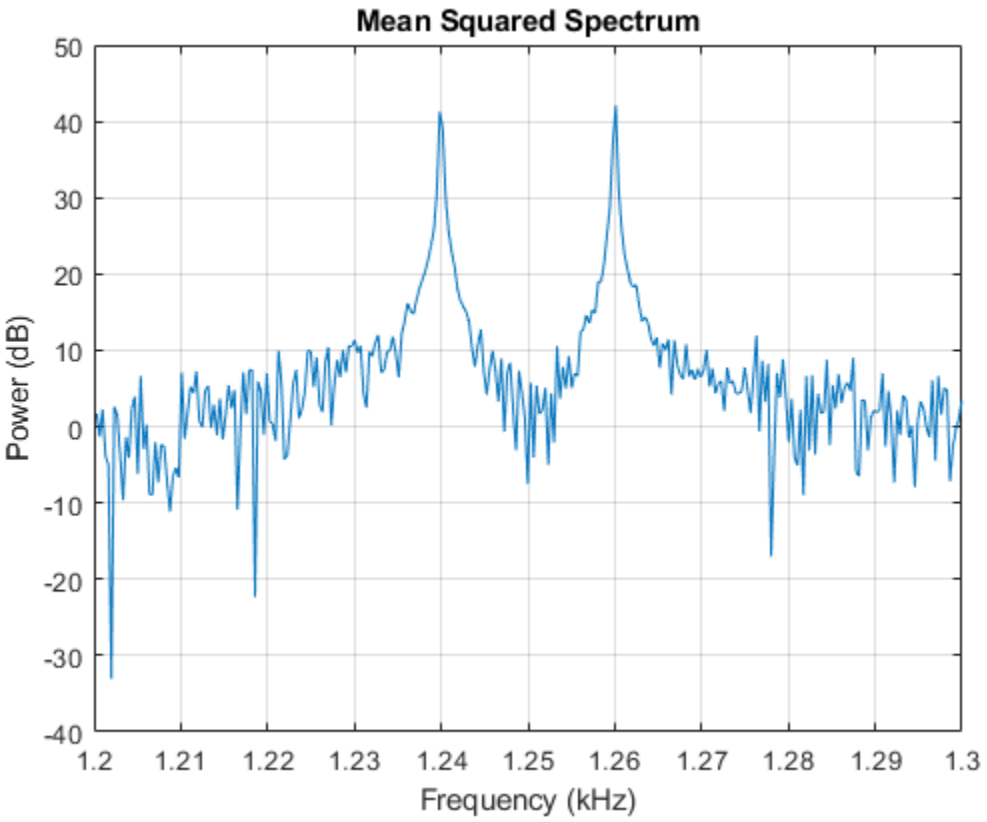
Generate the frequency vector. Use the Goertzel algorithm to compute the DFT. Restrict the range of frequencies to between 1.2 and 1.3 kHz.

```
N = (length(x)+1)/2;
f = (Fs/2)/N*(0:N-1);
indxs = find(f>1.2e3 & f<1.3e3);
X = goertzel(x,indxs);
```

Plot the mean squared spectrum expressed in decibels.

```
plot(f(indxs)/1e3,mag2db(abs(X)/length(X)))

title('Mean Squared Spectrum')
```

```
xlabel('Frequency (kHz)')
ylabel('Power (dB)')
grid
```



Discrete Fourier Transform of *N*-D Array

Generate a two-channel signal sampled at 3.2 kHz for 10 seconds and embedded in white Gaussian noise. The first channel of the signal is a 124 Hz sinusoid. The second channel is a complex exponential with a frequency of 126 Hz. Reshape the signal into a three-dimensional array such that the time axis runs along the third dimension.

```
fs = 3.2e3;
t = 0:1/fs:10-1/fs;

x = [cos(2*pi*t*124);exp(2j*pi*t*126)] + randn(2,length(t))/100;
x = permute(x,[3 1 2]);

size(x)
```

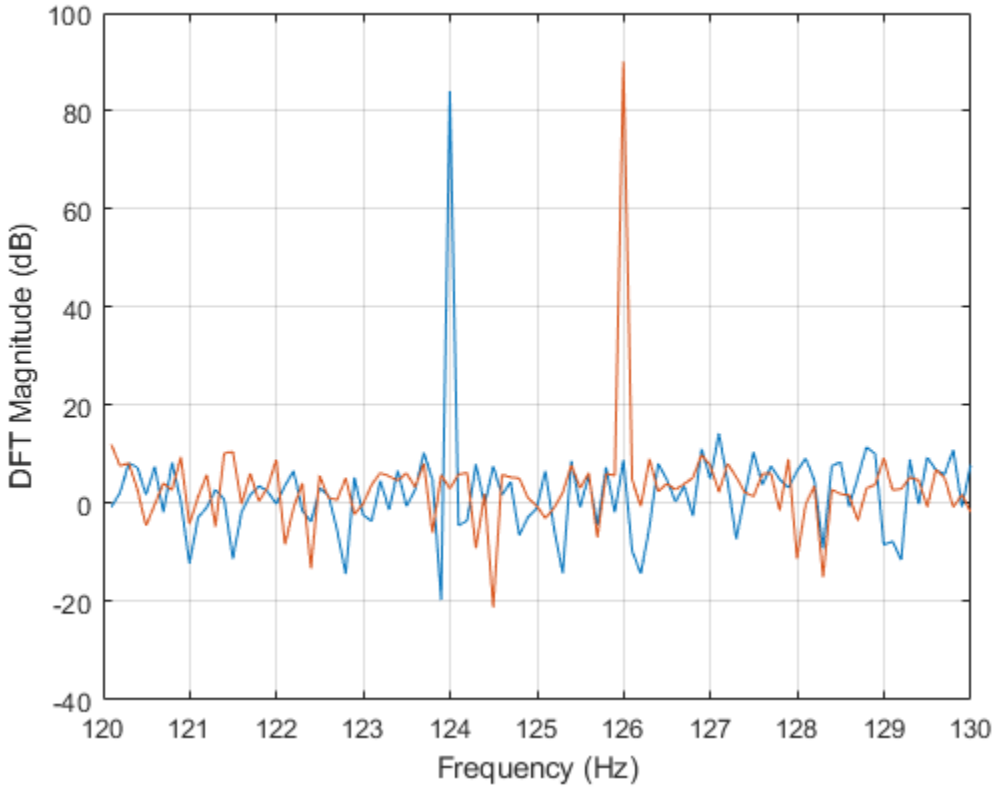
```
ans = 1×3

           1           2       32000
```

Compute the discrete Fourier transform of the signal using the Goertzel algorithm. Restrict the range of frequencies to between 120 Hz and 130 Hz.

```
N = (length(x)+1)/2;
f = (fs/2)/N*(0:N-1);
indxs = find(f>=120 & f<=130);

X = goertzel(x,indxs,3);
```

Plot the magnitude of the discrete Fourier transform expressed in decibels.

```
plot(f(indxs),mag2db(abs(squeeze(X))))
xlabel('Frequency (Hz)')
ylabel('DFT Magnitude (dB)')
grid
```



Input Arguments

data — Input array

vector | matrix | *N*-D array

Input array, specified as a vector, matrix, or *N*-D array.

Example: `sin(2*pi*(0:255)/4)` specifies a sinusoid as a row vector.

Example: `sin(2*pi*[0.1;0.3]*(0:39))'` specifies a two-channel sinusoid.

Data Types: `single` | `double`
Complex Number Support: Yes

`findx` — Frequency indices
vector

Frequency indices, specified as a vector. The indices can correspond to integer or noninteger multiples of f_s/N, where f_s is the sample rate and N is the signal length.

Data Types: `single` | `double`

`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar.

Data Types: `single` | `double`

Output Arguments

`dft` — Discrete Fourier transform
vector | matrix | *N*-D array

Discrete Fourier transform, returned as a vector, matrix, or *N*-D array.

Algorithms

The Goertzel algorithm implements the discrete Fourier transform $X(k)$ as the convolution of an N-point input $x(n)$, $n = 0, 1, \ldots, N - 1$, with the impulse response

$$h_k(n) = e^{-j2\pi k}\, e^{j2\pi kn/N}\, u(n) \equiv e^{-j2\pi k}\, W_N^{-kn}\, u(n),$$

where $u(n)$, the unit step sequence, is 1 for $n \geq 0$ and 0 otherwise. k does not have to be an integer. At a frequency $f = kf_s/N$, where f_s is the sample rate, the transform has a value

$$X(k) = y_k(n)|_{n = N},$$

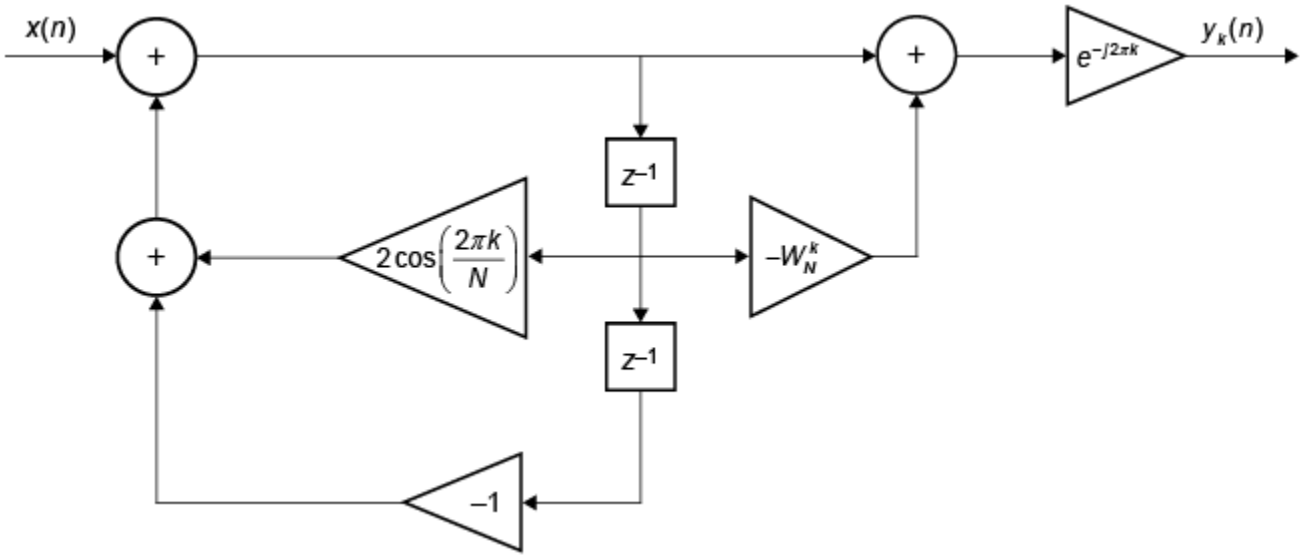
where

$$y_k(n) = \sum_{m = 0}^{N} x(m)\, h_k(n - m)$$

and $x(N) = 0$. The Z-transform of the impulse response is

$$H_k(z) = \frac{\left(1 - W_N^k z^{-1}\right) e^{-j2\pi k}}{1 - 2\cos\left(\frac{2\pi k}{N}\right) z^{-1} + z^{-2}},$$

with this direct form II implementation:



Compare the output of `goertzel` to the result of a direct implementation of the Goertzel algorithm. For the input signal, use a chirp sampled at 50 Hz for 10 seconds and embedded in white Gaussian noise. The chirp's frequency increases linearly from 15 Hz to 20 Hz during the measurement. Compute the discrete Fourier transform at a frequency that is not an integer multiple of f_s/N. When calling `goertzel`, keep in mind that MATLAB vectors run from 1 to N instead of from 0 to N – 1. The results agree to high precision.

```
fs = 50;
t = 0:1/fs:10-1/fs;
N = length(t);
xn = chirp(t,15,t(end),20)+randn(1,N)/100;

f0 = 17.36;
k = N*f0/fs;

ykn = filter([1 -exp(-2j*pi*k/N)],[1 -2*cos(2*pi*k/N) 1],[xn 0]);
Xk = exp(-2j*pi*k)*ykn(end);

dft = goertzel(xn,k+1);

df = abs(Xk-dft)

df =
   4.3634e-12
```

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. `fft` is more efficient than `goertzel` when you need to evaluate the transform at more than $\log_2 N$ frequencies, where N is the length of the input signal.
- `czt`: `czt` calculates the chirp Z-transform of an input signal on a circular or spiral contour and includes the DFT as a special case.

References

[1] Burrus, C. Sidney, and Thomas W. Parks. *DFT/FFT and Convolution Algorithms: Theory and Implementation*. New York: John Wiley & Sons, 1985.

[2] Proakis, John G., and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. 3rd Edition. Upper Saddle River, NJ: Prentice Hall, 1996.

[3] Sysel, Petr, and Pavel Rajmic. "Goertzel Algorithm Generalized to Non-Integer Multiples of Fundamental Frequency." *EURASIP Journal on Advances in Signal Processing*. Vol. 2012, Number 1, December 2012, pp. 56-1–56-8. https://doi.org/10.1186/1687-6180-2012-56.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "Automatic dimension restriction" (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- For a single-channel input, executing this function on the GPU offers no performance gains. Performance on the GPU increases as the number of channels increases.

For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

See Also

`czt` | `fft`

Introduced before R2006a

grpdelay

Average filter delay (group delay)

Syntax

```
[gd,w] = grpdelay(b,a,n)
[gd,w] = grpdelay(sos,n)
[gd,w] = grpdelay(d,n)
[gd,w] = grpdelay(___,'whole')

[gd,f] = grpdelay(___,n,fs)
[gd,f] = grpdelay(___,n,'whole',fs)

gd = grpdelay(___,win)
gd = grpdelay(___,fin,fs)

grpdelay(___)
```

Description

`[gd,w] = grpdelay(b,a,n)` returns the `n`-point group delay response vector, `gd`, and the corresponding angular frequency vector, `w`, for the digital filter with transfer function coefficients stored in `b` and `a`.

`[gd,w] = grpdelay(sos,n)` returns the `n`-point group delay response corresponding to the second-order sections matrix `sos`.

`[gd,w] = grpdelay(d,n)` returns the `n`-point group delay response for the digital filter `d`.

`[gd,w] = grpdelay(___,'whole')` returns the group delay at `n` sample points around the entire unit circle.

`[gd,f] = grpdelay(___,n,fs)` returns the group delay response vector `gd` and the corresponding physical frequency vector `f` for a digital filter designed to filter signals sampled at a rate `fs`.

`[gd,f] = grpdelay(___,n,'whole',fs)` returns the frequency vector at `n` points ranging between 0 and `fs`.

`gd = grpdelay(___,win)` returns the group delay response vector `gd` evaluated at the normalized frequencies supplied in `win`.

`gd = grpdelay(___,fin,fs)` returns the group delay response vector `gd` evaluated at the physical frequencies supplied in `fin`.

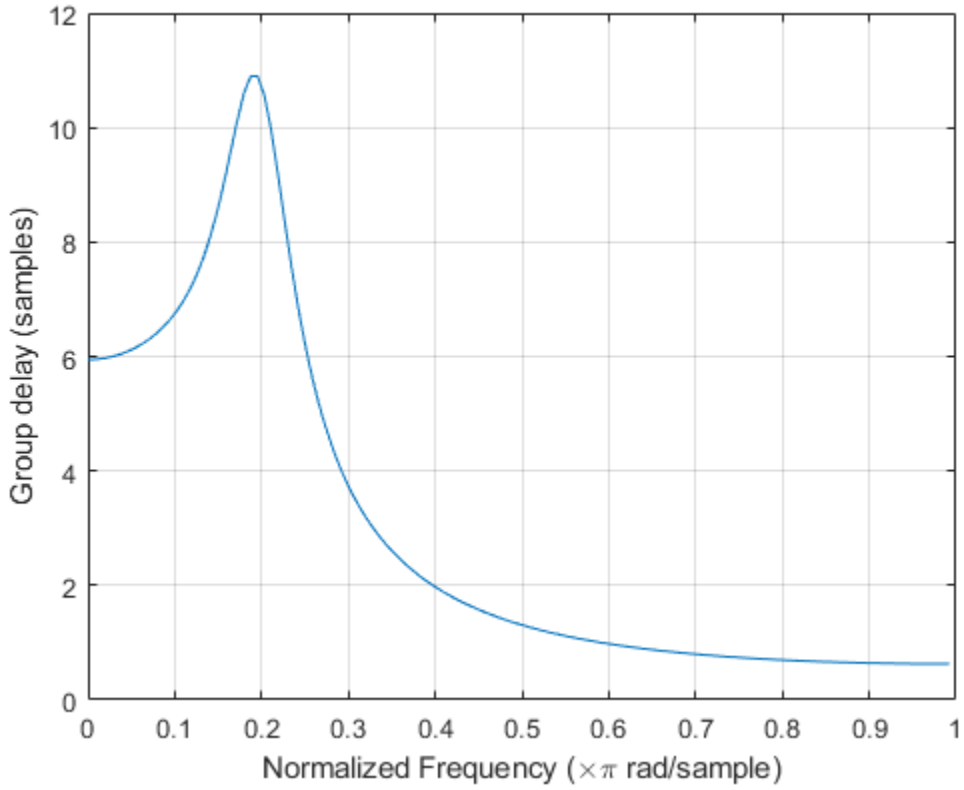
`grpdelay(___)` with no output arguments plots the group delay response of the filter.

Examples

Group Delay of a Butterworth Filter

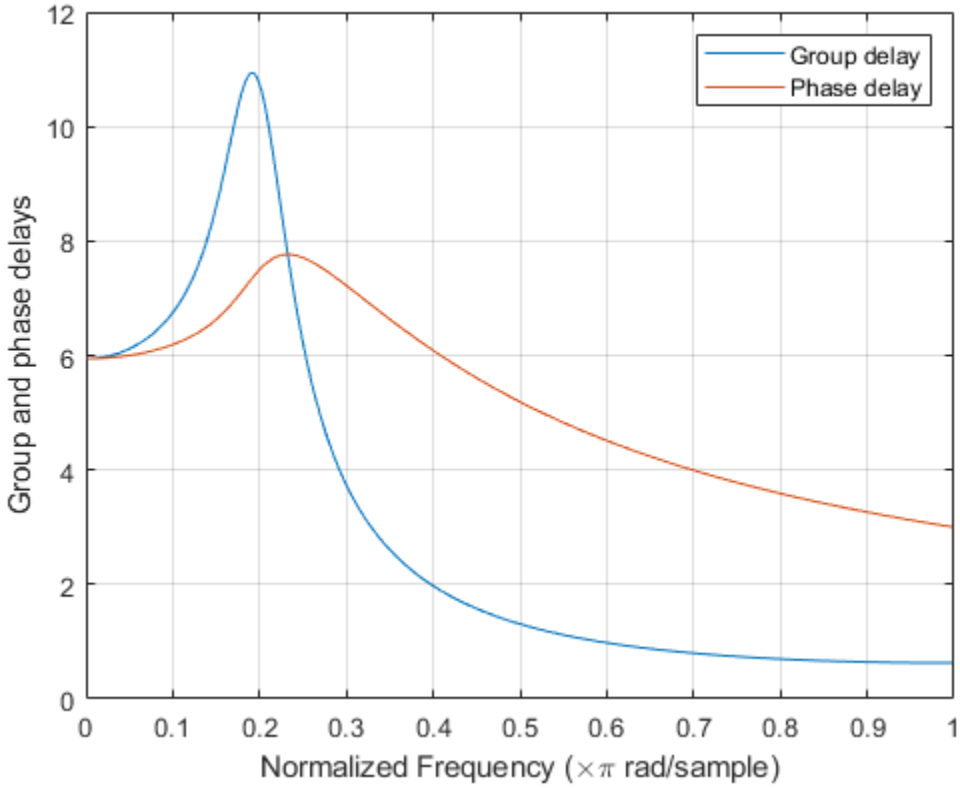
Design a Butterworth filter of order 6 with normalized 3-dB frequency 0.2π rad/sample. Use `grpdelay` to display the group delay.

```
[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.

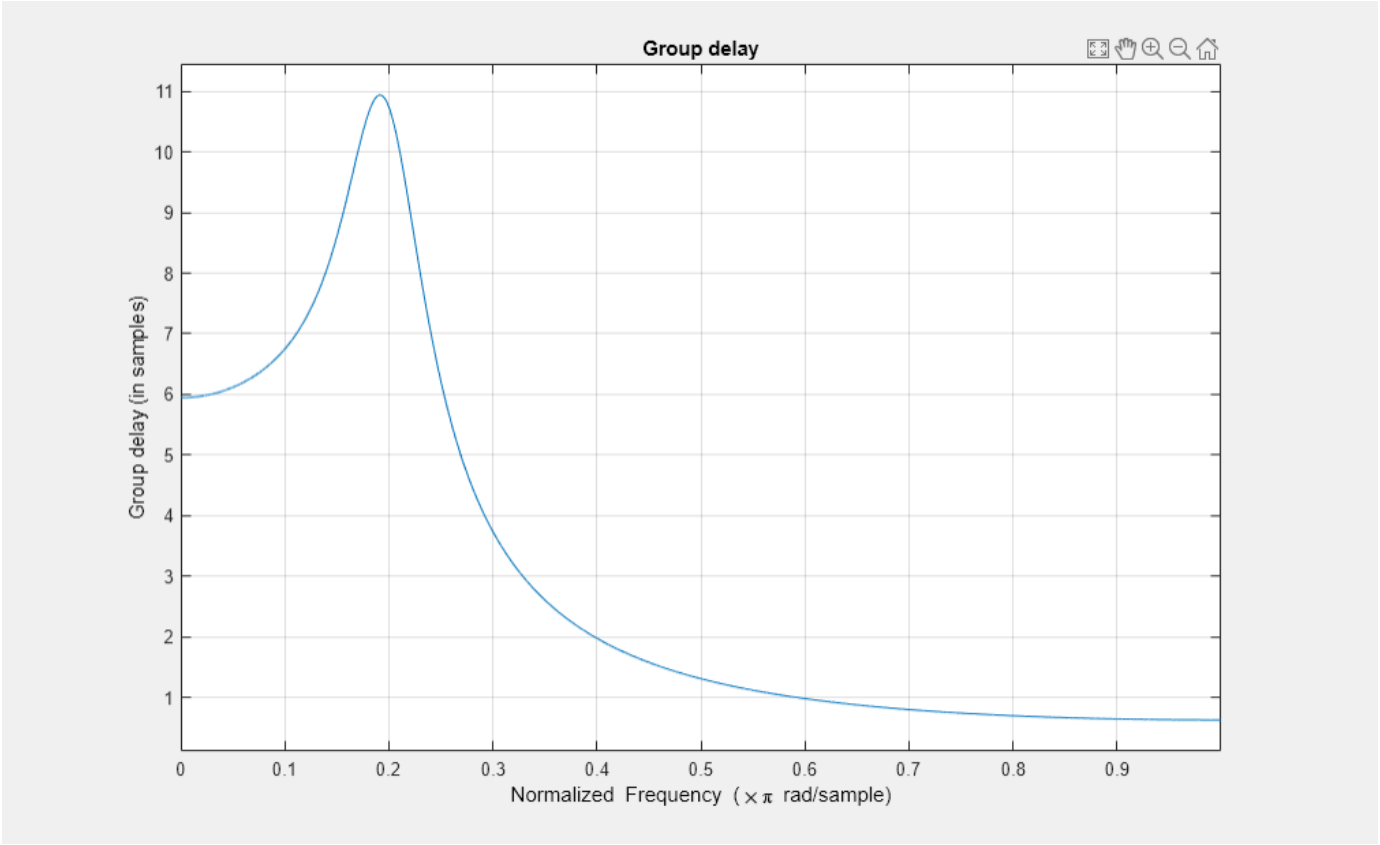
```
gd = grpdelay(sos,512);

[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 0.2π rad/sample. Display its group delay response.

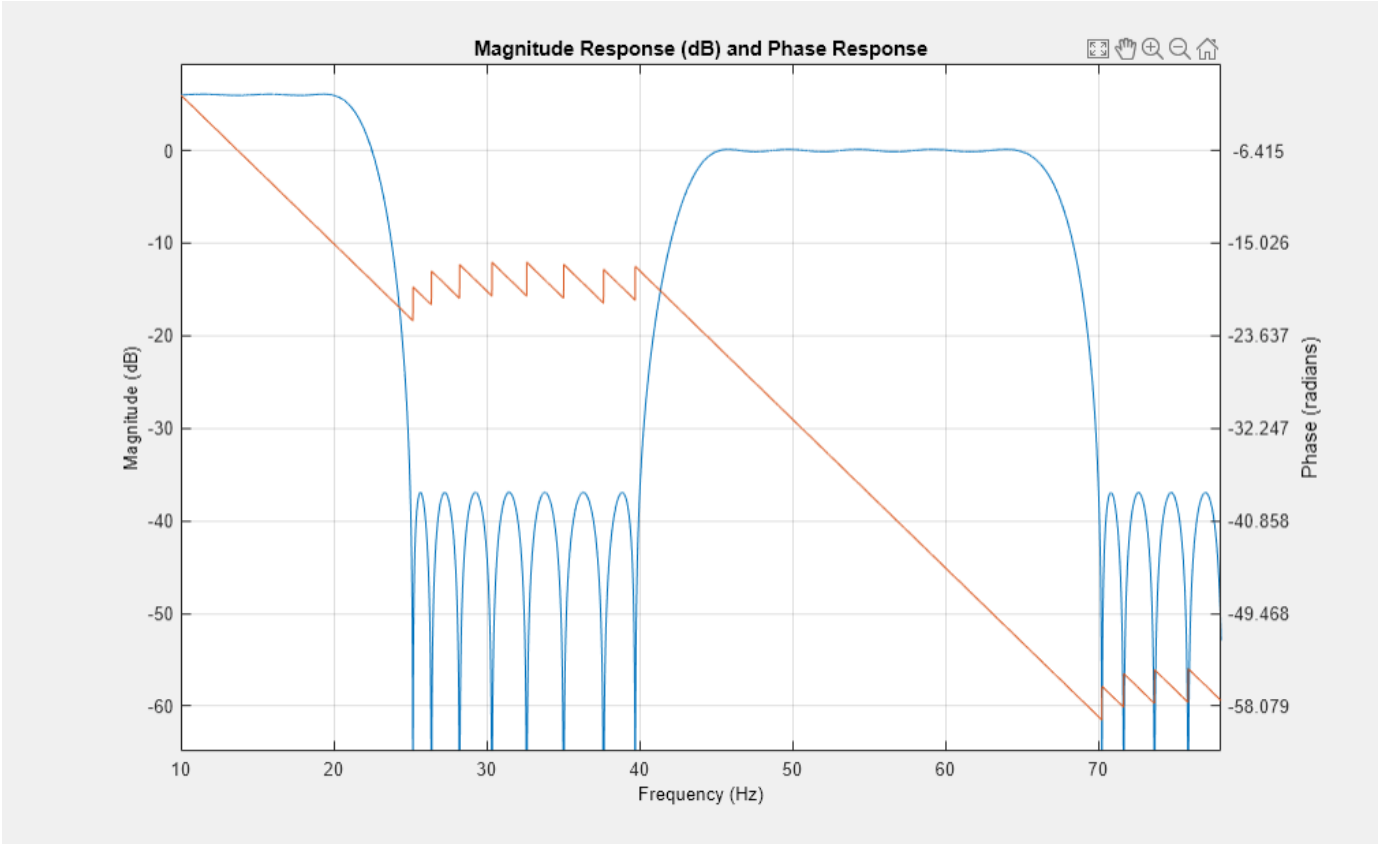
```
d = designfilt('lowpassiir','FilterOrder',6, ...
    'HalfPowerFrequency',0.2,'DesignMethod','butter');
grpdelay(d)
```



Group Delay Response of Arbitrary Magnitude Response FIR Filter

Design an 88th-order FIR filter of arbitrary magnitude response. The filter has two passbands and two stopbands. The lower-frequency passband has twice the gain of the higher-frequency passband. Specify a sample rate of 200 Hz. Visualize the magnitude response and the phase response of the filter from 10 Hz to 78 Hz.

```
fs = 200;
d = designfilt('arbmagfir', ...
       'FilterOrder',88, ...
       'NumBands',4, ...
       'BandFrequencies1',[0 20], ...
       'BandFrequencies2',[25 40], ...
       'BandFrequencies3',[45 65], ...
       'BandFrequencies4',[70 100], ...
       'BandAmplitudes1',[2 2], ...
       'BandAmplitudes2',[0 0], ...
       'BandAmplitudes3',[1 1], ...
       'BandAmplitudes4',[0 0], ...
       'SampleRate',fs);
freqz(d,10:1/fs:78,fs)
```

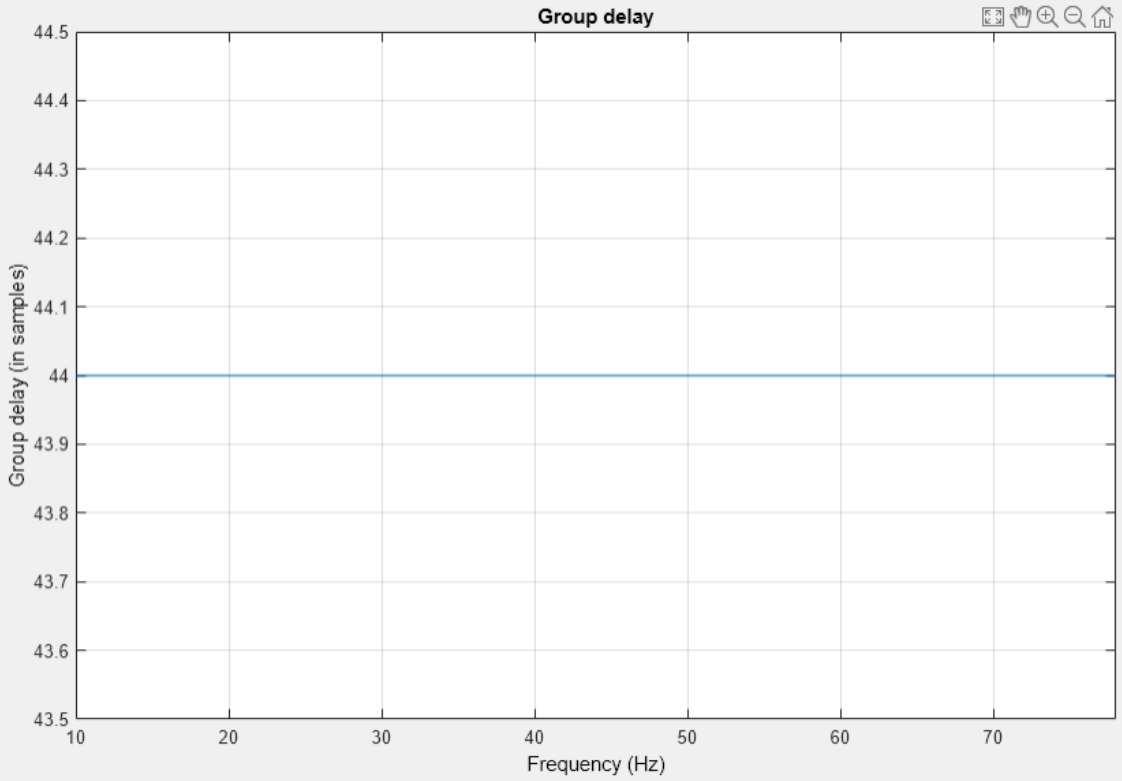


Compute and display the group delay response of the filter over the same frequency range. Verify that it is one-half of the filter order.

```
filtord(d)
```

```
ans = 88
```

```
grpdelay(d,10:1/fs:78,fs)
```



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

$$H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})} = \frac{\mathrm{b(1)}+\mathrm{b(2)}\,e^{-j\omega}+\mathrm{b(3)}\,e^{-j2\omega}+\cdots+\mathrm{b(M)}\,e^{-j(M-1)\omega}}{\mathrm{a(1)}+\mathrm{a(2)}\,e^{-j\omega}+\mathrm{a(3)}\,e^{-j2\omega}+\cdots+\mathrm{a(N)}\,e^{-j(N-1)\omega}}.$$

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 0.5π 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.

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, the function treats the input as 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 0.5π 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 0.5π rad/sample.

`fs` — Sample rate
positive scalar

Sample rate, specified as a positive scalar. When the unit of time is seconds, `fs` is expressed in hertz.

Data Types: `double`

`win` — Angular frequencies
vector

Angular frequencies, specified as a vector and expressed in rad/sample. `win` must have at least two elements, because otherwise the function interprets it as `n`. `win` = *π* corresponds to the Nyquist frequency.

`fin` — Frequencies
vector

Frequencies, specified as a vector. `fin` must have at least two elements, because otherwise the function interprets it as `n`. When the unit of time is seconds, `fin` is expressed in hertz.

Data Types: `double`

Output Arguments

`gd` — Group delay response
vector

Group delay response, returned as a vector. If you specify `n`, then `gd` has length `n`. If you do not specify `n`, or specify `n` as the empty vector, then `gd` has length 512.

If the input to `grpdelay` is single precision, the function computes the group delay using single-precision arithmetic. The output `h` is single precision.

`w` — Angular frequencies
vector

Angular frequencies, returned as a vector. `w` has values ranging from 0 to π. If you specify `'whole'` in your input, the values in `w` range from 0 to 2π. If you specify `n`, `w` has length `n`. If you do not specify `n`, or specify `n` as the empty vector, then `w` has length 512.

`f` — Frequencies
vector

Frequencies, returned as a vector expressed in hertz. `f` has values ranging from 0 to `fs`/2 Hz. If you specify `'whole'` in your input, the values in `f` range from 0 to `fs` Hz. If you specify `n`, `f` has length `n`. If you do not specify `n`, or specify `n` as the empty vector, then `f` has length 512.

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 $H(e^{j\omega})$.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The size of the first input must be a compile-time constant.

See Also
`cceps` | `designfilt` | `digitalFilter` | `fft` | `freqz` | **FVTool** | `hilbert` | `icceps` | `phasedelay` | `rceps`

Introduced before R2006a

hamming

Hamming window

Syntax

```
w = hamming(L)
w = hamming(L,sflag)
```

Description

`w = hamming(L)` returns an `L`-point symmetric Hamming window.

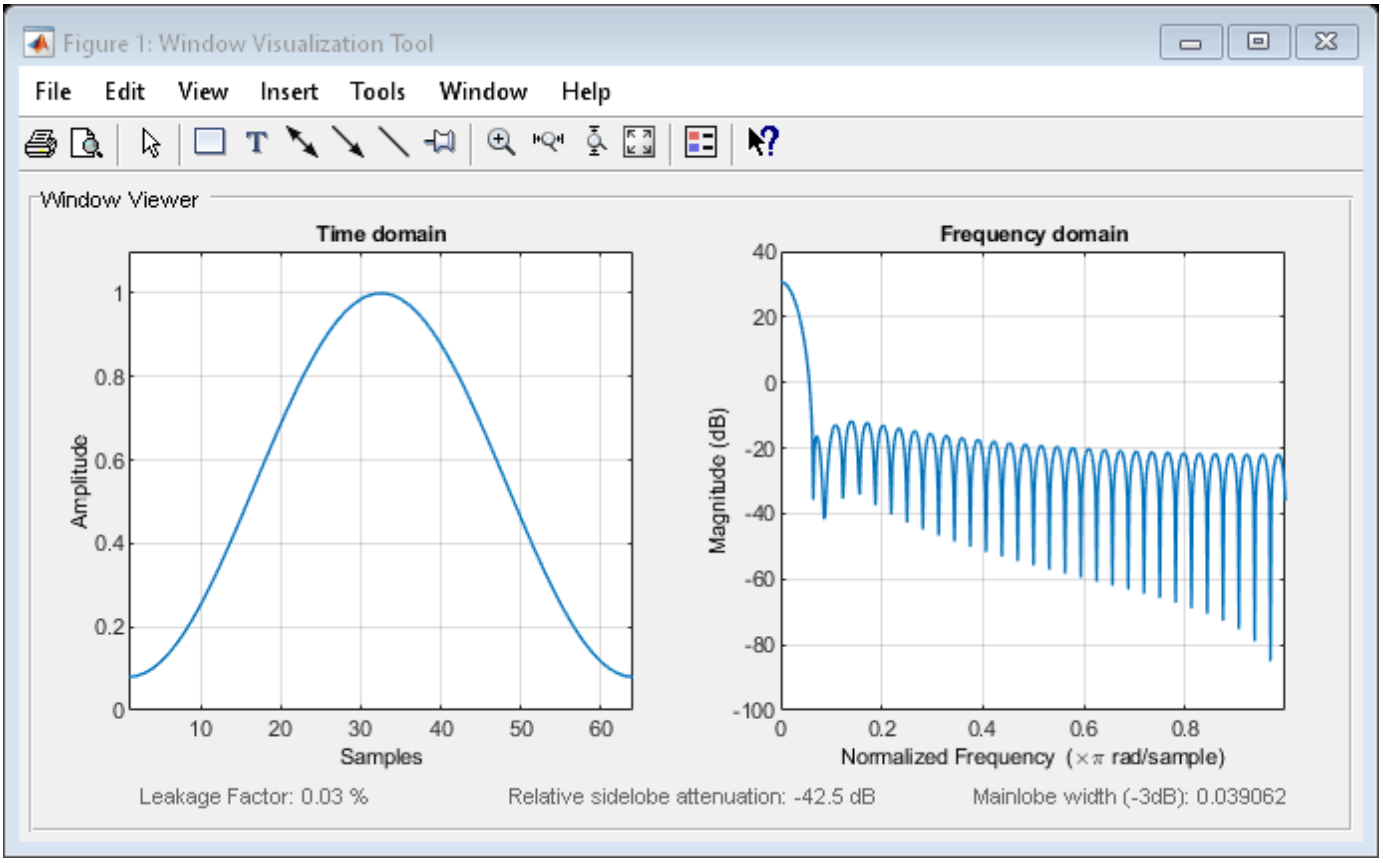
`w = hamming(L,sflag)` returns a Hamming window using the window sampling specified by `sflag`.

Examples

Hamming Window

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

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



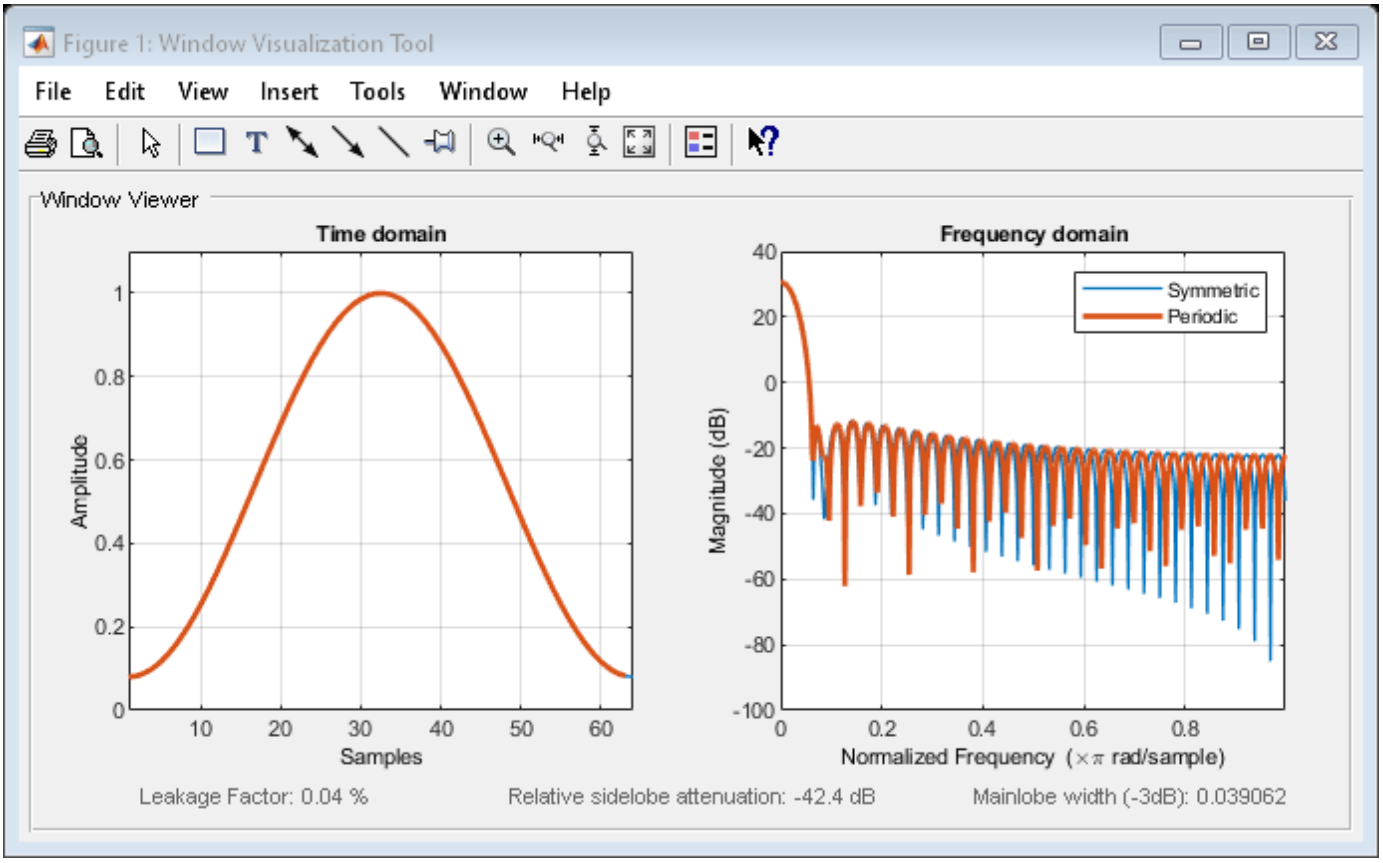
Comparison of Periodic and Symmetric Hamming Windows

Design two Hamming windows:

- The first window has $N = 64$ and is symmetric.
- The second window has $N = 63$ and is periodic.

Display the two windows.

```
Hs = hamming(64,'symmetric');
Hp = hamming(63,'periodic');
wvt = wvtool(Hs,Hp);
legend(wvt.CurrentAxes,'Symmetric','Periodic')
```



Input Arguments

`L` — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

`sflag` — Window sampling

`'symmetric'` (default) | `'periodic'`

Window sampling method, specified as:

- `'symmetric'` — Use this option when using windows for filter design.
- `'periodic'` — This option is useful for spectral analysis because it enables a windowed signal to have the perfect periodic extension implicit in the discrete Fourier transform. When `'periodic'` is specified, the function computes a window of length `L + 1` and returns the first `L` points.

Output Arguments

`w` — Hamming window

column vector

Hamming window, returned as a column vector.

Algorithms

The following equation generates the coefficients of a Hamming window:

$$w(n) = 0.54 - 0.46\cos\left(2\pi\frac{n}{N}\right),\ 0 \le n \le N\,.$$

The window length $L = N + 1$.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps

Window Designer

Functions

`blackman` | `flattopwin` | `hann` | **WVTool**

Introduced before R2006a

hampel

Outlier removal using Hampel identifier

Syntax

```
y = hampel(x)
y = hampel(x,k)
y = hampel(x,k,nsigma)

[y,j] = hampel( ___ )
[y,j,xmedian,xsigma] = hampel( ___ )

hampel( ___ )
```

Description

`y = hampel(x)` applies a Hampel filter to the input vector, `x`, to detect and remove outliers. For each sample of `x`, the function computes the median of a window composed of the sample and its six surrounding samples, three per side. It also estimates the standard deviation of each sample about its window median using the median absolute deviation. If a sample differs from the median by more than three standard deviations, it is replaced with the median. If `x` is a matrix, then `hampel` treats each column of `x` as an independent channel.

`y = hampel(x,k)` specifies the number of neighbors, `k`, on either side of each sample of `x` in the measurement window. `k` defaults to 3.

`y = hampel(x,k,nsigma)` specifies a number of standard deviations, `nsigma`, by which a sample of `x` must differ from the local median for it to be replaced with the median. `nsigma` defaults to 3.

`[y,j] = hampel( ___ )` also returns a logical matrix that is true at the locations of all points identified as outliers. This syntax accepts any of the input arguments from previous syntaxes.

`[y,j,xmedian,xsigma] = hampel( ___ )` also returns the local medians and the estimated standard deviations for each element of `x`.

`hampel( ___ )` with no output arguments plots the filtered signal and annotates the outliers that were removed.

Examples

Remove Spikes from Sinusoid

Generate 100 samples of a sinusoidal signal. Replace the sixth and twentieth samples with spikes.

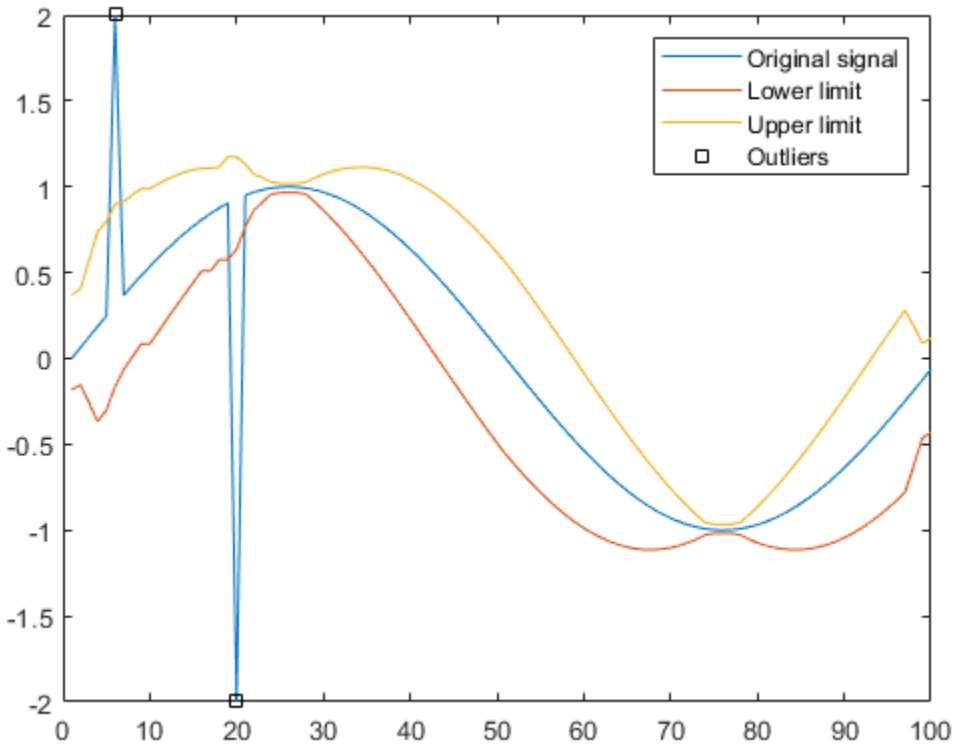
```
x = sin(2*pi*(0:99)/100);
x(6) = 2;
x(20) = -2;
```

Use `hampel` to locate every sample that differs by more than three standard deviations from the local median. The measurement window is composed of the sample and its six surrounding samples, three per side.

```
[y,i,xmedian,xsigma] = hampel(x);
```

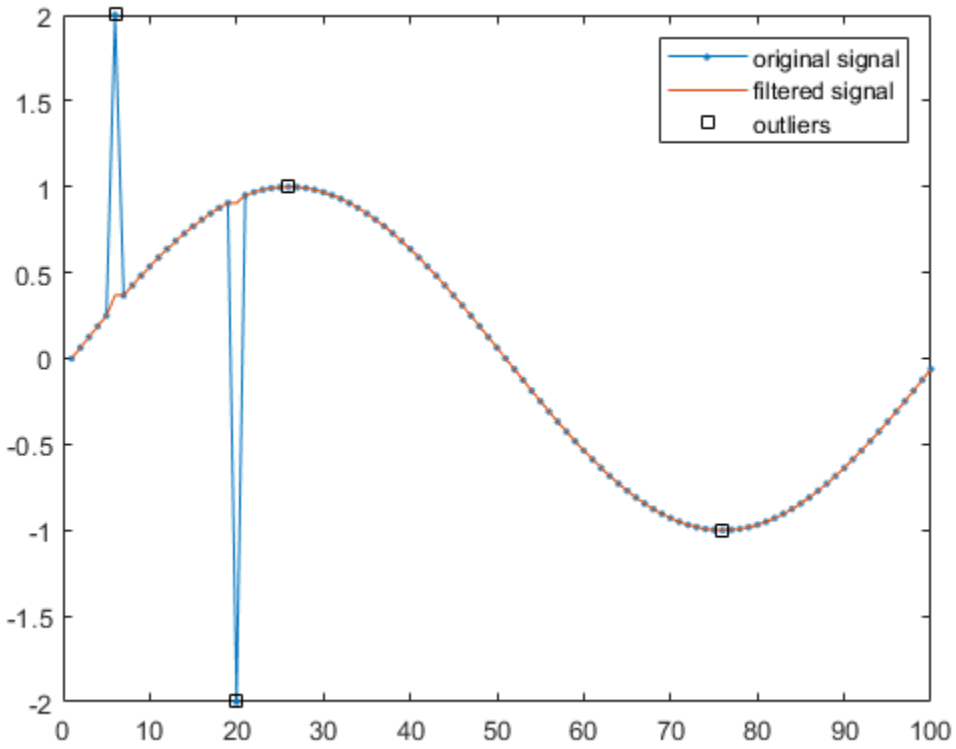
Plot the filtered signal and annotate the outliers.

```
n = 1:length(x);
plot(n,x)
hold on
plot(n,xmedian-3*xsigma,n,xmedian+3*xsigma)
plot(find(i),x(i),'sk')
hold off
legend('Original signal','Lower limit','Upper limit','Outliers')
```



Repeat the computation, but now take just one adjacent sample on each side when computing the median. The function considers the extrema as outliers.

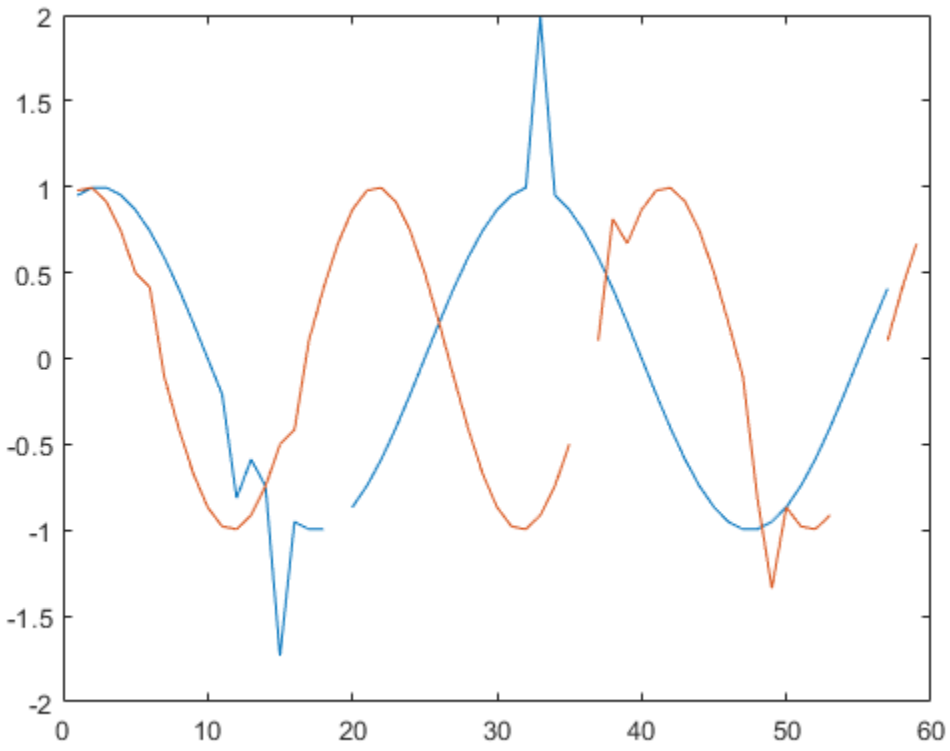
```
hampel(x,1)
```



Hampel Filtering of Multichannel Signal

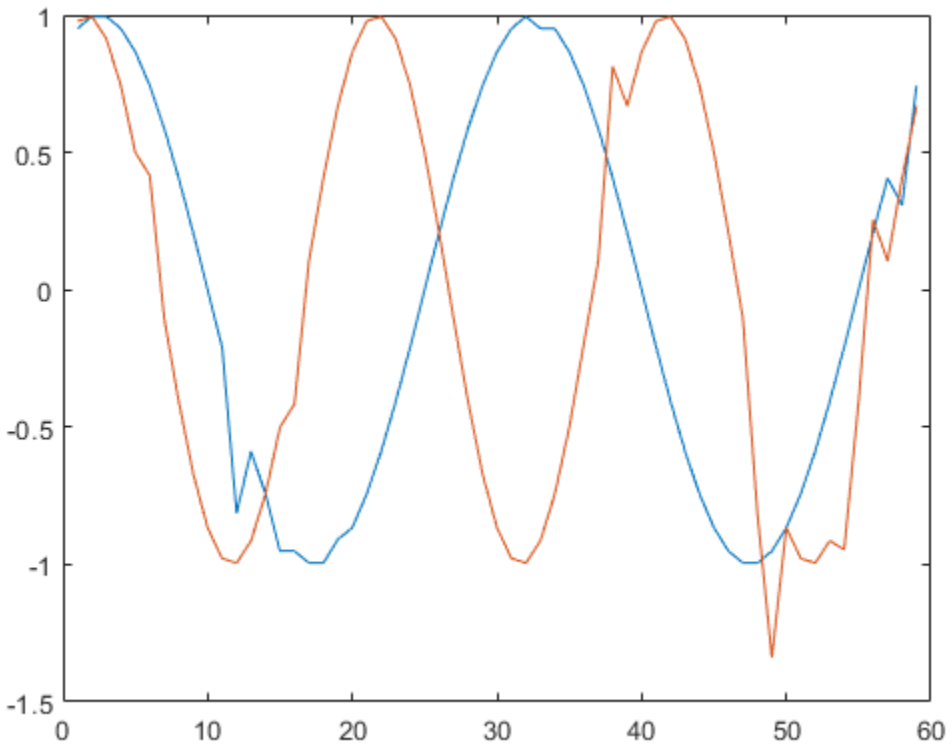
Generate a two-channel signal consisting of sinusoids of different frequencies. Place spikes in random places. Use NaNs to add missing samples at random. Reset the random number generator for reproducible results. Plot the signal.

```
rng('default')

n = 59;
x = sin(pi./[15 10]'*(1:n)+pi/3)';

spk = randi(2*n,9,1);
x(spk) = x(spk)*2;
x(randi(2*n,6,1)) = NaN;

plot(x)
```



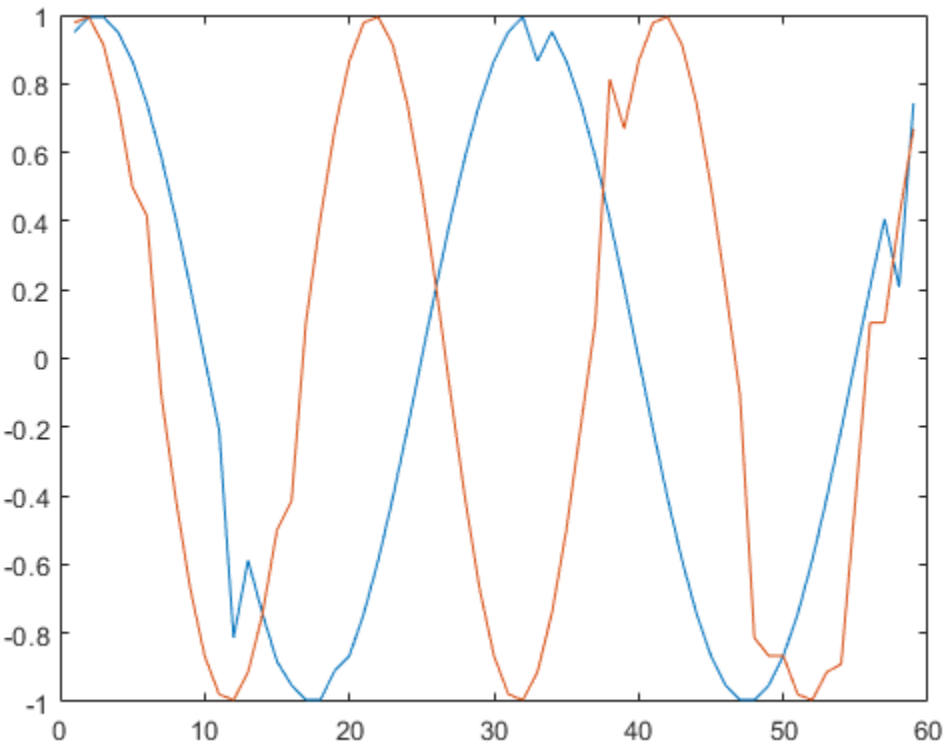
Filter the signal using `hampel` with the default settings.

```
y = hampel(x);
plot(y)
```



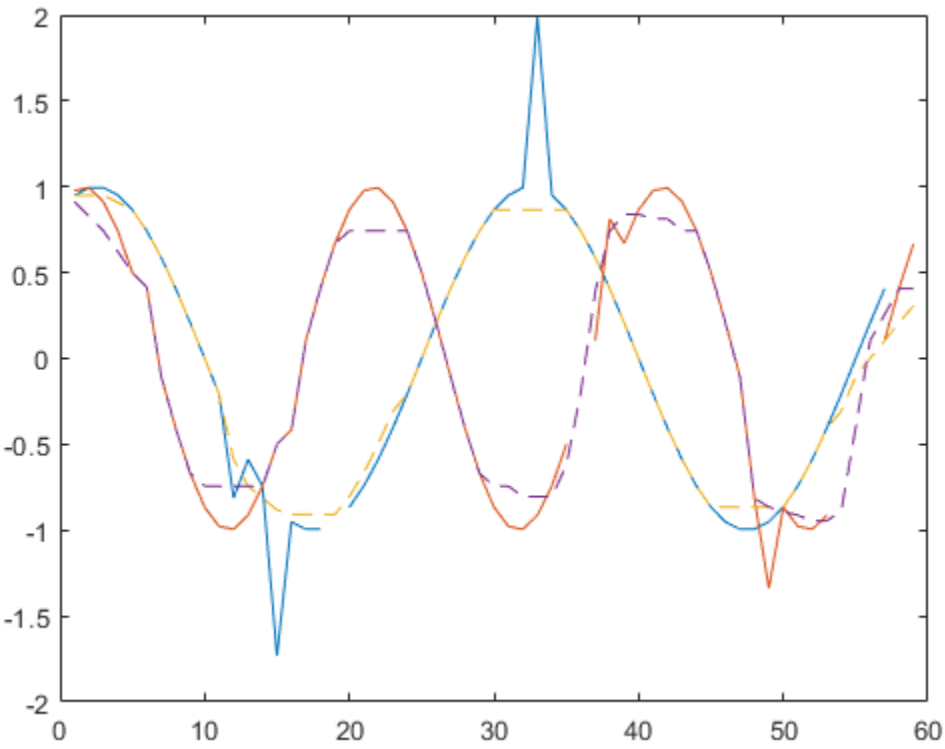
Increase the length of the moving window and decrease the threshold to treat a sample as an outlier.

```
y = hampel(x,4,2);
plot(y)
```



Output the running median for each channel. Overlay the medians on a plot of the signal.

```
[y,j,xmd,xsd] = hampel(x,4,2);
plot(x)
hold on
plot(xmd,'--')
```



Find Outliers in Multichannel Signal

Generate a multichannel signal that consists of two sinusoids of different frequencies embedded in white Gaussian noise of unit variance.

```
rng('default')

t = 0:60;
x = sin(pi./[10;2]*t)'+randn(numel(t),2);
```

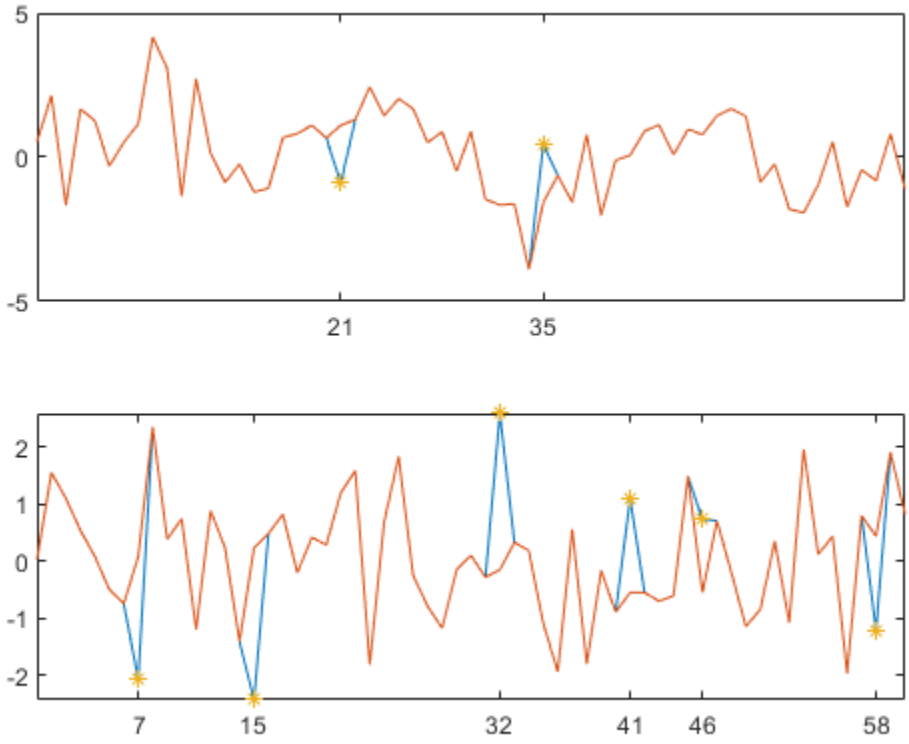
Apply a Hampel filter to the signal. Take as outliers those points that differ by more than two standard deviations from the median of a surrounding nine-sample window. Output a logical matrix that is true at the locations of the outliers.

```
k = 4;
nsig = 2;

[y,h] = hampel(x,k,nsig);
```

Plot each channel of the signal in its own set of axes. Draw the original signal, the filtered signal, and the outliers. Annotate the outlier locations.

```
for k = 1:2
    hk = h(:,k);
    ax = subplot(2,1,k);
```

```
    plot(t,x(:,k))
    hold on
    plot(t,y(:,k))
    plot(t(hk),x(hk,k),'*')
    hold off
    ax.XTick = t(hk);
end
```



Signal Statistics Returned by Hampel Filter

Generate 100 samples of a sinusoidal signal. Replace the sixth and twentieth samples with spikes.

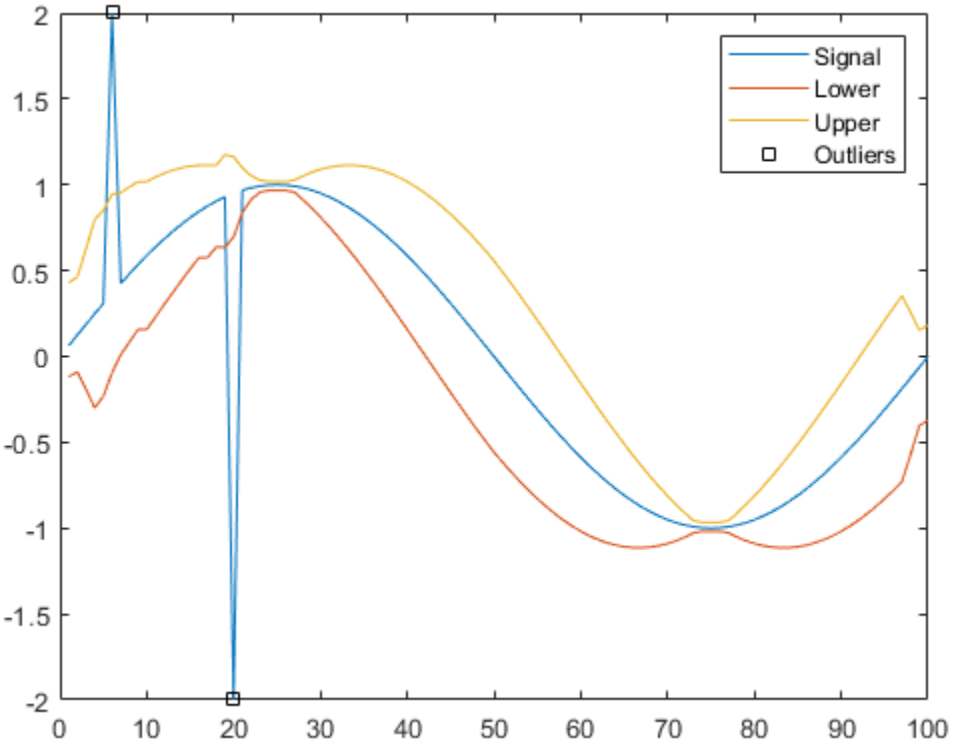
```
n = 1:100;
x = sin(2*pi*n/100);
x(6) = 2;
x(20) = -2;
```

Use `hampel` to compute the local median and estimated standard deviation for every sample. Use the default values of the input parameters:

- The window size is 2 × 3 + 1 = 7.
- The points that differ from their window median by more than three standard deviations are considered outliers.

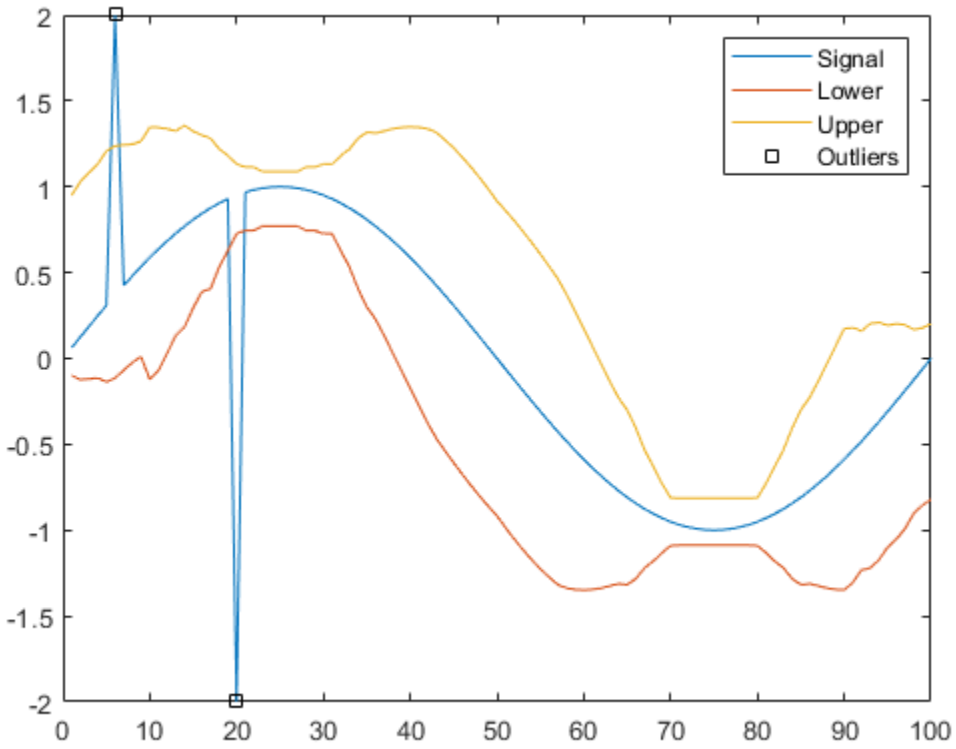
Plot the result.

```
[y,i,xmedian,xsigma] = hampel(x);

plot(n,x)
hold on
plot(n,[1;1]*xmedian+3*[-1;1]*xsigma)
plot(find(i),x(i),'sk')
hold off
legend('Signal','Lower','Upper','Outliers')
```



Repeat the calculation using a window size of 2 × 10 + 1 = 21 and two standard deviations as the criteria for identifying outliers.

```
sds = 2;
adj = 10;
[y,i,xmedian,xsigma] = hampel(x,adj,sds);

plot(n,x)
hold on
plot(n,[1;1]*xmedian+sds*[-1;1]*xsigma)
plot(find(i),x(i),'sk')
hold off
legend('Signal','Lower','Upper','Outliers')
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If `x` is a matrix, then `hampel` treats each column of `x` as an independent channel.

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

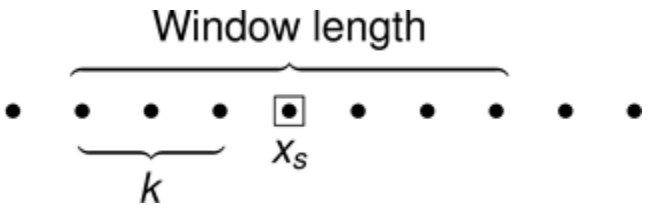
Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal.

Data Types: `single` | `double`

k — Number of neighbors on either side

3 (default) | integer scalar

Number of neighbors on either side of the sample x_s, specified as an integer scalar. Samples close to the signal edges that have fewer than `k` samples on one side are compared to the median of a smaller window.



Data Types: `single` | `double`

`nsigma` — Number of standard deviations
3 (default) | real scalar

Number of standard deviations by which a sample of `x` must differ from its local median to be considered an outlier. Specify `nsigma` as a real scalar. The function estimates the standard deviation by scaling the local median absolute deviation (MAD) by a factor of $\kappa = \frac{1}{\sqrt{2}\text{erf}^{-1}(1/2)} \approx 1.4826$.

Data Types: `single` | `double`

Output Arguments

`y` — Filtered signal
vector | matrix

Filtered signal, returned as a vector or matrix of the same size as `x`.

Data Types: `single` | `double`

`j` — Outlier index
vector | matrix

Outlier index, returned as a vector or matrix of the same size as `x`.

Data Types: `logical`

`xmedian` — Local medians
vector | matrix

Local medians, returned as a vector or matrix of the same size as `x`.

Data Types: `single` | `double`

`xsigma` — Estimated standard deviations
vector | matrix

Estimated standard deviations, returned as a vector or matrix of the same size as `x`.

Data Types: `single` | `double`

More About

Hampel Identifier

The Hampel identifier is a variation of the three-sigma rule of statistics that is robust against outliers.

Given a sequence $x_1, x_2, x_3, \ldots, x_n$ and a sliding window of length k, define point-to-point median and standard-deviation estimates using:

- Local median — $m_i = \text{median}(x_{i-k}, x_{i-k+1}, x_{i-k+2}, \ldots, x_i, \ldots, x_{i+k-2}, x_{i+k-1}, x_{i+k})$
- Standard deviation — $\sigma_i = \kappa\text{median}(|x_{i-k} - m_i|, \ldots, |x_{i+k} - m_i|)$, where $\kappa = \frac{1}{\sqrt{2}\text{erf}^{-1}(1/2)} \approx 1.4826$

The quantity σ_i / κ is known as the median absolute deviation (MAD).

If a sample x_i is such that

$$|x_i - m_i| > n_\sigma \sigma_i$$

for a given threshold n_σ, then the Hampel identifier declares x_i an outlier and replaces it with m_i.

Near the sequence endpoints, the function truncates the window used to compute m_i and σ_i:

- $i < k + 1$

  $m_i = \text{median}(x_1, x_2, x_3, \ldots, x_i, \ldots, x_{i+k-2}, x_{i+k-1}, x_{i+k})$

  $\sigma_i = \kappa\text{median}(|x_1 - m_1|, \ldots, |x_{i+k} - m_i|)$
- $i > n - k$

  $m_i = \text{median}(x_{i-k}, x_{i-k+1}, x_{i-k+2}, \ldots, x_i, \ldots, x_{n-2}, x_{n-1}, x_n)$

  $\sigma_i = \kappa\text{median}(|x_{i-k} - m_i|, \ldots, |x_n - m_n|)$

References

[1] Liu, Hancong, Sirish Shah, and Wei Jiang. “On-line outlier detection and data cleaning.” *Computers and Chemical Engineering*. Vol. 28, March 2004, pp. 1635–1647.

[2] Suomela, Jukka. “Median Filtering Is Equivalent to Sorting.” 2014.

See Also

`medfilt1` | `median` | `filloutliers` | `filter` | `isoutlier` | `mad` | `movmad` | `movmedian` | `sgolayfilt`

Topics

“Eliminate Outliers Using Hampel Identifier”

Introduced in R2015b

hann

Hann (Hanning) window

Syntax

```
w = hann(L)
w = hann(L,sflag)
```

Description

`w = hann(L)` returns an L-point symmetric Hann window.

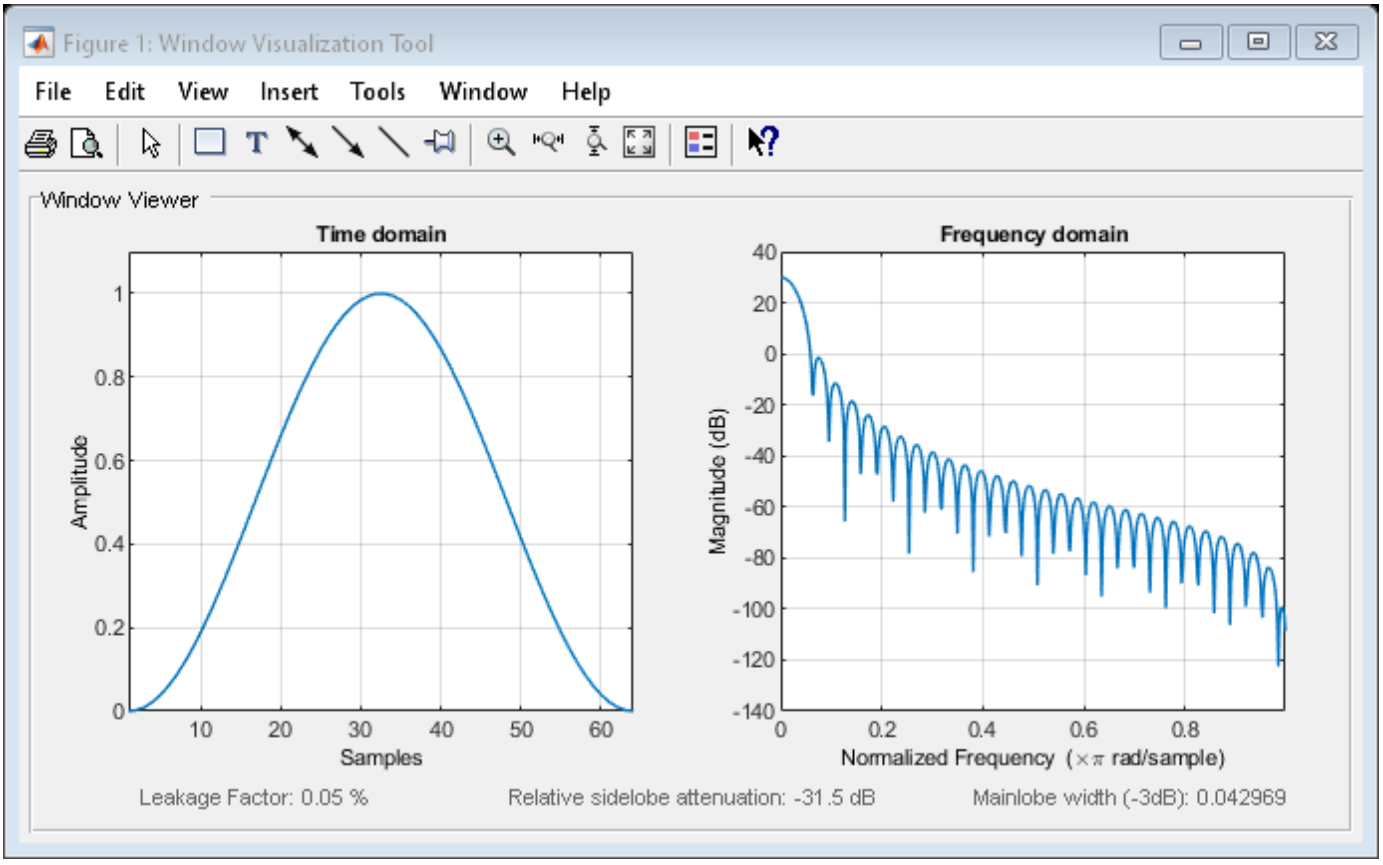
`w = hann(L,sflag)` returns a Hann window using the window sampling specified by `sflag`.

Examples

Hann Window

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

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



Input Arguments

`L` — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

`sflag` — Window sampling

`'symmetric'` (default) | `'periodic'`

Window sampling, specified as one of the following:

- `'symmetric'` — Use this option when using windows for filter design.
- `'periodic'` — This option is useful for spectral analysis because it enables a windowed signal to have the perfect periodic extension implicit in the discrete Fourier transform. When `'periodic'` is specified, `hann` computes a window of length $L + 1$ and returns the first L points.

Output Arguments

`w` — Hann window

column vector

Hann window, returned as a column vector.

Algorithms

The following equation generates the coefficients of a Hann window:

$$w(n) = 0.5\left(1 - \cos\left(2\pi\frac{n}{N}\right)\right),\ 0 \le n \le N\,.$$

The window length $L = N + 1$.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps
Window Designer

Functions
`blackman` | `flattopwin` | `hamming` | **WVTool**

Introduced before R2006a

hht

Hilbert-Huang transform

Syntax

```
hs = hht(imf)
hs = hht(imf,fs)
[hs,f,t] = hht( ___ )
[hs,f,t,imfinsf,imfinse] = hht( ___ )
[ ___ ] = hht( ___ ,Name,Value)

hht( ___ )
hht( ___ ,freqlocation)
```

Description

`hs = hht(imf)` returns the Hilbert spectrum `hs` of the signal specified by intrinsic mode functions `imf`. `hs` is useful for analyzing signals that comprise a mixture of signals whose spectral content changes in time. Use `hht` to perform Hilbert spectral analysis on signals to identify localized features.

`hs = hht(imf,fs)` returns the Hilbert spectrum `hs` of a signal sampled at a rate `fs`.

`[hs,f,t] = hht( ___ )` returns frequency vector `f` and time vector `t` in addition to `hs`. These output arguments can be used with either of the previous input syntaxes.

`[hs,f,t,imfinsf,imfinse] = hht( ___ )` also returns the instantaneous frequencies `imfinsf` and the instantaneous energies `imfinse` of the intrinsic mode functions for signal diagnostics.

`[ ___ ] = hht( ___ ,Name,Value)` estimates Hilbert spectrum parameters with additional options specified by one or more `Name,Value` pair arguments.

`hht( ___ )` with no output arguments plots the Hilbert spectrum in the current figure window. You can use this syntax with any of the input arguments in previous syntaxes.

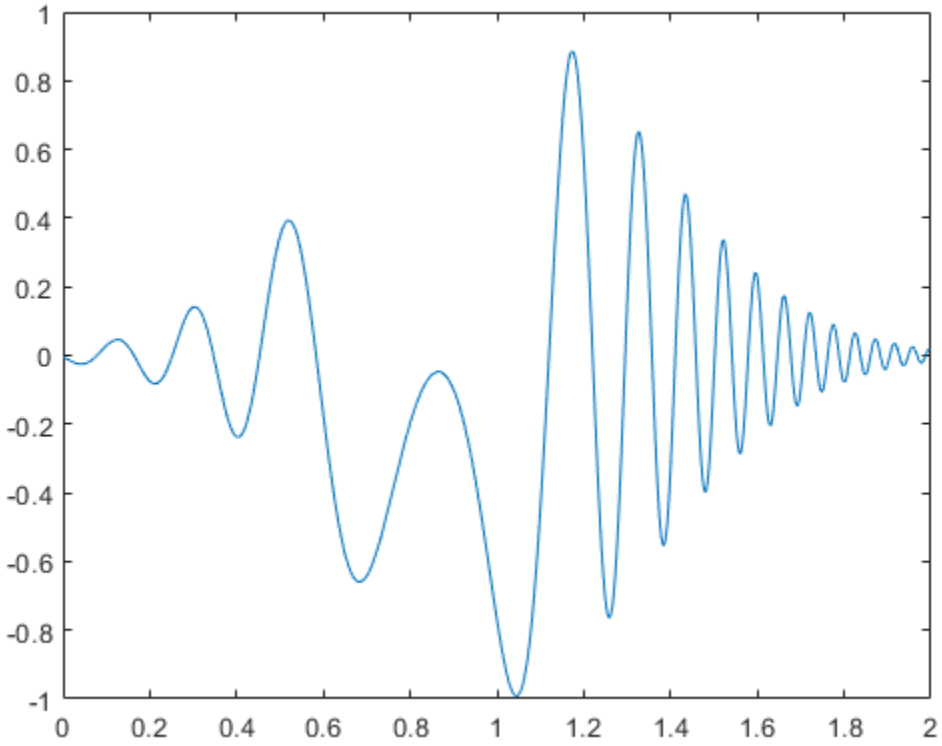
`hht( ___ ,freqlocation)` plots the Hilbert spectrum with the optional `freqlocation` argument to specify the location of the frequency axis. Frequency is represented on the *y*-axis by default.

Examples

Hilbert Spectrum of Quadratic Chirp

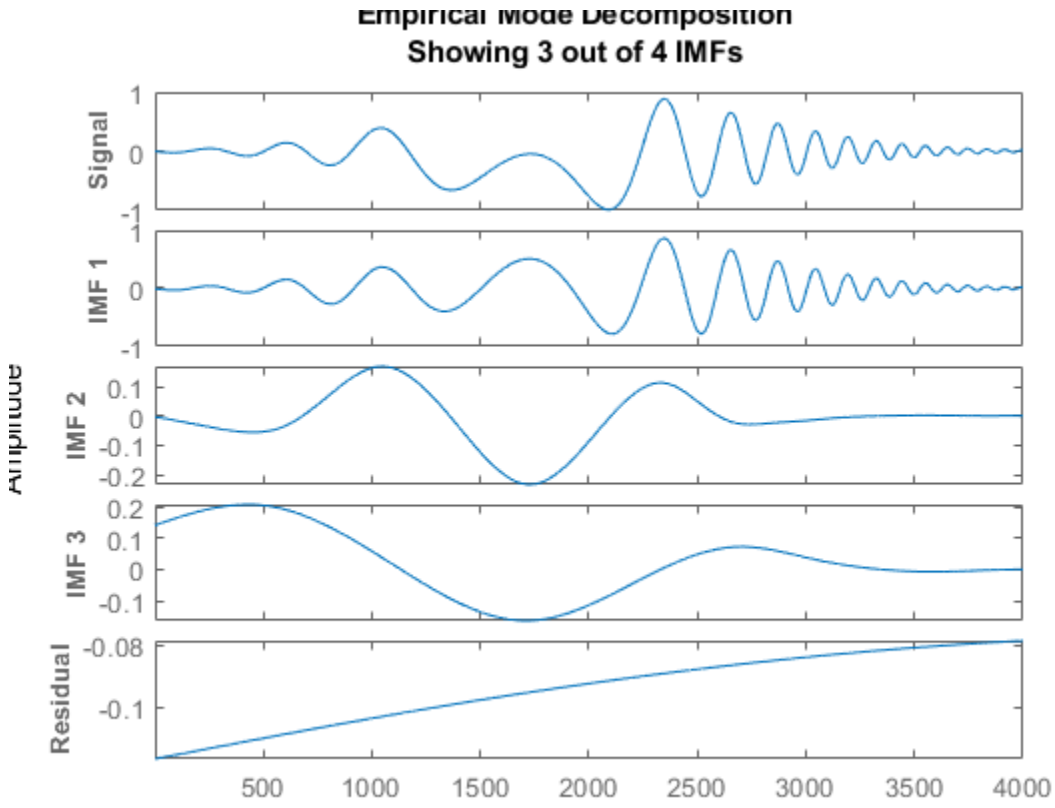
Generate a Gaussian-modulated quadratic chirp. Specify a sample rate of 2 kHz and a signal duration of 2 seconds.

```
fs = 2000;
t = 0:1/fs:2-1/fs;
q = chirp(t-2,4,1/2,6,'quadratic',100,'convex').*exp(-4*(t-1).^2);
plot(t,q)
```



Use `emd` to visualize the intrinsic mode functions (IMFs) and the residual.

```
emd(q)
```



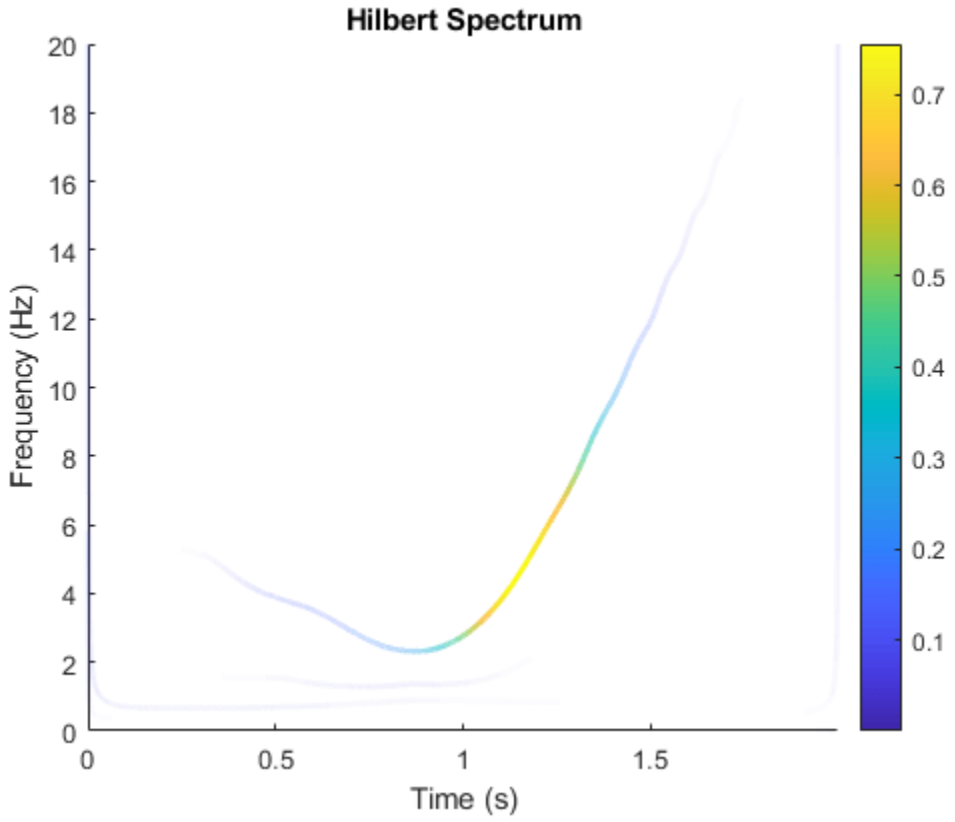
Compute the IMFs of the signal. Use the `'Display'` name-value pair to output a table showing the number of sifting iterations, the relative tolerance, and the sifting stop criterion for each IMF.

```
imf = emd(q,'Display',1);
```

```
Current IMF  |  #Sift Iter  |  Relative Tol  |  Stop Criterion Hit
      1      |        2     |    0.0063952   |  SiftMaxRelativeTolerance
      2      |        2     |       0.1007   |  SiftMaxRelativeTolerance
      3      |        2     |      0.01189   |  SiftMaxRelativeTolerance
      4      |        2     |    0.0075124   |  SiftMaxRelativeTolerance
Decomposition stopped because the number of extrema in the residual signal is less than the 'MaxI
```

Use the computed IMFs to plot the Hilbert spectrum of the quadratic chirp. Restrict the frequency range from 0 Hz to 20 Hz.

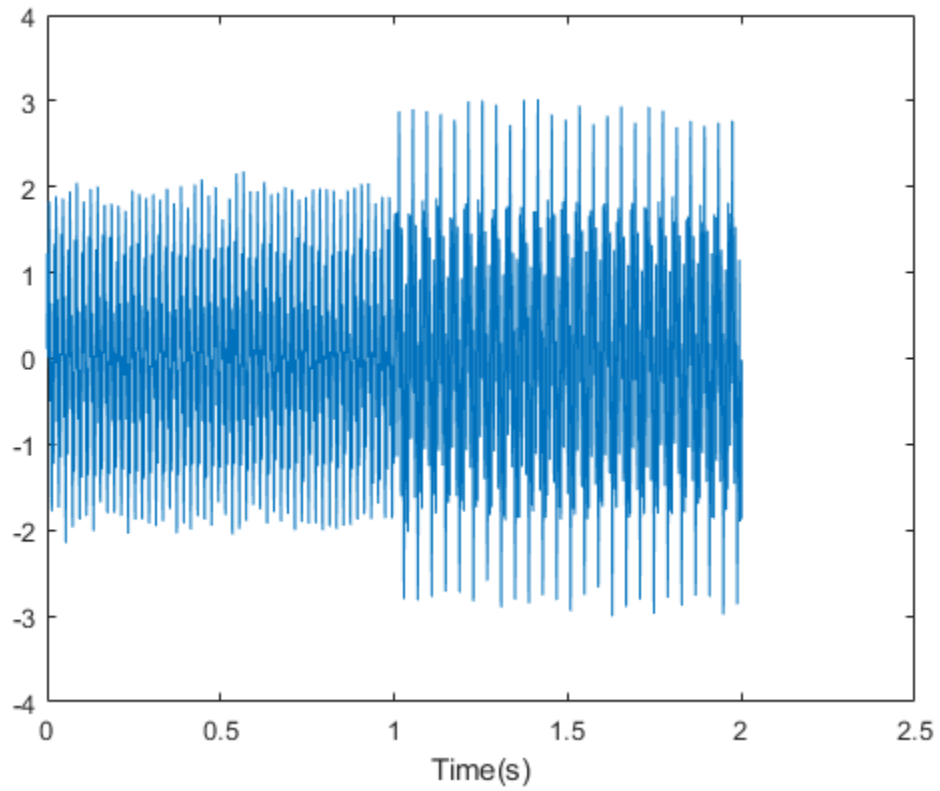
```
hht(imf,fs,'FrequencyLimits',[0 20])
```



Perform Empirical Mode Decomposition and Visualize Hilbert Spectrum of Signal

Load and visualize a nonstationary continuous signal composed of sinusoidal waves with a distinct change in frequency. The vibration of a jackhammer and the sound of fireworks are examples of nonstationary continuous signals. The signal is sampled at a rate `fs`.

```
load('sinusoidalSignalExampleData.mat','X','fs')
t = (0:length(X)-1)/fs;
plot(t,X)
xlabel('Time(s)')
```



The mixed signal contains sinusoidal waves with different amplitude and frequency values.

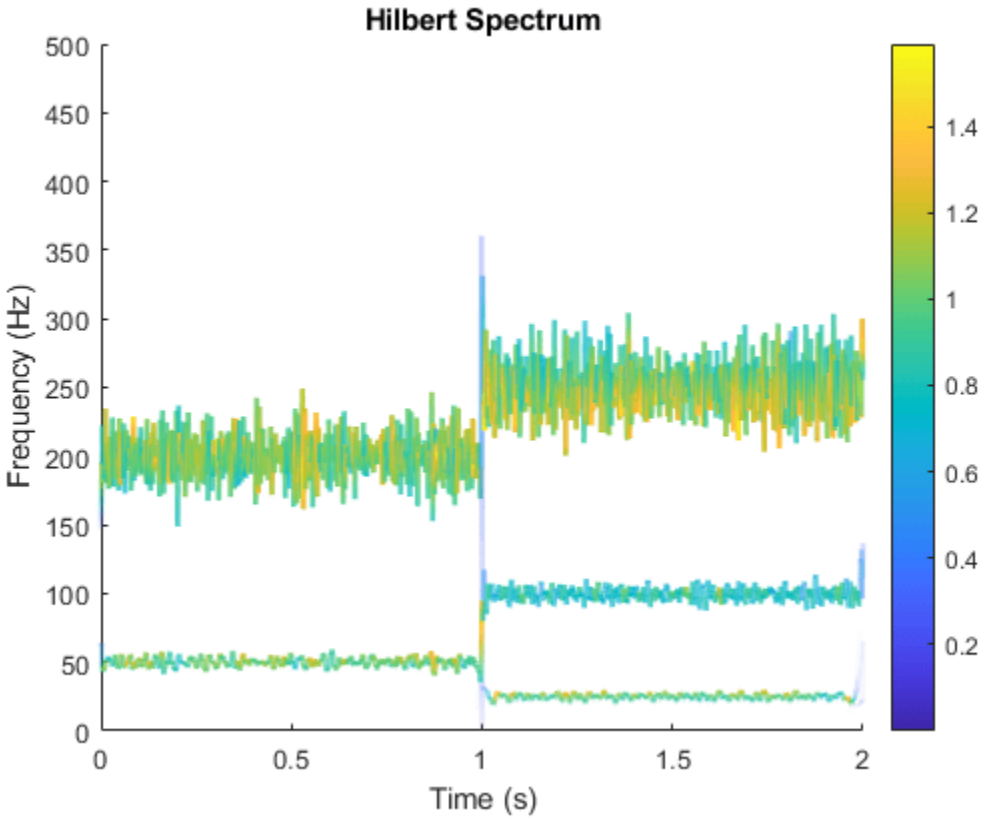
To create the Hilbert spectrum plot, you need the intrinsic mode functions (IMFs) of the signal. Perform empirical mode decomposition to compute the IMFs and residuals of the signal. Since the signal is not smooth, specify '`pchip`' as the interpolation method.

```
[imf,residual,info] = emd(X,'Interpolation','pchip');
```

The table generated in the command window indicates the number of sift iterations, the relative tolerance, and the sift stop criterion for each generated IMF. This information is also contained in `info`. You can hide the table by adding the `'Display',0` name value pair.

Create the Hilbert spectrum plot using the `imf` components obtained using empirical mode decomposition.

```
hht(imf,fs)
```

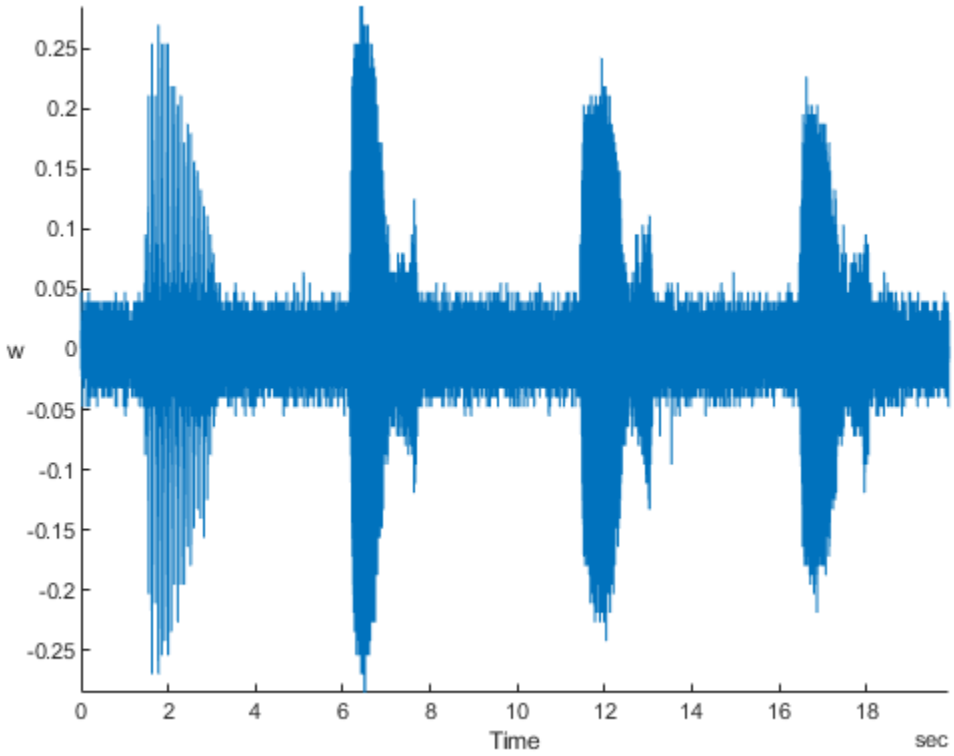


The frequency versus time plot is a sparse plot with a vertical color bar indicating the instantaneous energy at each point in the IMF. The plot represents the instantaneous frequency spectrum of each component decomposed from the original mixed signal. Three IMFs appear in the plot with a distinct change in frequency at 1 second.

Hilbert Spectrum of Whale Song

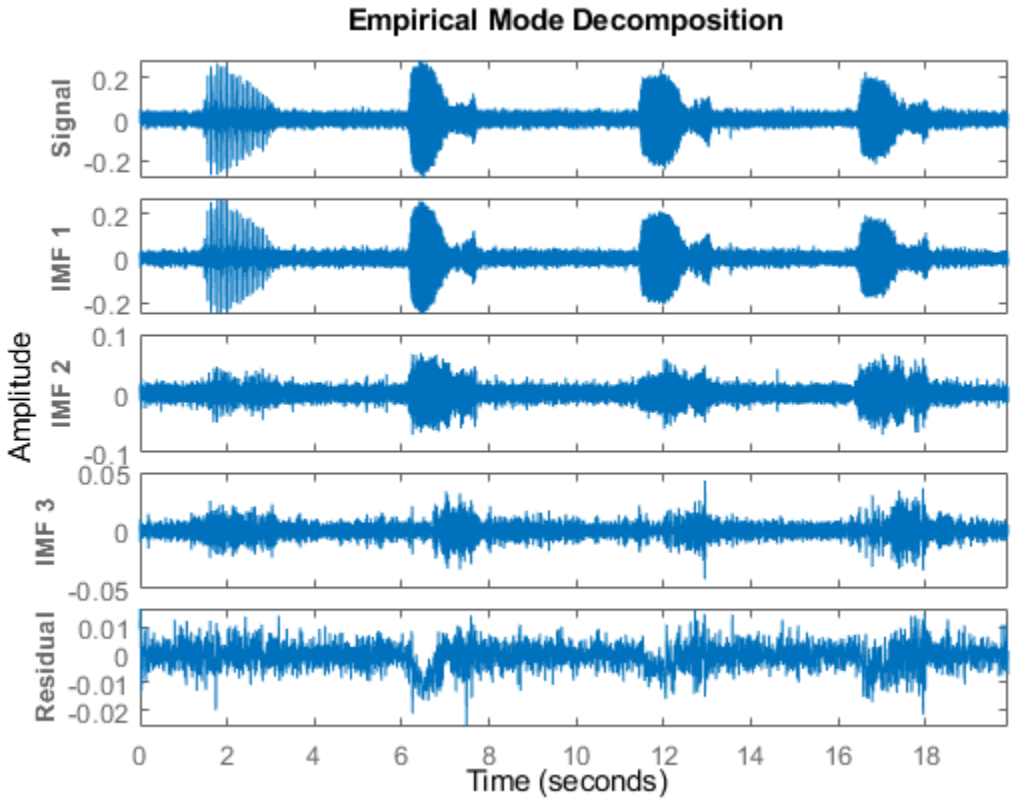
Load a file that contains audio data from a Pacific blue whale, sampled at 4 kHz. The file is from the library of animal vocalizations maintained by the Cornell University Bioacoustics Research Program. The time scale in the data is compressed by a factor of 10 to raise the pitch and make the calls more audible. Convert the signal to a MATLAB® timetable and plot it. Four features stand out from the noise in the signal. The first is known as a *trill*, and the other three are known as *moans*.

```
[w,fs] = audioread('bluewhale.wav');
whale = timetable(w,'SampleRate',fs);
stackedplot(whale);
```



Use `emd` to visualize the first three intrinsic mode functions (IMFs) and the residual.

```
emd(whale,'MaxNumIMF',3)
```



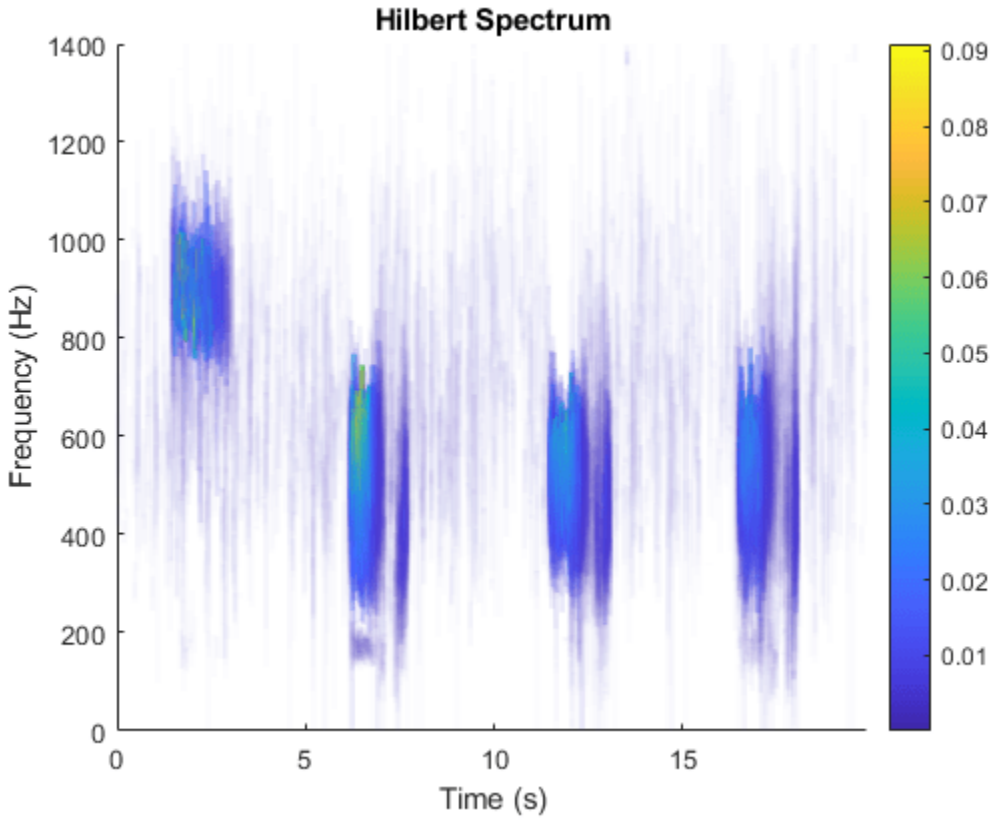
Compute the first three IMFs of the signal. Use the `'Display'` name-value pair to output a table showing the number of sifting iterations, the relative tolerance, and the sifting stop criterion for each IMF.

```
imf = emd(whale,'MaxNumIMF',3,'Display',1);
```

```
Current IMF  |  #Sift Iter  |  Relative Tol  |  Stop Criterion Hit
      1      |        1     |      0.13523   |  SiftMaxRelativeTolerance
      2      |        2     |     0.030198   |  SiftMaxRelativeTolerance
      3      |        2     |      0.01908   |  SiftMaxRelativeTolerance
Decomposition stopped because maximum number of intrinsic mode functions was extracted.
```

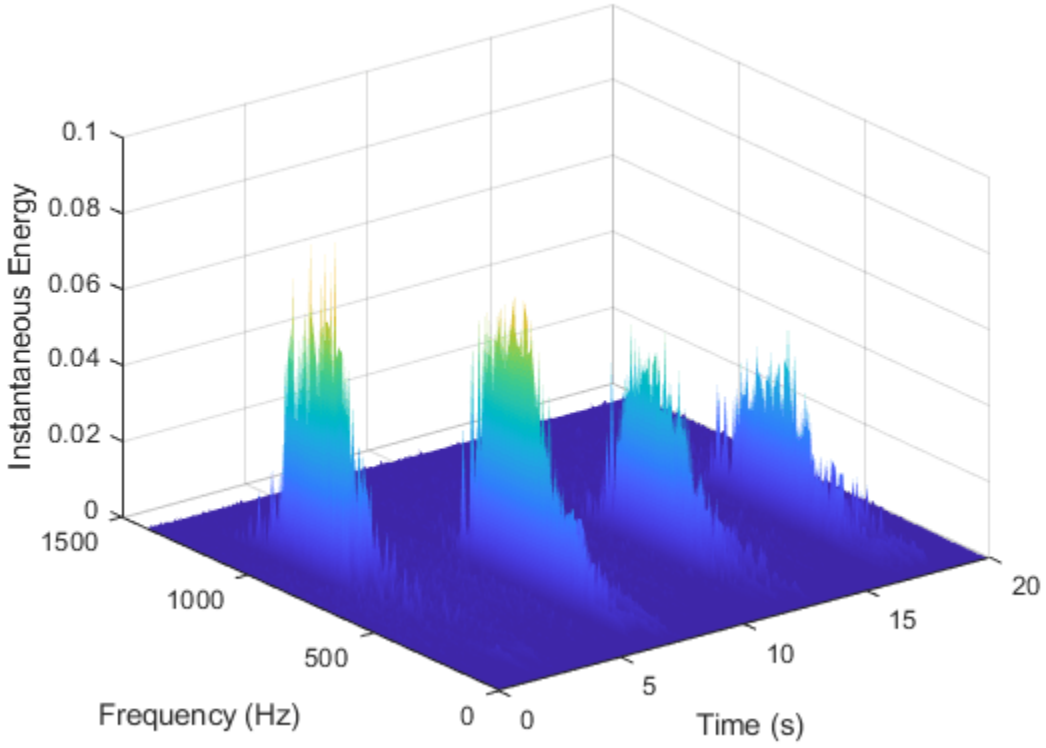
Use the computed IMFs to plot the Hilbert spectrum of the signal. Restrict the frequency range from 0 Hz to 1400 Hz.

```
hht(imf,'FrequencyLimits',[0 1400])
```



Compute the Hilbert spectrum for the same range of frequencies. Visualize the Hilbert spectra of the trill and moans as a mesh plot.

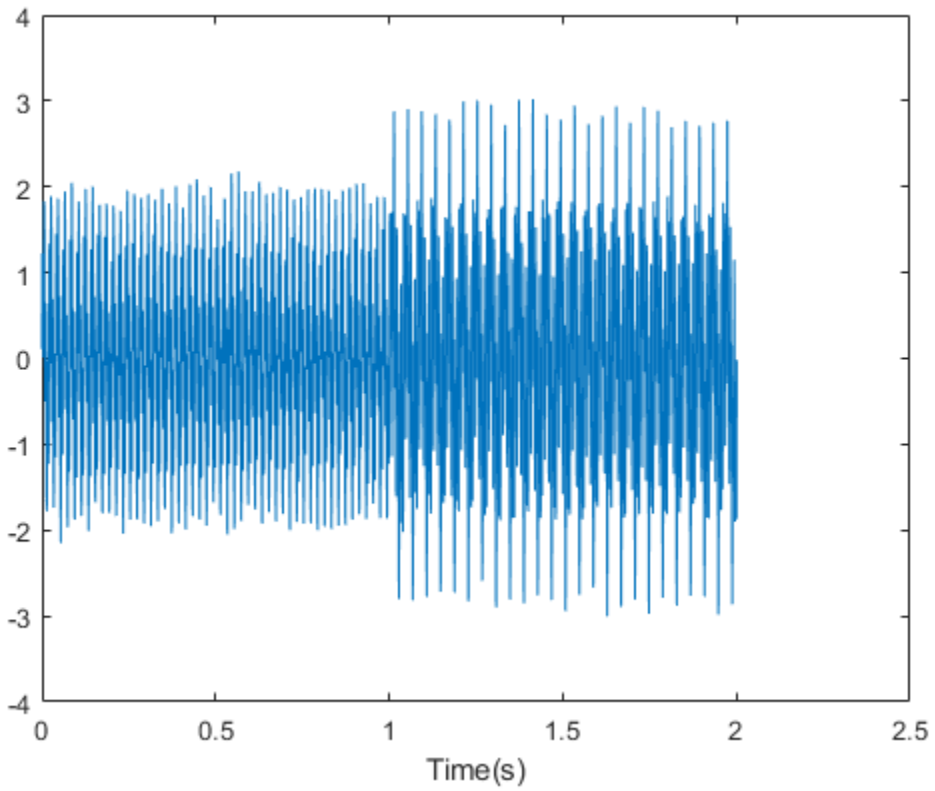
```
[hs,f,t] = hht(imf,'FrequencyLimits',[0 1400]);

mesh(seconds(t),f,hs,'EdgeColor','none','FaceColor','interp')
xlabel('Time (s)')
ylabel('Frequency (Hz)')
zlabel('Instantaneous Energy')
```



Compute Hilbert Spectrum Parameters of Signal

Load and visualize a nonstationary continuous signal composed of sinusoidal waves with a distinct change in frequency. The vibration of a jackhammer and the sound of fireworks are examples of nonstationary continuous signals. The signal is sampled at a rate `fs`.

```
load('sinusoidalSignalExampleData.mat','X','fs')
t = (0:length(X)-1)/fs;
plot(t,X)
xlabel('Time(s)')
```



The mixed signal contains sinusoidal waves with different amplitude and frequency values.

To compute the Hilbert spectrum parameters, you need the IMFs of the signal. Perform empirical mode decomposition to compute the intrinsic mode functions and residuals of the signal. Since the signal is not smooth, specify `'pchip'` as the interpolation method.

```
[imf,residual,info] = emd(X,'Interpolation','pchip');
```

The table generated in the command window indicates the number of sift iterations, the relative tolerance, and the sift stop criterion for each generated IMF. This information is also contained in `info`. You can hide the table by specifying `'Display'` as `0`.

Compute the Hilbert spectrum parameters: Hilbert spectrum `hs`, frequency vector `f`, time vector `t`, instantaneous frequency `imfinsf`, and instantaneous energy `imfinse`.

```
[hs,f,t,imfinsf,imfinse] = hht(imf,fs);
```

Use the computed Hilbert spectrum parameters for time-frequency analysis and signal diagnostics.

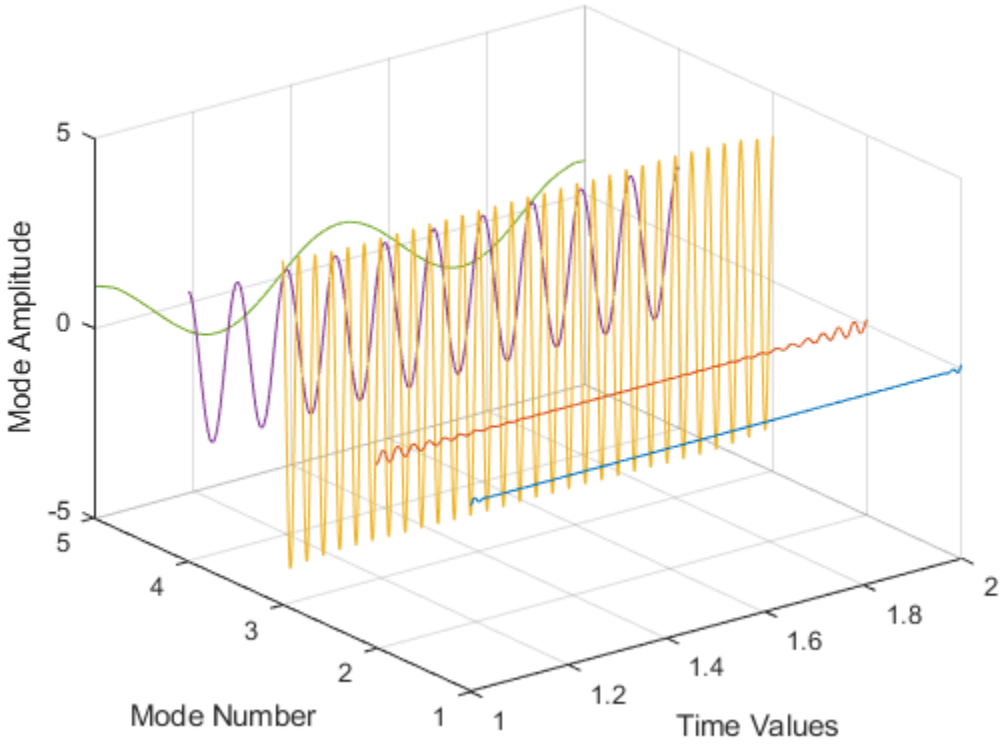
VMD of Multicomponent Signal

Generate a multicomponent signal consisting of three sinusoids of frequencies 2 Hz, 10 Hz, and 30 Hz. The sinusoids are sampled at 1 kHz for 2 seconds. Embed the signal in white Gaussian noise of variance 0.01^2.

```
fs = 1e3;
t = 1:1/fs:2-1/fs;
x = cos(2*pi*2*t) + 2*cos(2*pi*10*t) + 4*cos(2*pi*30*t) + 0.01*randn(1,length(t));
```

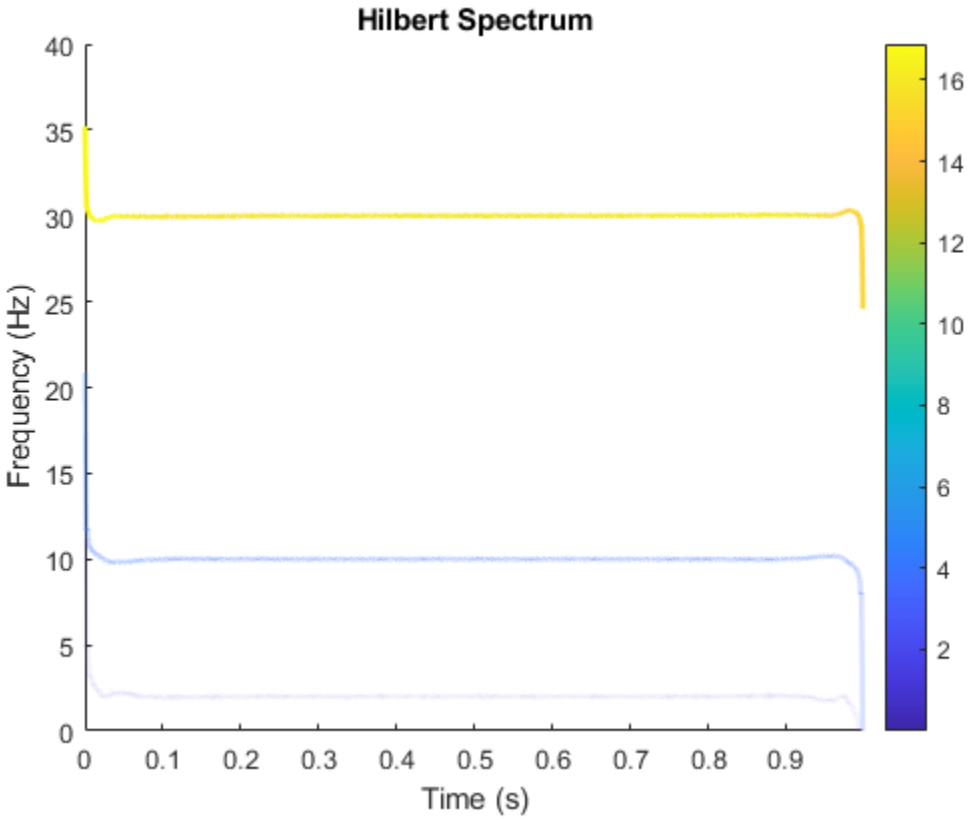
Compute the IMFs of the noisy signal and visualize them in a 3-D plot.

```
imf = vmd(x);
[p,q] = ndgrid(t,1:size(imf,2));
plot3(p,q,imf)
grid on
xlabel('Time Values')
ylabel('Mode Number')
zlabel('Mode Amplitude')
```



Use the computed IMFs to plot the Hilbert spectrum of the multicomponent signal. Restrict the frequency range to [0, 40] Hz.

```
hht(imf,fs,'FrequencyLimits',[0,40])
```

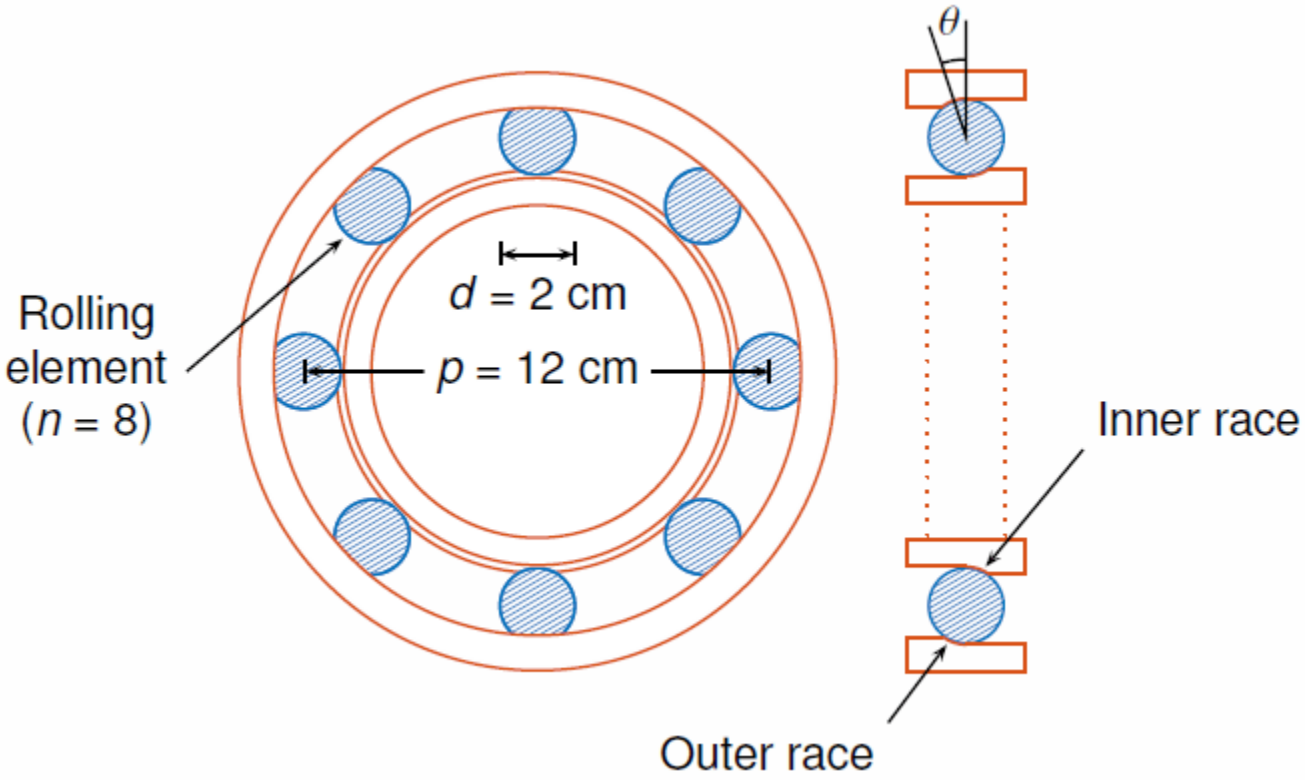


Compute Hilbert Spectrum of Vibration Signal

Simulate a vibration signal from a damaged bearing. Compute the Hilbert spectrum of this signal and look for defects.

A bearing with a pitch diameter of 12 cm has eight rolling elements. Each rolling element has a diameter of 2 cm. The outer race remains stationary as the inner race is driven at 25 cycles per second. An accelerometer samples the bearing vibrations at 10 kHz.

```
fs = 10000;
f0 = 25;
n = 8;
d = 0.02;
p = 0.12;
```



The vibration signal from the healthy bearing includes several orders of the driving frequency.

```
t = 0:1/fs:10-1/fs;
yHealthy = [1 0.5 0.2 0.1 0.05]*sin(2*pi*f0*[1 2 3 4 5]'.*t)/5;
```

A resonance is excited in the bearing vibration halfway through the measurement process.

```
yHealthy = (1+1./(1+linspace(-10,10,length(yHealthy)).^4)).*yHealthy;
```

The resonance introduces a defect in the outer race of the bearing that results in progressive wear. The defect causes a series of impacts that recur at the ball pass frequency outer race (BPFO) of the bearing:

$$\text{BPFO} = \frac{1}{2} n f_0 \left[1 - \frac{d}{p}\cos\theta\right],$$

where f_0 is the driving rate, n is the number of rolling elements, d is the diameter of the rolling elements, p is the pitch diameter of the bearing, and θ is the bearing contact angle. Assume a contact angle of 15° and compute the BPFO.

```
ca = 15;
bpfo = n*f0/2*(1-d/p*cosd(ca));
```

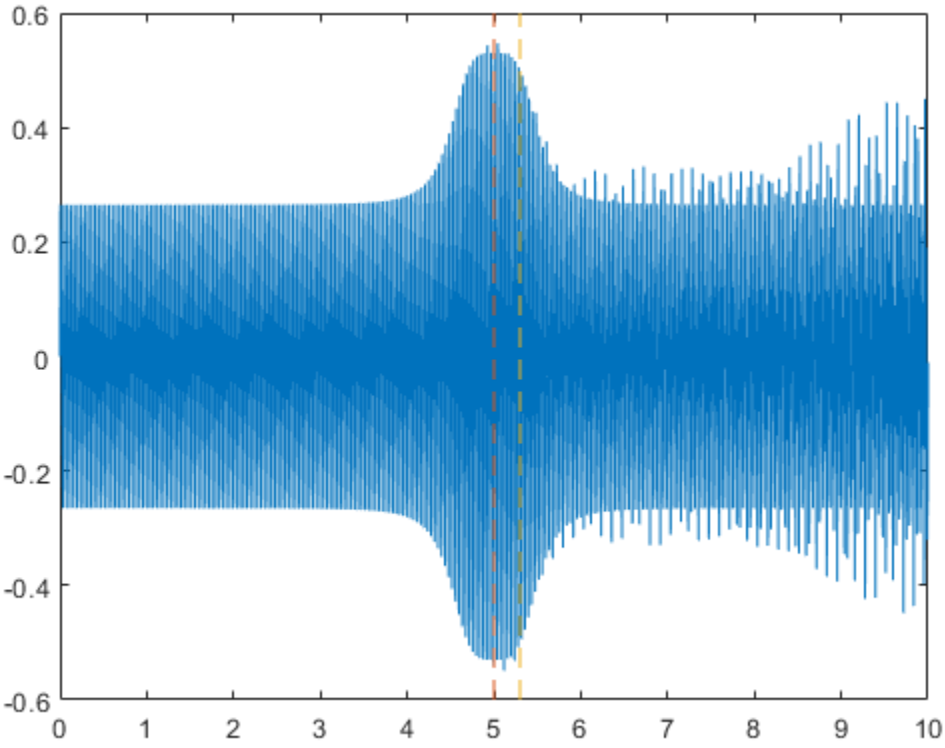
Use the `pulstran` function to model the impacts as a periodic train of 5-millisecond sinusoids. Each 3 kHz sinusoid is windowed by a flat top window. Use a power law to introduce progressive wear in the bearing vibration signal.

```
fImpact = 3000;
tImpact = 0:1/fs:5e-3-1/fs;
```

```
wImpact = flattopwin(length(tImpact))'/10;
xImpact = sin(2*pi*fImpact*tImpact).*wImpact;

tx = 0:1/bpfo:t(end);
tx = [tx; 1.3.^tx-2];

nWear = 49000;
nSamples = 100000;
yImpact = pulstran(t,tx',xImpact,fs)/5;
yImpact = [zeros(1,nWear) yImpact(1,(nWear+1):nSamples)];
```

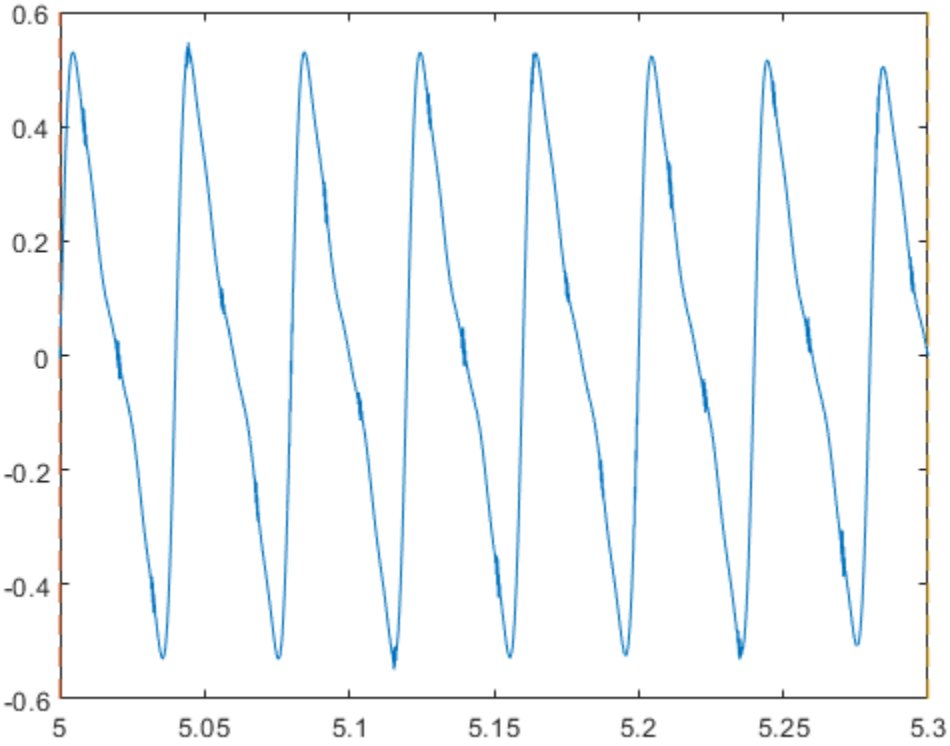
Generate the BPFO vibration signal by adding the impacts to the healthy bearing signal. Plot the signal and select a 0.3-second interval starting at 5.0 seconds.

```
yBPFO = yImpact + yHealthy;

xLimLeft = 5.0;
xLimRight = 5.3;
yMin = -0.6;
yMax = 0.6;

plot(t,yBPFO)

hold on
[limLeft,limRight] = meshgrid([xLimLeft xLimRight],[yMin yMax]);
plot(limLeft,limRight,'--')
hold off
```



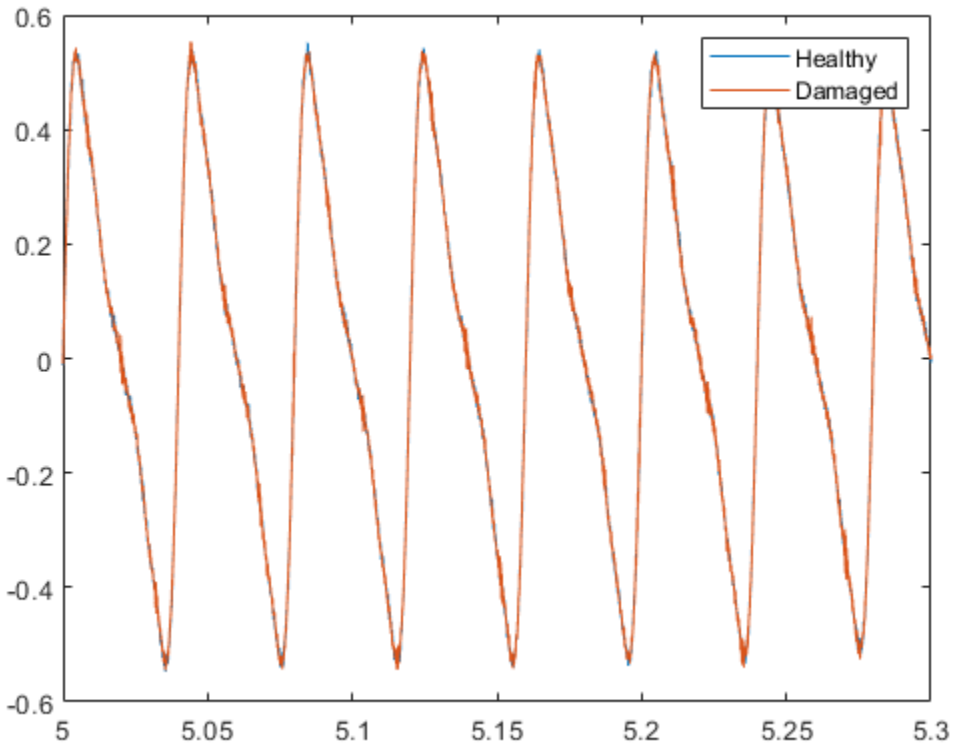
Zoom in on the selected interval to visualize the effect of the impacts.

```
xlim([xLimLeft xLimRight])
```



Add white Gaussian noise to the signals. Specify a noise variance of $1/150^2$.

```
rn = 150;
yGood = yHealthy + randn(size(yHealthy))/rn;
yBad = yBPFO + randn(size(yHealthy))/rn;

plot(t,yGood,t,yBad)
xlim([xLimLeft xLimRight])
legend('Healthy','Damaged')
```



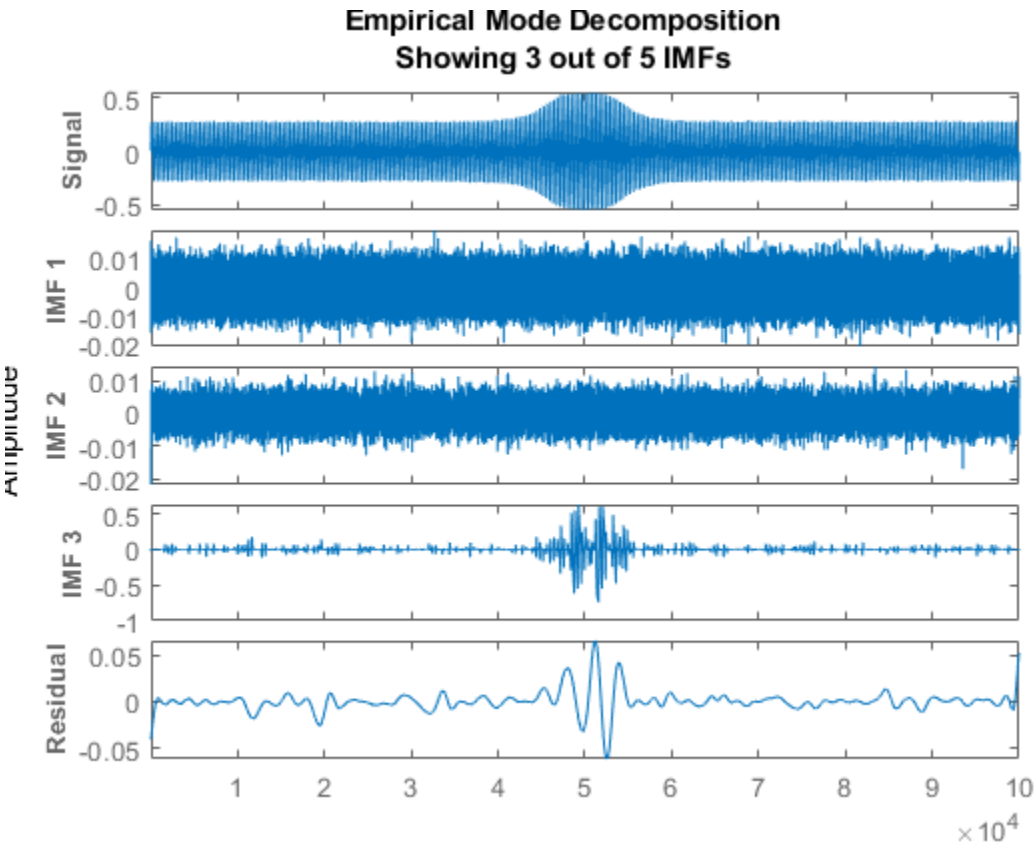
Use `emd` to perform an empirical mode decomposition of the healthy bearing signal. Compute the first five intrinsic mode functions (IMFs). Use the `'Display'` name-value argument to output a table showing the number of sifting iterations, the relative tolerance, and the sifting stop criterion for each IMF.

```
imfGood = emd(yGood,'MaxNumIMF',5,'Display',1);
```

```
Current IMF  |  #Sift Iter  |  Relative Tol  |  Stop Criterion Hit
      1      |        3     |     0.017132   |  SiftMaxRelativeTolerance
      2      |        3     |      0.12694   |  SiftMaxRelativeTolerance
      3      |        6     |      0.14582   |  SiftMaxRelativeTolerance
      4      |        1     |     0.011082   |  SiftMaxRelativeTolerance
      5      |        2     |      0.03463   |  SiftMaxRelativeTolerance
Decomposition stopped because maximum number of intrinsic mode functions was extracted.
```

Use `emd` without output arguments to visualize the first three IMFs and the residual.

```
emd(yGood,'MaxNumIMF',5)
```

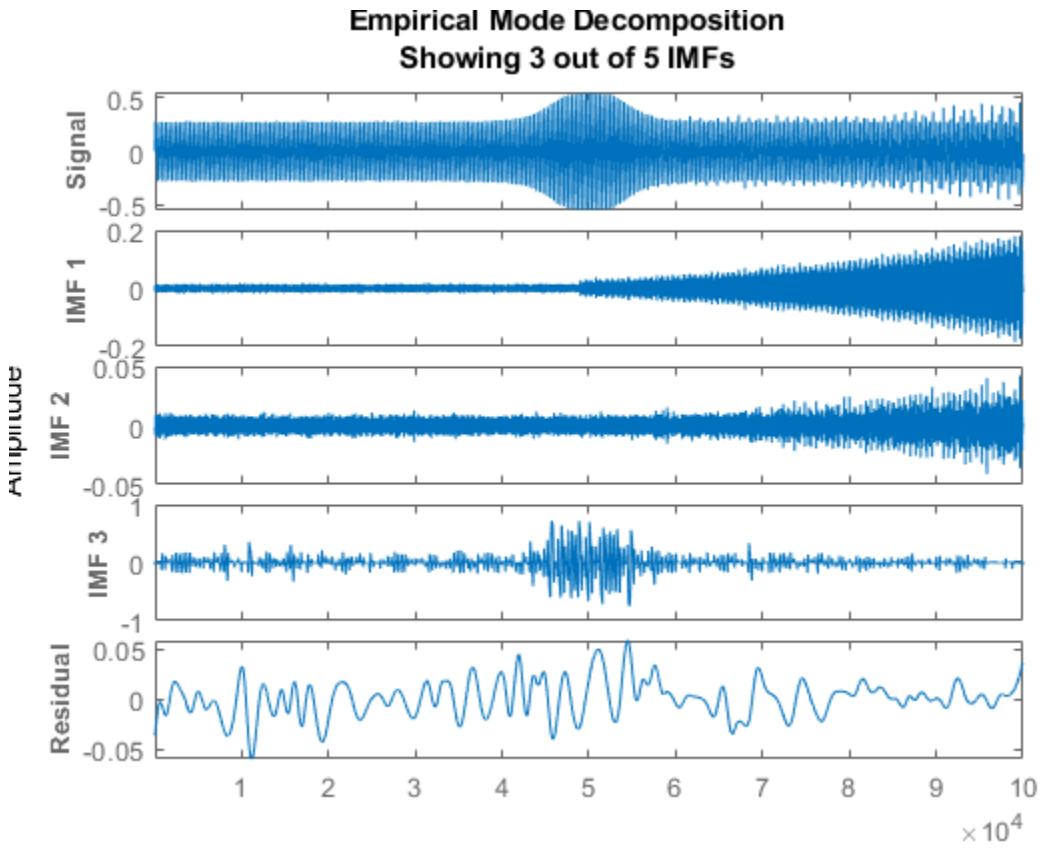


Compute and visualize the IMFs of the defective bearing signal. The first empirical mode reveals the high-frequency impacts. This high-frequency mode increases in energy as the wear progresses.

```
imfBad = emd(yBad,'MaxNumIMF',5,'Display',1);
```

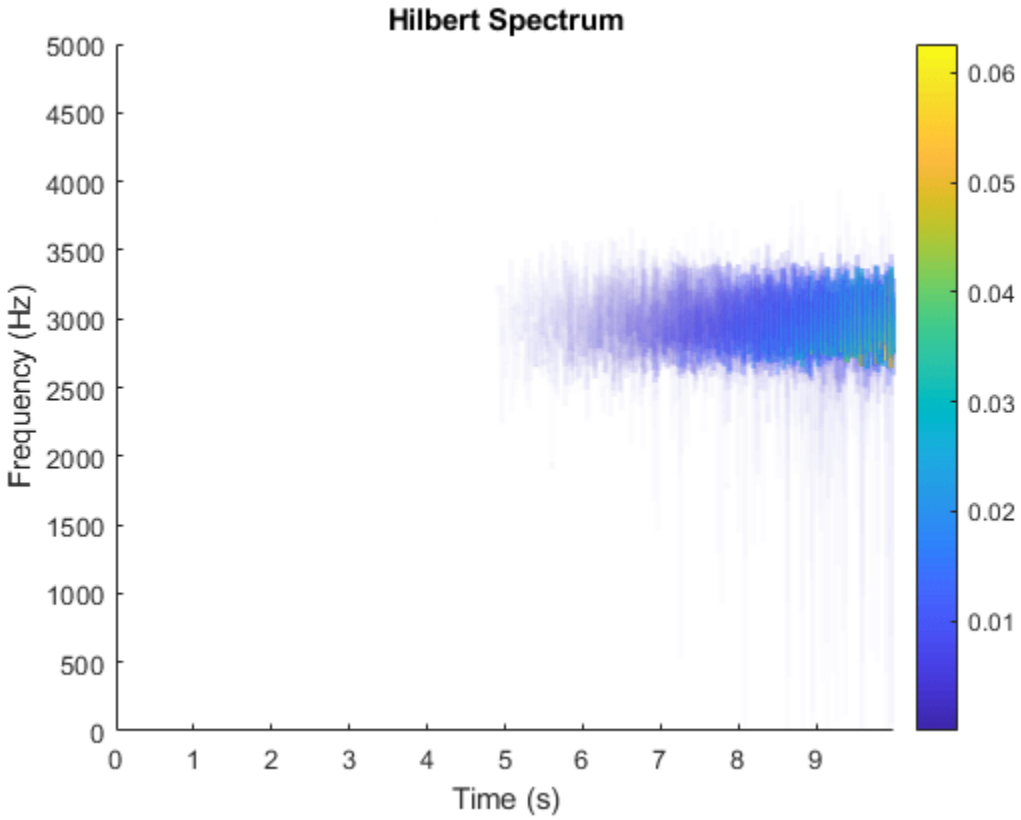
```
Current IMF  |  #Sift Iter  |  Relative Tol  |  Stop Criterion Hit
      1      |        2     |     0.041274   |  SiftMaxRelativeTolerance
      2      |        3     |      0.16695   |  SiftMaxRelativeTolerance
      3      |        3     |      0.18428   |  SiftMaxRelativeTolerance
      4      |        1     |     0.037177   |  SiftMaxRelativeTolerance
      5      |        2     |     0.095861   |  SiftMaxRelativeTolerance
Decomposition stopped because maximum number of intrinsic mode functions was extracted.
```

```
emd(yBad,'MaxNumIMF',5)
```



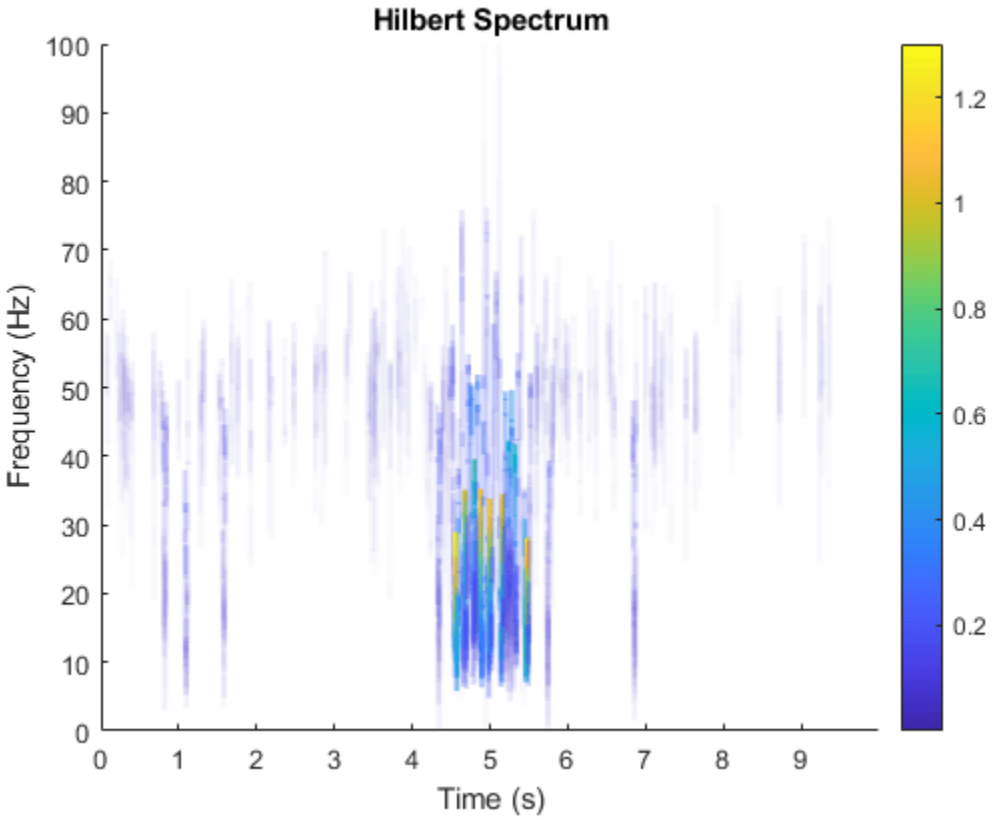
Plot the Hilbert spectrum of the first empirical mode of the defective bearing signal. The first mode captures the effect of high-frequency impacts. The energy of the impacts increases as the bearing wear progresses.

```
figure
hht(imfBad(:,1),fs)
```



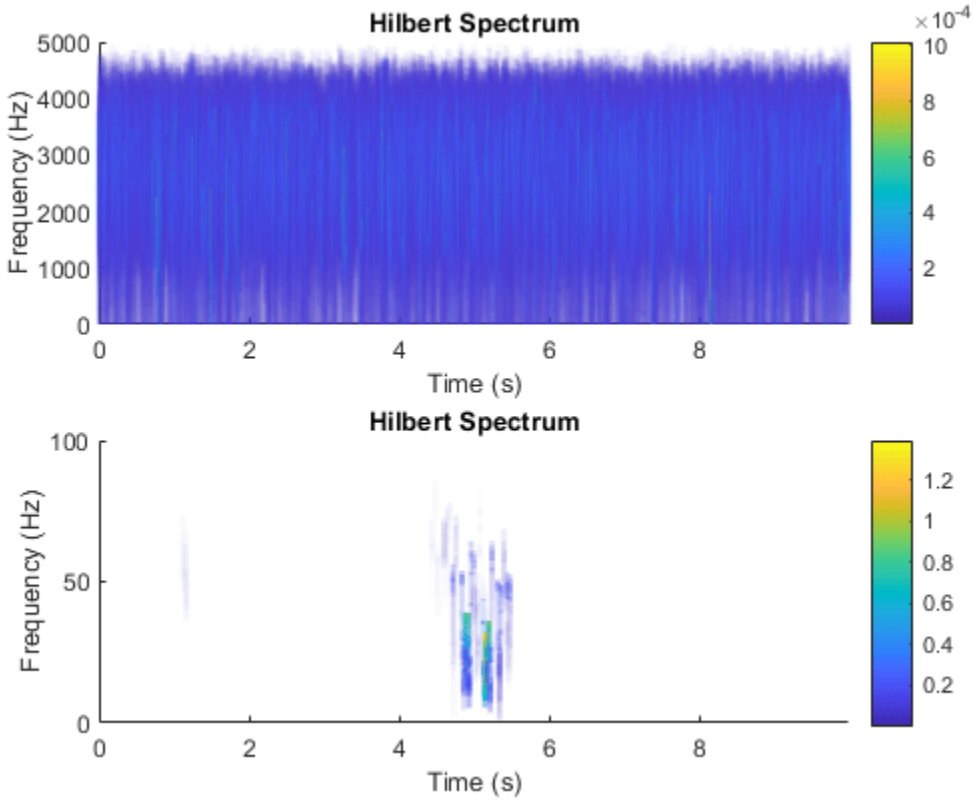
The Hilbert spectrum of the third mode shows the resonance in the vibration signal. Restrict the frequency range from 0 Hz to 100 Hz.

```
hht(imfBad(:,3),fs,'FrequencyLimits',[0 100])
```



For comparison, plot the Hilbert spectra of the first and third modes of the healthy bearing signal.

```
subplot(2,1,1)
hht(imfGood(:,1),fs)
subplot(2,1,2)
hht(imfGood(:,3),fs,'FrequencyLimits',[0 100])
```



Input Arguments

`imf` — Intrinsic mode function

matrix | timetable

Intrinsic mode function, specified as a matrix or timetable. `imf` is any signal whose envelope is symmetric with respect to zero and whose numbers of extrema and zero crossings differ by at most one. `emd` is used to decompose and simplify complicated signals into a finite number of intrinsic mode functions required to perform Hilbert spectral analysis.

`hht` treats each column in `imf` as an intrinsic mode function. For more information on computing `imf`, see `emd`.

`fs` — Sample Rate

2π (default) | positive scalar

Sample rate, specified as a positive scalar. If `fs` is not supplied, a normalized frequency of 2π is used to compute the Hilbert spectrum. If `imf` is specified as a timetable, the sample rate is inferred from it.

`freqlocation` — Location of frequency axis on plot

`'yaxis'` (default) | `'xaxis'`

Location of frequency axis on the plot, specified as `'yaxis'` or `'xaxis'`. To display frequency data on the *y*-axis or *x*-axis of the plot, specify `freqlocation` as `'yaxis'` or `'xaxis'` respectively.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'FrequencyResolution',1`

`FrequencyLimits` — Frequency limits to compute Hilbert spectrum

`[0,fs/2]` (default) | 1-by-2 integer-valued vector

Frequency limits to compute Hilbert spectrum, specified as the comma-separated pair consisting of '`FrequencyLimits`' and a 1-by-2 integer-valued vector. `FrequencyLimits` is specified in Hz.

`FrequencyResolution` — Frequency resolution to discretize frequency range

(f_high-f_low)/100 (default) | positive scalar

Frequency resolution to discretize frequency limits, specified as the comma-separated pair consisting of '`FrequencyResolution`' and a positive scalar.

Specify `FrequencyResolution` in Hz. If `'FrequencyResolution'` is not specified, a value of (f_{high}-f_{low})/100 is inferred from `FrequencyLimits`. Here, f_{high} is the upper limit of `FrequencyLimits` and f_{low} is the lower limit.

`MinThreshold` — Minimum threshold value of Hilbert spectrum

`-inf` (default) | scalar

Minimum threshold value of Hilbert spectrum, specified as the comma-separated pair consisting of '`MinThreshold`' and a scalar.

`MinThreshold` sets elements of `hs` to 0 when the corresponding elements of $10\log_{10}(hs)$ are less than `MinThreshold`.

Output Arguments

`hs` — Hilbert spectrum of signal

sparse matrix

Hilbert spectrum of the signal, returned as a sparse matrix. Use `hs` for time-frequency analysis and to identify localized features in the signal.

`f` — Frequency values

vector

Frequency values of the signal, returned as a vector. `hht` uses the frequency vector `f` and the time vector `t` to create the Hilbert spectrum plot.

Mathematically, `f` is denoted as: $f = f_{low} : f_{res} : f_{high}$, where f_{res} is the frequency resolution.

`t` — Time values

vector | `duration` array

Time values of the signal, returned as a vector or a `duration` array. `hht` uses the time vector `t` and the frequency vector `f` to create the Hilbert spectrum plot.

`t` is returned as:

- An array, if `imf` is specified as an array.
- A `duration` array, if `imf` is specified as a uniformly sampled timetable.

`imfinsf` — Instantaneous frequency of each IMF
vector | matrix | timetable

Instantaneous frequency of each IMF, returned as a vector, a matrix, or a timetable.

`imfinsf` has the same number of columns as `imf` and is returned as:

- A vector, if `imf` is specified as a vector.
- A matrix, if `imf` is specified as a matrix.
- A timetable, if `imf` is specified as a uniformly sampled timetable.

`imfinse` — Instantaneous energy of each IMF
vector | matrix | timetable

Instantaneous energy of each IMF, returned as a vector, a matrix, or a timetable.

`imfinse` has the same number of columns as `imf` and is returned as:

- A vector, if `imf` is specified as a vector.
- A matrix, if `imf` is specified as a matrix.
- A timetable, if `imf` is specified as a uniformly sampled timetable.

Algorithms

The Hilbert-Huang transform is useful for performing time-frequency analysis of nonstationary and nonlinear data. The Hilbert-Huang procedure consists of the following steps:

1. `emd` or `vmd` decomposes the data set x into a finite number of intrinsic mode functions.
2. For each intrinsic mode function, x_i, the function `hht`:
   a. Uses `hilbert` to compute the analytic signal, $z_i(t) = x_i(t) + jH\{x_i(t)\}$, where $H\{x_i\}$ is the Hilbert transform of x_i.
   b. Expresses z_i as $z_i(t) = a_i(t)\, e^{j\theta_i(t)}$, where $a_i(t)$ is the instantaneous amplitude and $\theta_i(t)$ is the instantaneous phase.
   c. Computes the instantaneous energy, $|a_i(t)|^2$, and the instantaneous frequency, $\omega_i(t) \equiv d\theta_i(t)/dt$. If given a sample rate, `hht` converts $\omega_i(t)$ to a frequency in Hz.
   d. Outputs the instantaneous energy in `imfinse` and the instantaneous frequency in `imfinsf`.
3. When called with no output arguments, `hht` plots the energy of the signal as a function of time and frequency, with color proportional to amplitude.

References

[1] Huang, Norden E, and Samuel S P Shen. *Hilbert–Huang Transform and Its Applications*. 2nd ed. Vol. 16. Interdisciplinary Mathematical Sciences. WORLD SCIENTIFIC, 2014. https://doi.org/10.1142/8804.

[2] Huang, Norden E., Zhaohua Wu, Steven R. Long, Kenneth C. Arnold, Xianyao Chen, and Karin Blank. “ON INSTANTANEOUS FREQUENCY.” *Advances in Adaptive Data Analysis* 01, no. 02 (April 2009): 177–229. https://doi.org/10.1142/S1793536909000096.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Arguments specified using name-value pairs must be compile-time constants.
- Timetables are not supported for code generation.

See Also

`emd` | `hilbert` | `vmd`

Topics

“Time-Frequency Gallery”
“Analytic Signal and Hilbert Transform”
“Hilbert Transform and Instantaneous Frequency”

Introduced in R2018a

highpass

Highpass-filter signals

Syntax

```
y = highpass(x,wpass)
y = highpass(x,fpass,fs)
y = highpass(xt,fpass)

y = highpass( ___ ,Name,Value)

[y,d] = highpass( ___ )

highpass( ___ )
```

Description

`y = highpass(x,wpass)` filters the input signal `x` using a highpass filter with normalized passband frequency `wpass` in units of π rad/sample. `highpass` uses a minimum-order filter with a stopband attenuation of 60 dB and compensates for the delay introduced by the filter. If `x` is a matrix, the function filters each column independently.

`y = highpass(x,fpass,fs)` specifies that `x` has been sampled at a rate of `fs` hertz. `fpass` is the passband frequency of the filter in hertz.

`y = highpass(xt,fpass)` highpass-filters the data in timetable `xt` using a filter with a passband frequency of `fpass` hertz. The function independently filters all variables in the timetable and all columns inside each variable.

`y = highpass( ___ ,Name,Value)` specifies additional options for any of the previous syntaxes using name-value pair arguments. You can change the stopband attenuation, the “Highpass Filter Steepness” on page 1-1024, and the type of impulse response of the filter.

`[y,d] = highpass( ___ )` also returns the `digitalFilter` object `d` used to filter the input.

`highpass( ___ )` with no output arguments plots the input signal and overlays the filtered signal.

Examples

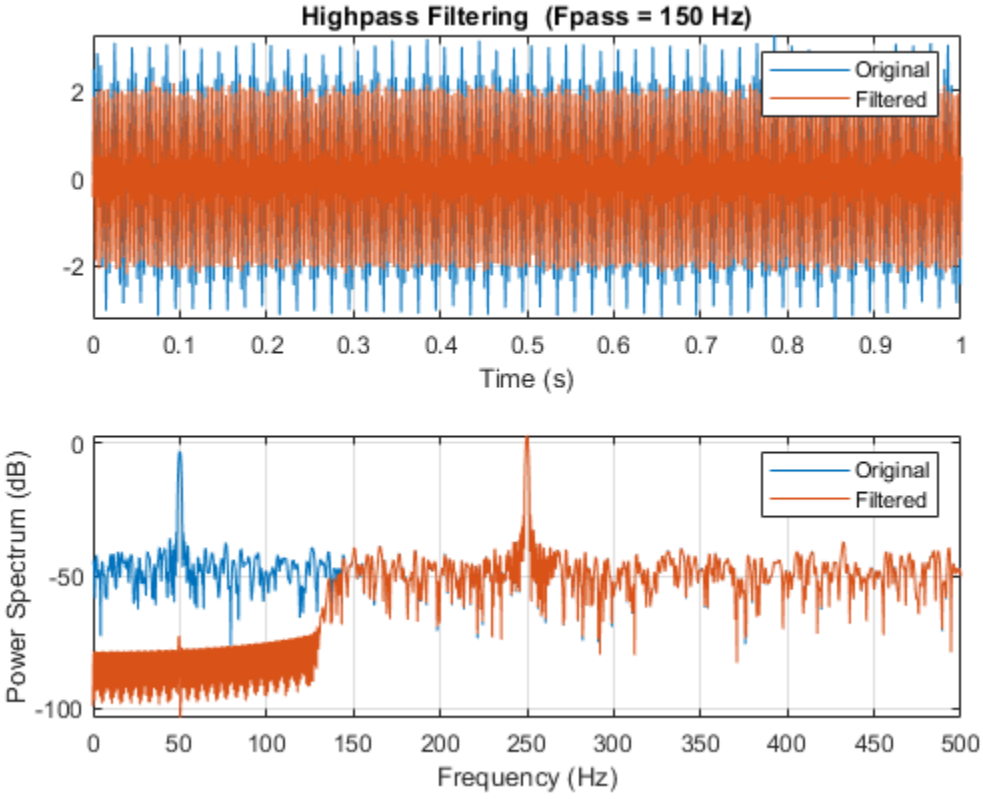
Highpass Filtering of Tones

Create a signal sampled at 1 kHz for 1 second. The signal contains two tones, one at 50 Hz and the other at 250 Hz, embedded in Gaussian white noise of variance 1/100. The high-frequency tone has twice the amplitude of the low-frequency tone.

```
fs = 1e3;
t = 0:1/fs:1;

x = [1 2]*sin(2*pi*[50 250]'.*t) + randn(size(t))/10;
```

Highpass-filter the signal to remove the low-frequency tone. Specify a passband frequency of 150 Hz. Display the original and filtered signals, and also their spectra.

```
highpass(x,150,fs)
```

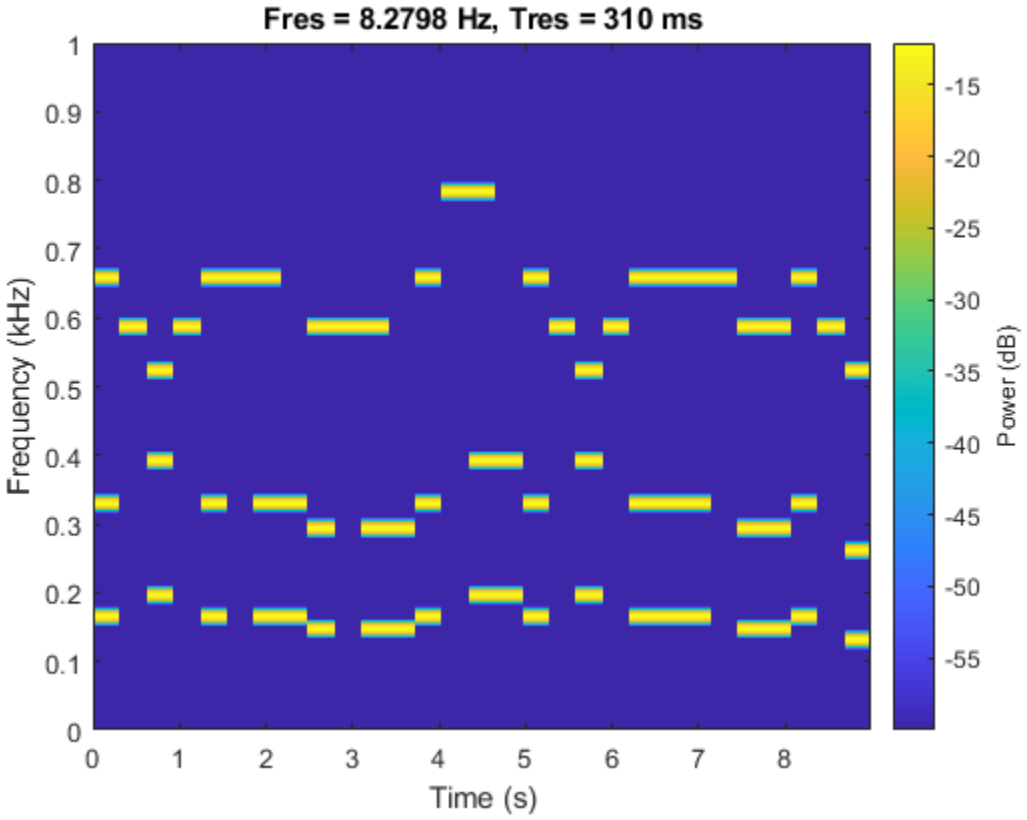


Highpass Filtering of Musical Signal

Implement a basic digital music synthesizer and use it to play a traditional song. Specify a sample rate of 2 kHz. Plot the spectrogram of the song.

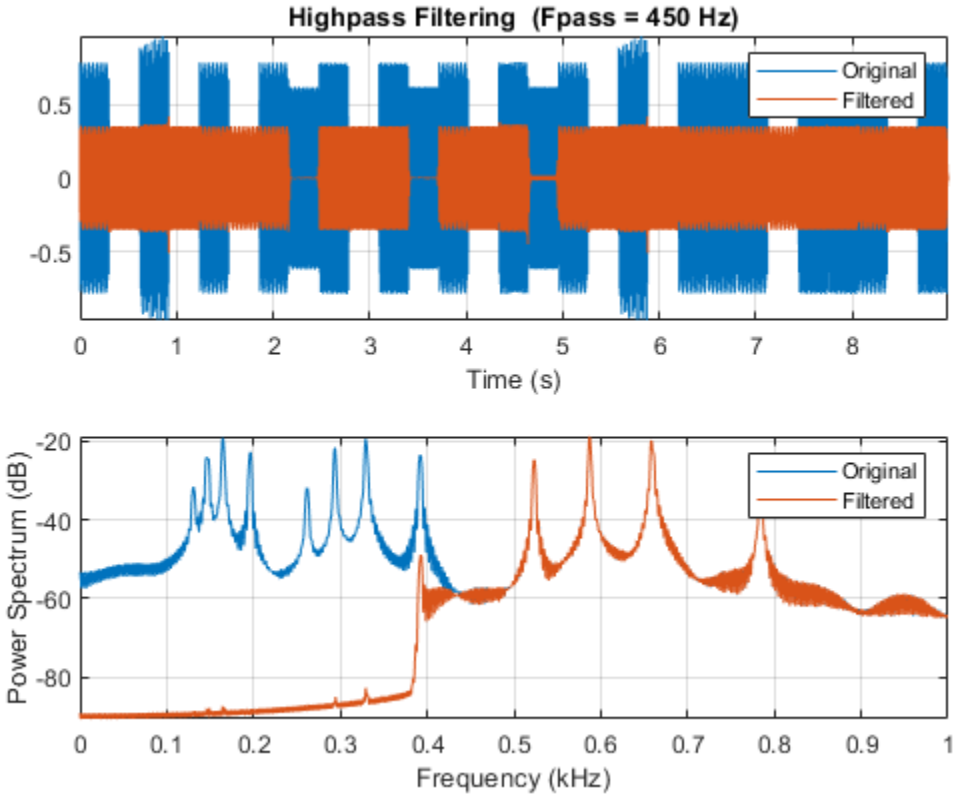
```
fs = 2e3;
t = 0:1/fs:0.3-1/fs;

l = [0 130.81 146.83 164.81 174.61 196.00 220 246.94];
m = [0 261.63 293.66 329.63 349.23 392.00 440 493.88];
h = [0 523.25 587.33 659.25 698.46 783.99 880 987.77];
note = @(f,g) [1 1 1]*sin(2*pi*[l(g) m(g) h(f)]'.*t);

mel = [3 2 1 2 3 3 3 0 2 2 2 0 3 5 5 0 3 2 1 2 3 3 3 3 2 2 3 2 1]+1;
acc = [3 0 5 0 3 0 3 3 2 0 2 2 3 0 5 5 3 0 5 0 3 3 3 0 2 2 3 0 1]+1;

song = [];
for kj = 1:length(mel)
    song = [song note(mel(kj),acc(kj)) zeros(1,0.01*fs)];
end
```

```
song = song/(max(abs(song))+0.1);

% To hear, type sound(song,fs)

pspectrum(song,fs,'spectrogram','TimeResolution',0.31, ...
    'OverlapPercent',0,'MinThreshold',-60)
```



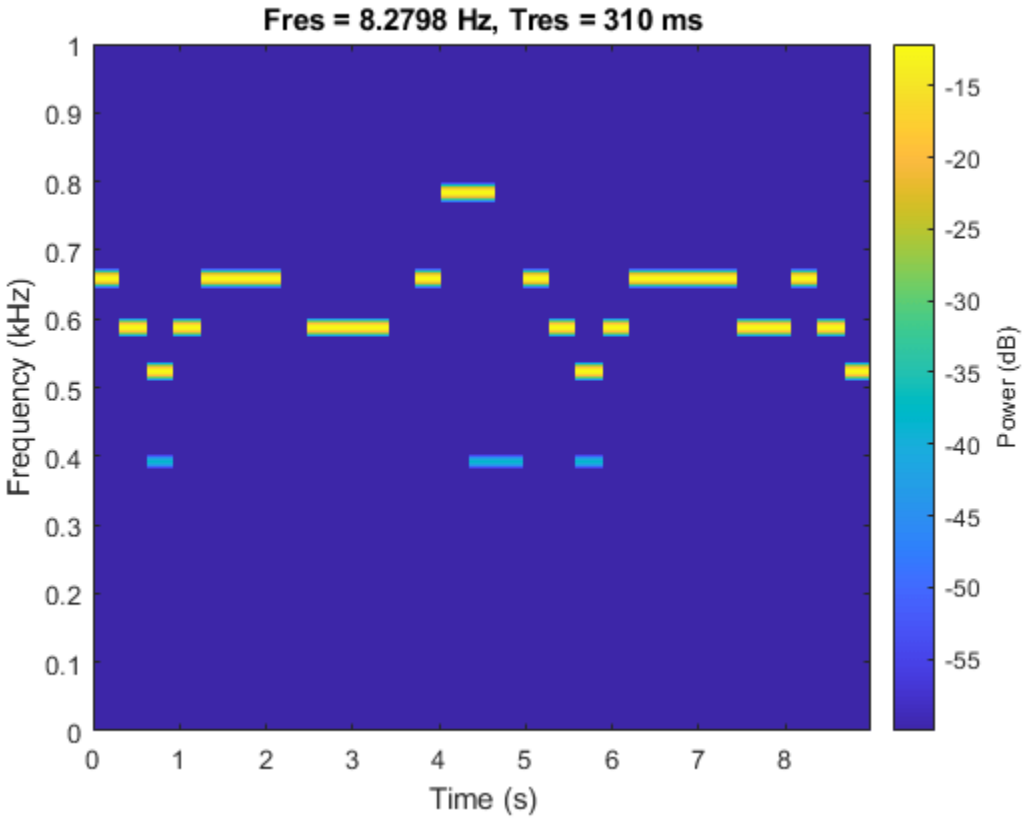
Highpass-filter the signal to separate the melody from the accompaniment. Specify a passband frequency of 450 Hz. Plot the original and filtered signals in the time and frequency domains.

```
hong = highpass(song,450,fs);

% To hear, type sound(hong,fs)

highpass(song,450,fs)
```



Plot the spectrogram of the melody.

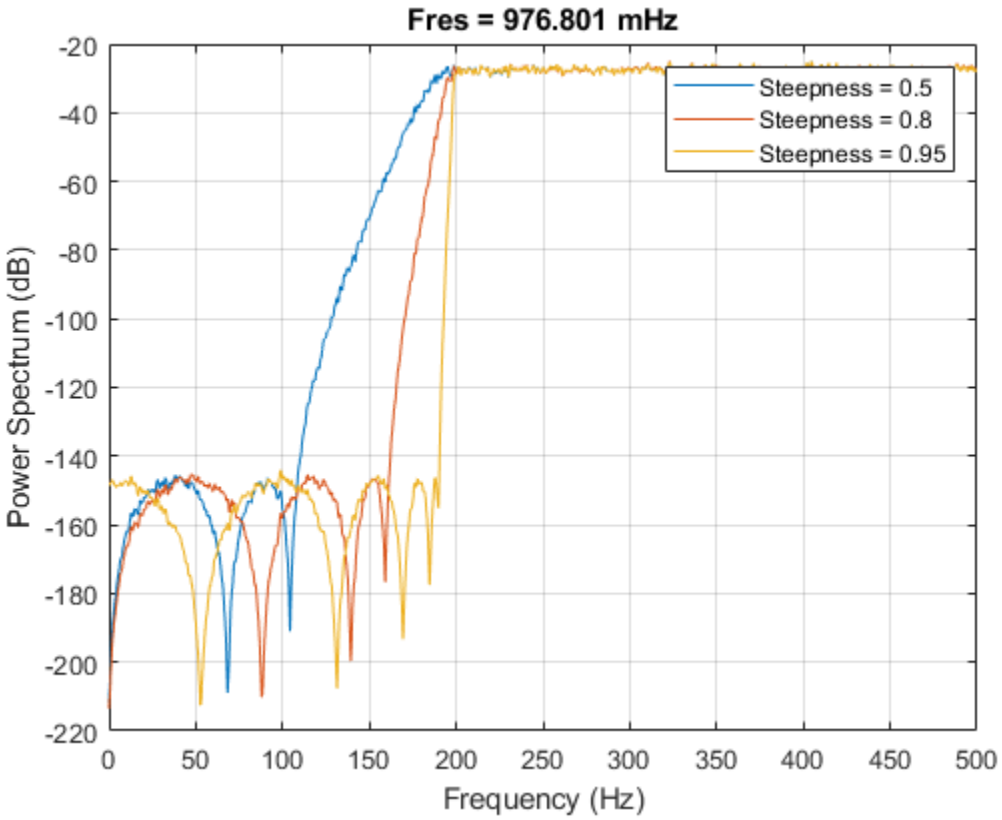
```
figure
pspectrum(hong,fs,'spectrogram','TimeResolution',0.31, ...
    'OverlapPercent',0,'MinThreshold',-60)
```



Highpass Filter Steepness

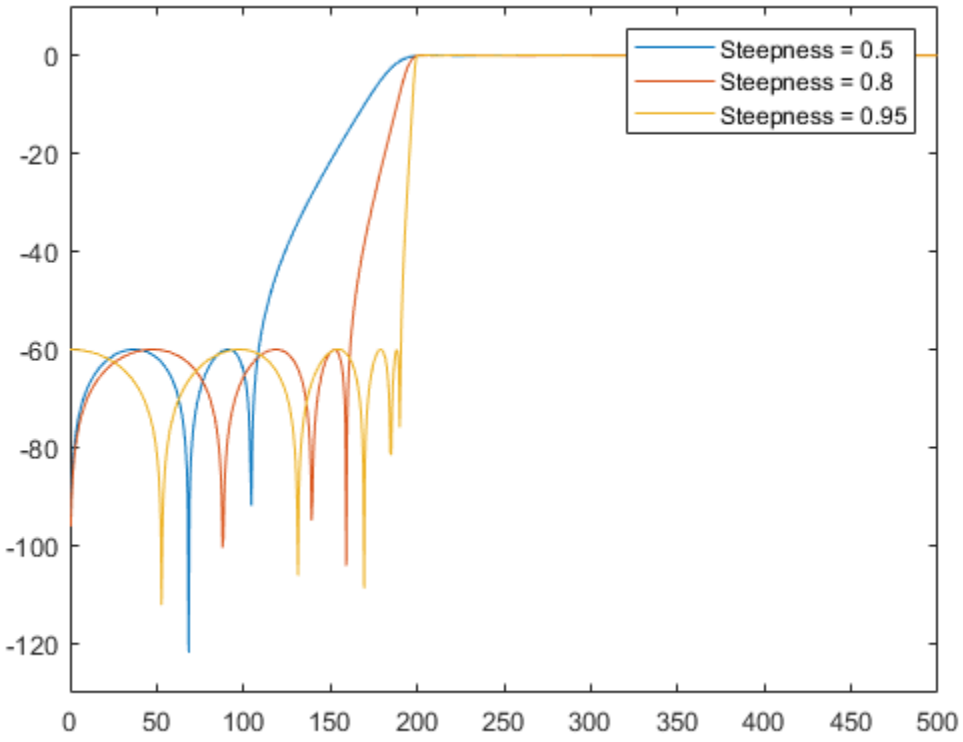
Filter white noise sampled at 1 kHz using an infinite impulse response highpass filter with a passband frequency of 200 Hz. Use different steepness values. Plot the spectra of the filtered signals.

```
fs = 1000;
x = randn(20000,1);

[y1,d1] = highpass(x,200,fs,'ImpulseResponse','iir','Steepness',0.5);
[y2,d2] = highpass(x,200,fs,'ImpulseResponse','iir','Steepness',0.8);
[y3,d3] = highpass(x,200,fs,'ImpulseResponse','iir','Steepness',0.95);

pspectrum([y1 y2 y3],fs)
legend('Steepness = 0.5','Steepness = 0.8','Steepness = 0.95')
```



Compute and plot the frequency responses of the filters.

```
[h1,f] = freqz(d1,1024,fs);
[h2,~] = freqz(d2,1024,fs);
[h3,~] = freqz(d3,1024,fs);

plot(f,mag2db(abs([h1 h2 h3])))
legend('Steepness = 0.5','Steepness = 0.8','Steepness = 0.95')
ylim([-130 10])
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix.

Example: `sin(2*pi*(0:127)/16)+randn(1,128)/100` specifies a noisy sinusoid

Example: `[2 1].*sin(2*pi*(0:127)'./[16 64])` specifies a two-channel sinusoid.

Data Types: `single` | `double`
Complex Number Support: Yes

wpass — Normalized passband frequency

scalar in (0, 1)

Normalized passband frequency, specified as a scalar in the interval (0, 1).

fpass — Passband frequency

scalar in (0, `fs`/2)

Passband frequency, specified as a scalar in the interval (0, `fs`/2).

fs — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar.

`xt` — Input timetable
timetable

Input timetable. `xt` must contain increasing, finite, and equally spaced row times of type `duration` in seconds.

If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

Example: `timetable(seconds(0:4)',randn(5,1))` specifies a random variable sampled at 1 Hz for 4 seconds.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ImpulseResponse','iir','StopbandAttenuation',30` filters the input using a minimum-order IIR filter that attenuates frequencies lower than `fpass` by 30 dB.

`ImpulseResponse` — Type of impulse response
`'auto'` (default) | `'fir'` | `'iir'`

Type of impulse response of the filter, specified as the comma-separated pair consisting of `'ImpulseResponse'` and `'fir'`, `'iir'`, or `'auto'`.

- `'fir'` — The function designs a minimum-order, linear-phase, finite impulse response (FIR) filter. To compensate for the delay, the function appends to the input signal $N/2$ zeros, where N is the filter order. The function then filters the signal and removes the first $N/2$ samples of the output.

  In this case, the input signal must be at least twice as long as the filter that meets the specifications.
- `'iir'` — The function designs a minimum-order infinite impulse response (IIR) filter and uses the `filtfilt` function to perform zero-phase filtering and compensate for the filter delay.

  If the signal is not at least three times as long as the filter that meets the specifications, the function designs a filter with smaller order and thus smaller steepness.
- `'auto'` — The function designs a minimum-order FIR filter if the input signal is long enough, and a minimum-order IIR filter otherwise. Specifically, the function follows these steps:

  - Compute the minimum order that an FIR filter must have to meet the specifications. If the signal is at least twice as long as the required filter order, design and use that filter.
  - If the signal is not long enough, compute the minimum order that an IIR filter must have to meet the specifications. If the signal is at least three times as long as the required filter order, design and use that filter.
  - If the signal is not long enough, truncate the order to one-third the signal length and design an IIR filter of that order. The reduction in order comes at the expense of transition band steepness.

- Filter the signal and compensate for the delay.

`Steepness` — Transition band steepness
0.85 (default) | scalar in the interval [0.5, 1)

Transition band steepness, specified as the comma-separated pair consisting of `'Steepness'` and a scalar in the interval [0.5, 1). As the steepness increases, the filter response approaches the ideal highpass response, but the resulting filter length and the computational cost of the filtering operation also increase. See “Highpass Filter Steepness” on page 1-1024 for more information.

`StopbandAttenuation` — Filter stopband attenuation
`60` (default) | positive scalar in dB

Filter stopband attenuation, specified as the comma-separated pair consisting of `'StopbandAttenuation'` and a positive scalar in dB.

Output Arguments

`y` — Filtered signal
vector | matrix | timetable

Filtered signal, returned as a vector, a matrix, or a timetable with the same dimensions as the input.

`d` — Highpass filter
`digitalFilter` object

Highpass filter used in the filtering operation, returned as a `digitalFilter` object.

- Use `filter(d,x)` to filter a signal `x` using `d`.
- Use **FVTool** to visualize the filter response.
- Use `designfilt` to edit or generate a digital filter based on frequency-response specifications.

More About

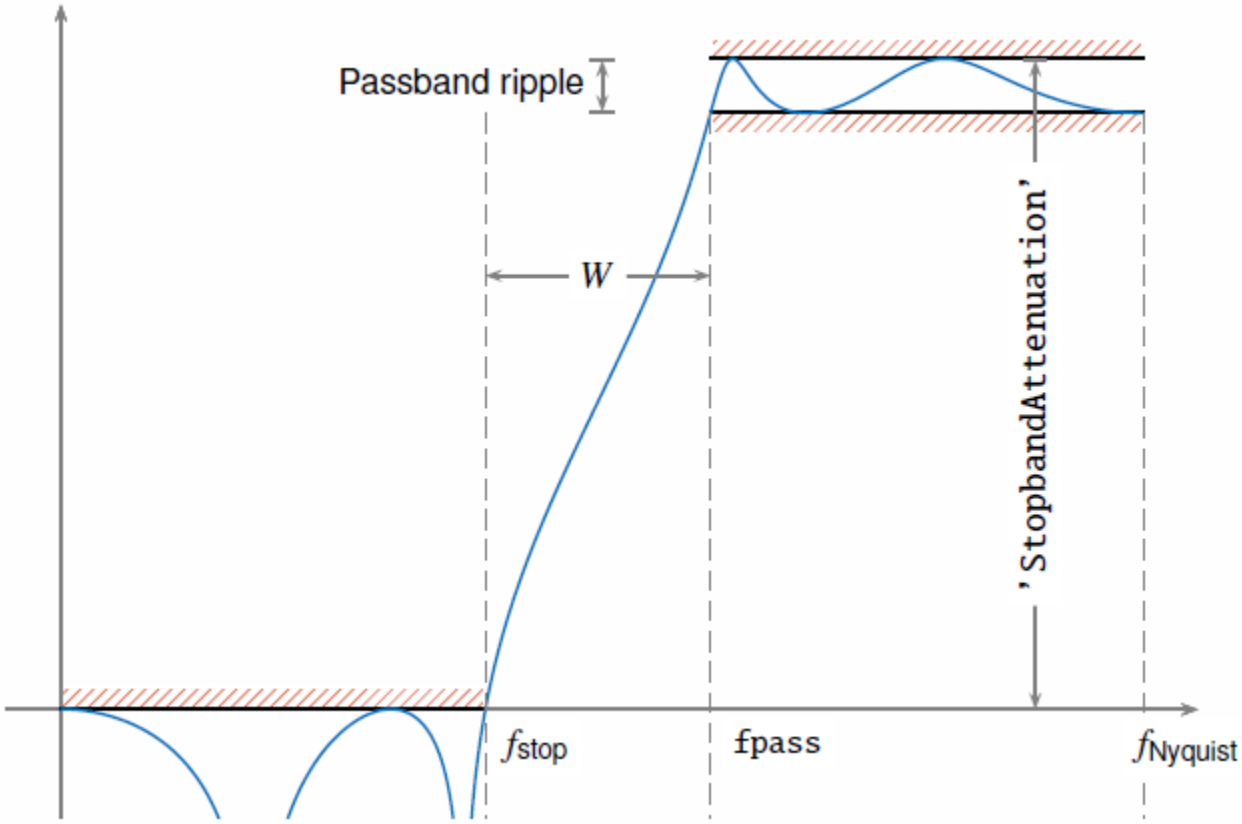
Highpass Filter Steepness

The `'Steepness'` argument controls the width of a filter's transition region. The lower the steepness, the wider the transition region. The higher the steepness, the narrower the transition region.

To interpret the filter steepness, consider the following definitions:

- The Nyquist frequency, f_{Nyquist}, is the highest frequency component of a signal that can be sampled at a given rate without aliasing. f_{Nyquist} is 1 ($\times\pi$ rad/sample) when the input signal has no time information, and `fs`/2 hertz when the input signal is a timetable or when you specify a sample rate.
- The stopband frequency of the filter, f_{stop}, is the frequency below which the attenuation is equal to or greater than the value specified using `'StopbandAttenuation'`.
- The transition width of the filter, W, is `fpass` – f_{stop}, where `fpass` is the specified passband frequency.

- Most nonideal filters also attenuate the input signal across the passband. The maximum value of this frequency-dependent attenuation is called the passband ripple. Every filter used by `highpass` has a passband ripple of 0.1 dB.



When you specify a value, *s*, for `'Steepness'`, the function computes the transition width as $W = (1 - s) \times$ `fpass`.

- When `'Steepness'` is equal to 0.5, the transition width is 50% of `fpass`.
- As `'Steepness'` approaches 1, the transition width becomes progressively narrower until it reaches a minimum value of 1% of `fpass`.
- The default value of `'Steepness'` is 0.85, which corresponds to a transition width that is 15% of `fpass`.

See Also

Apps
Signal Analyzer

Functions
`bandpass` | `bandstop` | `designfilt` | `filter` | `filtfilt` | `fir1` | `lowpass`

Introduced in R2018a

hilbert

Discrete-time analytic signal using Hilbert transform

Syntax

```
x = hilbert(xr)
x = hilbert(xr,n)
```

Description

`x = hilbert(xr)` returns the analytic signal, `x`, from a real data sequence, `xr`. If `xr` is a matrix, then `hilbert` finds the analytic signal corresponding to each column.

`x = hilbert(xr,n)` uses an `n`-point fast Fourier transform (FFT) to compute the Hilbert transform. The input data is zero-padded or truncated to length `n`, as appropriate.

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 = 1×4 complex

   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)
```

```
imx = 1×4

     1    -1    -1     1
```

```
rex = real(x)
```

```
rex = 1×4

     1     2     3     4
```

The last half of the discrete Fourier transform (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 `fft(x)` are purely real.

```
dft = fft(x)
```

```
dft = 1×4 complex
```

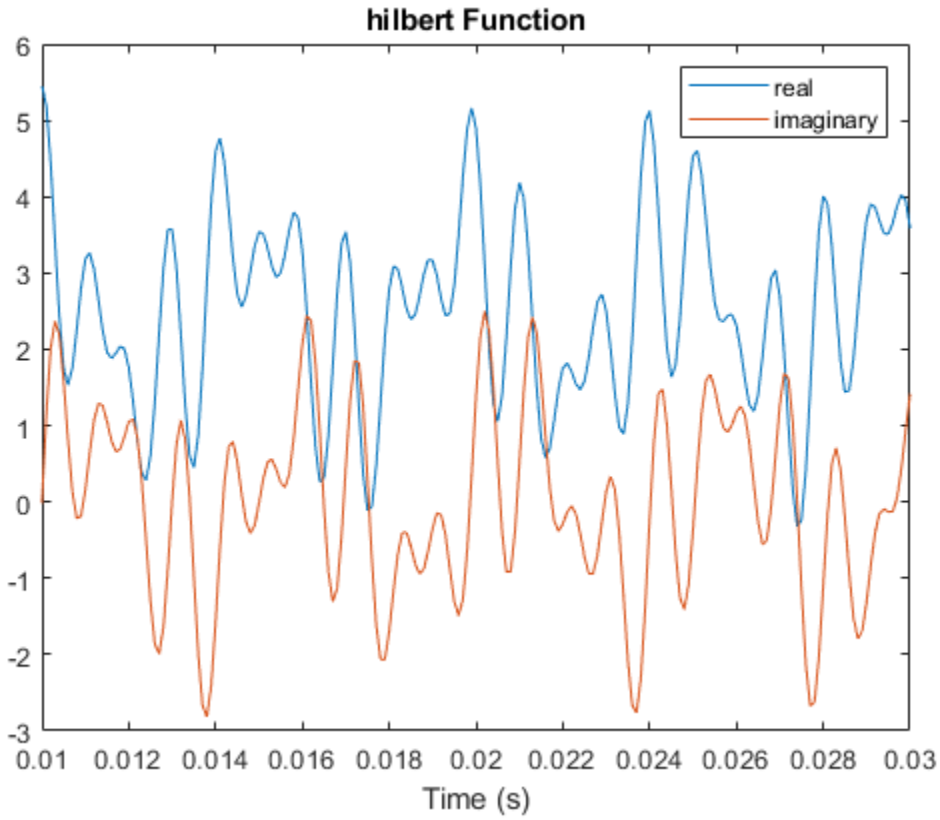
```
  10.0000 + 0.0000i  -4.0000 + 4.0000i  -2.0000 + 0.0000i   0.0000 + 0.0000i
```

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 finite impulse response (FIR) Hilbert transformer filter to compute an approximation to the imaginary part.

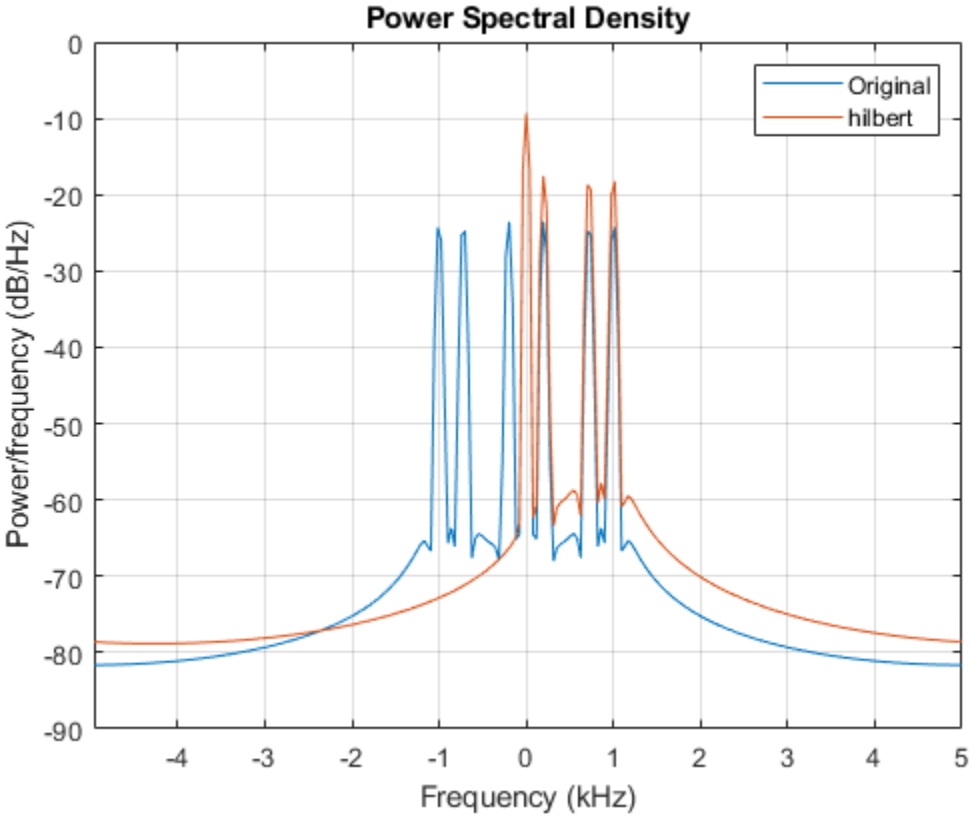
Generate a sequence composed of three sinusoids with frequencies 203, 721, and 1001 Hz. The sequence is sampled at 10 kHz for about 1 second. Use the `hilbert` function to compute the analytic signal. Plot it between 0.01 seconds and 0.03 seconds.

```
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')
xlabel('Time (s)')
```



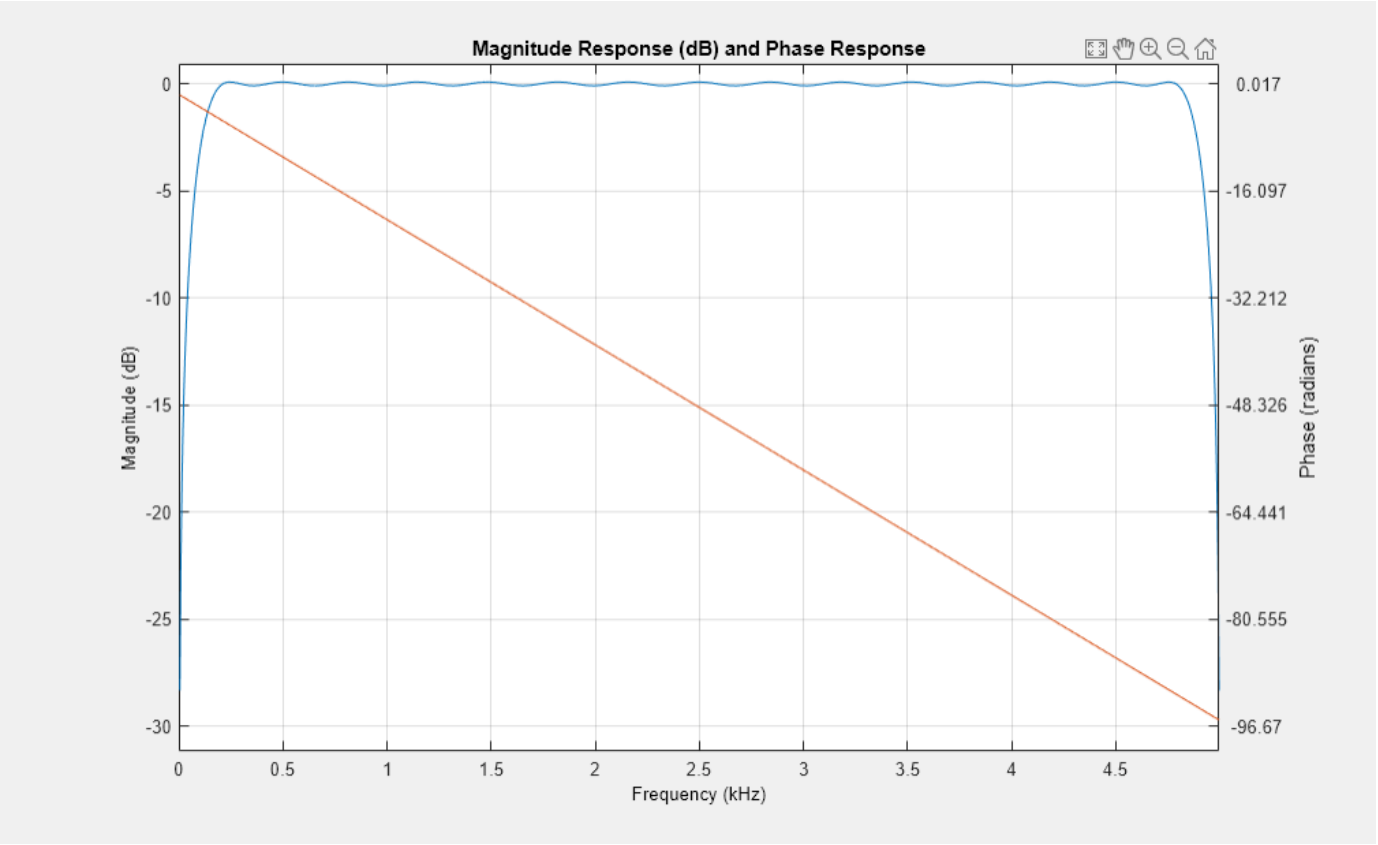
Compute Welch estimates of the power spectral densities of the original sequence and the analytic signal. Divide the sequences into Hamming-windowed, nonoverlapping sections of length 256. Verify that the analytic signal has no power at negative frequencies.

```
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.

```
fo = 60;

d = designfilt('hilbertfir','FilterOrder',fo, ...
       'TransitionWidth',400,'SampleRate',fs);

freqz(d,1024,fs)
```

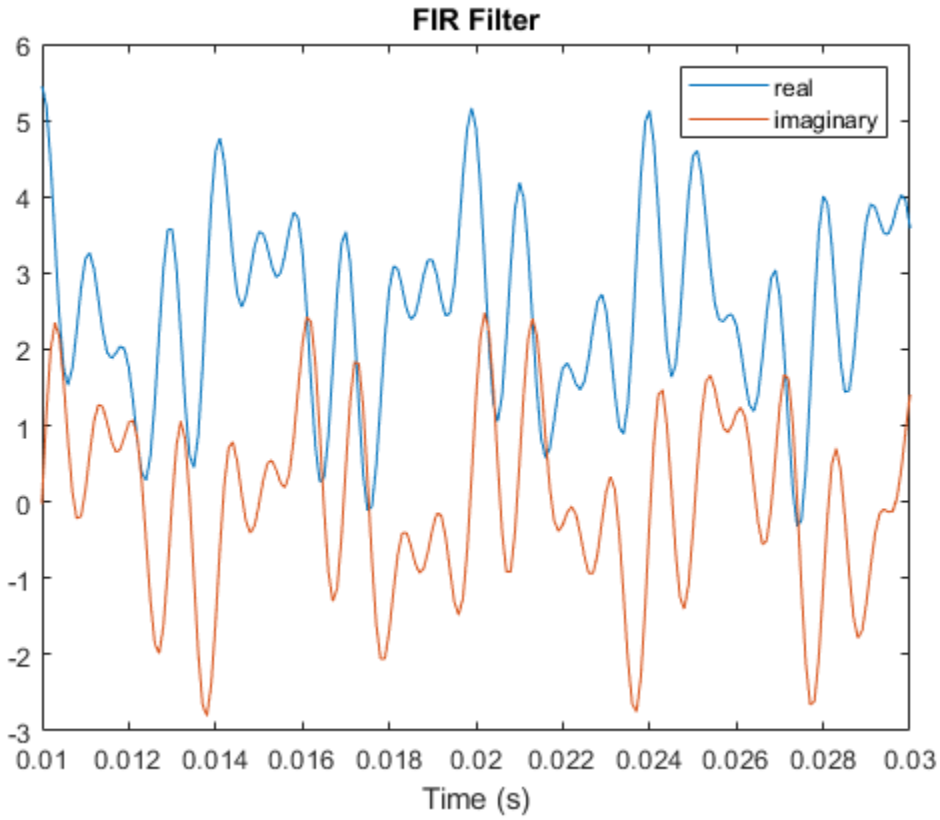


Filter the sinusoidal sequence to approximate the imaginary part of the analytic signal.

```
hb = filter(d,x);
```

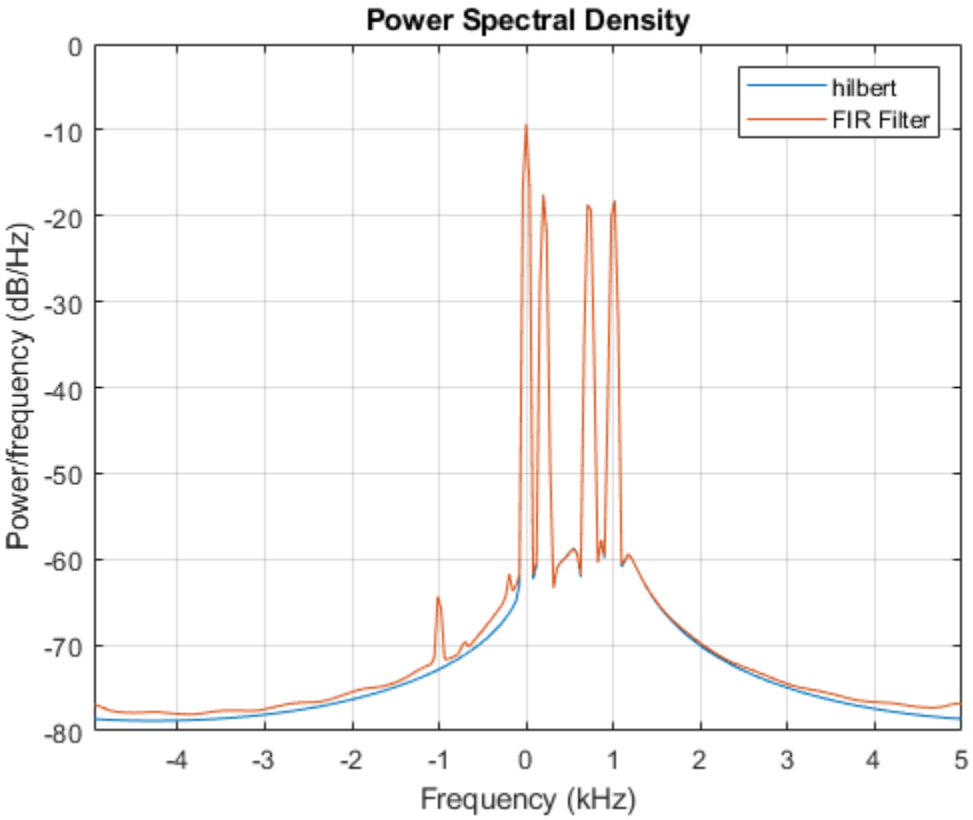
The group delay of the filter, `grd`, is equal to one-half the filter order. Compensate for this delay. Remove the first `grd` samples of the imaginary part and the last `grd` samples of the real part and the time vector. Plot the result between 0.01 seconds and 0.03 seconds.

```
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')
xlabel('Time (s)')
```



Estimate the power spectral density (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')
```



Input Arguments

`xr` — Input signal

vector | matrix

Input signal, specified as a real-valued vector or matrix. If `xr` is complex, then `hilbert` ignores its imaginary part.

Example: `sin(2*pi*(0:15)/16)` specifies one period of a sinusoid.

Example: `sin(2*pi*(0:15)'./[16 8])` specifies a two-channel sinusoidal signal.

Data Types: `single` | `double`

`n` — DFT length

positive integer scalar

DFT length, specified as a positive integer scalar.

Data Types: `single` | `double`

Output Arguments

`x` — Analytic signal

vector | matrix

Analytic signal, returned as a vector or matrix.

More About

Analytic Signal

`hilbert` returns a complex helical sequence, sometimes called the *analytic signal*, from a real data sequence.

The analytic signal $x = x_r + jx_i$ has a real part, x_r, which is the original data, and an imaginary part, x_i, 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, cosines are transformed to sines. The Hilbert-transformed series has the same amplitude and frequency content as the original sequence. The transform includes phase information that depends on the phase of the original.

The Hilbert transform is useful in calculating instantaneous attributes of a time series, especially the amplitude and the 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.

Algorithms

The analytic signal for a sequence `xr` 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.

`hilbert` uses a four-step algorithm:

1. Calculate the FFT of the input sequence, storing the result in a vector `x`.
2. Create a vector `h` whose elements `h(i)` have the values:
   - 1 for `i` = 1, `(n/2)+1`
   - 2 for `i` = 2, 3, ... , `(n/2)`
   - 0 for `i` = `(n/2)+2`, ... , `n`
3. Calculate the element-wise product of `x` and `h`.
4. Calculate 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.

[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. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`fft` | `ifft` | `rceps`

Introduced before R2006a

icceps

Inverse complex cepstrum

Syntax

```
x = icceps(xhat,nd)
```

Description

`x = icceps(xhat,nd)` returns the inverse complex cepstrum of the real data sequence `xhat`, removing `nd` samples of delay.

Examples

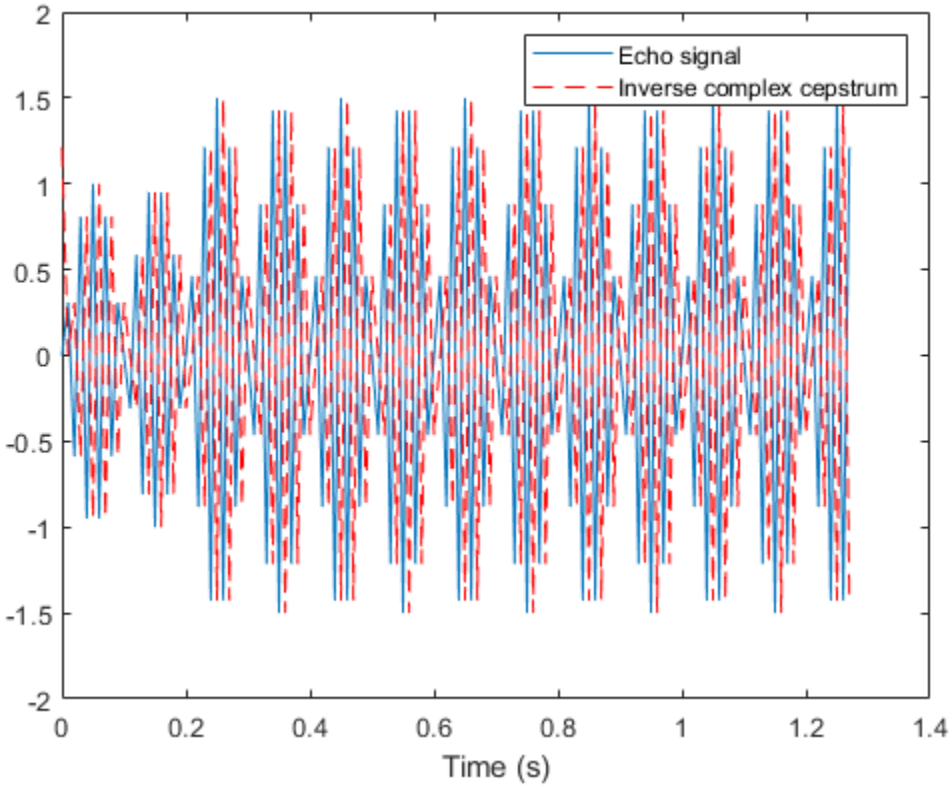
Inverse Complex Cepstrum of Echo

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.

```
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);
```

Compute the inverse complex cepstrum. Plot the echo data and its inverse complex cepstrum.

```
x = icceps(c);
plot(t,s2,t,x,'r--')
xlabel('Time (s)')
legend('Echo signal','Inverse complex cepstrum')
```



Input Arguments

`xhat` — Data sequence

real vector

Data sequence, specified as a real vector. If `xhat` was obtained with `cceps`, then the amount of delay that was added to `x` was the element of `round(unwrap(angle(fft(x)))/pi)` corresponding to π radians.

`nd` — Number of samples of delay

real positive scalar

Number of samples of delay, specified as a real positive scalar.

Output Arguments

`x` — Inverse complex cepstrum

vector

Inverse complex cepstrum, returned as a vector.

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`cceps` | `hilbert` | `rceps` | `unwrap`

Introduced before R2006a

idct

Inverse discrete cosine transform

Syntax

```
x = idct(y)
x = idct(y,n)

x = idct(y,n,dim)

y = idct( ___ ,'Type',dcttype)
```

Description

`x = idct(y)` returns the inverse discrete cosine transform of input array `y`. The output `x` has the same size as `y`. If `y` has more than one dimension, then `idct` operates along the first array dimension with size greater than 1.

`x = idct(y,n)` zero-pads or truncates the relevant dimension of `y` to length `n` before transforming.

`x = idct(y,n,dim)` computes the transform along dimension `dim`. To input a dimension and use the default value of `n`, specify the second argument as empty, `[]`.

`y = idct( ___ ,'Type',dcttype)` specifies the type of inverse discrete cosine transform to compute. See “Inverse Discrete Cosine Transform” on page 1-1040 for details. This option can be combined with any of the previous syntaxes.

Examples

Signal Reconstruction Using Inverse Discrete Cosine Transform

Generate a signal that consists of a 25 Hz sinusoid sampled at 1000 Hz for 1 second. The sinusoid is embedded in white Gaussian noise with variance 0.01.

```
rng('default')

Fs = 1000;
t = 0:1/Fs:1-1/Fs;
x = sin(2*pi*25*t) + randn(size(t))/10;
```

Compute the discrete cosine transform of the sequence. Determine how many of the 1000 DCT coefficients are significant. Choose 1 as the threshold for significance.

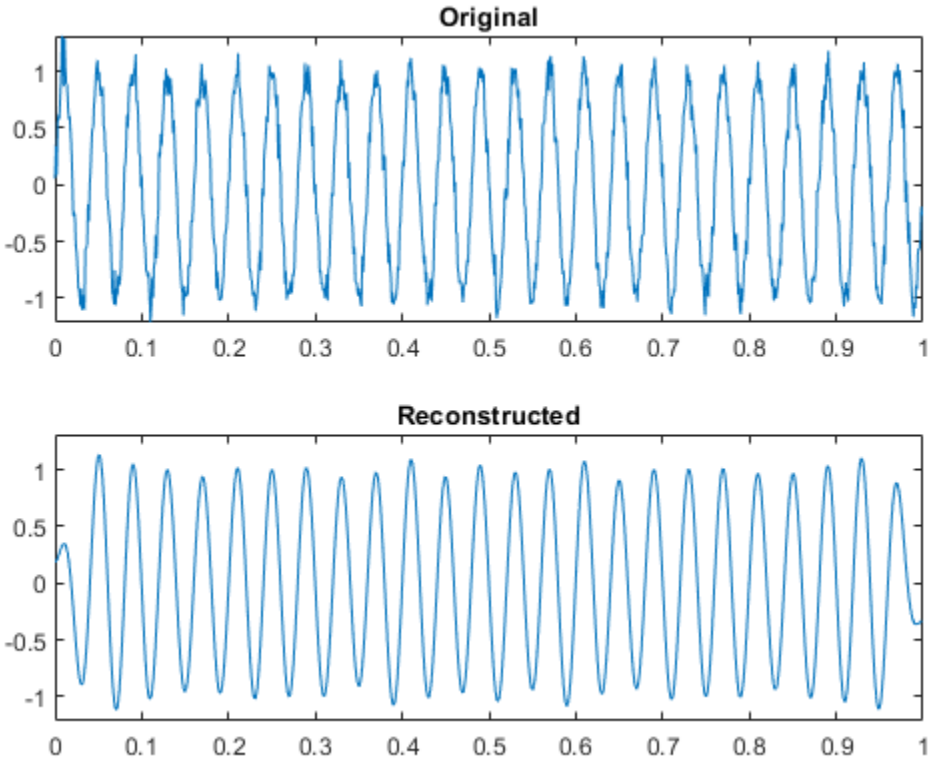
```
y = dct(x);

sigcoeff = abs(y) >= 1;

howmany = sum(sigcoeff)

howmany = 17
```

Reconstruct the signal using only the significant components.

```
y(~sigcoeff) = 0;

z = idct(y);
```

Plot the original and reconstructed signals.

```
subplot(2,1,1)
plot(t,x)
yl = ylim;
title('Original')

subplot(2,1,2)
plot(t,z)
ylim(yl)
title('Reconstructed')
```



DCT Orthogonality

Verify that the different variants of the discrete cosine transform are orthogonal, using a random signal as a benchmark.

Start by generating the signal.

```
s = randn(1000,1);
```

Verify that DCT-1 and DCT-4 are their own inverses.

```
dct1 = dct(s,'Type',1);
idt1 = idct(s,'Type',1);

max(abs(dct1-idt1))
```

```
ans = 1.1102e-15
```

```
dct4 = dct(s,'Type',4);
idt4 = idct(s,'Type',4);

max(abs(dct4-idt4))
```

```
ans = 1.3323e-15
```

Verify that DCT-2 and DCT-3 are inverses of each other.

```
dct2 = dct(s,'Type',2);
idt2 = idct(s,'Type',3);

max(abs(dct2-idt2))
```

```
ans = 4.4409e-16
```

```
dct3 = dct(s,'Type',3);
idt3 = idct(s,'Type',2);

max(abs(dct3-idt3))
```

```
ans = 1.5543e-15
```

Input Arguments

y — Input discrete cosine transform

vector | matrix | *N*-D array | `gpuArray` object

Input discrete cosine transform, specified as a real-valued or complex-valued vector, matrix, *N*-D array, or `gpuArray` object.

See “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox) and “GPU Support by Release” (Parallel Computing Toolbox) for details on `gpuArray` objects.

Example: `dct(sin(2*pi*(0:255)/4))` specifies the discrete cosine transform of a sinusoid.

Example: `dct(sin(2*pi*[0.1;0.3]*(0:39))')` specifies the discrete cosine transform of a two-channel sinusoid.

Data Types: `single` | `double`
Complex Number Support: Yes

n — Inverse transform length

positive integer scalar

Inverse transform length, specified as a positive integer scalar.

Data Types: `single` | `double`

`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar.

Data Types: `single` | `double`

`dcttype` — Inverse discrete cosine transform type
2 (default) | 1 | 3 | 4

Inverse discrete cosine transform type, specified as a positive integer scalar from 1 to 4.

Data Types: `single` | `double`

Output Arguments

`x` — Inverse discrete cosine transform
vector | matrix | *N*-D array | `gpuArray` object

Inverse discrete cosine transform, returned as a real-valued or complex-valued vector, matrix, *N*-D array, or `gpuArray` object.

More About

Inverse Discrete Cosine Transform

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.

The DCT has four standard variants. For a transformed signal y of length N, and with $\delta_{k\ell}$ the Kronecker delta, the inverses are defined by:

- Inverse of DCT-1:

$$x(n) = \sqrt{\frac{2}{N-1}} \sum_{k=1}^{N} y(k) \frac{1}{\sqrt{1+\delta_{k1}+\delta_{kN}}} \frac{1}{\sqrt{1+\delta_{n1}+\delta_{nN}}} \cos\left(\frac{\pi}{N-1}(k-1)(n-1)\right)$$

- Inverse of DCT-2:

$$x(n) = \sqrt{\frac{2}{N}} \sum_{k=1}^{N} y(k) \frac{1}{\sqrt{1+\delta_{k1}}} \cos\left(\frac{\pi}{2N}(k-1)(2n-1)\right)$$

- Inverse of DCT-3:

$$x(n) = \sqrt{\frac{2}{N}} \sum_{k=1}^{N} y(k) \frac{1}{\sqrt{1+\delta_{n1}}} \cos\left(\frac{\pi}{2N}(2k-1)(n-1)\right)$$

- Inverse of DCT-4:

$$x(n) = \sqrt{\frac{2}{N}} \sum_{k=1}^{N} y(k) \cos\left(\frac{\pi}{4N}(2k-1)(2n-1)\right)$$

The series are 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 $N – 1$.

All variants of the DCT are *unitary* (or, equivalently, *orthogonal*): To find the forward transforms, switch k and n in each definition. DCT-1 and DCT-4 are their own inverses. DCT-2 and DCT-3 are inverses of each other.

References

[1] Jain, A. K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

[2] Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

[3] Pennebaker, W. B., and J. L. Mitchell. *JPEG Still Image Data Compression Standard*. New York: Van Nostrand Reinhold, 1993.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- C and C++ code generation for `dct` requires DSP System Toolbox software.
- The length of the transform dimension must be a power of two. If specified, the pad or truncation value must be constant. Expressions or variables are allowed if their values do not change.
- Inputs must be double precision.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- *N*-D input arrays are not supported.
- The `dim` and `dcttype` input arguments are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`dct` | `dct2` | `idct2` | `ifft`

Topics

“DCT for Speech Signal Compression”

Introduced before R2006a

ifsst

Inverse Fourier synchrosqueezed transform

Syntax

```
x = ifsst(s)

x = ifsst(s,window)

x = ifsst(s,window,f,freqrange)

x = ifsst(s,window,iridge)
x = ifsst(s,window,iridge,'NumFrequencyBins',nbins)
```

Description

`x = ifsst(s)` returns the inverse Fourier synchrosqueezed transform of `s`. `x` is reconstructed using the entire time-frequency plane in `s`.

`x = ifsst(s,window)` reconstructs the signal whose Fourier synchrosqueezed transform was computed using `window`.

`x = ifsst(s,window,f,freqrange)` inverts the synchrosqueezed transform assuming it was sampled at the frequencies `f`, which lie within `freqrange`. The synchrosqueezed transform is inverted for the bins in `s` whose frequencies are within `freqrange`.

`x = ifsst(s,window,iridge)` inverts the synchrosqueezed transform along the time-frequency ridges specified by the index vector or matrix `iridge`. If `iridge` is a matrix, then `ifsst` initially performs the inversion along the first column of `iridge` and then proceeds iteratively along the subsequent columns. The output is a vector or matrix with the same size as `iridge`.

`x = ifsst(s,window,iridge,'NumFrequencyBins',nbins)` specifies the number of frequency bins around the indices in `iridge` to use in the reconstruction.

Examples

Inverse Fourier Synchrosqueezed Transform of Speech Signal

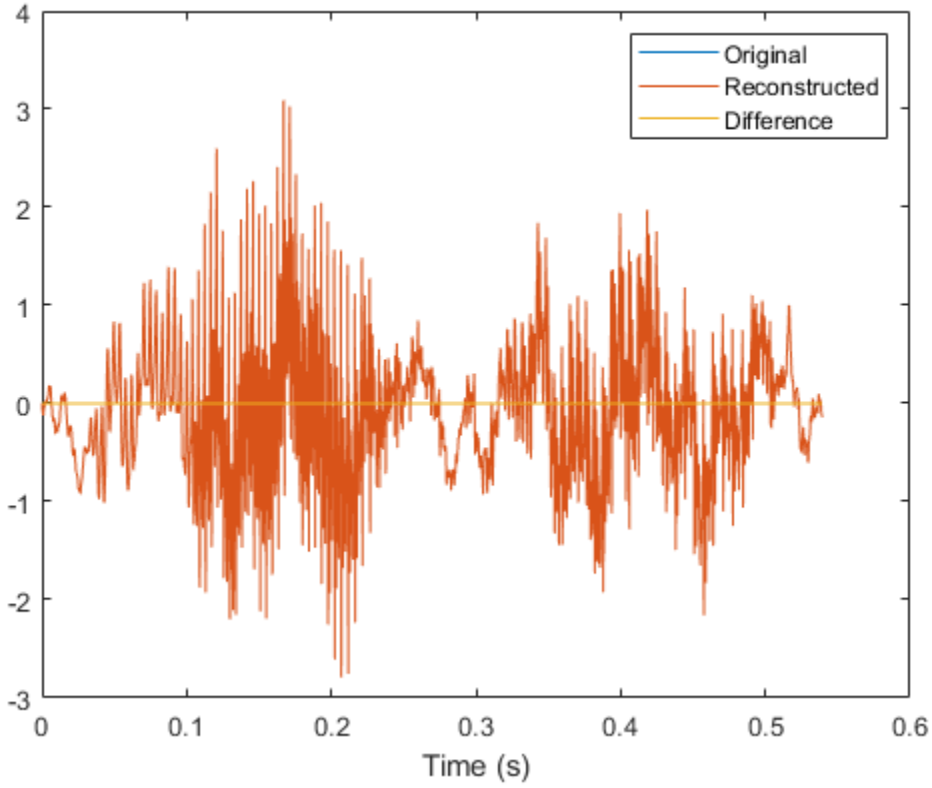
Load a speech signal sampled at F_s = 7418 Hz. The file contains a recording of a female voice saying the word "MATLAB®." Compute the Fourier synchrosqueezed transform of the signal.

```
load mtlb         % To hear, type sound(mtlb,Fs)

[sst,f] = fsst(mtlb,Fs);
```

Invert the transform to reconstruct the signal. Plot the original and reconstructed signals, as well as the difference between them.

```
xrec = ifsst(sst);
```

```
t = (0:length(mtlb)-1)/Fs;
plot(t,mtlb,t,xrec,t,mtlb-xrec)

xlabel('Time (s)')
legend('Original','Reconstructed','Difference')
```



Check the accuracy of the reconstruction by computing the ℓ_∞ norm of the difference between the original signal and the inverse transform.

```
Linf = norm(abs(mtlb-xrec),Inf)
```

```
Linf = 1.9762e-14
```

```
% To hear, type sound(mtlb-xrec,Fs)
```

Fourier Synchrosqueezed Transform and Its Inverse

Generate a signal sampled at 1024 Hz for 2 seconds.

```
nSamp = 2048;
Fs = 1024;
t = (0:nSamp-1)'/Fs;
```

During the first second, the signal consists of a 400 Hz sinusoid and a concave quadratic chirp. Specify a chirp that is symmetric about the interval midpoint, starts and ends at a frequency of 250 Hz, and attains a minimum of 150 Hz.

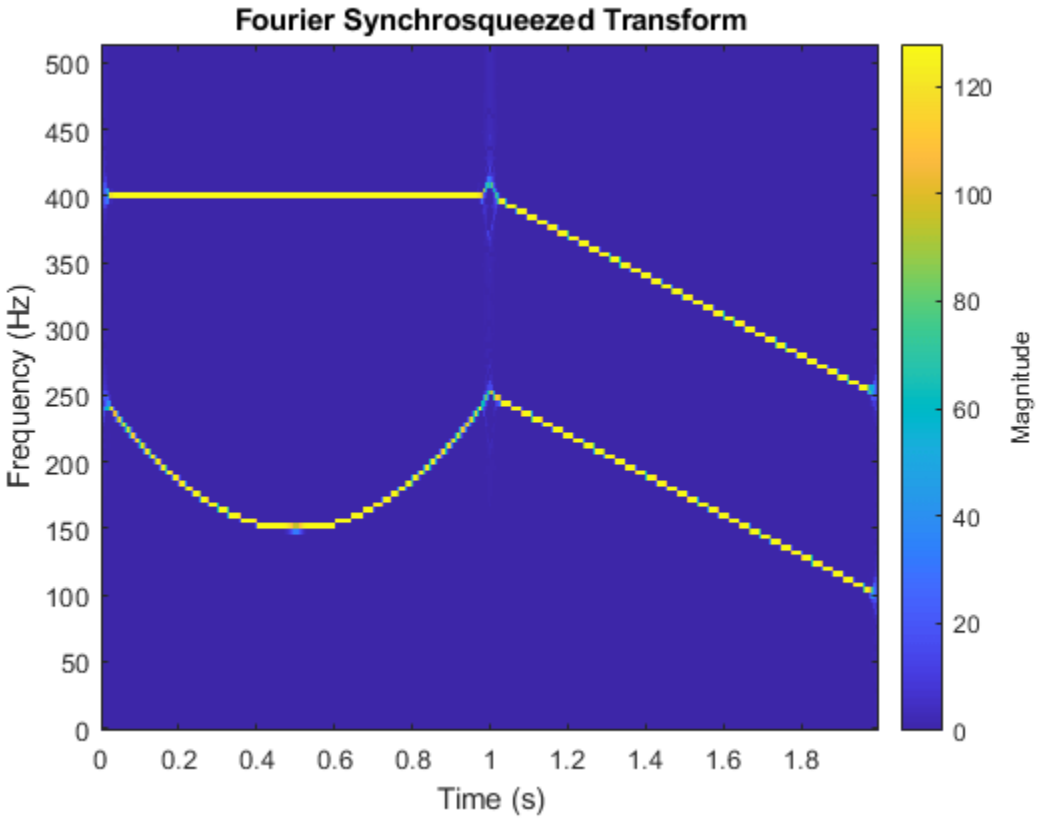
```
t1 = t(1:nSamp/2);

x11 = sin(2*pi*400*t1);
x12 = chirp(t1-t1(nSamp/4),150,nSamp/Fs,1750,'quadratic');
x1 = x11+x12;
```

The rest of the signal consists of two linear chirps of decreasing frequency. One chirp has an initial frequency of 250 Hz that decreases to 100 Hz. The other chirp has an initial frequency of 400 Hz that decreases to 250 Hz.

```
t2 = t(nSamp/2+1:nSamp);

x21 = chirp(t2,400,nSamp/Fs,100);
x22 = chirp(t2,550,nSamp/Fs,250);
x2 = x21+x22;
```

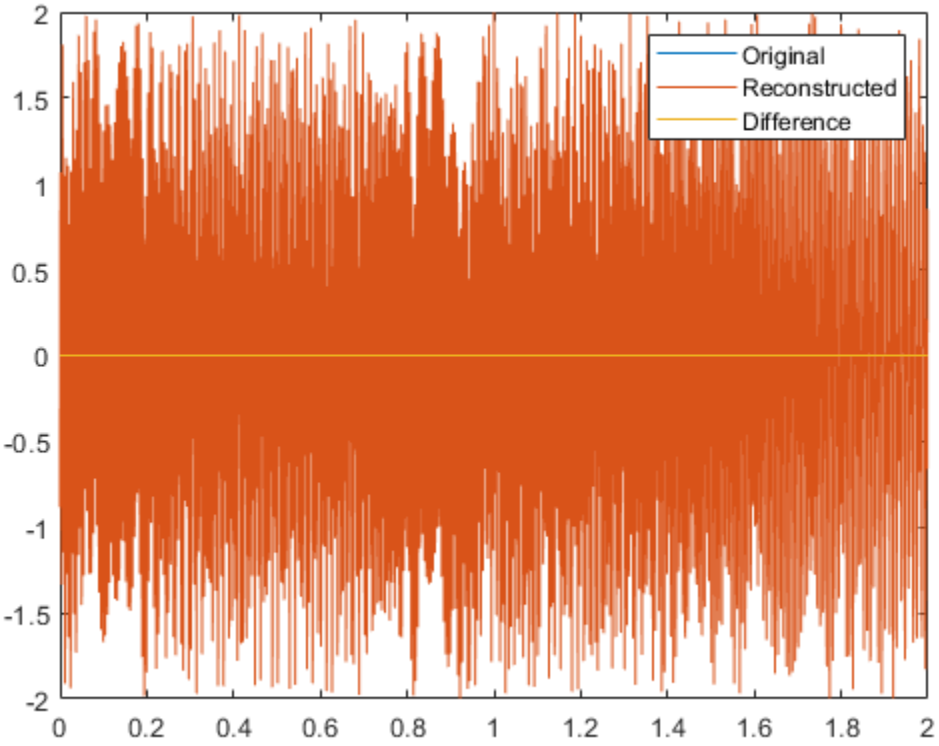
Compute the Fourier synchrosqueezed transform of the signal. Specify a 256-sample Kaiser window with a shape parameter $\beta = 100$. Use the plotting functionality of `fsst` to display the result.

```
sig = [x1;x2];
wind = kaiser(256,120);

[sigtr,ftr,ttr] = fsst(sig,Fs,wind);

fsst(sig,Fs,wind,'yaxis')
```



Invert the transform to reconstruct the function. Plot the original and inverted signals and the difference between them.

```
x = ifsst(sigtr,wind);

plot(t,sig,t,x,t,x-sig)
legend('Original','Reconstructed','Difference')
```




```
diffnorm = norm(x-sig)
```

```
diffnorm = 3.9032e-13
```

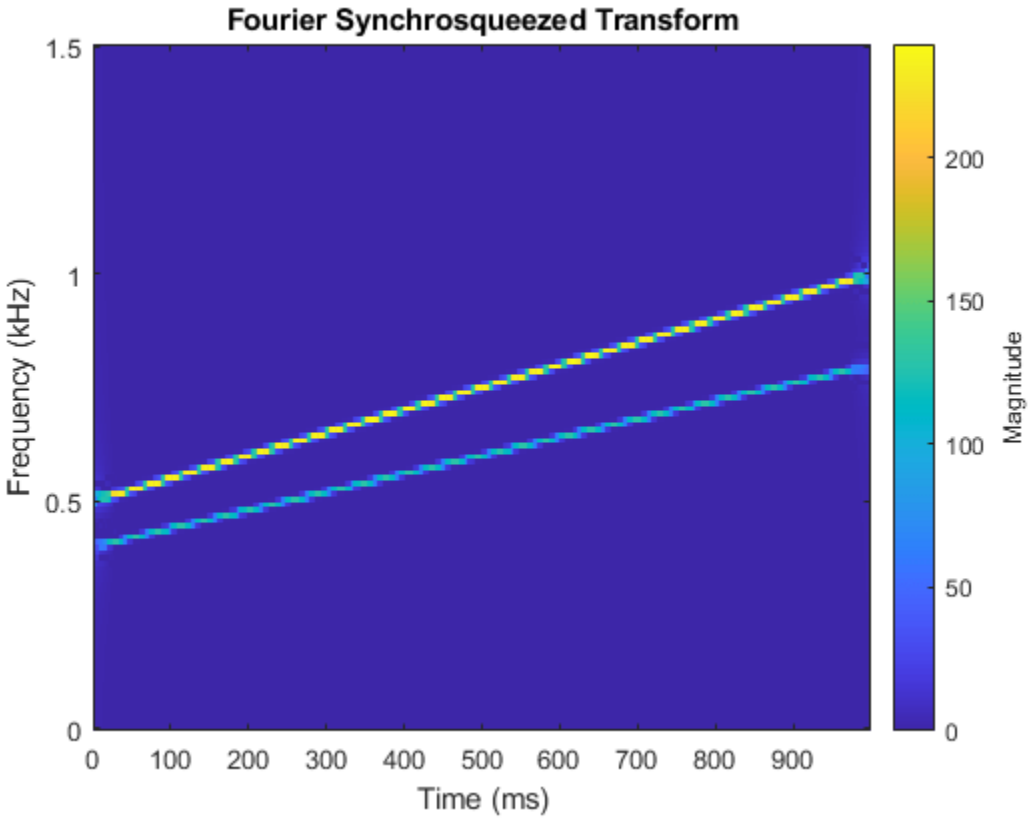
Reconstruction of Linear Chirps

Generate a signal that consists of two chirps. The signal is sampled at 3 kHz for one second. The first chirp has an initial frequency of 400 Hz and reaches 800 Hz at the end of the sampling. The second chirp starts at 500 Hz and reaches 1000 Hz at the end. The second chirp has twice the amplitude of the first chirp.

```
fs = 3000;
t = 0:1/fs:1-1/fs;

x1 = chirp(t,400,t(end),800);
x2 = 2*chirp(t,500,t(end),1000);
```

Compute and plot the Fourier synchrosqueezed transform of the signal. Display the time on the *x*-axis and the frequency on the *y*-axis.

```
[sst,f] = fsst(x1+x2,fs);
fsst(x1+x2,fs,'yaxis')
```

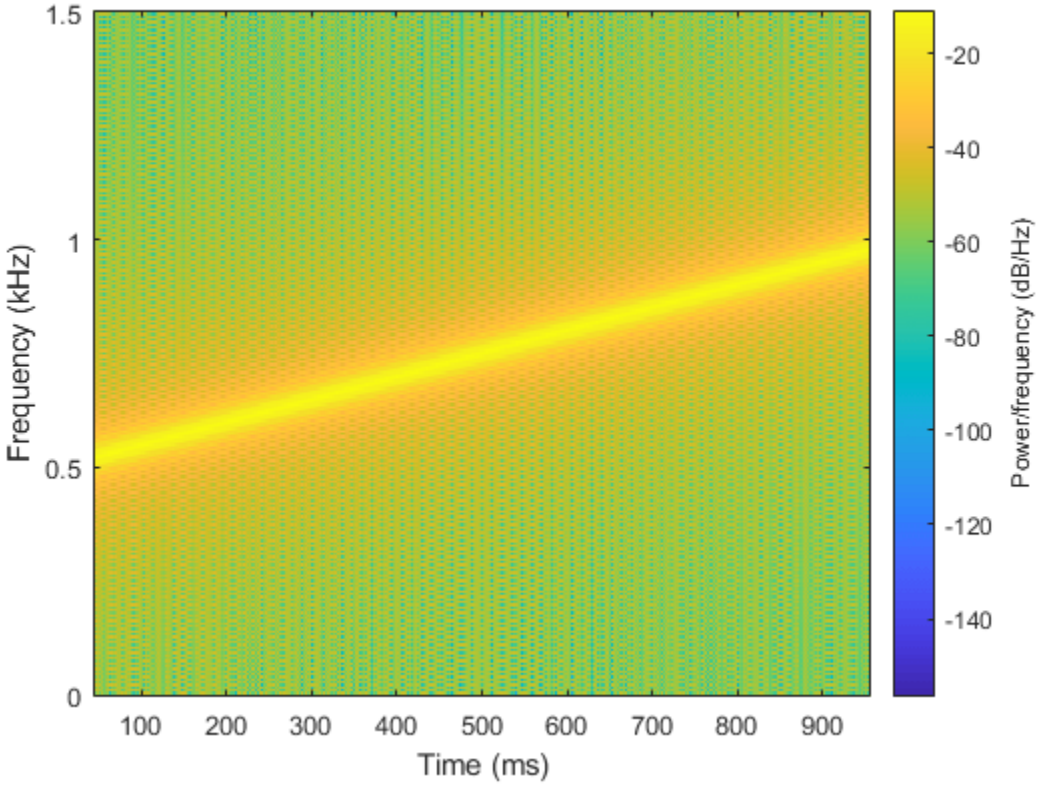


Extract the ridge corresponding to the higher-energy component of the signal, which is the chirp with the larger amplitude. Use the ridge to reconstruct the signal.

```
[~,iridge] = tfridge(sst,f);

xrec = ifsst(sst,[],iridge);
```

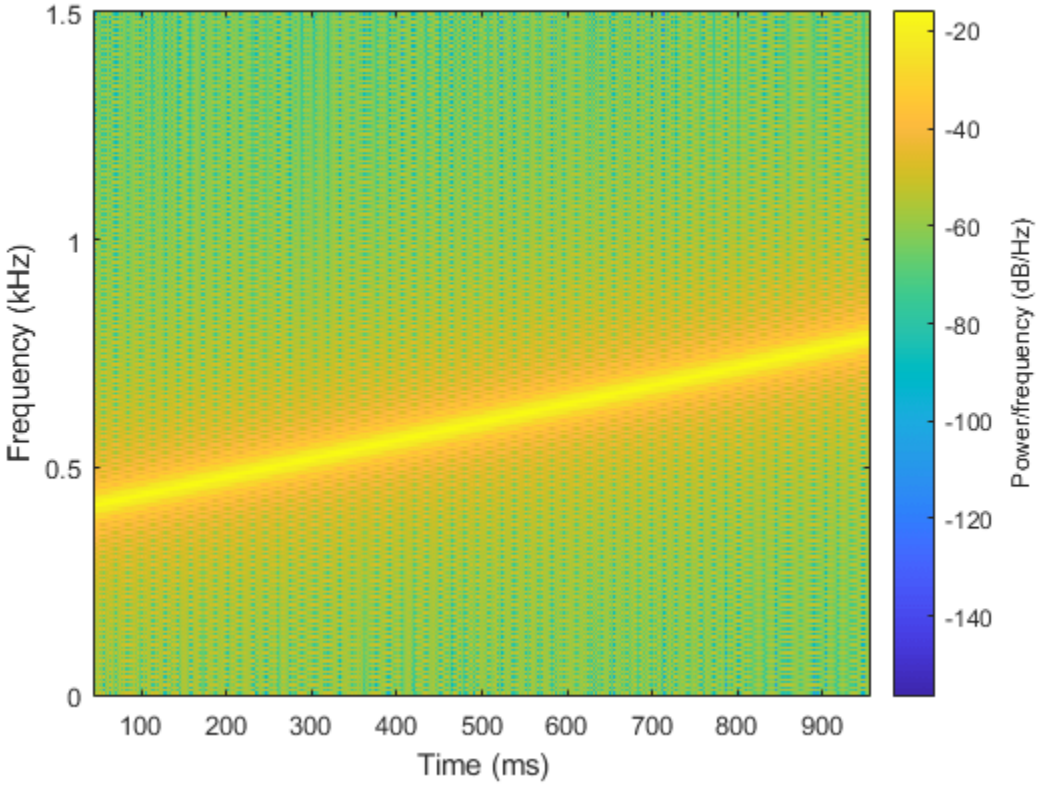
Plot the spectrogram for the higher-energy component. Divide the component into 256-sample sections and specify an overlap of 255 samples. Use 512 DFT points and a rectangular window.

```
spectrogram(xrec,rectwin(256),255,512,fs,'yaxis')
```



To extract the second chirp, specify that `tfridge` search for two ridges. The second column of the output is the lower-energy component of the signal.

```
[~,iridge] = tfridge(sst,f,'NumRidges',2);

xrec = ifsst(sst,[],iridge(:,2));

spectrogram(xrec,rectwin(256),255,512,fs,'yaxis')
```



Input Arguments

`s` — Input synchrosqueezed transform

matrix

Input synchrosqueezed transform, specified as a matrix.

Example: `fsst(cos(pi/4*(0:159)))` specifies the synchrosqueezed transform of a sinusoid.

Data Types: `single` | `double`
Complex Number Support: Yes

`window` — Spectral window

`kaiser(256,10)` (default) | integer | vector | `[]`

Spectral window, specified as an integer or as a row or column vector.

- If `window` is an integer, then `ifsst` assumes that the synchrosqueezed transform, `s`, was computed using a Kaiser window of length `window` and $\beta = 10$.
- If `window` is a vector, then `ifsst` assumes that `s` was computed by windowing each segment of the original signal using `window`.
- If `window` is not specified, then `ifsst` assumes that `s` was computed using a Kaiser window of length 256 and $\beta = 10$. If the signal to be reconstructed, `x`, has fewer than 256 samples, then you must provide a window length or window vector consistent with the length of `x`.

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: `double` | `single`

`f` — Sampling frequencies
vector

Sampling frequencies, specified as a vector. The length of `f` must equal the number of elements in `s`.

Data Types: `single` | `double`

`freqrange` — Frequency range
two-element vector

Frequency range, specified as a two-element vector. The values of `freqrange` must be strictly increasing and must lie in the range comprised by `f`.

Data Types: `single` | `double`

`iridge` — Time-frequency ridge indices
vector | matrix

Time-frequency ridge indices, specified as a vector or matrix. `iridge` is an output of `tfridge`.

Data Types: `single` | `double`

`nbins` — Number of neighboring bins
4 (default) | positive integer scalar

Number of neighboring bins on either side of the time-frequency ridges of interest, specified as the comma-separated pair consisting of `'NumFrequencyBins'` and a positive integer scalar. Indices close to the frequency edges that have fewer than `nbins` bins on one side are reconstructed using a smaller number of bins.

Data Types: `single` | `double`

Output Arguments

`x` — Inverse synchrosqueezed transform
vector

Inverse synchrosqueezed transform, returned as a vector. The length of `x` equals the number of columns in `s`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If `s` is variable size, you must extract the real part of `x` before inputting it to an audio plugin.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- The length of the window must be smaller than or equal to the length of the input signal.

See Also

Apps

Signal Analyzer

Functions

`fsst` | `pspectrum` | `spectrogram` | `tfridge`

Topics

“Hilbert Transform and Instantaneous Frequency”
“Practical Introduction to Time-Frequency Analysis”
“Detect Closely Spaced Sinusoids”
“Fourier Synchrosqueezed Transform” on page 1-922
“Time-Frequency Gallery”

Introduced in R2016b

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 signal `x`.

`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.

Examples

Walsh-Hadamard Transform for Spectral Analysis and Compression of ECG Signals

Use an electrocardiogram (ECG) signal to illustrate working with the Walsh-Hadamard transform. ECG signals typically are very large and need to be stored for analysis and retrieval at a future time. Walsh-Hadamard transforms are particularly well-suited to this application because they provide compression and thus require less storage space. They also provide rapid signal reconstruction.

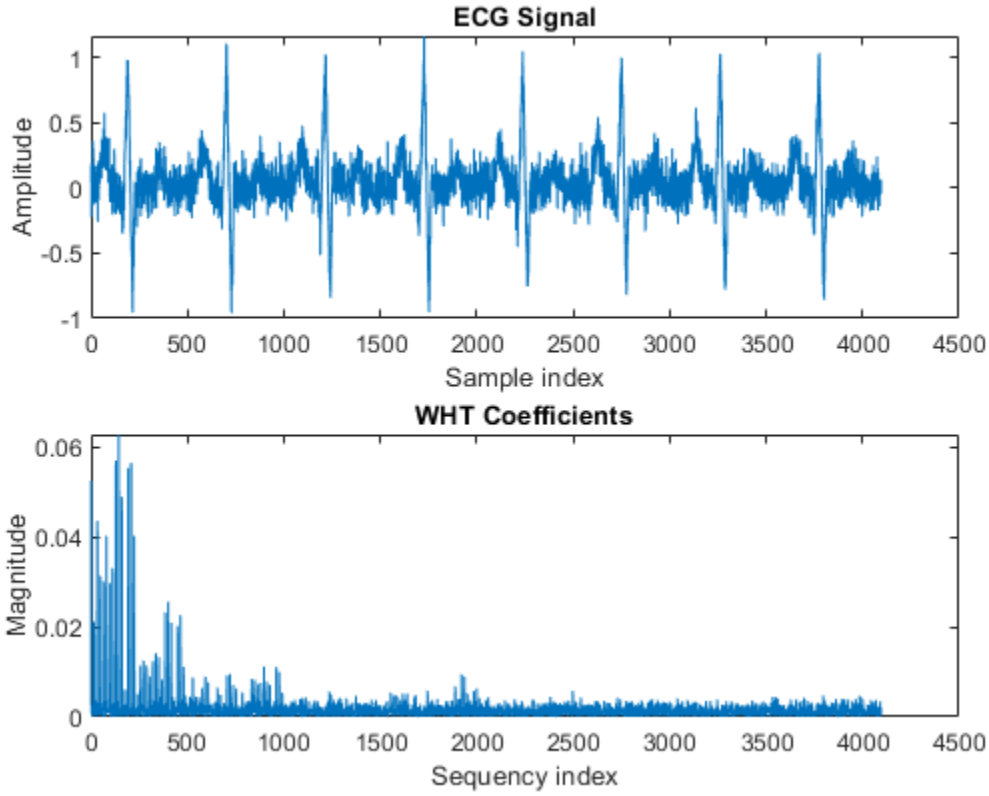
Start with an ECG signal. Replicate it to create a longer signal and insert some additional random noise.

```
xe = ecg(512);
xr = repmat(xe,1,8);
x = xr + 0.1.*randn(1,length(xr));
```

Transform the signal using the fast Walsh-Hadamard transform. Plot the original signal and the transformed signal.

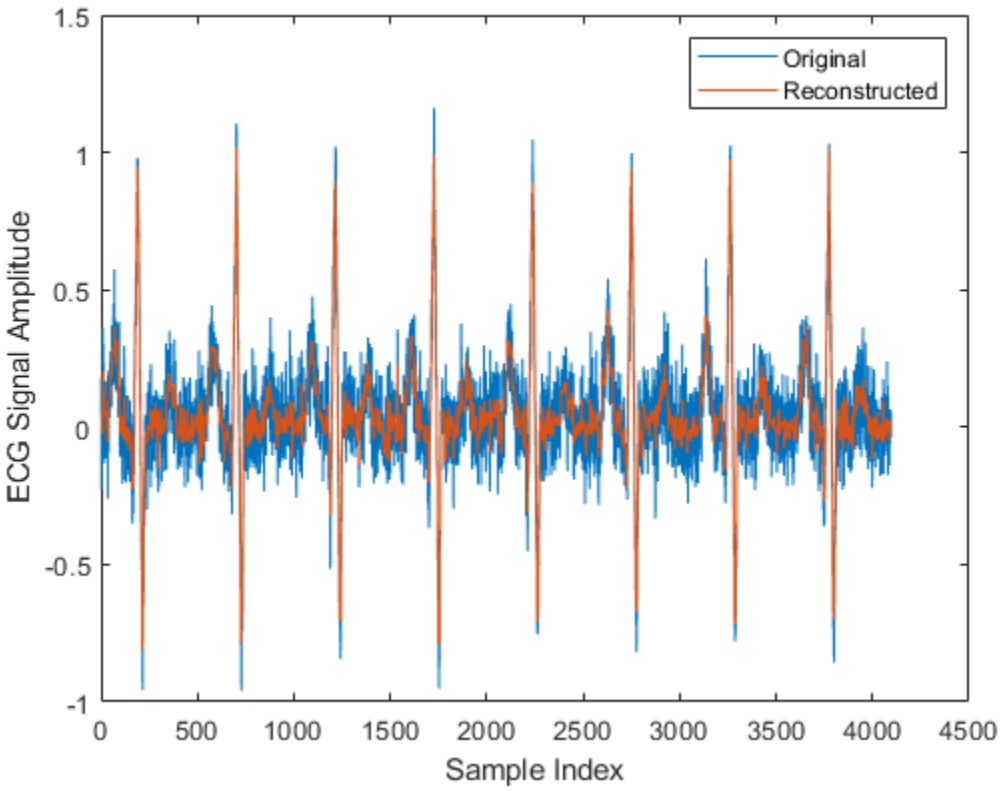
```
y = fwht(x);

subplot(2,1,1)
plot(x)
xlabel('Sample index')
ylabel('Amplitude')
title('ECG Signal')

subplot(2,1,2)
plot(abs(y))
xlabel('Sequency index')
```

```
ylabel('Magnitude')
title('WHT Coefficients')
```



The plot shows that most of the signal energy is in the lower sequency values, below approximately 1100. Store only the first 1024 coefficients (out of 4096). Try to reconstruct the signal accurately from only these stored coefficients.

```
y(1025:length(x)) = 0;
xHat = ifwht(y);

figure
plot(x)
hold on
plot(xHat)
xlabel('Sample Index')
ylabel('ECG Signal Amplitude')
legend('Original','Reconstructed')
```



The reproduced signal is very close to the original but has been compressed to a quarter of the size. Storing more coefficients is a tradeoff between increased resolution and increased noise, while storing fewer coefficients can cause loss of peaks.

Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If x is a matrix, the inverse fast Walsh-Hadamard transform is calculated on each column of x. The function operates only on signals with length equal to a power of 2. If the length of x is less than a power of 2, the function pads x with zeros to the next greater power of two before processing.

Data Types: `single` | `double`

n — Number of points in inverse discrete Walsh-Hadamard transform

positive even integer scalar

Number of points in inverse discrete Walsh-Hadamard transform, specified as a positive even integer scalar. n must be a power of 2.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ordering` — Order of Walsh

`"sequency"` (default) | `"hadamard"` | `"dyadic"`

Order of Walsh-Hadamard transform coefficients, specified as one of these:

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.

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`. For more information on the Walsh functions and ordering, see “Walsh-Hadamard Transform”.

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. For more details, see the references.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`fwht` | `dct` | `idct` | `fft` | `ifft`

Introduced in R2008b

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`, where `fs` is the sample rate.

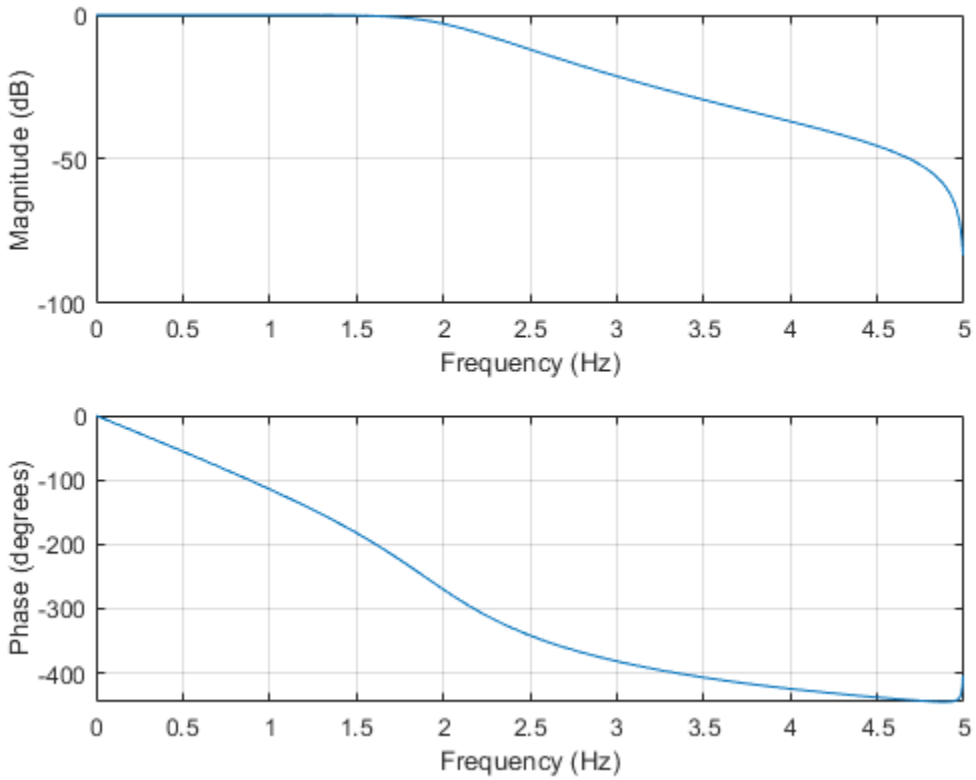
`[bz,az] = impinvar(b,a,fs,tol)` uses the tolerance specified by `tol` to determine whether poles are repeated.

Examples

Analog and Digital Butterworth Filters

Convert a sixth-order analog Butterworth lowpass filter to a digital filter using impulse invariance. Specify a sample rate of 10 Hz and a cutoff frequency of 2 Hz. Display the frequency response of the filter.

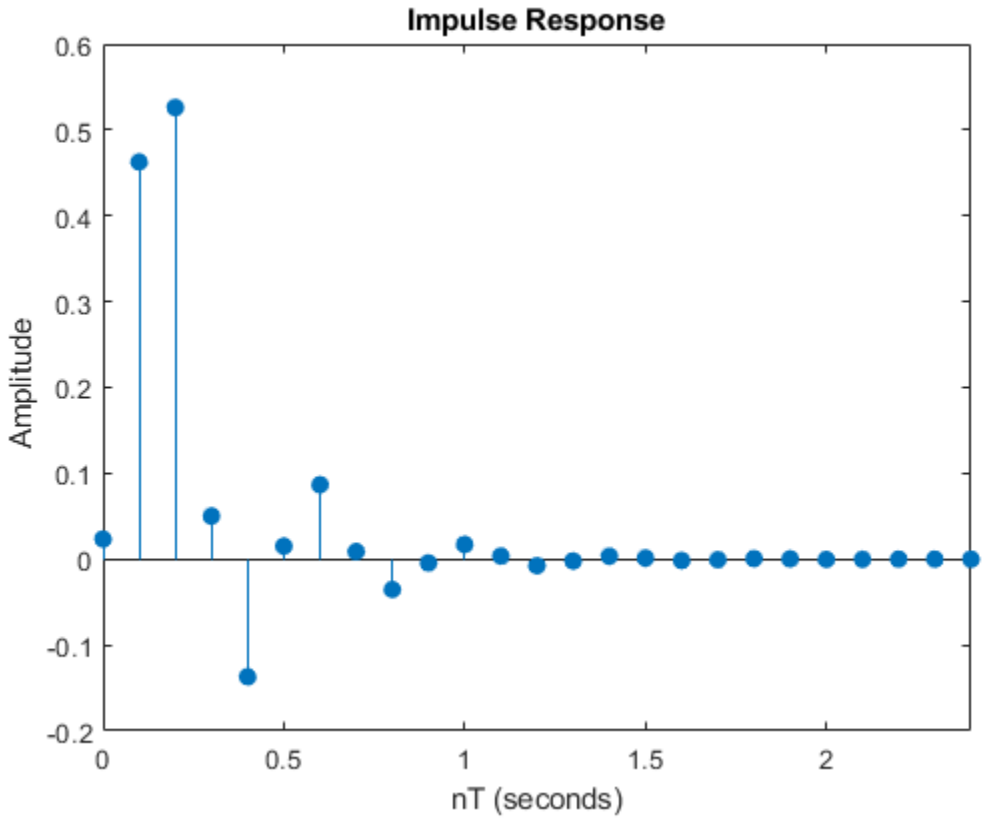
```
f = 2;
fs = 10;

[b,a] = butter(6,2*pi*f,'s');
[bz,az] = impinvar(b,a,fs);

freqz(bz,az,1024,fs)
```



Analog and Digital Impulse Responses

Convert a third-order analog elliptic filter to a digital filter using impulse invariance. Specify a sample rate f_s = 10 Hz, a passband edge frequency of 2.5 Hz, a passband ripple of 1 dB, and a stopband attenuation of 60 dB. Display the impulse response of the digital filter.

```
fs = 10;

[b,a] = ellip(3,1,60,2*pi*2.5,'s');
[bz,az] = impinvar(b,a,fs);

impz(bz,az,[],fs)
```

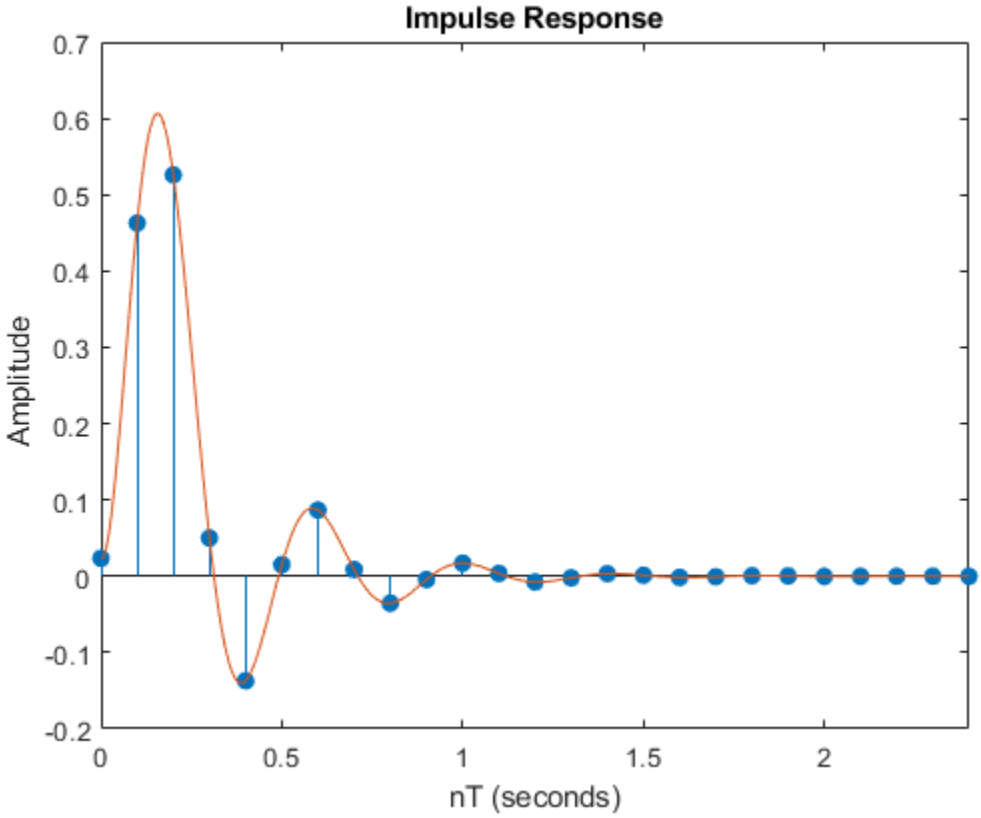


Derive the impulse response of the analog filter by finding the residues, r_k, and poles, p_k, of the transfer function and inverting the Laplace transform explicitly using

$$H(s) = \sum_k \frac{r_k}{s - p_k} \Longleftrightarrow h(t) = \sum_k r_k e^{p_k t}.$$

Overlay the impulse response of the analog filter. Impulse invariance introduces a gain of $1/f_s$ to the digital filter. Multiply the analog impulse response by this gain to enable meaningful comparison.

```
[r,p] = residue(b,a);
t = linspace(0,4,1000);
h = real(r.'*exp(p.*t)/fs);

hold on
plot(t,h)
hold off
```



Input Arguments

`b, a` — Analog filter transfer function coefficients

vectors

Analog filter transfer function coefficients, specified as vectors.

Example: `[b,a] = butter(6,2*pi*10,'s')` specifies a 6th-order Butterworth filter with a cutoff frequency of 10 Hz.

Data Types: `single` | `double`

`fs` — Sample rate

1 Hz (default) | positive scalar

Sample rate, specified as a positive scalar.

Data Types: `single` | `double`

`tol` — Tolerance

`0.001` (default) | positive scalar

Tolerance, specified as a positive scalar. The tolerance determines whether poles are repeated. A larger tolerance increases the likelihood that `impinvar` interprets closely located poles as multiplicities (repeated ones). The default tolerance corresponds to 0.1% of a pole magnitude. The accuracy of the pole values is still limited to the accuracy obtainable by the `roots` function.

Data Types: `single` | `double`

Output Arguments

bz, az — Digital filter transfer function coefficients

vectors

Digital filter transfer function coefficients, returned as vectors.

Algorithms

`impinvar` performs the impulse-invariant method of analog-to-digital transfer function conversion discussed in reference [2]:

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] Antoniou, Andreas. *Digital Filters*. New York: McGraw-Hill, Inc., 1993.

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

See Also

`bilinear` | `lp2bp` | `lp2bs` | `lp2hp` | `lp2lp`

Introduced before R2006a

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 digital filter with numerator coefficients `b` and denominator coefficients `a`. The function chooses the number of samples and returns the response coefficients in `h` and the sample times in `t`.

`[h,t] = impz(sos)` returns the impulse response of the filter specified by the second-order sections matrix `sos`.

`[h,t] = impz(d)` returns the impulse response of the digital filter `d`. Use `designfilt` to generate `d` based on frequency-response specifications.

`[h,t] = impz(___,n)` specifies what impulse-response samples to compute. You can specify the filter using any of the previous syntaxes.

`[h,t] = impz(___,n,fs)` returns a vector `t` with consecutive samples spaced 1/`fs` units apart.

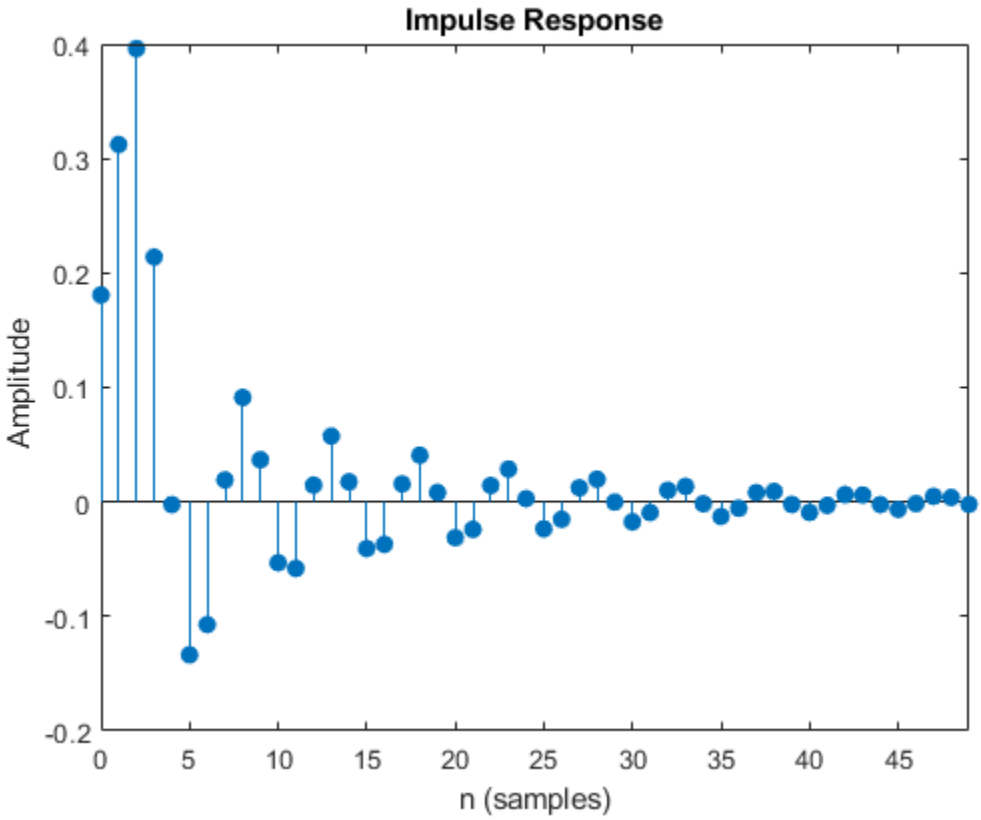
`impz(___)` with no output arguments plots the impulse response of the filter.

Examples

Impulse Response of Elliptic Lowpass Filter

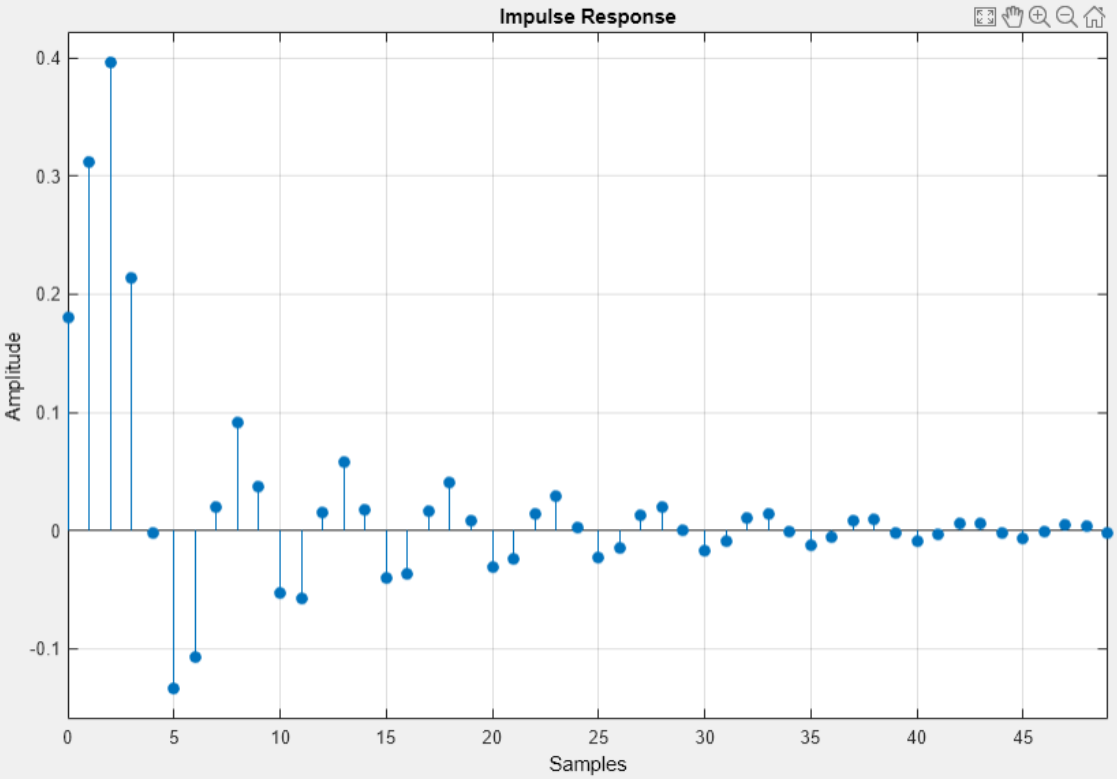
Design a fourth-order lowpass elliptic filter with normalized passband frequency 0.4 rad/sample. Specify a passband ripple of 0.5 dB and a stopband attenuation of 20 dB. Plot the first 50 samples of the impulse response.

```
[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.

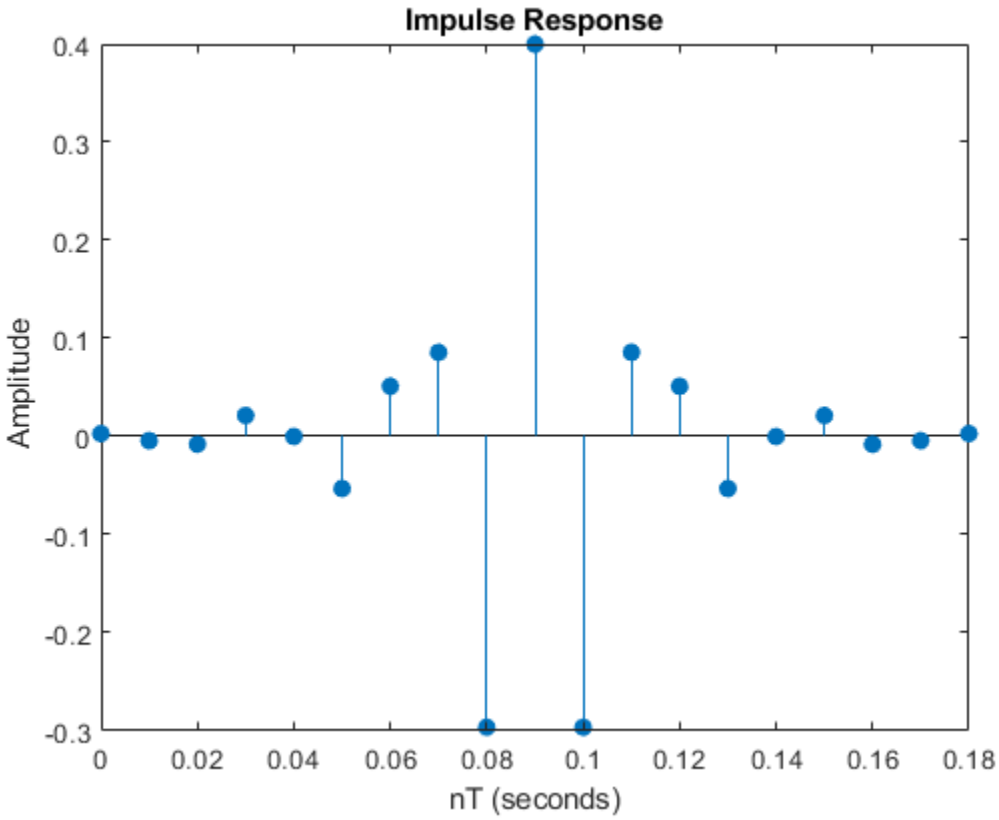
```
d = designfilt('lowpassiir','DesignMethod','ellip','FilterOrder',4, ...
               'PassbandFrequency',0.4, ...
               'PassbandRipple',0.5,'StopbandAttenuation',20);
impz(d,50)
```



Impulse Response of Highpass FIR Filter

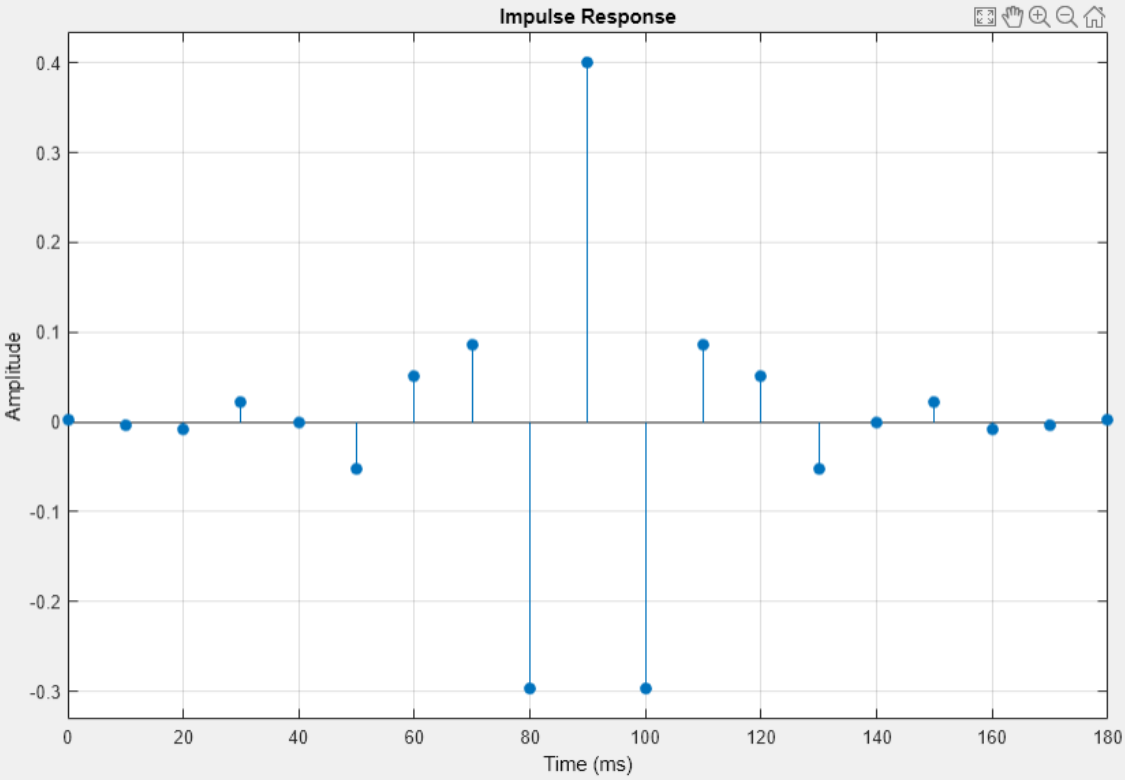
Design an FIR highpass filter of order 18 using a Kaiser window with $\beta = 4$. Specify a sample rate of 100 Hz and a cutoff frequency of 30 Hz. Display the impulse response of the filter.

```
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)
```



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

$$H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})} = \frac{\mathrm{b(1)}+\mathrm{b(2)}\,e^{-j\omega}+\mathrm{b(3)}\,e^{-j2\omega}+\cdots+\mathrm{b(M)}\,e^{-j(M-1)\omega}}{\mathrm{a(1)}+\mathrm{a(2)}\,e^{-j\omega}+\mathrm{a(3)}\,e^{-j2\omega}+\cdots+\mathrm{a(N)}\,e^{-j(N-1)\omega}}.$$

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 0.5π rad/sample.

Data Types: `double` | `single`
Complex Number Support: Yes

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, the function treats the input as 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 0.5π 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 0.5π rad/sample.

`n` — Sample numbers
positive integer | vector of nonnegative integers | `[]`

Sample numbers, specified as a positive integer, a vector of nonnegative integers, or an empty vector.

- If `n` is a positive integer, `impz` computes the first `n` samples of the impulse response and returns `t` as `(0:n-1)'`.
- If `n` is a vector of nonnegative integers, `impz` computes the impulse response at the locations specified in the vector.
- If `n` is an empty vector, `impz` computes the number of samples automatically. See “Algorithms” on page 1-1067 for more information.

Example: `impz([2 4 2 6 0 2;3 3 0 6 0 0],5)` computes the first five samples of the impulse response of a Butterworth filter.

Example: `impz([2 4 2 6 0 2;3 3 0 6 0 0],[0 3 2 1 4 5])` computes the first six samples of the impulse response of a Butterworth filter.

Example: `impz([2 4 2 6 0 2;3 3 0 6 0 0],[],5e3)` computes the impulse response of a Butterworth filter designed to filter signals sampled at 5 kHz.

`fs` — Sample rate
positive scalar

Sample rate, specified as a positive scalar. When the unit of time is seconds, `fs` is expressed in hertz.

Data Types: `double`

Output Arguments

`h` — Impulse response coefficients
column vector

Impulse response coefficients, returned as a column vector.

`t` — Sample times
column vector

Sample times, returned as a column vector.

Algorithms

`impz` filters a length-`n` impulse sequence using

```
filter(b,a,[1 zeros(1,n-1)])
```

and plots the result using `stem`.

Note If the input to `impz` is single precision, the function computes the impulse response using single-precision arithmetic and returns single-precision output.

When `impz` calculates `n` automatically, the algorithm depends on the properties of the filter:

- FIR filters — `n` is the length of `b`.
- IIR filters — `impz` first finds the poles of the transfer function using `roots`.
  - 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 as the point at which the term from the largest-amplitude pole is 5×10^{-5} times its original amplitude.
  - If the filter is oscillatory with 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 the greater of five periods of the slowest oscillation, or the point at which the term due to the largest pole is 5×10^{-5} times its original amplitude.

`impz` also allows for delays in the numerator polynomial. The number of delays is incorporated into the computation of the number of samples.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If the first input to `impz` is a variable-size matrix at compile time, then it must not become a vector at runtime.

See Also

`designfilt` | `digitalFilter` | `impulse` | `impzlength` | `stem`

Introduced before R2006a

impzlength

Impulse response length

Syntax

```
len = impzlength(b,a)
len = impzlength(sos)
len = impzlength(d)

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( ___ ,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

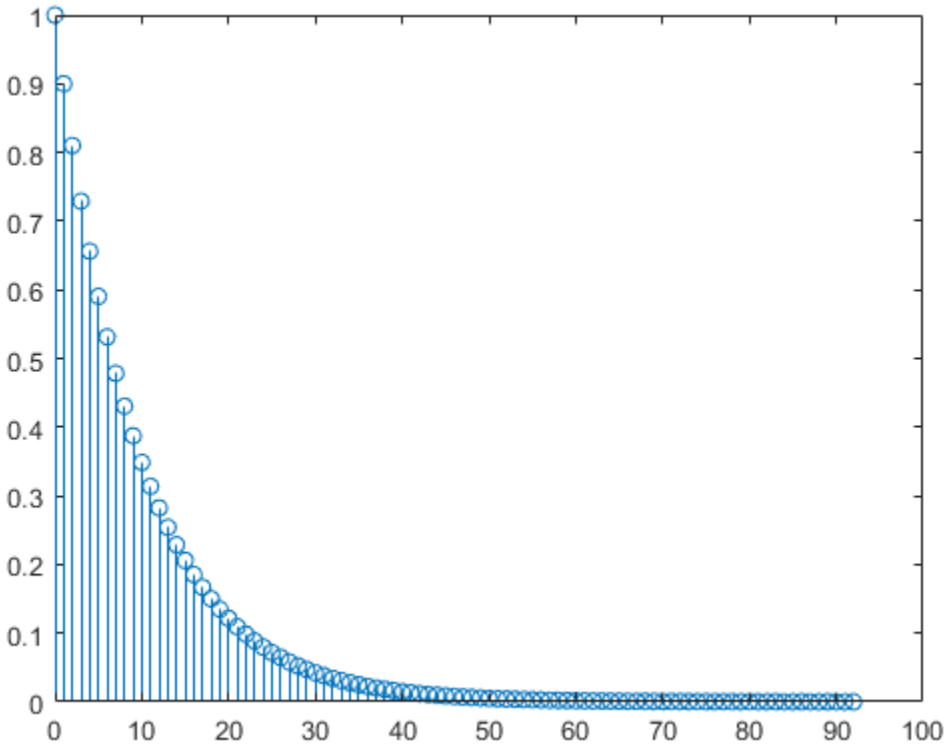
IIR Filter Effective Impulse Response Length — Coefficients

Create a lowpass allpole IIR filter with a pole at 0.9. Calculate the effective impulse response length. Obtain the impulse response. Plot the result.

```
b = 1;
a = [1 -0.9];
len = impzlength(b,a)
```

```
len = 93
```

```
[h,t] = impz(b,a);
stem(t,h)
```




```
h(len)
```

```
ans = 6.1704e-05
```

IIR Filter Effective Impulse Response Length — Second-Order Sections

Design a 4th-order lowpass elliptic filter with a cutoff frequency of 0.4π 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 section matrix.

```
[z,p,k] = ellip(4,1,60,.4);
[sos,g] = zp2sos(z,p,k);
len = impzlength(sos)
```

```
len = 80
```

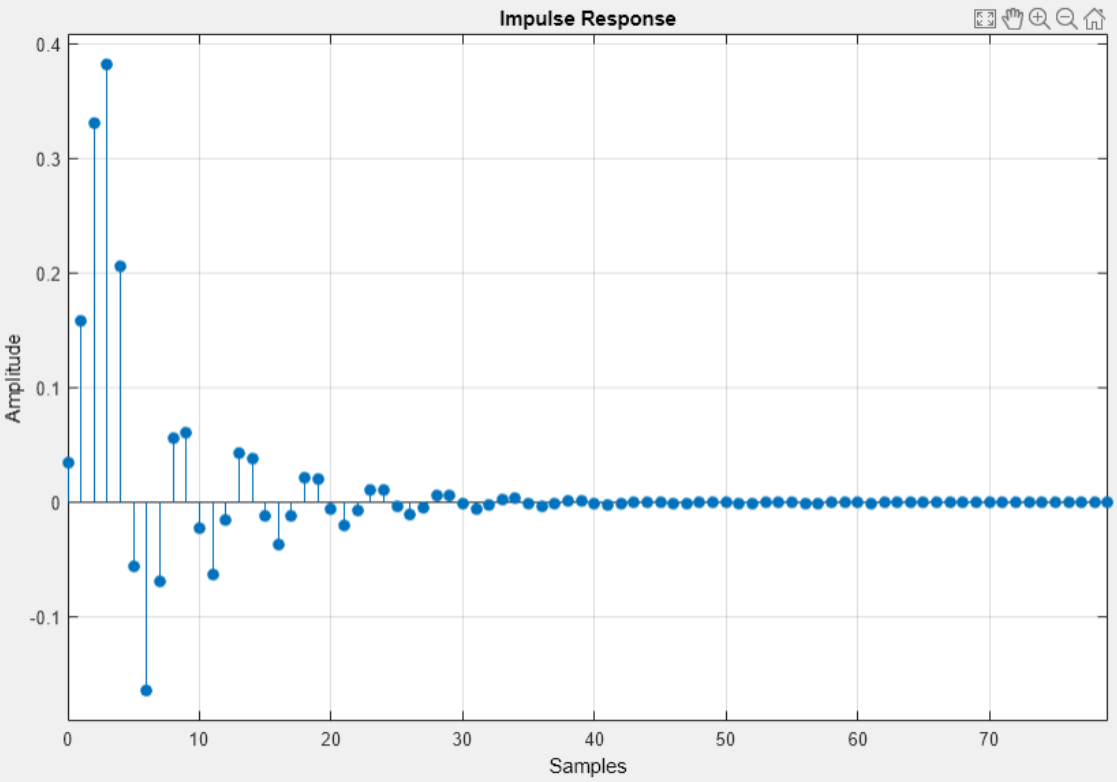
IIR Filter Effective Impulse Response Length — Digital Filter

Use `designfilt` to design a 4th-order lowpass elliptic filter with normalized passband frequency 0.4π 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)
```

```
len = 80
```

```
impz(d)
```



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.

$$\left[B_k(1)\ \ B_k(2)\ \ B_k(3)\ \ A_k(1)\ \ A_k(2)\ \ A_k(3)\right]$$

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 0.5π rad/sample.

`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.

Algorithms

To compute the impulse response for an FIR filter, `impzlength` uses the length of `b`. For IIR filters, the function first finds the poles of the transfer function using `roots`.

If the filter is unstable, the length extends to the point at which the term from the largest pole reaches 10^6 times its original value.

If the filter is stable, the length extends to the point at which the term from the largest-amplitude pole is `tol` times its original amplitude.

If the filter is oscillatory, with poles on the unit circle only, then `impzlength` computes five periods of the slowest oscillation.

If the filter has both oscillatory and damped terms, the length extends to the greater of these values:

- Five periods of the slowest oscillation.
- The point at which the term due to the largest pole is `tol` times its original amplitude.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If the first input to `impzlength` is a variable-size matrix at compile time, then it must not become a vector at runtime.

See Also

`designfilt` | `digitalFilter` | `impz` | `zp2sos`

Introduced in R2013a

info

Information about digital filter

Syntax

```
s = info(d)
```

Description

`s = info(d)` returns a character array with information about the digital filter, `d`.

Examples

Information on a Lowpass FIR Filter

Design a lowpass FIR filter with normalized passband frequency 0.4π rad/sample and normalized stopband frequency 0.45π rad/sample. Obtain information about the filter just designed.

```
d = designfilt('lowpassfir','PassbandFrequency',0.4,'StopbandFrequency',0.45);
s = info(d)
```

```
s = 17x44 char array
    'FIR Digital Filter (real)                  '
    '--------------------------                 '
    'Filter Length  : 81                        '
    'Stable         : Yes                       '
    'Linear Phase   : Yes (Type 1)              '
    '                                           '
    'Design Method Information                  '
    'Design Algorithm : Equiripple              '
    '                                           '
    'Design Specifications                      '
    'Sample Rate     : N/A (normalized frequency)'
    'Response        : Lowpass                  '
    'Specification   : Fp,Fst,Ap,Ast            '
    'Stopband Atten. : 60 dB                    '
    'Passband Ripple : 1 dB                     '
    'Stopband Edge   : 0.45                     '
    'Passband Edge   : 0.4                      '
```

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 0.5π rad/sample.

Output Arguments

s — Information table

character array

Information table, returned as a character array.

See Also

`designfilt` | `digitalFilter`

Introduced in R2014a

instbw

Estimate instantaneous bandwidth

Syntax

```
ibw = instbw(x,fs)
ibw = instbw(x,t)
ibw = instbw(xt)

ibw = instbw(tfd,fd,td)

ibw = instbw( ___ ,Name,Value)

[ibw,t] = instbw( ___ )

instbw( ___ )
```

Description

`ibw = instbw(x,fs)` estimates the instantaneous bandwidth of a signal, `x`, sampled at a rate `fs`. If `x` is a matrix, then the function estimates the instantaneous bandwidth independently for each column and returns the result in the corresponding column of `ibw`.

`ibw = instbw(x,t)` estimates the instantaneous bandwidth of `x` sampled at the time values stored in `t`.

`ibw = instbw(xt)` estimates the instantaneous bandwidth of a signal stored in the MATLAB timetable `xt`. The function treats all variables in the timetable and all columns inside each variable independently.

`ibw = instbw(tfd,fd,td)` estimates the instantaneous bandwidth of the signal whose time-frequency distribution, `tfd`, is sampled at the bandwidth values stored in `fd` and the time values stored in `td`.

`ibw = instbw( ___ ,Name,Value)` specifies additional options for any of the previous syntaxes using name-value arguments. You can specify the scale factor or the frequency limits used in the computation. For example, `'FrequencyLimits',[10 20]` computes the instantaneous bandwidth of the input in the range from 10 Hz to 20 Hz.

`[ibw,t] = instbw( ___ )` also returns `t`, a vector of sample times corresponding to `ibw`.

`instbw( ___ )` with no output arguments plots the estimated instantaneous bandwidth.

Examples

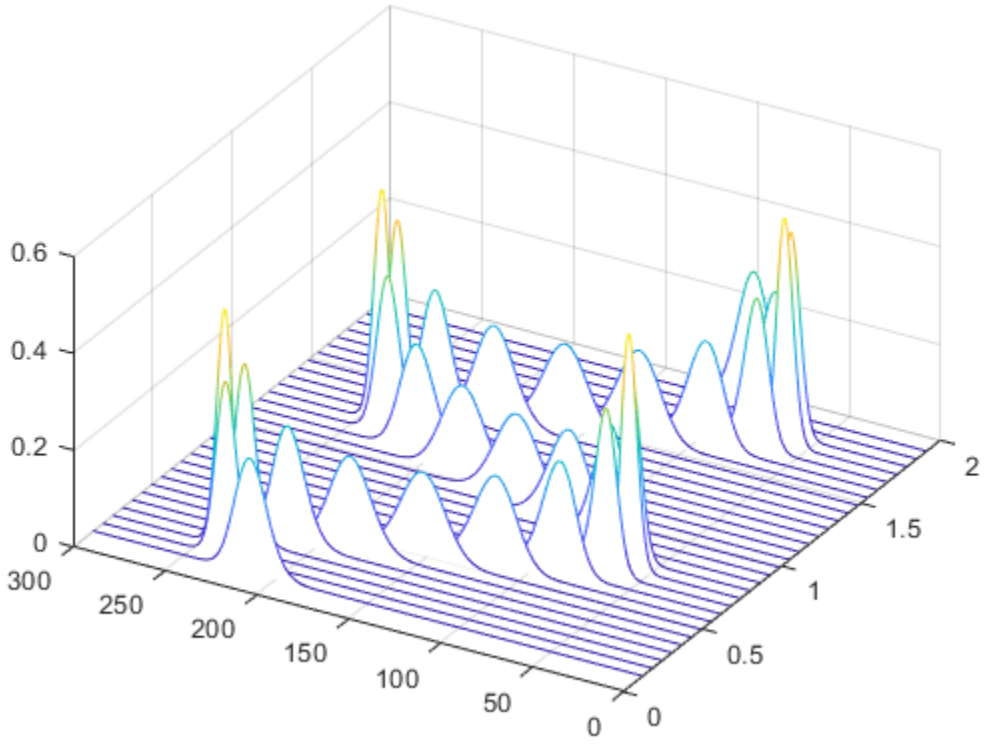
Instantaneous Bandwidth of Sinusoidal Chirp

Generate a signal sampled at 600 Hz for 2 seconds. The signal consists of a chirp with sinusoidally varying frequency content.

```
fs = 6e2;

x = vco(sin(2*pi*(0:1/fs:2)),[0.1 0.4]*fs,fs);
```

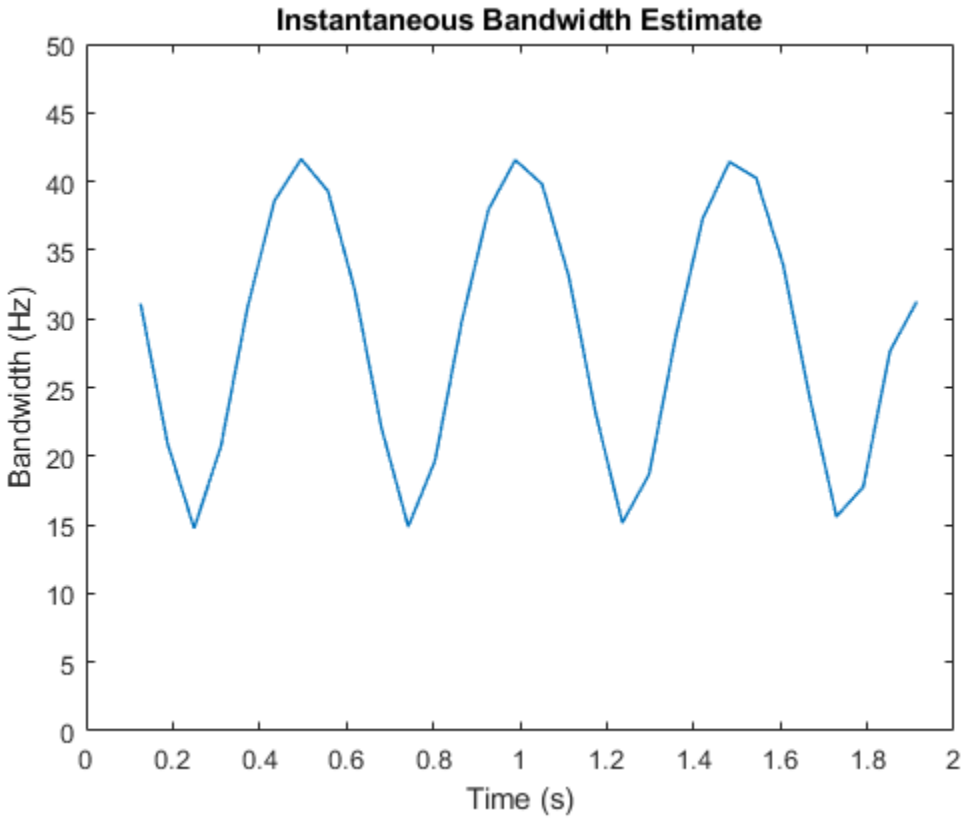
Compute the spectrogram of the signal and display it as a waterfall plot.

```
[p,f,t] = pspectrum(x,fs,'spectrogram');

waterfall(f,t,p')
ax = gca;
ax.XDir = 'reverse';
view(30,45)
```



Estimate and plot the instantaneous bandwidth of the signal.

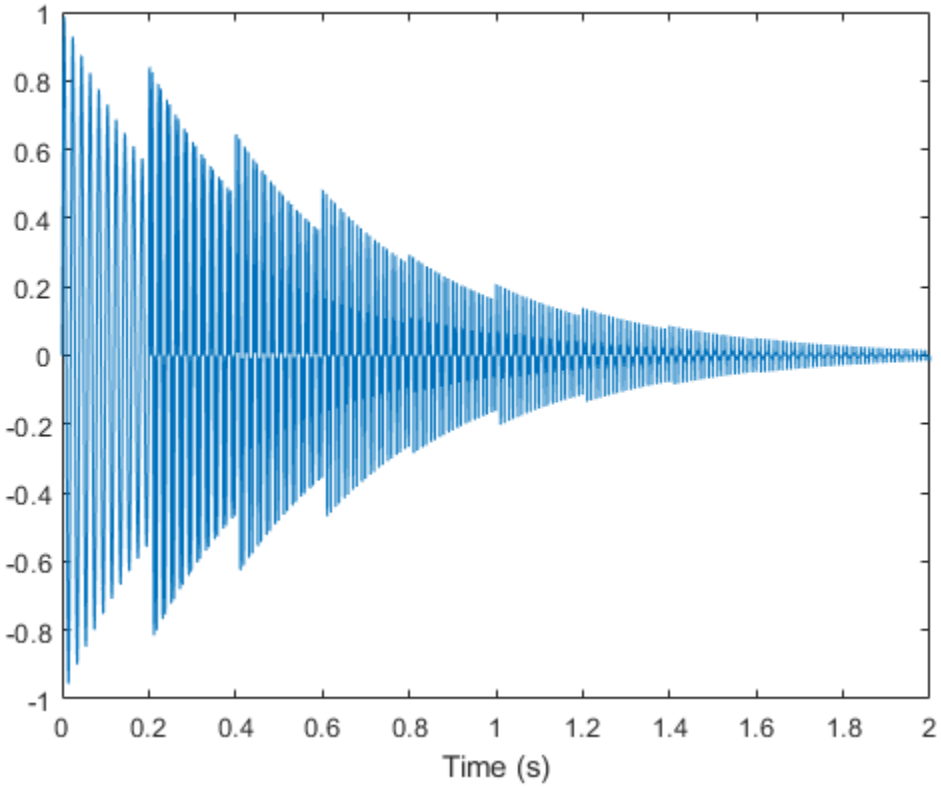
```
instbw(x,fs)
ylim([0 50])
```



Instantaneous Bandwidth from Spectrogram

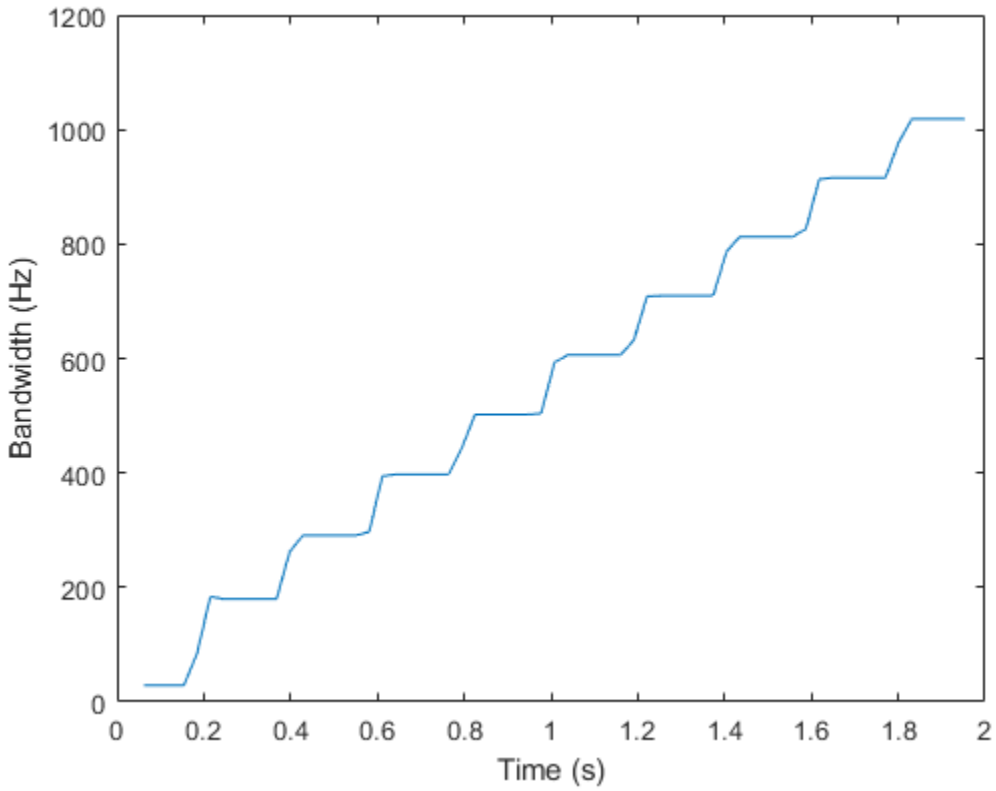
Generate a signal sampled at 2 kHz for 2 seconds. The signal consists of a superposition of exponentially damped sinusoids of increasing frequency that are added at regular intervals. Plot the signal.

```
fs = 2000;
t = 0:1/fs:2-1/fs;

frq = (50:100:950)';

amp = (t > 4*(frq-frq(1))/fs);
x = sum(amp.*sin(2*pi*t.*frq).*exp(-3*t));

% To hear, type sound(x,fs)

plot(t,x)
xlabel('Time (s)')
```



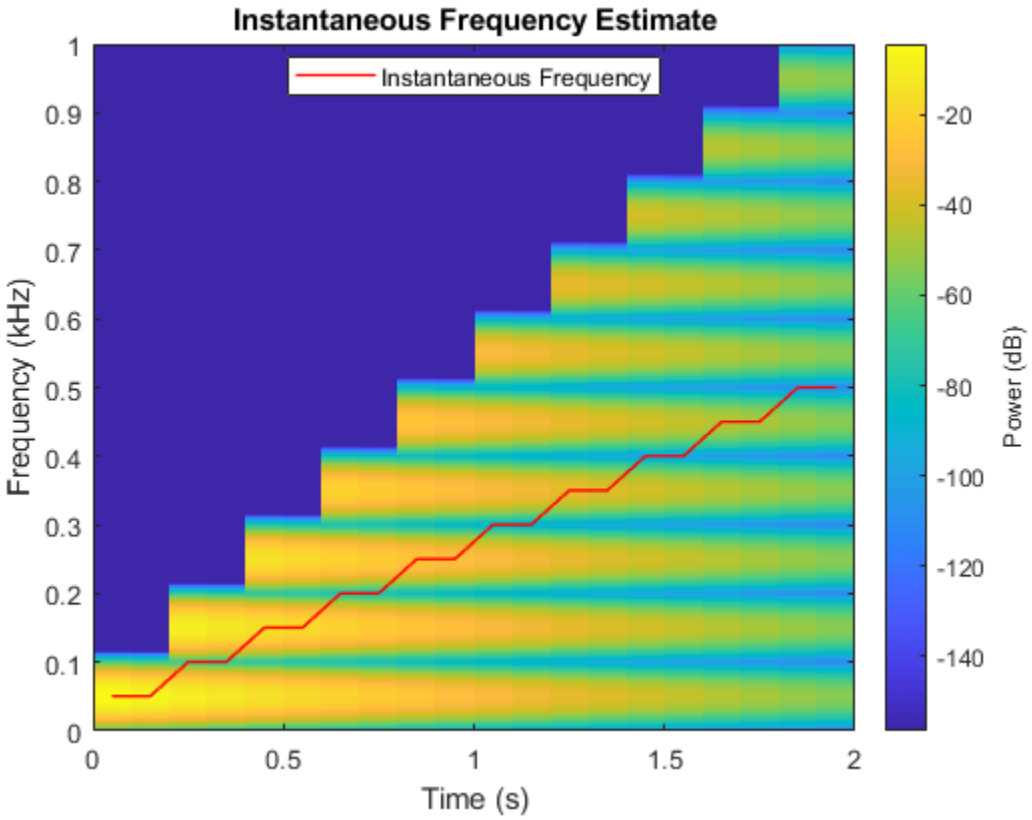
Compute and display the instantaneous bandwidth of the signal.

```
[bw,bt] = instbw(x,t);

plot(bt,bw)
xlabel('Time (s)')
ylabel('Bandwidth (Hz)')
```



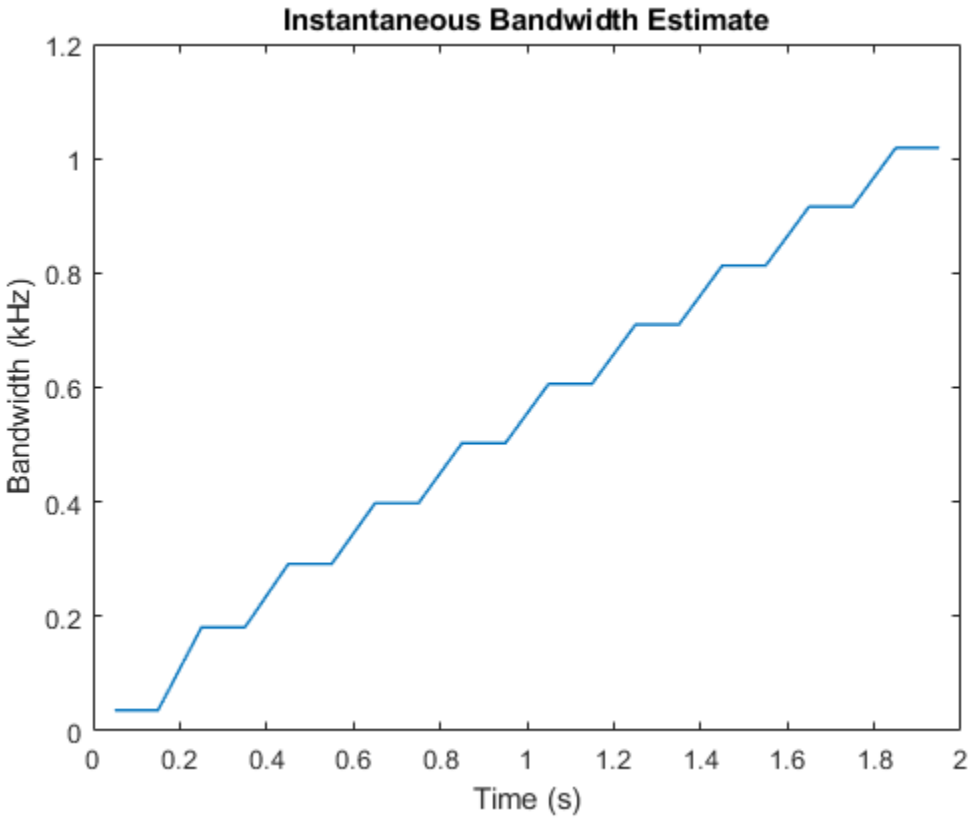
Compute the spectrogram of the signal. Specify a time resolution of 100 milliseconds and 0 overlap between adjoining segments. Use the spectrogram to estimate the instantaneous frequency of the signal.

```
[p,ff,tt] = pspectrum(x,t,'spectrogram','TimeResolution',0.1,'OverlapPercent',0);

instfreq(p,ff,tt)
```



Use the spectrogram to compute the instantaneous bandwidth.

```
instbw(p,ff,tt)
```



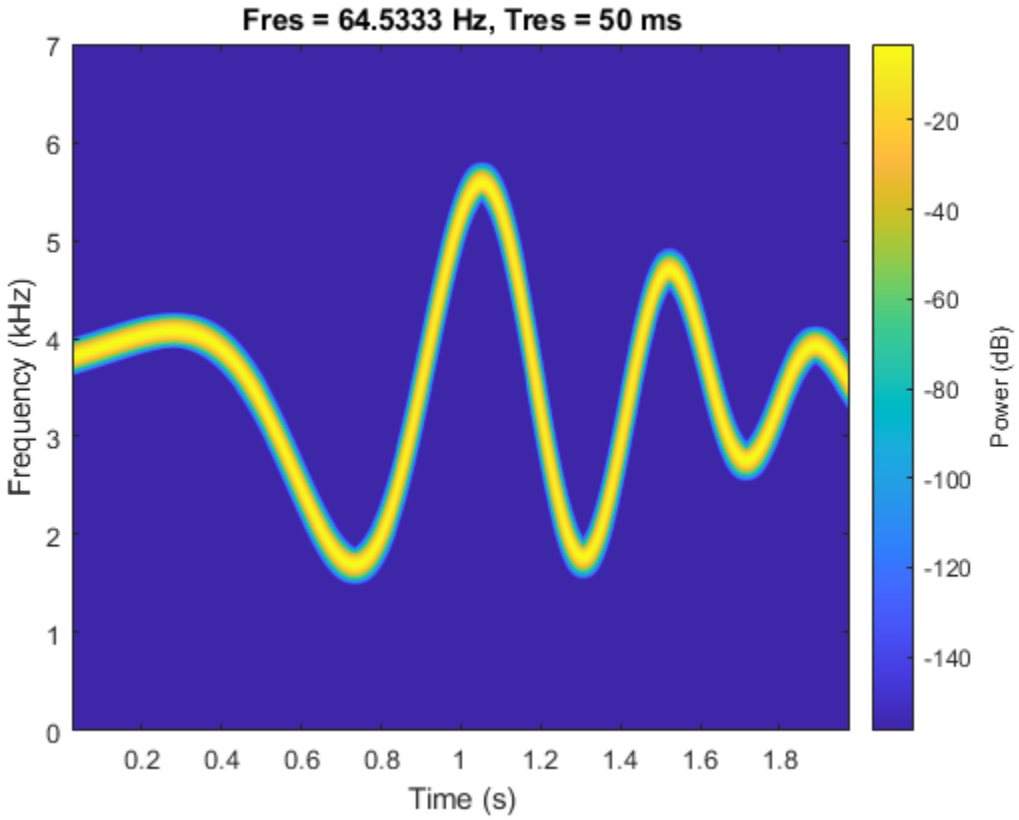
Instantaneous Bandwidth of Timetable

Generate a signal sampled at 14 kHz for 2 seconds. The frequency of the signal varies as a chirp modulated by a Gaussian. Save the signal as a MATLAB® timetable.

```
fs = 14000;
t = (0:1/fs:2)';
s = vco(chirp(t+.1,0,t(end),3).*exp(-2*(t-1).^2),[0.1 0.4]*fs,fs);

sx = timetable(s,'SampleRate',fs);
```

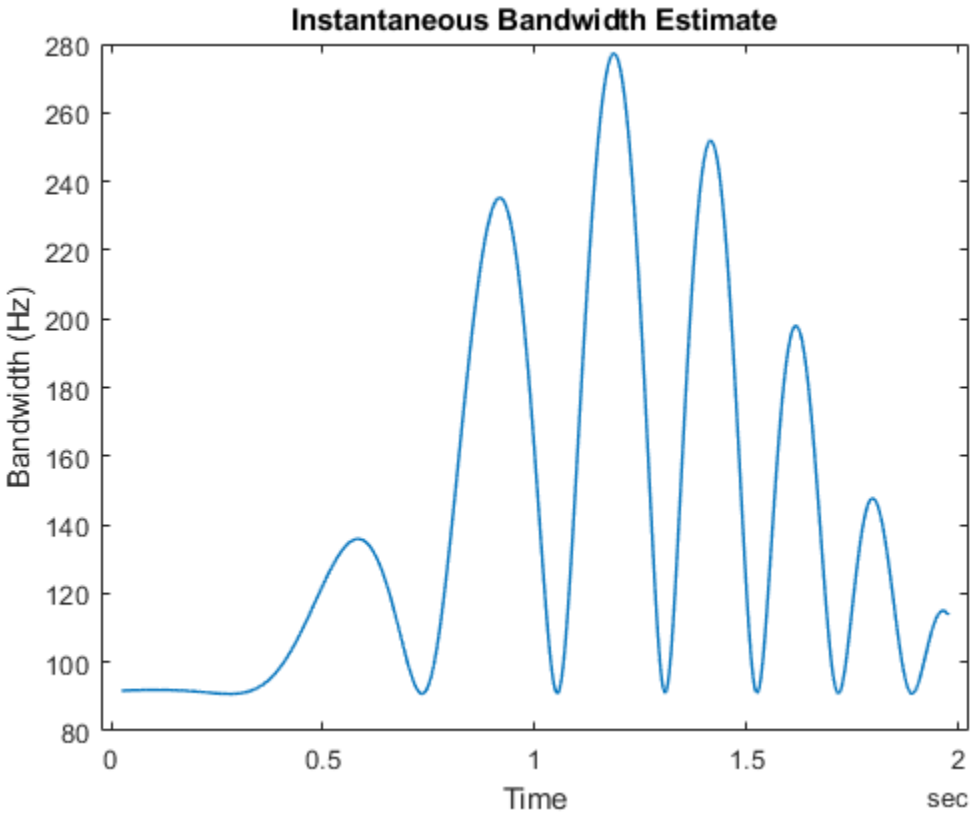
Compute the spectrogram of the signal. Specify a leakage of 0.2, a time resolution of 50 milliseconds, and 99% of overlap between adjoining segments. Display the spectrogram.

```
opts = {'spectrogram','Leakage',0.2,'TimeResolution',0.05,'OverlapPercent',99};

[p,ff,tt] = pspectrum(sx,opts{:});

pspectrum(sx,opts{:})
```



Estimate and display the instantaneous bandwidth of the signal.

```
instbw(p,ff,tt)
```



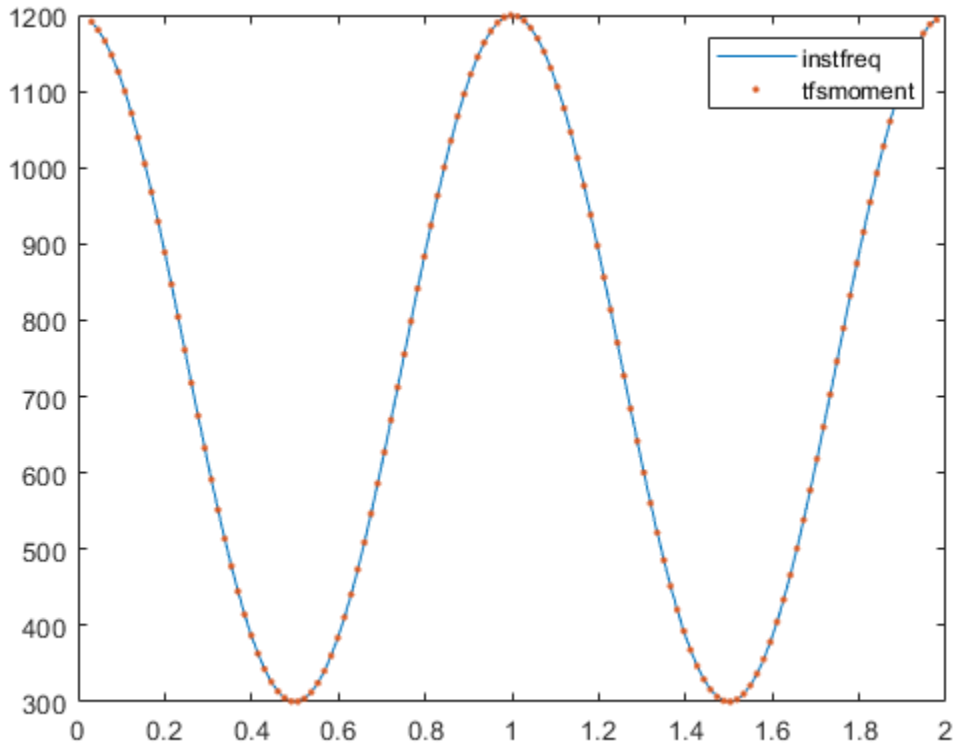
Instantaneous Frequency and Bandwidth as Conditional Spectral Moments

Generate a signal that consists of a chirp whose frequency varies sinusoidally between 300 Hz and 1200 Hz. The signal is sampled at 3 kHz for 2 seconds.

```
fs = 3e3;
t = 0:1/fs:2;
y = chirp(t,100,1,200,"quadratic");
y = vco(cos(2*pi*t),[0.1 0.4]*fs,fs);
```

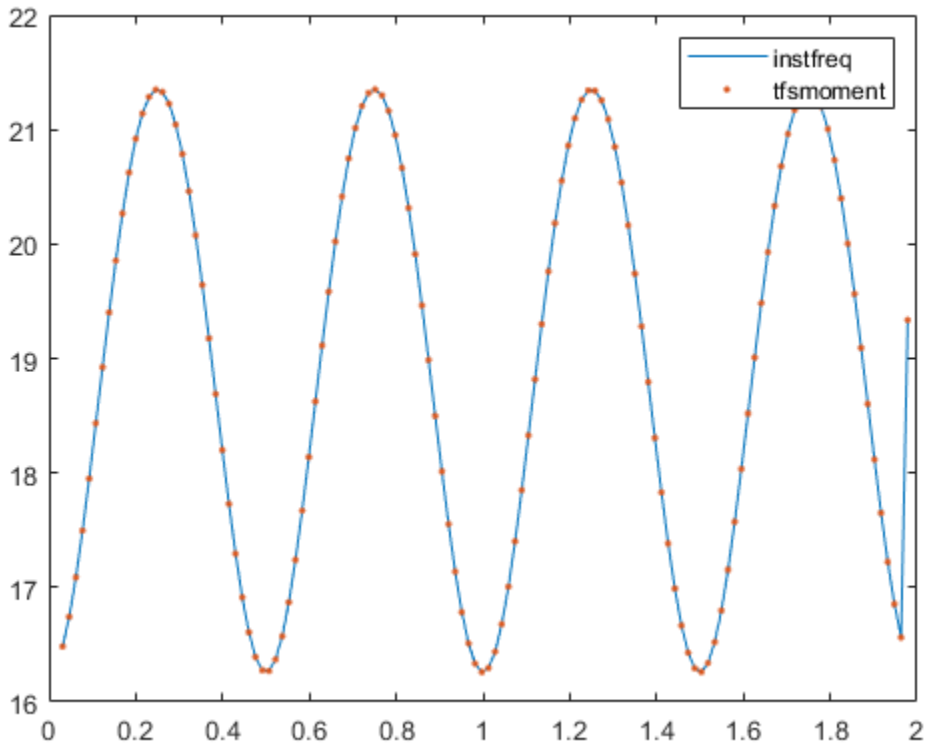
Use `instfreq` to compute the instantaneous frequency of the signal and the corresponding sample times. Verify that the output corresponds to the centralized first-order conditional spectral moment of the time-frequency distribution of the signal as computed by `tfsmoment` (Predictive Maintenance Toolbox).

```
[z,tz] = instfreq(y,fs);
[a,ta] = tfsmoment(y,fs,1,Centralize=false);

plot(tz,z,ta,a,'.')
legend("instfreq","tfsmoment")
```



Use `instbw` to compute the instantaneous bandwidth of the signal and the corresponding sample times. Specify a scale factor of 1. Verify that the output corresponds to the square root of the noncentralized second-order conditional spectral moment of the time-distribution of the signal. In other words, `instbw` generates a standard deviation and `tfsmoment` generates a variance.

```
[w,tw] = instbw(y,fs,ScaleFactor=1);
[m,tm] = tfsmoment(y,fs,2);

plot(tw,w,tm,sqrt(m),'.')
legend("instfreq","tfsmoment")
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If `x` is a vector, then `instbw` treats it as a single channel. If `x` is a matrix, then `instbw` computes the instantaneous bandwidth independently for each column and returns the result in the corresponding column of `ibw`.

Example: `sin(2*pi*(0:127)/16)+randn(1,128)/100` specifies a noisy sinusoid.

Example: `[2 1].*sin(2*pi*(0:127)'./[16 64])` specifies a two-channel sinusoid.

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`

t — Sample times

real vector | `duration` scalar | `duration` array | `datetime` array

Sample times, specified as a real vector, a `duration` scalar, a `duration` array, or a `datetime` array.

- `duration` scalar — The time interval between consecutive samples of `x`.
- Real vector, `duration` array, or `datetime` array — The time instant corresponding to each element of `x`.

Example: `seconds(1)` specifies a 1-second lapse between consecutive measurements of a signal.

Example: `seconds(0:8)` specifies that a signal is sampled at 1 Hz for 8 seconds.

Data Types: `single` | `double` | `duration` | `datetime`

`xt` — Input timetable
timetable

Input timetable. `xt` must contain increasing, finite row times.

If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

Example: `timetable(seconds(0:4)',randn(5,1))` specifies a random process sampled at 1 Hz for 4 seconds.

Example: `timetable(seconds(0:4)',randn(5,3),randn(5,4))` contains a three-channel random process and a four-channel random process, both sampled at 1 Hz for 4 seconds.

Data Types: `single` | `double`

`tfd` — Time-frequency distribution
matrix

Time-frequency distribution, specified as a matrix sampled at the frequencies stored in `fd` and the time values stored in `td`. This input argument is supported only when `'Method'` is set to `'tfmoment'`.

Example: `[p,f,t] = pspectrum(sin(2*pi*(0:511)/4),4,'spectrogram')` specifies the time-frequency distribution of a 1 Hz sinusoid sampled at 4 Hz for 128 seconds, and also the frequencies and times at which it is computed.

Data Types: `single` | `double`

`fd`, `td` — Frequency and time values for time-frequency distribution
vectors

Frequency and time values for time-frequency distribution, specified as vectors. These input arguments are supported only when `'Method'` is set to `'tfmoment'`.

Example: `[p,f,t] = pspectrum(sin(2*pi*(0:511)/4),4,'spectrogram')` specifies the time-frequency distribution of a 1 Hz sinusoid sampled at 4 Hz for 128 seconds, and also the frequencies and times at which it is computed.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'FrequencyLimits',[25 50]` computes the instantaneous bandwidth of the input in the range from 25 Hz to 50 Hz.

`FrequencyLimits` — Frequency range
`[0 fs/2]` (default for real-valued signals) | `[-fs/2 fs/2]` (default for complex-valued signals) | two-element vector in Hz

Frequency range, specified as a two-element vector in Hz. If not specified, `'FrequencyLimits'` defaults to `[0 fs/2]` for real-valued signals and to `[-fs/2 fs/2]` for complex-valued signals.

Data Types: `single` | `double`

`ScaleFactor` — Scaling factor for spectral moment
`sqrt(4*pi)` (default) | real scalar

Scaling factor for spectral moment, specified as a real scalar.

Data Types: `single` | `double`

Output Arguments

`ibw` — Instantaneous bandwidth
vector | matrix | timetable

Instantaneous bandwidth, returned as a vector, a matrix, or a timetable with the same dimensions as the input.

`t` — Times of bandwidth estimates
real vector | `duration` array | `datetime` array

Times of bandwidth estimates, returned as a real vector, a `duration` array, or a `datetime` array.

More About

Instantaneous Bandwidth

The instantaneous bandwidth of a nonstationary signal is a time-varying parameter that relates to the spread of the instantaneous frequency about its average at a given time instant [1], [2].

`instbw` estimates the instantaneous bandwidth as the square-root of the second conditional spectral moment of the time-frequency distribution of the input signal. The function:

1. Computes the spectrogram power spectrum $P(t,f)$ of the input using the `pspectrum` function and uses the spectrum as a time-frequency distribution.
2. Estimates the instantaneous bandwidth using

$$\sigma_f{}^2(t) = \frac{\int_0^\infty (f - f_{\text{inst}}(t))^2 P(t,f)\, df}{\int_0^\infty P(t,f)\, df},$$

where $f_{\text{inst}}(t)$ is the instantaneous frequency returned by `instfreq` and estimated by

$$f_{\text{inst}}(t) = \frac{\int_0^\infty f\, P(t,f)\, df}{\int_0^\infty P(t,f)\, df}.$$

References

[1] Boashash, Boualem. “Estimating and Interpreting the Instantaneous Frequency of a Signal. I. Fundamentals.” *Proceedings of the IEEE* 80, no. 4 (April 1992): 520–538. https://doi.org/10.1109/5.135376.

[2] Boashash, Boualem. "Estimating and Interpreting The Instantaneous Frequency of a Signal. II. Algorithms and Applications." *Proceedings of the IEEE* 80, no. 4 (May 1992): 540–568. https://doi.org/10.1109/5.135378.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Timetables are not supported for code generation.

See Also

`instfreq` | `pspectrum` | `tfmoment` | `tfsmoment` | `tftmoment`

Introduced in R2021a

instfreq

Estimate instantaneous frequency

Syntax

```
ifq = instfreq(x,fs)
ifq = instfreq(x,t)
ifq = instfreq(xt)

ifq = instfreq(tfd,fd,td)

ifq = instfreq( ___ ,Name,Value)

[ifq,t] = instfreq( ___ )

instfreq( ___ )
```

Description

`ifq = instfreq(x,fs)` estimates the instantaneous frequency of a signal, `x`, sampled at a rate `fs`. If `x` is a matrix, then the function estimates the instantaneous frequency independently for each column and returns the result in the corresponding column of `ifq`.

`ifq = instfreq(x,t)` estimates the instantaneous frequency of `x` sampled at the time values stored in `t`.

`ifq = instfreq(xt)` estimates the instantaneous frequency of a signal stored in the MATLAB timetable `xt`. The function treats all variables in the timetable and all columns inside each variable independently.

`ifq = instfreq(tfd,fd,td)` estimates the instantaneous frequency of the signal whose time-frequency distribution, `tfd`, is sampled at the frequency values stored in `fd` and the time values stored in `td`.

`ifq = instfreq( ___ ,Name,Value)` specifies additional options for any of the previous syntaxes using name-value pair arguments. You can specify the algorithm used to estimate the instantaneous frequency or the frequency limits used in the computation.

`[ifq,t] = instfreq( ___ )` also returns `t`, a vector of sample times corresponding to `ifq`.

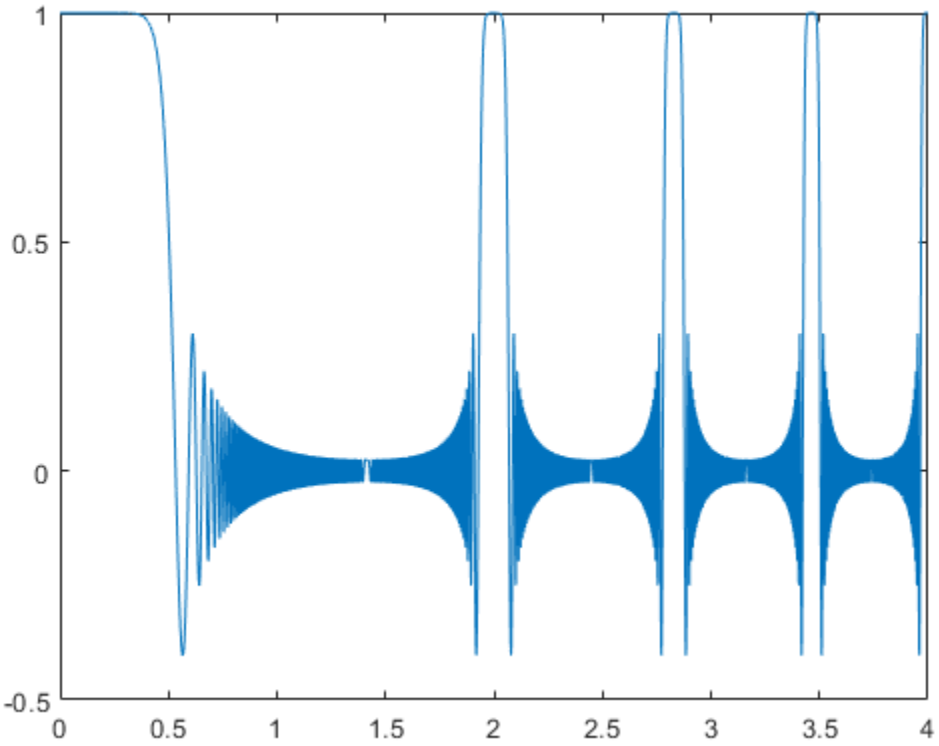
`instfreq( ___ )` with no output arguments plots the estimated instantaneous frequency.

Examples

Instantaneous Frequency of Nonstationary Signal

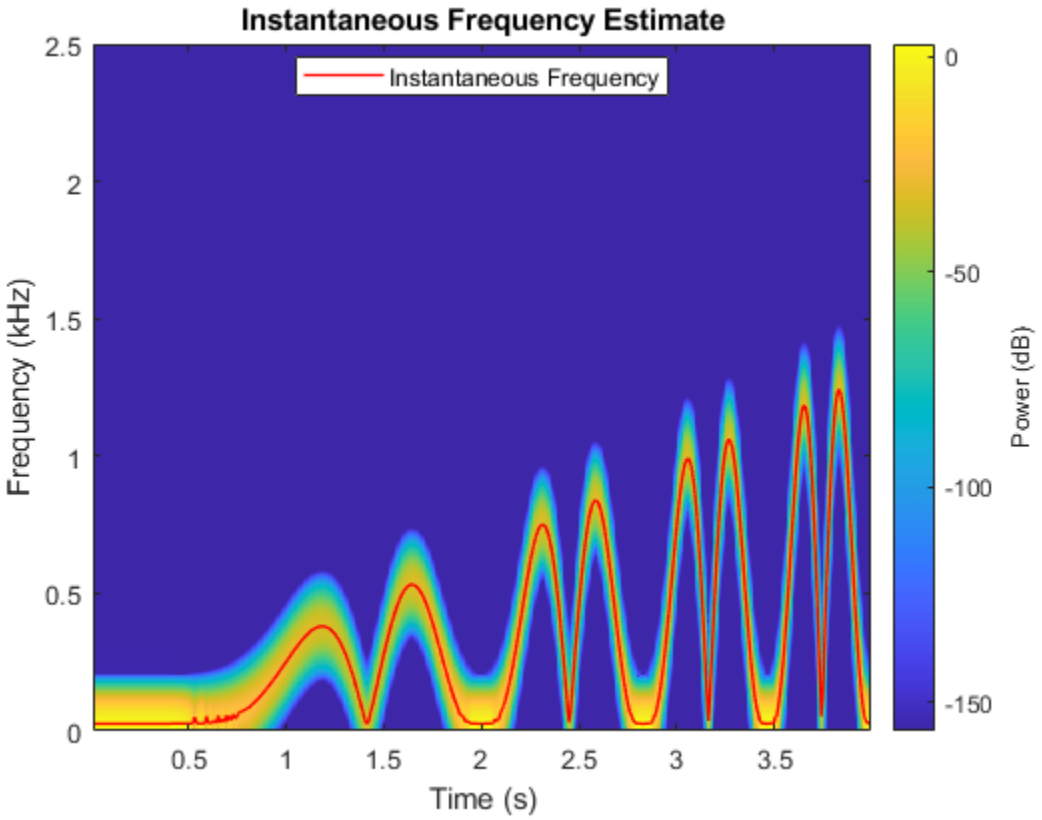
Generate a signal sampled at 5 kHz for 4 seconds. The signal consists of a set of pulses of decreasing duration separated by regions of oscillating amplitude and fluctuating frequency with an increasing trend. Plot the signal.

```
fs = 5000;
t = 0:1/fs:4-1/fs;

s = besselj(0,1000*(sin(2*pi*t.^2/8).^4));

% To hear, type sound(s,fs)

plot(t,s)
```



Estimate the time-dependent frequency of the signal as the first moment of the power spectrogram. Plot the power spectrogram and overlay the instantaneous frequency.

```
instfreq(s,fs)
```



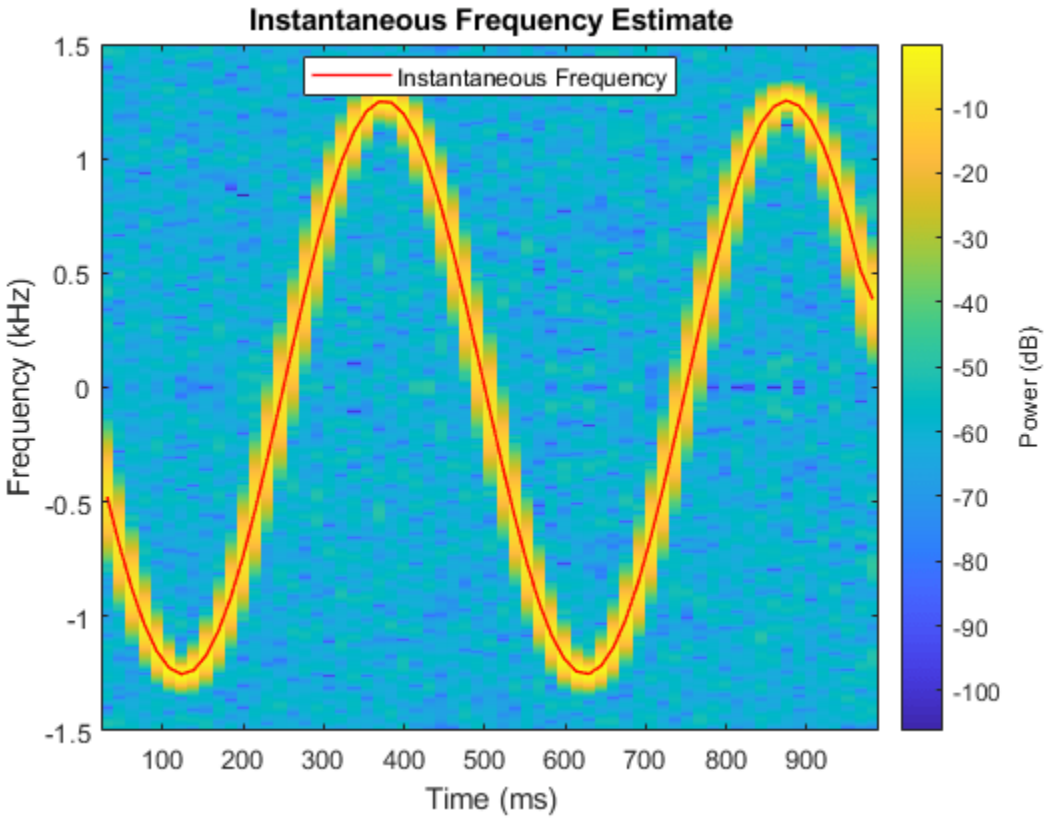
Instantaneous Frequency of Complex-Valued Signal

Generate a complex-valued signal that consists of a chirp with sinusoidally varying frequency content. The signal is sampled at 3 kHz for 1 second and is embedded in white Gaussian noise.

```
fs = 3000;
t = 0:1/fs:1-1/fs;
x = exp(2j*pi*100*cos(2*pi*2*t))+randn(size(t))/100;
```

Estimate the time-dependent frequency of the signal as the first moment of the power spectrogram. This is the only method that `instfreq` supports for complex-valued signals. Plot the power spectrogram and overlay the instantaneous frequency.

```
instfreq(x,t)
```



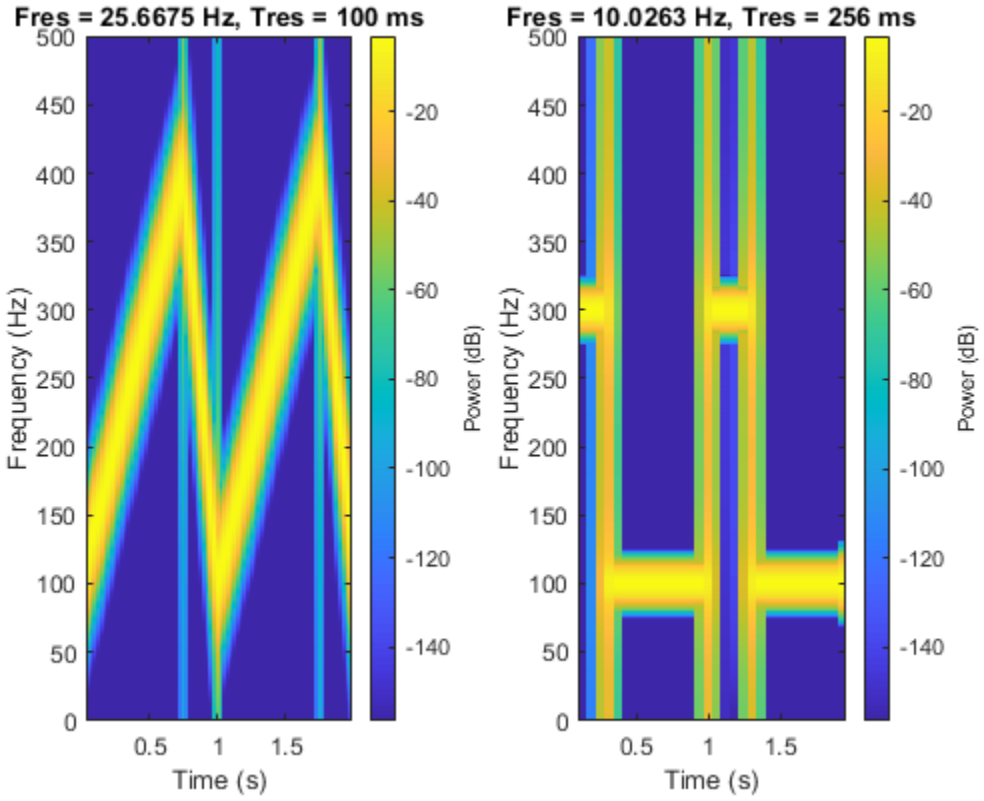
Instantaneous Frequency of Multichannel Signal

Create a two-channel signal, sampled at 1 kHz for 2 seconds, consisting of two voltage-controlled oscillators.

- In one channel, the instantaneous frequency varies with time as a sawtooth wave whose maximum is at 75% of the period.
- In the other channel, the instantaneous frequency varies with time as a square wave with a duty cycle of 30%.

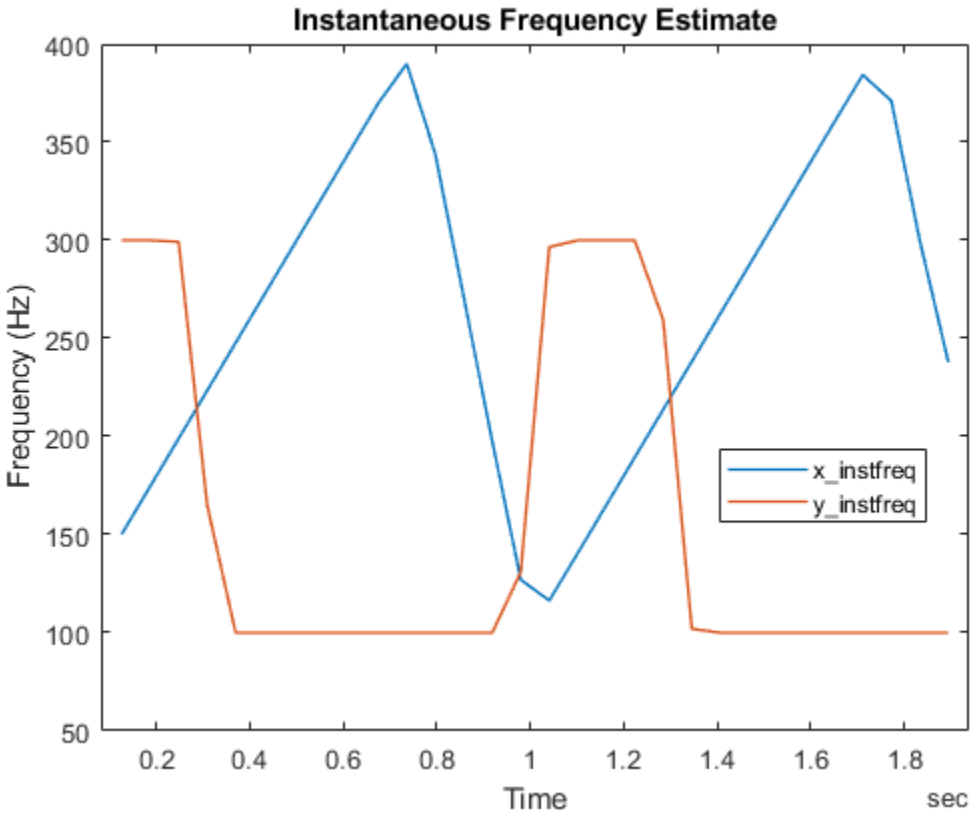
Plot the spectrograms of the two channels. Specify a time resolution of 0.1 second for the sawtooth channel and a frequency resolution of 10 Hz for the square channel.

```
fs = 1000;
t = (0:1/fs:2)';
x = vco(sawtooth(2*pi*t,0.75),[0.1 0.4]*fs,fs);
y = vco(square(2*pi*t,30),[0.1 0.3]*fs,fs);

subplot(1,2,1)
pspectrum(x,fs,'spectrogram','TimeResolution',0.1)
subplot(1,2,2)
pspectrum(y,fs,'spectrogram','FrequencyResolution',10)
```



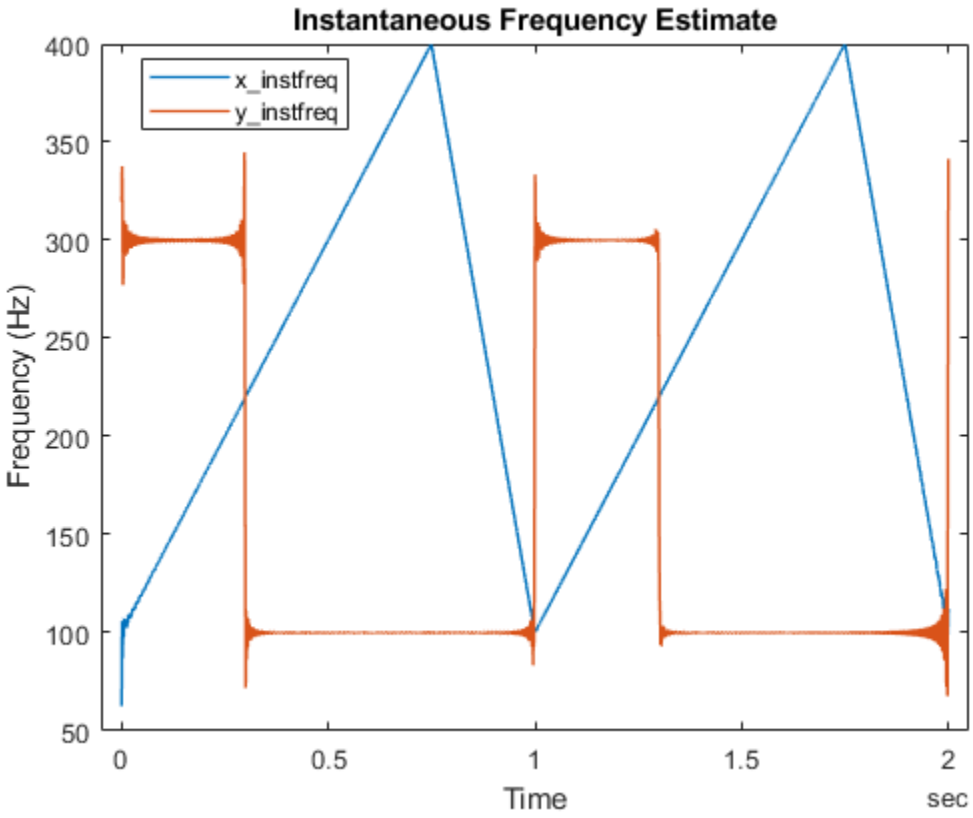
Store the signal in a timetable. Compute and display the instantaneous frequency.

```
xt = timetable(seconds(t),x,y);

clf
instfreq(xt)
```



Repeat the computation using the analytic signal.

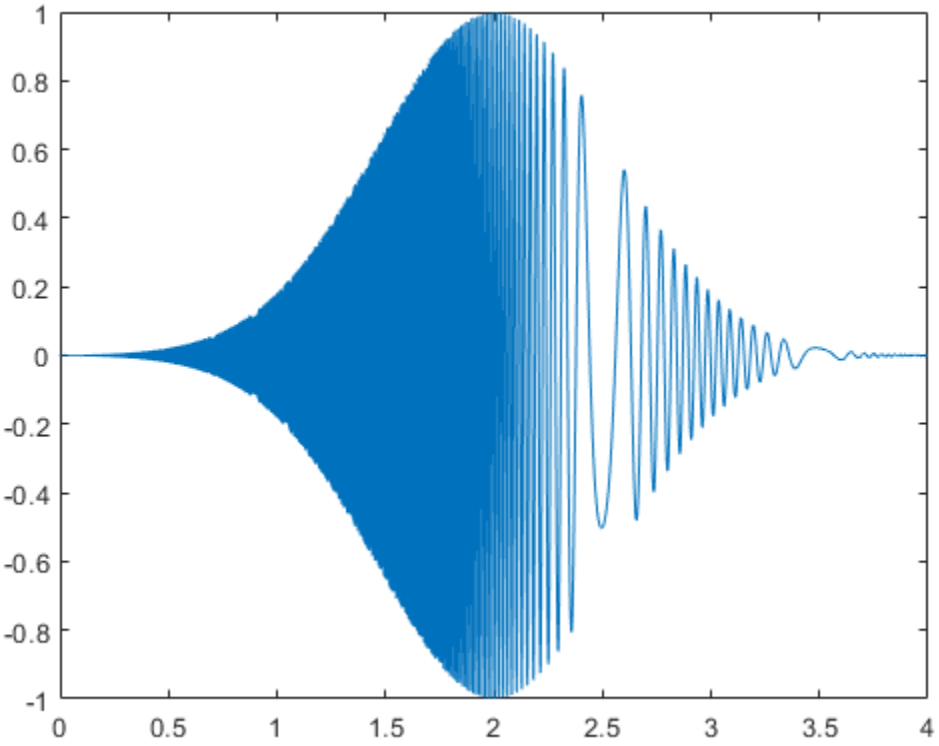
```
instfreq(xt,'Method','hilbert')
```



Instantaneous Frequency of Chirp

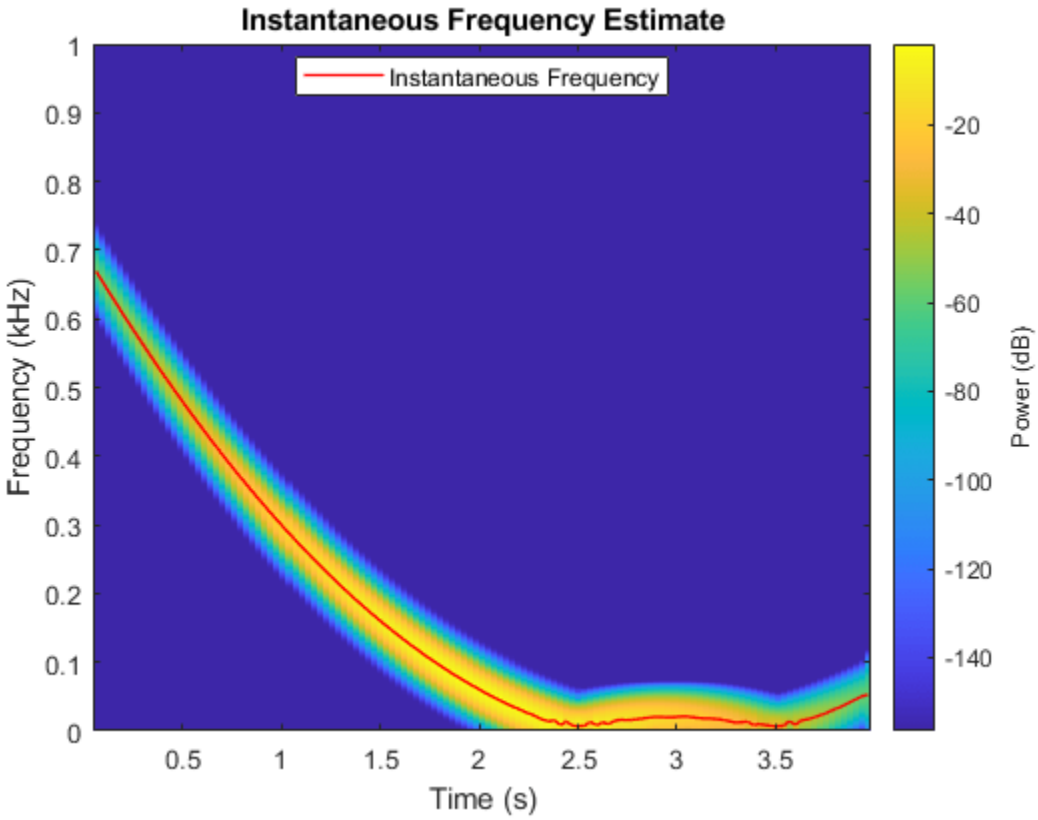
Generate a quadratic chirp modulated by a Gaussian. Specify a sample rate of 2 kHz and a signal duration of 4 seconds.

```
fs = 2000;
t = 0:1/fs:4-1/fs;

q = chirp(t-1,0,1/2,20,'quadratic',100,'convex').*exp(-1.7*(t-2).^2);
plot(t,q)
```



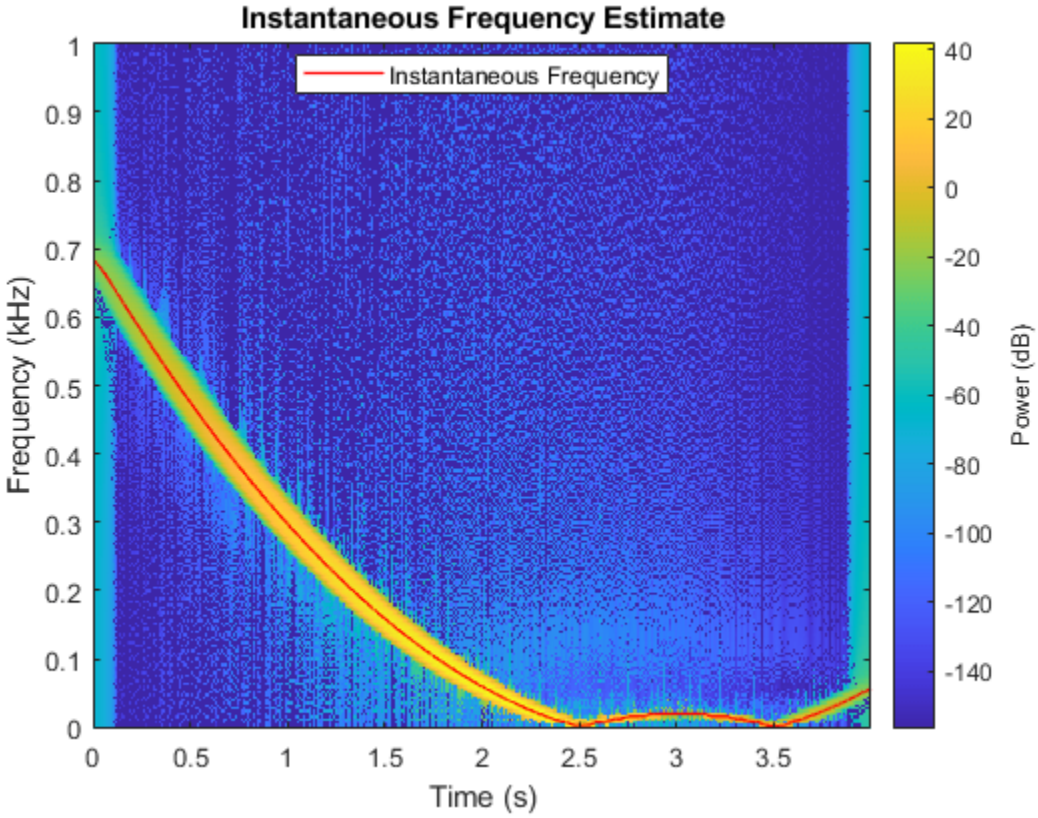
Use the `pspectrum` function with default settings to estimate the power spectrum of the signal. Use the estimate to compute the instantaneous frequency.

```
[p,f,t] = pspectrum(q,fs,'spectrogram');

instfreq(p,f,t)
```



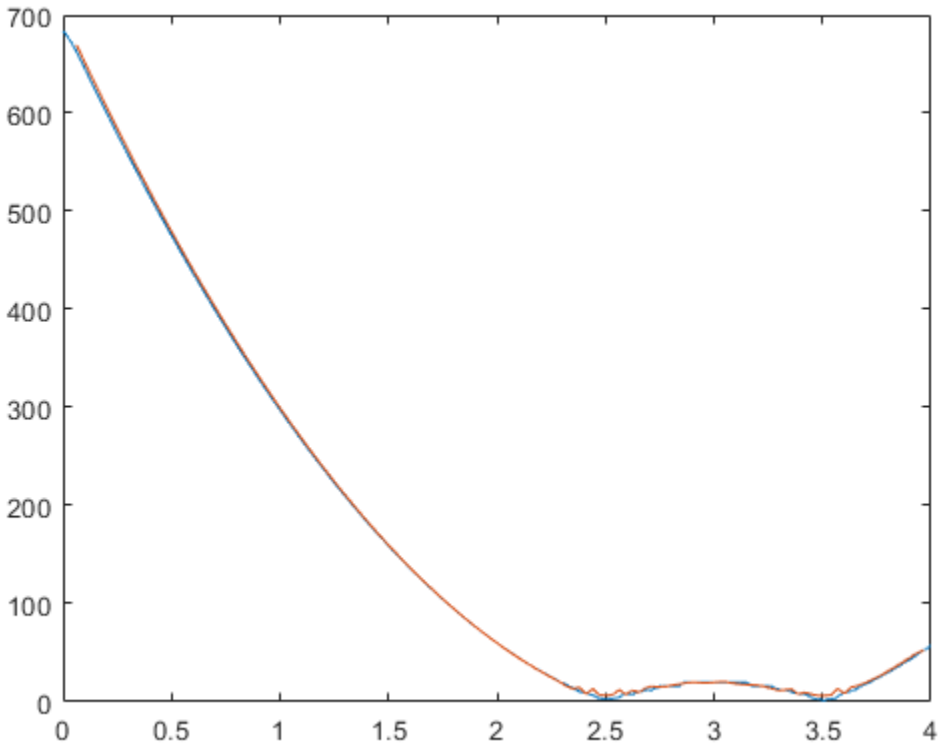
Repeat the calculation using the synchrosqueezed Fourier transform. Use a 500-sample Hann window to divide the signal into segments and window them.

```
[s,sf,st] = fsst(q,fs,hann(500));

instfreq(abs(s).^2,sf,st)
```



Compare the instantaneous frequencies found using the two different methods.

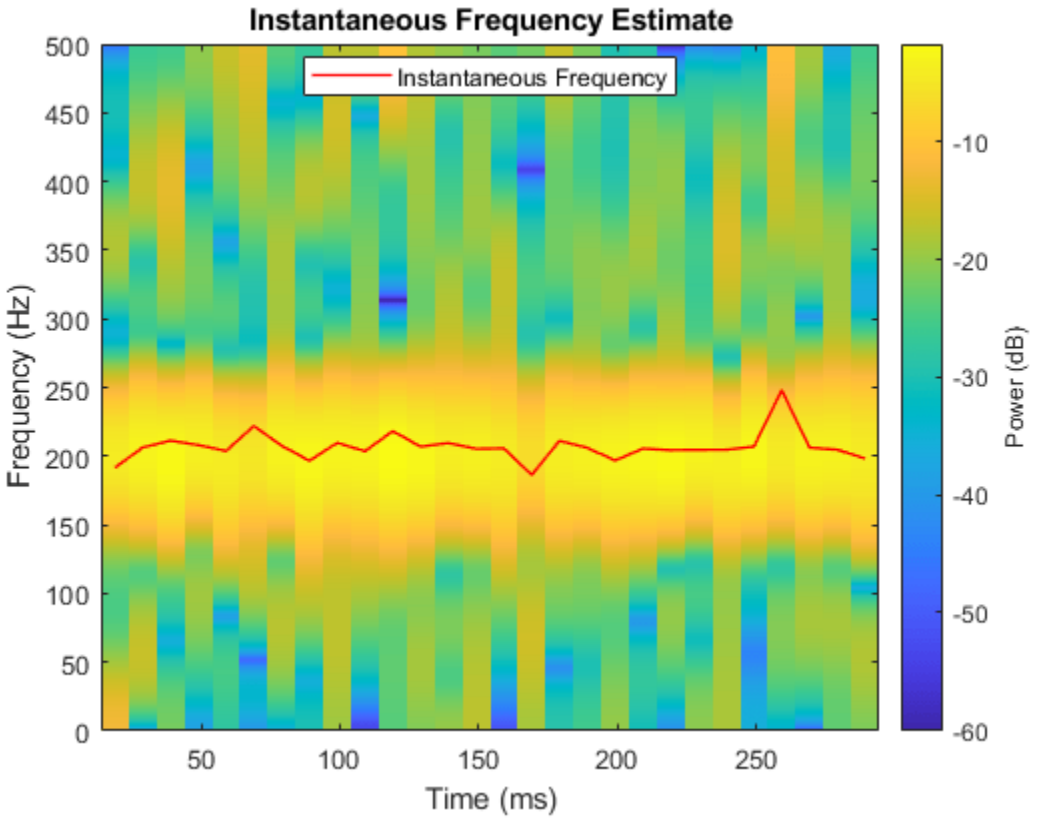
```
[psf,pst] = instfreq(p,f,t);
[fsf,fst] = instfreq(abs(s).^2,sf,st);

plot(fst,fsf,pst,psf)
```



Instantaneous Frequency of Sinusoid

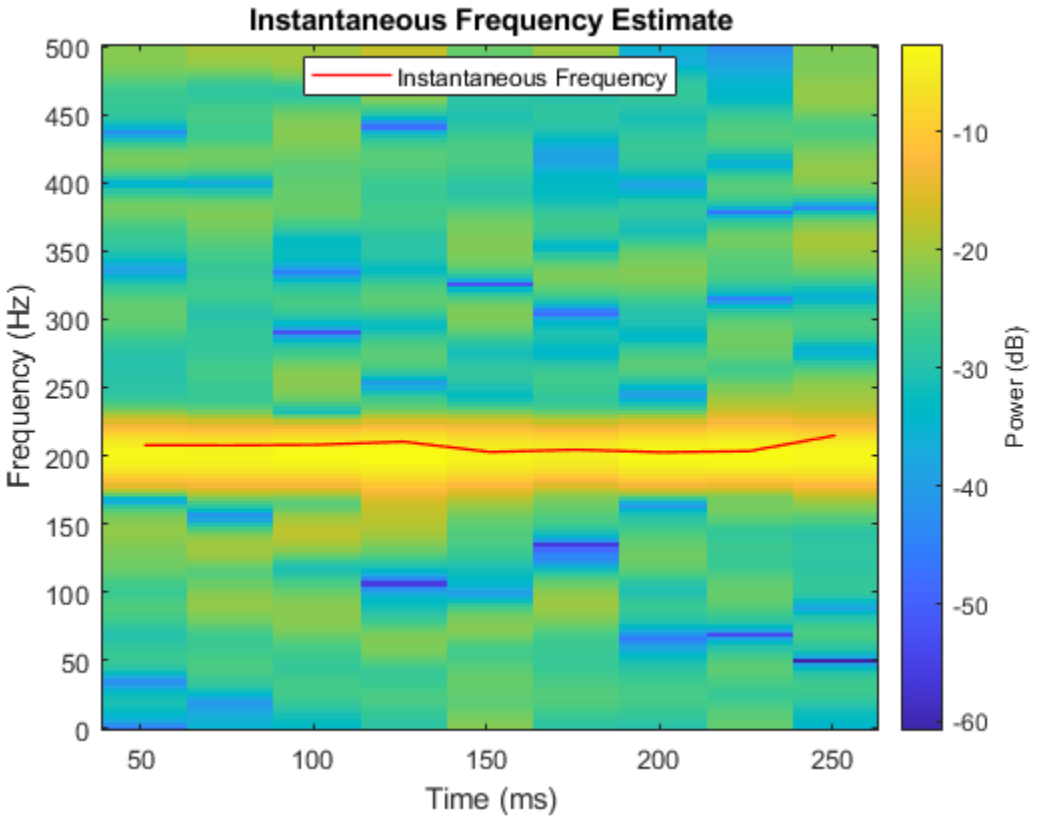
Generate a sinusoidal signal sampled at 1 kHz for 0.3 second and embedded in white Gaussian noise of variance 1/16. Specify a sinusoid frequency of 200 Hz. Estimate and display the instantaneous frequency of the signal.

```
fs = 1000;
t = (0:1/fs:0.3-1/fs)';

x = sin(2*pi*200*t) + randn(size(t))/4;

instfreq(x,t)
```



Estimate the instantaneous frequency of the signal again, but now use a time-frequency distribution with a coarse frequency resolution of 25 Hz as input.

```
[p,fd,td] = pspectrum(x,t,'spectrogram','FrequencyResolution',25);

instfreq(p,fd,td)
```



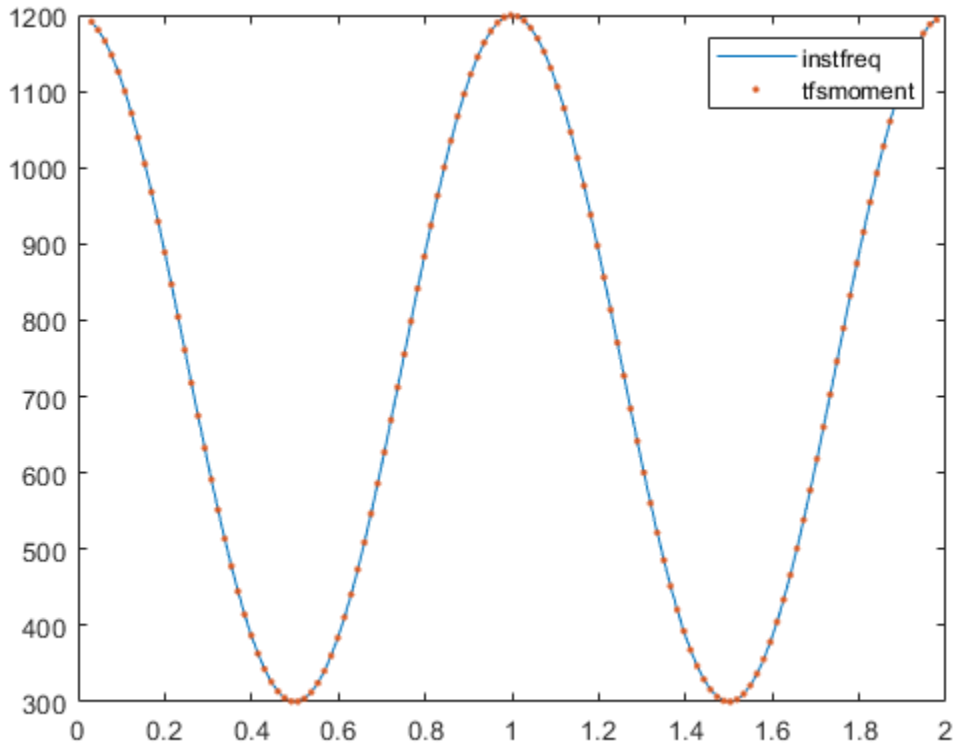
Instantaneous Frequency and Bandwidth as Conditional Spectral Moments

Generate a signal that consists of a chirp whose frequency varies sinusoidally between 300 Hz and 1200 Hz. The signal is sampled at 3 kHz for 2 seconds.

```
fs = 3e3;
t = 0:1/fs:2;
y = chirp(t,100,1,200,"quadratic");
y = vco(cos(2*pi*t),[0.1 0.4]*fs,fs);
```

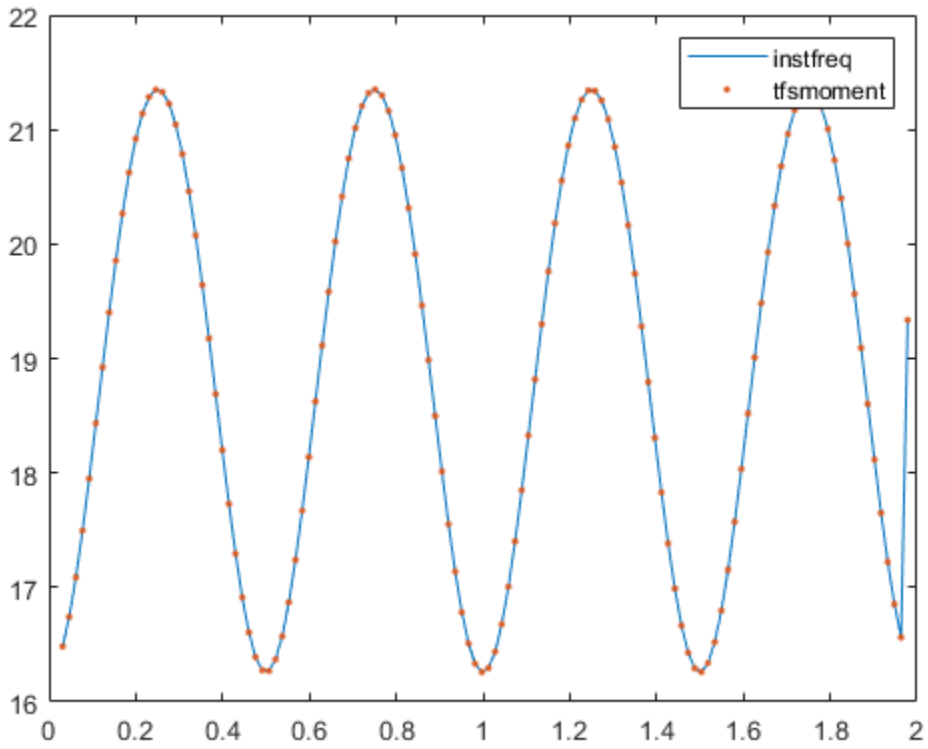
Use `instfreq` to compute the instantaneous frequency of the signal and the corresponding sample times. Verify that the output corresponds to the centralized first-order conditional spectral moment of the time-frequency distribution of the signal as computed by `tfsmoment` (Predictive Maintenance Toolbox).

```
[z,tz] = instfreq(y,fs);
[a,ta] = tfsmoment(y,fs,1,Centralize=false);

plot(tz,z,ta,a,'.')
legend("instfreq","tfsmoment")
```



Use `instbw` to compute the instantaneous bandwidth of the signal and the corresponding sample times. Specify a scale factor of 1. Verify that the output corresponds to the square root of the noncentralized second-order conditional spectral moment of the time-distribution of the signal. In other words, `instbw` generates a standard deviation and `tfsmoment` generates a variance.

```
[w,tw] = instbw(y,fs,ScaleFactor=1);
[m,tm] = tfsmoment(y,fs,2);

plot(tw,w,tm,sqrt(m),'.')
legend("instfreq","tfsmoment")
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If `x` is a vector, then `instfreq` treats it as a single channel. If `x` is a matrix, then `instfreq` computes the instantaneous frequency independently for each column and returns the result in the corresponding column of `ifq`.

Example: `sin(2*pi*(0:127)/16)+randn(1,128)/100` specifies a noisy sinusoid

Example: `[2 1].*sin(2*pi*(0:127)'./[16 64])` specifies a two-channel sinusoid.

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`

t — Sample times

real vector | `duration` scalar | `duration` array | `datetime` array

Sample times, specified as a real vector, a `duration` scalar, a `duration` array, or a `datetime` array.

- `duration` scalar — The time interval between consecutive samples of `x`.
- Real vector, `duration` array, or `datetime` array — The time instant corresponding to each element of `x`.

Example: `seconds(1)` specifies a 1-second lapse between consecutive measurements of a signal.

Example: `seconds(0:8)` specifies that a signal is sampled at 1 Hz for 8 seconds.

Data Types: `single` | `double` | `duration` | `datetime`

`xt` — Input timetable
timetable

Input timetable. `xt` must contain increasing, finite row times.

If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

Example: `timetable(seconds(0:4)',randn(5,1))` specifies a random process sampled at 1 Hz for 4 seconds.

Example: `timetable(seconds(0:4)',randn(5,3),randn(5,4))` contains a three-channel random process and a four-channel random process, both sampled at 1 Hz for 4 seconds.

Data Types: `single` | `double`

`tfd` — Time-frequency distribution
matrix

Time-frequency distribution, specified as a matrix sampled at the frequencies stored in `fd` and the time values stored in `td`. This input argument is supported only when `'Method'` is set to `'tfmoment'`.

Example: `[p,f,t] = pspectrum(sin(2*pi*(0:511)/4),4,'spectrogram')` specifies the time-frequency distribution of a 1 Hz sinusoid sampled at 4 Hz for 128 seconds, and also the frequencies and times at which it is computed.

Data Types: `single` | `double`

`fd`, `td` — Frequency and time values for time-frequency distribution
vectors

Frequency and time values for time-frequency distribution, specified as vectors. These input arguments are supported only when `'Method'` is set to `'tfmoment'`.

Example: `[p,f,t] = pspectrum(sin(2*pi*(0:511)/4),4,'spectrogram')` specifies the time-frequency distribution of a 1 Hz sinusoid sampled at 4 Hz for 128 seconds, and also the frequencies and times at which it is computed.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Method','tfmoment','FrequencyLimits',[25 50]` computes the instantaneous frequency of the input in the range from 25 Hz to 50 Hz by finding the first conditional spectral moment of the time-frequency distribution.

`FrequencyLimits` — Frequency range
`[0 fs/2]` (default for real-valued signals) | `[-fs/2 fs/2]` (default for complex-valued signals) | two-element vector in Hz

Frequency range, specified as the comma-separated pair consisting of `'FrequencyLimits'` and a two-element vector in Hz. If not specified, `'FrequencyLimits'` defaults to `[0 fs/2]` for real-valued signals and to `[-fs/2 fs/2]` for complex-valued signals. This argument is supported only when `'Method'` is set to `'tfmoment'`.

Data Types: `single` | `double`

`Method` — Computation method
`'tfmoment'` (default) | `'hilbert'`

Computation method, specified as the comma-separated pair consisting of `'Method'` and either `'tfmoment'` or `'hilbert'`.

- `'tfmoment'` — Compute the instantaneous frequency as the first conditional spectral moment of the time-frequency distribution of `x`. If `x` is nonuniformly sampled, then `instfreq` interpolates the signal to a uniform grid to compute instantaneous frequencies.
- `'hilbert'` — Compute the instantaneous frequency as the derivative of the phase of the analytic signal of `x` found using the Hilbert transform. This method accepts only uniformly sampled, real-valued signals and does not support time-frequency distribution input.

Output Arguments

`ifq` — Instantaneous frequency
vector | matrix | timetable

Instantaneous frequency, returned as a vector, a matrix, or a timetable with the same dimensions as the input.

`t` — Times of frequency estimates
real vector | `duration` array | `datetime` array

Times of frequency estimates, returned as a real vector, a `duration` array, or a `datetime` array.

More About

Instantaneous Frequency

The instantaneous frequency of a nonstationary signal is a time-varying parameter that relates to the average of the frequencies present in the signal as it evolves [1], [2].

- If `'Method'` is set to `'tfmoment'`, then `instfreq` estimates the instantaneous frequency as the first conditional spectral moment of the time-frequency distribution of the input signal. The function:

  1. Computes the spectrogram power spectrum $P(t,f)$ of the input using the `pspectrum` function and uses the spectrum as a time-frequency distribution.

2. Estimates the instantaneous frequency using

$$f_{\text{inst}}(t) = \frac{\int_0^\infty f\, P(t,f)\, df}{\int_0^\infty P(t,f)\, df}.$$

- If `'Method'` is set to `'hilbert'`, then `instfreq` estimates the instantaneous frequency as the derivative of the phase of the analytic signal of the input. The function:

  1. Computes the analytic signal, x_A, of the input using the `hilbert` function.
  2. Estimates the instantaneous frequency using

$$f_{\text{inst}}(t) = \frac{1}{2\pi}\frac{d\phi}{dt},$$

     where ϕ is the phase of the analytic signal of the input.

References

[1] Boashash, Boualem. “Estimating and Interpreting the Instantaneous Frequency of a Signal. I. Fundamentals.” *Proceedings of the IEEE* 80, no. 4 (April 1992): 520–538. https://doi.org/10.1109/5.135376.

[2] Boashash, Boualem. "Estimating and Interpreting The Instantaneous Frequency of a Signal. II. Algorithms and Applications." *Proceedings of the IEEE* 80, no. 4 (May 1992): 540–568. https://doi.org/10.1109/5.135378.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Arguments specified using name-value pairs must be compile-time constants.
- Timetables are not supported for code generation.

See Also

`hilbert` | `instbw` | `pspectrum` | `tfmoment` | `tfsmoment` | `tftmoment`

Topics

“Hilbert Transform and Instantaneous Frequency”
“Instantaneous Frequency of Complex Chirp”

Introduced in R2018a

interp

Interpolation — increase sample rate by integer factor

Syntax

```
y = interp(x,r)
y = interp(x,r,n,cutoff)
[y,b] = interp(x,r,n,cutoff)
```

Description

`y = interp(x,r)` increases the sample rate of `x`, the input signal, by a factor of `r`.

`y = interp(x,r,n,cutoff)` specifies two additional values:

- `n` is half the number of original sample values used to interpolate the expanded signal.
- `cutoff` is the normalized cutoff frequency of the input signal, specified as a fraction of the Nyquist frequency.

`[y,b] = interp(x,r,n,cutoff)` also returns a vector, `b`, with the filter coefficients used for the interpolation.

Examples

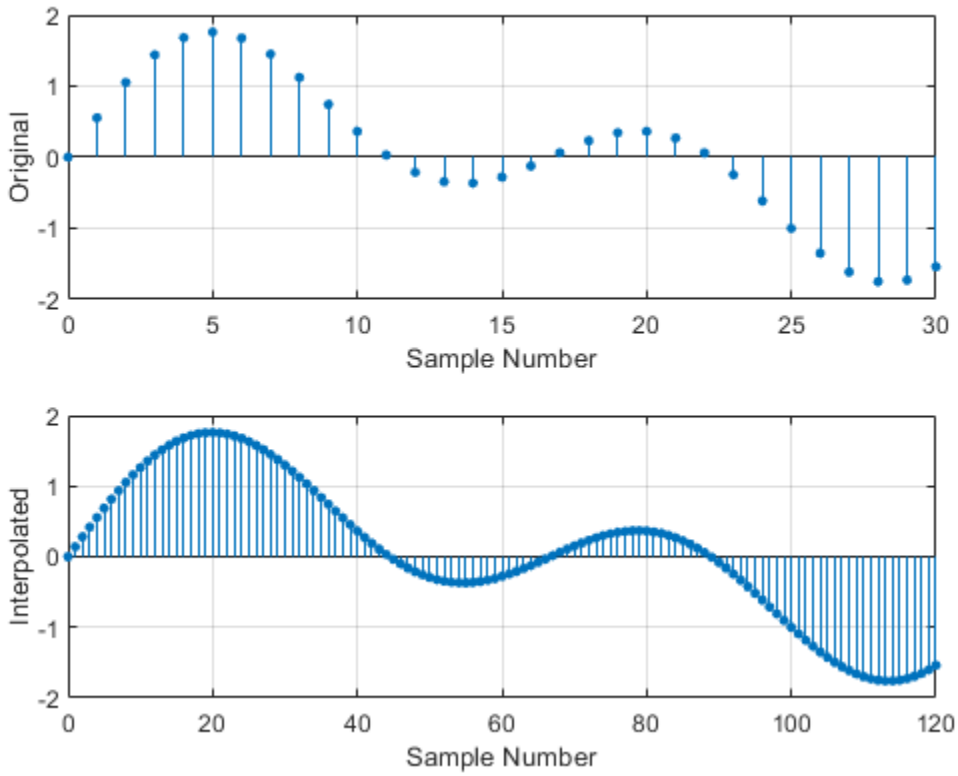
Interpolate Signal

Create a sinusoidal signal sampled at 1 kHz. Interpolate it by a factor of four.

```
t = 0:1/1e3:1;
x = sin(2*pi*30*t) + sin(2*pi*60*t);
y = interp(x,4);
```

Plot the original and interpolated signals.

```
subplot(2,1,1)
stem(0:30,x(1:31),'filled','MarkerSize',3)
grid on
xlabel('Sample Number')
ylabel('Original')

subplot(2,1,2)
stem(0:120,y(1:121),'filled','MarkerSize',3)
grid on
xlabel('Sample Number')
ylabel('Interpolated')
```



Input Arguments

x — Input signal

vector

Input signal, specified as a vector.

Data Types: `double` | `single`

r — Interpolation factor

positive integer

Interpolation factor, specified as a positive integer.

Data Types: `double` | `single`

n — Half the number of input samples used for interpolation

4 (default) | positive integer

Half the number of input samples used for interpolation, specified as a positive integer. For best results, use `n` no larger than 10. The lowpass interpolation filter has length $2 \times n \times r + 1$.

Data Types: `double` | `single`

cutoff — 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 that represents a fraction of the Nyquist frequency. A value of 1 means that the signal occupies the full Nyquist interval.

Data Types: `double` | `single`

Output Arguments

y — Interpolated signal

vector

Interpolated signal, returned as a vector. `y` is `r` times as long as the original input, `x`.

Data Types: `double` | `single`

b — Lowpass interpolation filter coefficients

column vector

Lowpass interpolation filter coefficients, returned as a column vector.

Data Types: `double` | `single`

Algorithms

Interpolation increases the original sample rate of a sequence to a higher rate. It is the opposite of decimation. `interp` inserts zeros into the original signal and then applies a lowpass interpolating filter to the expanded sequence. The function uses the lowpass interpolation algorithm 8.1 described in [1]:

1. Expand the input vector to the correct length by inserting 0s between the original data values.
2. Design 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. Apply 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.

[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, No. 3, June 1975, pp. 301–309.

See Also

`decimate` | `downsample` | `interp1` | `intfilt` | `resample` | `spline` | `upfirdn` | `upsample`

Introduced before R2006a

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*p` nonzero samples, when used on a sequence interleaved with `l`-1 consecutive zeros every `l` samples, assuming an original bandlimitedness of `alpha` times the Nyquist frequency. The returned filter `b` is identical to that used by `interp`.

`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 `l` samples.

Examples

Digital Interpolation Filter

Design a digital interpolation filter to upsample a signal by seven, using the bandlimited method. Specify a "bandlimitedness" factor of 0.5 and use 2 × 2 samples in the interpolation.

```
upfac = 7;
alpha = 0.5;
h1 = intfilt(upfac,2,alpha);
```

The filter works best when the original signal is bandlimited to `alpha` times the Nyquist frequency. Create a bandlimited noise signal by generating 200 Gaussian random numbers and filtering the sequence with a 40th-order FIR lowpass filter. Reset the random number generator for reproducible results.

```
lowp = fir1(40,alpha);

rng('default')
x = filter(lowp,1,randn(200,1));
```

Increase the sample rate of the signal by inserting zeros between each pair of samples of `x`.

```
xr = upsample(x,upfac);
```

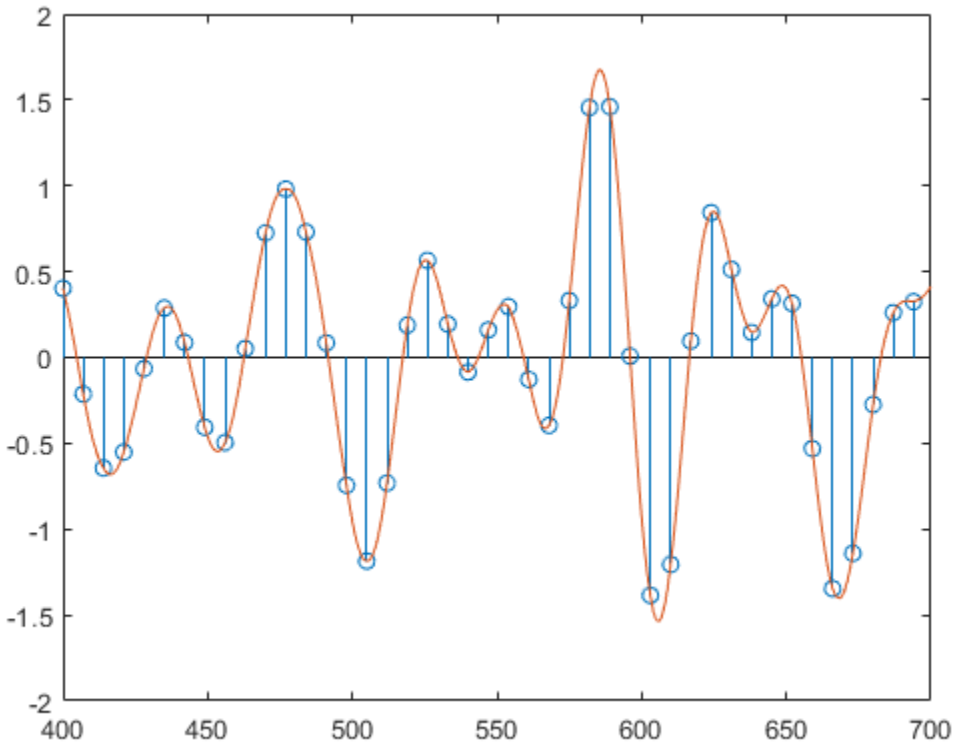
Use the `filter` function to produce an interpolated signal.

```
y = filter(h1,1,xr);
```

Compensate for the delay introduced by the filter. Plot the original and interpolated signals.

```
delay = mean(grpdelay(h1));
```

```
y(1:delay) = [];

stem(1:upfac:upfac*length(x),x)
hold on
plot(y)

xlim([400 700])
```

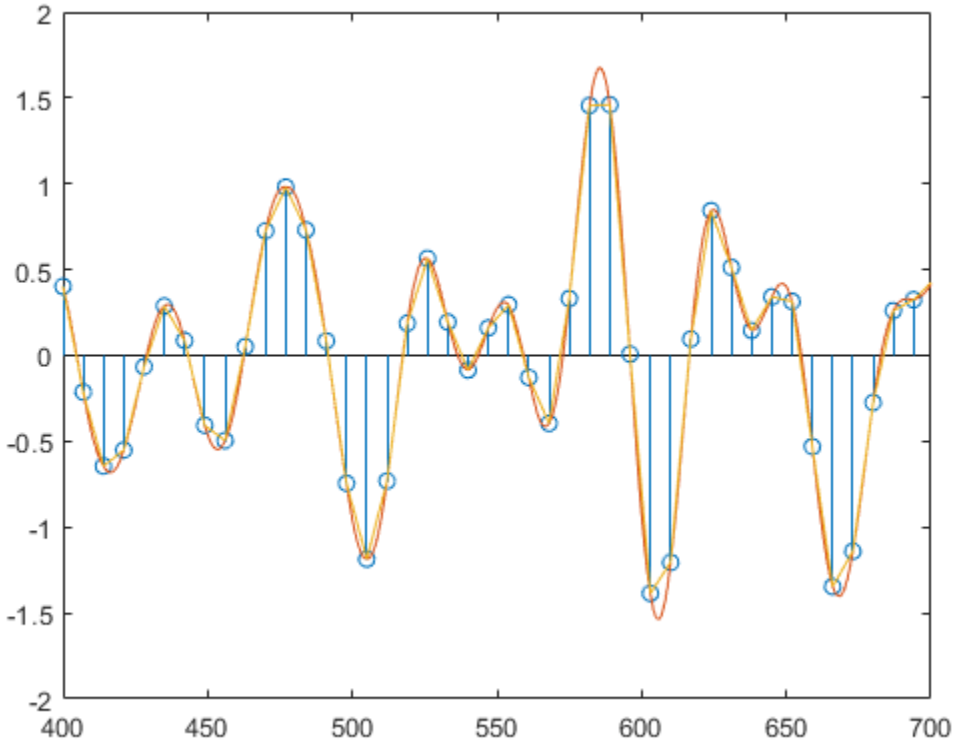


`intfilt` also performs Lagrange polynomial interpolation.

- First-order polynomial interpolation is just linear interpolation, which is accomplished with a triangular filter.
- Zeroth-order interpolation is accomplished with a moving average filter and resembles the output of a sample-and-hold display.

Interpolate the original signal and overlay the result.

```
h2 = intfilt(upfac,1,'Lagrange');

y2 = filter(h2,1,xr);
y2(1:floor(mean(grpdelay(h2)))) = [];

plot(y2)
hold off
```



Input Arguments

`l` — Number of samples

positive integer scalar

Number of samples, specified as a positive integer scalar. `intfilt` designs a linear phase FIR filter using a sequence interspersed with `l`-1 consecutive zeros every `l` samples.

`p` — Number of nonzero samples

positive integer scalar

Number of nonzero samples, specified as a positive integer scalar. `intfilt` designs a linear phase FIR filter that performs bandlimited interpolation using the nearest 2*`p` nonzero samples.

`alpha` — Inverse measure of transition bandwidth

scalar

Inverse measure of transition bandwidth, specified as a scalar. `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 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.

n — Order of Lagrange polynomial
positive integer scalar

Order of Lagrange polynomial specified as a positive integer scalar. The FIR filter performs nth-order Lagrange polynomial interpolation on a sequence interleaved with `l`-1 consecutive zeros every `l` samples. If both `n` and `l` are even, the filter designed is not linear phase.

'Lagrange' — Polynomial interpolation method
'Lagrange'

Polynomial interpolation method, specified as 'Lagrange'.

Output Arguments

b — Filter coefficients
vector

Filter coefficients, returned as a vector. Elements of `b` are the coefficients of an FIR filter. If `alpha` is specified, it assumes an original bandlimitedness of `alpha` times the Nyquist frequency. `b` is length 2*`l`*p-1.

For the nth-order Lagrange polynomial interpolation, `b` has length (`n`+1)*`l` for `n` even, and length (`n`+1)*`l`-1 for `n` odd.

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. Both types of filters are basically lowpass and have a gain of `l` in the passband.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also
`decimate` | `downsample` | `interp` | `resample` | `upsample`

Introduced before R2006a

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(___,iter)
[b,a] = invfreqs(___,tol)
[b,a] = invfreqs(___,'trace')
[b,a] = invfreqs(h,w,'complex',n,m,___)
```

Description

`[b,a] = invfreqs(h,w,n,m)` returns the real numerator and denominator coefficient vectors `b` and `a` of the transfer function `h`.

`[b,a] = invfreqs(h,w,n,m,wt)` weights the fit-errors versus frequency using `wt`.

`[b,a] = invfreqs(___,iter)` provides an algorithm that guarantees stability of the resulting linear system by searching for the best fit using a numerical, iterative scheme. This syntax can include any combination of input arguments from the previous syntaxes.

`[b,a] = invfreqs(___,tol)` uses `tol` to decide convergence of the iterative algorithm.

`[b,a] = invfreqs(___,'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 π.

Examples

Transfer Function to Frequency Response Conversion

Convert a simple transfer function to frequency-response data and then back to the original filter coefficients.

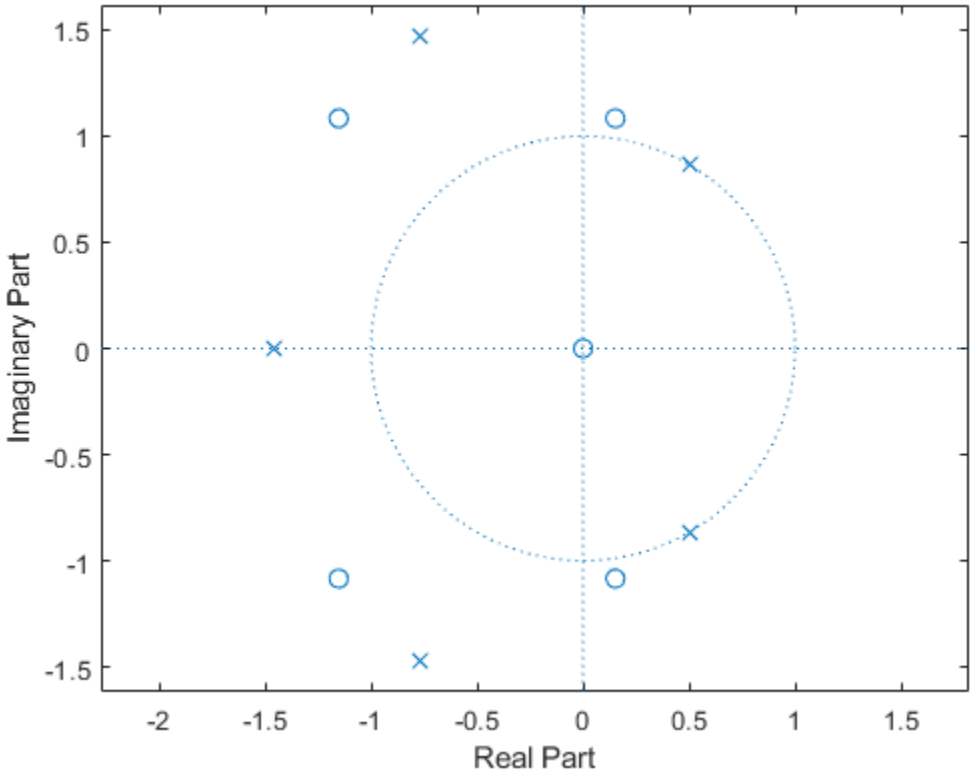
```
a = [1 2 3 2 1 4];
b = [1 2 3 2 3];

[h,w] = freqs(b,a,64);
[bb,aa] = invfreqs(h,w,4,5)
```

```
bb = 1×5

    1.0000    2.0000    3.0000    2.0000    3.0000


aa = 1×6
```

```
    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 `aa` has poles with positive real part. View the poles of `bb` and `aa`.

```
zplane(bb,aa)
```



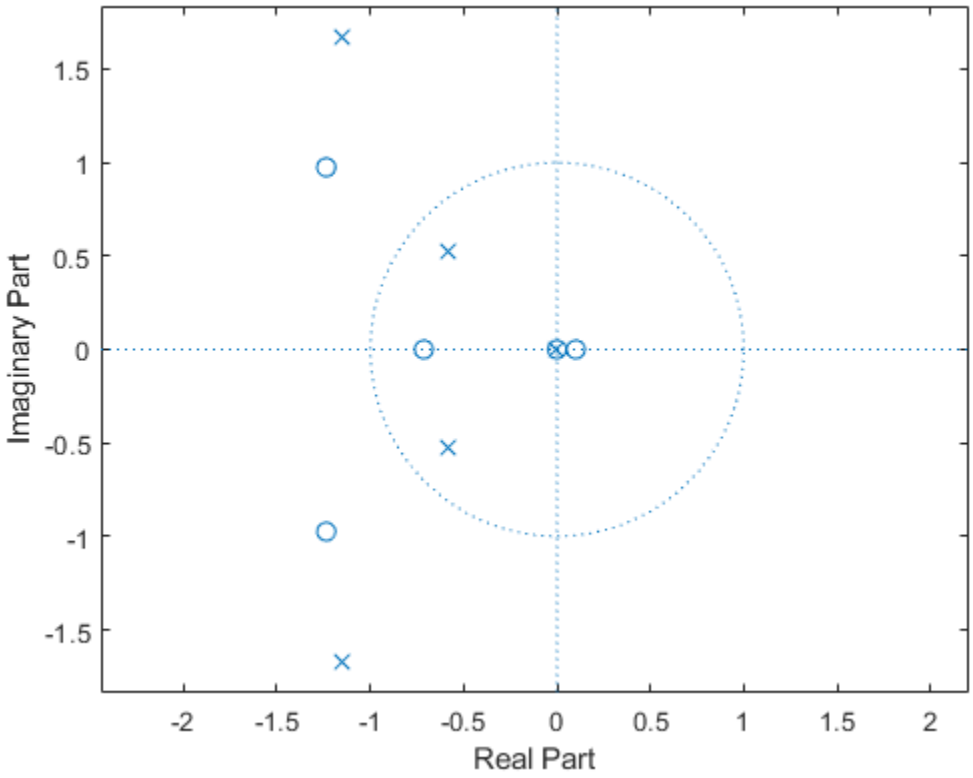
Use the iterative algorithm of `invfreqs` to find a stable approximation to the system.

```
[bbb,aaa] = invfreqs(h,w,4,5,[],30)
```

```
bbb = 1×5

    0.6816    2.1015    2.6694    0.9113   -0.1218

aaa = 1×6

    1.0000    3.4676    7.4060    6.2102    2.5413    0.0001
```

Verify that the system is stable by plotting the new poles.

```
zplane(bbb,aaa)
```



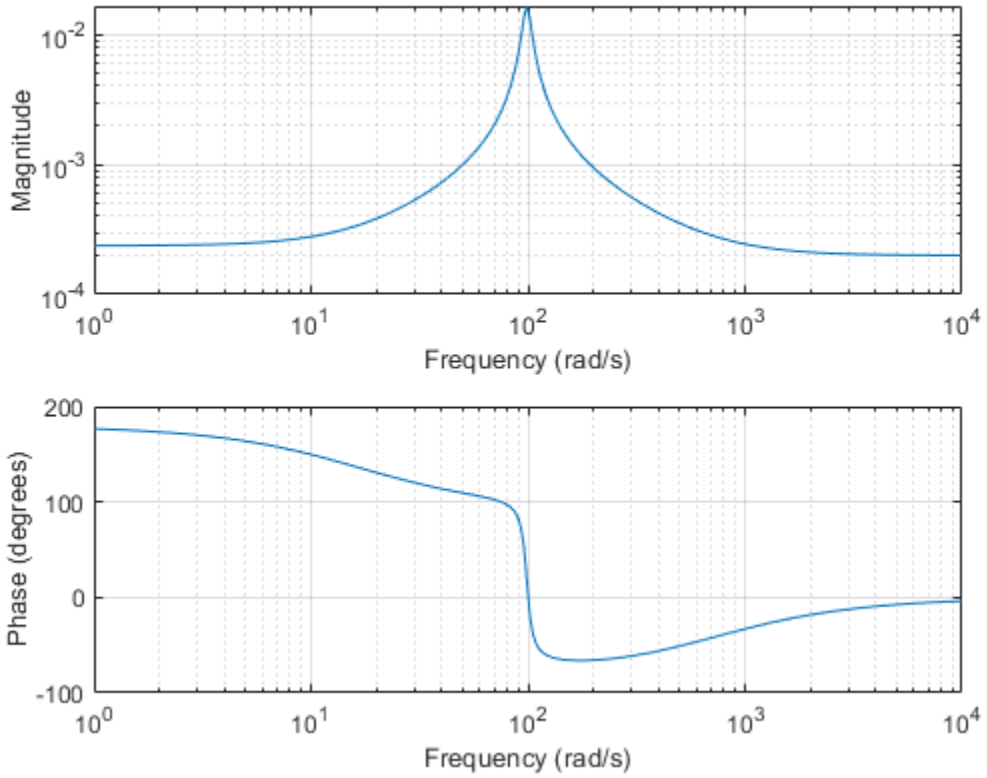
Continuous-Time Transfer Function

Generate two vectors, `mag` and `phase`, that simulate magnitude and phase data gathered in a laboratory. Also generate a vector, `w`, of frequencies.

```
rng('default')

fs = 1000;
t = 0:1/fs:2;
mag = periodogram(sin(2*pi*100*t)+randn(size(t))/10,[],[],fs);
phase = randn(size(mag))/10;
w = linspace(0,fs/2,length(mag))';
```

Use `invfreqs` to convert the data into a continuous-time transfer function. Plot the result.

```
[b,a] = invfreqs(mag.*exp(1j*phase),w,2,2,[],4);

freqs(b,a)
```



Input Arguments

`h` — Frequency response

vector

Frequency response, specified as a vector.

`w` — Angular frequencies

vector

Angular frequencies at which `h` is computed, specified as a vector.

`n`, `m` — Desired order

positive integer scalar

Desired order of the numerator and denominator polynomials, specified as positive integer scalars.

Data Types: `single` | `double`

`wt` — Weighting factors

vector

Weighting factors, specified as a vector. `wt` is a vector of weighting factors that is the same length as `w`.

Data Types: `single` | `double`

`iter` — Number of iterations in the search algorithm
positive real scalar

Number of iterations in the search algorithm, specified as a positive real scalar. The `iter` parameter tells `invfreqs` to end the iteration when the algorithm has converged to a solution, or after `iter` iterations, whichever occurs first.

`tol` — Tolerance
`0.01` (default) | scalar

Tolerance, specified as a scalar. `invfreqs` defines convergence as occurring when the norm of the (modified) gradient vector is less than `tol`.

To obtain a weight vector of all ones, use

```
invfreqs(h,w,n,m,[],iter,tol)
```

Output Arguments

`b`, `a` — Transfer function coefficients
vectors

Transfer function coefficients, returned as vectors. Express the transfer function in terms of `b` and `a` as

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

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 0.5π rad/sample.

Data Types: `double` | `single`
Complex Number Support: Yes

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

$$\min_{b,a} \sum_{k=1}^{n} wt(k)\left|h(k)A(w(k)) - B(w(k))\right|^2$$

by creating a system of linear equations and solving them with the MATLAB \ operator. Here $A(w(k))$ and $B(w(k))$ are the Fourier transforms of the polynomials `a` and `b`, respectively, at the frequency $w(k)$, and n is the number of frequency points (the length of `h` and `w`). This algorithm is based on Levi [1]. Several variants have been suggested in the literature, where the weighting function `wt` gives less attention to high frequencies.

The superior (“output-error”) algorithm uses the damped Gauss-Newton method for iterative search [2], with the output of the first algorithm as the initial estimate. This solves the direct problem of minimizing the weighted sum of the squared error between the actual and the desired frequency response points.

$$\min_{b,a} \sum_{k=1}^{n} wt(k)\left|h(k) - \frac{B(w(k))}{A(w(k))}\right|^2$$

References

[1] Levi, E. C. “Complex-Curve Fitting.” *IRE Trans. on Automatic Control*. Vol. AC-4, 1959, pp. 37–44.

[2] Dennis, J. E., Jr., and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Englewood Cliffs, NJ: Prentice-Hall, 1983.

See Also

`freqs` | `freqz` | `invfreqz` | `prony`

Introduced before R2006a

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(___,iter)
[b,a] = invfreqz(___,tol)
[b,a] = invfreqz(___,'trace')
[b,a] = invfreqz(h,w,'complex',n,m,___)
```

Description

`[b,a] = invfreqz(h,w,n,m)` returns the real numerator and denominator coefficient vectors `b` and `a` of the transfer function `h`.

`[b,a] = invfreqz(h,w,n,m,wt)` weights the fit-errors versus frequency using `wt`.

`[b,a] = invfreqz(___,iter)` provides an algorithm that guarantees stability of the resulting linear system by searching for the best fit using a numerical, iterative scheme. This syntax can include any combination of input arguments from the previous syntaxes.

`[b,a] = invfreqz(___,tol)` uses `tol` to decide convergence of the iterative algorithm.

`[b,a] = invfreqz(___,'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 π.

Examples

Stable Approximate Transfer Function

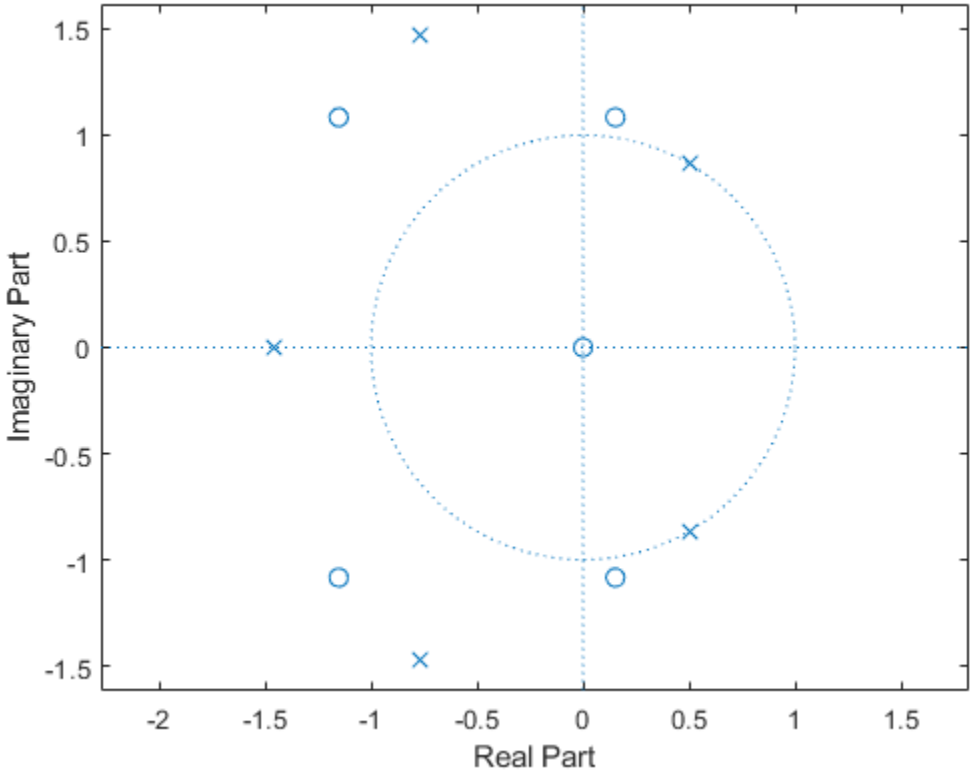
Convert a simple transfer function to frequency response data and then back to the original filter coefficients. Sketch the zeros and poles of the function.

```
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)
```

```
bb = 1×5

    1.0000    2.0000    3.0000    2.0000    3.0000


aa = 1×6
```

```
    1.0000    2.0000    3.0000    2.0000    1.0000    4.0000
```

```
zplane(bb,aa)
```

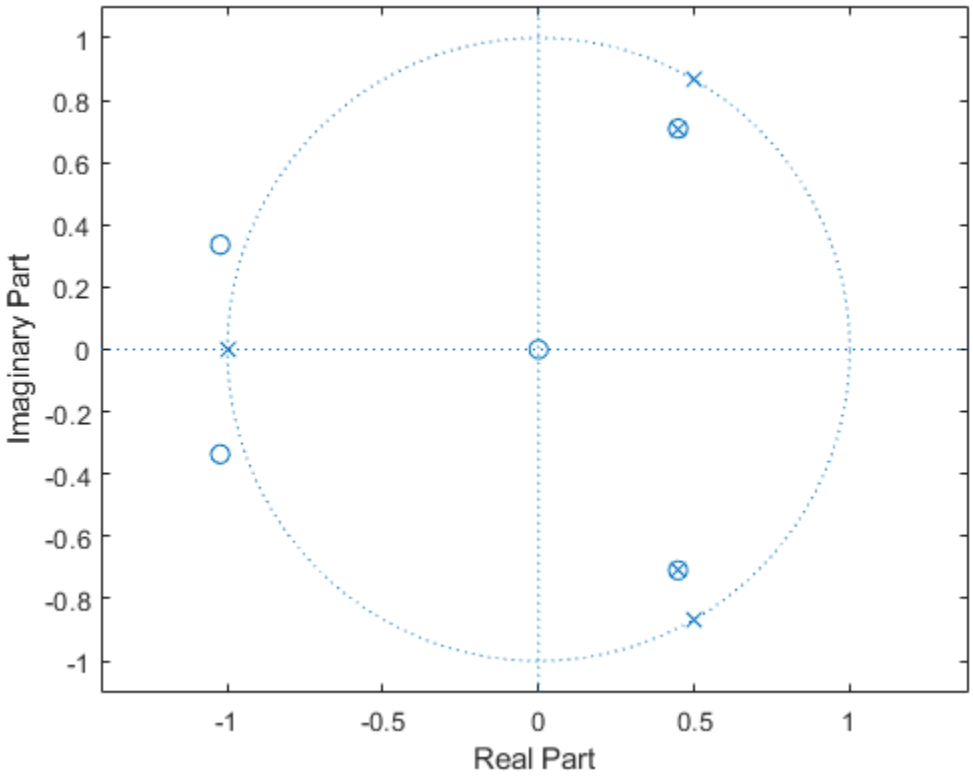


`bb` and `aa` are equivalent to `b` and `a`, respectively. However, the system is unstable because it has poles outside the unit circle. Use `invfreqz`'s iterative algorithm to find a stable approximation to the system. Verify that the poles are within the unit circle.

```
[bbb,aaa] = invfreqz(h,w,4,5,[],30)
```

```
bbb = 1×5

    0.2427    0.2788    0.0069    0.0971    0.1980

aaa = 1×6

    1.0000   -0.8944    0.6954    0.9997   -0.8933    0.6949
```

```
zplane(bbb,aaa)
```



Input Arguments

`h` — Frequency response
vector

Frequency response, specified as a vector.

`w` — Angular frequencies
vector

Angular frequencies at which `h` is computed, specified as a vector.

`n`, `m` — Desired order
positive integer scalar

Desired order of the numerator and denominator polynomials, specified as positive integer scalars.

`wt` — Weighting factors
vector

Weighting factors, specified as a vector. `wt` is a vector of weighting factors that is the same length as `w`.

`iter` — Number of iterations in the search algorithm
positive real scalar

Number of iterations in the search algorithm, specified as a positive real scalar. The `iter` parameter tells `invfreqz` to end the iteration when the solution has converged, or after `iter` iterations, whichever comes first.

`tol` — Tolerance
`0.01` (default) | scalar

Tolerance, specified as a scalar. `invfreqz` defines convergence as occurring when the norm of the (modified) gradient vector is less than `tol`.

To obtain a weight vector of all ones, use

```
invfreqz(h,w,n,m,[],iter,tol)
```

Output Arguments

`b`, `a` — Transfer function coefficients
vectors

Transfer function coefficients, returned as vectors. Express the transfer function in terms of `b` and `a` as

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

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 0.5π rad/sample.

Data Types: `double` | `single`
Complex Number Support: Yes

Algorithms

By default, `invfreqz` uses an equation error method to identify the best model from the data. This finds `b` and `a` in

$$\min_{b,a} \sum_{k=1}^{n} wt(k)\left|h(k)A(w(k)) - B(w(k))\right|^2$$

by creating a system of linear equations and solving them with the MATLAB \ operator. Here $A(\omega(k))$ and $B(\omega(k))$ are the Fourier transforms of the polynomials `a` and `b`, respectively, at the frequency $\omega(k)$, and n is the number of frequency points (the length of `h` and `w`). This algorithm is a based on Levi [1].

The superior (“output-error”) algorithm uses the damped Gauss-Newton method for iterative search [2], with the output of the first algorithm as the initial estimate. This solves the direct problem of minimizing the weighted sum of the squared error between the actual and the desired frequency response points.

$$\min_{b,a} \sum_{k=1}^{n} wt(k)\left|h(k) - \frac{B(w(k))}{A(w(k))}\right|^2$$

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`

Introduced before R2006a

isallpass

Determine whether filter is allpass

Syntax

```
flag = isallpass(b,a)
flag = isallpass(sos)
flag = isallpass(d)
flag = isallpass(...,tol)
```

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.

Examples

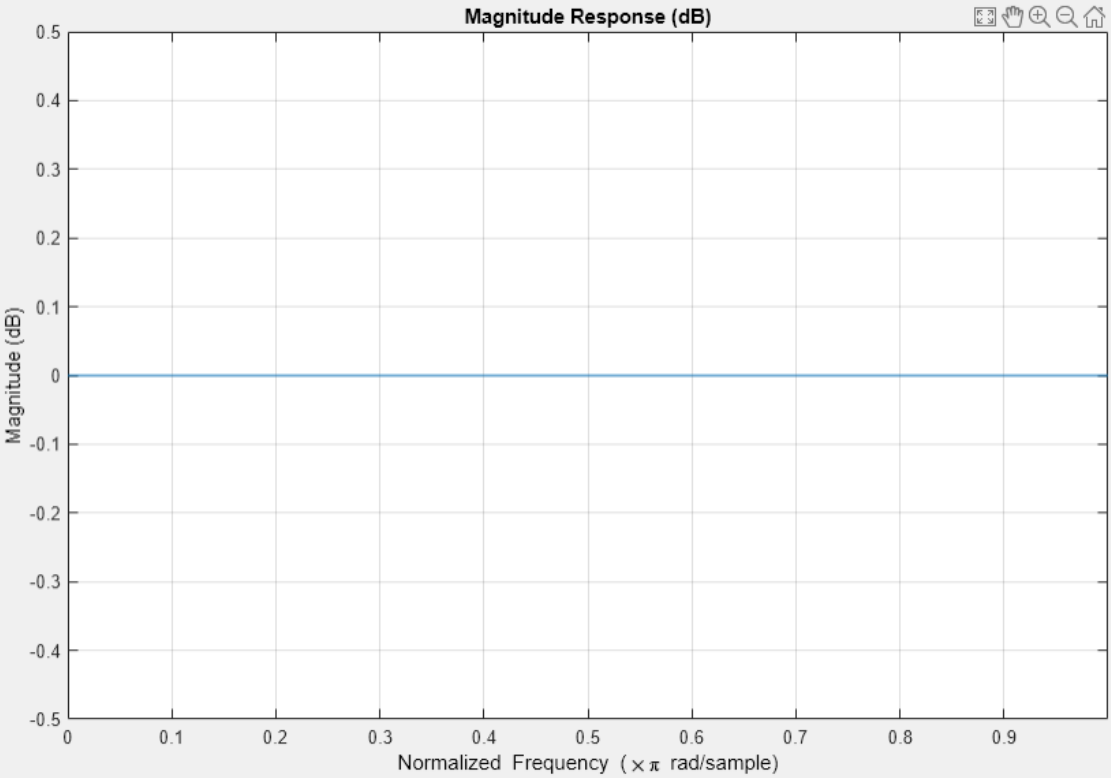
Allpass Filters

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)
```

```
flag = logical
   1
```

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

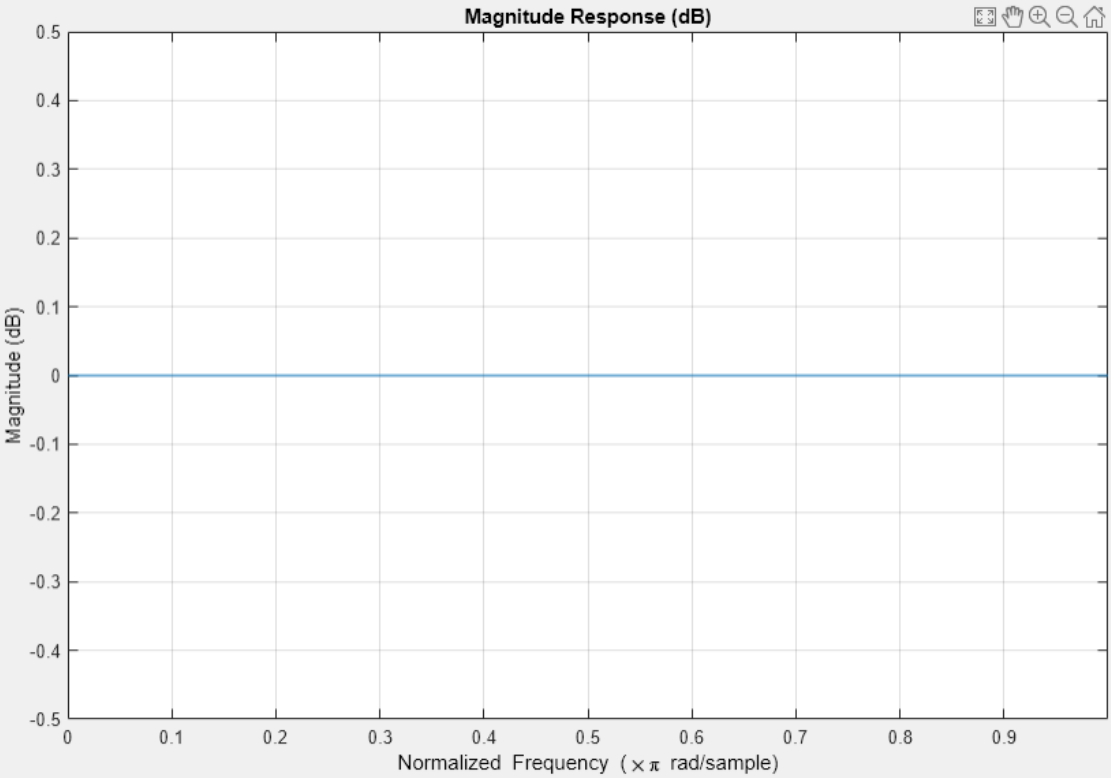


Create a lattice allpass filter and verify that the filter is allpass.

```
k = [1/2 1/3 1/4 1/5];
[b,a] = latc2tf(k,'allpass');
flag_isallpass = isallpass(b,a)
```

```
flag_isallpass = logical
   1
```

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



See Also

`designfilt` | `digitalFilter` | `islinphase` | `ismaxphase` | `isminphase` | `isstable`

Introduced in R2013a

iscola

Determine whether window-overlap combination is COLA compliant

Syntax

```
tf = iscola(window,noverlap)
tf = iscola(window,noverlap,method)

[tf,m] = iscola( ___ )
[tf,m,maxDeviation] = iscola( ___ )
```

Description

`tf = iscola(window,noverlap)` checks that the specified window and overlap satisfy the "Constant Overlap-Add (COLA) Constraint" on page 1-1132 to ensure that the "Inverse Short-Time Fourier Transform" on page 1-1130 results in perfect reconstruction for nonmodified spectra.

`tf = iscola(window,noverlap,method)` specifies the inversion method to use.

`[tf,m] = iscola( ___ )` also returns the median of the COLA summation. You can use these output arguments with any of the previous input syntaxes.

`[tf,m,maxDeviation] = iscola( ___ )` returns the maximum deviation from the median `m`.

Examples

Check COLA Compliance For Root-Hann window

Create a periodic root-Hann window of length 120. Test whether the window is COLA compliant with a 50% overlap.

```
win = sqrt(hann(120,'periodic'));
noverlap = 60;
```

Check whether the window is COLA compliant with a 50% overlap.

```
tf = iscola(win,noverlap)
```

```
tf = logical
   1
```

COLA Compliance of Periodic Hamming Window

Create a periodic Hamming window of length 256. Set the method of Overlap-Add as `'ola'`.

```
window = hamming(256,'periodic');
method = 'ola';
noverlap = 128;
```

Test whether the window is COLA compliant with a 50% overlap. Also calculate the median of the COLA summation and the maximum deviation from that summation.

```
[tf,m,maxDeviation] =  iscola(window,noverlap,method)
```

```
tf = logical
   1


m = 1.0800

maxDeviation = 2.2204e-16
```

Input Arguments

`window` — Analysis window

vector

Analysis window, specified as a vector.

Example: `win = bartlett(120)` is a Bartlett window of length 120.

Data Types: `double` | `single`

`noverlap` — Number of overlapped samples

positive scalar

Number of overlapped samples, specified as a positive integer smaller than the length of `window`.

Data Types: `double` | `single`

`method` — Method of overlap-add

`'wola'` (default) | `'ola'`

Method of overlap-add, specified as:

- `'wola'` — Weighted Overlap-Add.
- `'ola'` — Overlap-Add.

Output Arguments

`tf` — COLA compliance

logical scalar

COLA compliance, returned as a logical scalar. If the function returns a `1` (true), then the window and overlap length satisfy the COLA constraint.

`m` — Median

real scalar

Median of the COLA summation, returned as a real scalar. If the inputs are COLA compliant, then `m` is equal to the COLA summation constant.

`maxDeviation` — Maximum deviation
real scalar

Maximum deviation from the median `m`. If `window` and `noverlap` are COLA compliant, the `maxDeviation` is close to the expected numeric precision error of the COLA summation.

Note You can conclude strong COLA-compliance if `m = 1` and `maxDeviation` is close to the numeric precision error.

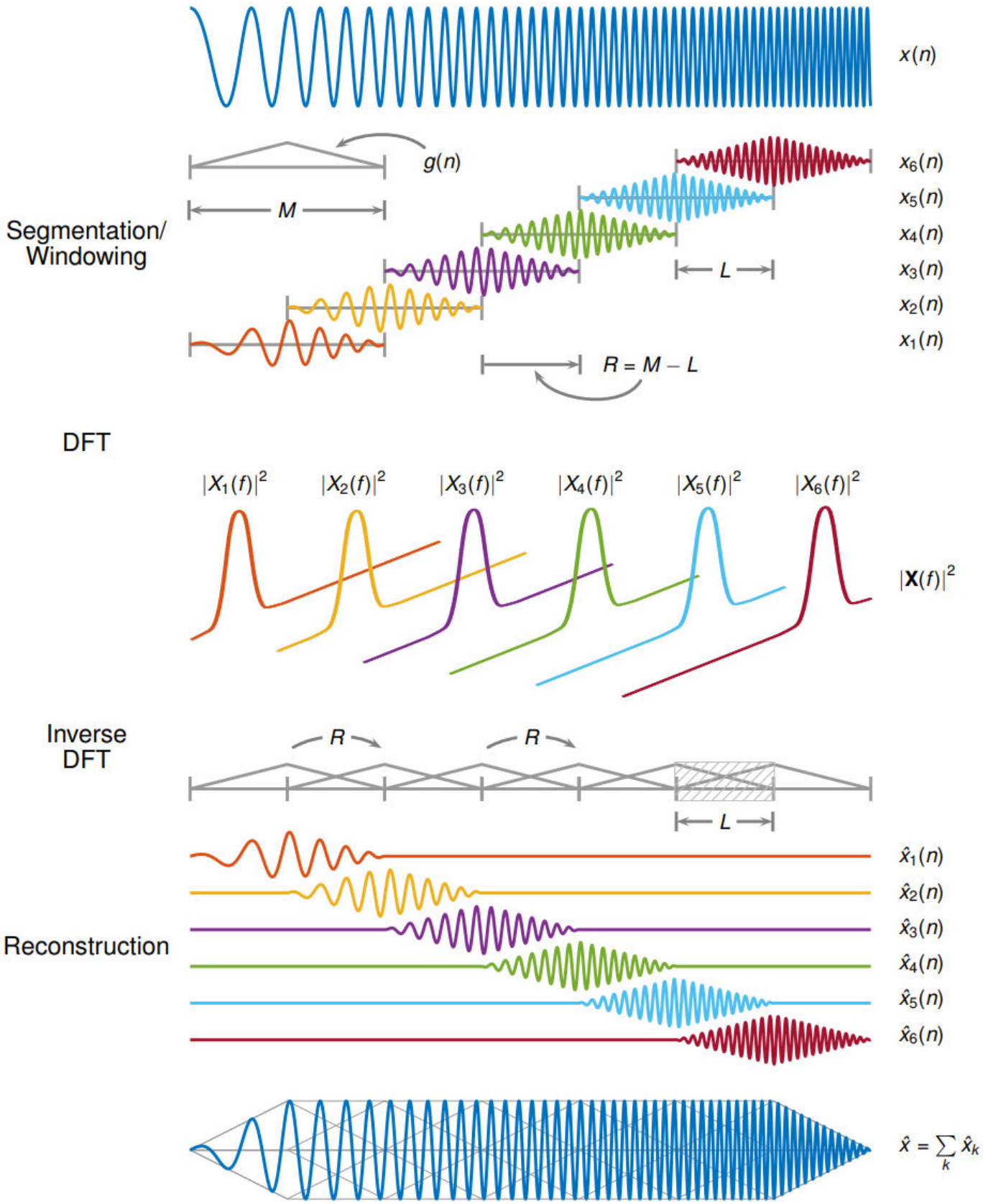
More About

Inverse Short-Time Fourier Transform

The inverse short-time Fourier transform is computed by taking the IFFT of each DFT vector of the STFT and overlap-adding the inverted signals. The ISTFT is calculated as follows:

$$\begin{aligned} x(n) &= \int_{-1/2}^{1/2} \sum_{m=-\infty}^{\infty} X_m(f) e^{j2\pi f n} df \\ &= \sum_{m=-\infty}^{\infty} \int_{-1/2}^{1/2} X_m(f) e^{j2\pi f n} df \\ &= \sum_{m=-\infty}^{\infty} x_m(n) \end{aligned}$$

where R is the hop size between successive DFTs, X_m is the DFT of the windowed data centered about time mR and $x_m(n) = x(n)\, g(n - mR)$. The inverse STFT is a perfect reconstruction of the original signal as long as $\sum_{m=-\infty}^{\infty} g^{a+1}(n - mR) = c \;\forall n \in \mathbb{Z}$ where the analysis window $g(n)$ was used to window the original signal and c is a constant. The following figure depicts the steps followed in reconstructing the original signal.


x(n)
g(n)
M
L
R = M − L
x6(n)
x5(n)
x4(n)
x3(n)
x2(n)
x1(n)
Segmentation/
Windowing
DFT
|X1(f)|2
|X2(f)|2
|X3(f)|2
|X4(f)|2
|X5(f)|2
|X6(f)|2
|X(f)|2
Inverse
DFT
R
R
L
Reconstruction
x̂1(n)
x̂2(n)
x̂3(n)
x̂4(n)
x̂5(n)
x̂6(n)
x̂ = ∑k x̂k

Constant Overlap-Add (COLA) Constraint

To ensure successful reconstruction of nonmodified spectra, the analysis window must satisfy the COLA constraint. In general, if the analysis window satisfies the condition $\sum_{m=-\infty}^{\infty} g^{a+1}(n-mR) = c \ \forall n \in \mathbb{Z}$, the window is considered to be COLA-compliant. Additionally, COLA compliance can be described as either weak or strong.

- Weak COLA compliance implies that the Fourier transform of the analysis window has zeros at frame-rate harmonics such that

  $$G(f_k) = 0, \qquad k = 1, 2, \ldots, R-1, \qquad f_k \triangleq \frac{k}{R}.$$

  Alias cancellation is disturbed by spectral modifications. Weak COLA relies on alias cancellation in the frequency domain. Therefore, perfect reconstruction is possible using weakly COLA-compliant windows as long as the signal has not undergone any spectral modifications.
- For strong COLA compliance, the Fourier transform of the window must be bandlimited consistently with downsampling by the frame rate such that

  $$G(f) = 0, \qquad f \geq \frac{1}{2R}.$$

  This equation shows that no aliasing is allowed by the strong COLA constraint. Additionally, for strong COLA compliance, the value of the constant c must equal 1. In general, if the short-time spectrum is modified in any way, a stronger COLA compliant window is preferred.

You can use the `iscola` function to check for weak COLA compliance. The number of summations used to check COLA compliance is dictated by the window length and hop size. In general, it is common to use $a = 1$ in $\sum_{m=-\infty}^{\infty} g^{a+1}(n-mR) = c \ \forall n \in \mathbb{Z}$ for weighted overlap-add (WOLA), and $a = 0$ for overlap-add (OLA). By default, `istft` uses the WOLA method, by applying a synthesis window before performing the overlap-add method.

In general, the synthesis window is the same as the analysis window. You can construct useful WOLA windows by taking the square root of a strong OLA window. You can use this method for all nonnegative OLA windows. For example, the root-Hann window is a good example of a WOLA window.

Perfect Reconstruction

In general, computing the STFT of an input signal and inverting it does not result in perfect reconstruction. If you want the output of ISTFT to match the original input signal as closely as possible, the signal and the window must satisfy the following conditions:

- Input size — If you invert the output of `stft` using `istft` and want the result to be the same length as the input signal `x`, the value of $k = \frac{(length(x) - noverlap)}{(length(window) - noverlap)}$ must be an integer.
- COLA compliance — Use COLA-compliant windows, assuming that you have not modified the short-time Fourier transform of the signal.
- Padding — If the length of the input signal is such that the value of k is not an integer, zero-pad the signal before computing the short-time Fourier transform. Remove the extra zeros after inverting the signal.

References

[1] Allen, J. B. "Short Term Spectral Analysis, Synthesis, and Modification by Discrete Fourier Transform." *IEEE Transactions on Acoustics, Speech and Signal Processing.* Vol. 25, Number 3, June 1977, pp. 235–238.

[2] Griffin, Daniel W., and Jae S. Lim. "Signal Estimation from Modified Short-Time Fourier Transform." *IEEE Transactions on Acoustics, Speech and Signal Processing.* Vol. 32, Number 2, April 1984, pp. 236–243.

[3] Sharpe, Bruce. *Invertibility of Overlap-Add Processing*. https://gauss256.github.io/blog/cola.html, accessed July 2019.

[4] Smith, Julius Orion. *Spectral Audio Signal Processing*. https://ccrma.stanford.edu/~jos/sasp/, online book, 2011 edition, accessed Nov 2018.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`pspectrum` | `stft` | `istft` | `bartlett`

Introduced in R2019a

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 0.6π 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)
```

```
isd = logical
   1
```

```
fs = single(fd);
iss = isdouble(fs)
```

```
iss = logical
   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 0.5π rad/sample.

Output Arguments

flag — Type identification

logical scalar

Type identification, returned as a logical scalar.

See Also

`designfilt` | `digitalFilter` | `double` | `issingle` | `single`

Introduced in R2014a

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.

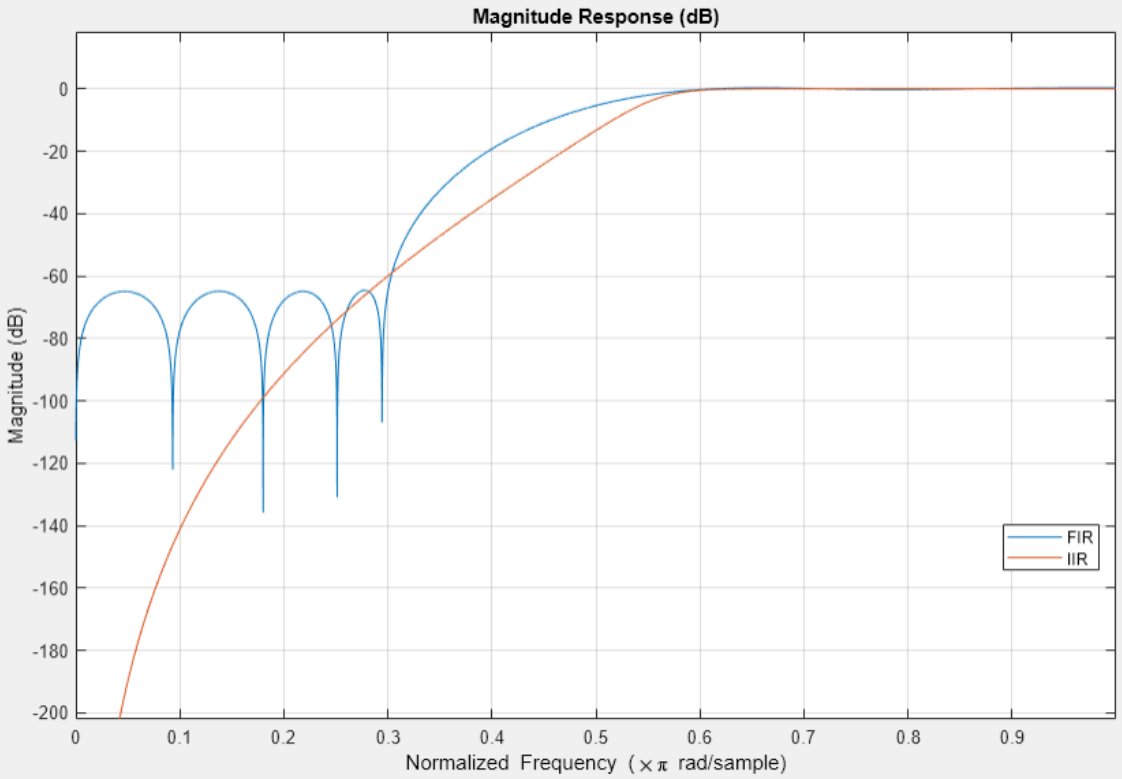
```
fir = designfilt('highpassfir','StopbandFrequency',0.3,'PassbandFrequency',0.6);
iir = designfilt('highpassiir','StopbandFrequency',0.3,'PassbandFrequency',0.6);
isfirFIR = isfir(fir)
```

```
isfirFIR = logical
   1
```

```
isiirFIR = isfir(iir)
```

```
isiirFIR = logical
   0
```

```
fvt = fvtool(fir,iir);
legend(fvt,'FIR','IIR')
```



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 0.5π 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`

Introduced in R2014a

islinphase

Determine whether filter has linear phase

Syntax

```
flag = islinphase(b,a)
flag = islinphase(sos)
flag = islinphase(d)
flag = islinphase(...,tol)
```

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)`.

Examples

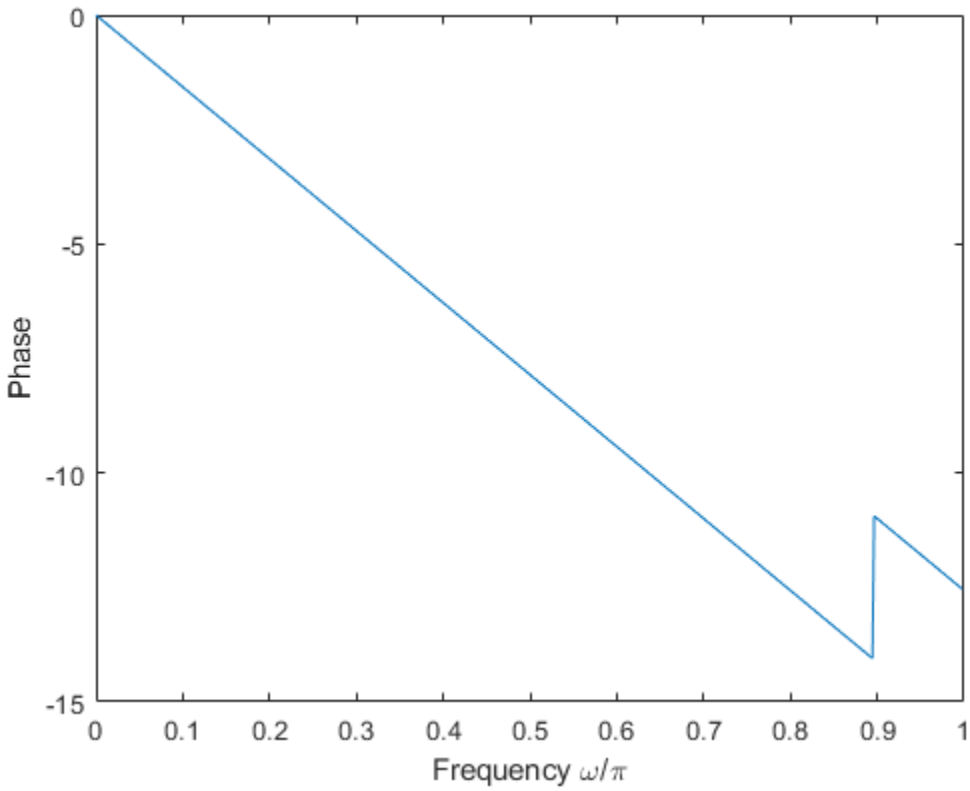
Linear and Nonlinear Phase

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)
```

```
flag = logical
   1
```

```
[phs,w] = phasez(d);

plot(w/pi,phs)
xlabel('Frequency \omega/\pi')
ylabel('Phase')
```

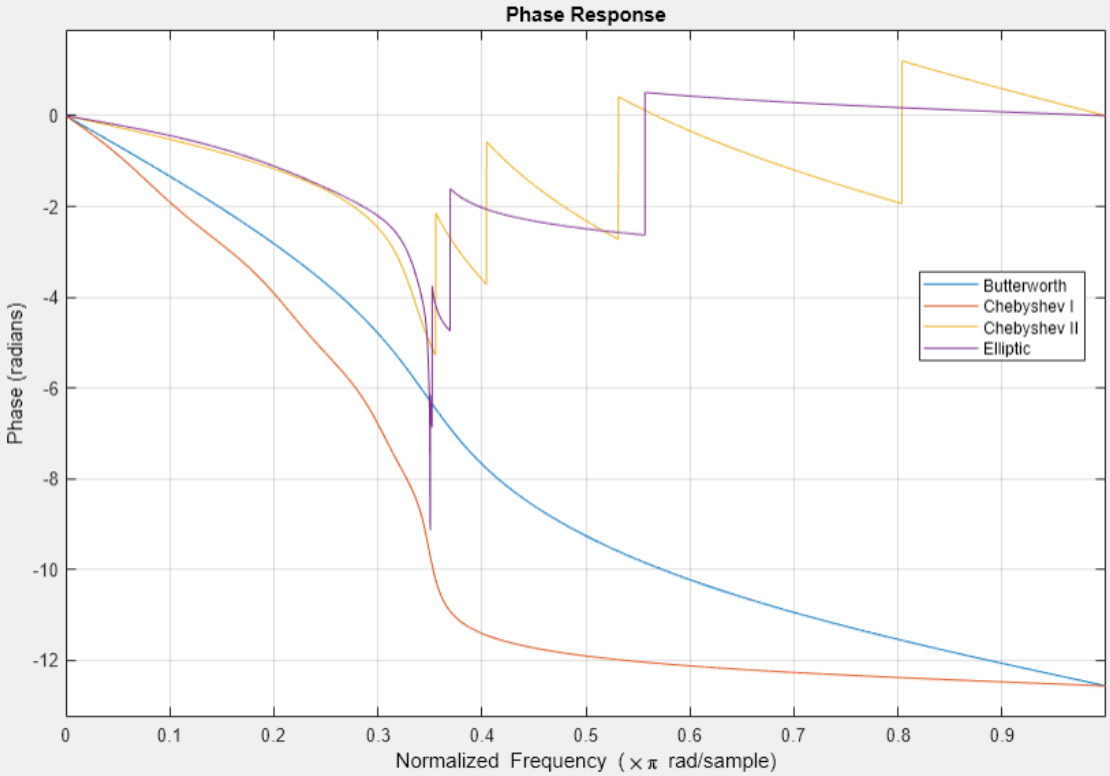


IIR filters in general do not have linear phase. Verify the statement by constructing eighth-order Butterworth, Chebyshev, and elliptic filters with similar specifications.

```
ord = 8;
Wcut = 0.35;
atten = 20;
rippl = 1;

[zb,pb,kb] = butter(ord,Wcut);
sosb = zp2sos(zb,pb,kb);

[zc,pc,kc] = cheby1(ord,rippl,Wcut);
sosc = zp2sos(zc,pc,kc);

[zd,pd,kd] = cheby2(ord,atten,Wcut);
sosd = zp2sos(zd,pd,kd);

[ze,pe,ke] = ellip(ord,rippl,atten,Wcut);
sose = zp2sos(ze,pe,ke);
```

Plot the phase responses of the filters. Determine whether they have linear phase.

```
fv = fvtool(sosb,sosc,sosd,sose,'Analysis','phase');
legend(fv,'Butterworth','Chebyshev I','Chebyshev II','Elliptic')
```




```
phs = [islinphase(sosb) islinphase(sosc) ...
       islinphase(sosd) islinphase(sose)]
```

```
phs = 1x4 logical array

   0   0   0   0
```

See Also

designfilt | digitalFilter | isallpass | ismaxphase | isminphase | isstable

Introduced in R2013a

isminphase

Determine whether filter is minimum phase

Syntax

```
flag = isminphase(b,a)
flag = isminphase(sos)
flag = isminphase(d)
flag = isminphase( ___ ,tol)
```

Description

`flag = isminphase(b,a)` returns a logical output equal to `1` if the filter specified by numerator coefficients `b` and denominator coefficients `a` is minimum phase.

`flag = isminphase(sos)` returns `1` if the filter specified by second order sections matrix `sos` is minimum phase.

`flag = isminphase(d)` returns `1` if the digital filter `d` is 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.

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 is minimum phase.

```
[z,p,k] = butter(6,0.15);
SOS = zp2sos(z,p,k);
min_flag = isminphase(SOS)
```

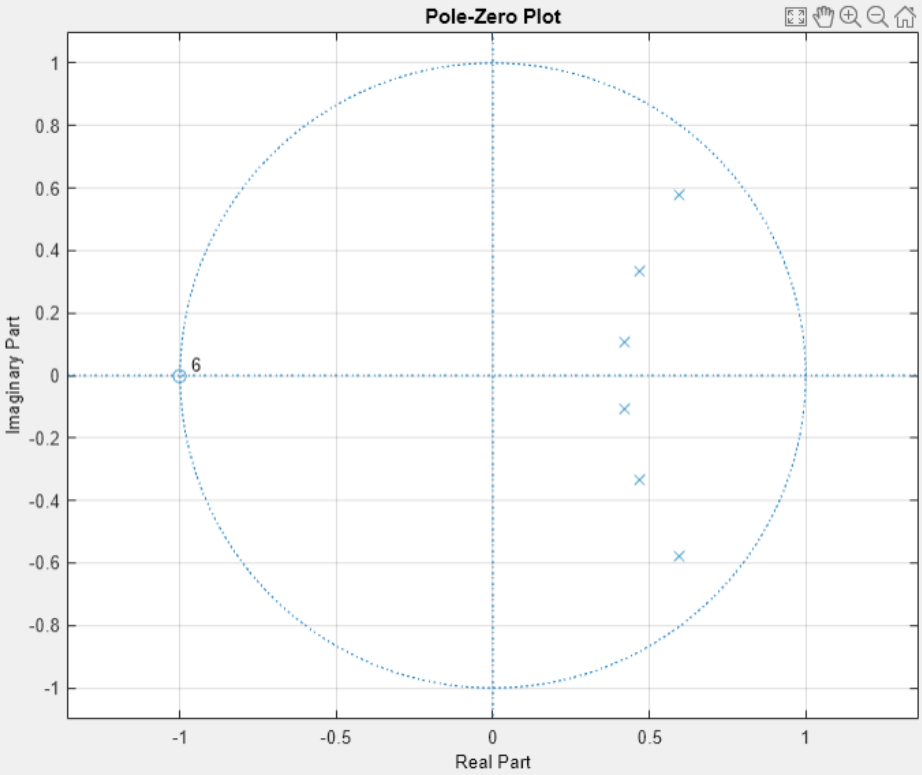
```
min_flag = logical
   1
```

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)
```

```
d_flag = logical
   1
```

```
zplane(d)
```



Given a filter defined with a set of single-precision numerator and denominator coefficients, check if it is minimum phase for different tolerance values.

```
b = single([1 1.00001]);
a = single([1 0.45]);
min_flag1 = isminphase(b,a)
```

```
min_flag1 = logical
   0
```

```
min_flag2 = isminphase(b,a,1e-3)
```

```
min_flag2 = logical
   1
```

Input Arguments

b, a — Filter coefficients

vector

Filter coefficients, specified as vectors.

Data Types: `single` | `double`

`sos` — Second order sections
k-by-6 matrix

Second order sections, specified as 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)]`.

Data Types: `double`

`d` — Digital filter
`digitalFilter` object

Digital filter, specified as a `digitalFilter` object.

`tol` — Tolerance
`eps^(2/3)` (default) | positive scalar

Tolerance, specified as a positive scalar. The tolerance value determines when two numbers are close enough to be considered equal.

Data Types: `double`

Output Arguments

`flag` — Logical output
`1` (`true`) | `0` (`false`)

Logical output, returned as `1` or `0`. The function returns `1` when the input is a minimum phase filter.

See Also
`designfilt` | `digitalFilter` | `isallpass` | `islinphase` | `ismaxphase` | `isstable`

Introduced in R2013a

ismaxphase

Determine whether filter is maximum phase

Syntax

```
flag = ismaxphase(b,a)
flag = ismaxphase(sos)
flag = ismaxphase(d)
flag = ismaxphase(...,tol)
```

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)`.

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)
```

```
max_flag = logical
   1
```

```
min_flag = isminphase(bmin)
```

```
min_flag = logical
   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_flag1 = logical
   0
```

```
max_flag2 = ismaxphase(b,a,1e-3)
```

```
max_flag2 = logical
   1
```

See Also

designfilt | digitalFilter | isallpass | islinphase | isminphase | isstable

Introduced in R2013a

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 0.6π 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)
```

```
isd = logical
   0
```

```
fs = single(fd);
iss = issingle(fs)
```

```
iss = logical
   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 0.5π rad/sample.

Output Arguments

flag — Type identification

logical scalar

Type identification, returned as a logical scalar.

See Also

`designfilt` | `digitalFilter` | `double` | `single` | `isdouble`

Introduced in R2014a

isstable

Determine whether filter is stable

Syntax

```
flag = isstable(b,a)
flag = isstable(sos)
flag = isstable(d)
```

Description

`flag = isstable(b,a)` returns a logical output equal to `1` if the filter specified by numerator coefficients `b` and denominator coefficients `a` is stable.

`flag = isstable(sos)` returns `1` if the filter specified by second order sections matrix `sos` is stable.

`flag = isstable(d)` returns `1` if the digital filter `d` is stable. Use `designfilt` to generate `d` based on frequency-response specifications.

Examples

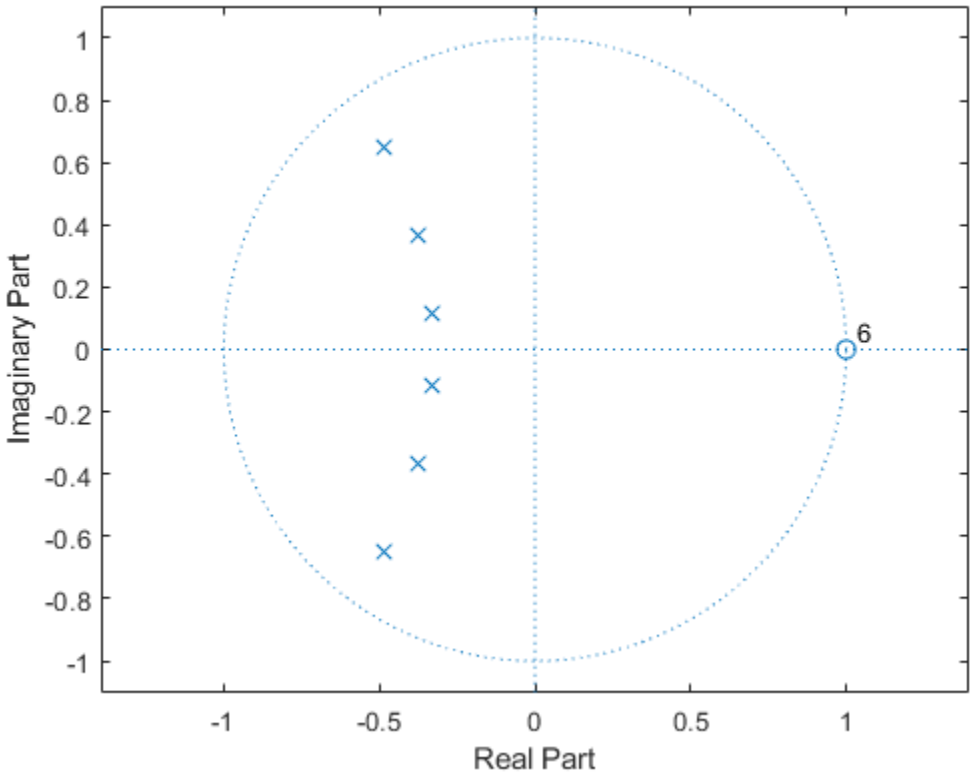
Filter Stability

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)
```

```
flag = logical
   1
```

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

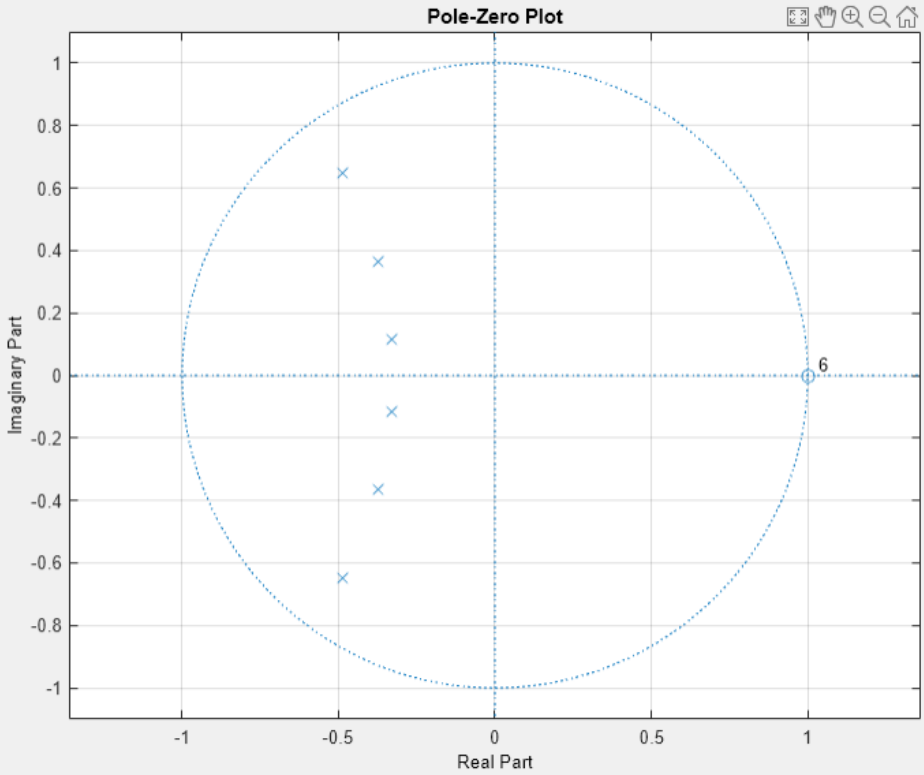


Redesign the filter using `designfilt` and check it for stability.

```
d = designfilt("highpassiir",DesignMethod="butter",FilterOrder=6, ...
               HalfPowerFrequency=0.7);
dflg = isstable(d)
```

```
dflg = logical
   1
```

```
zplane(d)
```



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_flag1 = logical
   1
```

```
act_flag2 = isstable(single(b),single(a))
```

```
act_flag2 = logical
   0
```

Input Arguments

b, a — Filter coefficients

vector

Filter coefficients, specified as vectors.

Data Types: `single` | `double`

sos — Second order sections

k-by-6 matrix

Second order sections, specified as 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 matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

Data Types: `double`

d — Digital filter
`digitalFilter` object

Digital filter, specified as a `digitalFilter` object.

Output Arguments

`flag` — Logical output
`1` (`true`) | `0` (`false`)

Logical output, returned as `1` or `0`. If the poles lie on or outside the circle, the function returns `0`. If the poles are inside the circle, the function returns `1`.

Data Types: `logical`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the first input to `isstable` is a variable-sized matrix during code generation, then it must not reduce to a vector at runtime.

See Also
`designfilt` | `digitalFilter` | `isallpass` | `islinphase` | `ismaxphase` | `isminphase` | `zplane`

Introduced in R2013a

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 = 1×5

    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`

Introduced before R2006a

istft

Inverse short-time Fourier transform

Syntax

```
x = istft(s)
x = istft(s,fs)
x = istft(s,ts)

x = istft(___,Name,Value)

[x,t] = istft(___)
```

Description

`x = istft(s)` returns the "Inverse Short-Time Fourier Transform" on page 1-1165 (ISTFT) of `s`.

`x = istft(s,fs)` returns the ISTFT of `s` using sample rate `fs`.

`x = istft(s,ts)` returns the ISTFT using sample time `ts`.

`x = istft(___,Name,Value)` specifies additional options using name-value pair arguments. Options include the FFT window length and number of overlapped samples. These arguments can be added to any of the previous input syntaxes.

`[x,t] = istft(___)` returns the signal times at which the ISTFT is evaluated.

Examples

ISTFT of Multichannel Signals

Generate a three-channel signal consisting of three different chirps sampled at 1 kHz for 1 second.

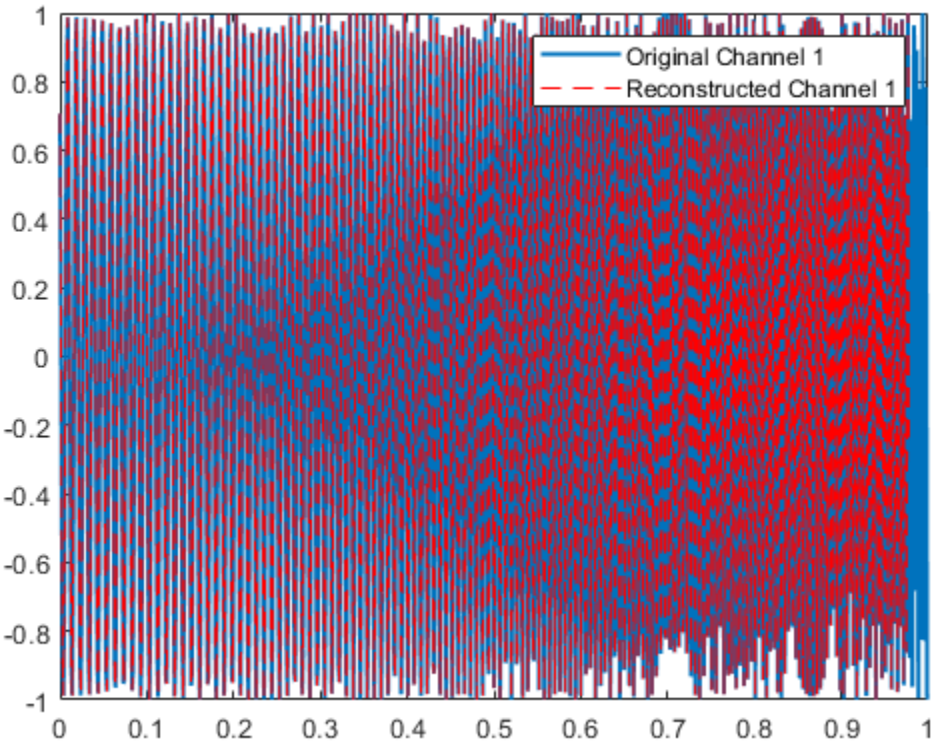
1. The first channel consists of a concave quadratic chirp with instantaneous frequency 100 Hz at $t = 0$ and crosses 300 Hz at $t = 1$ second. It has an initial phase equal to 45 degrees.
2. The second channel consists of a convex quadratic chirp with instantaneous frequency 200 Hz at $t = 0$ and crosses 600 Hz at $t = 1$ second.
3. The third channel consists of a logarithmic chirp with instantaneous frequency 300 Hz at $t = 0$ and crosses 500 Hz at $t = 1$ second.

Compute the STFT of the multichannel signal using a periodic Hamming window of length 256 and an overlap length of 15 samples.

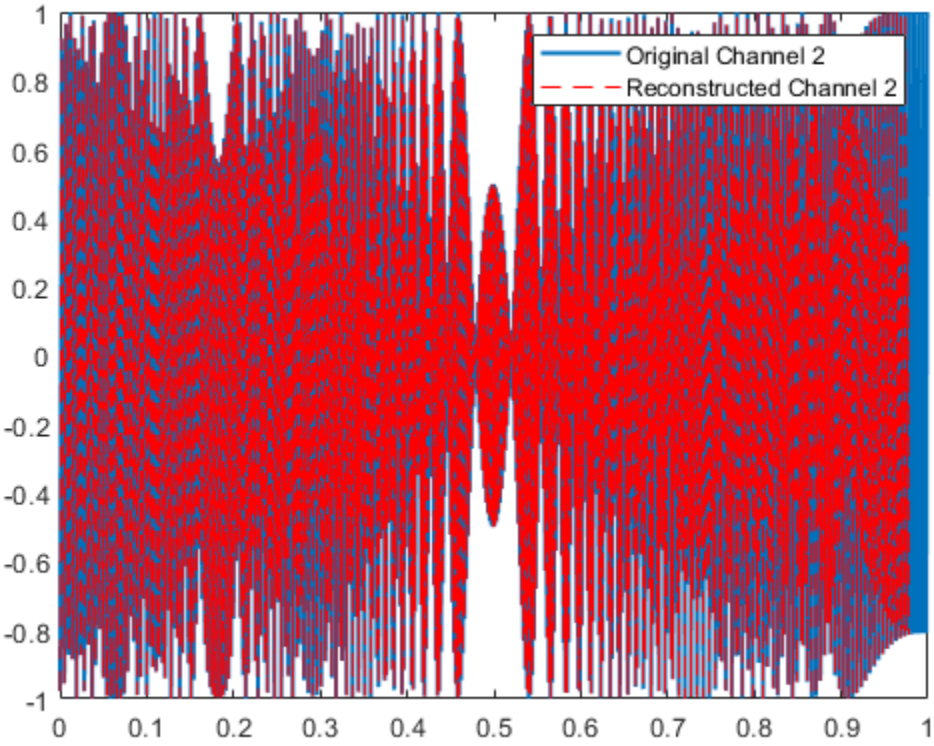
```
fs = 1e3;
t = 0:1/fs:1-1/fs;
x = [chirp(t,100,1,300,'quadratic',45,'concave');
      chirp(t,200,1,600,'quadratic',[],'convex');
      chirp(t,300,1,500,'logarithmic')]';

[S,F,T] = stft(x,fs,'Window',hamming(256,'periodic'),'OverlapLength',15);
```

Plot the original and reconstructed versions of the first and second channels.

```
[ix,ti] = istft(S,fs,'Window',hamming(256,'periodic'),'OverlapLength',15);

plot(t,x(:,1)','LineWidth',1.5)
hold on
plot(ti,ix(:,1)','r--')
hold off
legend('Original Channel 1','Reconstructed Channel 1')
```

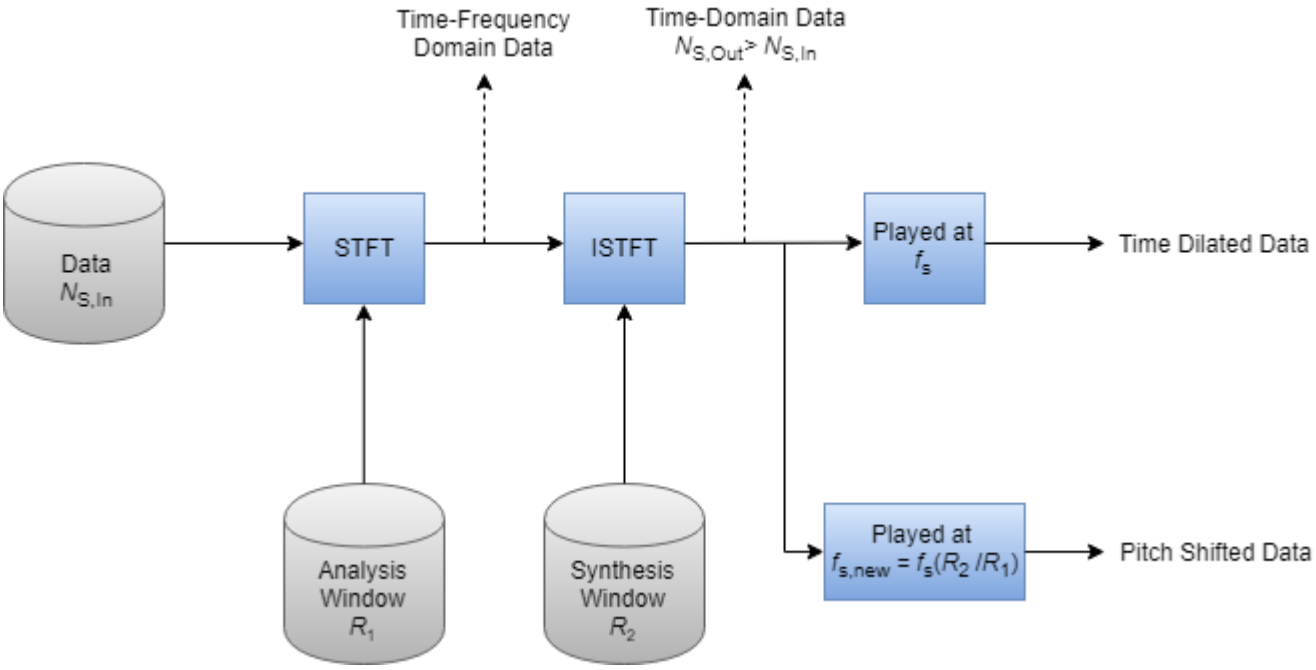



```
plot(t,x(:,2)','LineWidth',1.5)
hold on
plot(ti,ix(:,2)','r--')

legend('Original Channel 2','Reconstructed Channel 2')
```



Phase Vocoder with Different Synthesis and Analysis Windows

The phase vocoder performs time stretching and pitch scaling by transforming the audio into the frequency domain. This diagram shows the operations involved in the phase vocoder implementation.



The phase vocoder takes the STFT of a signal with an analysis window of hop size R_1 and then performs an ISTFT with a synthesis window of hop size R_2. The vocoder thus takes advantage of the WOLA method. To time stretch a signal, the analysis window uses a larger number of overlap samples than the synthesis. As a result, there are more samples at the output than at the input ($N_{S,\,\mathrm{Out}} > N_{S,\,\mathrm{In}}$), although the frequency content remains the same. Now, you can pitch scale this signal by playing it back at a higher sample rate, which produces a signal with the original duration but a higher pitch.

Load an audio file containing a fragment of Handel's "Hallelujah Chorus" sampled at 8192 Hz.

```
load handel
```

Design a root-Hann window of length 512. Set analysis overlap length as 192 and synthesis overlap length as 166.

```
wlen = 512;
win = sqrt(hann(wlen,'periodic'));
noverlapA = 192;
noverlapS = 166;
```

Implement the phase vocoder by using an analysis window of overlap 192 and a synthesis window of overlap 166.

```
S = stft(y,Fs,'Window',win,'OverlapLength',noverlapA);
iy = istft(S,Fs,'Window',win,'OverlapLength',noverlapS);


%To hear, type soundsc(w,Fs), pause(10), soundsc(iw,Fs);
```

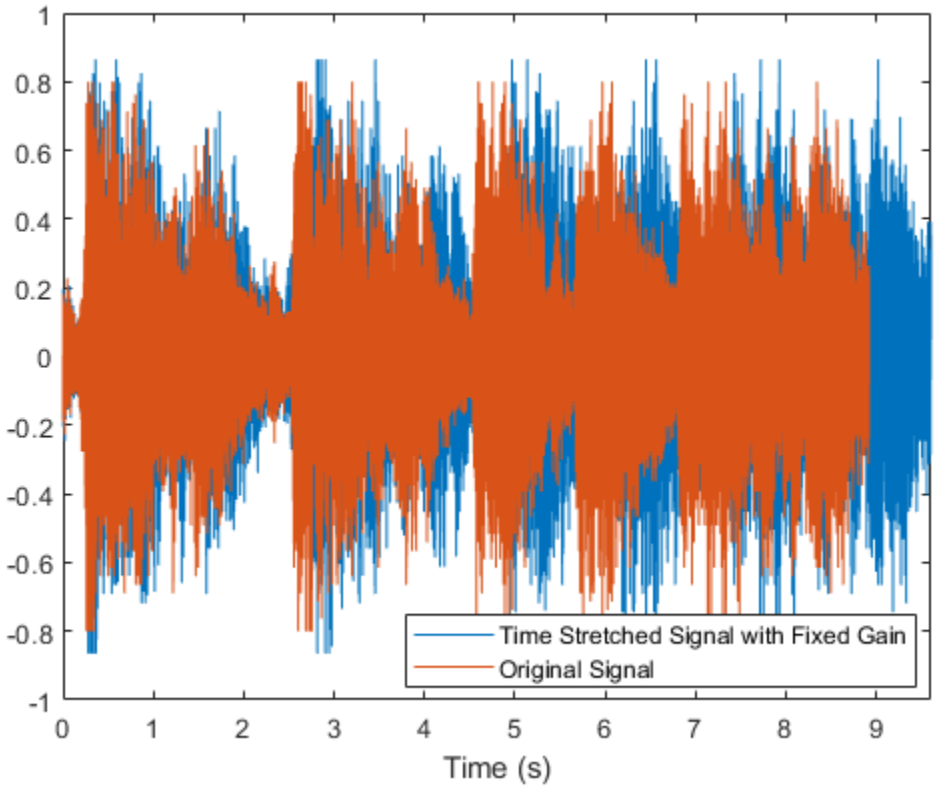
If the analysis and synthesis windows are the same but the overlap length is changed, there will be an additional gain/loss that you will need to adjust. This is a common approach to implementing a phase vocoder.

Calculate the hop ratio and use it to adjust the gain of the reconstructed signal. Also calculate frequency of pitch-shifted data using the hop ratio.

```
hopRatio = (wlen-noverlapS)/(wlen-noverlapA);
iyg = iy*hopRatio;
Fp = Fs*hopRatio;


%To hear, type soundsc(iwg,Fs), pause(15), soundsc(iwg,Fp);
```

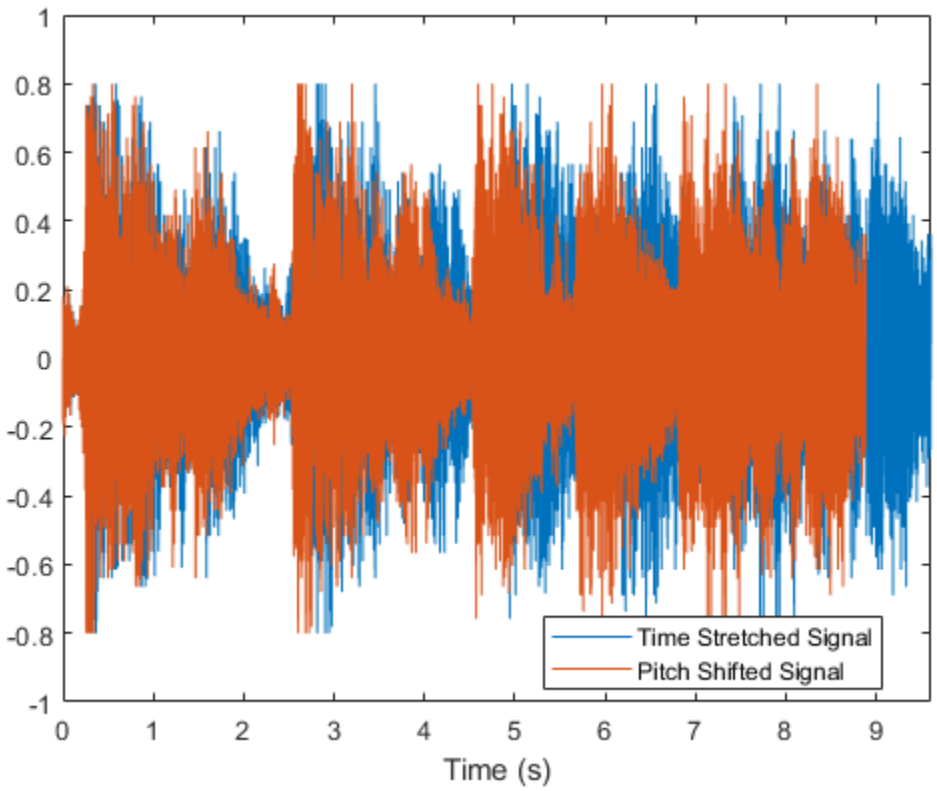
Plot the original signal and the time stretched signal with fixed gain.

```
plot((0:length(iyg)-1)/Fs,iyg,(0:length(y)-1)/Fs,y)
xlabel('Time (s)')
xlim([0 (length(iyg)-1)/Fs])
legend('Time Stretched Signal with Fixed Gain','Original Signal','Location','best')
```



Compare the time-stretched signal and the pitch shifted signal on the same plot.

```
plot((0:length(iy)-1)/Fs,iy,(0:length(iy)-1)/Fp,iy)
xlabel('Time (s)')
xlim([0 (length(iyg)-1)/Fs])
legend('Time Stretched Signal','Pitch Shifted Signal','Location','best')
```



To better understand the effect of pitch shifting data, consider the following sinusoid of frequency `Fs` over 2 seconds.

```
t =  0:1/Fs:2;
x = sin(2*pi*10*t);
```

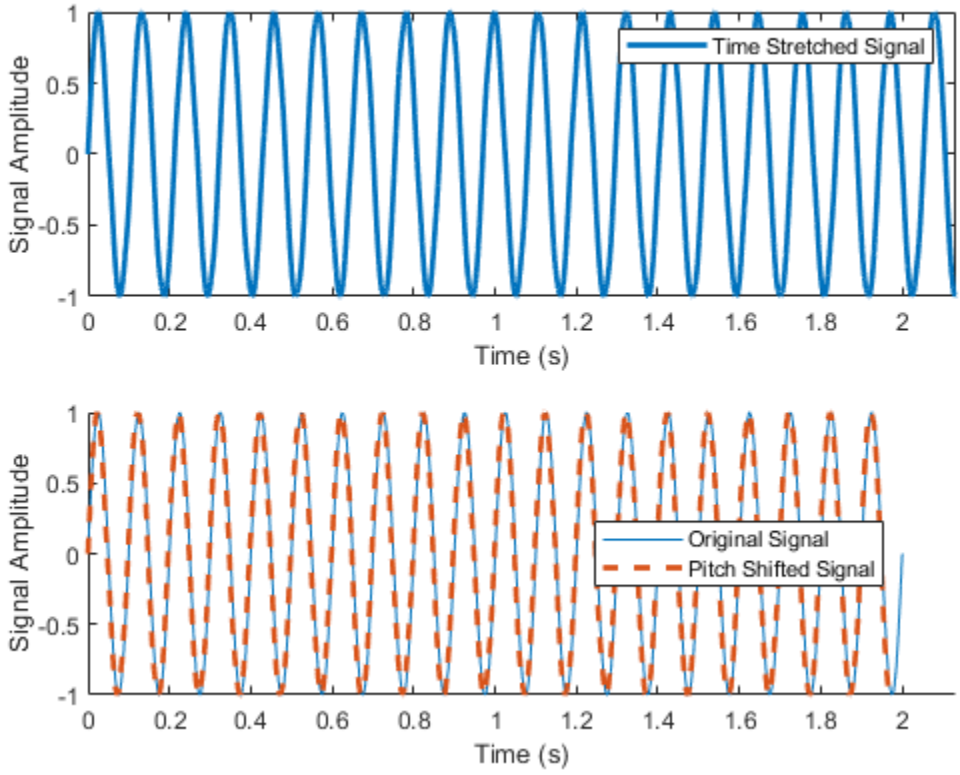
Calculate the short-time Fourier transform and the inverse short-time Fourier transform with overlap lengths 192 and 166 respectively.

```
Sx = stft(x,Fs,'Window',win,'OverlapLength',noverlapA);
ix = istft(Sx,Fs,'Window',win,'OverlapLength',noverlapS);
```

Plot the original signal on one plot and the time-stretched and pitch shifted signal on another.

```
subplot(2,1,1)
plot((0:length(ix)-1)/Fs,ix,'LineWidth',2)
xlabel('Time (s)')
ylabel('Signal Amplitude')
xlim([0 (length(ix)-1)/Fs])
legend('Time Stretched Signal')



subplot(2,1,2)
hold on
plot((0:length(x)-1)/Fs,x)
plot((0:length(ix)-1)/Fp,ix,'--','LineWidth',2)
legend('Original Signal','Pitch Shifted Signal','Location','best')
```

```
hold off
xlabel('Time (s)')
ylabel('Signal Amplitude')
xlim([0 (length(ix)-1)/Fs])
```



ISTFT of Zero-Padded Complex Signal

Generate a complex sinusoid of frequency 1 kHz and duration 2 seconds.

```
fs = 1e3;
ts = 0:1/fs:2-1/fs;

x = exp(2j*pi*100*cos(2*pi*2*ts));
```

Design a periodic Hann window of length 100 and set the number of overlap samples to 75. Check the window and overlap length for COLA compliance.

```
nwin = 100;
win = hann(nwin,'periodic');
noverlap = 75;

tf = iscola(win,noverlap)
```

```
tf = logical
   1
```

Zero-pad the signal to remove edge-effects. To avoid truncation, pad the input signal with zeros such that

$$\frac{\text{length(xZero)} - \text{noverlap}}{\text{nwin} - \text{noverlap}}$$

is an integer. Set the FFT length to 128. Compute the short-time Fourier transform of the complex signal.

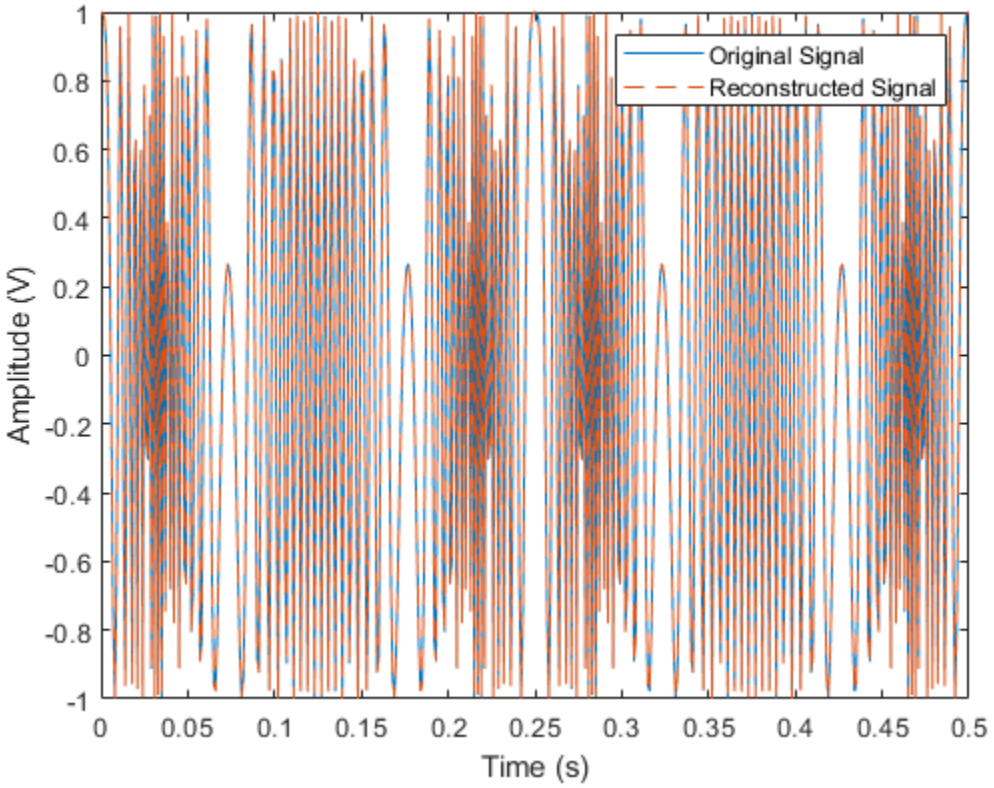
```
nPad = 100;
xZero = [zeros(1,nPad) x zeros(1,nPad)];
fftlen = 128;
s = stft(xZero,fs,'Window',win,'OverlapLength',noverlap,'FFTLength',fftlen);
```

Calculate the inverse short-time Fourier transform and remove the zeros for perfect reconstruction.

```
[is,ti] = istft(s,fs,'Window',win,'OverlapLength',noverlap,'FFTLength',fftlen);
is(1:nPad) = [];
is(end-nPad+1:end) = [];
ti = ti(1:end-2*nPad);
```

Plot the real parts of the original and reconstructed signals. The imaginary part of the signal is also reconstructed perfectly.

```
plot(ts,real(x))
hold on
plot(ti,real(is),'--')
xlim([0 0.5])
xlabel('Time (s)')
ylabel('Amplitude (V)')
legend('Original Signal','Reconstructed Signal')
hold off
```



ISTFT of Real Signal Using COLA Compliant Window and Overlap

Generate a sinusoid sampled at 2 kHz for 1 second.

```
fs = 2e3;
t = 0:1/fs:1-1/fs;
x = 5*sin(2*pi*10*t);
```

Design a periodic Hamming window of length 120. Check the COLA constraint for the window with an overlap of 80 samples. The window-overlap combination is COLA compliant.

```
win = hamming(120,'periodic');
noverlap = 80;
tf = iscola(win,noverlap)
```

```
tf = logical
   1
```

Set the FFT length to 512. Compute the short-time Fourier transform.

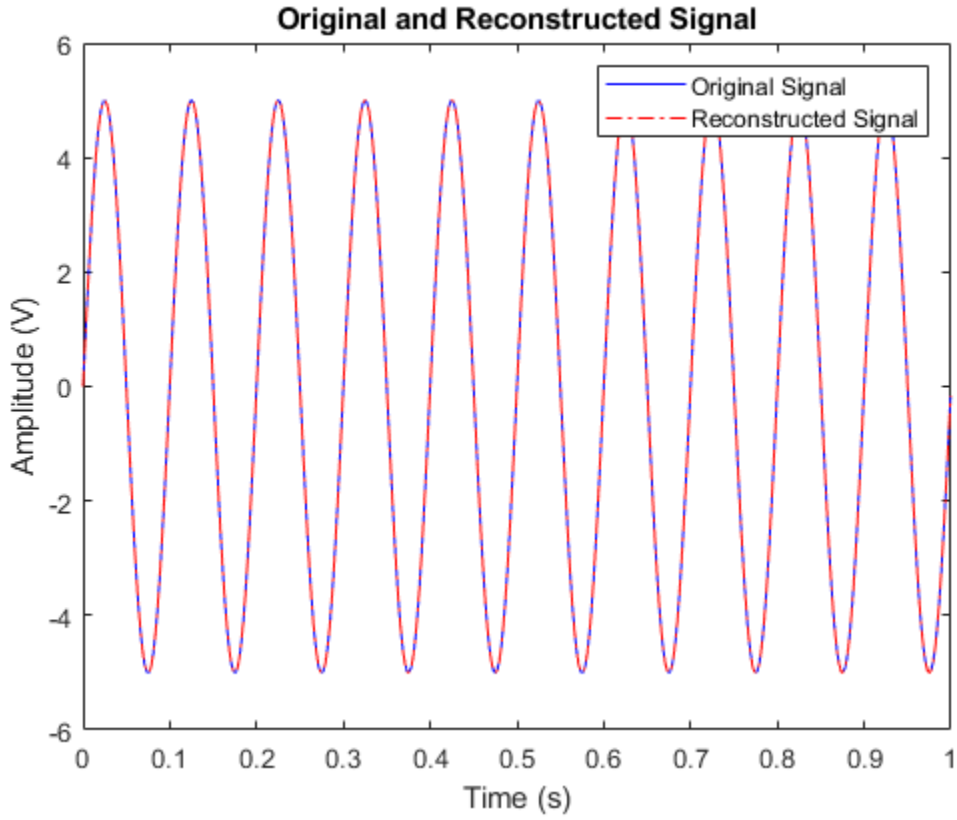
```
fftlen = 512;
s = stft(x,fs,'Window',win,'OverlapLength',noverlap,'FFTLength',fftlen);
```

Calculate the inverse short-time Fourier transform.

```
[X,T] = istft(s,fs,'Window',win,'OverlapLength',noverlap,'FFTLength',fftlen,'Method','ola','Conju
```

Plot the original and reconstructed signals.

```
plot(t,x,'b')
hold on
plot(T,X,'-.r')
xlabel('Time (s)')
ylabel('Amplitude (V)')
title('Original and Reconstructed Signal')
legend('Original Signal','Reconstructed Signal')
hold off
```



Input Arguments

s — Short-time Fourier transform

matrix | 3-D array

Short-time Fourier transform, specified as a matrix or a 3-D array. For single-channel signals, specify `s` as a matrix with time increasing across the columns and frequency increasing down the rows. For multichannel signals, specify `s` as a 3-D array with the third dimension corresponding to the channels. The frequency and time vectors are obtained as outputs of `stft`.

Note If you invert `s` using `istft` and want the result to be the same length as `x`, the value of `(length(x)-noverlap)/(length(window)-noverlap)` must be an integer.

Data Types: `double` | `single`
Complex Number Support: Yes

`fs` — Sample rate

2π (default) | positive scalar

Sample rate in hertz, specified as a positive scalar.

Data Types: `double` | `single`

`ts` — Sample time

duration scalar

Sample time, specified as a `duration` scalar.

Example: `seconds(1)` is a `duration` scalar representing a 1-second time difference between consecutive signal samples.

Data Types: `duration`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `istft(s,'Window',win,'OverlapLength',50,'FFTLength',128)` windows the data using the window `win`, with 50 samples overlap between adjoining segments and 128 DFT points.

`Window` — Windowing function

`hann(128,'periodic')` (default) | vector

Windowing function, specified as the comma-separated pair consisting of `'Window'` and a vector. If you do not specify the window or specify it as empty, the function uses a periodic Hann window of length 128. The length of `Window` must be greater than or equal to 2.

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: `double` | `single`

`OverlapLength` — Number of overlapped samples

75% of window length (default) | nonnegative integer

Number of overlapped samples, specified as the comma-separated pair consisting of `'OverlapLength'` and a positive integer smaller than the length of `window`. If you omit `'OverlapLength'` or specify it as empty, it is set to the largest integer less than 75% of the window length, which turns out to be 96 samples for the default Hann window.

Data Types: `double` | `single`

`FFTLength` — Number of DFT points

128 (default) | positive integer

Number of DFT points, specified as the comma-separated pair consisting of `'FFTLength'` and a positive integer. To achieve perfect time-domain reconstruction, you should set the number of DFT points to match that used in `stft`.

Data Types: `double` | `single`

`Method` — Method of overlap-add
`'wola'` (default) | `'ola'`

Method of overlap-add, specified as the comma-separated pair consisting of `'Method'` and one of these:

- `'wola'` — Weighted overlap-add
- `'ola'` — Overlap-add

`ConjugateSymmetric` — Conjugate symmetry of original signal
`false` (default) | `true`

Conjugate symmetry of the original signal, specified as the comma-separated pair consisting of `'ConjugateSymmetric'` and `true` or `false`. If this option is set to `true`, `istft` assumes that the input `s` is symmetric, otherwise no symmetric assumption is made. When `s` is not exactly conjugate symmetric due to round-off error, setting the name-value pair to `true` ensures that the STFT is treated as if it were conjugate symmetric. If `s` is conjugate symmetric, then the inverse transform computation is faster, and the output is real.

`FrequencyRange` — STFT frequency range
`'centered'` (default) | `'twosided'` | `'onesided'`

STFT frequency range, specified as the comma-separated pair consisting of `'FrequencyRange'` and `'centered'`, `'twosided'`, or `'onesided'`.

- `'centered'` — Treat `s` as a two-sided, centered STFT. If `nfft` is even, then `s` is considered to be computed over the interval $(-\pi, \pi]$ rad/sample. If `nfft` is odd, then `s` is considered to be computed over the interval $(-\pi, \pi)$ rad/sample. If you specify time information, then the intervals are $(-f_s, f_s/2]$ cycles/unit time and $(-f_s, f_s/2)$ cycles/unit time, respectively, where f_s is the sample rate.
- `'twosided'` — Treat `s` as a two-sided STFT computed over the interval $[0, 2\pi)$ rad/sample. If you specify time information, then the interval is $[0, f_s)$ cycles/unit time.
- `'onesided'` — Treat `s` as a one-sided STFT. If `nfft` is even, then `s` is considered to be computed over the interval $[0, \pi]$ rad/sample. If `nfft` is odd, then `s` is considered to be computed over the interval $[0, \pi)$ rad/sample. If you specify time information, then the intervals are $[0, f_s/2]$ cycles/unit time and $[0, f_s/2)$ cycles/unit time, respectively, where f_s is the sample rate.

  Note When this argument is set to `'onesided'`, `istft` assumes the values in the positive Nyquist range were computed without conserving the total power.

For an example, see “STFT Frequency Ranges” on page 1-2554.

Data Types: `char` | `string`

`InputTimeDimension` — Input time dimension
`'acrosscolumns'` (default) | `'downrows'`

Input time dimension, specified as the comma-separated pair consisting of `'InputTimeDimension'` and `'acrosscolumns'` or `'downrows'`. If this value is set to `'downrows'`, `istft` assumes that the

time dimension of `s` is down the rows and the frequency is across the columns. If this value is set to `'acrosscolumns'`, the function `istft` assumes that the time dimension of `s` is across the columns and frequency dimension is down the rows.

Output Arguments

x — Reconstructed signal

vector | matrix

Reconstructed signal in the time domain, returned as a vector or a matrix.

Data Types: `single` | `double`

t — Time instants

vector

Time instants, returned as a vector.

- If a sample rate `fs` is provided, then `t` contains time values in seconds.
- If a duration `ts` is provided, then `t` has the same time format as the input duration and is a duration array.
- If no time information is provided, then `t` contains sample numbers.

Data Types: `double` | `single`

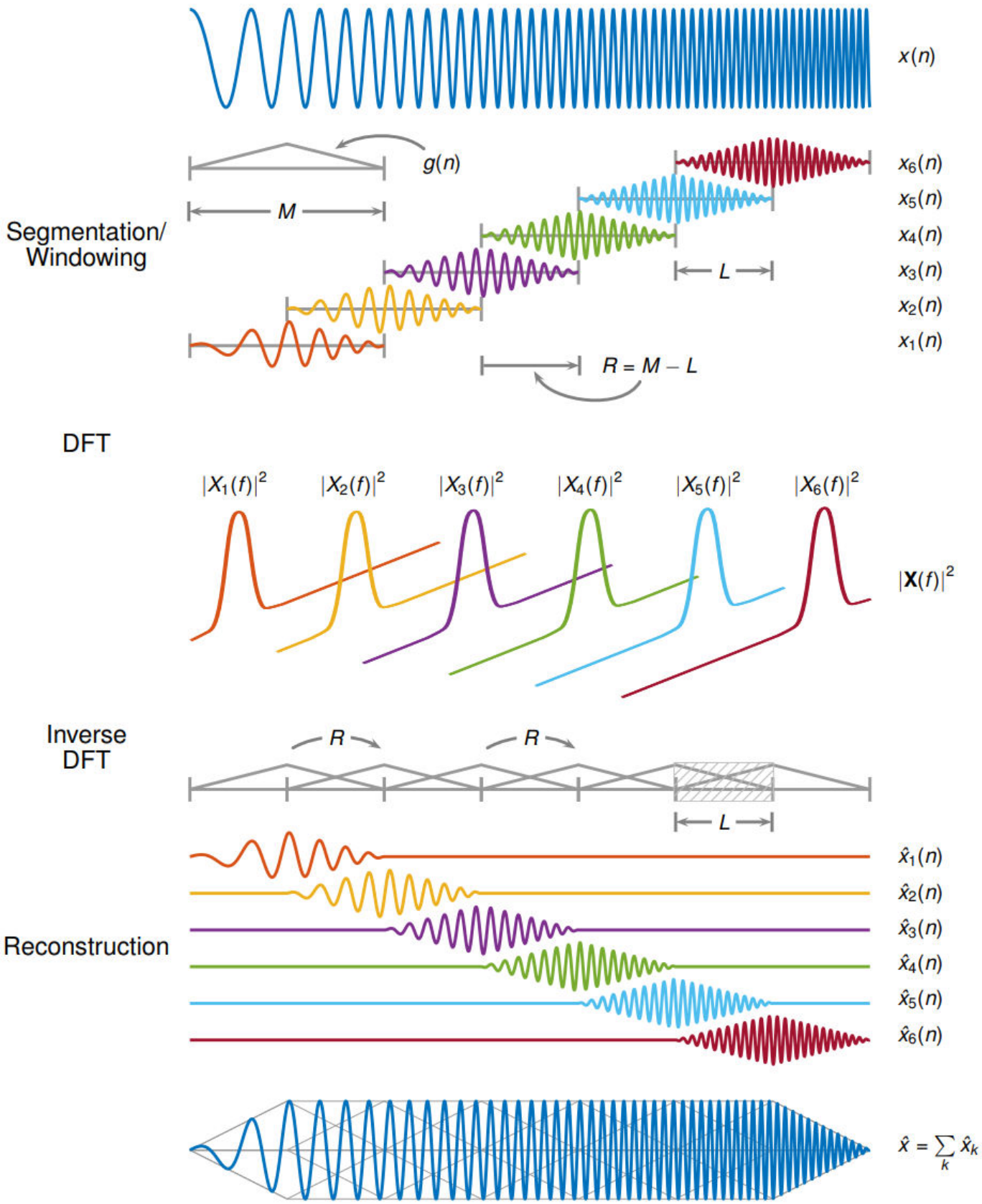
More About

Inverse Short-Time Fourier Transform

The inverse short-time Fourier transform is computed by taking the IFFT of each DFT vector of the STFT and overlap-adding the inverted signals. The ISTFT is calculated as follows:

$$\begin{aligned} x(n) &= \int_{-1/2}^{1/2} \sum_{m=-\infty}^{\infty} X_m(f) e^{j2\pi f n} df \\ &= \sum_{m=-\infty}^{\infty} \int_{-1/2}^{1/2} X_m(f) e^{j2\pi f n} df \\ &= \sum_{m=-\infty}^{\infty} x_m(n) \end{aligned}$$

where R is the hop size between successive DFTs, X_m is the DFT of the windowed data centered about time mR and $x_m(n) = x(n)\ g(n - mR)$. The inverse STFT is a perfect reconstruction of the original signal as long as $\sum_{m=-\infty}^{\infty} g^{a+1}(n - mR) = c\ \forall n \in \mathbb{Z}$ where the analysis window $g(n)$ was used to window the original signal and c is a constant. The following figure depicts the steps followed in reconstructing the original signal.


x(n)
g(n)
M
L
R = M − L
Segmentation/
Windowing
x6(n)
x5(n)
x4(n)
x3(n)
x2(n)
x1(n)
DFT
|X1(f)|2
|X2(f)|2
|X3(f)|2
|X4(f)|2
|X5(f)|2
|X6(f)|2
|X(f)|2
Inverse
DFT
R
R
L
Reconstruction
x̂1(n)
x̂2(n)
x̂3(n)
x̂4(n)
x̂5(n)
x̂6(n)
x̂ = ∑k x̂k

Constant Overlap-Add (COLA) Constraint

To ensure successful reconstruction of nonmodified spectra, the analysis window must satisfy the COLA constraint. In general, if the analysis window satisfies the condition $\sum_{m=-\infty}^{\infty} g^{a+1}(n-mR) = c \ \forall n \in \mathbb{Z}$, the window is considered to be COLA-compliant. Additionally, COLA compliance can be described as either weak or strong.

- Weak COLA compliance implies that the Fourier transform of the analysis window has zeros at frame-rate harmonics such that

  $$G(f_k) = 0, \qquad k = 1, 2, \ldots, R-1, \qquad f_k \triangleq \frac{k}{R}.$$

  Alias cancellation is disturbed by spectral modifications. Weak COLA relies on alias cancellation in the frequency domain. Therefore, perfect reconstruction is possible using weakly COLA-compliant windows as long as the signal has not undergone any spectral modifications.
- For strong COLA compliance, the Fourier transform of the window must be bandlimited consistently with downsampling by the frame rate such that

  $$G(f) = 0, \qquad f \geq \frac{1}{2R}.$$

  This equation shows that no aliasing is allowed by the strong COLA constraint. Additionally, for strong COLA compliance, the value of the constant c must equal 1. In general, if the short-time spectrum is modified in any way, a stronger COLA compliant window is preferred.

You can use the `iscola` function to check for weak COLA compliance. The number of summations used to check COLA compliance is dictated by the window length and hop size. In general, it is common to use $a = 1$ in $\sum_{m=-\infty}^{\infty} g^{a+1}(n-mR) = c \ \forall n \in \mathbb{Z}$ for weighted overlap-add (WOLA), and $a = 0$ for overlap-add (OLA). By default, `istft` uses the WOLA method, by applying a synthesis window before performing the overlap-add method.

In general, the synthesis window is the same as the analysis window. You can construct useful WOLA windows by taking the square root of a strong OLA window. You can use this method for all nonnegative OLA windows. For example, the root-Hann window is a good example of a WOLA window.

Perfect Reconstruction

In general, computing the STFT of an input signal and inverting it does not result in perfect reconstruction. If you want the output of ISTFT to match the original input signal as closely as possible, the signal and the window must satisfy the following conditions:

- Input size — If you invert the output of `stft` using `istft` and want the result to be the same length as the input signal `x`, the value of $k = \frac{(length(x) - noverlap)}{(length(window) - noverlap)}$ must be an integer.
- COLA compliance — Use COLA-compliant windows, assuming that you have not modified the short-time Fourier transform of the signal.
- Padding — If the length of the input signal is such that the value of k is not an integer, zero-pad the signal before computing the short-time Fourier transform. Remove the extra zeros after inverting the signal.

References

[1] Crochiere, R. E. "A Weighted Overlap-Add Method of Short-Time Fourier Analysis/Synthesis." *IEEE Transactions on Acoustics, Speech and Signal Processing.* Vol. 28, Number 1, Feb. 1980, pp. 99–102.

[2] Gotzen, A. D., N. Bernardini, and D. Arfib. "Traditional Implementations of a Phase-Vocoder: The Tricks of the Trade." *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00)*, Verona, Italy, Dec 7–9, 2000.

[3] Griffin, Daniel W., and Jae S. Lim. "Signal Estimation from Modified Short-Time Fourier Transform." *IEEE Transactions on Acoustics, Speech and Signal Processing.* Vol. 32, Number 2, April 1984, pp. 236–243.

[4] Portnoff, M. R. "Time-Frequency Representation of Digital Signals and Systems Based on Short-Time Fourier analysis." *IEEE Transactions on Acoustics, Speech and Signal Processing.* Vol. 28, Number 1, Feb 1980, pp. 55–69.

[5] Smith, Julius Orion. *Spectral Audio Signal Processing*. https://ccrma.stanford.edu/~jos/sasp/, online book, 2011 edition, accessed Nov. 2018.

[6] Sharpe, Bruce. *Invertibility of Overlap-Add Processing*. https://gauss256.github.io/blog/cola.html, accessed July 2019.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

`'InputTimeDimension'` must be always specified and set to `'downrows'`.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The `'ConjugateSymmetric'` argument is not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

The `'ConjugateSymmetric'` argument is not supported for code generation.

Thread-Based Environment

Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

This function fully supports thread-based environments. For more information, see “Run MATLAB Functions in Thread-Based Environment”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

Unless `'ConjugateSymmetric'` is set to `true`, the output `x` is always complex even if all the imaginary parts are zero.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Functions

`iscola` | `pspectrum` | `stft` | `stftmag2sig`

Topics

“Time-Frequency Gallery”

Introduced in R2019a

kaiser

Kaiser window

Syntax

```
w = kaiser(L,beta)
```

Description

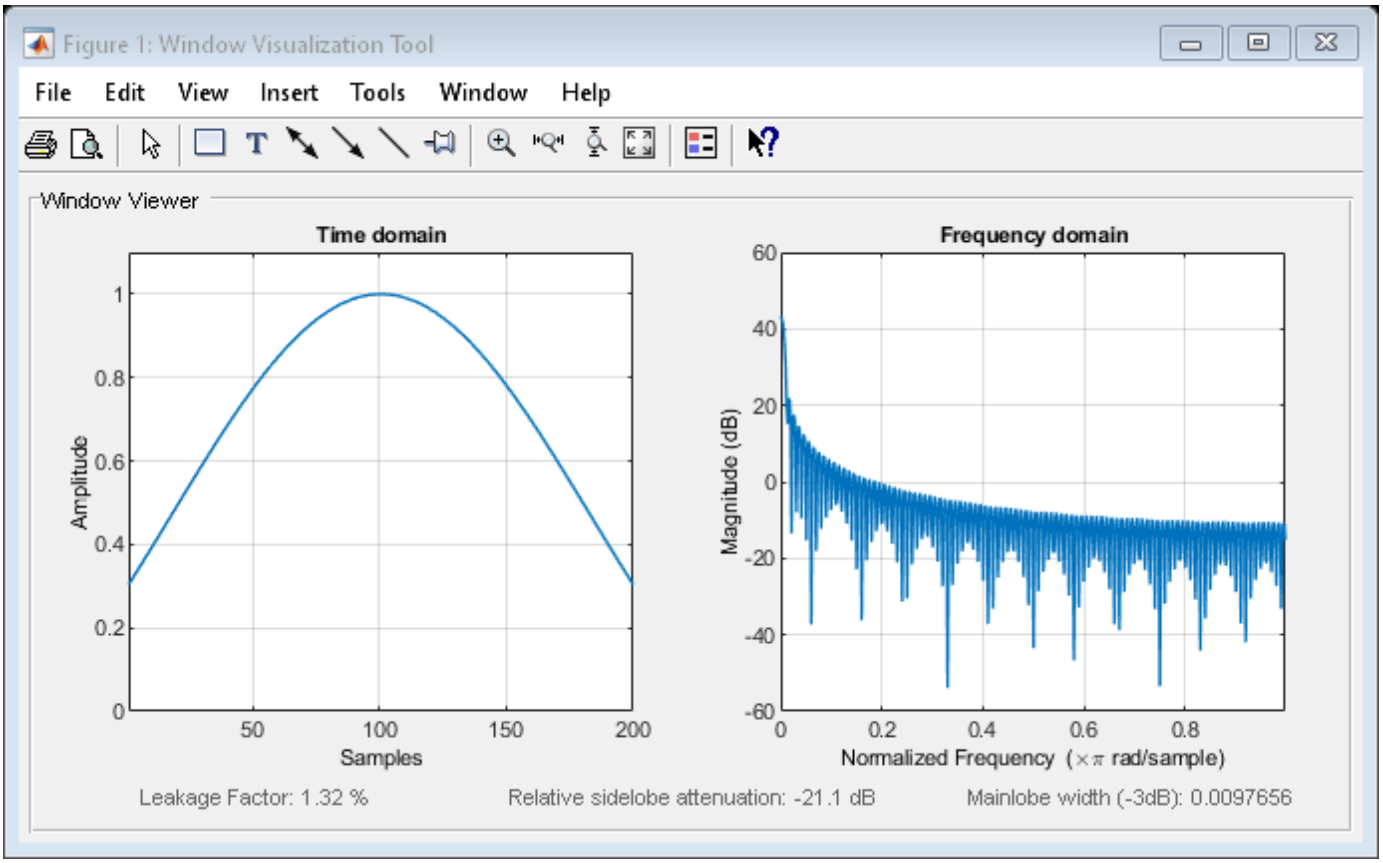
`w = kaiser(L,beta)` returns an L-point Kaiser window with shape factor `beta`.

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)
```



Input Arguments

`L` — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

`beta` — Shape factor

`0.5` (default) | positive real scalar

Shape factor, specified as a positive real scalar. The parameter `beta` affects the sidelobe attenuation of the Fourier transform of the window.

Data Types: `single` | `double`

Output Arguments

`w` — Kaiser window

column vector

Kaiser window, returned as a column vector.

Algorithms

The coefficients of a Kaiser window are computed from the following equation:

$$w(n) = \frac{I_0\left(\beta\sqrt{1-\left(\frac{n-N/2}{N/2}\right)^2}\right)}{I_0(\beta)}, \quad 0 \le n \le N,$$

where I_0 is the zeroth-order modified Bessel function of the first kind. The length $L = N + 1$. `kaiser(L,beta)` is equivalent to

```
besseli(0,beta*sqrt(1-(((0:L-1)-(L-1)/2)/((L-1)/2)).^2))/besseli(0,beta)
```

To obtain a Kaiser window that represents an FIR filter with sidelobe attenuation of α dB, use the following β.

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \ge \alpha \ge 21 \\ 0, & \alpha < 21 \end{cases}$$

Increasing β widens the mainlobe and decreases the amplitude of the sidelobes (i.e., increases the attenuation).

References

[1] 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.

[2] 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.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps

Window Designer | **Signal Analyzer**

Functions

`chebwin` | `gausswin` | `kaiserord` | `tukeywin` | **WVTool** | `pspectrum`

Introduced before R2006a

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

`[n,Wn,beta,ftype] = kaiserord(f,a,dev)` returns a filter order `n`, normalized frequency band edges `Wn`, and a shape factor `beta` that specify a Kaiser window for use with the `fir1` function. To design an FIR filter `b` that approximately meets the specifications given by `f`, `a`, and `dev`, use `b = fir1(n,Wn,kaiser(n+1,beta),ftype,'noscale')`.

`[n,Wn,beta,ftype] = kaiserord(f,a,dev,fs)` uses a sample rate `fs` in Hz.

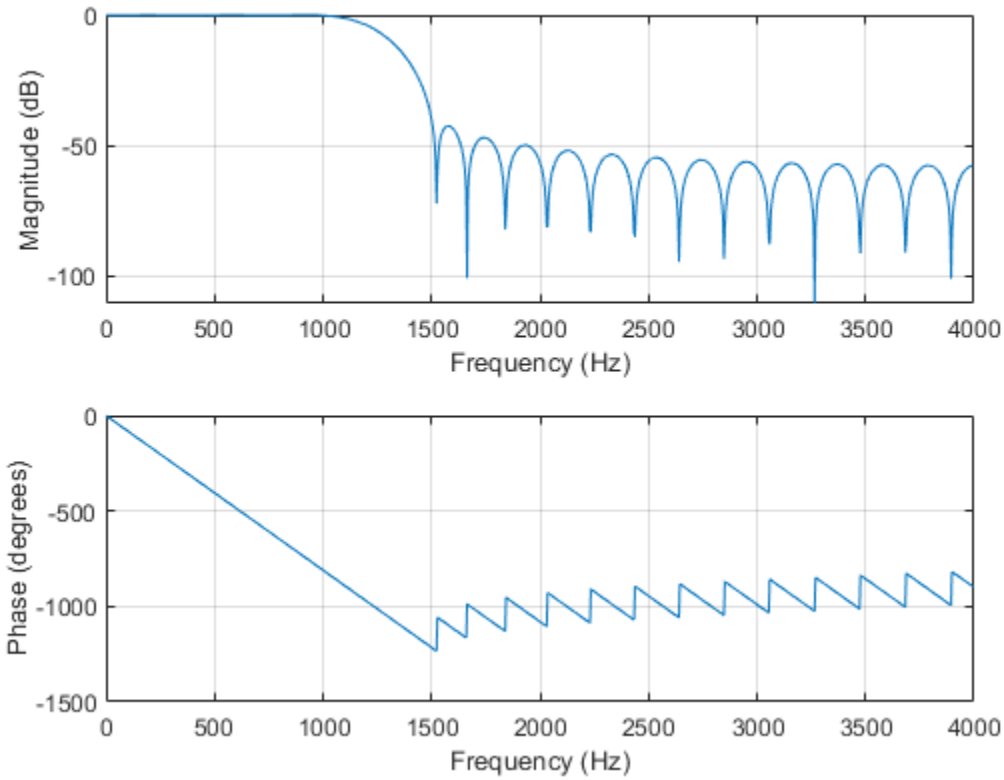
`c = kaiserord(f,a,dev,fs,'cell')` returns a cell array whose elements are the parameters to `fir1`.

Examples

Kaiser Window Lowpass 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.

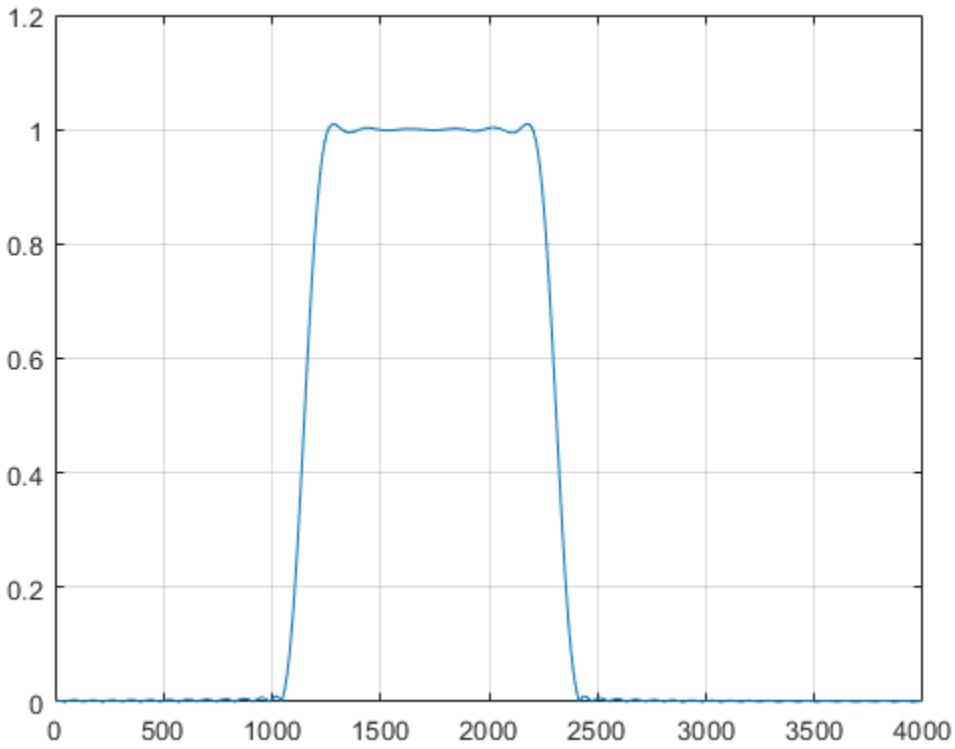
```
fsamp = 8000;
fcuts = [1000 1500];
mags = [1 0];
devs = [0.05 0.01];

[n,Wn,beta,ftype] = kaiserord(fcuts,mags,devs,fsamp);
hh = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');

freqz(hh,1,1024,fsamp)
```



Kaiser Window Bandpass Filter Design

Design an odd-length bandpass filter. Note that odd length means even order, so the input to `fir1` must be an even integer.

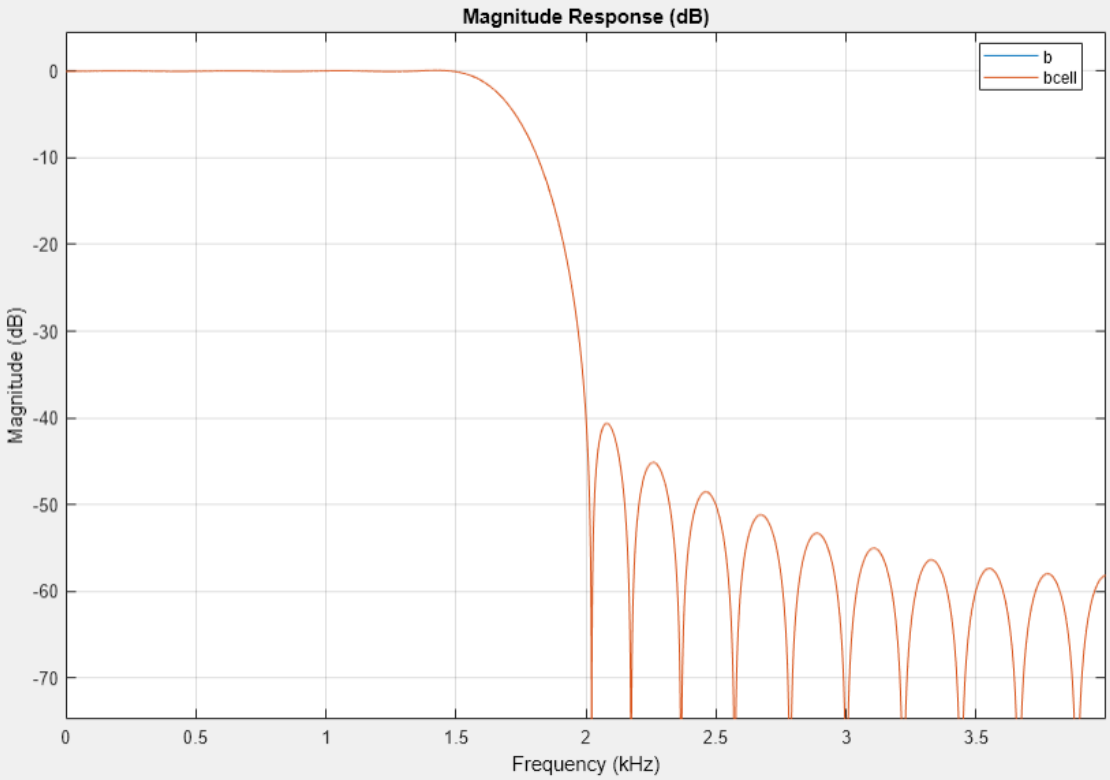
```
fsamp = 8000;
fcuts = [1000 1300 2210 2410];
mags = [0 1 0];
devs = [0.01 0.05 0.01];

[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
```



Lowpass Filter Design with `'cell'` Option

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.

```
fs = 8000;
[n,Wn,beta,ftype] = kaiserord([1500 2000],[1 0],...
    [0.01 0.1],fs);
b = fir1(n,Wn,ftype,kaiser(n+1,beta),'noscale');

c = kaiserord([1500 2000],[1 0],[0.01 0.1],fs,'cell');
bcell = fir1(c{:});

hfvt = fvtool(b,1,bcell,1,'Fs',fs);
legend(hfvt,'b','bcell')
```



Input Arguments

`f` — Band edges

vector

Band edges, specified as a vector. The length of `f` is `2*length(a)-2`.

`a` — Band amplitude

vector

Band amplitude, specified as a vector. The amplitude is specified on the bands defined by `f`. Together, `f` and `a` define a piecewise-constant response function.

`dev` — Maximum allowable deviation

positive vector

Maximum allowable deviation, specified as a vector. `dev` is a vector the same size as `a` that specifies the maximum allowable deviation between the frequency response of the output filter and its band amplitude, for each band. The entries in `dev` specify the passband ripple and the stopband attenuation. Specify each entry in `dev` as a positive number, representing absolute filter gain (unitless).

`fs` — Sample rate

2 (default) | positive scalar

Sample rate, specified as a positive scalar measured in Hz. If you do not specify the argument `fs`, or if you specify it as the empty vector `[]`, the sample rate defaults to 2 Hz, and the Nyquist frequency is 1 Hz. Use this syntax to specify band edges scaled to a particular application's sample rate. The frequency band edges in `f` must be from 0 to `fs`/2.

Output Arguments

`n` — Filter order

positive integer

Filter order, returned as a positive integer.

`Wn` — Normalized frequency band edges

real vector

Normalized frequency band edges, returned as a real vector.

`beta` — Shape factor

positive real scalar

Shape factor, returned as a positive real scalar. The parameter `beta` affects the sidelobe attenuation of the Fourier transform of the window.

`ftype` — Filter type

`'low'` | `'bandpass'` | `'high'` | `'stop'` | `'DC-0'` | `'DC-1'`

Filter type, intended for use with `fir1`, returned as:

- `'low'` — lowpass filter with cutoff frequency `Wn`.
- `'high'` — highpass filter with cutoff frequency `Wn`.
- `'bandpass'` — bandpass filter if `Wn` is a two-element vector.
- `'stop'` — bandstop filter if `Wn` is a two-element vector.
- `'DC-0'` — the first band of a multiband filter is a stopband.
- `'DC-1'` — the first band of a multiband filter is a passband.

`c` — FIR parameters

cell array

FIR parameters, returned as a cell array.

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.
- 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.

- 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%).

Algorithms

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.

`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

$$\beta = \begin{cases} 0.1102(\alpha - 8.7), & \alpha > 50 \\ 0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 21 \le \alpha \le 50 \\ 0, & \alpha < 21 \end{cases}$$

where $\alpha = -20\log_{10}\delta$ is the stopband attenuation expressed in decibels, and

$$n = \frac{\alpha - 7.95}{2.285(\Delta\omega)}$$

where n is the filter order and $\Delta\omega$ is the width of the smallest transition region.

References

[1] 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.

[2] 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.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`fir1` | `kaiser` | `firpmord`

Introduced before R2006a

kaiserwin

Kaiser window filter from specification object

Syntax

```
h = design(d,'kaiserwin')
h = design(d,'kaiserwin',Name,Value)
```

Description

`h = design(d,'kaiserwin')` designs a digital filter using 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',Name,Value)` returns a filter designed with the Kaiser window technique and with design options specified as `Name,Value` pairs.

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.

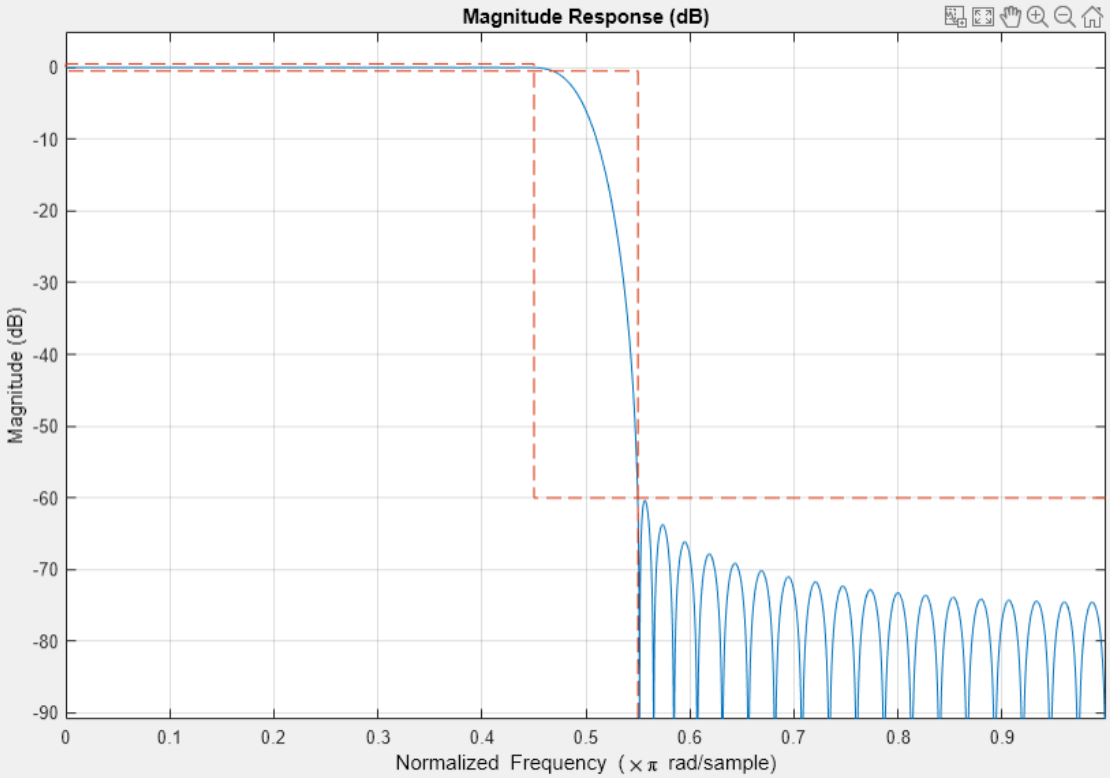
```
help(d,'kaiserwin')
```

Examples

Design Filter Using Kaiser Window

Design a direct-form FIR filter starting from the default lowpass filter specification object. Use a Kaiser window for the design. Visualize the magnitude response.

```
d = fdesign.lowpass;
Hd = design(d,'kaiserwin');

fvtool(Hd)
```



See Also

Apps
Filter Designer

Functions
`designfilt` | `fdesign`

Introduced in R2009a

kurtogram

Visualize spectral kurtosis

Syntax

```
kgram = kurtogram(x)
kgram = kurtogram(x,sampx)
kgram = kurtogram(xt)
kgram = kurtogram( ___ ,level)

[kgram,f,w,fc,wc,bw] = kurtogram( ___ )

kurtogram( ___ )
```

Description

`kgram = kurtogram(x)` returns the fast kurtogram on page 1-1190 `kgram` of signal vector `x` as a matrix. `kurtogram` uses normalized frequency (evenly spaced frequency vector spanning [0 π]) to compute the time values.

`kgram = kurtogram(x,sampx)` returns the fast kurtogram on page 1-1190 of signal vector `x` sampled at rate or time interval `sampx`, as a matrix.

`kgram = kurtogram(xt)` returns the fast kurtogram on page 1-1190 `kgram` of timetable `xt` as a matrix.

`kgram = kurtogram( ___ ,level)` returns the fast kurtogram on page 1-1190 using a specified `level`. `level` determines the level of window resolution to use, and therefore how many spectral kurtosis cases to calculate.

`[kgram,f,w,fc,wc,bw] = kurtogram( ___ )` returns the fast kurtogram on page 1-1190 along with a set of parameters you can use for follow-on bandpass filter design and spectral kurtosis:

- `f` — Frequency vector for `kgram`
- `w` — Window size vector for `kgram`
- `fc` — Frequency where the maximal spectral kurtosis is located
- `wc` — Window size where the maximal spectral kurtosis on the kurtogram is located
- `bw` — Suggested bandwidth for the optimal bandpass filter

You can use this syntax with any of the input arguments in previous syntaxes.

`kurtogram( ___ )` plots the kurtogram, along with key critical optimization parameters, without returning any data. You can use this syntax with any of the input arguments in previous syntaxes.

Examples

Compute the Kurtogram of a Nonstationary Signal

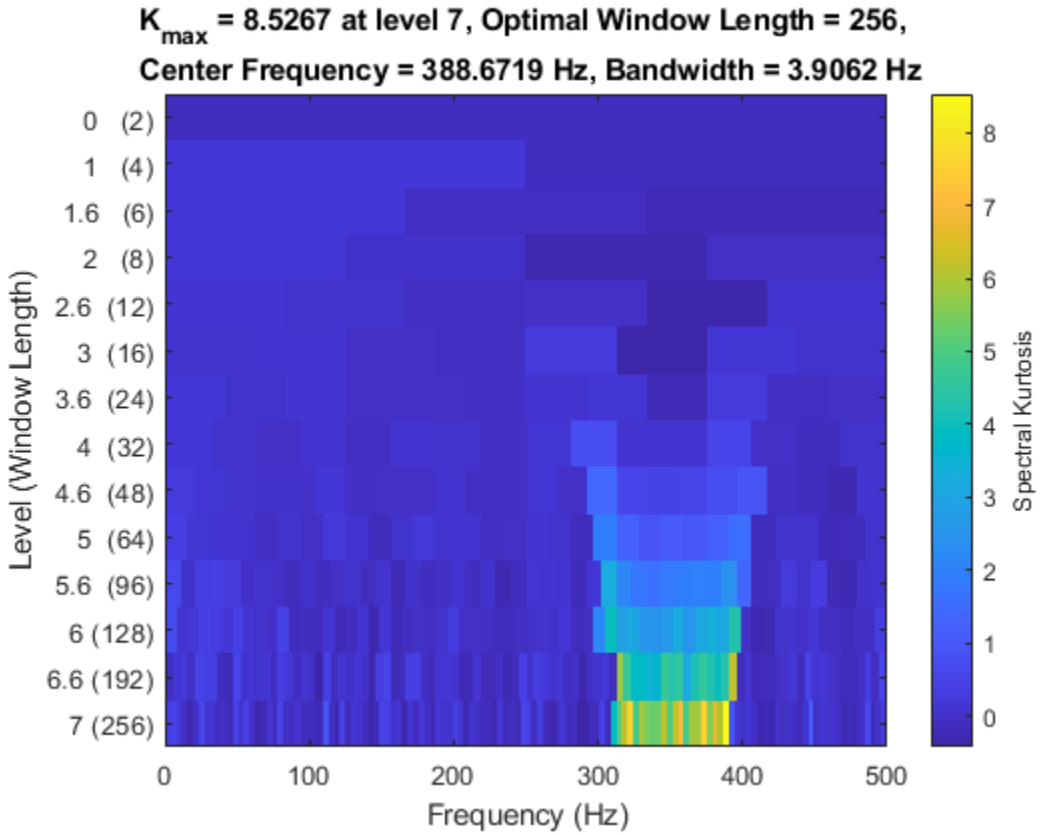
Compute the kurtogram of a nonstationary signal. Compare different level settings for the kurtogram. Examine a kurtogram that uses normalized frequency. Use the kurtogram to provide filter settings that can be used to preprocess the signal to enhance transient detection.

Generate a signal with a chirp component and white Gaussian noise.

```
fs = 1000;
t = 0:1/fs:10;
f1 = 300;
f2 = 400;
xc = chirp(t,f1,10,f2);
x = xc+randn(1,length(t));
```

Plot the kurtogram using the sample rate `fs`.

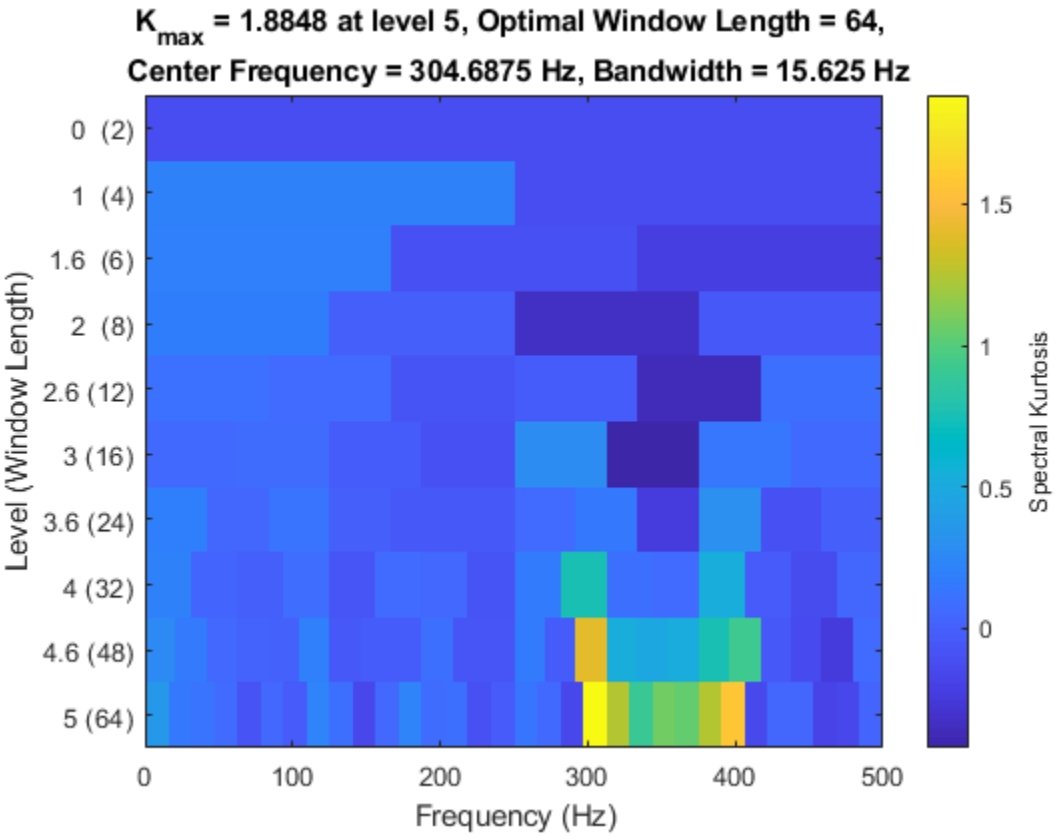
```
kurtogram(x,fs)
```



The kurtogram shows kurtosis results for a range of window lengths and frequencies. A high kurtosis level corresponds to a high level of nonstationary or non-Gaussian behavior. The peak kurtosis is provided in the text at the top, along with the window length and center frequency associated with it. The bandwidth is a function of the window length.

Explore the effects of lowering the maximum level to 5.

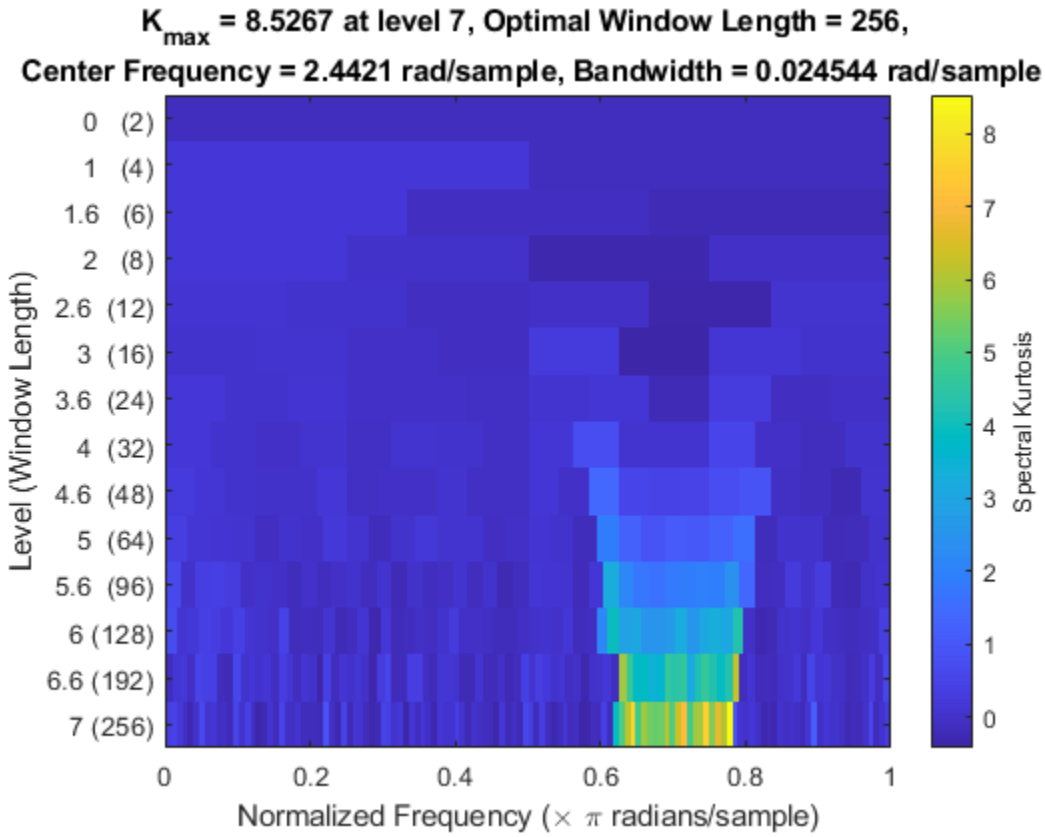
```
level = 5;
kurtogram(x,fs,level)
```



The lower resolution is apparent and leads to a lower peak kurtosis value and a displaced center frequency.

Now plot the kurtosis without specifying sample rate or time.

```
kurtogram(x)
```



The kurtogram is now shown with normalized frequency.

The parameters at the top of the plot provide recommendations for a bandpass filter that could be used to prefilter the data and enhance the differentiation of the nonstationary component. You can also have `kurtogram` return these values so they can be input more directly into filtering or spectral kurtosis functions.

```
[kgram,f,w,fc,wc,bw] = kurtogram(x);
wc
```

```
wc = 256
```

```
fc
```

```
fc = 2.4421
```

```
bw
```

```
bw = 0.0245
```

These values match the optimal window size, center frequency, and bandwidth of the first plot. `kgram` is the actual kurtogram matrix, and `f` and `w` are the frequency and window-size vectors that accompany it.

Plot Spectral Kurtosis Using a Customized Window Size

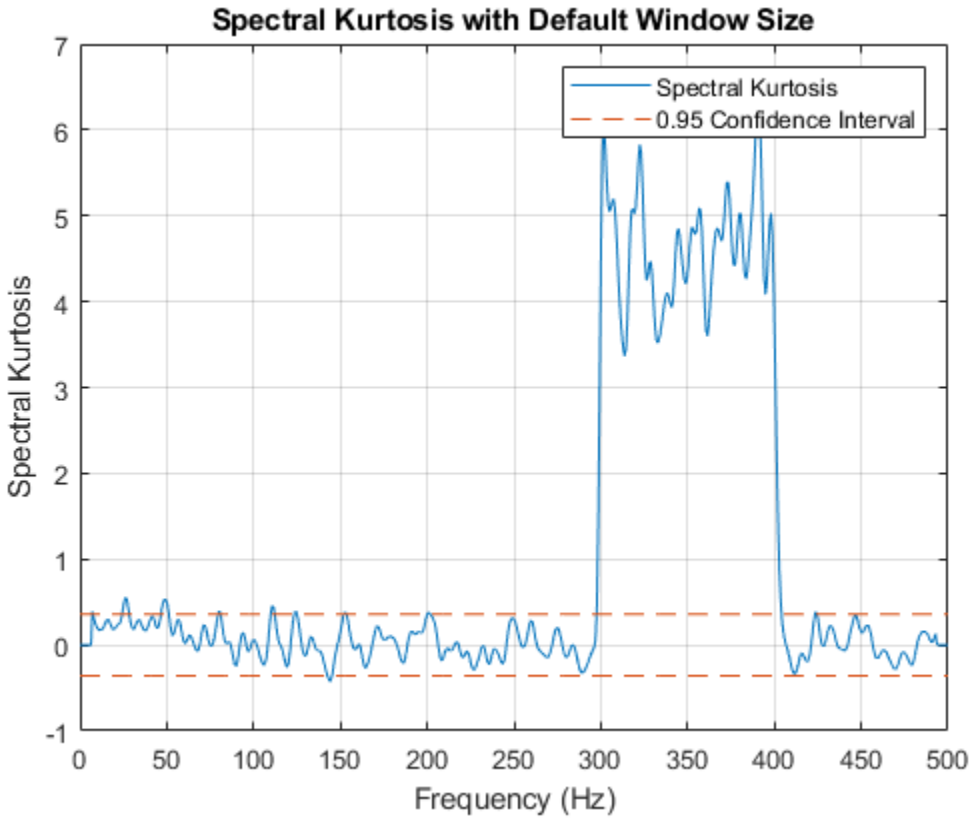
The `pkurtosis` function uses the default `pspectrum` window size (time resolution). You can specify the window size to use instead. In this example, use the function `kurtogram` to return an optimal window size and use that result for `pkurtosis`.

Create a chirp signal with white Gaussian noise.

```
fs = 1000;
t = 0:1/fs:10;
f1 = 300;
f2 = 400;
x = chirp(t,f1,10,f2)+randn(1,length(t));
```

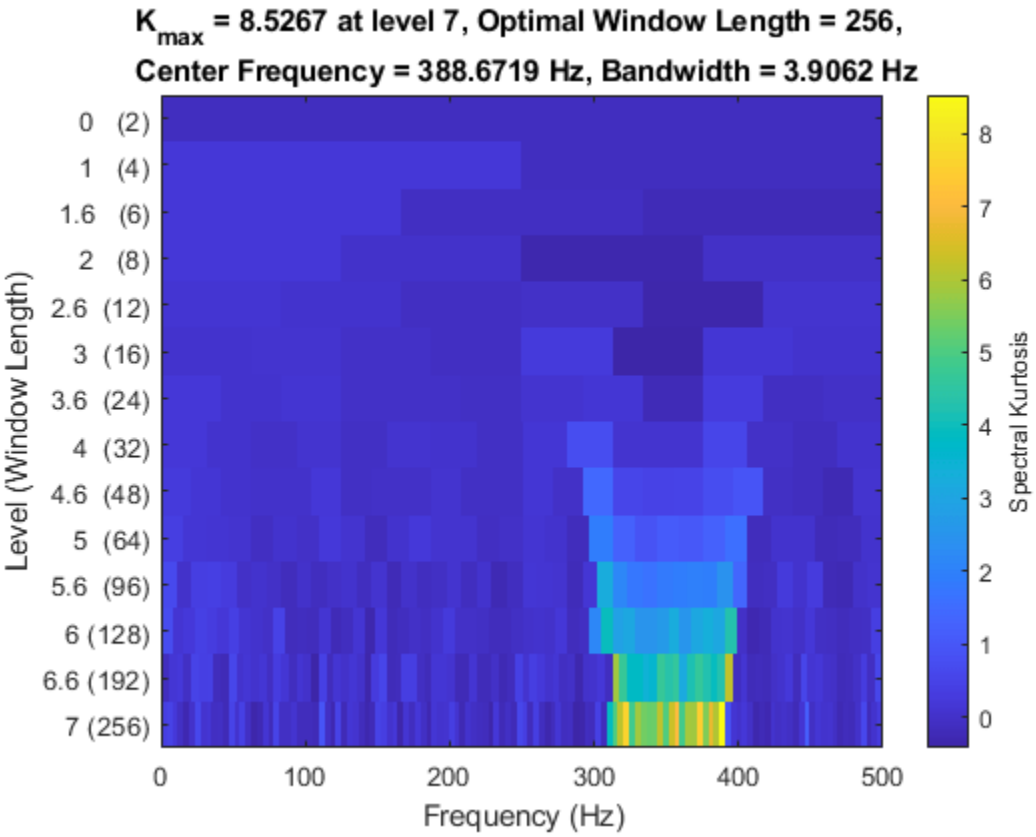
Plot the spectral kurtosis with the default window size.

```
pkurtosis(x,fs)
title('Spectral Kurtosis with Default Window Size')
```



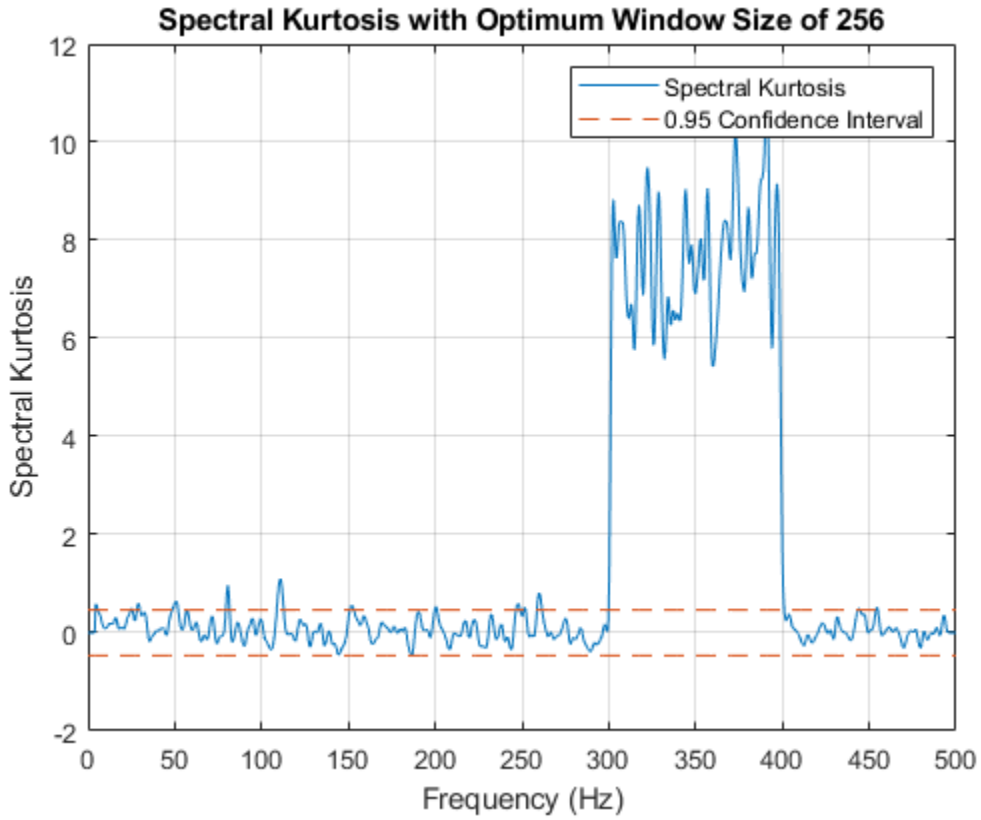
Now compute the optimal window size using `kurtogram`.

```
kurtogram(x,fs)
```



The kurtogram plot also illustrates the chirp between 300 and 400 Hz, and shows that the optimum window size is 256. Feed `w0` into `pkurtosis`.

```
w0 = 256;
pkurtosis(x,fs,w0)
title('Spectral Kurtosis with Optimum Window Size of 256')
```



The main excursion has higher kurtosis values. The higher values improve the differentiation between stationary and nonstationary components, and enhance your ability to extract the nonstationary component as a feature.

Input Arguments

x — Time-series signal

vector

Time-series signal for which `kurtogram` returns the fast kurtogram, specified as a vector.

sampx — Sample rate or sample time of signal

normalized frequency (default) | positive numeric scalar | `duration` scalar | numeric vector in seconds | `duration` array | `datetime` array

Sample rate or sample time, specified as one of the following:

- Positive numeric scalar — Frequency in hertz
- `duration` scalar — Time interval between consecutive samples of `X`
- Vector, `duration` array, or `datetime` array — Time instant or duration corresponding to each element of `x`

For an example, see “Compute the Kurtogram of a Nonstationary Signal” on page 1-1182.

When `sampx` represents a time vector, time samples can be nonuniform, with the `pspectrum` constraint that the median time interval and the mean time interval must obey:

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

If you specify `sampx` as empty, then `kurtogram` uses normalized frequency. In other words, it assumes an evenly spaced frequency vector spanning [0 π].

`xt` — Signal timetable
timetable

Signal timetable from which `kurtogram` returns the fast kurtogram, specified as a `timetable` that contains a single variable with a single column. `xt` must contain increasing, finite row times. If the timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”. `xt` can be nonuniformly sampled, with the `pspectrum` constraint that the median time interval and the mean time interval must obey:

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

`level` — maximum kurtogram level
positive integer

Maximum kurtogram level, which drives number of cases to compute, specified as a positive integer. Level drives the frequency window sizes that `kurtogram` uses, and therefore, the frequency resolution. As frequency resolution, goes up, time resolution goes down. The spectral kurtosis estimate will be poor if either resolution is too low. When you specify `level`, balance the impact on both time and frequency resolution.

Output Arguments

`kgram` — Fast kurtogram
matrix

Fast kurtogram, returned as a matrix with dimensions defined by `level`. `kgram` has 2*`level` rows and 3* $2^{\texttt{level}}$ columns. Each row of the matrix represents the spectral kurtosis results for each element in the frequency vector, and for the window size defined by the position of the row, with respect to the sequence:

$$[0, 1, \log_2(3), 2, 1 + \log_2(3), 3, 3 + \log_2(3), \ldots, n, n + \log_2(3), \ldots, level],$$

where the equivalent window size for a level n is 2^{n+1} samples.

`f` — Frequency vector
vector

Frequency vector associated with `kgram`, returned as a vector. The length of `f` is equal to the number of columns in `kgram`.

`w` — Window-size vector
vector

Window-size vector associated with `kgram`, returned as a vector. The length of `f` is equal to the number of columns in `kgram`.

`fc` — Frequency of maximal spectral kurtosis value

scalar

Frequency of maximal spectral kurtosis value in `kgram`, returned as a scalar:

- In rad/second, if you have not specified `sampx`, causing `kurtogram` to use normalized frequency
- In hertz, if `sampx` is defined

You can use `fc` as the central frequency for an optimal bandpass filter that maximizes the kurtosis of the envelope of the filtered signal. Maximizing the envelope kurtosis allows you to more easily extract the resulting impulsive component as a feature.

`wc` — Window size of maximal spectral kurtosis value

scalar

Window size of maximal spectral kurtosis value in `kgram`, returned as a scalar in samples. You can use `wc` to provide the optimal window size for `pkurtosis`. For an example, see “Plot Spectral Kurtosis Using a Customized Window Size” on page 1-1185.

`bw` — Suggested bandwidth for optimal bandpass filter

scalar

Suggested bandwidth for optimal bandpass filter, returned as a scalar:

- In rad/second, if you have not specified `sampx`, causing `kurtogram` to use normalized frequency
- In hertz, if you have specified `sampx`

You can use `bw` to create a filter that maximizes the kurtosis of the envelope of the filtered signal. `bw` is equal to `fx`/`wc`, where `fx` is the signal sample frequency that `kurtogram` derives from `sampx`.

More About

Kurtogram

The `kurtogram` function provides key information that you can use when you are performing “Spectral Kurtosis” on page 1-1616 analysis using `pkurtosis`. `kurtogram` calculates the spectral kurtosis for multiple window sizes using a fast kurtogram algorithm. Along with the kurtogram and its associated frequency and window vectors, `kurtogram` returns the optimal window size and other filter-tuning parameters. And it can visualize the results of its computations.

The fast kurtogram algorithm uses bandpass filtering along with a simplified computation to approximate the spectral kurtosis for each window size and frequency rather than compute the short-time Fourier transform (STFT) as the higher-fidelity `pkurtosis` does. It also reduces the number of iterations the algorithm requires to span the frequency-window plane relative to the full kurtogram [1].

References

[1] Antoni, J., and R. B. Randall. "Fast Computation of the Kurtogram for the Detection of Transient Faults." *Mechanical Systems and Signal Processing* . Vol. 20, Issue 1, 2007, pp. 108–124.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Inputs must be double precision.
- Timetables are not supported for code generation.

See Also

`pkurtosis` | `pentropy` | `pspectrum`

Introduced in R2018a

labelDefinitionsHierarchy

Get hierarchical list of label and sublabel names

Syntax

```
str = labelDefinitionsHierarchy(lbldefs)
str = labelDefinitionsHierarchy(lss)
```

Description

`str = labelDefinitionsHierarchy(lbldefs)` returns a character array with a hierarchical list of label and sublabel names contained in `lbldefs`, a vector of `signalLabelDefinition` objects.

`str = labelDefinitionsHierarchy(lss)` returns a character array with a hierarchical list of label and sublabel names contained in the `labeledSignalSet` object `lss`.

Examples

Label Hierarchy

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

             Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Visualize the label hierarchy of the set.

```
labelDefinitionsHierarchy(lss)
```

```
ans =
    'WhaleType
       Sublabels: []
     MoanRegions
       Sublabels: []
     TrillRegions
       Sublabels: TrillPeaks
     '
```

Input Arguments

`lbldefs` — Signal label definitions

`signalLabelDefinition` object | vector of `signalLabelDefinition` objects

Signal label definitions, specified as a `signalLabelDefinition` object or a vector of `signalLabelDefinition` objects.

Example: `signalLabelDefinition("Asleep",'LabelType','roi','LabelDataType','logical')` can label a region of a signal in which a patient is asleep.

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

Output Arguments

`str` — List of label and sublabel names

character array

List of label and sublabel names, returned as a character array.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

labelDefinitionsSummary

Get summary table of signal label definitions

Syntax

```
T = labelDefinitionsSummary(lbldefs)
T = labelDefinitionsSummary(lss)

T = labelDefinitionsSummary( ___ ,lblname)
T = labelDefinitionsSummary( ___ ,lblname,'sublbls')
```

Description

`T = labelDefinitionsSummary(lbldefs)` returns a table, `T`, with the properties of the label definitions contained in `lbldefs`, a vector of `signalLabelDefinition` objects.

`T = labelDefinitionsSummary(lss)` returns a table, `T`, with the properties of the label definitions contained in the `labeledSignalSet` object `lss`.

`T = labelDefinitionsSummary( ___ ,lblname)` returns a table with the properties of the label `lblname`.

`T = labelDefinitionsSummary( ___ ,lblname,'sublbls')` returns a table of the properties of the sublabels defined for `lblname`.

Examples

Label Properties

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
        NumMembers: 2
   TimeInformation: "sampleRate"
        SampleRate: 4000
            Labels: [2x3 table]
       Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Visualize the label properties of the set.

```
labelDefinitionsSummary(lss)
```

```
ans=3×9 table
      LabelName        LabelType     LabelDataType     Categories     ValidationFunction    Defau
    _____________    ___________    _____________    ____________    __________________    _____

    "WhaleType"      "attribute"    "categorical"    {3x1 string}       {["N/A"   ]}       {0x0
    "MoanRegions"    "roi"          "logical"        {["N/A"   ]}       {0x0 double}       {0x0
    "TrillRegions"   "roi"          "logical"        {["N/A"   ]}       {0x0 double}       {0x0
```

Visualize the properties of the `TrillRegions` label.

```
labelDefinitionsSummary(lss,"TrillRegions")
```

```
ans=1×9 table
      LabelName        LabelType    LabelDataType    Categories    ValidationFunction    DefaultVa
    _____________    ___________    _____________    __________    __________________    _________

    "TrillRegions"      "roi"         "logical"      {["N/A"]}        {0x0 double}       {0x0 doub
```

Visualize the properties of the `TrillRegions` sublabels.

```
labelDefinitionsSummary(lss,"TrillRegions",'sublbls')
```

```
ans=1×8 table
     LabelName       LabelType    LabelDataType    Categories    ValidationFunction    DefaultValu
    ____________    ___________    _____________    __________    __________________    ___________

    "TrillPeaks"      "point"       "numeric"      {["N/A"]}        {0x0 double}       {0x0 double
```

Input Arguments

`lbldefs` — Signal label definitions

`signalLabelDefinition` object | vector of `signalLabelDefinition` objects

Signal label definitions, specified as a `signalLabelDefinition` object or a vector of `signalLabelDefinition` objects.

Example: `signalLabelDefinition("Asleep",'LabelType','roi','LabelDataType','logical')` can label a region of a signal in which a patient is asleep.

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`lblname` — Label or sublabel name

character vector | string scalar | cell array of character vectors | string array

Label or sublabel name. To specify a label, use a character vector or a string scalar. To specify a sublabel, use a two-element cell array of character vectors or a two-element string array:

- The first element is the name of the parent label.
- The second element is the name of the sublabel.

Example: `signalLabelDefinition("Asleep",'LabelType','roi')` specifies a label of name `"Asleep"` for a region of a signal in which a patient is asleep during a clinical trial.

Example: `{'Asleep' 'REM'}` or `["Asleep" "REM"]` specifies a region of a signal in which a patient undergoes REM sleep.

Output Arguments

T — Summary table

table

Summary table with the properties of a label.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

labeledSignalSet

Create labeled signal set

Description

Use `labeledSignalSet` to store labeled signals along with the label definitions. Create signal label definitions using `signalLabelDefinition`.

Creation

Syntax

```
lss = labeledSignalSet

lss = labeledSignalSet(src)
lss = labeledSignalSet(src,lbldefs)

lss = labeledSignalSet(src,lbldefs,'MemberNames',mnames)
lss = labeledSignalSet(src,lbldefs,Name,Value)
```

Description

`lss = labeledSignalSet` creates an empty labeled signal set. Use `addMembers` to add signals to the set. Use `addLabelDefinitions` to add label definitions to the set.

`lss = labeledSignalSet(src)` creates a labeled signal set for the input data source `src`. Use `addLabelDefinitions` to add label definitions to the set.

`lss = labeledSignalSet(src,lbldefs)` creates a labeled signal set for the input data source `src` using the signal label definitions `lbldefs`. Use `signalLabelDefinition` to create signal label definitions.

`lss = labeledSignalSet(src,lbldefs,'MemberNames',mnames)` creates a labeled signal set for the input data source `src` and specifies names for the members of the set. Use `setMemberNames` to modify the member names. `lbldefs` is optional.

`lss = labeledSignalSet(src,lbldefs,Name,Value)` sets “Properties” on page 1-1199 using name-value arguments. You can specify multiple name-value arguments. Enclose each property name in quotes. `lbldefs` is optional.

Input Arguments

`src` — Input data source
matrix | cell array | timetable | `signalDatastore` object | `audioDatastore` object

Input data source, specified as a matrix, a cell array, a timetable, a `signalDatastore` object, or an `audioDatastore` object. `src` implicitly specifies the number of members of the set, the number of signals in each member, and the data in each signal.

Example: `{randn(10,3),randn(17,9)}` has two members. The first member contains three 10-sample signals. The second member contains nine 17-sample signals.

Example: `{{randn(10,1)},{randn(17,1),randn(27,1)}}` has two members. The first member contains one 10-sample signal. The second member contains a 17-sample signal and a 27-sample signal.

Example: `{{timetable(seconds(1:10)',randn(10,3)),timetable(seconds(1:7)',randn(7,2))}, {timetable(seconds(1:3)',randn(3,1))}}` has two members. The first member contains three signals sampled at 1 Hz for 10 seconds and two signals sampled at 1 Hz for 7 seconds. The second member contains one signal sampled at 1 Hz for 3 seconds.

Example: signalDatastore Object Pointing to Files

Specify the path to a set of sample sound signals included as MAT-files with MATLAB®. Each file contains a signal variable and a sample rate. List the names of the files.

```
folder = fullfile(matlabroot,"toolbox","matlab","audiovideo");
lst = dir(append(folder,"/*.mat"));
nms = {lst(:).name}'
```

```
nms = 7x1 cell
    {'chirp.mat'   }
    {'gong.mat'    }
    {'handel.mat'  }
    {'laughter.mat'}
    {'mtlb.mat'    }
    {'splat.mat'   }
    {'train.mat'   }
```

Create a signal datastore that points to the specified folder. Set the sample rate variable name to `Fs`, which is common to all files. Generate a subset of the datastore that excludes the file `mtlb.mat`, which differs from the other files in that the signal variable is not called `y`.

```
sds = signalDatastore(folder,"SampleRateVariableName","Fs");
sdss = subset(sds,~strcmp(nms,"mtlb.mat"));
```

Use the subset datastore as the source for a `labeledSignalSet` object.

```
lss = labeledSignalSet(sdss)
```

```
lss =
  labeledSignalSet with properties:

            Source: [1x1 signalDatastore]
         NumMembers: 6
    TimeInformation: "inherent"
            Labels: [6x0 table]
       Description: ""

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

`lbldefs` — Label definitions

vector of `signalLabelDefinition` objects

Label definitions, specified as a vector of `signalLabelDefinition` objects.

`mnames` — Member names
character vector | string scalar | cell array of character vectors | string array

Member names, specified as a character vector, a string scalar, a cell array of character vectors, or a string array.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},'MemberNames',{'llama' 'alpaca'})` specifies a set of random signals with two members, `'llama'` and `'alpaca'`.

Properties

`Description` — Labeled signal set description
character vector | string scalar

Labeled signal set description, specified as a character vector or string scalar.

Example: `'Description','Sleep test patients by sex and age'`

Data Types: `char` | `string`

`SampleRate` — Sample rate values
positive scalar | vector

This property is read-only.

Sample rate values, specified as a positive scalar or a vector. This property is valid only when the data source does not contain inherent time information.

- Set `SampleRate` to a positive numeric scalar to specify the same sample rate for all signals in the labeled set.
- Set `SampleRate` to a vector to specify that each member of the labeled set has signals sampled at the same rate, but the sample rates differ from member to member. The vector must have a number of elements equal to the number of members of the set. If a member of a set has signals with different sample rates, then specify the sample rates using timetables.

Example: `'SampleRate',[1e2 1e3]` specifies that the signals in the first member of a set are sampled at a rate of 100 Hz and the signals in the second member are sampled at 1 kHz.

`SampleTime` — Sample time values
positive scalar | vector | `duration` scalar | `duration` vector

This property is read-only.

Sample time values, specified as a positive scalar, a vector, a `duration` scalar, or a `duration` vector. This property is valid only when the data source does not contain inherent time information.

- Set `SampleTime` to a numeric or `duration` scalar to specify the same sample time for all signals in the labeled set.
- Set `SampleTime` to a numeric or `duration` vector to specify that each member of the labeled set has signals with the same time interval between samples, but the intervals differ from member to member. The vector must have a number of elements equal to the number of members of the set. If a member of a set has signals with different sample times, then specify the sample times using timetables.

Example: `'SampleTime',seconds([1e-2 1e-3])` specifies that the signals in the first member of a set have 0.01 second between samples, and the signals in the second member have 1 millisecond between samples.

`TimeValues` — Time values

vector | `duration` vector | matrix | cell array

This property is read-only.

Time values, specified as a vector, a `duration` vector, a matrix, or a cell array. This property is valid only when the data source does not contain inherent time information. Time values must be unique and increasing.

- Set `TimeValues` to a numeric or `duration` vector to specify the same time values for all signals in the labeled set. The vector must have the same length as all the signals in the set.
- Set `TimeValues` to a numeric or `duration` matrix or cell array to specify that each member of the labeled set has signals with the same time values, but the time values differ from member to member.
  - If `TimeValues` is a matrix, then it must have a number of columns equal to the number of members of the set. All signals in the set must have a length equal to the number of rows of the matrix.
  - If `TimeValues` is a cell array, then it must contain a number of vectors equal to the number of members of the set. All signals in a member must have a length equal to the number of elements of the corresponding vector in the cell array.

If a member of a set has signals with different time values, then specify the time values using timetables.

Example: `'TimeValues',[1:1000;0:1/500:2-1/500]'` specifies that the signals in the first member of a set are sampled 1 Hz for 1000 seconds. The signals in the second member are sampled at 500 Hz for 2 seconds.

Example: `'TimeValues',seconds([1:1000;0:1/500:2-1/500]')` specifies that the signals in the first member of a set are sampled 1 Hz for 1000 seconds. The signals in the second member are sampled at 500 Hz for 2 seconds.

Example: `'TimeValues',{1:1000,0:1/500:2-1/500}` specifies that the signals in the first member of a set are sampled 1 Hz for 1000 seconds. The signals in the second member are sampled at 500 Hz for 2 seconds.

Example: `'TimeValues',{seconds(1:1000),seconds(0:1/500:2-1/500)}` specifies that the signals in the first member of a set are sampled 1 Hz for 1000 seconds. The signals in the second member are sampled at 500 Hz for 2 seconds.

`NumMembers` — Number of members in set

positive integer

This property is read-only.

Number of members in set, specified as a positive integer.

`Labels` — Labels table

table

This property is read-only.

Labels table, specified as a MATLAB table. Each variable of `Labels` corresponds to a label defined for the set. Each row of `Labels` corresponds to a member of the data source. The row names of `Labels` are the member names.

Data Types: `table`

`TimeInformation` — Time information of source
`'none'` | `'sampleRate'` | `'sampleTime'` | `'timeValues'` | `'inherent'`

Time information of source, specified as one of the following:

- `'none'` — The signals in the source have no time information.
- `'sampleRate'` — The signals in the source are sampled at a specified rate.
- `'sampleTime'` — The signals in the source have a specified time interval between samples.
- `'timeValues` — The signals in the source have a time value corresponding to each sample.
- `'inherent'` — The signals in the source contain inherent time information. MATLAB timetables are an example of such signals.

Data Types: `char` | `string`

`Source` — Data source of labeled signal set
matrix | cell array | timetable

This property is read-only.

Data source of labeled signal set, specified as a matrix, a timetable, a cell array, or an audio datastore.

- If `Source` is a numeric matrix, then the labeled signal set has one member that contains a number of signals equal to the number of matrix columns.

  Example: `labeledSignalSet(randn(10,3))` has one member that contains three 10-sample signals.
- If `Source` is a cell array of matrices, then the labeled signal set has a number of members equal to the number of matrices in the cell array. Each member contains a number of signals equal to the number of columns of the corresponding matrix.

  Example: `labeledSignalSet({randn(10,3),randn(17,9)})` has two members. The first member contains three 10-sample signals. The second member contains nine 17-sample signals.
- If `Source` is a cell array, and each element of the cell array is a cell array of numeric vectors, then the labeled signal set has a number of members equal to the number of cell array elements. Each signal within a member can have any length.

  Example: `labeledSignalSet({{randn(10,1)},{randn(17,1),randn(27,1)}})` has two members. The first member contains one 10-sample signal. The second member contains a 17-sample signal and a 27-sample signal.
- If `Source` is a timetable with variables containing numeric values, then the labeled signal set has one member that contains a number of signals equal to the number of variables. The time values of the timetable must be of type `duration`, unique, and increasing.

  Example: `labeledSignalSet(timetable(seconds(1:10)',randn(10,3)))` has one member that contains three signals sampled at 1 Hz for 10 seconds.

- If `Source` is a cell array of timetables, and each timetable has an arbitrary number of variables with numeric values, then the labeled signal set has a number of members equal to the number of timetables. Each member contains a number of signals equal to the number of variables in the corresponding timetable.

  Example:
  `labeledSignalSet({timetable(seconds(1:10)',randn(10,3)),timetable(seconds(1:5)',randn(5,13))})` has two members. The first member contains three signals sampled at 1 Hz for 10 seconds. The second member contains 13 signals sampled at 1 Hz for 5 seconds.
- If `Source` is a cell array, and each element of the cell array is a cell array of timetables, then the labeled signal set has a number of members equal to the number of cell array elements. Each member can have any number of timetables, and each timetable within a member can have any number of variables.

  Example:
  `labeledSignalSet({{timetable(seconds(1:10)',randn(10,3)),timetable(seconds(1:7)',randn(7,2))},{timetable(seconds(1:3)',randn(3,1))}})` has two members. The first member contains three signals sampled at 1 Hz for 10 seconds and two signals sampled at 1 Hz for 7 seconds. The second member contains one signal sampled at 1 Hz for 3 seconds.
- If the input data source, `src`, is an audio datastore, then the labeled signal set has a number of members equal to the number of files to which the datastore points. The `Source` property contains a cell array of character vectors with the file names. Each member contains all the signals returned by the read of the corresponding datastore file.

Object Functions

addLabelDefinitions	Add label definitions to labeled signal set
addMembers	Add members to labeled signal set
countLabelValues	Count label values
createDatastores	Create datastores pointing to signal and label data
createFeatureData	Create feature table or matrix and response vectors
editLabelDefinition	Edit label definition properties
getAlternateFileSystemRoots	Get alternate file system roots when data source of labeled signal set is a datastore
getLabelDefinitions	Get label definitions in labeled signal set
getLabeledSignal	Get labeled signals from labeled signal set
getLabelIndices	Get label indices pointing to label definitions in labeled signal set
getLabelNames	Get label names in labeled signal set
getLabelValues	Get label values from labeled signal set
getMemberNames	Get member names in labeled signal set
getSignal	Get signals from labeled signal set
head	Get top rows of labels table
labelDefinitionsHierarchy	Get hierarchical list of label and sublabel names
labelDefinitionsSummary	Get summary table of signal label definitions
merge	Merge two or more labeled signal sets
removeLabelDefinition	Remove label definition from labeled signal set
removeMembers	Remove members from labeled signal set
removePointValue	Remove row from point label
removeRegionValue	Remove row from ROI label
resetLabelValues	Reset labels to default values
setAlternateFileSystemRoots	Set alternate file system roots when data source of labeled signal set is a datastore

setLabelValue	Set label value in labeled signal set
setMemberNames	Set member names in labeled signal set
subset	Get new labeled signal set with subset of members

Examples

Label Definitions for Whale Songs

Consider a set of whale sound recordings. The recorded whale sounds consist of trills and moans. *Trills* sound like series of clicks. *Moans* are low-frequency cries similar to the sound made by a ship's horn. You want to look at each signal and label it to identify the whale type, the trill regions, and the moan regions. For each trill region, you also want to label the signal peaks higher than a certain threshold.

Signal Label Definitions

Define an attribute label to store whale types. The possible categories are blue whale, humpback whale, and white whale.

```
dWhaleType = signalLabelDefinition('WhaleType',...
   'LabelType','attribute',...
   'LabelDataType','categorical',...
   'Categories',{'blue','humpback','white'},...
   'Description','Whale type');
```

Define a region-of-interest (ROI) label to capture moan regions. Define another ROI label to capture trill regions.

```
dMoans = signalLabelDefinition('MoanRegions',...
   'LabelType','roi',...
   'LabelDataType','logical',...
   'Description','Regions where moans occur');

dTrills = signalLabelDefinition('TrillRegions',...
   'LabelType','roi',...
   'LabelDataType','logical',...
   'Description','Regions where trills occur');
```

Finally, define a point label to capture the trill peaks. Set this label as a sublabel of the `dTrills` definition.

```
dTrillPeaks = signalLabelDefinition('TrillPeaks',...
   'LabelType','point',...
   'LabelDataType','numeric',...
   'Description','Trill peaks');

dTrills.Sublabels = dTrillPeaks;
```

Labeled Signal Set

Create a `labeledSignalSet` with the whale signals and the label definitions. Add label values to identify the whale type, the moan and trill regions, and the peaks of the trills.

```
load labelwhalesignals
lbldefs = [dWhaleType dMoans dTrills];
```

```
lss = labeledSignalSet({whale1 whale2},lbldefs,'MemberNames',{'Whale1','Whale2'}, ...
   'SampleRate',Fs,'Description','Characterize whale song regions');
```

Visualize the label hierarchy and label properties using `labelDefinitionsHierarchy` and `labelDefinitionsSummary`.

```
labelDefinitionsHierarchy(lss)
```

```
ans =
    'WhaleType
       Sublabels: []
     MoanRegions
       Sublabels: []
     TrillRegions
       Sublabels: TrillPeaks
     '
```

```
labelDefinitionsSummary(lss)
```

```
ans=3×9 table
      LabelName        LabelType     LabelDataType     Categories     ValidationFunction    Defau
    _____________    ____________    _____________    ____________    __________________    _____

    "WhaleType"      "attribute"     "categorical"    {3x1 string}       {["N/A"   ]}       {0x0
    "MoanRegions"    "roi"           "logical"        {["N/A"   ]}       {0x0 double}      {0x0
    "TrillRegions"   "roi"           "logical"        {["N/A"   ]}       {0x0 double}      {0x0
```

The signals in the loaded data correspond to songs of two blue whales. Set the `'WhaleType'` values for both signals.

```
setLabelValue(lss,1,'WhaleType','blue');
setLabelValue(lss,2,'WhaleType','blue');
```

Visualize the `'Labels'` property. The table has the newly added `'WhaleType'` values for both signals.

```
lss.Labels
```

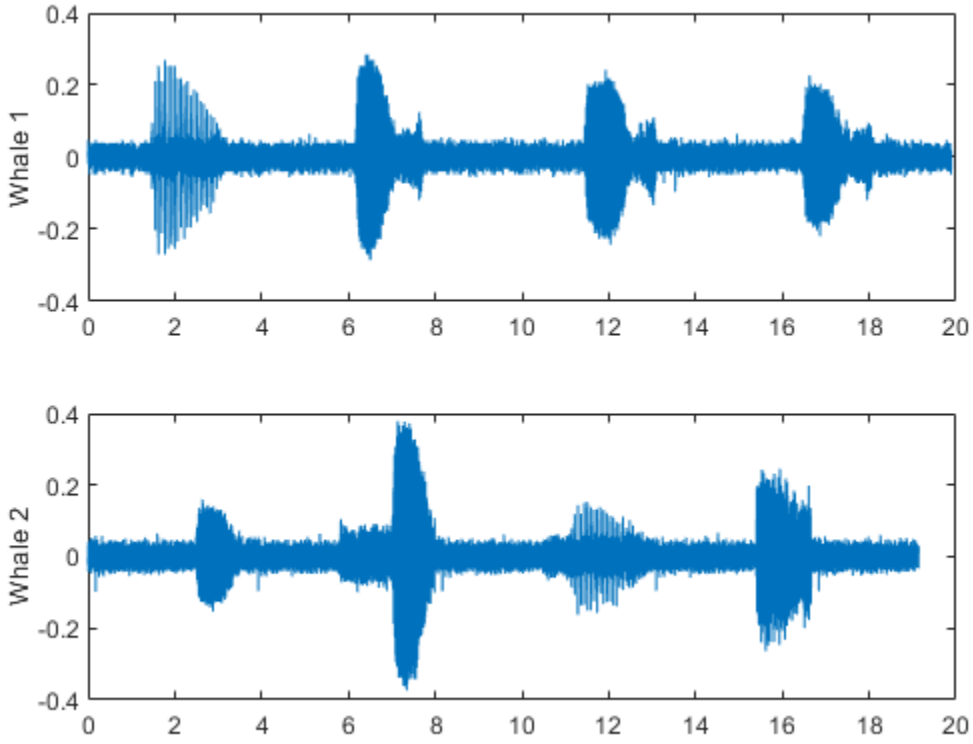
```
ans=2×3 table
              WhaleType    MoanRegions    TrillRegions
              _________    ___________    ____________

    Whale1      blue       {0x2 table}    {0x3 table}
    Whale2      blue       {0x2 table}    {0x3 table}
```

Visualize Region Labels

Visualize the whale songs to identify the trill and moan regions.

```
subplot(2,1,1)
plot((0:length(whale1)-1)/Fs,whale1)
ylabel('Whale 1')

subplot(2,1,2)
```

```
plot((0:length(whale2)-1)/Fs,whale2)
ylabel('Whale 2')
```



Moan regions are sustained low-frequency wails.

- `whale1` has moans centered at about 7 seconds, 12 seconds, and 17 seconds.
- `whale2` has moans centered at about 3 seconds, 7 seconds, and 16 seconds.

Add the moan regions to the labeled set. Specify the ROI limits in seconds and the label values.

```
moanRegionsWhale1 = [6.1 7.7; 11.4 13.1; 16.5 18.1];
mrsz1 = [size(moanRegionsWhale1,1) 1];
setLabelValue(lss,1,'MoanRegions',moanRegionsWhale1,true(mrsz1));

moanRegionsWhale2 = [2.5 3.5; 5.8 8; 15.4 16.7];
mrsz2 = [size(moanRegionsWhale2,1) 1];
setLabelValue(lss,2,'MoanRegions',moanRegionsWhale2,true(mrsz2));
```

Trill regions have distinct bursts of sound punctuated by silence.

- `whale1` has a trill centered at about 2 seconds.
- `whale2` has a trill centered at about 12 seconds.

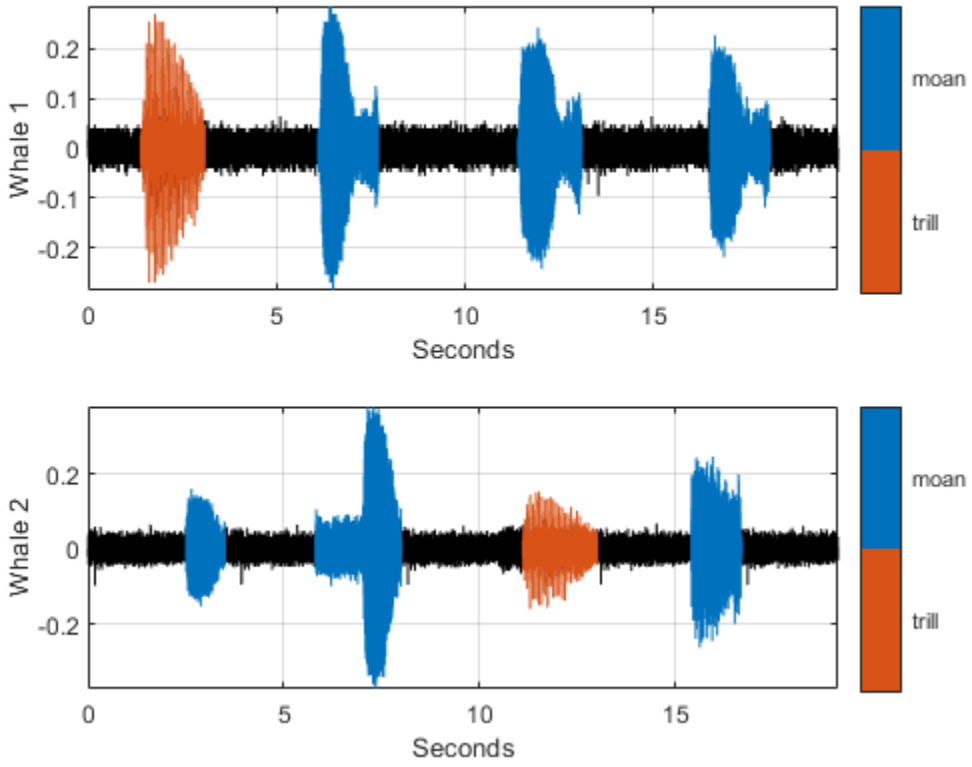
Add the trill regions to the labeled set.

```
trillRegionWhale1 = [1.4 3.1];
trsz1 = [size(trillRegionWhale1,1) 1];
```

```
setLabelValue(lss,1,'TrillRegions',trillRegionWhale1,true(trsz1));

trillRegionWhale2 = [11.1 13];
trsz2 = [size(trillRegionWhale1,1) 1];
setLabelValue(lss,2,'TrillRegions',trillRegionWhale2,true(trsz2));
```

Create a `signalMask` object for each whale song and use it to visualize and label the different regions. For better visualization, change the label values from logical to categorical.

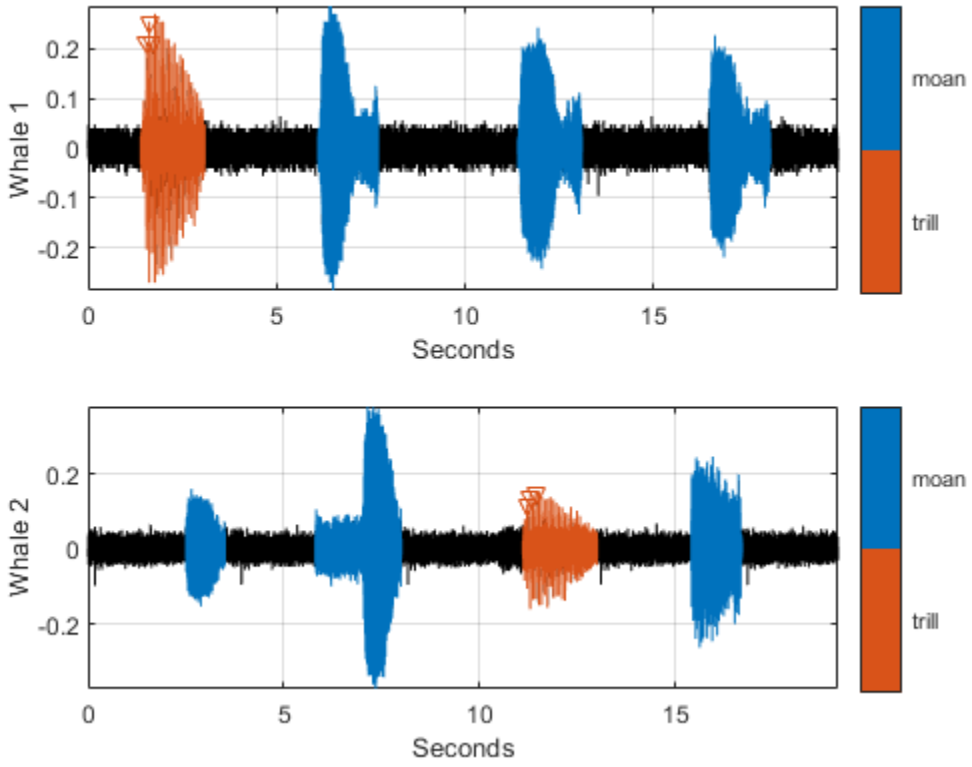
```
mr1 = getLabelValues(lss,1,'MoanRegions');
mr1.Value = categorical(repmat("moan",mrsz1));
tr1 = getLabelValues(lss,1,'TrillRegions');
tr1.Value = categorical(repmat("trill",trsz1));

msk1 = signalMask([mr1;tr1],'SampleRate',Fs);

subplot(2,1,1)
plotsigroi(msk1,whale1)
ylabel('Whale 1')
hold on

mr2 = getLabelValues(lss,2,'MoanRegions');
mr2.Value = categorical(repmat("moan",mrsz2));
tr2 = getLabelValues(lss,2,'TrillRegions');
tr2.Value = categorical(repmat("trill",trsz2));

msk2 = signalMask([mr2;tr2],'SampleRate',Fs);

subplot(2,1,2)
plotsigroi(msk2,whale2)
ylabel('Whale 2')
hold on
```



Visualize Point Labels

Label three peaks for each trill region. For point labels, you specify the point locations and the label values. In this example, the point locations are in seconds.

```
peakLocsWhale1 = [1.553 1.626 1.7];
peakValsWhale1 = [0.211 0.254 0.211];

setLabelValue(lss,1,{'TrillRegions','TrillPeaks'}, ...
   peakLocsWhale1,peakValsWhale1,'LabelRowIndex',1);

subplot(2,1,1)
plot(peakLocsWhale1,peakValsWhale1,'v')
hold off

peakLocsWhale2 = [11.214 11.288 11.437];
peakValsWhale2 = [0.119 0.14 0.15];

setLabelValue(lss,2,{'TrillRegions','TrillPeaks'}, ...
   peakLocsWhale2,peakValsWhale2,'LabelRowIndex',1);

subplot(2,1,2)
plot(peakLocsWhale2,peakValsWhale2,'v')
hold off
```



Explore Label Values

Explore the label values using `getLabelValues`.

```
getLabelValues(lss)
```

```
ans=2×3 table
              WhaleType    MoanRegions    TrillRegions
              _________    ___________    ____________

    Whale1      blue       {3x2 table}    {1x3 table}
    Whale2      blue       {3x2 table}    {1x3 table}
```

Retrieve the moan regions for the first member of the labeled set.

```
getLabelValues(lss,1,'MoanRegions')
```

```
ans=3×2 table
     ROILimits      Value
    ____________    _____

     6.1     7.7    {[1]}
    11.4    13.1    {[1]}
    16.5    18.1    {[1]}
```

Use a second output argument to list the sublabels of a label.

```
[value,valueWithSublabel] = getLabelValues(lss,1,'TrillRegions')
```

```
value=1×2 table
    ROILimits     Value
    __________    _____

    1.4    3.1    {[1]}


valueWithSublabel=1×3 table
    ROILimits     Value     Sublabels
                           TrillPeaks
    __________    _____    ___________

    1.4    3.1    {[1]}    {3x2 table}
```

To retrieve the values in a sublabel, express the label name as a two-element array.

```
getLabelValues(lss,1,{'TrillRegions','TrillPeaks'})
```

```
ans=3×2 table
    Location      Value
    ________    __________

     1.553      {[0.2110]}
     1.626      {[0.2540]}
       1.7      {[0.2110]}
```

Find the value of the third trill peak corresponding to the second member of the set.

```
getLabelValues(lss,2,{'TrillRegions','TrillPeaks'}, ...
    'LabelRowIndex',1,'SublabelRowIndex',3)
```

```
ans=1×2 table
    Location      Value
    ________    __________

     11.437     {[0.1500]}
```

Count Label Values and Create Datastores

Specify the path to a set of audio signals included as MAT-files with MATLAB®. Each file contains a signal variable and a sample rate. List the names of the files.

```
folder = fullfile(matlabroot,"toolbox","matlab","audiovideo");
lst = dir(append(folder,"/*.mat"));
nms = {lst(:).name}'
```

```
nms = 7x1 cell
    {'chirp.mat'   }
    {'gong.mat'    }
    {'handel.mat'  }
    {'laughter.mat'}
    {'mtlb.mat'    }
```

```
    {'splat.mat'   }
    {'train.mat'   }
```

Create a signal datastore that points to the specified folder. Set the sample rate variable name to `Fs`, which is common to all files. Generate a subset of the datastore that excludes the file `mtlb.mat`. Use the subset datastore as the source for a `labeledSignalSet` object.

```
sds = signalDatastore(folder,"SampleRateVariableName","Fs");
sds = subset(sds,~strcmp(nms,"mtlb.mat"));
lss = labeledSignalSet(sds);
```

Create three label definitions to label the signals:

- Define a logical attribute label that is true for signals that contain human voices.
- Define a numeric point label that marks the location and amplitude of the maximum of each signal.
- Define a categorical region-of-interest (ROI) label to pick out nonoverlapping, uniform-length random regions of each signal.

Add the signal label definitions to the labeled signal set.

```
vc = signalLabelDefinition("Voice",'LabelType','attribute', ...
    'LabelDataType','logical','DefaultValue',false);
mx = signalLabelDefinition("Maximum",'LabelType','point', ...
    'LabelDataType','numeric');
rs = signalLabelDefinition("RanROI",'LabelType','ROI', ...
    'LabelDataType','categorical','Categories',["ROI" "other"]);
addLabelDefinitions(lss,[vc mx rs])
```

Label the signals:

- Label `'handel.mat'` and `'laughter.mat'` as having human voices.
- Use the `islocalmax` function to find the maximum of each signal. Label its location and value.
- Use the `randROI` on page 1-0 function to generate as many regions of length $N/10$ samples as can fit in a signal of length N given a minimum separation of $N/6$ samples between regions. Label their locations and assign them to the `ROI` category.

When labeling points and regions, convert sample values to time values. Subtract 1 to account for MATLAB® array indexing and divide by the sample rate.

```
kj = 1;
while hasdata(sds)

    [sig,info] = read(sds);
    fs = info.SampleRate;

    [~,fn] = fileparts(info.FileName);
    if fn=="handel" || fn=="laughter"
        setLabelValue(lss,kj,"Voice",true)
    end

    xm = find(islocalmax(sig,'MaxNumExtrema',1));
    setLabelValue(lss,kj,"Maximum",(xm-1)/fs,sig(xm))

    N = length(sig);
```

```
    rois = randROI(N,round(N/10),round(N/6));
    setLabelValue(lss,kj,"RanROI",(rois-1)/fs,repelem("ROI",size(rois,1)))

    kj = kj+1;

end
```

Verify that only two signals contain voices.

```
countLabelValues(lss,"Voice")
```

```
ans=2×3 table
    Voice    Count    Percent
    _____    _____    _______

    false      4      66.667
    true       2      33.333
```

Verify that two signals have a maximum amplitude of 1.

```
countLabelValues(lss,"Maximum")
```

```
ans=5×4 table
           Maximum             Count    Percent    MemberCount
    ___________________________    _____    _______    ___________

    0.80000000000000004441         1      16.667          1
    0.89113331915798421612         1      16.667          1
    0.94730769230769229505         1      16.667          1
    1                              2      33.333          2
    1.0575668990330560071          1      16.667          1
```

Verify that each signal has four nonoverlapping random regions of interest.

```
countLabelValues(lss,"RanROI")
```

```
ans=2×4 table
    RanROI    Count    Percent    MemberCount
    ______    _____    _______    ___________

    ROI        24        100           6
    other       0          0           0
```

Create two datastores with the data in the labeled signal set:

- The `signalDatastore` object `sd` contains the signal data.
- The `arrayDatastore` object `ld` contains the labeling information. Specify that you want to include the information corresponding to all the labels you created.

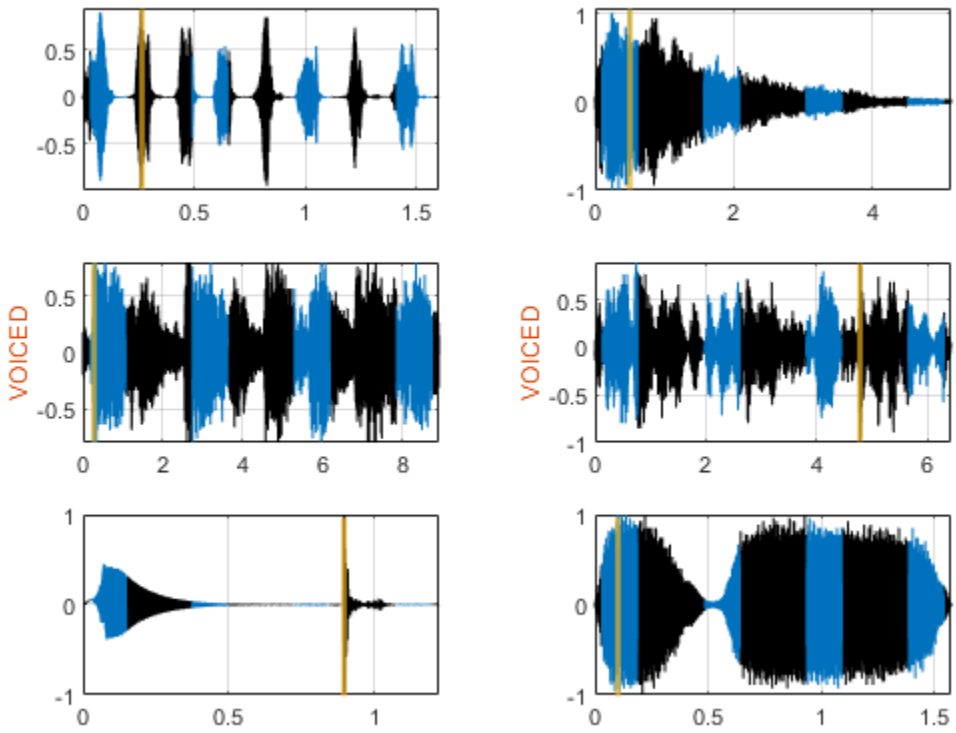
```
[sd,ld] = createDatastores(lss,["Voice" "RanROI" "Maximum"]);
```

Use the information in the datastores to plot the signals and display their labels.

- Use a `signalMask` object to highlight the regions of interest in blue.
- Plot yellow lines to mark the locations of the maxima.

- Add a red axis label to the signals that contain human voices.

```
tiledlayout flow

while hasdata(sd)

    [sg,nf] = read(sd);

    lbls = read(ld);

    nexttile

    msk = signalMask(lbls{:}.RanROI{:},'SampleRate',nf.SampleRate);
    plotsigroi(msk,sg)
    colorbar off
    xlabel('')

    xline(lbls{:}.Maximum{:}.Location, ...
        'LineWidth',2,'Color','#EDB120')

    if lbls{:}.Voice{:}
        ylabel('VOICED','Color','#D95319')
    end

end
```




```
function roilims = randROI(N,wid,sep)
```

```
num = floor((N+sep)/(wid+sep));
hq = histcounts(randi(num+1,1,N-num*wid-(num-1)*sep),(1:num+2)-1/2);
roilims = (1 + (0:num-1)*(wid+sep) + cumsum(hq(1:num)))' + [0 wid-1];

end
```

See Also

Apps
Signal Labeler

Objects
`signalLabelDefinition` | `signalMask`

Introduced in R2018b

addLabelDefinitions

Add label definitions to labeled signal set

Syntax

```
addLabelDefinitions(lss,lbldefs)
addLabelDefinitions(lss,lbldefs,lblname)
```

Description

`addLabelDefinitions(lss,lbldefs)` adds the labels defined in the vector of signal label definitions `lbldefs` to the labeled signal set `lss`.

`addLabelDefinitions(lss,lbldefs,lblname)` adds the labels defined in `lbldefs` as sublabels of the label `lblname`.

Examples

Add Label Definition

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Create a label definition that specifies whether a signal corresponds to a calf or to an adult whale.

```
calf = signalLabelDefinition('Calf','LabeldataType','logical','DefaultValue',false, ...
    'Description','Is the specimen a calf, or an adult?')
```

```
calf =
  signalLabelDefinition with properties:

                 Name: "Calf"
            LabelType: "attribute"
         LabelDataType: "logical"
    ValidationFunction: []
```

```
          DefaultValue: 0
             Sublabels: [0x0 signalLabelDefinition]
                   Tag: ""
           Description: "Is the specimen a calf, or an adult?"

 Use labeledSignalSet to create a labeled signal set.
```

Add the definition to the labeled signal set. Retrieve the names of the labels.

```
addLabelDefinitions(lss,calf)

getLabelNames(lss)
```

```
ans = 4x1 string
    "WhaleType"
    "MoanRegions"
    "TrillRegions"
    "Calf"
```

Create a label definition that specifies the sex of the whale. Add the label to the set as a sublabel of `'WhaleType'`.

```
sx = signalLabelDefinition('Sex','LabelDataType','categorical', ...
    'Categories',["male" "female"]);
addLabelDefinitions(lss,sx,'WhaleType')

labelDefinitionsHierarchy(lss)
```

```
ans =
    'WhaleType
       Sublabels: Sex
     MoanRegions
       Sublabels: []
     TrillRegions
       Sublabels: TrillPeaks
     Calf
       Sublabels: []
     '
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`lbldefs` — Signal label definitions

`signalLabelDefinition` object | vector of `signalLabelDefinition` objects

Signal label definitions, specified as a `signalLabelDefinition` object or a vector of `signalLabelDefinition` objects.

Example:
`signalLabelDefinition("Asleep",'LabelType','roi','LabelDataType','logical')`
can label a region of a signal in which a patient is asleep.

`lblname` — Label name
character vector | string scalar

Label name, specified as a character vector or a string scalar.

Example: `signalLabelDefinition("Asleep",'LabelType','roi')` specifies a label of name `"Asleep"` for a region of a signal in which a patient is asleep during a clinical trial.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

addMembers

Add members to labeled signal set

Syntax

```
addMembers(lss,src)
addMembers(lss,src,tinfo)
addMembers(lss,src,tinfo,mnames)
```

Description

`addMembers(lss,src)` adds members to the labeled signal set `lss` from the input data source `src`.

`addMembers(lss,src,tinfo)` sets the time information for the new members to `tinfo`.

`addMembers(lss,src,tinfo,mnames)` sets the names of the new members to `mnames`. The length of `mnames` must be equal to the number of new members.

Examples

Add Member to Labeled Signal Set

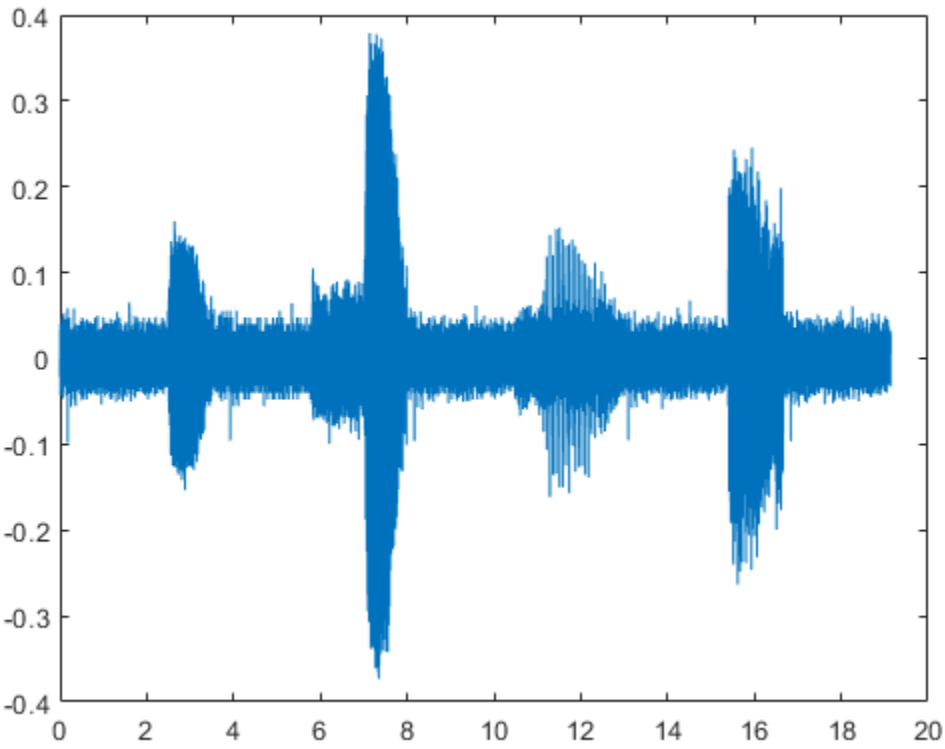
Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

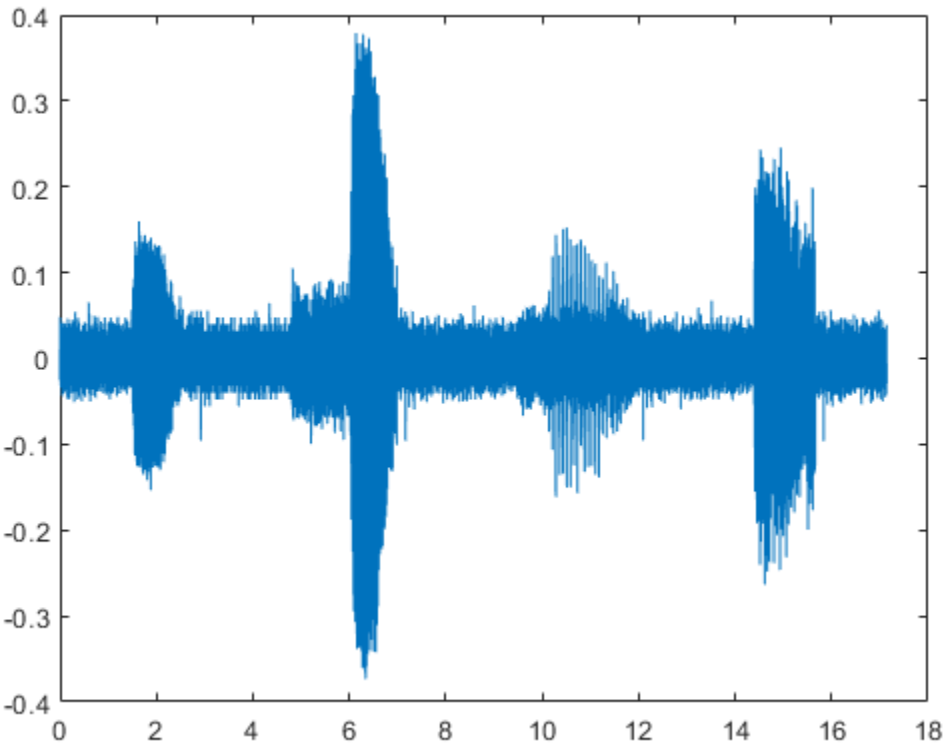
Retrieve the second member of the set and plot it.

```
[song,tinfo] = getSignal(lss,2);

t = (0:length(song)-1)/tinfo.SampleRate;

plot(t,song)
```



Remove the first and last seconds of the retrieved signal.

```
song2 = song(t>1 & t<t(end)-1);
t2 = (0:length(song2)-1)/tinfo.SampleRate;

plot(t2,song2)
```



Add the shorter signal as a new member of the labeled set.

```
addMembers(lss,song2)
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {3x1 cell}
        NumMembers: 3
    TimeInformation: "sampleRate"
        SampleRate: 4000
            Labels: [3x3 table]
       Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Flip the shorter signal upside-down and add it as a new member of the labeled set. Specify that the new member is sampled at 1 kHz.

```
addMembers(lss,flipud(song2),1000)
lss.SampleRate
```

```
ans = 4×1

        4000
```

```
4000
4000
1000
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`src` — Input data source

matrix | cell array | timetable | `signalDatastore` object | `audioDatastore` object

Input data source, specified as a matrix, a cell array, a timetable, a `signalDatastore` object, or an `audioDatastore` object. The particular form of `src` depends on the "Source" on page 1-0 property of `lss`.

- If "Source" on page 1-0 is a cell array of matrices:
  - Specify `src` as a matrix to add one member to the set.
  - Specify `src` as a cell array of matrices to add multiple members to the set.
- If "Source" on page 1-0 is a cell array containing cell arrays of vectors:
  - Specify `src` as a cell array of vectors to add one member to the set.
  - Specify `src` as a cell array containing cell arrays of vectors to add multiple members to the set.
- If "Source" on page 1-0 is a cell array of timetables:
  - Specify `src` as a timetable to add one member to the set.
  - Specify `src` as a cell array of timetables to add multiple members to the set.
- If "Source" on page 1-0 is a datastore, then add members by setting `src` as another datastore that points to new files.

Example: `{randn(10,3),randn(17,9)}` specifies two members. The first member contains three 10-sample signals. The second member contains nine 17-sample signals.

Example: `{{randn(10,1)},{randn(17,1),randn(27,1)}}` specifies two members. The first member contains one 10-sample signal. The second member contains a 17-sample signal and a 27-sample signal.

Example: `{{timetable(seconds(1:10)',randn(10,3)),timetable(seconds(1:7)',randn(7,2))}, {timetable(seconds(1:3)',randn(3,1))}}` specifies two members. The first member contains three signals sampled at 1 Hz for 10 seconds and two signals sampled at 1 Hz for 7 seconds. The second member contains one signal sampled at 1 Hz for 3 seconds.

Example: signalDatastore Object Pointing to Files

Specify the path to a set of sample sound signals included as MAT-files with MATLAB®. Each file contains a signal variable and a sample rate. List the names of the files.

```
folder = fullfile(matlabroot,"toolbox","matlab","audiovideo");
lst = dir(append(folder,"/*.mat"));
nms = {lst(:).name}'
```

```
nms = 7x1 cell
    {'chirp.mat'   }
    {'gong.mat'    }
    {'handel.mat'  }
    {'laughter.mat'}
    {'mtlb.mat'    }
    {'splat.mat'   }
    {'train.mat'   }
```

Create a signal datastore that points to the specified folder. Set the sample rate variable name to `Fs`, which is common to all files. Generate a subset of the datastore that excludes the file `mtlb.mat`, which differs from the other files in that the signal variable is not called `y`.

```
sds = signalDatastore(folder,"SampleRateVariableName","Fs");
sdss = subset(sds,~strcmp(nms,"mtlb.mat"));
```

Use the subset datastore as the source for a `labeledSignalSet` object.

```
lss = labeledSignalSet(sdss)
```

```
lss =
  labeledSignalSet with properties:

             Source: [1x1 signalDatastore]
         NumMembers: 6
    TimeInformation: "inherent"
             Labels: [6x0 table]
        Description: ""

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

`tinfo` — Time information for new members

scalar | vector | matrix | `duration` scalar | `duration` vector

Time information for new members, specified as a scalar, a vector, a matrix, a duration scalar, or a duration vector. This argument is valid only if the "TimeInformation" on page 1-0 property of `lss` is `'sampleRate'`, `'sampleTime'`, or `'timeValues'`.

- If "TimeInformation" on page 1-0 is `'sampleRate'`, then `tinfo` specifies sample rate values.
- If "TimeInformation" on page 1-0 is `'sampleTime'`, then `tinfo` specifies sample time values.
- If "TimeInformation" on page 1-0 is `'timeValues'`, then `tinfo` specifies time values.

If you add multiple members to a set, then specifying only one value of `tinfo` sets the same value for all members. If you want to specify a different value for each new member, then set `tinfo` to have multiple values.

When no source has been specified, or when the labeled signal set source is empty, you can change the "TimeInformation" on page 1-0 property to `'sampleRate'`, `'sampleTime'`, or `'timeValues'` to make `lss` interpret `tinfo` correctly.

Example: `addMembers(ks,{randn(10,5),randn(10,3)},seconds([1 2]))` adds two new members with different time information to `ks = labeledSignalSet(randn(10,3),'SampleTime',seconds(1))`.

Example: `addMembers(ks,{randn(10,5),randn(10,3)},[1:10;2:2:20]')` adds two new members with different time information to `ks = labeledSignalSet(randn(10,3),'TimeValues',1:10)`.

`mnames` — Member names

character vector | string scalar | cell array of character vectors | string array

Member names, specified as a character vector, a string scalar, a cell array of character vectors, or a string array.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},'MemberNames',{'llama' 'alpaca'})` specifies a set of random signals with two members, `'llama'` and `'alpaca'`.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

concatenate

Concatenate two or more labeled signal sets

Syntax

```
lssnew = concatenate(lss1,...,lssN)
```

Description

`lssnew = concatenate(lss1,...,lssN)` concatenates *N* labeled signal set objects, `lss1,...,lssN`, and returns a labeled signal set `lssnew` containing all the members and label values of the input sets.

Examples

Concatenate Labeled Signal Sets

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Create a new signal set with the same data source, time information, and labels as `lss`.

```
newlss = copy(lss)
```

```
newlss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
```

```
 Use setLabelValue to add data to the set.
```

Concatenate the two signal sets.

```
lssconcat = concatenate(lss,newlss)
```

```
lssconcat =
  labeledSignalSet with properties:

            Source: {4x1 cell}
        NumMembers: 4
    TimeInformation: "sampleRate"
        SampleRate: 4000
            Labels: [4x3 table]
       Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Input Arguments

`lss1,...,lssN` — Input labeled signal sets

`labeledSignalSet` objects

Input labeled signal sets, specified as `labeledSignalSet` objects. All input sets must have the same time information settings, label definitions, and data source type.

Output Arguments

`lssnew` — Concatenated labeled signal set

`labeledSignalSet` object

Concatenated labeled signal set, returned as a `labeledSignalSet` object. The set `lssnew` contains a signal source, label definitions, and label values that are independent of those in the input labeled signal sets. Changing any of the input labeled signal sets does not affect the concatenated labeled signal set. Changing the concatenated labeled signal set does not affect the input label signal sets.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

countLabelValues

Count label values

Syntax

```
cnt = countLabelValues(lss,lblname)
```

Description

`cnt = countLabelValues(lss,lblname)` counts the values of the label named `lblname` and returns results in table `cnt`. `cnt` contains label value counts and percentages. When `lblname` is an ROI or point label, `cnt` also contains the number of members with at least one value of a particular category. `countLabelValues` does not support:

- Sublabels
- Label definitions with the LabelDataType property set to `'table'` or `'timetable'`
- Labels with instance values that cannot be converted to a vector with a discrete set of categories. It must be possible to group label values using a set of unique discrete categories. Examples of labels that are not supported include:
  - Cell arrays of timetables
  - Cell arrays containing matrices of different sizes

Examples

Count Label Values

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Get the names of the labels in the set.

```
getLabelNames(lss)
```

```
ans = 3x1 string
    "WhaleType"
    "MoanRegions"
    "TrillRegions"
```

Verify that the two members of the set are blue whales.

```
countLabelValues(lss,"WhaleType")
```

```
ans=3×3 table
    WhaleType    Count    Percent
    _________    _____    _______

    blue           2        100
    humpback       0          0
    white          0          0
```

Verify that each member has three moan regions.

```
countLabelValues(lss,"MoanRegions")
```

```
ans=2×4 table
    MoanRegions    Count    Percent    MemberCount
    ___________    _____    _______    ___________

       false         0          0           0
       true          6        100           2
```

Verify that each member has one trill region.

```
countLabelValues(lss,"TrillRegions")
```

```
ans=2×4 table
    TrillRegions    Count    Percent    MemberCount
    ____________    _____    _______    ___________

       false          0          0           0
       true           2        100           2
```

Count Label Values and Create Datastores

Specify the path to a set of audio signals included as MAT-files with MATLAB®. Each file contains a signal variable and a sample rate. List the names of the files.

```
folder = fullfile(matlabroot,"toolbox","matlab","audiovideo");
lst = dir(append(folder,"/*.mat"));
nms = {lst(:).name}'
```

```
nms = 7x1 cell
    {'chirp.mat'   }
    {'gong.mat'    }
    {'handel.mat'  }
```

```
    {'laughter.mat'}
    {'mtlb.mat'    }
    {'splat.mat'   }
    {'train.mat'   }
```

Create a signal datastore that points to the specified folder. Set the sample rate variable name to `Fs`, which is common to all files. Generate a subset of the datastore that excludes the file `mtlb.mat`. Use the subset datastore as the source for a `labeledSignalSet` object.

```
sds = signalDatastore(folder,"SampleRateVariableName","Fs");
sds = subset(sds,~strcmp(nms,"mtlb.mat"));
lss = labeledSignalSet(sds);
```

Create three label definitions to label the signals:

- Define a logical attribute label that is true for signals that contain human voices.
- Define a numeric point label that marks the location and amplitude of the maximum of each signal.
- Define a categorical region-of-interest (ROI) label to pick out nonoverlapping, uniform-length random regions of each signal.

Add the signal label definitions to the labeled signal set.

```
vc = signalLabelDefinition("Voice",'LabelType','attribute', ...
    'LabelDataType','logical','DefaultValue',false);
mx = signalLabelDefinition("Maximum",'LabelType','point', ...
    'LabelDataType','numeric');
rs = signalLabelDefinition("RanROI",'LabelType','ROI', ...
    'LabelDataType','categorical','Categories',["ROI" "other"]);
addLabelDefinitions(lss,[vc mx rs])
```

Label the signals:

- Label `'handel.mat'` and `'laughter.mat'` as having human voices.
- Use the `islocalmax` function to find the maximum of each signal. Label its location and value.
- Use the `randROI` on page 1-0 function to generate as many regions of length $N/10$ samples as can fit in a signal of length N given a minimum separation of $N/6$ samples between regions. Label their locations and assign them to the `ROI` category.

When labeling points and regions, convert sample values to time values. Subtract 1 to account for MATLAB® array indexing and divide by the sample rate.

```
kj = 1;
while hasdata(sds)

    [sig,info] = read(sds);
    fs = info.SampleRate;

    [~,fn] = fileparts(info.FileName);
    if fn=="handel" || fn=="laughter"
        setLabelValue(lss,kj,"Voice",true)
    end

    xm = find(islocalmax(sig,'MaxNumExtrema',1));
    setLabelValue(lss,kj,"Maximum",(xm-1)/fs,sig(xm))
```

```
    N = length(sig);
    rois = randROI(N,round(N/10),round(N/6));
    setLabelValue(lss,kj,"RanROI",(rois-1)/fs,repelem("ROI",size(rois,1)))

    kj = kj+1;

end
```

Verify that only two signals contain voices.

```
countLabelValues(lss,"Voice")
```

```
ans=2×3 table
    Voice    Count    Percent
    _____    _____    _______

    false      4      66.667
    true       2      33.333
```

Verify that two signals have a maximum amplitude of 1.

```
countLabelValues(lss,"Maximum")
```

```
ans=5×4 table
           Maximum            Count    Percent    MemberCount
    ______________________    _____    _______    ___________

    0.80000000000000004441      1      16.667          1
    0.89113331915798421612      1      16.667          1
    0.94730769230769229505      1      16.667          1
    1                           2      33.333          2
    1.0575668990330560071       1      16.667          1
```

Verify that each signal has four nonoverlapping random regions of interest.

```
countLabelValues(lss,"RanROI")
```

```
ans=2×4 table
    RanROI    Count    Percent    MemberCount
    ______    _____    _______    ___________

    ROI        24        100           6
    other       0          0           0
```

Create two datastores with the data in the labeled signal set:

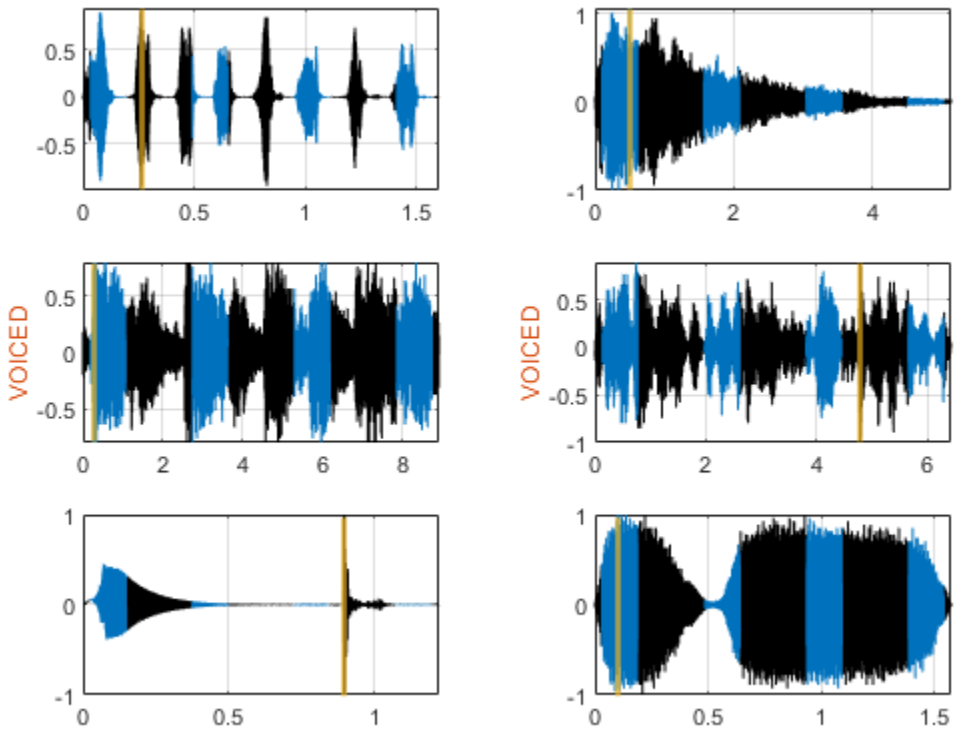
- The `signalDatastore` object `sd` contains the signal data.
- The `arrayDatastore` object `ld` contains the labeling information. Specify that you want to include the information corresponding to all the labels you created.

```
[sd,ld] = createDatastores(lss,["Voice" "RanROI" "Maximum"]);
```

Use the information in the datastores to plot the signals and display their labels.

- Use a `signalMask` object to highlight the regions of interest in blue.
- Plot yellow lines to mark the locations of the maxima.
- Add a red axis label to the signals that contain human voices.

```
tiledlayout flow

while hasdata(sd)

    [sg,nf] = read(sd);

    lbls = read(ld);

    nexttile

    msk = signalMask(lbls{:}.RanROI{:},'SampleRate',nf.SampleRate);
    plotsigroi(msk,sg)
    colorbar off
    xlabel('')

    xline(lbls{:}.Maximum{:}.Location, ...
        'LineWidth',2,'Color','#EDB120')

    if lbls{:}.Voice{:}
        ylabel('VOICED','Color','#D95319')
    end

end
```



```
function roilims = randROI(N,wid,sep)

num = floor((N+sep)/(wid+sep));
hq = histcounts(randi(num+1,1,N-num*wid-(num-1)*sep),(1:num+2)-1/2);
roilims = (1 + (0:num-1)*(wid+sep) + cumsum(hq(1:num)))' + [0 wid-1];

end
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`lblname` — Label name

character vector | string scalar

Label name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Output Arguments

`cnt` — Results table

table

Results table, returned as a table with the following variables:

- `Count` — Number of label values for a particular category.
- `Percent` — Number of label values for a particular category as a percentage of all label values.
- `MemberCount` — Number of members with at least one value of a particular category. This variable is returned only for an ROI or a point label.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2021a

createDatastores

Create datastores pointing to signal and label data

Syntax

```
[sigdata,lbldata] = createDatastores(lss,lblnames)
```

Description

`[sigdata,lbldata] = createDatastores(lss,lblnames)` creates a datastore, `sigdata`, containing signal member data, and a datastore, `lbldata`, containing label data from labels specified in the string array `lblnames`. `createDatastores` does not apply to sublabels. Set `lblnames` to one or more parent label names to get the parent labels and the corresponding sublabel values.

Examples

Create Datastores

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
        NumMembers: 2
    TimeInformation: "sampleRate"
        SampleRate: 4000
            Labels: [2x3 table]
       Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Display the labels for the first member of the set.

```
lss.Labels(1,:)
```

```
ans=1×3 table
                WhaleType    MoanRegions    TrillRegions
                _________    ___________    ____________

    Member{1}      blue       {3x2 table}    {1x3 table}
```

Get the names of the labels in the set. Create a signal datastore with the signal information and an array datastore with the label information.

```
lbls = getLabelNames(lss);
[sgd,lbd] = createDatastores(lss,lbls)

sgd =
  signalDatastore with properties:

    MemberNames:{
                'Member{1}';
                'Member{2}'
                }
       Members: {2x1 cell}
      ReadSize: 1
    SampleRate: 4000


lbd =
  ArrayDatastore with properties:

              ReadSize: 1
    IterationDimension: 1
            OutputType: "cell"
```

Display the labels for the first member of the set.

```
lbls = read(lbd);
lbls{:}

ans=1×3 table
    WhaleType    MoanRegions    TrillRegions
    _________    ___________    ____________

      blue       {3x2 table}    {1x3 table}
```

Count Label Values and Create Datastores

Specify the path to a set of audio signals included as MAT-files with MATLAB®. Each file contains a signal variable and a sample rate. List the names of the files.

```
folder = fullfile(matlabroot,"toolbox","matlab","audiovideo");
lst = dir(append(folder,"/*.mat"));
nms = {lst(:).name}'

nms = 7x1 cell
    {'chirp.mat'   }
    {'gong.mat'    }
    {'handel.mat'  }
    {'laughter.mat'}
    {'mtlb.mat'    }
    {'splat.mat'   }
    {'train.mat'   }
```

Create a signal datastore that points to the specified folder. Set the sample rate variable name to `Fs`, which is common to all files. Generate a subset of the datastore that excludes the file `mtlb.mat`. Use the subset datastore as the source for a `labeledSignalSet` object.

```
sds = signalDatastore(folder,"SampleRateVariableName","Fs");
sds = subset(sds,~strcmp(nms,"mtlb.mat"));
lss = labeledSignalSet(sds);
```

Create three label definitions to label the signals:

- Define a logical attribute label that is true for signals that contain human voices.
- Define a numeric point label that marks the location and amplitude of the maximum of each signal.
- Define a categorical region-of-interest (ROI) label to pick out nonoverlapping, uniform-length random regions of each signal.

Add the signal label definitions to the labeled signal set.

```
vc = signalLabelDefinition("Voice",'LabelType','attribute', ...
    'LabelDataType','logical','DefaultValue',false);
mx = signalLabelDefinition("Maximum",'LabelType','point', ...
    'LabelDataType','numeric');
rs = signalLabelDefinition("RanROI",'LabelType','ROI', ...
    'LabelDataType','categorical','Categories',["ROI" "other"]);
addLabelDefinitions(lss,[vc mx rs])
```

Label the signals:

- Label `'handel.mat'` and `'laughter.mat'` as having human voices.
- Use the `islocalmax` function to find the maximum of each signal. Label its location and value.
- Use the `randROI` on page 1-0 function to generate as many regions of length $N/10$ samples as can fit in a signal of length N given a minimum separation of $N/6$ samples between regions. Label their locations and assign them to the `ROI` category.

When labeling points and regions, convert sample values to time values. Subtract 1 to account for MATLAB® array indexing and divide by the sample rate.

```
kj = 1;
while hasdata(sds)

    [sig,info] = read(sds);
    fs = info.SampleRate;

    [~,fn] = fileparts(info.FileName);
    if fn=="handel" || fn=="laughter"
        setLabelValue(lss,kj,"Voice",true)
    end

    xm = find(islocalmax(sig,'MaxNumExtrema',1));
    setLabelValue(lss,kj,"Maximum",(xm-1)/fs,sig(xm))

    N = length(sig);
    rois = randROI(N,round(N/10),round(N/6));
    setLabelValue(lss,kj,"RanROI",(rois-1)/fs,repelem("ROI",size(rois,1)))

    kj = kj+1;
```

```
end
```

Verify that only two signals contain voices.

```
countLabelValues(lss,"Voice")
```

```
ans=2×3 table
    Voice    Count    Percent
    _____    _____    _______

    false      4      66.667
    true       2      33.333
```

Verify that two signals have a maximum amplitude of 1.

```
countLabelValues(lss,"Maximum")
```

```
ans=5×4 table
           Maximum            Count    Percent    MemberCount
    ______________________    _____    _______    ___________

    0.80000000000000004441      1      16.667          1
    0.89113331915798421612      1      16.667          1
    0.94730769230769229505      1      16.667          1
    1                           2      33.333          2
    1.0575668990330560071       1      16.667          1
```

Verify that each signal has four nonoverlapping random regions of interest.

```
countLabelValues(lss,"RanROI")
```

```
ans=2×4 table
    RanROI    Count    Percent    MemberCount
    ______    _____    _______    ___________

    ROI        24        100           6
    other       0          0           0
```

Create two datastores with the data in the labeled signal set:

- The `signalDatastore` object `sd` contains the signal data.
- The `arrayDatastore` object `ld` contains the labeling information. Specify that you want to include the information corresponding to all the labels you created.

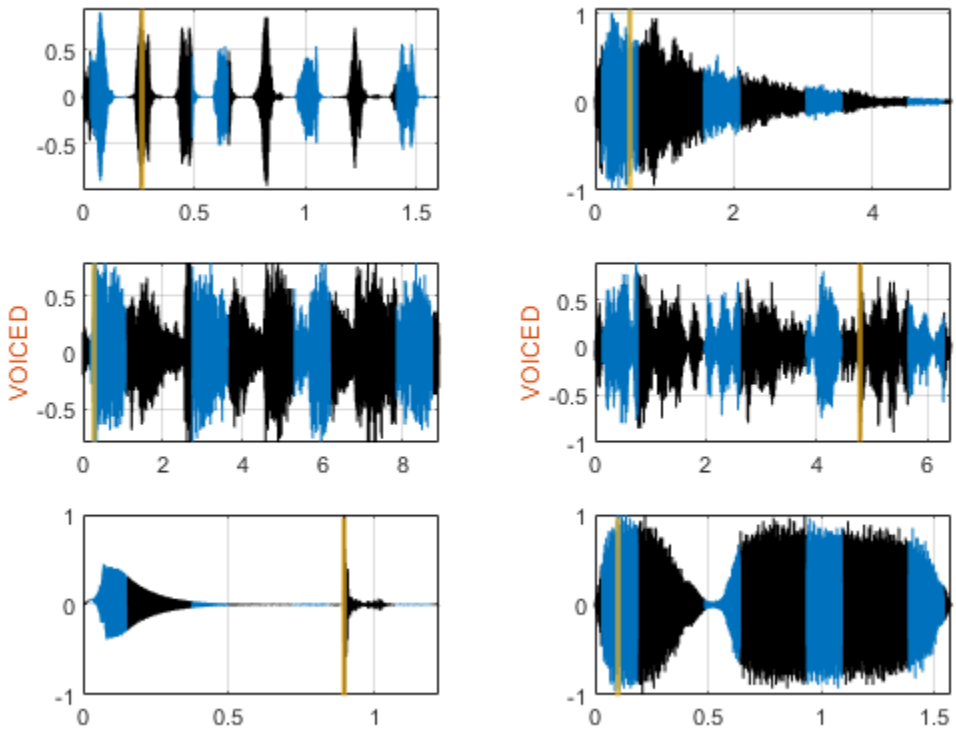
```
[sd,ld] = createDatastores(lss,["Voice" "RanROI" "Maximum"]);
```

Use the information in the datastores to plot the signals and display their labels.

- Use a `signalMask` object to highlight the regions of interest in blue.
- Plot yellow lines to mark the locations of the maxima.
- Add a red axis label to the signals that contain human voices.

```
tiledlayout flow
```

```
while hasdata(sd)

    [sg,nf] = read(sd);

    lbls = read(ld);

    nexttile

    msk = signalMask(lbls{:}.RanROI{:},'SampleRate',nf.SampleRate);
    plotsigroi(msk,sg)
    colorbar off
    xlabel('')

    xline(lbls{:}.Maximum{:}.Location, ...
        'LineWidth',2,'Color','#EDB120')

    if lbls{:}.Voice{:}
        ylabel('VOICED','Color','#D95319')
    end

end
```




```
function roilims = randROI(N,wid,sep)

num = floor((N+sep)/(wid+sep));
hq = histcounts(randi(num+1,1,N-num*wid-(num-1)*sep),(1:num+2)-1/2);
roilims = (1 + (0:num-1)*(wid+sep) + cumsum(hq(1:num)))' + [0 wid-1];
```

```
end
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`lblnames` — Label names

character vector | string scalar | cell array of character vectors | string array

Label names, specified as a character vector, a string scalar, a cell array of character vectors, or a string array.

Data Types: `char` | `string`

Output Arguments

`sigdata` — Signal data

`signalDatastore` object | `audioDatastore` object

Signal data, returned as a `signalDatastore` object or an `audioDatastore` object.

`lbldata` — Label data

`arrayDatastore` object

Label data, returned as an `arrayDatastore` object.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2021a

createFeatureData

Create feature table or matrix and response vectors

Syntax

```
ftdata = createFeatureData(lss)
[ftdata,respdata] = createFeatureData(lss,Name=Value)
```

Description

`ftdata = createFeatureData(lss)` creates a table of feature data `ftdata` with features corresponding to all the attribute and region-of-interest (ROI) feature labels in the input labeled signal set.

`[ftdata,respdata] = createFeatureData(lss,Name=Value)` returns a table of responses `respdata` where each variable corresponds to the data for the specified response label. You can specify a list of label names that the function adds as responses to `respdata` and other optional inputs as name-value arguments. For example, `Responses="Species"` specifies `Species` as a response label.

Note The label data for all specified label names must be vertically and horizontally concatenable.

Examples

Create Feature and Response Tables from Label Names in Labeled Signal Set

Create a set of label definitions.

- Define three numeric attribute feature labels that correspond to mean frequency, band power, and peak amplitude.
- Define two numeric region-of-interest (ROI) feature labels that correspond to mean and signal-to-noise ratio (SNR).
- Define one categorical attribute label that has these categories: `A`, `B`, and `C`.

Create a labeled signal set that contains all six label definitions.

```
ld1 = signalLabelDefinition("MeanFrequency",LabelType="attributeFeature",LabelDataType="numeric")
ld2 = signalLabelDefinition("BandPower",LabelType="attributeFeature",LabelDataType="numeric");
ld3 = signalLabelDefinition("PeakAmplitude",LabelType="attributeFeature",LabelDataType="numeric")

ld4 = signalLabelDefinition("Mean",LabelType="roiFeature",LabelDataType="numeric", ...
      FrameSize=500,FrameOverlapLength=250);
ld5 = signalLabelDefinition("SNR",LabelType="roiFeature",LabelDataType="numeric", ...
      FrameSize=500,FrameOverlapLength=250);

catValues = ["A" "B" "C"];
ld6 = signalLabelDefinition("Class",LabelType="attribute",LabelDataType="categorical", ...
      Categories=catValues);
```

```
lss = labeledSignalSet([],[ld1,ld2,ld3,ld4,ld5,ld6]);
```

Create a `signalFrequencyFeatureExtractor` object to extract the mean frequency, band power, and peak amplitude values across an entire signal. Create a `signalTimeFeatureExtractor` object to extract the mean and SNR values from signal frames that are 500 samples long and have 250 samples of overlap. Set the output format of each extractor object to a table.

```
H = signalFrequencyFeatureExtractor();
H.FeatureFormat = "table";
H.MeanFrequency = true;
H.BandPower = true;
H.PeakAmplitude= true;
H.setExtractorParameters("PeakAmplitude",MaxNumExtrema=5)

G = signalTimeFeatureExtractor();
G.FeatureFormat = "table";
G.FrameSize = 500;
G.FrameOverlapLength = 250;
G.Mean = true;
G.SNR = true;
```

Create a signal, `x`, made up of several sinusoids with varying frequencies. For ten iterations, add random noise to `x` to generate `xn`, and add this new signal as a member to `lss`. Extract the specified features from `xn` and set the label values in the labeled signal set equal to the values of the extracted features. Display the labels contained in the labeled signal set.

```
t = (1:1000)'/1e3;
x = sin(2*pi*10*t) + sin(2*pi*100*t) + sin(2*pi*200*t) + sin(2*pi*350*t) + sin(2*pi*450*t);

for idx = 1:10

    xn = x + randn(size(t));
    p = extract(H,xn);

    addMembers(lss,xn);

    setLabelValue(lss,idx,"MeanFrequency",p.MeanFrequency);
    setLabelValue(lss,idx,"BandPower",p.BandPower);
    setLabelValue(lss,idx,"PeakAmplitude",p.PeakAmplitude);

    q = extract(G,xn);
    rois = q{:,[1,2]};
    setLabelValue(lss,idx,"Mean",rois,q.Mean);
    setLabelValue(lss,idx,"SNR",rois,q.SNR);

    setLabelValue(lss,idx,"Class",catValues(randi(3,[1 1])));
end

LBLS = lss.Labels
```

```
LBLS=10×6 table
                MeanFrequency    BandPower                    PeakAmplitude
                _____________    _________    ____________________________________________

    Member{1}      {[1.4789]}      {[3.4261]}    {[ 9.0290 11.1191 11.4532 12.0216 13.2064]}
    Member{2}      {[1.4040]}      {[3.4643]}    {[ 11.3509 9.5855 12.0983 11.9682 12.5628]}
    Member{3}      {[1.4642]}      {[3.5901]}    {[10.9591 10.3477 14.7581 11.1874 12.6932]}
```

```
    Member{4}      {[1.3684]}      {[3.5572]}    {[ 12.9701 11.7092 13.0650 9.5547 12.5720]}
    Member{5}      {[1.4672]}      {[3.4403]}    {[ 10.7361 9.3758 13.8124 10.7749 13.3737]}
    Member{6}      {[1.4241]}      {[3.5225]}    {[10.3302 11.5979 11.6696 11.2745 12.2777]}
    Member{7}      {[1.4070]}      {[3.3863]}    {[ 10.5294 11.4855 12.5187 9.8425 12.6257]}
    Member{8}      {[1.4529]}      {[3.4818]}    {[ 9.8225 10.6672 13.3820 10.7357 12.6096]}
    Member{9}      {[1.4181]}      {[3.5552]}    {[12.4350 11.5933 12.0069 11.1148 12.9654]}
    Member{10}     {[1.4198]}      {[3.5516]}    {[11.3011 10.5433 13.6415 11.5888 13.1569]}
```

Create a table that contains feature data corresponding to all feature labels in `lss`. The `Class` data is not included in the table because `Class` is not a feature label.

```
FTs = createFeatureData(lss,ConvertFeaturesToRows=true)
```

```
FTs=10×5 table
    MeanFrequency    BandPower                    PeakAmplitude
    _____________    _________    ____________________________________________    ____________

       1.4789         3.4261       9.029    11.119    11.453    12.022    13.206    -0.021921
        1.404         3.4643      11.351    9.5855    12.098    11.968    12.563     0.058634
       1.4642         3.5901      10.959    10.348    14.758    11.187    12.693     0.023716
       1.3684         3.5572       12.97    11.709    13.065    9.5547    12.572      0.04709
       1.4672         3.4403      10.736    9.3758    13.812    10.775    13.374     0.014723
       1.4241         3.5225       10.33    11.598     11.67    11.274    12.278    -0.032001
        1.407         3.3863      10.529    11.486    12.519    9.8425    12.626     0.016776
       1.4529         3.4818      9.8225    10.667    13.382    10.736     12.61    -0.003218
       1.4181         3.5552      12.435    11.593    12.007    11.115    12.965    -0.092008
       1.4198         3.5516      11.301    10.543    13.641    11.589    13.157     0.066601
```

Create another table containing feature data that corresponds only to attribute feature labels. Use the `getLabelNames` function to specify label names corresponding to attribute features. Specify `Class` as a response label to obtain a second output table that contains the category each member belongs to.

```
[attFTs,R] = createFeatureData(lss,Features=getLabelNames(lss,LabelType="attributeFeature"), ...
             ConvertFeaturesToRows=true,Responses="Class")
```

```
attFTs=10×3 table
    MeanFrequency    BandPower                    PeakAmplitude
    _____________    _________    ____________________________________________

       1.4789         3.4261       9.029    11.119    11.453    12.022    13.206
        1.404         3.4643      11.351    9.5855    12.098    11.968    12.563
       1.4642         3.5901      10.959    10.348    14.758    11.187    12.693
       1.3684         3.5572       12.97    11.709    13.065    9.5547    12.572
       1.4672         3.4403      10.736    9.3758    13.812    10.775    13.374
       1.4241         3.5225       10.33    11.598     11.67    11.274    12.278
        1.407         3.3863      10.529    11.486    12.519    9.8425    12.626
       1.4529         3.4818      9.8225    10.667    13.382    10.736     12.61
       1.4181         3.5552      12.435    11.593    12.007    11.115    12.965
       1.4198         3.5516      11.301    10.543    13.641    11.589    13.157


R=10×1 table
    Class
    _____
```

```
      C
      C
      A
      B
      A
      C
      A
      B
      B
      C
```

You can use the `getLabelIndices` function to specify indices for labels you want to include in the feature table. Create a table containing feature data that corresponds only to ROI feature labels. Use the `ExpandResponseLabels` argument to repeat response labels for each member.

```
[roiFTs,R] = createFeatureData(lss,Features=getLabelIndices(lss,LabelType="roiFeature"), ...
             Responses="Class",ExpandResponseLabels=true)
```

```
roiFTs=30×2 table
      Mean          SNR
    _________    _______

    -0.021921    -4.7744
     -0.04995    -6.8315
    -0.043343    -5.7233
     0.058634    -6.6994
     0.037199    -4.8069
     0.014408     -6.049
     0.023716    -4.4364
     0.022439    -5.5243
     0.052243    -5.9047
      0.04709    -5.2615
      0.04225    -5.9997
    0.0099387    -4.9845
     0.014723    -4.5713
     0.023537    -5.9291
     0.044529    -5.9603
    -0.032001    -5.1732
      ⋮


R=30×1 table
    Class
    _____

      C
      C
      C
      C
      C
      C
      A
      A
      A
      B
      B
      B
```

```
      A
      A
      A
      C
      ⋮
```

Create Feature Matrix from Label Definitions in Labeled Signal Set

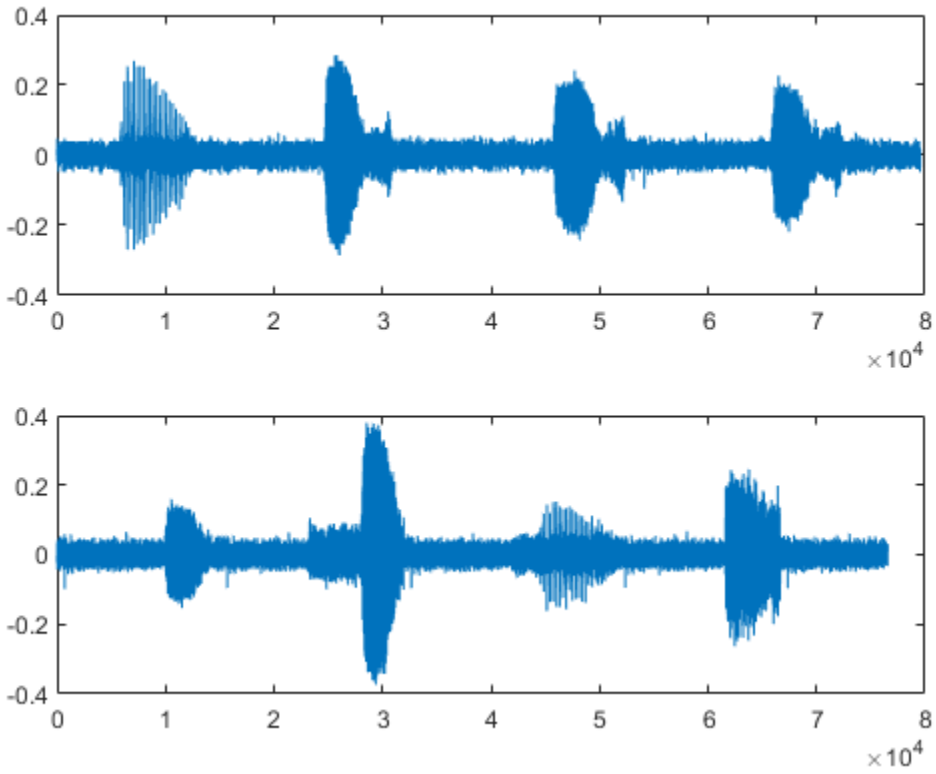
Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
        NumMembers: 2
    TimeInformation: "sampleRate"
        SampleRate: 4000
            Labels: [2x3 table]
       Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Retrieve and plot the signals contained in `lss`. Each whale song consists of four sounds.

```
for k = 1:lss.NumMembers
    [s,info] = getSignal(lss,k);
    fs = lss.SampleRate;
    t = 0:length(s)-1/fs;
    subplot(2,1,k)
    plot(t,s)
end
```



Add numeric region-of-interest (ROI) feature label definitions to `lss` that you will use to specify the shape factor and peak values of each signal. Specify a frame size equal to 25% of the signal length, so that each region corresponds to a different sound.

Create a `signalTimeFeatureExtractor` object to extract the shape factor and peak value of each ROI. Set the `roiFeature` label values in the labeled signal set to the values of the extracted features.

```
sld1 = signalLabelDefinition("ShapeROI",LabelType="roiFeature", ...
      LabelDataType="numeric",FrameSize=round(0.25*length(s))-1);

sld2 = signalLabelDefinition("PeakROI",LabelType="roiFeature", ...
      LabelDataType="numeric",FrameSize=round(0.25*length(s))-1);

addLabelDefinitions(lss,[sld1 sld2]);

sFE = signalTimeFeatureExtractor(SampleRate=fs,FrameSize=round(0.25*length(s))-1);
sFE.FeatureFormat = "table";
sFE.ShapeFactor = true;
sFE.PeakValue = true;

for i = 1:lss.NumMembers
    sig = getSignal(lss,i);
    fts = extract(sFE,sig);

    setLabelValue(lss,i,"ShapeROI",[fts.FrameStartTime fts.FrameEndTime],fts.ShapeFactor);
```

```
    setLabelValue(lss,i,"PeakROI",[fts.FrameStartTime fts.FrameEndTime],fts.PeakValue);
end
```

Use the `getLabelDefinitions` function to obtain the label names corresponding to `roiFeature` labels in `lss`. Create a matrix of data that corresponds to the ROI feature labels in the labeled signal set, where each column is a feature.

```
lbldefs = getLabelDefinitions(lss,LabelType="roiFeature");
lbldefs(:).Name
```

```
ans =
"ShapeROI"

ans =
"PeakROI"
```

```
ftdata = createFeatureData(lss,Features=[lbldefs(1).Name lbldefs(2).Name],OutputFormat="matrix")
```

```
ftdata = 8×2

    1.6908    0.2694
    1.7483    0.2850
    1.6517    0.2421
    1.5930    0.2264
    1.4750    0.1599
    1.8366    0.3791
    1.3921    0.1612
    1.6176    0.2633
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object. To obtain a list of feature label definitions available in `lss`, use the `getLabelDefinitions` or `getLabelNames` functions and specify `"attributeFeature"` or `"roiFeature"` as the label type.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `createFeatureData(lss,Features=getLabelDefinitions(lss,LabelType="roiFeature",FrameSize=60),OutputFormat="matrix")` returns a matrix of features corresponding to region-of-interest (ROI) feature labels in the input labeled signal set with the frame size of 60.

`Features` — List of label names corresponding to features

string vector | numeric vector

List of label names corresponding to features, specified as a string vector containing label names or a numeric vector containing indices pointing to the label definitions in `lss`. The function adds the specified label names as features to `ftdata`. You can get a list of all label definitions, label names, or

label definition indices in `lss` using `getLabelDefinitions`, `getLabelNames`, and `getLabelIndices`, respectively. If you do not specify `Features`, then the function uses the list of attribute and ROI feature labels available in `lss`.

Example: `Features=getLabelNames(lss,LabelType="point")` specifies the label names corresponding to point labels in the input labeled signal set.

Data Types: `double` | `logical` | `string` | `categorical`

`Responses` — List of label names corresponding to responses
string vector | numeric vector

List of label names corresponding to responses, specified as a string vector containing label names or a numeric vector containing indices pointing to the label definitions in `lss`. The function adds the specified label names as responses to `respdata`. You cannot specify feature labels as responses. If you do not specify `Responses`, then `respdata` is empty.

Example: `Responses=[1 5 7 12]` specifies responses corresponding to label definitions at the provided indices in the input labeled signal set.

Note `Features` and `Responses` must contain different label names or indices.

Data Types: `double` | `logical` | `string` | `categorical`

`OutputFormat` — Output data format
`"table"` (default) | `"matrix"`

Output data format, specified as `"table"` or `"matrix"`. If you specify `OutputFormat` as `"table"`, then the function returns `ftdata` and `respdata` as tables. If you specify `OutputFormat` as `"matrix"`, then the function returns `ftdata` and `respdata` as matrices.

Data Types: `char` | `string`

`ExpandResponseLabels` — Option to apply scalar expansion to response labels
`0` (`false`) (default) | `1` (`true`)

Option to apply scalar expansion to response labels, specified as logical scalar. This argument applies when the corresponding features contain multiple rows. For example, you can set this argument to true when you compute frame-based features and want to assign the same response label to each frame. When `ExpandResponseLabels` is set to false, the number of response label values per member must match the number of feature rows per member.

Data Types: `logical`

`ConvertFeaturesToRows` — Option to convert features containing multiple rows to a single row vector
`0` (`false`) (default) | `1` (`true`)

Option to convert features containing multiple rows to a single row vector, specified as a logical scalar. If you specify `ConvertFeaturesToRows` as true, then the function converts the features corresponding to column vectors or matrices to row vectors. Use this argument when all features do not have the same number of rows.

Data Types: `logical`

Output Arguments

`ftdata` — Feature data

table | matrix

Feature data, returned as a table or matrix. By default, the function returns `ftdata` as a table. To output the feature data as a matrix, specify `OutputFormat` as `"matrix"`.

`respdata` — Response data

table | matrix

Response data, returned as a table or matrix. By default, the function returns `respdata` as a table. To output the response data as a matrix, specify `OutputFormat` as `"matrix"`.

See Also

Objects

`labeledSignalSet` | `signalLabelDefinition`

Functions

`getLabelDefinitions` | `getLabelNames` | `getLabelIndices`

Introduced in R2022a

editLabelDefinition

Edit label definition properties

Syntax

```
editLabelDefinition(lss,lblname,propname,val)
```

Description

`editLabelDefinition(lss,lblname,propname,val)` changes the `propname` property of the label or sublabel definition `lblname` to `val`.

The function can edit only the “Name” on page 1-0 , “DefaultValue” on page 1-0 , “Tag” on page 1-0 , “Description” on page 1-0 , and “Categories” on page 1-0 properties. To change any other property of the label definition, remove the definition using `removeLabelDefinition` and add a definition with the desired property values using `addLabelDefinitions`.

- If you edit the “DefaultValue” on page 1-0 property, all existing label values remain unchanged. The new default value applies only to new members, new regions, or new points.
- You can edit the “Categories” on page 1-0 property only when the “LabelDataType” on page 1-0 of the target label or sublabel definition is `'Categorical'`.

New specified categories do not replace any existing categories. They are appended to the existing values.

Examples

Edit Label Definition

Load a labeled signal set containing recordings of whale songs. Get the names of the labels.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
        NumMembers: 2
    TimeInformation: "sampleRate"
        SampleRate: 4000
            Labels: [2x3 table]
       Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

```
getLabelNames(lss)
```

```
ans = 3x1 string
    "WhaleType"
    "MoanRegions"
    "TrillRegions"
```

The first label corresponds to the type of whale. Get the types available in the set.

```
lbldefs = getLabelDefinitions(lss);
types = lbldefs(1)
```

```
types =
  signalLabelDefinition with properties:

             Name: "WhaleType"
        LabelType: "attribute"
    LabelDataType: "categorical"
       Categories: [3x1 string]
     DefaultValue: []
        Sublabels: [0x0 signalLabelDefinition]
              Tag: ""
      Description: "Whale type"

 Use labeledSignalSet to create a labeled signal set.
```

```
types = types.Categories
```

```
types = 3x1 string
    "blue"
    "humpback"
    "white"
```

Modify the label to incorporate sperm whales and killer whales. Verify that the labeled signal set includes the two new whale types.

```
editLabelDefinition(lss,'WhaleType', ...
    'Categories',{'sperm','killer'})

lbldefs = getLabelDefinitions(lss);
types = lbldefs(1).Categories
```

```
types = 5x1 string
    "blue"
    "humpback"
    "white"
    "sperm"
    "killer"
```

The definition for trill regions has a sublabel that identifies peaks.

```
lbldefs(3).Sublabels
```

```
ans =
  signalLabelDefinition with properties:

                     Name: "TrillPeaks"
                LabelType: "point"
```

```
             LabelDataType: "numeric"
        ValidationFunction: []
    PointLocationsDataType: "double"
              DefaultValue: []
                 Sublabels: [0x0 signalLabelDefinition]
                       Tag: ""
               Description: "Trill peaks"

 Use labeledSignalSet to create a labeled signal set.
```

Change the description of the sublabel.

```
editLabelDefinition(lss,["TrillRegions" "TrillPeaks"],'Description','Peaks of trill regions')

lbldefs = getLabelDefinitions(lss);
lbldefs(3).Sublabels
```

```
ans =
  signalLabelDefinition with properties:

                     Name: "TrillPeaks"
                LabelType: "point"
            LabelDataType: "numeric"
        ValidationFunction: []
    PointLocationsDataType: "double"
              DefaultValue: []
                 Sublabels: [0x0 signalLabelDefinition]
                       Tag: ""
               Description: "Peaks of trill regions"

 Use labeledSignalSet to create a labeled signal set.
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`lblname` — Label or sublabel name

character vector | string scalar | cell array of character vectors | string array

Label or sublabel name. To specify a label, use a character vector or a string scalar. To specify a sublabel, use a two-element cell array of character vectors or a two-element string array:

- The first element is the name of the parent label.
- The second element is the name of the sublabel.

Example: `signalLabelDefinition("Asleep",'LabelType','roi')` specifies a label of name `"Asleep"` for a region of a signal in which a patient is asleep during a clinical trial.

Example: `{'Asleep' 'REM'}` or `["Asleep" "REM"]` specifies a region of a signal in which a patient undergoes REM sleep.

`propname` — Property name
`'Name'` | `'DefaultValue'` | `'Tag'` | `'Description'` | `'Categories'`

Property name, specified as `'Name'`, `'DefaultValue'`, `'Tag'`, `'Description'`, or `'Categories'`.

Data Types: `char` | `string`

`val` — Property value
numeric value | logical value | character vector | string | vector of strings | cell array of character vectors

Label values, specified as a numeric or logical value, a character vector or string, a vector of strings, or a cell array of character vectors. `val` must be of the data type specified for `propname`.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

getAlternateFileSystemRoots

Get alternate file system roots when data source of labeled signal set is a datastore

Syntax

```
altroots = getAlternateFileSystemRoots(lss)
```

Description

`altroots = getAlternateFileSystemRoots(lss)` returns the current alternate file system roots set in the underlying datastore source of the labeled signal set `lss`. The function applies only to labeled signal sets whose data source is a datastore that points to files.

Examples

Get Alternate Root Paths for Labeled Signal Set

Create a `labeledSignalSet` to store labels for electromyographic (EMG) signals and get alternate file paths for the underlying data source which you can use to access the signals on a different machine. Download the files from this location: https://ssd.mathworks.com/supportfiles/SPT/data/MyoelectricData.zip.

Use a `signalDatastore` to consume the signal files from the EMG data set. Modify `folder` to correspond to the location where you download the files. The data set is organized into subfolders, so enable the `IncludeSubfolders` flag.

```
folder = "C:\path\Data";
sds = signalDatastore(folder,IncludeSubfolders=true);
```

Create a `labeledSignalSet` object with the datastore as the input data source. To access the source data of the underlying datastore on different machines and operating systems, use `setAlternateFileSystemRoots` to set alternate file paths to the data. Replace `folder2` with your desired root path(s). You can also set alternate file system roots for the datastore before creating the labeled signal set.

```
lss = labeledSignalSet(sds);
folder2 = "E:\path2\Data";
setAlternateFileSystemRoots(lss,[folder folder2]);
```

Get the current alternate file system roots in the underlying datastore of the labeled signal set to confirm that all the file paths are included.

```
altroots = getAlternateFileSystemRoots(lss);
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object. The function applies only to labeled signal sets whose data source is a datastore that points to files.

Output Arguments

`altroots` — Alternate file system roots

string array

Alternate file system roots, returned as a string array.

See Also

Objects

`labeledSignalSet` | `signalDatastore` | `audioDatastore`

Functions

`setAlternateFileSystemRoots`

Introduced in R2022a

getLabelDefinitions

Get label definitions in labeled signal set

Syntax

```
lbldefs = getLabelDefinitions(lss)
getLabelDefinitions(lss,Name=Value)
```

Description

`lbldefs = getLabelDefinitions(lss)` returns a vector of `signalLabelDefinition` objects with the labels of the labeled signal set `lss`.

Changing `lbldefs` does not affect the labeled set. To modify label definitions, use `editLabelDefinition`, `addLabelDefinitions`, and `removeLabelDefinition`.

`getLabelDefinitions(lss,Name=Value)` returns a vector of `signalLabelDefintion` objects that have a label type and frame policy equal to the values you specify using name-value arguments.

Examples

Get Label Definitions

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
        NumMembers: 2
    TimeInformation: "sampleRate"
        SampleRate: 4000
            Labels: [2x3 table]
       Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Retrieve the definitions of the labels in the set.

```
dfs = getLabelDefinitions(lss);

for k = 1:length(dfs)
    dfs(k)
end
```

```
ans =
  signalLabelDefinition with properties:
```

```
            Name: "WhaleType"
       LabelType: "attribute"
   LabelDataType: "categorical"
      Categories: [3x1 string]
    DefaultValue: []
       Sublabels: [0x0 signalLabelDefinition]
             Tag: ""
     Description: "Whale type"

 Use labeledSignalSet to create a labeled signal set.

ans =
  signalLabelDefinition with properties:

                Name: "MoanRegions"
           LabelType: "roi"
        LabelDataType: "logical"
   ValidationFunction: []
    ROILimitsDataType: "double"
         DefaultValue: []
            Sublabels: [0x0 signalLabelDefinition]
                  Tag: ""
          Description: "Regions where moans occur"

 Use labeledSignalSet to create a labeled signal set.

ans =
  signalLabelDefinition with properties:

                Name: "TrillRegions"
           LabelType: "roi"
        LabelDataType: "logical"
   ValidationFunction: []
    ROILimitsDataType: "double"
         DefaultValue: []
            Sublabels: [1x1 signalLabelDefinition]
                  Tag: ""
          Description: "Regions where trills occur"

 Use labeledSignalSet to create a labeled signal set.
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `getLabelDefinitions(lss,LabelType="roiFeature",FrameSize=50,FrameOverlapLength=5)` returns `signalLabelDefinition` objects that contain `roiFeature` labels which have a frame size equal to 50 and a frame overlap length equal to 5.

`LabelType` — Label type

`"attribute"` | `"roi"` | `"point"` | `"attributeFeature"` | `"roiFeature"`

Label type, specified as `"attribute"`, `"roi"`, `"point"`, `"attributeFeature"`, or `"roiFeature"`.

Example: `LabelType="attribute"`

Data Types: `char` | `string`

`FrameSize` — Frame size

numeric scalar

Frame size, specified as a numeric scalar. To enable this argument, set `LabelType` to `"roiFeature"`.

Example: `LabelType="roiFeature",FrameSize=50`

Data Types: `double`

`FrameOverlapLength` — Frame overlap length

numeric scalar

Frame overlap length, specified as a numeric scalar. To enable this argument, set `LabelType` to `"roiFeature"`. You cannot specify `FrameOverlapLength` and `FrameRate` simultaneously.

When you specify a frame overlap length, the function returns `signalLabelDefinition` objects that contain `roiFeature` labels that have `FrameSize` and `FrameOverlapLength` equal to the values you specify.

Example: `LabelType="roiFeature",FrameSize=50,FrameOverlapLength=5`

Data Types: `double`

`FrameRate` — Frame rate

numeric scalar

Frame rate, specified as a numeric scalar. To enable this argument, set `LabelType` to `"roiFeature"`. You cannot specify `FrameRate` and `FrameOverlapLength` simultaneously.

When you specify a frame rate, the function returns `signalLabelDefinition` objects that contain `roiFeature` labels that have `FrameSize` and `FrameRate` equal to the values you specify.

Example: `LabelType="roiFeature",FrameSize=50,FrameRate=45`

Data Types: `double`

Output Arguments

`lbldefs` — Signal label definitions

`signalLabelDefinition` object

Signal label definitions, returned as a `signalLabelDefinition` object or a vector of such objects.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

getLabeledSignal

Get labeled signals from labeled signal set

Syntax

```
[t,info] = getLabeledSignal(lss)
[t,info] = getLabeledSignal(lss,midx)
```

Description

`[t,info] = getLabeledSignal(lss)` returns a table with all the signals and labeled data in the labeled signal set `lss`.

`[t,info] = getLabeledSignal(lss,midx)` returns a table with the signals specified in `midx`.

Examples

Get Labeled Signal

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Get a table with all the signals in `lss`.

```
t = getLabeledSignal(lss)
```

```
t=2×4 table
                     Signal          WhaleType    MoanRegions    TrillRegions
                ________________    _________    ___________    ____________

    Member{1}    {79572x1 double}      blue       {3x2 table}    {1x3 table}
    Member{2}    {76579x1 double}      blue       {3x2 table}    {1x3 table}
```

Identify the sublabels of the trill regions.

```
d = getLabelNames(lss,'TrillRegions')
```

```
d =
"TrillPeaks"
```

Get the labeled signal corresponding to the second member of the set. Determine the sample rate.

```
idx = 2;
```

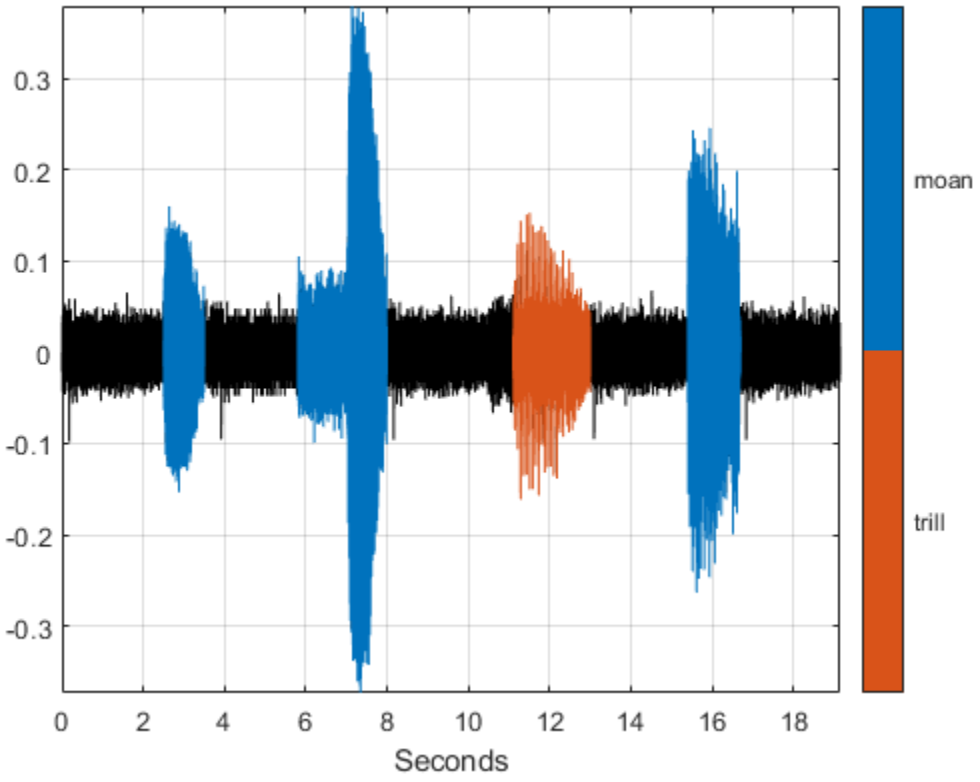
```
[lbs,info] = getLabeledSignal(lss,idx)
```

```
lbs=1×4 table
                    Signal          WhaleType    MoanRegions    TrillRegions
                ________________    _________    ___________    ____________

    Member{2}    {76579x1 double}      blue       {3x2 table}    {1x3 table}


info = struct with fields:
    TimeInformation: "sampleRate"
         SampleRate: 4000
```

```
fs = info.SampleRate;
```

Identify the moan and trill regions of interest. Use a `signalMask` object to plot the signal and highlight the moans and trills.

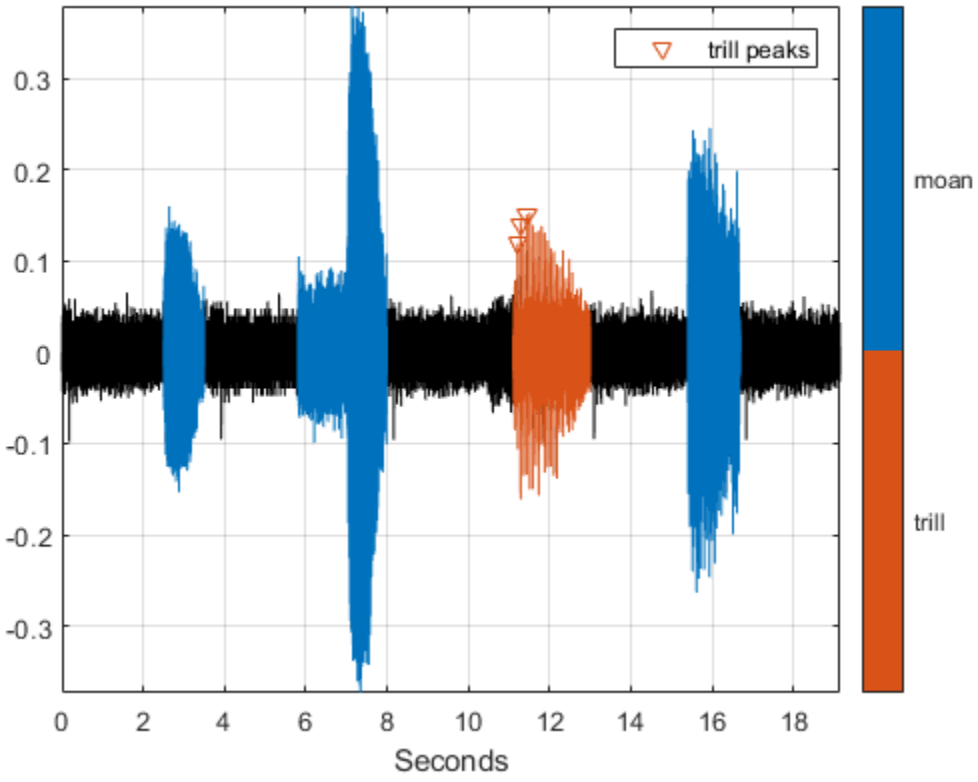
```
mvals = getLabelValues(lss,idx,'MoanRegions');
tvals = getLabelValues(lss,idx,'TrillRegions');

tb = [mvals;tvals];
tb.Value = categorical( ...
    [repmat("moan",height(mvals),1);repmat("trill",height(tvals),1)], ...
    ["moan" "trill"]);
sm = signalMask(tb,"SampleRate",fs);
plotsigroi(sm,getSignal(lss,idx))
```



Identify three peaks of the trill region and plot them.

```
peaks = getLabelValues(lss,idx,{'TrillRegions','TrillPeaks'});
```

```
hold on
pk = plot(peaks.Location,cell2mat(peaks.Value),'v');
hold off
legend(pk,'trill peaks')
```



Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`midx` — Member row number

positive integer

Member row number, specified as a positive integer. `midx` specifies the member row number as it appears in the "Labels" on page 1-0 table of a labeled signal set.

Output Arguments

`t` — Labeled signal

table

Labeled signal, specified as a table.

`info` — Time information
structure

Time information, returned as a structure.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

getLabelIndices

Get label indices pointing to label definitions in labeled signal set

Syntax

```
idxs = getLabelIndices(lss)
getLabelIndices(lss,Name=Value)
```

Description

`idxs = getLabelIndices(lss)` returns a vector of indices `idxs` pointing to label definitions in labeled signal set `lss`. Use this function to obtain label definition indices when you want to create feature data using `createFeatureData`.

`getLabelIndices(lss,Name=Value)` returns a vector of indices pointing to label definitions in `lss` that have a label type and frame policy equal to the values you specify using name-value arguments. For example, `LabelType="point"` returns indices for point label definitions.

Examples

Get Label Indices

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
        NumMembers: 2
    TimeInformation: "sampleRate"
        SampleRate: 4000
            Labels: [2x3 table]
       Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

`lss` contains one attribute label and two region-of-interest (ROI) labels. Return label indices for the ROI labels.

```
idx = getLabelIndices(lss,LabelType="roi")
```

```
idx = 2×1

     2
     3
```

Add an ROI feature label definition to `lss`. Return the label index corresponding to the ROI feature label.

```
sld = signalLabelDefinition("roiFT",LabelType="roiFeature", ...
      LabelDataType="numeric",FrameSize=200,FrameOverlapLength=50);

addLabelDefinitions(lss,sld);
getLabelIndices(lss,LabelType="roiFeature")
```

```
ans = 4
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `getLabelIndices(lss,LabelType="roiFeature",FrameSize=100,FrameOverlapLength=25)` returns indices corresponding to region-of-interest (ROI) feature label definitions in the input labeled signal set that have a frame size equal to 100 and a frame overlap length equal to 25.

`LabelType` — Label type

`"attribute"` | `"roi"` | `"point"` | `"attributeFeature"` | `"roiFeature"`

Label type, specified as `"attribute"`, `"roi"`, `"point"`, `"attributeFeature"`, or `"roiFeature"`.

Data Types: `char` | `string`

`FrameSize` — Frame size

numeric scalar

Frame size, specified as a numeric scalar. To enable this argument, set `LabelType` to `"roiFeature"`.

Example: `LabelType="roiFeature",FrameSize=50` specifies an ROI feature label type and a frame policy in which frame size is equal to 50.

Data Types: `double`

`FrameOverlapLength` — Frame overlap length

numeric scalar

Frame overlap length, specified as a numeric scalar. To enable this argument, set `LabelType` to `"roiFeature"`. You cannot specify `FrameOverlapLength` and `FrameRate` simultaneously.

Example: `LabelType="roiFeature",FrameSize=50,FrameOverlapLength=5` specifies an ROI feature label type and a frame policy in which frame size is equal to 50 and frame overlap length is equal to 5.

Data Types: `double`

`FrameRate` — Frame rate
numeric scalar

Frame rate, specified as a numeric scalar. To enable this argument, set `LabelType` to `"roiFeature`. You cannot specify `FrameRate` and `FrameOverlapLength` simultaneously.

Example: `LabelType="roiFeature",FrameSize=50,FrameRate=45` specifies an ROI feature label type and a frame policy in which frame size is equal to 50 and frame rate is equal to 45.

Data Types: `double`

Output Arguments

`idxs` — Indices
vector

Indices pointing to label definitions in `lss`, returned as a vector.

Note The function returns indices corresponding to parent labels only and does not consider sublabels.

See Also

Objects
`labeledSignalSet` | `signalLabelDefinition`

Functions
`getLabelDefinitions` | `getLabelNames`

Introduced in R2022a

getLabelNames

Get label names in labeled signal set

Syntax

```
lblnames = getLabelNames(lss)
sublblnames = getLabelNames(lss,lblname)
getLabelNames(lss,Name=Value)
```

Description

`lblnames = getLabelNames(lss)` returns a string array containing the label names in the labeled signal set `lss`.

`sublblnames = getLabelNames(lss,lblname)` returns a string array containing the sublabel names for the label named `lblname` in the labeled signal set `lss`.

`getLabelNames(lss,Name=Value)` returns a string array containing the label names in the labeled signal set for labels which have a label type and frame policy equal to the values you specify using name-value arguments.

Examples

Get Label Names

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

             Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Get the names of the labels in the set.

```
str = getLabelNames(lss)
```

```
str = 3x1 string
    "WhaleType"
    "MoanRegions"
```

```
    "TrillRegions"
```

Verify that only the `'TrillRegions'` label has sublabels.

```
for kj = 1:length(str)
    sbstr = str{kj};
    sbl = [sbstr getLabelNames(lss,sbstr)]
end
```

```
sbl =
"WhaleType"

sbl =
"MoanRegions"

sbl = 1x2 string
    "TrillRegions"    "TrillPeaks"
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`lblname` — Label name

character vector | string scalar

Label name, specified as a character vector or a string scalar.

Example: `signalLabelDefinition("Asleep",'LabelType','roi')` specifies a label of name `"Asleep"` for a region of a signal in which a patient is asleep during a clinical trial.

Name-Value Arguments

Example: `getLabelNames(lss,LabelType="roiFeature",FrameSize=50,FrameOverlapLength=5)` returns the names of `roiFeature` labels which have a frame size equal to 50 and a frame overlap length equal to 5.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

`LabelType` — Label type

`"attribute"` | `"roi"` | `"point"` | `"attributeFeature"` | `"roiFeature"`

Label type, specified as `"attribute"`, `"roi"`, `"point"`, `"attributeFeature"`, or `"roiFeature"`.

Example: `LabelType="attribute"`

Data Types: `char` | `string`

`FrameSize` — Frame size
numeric scalar

Frame size, specified as a numeric scalar. To enable this argument, set `LabelType` to `"roiFeature"`.

Example: `LabelType="roiFeature",FrameSize=50`

Data Types: `double`

`FrameOverlapLength` — Frame overlap length
numeric scalar

Frame overlap length, specified as a numeric scalar. To enable this argument, set `LabelType` to `"roiFeature"`. You cannot specify `FrameOverlapLength` and `FrameRate` simultaneously.

Example: `LabelType="roiFeature",FrameSize=50,FrameOverlapLength=5`

Data Types: `double`

`FrameRate` — Frame rate
numeric scalar

Frame rate, specified as a numeric scalar. To enable this argument, set `LabelType` to `"roiFeature"`. You cannot specify `FrameRate` and `FrameOverlapLength` simultaneously.

Example: `LabelType="roiFeature",FrameSize=50,FrameRate=45`

Data Types: `double`

Output Arguments

`lblnames` — Label names
string array

Label names, returned as a string array.

`sublblnames` — Sublabel names
string array

Sublabel names, returned as a string array.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

getLabelValues

Get label values from labeled signal set

Syntax

```
val = getLabelValues(lss)
val = getLabelValues(lss,midx)

[val,sublbltbl] = getLabelValues(lss,midx,lblname)

[___] = getLabelValues(___,'LabelRowIndex',ridx)
[___] = getLabelValues(___,'SublabelRowIndex',sridx)
```

Description

`val = getLabelValues(lss)` returns a table containing the label values for all members of the labeled signal set `lss`.

`val = getLabelValues(lss,midx)` returns a table containing the label values for the member specified by `midx`.

`[val,sublbltbl] = getLabelValues(lss,midx,lblname)` returns the value of the label named `lblname`. If `lblname` has sublabels, then the table `sublbltbl` shows the structure of the label value and its sublabel variables.

`[___] = getLabelValues(___,'LabelRowIndex',ridx)` specifies the row index, `ridx`, of an ROI or point label whose value you want to get.

`[___] = getLabelValues(___,'SublabelRowIndex',sridx)` specifies the row index, `sridx`, of an ROI or point sublabel whose value you want to get.

Examples

Get Label Values

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
        NumMembers: 2
    TimeInformation: "sampleRate"
        SampleRate: 4000
            Labels: [2x3 table]
       Description: "Characterize wave song regions"
```

```
 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Get the values of the labels.

```
lbls = getLabelValues(lss)
```

```
lbls=2×3 table
                 WhaleType    MoanRegions    TrillRegions
                 _________    ___________    ____________

    Member{1}      blue       {3x2 table}    {1x3 table}
    Member{2}      blue       {3x2 table}    {1x3 table}
```

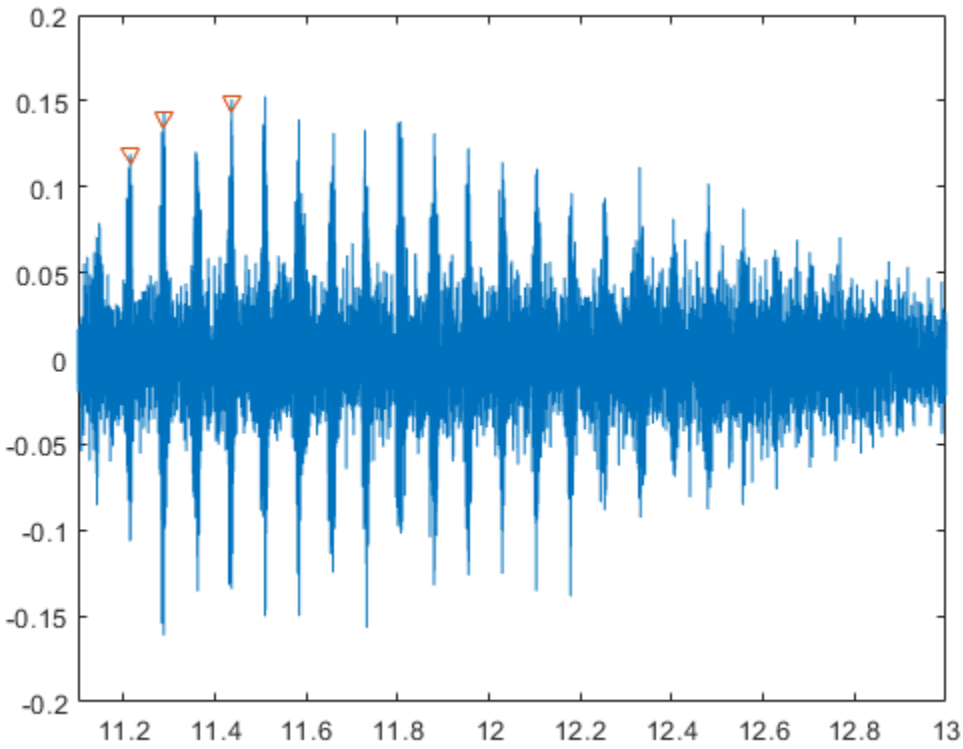
Display the moan ROI limits for the second signal of the set.

```
lbb = getLabelValues(lss,2,'MoanRegions')
```

```
lbb=3×2 table
     ROILimits      Value
    ____________    _____

     2.5     3.5    {[1]}
     5.8       8    {[1]}
    15.4    16.7    {[1]}
```

Plot the trill region of the signal between the ROI limits. Display the labeled trill peaks.

```
tvals = getLabelValues(lss,2,'TrillRegions');
peaks = getLabelValues(lss,2,{'TrillRegions','TrillPeaks'});

sg = getSignal(lss,2);
plot((0:length(sg)-1)/lss.SampleRate,sg)
xlim(tvals.ROILimits)
hold on
plot(peaks.Location,cell2mat(peaks.Value),'v')
hold off
```



Display the coordinates of the third trill peak.

```
pcoor = getLabelValues(lss,2,{'TrillRegions','TrillPeaks'}, ...
    'LabelRowIndex',1,'SublabelRowIndex',3)
```

```
pcoor=1×2 table
    Location      Value
    ________    __________

     11.437     {[0.1500]}
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`midx` — Member row number

positive integer

Member row number, specified as a positive integer. `midx` specifies the member row number as it appears in the “Labels” on page 1-0 table of a labeled signal set.

`lblname` — Label or sublabel name
character vector | string scalar | cell array of character vectors | string array

Label or sublabel name. To specify a label, use a character vector or a string scalar. To specify a sublabel, use a two-element cell array of character vectors or a two-element string array:

- The first element is the name of the parent label.
- The second element is the name of the sublabel.

Example: `signalLabelDefinition("Asleep",'LabelType','roi')` specifies a label of name `"Asleep"` for a region of a signal in which a patient is asleep during a clinical trial.

Example: `{'Asleep' 'REM'}` or `["Asleep" "REM"]` specifies a region of a signal in which a patient undergoes REM sleep.

`ridx` — Label row index
positive integer

Label row index, specified as a positive integer. This argument applies only for ROI and point labels.

`sridx` — Sublabel row index
positive integer

Sublabel row index, specified as a positive integer. This argument applies only when a label and sublabel pair has been specified in `lblname` and the sublabel is of type ROI or point.

Output Arguments

`val` — Label values
table

Label values, returned as a table.

`sublbltbl` — Sublabel values
table

Sublabel values, returned as a table showing the structure of the label value and its sublabel variables.

- If `lblname` has no sublabels, then `sublbltbl` is empty.
- If you specify `lblname` as a string or cell array, then `sublbltbl` is empty.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

getMemberNames

Get member names in labeled signal set

Syntax

```
mnames = getMemberNames(lss)
```

Description

`mnames = getMemberNames(lss)` returns a string array containing the member names in the order in which they are stored in the labeled signal set `lss`.

Examples

Get Member Names

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
        NumMembers: 2
    TimeInformation: "sampleRate"
        SampleRate: 4000
            Labels: [2x3 table]
       Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Return a string array with the names of the members.

```
getMemberNames(lss)
```

```
ans = 2x1 string
    "Member{1}"
    "Member{2}"
```

Set the names of the set members to the whales' nicknames.

```
setMemberNames(lss,{'Brutus' 'Lucy'})
```

Verify that the members have the nicknames as names.

```
getMemberNames(lss)
```

```
ans = 2x1 string
    "Brutus"
    "Lucy"
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

Output Arguments

`mnames` — Member names

string array

Member names, returned as a string array.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

getSignal

Get signals from labeled signal set

Syntax

```
[s,info] = getSignal(lss,midx)
```

Description

`[s,info] = getSignal(lss,midx)` returns the values for the signals contained in member `midx` of the labeled signal set `lss`.

Examples

Get Signal

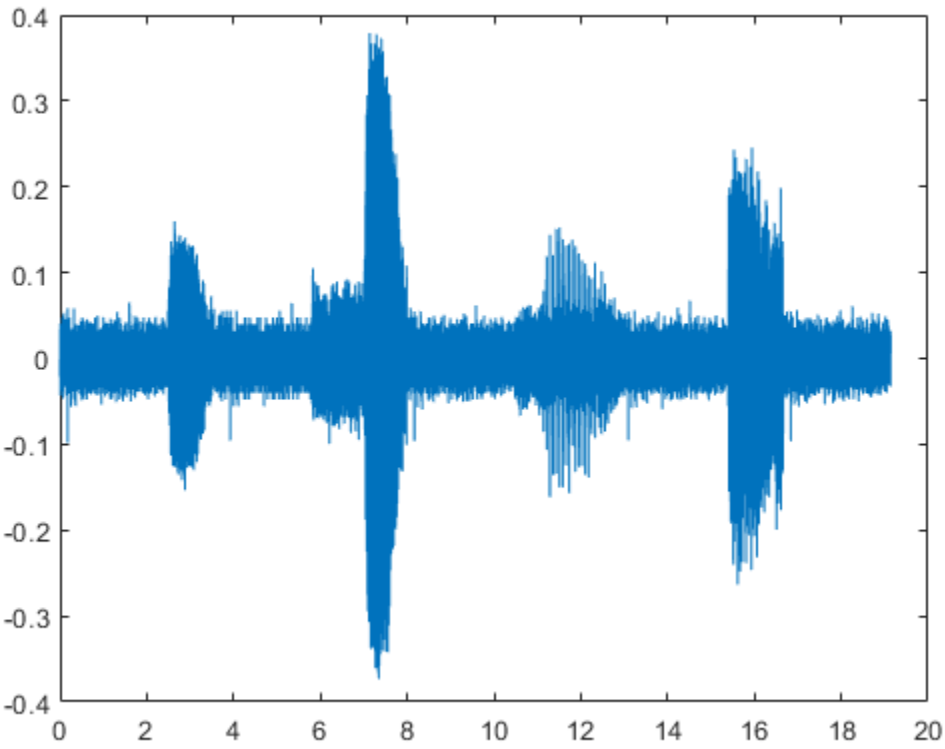
Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
        NumMembers: 2
    TimeInformation: "sampleRate"
        SampleRate: 4000
            Labels: [2x3 table]
       Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Retrieve the second member of the set and plot it.

```
[song,tinfo] = getSignal(lss,2);
t = (0:length(song)-1)/tinfo.SampleRate;
plot(t,song)
```



Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`midx` — Member row number

positive integer

Member row number, specified as a positive integer. `midx` specifies the member row number as it appears in the "Labels" on page 1-0 table of a labeled signal set.

Output Arguments

`s` — Signal values

vector | matrix | timetable | cell array

Signal values, returned as vector, matrix, timetable, or cell array.

`info` — Time information
structure

Time information, returned as a structure.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

head

Get top rows of labels table

Syntax

```
val = head(lss)
```

Description

`val = head(lss)` returns the top rows of the labels table of the labeled signal set `lss`.

Examples

Top Label Values

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

             Source: {2x1 cell}
          NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Get the top rows of the labels table.

```
head(lss)
```

```
ans=2×3 table
                WhaleType    MoanRegions    TrillRegions
                _________    ___________    ____________

    Member{1}      blue       {3x2 table}    {1x3 table}
    Member{2}      blue       {3x2 table}    {1x3 table}
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

Output Arguments

`val` — Top rows of labels

table

Top rows of labels, returned as a table.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

merge

Merge two or more labeled signal sets

Syntax

```
lssnew = merge(lss1,...,lssN)
```

Description

`lssnew = merge(lss1,...,lssN)` merges *N* labeled signal set objects, `lss1,...,lssN`, and returns a labeled signal set `lssnew` containing all the members and label values of the input sets.

Examples

Merge Labeled Signal Sets

Load a labeled signal set containing recordings of whale songs. Display the names of the set's members and a summary of its label definitions.

```
load whales

getMemberNames(lss)
```

```
ans = 2x1 string
    "Member{1}"
    "Member{2}"
```

```
labelDefinitionsSummary(lss)
```

```
ans=3×9 table
      LabelName        LabelType     LabelDataType     Categories     ValidationFunction    Defau
    _____________    ____________    _____________    ____________    __________________    _____

    "WhaleType"      "attribute"    "categorical"    {3x1 string}       {["N/A"   ]}       {0x0
    "MoanRegions"    "roi"          "logical"        {["N/A"   ]}       {0x0 double}       {0x0
    "TrillRegions"   "roi"          "logical"        {["N/A"   ]}       {0x0 double}       {0x0
```

Create a new signal set with the same data source, time information, and labels as `lss`. Remove the first member of the new set and change the name of the remaining one. Display the names of the new set's members.

```
newlss = copy(lss);

removeMembers(newlss,1)
setMemberNames(newlss,"YoungOne")

getMemberNames(newlss)
```

```
ans =
"YoungOne"
```

Create a label definition that specifies whether a signal corresponds to a calf or to an adult whale. Add the definition to the new labeled signal set and label the member. Remove the label that specifies the moan regions. Display a summary of the new member's label definitions

```
calf = signalLabelDefinition('Calf','LabeldataType','logical','DefaultValue',false, ...
    'Description','Is the specimen a calf, or an adult?');

addLabelDefinitions(newlss,calf)
setLabelValue(newlss,1,"Calf",true)

removeLabelDefinition(newlss,"MoanRegions")
labelDefinitionsSummary(newlss)
```

```
ans=3×9 table
      LabelName        LabelType     LabelDataType     Categories     ValidationFunction    Defau
    _____________    ___________    _____________    ____________    __________________    _____

    "WhaleType"      "attribute"    "categorical"    {3x1 string}       {["N/A"   ]}      {0x0
    "TrillRegions"   "roi"          "logical"        {["N/A"   ]}       {0x0 double}      {0x0
    "Calf"           "attribute"    "logical"        {["N/A"   ]}       {0x0 double}      {[
```

Merge the two labeled signal sets. Verify that the merged set contains the members, definitions, and labels of the original sets.

```
lssmerge = merge(lss,newlss);

getMemberNames(lssmerge)
```

```
ans = 3x1 string
    "Member{1}"
    "Member{2}"
    "YoungOne"
```

```
labelDefinitionsSummary(lssmerge)
```

```
ans=4×9 table
      LabelName        LabelType     LabelDataType     Categories     ValidationFunction    Defau
    _____________    ___________    _____________    ____________    __________________    _____

    "WhaleType"      "attribute"    "categorical"    {3x1 string}       {["N/A"   ]}      {0x0
    "MoanRegions"    "roi"          "logical"        {["N/A"   ]}       {0x0 double}      {0x0
    "TrillRegions"   "roi"          "logical"        {["N/A"   ]}       {0x0 double}      {0x0
    "Calf"           "attribute"    "logical"        {["N/A"   ]}       {0x0 double}      {[
```

Input Arguments

`lss1,...,lssN` — Input labeled signal sets

`labeledSignalSet` objects

Input labeled signal sets, specified as `labeledSignalSet` objects. All input sets must have the same time information settings and data source type.

Output Arguments

`lssnew` — Merged labeled signal set

`labeledSignalSet` object

Merged labeled signal set, returned as a `labeledSignalSet` object. The set `lssnew` contains a signal source, label definitions, and label values that are independent of those in the input labeled signal sets.

- Changing any of the input labeled signal sets does not affect the merged labeled signal set.
- Changing the merged labeled signal set does not affect any of the input labeled signal sets.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2020a

removeLabelDefinition

Remove label definition from labeled signal set

Syntax

```
removeLabelDefinition(lss,lblname)
```

Description

`removeLabelDefinition(lss,lblname)` removes the label definition `lblname` from the labeled signal set `lss`. If you want to remove a sublabel, specify `lblname` as a two-element string array or two-element cell array of character vectors:

- The first element is the name of the parent label.
- The second element is the name of the sublabel.

Examples

Remove Label Definition

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Retrieve a hierarchical list of labels and sublabels.

```
labelDefinitionsHierarchy(lss)
```

```
ans =
    'WhaleType
       Sublabels: []
     MoanRegions
       Sublabels: []
     TrillRegions
       Sublabels: TrillPeaks
```

```
    '
```

Remove the sublabel that labels peaks in the trill regions.

```
removeLabelDefinition(lss,{'TrillRegions' 'TrillPeaks'})

labelDefinitionsHierarchy(lss)
```

```
ans =
    'WhaleType
       Sublabels: []
     MoanRegions
       Sublabels: []
     TrillRegions
       Sublabels: []
     '
```

Remove the label that specifies the whale type.

```
removeLabelDefinition(lss,"WhaleType")

getLabelNames(lss)
```

```
ans = 2x1 string
    "MoanRegions"
    "TrillRegions"
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`lblname` — Label or sublabel name

character vector | string scalar | cell array of character vectors | string array

Label or sublabel name. To specify a label, use a character vector or a string scalar. To specify a sublabel, use a two-element cell array of character vectors or a two-element string array:

- The first element is the name of the parent label.
- The second element is the name of the sublabel.

Example: `signalLabelDefinition("Asleep",'LabelType','roi')` specifies a label of name `"Asleep"` for a region of a signal in which a patient is asleep during a clinical trial.

Example: `{'Asleep' 'REM'}` or `["Asleep" "REM"]` specifies a region of a signal in which a patient undergoes REM sleep.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

removeMembers

Remove members from labeled signal set

Syntax

```
removeMembers(lss,midxvect)
```

Description

`removeMembers(lss,midxvect)` removes the members specified in `midxvect` from the labeled signal set `lss`.

Examples

Remove Member

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
            Labels: [2x3 table]
       Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Remove the second member of the set.

```
removeMembers(lss,2)
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {[79572x1 double]}
         NumMembers: 1
    TimeInformation: "sampleRate"
         SampleRate: 4000
            Labels: [1x3 table]
       Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
```

```
Use setLabelValue to add data to the set.
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`midxvect` — Subset member row numbers

vector of positive integers

Subset member row numbers, specified as a vector of positive integers. Each element of `midxvect` specifies a member row number as it appears in the "Labels" on page 1-0 table of the `labeledSignalSet` object `lss`.

Example: `[2 3 5 7 11 13 17]` chooses a subset of signals indexed by prime numbers.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

removePointValue

Remove row from point label

Syntax

```
removePointValue(lss,midx,lblname)
removePointValue(lss,midx,lblname,'LabelRowIndex',ridx)
removePointValue(lss,midx,lblname,'SublabelRowIndex',sridx)
removePointValue(lss,midx,lblname,'LabelRowIndex',ridx,'SublabelRowIndex',
sridx)
```

Description

`removePointValue(lss,midx,lblname)` removes all rows of the point label `lblname` for the member specified by `midx`.

- If `lblname` is a character vector or a string scalar, the function targets a parent label.
- If `lblname` is a two-element string array or a two-element cell array of character vectors, the function:
  - Interprets the first element as the name of a parent label.
  - Interprets the second element as the sublabel name of a point label.
  - Removes all the points of the sublabel.

`removePointValue(lss,midx,lblname,'LabelRowIndex',ridx)` removes a row, specified by `ridx`, of the point label `lblname` for the member `midx`.

If `lblname` is a two-element string array or a two-element cell array of character vectors, the function:

- Interprets the first element as the name of a parent label.
- Interprets the second element as the sublabel name of a point label.
- Removes all the points of the sublabel contained in row `ridx`.

`removePointValue(lss,midx,lblname,'SublabelRowIndex',sridx)` removes the sublabel row specified by `sridx`. In this case, `lblname` must be a two-element string array or a two-element cell array of character vectors:

- The first element is the name of a parent attribute label.
- The second element is the sublabel name of a point label.

`removePointValue(lss,midx,lblname,'LabelRowIndex',ridx,'SublabelRowIndex',
sridx)` removes the sublabel row specified by `sridx` of the ROI or point label row specified by `ridx`. In this case, `lblname` must be a two-element string array or a two-element cell array of character vectors:

- The first element is the name of a parent ROI or point label.
- The second element is the sublabel name of a point label.

Examples

Remove Point Value

Load a labeled signal set containing recordings of whale songs. Get the names of the labels and the number of members.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
        NumMembers: 2
    TimeInformation: "sampleRate"
        SampleRate: 4000
            Labels: [2x3 table]
       Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

```
nm = lss.NumMembers;
```

Define a point label associated with the signal maximum.

```
themax = signalLabelDefinition('Maximum','LabelType','point', ...
    'LabelDataType','numeric')
```

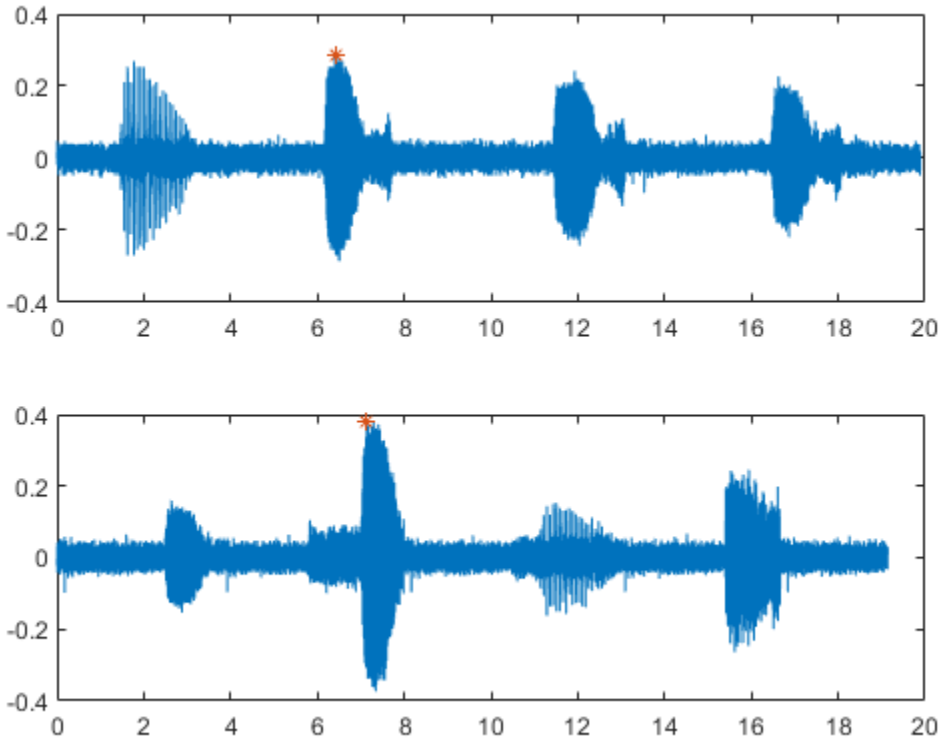
```
themax =
  signalLabelDefinition with properties:

                    Name: "Maximum"
               LabelType: "point"
           LabelDataType: "numeric"
        ValidationFunction: []
    PointLocationsDataType: "double"
            DefaultValue: []
               Sublabels: [0x0 signalLabelDefinition]
                     Tag: ""
             Description: ""

 Use labeledSignalSet to create a labeled signal set.
```

```
addLabelDefinitions(lss,themax)
```

Find the maxima of the signals and add their values to the labeled set.

```
figure
for idx = 1:nm
    sg = getSignal(lss,idx);
    [mx,ix] = max(sg);
    setLabelValue(lss,idx,'Maximum',ix,mx)

    subplot(nm,1,idx)
```

```
    plot((0:length(sg)-1)/lss.SampleRate,sg,ix/lss.SampleRate,mx,'*')
end
```



Verify that the set includes the new point label.

```
getLabelValues(lss)
```

```
ans=2×4 table
                WhaleType    MoanRegions    TrillRegions      Maximum
                _________    ___________    ____________    ___________

    Member{1}      blue       {3x2 table}    {1x3 table}     {1x2 table}
    Member{2}      blue       {3x2 table}    {1x3 table}     {1x2 table}
```

Remove the `'Maximum'` value for the first member of the set. Verify that the label is empty for the first member.

```
removePointValue(lss,1,'Maximum')

getLabelValues(lss,1)
```

```
ans=1×4 table
                WhaleType    MoanRegions    TrillRegions      Maximum
                _________    ___________    ____________    ___________

    Member{1}      blue       {3x2 table}    {1x3 table}     {0x2 table}
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`midx` — Member row number

positive integer

Member row number, specified as a positive integer. `midx` specifies the member row number as it appears in the "Labels" on page 1-0 table of a labeled signal set.

`lblname` — Label or sublabel name

character vector | string scalar | cell array of character vectors | string array

Label or sublabel name. To specify a label, use a character vector or a string scalar. To specify a sublabel, use a two-element cell array of character vectors or a two-element string array:

- The first element is the name of the parent label.
- The second element is the name of the sublabel.

Example: `signalLabelDefinition("Asleep",'LabelType','roi')` specifies a label of name `"Asleep"` for a region of a signal in which a patient is asleep during a clinical trial.

Example: `{'Asleep' 'REM'}` or `["Asleep" "REM"]` specifies a region of a signal in which a patient undergoes REM sleep.

`ridx` — Label row index

positive integer

Label row index, specified as a positive integer. This argument applies only for ROI and point labels.

`sridx` — Sublabel row index

positive integer

Sublabel row index, specified as a positive integer. This argument applies only when a label and sublabel pair has been specified in `lblname` and the sublabel is of type ROI or point.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

removeRegionValue

Remove row from ROI label

Syntax

```
removeRegionValue(lss,midx,lblname)
removeRegionValue(lss,midx,lblname,'LabelRowIndex',ridx)
removeRegionValue(lss,midx,lblname,'SublabelRowIndex',sridx)
removeRegionValue(lss,midx,lblname,'LabelRowIndex',ridx,'SublabelRowIndex',
sridx)
```

Description

`removeRegionValue(lss,midx,lblname)` removes all rows of the ROI label `lblname` for the member specified by `midx`.

- If `lblname` is a character vector or a string scalar, the function targets a parent label.
- If `lblname` is a two-element string array or a two-element cell array of character vectors, the function:
  - Interprets the first element as the name of a parent label.
  - Interprets the second element as the sublabel name of an ROI label.
  - Removes all the regions of the sublabel.

`removeRegionValue(lss,midx,lblname,'LabelRowIndex',ridx)` removes a row, specified by `ridx`, of the ROI label `lblname` for the member `midx`.

If `lblname` is a two-element string array or a two-element cell array of character vectors, the function:

- Interprets the first element as the name of a parent label.
- Interprets the second element as the sublabel name of an ROI label.
- Removes all the regions of the sublabel contained in row `ridx`.

`removeRegionValue(lss,midx,lblname,'SublabelRowIndex',sridx)` removes the sublabel row specified by `sridx`. In this case, `lblname` must be a two-element string array or a two-element cell array of character vectors:

- The first element is the name of a parent attribute label.
- The second element is the sublabel name of an ROI label.

`removeRegionValue(lss,midx,lblname,'LabelRowIndex',ridx,'SublabelRowIndex',
sridx)` removes the sublabel row specified by `sridx` of the ROI or point label row specified by `ridx`. In this case, `lblname` must be a two-element string array or a two-element cell array of character vectors:

- The first element is the name of a parent ROI or point label.
- The second element is the sublabel name of an ROI label.

Examples

Remove Region Value

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Get the names and values of the labels in the set. For the following, concentrate on the second member of the set.

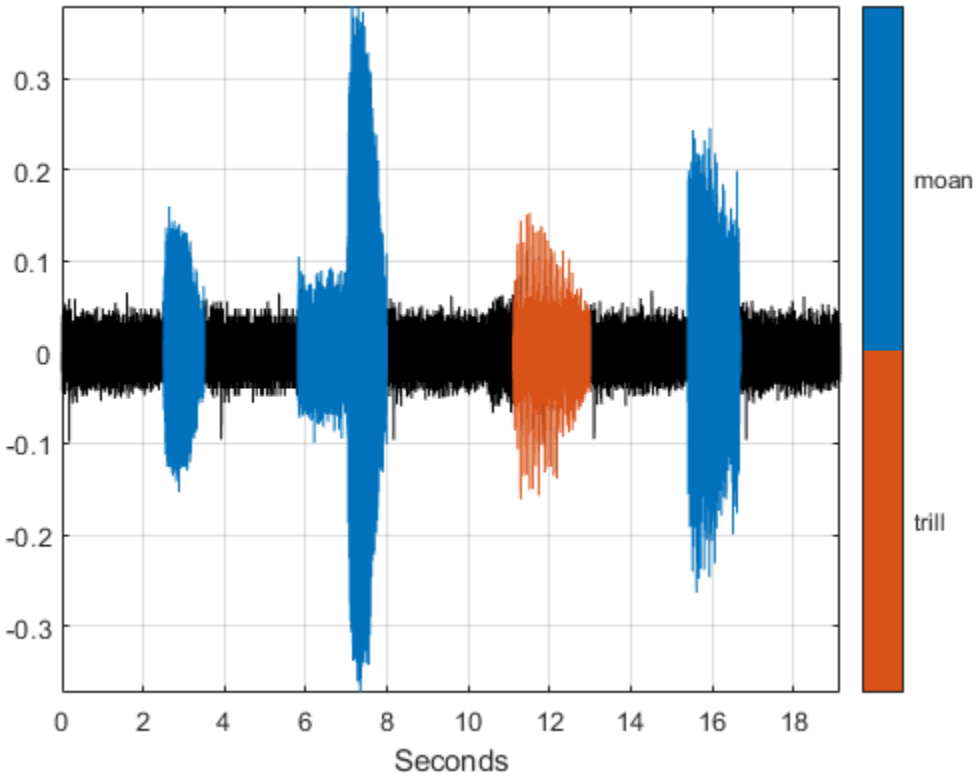
```
lbldefs = getLabelValues(lss)
```

```
lbldefs=2×3 table
                 WhaleType    MoanRegions    TrillRegions
                 _________    ___________    ____________

    Member{1}      blue       {3x2 table}    {1x3 table}
    Member{2}      blue       {3x2 table}    {1x3 table}
```

```
idx = 2;
```

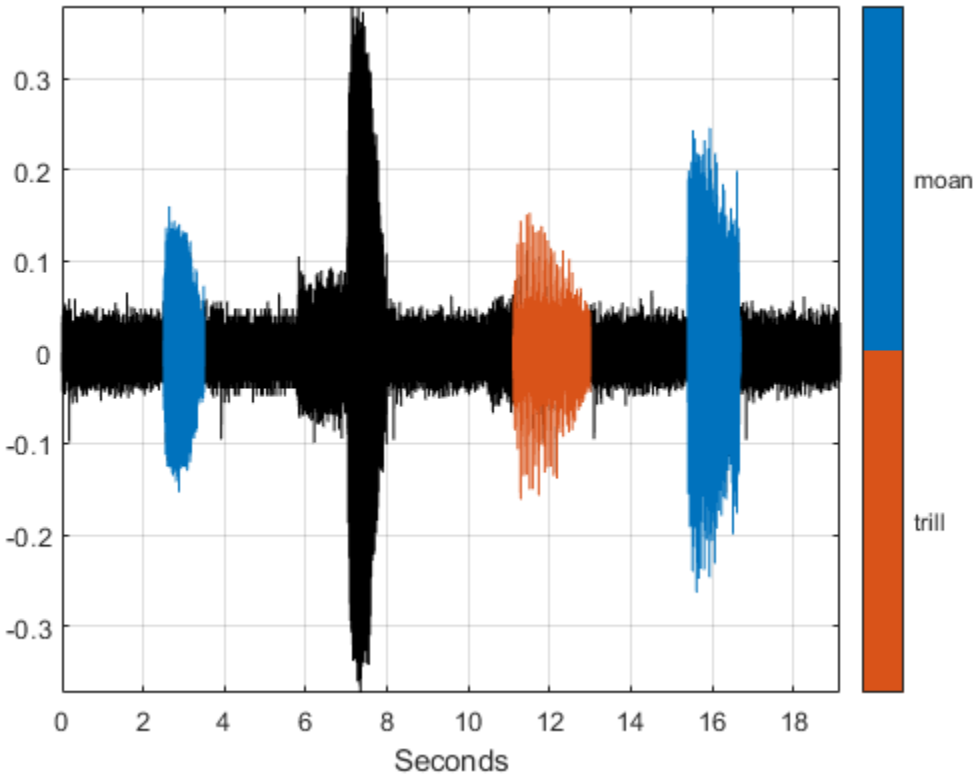
Retrieve the moan and trill regions. Use a `signalMask` object to plot the signal and highlight the moans and trills.

```
mvals = getLabelValues(lss,idx,'MoanRegions');
tvals = getLabelValues(lss,idx,'TrillRegions');

tb = [mvals;tvals];
tb.Value = categorical( ...
    [repmat("moan",height(mvals),1);repmat("trill",height(tvals),1)], ...
    ["moan" "trill"]);

sm = signalMask(tb,"SampleRate",lss.SampleRate);

plotsigroi(sm,getSignal(lss,idx))
```



Remove the second moan from the labels. Plot the signal again. Highlight the moans and trills.

```
removeRegionValue(lss,idx,'MoanRegions','LabelRowIndex',2)

mvals = getLabelValues(lss,idx,'MoanRegions');

tb = [mvals;tvals];
tb.Value = categorical( ...
    [repmat("moan",height(mvals),1);repmat("trill",height(tvals),1)], ...
    ["moan" "trill"]);

sm = signalMask(tb,"SampleRate",lss.SampleRate);

plotsigroi(sm,getSignal(lss,idx))
```



Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`midx` — Member row number

positive integer

Member row number, specified as a positive integer. `midx` specifies the member row number as it appears in the "Labels" on page 1-0 table of a labeled signal set.

`lblname` — Label or sublabel name

character vector | string scalar | cell array of character vectors | string array

Label or sublabel name. To specify a label, use a character vector or a string scalar. To specify a sublabel, use a two-element cell array of character vectors or a two-element string array:

- The first element is the name of the parent label.

- The second element is the name of the sublabel.

Example: `signalLabelDefinition("Asleep",'LabelType','roi')` specifies a label of name `"Asleep"` for a region of a signal in which a patient is asleep during a clinical trial.

Example: `{'Asleep' 'REM'}` or `["Asleep" "REM"]` specifies a region of a signal in which a patient undergoes REM sleep.

`ridx` — Label row index
positive integer

Label row index, specified as a positive integer. This argument applies only for ROI and point labels.

`sridx` — Sublabel row index
positive integer

Sublabel row index, specified as a positive integer. This argument applies only when a label and sublabel pair has been specified in `lblname` and the sublabel is of type ROI or point.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

resetLabelValues

Reset labels to default values

Syntax

```
resetLabelValues(lss)
resetLabelValues(lss,midx)

resetLabelValues(lss,midx,lblname)
resetLabelValues( ___ ,'LabelRowIndex',ridx)
```

Description

`resetLabelValues(lss)` resets all label values for all members of the labeled signal set `lss`.

`resetLabelValues(lss,midx)` resets all label values for the signals in the member specified by `midx`.

`resetLabelValues(lss,midx,lblname)` resets the values of label `lblname` for the signals in the member specified by `midx`. To reset a sublabel, make `lblname` a two-element string array or a two-element cell array of character vectors, with the first element containing the parent label name and the second element containing the sublabel name.

By default, the function resets all sublabels of a parent label. To target a sublabel of an ROI or point parent label, specify the parent label row index using `ridx`.

`resetLabelValues( ___ ,'LabelRowIndex',ridx)` specifies the row index of the ROI or point parent label for which you want to reset a sublabel value.

Examples

Reset Label Values

Load a labeled signal set containing recordings of whale songs. Get the names of the labels.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

             Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

```
getLabelNames(lss)

ans = 3x1 string
    "WhaleType"
    "MoanRegions"
    "TrillRegions"
```

Get the label values corresponding to the trill regions for the second signal in the set.

```
idx = 2;
getLabelValues(lss,idx,'TrillRegions')

ans=1×2 table
     ROILimits      Value
    ____________    _____

    11.1      13    {[1]}
```

Reset the values. Verify that `'TrillRegions'` becomes an empty array.

```
resetLabelValues(lss,idx,'TrillRegions')

getLabelValues(lss,idx,'TrillRegions')

ans =

  0x2 empty table

getLabelValues(lss,idx)

ans=1×3 table
                  WhaleType    MoanRegions    TrillRegions
                  _________    ___________    ____________

    Member{2}       blue       {3x2 table}    {0x3 table}
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`midx` — Member row number

positive integer

Member row number, specified as a positive integer. `midx` specifies the member row number as it appears in the “Labels” on page 1-0 table of a labeled signal set.

`lblname` — Label or sublabel name

character vector | string scalar | cell array of character vectors | string array

Label or sublabel name. To specify a label, use a character vector or a string scalar. To specify a sublabel, use a two-element cell array of character vectors or a two-element string array:

- The first element is the name of the parent label.
- The second element is the name of the sublabel.

Example: `signalLabelDefinition("Asleep",'LabelType','roi')` specifies a label of name `"Asleep"` for a region of a signal in which a patient is asleep during a clinical trial.

Example: `{'Asleep' 'REM'}` or `["Asleep" "REM"]` specifies a region of a signal in which a patient undergoes REM sleep.

`ridx` — Label row index

positive integer

Label row index, specified as a positive integer. This argument applies only for ROI and point labels.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

setAlternateFileSystemRoots

Set alternate file system roots when data source of labeled signal set is a datastore

Syntax

```
setAlternateFileSystemRoots(lss,altroots)
```

Description

`setAlternateFileSystemRoots` facilitates collaborative data set labeling and analysis. Use `setAlternateFileSystemRoots` to change the path to signal data in a datastore when you want to access the data in the labeled signal set on different machines.

`setAlternateFileSystemRoots(lss,altroots)` specifies alternate file system root paths `altroots` for the source data files in the labeled signal set `lss`. The function applies only to labeled signal sets whose data source is a datastore that points to files.

Note The `setAlternateFileSystemRoots` function overwrites the existing alternate file system roots. To append new alternate roots, use the `getAlternateFileSystemRoots` function to get the current roots, append the new roots to the current array, then set the extended array with `setAlternateFileSystemRoots`.

Examples

Set Alternate Root Paths for Labeled Signal Set

Create a `labeledSignalSet` to store labels for electromyographic (EMG) signals and set alternate file paths to access the data on a different machine. Download the data set from this location: https://ssd.mathworks.com/supportfiles/SPT/data/MyoelectricData.zip.

Use a `signalDatastore` to manage the EMG files. Modify `folder` to correspond to the location where you download the files. The data set is organized into subfolders, so enable the `IncludeSubfolders` flag.

```
folder = "C:\path\Data";
sds = signalDatastore(folder,IncludeSubfolders=true);
```

The data set contains signal data and index files, appended by "`d.mat`" and "`i.mat`", respectively. Create a subset of the datastore that includes only the signal data files.

```
listing = dir(append(sds.Folders{1},"\**\*.mat"));
files = {listing(:).name}';
idx = contains(files,"d.mat");
subds = subset(sds,idx);
```

Use the subset datastore as the data source for a `labeledSignalSet` object. You can import the labeled signal set into **Signal Labeler** to interactively label the signal data, or use `addLabelDefinitions` to add labels to the labeled signal set at the command line.

```
lss = labeledSignalSet(subds);
```

To continue labeling the data on a different machine, or share the labeled signal set with another user, identify the root paths for the EMG data on your local machine and the second machine. Set the alternate file paths so that you can read the data on the second machine.

```
altroots = ["C:\path\Data" "C:\secondpath\Data"];
setAlternateFileSystemRoots(lss,altroots)
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object. The function updates the file paths of the underlying datastore source if the current paths do not exist. The function applies only to labeled signal sets whose data source is a datastore that points to files.

`altroots` — Alternate file system roots

string array | cell array

Alternate file system roots, specified as a string array or cell array. `altroots` contains one or more rows, where each row specifies a set of equivalent root paths.

Example: `["Z:\datasets" "/mynetwork/datasets"]`

Example: `{["Z:\datasets" "/mynetwork/datasets"];["Y:\datasets" "/mynetwork2/datasets" "S:\datasets"]}`

See Also

Objects

`labeledSignalSet` | `signalDatastore` | `audioDatastore`

Functions

`getAlternateFileSystemRoots`

Introduced in R2022a

setLabelValue

Set label value in labeled signal set

Syntax

```
setLabelValue(lss,midx,lblname,val)
setLabelValue(lss,midx,lblname,limits,val)
setLabelValue(lss,midx,lblname,locs,val)
setLabelValue( ___ ,'LabelRowIndex',ridx)
setLabelValue( ___ ,'SublabelRowIndex',sridx)
```

Description

`setLabelValue(lss,midx,lblname,val)` sets the attribute label `lblname` to value `val`, for the member of labeled signal set `lss` specified in `midx`. Omit `val` if `lblname` has a default value and you want to set the label to the default value.

`setLabelValue(lss,midx,lblname,limits,val)` adds regions delimited by `limits` to the ROI label named `lblname`. The number of rows of `limits` specifies the number of added regions.

`setLabelValue(lss,midx,lblname,locs,val)` adds points to the point label named `lblname`. `locs` specifies the number of added points and their locations.

`setLabelValue( ___ ,'LabelRowIndex',ridx)` specifies the row index, `ridx`, of an ROI or point label. The specified value replaces the current value of that row. If you omit this argument, the function appends ROI or point values to any existing label values.

`setLabelValue( ___ ,'SublabelRowIndex',sridx)` specifies the row index, `sridx`, of an ROI or point sublabel. The specified value replaces the current value of that sublabel row.

Examples

Set Label Value

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

            Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
```

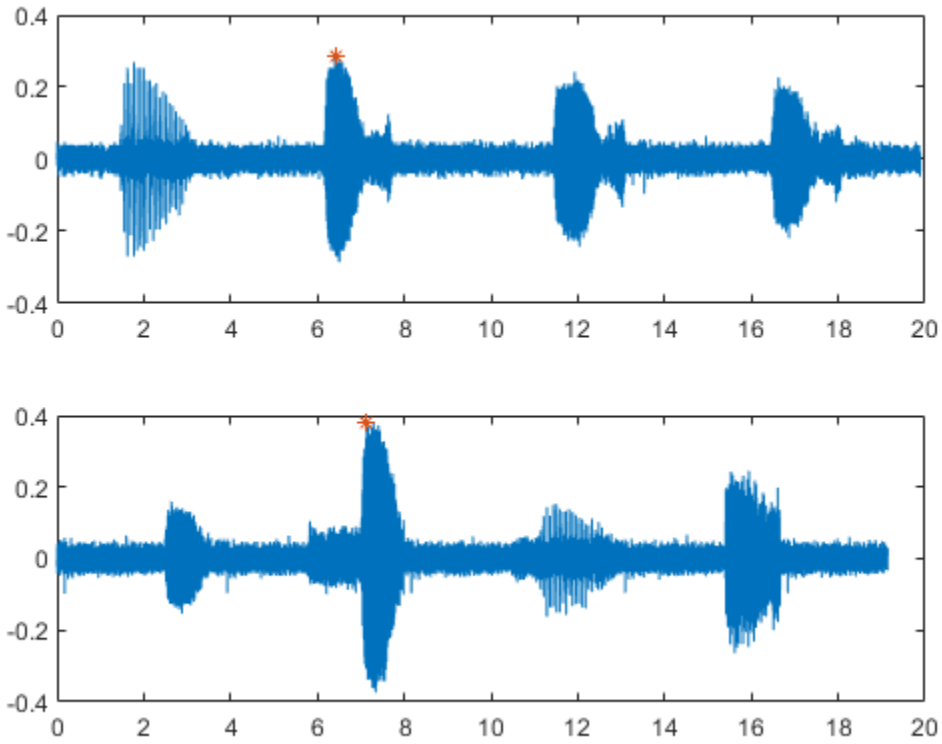
```
 Use setLabelValue to add data to the set.
```

Add a new label to the signal set, corresponding to the maximum value of each member.

```
theMax = signalLabelDefinition('Maximum', ...
    'LabelDataType','numeric', ...
    'Description','Maximum value of the signal');
addLabelDefinitions(lss,theMax)
```

For each labeled signal, set the value of the new label to the signal maximum. Plot the signals and their maxima.

```
fs = lss.SampleRate;
for k = 1:lss.NumMembers
    sg = getSignal(lss,k);
    [mx,ix] = max(sg);

    setLabelValue(lss,k,'Maximum',mx)

    subplot(2,1,k)
    plot((0:length(sg)-1)/fs,sg,ix/fs,mx,'*')
end
```



Display the names and values of the labels in the set.

```
lbldefs = getLabelValues(lss)
```

```
lbldefs=2×4 table
                 WhaleType    MoanRegions    TrillRegions     Maximum
                 _________    ___________    ____________    __________

    Member{1}      blue       {3x2 table}    {1x3 table}     {[0.2850]}
    Member{2}      blue       {3x2 table}    {1x3 table}     {[0.3791]}
```

Decide that the signal maximum is better represented as a point label than as an attribute. Remove the numeric definition and redefine the maximum.

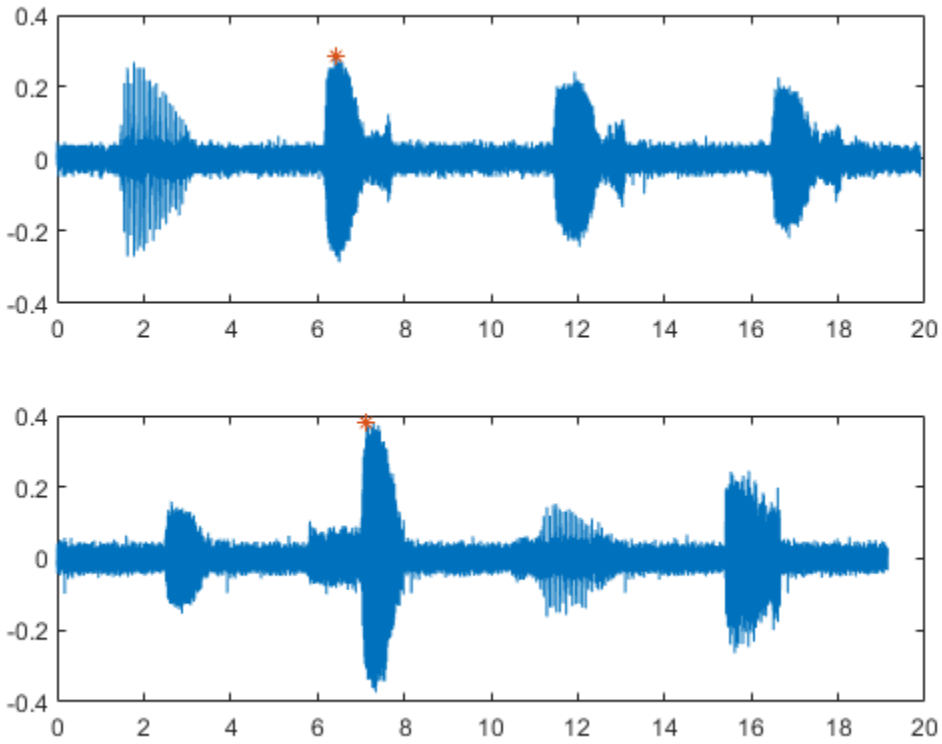
```
removeLabelDefinition(lss,'Maximum')
theMax = signalLabelDefinition('Maximum', ...
    'LabelType','point','LabelDataType','numeric', ...
    'Description','Maximum value of the signal');
addLabelDefinitions(lss,theMax)
```

For each labeled signal, set the value of the new label to the signal maximum.

```
for k = 1:lss.NumMembers
    sg = getSignal(lss,k);
    [mx,ix] = max(sg);
    setLabelValue(lss,k,'Maximum',ix/fs,mx)
end
```

Plot the signals and their maxima.

```
for k = 1:lss.NumMembers
    subplot(2,1,k)
    sg = getSignal(lss,k);
    peaks = getLabelValues(lss,k,'Maximum');
    plot((0:length(sg)-1)/fs,sg, ...
        peaks.Location,cell2mat(peaks.Value),'*')
end
```



Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`midx` — Member row number

positive integer

Member row number, specified as a positive integer. `midx` specifies the member row number as it appears in the "Labels" on page 1-0 table of a labeled signal set.

`lblname` — Label or sublabel name

character vector | string scalar | cell array of character vectors | string array

Label name, specified as a character vector or string scalar.

Label or sublabel name. To specify a label, use a character vector or a string scalar. To specify a sublabel, use a two-element cell array of character vectors or a two-element string array:

- The first element is the name of the parent label.
- The second element is the name of the sublabel.

When targeting a sublabel of an ROI or point label, you must also specify the `'LabelRowIndex'` of the parent label whose label you want to set. The row of the parent must already exist before you can set a sublabel value to it.

Example: `signalLabelDefinition("Asleep",'LabelType','roi')` specifies a label of name `"Asleep"` for a region of a signal in which a patient is asleep during a clinical trial.

Example: `{'Asleep' 'REM'}` or `["Asleep" "REM"]` specifies a region of a signal in which a patient undergoes REM sleep.

`val` — Label values

numeric value or array | logical value or array | categorical value or array | character vector or cell array of character vectors | string or string array | table or table array | timetable or timetable array

Label values, specified as a numeric, logical, or categorical value, as a string, as a table, or as a timetable. `val` can also be an array of any of the previous types. `val` must be of the data type specified for `lblname`.

- If you specify `locs`, then `val` must have the same number of elements as `locs`.
- If you specify `limits`, then `val` must have a number of elements equal to the number of rows in `limits`.
  - If `limits` has more than one row, and `lblname` is of type `'numeric'` or `'logical'`, then `val` must be a vector or a cell array.
  - If `limits` has more than one row, and `lblname` is of type `'string'` or `'categorical'`, then `val` must be a string array or a cell array of character vectors.
  - If `limits` has more than one row, and `lblname` is of type `'table'` or `'timetable'`, then `val` must be a cell array of tables or timetables.

Assign Nonscalar Label Values

To assign nonscalar label values to several points or regions of interest, you must use cell arrays. For example, given the labeled signal set

```
lss = labeledSignalSet(randn(10,1), [...
    signalLabelDefinition('pl','LabelType','point', ...
                          'LabelDataType','numeric') ...
    signalLabelDefinition('rl','LabelType','ROI', ...
                          'LabelDataType','numeric')]);
```

the commands

```
setLabelValue(lss,1,'pl',5,{[3 4]'})
setLabelValue(lss,1,'rl',[2 3; 8 9],{[2 1]' [6 7]})
```

label point 5 with the column vector `[3 4]'`, the region limited by 2 and 3 with the column vector `[2 1]'`, and the region limited by 8 and 9 with the row vector `[6 7]`.

`limits` — Region limits

two-column matrix

Region limits, specified as a two-column matrix.

- If `lss` does not have time information, then `limits` defines the minimum and maximum indices over which the regions are defined.
- If `lss` has time information, then `limits` defines the minimum and maximum instants over which the regions are defined.

`limits` must be of the data type specified by the “ROILimitsDataType” on page 1-0 property of the label definition for `lblname`.

Example: `seconds([0:3;1:4]')`

Example: `[0:3;1:4]'`

`locs` — Point locations
vector

Point locations, specified as a vector.

- If `lss` does not have time information, then `locs` defines the indices corresponding to the point locations.
- If `lss` has time information, then `locs` defines the instants corresponding to the point locations.

`locs` must be of the data type specified by the “PointLocationsDataType” on page 1-0 property of the label definition for `lblname`.

`ridx` — Label row index
positive integer

Label row index, specified as a positive integer. This argument applies only for ROI and point labels.

`sridx` — Sublabel row index
positive integer

Sublabel row index, specified as a positive integer. This argument applies only when a label and sublabel pair has been specified in `lblname` and the sublabel is of type ROI or point.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

setMemberNames

Set member names in labeled signal set

Syntax

```
setMemberNames(lss,mnames)
setMemberNames(lss,mnames,midx)
```

Description

`setMemberNames(lss,mnames)` sets the names of the members of the labeled signal set `lss` to `mnames`. The length of `mnames` must be equal to the number of members.

`setMemberNames(lss,mnames,midx)` sets the name of the member specified by `midx`.

Examples

Set Member Names

Load a labeled signal set containing recordings of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

             Source: {2x1 cell}
          NumMembers: 2
    TimeInformation: "sampleRate"
          SampleRate: 4000
              Labels: [2x3 table]
         Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Set the names of the set members to the whales' nicknames.

```
setMemberNames(lss,{'Brutus' 'Lucy'})
```

Return a string array with the names of the members.

```
getMemberNames(lss)
```

```
ans = 2x1 string
    "Brutus"
    "Lucy"
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`mnames` — Member names

character vector | string scalar | cell array of character vectors | string array

Member names, specified as a character vector, a string scalar, a cell array of character vectors, or a string array.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},'MemberNames',{'llama' 'alpaca'})` specifies a set of random signals with two members, `'llama'` and `'alpaca'`.

`midx` — Member row number

positive integer

Member row number, specified as a positive integer. `midx` specifies the member row number as it appears in the “Labels” on page 1-0 table of a labeled signal set.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2019a

subset

Get new labeled signal set with subset of members

Syntax

```
lssnew = subset(lss,midxvect)
```

Description

`lssnew = subset(lss,midxvect)` returns a new labeled signal set containing the members specified in `midxvect`.

Examples

Labeled Subset

Load a labeled signal set of whale songs.

```
load whales
lss
```

```
lss =
  labeledSignalSet with properties:

             Source: {2x1 cell}
         NumMembers: 2
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [2x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Create a new labeled signal set consisting of the second member of the original set.

```
lssnew = subset(lss,2)
```

```
lssnew =
  labeledSignalSet with properties:

             Source: {[76579x1 double]}
         NumMembers: 1
    TimeInformation: "sampleRate"
         SampleRate: 4000
             Labels: [1x3 table]
        Description: "Characterize wave song regions"

 Use labelDefinitionsHierarchy to see a list of labels and sublabels.
 Use setLabelValue to add data to the set.
```

Input Arguments

`lss` — Labeled signal set

`labeledSignalSet` object

Labeled signal set, specified as a `labeledSignalSet` object.

Example: `labeledSignalSet({randn(100,1) randn(10,1)},signalLabelDefinition('female'))` specifies a two-member set of random signals containing the attribute `'female'`.

`midxvect` — Subset member row numbers

vector of positive integers

Subset member row numbers, specified as a vector of positive integers. Each element of `midxvect` specifies a member row number as it appears in the "Labels" on page 1-0 table of the `labeledSignalSet` object `lss`.

Example: `[2 3 5 7 11 13 17]` chooses a subset of signals indexed by prime numbers.

Output Arguments

`lssnew` — New labeled signal set

`labeledSignalSet` object

New labeled signal set, returned as a `labeledSignalSet` object.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition`

Introduced in R2018b

lar2rc

Convert log area ratio parameters to reflection coefficients

Syntax

```
k = lar2rc(g)
```

Description

`k = lar2rc(g)` returns a vector of reflection coefficients `k` from a vector of log area ratio parameters `g`.

Examples

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 = 1×5

    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`

Introduced before R2006a

latc2tf

Convert lattice filter parameters to transfer function form

Syntax

```
[num,den] = latc2tf(k,v)
[num,den] = latc2tf(k,'iiroption')
num = latc2tf(k,'firoption')
```

Description

`[num,den] = latc2tf(k,v)` finds the transfer function numerator `num` and denominator `den` from the IIR lattice coefficients `k` and ladder coefficients `v`.

`[num,den] = latc2tf(k,'iiroption')` produces an IIR filter transfer function according to the value of '*iiroption*':

- `'allpole'`: Produces an all-pole filter transfer function from the associated all-pole IIR lattice filter coefficients `k`.
- `'allpass'`: Produces an allpass filter transfer function from the associated allpass IIR lattice filter coefficients `k`.

`num = latc2tf(k,'firoption')` produces an FIR filter according to the value of '*firoption*':

- `'min'`: Produces a minimum-phase FIR filter numerator from the associated minimum-phase FIR lattice filter coefficients `k`.
- `'max'`: Produces a maximum-phase FIR filter numerator from the associated maximum-phase FIR lattice filter coefficients `k`.
- `'FIR'`: Produces a general FIR filter numerator from the lattice filter coefficients `k` (this is equivalent to not specifying '*iiroption*' or '*firoption*').

See Also

`latcfilt` | `tf2latc`

Introduced before R2006a

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

`[f,g] = latcfilt(k,x)` filters input signal `x` with the FIR lattice coefficients specified by `k` and returns the forward lattice filter result `f` and backward filter result `g`.

`[f,g] = latcfilt(k,v,x)` filters input signal `x` with the IIR lattice coefficients specified by `k` and ladder coefficients specified by `v`. Both `k` and `v` must be vectors, while `x` can be a matrix.

`[f,g] = latcfilt(k,1,x)` filters input signal `x` with the IIR lattice coefficients specified by `k` and returns the all-pole lattice filter result `f` and the allpass filter result `g`.

`[f,g,zf] = latcfilt(___,"ic",zi)` specifies the initial conditions of the lattice states `zi` and returns the final conditions of the lattice states `zf`.

`[f,g,zf] = latcfilt(___,dim)` filters `x` along the dimension `dim`.

Examples

FIR Lattice Filter

Generate a signal with 512 samples of white Gaussian noise.

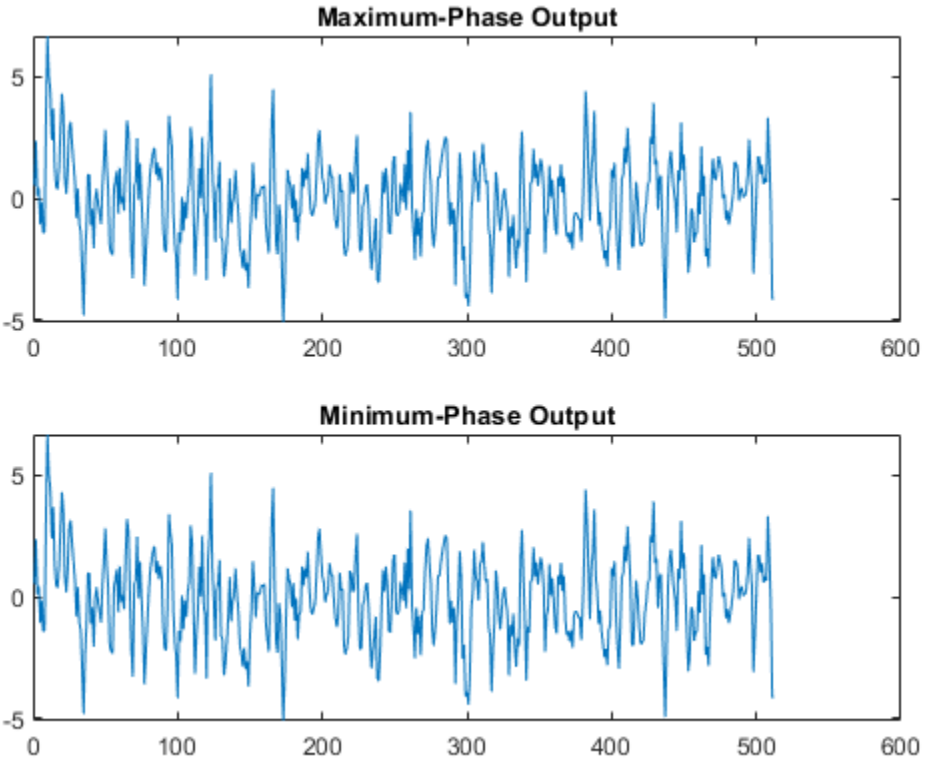
```
x = randn(512,1);
```

Filter the data with an FIR lattice filter. Specify the reflection coefficients so that the lattice filter is equivalent to a 3rd-order moving average filter.

```
[f,g] = latcfilt([1/2 1],x);
```

Plot the maximum- and minimum-phase outputs of the lattice filter in separate plots

```
subplot(2,1,1)
plot(f)
title('Maximum-Phase Output')

subplot(2,1,2)
plot(g)
title('Minimum-Phase Output')
```



Input Arguments

k — Lattice coefficients

vector | matrix

Lattice coefficients, specified as a vector or matrix. If $|k| \le 1$, `f` corresponds to the minimum-phase output, and `g` corresponds to the maximum-phase output.

When filtering data, lattice coefficients can be used to represent

- FIR filters
- All-pole IIR filters
- Allpass IIR filters
- General IIR filters

Data Types: `double`

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix.

Data Types: `double`

v — Ladder coefficients
vector

Ladder coefficients, specified as a vector.

Data Types: `double`

zi — Initial conditions
vector | matrix

Initial conditions of the lattice states, specified as a vector or matrix. If `zi` is a vector, then it must have the same length as `k`. If `zi` is a matrix, then each column must have a length equal to the length of `k`.

Data Types: `double`

dim — Dimension
positive integer scalar

Dimension along which to operate, specified as a positive integer scalar. 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.

Data Types: `double`

Output Arguments

f — Forward lattice filter
vector | matrix

Forward lattice filter, returned as a vector or matrix.

If `k` and `x` are vectors, `f` is a vector. Matrix arguments are permitted under these rules:

- If `x` is a matrix and `k` is a vector, the function processes each column of `x` through the lattice filter specified by `k`.
- If `x` is a vector and `k` is a matrix, the function uses each column of `k` to filter `x` and returns a matrix.
- If `x` and `k` are both matrices with the same number of columns, then the function uses the *i*th column of `k` to filter the *i*th column of `x` and returns a matrix.

g — Backward lattice filter
vector | matrix

Backward lattice filter, returned as a vector or matrix.

zf — Final conditions
vector | matrix

Final conditions of the lattice states, returned as a vector or matrix. If `zf` is a vector, then it has the same length as `k`. If `zf` is a matrix, then each column has a length equal to the length of `k`. `zf` returns the final conditions in columns, regardless of the shape of `x`.

See Also
`filter` | `latc2tf` | `tf2latc`

Introduced before R2006a

levinson

Levinson-Durbin recursion

Syntax

```
a = levinson(r,n)
[a,e,k] = levinson( ___ )
```

Description

`a = levinson(r,n)` returns the coefficients of an autoregressive linear process of order `n` that has `r` as its autocorrelation sequence.

`[a,e,k] = levinson( ___ )` also returns the prediction error `e`, and the reflection coefficients, `k`.

Examples

Autoregressive Process Coefficients

Estimate the coefficients of an autoregressive process given by

$$x(n) = 0.1\,x(n-1) - 0.8\,x(n-2) - 0.27\,x(n-3) + w(n).$$

```
a = [1 0.1 -0.8 -0.27];
```

Generate a realization of the process by filtering white noise of variance 0.4.

```
v = 0.4;
w = sqrt(v)*randn(15000,1);
x = filter(1,a,w);
```

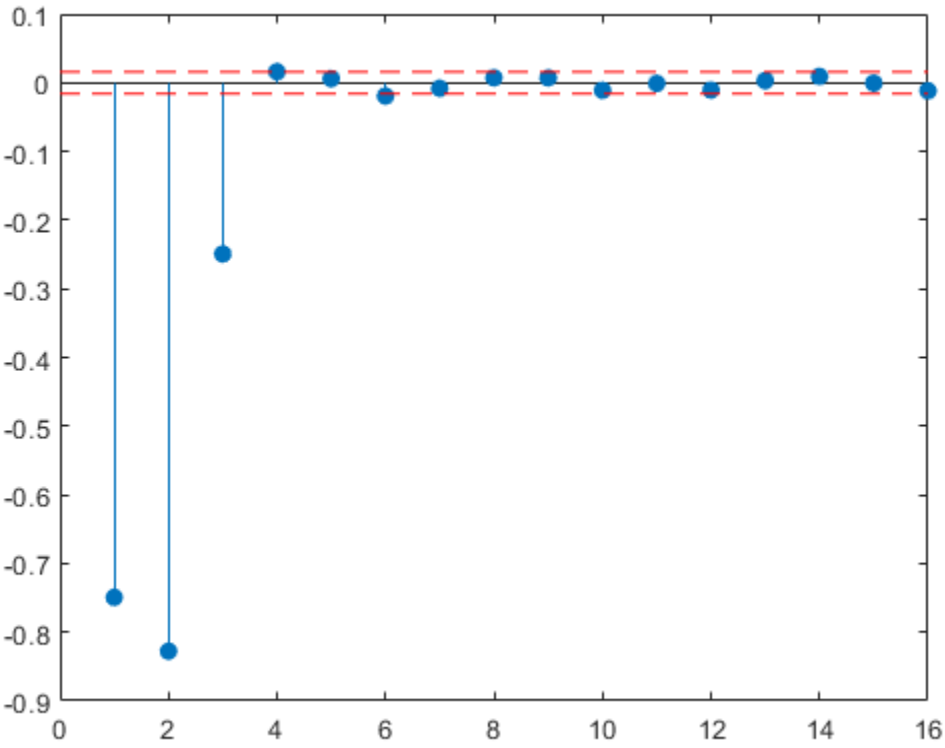
Estimate the correlation function. Discard the correlation values at negative lags. Use the Levinson-Durbin recursion to estimate the model coefficients. Verify that the prediction error corresponds to the variance of the input.

```
[r,lg] = xcorr(x,'biased');
r(lg<0) = [];

[ar,e] = levinson(r,numel(a)-1)
```

```
ar = 1×4

    1.0000    0.0772   -0.7954   -0.2493


e = 0.3909
```

Estimate the reflection coefficients for a 16th-order model. Verify that the only reflection coefficients that lie outside the 95% confidence bounds are the ones that correspond to the correct model order. See “AR Order Selection with Partial Autocorrelation Sequence” for more details.

```
[~,~,k] = levinson(r,16);
stem(k,'filled')

conf = sqrt(2)*erfinv(0.95)/sqrt(15000);
hold on
[X,Y] = ndgrid(xlim,conf*[-1 1]);
plot(X,Y,'--r')
hold off
```



Prediction Errors for Multiple Realizations

Generate the coefficients of an autoregressive process given by

$$x(n) = 0.1\,x(n-1) - 0.8\,x(n-2) - 0.27\,x(n-3) + w(n).$$

```
a = [1 0.1 -0.8 -0.27];
```

Generate five realizations of the process by filtering white noise with different variances.

```
nr = 5;
v = rand(1,nr)
```

```
v = 1×5

    0.8147    0.9058    0.1270    0.9134    0.6324
```

```
w = sqrt(v).*randn(15000,nr);
x = filter(1,a,w);
```

Estimate the correlation function. Discard cross-correlation terms and correlation values at negative lags. Use the Levinson-Durbin recursion to estimate the prediction errors for the correct model order and verify that the prediction errors correspond to the variances of the input noise signals.

```
[r,lg] = xcorr(x,'biased');

[~,e] = levinson(r(lg>=0,1:nr+1:end),numel(a)-1)
```

```
e = 5×1

    0.7957
    0.9045
    0.1255
    0.9290
    0.6291
```

Input Arguments

r — Autocorrelation sequence

vector | matrix

Autocorrelation sequence, specified as a vector or matrix. If `r` is a matrix, the function finds the coefficients for each column of `r` and returns them in the rows of `a`.

Example: `[r,lg] = xcorr(randn(1000,1),'biased'); r(lg<0) = []` estimates the autocorrelation sequence of a 1000-sample random signal for positive lags.

Data Types: `single` | `double`
Complex Number Support: Yes

n — Model order

`length(r)` – 1 (default) | positive integer scalar

Model order, specified as a positive integer scalar.

Data Types: `single` | `double`

Output Arguments

a — Autoregressive linear process coefficients

row vector | matrix

Autoregressive linear process coefficients, returned as a row vector or matrix. The filter coefficients are ordered in descending powers of z^{-1}:

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

If `r` is a matrix, then each row of `a` corresponds to a column of `r`.

e — Prediction error

scalar | column vector

Prediction error, returned as a scalar or column vector. If `r` is a matrix, then each element of `e` corresponds to a column of `r`.

`k` — Reflection coefficients
column vector | matrix

Reflection coefficients, returned as a column vector of length `n`. If `r` is a matrix, then each column of `k` corresponds to a column of `r`.

Note `k` is computed internally while computing the `a` coefficients, so returning `k` simultaneously is more efficient than converting `a` to `k` with `tf2latc`.

Algorithms

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.

`levinson` solves the symmetric Toeplitz system of linear equations

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(n)^* \\ r(2) & r(1) & \cdots & r(n-1)^* \\ \vdots & \vdots & \ddots & \vdots \\ r(n) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(n+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(n+1) \end{bmatrix},$$

where `r = [`$r(1)$ ... $r(n+1)$`]` is the input autocorrelation vector, and $r(i)^*$ denotes the complex conjugate of $r(i)$. The input `r` is typically a vector of autocorrelation coefficients where lag 0 is the first element, $r(1)$.

Note If `r` is not a valid autocorrelation sequence, the `levinson` function might return `NaN`s even if the solution exists.

The algorithm requires $O(n^2)$ flops and is thus much more efficient than the MATLAB backslash command for large `n`. However, the `levinson` function uses `\` for low orders to provide the fastest possible execution.

References

[1] Ljung, Lennart. *System Identification: Theory for the User*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation for this function requires the DSP System Toolbox software.
- Input `n`, when specified, must be a scalar.

See Also

`lpc` | `prony` | `rlevinson` | `schurrc` | `stmcb`

Introduced before R2006a

lowpass

Lowpass-filter signals

Syntax

```
y = lowpass(x,wpass)
y = lowpass(x,fpass,fs)
y = lowpass(xt,fpass)

y = lowpass( ___ ,Name,Value)

[y,d] = lowpass( ___ )

lowpass( ___ )
```

Description

`y = lowpass(x,wpass)` filters the input signal `x` using a lowpass filter with normalized passband frequency `wpass` in units of π rad/sample. `lowpass` uses a minimum-order filter with a stopband attenuation of 60 dB and compensates for the delay introduced by the filter. If `x` is a matrix, the function filters each column independently.

`y = lowpass(x,fpass,fs)` specifies that `x` has been sampled at a rate of `fs` hertz. `fpass` is the passband frequency of the filter in hertz.

`y = lowpass(xt,fpass)` lowpass-filters the data in timetable `xt` using a filter with a passband frequency of `fpass` hertz. The function independently filters all variables in the timetable and all columns inside each variable.

`y = lowpass( ___ ,Name,Value)` specifies additional options for any of the previous syntaxes using name-value pair arguments. You can change the stopband attenuation, the “Lowpass Filter Steepness” on page 1-1329, and the type of impulse response of the filter.

`[y,d] = lowpass( ___ )` also returns the `digitalFilter` object `d` used to filter the input.

`lowpass( ___ )` with no output arguments plots the input signal and overlays the filtered signal.

Examples

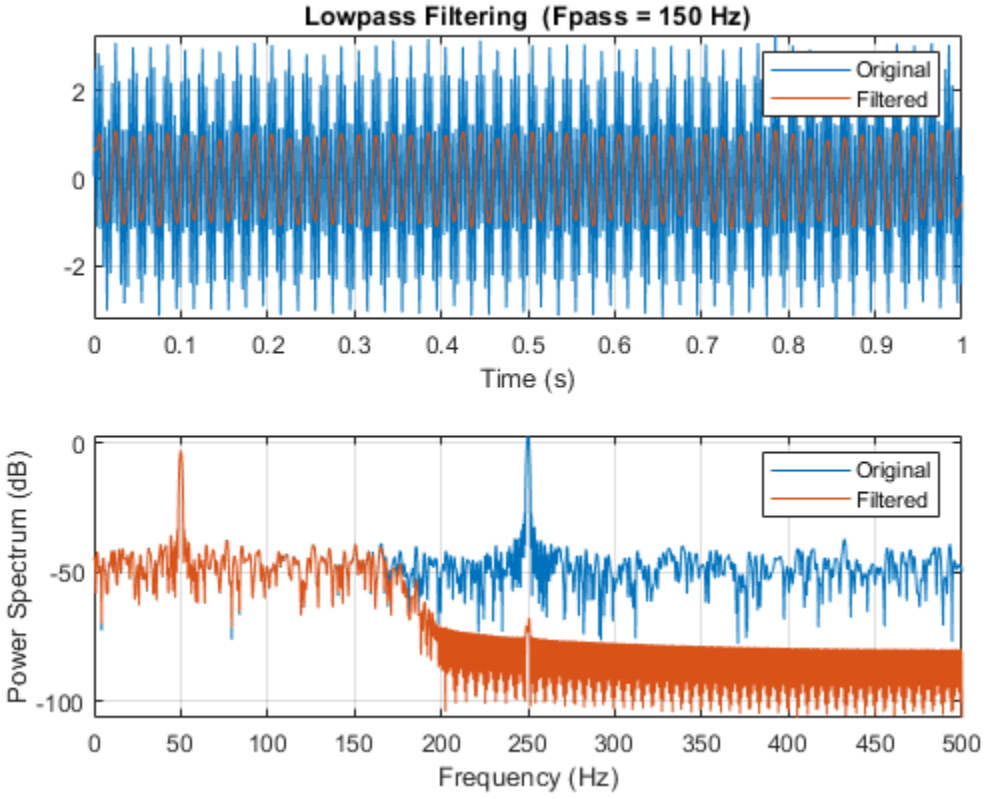
Lowpass Filtering of Tones

Create a signal sampled at 1 kHz for 1 second. The signal contains two tones, one at 50 Hz and the other at 250 Hz, embedded in Gaussian white noise of variance 1/100. The high-frequency tone has twice the amplitude of the low-frequency tone.

```
fs = 1e3;
t = 0:1/fs:1;

x = [1 2]*sin(2*pi*[50 250]'.*t) + randn(size(t))/10;
```

Lowpass-filter the signal to remove the high-frequency tone. Specify a passband frequency of 150 Hz. Display the original and filtered signals, and also their spectra.

```
lowpass(x,150,fs)
```

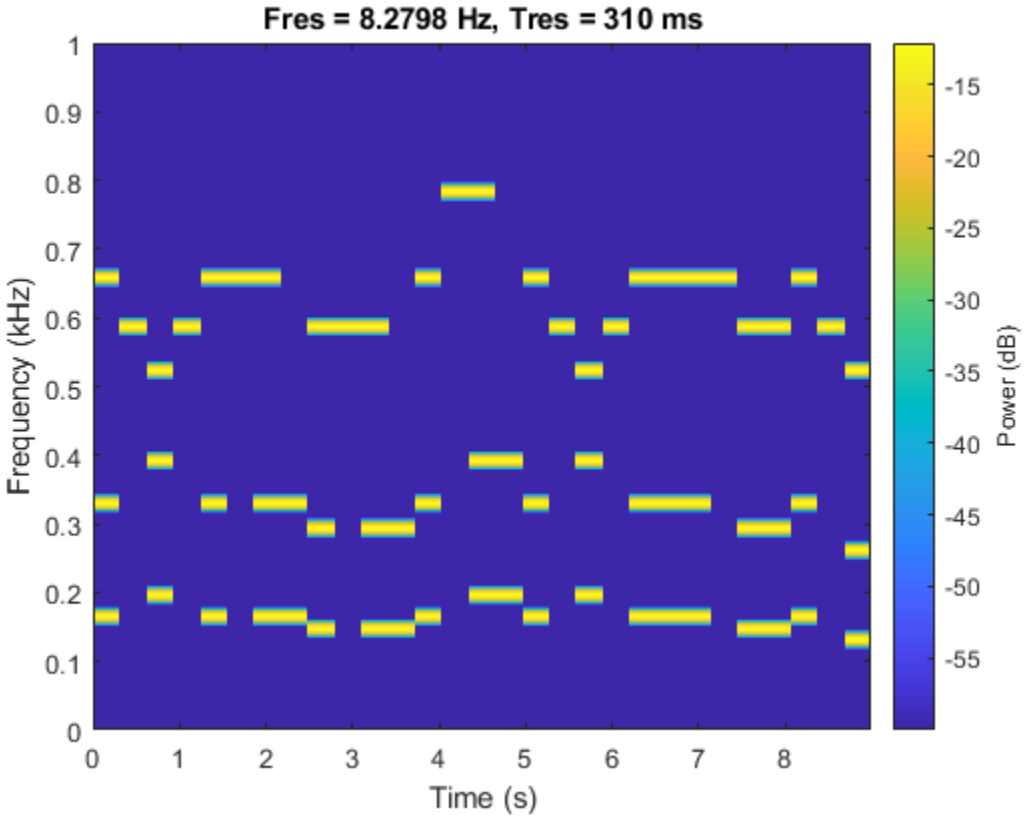


Lowpass Filtering of Musical Signal

Implement a basic digital music synthesizer and use it to play a traditional song. Specify a sample rate of 2 kHz. Plot the spectrogram of the song.

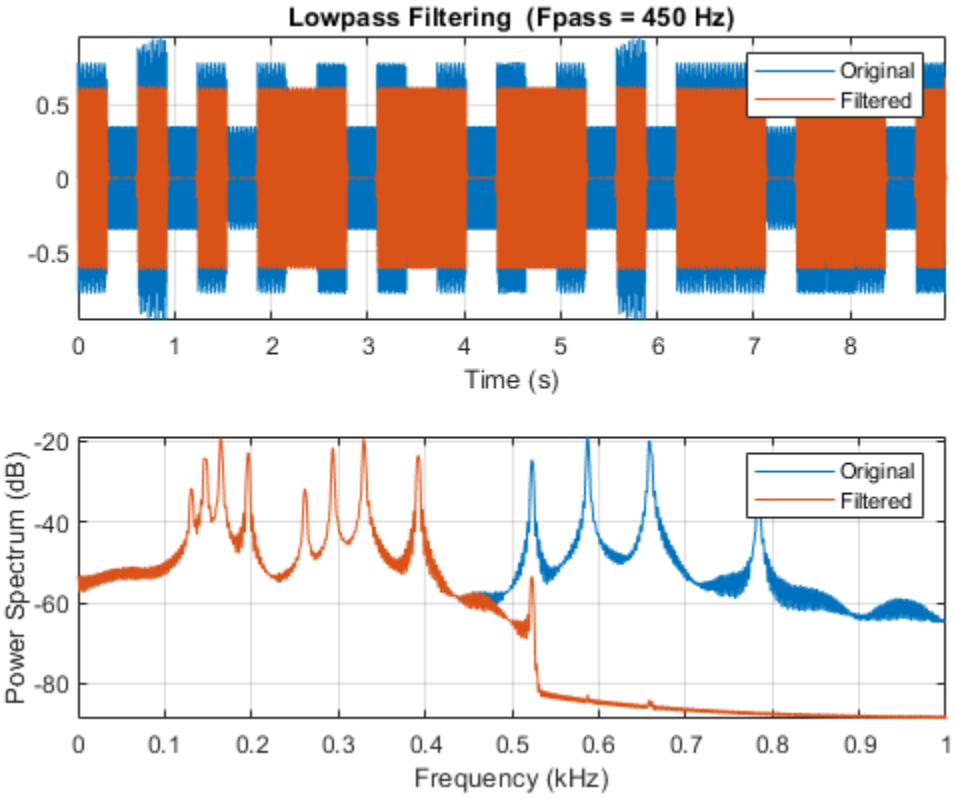
```
fs = 2e3;
t = 0:1/fs:0.3-1/fs;

l = [0 130.81 146.83 164.81 174.61 196.00 220 246.94];
m = [0 261.63 293.66 329.63 349.23 392.00 440 493.88];
h = [0 523.25 587.33 659.25 698.46 783.99 880 987.77];
note = @(f,g) [1 1 1]*sin(2*pi*[l(g) m(g) h(f)]'.*t);

mel = [3 2 1 2 3 3 3 0 2 2 2 0 3 5 5 0 3 2 1 2 3 3 3 3 2 2 3 2 1]+1;
acc = [3 0 5 0 3 0 3 3 2 0 2 2 3 0 5 5 3 0 5 0 3 3 3 0 2 2 3 0 1]+1;

song = [];
for kj = 1:length(mel)
    song = [song note(mel(kj),acc(kj)) zeros(1,0.01*fs)];
end
```

```
song = song/(max(abs(song))+0.1);

% To hear, type sound(song,fs)

pspectrum(song,fs,'spectrogram','TimeResolution',0.31, ...
    'OverlapPercent',0,'MinThreshold',-60)
```



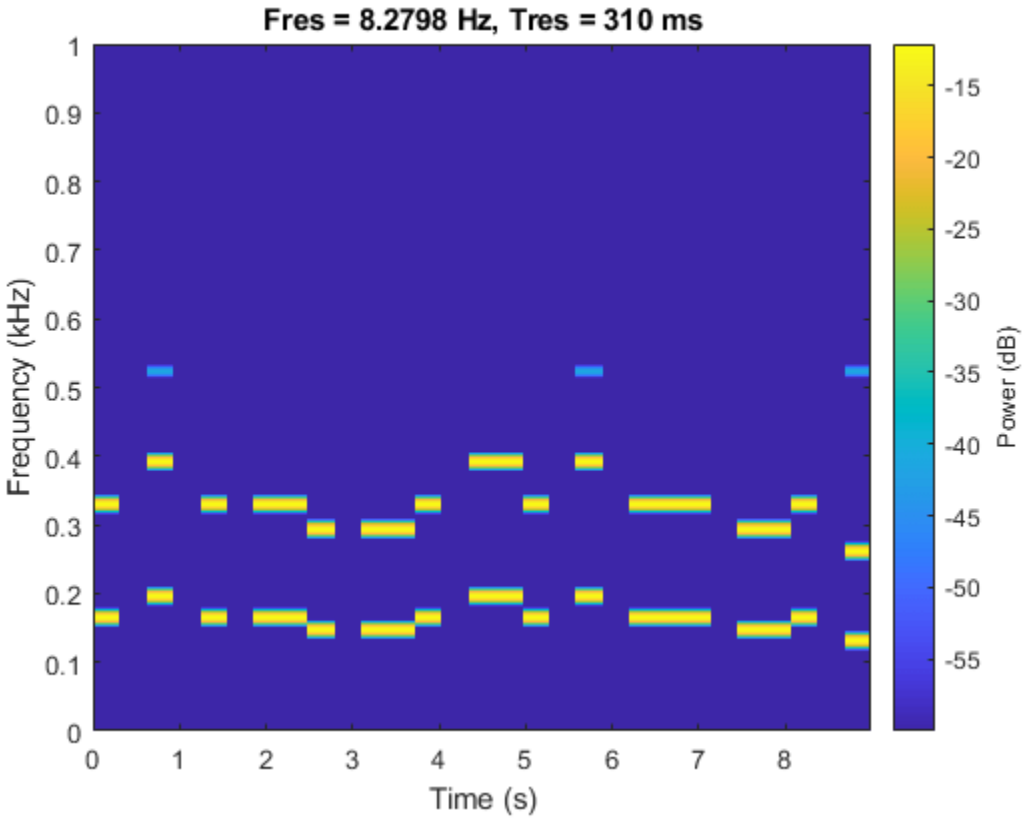
Lowpass-filter the signal to separate the melody from the accompaniment. Specify a passband frequency of 450 Hz. Plot the original and filtered signals in the time and frequency domains.

```
long = lowpass(song,450,fs);

% To hear, type sound(long,fs)

lowpass(song,450,fs)
```



Plot the spectrogram of the accompaniment.

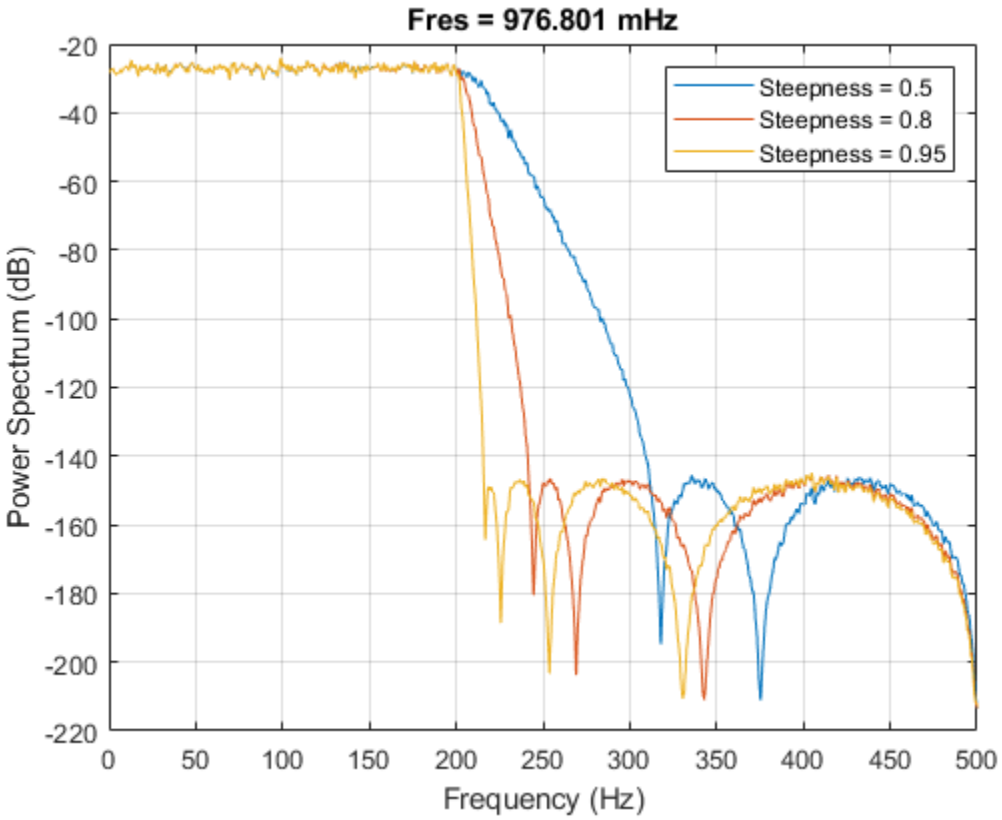
```
figure
pspectrum(long,fs,'spectrogram','TimeResolution',0.31, ...
    'OverlapPercent',0,'MinThreshold',-60)
```



Lowpass Filter Steepness

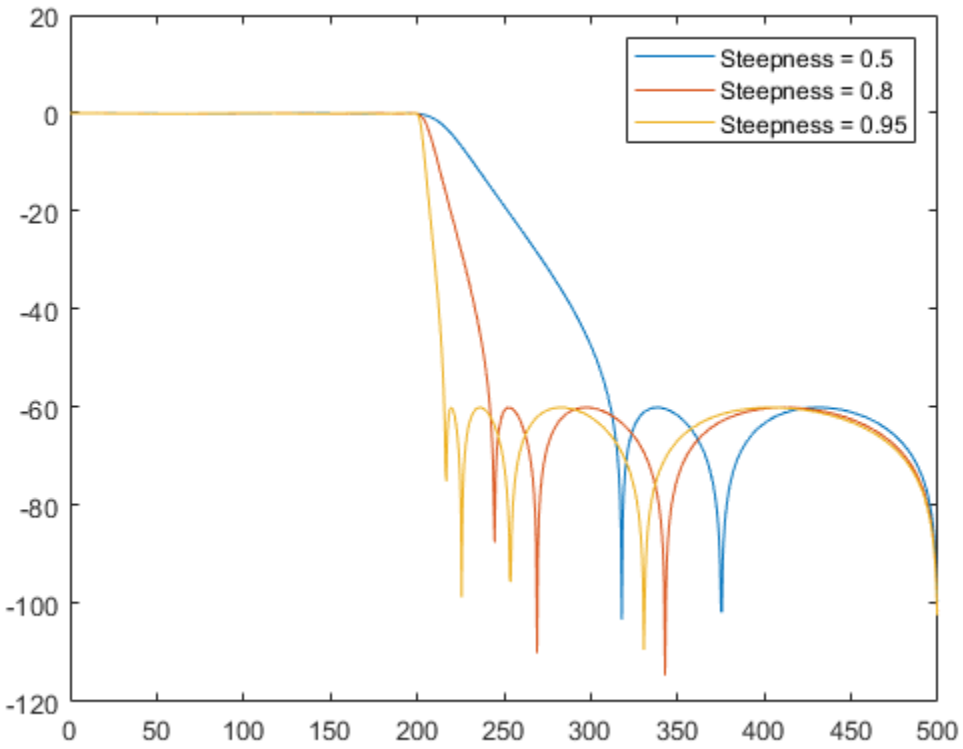
Filter white noise sampled at 1 kHz using an infinite impulse response lowpass filter with a passband frequency of 200 Hz. Use different steepness values. Plot the spectra of the filtered signals.

```
fs = 1000;
x = randn(20000,1);

[y1,d1] = lowpass(x,200,fs,'ImpulseResponse','iir','Steepness',0.5);
[y2,d2] = lowpass(x,200,fs,'ImpulseResponse','iir','Steepness',0.8);
[y3,d3] = lowpass(x,200,fs,'ImpulseResponse','iir','Steepness',0.95);

pspectrum([y1 y2 y3],fs)
legend('Steepness = 0.5','Steepness = 0.8','Steepness = 0.95')
```



Compute and plot the frequency responses of the filters.

```
[h1,f] = freqz(d1,1024,fs);
[h2,~] = freqz(d2,1024,fs);
[h3,~] = freqz(d3,1024,fs);

plot(f,mag2db(abs([h1 h2 h3])))
legend('Steepness = 0.5','Steepness = 0.8','Steepness = 0.95')
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix.

Example: `sin(2*pi*(0:127)/16)+randn(1,128)/100` specifies a noisy sinusoid

Example: `[2 1].*sin(2*pi*(0:127)'./[16 64])` specifies a two-channel sinusoid.

Data Types: `single` | `double`
Complex Number Support: Yes

wpass — Normalized passband frequency

scalar in (0, 1)

Normalized passband frequency, specified as a scalar in the interval (0, 1).

fpass — Passband frequency

scalar in (0, `fs`/2)

Passband frequency, specified as a scalar in the interval (0, `fs`/2).

fs — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar.

`xt` — Input timetable

timetable

Input timetable. `xt` must contain increasing, finite, and equally spaced row times of type `duration` in seconds.

If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

Example: `timetable(seconds(0:4)',randn(5,1),randn(5,2))` contains a single-channel random signal and a two-channel random signal, sampled at 1 Hz for 4 seconds.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ImpulseResponse','iir','StopbandAttenuation',30` filters the input using a minimum-order IIR filter that attenuates frequencies higher than `fpass` by 30 dB.

`ImpulseResponse` — Type of impulse response

`'auto'` (default) | `'fir'` | `'iir'`

Type of impulse response of the filter, specified as the comma-separated pair consisting of `'ImpulseResponse'` and `'fir'`, `'iir'`, or `'auto'`.

- `'fir'` — The function designs a minimum-order, linear-phase, finite impulse response (FIR) filter. To compensate for the delay, the function appends to the input signal $N/2$ zeros, where N is the filter order. The function then filters the signal and removes the first $N/2$ samples of the output.

  In this case, the input signal must be at least twice as long as the filter that meets the specifications.
- `'iir'` — The function designs a minimum-order infinite impulse response (IIR) filter and uses the `filtfilt` function to perform zero-phase filtering and compensate for the filter delay.

  If the signal is not at least three times as long as the filter that meets the specifications, the function designs a filter with smaller order and thus smaller steepness.
- `'auto'` — The function designs a minimum-order FIR filter if the input signal is long enough, and a minimum-order IIR filter otherwise. Specifically, the function follows these steps:

  - Compute the minimum order that an FIR filter must have to meet the specifications. If the signal is at least twice as long as the required filter order, design and use that filter.
  - If the signal is not long enough, compute the minimum order that an IIR filter must have to meet the specifications. If the signal is at least three times as long as the required filter order, design and use that filter.
  - If the signal is not long enough, truncate the order to one-third the signal length and design an IIR filter of that order. The reduction in order comes at the expense of transition band steepness.

- Filter the signal and compensate for the delay.

`Steepness` — Transition band steepness
`0.85` (default) | scalar in the interval [0.5, 1)

Transition band steepness, specified as the comma-separated pair consisting of `'Steepness'` and a scalar in the interval [0.5, 1). As the steepness increases, the filter response approaches the ideal lowpass response, but the resulting filter length and the computational cost of the filtering operation also increase. See “Lowpass Filter Steepness” on page 1-1329 for more information.

`StopbandAttenuation` — Filter stopband attenuation
`60` (default) | positive scalar in dB

Filter stopband attenuation, specified as the comma-separated pair consisting of `'StopbandAttenuation'` and a positive scalar in dB.

Output Arguments

`y` — Filtered signal
vector | matrix | timetable

Filtered signal, returned as a vector, a matrix, or a timetable with the same dimensions as the input.

`d` — Lowpass filter
`digitalFilter` object

Lowpass filter used in the filtering operation, returned as a `digitalFilter` object.

- Use `filter(d,x)` to filter a signal `x` using `d`.
- Use **FVTool** to visualize the filter response.
- Use `designfilt` to edit or generate a digital filter based on frequency-response specifications.

More About

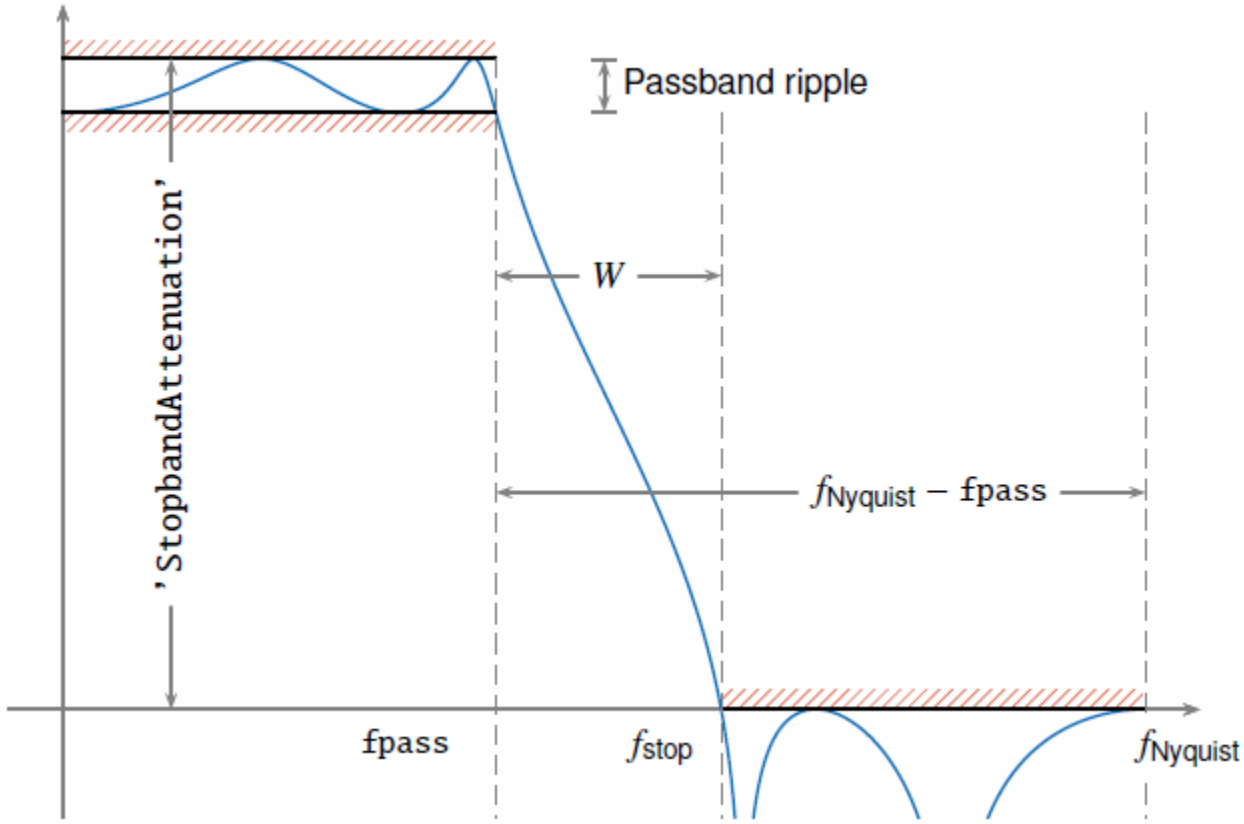
Lowpass Filter Steepness

The `'Steepness'` argument controls the width of a filter's transition region. The lower the steepness, the wider the transition region. The higher the steepness, the narrower the transition region.

To interpret the filter steepness, consider the following definitions:

- The Nyquist frequency, f_{Nyquist}, is the highest frequency component of a signal that can be sampled at a given rate without aliasing. f_{Nyquist} is 1 ($\times\pi$ rad/sample) when the input signal has no time information, and `fs`/2 hertz when the input signal is a timetable or when you specify a sample rate.
- The stopband frequency of the filter, f_{stop}, is the frequency beyond which the attenuation is equal to or greater than the value specified using `'StopbandAttenuation'`.
- The transition width of the filter, W, is f_{stop} – `fpass`, where `fpass` is the specified passband frequency.

- Most nonideal filters also attenuate the input signal across the passband. The maximum value of this frequency-dependent attenuation is called the passband ripple. Every filter used by `lowpass` has a passband ripple of 0.1 dB.



When you specify a value, s, for `'Steepness'`, the function computes the transition width as $W = (1 - s) \times (f_{Nyquist} - \texttt{fpass})$.

- When `'Steepness'` is equal to 0.5, the transition width is 50% of ($f_{Nyquist}$ – `fpass`).
- As `'Steepness'` approaches 1, the transition width becomes progressively narrower until it reaches a minimum value of 1% of ($f_{Nyquist}$ – `fpass`).
- The default value of `'Steepness'` is 0.85, which corresponds to a transition width that is 15% of ($f_{Nyquist}$ – `fpass`).

See Also

Apps
Signal Analyzer

Functions
`bandpass` | `bandstop` | `designfilt` | `filter` | `filtfilt` | `fir1` | `highpass`

Introduced in R2018a

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

`[bt,at] = lp2bp(b,a,Wo,Bw)` transforms an analog lowpass filter prototype given by polynomial coefficients (specified by row vectors `b` and `a`) into a bandpass filter with center frequency `Wo` and bandwidth `Bw`. The input system must be an analog filter prototype.

`[At,Bt,Ct,Dt] = lp2bp(A,B,C,D,Wo,Bw)` converts the continuous-time state-space lowpass filter prototype (specified by matrices `A`, `B`, `C`, and `D`) to a bandpass filter with center frequency `Wo` and bandwidth `Bw`. The input system must be an analog filter prototype.

Examples

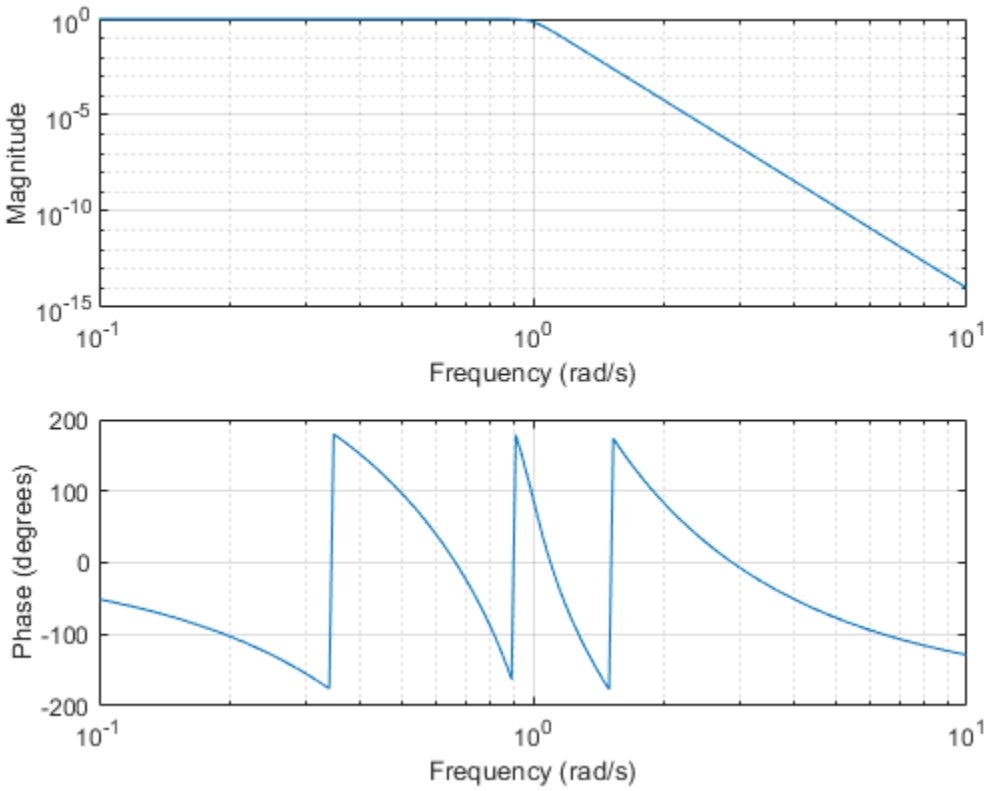
Lowpass to Bandpass Transformation

Design a 14th-order lowpass Butterworth analog filter prototype.

```
n = 14;
[z,p,k] = buttap(n);
```

Convert the prototype to transfer function form. Display its magnitude and frequency responses.

```
[b,a] = zp2tf(z,p,k);
freqs(b,a)
```

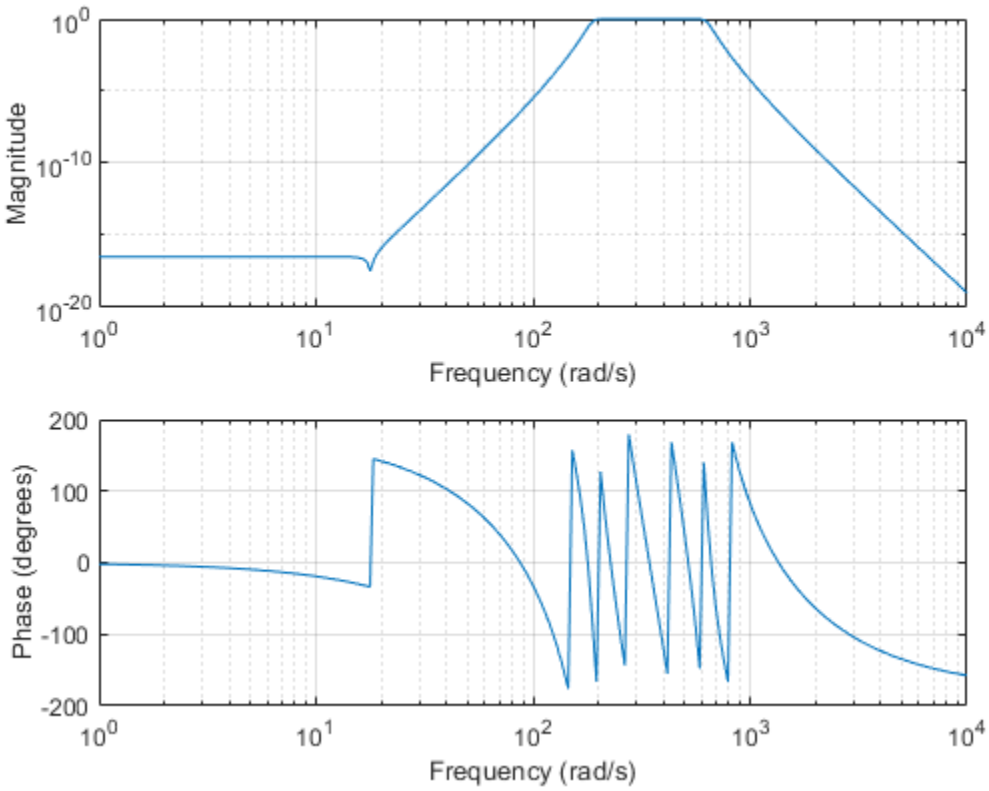


Transform the prototype to a bandpass filter with a passband from 30 Hz to 100 Hz. Specify the center frequency and bandwidth in rad/s.

```
fl = 30;
fh = 100;

Wo = 2*pi*sqrt(fl*fh); % center frequency
Bw = 2*pi*(fh-fl); % bandwidth

[bt,at] = lp2bp(b,a,Wo,Bw);
```

Display the magnitude and frequency responses of the transformed filter.

```
freqs(bt,at)
```



Input Arguments

b, a — Prototype numerator and denominator coefficients

row vectors

Prototype numerator and denominator coefficients, specified as row vectors. `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of s:

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

Data Types: `single` | `double`

A, B, C, D — Prototype state-space representation

matrices

Prototype state-space representation, specified as matrices. The state-space matrices relate the state vector x, the input u, and the output y through

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

Data Types: `single` | `double`

`Wo` — Center frequency
scalar

Center frequency, specified as a scalar. For a filter with lower band edge `w1` and upper band edge `w2`, use `Wo` = sqrt(`w1`*`w2`). Express `Wo` in units of rad/s.

Data Types: `single` | `double`

`Bw` — Bandwidth
scalar

Bandwidth, specified as a scalar. For a filter with lower band edge `w1` and upper band edge `w2`, use `Bw` = `w2`–`w1`. Express `Bw` in units of rad/s.

Data Types: `single` | `double`

Output Arguments

`bt`, `at` — Transformed numerator and denominator coefficients
row vectors

Transformed numerator and denominator coefficients, returned as row vectors.

`At`, `Bt`, `Ct`, `Dt` — Transformed state-space representation
matrices

Transformed state-space representation, returned as matrices.

Algorithms

`lp2bp` transforms analog lowpass filter prototypes with a cutoff angular frequency of 1 rad/s into bandpass filters with the 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` 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 has center frequency ω_0 and bandwidth B_w, the standard s-domain transformation is

$$s = Q(p^2 + 1)/p$$

where $Q = \omega_0/B_w$ and $p = s/\omega_0$. Substituting this for s in the Laplace transformed state-space equation and considering the operator p as d/dt results in

$$Q\ddot{x} + Qx = \dot{A}x + B\dot{u}$$

or

$$Q\ddot{x} - \dot{A}x - B\dot{u} = -Qx$$

Now define

$$Q\dot{\omega} = -Qx$$

which, when substituted, leads to

$$Q\dot{x} = Ax + Q\omega + Bu$$

The last two equations give equations of state. Write them in standard form and multiply the differential equations by ω_0 to recover the time or frequency scaling represented by p and find state matrices for the bandpass filter:

$$Q = \frac{\omega_0}{B_w}$$

$$At = \omega_0\left[\frac{A}{Q}\ \text{eye}(ma, m);\ -\text{eye}(ma, m)\ \text{zeros}(ma, m)\right]$$

$$Bt = \omega_0\left[\frac{B}{Q}; \text{zeros}(ma, n)\right]$$

$$Ct = [C\,\text{zeros}(mc, ma)]$$

$$Dt = D$$

where $[ma, m] = \text{size}(A)$.

`lp2bp` can perform the transformation on two different linear system representations: transfer function form and state-space form. If the input to `lp2bp` is in transfer function form, the function transforms it into state-space form before applying this algorithm.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The input transfer function coefficients, `num` and `den`, must be real.

See Also

`bilinear` | `impinvar` | `lp2bs` | `lp2hp` | `lp2lp`

Introduced before R2006a

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

`[bt,at] = lp2bs(b,a,Wo,Bw)` transforms an analog lowpass filter prototype given by polynomial coefficients (specified by row vectors `b` and `a`) into a bandstop filter with center frequency `Wo` and bandwidth `Bw`. The input system must be an analog filter prototype.

`[At,Bt,Ct,Dt] = lp2bs(A,B,C,D,Wo,Bw)` converts the continuous-time state-space lowpass filter prototype (specified by matrices `A`, `B`, `C`, and `D`) to a bandstop filter with center frequency `Wo` and bandwidth `Bw`. The input system must be an analog filter prototype.

Examples

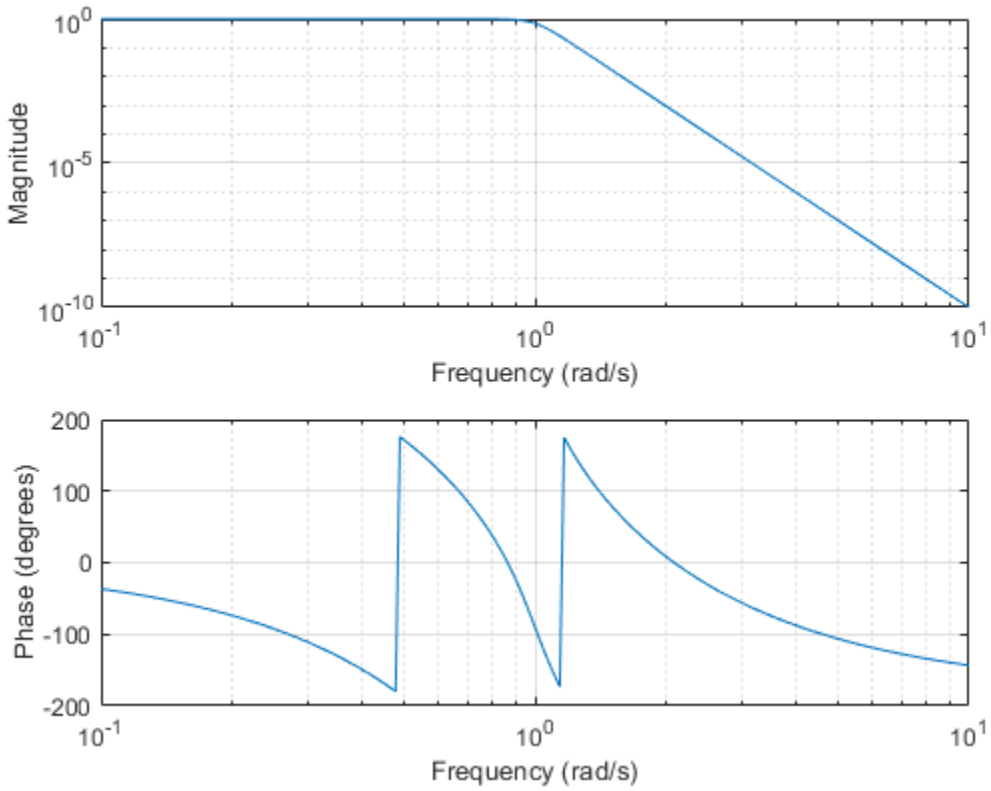
Lowpass to Bandstop Transformation

Design a 10th-order lowpass Butterworth analog filter prototype.

```
n = 10;
[z,p,k] = buttap(n);
```

Convert the prototype to transfer function form. Display its magnitude and frequency responses.

```
[b,a] = zp2tf(z,p,k);
freqs(b,a)
```

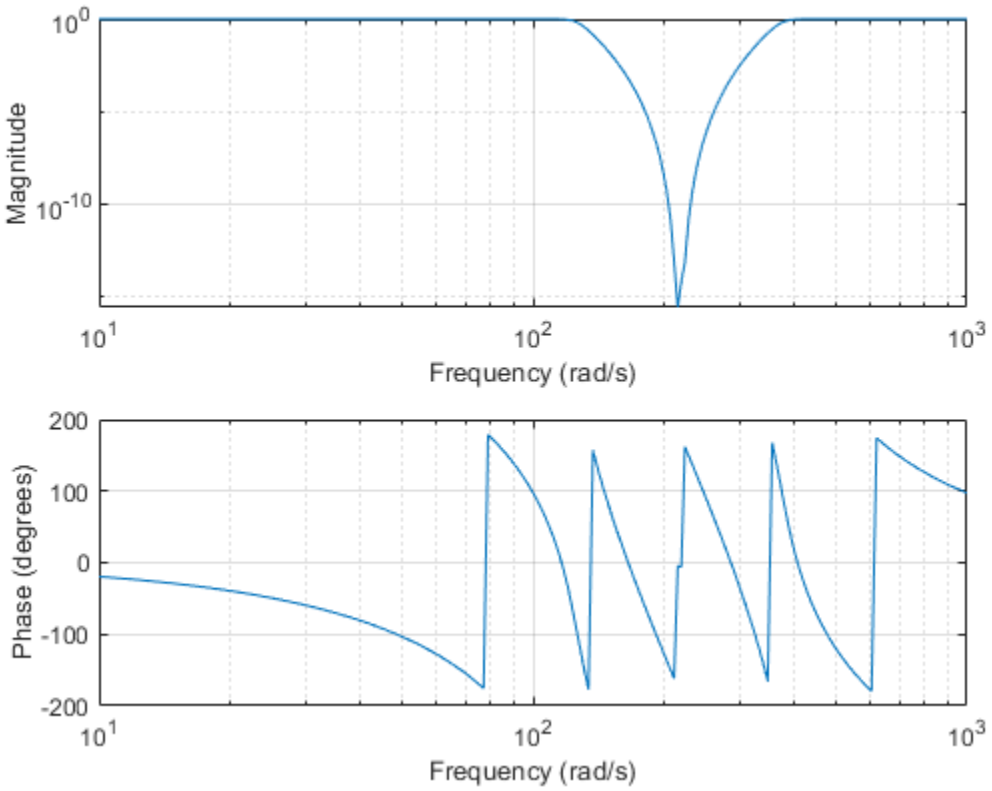


Transform the prototype to a bandstop filter with a stopband from 20 Hz to 60 Hz. Specify the center frequency and bandwidth in rad/s.

```
fl = 20;
fh = 60;

Wo = 2*pi*sqrt(fl*fh); % center frequency
Bw = 2*pi*(fh-fl); % bandwidth

[bt,at] = lp2bs(b,a,Wo,Bw);
```

Display the magnitude and frequency responses of the transformed filter.

```
freqs(bt,at)
```



Input Arguments

b, a — Prototype numerator and denominator coefficients

row vectors

Prototype numerator and denominator coefficients, specified as row vectors. `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of s:

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

Data Types: `single` | `double`

A, B, C, D — Prototype state-space representation

matrices

Prototype state-space representation, specified as matrices. The state-space matrices relate the state vector x, the input u, and the output y through

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

Data Types: `single` | `double`

`Wo` — Center frequency
scalar

Center frequency, specified as a scalar. For a filter with lower band edge `w1` and upper band edge `w2`, use `Wo` = sqrt(`w1`*`w2`). Express `Wo` in units of rad/s.

Data Types: `single` | `double`

`Bw` — Bandwidth
scalar

Bandwidth, specified as a scalar. For a filter with lower band edge `w1` and upper band edge `w2`, use `Bw` = `w2`–`w1`. Express `Bw` in units of rad/s.

Data Types: `single` | `double`

Output Arguments

`bt`, `at` — Transformed numerator and denominator coefficients
row vectors

Transformed numerator and denominator coefficients, returned as row vectors.

`At`, `Bt`, `Ct`, `Dt` — Transformed state-space representation
matrices

Transformed state-space representation, returned as matrices.

Algorithms

`lp2bs` transforms analog lowpass filter prototypes with a cutoff angular frequency of 1 rad/s into bandstop filters with the 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` is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a bandstop filter has center frequency ω_0 and bandwidth B_w, the standard s-domain transformation is

$$s = \frac{p}{Q(p^2 + 1)}$$

where $Q = \omega_0/B_w$ and $p = s/\omega_0$. The state-space version of this transformation is

$$Q = \frac{\omega_0}{B_w}$$

$$At = \left[\frac{\omega_0}{Q \cdot A^{-1}}\ \omega_0 \cdot \mathrm{eye}(ma);\ -\omega_0 \cdot \mathrm{eye}(ma)\ \mathrm{zeros}(ma)\right]$$

$$Bt = -\left[\frac{\omega_0}{Q(A \backslash B)}; \mathrm{zeros}(ma, n)\right]$$

$$Ct = \left[\frac{C}{A}\ \ \mathrm{zeros}(mc, ma)\right]$$

$$Dt = D - C/A \cdot B$$

`lp2bs` can perform the transformation on two different linear system representations: transfer function form and state-space form. See `lp2bp` for a derivation of the bandpass version of this transformation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The input transfer function coefficients, `num` and `den`, must be real.

See Also

`bilinear` | `impinvar` | `lp2bp` | `lp2hp` | `lp2lp`

Introduced before R2006a

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

`[bt,at] = lp2hp(b,a,Wo)` transforms an analog lowpass filter prototype given by polynomial coefficients (specified by row vectors `b` and `a`) into a highpass analog filter with cutoff angular frequency `Wo`. The input system must be an analog filter prototype.

`[At,Bt,Ct,Dt] = lp2hp(A,B,C,D,Wo)` converts the continuous-time state-space lowpass filter prototype (specified by matrices `A`, `B`, `C`, and `D`) to a highpass analog filter with cutoff angular frequency `Wo`. The input system must be an analog filter prototype.

Examples

Lowpass to Highpass Transformation

Design a 5th-order highpass elliptic filter with a cutoff frequency of 100 Hz, 3 dB of passband ripple, and 30 dB of stopband attenuation

Design the prototype. Convert the zero-pole-gain output to a transfer function.

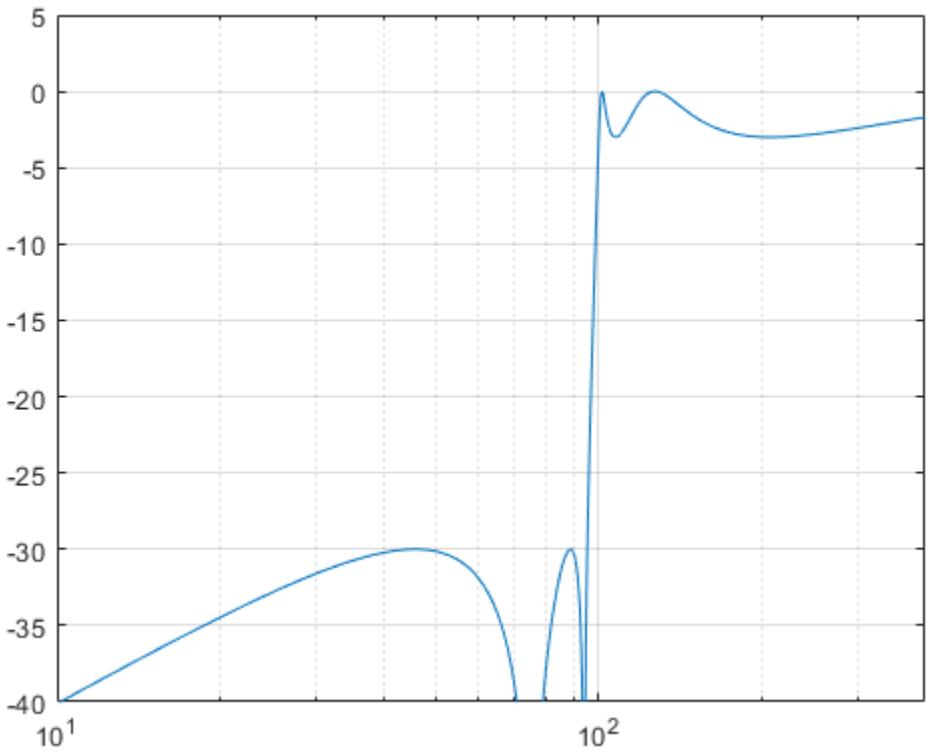
```
f = 100;

[ze,pe,ke] = ellipap(5,3,30);
[be,ae] = zp2tf(ze,pe,ke);
```

Transform the prototype to a highpass filter. Specify the cutoff frequency in rad/s.

```
[bh,ah] = lp2hp(be,ae,2*pi*f);
```

Compute and plot the frequency response of the filter. Divide the normalized frequency by 2π so the x-axis of the plot is in Hz.

```
[hh,wh] = freqs(bh,ah,4096);

semilogx(wh/2/pi,mag2db(abs(hh)))
axis([10 400 -40 5])
grid
```



Input Arguments

b, a — Prototype numerator and denominator coefficients

row vectors

Prototype numerator and denominator coefficients, specified as row vectors. `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of s:

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

Data Types: `single` | `double`

A, B, C, D — Prototype state-space representation

matrices

Prototype state-space representation, specified as matrices. The state-space matrices relate the state vector x, the input u, and the output y through

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

Data Types: `single` | `double`

`Wo` — Cutoff angular frequency
scalar

Cutoff angular frequency, specified as a scalar. Express the cutoff angular frequency in rad/s.

Data Types: `single` | `double`

Output Arguments

`bt`, `at` — Transformed numerator and denominator coefficients
row vectors

Transformed numerator and denominator coefficients, returned as row vectors.

`At`, `Bt`, `Ct`, `Dt` — Transformed state-space representation
matrices

Transformed state-space representation, returned as matrices.

Algorithms

`lp2hp` transforms analog lowpass filter prototypes with a cutoff angular frequency of 1 rad/s into highpass filters with a desired cutoff angular frequency. The transformation is one step in the digital filter design process for the `butter`, `cheby1`, `cheby2`, and `ellip` functions.

`lp2hp` is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a highpass filter is to have a cutoff angular frequency ω_0, the standard s-domain transformation is

$$s = \frac{\omega_0}{p}.$$

The state-space version of this transformation is:

```
At = Wo*inv(A);
Bt = -Wo*(A\B);
Ct = C/A;
Dt = D - C/A*B;
```

See `lp2bp` for a derivation of the bandpass version of this transformation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The input transfer function coefficients, `num` and `den`, must be real.

See Also

`bilinear` | `impinvar` | `lp2bp` | `lp2bs` | `lp2lp`

Introduced before R2006a

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

`[bt,at] = lp2lp(b,a,Wo)` transforms an analog lowpass filter prototype given by polynomial coefficients (specified by row vectors `b` and `a`) into a lowpass filter with cutoff angular frequency `Wo`. The input system must be an analog filter prototype.

`[At,Bt,Ct,Dt] = lp2lp(A,B,C,D,Wo)` converts the continuous-time state-space lowpass filter prototype (specified by matrices `A`, `B`, `C`, and `D`) to a lowpass filter with cutoff angular frequency `Wo`. The input system must be an analog filter prototype.

Examples

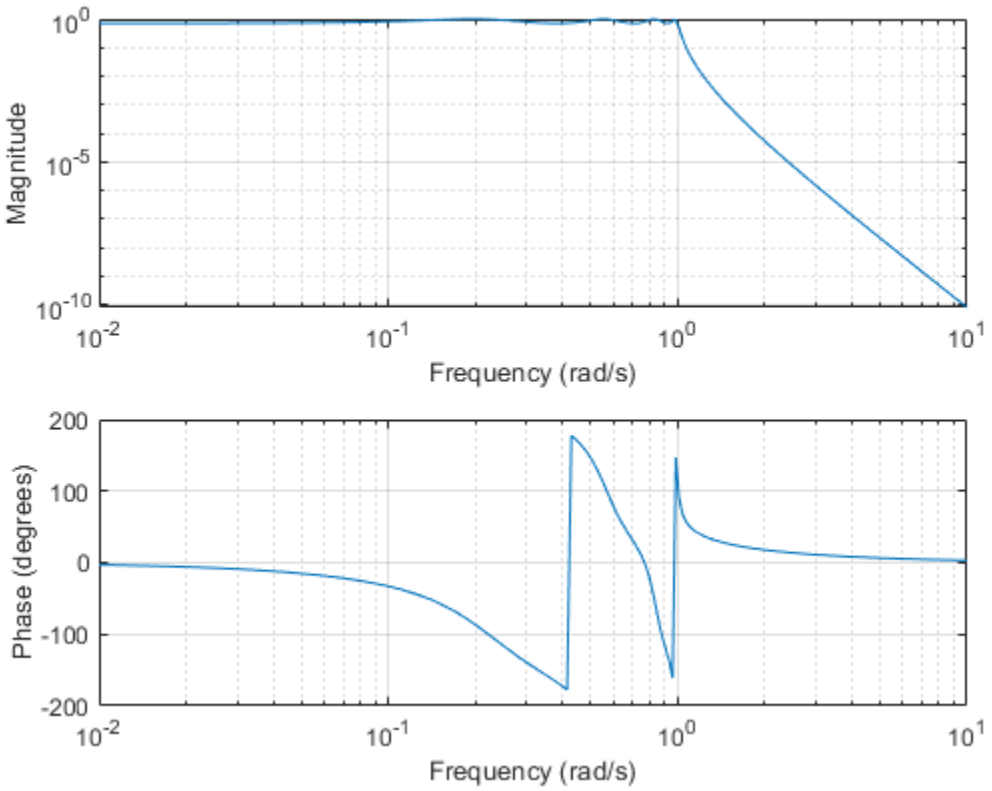
Lowpass to Lowpass Transformation

Design an 8th-order Chebyshev Type I analog lowpass filter prototype with 3 dB of ripple in the passband.

```
[z,p,k] = cheb1ap(8,3);
```

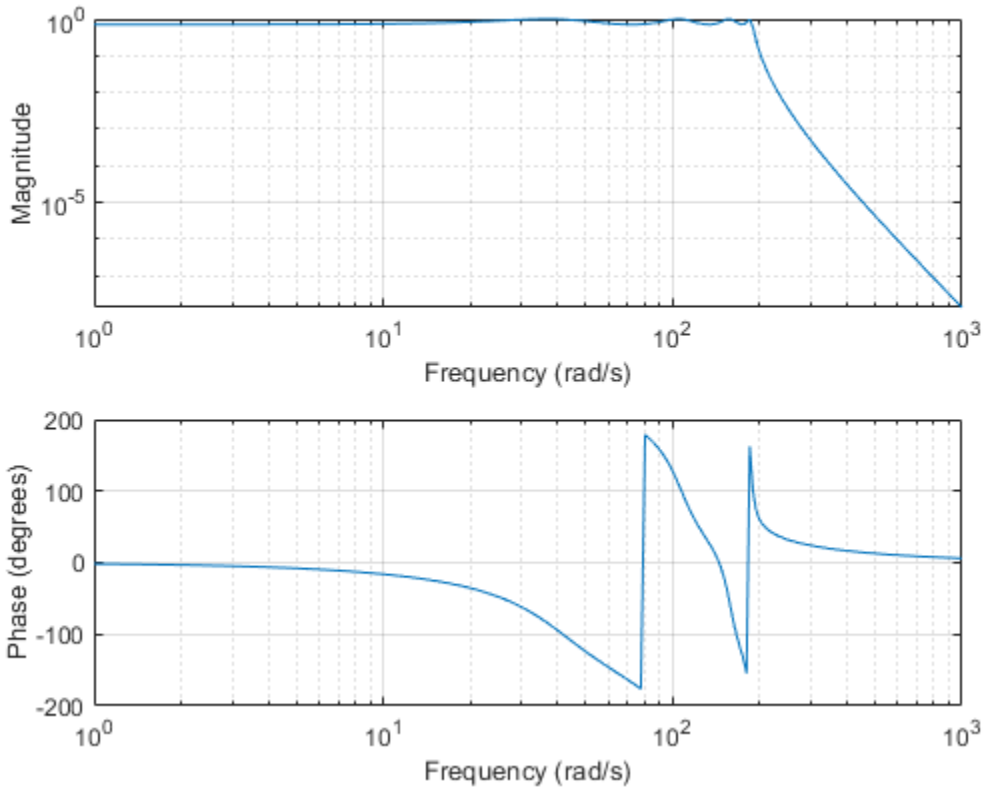
Convert the prototype to transfer function form and display its magnitude and frequency responses.

```
[b,a] = zp2tf(z,p,k);
freqs(b,a)
```



Transform the prototype to a lowpass filter with a cutoff frequency of 30 Hz. Specify the cutoff frequency in rad/s. Display the magnitude and frequency responses of the transformed filter.

```
Wo = 2*pi*30;

[bt,at] = lp2lp(b,a,Wo);
freqs(bt,at)
```



Input Arguments

b, a — Prototype numerator and denominator coefficients

row vectors

Prototype numerator and denominator coefficients, specified as row vectors. `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of s:

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

Data Types: `single` | `double`

A, B, C, D — Prototype state-space representation

matrices

Prototype state-space representation, specified as matrices. The state-space matrices relate the state vector x, the input u, and the output y through

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

Data Types: `single` | `double`

Wo — Cutoff angular frequency
scalar

Cutoff angular frequency, specified as a scalar. Express `Wo` in units of rad/s.

Data Types: `single` | `double`

Output Arguments

bt, at — Transformed numerator and denominator coefficients
row vectors

Transformed numerator and denominator coefficients, returned as row vectors.

At, Bt, Ct, Dt — Transformed state-space representation
matrices

Transformed state-space representation, returned as matrices.

Algorithms

`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.

`lp2lp` is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a lowpass filter has cutoff angular frequency ω_0, the standard s-domain transformation is

$$s = p/\omega_0$$

The state-space version of this transformation is

$$At = \omega_0 \cdot A$$

$$Bt = \omega_0 \cdot B$$

$$Ct = C$$

$$Dt = D$$

The `lp2lp` function can perform the transformation on two different linear system representations: transfer function form and state-space form. See `lp2bp` for a derivation of the bandpass version of this transformation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The input transfer function coefficients, `num` and `den`, must be real.

See Also

`bilinear` | `impinvar` | `lp2bp` | `lp2bs` | `lp2hp`

Introduced before R2006a

lpc

Linear prediction filter coefficients

Syntax

```
[a,g] = lpc(x,p)
```

Description

`[a,g] = lpc(x,p)` finds the coefficients of a `p`th-order linear predictor, an FIR filter that predicts the current value of the real-valued time series `x` based on past samples. The function also returns `g`, the variance of the prediction error. If `x` is a matrix, the function treats each column as an independent channel.

Examples

Estimate Series Using 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 (AR) process driven by normalized white Gaussian noise. Use the last 4096 samples of the AR process output to avoid startup transients.

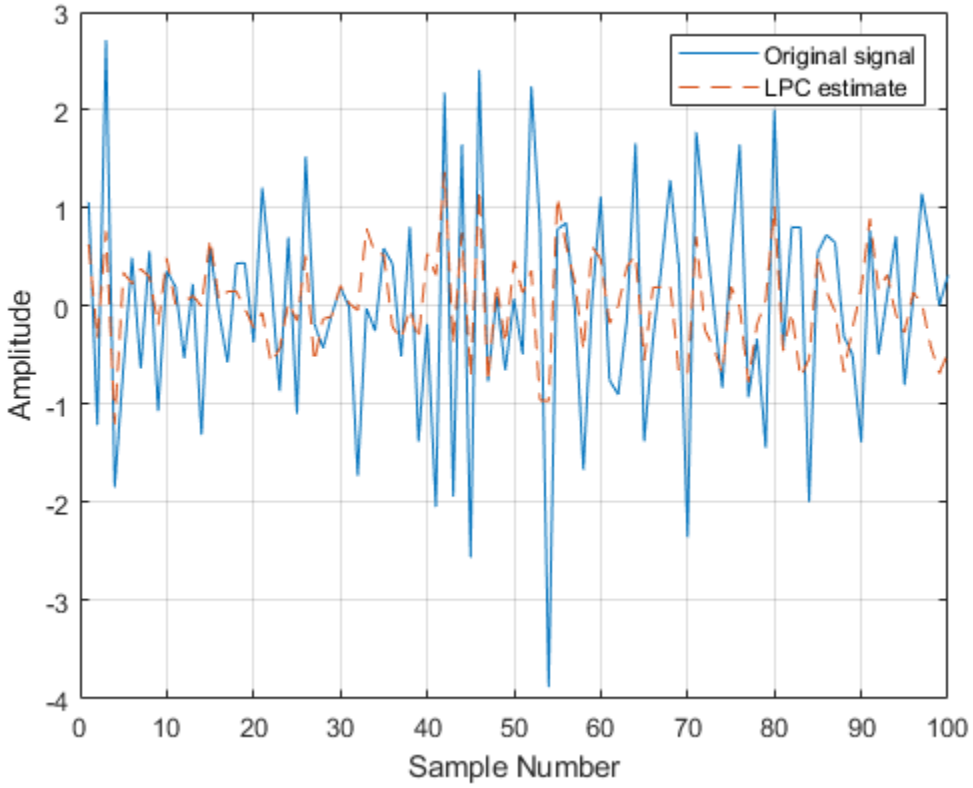
```
noise = randn(50000,1);
x = filter(1,[1 1/2 1/3 1/4],noise);
x = x(end-4096+1:end);
```

Compute the predictor coefficients and the estimated signal.

```
a = lpc(x,3);
est_x = filter([0 -a(2:end)],1,x);
```

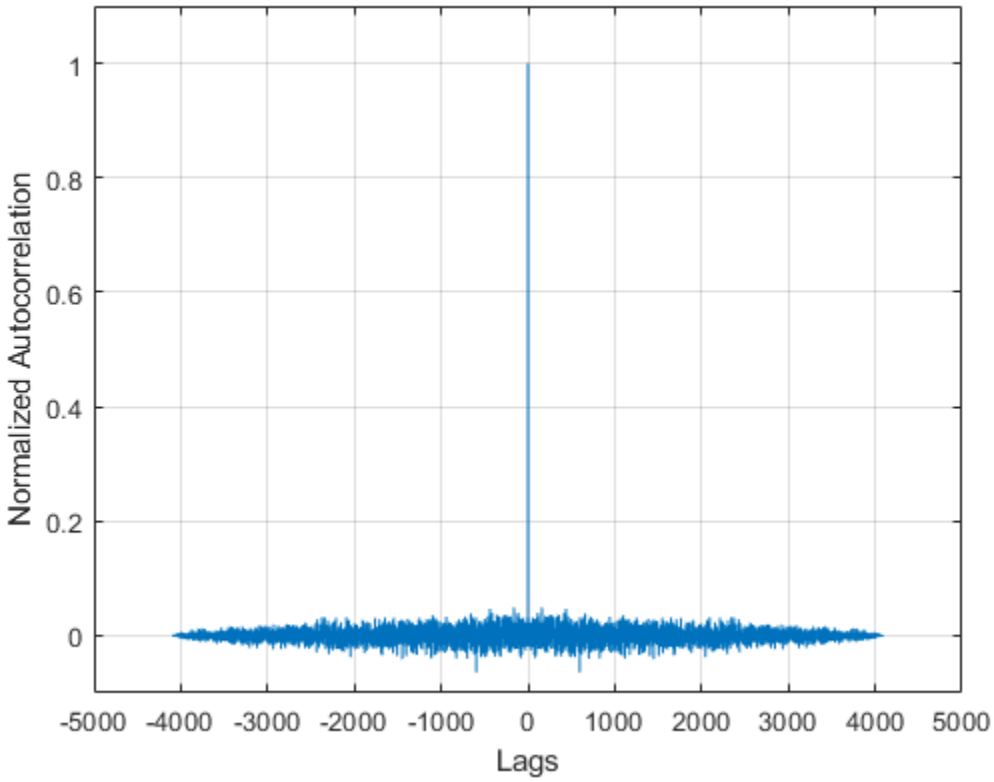
Compare the predicted signal to the original signal by plotting the last 100 samples of each.

```
plot(1:100,x(end-100+1:end),1:100,est_x(end-100+1:end),'--')
grid
xlabel('Sample Number')
ylabel('Amplitude')
legend('Original signal','LPC estimate')
```



Compute the prediction error and the autocorrelation sequence of the prediction error. Plot the autocorrelation. The prediction error is approximately white Gaussian noise, as expected for a third-order AR input process.

```
e = x-est_x;
[acs,lags] = xcorr(e,'coeff');

plot(lags,acs)
grid
xlabel('Lags')
ylabel('Normalized Autocorrelation')
ylim([-0.1 1.1])
```



Input Arguments

x — Input array

vector | matrix

Input array, specified as a vector or matrix. If `x` is a matrix, then the function treats each column as an independent channel.

p — Prediction filter polynomial order

`length(x)-1` (default) | positive integer

Prediction filter polynomial order, specified as a positive integer. `p` must be less than or equal to the length of `x`.

Output Arguments

a — Linear predictor coefficients

row vector | matrix

Linear predictor coefficients, returned as a row vector or a matrix. The coefficients relate the past `p` samples of `x` to the current value:

$$\widehat{x}(n) = -a(2)x(n-1) - a(3)x(n-2) - \cdots - a(p+1)x(n-p).$$

g — Prediction error variance

scalar | vector

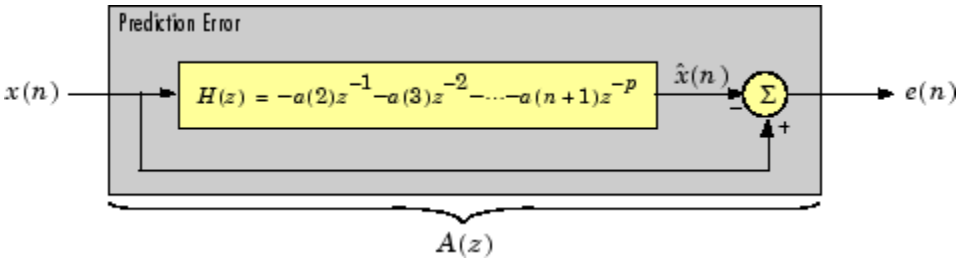
Prediction error variance, returned as a scalar or vector.

More About

Prediction Error

The prediction error, $e(n)$, can be viewed as the output of the prediction error filter $A(z)$, where

- $H(z)$ is the optimal linear predictor.
- $x(n)$ is the input signal.
- $\hat{x}(n)$ is the predicted signal.



Algorithms

`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.

`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, because the autocorrelation method implicitly windows the data. In other words, the method assumes that signal samples beyond the length of `x` are 0.

`lpc` computes the least-squares solution to $Xa = b$, where

$$X = \begin{bmatrix} x(1) & 0 & \cdots & 0 \\ x(2) & x(1) & \cdots & \vdots \\ \vdots & x(2) & \cdots & 0 \\ x(m) & \vdots & \vdots & x(1) \\ 0 & x(m) & \cdots & x(2) \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & x(m) \end{bmatrix}, \; a = \begin{bmatrix} 1 \\ a(2) \\ \vdots \\ a(p+1) \end{bmatrix}, \; b = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

and m is the length of x. Solving the least-squares problem using the normal equations $X^H Xa = X^H b$ leads to the Yule-Walker equations

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \cdots & \vdots \\ \vdots & \vdots & \ddots & r(2)^* \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(p+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(p+1) \end{bmatrix},$$

where `r = [`$r(1)$ $r(2)$ ... $r(p+1)$`]` is an autocorrelation estimate for `x` computed using `xcorr`. The Levinson-Durbin algorithm (see `levinson`) solves the Yule-Walker equations in $O(p^2)$ flops.

References

[1] Jackson, L. B. *Digital Filters and Signal Processing*. 2nd Edition. Boston: Kluwer Academic Publishers, 1989, pp. 255–257.

See Also

`aryule` | `levinson` | `prony` | `pyulear` | `stmcb`

Introduced before R2006a

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 = 1×6

    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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`ac2poly` | `poly2lsf` | `rc2poly`

Introduced before R2006a

mag2db

Convert magnitude to decibels

Syntax

```
ydb = mag2db(y)
```

Description

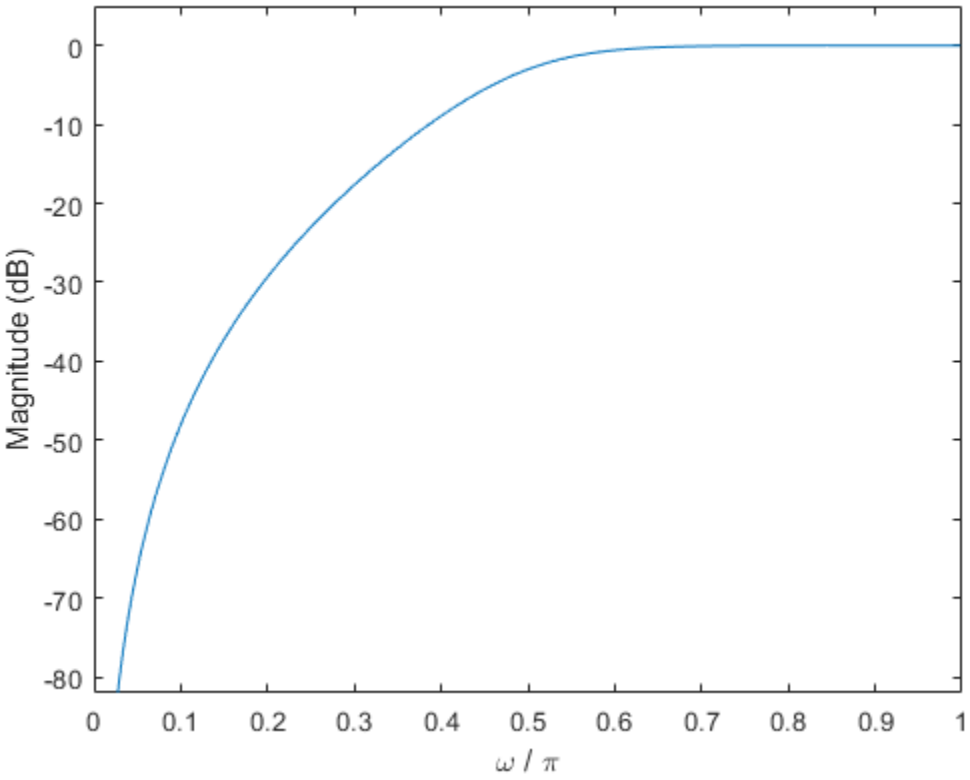
`ydb = mag2db(y)` expresses in decibels (dB) the magnitude measurements specified in `y`. The relationship between magnitude and decibels is `ydb` = 20 $\log_{10}$(`y`).

Examples

Magnitude Response of a Highpass Filter

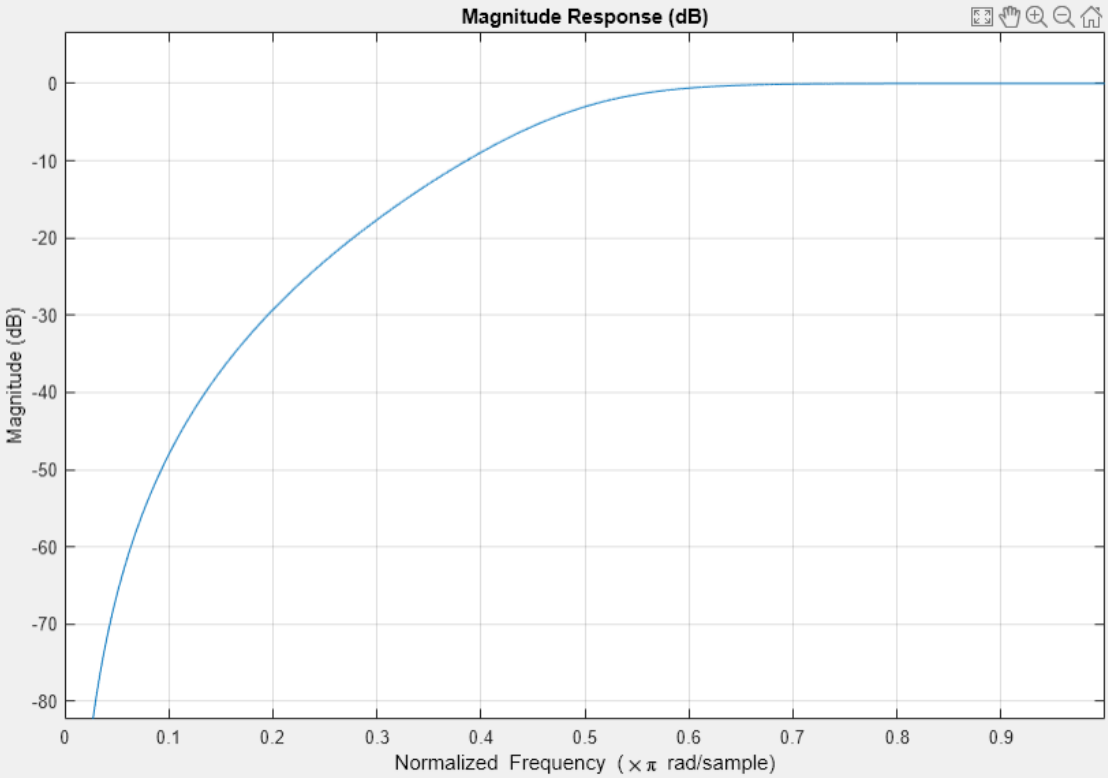
Design a 3rd-order highpass Butterworth filter having a normalized 3-dB frequency of 0.5π rad/sample. Compute its frequency response. Express the magnitude response in decibels and plot it.

```
[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`

Introduced in R2008a

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 order 1 for noncentrality parameter `a` and argument `b`. Use this syntax to calculate the special case of the generalized Marcum *Q*-function of order *m* with *m* = 1. For the definition of the generalized Marcum *Q*-function, see “Generalized Marcum Q-function” on page 1-1362.

`Q = marcumq(a,b,m)` computes the generalized Marcum *Q*-function of order `m`.

Examples

Generate and Plot Marcum *Q*-Function Data

Specify a range of values over which to generate data for the Marcum *Q*-function.

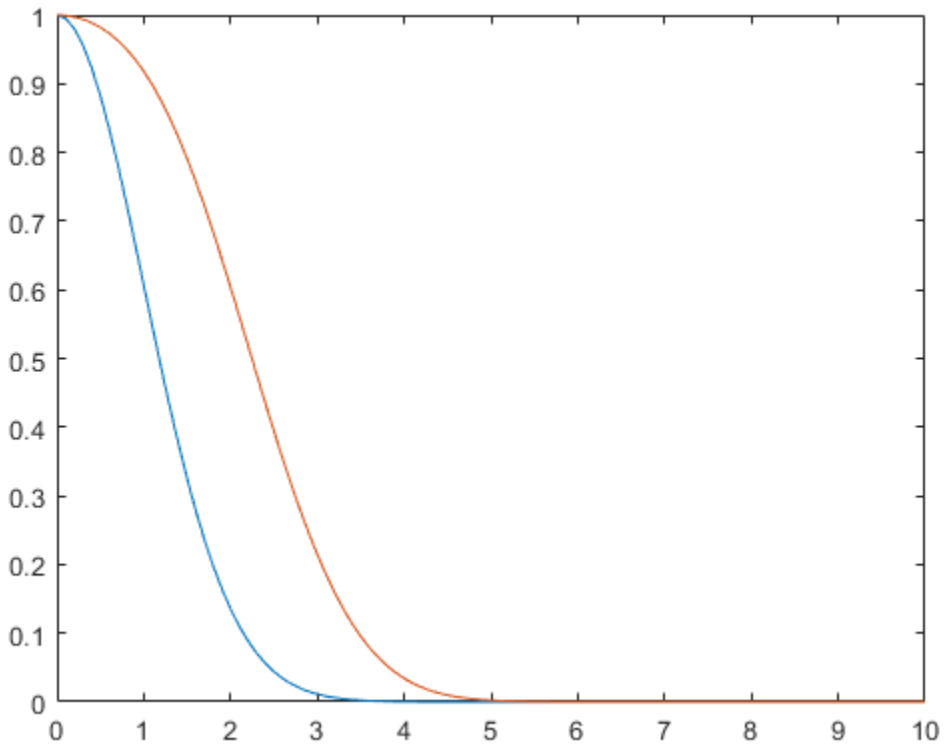
```
b = (0:0.1:10)';
```

Generate two output vectors comprising values of the Marcum *Q*-function of order one with noncentrality parameter values of `0` and `2`.

```
Q1 = marcumq(0,b);
Q2 = marcumq(2,b);
```

Plot the resultant Marcum *Q*-functions.

```
plot(b,[Q1 Q2])
```



Input Arguments

a — Noncentrality parameter

nonnegative scalar | array of nonnegative numbers

Noncentrality parameter, specified as an array of nonnegative numbers.

Note If you specify any input as an array, you must specify all other inputs as arrays of the same size or as scalars. If you specify any of the inputs as a scalar, the function expands that input to match the size of the other inputs.

Data Types: `double`

b — Argument of Marcum *Q*-function

nonnegative scalar | array of nonnegative numbers

Argument of the Marcum *Q*-function, specified as an array of nonnegative numbers.

Data Types: `double`

`m` — Order of generalized Marcum *Q*-function

positive integer | array of positive integers

Order of the generalized Marcum *Q*-function, specified as an array of positive integers.

Data Types: `double`

Output Arguments

`Q` — Value of Marcum *Q*-function

scalar in the range [0, 1] | array of numbers in the range [0, 1]

Value of the Marcum *Q*-function, returned as an array of numbers in the range [0, 1]. The size of this output matches the size of the input arguments.

Data Types: `double`

More About

Generalized Marcum *Q*-function

The generalized Marcum *Q*-function of order *m* is defined by

where I_{m-1} is the modified Bessel function of the first kind of order $m - 1$.

Algorithms

To compute the value of the generalized Marcum *Q*-function for the specified inputs, this function uses a calculation based on a method illustrated by Shnidman in [3]. This paper describes two error criteria: a relative error criterion and an absolute error criterion. This function uses the absolute error criterion.

References

[1] Cantrell, P., and A. Ojha. “Comparison of Generalized *Q*-function Algorithms (Corresp.).” *IEEE Transactions on Information Theory* 33, no. 4 (July 1987): 591–96. https://doi.org/10.1109/TIT.1987.1057323.

[2] Marcum, J. “A Statistical Theory of Target Detection by Pulsed Radar.” *IEEE Transactions on Information Theory* 6, no. 2 (April 1960): 59–267. https://doi.org/10.1109/TIT.1960.1057560.

[3] Shnidman, D.A. “The Calculation of the Probability of Detection and the Generalized Marcum *Q*-function,” *IEEE Transactions on Information Theory* 35, no. 2 (March 1989): 389–400. https://doi.org/10.1109/18.32133.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`besseli`

Introduced in R2008a

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,designflag)
```

Description

`[b,a] = maxflat(n,m,Wn)` returns the coefficients `b` and `a` of a lowpass Butterworth filter with normalized cutoff frequency `Wn`.

`b = maxflat(n,'sym',Wn)` returns the coefficients `b` of a symmetric FIR Butterworth filter. `n` must be even.

`[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)`).

`[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`.

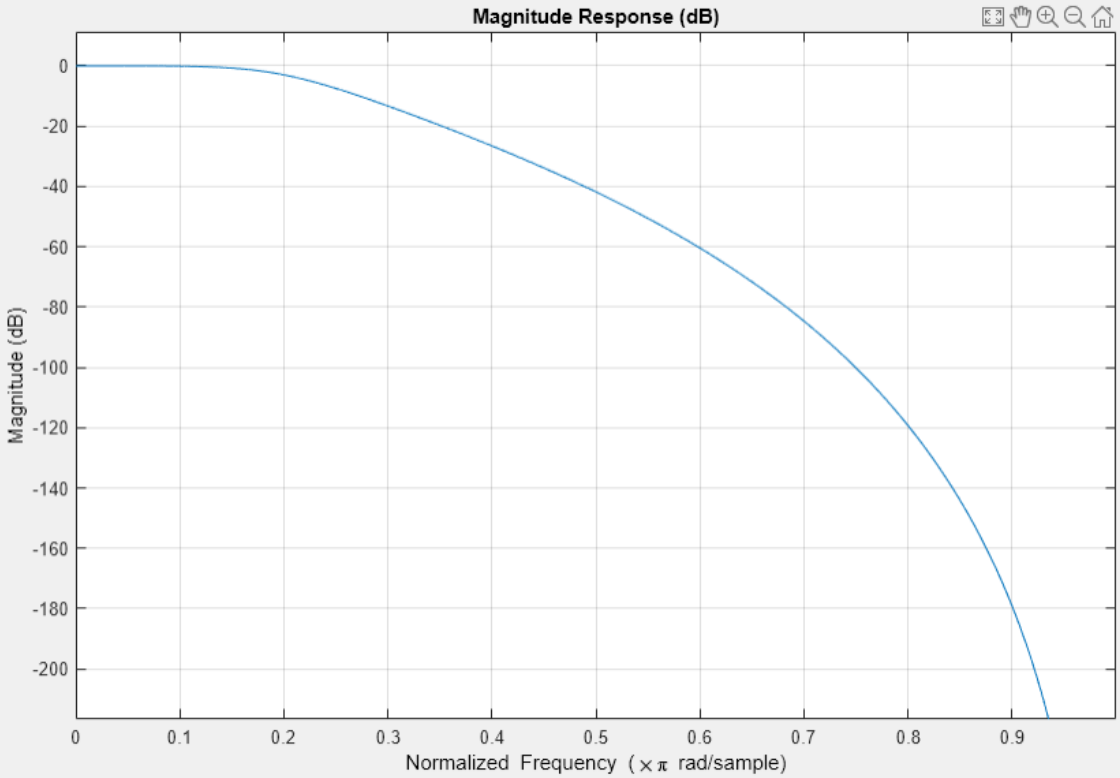
`[___] = maxflat(n,m,Wn,designflag)` specifies the option to display the filter design as a table, plot, or both using `designflag`. You can use any of the output combinations from previous syntaxes.

Examples

Generalized Butterworth Filter

Design a generalized Butterworth filter with normalized cutoff frequency 0.2π 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)
```

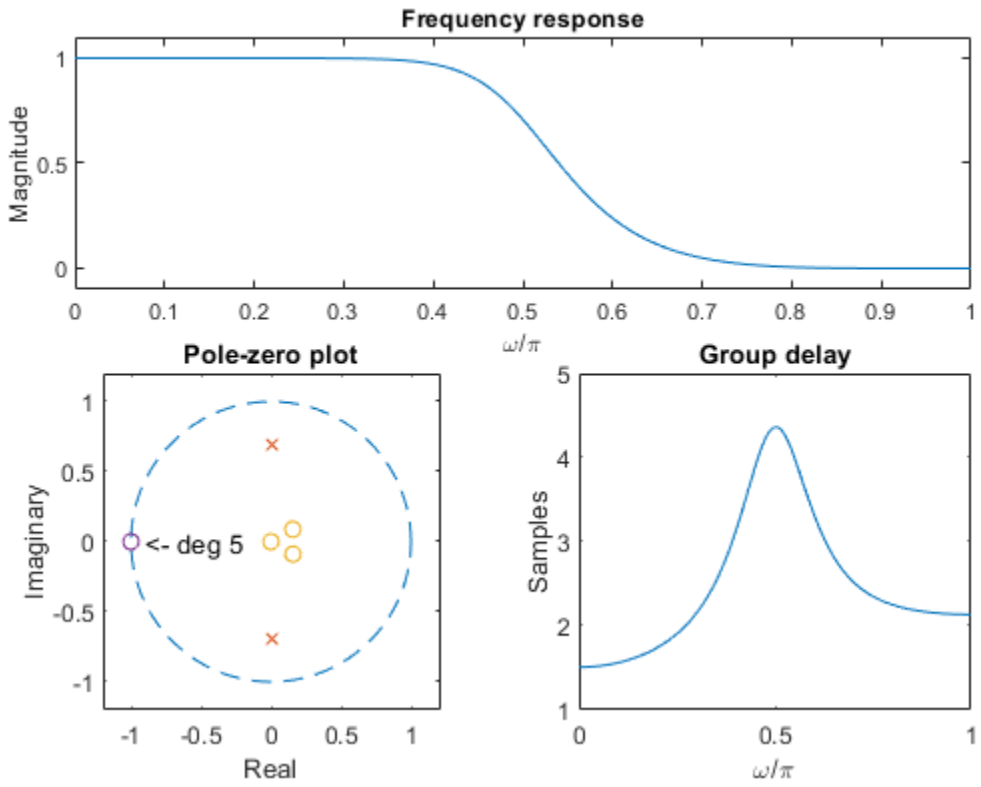


Monitor Filter Design with Display Option

Design a generalized Butterworth filter with normalized cutoff frequency 0.5π rad/s. Specify a numerator order of 8 and a denominator order of 2. Display the design table and the plots of the filter characteristics.

```
n = 8;
m = 2;
Wn = 0.5;
b = maxflat(n,m,Wn,'both');
```

```
 Table:

    L         M         N         wo_min/pi wo_max/pi

    8.0000         0    2.0000         0    0.2707
    7.0000    1.0000    2.0000    0.2707    0.3710
    6.0000    2.0000    2.0000    0.3710    0.4581
    5.0000    3.0000    2.0000    0.4581    0.5419
    4.0000    4.0000    2.0000    0.5419    0.6290
    3.0000    5.0000    2.0000    0.6290    0.7293
    2.0000    6.0000    2.0000    0.7293    1.0000
```



Input Arguments

`n` — Numerator coefficient order
real positive scalar

Numerator coefficient order, specified as a real positive scalar.

Data Types: `single` | `double`

`m` — Denominator coefficient order
real positive scalar

Denominator coefficient order, specified as a real positive scalar.

Data Types: `single` | `double`

`Wn` — Normalized cutoff frequency
scalar in the range [0, 1]

Normalized cutoff frequency at which the magnitude response of the filter is equal to $1/\sqrt{2}$, specified as a scalar in the range [0, 1], where 1 corresponds to the Nyquist frequency.

Data Types: `single` | `double`

`designflag` — Filter design display
`'trace'` | `'plots'` | `'both'`

Filter design display, specified as one of these values:

- `'trace'` for a textual display of the design table used in the design
- `'plots'` for plots of the filter magnitude, group delay, and zeros and poles
- `'both'` for both the textual display and plots

Output Arguments

b — Numerator coefficients
vector

Numerator coefficients, returned as a vector.

a — Denominator coefficients
vector

Denominator coefficients, returned as a vector.

b1, b2 — Polynomials
vectors

Polynomials, returned as vectors. The product of `b1` and `b2` is equal to the numerator polynomial `b`. `b1` contains all of the zeros at `z = -1`, and `b2` contains all of the other zeros.

sos — Second-order section coefficients
matrix

Second-order section coefficients, returned as a matrix.

g — Gain
real-valued scalar

Gain of the filter, returned as a real-valued scalar.

References

[1] Selesnick, Ivan W., and C. Sidney Burrus. “Generalized Digital Butterworth Filter Design.” *IEEE Transactions on Signal Processing* 46, no. 6, (June 1998): 1688–94.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

`butter` | `filter` | `freqz`

Introduced before R2006a

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. This syntax can include any combination of input arguments from previous syntaxes, as long as the second input argument is either `fs` or `f`. If the second input is passed as empty, normalized frequency will be assumed. 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

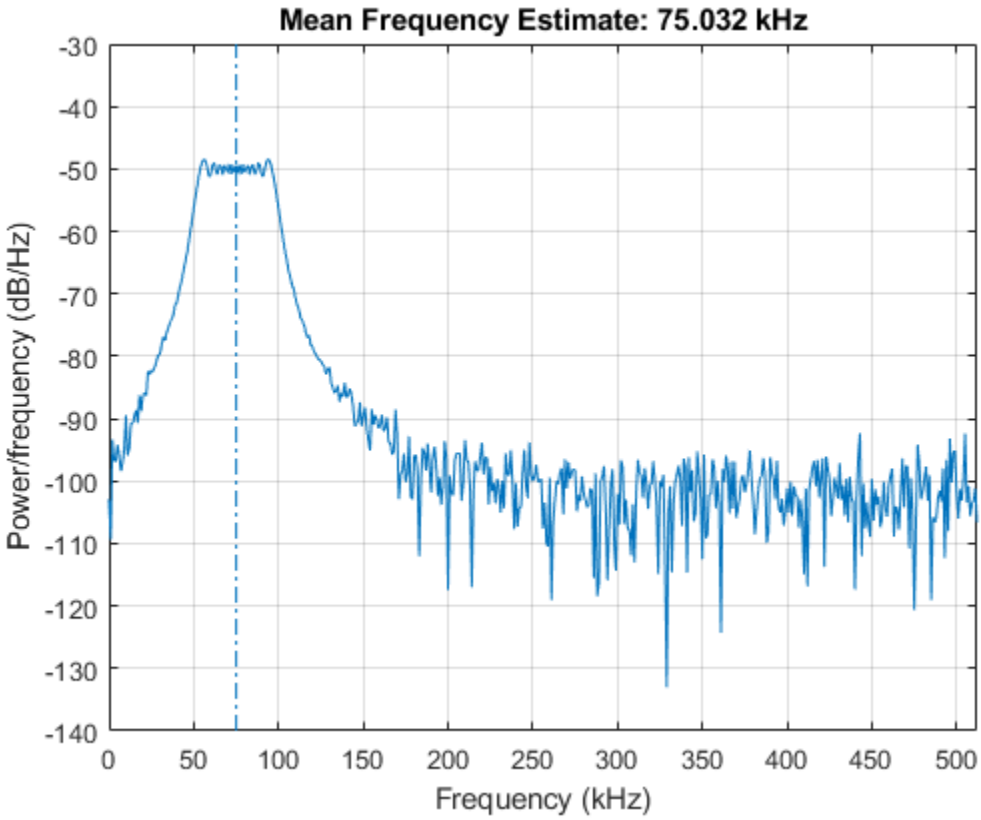
Mean Frequency of Chirps

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);
```

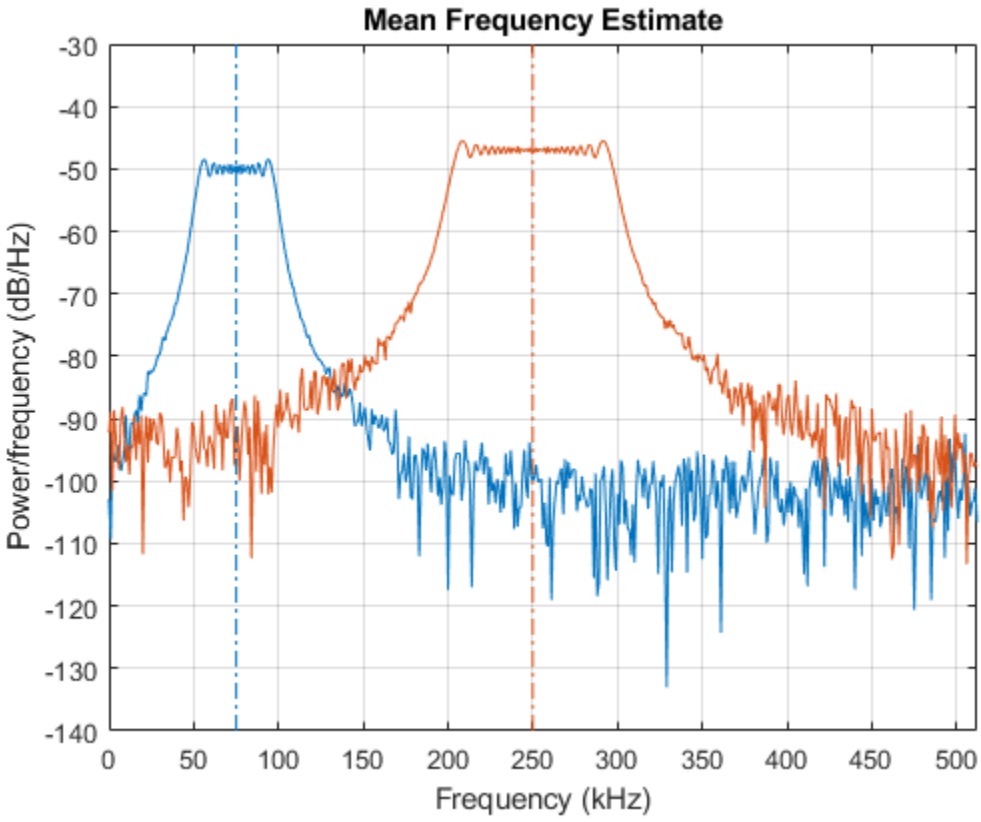
Concatenate the chirps to produce a two-channel signal. Estimate the mean frequency of each channel.

```
y = meanfreq([x x2],Fs)
```

```
y = 1×2
10^5 ×

    0.7503    2.4999
```

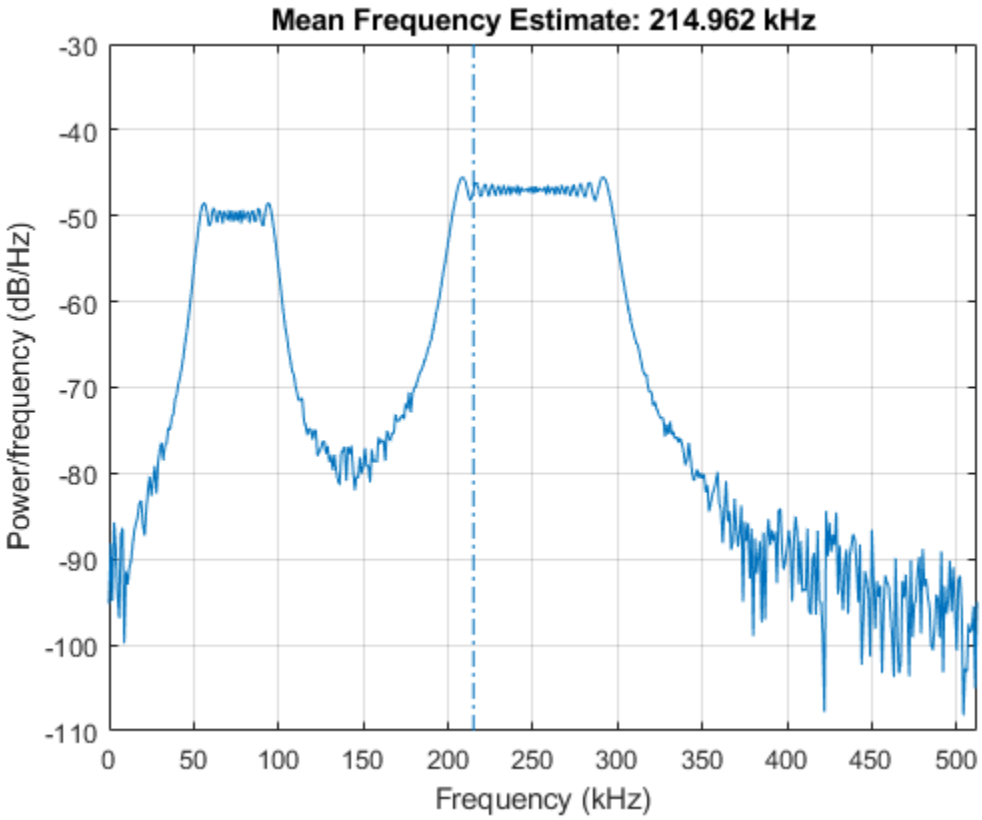
Plot the PSDs of the two channels and annotate their mean frequencies.

```
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
```

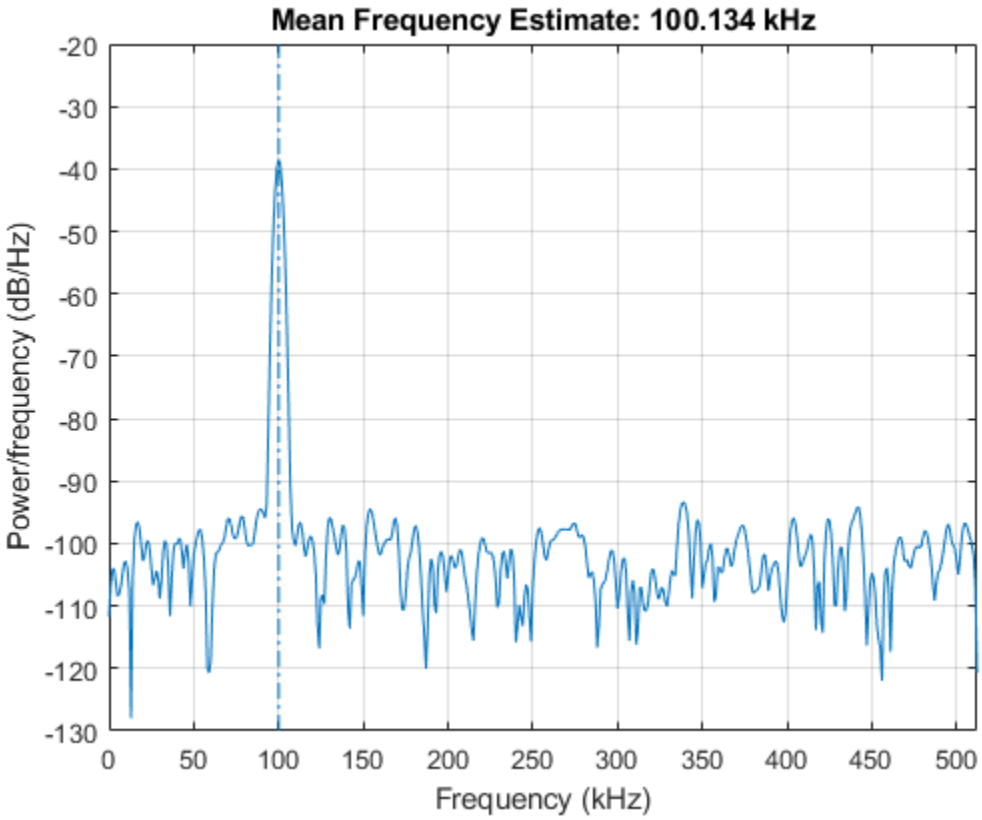
Mean Frequency of Sinusoids

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.

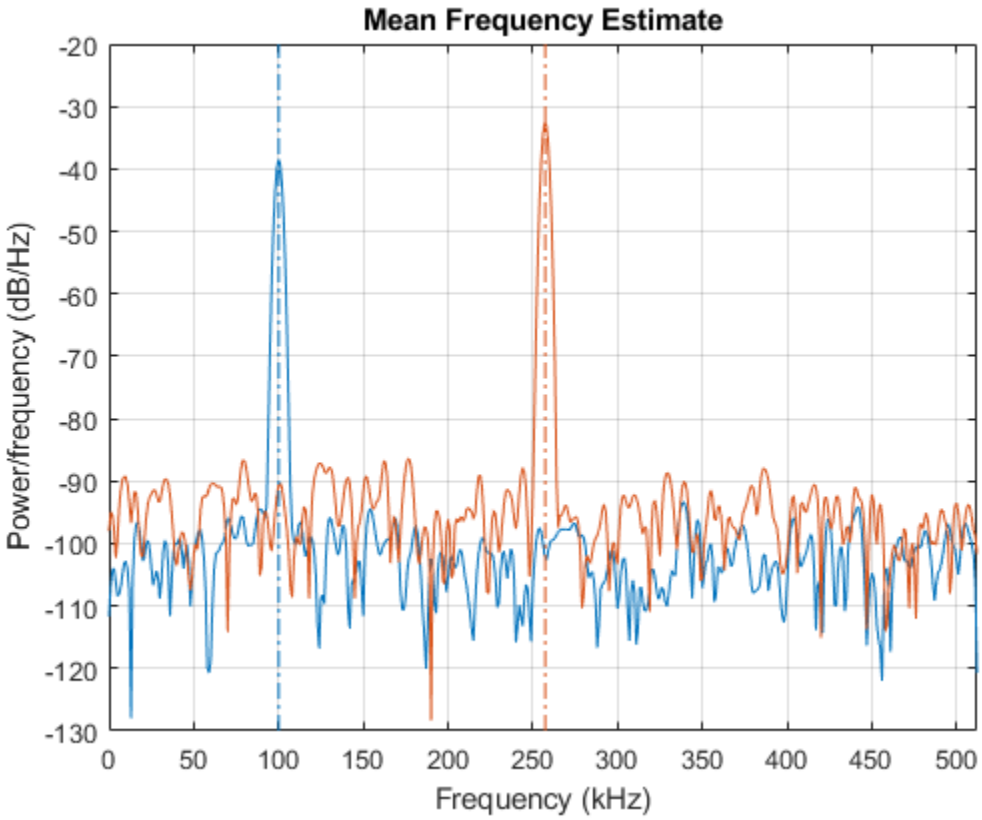
```
[Pyy,f] = periodogram([x x2],kaiser(nSamp,38),[],Fs);
```

```
y = meanfreq(Pyy,f)
```

```
y = 1×2
10^5 ×

    1.0013    2.5732
```

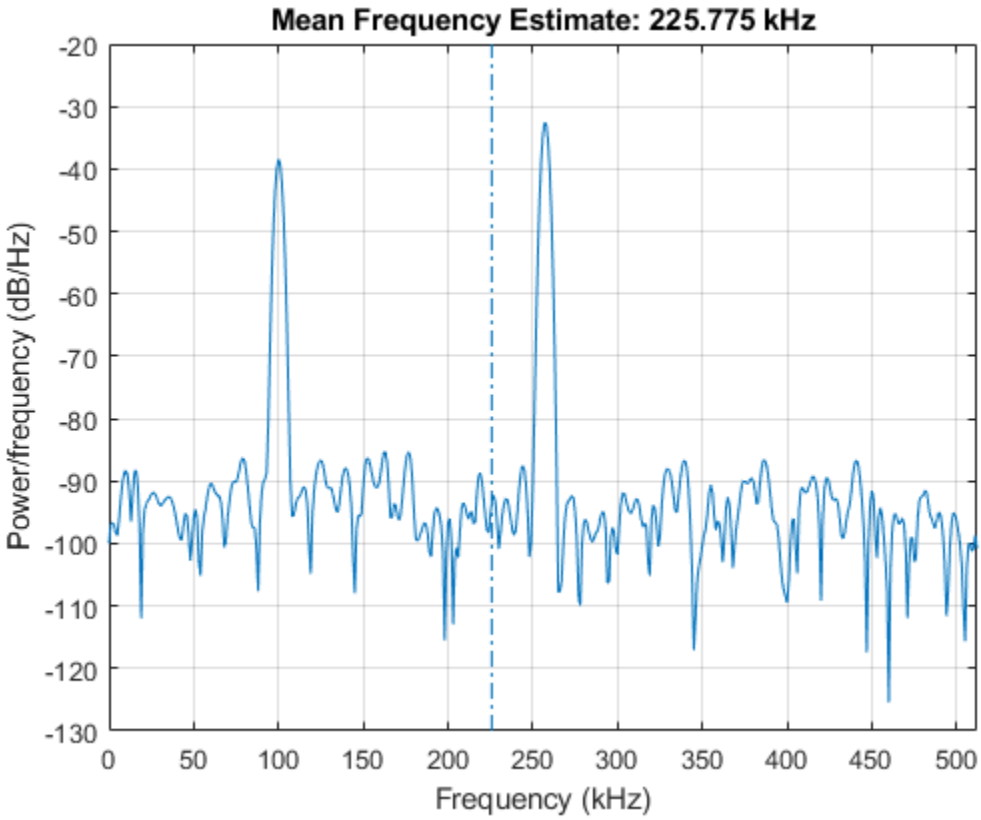
Annotate the mean frequencies of the two channels on a plot of the PSDs.

```
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);
```



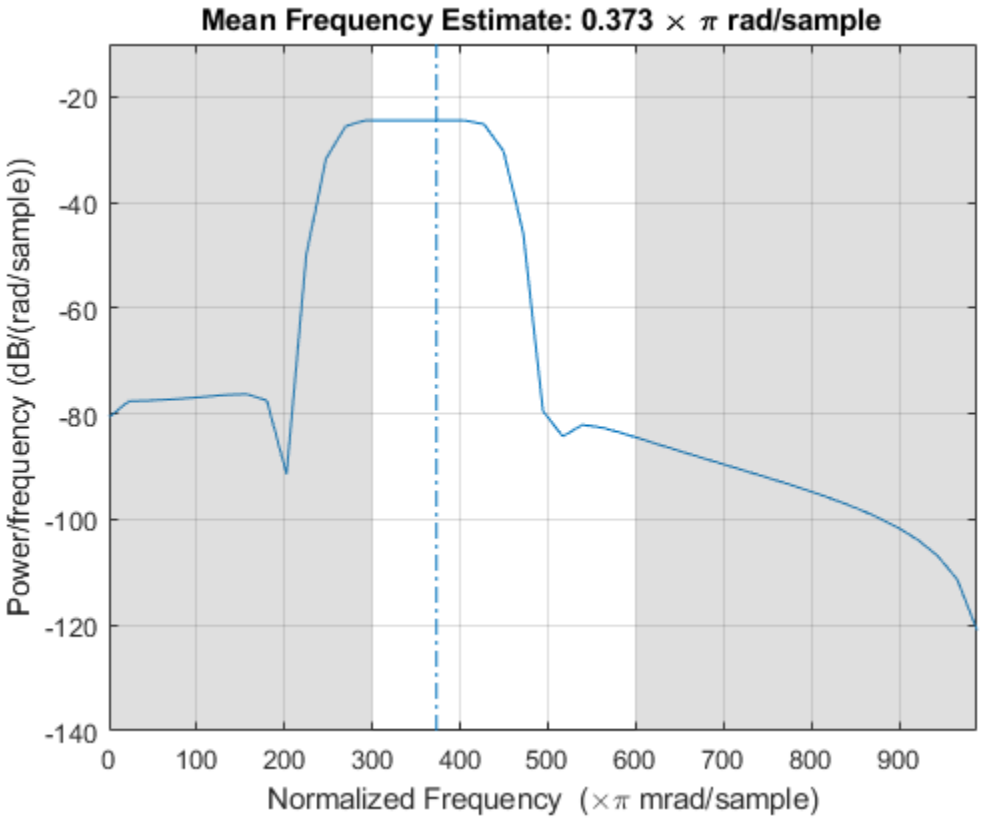
Mean Frequency of Bandlimited Signals

Generate a signal whose PSD resembles the frequency response of an 88th-order bandpass FIR filter with normalized cutoff frequencies 0.25π rad/sample and 0.45π rad/sample.

```
d = fir1(88,[0.25 0.45]);
```

Compute the mean frequency of the signal between 0.3π rad/sample and 0.6π 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 2π 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)
```

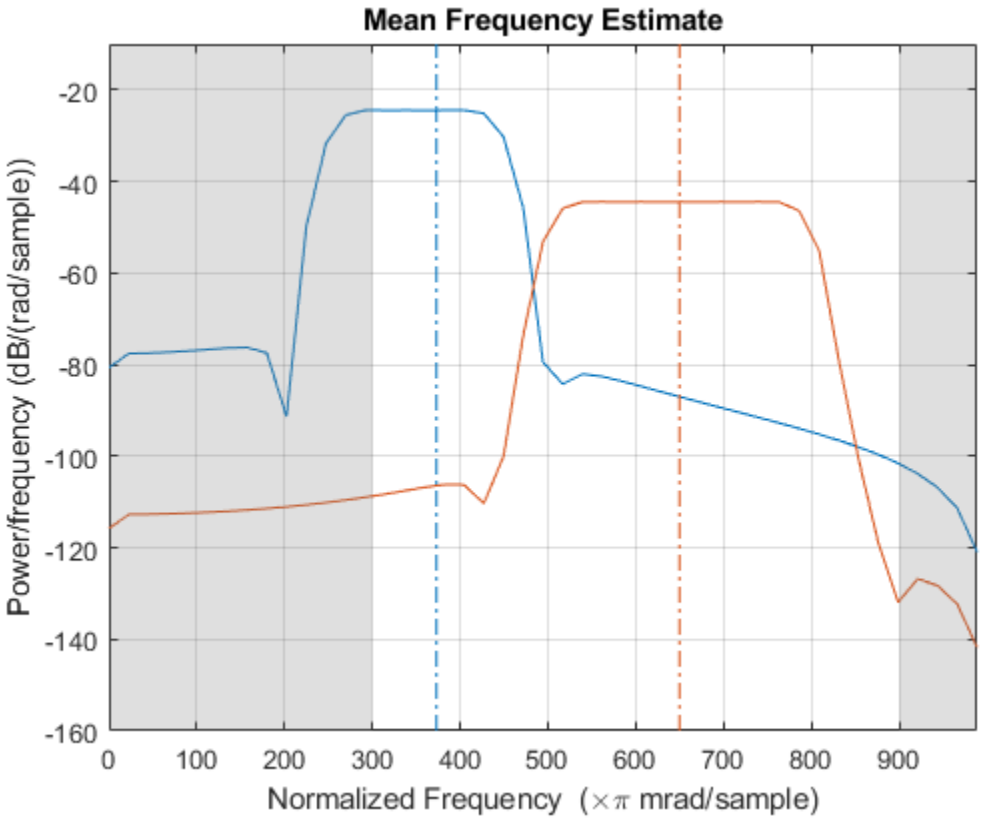
```
Mean = 0.373*pi, power = 75.6% of total
```

Add a second channel with normalized cutoff frequencies 0.5π rad/sample and 0.8π 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 0.3π rad/sample and 0.9π 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 π.

```
mnf = meanfreq(d,[],[0.3 0.9]*pi)/pi
```

```
mnf = 1×2

    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.

The power spectral density must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[pxx,f] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` specifies the periodogram PSD estimate of a noisy two-channel sinusoid sampled at 2π Hz and the frequencies at which it is computed.

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.

The power spectrum must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[sxx,w] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2),'power')` specifies the periodogram power spectrum estimate of a two-channel sinusoid embedded in white Gaussian noise and the normalized frequencies at which it is computed.

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.

References

[1] Phinyomark, Angkoon, Sirinee Thongpanja, Huosheng Hu, Pornchai Phukpattaranont, and Chusak Limsakul. "The Usefulness of Mean and Median Frequencies in Electromyography Analysis." In *Computational Intelligence in Electromyography Analysis – A Perspective on Current Applications and Future Challenges*, edited by Ganesh R. Naik. InTech, 2012. https://doi.org/10.5772/50639.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Thread-Based Environment

Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

Usage notes and limitations:

- The syntax with no output arguments is not supported.

For more information, see “Run MATLAB Functions in Thread-Based Environment”.

See Also

`findpeaks` | `medfreq` | `periodogram` | `plomb` | `pwelch`

Introduced in R2015a

medfilt1

1-D median filtering

Syntax

```
y = medfilt1(x)
y = medfilt1(x,n)

y = medfilt1(x,n,blksz,dim)
y = medfilt1(x,n,[],dim)

y = medfilt1( ___ ,nanflag,padding)
```

Description

`y = medfilt1(x)` applies a third-order one-dimensional median filter to the input vector, `x`. The function considers the signal to be 0 beyond the endpoints. The output, `y`, has the same length as `x`.

`y = medfilt1(x,n)` applies an `n`th-order one-dimensional median filter to `x`.

`y = medfilt1(x,n,blksz,dim)` or `y = medfilt1(x,n,[],dim)` specifies the dimension, `dim`, along which the filter operates. `blksz` is required for backward compatibility and is ignored.

`y = medfilt1( ___ ,nanflag,padding)` specifies how `NaN` values are treated over each segment, using any input arguments from previous syntaxes. This syntax also specifies `padding`, the type of filtering performed at the signal edges.

`nanflag` and `padding` can appear anywhere after `x` in the function call.

Examples

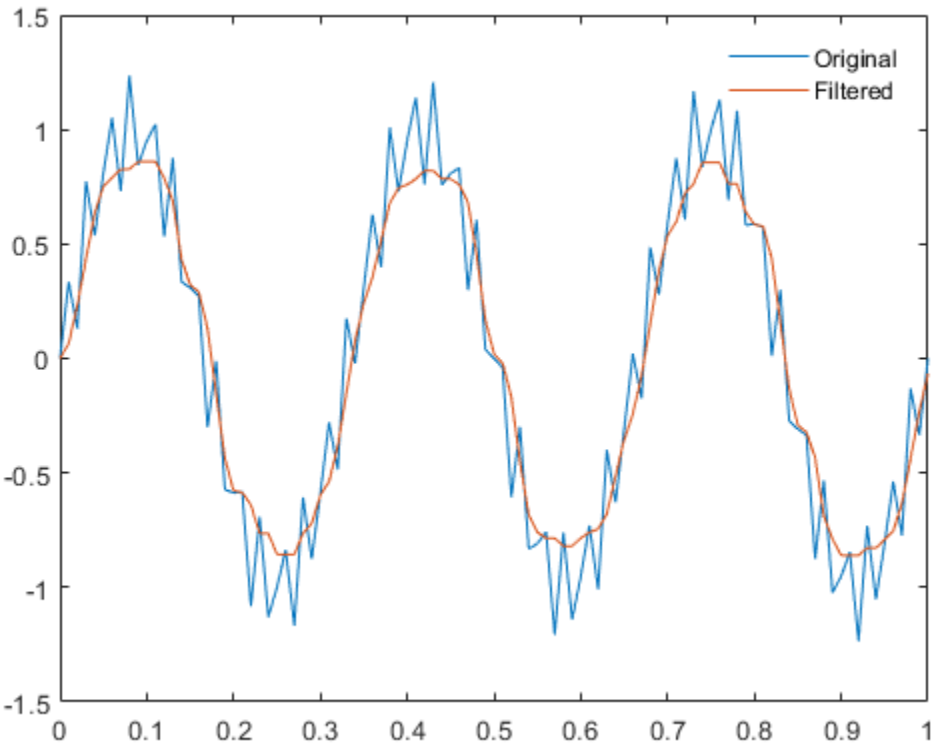
Noise Suppression by Median Filtering

Generate a sinusoidal signal sampled for 1 second at 100 Hz. Add a higher-frequency sinusoid to simulate noise.

```
fs = 100;
t = 0:1/fs:1;
x = sin(2*pi*t*3)+0.25*sin(2*pi*t*40);
```

Use a 10th-order median filter to smooth the signal. Plot the result.

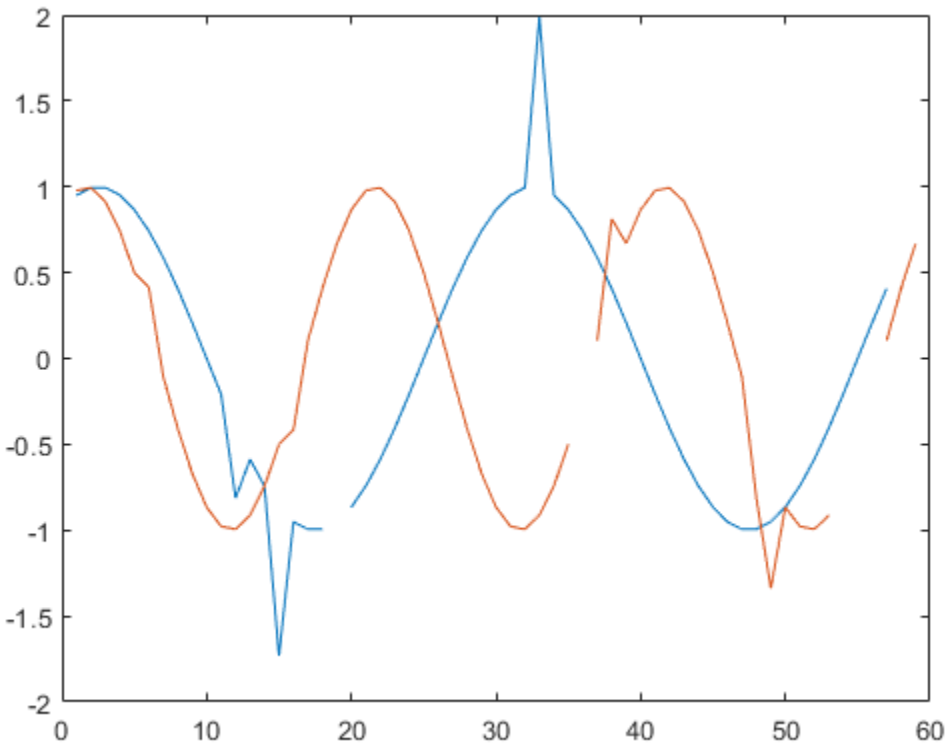
```
y = medfilt1(x,10);

plot(t,x,t,y)
legend('Original','Filtered')
legend('boxoff')
```



Multichannel Signal with Spikes and Missing Samples

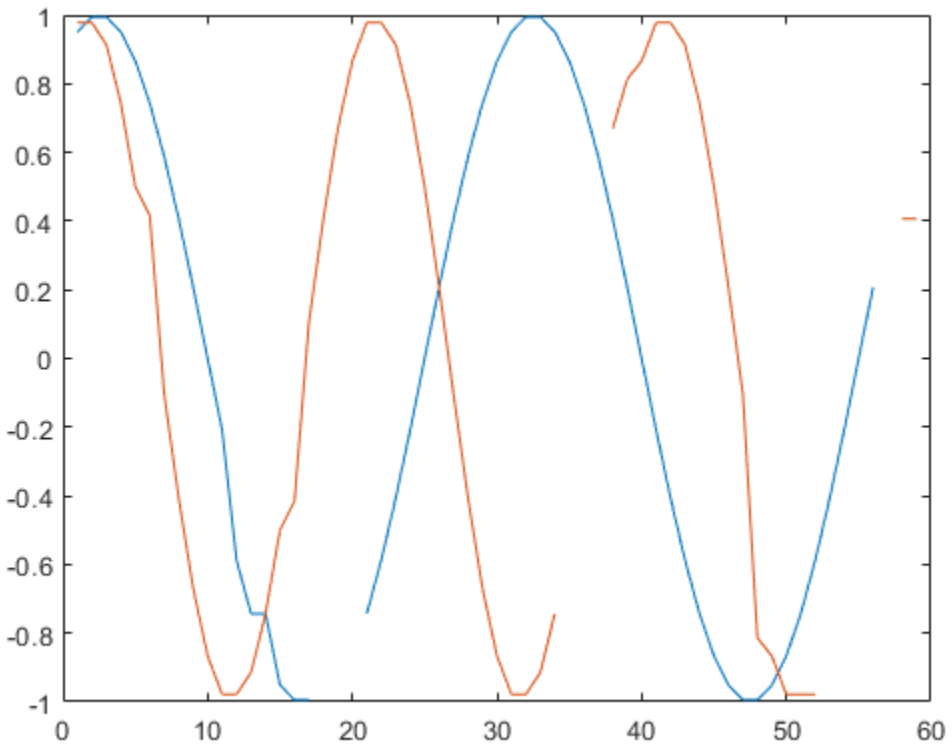
Generate a two-channel signal consisting of sinusoids of different frequencies. Place spikes in random places. Use NaNs to add missing samples at random. Reset the random number generator for reproducible results. Plot the signal.

```
rng('default')

n = 59;
x = sin(pi./[15 10]'*(1:n)+pi/3)';

spk = randi(2*n,9,1);
x(spk) = x(spk)*2;
x(randi(2*n,6,1)) = NaN;

plot(x)
```



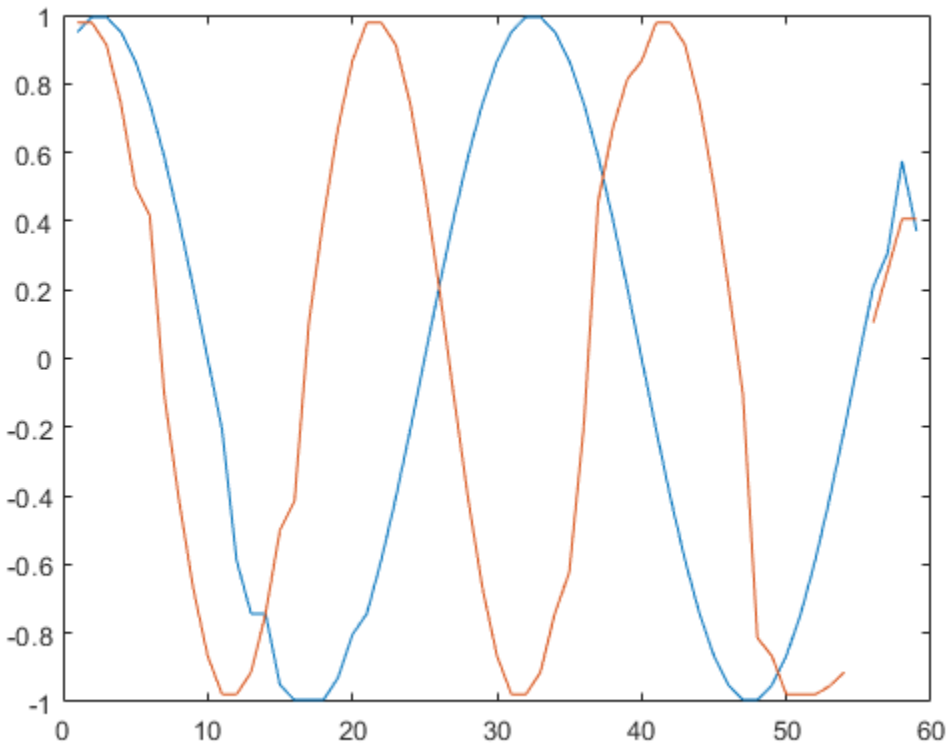
Filter the signal using `medfilt1` with the default settings. Plot the filtered signal. By default, the filter assigns NaN to the median of any segment with missing samples.

```
y = medfilt1(x);
plot(y)
```



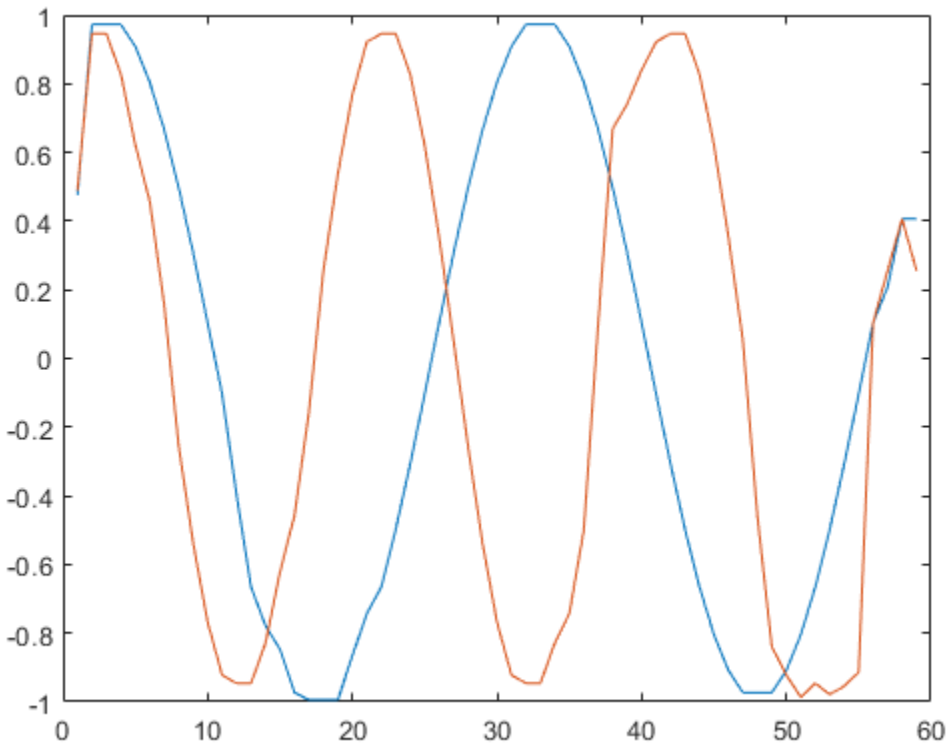
Transpose the original signal. Filter it again, specifying that the function work along the rows. Exclude the missing samples when computing the medians. If you leave the second argument empty, then `medfilt1` uses the default filter order of 3.

```
y = medfilt1(x',[],[],2,'omitnan');
plot(y')
```



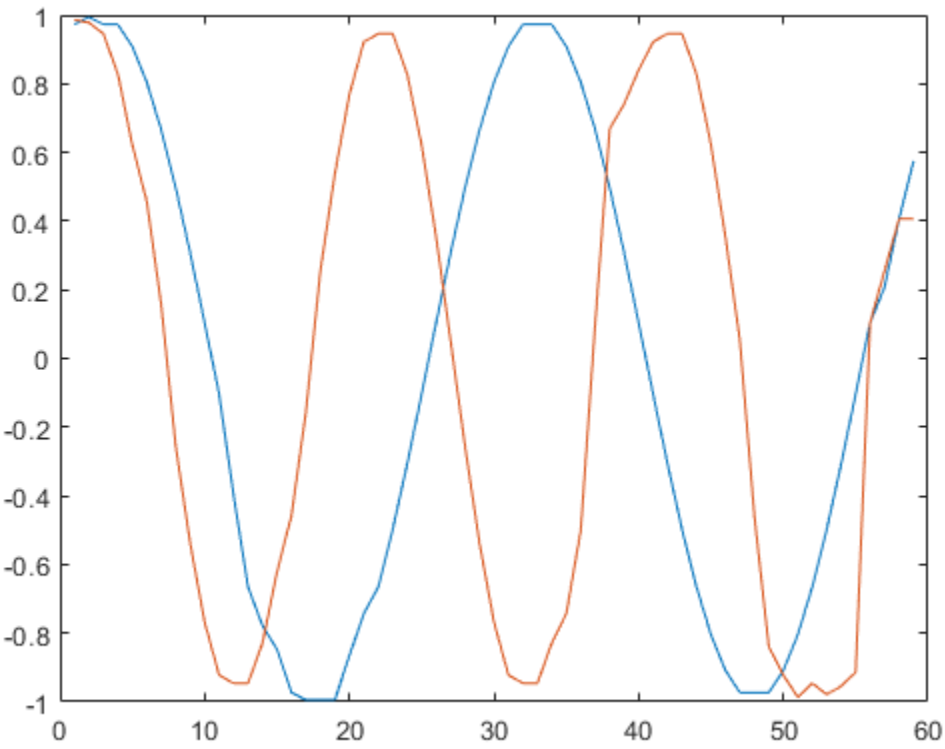
The function cannot assign a value to the segment that contains only NaNs. Increase the segment length to fix this issue. The change also removes the outlier more thoroughly.

```
y = medfilt1(x,4,'omitnan');
plot(y)
```



The default zero padding results in the function's underestimating the signal values at the edges. Lessen this effect by using decreasing windows to compute the medians at the ends.

```
y = medfilt1(x,4,'omitnan','truncate');
plot(y)
```



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: `single` | `double`

n — Filter order

3 (default) | positive integer scalar

Order of the one-dimensional median filter, specified as a positive integer scalar.

- When `n` is odd, `y(k)` is the median of `x(k-(n-1)/2:k+(n-1)/2)`.
- When `n` is even, `y(k)` is the median of `x(k-n/2:k+(n/2)-1)`. In this case, `medfilt1` sorts the numbers and takes the average of the two middle elements of the sorted list.

Example: If `n = 11`, then `y(k)` is the median of `x(k-5:k+5)`.

Example: If `n = 12`, then `y(k)` is the median of `x(k-6:k+5)`.

Data Types: `double`

dim — Dimension to filter along

positive integer scalar

Dimension to filter along, specified as a positive integer scalar. By default, `medfilt1` operates along the first nonsingleton dimension of `x`. In particular, if `x` is a matrix, the function filters its columns so that `y(:,i) = medfilt1(x(:,i),n)`.

Data Types: `double`

`nanflag` — `NaN` condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as `'includenan'` or `'omitnan'`.

- `'includenan'` — Returns the filtered signal so that the median of any segment containing `NaN`s is also `NaN`.
- `'omitnan'` — Returns the filtered signal so that the median of any segment containing `NaN`s is the median of the non-`NaN` values. If all elements of a segment are `NaN`s, the result is `NaN`.

`padding` — Endpoint filtering
`'zeropad'` (default) | `'truncate'`

Endpoint filtering, specified as `'zeropad'` or `'truncate'`.

- `'zeropad'` — Considers the signal to be zero beyond the endpoints.
- `'truncate'` — Computes medians of smaller segments as it reaches the signal edges.

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`

Tips

If you have a license for Image Processing Toolbox™ software, you can use the `medfilt2` function to perform two-dimensional median filtering.

References

[1] Pratt, William K. *Digital Image Processing*. 4th Ed. Hoboken, NJ: John Wiley & Sons, 2007.

See Also
`filter` | `hampel` | `median` | `movmedian` | `sgolayfilt`

Introduced before R2006a

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. This syntax can include any combination of input arguments from previous syntaxes, as long as the second input argument is either `fs` or `f`. If the second input is passed as empty, normalized frequency will be assumed. 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

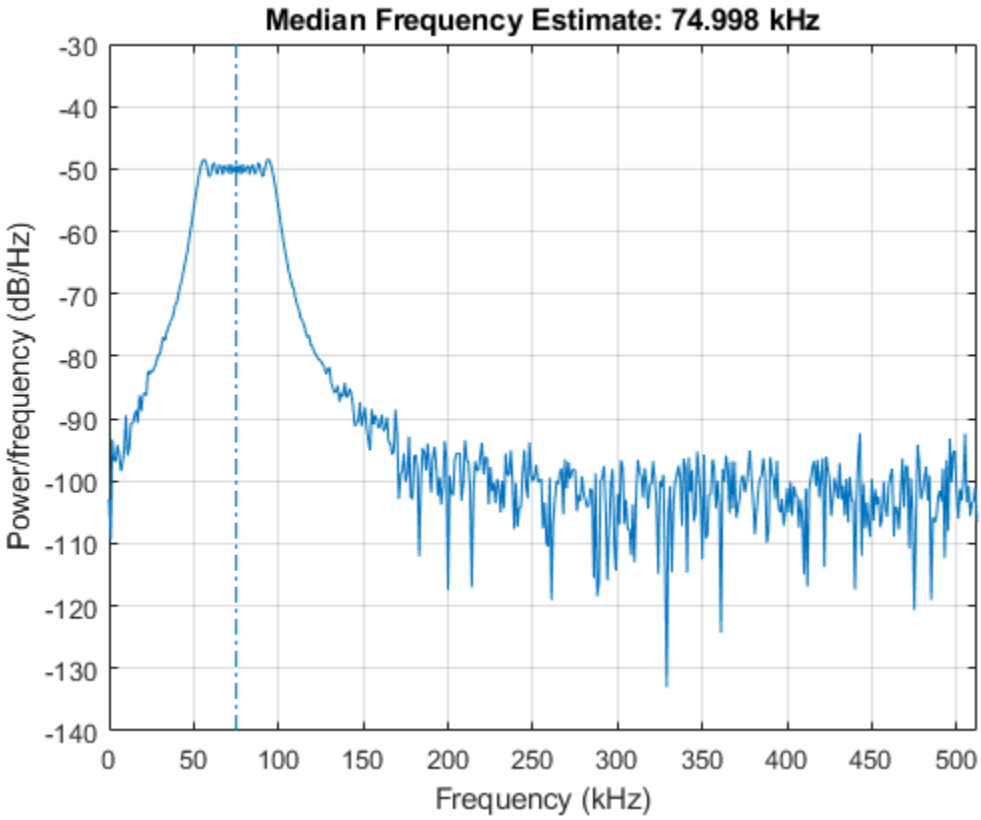
Median Frequency of Chirps

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);
```

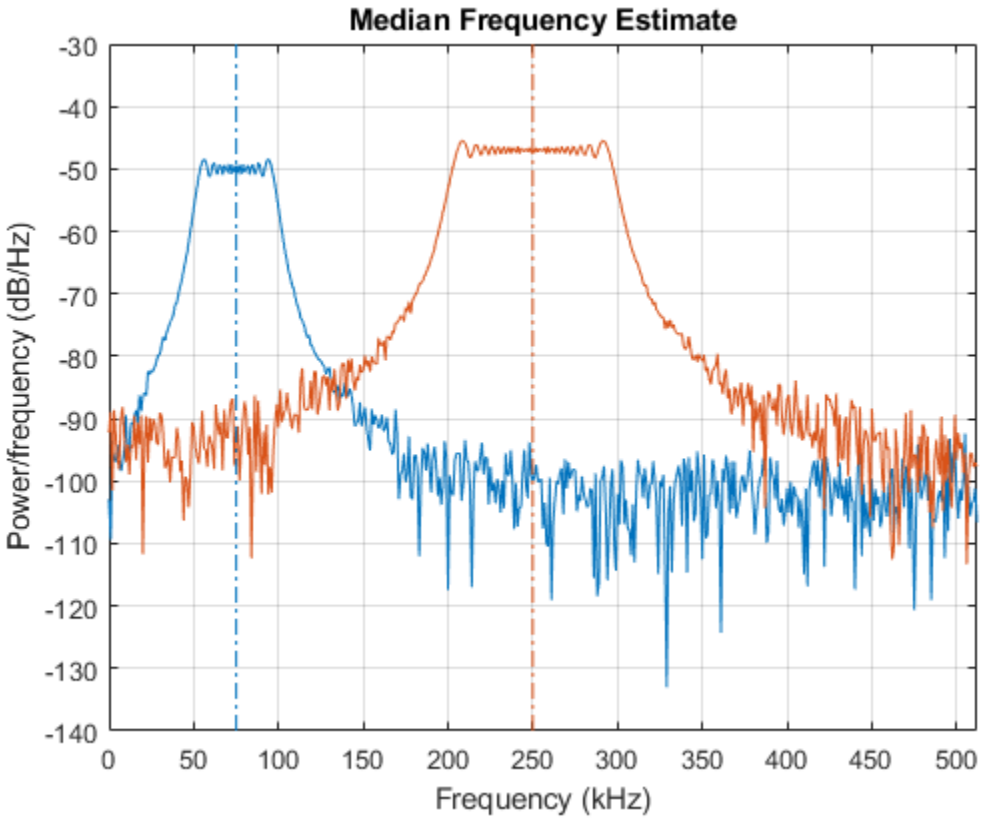
Concatenate the chirps to produce a two-channel signal. Estimate the median frequency of each channel.

```
y = medfreq([x x2],Fs)
```

```
y = 1×2
10^5 ×

    0.7500    2.4999
```

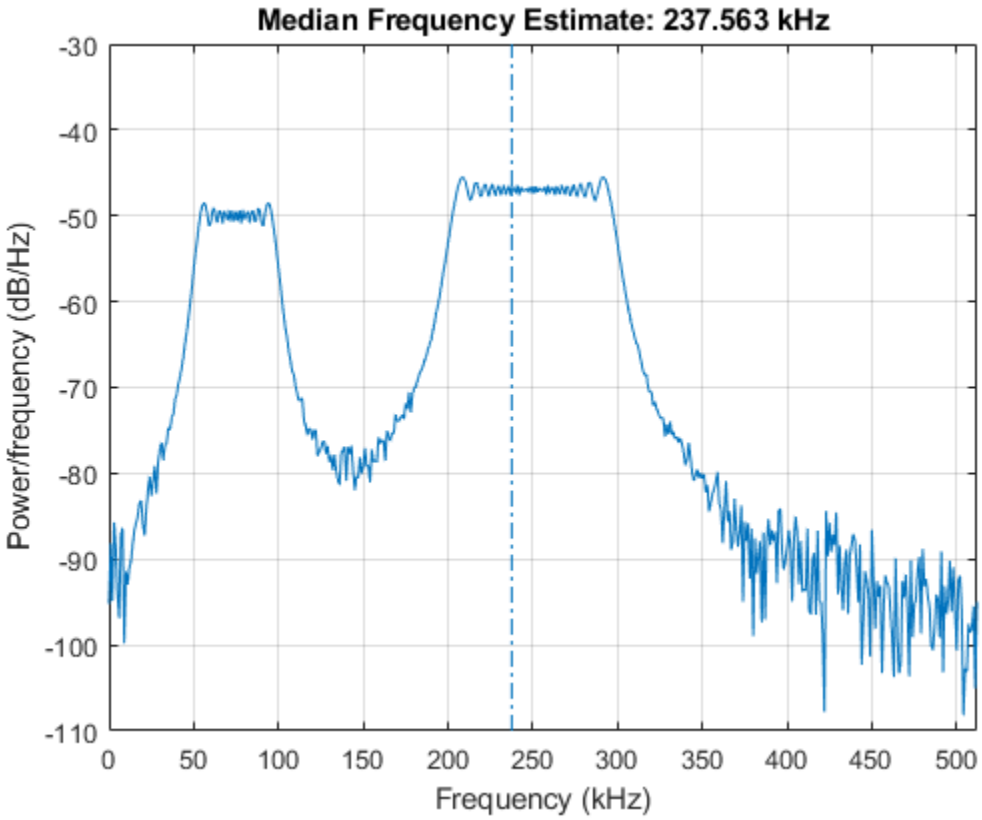
Plot the PSDs of the two channels and annotate their median frequencies.

```
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
```

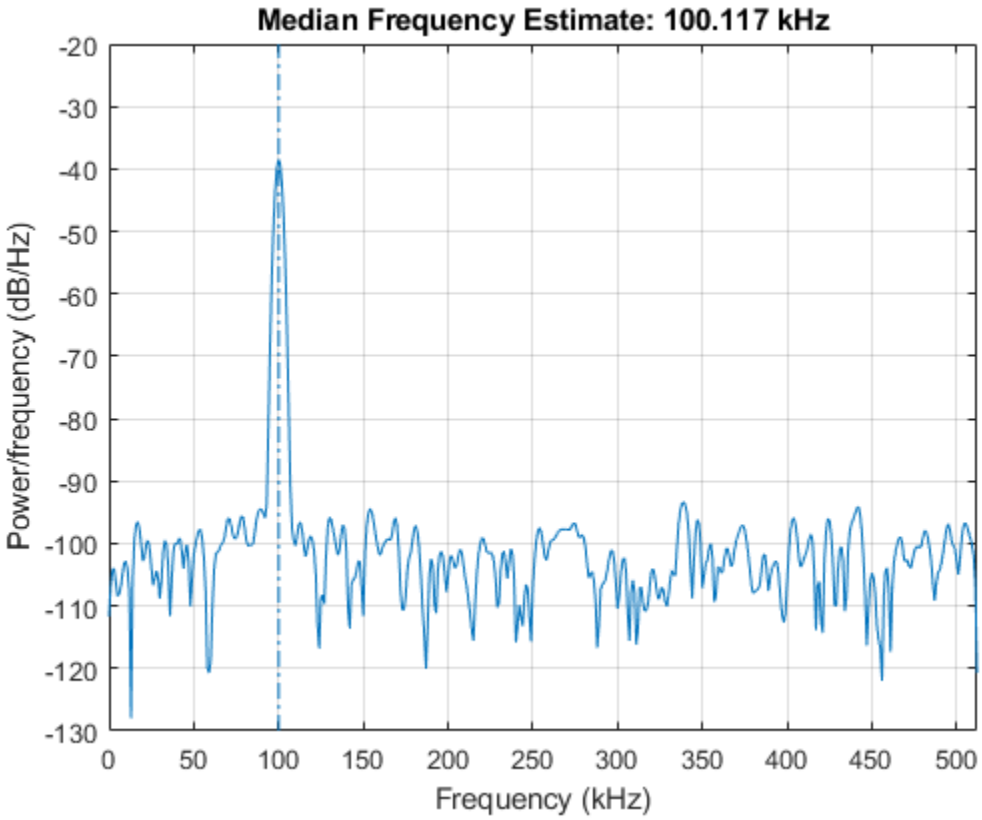
Median Frequency of Sinusoids

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.

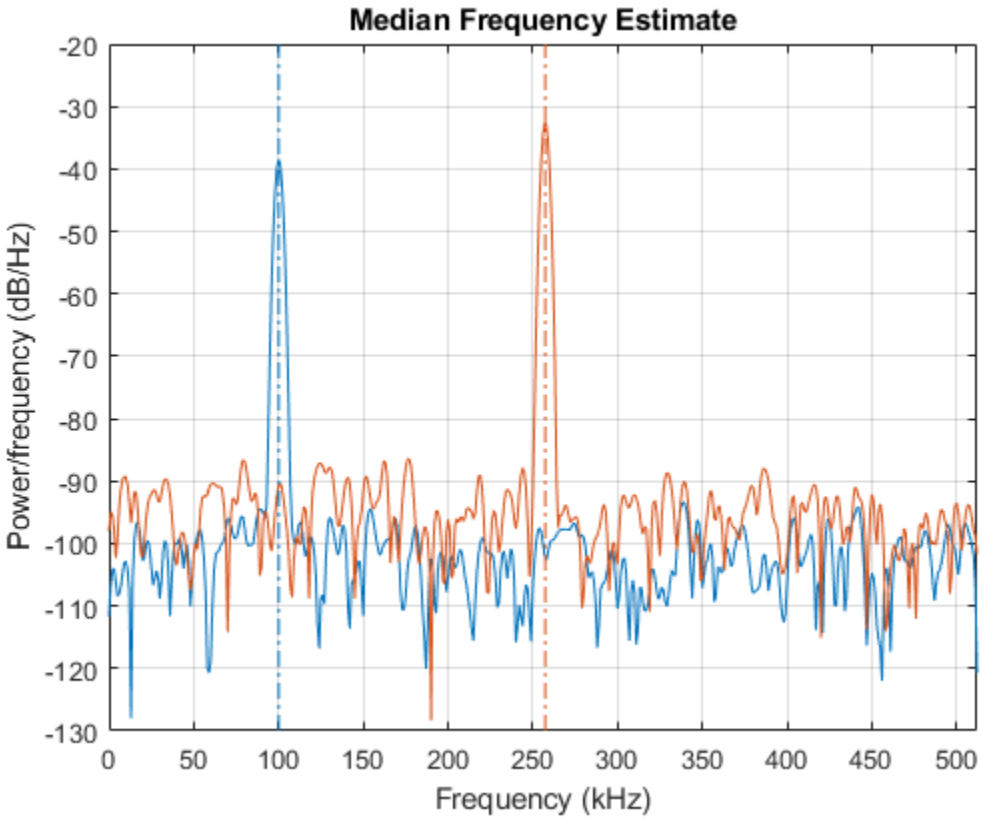
```
[Pyy,f] = periodogram([x x2],kaiser(nSamp,38),[],Fs);
```

```
y = medfreq(Pyy,f)
```

```
y = 1×2
10^5 ×

    1.0012    2.5731
```

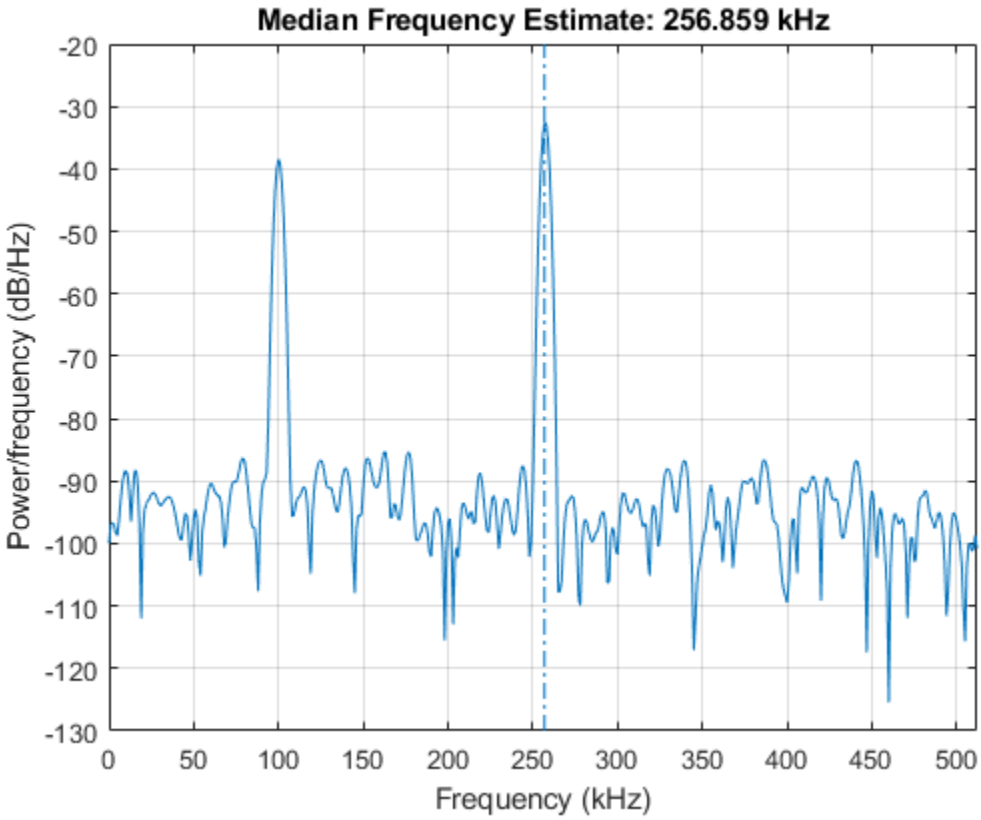
Annotate the median frequencies of the two channels on a plot of the PSDs.

```
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);
```



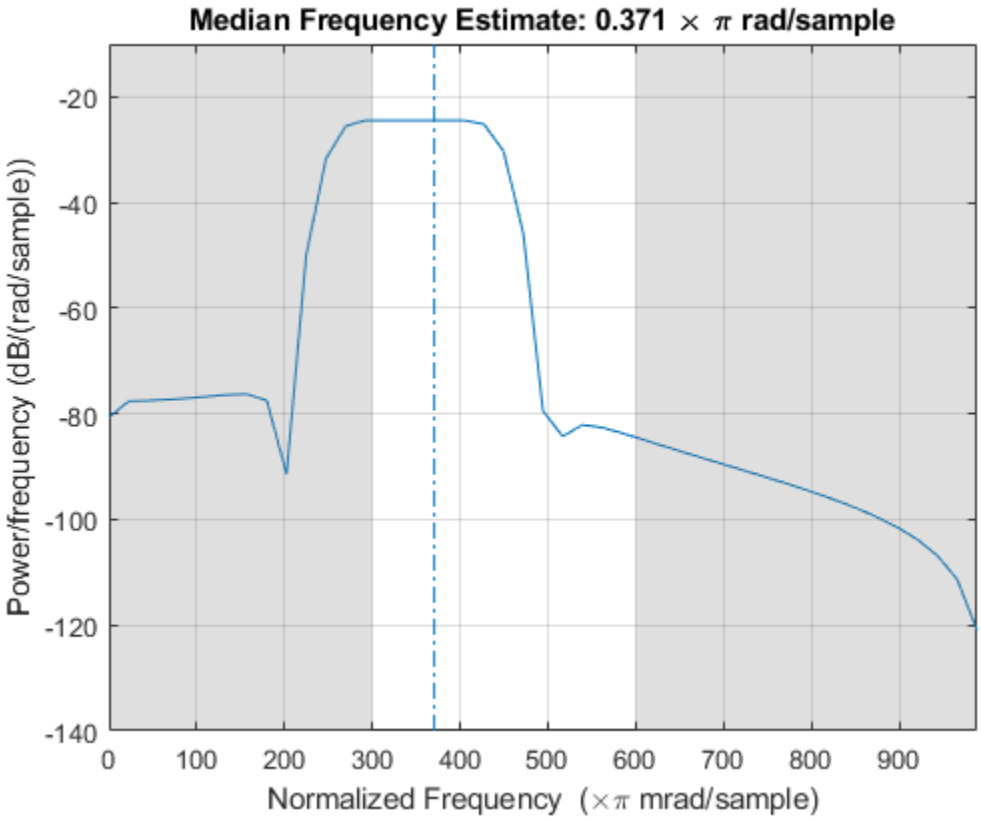
Median Frequency of Bandlimited Signals

Generate a signal whose PSD resembles the frequency response of an 88th-order bandpass FIR filter with normalized cutoff frequencies 0.25π rad/sample and 0.45π rad/sample.

```
d = fir1(88,[0.25 0.45]);
```

Compute the median frequency of the signal between 0.3π rad/sample and 0.6π 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 2π 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)
```

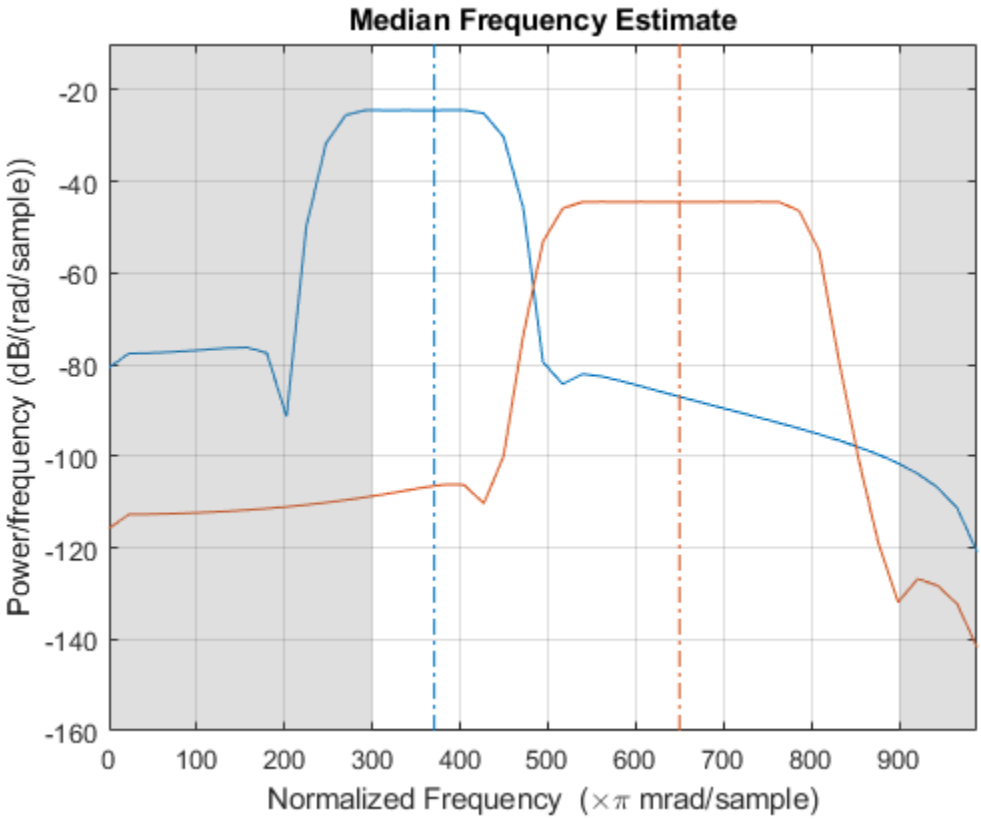
```
Mean = 0.371*pi, power = 77.4% of total
```

Add a second channel with normalized cutoff frequencies 0.5π rad/sample and 0.8π 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 0.3π rad/sample and 0.9π 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 π.

```
mdf = medfreq(d,[],[0.3 0.9]*pi)/pi
```

```
mdf = 1×2

    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.

The power spectral density must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[pxx,f] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` specifies the periodogram PSD estimate of a noisy two-channel sinusoid sampled at 2π Hz and the frequencies at which it is computed.

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.

The power spectrum must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[sxx,w] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2),'power')` specifies the periodogram power spectrum estimate of a two-channel sinusoid embedded in white Gaussian noise and the normalized frequencies at which it is computed.

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 `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.

References

[1] Phinyomark, Angkoon, Sirinee Thongpanja, Huosheng Hu, Pornchai Phukpattaranont, and Chusak Limsakul. "The Usefulness of Mean and Median Frequencies in Electromyography Analysis." In *Computational Intelligence in Electromyography Analysis – A Perspective on Current Applications and Future Challenges*, edited by Ganesh R. Naik. InTech, 2012. https://doi.org/10.5772/50639.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Thread-Based Environment

Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

Usage notes and limitations:

- The syntax with no output arguments is not supported.

For more information, see “Run MATLAB Functions in Thread-Based Environment”.

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 the time instants where each transition of the input signal `x` crosses the 50% reference level. To determine the transitions, the `midcross` function estimates the state levels of `x` by a histogram method and identifies all intervals which cross the upper-state boundary of the low state and the lower-state boundary of the high state.

`c = midcross(x,Fs)` specifies the sample rate `Fs`.

`c = midcross(x,t)` specifies the sample instants `t`.

`[c,midlev] = midcross( ___ )` returns the waveform value corresponding to the mid-reference level. You can specify an input combination from any of the previous syntaxes.

`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 arguments.

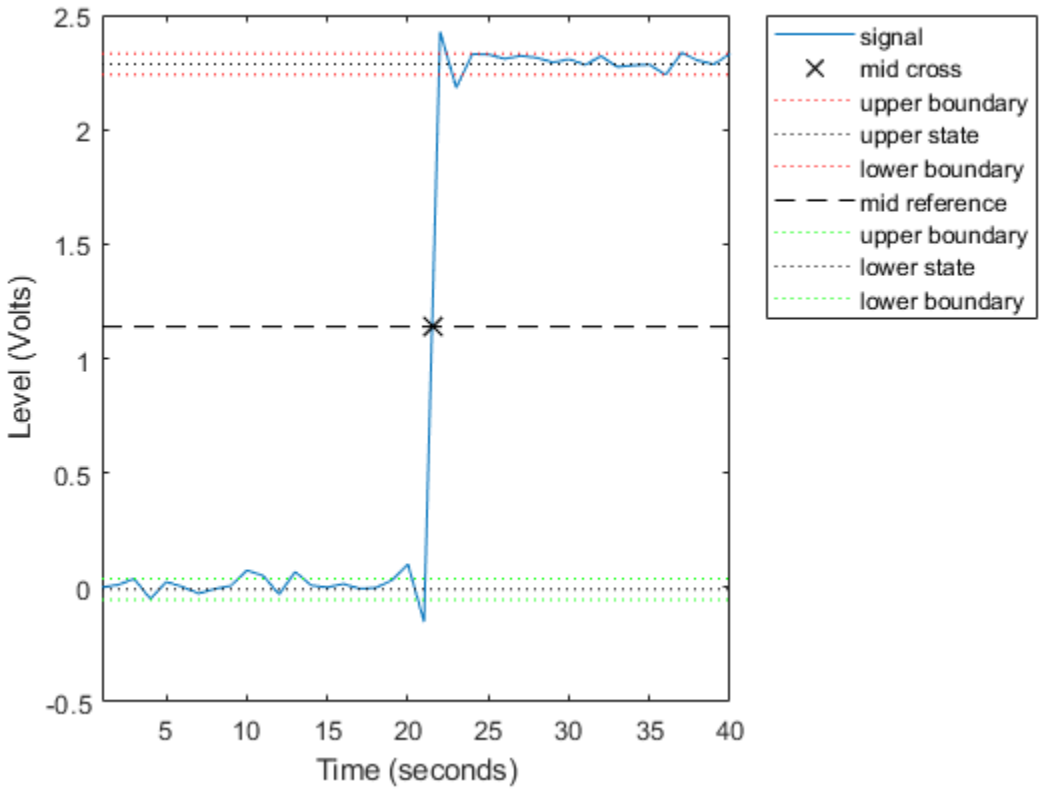
`midcross( ___ )` plots the signal and marks the location of the mid-crossings (mid-reference level instants) and the associated reference levels. The function also plots the state levels with upper and lower state boundaries.

Examples

Mid-Reference Level Instant of Bilevel Waveform

Assuming a sampling interval of 1, compute the mid-reference level instant of a bilevel waveform. Plot the result.

```
load('transitionex.mat','x')

midcross(x)
```




```
ans = 21.5000
```

The instant at which the waveform crosses the 50% reference level is 21.5. This is not a sampling instant present in the input vector. `midcross` uses interpolation to identify the mid-reference level crossing.

Mid-Reference Level Instant with Sample Rate

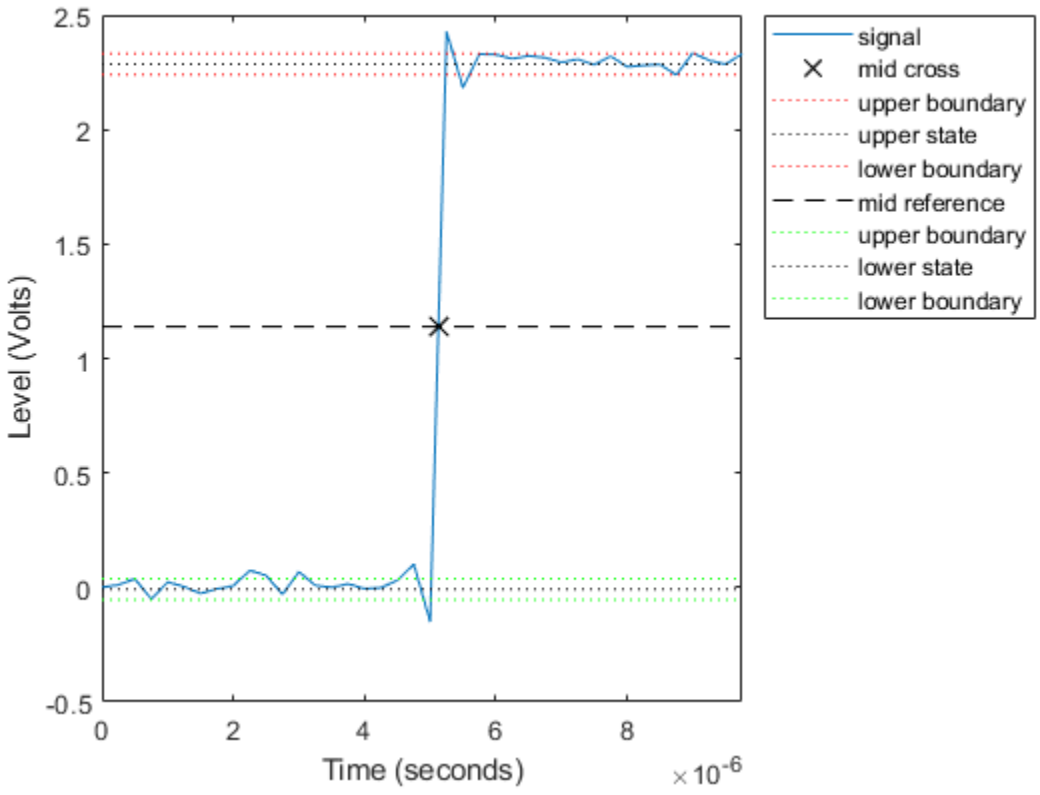
Compute the mid-reference level instant for a sampled bilevel waveform. Use the time information to determine the sample rate, which is 4 MHz.

```
load('transitionex.mat','x','t')
Fs = 1/(t(2)-t(1))
```

```
Fs = 4000000
```

Use the sample rate to express the mid-reference level instant in seconds. Plot the waveform and annotate the result.

```
midcross(x,Fs)
```




```
ans = 5.1250e-06
```

Mid-Reference Level Instant Using Sample Instants

Compute the mid-reference level instant using a vector of sample times equal in length to the bilevel waveform. The sample rate is 4 MHz.

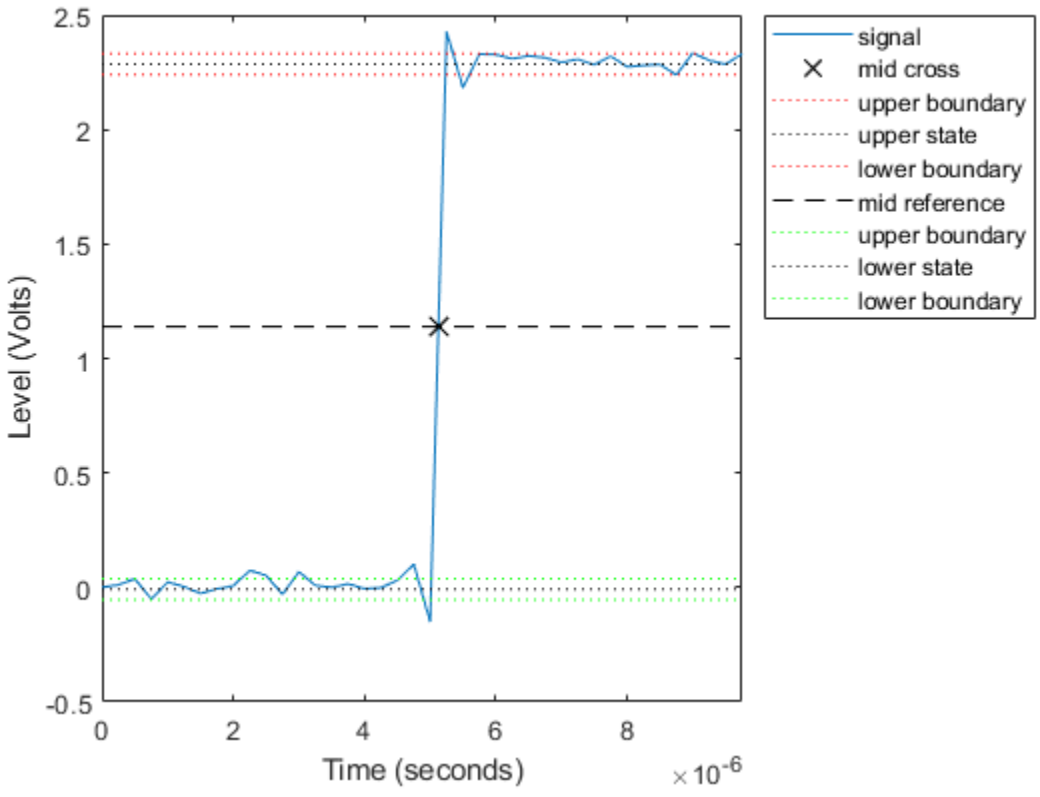
```
load('transitionex.mat','x','t')
```

```
C = midcross(x,t)
```

```
C = 5.1250e-06
```

Annotate the result on a plot of the waveform.

```
midcross(x,t);
```



Mid-Reference Level Value of Bilevel Waveform

Compute the level corresponding to the mid-reference level instant.

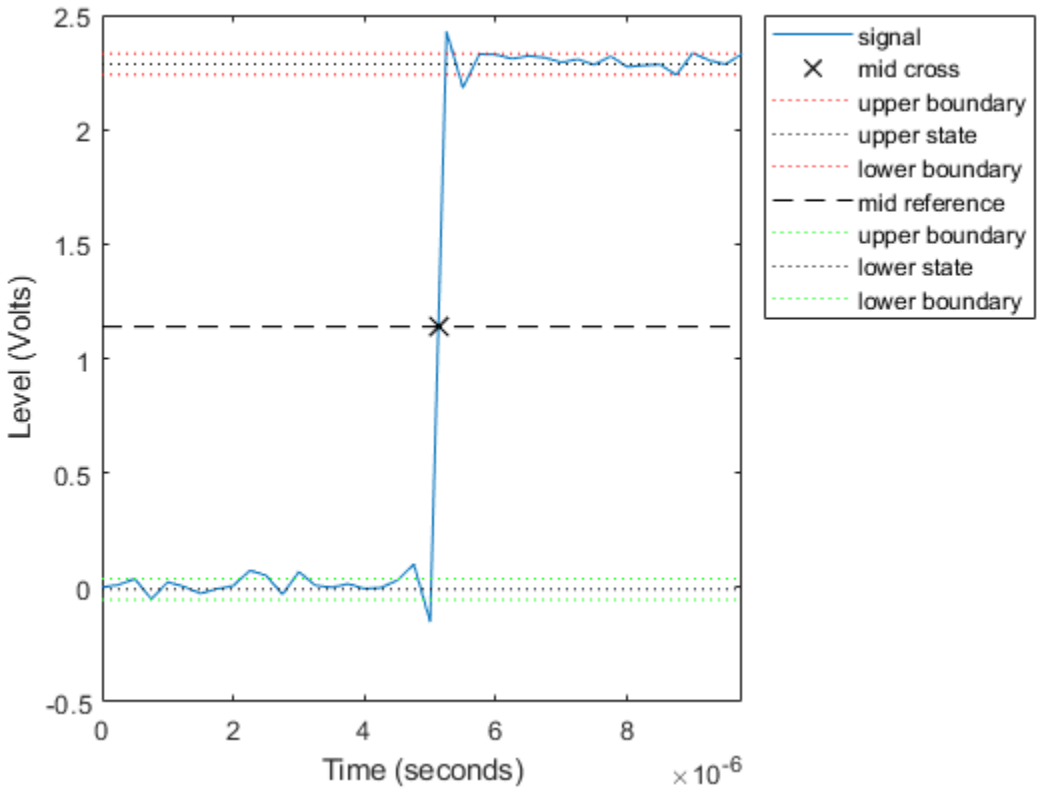
```
load('transitionex.mat','x','t')
```

```
[~,midlev] = midcross(x,t)
```

```
midlev = 1.1388
```

Annotate the result on a plot of the waveform.

```
midcross(x,t);
```



Sixty Percent Reference Level Instant and Waveform Value

Obtain the 60% reference level instant and value for a bilevel waveform sampled at 4 MHz.

```
load("transitionex.mat","x","t")
```

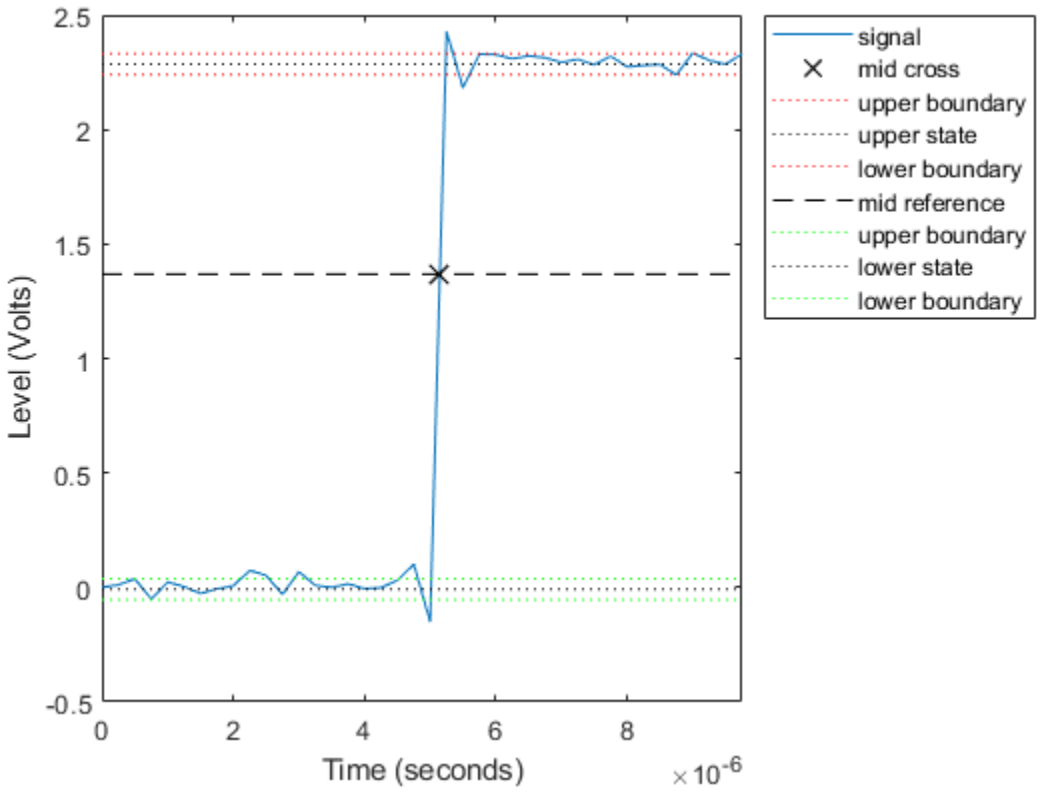
```
[mc,Lev60] = midcross(x,t,MidPercentReferenceLevel=60)
```

```
mc = 5.1473e-06
```

```
Lev60 = 1.3682
```

Annotate the result on a plot of the waveform.

```
midcross(x,t,MidPercentReferenceLevel=60);
```



Input Arguments

x — Bilevel waveform

real-valued vector

Bilevel waveform, specified as a real-valued vector. The first sample instant corresponds to t=0.

Fs — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar in hertz.

t — Sample instants

vector

Sample instants, specified as a vector. The length of `t` must equal the length of the input bilevel waveform `x`. The sample instants correspond to the indices of the input vector.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `c = midcross(x,t,MidPercentReferenceLevel=30)`

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `c = midcross(x,t,'MidPercentReferenceLevel',30)`

`MidPercentReferenceLevel` — Mid-reference level
`50` (default) | scalar

Mid-reference level as a percentage of the waveform amplitude, specified as a scalar. For more information, see “Mid-Reference Level” on page 1-1404.

`StateLevels` — Low- and high-state levels
1-by-2 real-valued vector

Low- and high-state levels, specified as a 1-by-2 real-valued vector. The first element is the low-state level and the second element is the high-state level. If you do not specify `'StateLevels'`, the function estimates the state levels from the input waveform using the histogram method.

`Tolerance` — Tolerance levels
2 (default) | scalar

Tolerance levels (lower- and upper-state boundaries), specified as a scalar and expressed as a percentage. 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. For more information, see “State-Level Tolerances” on page 1-1405.

Output Arguments

`c` — Time instants
vector

Time instants of the mid-reference level crossings, returned as a vector.

Note Because `midcross` uses interpolation to determine the crossing instant, `c` may contain values that do not correspond to sampling instants.

`midlev` — Mid-reference level
scalar

Mid-reference level, returned as a scalar.

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

$$t_{50\%} = t_{50\%} + (\frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}})(y_{50\%_+} - y_{50\%_-})$$

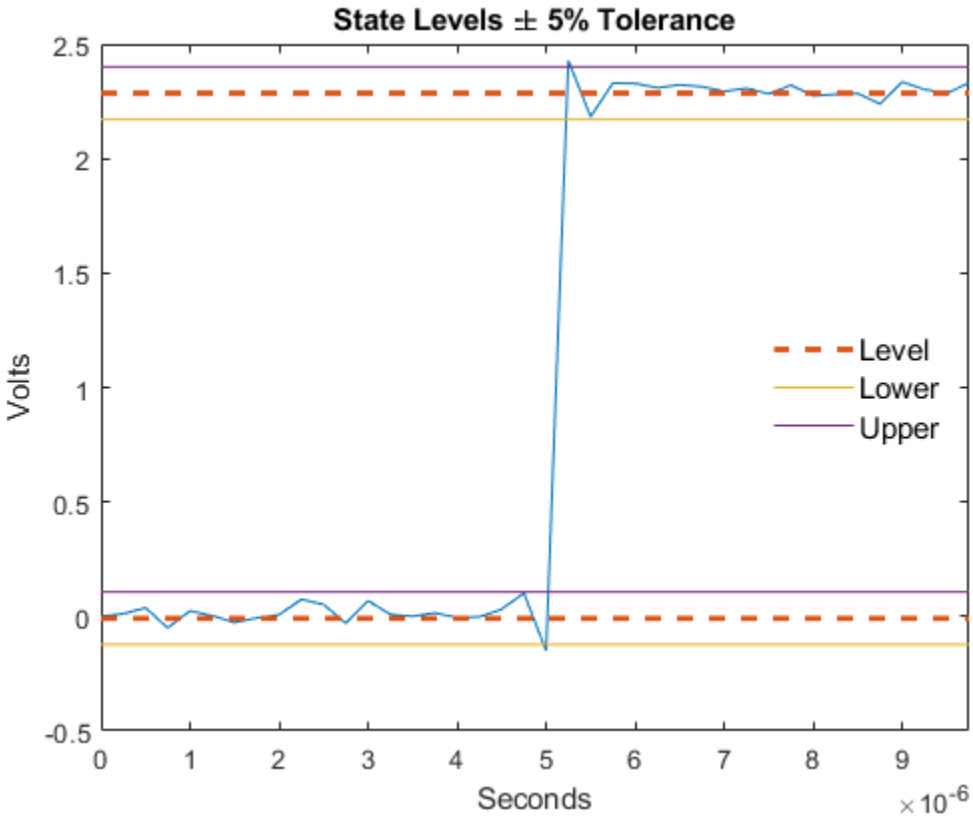
State-Level Tolerances

You can specify lower- and upper-state boundaries for each state level. Define the boundaries as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, specify the scalar as a small number such as 2/100 or 3/100. In general, the $\alpha\%$ 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.

This figure shows lower and upper 5% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The thick dashed lines indicate the estimated state levels.



References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003. p. 20.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`falltime` | `pulsewidth` | `risetime` | `settlingtime` | `statelevels`

Introduced in R2012a

modalfit

Modal parameters from frequency-response functions

Syntax

```
fn = modalfit(frf,f,fs,mnum)

fn = modalfit(frf,f,fs,mnum,Name,Value)

[fn,dr,ms] = modalfit(___)
[fn,dr,ms,ofrf] = modalfit(___)

[___] = modalfit(sys,f,mnum,Name,Value)
```

Description

`fn = modalfit(frf,f,fs,mnum)` estimates the natural frequencies of `mnum` modes of a system with measured frequency-response functions `frf` defined at frequencies `f` and for a sample rate `fs`. Use `modalfrf` to generate a matrix of frequency-response functions from measured data. `frf` is assumed to be in dynamic flexibility (receptance) format.

`fn = modalfit(frf,f,fs,mnum,Name,Value)` specifies additional options using name-value arguments.

`[fn,dr,ms] = modalfit(___)` also returns the damping ratios and mode-shape vectors corresponding to each natural frequency in `fn`, using any combination of inputs from previous syntaxes.

`[fn,dr,ms,ofrf] = modalfit(___)` also returns a reconstructed frequency-response function array based on the estimated modal parameters.

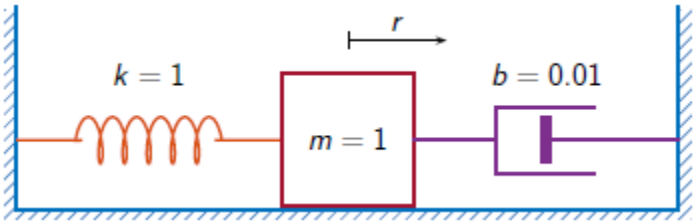
`[___] = modalfit(sys,f,mnum,Name,Value)` estimates the modal parameters of the identified model `sys`. Use estimation commands like `ssest` or `tfest` to create `sys` starting from a measured frequency-response function or from time-domain input and output signals. This syntax allows use of the `'DriveIndex'`, `'FreqRange'`, and `'PhysFreq'` name-value arguments. It typically requires less data than syntaxes that use nonparametric methods. You must have a System Identification Toolbox™ license to use this syntax.

Examples

Frequency-Response Function of SISO System

Estimate the frequency-response function for a simple single-input/single-output system and compare it to the definition.

A one-dimensional discrete-time oscillating system consists of a unit mass, m, attached to a wall by a spring with elastic constant $k = 1$. A sensor samples the displacement of the mass at $F_s = 1$ Hz. A damper impedes the motion of the mass by exerting on it a force proportional to speed, with damping constant $b = 0.01$.



Generate 3000 time samples. Define the sampling interval $\Delta t = 1/F_s$.

```
Fs = 1;
dt = 1/Fs;
N = 3000;
t = dt*(0:N-1);
b = 0.01;
```

The system can be described by the state-space model

$$x(k+1) = Ax(k) + Bu(k),$$
$$y(k) = Cx(k) + Du(k),$$

where $x = [r\ v]^T$ is the state vector, r and v are respectively the displacement and velocity of the mass, u is the driving force, and $y = r$ is the measured output. The state-space matrices are

$$A = \exp(A_c \Delta t), \quad B = A_c^{-1}(A - I)B_c, \quad C = [1\ 0], \quad D = 0,$$

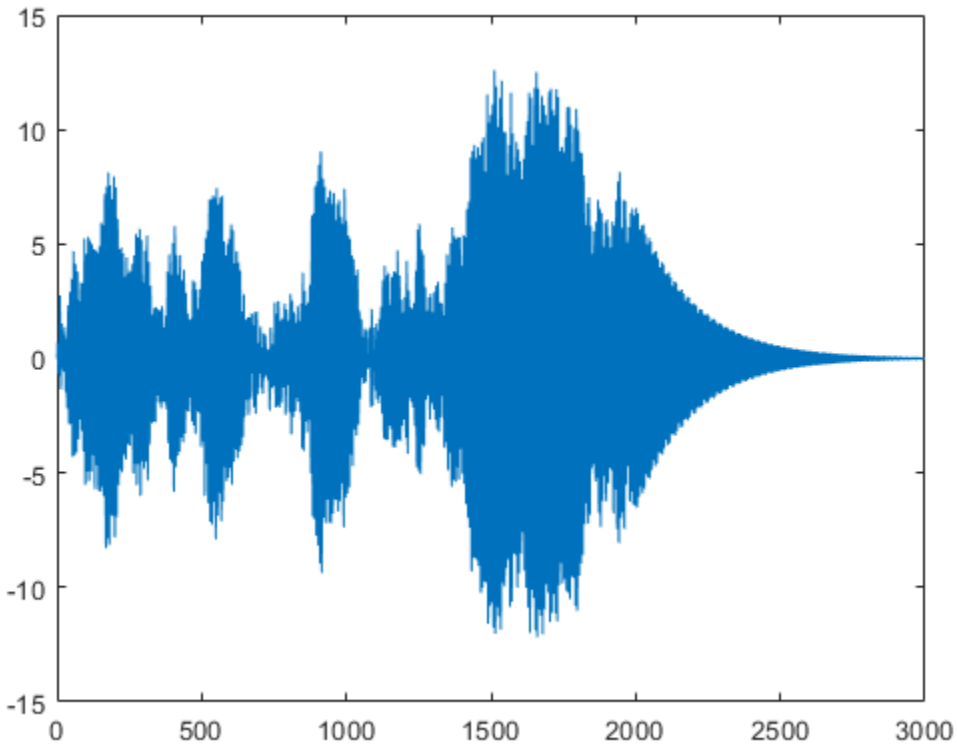
I is the 2×2 identity, and the continuous-time state-space matrices are

$$A_c = \begin{bmatrix} 0 & 1 \\ -1 & -b \end{bmatrix}, \quad B_c = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

```
Ac = [0 1;-1 -b];
A = expm(Ac*dt);

Bc = [0;1];
B = Ac\(A-eye(2))*Bc;

C = [1 0];
D = 0;
```

The mass is driven by random input for the first 2000 seconds and then left to return to rest. Use the state-space model to compute the time evolution of the system starting from an all-zero initial state. Plot the displacement of the mass as a function of time.

```
rng default
u = randn(1,N)/2;
u(2001:end) = 0;

y = 0;
x = [0;0];
for k = 1:N
    y(k) = C*x + D*u(k);
    x = A*x + B*u(k);
end

plot(t,y)
```

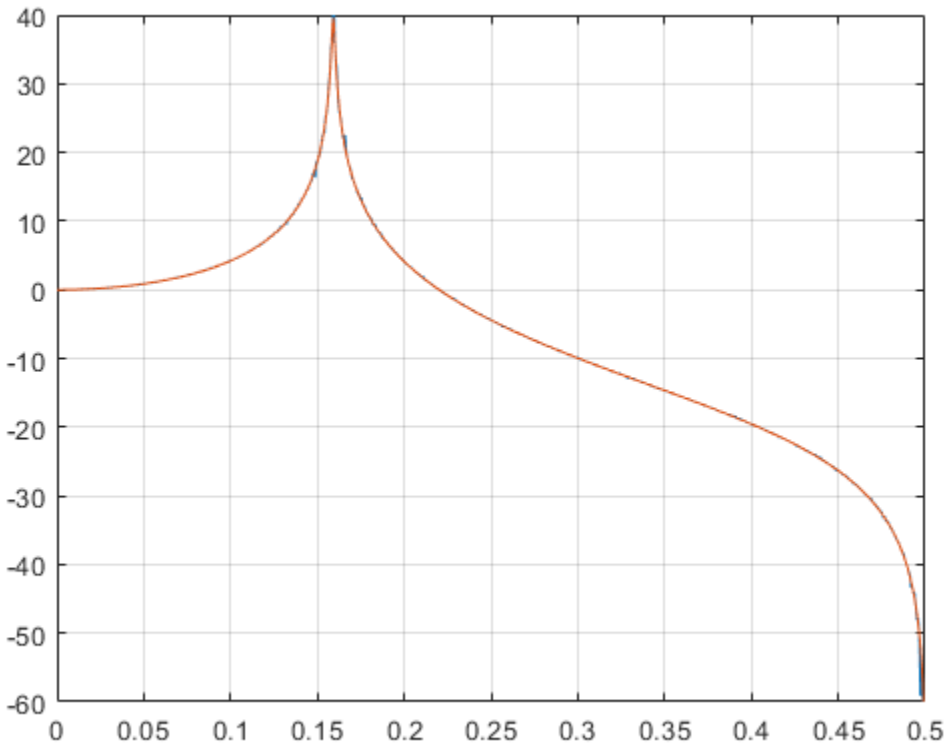


Estimate the modal frequency-response function of the system. Use a Hann window half as long as the measured signals. Specify that the output is the displacement of the mass.

```
wind = hann(N/2);

[frf,f] = modalfrf(u',y',Fs,wind,'Sensor','dis');
```

The frequency-response function of a discrete-time system can be expressed as the Z-transform of the time-domain transfer function of the system, evaluated at the unit circle. Compare the `modalfrf` estimate with the definition.

```
[b,a] = ss2tf(A,B,C,D);

nfs = 2048;
fz = 0:1/nfs:1/2-1/nfs;
z = exp(2j*pi*fz);
ztf = polyval(b,z)./polyval(a,z);

plot(f,20*log10(abs(frf)))
hold on
plot(fz*Fs,20*log10(abs(ztf)))
hold off
grid
ylim([-60 40])
```



Estimate the natural frequency and the damping ratio for the vibration mode.

```
[fn,dr] = modalfit(frf,f,Fs,1,'FitMethod','PP')
```

```
fn = 0.1593
```

```
dr = 0.0043
```

Compare the natural frequency to $1/2\pi$, which is the theoretical value for the undamped system.

```
theo = 1/(2*pi)
```

```
theo = 0.1592
```

Modal Parameters Using Least-Squares Rational Function Method

Compute the modal parameters of a Space Station module starting from its frequency-response function (FRF) array.

Load a structure containing the three-input/three-output FRF array. The system is sampled at 320 Hz.

```
load modaldata SpaceStationFRF
```

```
frf = SpaceStationFRF.FRF;
```

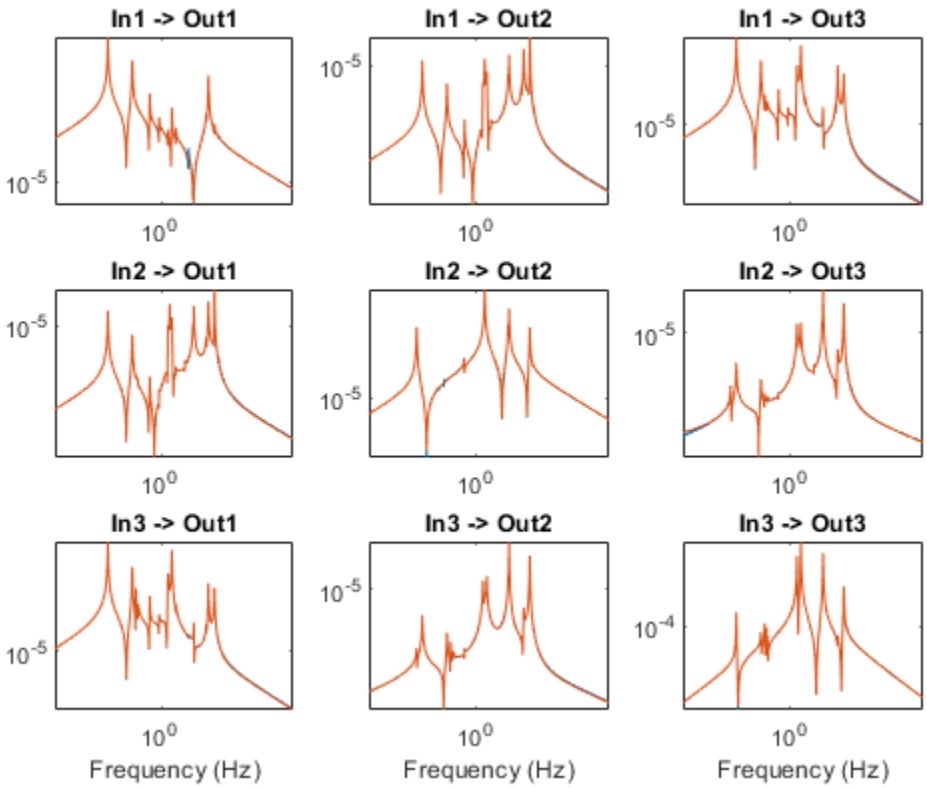
```
f = SpaceStationFRF.f;
fs = SpaceStationFRF.Fs;
```

Extract the modal parameters of the lowest 24 modes using the least-squares rational function method.

```
[fn,dr,ms,ofrf] = modalfit(frf,f,fs,24,'FitMethod','lsrf');
```

Compare the reconstructed FRF array to the measured one.

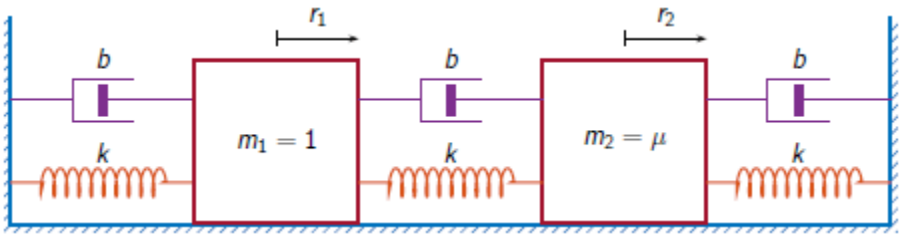
```
for ij = 1:3
 for ji = 1:3
    subplot(3,3,3*(ij-1)+ji)
    loglog(f,abs(frf(:,ji,ij)))
    hold on
    loglog(f,abs(ofrf(:,ji,ij)))
    hold off
    axis tight
    title(sprintf('In%d -> Out%d',ij,ji))
    if ij==3
        xlabel('Frequency (Hz)')
    end
 end
end
```



Modal Parameters of Two-Body Oscillator

Estimate the frequency-response function and modal parameters of a simple multi-input/multi-output system.

An ideal one-dimensional oscillating system consists of two masses, m_1 and m_2, confined between two walls. The units are such that $m_1 = 1$ and $m_2 = \mu$. Each mass is attached to the nearest wall by a spring with an elastic constant k. An identical spring connects the two masses. Three dampers impede the motion of the masses by exerting on them forces proportional to speed, with damping constant b. Sensors sample r_1 and r_2, the displacements of the masses, at $F_s = 50$ Hz.



Generate 30,000 time samples, equivalent to 600 seconds. Define the sampling interval $\Delta t = 1/F_s$.

```
Fs = 50;
dt = 1/Fs;
N = 30000;
t = dt*(0:N-1);
```

The system can be described by the state-space model

$$x(k+1) = Ax(k) + Bu(k),$$
$$y(k) = Cx(k) + Du(k),$$

where $x = [r_1\ v_1\ r_2\ v_2]^T$ is the state vector, r_i and v_i are respectively the location and the velocity of the ith mass, $u = [u_1\ u_2]^T$ is the vector of input driving forces, and $y = [r_1\ r_2]^T$ is the output vector. The state-space matrices are

$$A = \exp(A_c \Delta t), \quad B = A_c^{-1}(A - I)B_c, \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix},$$

I is the 4×4 identity, and the continuous-time state-space matrices are

$$A_c = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -2k & -2b & k & b \\ 0 & 0 & 0 & 1 \\ k/\mu & b/\mu & -2k/\mu & -2b/\mu \end{bmatrix}, \quad B_c = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1/\mu \end{bmatrix}.$$

Set $k = 400$, $b = 0.1$, and $\mu = 1/10$.

```
k = 400;
b = 0.1;
m = 1/10;

Ac = [0 1 0 0;-2*k -2*b k b;0 0 0 1;k/m b/m -2*k/m -2*b/m];
```

```
A = expm(Ac*dt);
Bc = [0 0;1 0;0 0;0 1/m];
B = Ac\(A-eye(4))*Bc;
C = [1 0 0 0;0 0 1 0];
D = zeros(2);
```

The masses are driven by random input throughout the measurement. Use the state-space model to compute the time evolution of the system starting from an all-zero initial state.

```
rng default
u = randn(2,N);

y = [0;0];
x = [0;0;0;0];
for kk = 1:N
    y(:,kk) = C*x + D*u(:,kk);
    x = A*x + B*u(:,kk);
end
```

Use the input and output data to estimate the transfer function of the system as a function of frequency. Use a 15000-sample Hann window with 9000 samples of overlap between adjoining segments. Specify that the measured outputs are displacements.

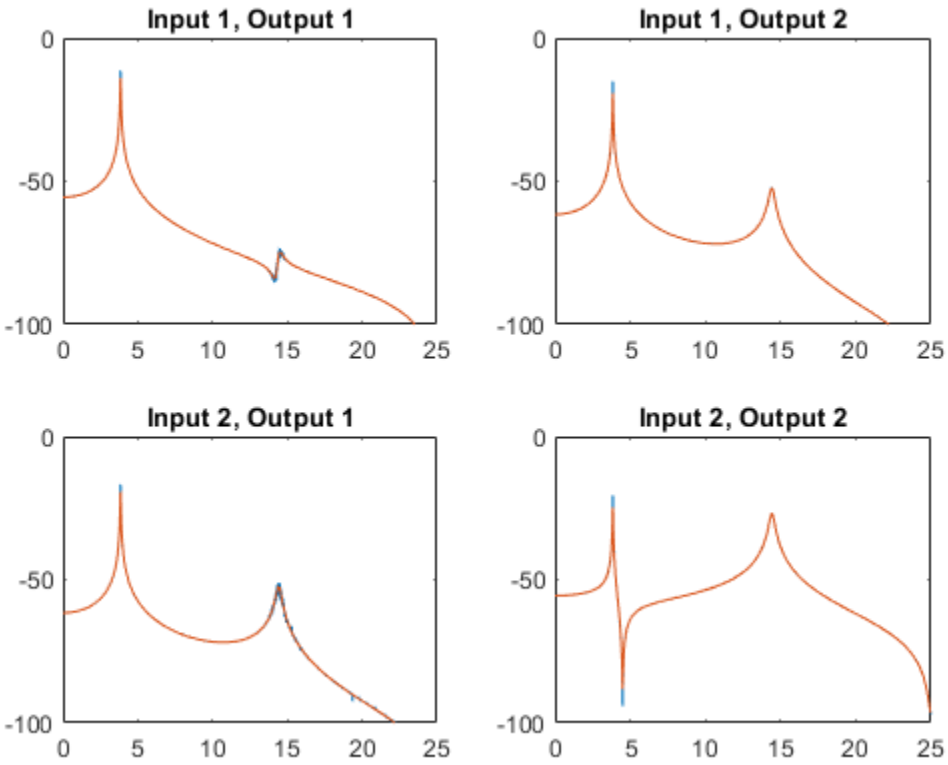
```
wind = hann(15000);
nove = 9000;
[FRF,f] = modalfrf(u',y',Fs,wind,nove,'Sensor','dis');
```

Compute the theoretical transfer function as the Z-transform of the time-domain transfer function, evaluated at the unit circle.

```
nfs = 2048;
fz = 0:1/nfs:1/2-1/nfs;
z = exp(2j*pi*fz);

[b1,a1] = ss2tf(A,B,C,D,1);
[b2,a2] = ss2tf(A,B,C,D,2);

frf(1,:,1) = polyval(b1(1,:),z)./polyval(a1,z);
frf(1,:,2) = polyval(b1(2,:),z)./polyval(a1,z);
frf(2,:,1) = polyval(b2(1,:),z)./polyval(a2,z);
frf(2,:,2) = polyval(b2(2,:),z)./polyval(a2,z);
```

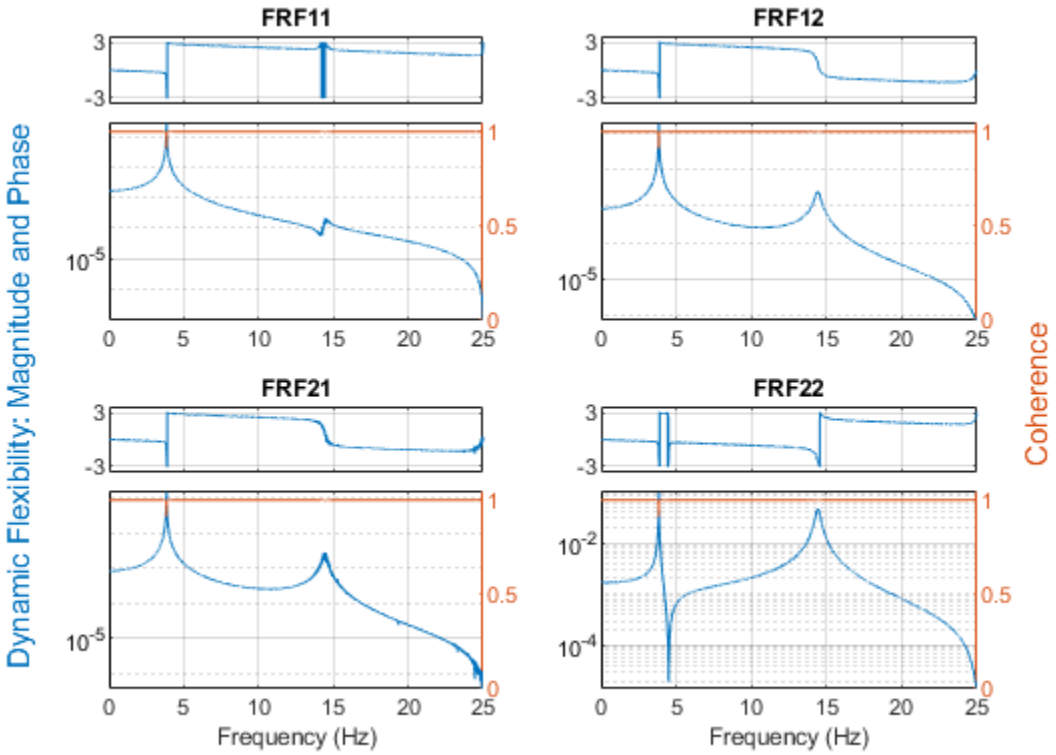
Plot the estimates and overlay the theoretical predictions.

```
for jk = 1:2
    for kj = 1:2
        subplot(2,2,2*(jk-1)+kj)
        plot(f,20*log10(abs(FRF(:,jk,kj))))
        hold on
        plot(fz*Fs,20*log10(abs(frf(jk,:,kj))))
        hold off
        axis([0 Fs/2 -100 0])
        title(sprintf('Input %d, Output %d',jk,kj))
    end
end
```



Plot the estimates by using the syntax of `modalfrf` with no output arguments.

```
figure
modalfrf(u',y',Fs,wind,nove,'Sensor','dis')
```



Estimate the natural frequencies, damping ratios, and mode shapes of the system. Use the peak-picking method for the calculation.

```
[fn,dr,ms] = modalfit(FRF,f,Fs,2,'FitMethod','pp');
fn
```

```
fn =
fn(:,:,1) =

    3.8466    3.8466
    3.8495    3.8495


fn(:,:,2) =

    3.8492    3.8490
    3.8552   14.4684
```

Compare the natural frequencies to the theoretical predictions for the undamped system.

```
undamped = sqrt(eig([2*k -k;-k/m 2*k/m]))/2/pi
```

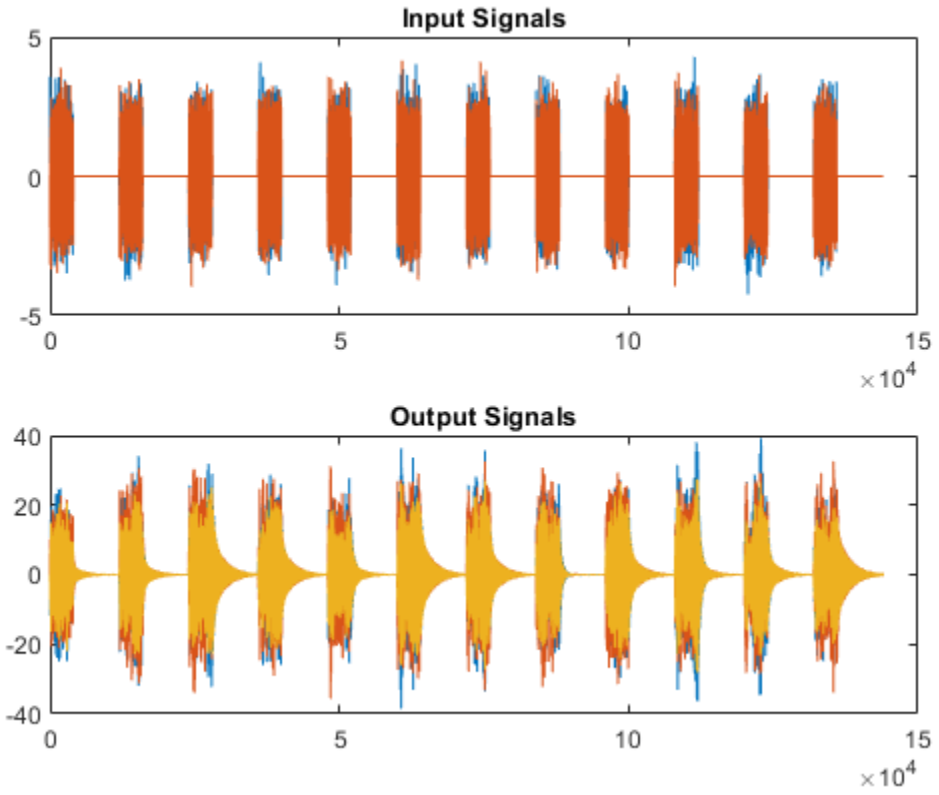
```
undamped = 2×1

    3.8470
   14.4259
```

Modal Parameters of MIMO System

Compute the natural frequencies, the damping ratios, and the mode shapes for a two-input/three-output system excited by several bursts of random noise. Each burst lasts for 1 second, and there are 2 seconds between the end of each burst and the start of the next. The data are sampled at 4 kHz.

Load the data file. Plot the input signals and the output signals.

```
load modaldata

subplot(2,1,1)
plot(Xburst)
title('Input Signals')
subplot(2,1,2)
plot(Yburst)
title('Output Signals')
```

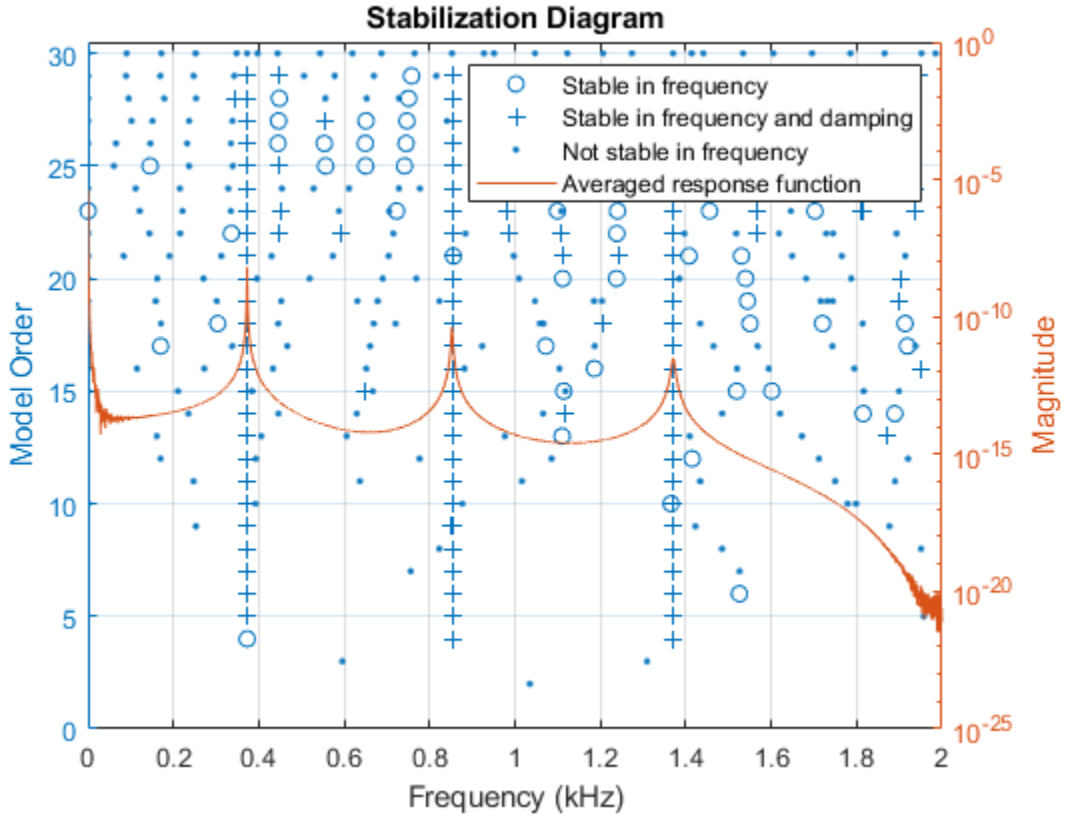


Compute the frequency-response functions. Specify a rectangular window with length equal to the burst period and no overlap between adjoining segments.

```
burstLen = 12000;
[frf,f] = modalfrf(Xburst,Yburst,fs,burstLen);
```

Visualize a stabilization diagram and return the stable natural frequencies. Specify a maximum model order of 30 modes.

```
figure
modalsd(frf,f,fs,'MaxModes',30);
```



Zoom in on the plot. The averaged response function has maxima at 373 Hz, 852 Hz, and 1371 Hz, which correspond to the physical frequencies of the system. Save the maxima to a variable.

```
phfr = [373 852 1371];
```

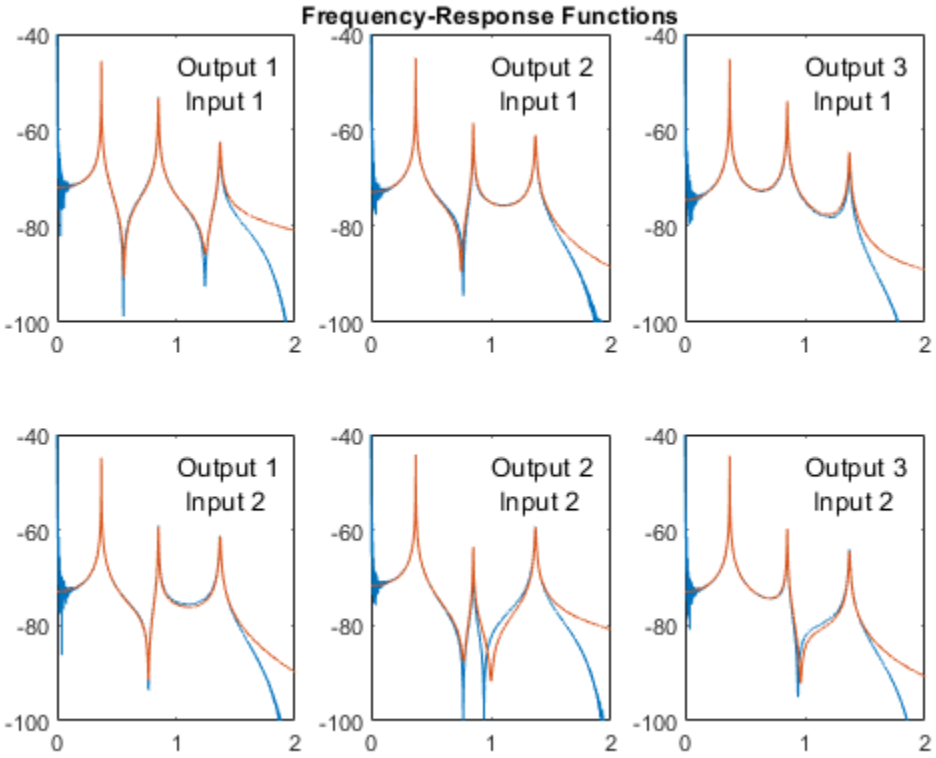
Compute the modal parameters using the least-squares complex exponential (LSCE) algorithm. Specify a model order of 6 modes and specify physical frequencies for the 3 modes determined from the stabilization diagram. The function generates one set of natural frequencies and damping ratios for each input reference.

```
[fn,dr,ms,ofrf] = modalfit(frf,f,fs,6,'PhysFreq',phfr);
```

Plot the reconstructed frequency-response functions and compare them to the original ones.

```
for k = 1:2
    for m = 1:3
        subplot(2,3,m+3*(k-1))
        plot(f/1000,10*log10(abs(frf(:,m,k))))
        hold on
        plot(f/1000,10*log10(abs(ofrf(:,m,k))))
        hold off
        text(1,-50,[['Output ';' Input '] num2str([m k]')])
        ylim([-100 -40])
    end
end
```

```
subplot(2,3,2)
title('Frequency-Response Functions')
```



Input Arguments

`frf` — Frequency-response functions

vector | matrix | 3-D array

Frequency-response functions, specified as a vector, matrix, or 3-D array. `frf` has size *p*-by-*m*-by-*n*, where *p* is the number of frequency bins, *m* is the number of response signals, and *n* is the number of excitation signals used to estimate the transfer function. `frf` is assumed to be in dynamic flexibility (receptance) format.

Use `modalfrf` to generate a matrix of frequency-response functions from measured data.

Example: Undamped Harmonic Oscillator

The motion of a simple undamped harmonic oscillator of unit mass and elastic constant sampled at a rate $f_s = 1/\Delta t$ is described by the transfer function

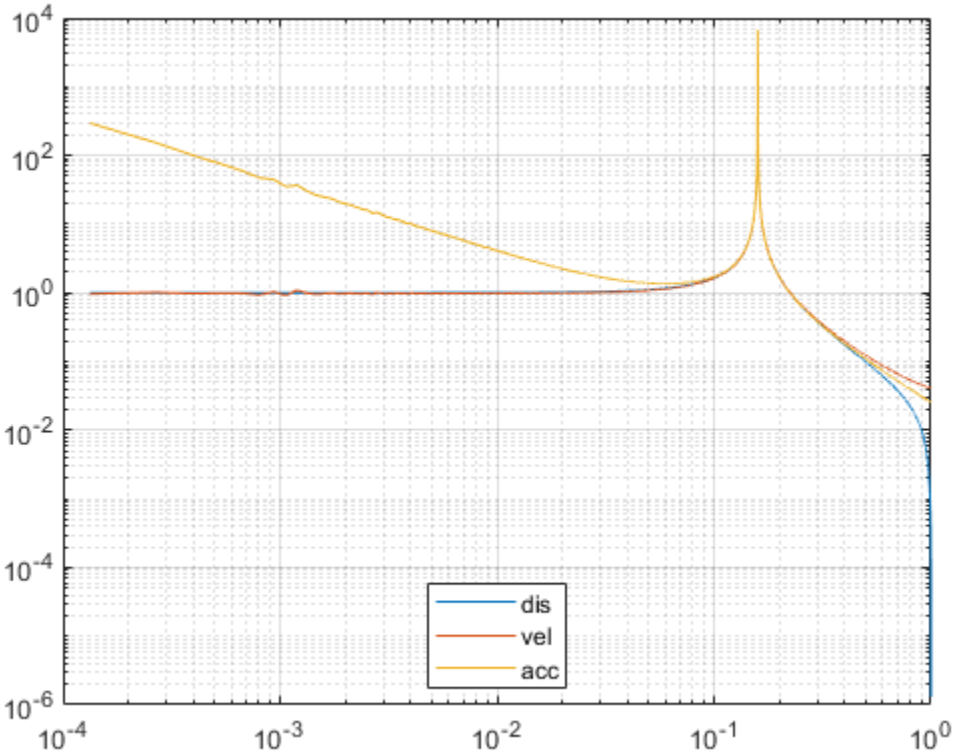
$$H(z) = \frac{N_{\text{Sensor}}(z)}{1 - 2z^{-1}\cos\Delta t + z^{-2}},$$

where the numerator depends on the magnitude being measured:

- Displacement: $N_{\text{dis}}(z) = \left(z^{-1} + z^{-2}\right)(1 - \cos\Delta t)$
- Velocity: $N_{\text{vel}}(z) = \left(z^{-1} - z^{-2}\right)\sin\Delta t$
- Acceleration: $N_{\text{acc}}(z) = \left(1 - z^{-1}\right) - \left(z^{-1} - z^{-2}\right)\cos\Delta t$

Compute the frequency-response function for the three possible system response sensor types. Use a sample rate of 2 Hz and 30,000 samples of white noise as input.

```
fs = 2;
dt = 1/fs;
N = 30000;

u = randn(N,1);

ydis = filter((1-cos(dt))*[0 1 1],[1 -2*cos(dt) 1],u);
[frfd,fd] = modalfrf(u,ydis,fs,hann(N/2),Sensor="dis");

yvel = filter(sin(dt)*[0 1 -1],[1 -2*cos(dt) 1],u);
[frfv,fv] = modalfrf(u,yvel,fs,hann(N/2),Sensor="vel");

yacc = filter([1 -(1+cos(dt)) cos(dt)],[1 -2*cos(dt) 1],u);
[frfa,fa] = modalfrf(u,yacc,fs,hann(N/2),Sensor="acc");

loglog(fd,abs(frfd),fv,abs(frfv),fa,abs(frfa))
grid
legend(["dis" "vel" "acc"],Location="best")
```



In all cases, the generated frequency-response function is in a format corresponding to displacement. Velocity and acceleration measurements are first and second time derivatives, respectively, of displacement measurements. The frequency-response functions are equivalent in the range around the natural frequency of the system. Away from the natural frequency, the frequency-response functions differ.

Data Types: `single` | `double`
Complex Number Support: Yes

`f` — Frequencies

vector

Frequencies, specified as a vector. The number of elements of `f` must equal the number of rows of `frf`.

Data Types: `single` | `double`

`fs` — Sample rate of measurement data

positive scalar

Sample rate of measurement data, specified as a positive scalar expressed in hertz.

Data Types: `single` | `double`

`mnum` — Number of modes

positive integer

Number of modes, specified as a positive integer.

Data Types: `single` | `double`

`sys` — Identified system

model with identified parameters

Identified system, specified as a model with identified parameters. Use estimation commands like `ssest`, `n4sid`, or `tfest` to create `sys` starting from a measured frequency-response function or from time-domain input and output signals. See “Modal Analysis of Identified Models” for an example. You must have a System Identification Toolbox license to use this input argument.

Example: `idss([0.5418 0.8373;-0.8373 0.5334],[0.4852;0.8373],[1 0],0,[0;0],[0;0],1)` generates an identified state-space model corresponding to a unit mass attached to a wall by a spring of unit elastic constant and a damper with constant 0.01. The displacement of the mass is sampled at 1 Hz.

Example: `idtf([0 0.4582 0.4566],[1 -1.0752 0.99],1)` generates an identified transfer-function model corresponding to a unit mass attached to a wall by a spring of unit elastic constant and a damper with constant 0.01. The displacement of the mass is sampled at 1 Hz.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'FitMethod','pp','FreqRange',[0 500]` uses the peak-picking method to perform the fit and restricts the frequency range to between 0 and 500 Hz.

`Feedthrough` — Presence of feedthrough in estimated transfer function
`false` (default) | `true`

Presence of feedthrough in estimated transfer function, specified as a logical value. This argument is available only if `'FitMethod'` is specified as `'lsrf'`.

Data Types: `logical`

`FitMethod` — Fitting algorithm
`'lsce'` (default) | `'lsrf'` | `'pp'`

Fitting algorithm, specified as `'lsce'`, `'lsrf'`, or `'pp'`.

- `'lsce'` — "Least-Squares Complex Exponential Method" on page 1-1422. If you specify `'lsce'`, then `fn` is a vector with `mnum` elements, independent of the size of `frf`.
- `'lsrf'` — Least-squares rational function estimation method. If you specify `'lsrf'`, then `fn` is a vector with `mnum` elements, independent of the size of `frf`. The method is described in [3]. See "Continuous-Time Transfer Function Estimation Using Continuous-Time Frequency-Domain Data" (System Identification Toolbox) for more information. This algorithm typically requires less data than nonparametric approaches and is the only one that works for nonuniform `f`.
- `'pp'` — "Peak-Picking Method" on page 1-1424. For an `frf` computed from *n* excitation signals and *m* response signals, `fn` is an `mnum`-by-*m*-by-*n* array with one estimate of `fn` and one estimate of `dr` per `frf`.

`FreqRange` — Frequency range
two-element vector of increasing positive values

Frequency range, specified as a two-element vector of increasing positive values contained within the range specified in `f`.

Data Types: `single` | `double`

`PhysFreq` — Natural frequencies for physical modes
vector

Natural frequencies for physical modes to include in the analysis, specified as a vector of frequency values within the range spanned by `f`. The function includes in the analysis those modes with natural frequencies closest to the values specified in the vector. If the vector contains *m* frequency values, then `fn` and `dr` have *m* rows each, and `ms` has *m* columns. If you do not specify this argument, then the function uses the entire frequency range in `f`.

Data Types: `single` | `double`

`DriveIndex` — Indices of the driving-point frequency-response function
`[1 1]` (default) | two-element vector of positive integers

Indices of the driving-point frequency-response function, specified as a two-element vector of positive integers. The first element of the vector must be less than or equal to the number of system responses. The second element of the vector must be less than or equal to the number of system excitations. Mode shapes are normalized to unity modal based on the driving point.

Example: `'DriveIndex',[2 3]` specifies that the driving-point frequency-response function is `frf(:,2,3)`.

Data Types: `single` | `double`

Output Arguments

`fn` — Natural frequencies

matrix | 3-D array

Natural frequencies, returned as a matrix or 3-D array. The size of `fn` depends on the choice of fitting algorithm specified with `'FitMethod'`:

- If you specify `'lsce'` or `'lsrf'`, then `fn` is a vector with `mnum` elements, independent of the size of `frf`. If the system has more than `mnum` oscillatory modes, then the `'lsrf'` method returns the first `mnum` least-damped modes sorted in order of increasing natural frequency.
- If you specify `'pp'`, then `fn` is an array of size `mnum`-by-*m*-by-*n* with one estimate of `fn` and one estimate of `dr` per `frf`.

`dr` — Damping ratios

matrix | 3-D array

Damping ratios for the natural frequencies in `fn`, returned as a matrix or 3-D array of the same size as `fn`.

`ms` — Mode-shape vectors

matrix

Mode-shape vectors, returned as a matrix. `ms` has `mnum` columns, each containing a mode-shape vector of length q, where q is the larger of the number of excitation channels and the number of response channels.

`ofrf` — Reconstructed frequency-response functions

vector | matrix | 3-D array

Reconstructed frequency-response functions, returned as a vector, matrix, or 3-D array with the same size as `frf`.

Algorithms

Least-Squares Complex Exponential Method

The least-squares complex exponential method computes the impulse response corresponding to each frequency-response function and fits to the response a set of complex damped sinusoids using Prony's method.

A sampled damped sinusoid can be cast in the form

$$s_i(n) = A_i e^{-b_i n/f_s}\cos(2\pi f_i n/f_s + \phi_i)$$

$$= \tfrac{1}{2}A_i e^{j\phi_i}\exp(-(b_i/f_s - j2\pi f_i/f_s)n) + \tfrac{1}{2}A_i e^{-j\phi_i}\exp(-(b_i/f_s + j2\pi f_i/f_s)n)$$

$$\equiv a_{i+}x_{i+}^n + a_{i-}x_{i-}^n,$$

where:

- f_s is the sample rate.
- f_i is the sinusoid frequency.

- b_i is the damping coefficient.
- A_i and ϕ_i are the amplitude and phase of the sinusoid.

The a_i are called *amplitudes* and the x_i are called *poles*. Prony's method expresses a sampled function $h(n)$ as a superposition of $N/2$ modes (and thus N amplitudes and poles):

$$\begin{aligned} h(0) &= a_1 x_1^0 + a_2 x_2^0 \cdots + a_N x_N^0 \\ h(1) &= a_1 x_1^1 + a_2 x_2^1 + \cdots + a_N x_N^1 \\ &\vdots \\ h(N-1) &= a_1 x_1^{N-1} + a_2 x_2^{N-1} + \cdots + a_N x_N^{N-1}. \end{aligned}$$

The poles are the roots of a polynomial with coefficients c_0, c_1, ..., c_{N-1}:

$$x_i^N + c_{N-1} x_i^{N-1} + \cdots + c_1 x_i^1 + c_0 x_i^0 = 0.$$

The coefficients are found using an autoregressive model of $L = 2N$ samples of h:

$$\begin{bmatrix} h(0) & h(1) & \cdots & h(N-1) \\ h(1) & h(2) & \cdots & h(N) \\ \vdots & \vdots & \ddots & \vdots \\ h(L-N-1) & h(L-N) & \cdots & h(L-2) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix} = -\begin{bmatrix} h(N) \\ h(N+1) \\ \vdots \\ h(L-1) \end{bmatrix}.$$

To find the poles, the algorithm uses the `roots` function. Once the poles are known, it is possible to determine the frequencies and damping factors by identifying the imaginary and real parts of the pole logarithms. The final step is solving for the amplitudes and reconstructing the impulse response using

$$\begin{bmatrix} h(0) \\ \vdots \\ h(N-1) \end{bmatrix} = \begin{bmatrix} x_1^0 & \cdots & x_N^0 \\ \vdots & \ddots & \vdots \\ x_1^{N-1} & \cdots & x_N^{N-1} \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix}.$$

The following naive MATLAB implementation summarizes the procedure:

```
N = 4;
L = 2*N;
h = rand(L,1);
c = hankel(h(1:N),h(L-N:L-1))\-h(N+1:L);
x = roots([1;c(N:-1:1)]).';
p = log(x);
hrec = x.^((0:L-1)')*(x.^((0:L-1)')\h(1:L));
sum(h-hrec)

ans =

   3.2613e-15 - 1.9297e-16i
```

The system can also be constructed to contain samples from multiple frequency-response functions and solved using least squares.

Peak-Picking Method

The peak-picking method assumes that each significant peak in the frequency-response function corresponds to exactly one natural mode. Close to a peak, the system is assumed to behave like a one-degree-of-freedom damped harmonic oscillator:

$$H(f) = \frac{-1}{(2\pi)^2}\frac{1/m}{f^2 - j2\zeta_r f_r f - f_r^2} \quad \Rightarrow \quad f_r^2 H(f) + j2\zeta_r f_r f H(f) - \frac{1}{(2\pi)^2 m} = f^2 H(f),$$

where H is the frequency-response function, f_r is the undamped resonance frequency, $\zeta_r = b/(4mk)^{1/2}$ is the relative damping, b is the damping constant, k is the elastic constant, and m is the mass.

Given a peak located at f_p, the procedure takes the peak and a fixed number of points to either side, replaces the mass term with a dummy variable, d, and computes the modal parameters by solving the system of equations

$$\begin{bmatrix} H(f_{p-k}) & j2f_{p-k}H(f_{p-k}) & -1 \\ \vdots & \vdots & \vdots \\ H(f_p) & j2f_pH(f_p) & -1 \\ \vdots & \vdots & \vdots \\ H(f_{p+k}) & j2f_{p+k}H(f_{p+k}) & -1 \end{bmatrix} \begin{bmatrix} f_r^2 \\ \zeta_r f_r \\ d \end{bmatrix} = \begin{bmatrix} f_{p-k}^2 H(f_{p-k}) \\ \vdots \\ f_p^2 H(f_p) \\ \vdots \\ f_{p+k}^2 H(f_{p+k}) \end{bmatrix}.$$

References

[1] Allemang, Randall J., and David L. Brown. “Experimental Modal Analysis and Dynamic Component Synthesis, Vol. III: Modal Parameter Estimation.” Technical Report AFWAL-TR-87-3069. Air Force Wright Aeronautical Laboratories, Wright-Patterson Air Force Base, OH, December 1987.

[2] Brandt, Anders. *Noise and Vibration Analysis: Signal Analysis and Experimental Procedures*. Chichester, UK: John Wiley & Sons, 2011.

[3] Ozdemir, Ahmet Arda, and Suat Gumussoy. "Transfer Function Estimation in System Identification Toolbox via Vector Fitting." *Proceedings of the 20th World Congress of the International Federation of Automatic Control*, Toulouse, France, July 2017.

See Also

`modalfrf` | `modalsd` | `n4sid` | `tfest` | `tfestimate`

Topics

“Modal Analysis of Identified Models”
“System Identification Overview” (System Identification Toolbox)
“System Identification Workflow” (System Identification Toolbox)
“Supported Continuous- and Discrete-Time Models” (System Identification Toolbox)

Introduced in R2017a

modalfrf

Frequency-response functions for modal analysis

Syntax

```
frf = modalfrf(x,y,fs,window)
frf = modalfrf(x,y,fs,window,noverlap)

frf = modalfrf(___,Name,Value)

[frf,f,coh] = modalfrf(___)

[frf,f] = modalfrf(sys)
frf = modalfrf(sys,f)

modalfrf(___)
```

Description

`frf = modalfrf(x,y,fs,window)` estimates a matrix of frequency response functions, `frf`, from the excitation signals, `x`, and the response signals, `y`, all sampled at a rate `fs`. The output, `frf`, is an H_1 estimate computed using Welch's method with `window` to window the signals. `x` and `y` must have the same number of rows. If `x` or `y` is a matrix, each column represents a signal.

The system response, `y`, is assumed to contain acceleration measurements. To compute a frequency-response function starting from displacement or velocity measurements, use the `'Sensor'` argument. `modalfrf` always outputs the frequency-response function in dynamic flexibility (receptance) format irrespective of the sensor type.

`frf = modalfrf(x,y,fs,window,noverlap)` specifies `noverlap` samples of overlap between adjoining segments.

`frf = modalfrf(___,Name,Value)` specifies options using name-value arguments, using any combination of inputs from previous syntaxes. Options include the estimator, the measurement configuration, and the type of sensor measuring the system response.

`[frf,f,coh] = modalfrf(___)` also returns the frequency vector corresponding to each frequency-response function, as well as the multiple coherence matrix.

`[frf,f] = modalfrf(sys)` computes the frequency-response function of the identified model `sys`. Use estimation commands like `ssest`, `n4sid`, or `tfest` to create `sys` from time-domain input and output signals. This syntax allows use only of the `'Sensor'` name-value argument. You must have a System Identification Toolbox license to use this syntax.

`frf = modalfrf(sys,f)` specifies the frequencies at which to compute `frf`. This syntax allows use only of the `'Sensor'` name-value argument. You must have a System Identification Toolbox license to use this syntax.

`modalfrf(___)` with no output arguments plots the frequency response functions in the current figure. The plots are limited to the first four excitations and four responses.

Examples

Frequency-Response Function of Hammer Excitation

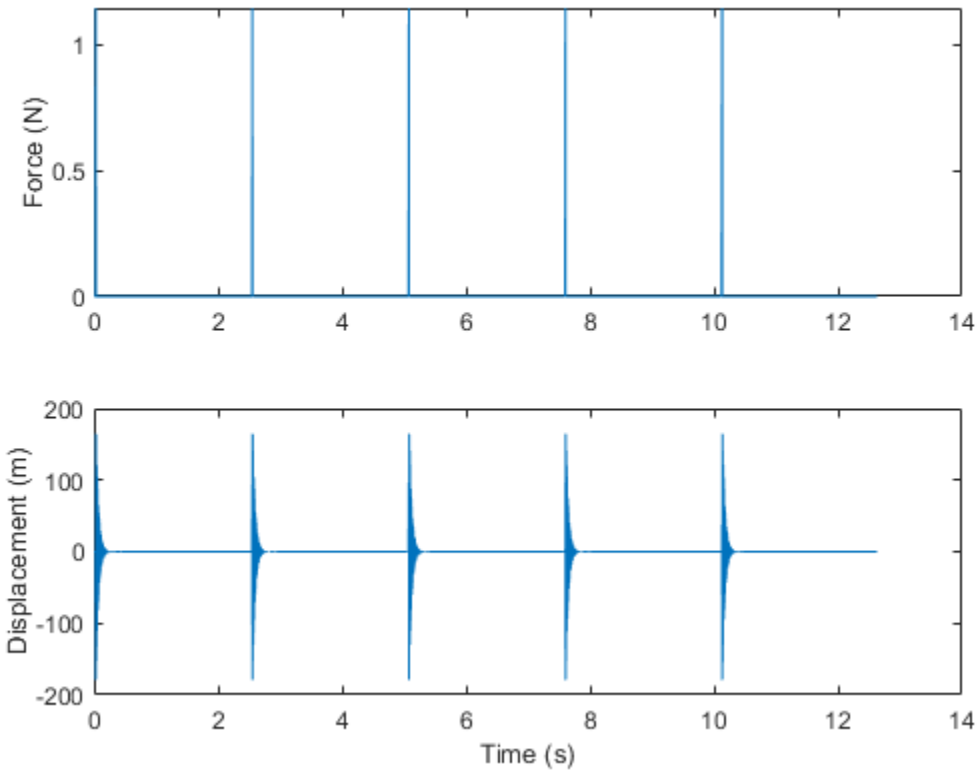
Visualize the frequency-response function of a single-input/single-output hammer excitation.

Load a data file that contains:

- `Xhammer` — An input excitation signal consisting of five hammer blows delivered periodically.
- `Yhammer` — The response of a system to the input. `Yhammer` is measured as a displacement.

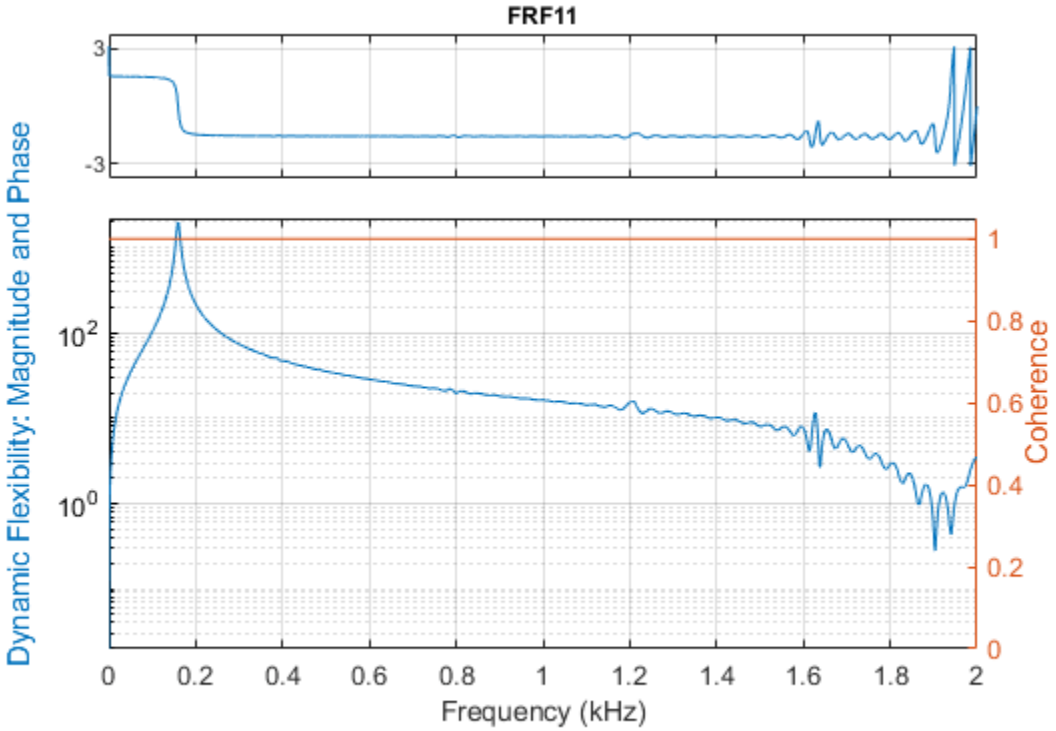
The signals are sampled at 4 kHz. Plot the excitation and output signals.

```
load modaldata

subplot(2,1,1)
plot(thammer,Xhammer(:))
ylabel('Force (N)')
subplot(2,1,2)
plot(thammer,Yhammer(:))
ylabel('Displacement (m)')
xlabel('Time (s)')
```



Compute and display the frequency-response function. Window the signals using a rectangular window. Specify that the window covers the period between hammer blows.

```
clf
winlen = size(Xhammer,1);
modalfrf(Xhammer(:),Yhammer(:),fs,winlen,'Sensor','dis')
```



MIMO Frequency-Response Functions

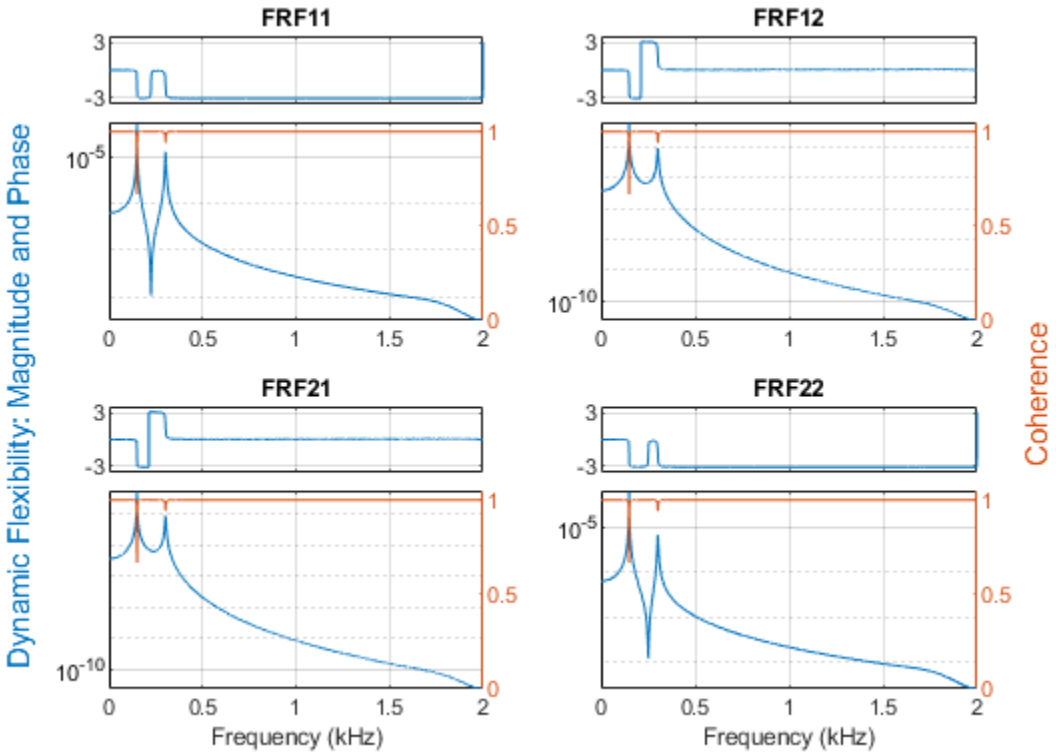
Compute the frequency-response functions for a two-input/two-output system excited by random noise.

Load a data file that contains `Xrand`, the input excitation signal, and `Yrand`, the system response. Compute the frequency-response functions using a 5000-sample Hann window and 50% overlap between adjoining data segments. Specify that the output measurements are displacements.

```
load modaldata
winlen = 5000;

frf = modalfrf(Xrand,Yrand,fs,hann(winlen),0.5*winlen,'Sensor','dis');
```

Use the plotting functionality of `modalfrf` to visualize the responses.

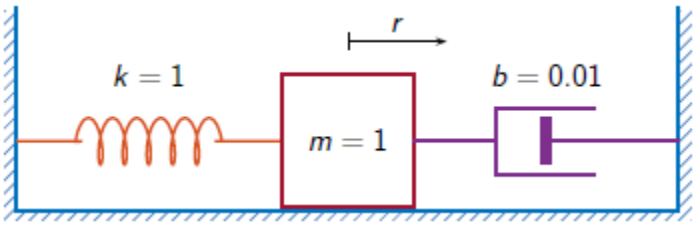
```
modalfrf(Xrand,Yrand,fs,hann(winlen),0.5*winlen,'Sensor','dis')
```



Frequency-Response Function of SISO System

Estimate the frequency-response function for a simple single-input/single-output system and compare it to the definition.

A one-dimensional discrete-time oscillating system consists of a unit mass, m, attached to a wall by a spring with elastic constant $k = 1$. A sensor samples the displacement of the mass at $F_s = 1$ Hz. A damper impedes the motion of the mass by exerting on it a force proportional to speed, with damping constant $b = 0.01$.



Generate 3000 time samples. Define the sampling interval $\Delta t = 1/F_s$.

```
Fs = 1;
dt = 1/Fs;
N = 3000;
```

```
t = dt*(0:N-1);
b = 0.01;
```

The system can be described by the state-space model

$$x(k+1) = Ax(k) + Bu(k),$$
$$y(k) = Cx(k) + Du(k),$$

where $x = [r\ v]^T$ is the state vector, r and v are respectively the displacement and velocity of the mass, u is the driving force, and $y = r$ is the measured output. The state-space matrices are

$$A = \exp(A_c \Delta t), \quad B = A_c^{-1}(A - I)B_c, \quad C = [1\ 0], \quad D = 0,$$

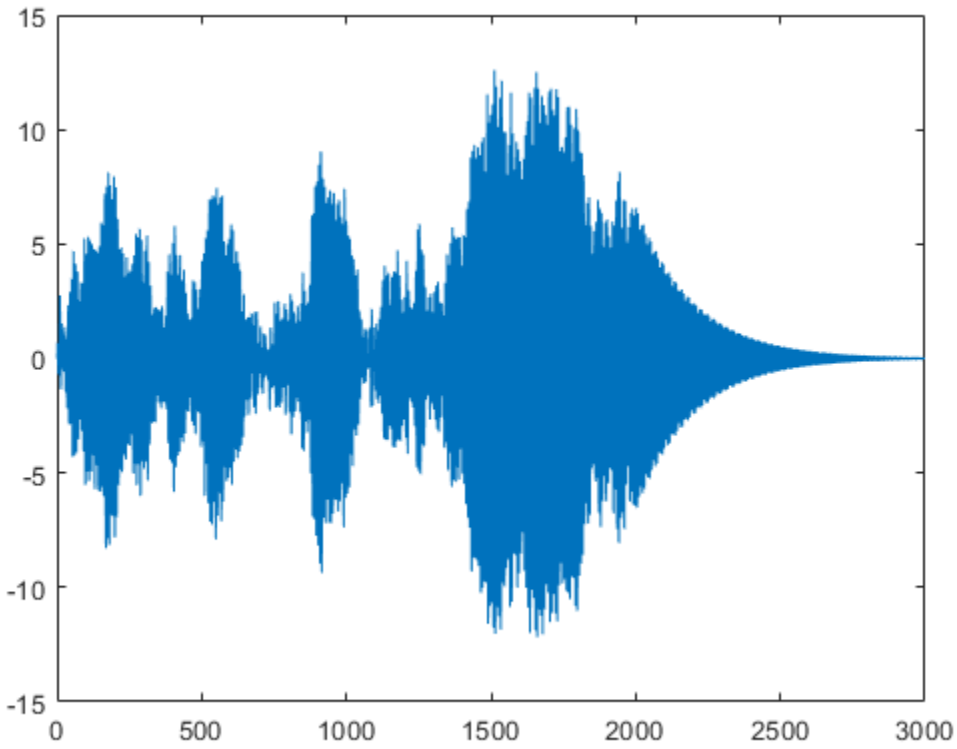
I is the 2×2 identity, and the continuous-time state-space matrices are

$$A_c = \begin{bmatrix} 0 & 1 \\ -1 & -b \end{bmatrix}, \quad B_c = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

```
Ac = [0 1;-1 -b];
A = expm(Ac*dt);

Bc = [0;1];
B = Ac\(A-eye(2))*Bc;

C = [1 0];
D = 0;
```

The mass is driven by random input for the first 2000 seconds and then left to return to rest. Use the state-space model to compute the time evolution of the system starting from an all-zero initial state. Plot the displacement of the mass as a function of time.

```
rng default
u = randn(1,N)/2;
u(2001:end) = 0;

y = 0;
x = [0;0];
for k = 1:N
    y(k) = C*x + D*u(k);
    x = A*x + B*u(k);
end

plot(t,y)
```

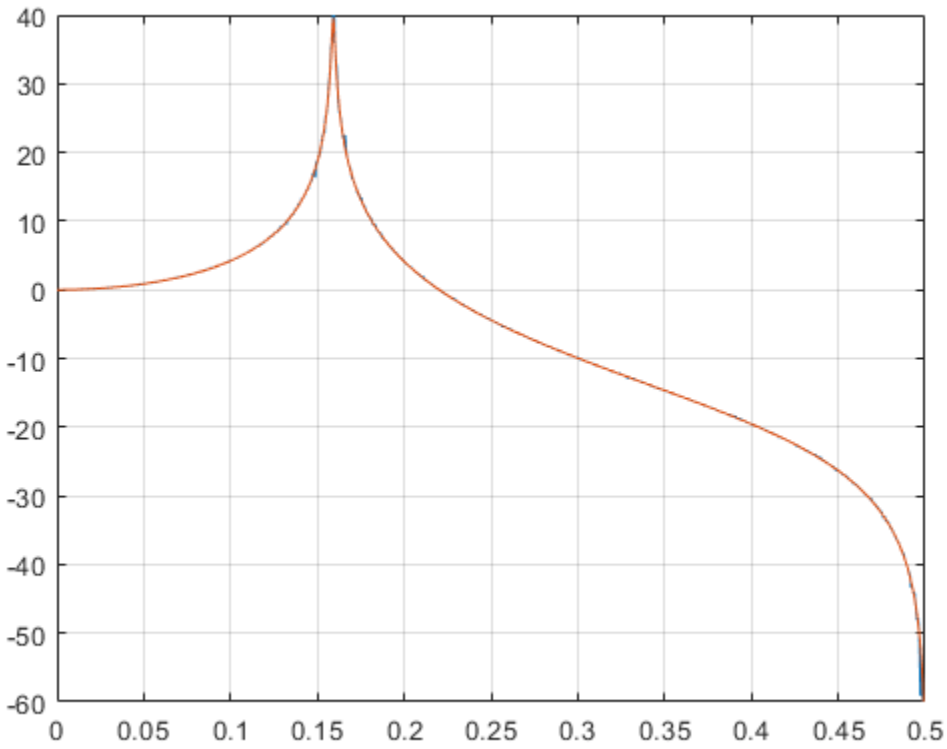


Estimate the modal frequency-response function of the system. Use a Hann window half as long as the measured signals. Specify that the output is the displacement of the mass.

```
wind = hann(N/2);

[frf,f] = modalfrf(u',y',Fs,wind,'Sensor','dis');
```

The frequency-response function of a discrete-time system can be expressed as the Z-transform of the time-domain transfer function of the system, evaluated at the unit circle. Compare the `modalfrf` estimate with the definition.

```
[b,a] = ss2tf(A,B,C,D);

nfs = 2048;
fz = 0:1/nfs:1/2-1/nfs;
z = exp(2j*pi*fz);
ztf = polyval(b,z)./polyval(a,z);

plot(f,20*log10(abs(frf)))
hold on
plot(fz*Fs,20*log10(abs(ztf)))
hold off
grid
ylim([-60 40])
```



Estimate the natural frequency and the damping ratio for the vibration mode.

```
[fn,dr] = modalfit(frf,f,Fs,1,'FitMethod','PP')
```

```
fn = 0.1593
```

```
dr = 0.0043
```

Compare the natural frequency to $1/2\pi$, which is the theoretical value for the undamped system.

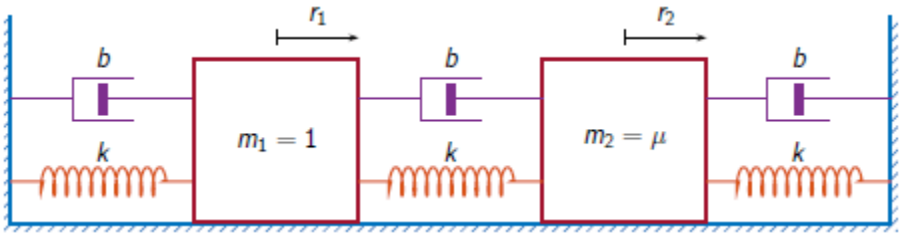
```
theo = 1/(2*pi)
```

```
theo = 0.1592
```

Modal Parameters of Two-Body Oscillator

Estimate the frequency-response function and modal parameters of a simple multi-input/multi-output system.

An ideal one-dimensional oscillating system consists of two masses, m_1 and m_2, confined between two walls. The units are such that $m_1 = 1$ and $m_2 = \mu$. Each mass is attached to the nearest wall by a spring with an elastic constant k. An identical spring connects the two masses. Three dampers impede the motion of the masses by exerting on them forces proportional to speed, with damping constant b. Sensors sample r_1 and r_2, the displacements of the masses, at $F_s = 50$ Hz.



Generate 30,000 time samples, equivalent to 600 seconds. Define the sampling interval $\Delta t = 1/F_s$.

```
Fs = 50;
dt = 1/Fs;
N = 30000;
t = dt*(0:N-1);
```

The system can be described by the state-space model

$$x(k+1) = Ax(k) + Bu(k),$$
$$y(k) = Cx(k) + Du(k),$$

where $x = [r_1\ v_1\ r_2\ v_2]^T$ is the state vector, r_i and v_i are respectively the location and the velocity of the ith mass, $u = [u_1\ u_2]^T$ is the vector of input driving forces, and $y = [r_1\ r_2]^T$ is the output vector. The state-space matrices are

$$A = \exp(A_c \Delta t), \quad B = A_c^{-1}(A - I)B_c, \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix},$$

I is the 4×4 identity, and the continuous-time state-space matrices are

$$A_c = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -2k & -2b & k & b \\ 0 & 0 & 0 & 1 \\ k/\mu & b/\mu & -2k/\mu & -2b/\mu \end{bmatrix}, \quad B_c = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1/\mu \end{bmatrix}.$$

Set $k = 400$, $b = 0.1$, and $\mu = 1/10$.

```
k = 400;
b = 0.1;
m = 1/10;

Ac = [0 1 0 0;-2*k -2*b k b;0 0 0 1;k/m b/m -2*k/m -2*b/m];
A = expm(Ac*dt);
Bc = [0 0;1 0;0 0;0 1/m];
B = Ac\(A-eye(4))*Bc;
C = [1 0 0 0;0 0 1 0];
D = zeros(2);
```

The masses are driven by random input throughout the measurement. Use the state-space model to compute the time evolution of the system starting from an all-zero initial state.

```
rng default
u = randn(2,N);
```

```
y = [0;0];
x = [0;0;0;0];
for kk = 1:N
    y(:,kk) = C*x + D*u(:,kk);
    x = A*x + B*u(:,kk);
end
```

Use the input and output data to estimate the transfer function of the system as a function of frequency. Use a 15000-sample Hann window with 9000 samples of overlap between adjoining segments. Specify that the measured outputs are displacements.

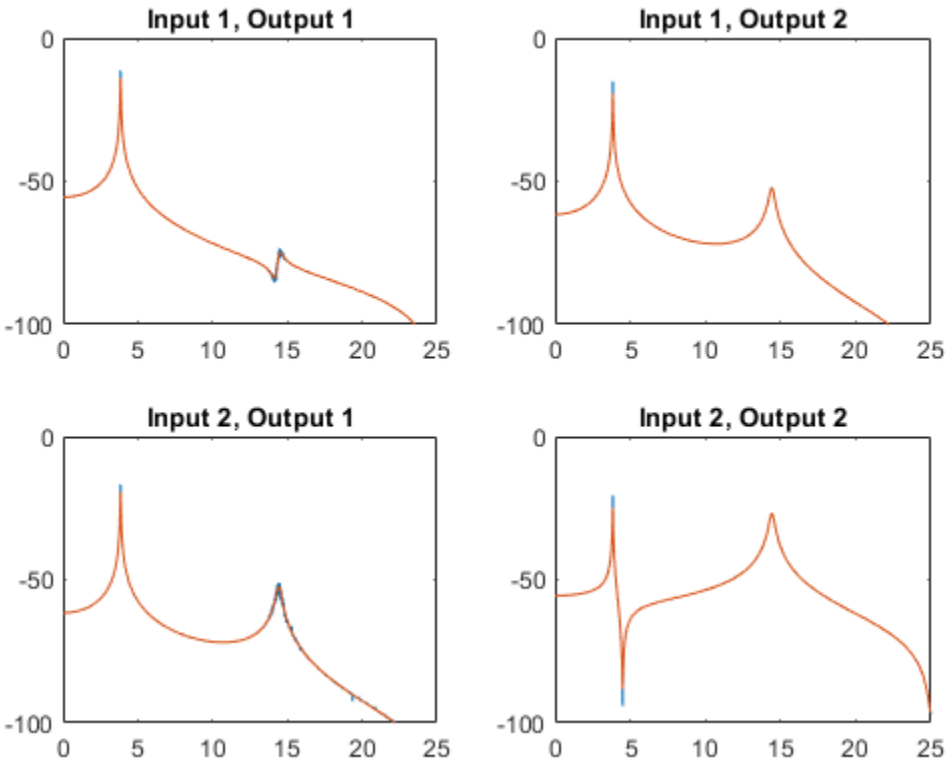
```
wind = hann(15000);
nove = 9000;
[FRF,f] = modalfrf(u',y',Fs,wind,nove,'Sensor','dis');
```

Compute the theoretical transfer function as the Z-transform of the time-domain transfer function, evaluated at the unit circle.

```
nfs = 2048;
fz = 0:1/nfs:1/2-1/nfs;
z = exp(2j*pi*fz);

[b1,a1] = ss2tf(A,B,C,D,1);
[b2,a2] = ss2tf(A,B,C,D,2);

frf(1,:,1) = polyval(b1(1,:),z)./polyval(a1,z);
frf(1,:,2) = polyval(b1(2,:),z)./polyval(a1,z);
frf(2,:,1) = polyval(b2(1,:),z)./polyval(a2,z);
frf(2,:,2) = polyval(b2(2,:),z)./polyval(a2,z);
```

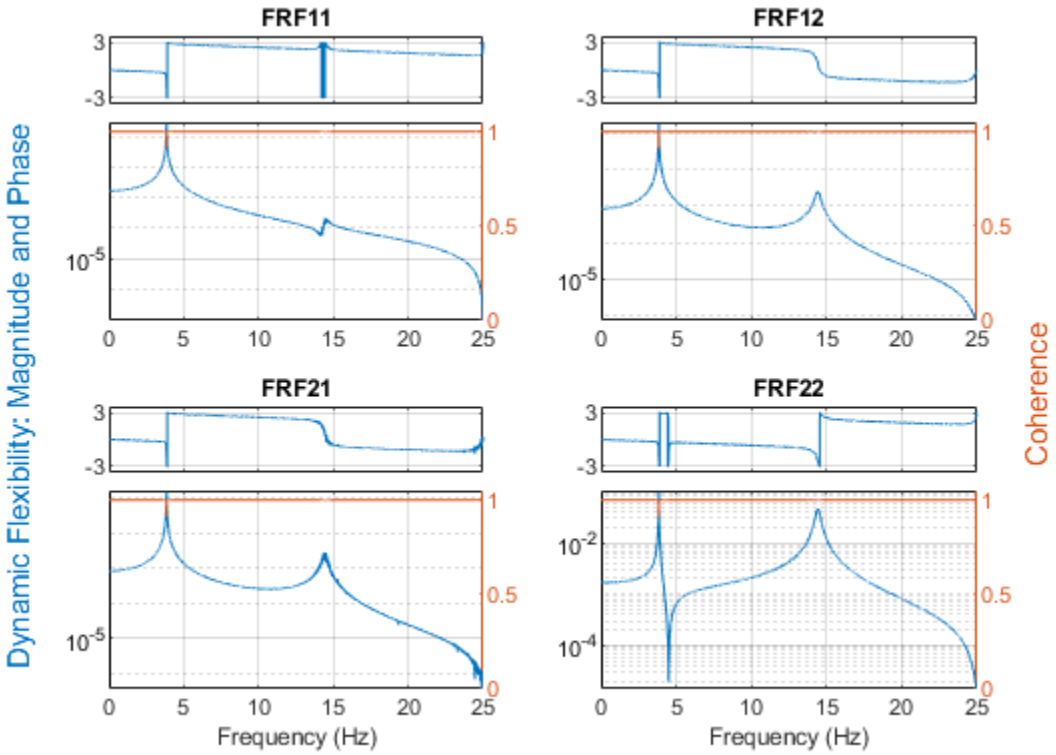
Plot the estimates and overlay the theoretical predictions.

```
for jk = 1:2
    for kj = 1:2
        subplot(2,2,2*(jk-1)+kj)
        plot(f,20*log10(abs(FRF(:,jk,kj))))
        hold on
        plot(fz*Fs,20*log10(abs(frf(jk,:,kj))))
        hold off
        axis([0 Fs/2 -100 0])
        title(sprintf('Input %d, Output %d',jk,kj))
    end
end
```



Plot the estimates by using the syntax of `modalfrf` with no output arguments.

```
figure
modalfrf(u',y',Fs,wind,nove,'Sensor','dis')
```



Estimate the natural frequencies, damping ratios, and mode shapes of the system. Use the peak-picking method for the calculation.

```
[fn,dr,ms] = modalfit(FRF,f,Fs,2,'FitMethod','pp');
fn
```

```
fn =
fn(:,:,1) =

    3.8466    3.8466
    3.8495    3.8495


fn(:,:,2) =

    3.8492    3.8490
    3.8552   14.4684
```

Compare the natural frequencies to the theoretical predictions for the undamped system.

```
undamped = sqrt(eig([2*k -k;-k/m 2*k/m]))/2/pi
```

```
undamped = 2×1

    3.8470
   14.4259
```

Frequency-Response Function Using Subspace Method

Compute the frequency-response function of a two-input/six-output data set corresponding to a steel frame.

Load a structure containing the input excitations and the output accelerometer measurements. The system is sampled at 1024 Hz for about 3.9 seconds.

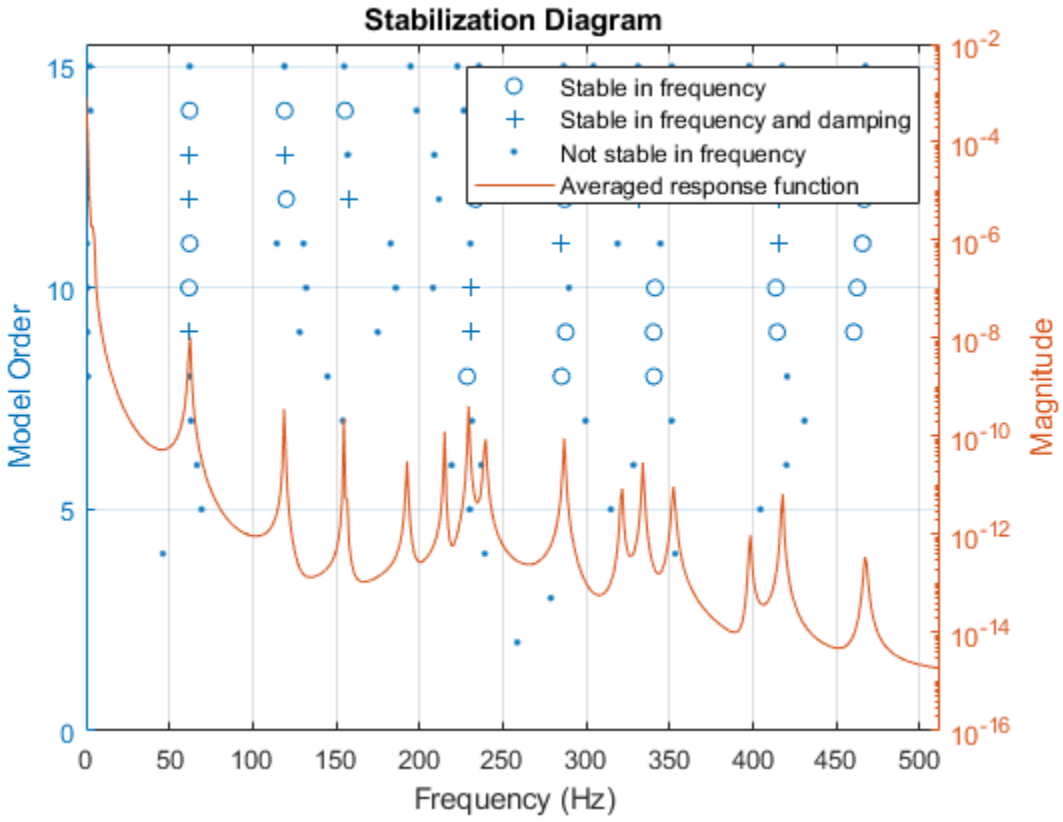
```
load modaldata SteelFrame
X = SteelFrame.Input;
Y = SteelFrame.Output;
fs = SteelFrame.Fs;
```

Use the subspace method to compute the frequency-response functions. Divide the input and output signals into nonoverlapping, 1000-sample segments. Window each segment using a rectangular window. Specify a model order of 36.

```
[frf,f] = modalfrf(X,Y,fs,1000,'Estimator','subspace','Order',36);
```

Visualize the stabilization diagram for the system. Identify up to 15 physical modes.

```
modalsd(frf,f,fs,'MaxModes',15)
```



Input Arguments

`x` — Excitation signals

vector | matrix

Excitation signals, specified as a vector or matrix.

Data Types: `single` | `double`

`y` — Response signals

vector | matrix

Response signals, specified as a vector or matrix.

Data Types: `single` | `double`

`fs` — Sample rate

positive scalar

Sample rate, specified as a positive scalar expressed in hertz.

Data Types: `single` | `double`

`window` — Window

integer | vector

Window, specified as an integer or as a row or column vector. Use `window` to divide the signal into segments:

- If `window` is an integer, then `modalfrf` divides `x` and `y` into segments of length `window` and windows each segment with a rectangular window of that length.
- If `window` is a vector, then `modalfrf` divides `x` and `y` into segments of the same length as the vector and windows each segment using `window`.
- If `'Estimator'` is specified as `'subspace'`, then `modalfrf` ignores the shape of `window` and uses its length to determine the number of frequency points in the returned frequency-response function.

If the length of `x` and `y` cannot be divided exactly into an integer number of segments with `noverlap` overlapping samples, then the signals are truncated accordingly.

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

0 (default) | positive integer

Number of overlapped samples, specified as a positive integer.

- If `window` is a scalar, then `noverlap` must be smaller than `window`.
- If `window` is a vector, then `noverlap` must be smaller than the length of `window`.

Data Types: `double` | `single`

sys — Identified system

model with identified parameters

Identified system, specified as a model with identified parameters. Use estimation commands like `ssest`, `n4sid`, or `tfest` to create `sys` from time-domain input and output signals. See “Modal Analysis of Identified Models” for an example. Syntaxes that use `sys` typically require less data than syntaxes that use nonparametric methods. You must have a System Identification Toolbox license to use this input argument.

Example: `idss([0.5418 0.8373;-0.8373 0.5334],[0.4852;0.8373],[1 0],0,[0;0],[0;0],1)` generates an identified state-space model corresponding to a unit mass attached to a wall by a spring of unit elastic constant and a damper with constant 0.01. The displacement of the mass is sampled at 1 Hz.

Example: `idtf([0 0.4582 0.4566],[1 -1.0752 0.99],1)` generates an identified transfer-function model corresponding to a unit mass attached to a wall by a spring of unit elastic constant and a damper with constant 0.01. The displacement of the mass is sampled at 1 Hz.

f — Frequencies

vector

Frequencies, specified as a vector expressed in Hz.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Sensor','vel','Est','H1'` specifies that the input signal consists of velocity measurements and that the estimator of choice is H1.

Estimator — Estimator

`'H1'` (default) | `'H2'` | `'Hv'` | `'subspace'`

Estimator, specified as `'H1'`, `'H2'`, `'Hv'`, or `'subspace'`. See “Transfer Function” on page 1-2648 for more information about the H_1 and H_2 estimators.

- Use `'H1'` when the noise is uncorrelated with the excitation signals.
- Use `'H2'` when the noise is uncorrelated with the response signals. In this case, the number of excitation signals must equal the number of response signals.
- Use `'Hv'` to minimize the discrepancy between modeled and estimated response data by minimizing the trace of the error matrix. H_v is the geometric mean of H_1 and H_2: $H_v = (H_1H_2)^{1/2}$

  The measurement must be single-input/single-output (SISO).
- Use `'subspace'` to compute the frequency-response functions using a state-space model. In this case, the `noverlap` argument is ignored. This method typically requires less data than nonparametric approaches. See `n4sid` for more information.

Feedthrough — Presence of feedthrough in state-space model

`false` (default) | `true`

Presence of feedthrough in state-space model, specified as a logical value. This argument is available only if `'Estimator'` is specified as `'subspace'`.

Data Types: `logical`

`Measurement` — Measurement configuration

`'fixed'` (default) | `'rovinginput'` | `'rovingoutput'`

Measurement configuration for equal numbers of excitation and response channels, specified as `'fixed'`, `'rovinginput'`, or `'rovingoutput'`.

- Use `'fixed'` when there are excitation sources and sensors at fixed locations of the system. Each excitation contributes to every response.
- Use `'rovinginput'` when the measurements result from a roving excitation (or roving hammer) test. A single sensor is kept at a fixed location of the system. A single excitation source is placed at multiple locations and produces one sensor response per location. The function output `frf(:,:,i) = modalfrf(x(:,i),y(:,i))`.
- Use `'rovingoutput'` when the measurements result from a roving sensor test. A single excitation source is kept at a fixed location of the system. A single sensor is placed at multiple locations and responds to one excitation per location. The function output `frf(:,i) = modalfrf(x(:,i),y(:,i))`.

`Order` — State-space model order

`1:10` (default) | integer | row vector of integers

State-space model order, specified as an integer or row vector of integers. If you specify a vector of integers, then the function selects an optimal order value from the specified range. This argument is available only if `'Estimator'` is specified as `'subspace'`.

Data Types: `single` | `double`

`Sensor` — Sensor type

`'acc'` (default) | `'dis'` | `'vel'`

Sensor type, specified as `'acc'`, `'vel'`, or `'dis'`.

- `'acc'` — Specifies that the response signal of the system is proportional to acceleration.
- `'vel'` — Specifies that the response signal of the system is proportional to velocity.
- `'dis'` — Specifies that the response signal of the system is proportional to displacement.

`modalfrf` always outputs the frequency-response function in dynamic flexibility (receptance) format irrespective of the sensor type.

Example: Undamped Harmonic Oscillator

The motion of a simple undamped harmonic oscillator of unit mass and elastic constant sampled at a rate $f_s = 1/\Delta t$ is described by the transfer function

$$H(z) = \frac{N_{\text{Sensor}}(z)}{1 - 2z^{-1}\cos\Delta t + z^{-2}},$$

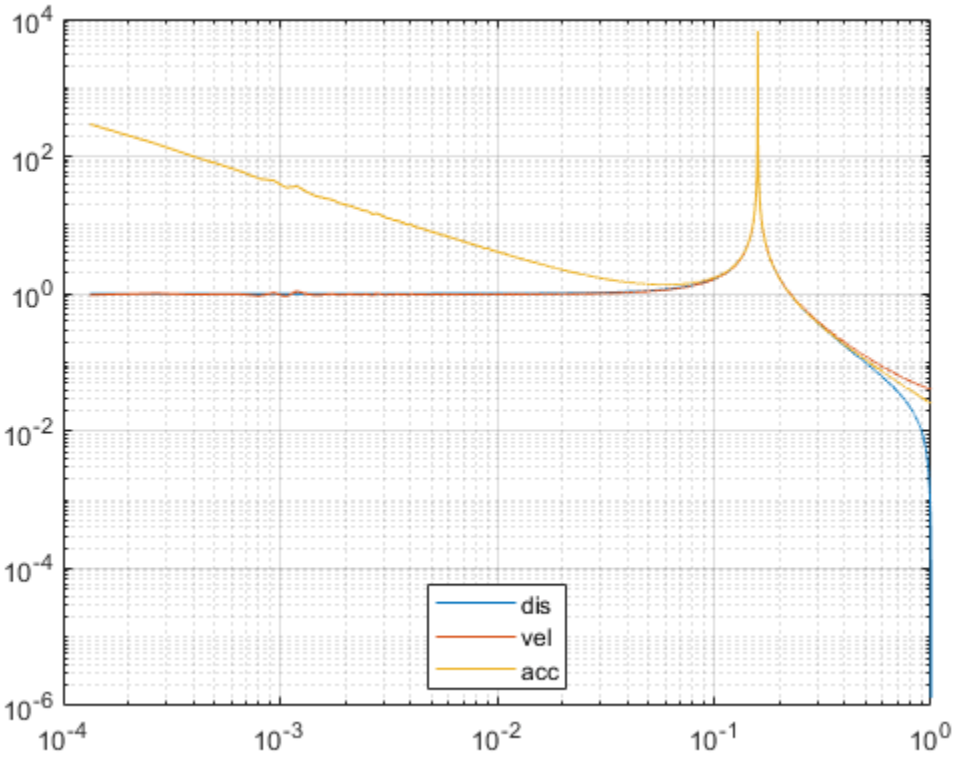
where the numerator depends on the magnitude being measured:

- Displacement: $N_{\text{dis}}(z) = \left(z^{-1} + z^{-2}\right)(1 - \cos\Delta t)$

- Velocity: $N_{\mathrm{vel}}(z) = \left(z^{-1} - z^{-2}\right)\sin\Delta t$
- Acceleration: $N_{\mathrm{acc}}(z) = \left(1 - z^{-1}\right) - \left(z^{-1} - z^{-2}\right)\cos\Delta t$

Compute the frequency-response function for the three possible system response sensor types. Use a sample rate of 2 Hz and 30,000 samples of white noise as input.

```
fs = 2;
dt = 1/fs;
N = 30000;

u = randn(N,1);

ydis = filter((1-cos(dt))*[0 1 1],[1 -2*cos(dt) 1],u);
[frfd,fd] = modalfrf(u,ydis,fs,hann(N/2),Sensor="dis");

yvel = filter(sin(dt)*[0 1 -1],[1 -2*cos(dt) 1],u);
[frfv,fv] = modalfrf(u,yvel,fs,hann(N/2),Sensor="vel");

yacc = filter([1 -(1+cos(dt)) cos(dt)],[1 -2*cos(dt) 1],u);
[frfa,fa] = modalfrf(u,yacc,fs,hann(N/2),Sensor="acc");

loglog(fd,abs(frfd),fv,abs(frfv),fa,abs(frfa))
grid
legend(["dis" "vel" "acc"],Location="best")
```



In all cases, the generated frequency-response function is in a format corresponding to displacement. Velocity and acceleration measurements are first and second time derivatives, respectively, of displacement measurements. The frequency-response functions are equivalent in the range around the natural frequency of the system. Away from the natural frequency, the frequency-response functions differ.

Output Arguments

`frf` — Frequency-response functions

vector | matrix | 3-D array

Frequency-response functions, returned as a vector, matrix, or 3-D array. `frf` has size *p*-by-*m*-by-*n*, where *p* is the number of frequency bins, *m* is the number of responses, and *n* is the number of excitation signals.

`modalfrf` always outputs the frequency-response function in dynamic flexibility (receptance) format irrespective of the sensor type.

`f` — Frequencies

vector

Frequencies, returned as a vector.

`coh` — Multiple coherence matrix

matrix

Multiple coherence matrix, returned as a matrix. `coh` has one column for each response signal.

References

[1] "Dynamic Stiffness, Compliance, Mobility, and more..." Siemens, last modified 2019, https://community.sw.siemens.com/s/article/dynamic-stiffness-compliance-mobility-and-more.

[2] Brandt, Anders. *Noise and Vibration Analysis: Signal Analysis and Experimental Procedures*. Chichester, UK: John Wiley & Sons, 2011.

[3] Irvine, Tom. "An Introduction to Frequency Response Functions," Vibrationdata, 2000, https://vibrationdata.com/tutorials2/frf.pdf.

[4] Vold, Håvard, John Crowley, and G. Thomas Rocklin. "New Ways of Estimating Frequency Response Functions." *Sound and Vibration*. Vol. 18, November 1984, pp. 34–38.

See Also

`modalfit` | `modalsd` | `n4sid` | `tfestimate`

Topics

“Modal Analysis of Identified Models”
“System Identification Overview” (System Identification Toolbox)
“System Identification Workflow” (System Identification Toolbox)
“Supported Continuous- and Discrete-Time Models” (System Identification Toolbox)

Introduced in R2017a

modalsd

Generate stabilization diagram for modal analysis

Syntax

```
modalsd(frf,f,fs)
modalsd(frf,f,fs,Name,Value)

fn = modalsd( ___ )
```

Description

`modalsd(frf,f,fs)` generates a stabilization diagram in the current figure. `modalsd` estimates the natural frequencies and damping ratios from 1 to 50 modes and generates the diagram using the least-squares complex exponential (LSCE) algorithm. `fs` is the sample rate. The frequency, `f`, is a vector with a number of elements equal to the number of rows of the frequency-response function, `frf`. You can use this diagram to differentiate between computational and physical modes.

`modalsd(frf,f,fs,Name,Value)` specifies options using name-value pair arguments.

`fn = modalsd( ___ )` returns a cell array of natural frequencies, `fn`, identified as being stable between consecutive model orders. The *i*th element contains a length-*i* vector of natural frequencies of stable poles. Poles that are not stable are returned as `NaN`s. This syntax accepts any combination of inputs from previous syntaxes.

Examples

MIMO Stabilization Diagram

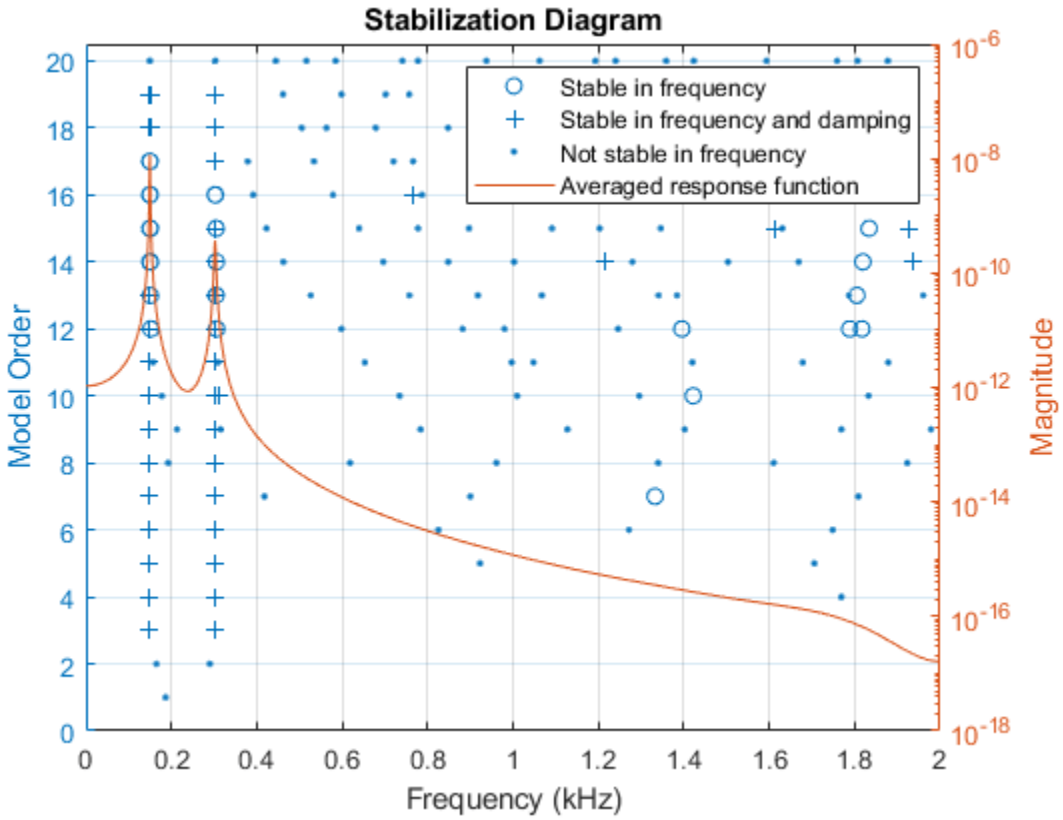
Compute the frequency-response functions for a two-input/two-output system excited by random noise.

Load the data file. Compute the frequency-response functions using a 5000-sample Hann window and 50% overlap between adjoining data segments. Specify that the output measurements are displacements.

```
load modaldata
winlen = 5000;

[frf,f] = modalfrf(Xrand,Yrand,fs,hann(winlen),0.5*winlen,'Sensor','dis');
```

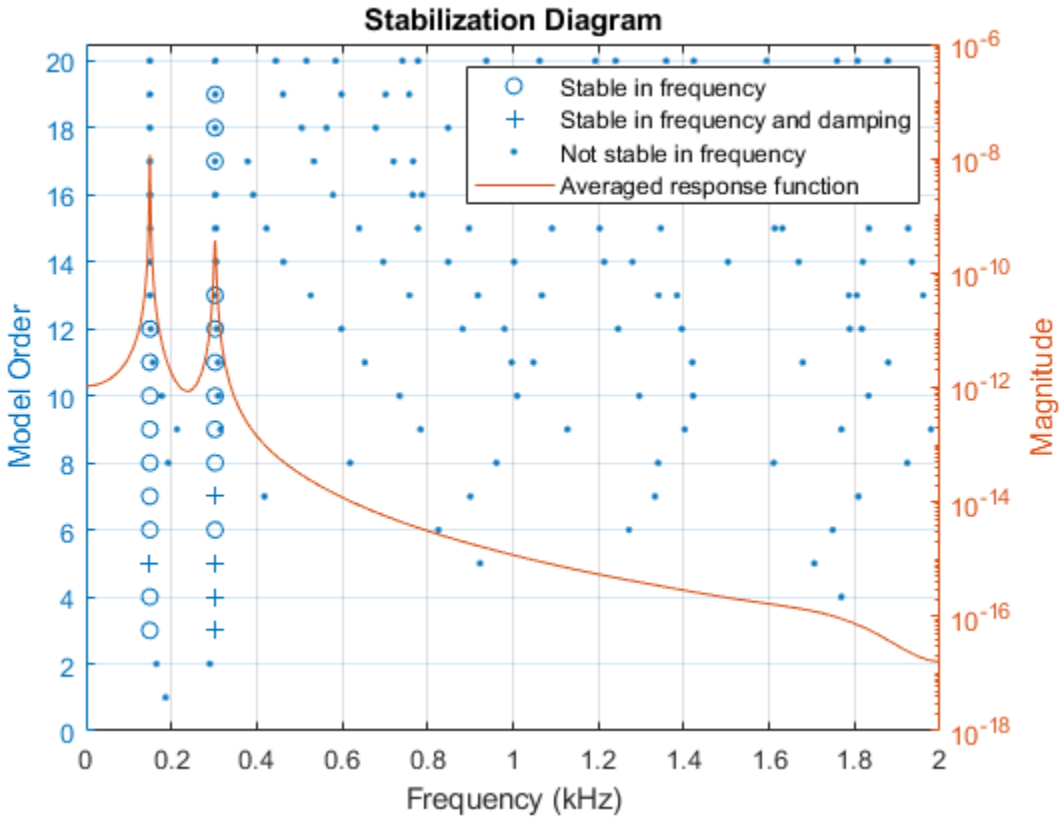
Generate a stabilization diagram to identify up to 20 physical modes.

```
modalsd(frf,f,fs,'MaxModes',20)
```



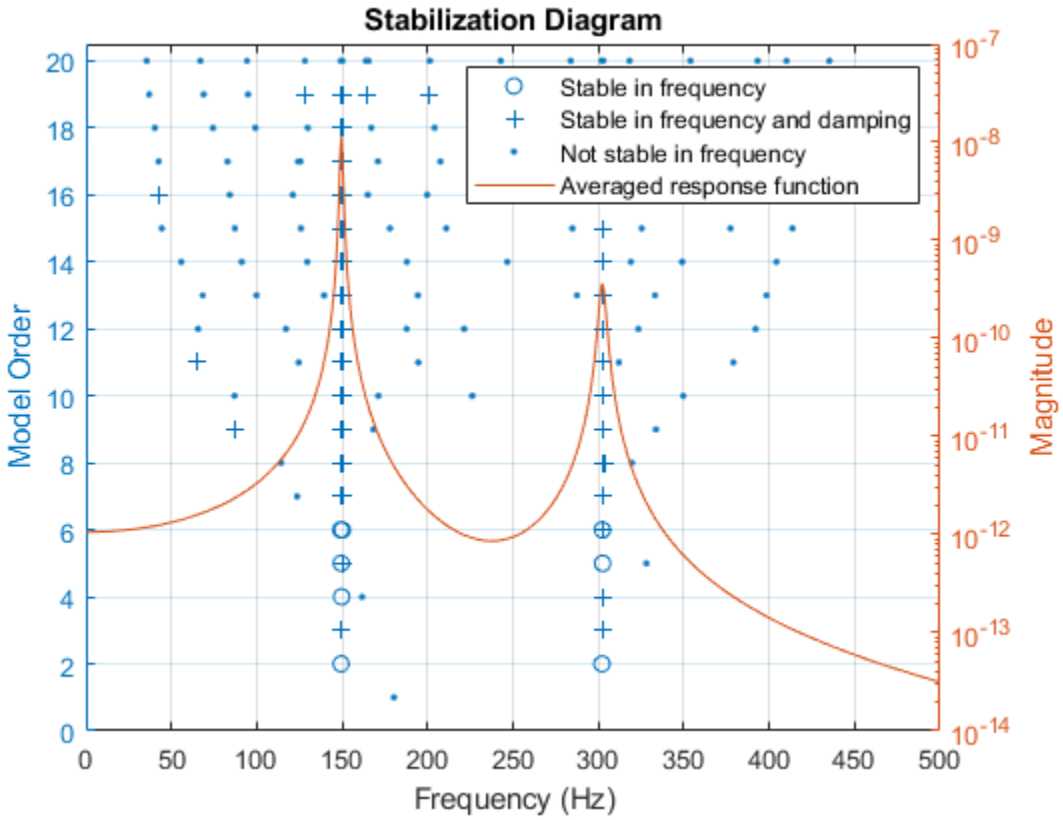
Repeat the computation, but now tighten the criteria for stability. Classify a given pole as stable in frequency if its natural frequency changes by less than 0.01% as the model order increases. Classify a given pole as stable in damping if the damping ratio estimate changes by less than 0.2% as the model order increases.

```
modalsd(frf,f,fs,'MaxModes',20,'SCriteria',[1e-4 0.002])
```



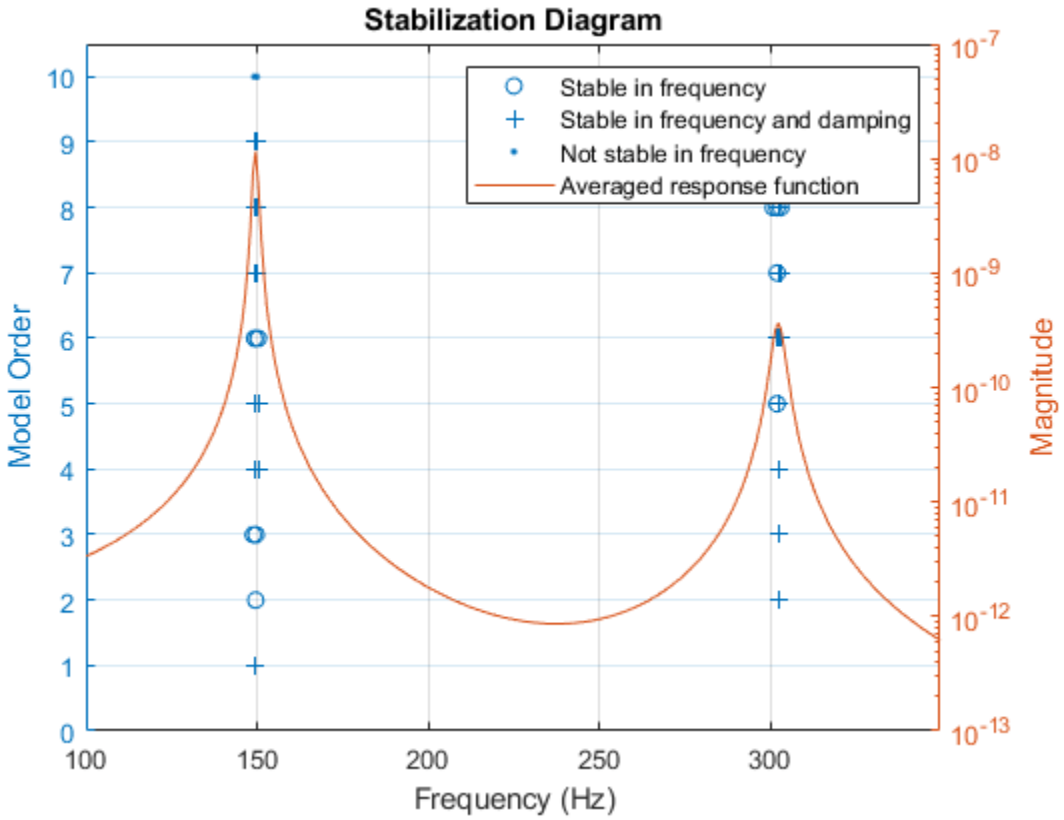
Restrict the frequency range to between 0 and 500 Hz. Relax the stability criteria to 0.5% for frequency and 10% for damping.

```
modalsd(frf,f,fs,'MaxModes',20,'SCriteria',[5e-3 0.1],'FreqRange',[0 500])
```



Repeat the computation using the least-squares rational function algorithm. Restrict the frequency range from 100 Hz to 350 Hz and identify up to 10 physical modes.

```
modalsd(frf,f,fs,'MaxModes',10,'FreqRange',[100 350],'FitMethod','lsrf')
```



Input Arguments

`frf` — Frequency-response functions

vector | matrix | 3-D array

Frequency-response functions, specified as a vector, matrix, or 3-D array. `frf` has size *p*-by-*m*-by-*n*, where *p* is the number of frequency bins, *m* is the number of response signals, and *n* is the number of excitation signals used to estimate the transfer function.

Example: `tfestimate(randn(1,1000),sin(2*pi*(1:1000)/4)+randn(1,1000)/10)` approximates the frequency response of an oscillator.

Data Types: `single` | `double`
Complex Number Support: Yes

`f` — Frequencies

vector

Frequencies, specified as a vector. The number of elements of `f` must equal the number of rows of `frf`.

Data Types: `single` | `double`

`fs` — Sample rate of measurement data

positive scalar

Sample rate of measurement data, specified as a positive scalar expressed in hertz.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MaxModes',20,'FreqRange',[0 500]` computes up to 20 physical modes and restricts the frequency range to between 0 and 500 Hz.

`FitMethod` — Fitting algorithm

`'lsce'` (default) | `'lsrf'`

Fitting algorithm, specified as the comma-separated pair consisting of `'FitMethod'` and `'lsce'` or `'lsrf'`.

- `'lsce'` — Least-squares complex exponential method.
- `'lsrf'` — Least-squares rational function estimation method. The method is described in [2]. See “Continuous-Time Transfer Function Estimation Using Continuous-Time Frequency-Domain Data” (System Identification Toolbox) for more information. This algorithm typically requires less data than nonparametric approaches.

`FreqRange` — Frequency range

two-element vector of positive values

Frequency range, specified as the comma-separated pair consisting of `'FreqRange'` and a two-element vector of increasing, positive values contained within the range specified in `f`.

Data Types: `single` | `double`

`MaxModes` — Maximum number of modes

`50` (default) | positive integer

Maximum number of modes, specified as the comma-separated pair consisting of `'MaxModes'` and a positive integer.

Data Types: `single` | `double`

`SCriteria` — Criteria to define consecutive stable natural frequencies and damping ratios

`[0.01 0.05]` (default) | two-element vector of positive values

Criteria to define stable natural frequencies and damping ratios between consecutive model degrees of freedom, specified as the comma-separated pair consisting of `'SCriteria'` and a two-element vector of positive values. `'SCriteria'` contains the maximum fractional differences between poles to be classified as stable. The first element of the vector applies to natural frequencies. The second element applies to damping ratios.

Data Types: `single` | `double`

Output Arguments

`fn` — Natural frequencies identified as stable

matrix

Natural frequencies identified as stable, returned as a matrix. The first *i* elements of the *i*th row contain natural frequencies. Poles that are nonphysical or not stable in frequency are returned as `NaN`s.

References

[1] Brandt, Anders. *Noise and Vibration Analysis: Signal Analysis and Experimental Procedures*. Chichester, UK: John Wiley & Sons, 2011.

[2] Ozdemir, Ahmet Arda, and Suat Gumussoy. "Transfer Function Estimation in System Identification Toolbox via Vector Fitting." *Proceedings of the 20th World Congress of the International Federation of Automatic Control*, Toulouse, France, July 2017.

[3] Vold, Håvard, John Crowley, and G. Thomas Rocklin. “New Ways of Estimating Frequency Response Functions.” *Sound and Vibration*. Vol. 18, November 1984, pp. 34–38.

See Also

`modalfit` | `modalfrf` | `tfest`

Introduced in R2017a

modulate

Modulation for communications simulation

Syntax

```
y = modulate(x,fc,fs)
[y,t] = modulate(x,fc,fs)
[___] = modulate(x,fc,fs,method)
[___] = modulate(x,fc,fs,method,opt)
```

Description

`y = modulate(x,fc,fs)` modulates the real message signal `x` with a carrier frequency `fc` and sample rate `fs`. If `x` is a matrix, the modulated signal is computed independently for each column and stored in the corresponding column of `y`.

`[y,t] = modulate(x,fc,fs)` also returns the internal time vector `t`.

`[___] = modulate(x,fc,fs,method)` modulates the real message signal using the modulation technique specified by `method`. You can use these inputs with either of the previous output syntaxes.

`[___] = modulate(x,fc,fs,method,opt)` uses the additional options specified in `opt` for some modulation methods.

Examples

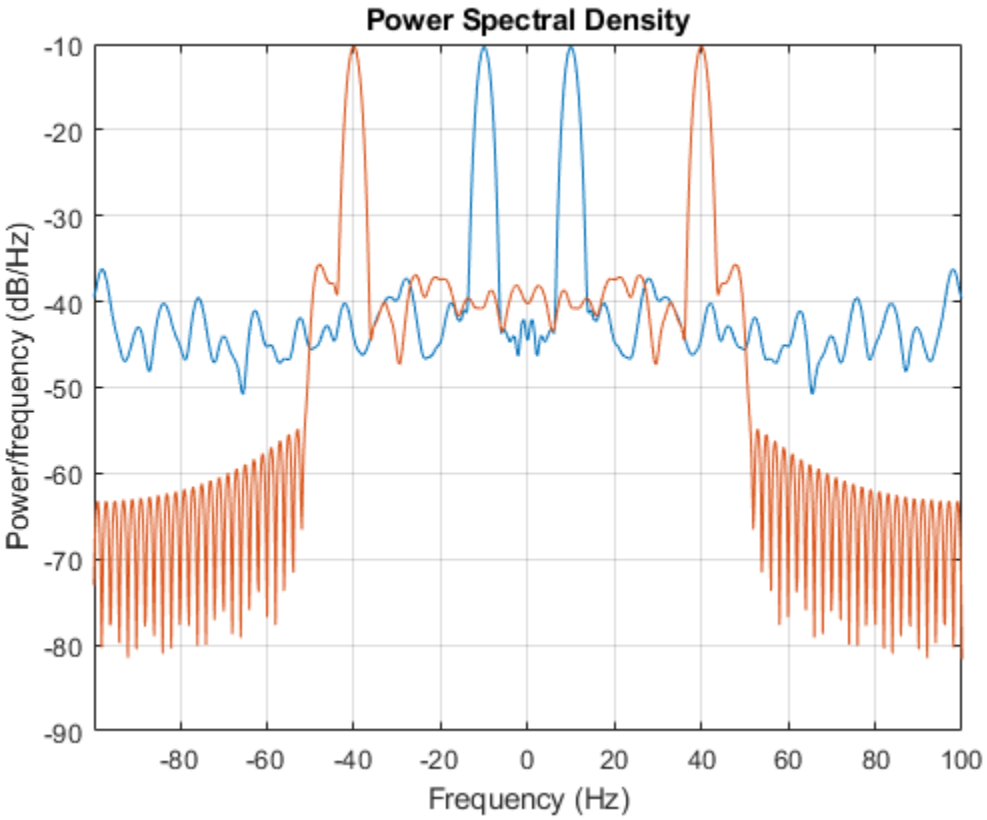
Single-Sideband Amplitude Modulation

Generate a 10 Hz sinusoidal signal sampled at a rate of 200 Hz for 1 second. Embed the sinusoid in white Gaussian noise of variance 0.01.

```
fs = 200;
t = 0:1/fs:1;
x = sin(2*pi*10*t) + randn(size(t))/10;
```

Single-sideband amplitude modulate the signal with a carrier frequency of 50 Hz. Compute and display the new Welch's power spectral density estimates.

```
y = modulate(x,50,fs,'amssb');

pwelch([x;y]',hamming(100),80,1024,fs,'centered')
```



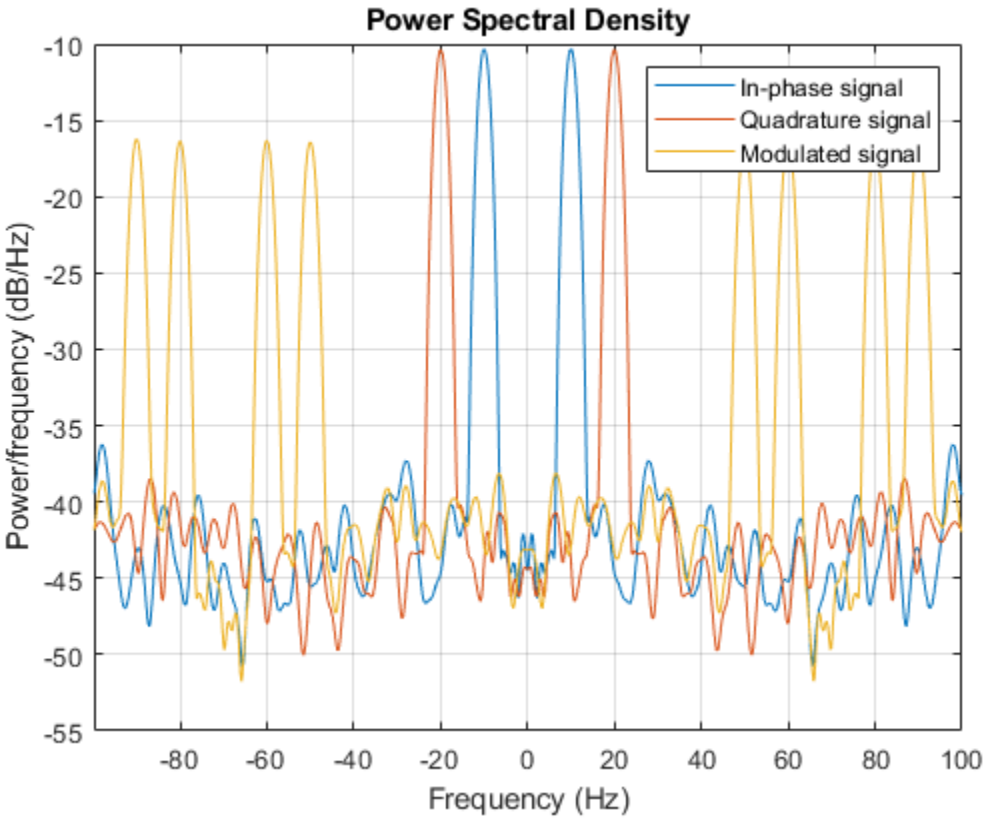
Quadrature Amplitude Modulation of Two Sinusoidal Signals

Generate two sinusoidal signals frequencies of 10 Hz and 20 Hz, sampled at a rate of 200 Hz for 1 second. Embed the sinusoids in white Gaussian noise of variance 0.01.

```
fs = 200;
t = 0:1/fs:1;
i = sin(2*pi*10*t) + randn(size(t))/10;
q = sin(2*pi*20*t) + randn(size(t))/10;
```

Create a quadrature amplitude modulated signal from signals `i` and `q` using a carrier frequency of 70 Hz. Compute the Welch power spectral density estimates of the original and modulated sequences. Use a 100-sample Hamming window with 80 samples of overlap. Specify an FFT length of 1024.

```
y = modulate(i,70,fs,'qam',q);

pwelch([i;q;y]',hamming(100),80,1024,fs,'centered')
legend('In-phase signal','Quadrature signal','Modulated signal')
```



Input Arguments

`x` — Message signal

real vector | real matrix

Message signal, specified as a real vector or matrix.

Example: `sin(2*pi*25*[0:(1/200):1])`

`fc` — Carrier frequency

real positive scalar

Carrier frequency used to modulate the message signal, specified as a real positive scalar.

`fs` — Sample rate

real positive scalar

Sample rate, specified as a real positive scalar.

`method` — Method of modulation used

`'am'` (default) | `'amdsb-tc'` | `'amssb'` | `'fm'` | `'pm'` | `'pwm'` | `'ppm'` | `'qam'`

Method of modulation used, specified as one of:

- `amdsb-sc` or `am` — Amplitude modulation, double sideband, suppressed carrier. Multiplies `x` by a sinusoid of frequency `fc`.

```
y = x.*cos(2*pi*fc*t)
```

- `amdsb-tc` — Amplitude modulation, double sideband, transmitted carrier. Subtracts scalar `opt` from `x` and multiplies the result by a sinusoid of frequency `fc`.

  ```
  y = (x-opt).*cos(2*pi*fc*t)
  ```

  If you do not specify the `opt` parameter, `modulate` uses a default of `min(min(x))` so that the message signal `(x-opt)` is entirely nonnegative and has a minimum value of 0.

- `amssb` — Amplitude modulation, single sideband. Multiplies `x` by a sinusoid of frequency `fc` and adds the result to the Hilbert transform of `x` multiplied by a phase-shifted sinusoid of frequency `fc`.

  ```
  y = x.*cos(2*pi*fc*t)+imag(hilbert(x)).*sin(2*pi*fc*t)
  ```

  This effectively removes the upper sideband.

- `fm` — Frequency modulation. Creates a sinusoid with instantaneous frequency that varies with the message signal `x`.

  ```
  y = cos(2*pi*fc*t + opt*cumsum(x))
  ```

  `cumsum` is a rectangular approximation of the integral of `x`. `modulate` uses `opt` as the constant of frequency modulation. If you do not specify the `opt` parameter, `modulate` uses a default of `opt = (fc/fs)*2*pi/(max(max(x)))` so the maximum frequency excursion from `fc` is `fc` Hz.

- `pm` — Phase modulation. Creates a sinusoid of frequency `fc` whose phase varies with the message signal `x`.

  ```
  y = cos(2*pi*fc*t + opt*x)
  ```

  `modulate` uses `opt` as the constant of phase modulation. If you do not specify the `opt` parameter, `modulate` uses a default of `opt = pi/(max(max(x)))` so the maximum phase excursion is π radians.

- `pwm` — Pulse-width modulation. Creates a pulse-width modulated signal from the pulse widths in `x`. The elements of `x` must be between 0 and 1, specifying the width of each pulse in fractions of a period. The pulses start at the beginning of each period, that is, they are left justified. `modulate(x,fc,fs,'pwm','centered')` yields pulses centered at the beginning of each period. The length of `y` is `length(x)*fs/fc`.

- `ppm` — Pulse-position modulation. Creates a pulse-position modulated signal from the pulse positions in `x`. The elements of `x` must be between 0 and 1, specifying the left edge of each pulse in fractions of a period. `opt` is a scalar between 0 and 1 that specifies the length of each pulse in fractions of a period. The default for `opt` is `0.1`. The length of `y` is `length(x)*fs/fc`.

- `qam`— Quadrature amplitude modulation. Creates a quadrature amplitude modulated signal from signals `x` and `opt`.

  ```
  y = x.*cos(2*pi*fc*t) + opt.*sin(2*pi*fc*t)
  ```

  The input argument `opt` must be the same size as `x`.

`opt` — Optional input for some methods
real vector

Optional input, specified for some methods. Refer to `method` for more details on how to use `opt`.

Output Arguments

`y` — Modulated signal

real vector | real matrix

Modulated message signal, returned as a real vector or matrix. Except for the methods `pwm` and `ppm`, `y` is the same size as `x`.

`t` — Internal time array

real vector

Internal time array used by `modulate` in its computations, specified as a real vector.

See Also

`demod` | `vco`

Introduced before R2006a

mscohere

Magnitude-squared coherence

Syntax

```
cxy = mscohere(x,y)

cxy = mscohere(x,y,window)
cxy = mscohere(x,y,window,noverlap)
cxy = mscohere(x,y,window,noverlap,nfft)

cxy = mscohere( ___ ,'mimo')

[cxy,w] = mscohere( ___ )
[cxy,f] = mscohere( ___ ,fs)

[cxy,w] = mscohere(x,y,window,noverlap,w)
[cxy,f] = mscohere(x,y,window,noverlap,f,fs)

[ ___ ] = mscohere(x,y, ___ ,freqrange)

mscohere( ___ )
```

Description

`cxy = mscohere(x,y)` finds the magnitude-squared coherence estimate, `cxy`, of the input signals, `x` and `y`.

- If `x` and `y` are both vectors, they must have the same length.
- If one of the signals is a matrix and the other is a vector, then the length of the vector must equal the number of rows in the matrix. The function expands the vector and returns a matrix of column-by-column magnitude-squared coherence estimates.
- If `x` and `y` are matrices with the same number of rows but different numbers of columns, then `mscohere` returns a multiple coherence matrix. The *m*th column of `cxy` contains an estimate of the degree of correlation between all the input signals and the *m*th output signal. See “Magnitude-Squared Coherence” on page 1-1470 for more information.
- If `x` and `y` are matrices of equal size, then `mscohere` operates column-wise: `cxy(:,n) = mscohere(x(:,n),y(:,n))`. To obtain a multiple coherence matrix, append `'mimo'` to the argument list.

`cxy = mscohere(x,y,window)` uses `window` to divide `x` and `y` into segments and perform windowing. You must use at least two segments. Otherwise, the magnitude-squared coherence is 1 for all frequencies. In the MIMO case, the number of segments must be greater than the number of input channels.

`cxy = mscohere(x,y,window,noverlap)` uses `noverlap` samples of overlap between adjoining segments.

`cxy = mscohere(x,y,window,noverlap,nfft)` uses `nfft` sampling points to calculate the discrete Fourier transform.

`cxy = mscohere( ___ ,'mimo')` computes a multiple coherence matrix for matrix inputs. This syntax can include any combination of input arguments from previous syntaxes.

`[cxy,w] = mscohere( ___ )` returns a vector of normalized frequencies, `w`, at which the magnitude-squared coherence is estimated.

`[cxy,f] = mscohere( ___ ,fs)` returns a vector of frequencies, `f`, expressed in terms of the sample rate, `fs`, at which the magnitude-squared coherence is estimated. `fs` must be the sixth numeric input to `mscohere`. To input a sample rate and still use the default values of the preceding optional arguments, specify these arguments as empty, `[]`.

`[cxy,w] = mscohere(x,y,window,noverlap,w)` returns the magnitude-squared coherence estimate at the normalized frequencies specified in `w`.

`[cxy,f] = mscohere(x,y,window,noverlap,f,fs)` returns the magnitude-squared coherence estimate at the frequencies specified in `f`.

`[ ___ ] = mscohere(x,y, ___ ,freqrange)` returns the magnitude-squared coherence estimate over the frequency range specified by `freqrange`. Valid options for `freqrange` are `'onesided'`, `'twosided'`, and `'centered'`.

`mscohere( ___ )` with no output arguments plots the magnitude-squared coherence estimate in the current figure window.

Examples

Coherence Estimate of Two Sequences

Compute and plot the coherence estimate between two colored noise sequences.

Generate a signal consisting of white Gaussian noise.

```
r = randn(16384,1);
```

To create the first sequence, bandpass filter the signal. Design a 16th-order filter that passes normalized frequencies between 0.2π and 0.4π rad/sample. Specify a stopband attenuation of 60 dB. Filter the original signal.

```
dx = designfilt('bandpassiir','FilterOrder',16, ...
    'StopbandFrequency1',0.2,'StopbandFrequency2',0.4, ...
    'StopbandAttenuation',60);
x = filter(dx,r);
```

To create the second sequence, design a 16th-order filter that stops normalized frequencies between 0.6π and 0.8π rad/sample. Specify a passband ripple of 0.1 dB. Filter the original signal.

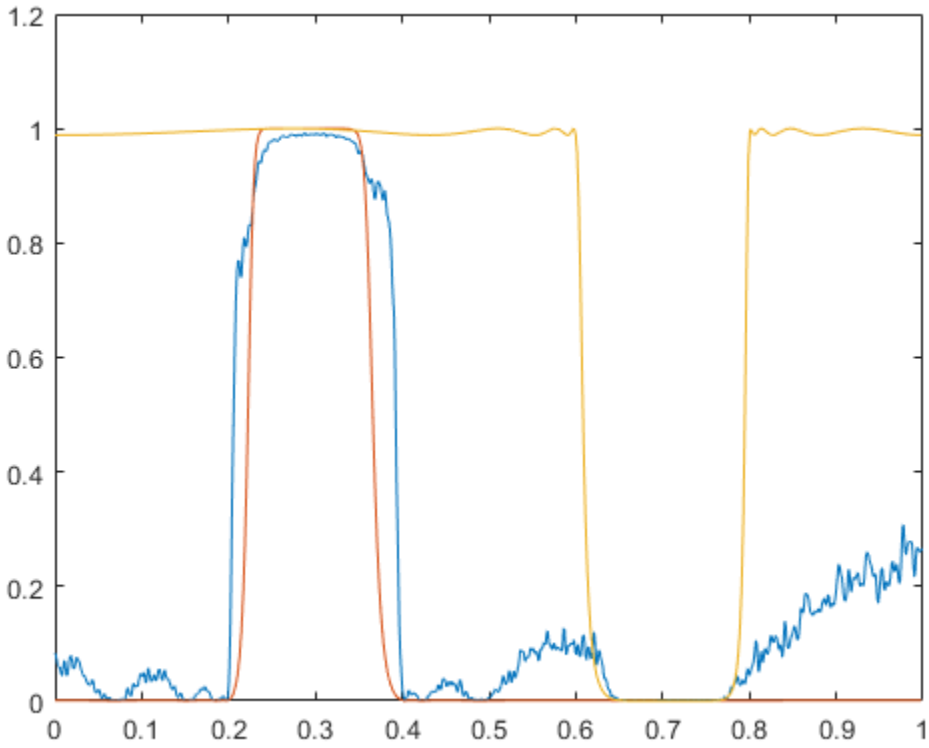
```
dy = designfilt('bandstopiir','FilterOrder',16, ...
    'PassbandFrequency1',0.6,'PassbandFrequency2',0.8, ...
    'PassbandRipple',0.1);
y = filter(dy,r);
```

Estimate the magnitude-squared coherence of `x` and `y`. Use a 512-sample Hamming window. Specify 500 samples of overlap between adjoining segments and 2048 DFT points.

```
[cxy,fc] = mscohere(x,y,hamming(512),500,2048);
```

Plot the coherence function and overlay the frequency responses of the filters.

```
[qx,f] = freqz(dx);
qy = freqz(dy);

plot(fc/pi,cxy)
hold on
plot(f/pi,abs(qx),f/pi,abs(qy))
hold off
```



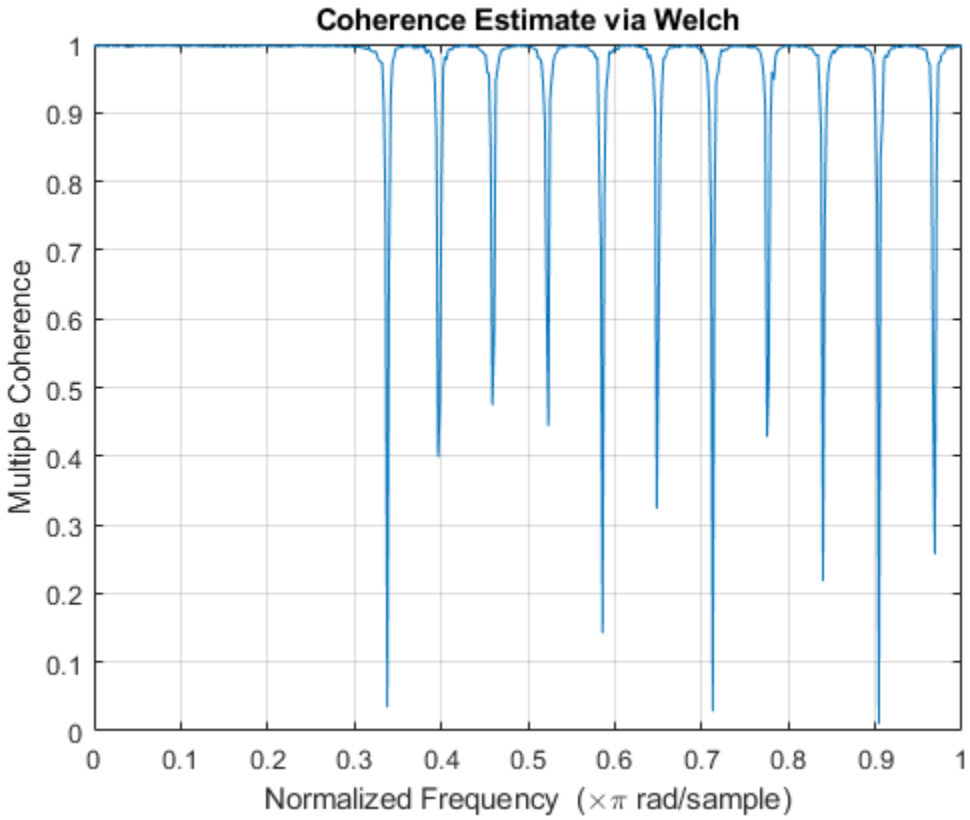
Multiple Coherence and Ordinary Coherence

Generate a random two-channel signal, `x`. Generate another signal, `y`, by lowpass filtering the two channels and adding them together. Specify a 30th-order FIR filter with a cutoff frequency of 0.3π and designed using a rectangular window.

```
h = fir1(30,0.3,rectwin(31));
x = randn(16384,2);
y = sum(filter(h,1,x),2);
```

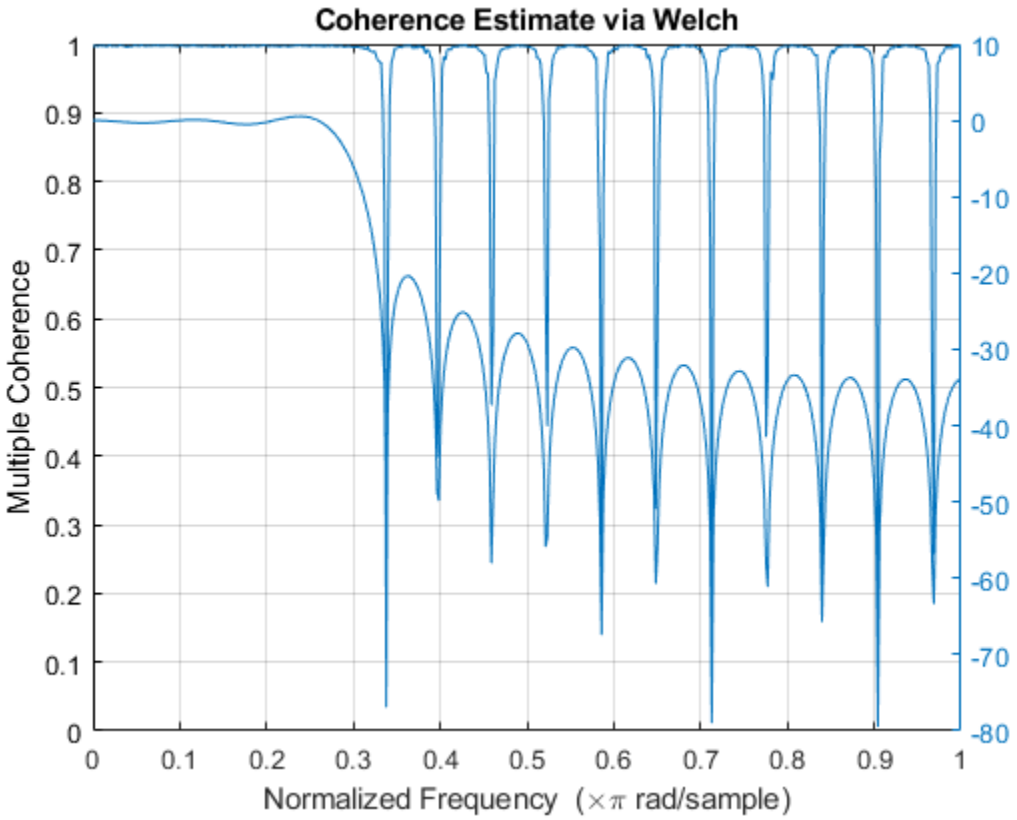
Compute the multiple-coherence estimate of `x` and `y`. Window the signals with a 1024-sample Hann window. Specify 512 samples of overlap between adjoining segments and 1024 DFT points. Plot the estimate.

```
noverlap = 512;
nfft = 1024;

mscohere(x,y,hann(nfft),noverlap,nfft,'mimo')
```



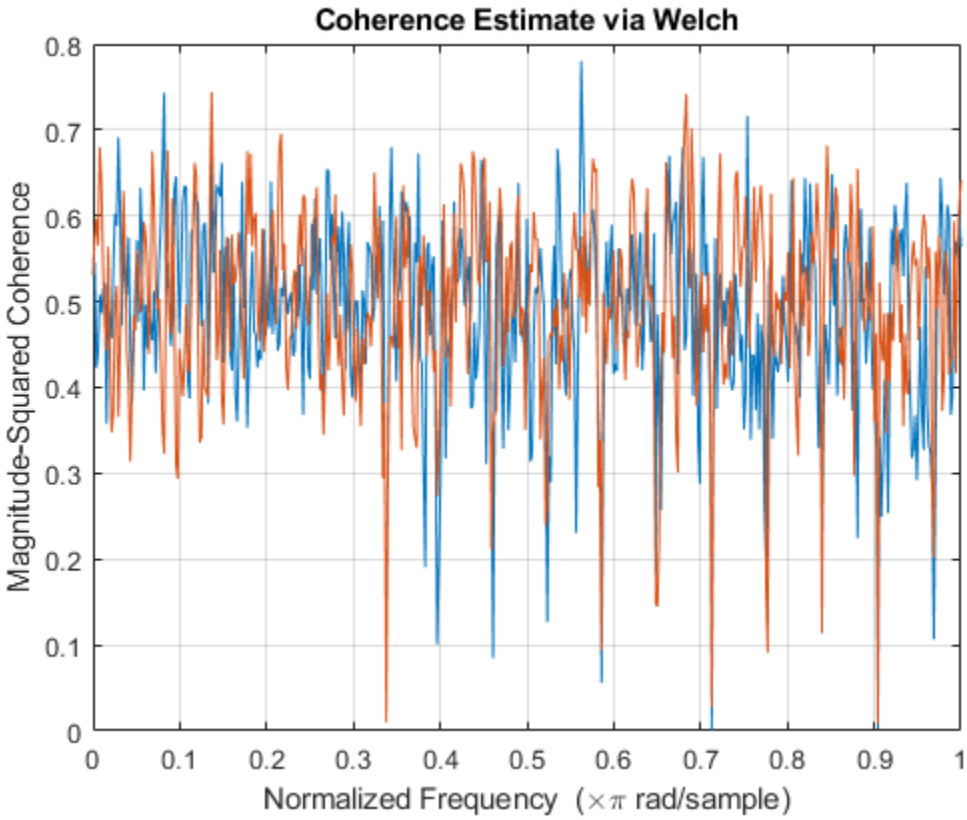
Compare the coherence estimate to the frequency response of the filter. The drops in coherence correspond to the zeros of the frequency response.

```
[H,f] = freqz(h);

hold on
yyaxis right
plot(f/pi,20*log10(abs(H)))
hold off
```



Compute and plot the ordinary magnitude-squared coherence estimate of `x` and `y`. The estimate does not reach 1 for any of the channels.

```
figure
mscohere(x,y,hann(nfft),noverlap,nfft)
```



Coherence of MIMO System

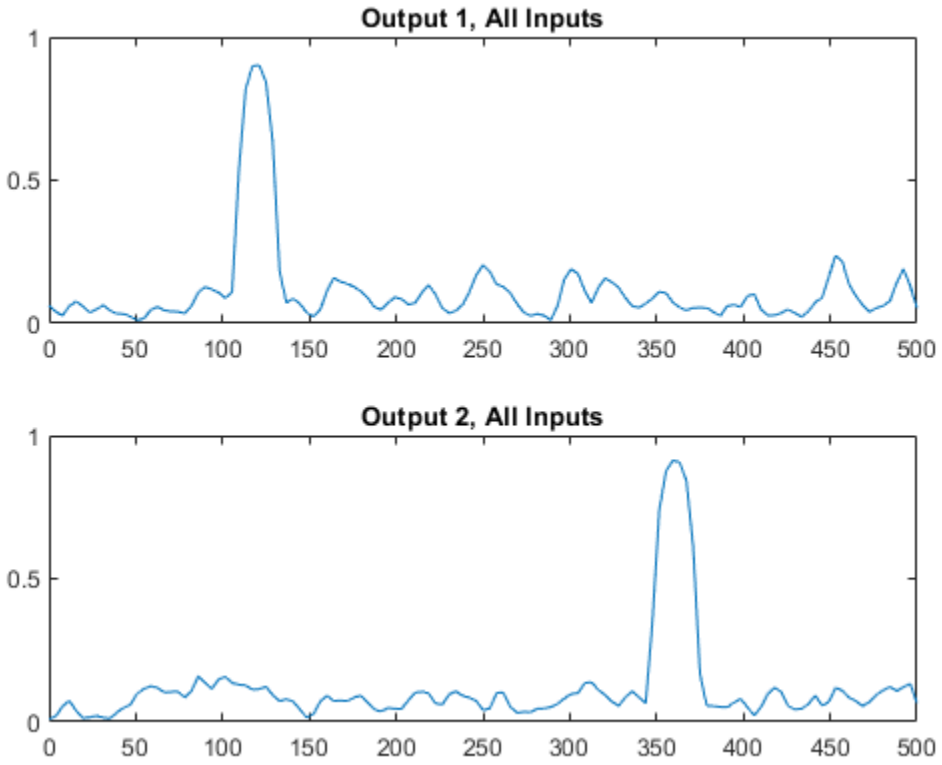
Generate two multichannel signals, each sampled at 1 kHz for 2 seconds. The first signal, the input, consists of three sinusoids with frequencies of 120 Hz, 360 Hz, and 480 Hz. The second signal, the output, is composed of two sinusoids with frequencies of 120 Hz and 360 Hz. One of the sinusoids lags the first signal by π/2. The other sinusoid has a lag of π/4. Both signals are embedded in white Gaussian noise.

```
fs = 1000;
f = 120;
t = (0:1/fs:2-1/fs)';

inpt = sin(2*pi*f*[1 3 4].*t);
inpt = inpt+randn(size(inpt));
oupt = sin(2*pi*f*[1 3].*t-[pi/2 pi/4]);
oupt = oupt+randn(size(oupt));
```

Estimate the degree of correlation between all the input signals and each of the output channels. Use a Hamming window of length 100 to window the data. `mscohere` returns one coherence function for each output channel. The coherence functions reach maxima at the frequencies shared by the input and the output.

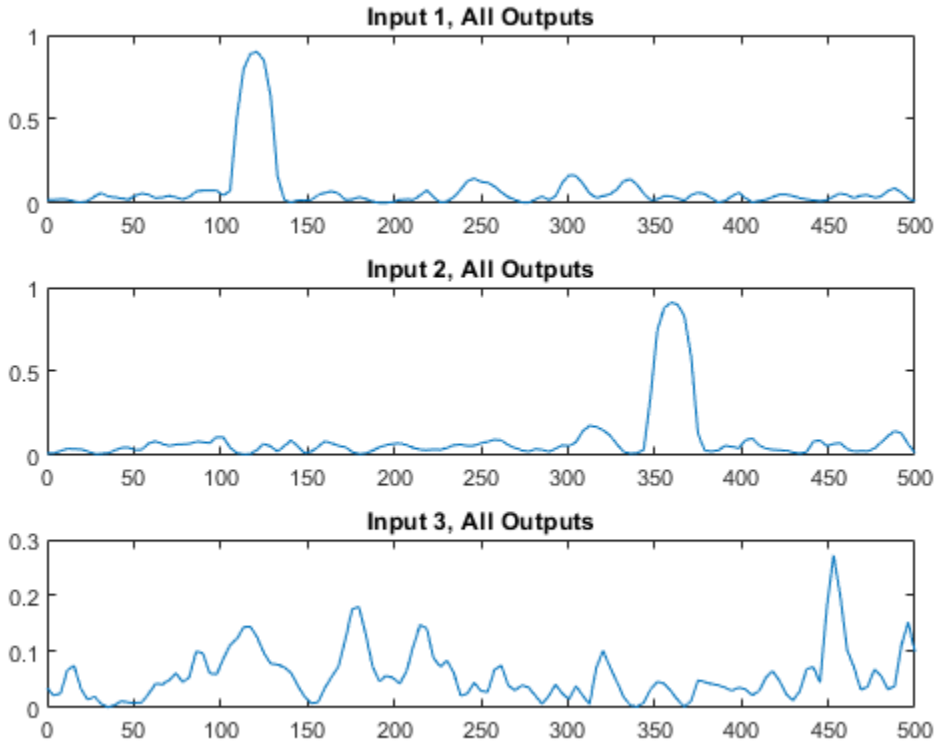
```
[Cxy,f] = mscohere(inpt,oupt,hamming(100),[],[],fs,'mimo');
```

```
for k = 1:size(oupt,2)
    subplot(size(oupt,2),1,k)
    plot(f,Cxy(:,k))
    title(['Output ' int2str(k) ', All Inputs'])
end
```



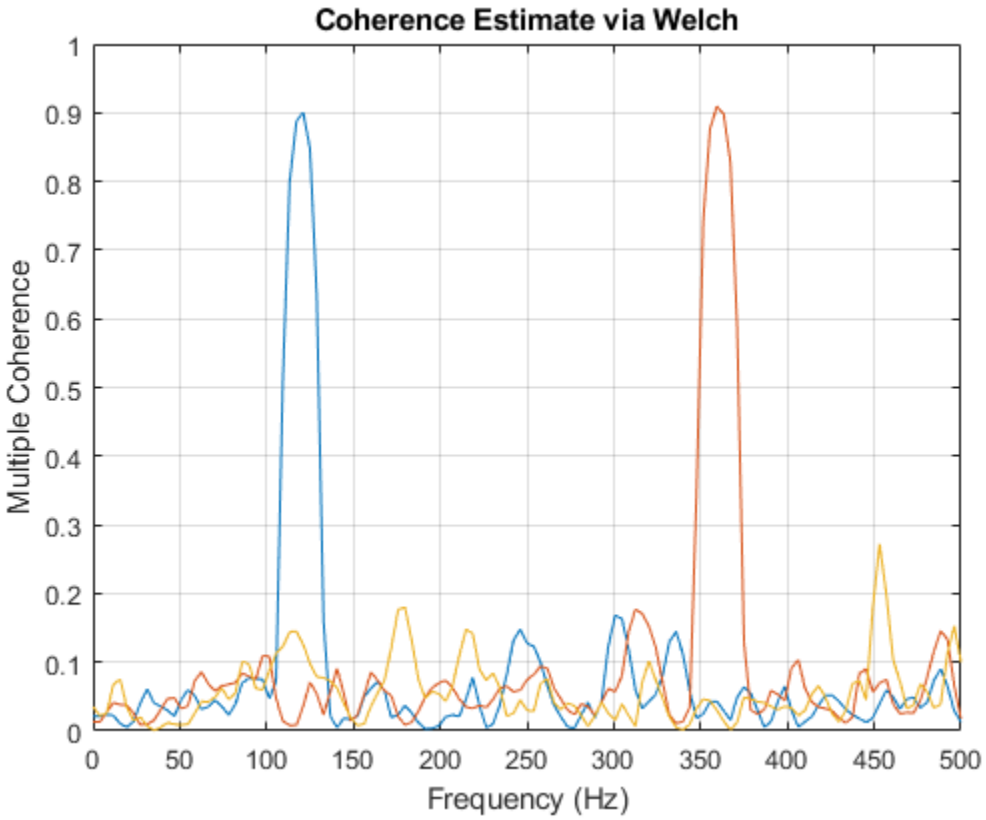
Switch the input and output signals and compute the multiple coherence function. Use the same Hamming window. There is no correlation between input and output at 480 Hz. Thus there are no peaks in the third correlation function.

```
[Cxy,f] = mscohere(oupt,inpt,hamming(100),[],[],fs,'mimo');

for k = 1:size(inpt,2)
    subplot(size(inpt,2),1,k)
    plot(f,Cxy(:,k))
    title(['Input ' int2str(k) ', All Outputs'])
end
```



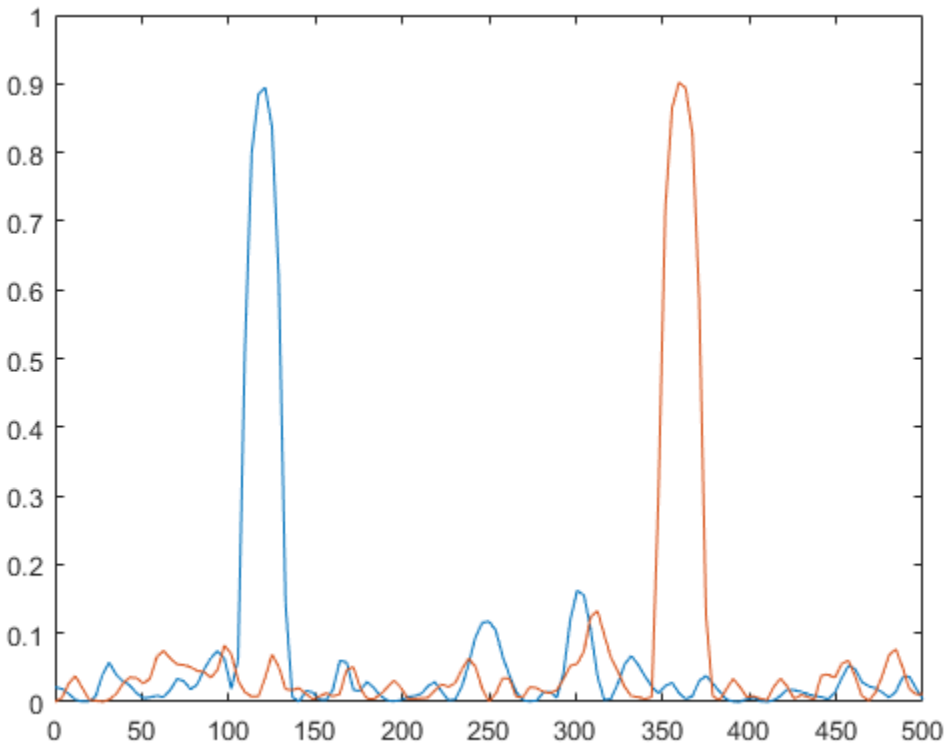
Repeat the computation, using the plotting functionality of `mscohere`.

```
clf
mscohere(oupt,inpt,hamming(100),[],[],fs,'mimo')
```



Compute the ordinary coherence function of the second signal and the first two channels of the first signal. The off-peak values differ from the multiple coherence function.

```
[Cxy,f] = mscohere(oupt,inpt(:,[1 2]),hamming(100),[],[],fs);
plot(f,Cxy)
```



Find the phase differences by computing the angle of the cross-spectrum at the points of maximum coherence.

```
Pxy = cpsd(oupt,inpt(:,[1 2]),hamming(100),[],[],fs);
[~,mxx] = max(Cxy);
for k = 1:2
    fprintf('Phase lag %d = %5.2f*pi\n',k,angle(Pxy(mxx(k),k))/pi)
end
```

```
Phase lag 1 = -0.51*pi
Phase lag 2 = -0.22*pi
```

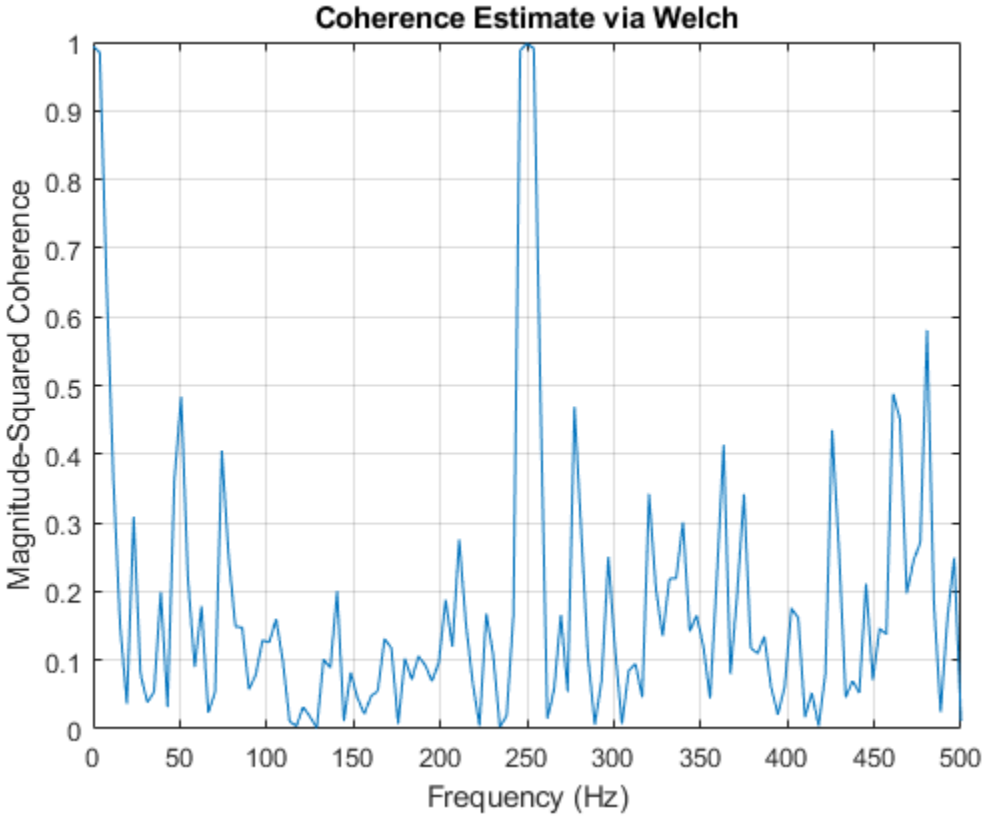
Modify Magnitude-Squared Coherence Plot

Generate two sinusoidal signals sampled for 1 second each at 1 kHz. Each sinusoid has a frequency of 250 Hz. One of the signals lags the other in phase by $\pi/3$ radians. Embed both signals in white Gaussian noise of unit variance.

```
fs = 1000;
f = 250;
t = 0:1/fs:1-1/fs;
um = sin(2*pi*f*t)+rand(size(t));
un = sin(2*pi*f*t-pi/3)+rand(size(t));
```

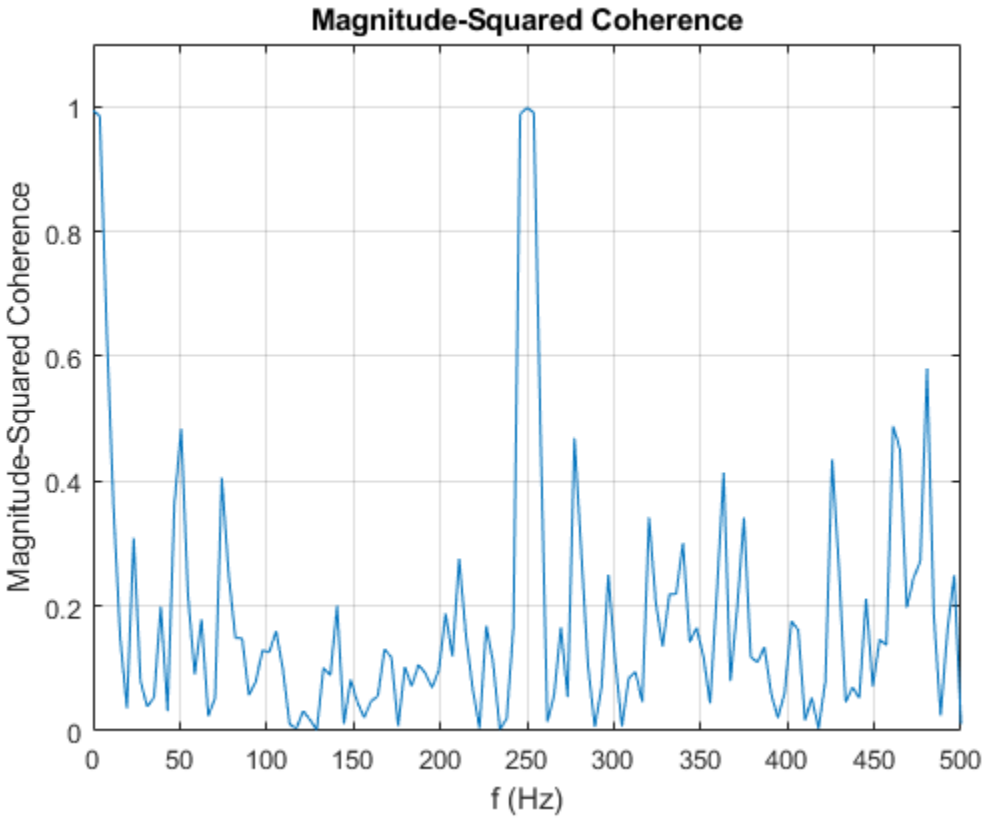
Use `mscohere` to compute and plot the magnitude-squared coherence of the signals.

```
mscohere(um,un,[],[],[],fs)
```



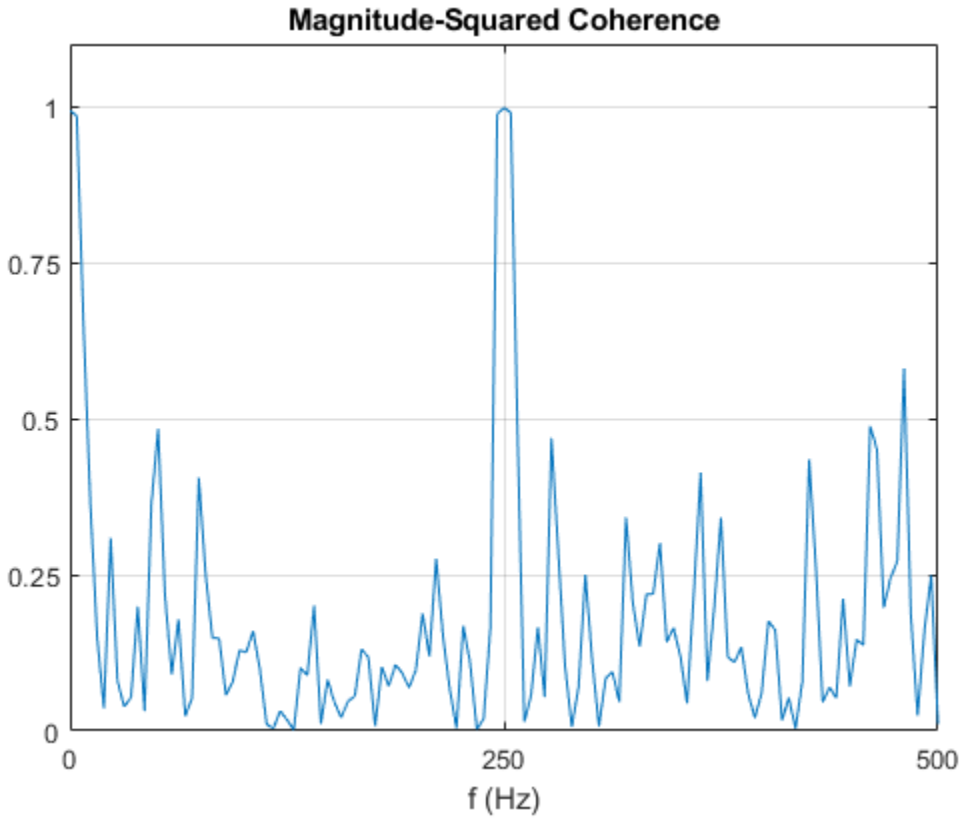
Modify the title of the plot, the label of the *x*-axis, and the limits of the *y*-axis.

```
title('Magnitude-Squared Coherence')
xlabel('f (Hz)')
ylim([0 1.1])
```



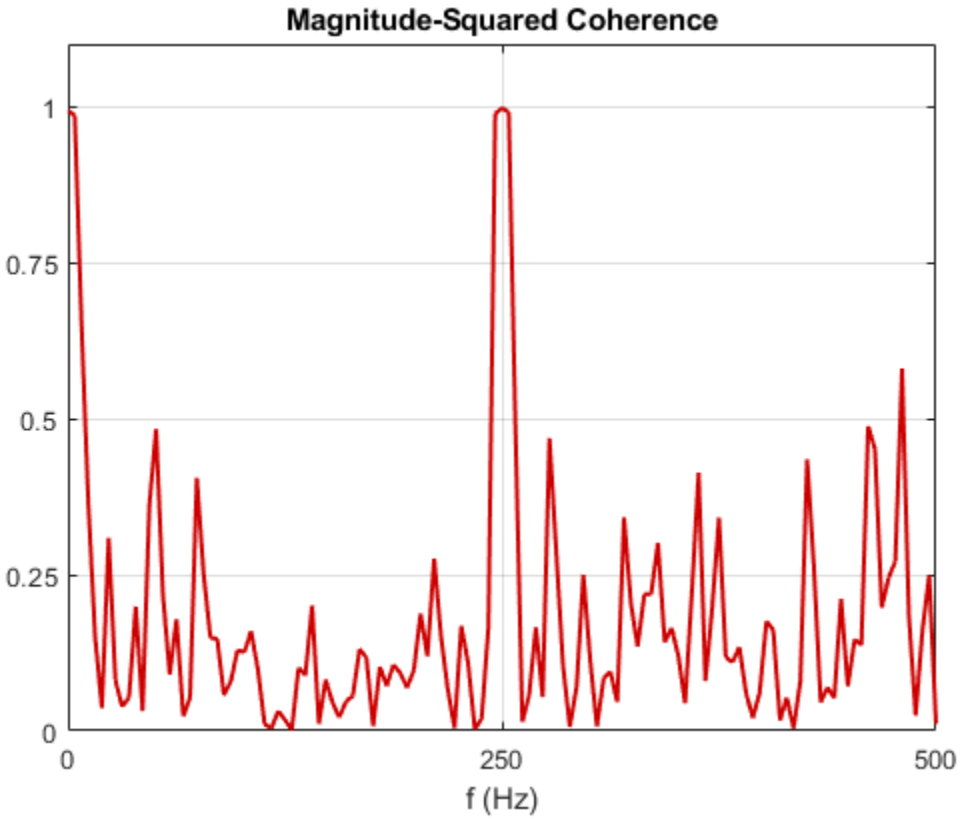
Use `gca` to obtain a handle to the current axes. Change the locations of the tick marks. Remove the label of the *y*-axis.

```
ax = gca;
ax.XTick = 0:250:500;
ax.YTick = 0:0.25:1;
ax.YLabel.String = [];
```



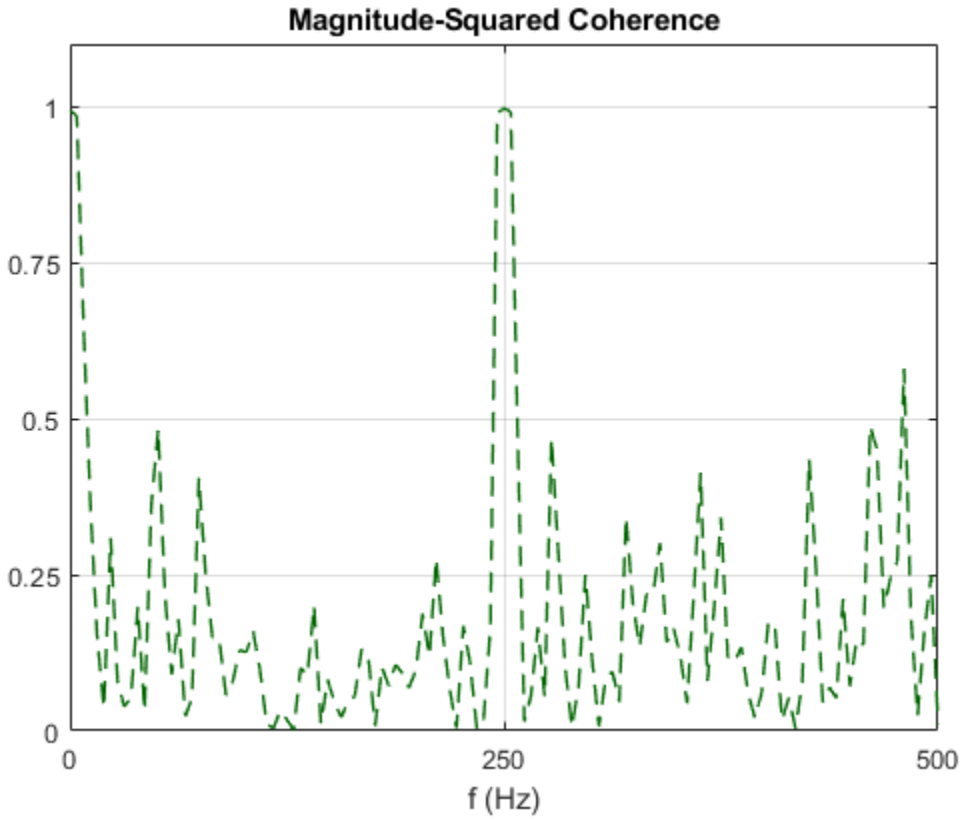
Call the `Children` property of the handle to change the color and width of the plotted line.

```
ln = ax.Children;
ln.Color = [0.8 0 0];
ln.LineWidth = 1.5;
```



Alternatively, use `set` and `get` to modify the line properties.

```
set(get(gca,'Children'),'Color',[0 0.4 0],'LineStyle','--','LineWidth',1)
```



Input Arguments

x, y — Input signals

vectors | matrices

Input signals, specified as vectors or matrices.

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 segments:

- If `window` is an integer, then `mscohere` divides `x` and `y` into segments of length `window` and windows each segment with a Hamming window of that length.
- If `window` is a vector, then `mscohere` divides `x` and `y` into segments of the same length as the vector and windows each segment using `window`.

If the length of `x` and `y` cannot be divided exactly into an integer number of segments with `noverlap` overlapping samples, then the signals are truncated accordingly.

If you specify `window` as empty, then `mscohere` uses a Hamming window such that `x` and `y` are divided into eight segments 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 `mscohere` uses a number that produces 50% overlap between segments. If the segment length is unspecified, the function sets `noverlap` to $\lfloor N/4.5 \rfloor$, where N is the length of the input and output signals.

Data Types: `double` | `single`

`nfft` — Number of DFT points
positive integer | `[]`

Number of DFT points, specified as a positive integer. If you specify `nfft` as empty, then `mscohere` sets this argument to $\max(256,2^p)$, where $p = \lceil \log_2 N \rceil$ for input signals of length N.

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 has units of Hz.

`w` — Normalized frequencies
vector

Normalized frequencies, specified as a row or column vector with at least two elements. Normalized frequencies are in rad/sample.

Example: `w = [pi/4 pi/2]`

Data Types: `double`

`f` — Frequencies
vector

Frequencies, specified as a row or column vector with at least two elements. The frequencies are in cycles per unit time. The unit time is specified by the sample rate, `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 magnitude-squared coherence estimate
`'onesided'` | `'twosided'` | `'centered'`

Frequency range for the magnitude-squared coherence estimate, specified as `'onesided'`, `'twosided'`, or `'centered'`. The default is `'onesided'` for real-valued signals and `'twosided'` for complex-valued signals.

- `'onesided'` — Returns the one-sided estimate of the magnitude-squared coherence estimate between two real-valued input signals, `x` and `y`. If `nfft` is even, `cxy` has `nfft`/2 + 1 rows and is computed over the interval [0,π] rad/sample. If `nfft` is odd, `cxy` has (`nfft` + 1)/2 rows and the interval is [0,π) rad/sample. If you specify `fs`, the corresponding intervals are [0,`fs`/2] cycles/unit time for even `nfft` and [0,`fs`/2) cycles/unit time for odd `nfft`.
- `'twosided'` — Returns the two-sided estimate of the magnitude-squared coherence estimate between two real-valued or complex-valued input signals, `x` and `y`. In this case, `cxy` has `nfft` rows and is computed over the interval [0,2π) rad/sample. If you specify `fs`, the interval is [0,`fs`) cycles/unit time.
- `'centered'` — Returns the centered two-sided estimate of the magnitude-squared coherence estimate between two real-valued or complex-valued input signals, `x` and `y`. In this case, `cxy` has `nfft` rows and is computed over the interval (–π,π] rad/sample for even `nfft` and (–π,π) rad/sample for odd `nfft`. If you specify `fs`, the corresponding intervals are (–`fs`/2, `fs`/2] cycles/unit time for even `nfft` and (–`fs`/2, `fs`/2) cycles/unit time for odd `nfft`.

Output Arguments

`cxy` — Magnitude-squared coherence estimate
vector | matrix | three-dimensional array

Magnitude-squared coherence estimate, returned as a vector, matrix, or three-dimensional array.

`w` — Normalized frequencies
vector

Normalized frequencies, returned as a real-valued column vector.

`f` — Frequencies
vector

Frequencies, returned as a real-valued column vector.

More About

Magnitude-Squared Coherence

The magnitude-squared coherence estimate is a function of frequency with values between 0 and 1. These values indicate 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 $P_{yy}(f)$, and the cross power spectral density, $P_{xy}(f)$, of `x` and `y`:

$$C_{xy}(f) = \frac{\left|P_{xy}(f)\right|^2}{P_{xx}(f)P_{yy}(f)}.$$

For multi-input/multi-output systems, the multiple-coherence function becomes

$$C_{Xy_i}(f) = \frac{P^{\dagger}_{Xy_i}(f) P^{-1}_{XX}(f) P_{Xy_i}(f)}{P_{y_i y_i}(f)}$$

$$= \begin{bmatrix} P^{*}_{x_1 y_i}(f) & \cdots & P^{*}_{x_m y_i}(f) \end{bmatrix} \begin{bmatrix} P_{x_1 x_1}(f) & P_{x_1 x_2}(f) & \cdots & P_{x_1 x_m}(f) \\ P_{x_2 x_1}(f) & P_{x_2 x_2}(f) & \cdots & P_{x_2 x_m}(f) \\ \vdots & \vdots & \ddots & \vdots \\ P_{x_m x_1}(f) & P_{x_m x_2}(f) & \cdots & P_{x_m x_m}(f) \end{bmatrix}^{-1} \begin{bmatrix} P_{x_1 y_i}(f) \\ \vdots \\ P_{x_m y_i}(f) \end{bmatrix} \frac{1}{P_{y_i y_i}(f)}$$

for the ith output signal, where:

- X corresponds to the array of m inputs.
- P_{Xy_i} is the m-dimensional vector of cross power spectral densities between the inputs and y_i.
- P_{XX} is the m-by-m matrix of power spectral densities and cross power spectral densities of the inputs.
- $P_{y_i y_i}$ is the power spectral density of the output.
- The dagger (†) stands for the complex conjugate transpose.

Algorithms

`mscohere` estimates the magnitude-squared coherence function [2] using Welch's overlapped averaged periodogram method [3], [5].

References

[1] Gómez González, A., J. Rodríguez, X. Sagartzazu, A. Schumacher, and I. Isasa. "Multiple Coherence Method in Time Domain for the Analysis of the Transmission Paths of Noise and Vibrations with Non-Stationary Signals." *Proceedings of the 2010 International Conference of Noise and Vibration Engineering, ISMA2010-USD2010*. pp. 3927–3941.

[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] Stoica, Petre, and Randolph Moses. *Spectral Analysis of Signals.* Upper Saddle River, NJ: Prentice Hall, 2005.

[5] 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`cpsd` | `periodogram` | `pwelch` | `tfestimate`

Topics

“Cross Spectrum and Magnitude-Squared Coherence”
“Compare the Frequency Content of Two Signals”

Introduced before R2006a

nuttallwin

Nuttall-defined minimum 4-term Blackman-Harris window

Syntax

```
w = nuttallwin(L)
w = nuttallwin(L,sflag)
```

Description

`w = nuttallwin(L)` returns a Nuttall defined `L`-point, four-term symmetric Blackman-Harris window. The coefficients for this window differ from the Blackman-Harris window coefficients computed with `blackmanharris` and produce slightly lower sidelobes.

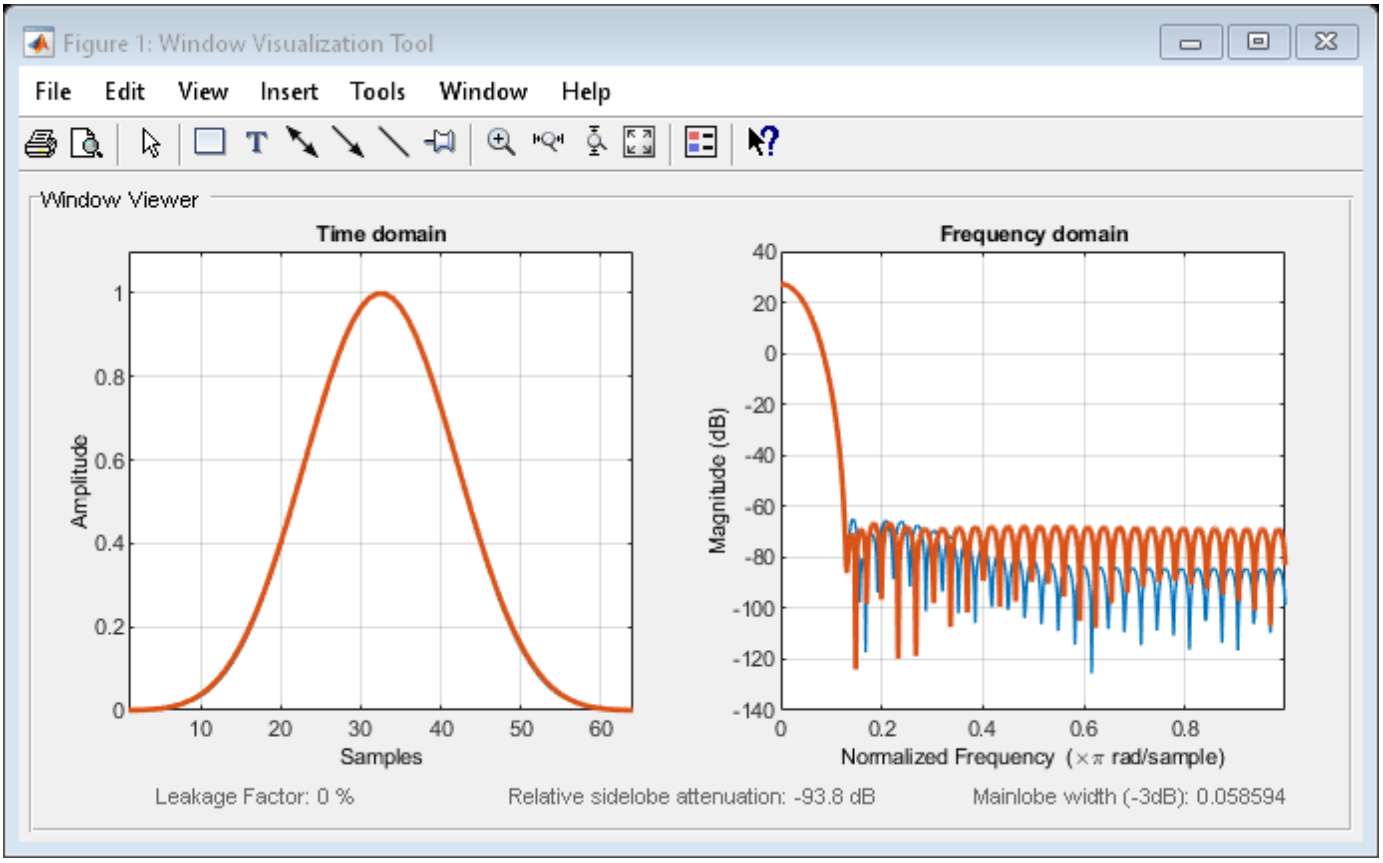
`w = nuttallwin(L,sflag)` uses `sflag` window sampling.

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
```

Input Arguments

L — Window length

real positive scalar

Window length, specified as a real positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sflag — Window sampling

"symmetric" (default) | "periodic"

Window sampling, specified as `"symmetric"` or `"periodic"`. For the equations that define the symmetric and periodic windows, see “Algorithms” on page 1-1475.

Output Arguments

w — Nuttall defined Blackman-Harris window

column vector

Nuttall defined Blackman-Harris window, returned as a column vector of length L. The function minimizes the maximum sidelobes of the window.

Algorithms

The equation for the *symmetric* Nuttall defined four-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, ... N-1.

The equation for the *periodic* Nuttall defined four-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, ... 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 = 0.0106411

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps

Window Designer

Functions

`barthannwin` | `bartlett` | `blackmanharris` | `bohmanwin` | `parzenwin` | `rectwin` | `triang` | **WVTool**

Introduced before R2006a

obw

Occupied bandwidth

Syntax

```
bw = obw(x)
bw = obw(x,fs)

bw = obw(pxx,f)
bw = obw(sxx,f,rbw)

bw = obw( ___ ,freqlims,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( ___ ,freqlims,p)` specifies the frequency interval over which to compute the occupied bandwidth. This syntax can include any combination of input arguments from previous syntaxes, as long as the second input argument is either `fs` or `f`. If the second input is passed as empty, normalized frequency will be assumed. 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

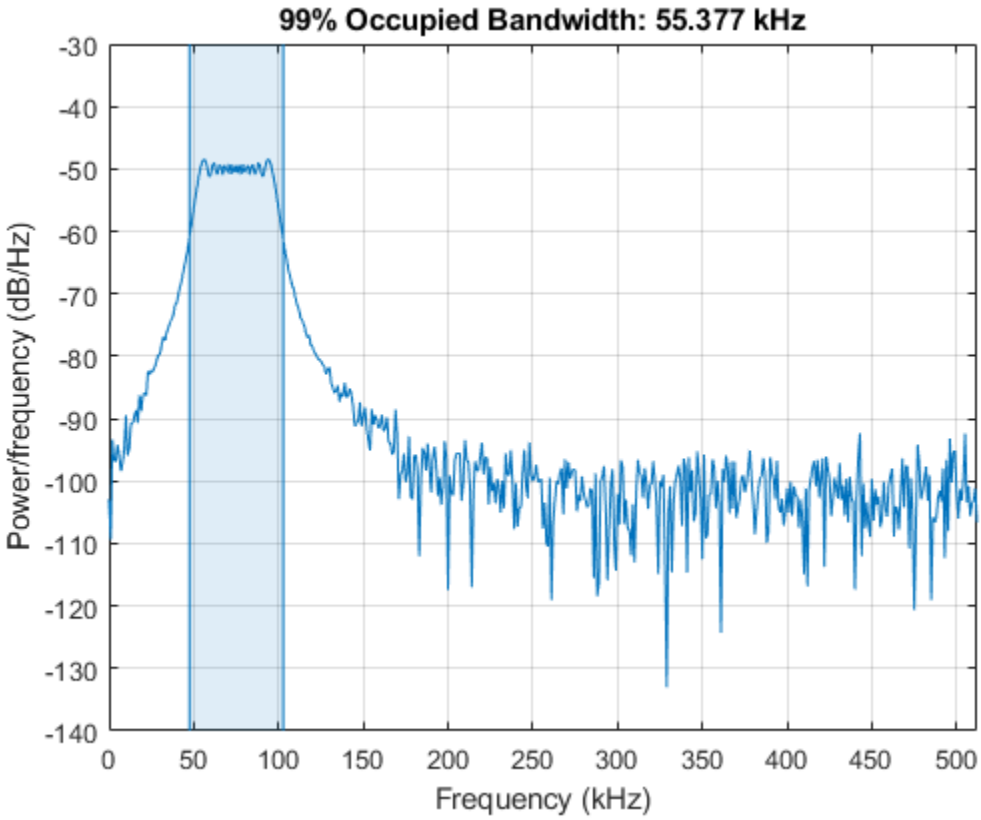
Occupied Bandwidth of Chirps

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);
```

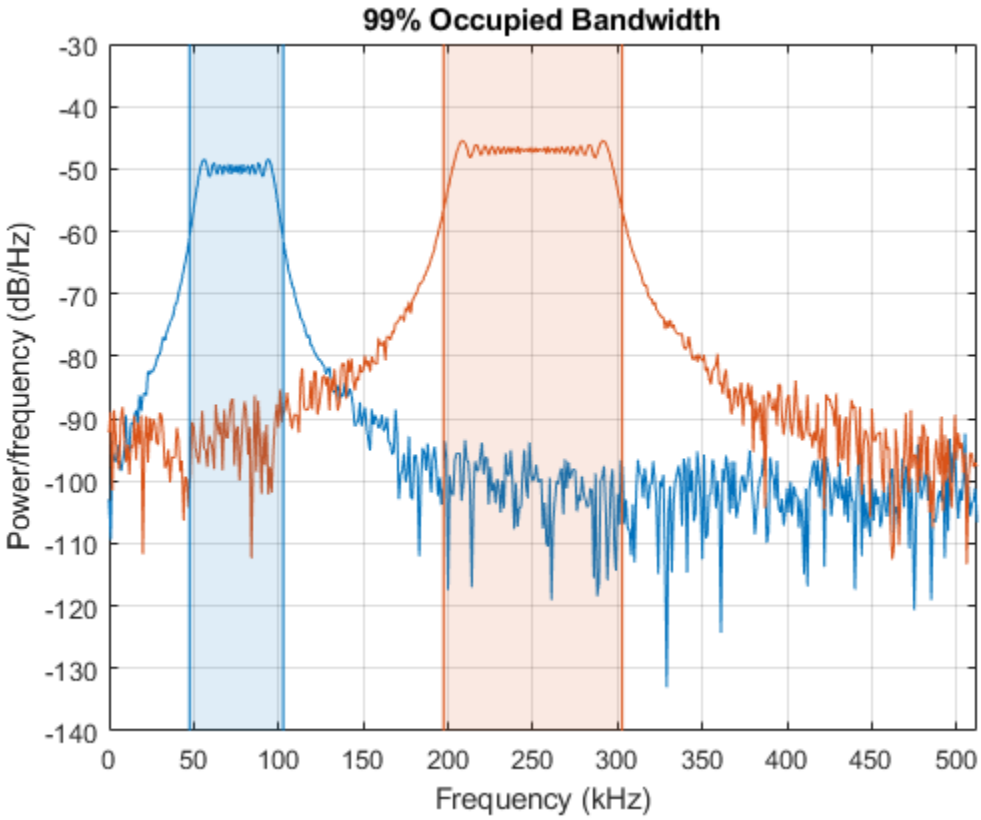
Concatenate the chirps to produce a two-channel signal. Estimate the occupied bandwidth of each channel.

```
y = obw([x x2],Fs)
```

```
y = 1×2
10^5 ×

    0.5538    1.0546
```

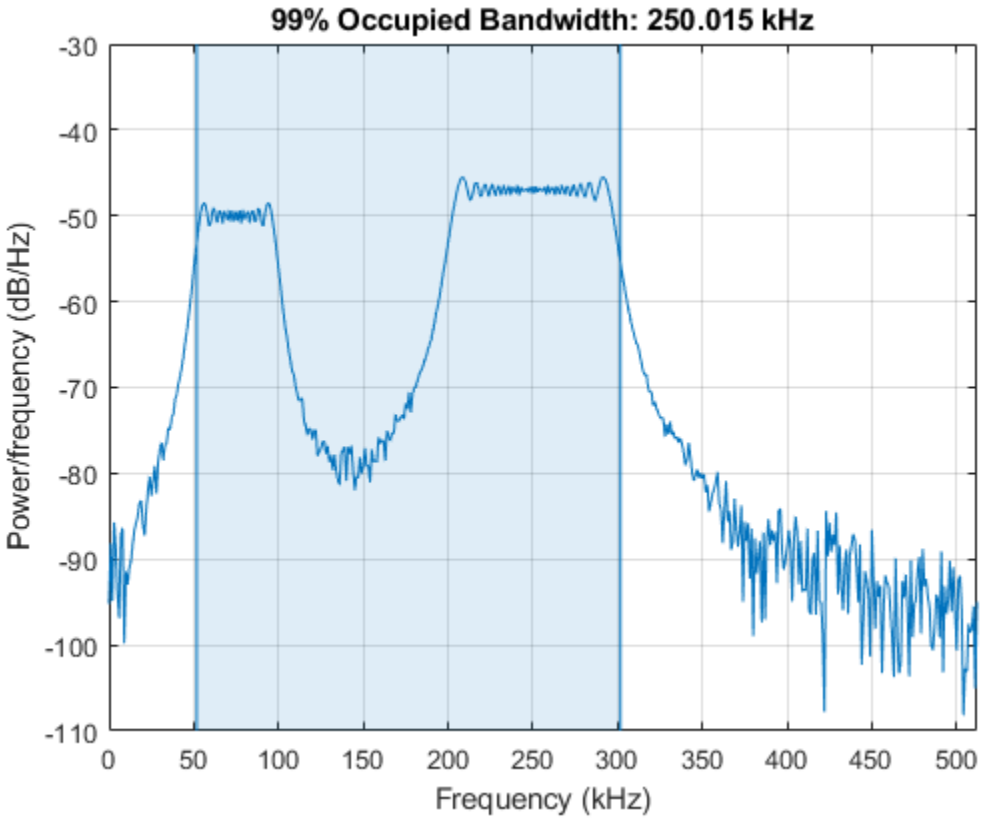
Annotate the occupied bandwidths of the two channels on a plot of the PSDs.

```
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);
```



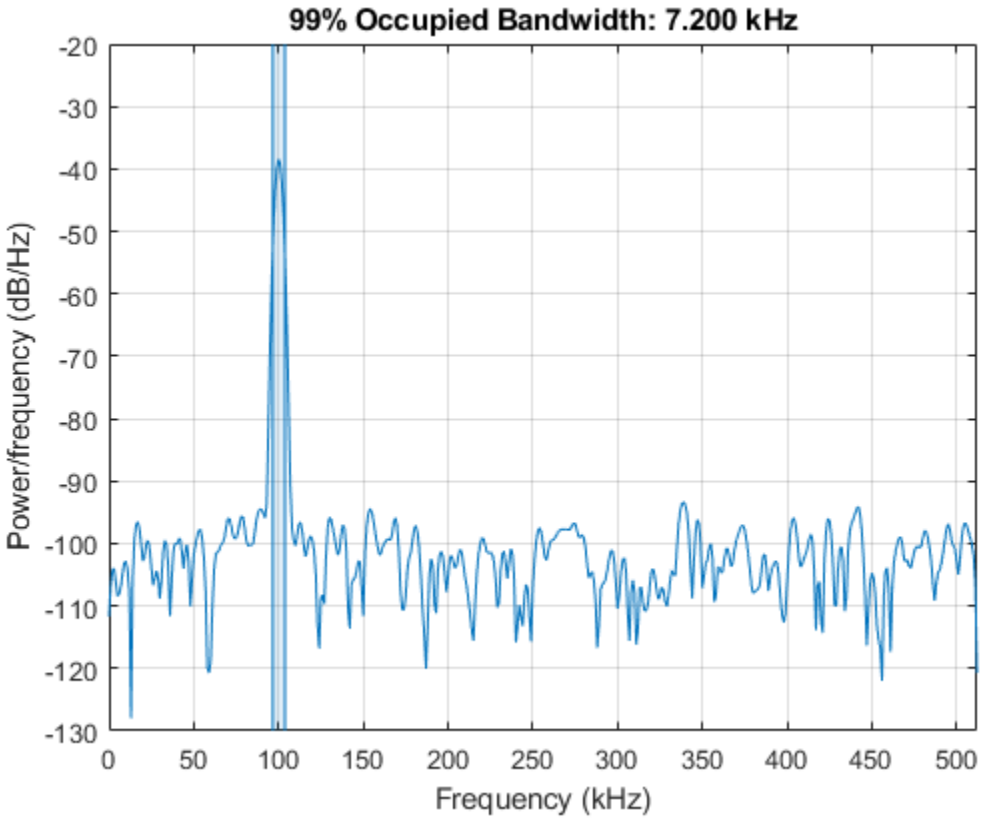
Occupied Bandwidth of Sinusoids

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.

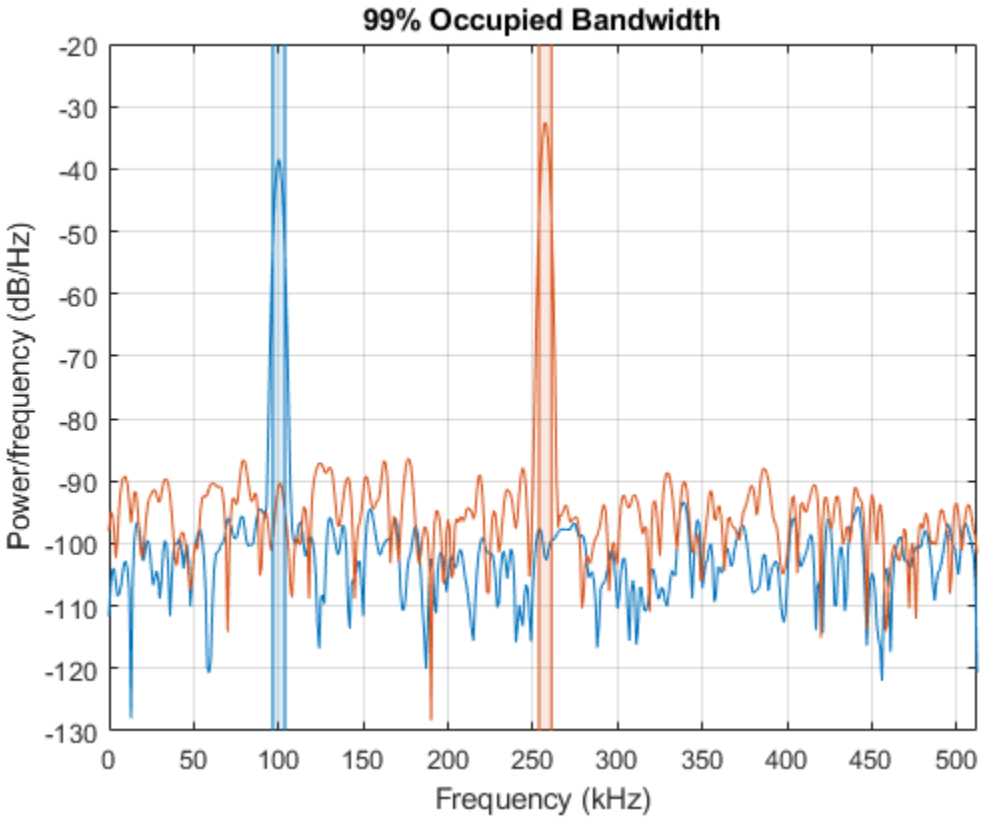
```
[Pyy,f] = periodogram([x x2],kaiser(nSamp,38),[],Fs);
```

```
y = obw(Pyy,f)
```

```
y = 1×2
10^3 ×

    7.2001    7.3777
```

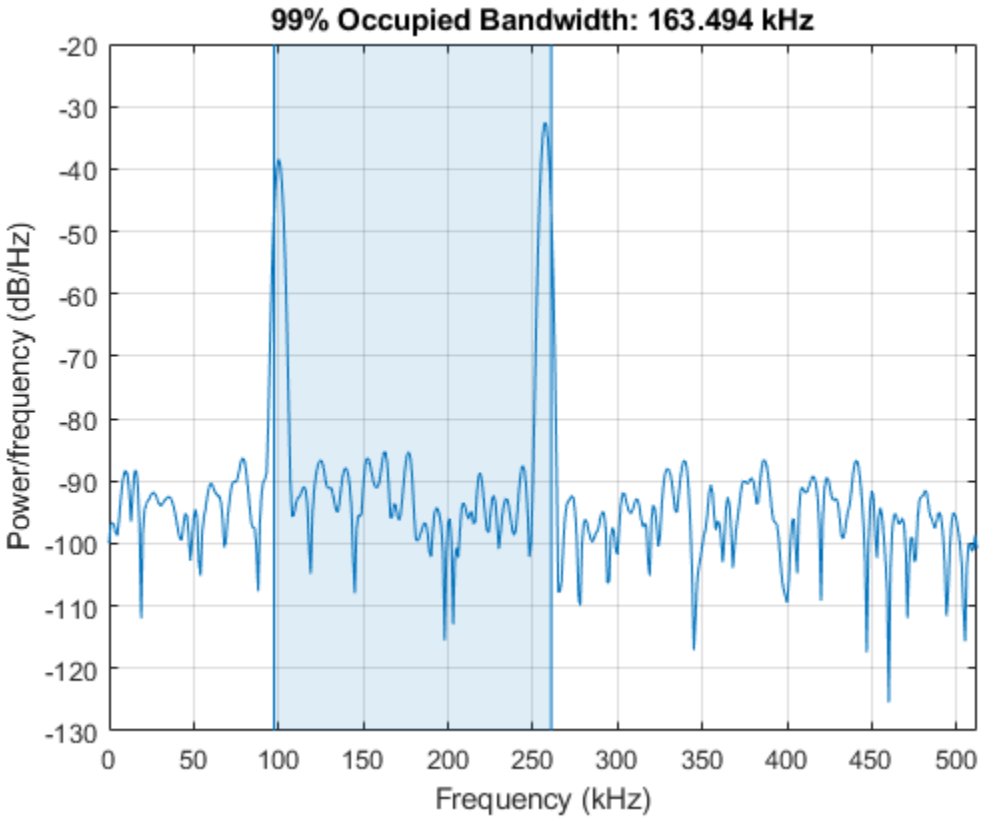
Annotate the occupied bandwidths of the two channels on a plot of the PSDs.

```
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);
```



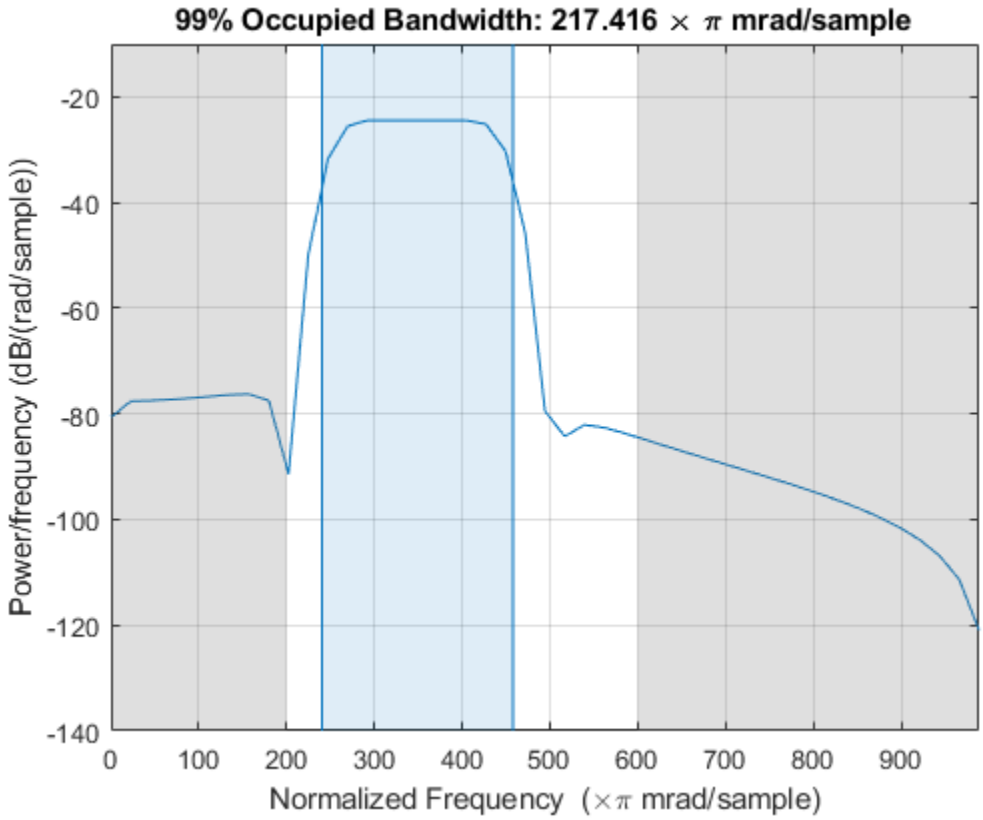
Occupied Bandwidth of Bandlimited Signals

Generate a signal whose PSD resembles the frequency response of an 88th-order bandpass FIR filter with normalized cutoff frequencies 0.25π rad/sample and 0.45π rad/sample.

```
d = fir1(88,[0.25 0.45]);
```

Compute the 99% occupied bandwidth of the signal between 0.2π rad/sample and 0.6π 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 2π 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)
```

```
bw = 0.217*pi, flo = 0.240*pi, fhi = 0.458*pi
```

```
fprintf('power = %.1f%% of total',power/bandpower(d)*100)
```

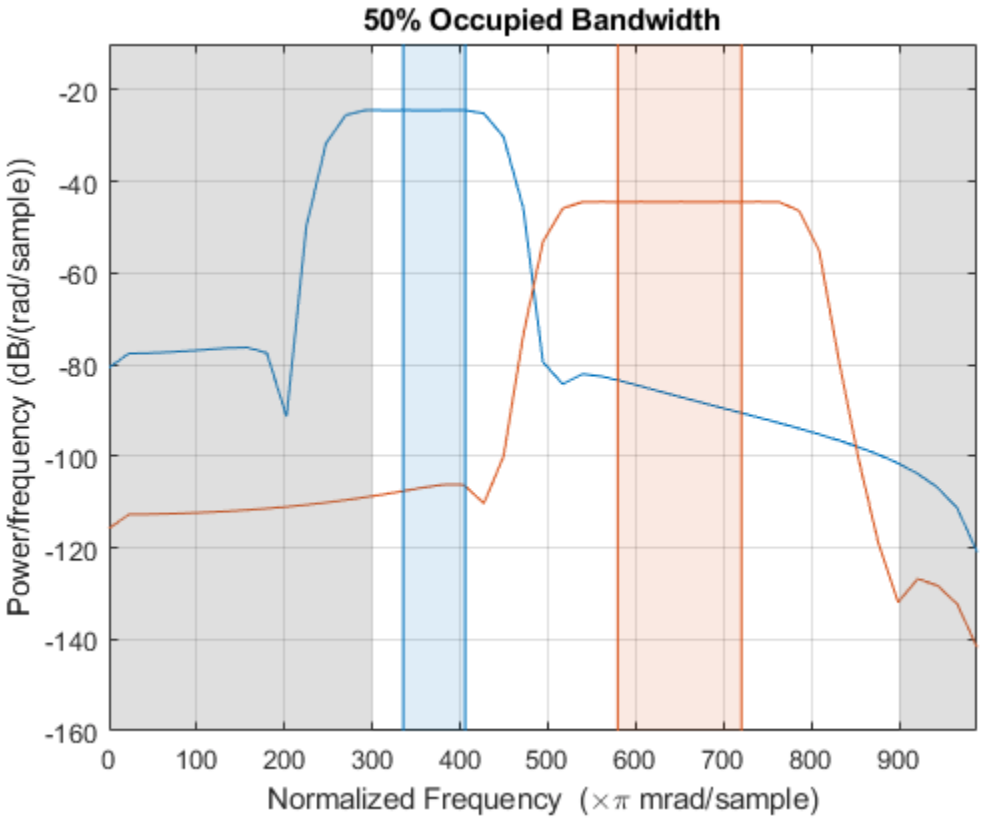
```
power = 99.0% of total
```

Add a second channel with normalized cutoff frequencies 0.5π rad/sample and 0.8π 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 0.3π rad/sample and 0.9π 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 π.

```
bw = obw(d,[],[0.3 0.9]*pi,50)/pi
```

```
bw = 1×2

    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 with real nonnegative elements. 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.

The power spectral density must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[pxx,f] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` specifies the periodogram PSD estimate of a noisy two-channel sinusoid sampled at 2π Hz and the frequencies at which it is computed.

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 with real nonnegative elements. If `sxx` is a matrix, then `obw` computes the occupied bandwidth of each column of `sxx` independently.

The power spectrum must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[sxx,w] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2),'power')` specifies the periodogram power spectrum estimate of a two-channel sinusoid embedded in white Gaussian noise and the normalized frequencies at which it is computed.

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`

`freqlims` — Frequency limits

two-element vector

Frequency limits, specified as a two-element vector of real values. If you do not specify `freqlims`, 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 ½(100 – `p`) and ½(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.

`flo`, `fhi` — 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.

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`bandpower` | `periodogram` | `powerbw` | `plomb` | `pwelch`

Introduced in R2015a

orderspectrum

Average spectrum versus order for vibration signal

Syntax

```
spec = orderspectrum(x,fs,rpm)
[spec,order] = orderspectrum(x,fs,rpm)

[spec,order] = orderspectrum(map,order)
[spec,order] = orderspectrum(map,order,'Amplitude',amp)

orderspectrum( ___ )
```

Description

`spec = orderspectrum(x,fs,rpm)` computes an average order-magnitude spectrum vector, `spec`, for an input signal, `x`, sampled at a rate of `fs` Hz. To compute the spectrum, `orderspectrum` windows a constant-phase, resampled version of `x` with a flat top window.

`[spec,order] = orderspectrum(x,fs,rpm)` also returns a vector of the orders corresponding to each average spectrum value.

`[spec,order] = orderspectrum(map,order)` computes an average order-magnitude spectrum vector starting from an order-RPM map and a vector of orders. Use `rpmordermap` to compute `map` and `order`. `map` must be linearly scaled. The returned amplitudes are the same as in `map`. The returned spectrum is scaled linearly.

`[spec,order] = orderspectrum(map,order,'Amplitude',amp)` specifies the type of amplitude to consider when computing an average order-magnitude spectrum starting from an order-RPM map.

`orderspectrum( ___ )` with no output arguments plots the RMS amplitude of the order spectrum, scaled linearly, on the current figure.

Examples

Average Order Spectrum of Chirp with Four Orders

Create a simulated signal sampled at 600 Hz for 5 seconds. The system that is being tested increases its rotational speed from 10 to 40 revolutions per second during the observation period.

Generate the tachometer readings.

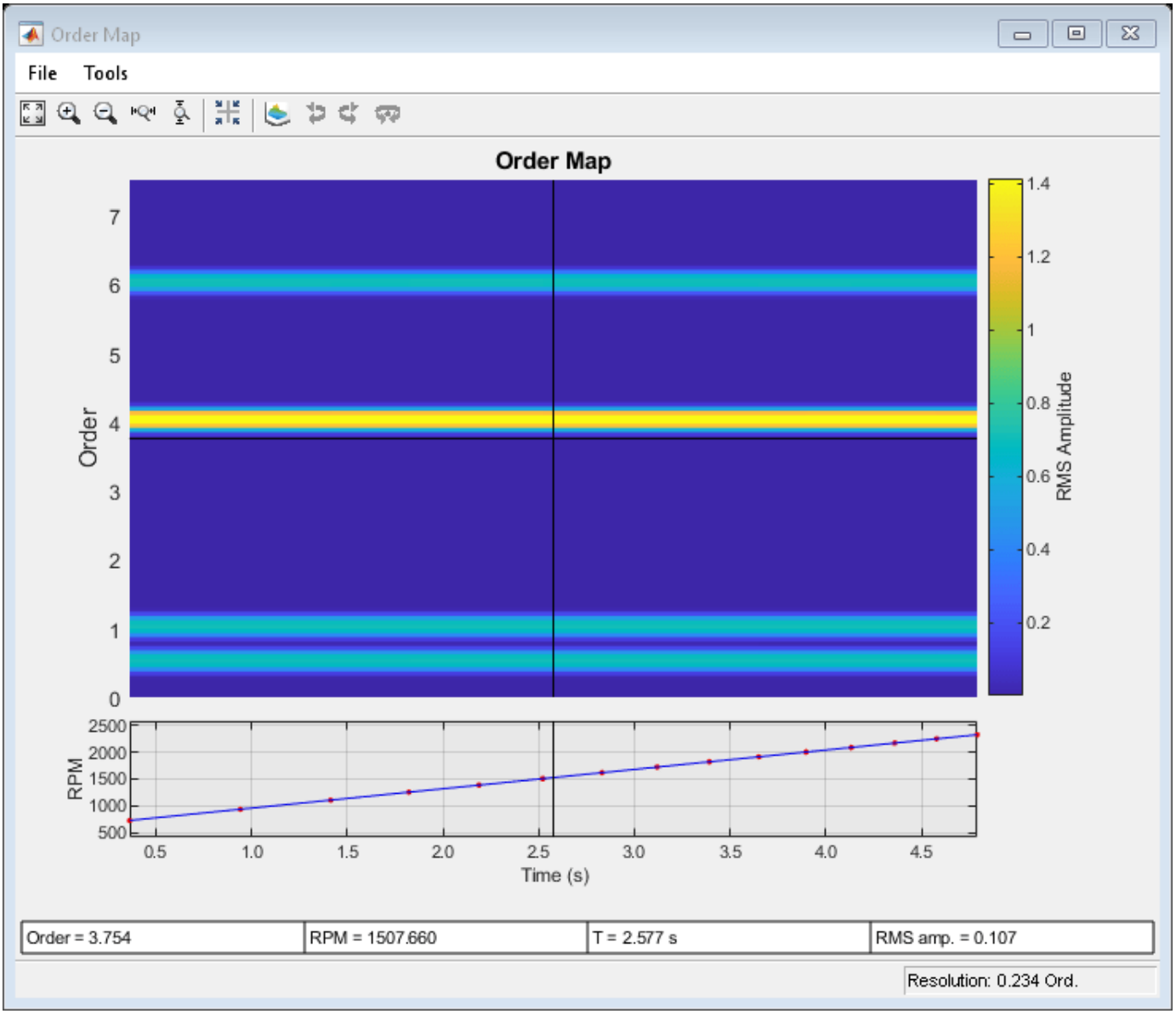
```
fs = 600;
t1 = 5;
t = 0:1/fs:t1;

f0 = 10;
f1 = 40;
rpm = 60*linspace(f0,f1,length(t));
```

The signal consists of four harmonically related chirps with orders 1, 0.5, 4, and 6. The order-4 chirp has twice the amplitude of the others. To generate the chirps, use the trapezoidal rule to express the phase as the integral of the rotational speed.

```
o1 = 1;
o2 = 0.5;
o3 = 4;
o4 = 6;

ph = 2*pi*cumtrapz(rpm/60)/fs;

x = [1 1 2 1]*cos([o1 o2 o3 o4]'*ph);
```

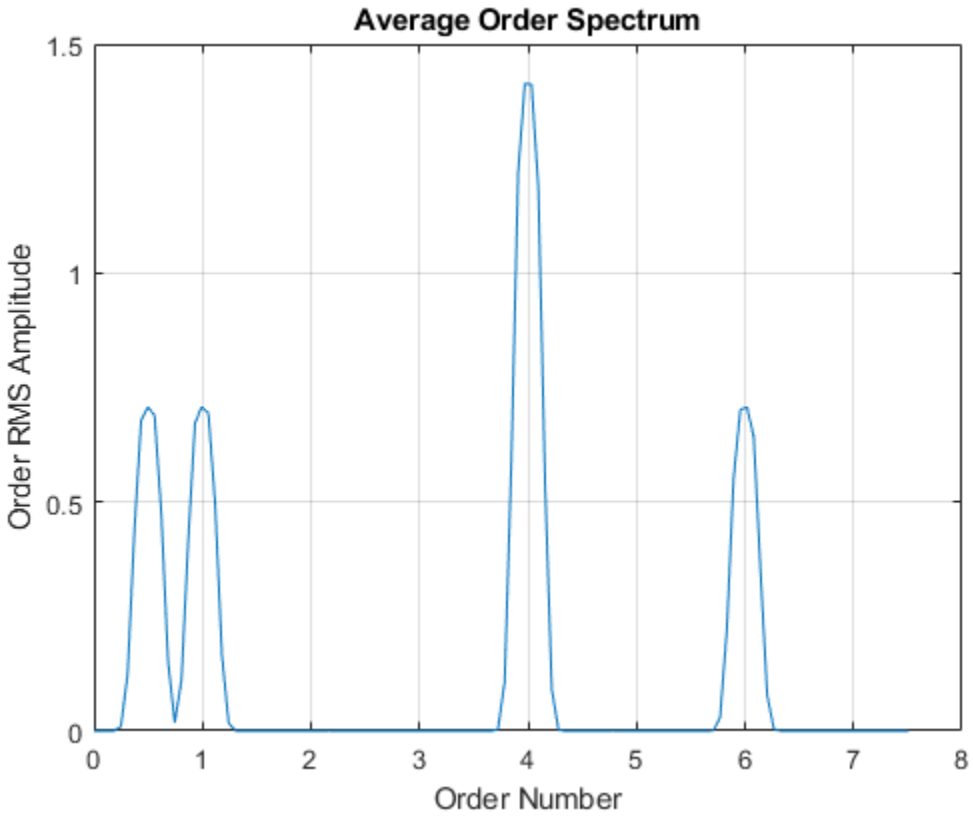
Visualize the order-RPM map of the signal.

```
rpmordermap(x,fs,rpm)
```



Visualize the average order spectrum of the signal. The peaks of the spectrum correspond to the ridges seen in the order-RPM map.

```
orderspectrum(x,fs,rpm)
```



Average Order Spectrum of Helicopter Vibration Data

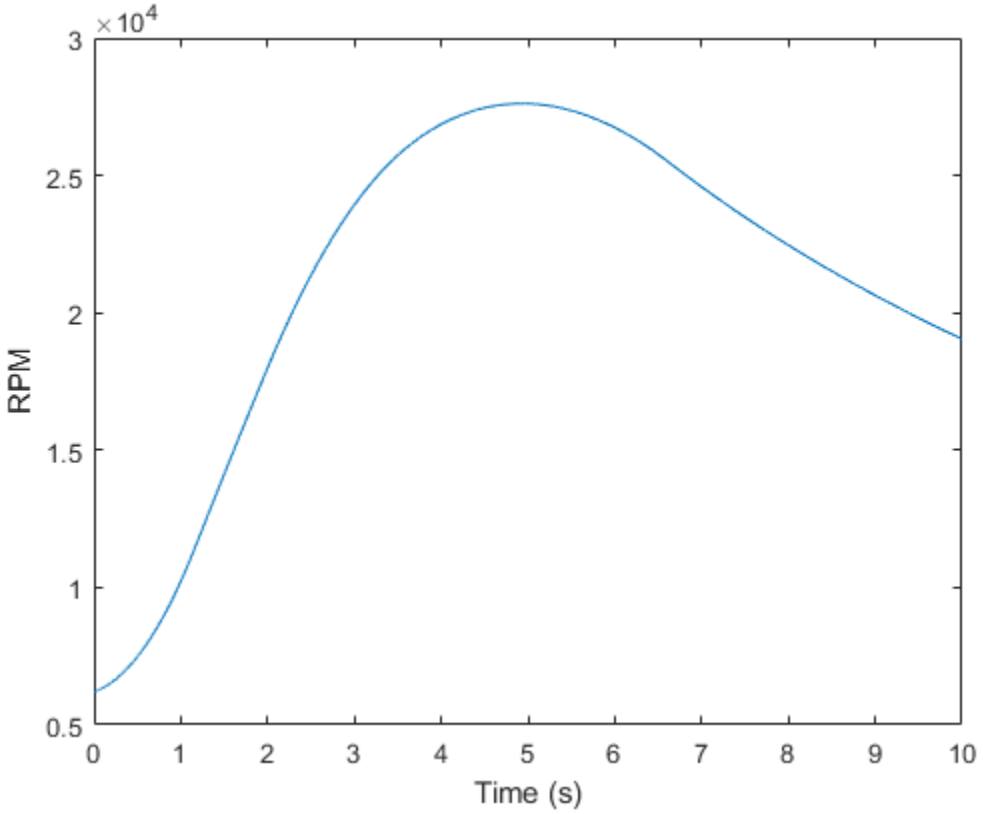
Analyze simulated data from an accelerometer placed in the cockpit of a helicopter.

Load the helicopter data. The vibrational measurements, `vib`, are sampled at a rate of 500 Hz for 10 seconds. The data has a linear trend. Remove the trend to prevent it from degrading the quality of the order estimation.

```
load('helidata.mat')

vib = detrend(vib);
```

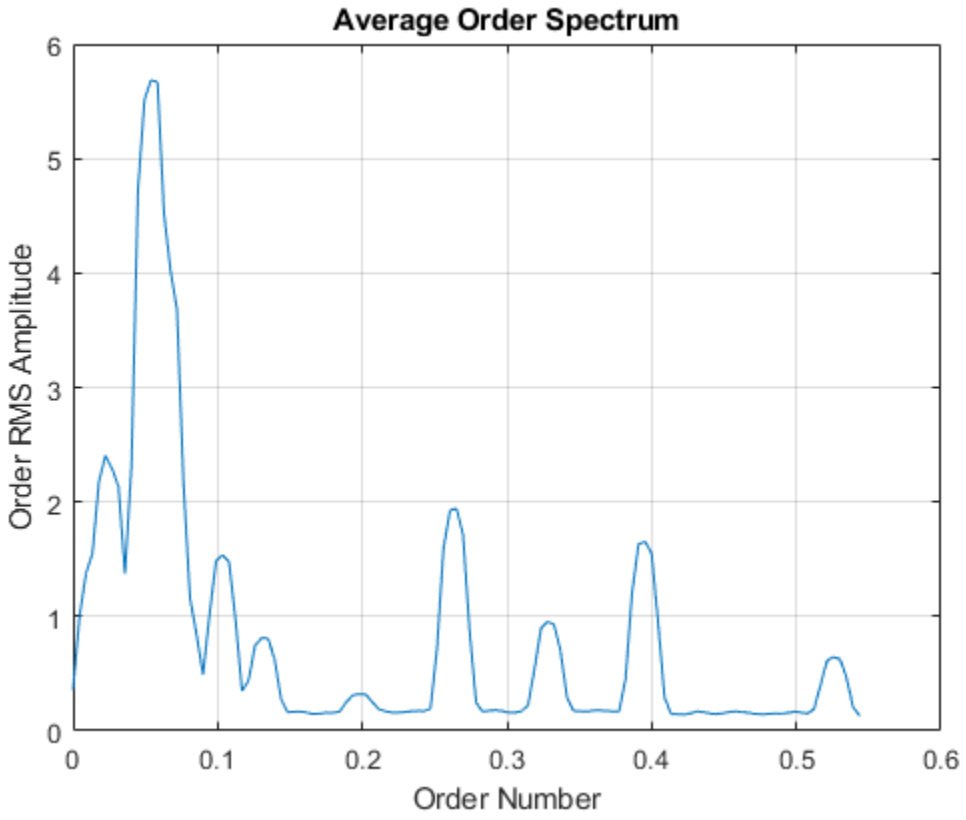
Plot the nonlinear RPM profile. The rotor runs up until it reaches a maximum rotational speed of about 27,600 revolutions per minute and then coasts down.

```
plot(t,rpm)
xlabel('Time (s)')
ylabel('RPM')
```



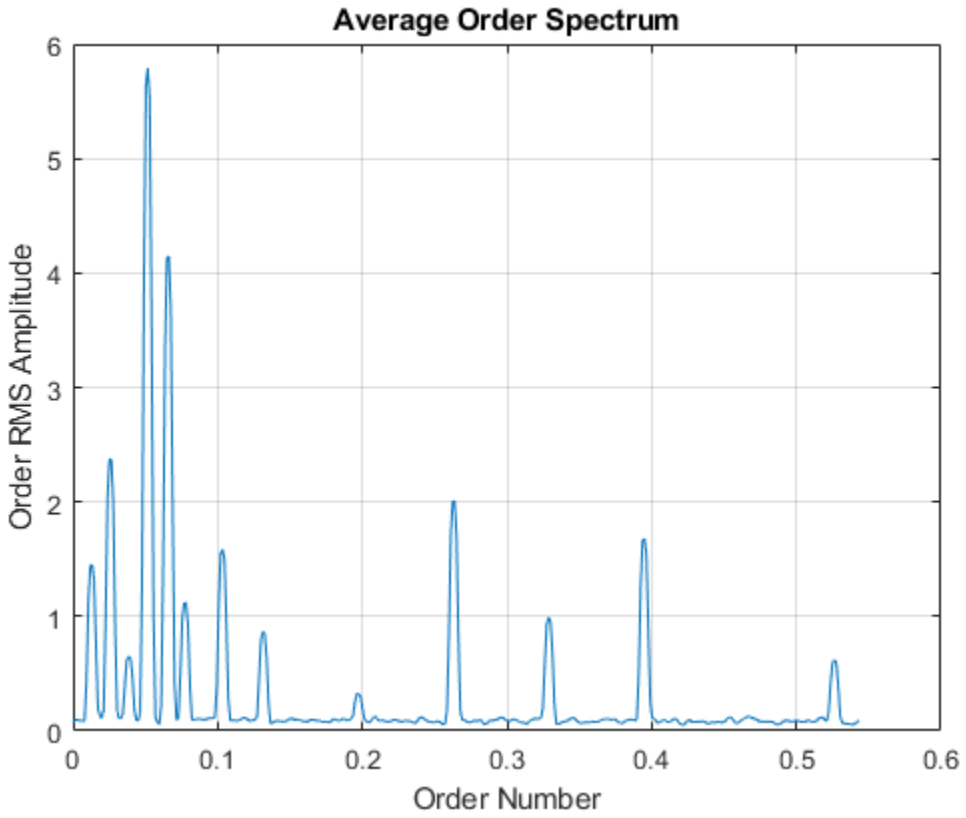
Compute the average order spectrum of the signal. Use the default order resolution.

```
orderspectrum(vib,fs,rpm)
```



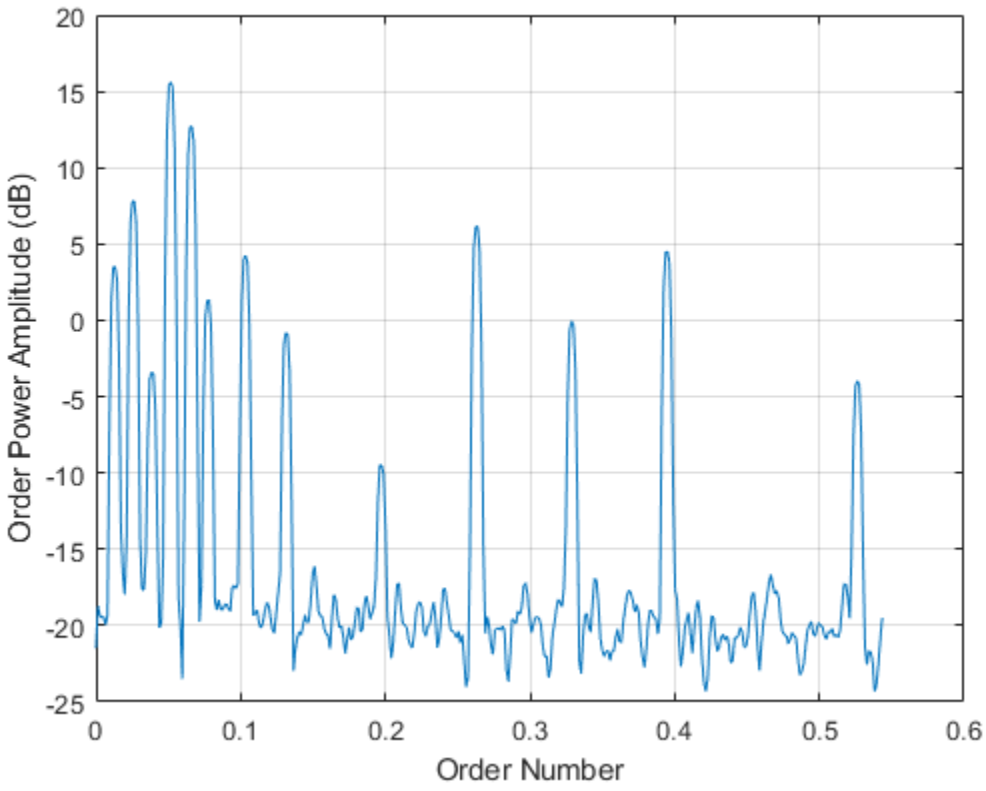
Use `rpmordermap` to repeat the computation with a finer order resolution. The lower orders are resolved more clearly.

```
[map,order] = rpmordermap(vib,fs,rpm,0.005);

orderspectrum(map,order)
```



Compute the power level for each estimated order. Display the result in decibels.

```
[map,order] = rpmordermap(vib,fs,rpm,0.005,'Amplitude','power');

spec = orderspectrum(map,order);

plot(order,pow2db(spec))
xlabel('Order Number')
ylabel('Order Power Amplitude (dB)')
grid on
```



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.

fs — Sample rate

positive scalar

Sample rate, specified as a positive scalar expressed in Hz.

rpm — Rotational speeds

vector of positive values

Rotational speeds, specified as a vector of positive values expressed in revolutions per minute. `rpm` must have the same length as `x`.

- If you have a tachometer pulse signal, use `tachorpm` to extract `rpm` directly.
- If you do not have a tachometer pulse signal, use `rpmtrack` to extract `rpm` from a vibration signal.

Example: `100:10:3000` specifies that a system rotates initially at 100 revolutions per minute and runs up to 3000 revolutions per minute in increments of 10.

`map` — Order-RPM map
matrix

Order-RPM map, specified as a matrix. Use `rpmordermap` to compute order-RPM maps.

`order` — Orders in order-RPM map syntax
vector

Orders in order-RPM map syntax, specified as a vector. The length of `order` must equal the number of rows in `map`.

`amp` — Order-RPM map amplitudes
`'rms'` (default) | `'peak'` | `'power'`

Order-RPM map amplitudes, specified as one of `'rms'`, `'peak'`, or `'power'`.

- `'rms'` — Assumes that the order-RPM map uses the root-mean-square amplitude for each estimated order.
- `'peak'` — Assumes that the order-RPM map uses the peak amplitude for each estimated order.
- `'power'` — Assumes that the order-RPM map uses the power level for each estimated order.

Output Arguments

`spec` — Average order-magnitude spectrum
vector

Average order-magnitude spectrum, returned as a vector of root-mean-square (RMS) amplitudes in linear scale. If you use `map` and `order` as input arguments, and set `'Amplitude'` to `'power'` when using `rpmordermap` to compute `map`, then `orderspectrum` returns `spec` in power units.

`order` — Output orders
real vector

Output orders, returned as a real vector.

References

[1] Brandt, Anders. *Noise and Vibration Analysis: Signal Analysis and Experimental Procedures*. Chichester, UK: John Wiley & Sons, 2011.

[2] Vold, Håvard, and Jan Leuridan. “High Resolution Order Tracking at Extreme Slew Rates Using Kalman Tracking Filters.” *Shock and Vibration*. Vol. 2, 1995, pp. 507–515.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`ordertrack` | `orderwaveform` | `rpmfreqmap` | `rpmordermap` | `tachorpm`

Topics

“Order Analysis of a Vibration Signal”

Introduced in R2016b

ordertrack

Track and extract order magnitudes from vibration signal

Syntax

```
mag = ordertrack(x,fs,rpm,orderlist)
[mag,rpm,time] = ordertrack(x,fs,rpm,orderlist)

[___] = ordertrack(x,fs,rpm,orderlist,rpmrefidx)

[___] = ordertrack(map,order,rpm,time,orderlist)

[___] = ordertrack(___,Name,Value)

ordertrack(___)
```

Description

`mag = ordertrack(x,fs,rpm,orderlist)` returns a matrix, `mag`, that contains time-dependent root-mean-square (RMS) amplitude estimates of a specified set of orders, `orderlist`, present in input signal `x`. `x` is measured at a set `rpm` of rotational speeds expressed in revolutions per minute. `fs` is the measurement sample rate in Hz.

`[mag,rpm,time] = ordertrack(x,fs,rpm,orderlist)` also returns vectors of RPM and time values corresponding to the columns of `mag`.

`[___] = ordertrack(x,fs,rpm,orderlist,rpmrefidx)` extracts order magnitudes using the first-order Vold-Kalman filter and returns any of the output arguments from previous syntaxes.

`[___] = ordertrack(map,order,rpm,time,orderlist)` computes a matrix of magnitude estimates starting from an order-RPM map, `map`, a vector of orders, `order`, and a vector of time instants, `time`. Use `rpmordermap` to compute `map`, `order`, and `time`. The returned amplitudes and scaling are the same as in `map`.

`[___] = ordertrack(___,Name,Value)` specifies further options using `Name,Value` pairs. Some of the options apply only to the Vold-Kalman tracking procedure.

`ordertrack(___)` with no output arguments plots in the current figure the time-dependent orders and RPM values.

Examples

Order Magnitudes of Chirp with Four Orders

Create a simulated signal sampled at 600 Hz for 5 seconds. The system that is being tested increases its rotational speed from 10 to 40 revolutions per second (or, equivalently, from 600 to 2400 revolutions per minute) during the observation period.

Generate the tachometer readings.

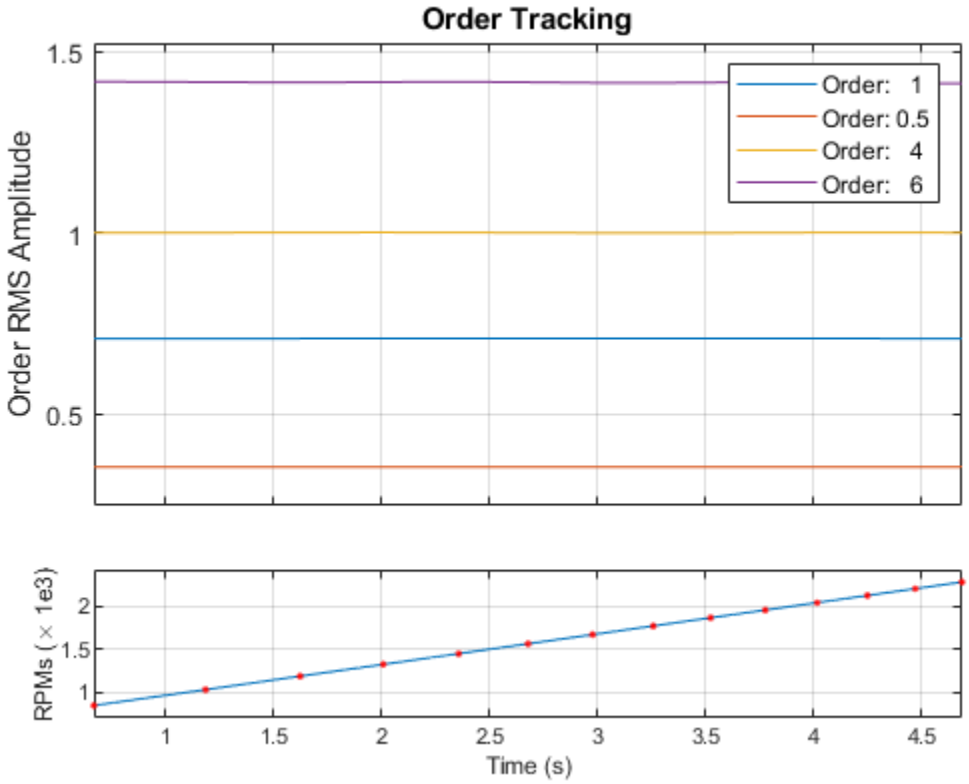
```
fs = 600;
t1 = 5;
t = 0:1/fs:t1;

f0 = 10;
f1 = 40;
rpm = 60*linspace(f0,f1,length(t));
```

The signal consists of four harmonically related chirps with orders 1, 0.5, 4, and 6. The amplitudes of the chirps are 1, 1/2, √2, and 2, respectively. To generate the chirps, use the trapezoidal rule to express the phase as the integral of the rotational speed.

```
o1 = 1;
o2 = 0.5;
o3 = 4;
o4 = 6;

a1 = 1;
a2 = 0.5;
a3 = sqrt(2);
a4 = 2;

ph = 2*pi*cumtrapz(rpm/60)/fs;

x = [a1 a2 a3 a4]*cos([o1 o2 o3 o4]'*ph);
```

Extract and visualize the magnitudes of the orders.

```
ordertrack(x,fs,rpm,[o1 o2 o3 o4])
```



Track Crossing Orders

Create a simulated vibration signal consisting of two crossing orders corresponding to two different motors. The signal is sampled at 300 Hz for 3 seconds. The first motor increases its rotational speed from 10 to 100 revolutions per second (or, equivalently, from 600 to 6000 revolutions per minute) during the measurement. The second motor increases its rotational speed from 50 to 70 revolutions per second (or 3000 to 4200 revolutions per minute) during the same period.

```
fs = 300;
nsamp = 3*fs;

rpm1 = linspace(10,100,nsamp)'*60;
rpm2 = linspace(50,70,nsamp)'*60;
```

The measured signal is of order 1.2 and amplitude $2\sqrt{2}$ with respect to the first motor. With respect to the second motor, the signal is of order 0.8 and amplitude $4\sqrt{2}$.

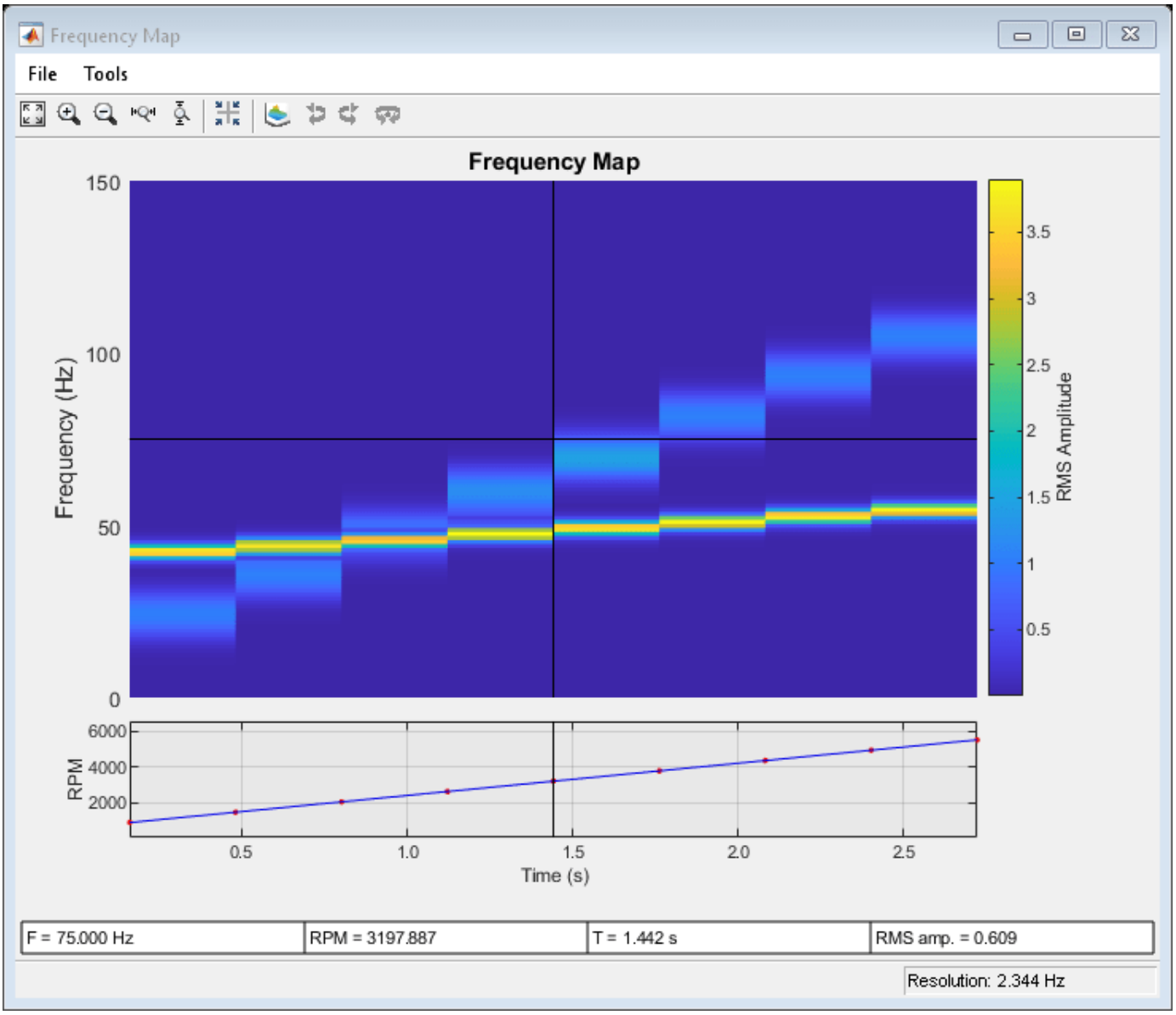
```
x = [2 4]*sqrt(2).*cos(2*pi*cumtrapz([1.2*rpm1 0.8*rpm2]/60)/fs);
```

Make the first motor excite a resonance at the middle of the frequency range.

```
rs = [1+1./(1+linspace(-10,10,nsamp).^4)'/2 ones(nsamp,1)];

x = sum(rs.*x,2);
```

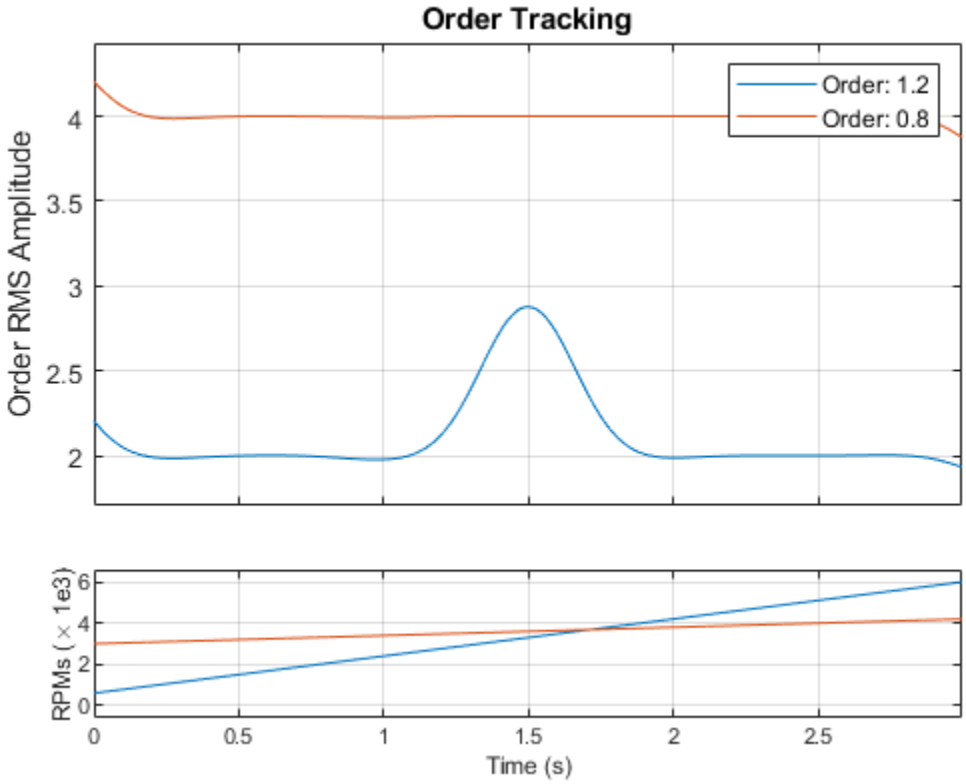
Visualize the orders using `rpmfreqmap`.

```
rpmfreqmap(x,fs,rpm1)
```



Compute the order magnitudes for both motors as a function of RPM. Use the Vold-Kalman algorithm to decouple the crossing orders.

```
ordertrack(x,fs,[rpm1 rpm2],[1.2 0.8],[1 2],'Decouple',true)
```



Track Orders of Helicopter Vibration Data

Analyze simulated data from an accelerometer placed in the cockpit of a helicopter.

Load the helicopter data. The vibrational measurements, `vib`, are sampled at a rate of 500 Hz for 10 seconds. Inspection of the data reveals that it has a linear trend. Remove the trend to prevent it from degrading the quality of the order estimation.

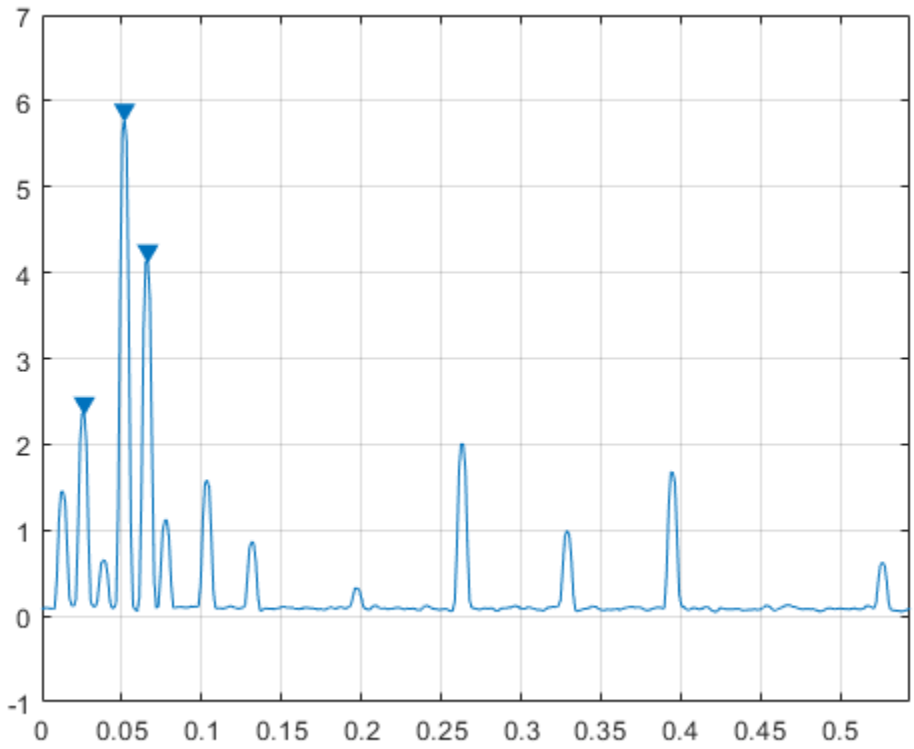
```
load('helidata.mat')

vib = detrend(vib);
```

Compute the order-RPM map. Specify an order resolution of 0.005.

```
[map,order,rpm,time,res] = rpmordermap(vib,fs,rpm,0.005);
```

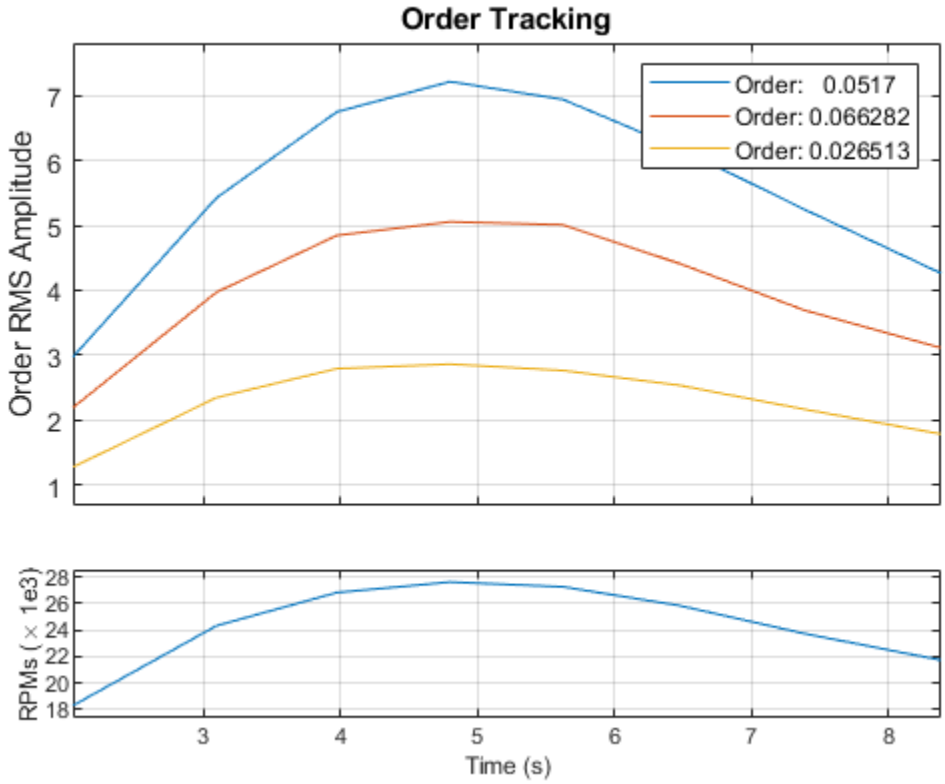
Compute and plot the average order spectrum of the signal. Find the three highest peaks of the spectrum.

```
[spectrum,specorder] = orderspectrum(map,order);

[~,pkords] = findpeaks(spectrum,specorder,'SortStr','descend','Npeaks',3);

findpeaks(spectrum,specorder,'SortStr','descend','Npeaks',3)
```



Track the amplitudes of the three highest peaks.

```
ordertrack(map,order,rpm,time,pkords)
```



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: `double` | `single`

fs — Sample rate

positive scalar

Sample rate, specified as a positive scalar expressed in Hz.

Data Types: `double` | `single`

rpm — Rotational speeds

vector of positive values | matrix of positive values

Rotational speeds, specified as a vector or matrix of positive values expressed in revolutions per minute. If `rpm` is a vector, it must have the same length as `x`. If `rpm` is a matrix, and `rpmrefidx` is specified, then `rpm` must have at least two columns, and each column must have as many elements as `x`.

- If you have a tachometer pulse signal, use `tachorpm` to extract `rpm` directly.
- If you do not have a tachometer pulse signal, use `rpmtrack` to extract `rpm` from a vibration signal.

Example: `100:10:3000` specifies that a system rotates initially at 100 revolutions per minute and runs up to 3000 revolutions per minute in increments of 10.

Data Types: `double` | `single`

`orderlist` — List of orders
vector

List of orders, specified as a vector. `orderlist` must not have values larger than `fs`/(2 × max(`rpm`/60)).

Data Types: `double` | `single`

`rpmrefidx` — RPM column indices
vector

RPM column indices, specified as a vector of the same size as `orderlist`. The presence of this argument specifies that the Vold-Kalman algorithm is to be used.

Data Types: `double` | `single`

`map` — Order-RPM map
matrix

Order-RPM map, specified as a matrix. Use `rpmordermap` to compute order-RPM maps.

Data Types: `double` | `single`

`order` — Orders in order-RPM map syntax
vector

Orders in order-RPM map syntax, specified as a vector. The length of `order` must equal the number of rows in `map`.

Data Types: `double` | `single`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Decouple',true,'Amplitude','peak'` extracts the specified orders simultaneously and returns the peak amplitude of each order.

`Amplitude` — Amplitude type
`'rms'` (default) | `'peak'` | `'power'`

Amplitude type, specified as the comma-separated pair consisting of `'Amplitude'` and one of `'rms'`, `'peak'`, or `'power'`.

- `'rms'` — Returns the root-mean-square amplitude for each estimated order.

- `'peak'` — Returns the peak amplitude for each estimated order.
- `'power'` — Returns the power level for each estimated order.

`Scale` — Magnitude scaling
`'linear'` (default) | `'dB'`

Magnitude scaling, specified as the comma-separated pair consisting of `'Scale'` and either `'linear'` or `'dB'`.

- `'linear'` — Returns magnitude values scaled in linear units.
- `'dB'` — Returns magnitude values scaled logarithmically and expressed in decibels.

`Bandwidth` — Approximate half-power bandwidth
`fs`/100 (default) | real scalar | real vector

Approximate half-power bandwidth, specified as the comma-separated pair consisting of `'Bandwidth'` and either a real scalar or a real vector with the same number of elements as `orderlist`. Smaller values of `'Bandwidth'` produce smooth, narrowband output. However, this output might not accurately reflect rapid changes in order amplitude. This argument applies only to the Vold-Kalman algorithm.

Data Types: `double` | `single`

`Decouple` — Mode decoupling option
`false` (default) | `true`

Mode decoupling option, specified as the comma-separated pair consisting of `'Decouple'` and a logical value. If this option is set to `true`, then `ordertrack` extracts order magnitudes simultaneously, enabling it to separate closely spaced or crossing orders. This argument applies only to the Vold-Kalman algorithm.

Data Types: `logical`

`SegmentLength` — Length of overlapping segments
integer

Length of overlapping segments, specified as the comma-separated pair consisting of `'SegmentLength'` and an integer. If you specify a segment length, then `ordertrack` divides the input signal into segments. It then computes the order magnitudes for each segment and combines the results to produce the output. If the segments are too short, the function might not properly capture localized events such as crossing orders. This argument applies only to the Vold-Kalman algorithm.

Data Types: `double` | `single`

Output Arguments

`mag` — Order-magnitude matrix
matrix

Order-magnitude matrix, returned as a matrix.

`rpm` — Rotational speeds
vector of positive values

Rotational speeds, returned as a vector of positive values expressed in revolutions per minute.

`time` — Time instants
vector

Time instants, returned as a vector.

References

[1] Brandt, Anders. *Noise and Vibration Analysis: Signal Analysis and Experimental Procedures*. Chichester, UK: John Wiley & Sons, 2011.

[2] Feldbauer, Christian, and Robert Höldrich. "Realization of a Vold-Kalman Tracking Filter — A Least Squares Problem." *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00)*. Verona, Italy, December 7–9, 2000.

[3] Vold, Håvard, and Jan Leuridan. "High Resolution Order Tracking at Extreme Slew Rates Using Kalman Tracking Filters." *Shock and Vibration*. Vol. 2, 1995, pp. 507–515.

[4] Tůma, Jiří. "Algorithms for the Vold-Kalman Multiorder Tracking Filter." *Proceedings of the 14th International Carpathian Control Conference (ICCC)*, 2013, pp. 388–94. https://doi.org/10.1109/CarpathianCC.2013.6560575.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the second input argument represents the sample rate, it must be specified as a scalar at compile time.
- If the function uses the Vold-Kalman algorithm, then the generated code results may show small numerical differences with respect to MATLAB results.

See Also

`orderspectrum` | `orderwaveform` | `rpmfreqmap` | `rpmordermap` | `tachorpm`

Topics
"Order Analysis of a Vibration Signal"

Introduced in R2016b

orderwaveform

Extract time-domain order waveforms from vibration signal

Syntax

```
xrec = orderwaveform(x,fs,rpm,orderlist)
xrec = orderwaveform(x,fs,rpm,orderlist,rpmrefidx)
xrec = orderwaveform(x,fs,rpm,orderlist,rpmrefidx,Name,Value)
```

Description

`xrec = orderwaveform(x,fs,rpm,orderlist)` returns the time-domain waveforms corresponding to a specified set of orders present in an input signal, `x`. `x` is measured at a set `rpm` of rotational speeds expressed in revolutions per minute. `fs` is the measurement sample rate in Hz. The vector `orderlist` specifies the desired orders, whose waveforms are returned in the corresponding columns of `xrec`. The function uses the Vold-Kalman filter for the computation.

`xrec = orderwaveform(x,fs,rpm,orderlist,rpmrefidx)` returns time-domain waveforms with multiple reference RPM signals, which are stored in the columns of `rpm`. `rpmrefidx` is a vector that relates each order in `orderlist` to an RPM signal.

`xrec = orderwaveform(x,fs,rpm,orderlist,rpmrefidx,Name,Value)` specifies further options for the Vold-Kalman procedure using `Name,Value` pairs.

Examples

Order Waveforms of Chirp with Four Orders

Create a simulated signal sampled at 600 Hz for 5 seconds. The system that is being tested increases its rotational speed from 10 to 40 revolutions per second (or, equivalently, from 600 to 2400 revolutions per minute) during the observation period.

Generate the tachometer readings.

```
fs = 600;
t1 = 5;
t = 0:1/fs:t1;

f0 = 10;
f1 = 40;
rpm = 60*linspace(f0,f1,length(t));
```

The signal consists of four harmonically related chirps with orders 1, 1/2, √2, and 2. The amplitudes of the chirps are 1, 1/2, √2, and 2, respectively. To generate the chirps, use the trapezoidal rule to express the phase as the integral of the rotational speed.

```
ord = [1 0.5 sqrt(2) 2];
amp = [1 0.5 sqrt(2) 2];

ph = 2*pi*cumtrapz(rpm/60)/fs;
```

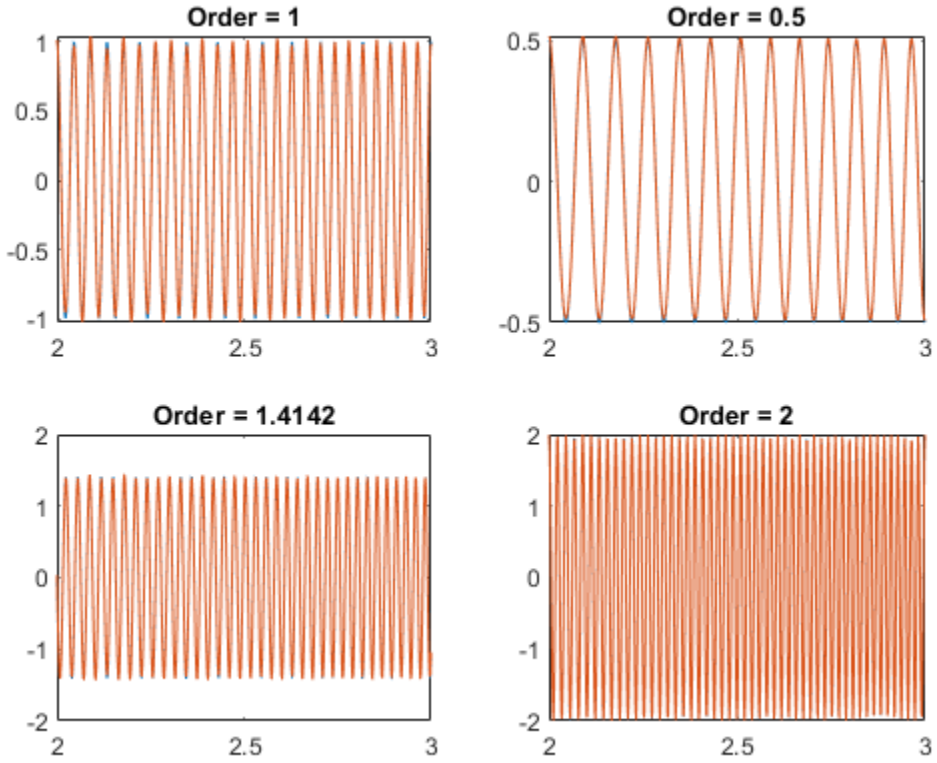
```
x(1,:) = amp(1)*cos(ord(1)*ph);
x(2,:) = amp(2)*cos(ord(2)*ph);
x(3,:) = amp(3)*cos(ord(3)*ph);
x(4,:) = amp(4)*cos(ord(4)*ph);

xsum = sum(x);
```

Reconstruct the time-domain waveforms that compose the signal.

```
xrec = orderwaveform(xsum,fs,rpm,ord);
```

Visualize the results. Zoom in on a time interval occurring after the transients have decayed.

```
for kj = 1:4
    subplot(2,2,kj)
    plot(t,x(kj,:),t,xrec(:,kj))
    title(['Order = ' num2str(ord(kj))])
    xlim([2 3])
end
```



Extract Waveforms of Crossing Orders

Create a simulated vibration signal consisting of two crossing orders corresponding to two different motors. The signal is sampled at 300 Hz for 3 seconds. The first motor increases its rotational speed

from 10 to 100 revolutions per second (or, equivalently, from 600 to 6000 rpm) during the measurement. The second motor increases its rotational speed from 50 to 70 revolutions per second (or 3000 to 4200 rpm) during the same period.

```
fs = 300;
nsamp = 3*fs;

rpm1 = linspace(10,100,nsamp)'*60;
rpm2 = linspace(50,70,nsamp)'*60;
```

The measured signal is of order 1.2 and amplitude $2\sqrt{2}$ with respect to the first motor. With respect to the second motor, the signal is of order 0.8 and amplitude $4\sqrt{2}$.

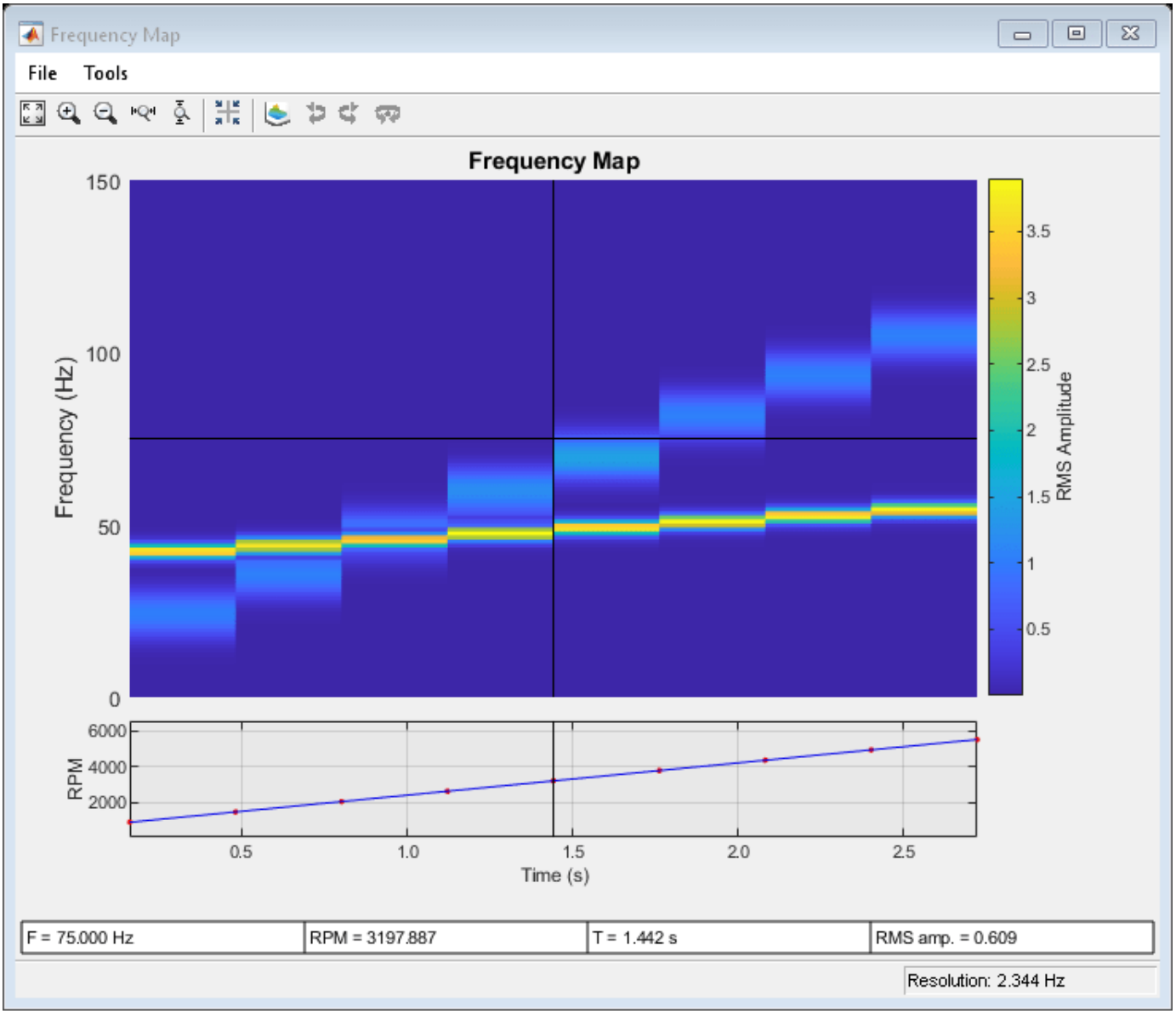
```
x = [2 4]*sqrt(2).*cos(2*pi*cumtrapz([1.2*rpm1 0.8*rpm2]/60)/fs);
```

Make the first motor excite a resonance at the middle of the frequency range.

```
y = [1+1./(1+linspace(-10,10,nsamp).^4)'/2 ones(nsamp,1)].*x;

x = sum(y,2);
```

Visualize the orders using `rpmfreqmap`.

```
rpmfreqmap(x,fs,rpm1)
```



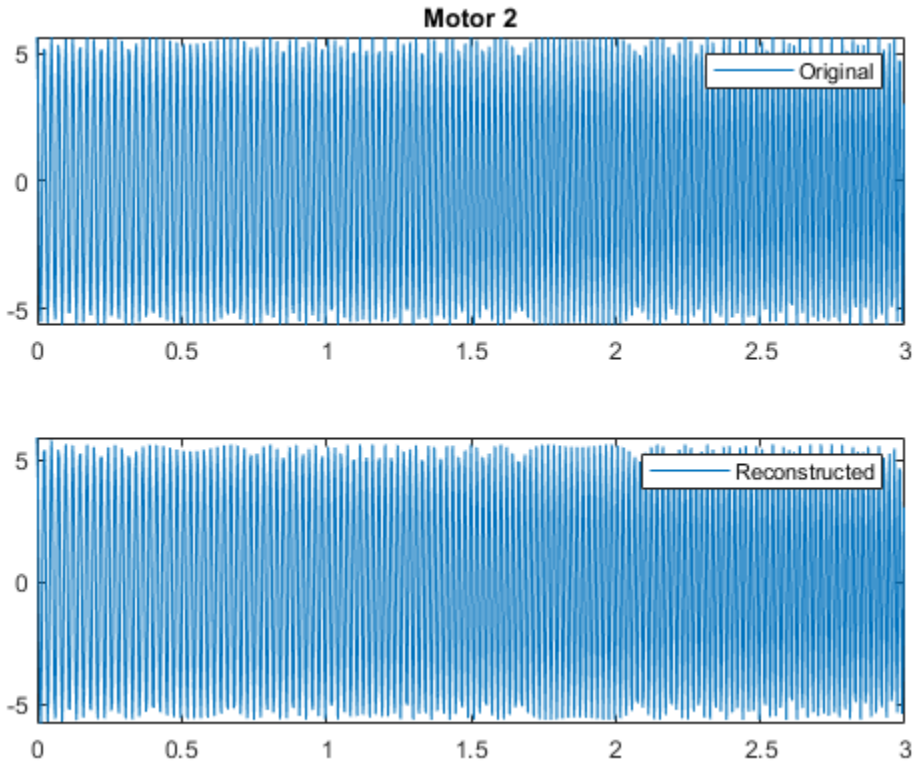
Reconstruct the time-domain waveforms that compose the signal. Use the Vold-Kalman algorithm to decouple the crossing orders.

```
xrec = orderwaveform(x,fs,[rpm1 rpm2],[1.2 0.8],[1 2],'Decouple',true);
```

Plot the original and reconstructed waveforms.

```
for kj = 1:2
    figure(kj)
    subplot(2,1,1)
    plot((0:nsamp-1)/fs,y(:,kj))
    legend('Original')
    title(['Motor ' int2str(kj)])
    subplot(2,1,2)
    plot((0:nsamp-1)/fs,xrec(:,kj))
```

```
    legend('Reconstructed')
end
```



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.

fs — Sample rate

positive scalar

Sample rate, specified as a positive scalar expressed in Hz.

rpm — Rotational speeds

vector of positive values

Rotational speeds, specified as a vector of positive values expressed in revolutions per minute. `rpm` must have the same length as `x`.

- If you have a tachometer pulse signal, use `tachorpm` to extract `rpm` directly.

- If you do not have a tachometer pulse signal, use `rpmtrack` to extract `rpm` from a vibration signal.

Example: `100:10:3000` specifies that a system rotates initially at 100 revolutions per minute and runs up to 3000 revolutions per minute in increments of 10.

`orderlist` — List of orders
vector

List of orders, specified as a vector. `orderlist` must not have values larger than `fs`/(2 × max(`rpm`/60)).

Data Types: `double` | `single`

`rpmrefidx` — RPM column indices
vector

RPM column indices, specified as a vector of the same size as `orderlist`.

Data Types: `double` | `single`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Decouple',true,'FilterOrder',2` extracts the specified order waveforms simultaneously and uses a second-order Vold-Kalman filter.

`FilterOrder` — Vold-Kalman filter order
1 (default) | 2

Vold-Kalman filter order, specified as the comma-separated pair consisting of `'FilterOrder'` and either 1 or 2.

Data Types: `double` | `single`

`Bandwidth` — Approximate half-power bandwidth
`fs`/100 (default) | real scalar | real vector

Approximate half-power bandwidth, specified as the comma-separated pair consisting of `'Bandwidth'` and either a real scalar or a real vector with the same number of elements as `orderlist`. Smaller values of `'Bandwidth'` produce smooth, narrowband output. However, this output might not accurately reflect rapid changes in order amplitude.

Data Types: `double` | `single`

`Decouple` — Mode decoupling option
`false` (default) | `true`

Mode decoupling option, specified as the comma-separated pair consisting of `'Decouple'` and a logical value. If this option is set to `true`, then `orderwaveform` extracts order waveforms simultaneously, enabling it to separate closely spaced or crossing orders.

Data Types: `logical`

`SegmentLength` — Length of overlapping segments
integer

Length of overlapping segments, specified as the comma-separated pair consisting of `'SegmentLength'` and an integer. If you specify a segment length, then `orderwaveform` divides the input signal into segments. It then computes the reconstructed waveforms for each segment and combines the results to produce the output. If the segments are too short, the function might not properly capture localized events such as crossing orders.

Data Types: `double` | `single`

Output Arguments

`xrec` — Reconstructed time-domain order waveforms
matrix

Reconstructed time-domain order waveforms, returned as a matrix with one waveform in each column.

References

[1] Feldbauer, Christian, and Robert Höldrich. “Realization of a Vold-Kalman Tracking Filter — A Least Squares Problem.” *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00)*. Verona, Italy, December 7–9, 2000.

[2] Vold, Håvard, and Jan Leuridan. “High Resolution Order Tracking at Extreme Slew Rates Using Kalman Tracking Filters.” *Shock and Vibration*. Vol. 2, 1995, pp. 507–515.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generated code results may show small numerical differences with respect to MATLAB results.

See Also

`orderspectrum` | `ordertrack` | `rpmfreqmap` | `rpmordermap` | `tachorpm`

Topics
“Order Analysis of a Vibration Signal”

Introduced in R2016b

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 overshoots expressed as a percentage of the difference between the low- and high-state levels in the input bilevel waveform. The values in `os` correspond to the greatest absolute deviations that are greater than the final state levels of each transition.

`os = overshoot(x,fs)` specifies the sample rate `fs` in hertz.

`os = overshoot(x,t)` specifies the sample instants `t`.

`[os,oslev,osinst] = overshoot( ___ )` returns the levels `oslev` and sample instants `osinst` of the overshoots for each transition. You can specify an input combination from any of the previous syntaxes.

`[ ___ ] = overshoot( ___ ,Name,Value)` specifies additional options using one or more `Name,Value` arguments. You can use any of the output combinations from previous syntaxes.

`overshoot( ___ )` plots the bilevel waveform and marks the location of the overshoot of each transition. The function also plots the lower and upper reference-level instants and associated reference levels and the state levels and associated lower- and upper-state boundaries.

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. Determine the maximum percent overshoot of the transition. Determine also the level and sample instant of the overshoot. In this example, the maximum overshoot in the posttransition region occurs near index 22.

```
load('transitionex.mat','x')
```

```
[oo,lv,nst] = overshoot(x)
```

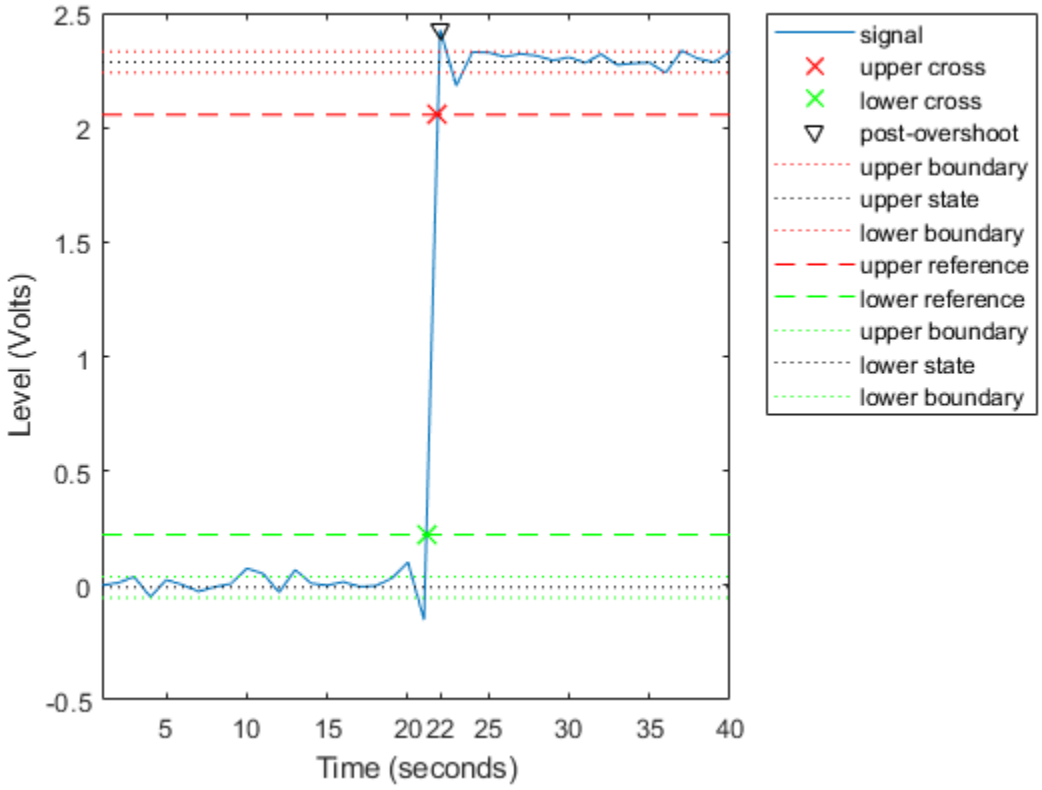
```
oo = 6.1798
```

```
lv = 2.4276
```

```
nst = 22
```

Plot the waveform. Annotate the overshoot and the corresponding sample instant.

```
overshoot(x);

ax = gca;
ax.XTick = sort([ax.XTick nst]);
```



Overshoot Percentage, Levels, and Time 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. The clock data are sampled at 4 MHz.

```
load('transitionex.mat','x','t')
```

Determine the maximum percent overshoot, the level of the overshoot in volts, and the time instant where the maximum overshoot occurs. Plot the result.

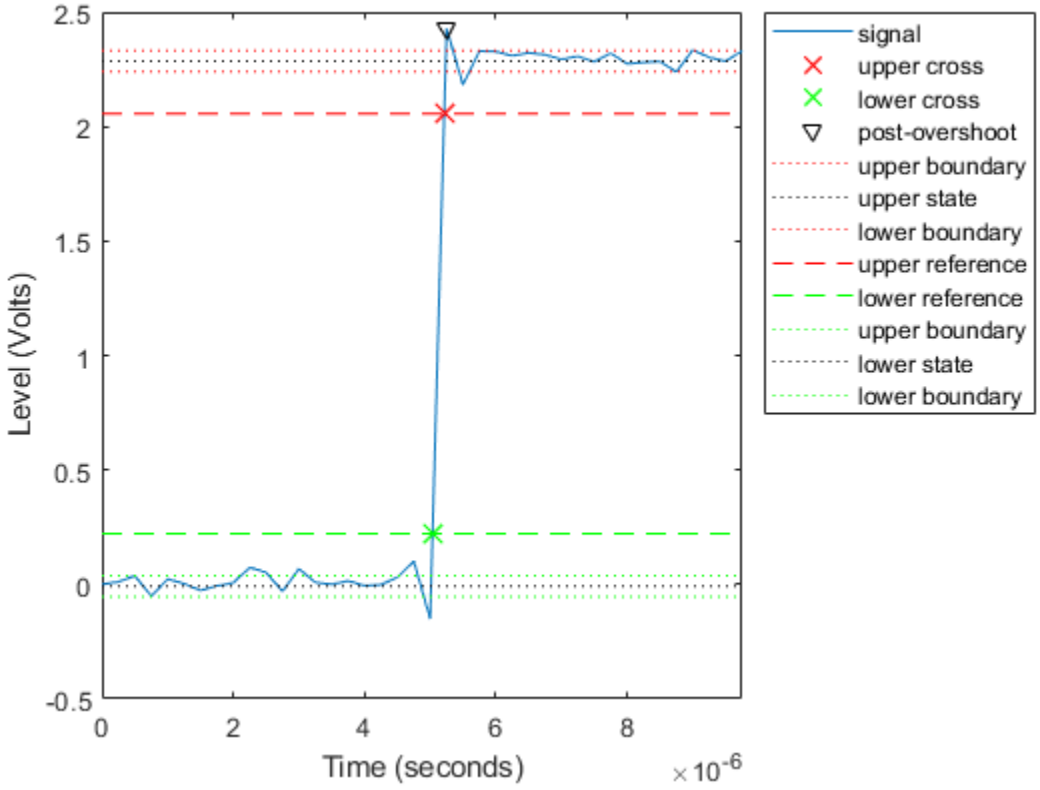
```
[os,oslev,osinst] = overshoot(x,t)
```

```
os = 6.1798
```

```
oslev = 2.4276
```

```
osinst = 5.2500e-06
```

```
overshoot(x,t);
```



Overshoot Percentage, Levels, and Time 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. The clock data are sampled at 4 MHz.

```
load('transitionex.mat','x','t')
```

Determine the maximum percent overshoot, the level of the overshoot in volts, and the sampling instant where the maximum overshoot occurs. Plot the result.

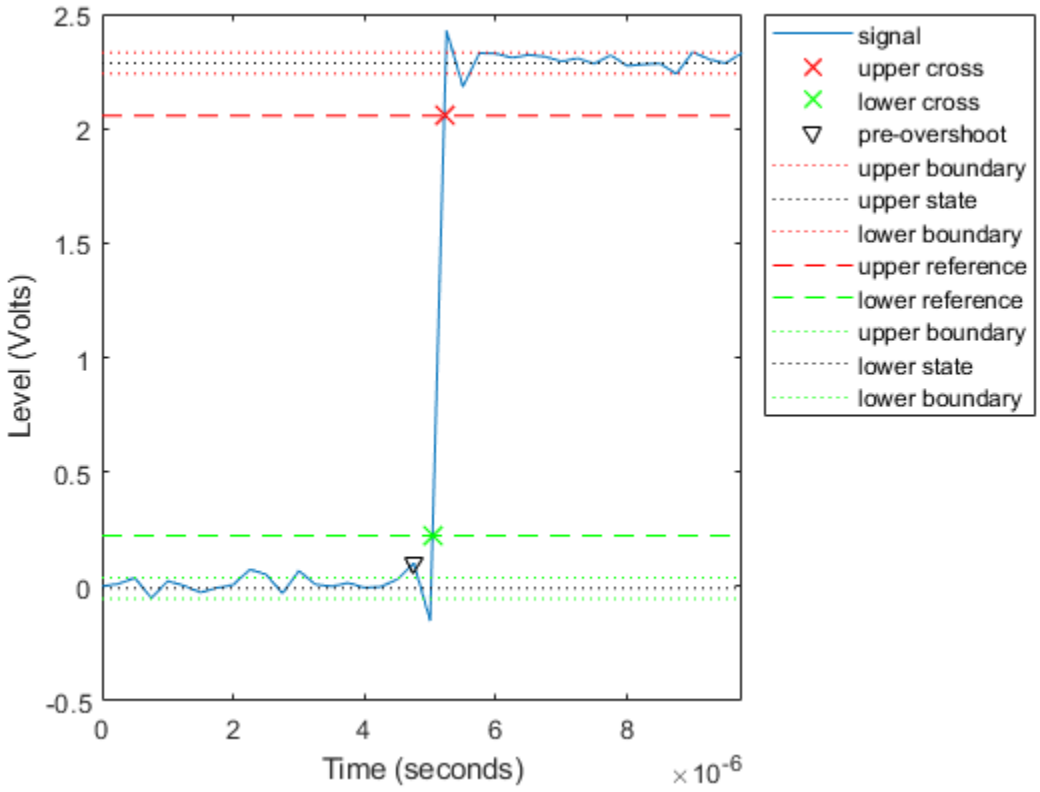
```
[os,oslev,osinst] = overshoot(x,t,'Region','Preshoot')
```

```
os = 4.8050
```

```
oslev = 0.1020
```

```
osinst = 4.7500e-06
```

```
overshoot(x,t,'Region','Preshoot');
```



Input Arguments

x — Bilevel waveform

real-valued vector

Bilevel waveform, specified as a real-valued vector. The sample instants in `x` correspond to the vector indices. The first sample instant in `x` corresponds to `t = 0`.

fs — Sample rate

real positive scalar

Sample rate in hertz, specified as a real positive scalar. The sample rate determines the sample instants corresponding to the elements in `x`.

t — Sample instants

vector

Sample instants, specified as a vector. The length of `t` must equal the length of the input bilevel waveform `x`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Region','Preshoot'` specifies the pretransition aberration region.

`PercentReferenceLevels` — Reference levels

`[10 90]` (default) | 1-by-2 real-valued vector

Reference levels as a percentage of the waveform amplitude, specified as a 1-by-2 real-valued vector. The function defines the lower-state level to be 0 percent and the upper-state level to be 100 percent. The first element corresponds to the lower percent reference level, and the second element corresponds to the upper percent reference level.

`Region` — Aberration region

`'Postshoot'` (default) | `'Preshoot'`

Aberration region over which to compute the overshoot, specified as `'Preshoot'` or `'Postshoot'`. If you specify `'Preshoot'`, the function defines the end of the pretransition aberration region as the last instant when the signal exits the first state. If you specify `'Postshoot'`, the function defines the start of the posttransition aberration region as the instant when the signal enters the second state. By default, the function computes overshoots for posttransition aberration regions.

`SeekFactor` — Aberration region duration

3 (default) | real-valued scalar

Aberration region duration, specified as a real-valued scalar. The function computes the overshoot over the specified duration 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 aberration region duration elapses, the duration is truncated to the edge of the waveform or the start of the intervening transition.

`StateLevels` — Low- and high-state levels

1-by-2 real-valued vector

Low- and high-state levels, specified as a 1-by-2 real-valued vector. The first element corresponds to the low-state level and the second element corresponds to the high-state level of the input waveform.

`Tolerance` — Tolerance level

2 (default) | real-valued scalar

Tolerance level, specified as a real-valued scalar. The function expresses tolerance as a percentage of the difference between the upper and lower state levels. The initial and final levels of each transition must be within the respective state levels.

Output Arguments

`os` — Overshoots

vector

Overshoots expressed as a percentage of the state levels, returned as a vector. The length of `OS` corresponds to the number of transitions detected in the input signal. For more information, see “Overshoot” on page 1-1518.

`oslev` — Overshoot level
column vector

Overshoot level, returned as a column vector.

`osinst` — Sample instants
column vector

Sample instants of pretransition or posttransition overshoots, returned as a column vector. If you specify `fs` or `t`, the overshoot instants are in seconds. If you do not specify`fs` or `t`, the overshoot instants are the indices of the input vector.

More About

State-Level Estimation

To determine the transitions, the `overshoot` function estimates the state levels of the input bilevel waveform `x` by using a histogram method with these steps.

1. Determine the minimum and maximum amplitudes of the data.
2. For the specified number of histogram bins, determine the bin width, which is the ratio of the amplitude range to the number of bins.
3. Sort the data values into the histogram bins.
4. Identify the lowest and highest indexed histogram bins with nonzero counts.
5. Divide the histogram into two subhistograms.
6. Compute the state levels by determining the mode or mean of the upper and lower histograms.

The function 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.

Overshoot

The function computes the overshoot percentages based on the greatest deviation from the final state level in each transition.

For a positive-going (positive-polarity) pulse, the overshoot is given by

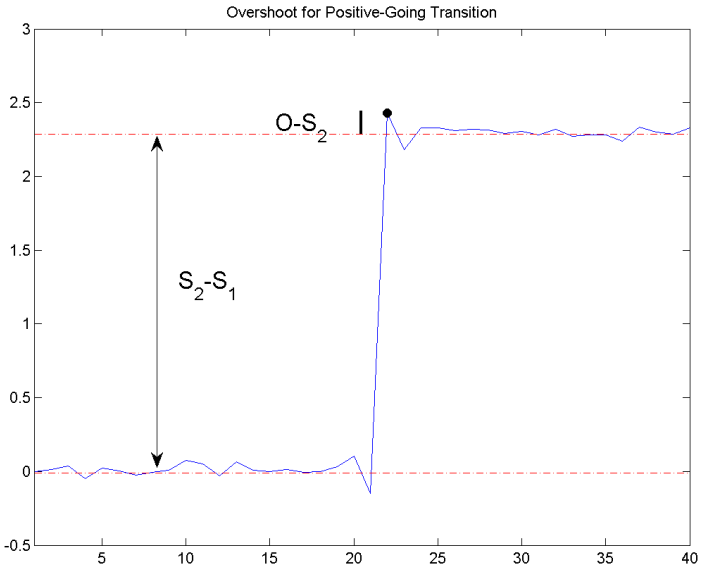
$$100\frac{(O - S_2)}{(S_2 - S_1)}$$

where O is the maximum deviation greater than the high-state level, S_2 is the high state, and S_1 is the low state.

For a negative-going (negative-polarity) pulse, the overshoot is given by

$$100\frac{(O - S_1)}{(S_2 - S_1)}$$

This figure shows 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.

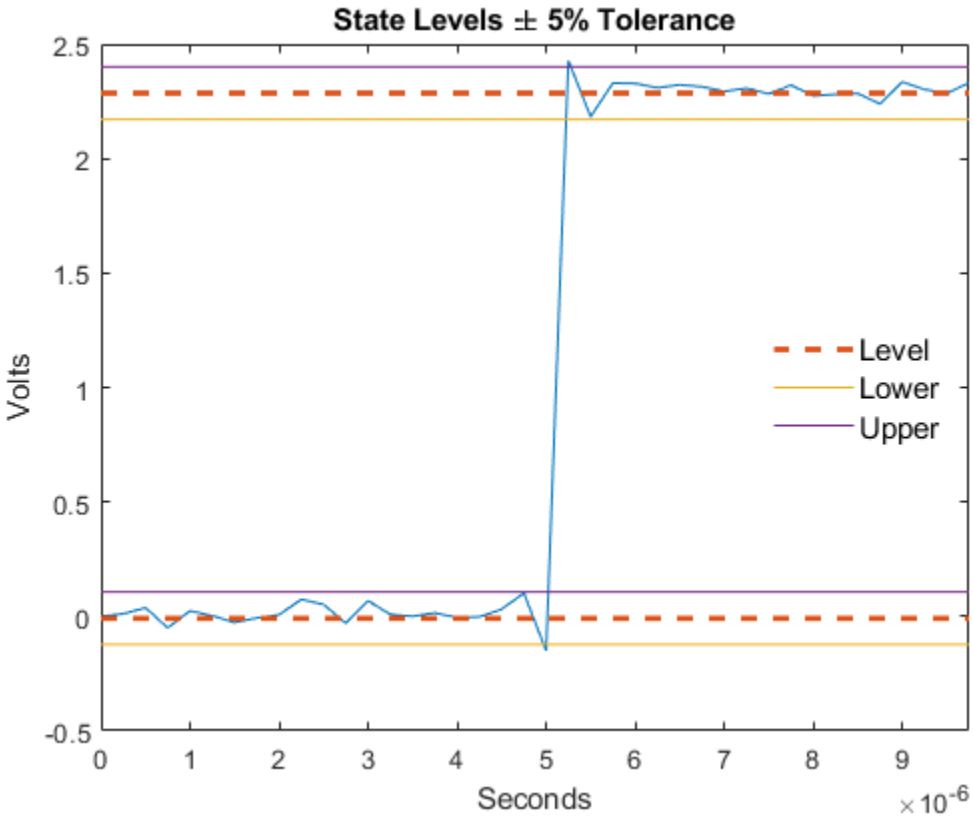
State-Level Tolerances

You can specify lower- and upper-state boundaries for each state level. Define the boundaries as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, specify the scalar as a small number such as 2/100 or 3/100. In general, the $\alpha\%$ 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.

This figure shows lower and upper 5% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The thick dashed lines indicate the estimated state levels.



References

[1] IEEE Standard 181. *IEEE Standard on Transitions, Pulses, and Related Waveforms* (2003): 15–17.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`settlingtime` | `statelevels`

Introduced in R2012a

parzenwin

Parzen (de la Vallée Poussin) window

Syntax

```
w = parzenwin(L)
```

Description

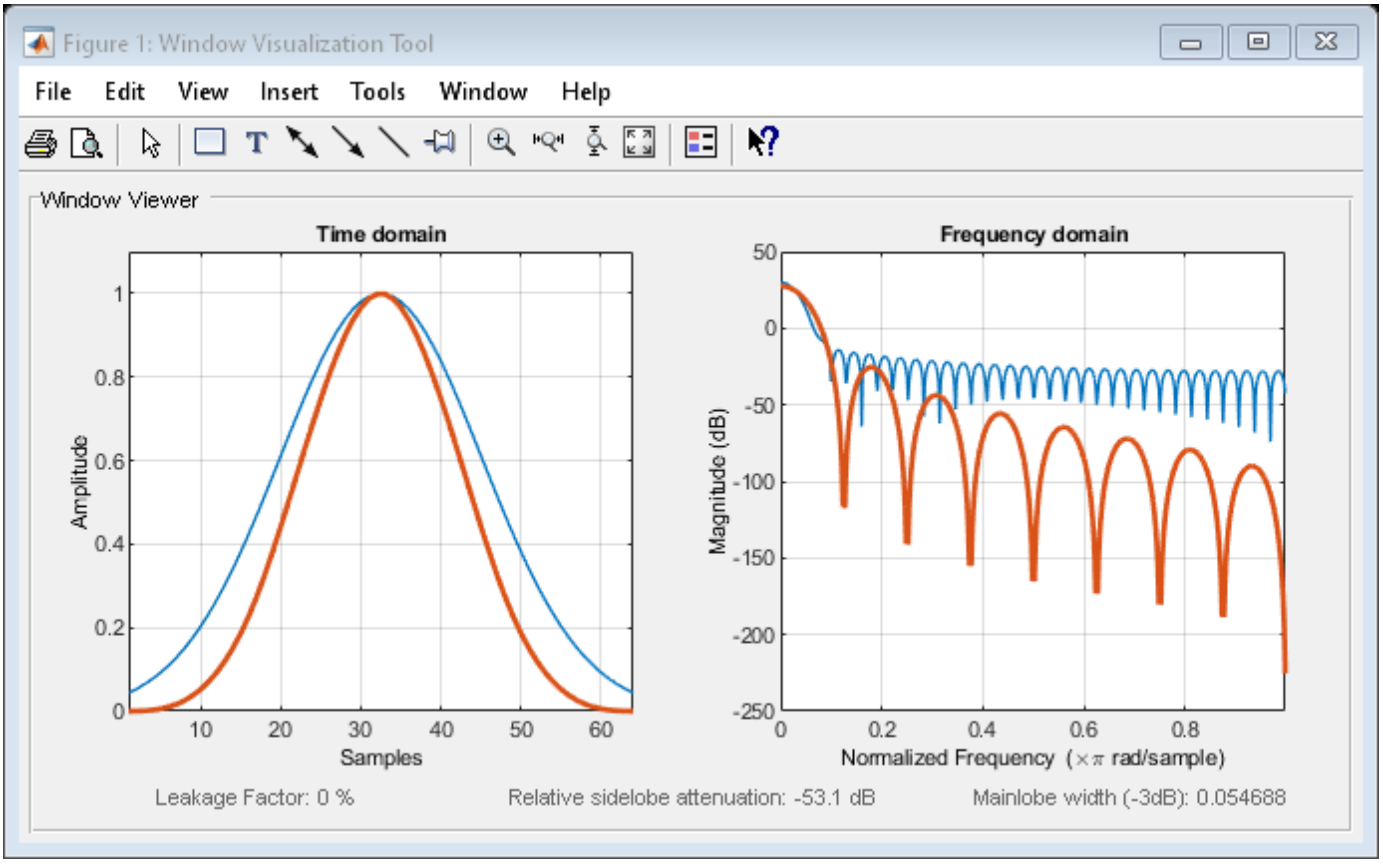
`w = parzenwin(L)` returns the `L`-point Parzen (de la Vallée Poussin) window.

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)
```



Input Arguments

L — Window length

real positive scalar

Window length, specified as a real positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

w — Parzen window

column vector

Parzen window, returned as a column vector of length L. For the equation that defines the Parzen window, see “Algorithms” on page 1-1522.

Algorithms

Parzen windows are piecewise-cubic approximations of Gaussian windows. Parzen window sidelobes fall off as $1/\omega^4$.

This equation defines the *N*–point Parzen window over the interval $-\frac{(N-1)}{2} \le n \le \frac{(N-1)}{2}$:

$$w(n) = \begin{cases} 1 - 6\left(\frac{|n|}{N/2}\right)^2 + 6\left(\frac{|n|}{N/2}\right)^3 & 0 \le |n| \le (N-1)/4 \\ 2\left(1 - \frac{|n|}{N/2}\right)^3 & (N-1)/4 < |n| \le (N-1)/2 \end{cases}$$

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps
Window Designer

Functions
`barthannwin` | `bartlett` | `blackmanharris` | `bohmanwin` | `nuttallwin` | `rectwin` | `triang` | **WVTool**

Introduced before R2006a

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`. If you omit `nfft`, or specify it as empty, then `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, because otherwise the function interprets it as `nfft`.

`[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, because otherwise the function interprets it as `nfft`. 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` × 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.

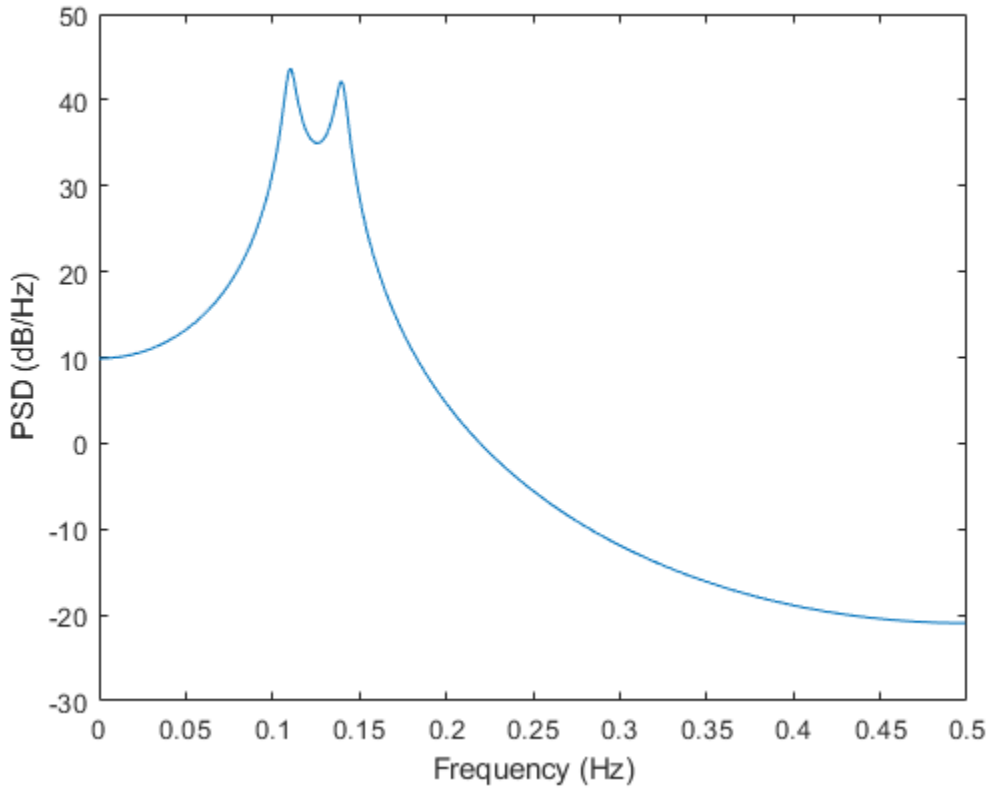
Examples

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.

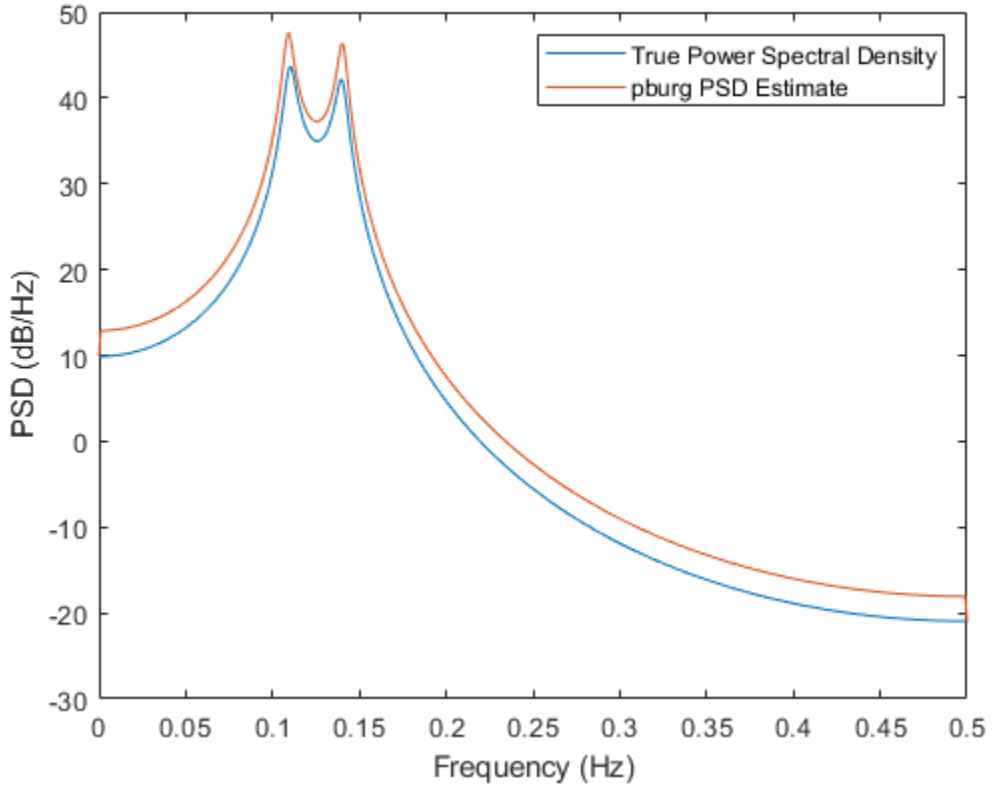
Create an AR(4) system function. Obtain the frequency response and plot the PSD of the system.

```
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')
```

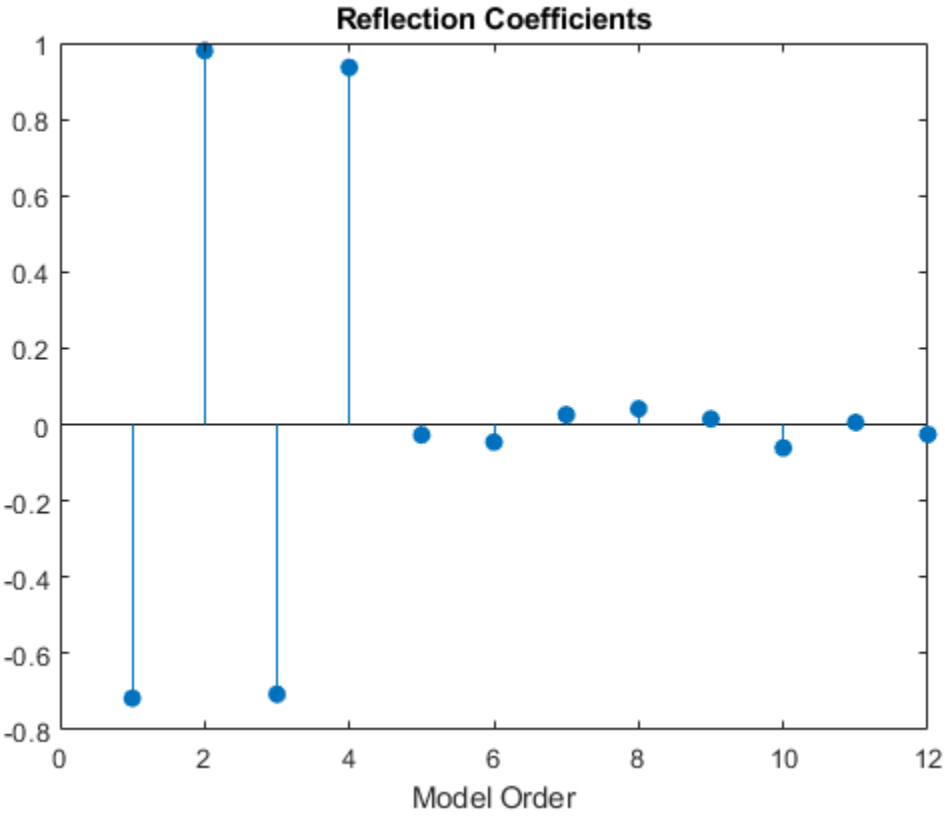


Reflection Coefficients for Model Order Determination

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.

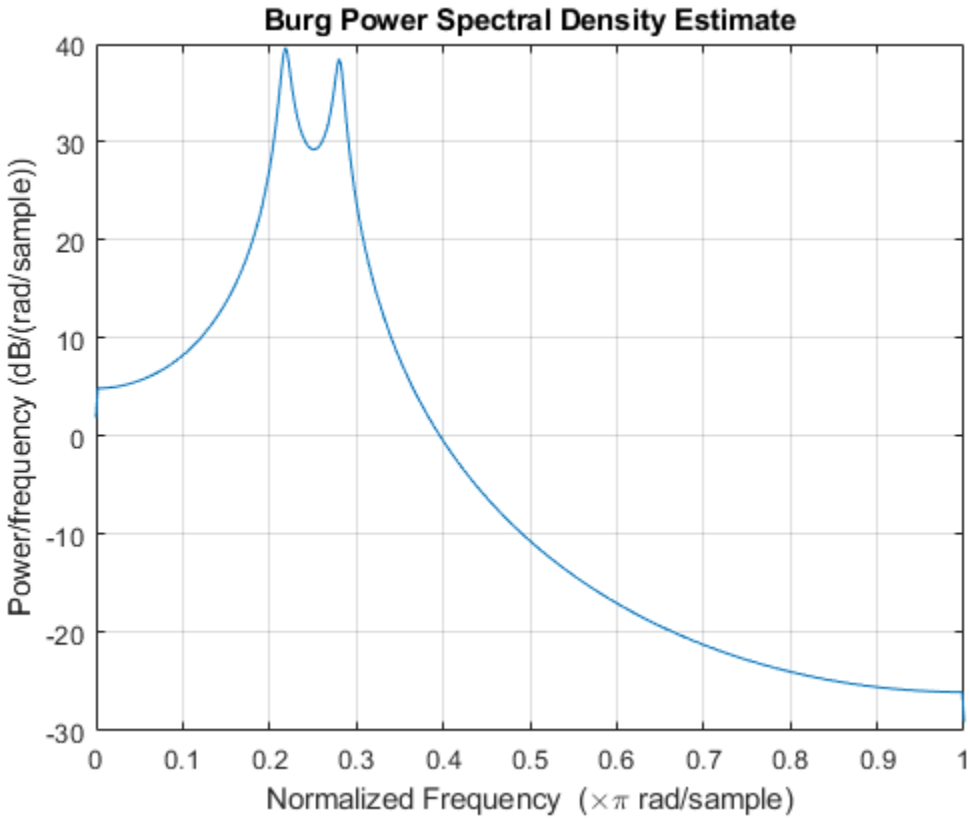
```
A = [1 -2.7607 3.8106 -2.6535 0.9238];

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))
```



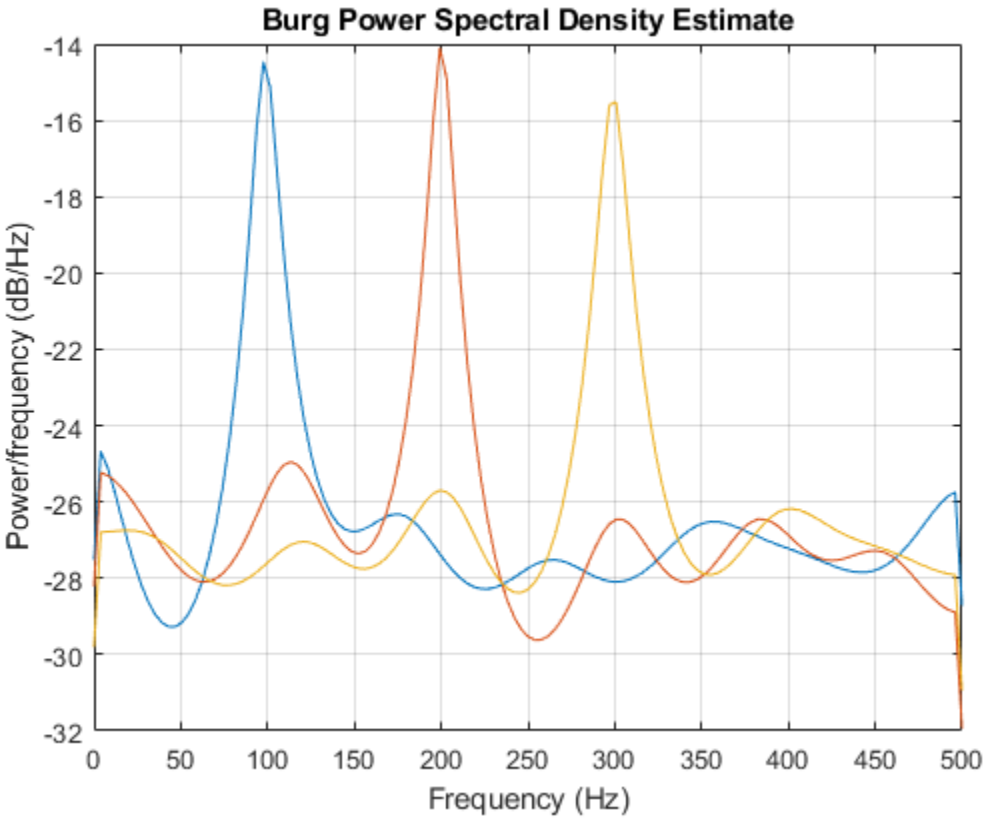
Burg PSD Estimate of a Multichannel Signal

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 — 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 has units of Hz.

w — Normalized frequencies

vector

Normalized frequencies, specified as a row or column vector with at least two elements. Normalized frequencies are in rad/sample.

Example: w = [pi/4 pi/2]

Data Types: double

f — Frequencies

vector

Frequencies, specified as a row or column vector with at least two elements. The frequencies are in cycles per unit time. The unit time is specified by the sample rate, 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,π] rad/sample. If nfft is odd, the length of pxx is (nfft + 1)/2 and the interval is [0,π) 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.

  The function multiplies the power by 2 at all frequencies except 0 and the Nyquist frequency to conserve the total power.
- '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 [0,2π) 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 (–π,π] rad/sample for even length nfft and (–π,π) 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.

`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` × 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 Ω and specify the sample rate 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,π] if `nfft` is even and [0,π) if `nfft` is odd. If `pxx` is a two-sided PSD estimate, `w` spans the interval [0,2π). For a DC-centered PSD estimate, `w` spans the interval (–π,π] for even `nfft` and (–π,π) for odd `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` | `single`

`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`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If the `nfft` argument is variable-size at compile time, then it must not become a scalar or an empty array at runtime.

See Also

`pcov` | `pmcov` | `pyulear`

Introduced before R2006a

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`. If you omit `nfft`, or specify it as empty, then `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, π] when `nfft` is even and [0,π) when `nfft` is odd. For complex–valued signals, `w` always spans the interval [0,2π].

`[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, because otherwise the function interprets it as `nfft`.

`[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, because otherwise the

function interprets it as `nfft`. 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.

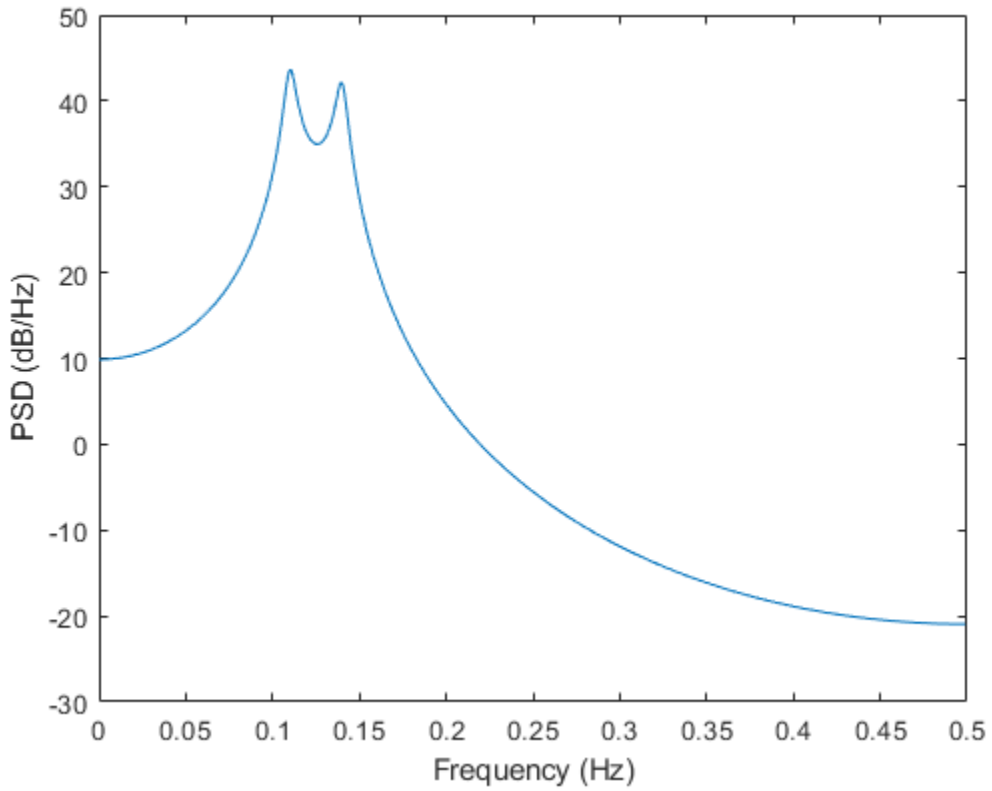
Examples

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.

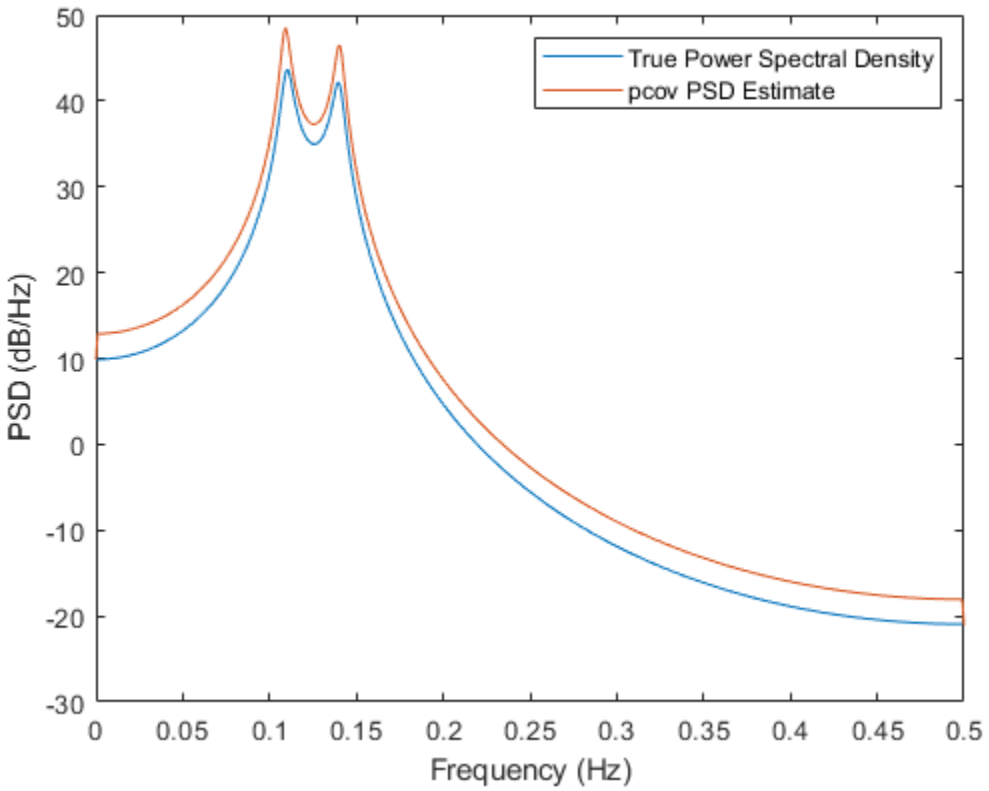
Create an AR(4) system function. Obtain the frequency response and plot the PSD of the system.

```
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')
```



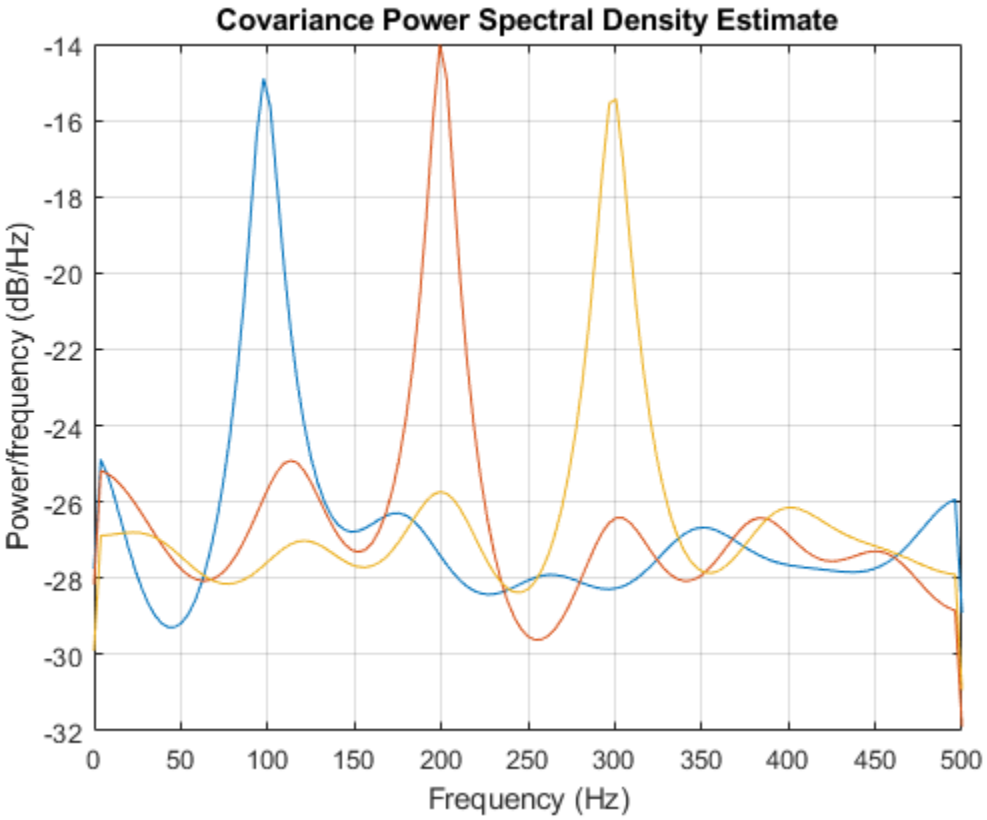
Covariance-Method PSD Estimate of a Multichannel Signal

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 — 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 has units of Hz.

w — Normalized frequencies

vector

Normalized frequencies, specified as a row or column vector with at least two elements. Normalized frequencies are in rad/sample.

Example: w = [pi/4 pi/2]

Data Types: double

f — Frequencies

vector

Frequencies, specified as a row or column vector with at least two elements. The frequencies are in cycles per unit time. The unit time is specified by the sample rate, 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,π] rad/sample. If nfft is odd, the length of pxx is (nfft + 1)/2 and the interval is [0,π) 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.

  The function multiplies the power by 2 at all frequencies except 0 and the Nyquist frequency to conserve the total power.
- '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 [0,2π) 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 (–π,π] rad/sample for even length nfft and (–π,π) 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.

`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` × 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 Ω and specify the sample rate 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,π] if `nfft` is even and [0,π) if `nfft` is odd. If `pxx` is a two-sided PSD estimate, `w` spans the interval [0,2π). For a DC-centered PSD estimate, `w` spans the interval (–π,π] for even `nfft` and (–π,π) for odd `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` | `single`

`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`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If the `nfft` argument is variable-size at compile time, then it must not become a scalar or an empty array at runtime.

See Also

`pburg` | `pmcov` | `pyulear`

Introduced before R2006a

peak2peak

Maximum-to-minimum difference

Syntax

```
y = peak2peak(x)
y = peak2peak(x,dim)
```

Description

`y = peak2peak(x)` returns the difference between the maximum and minimum values in `x`.

`y = peak2peak(x,dim)` computes the maximum-to-minimum differences of `x` along dimension `dim`.

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)
```

```
y = 2
```

Peak-to-Peak Difference of Complex Exponential

Create a complex exponential with a frequency of $\pi/4$ rad/sample. Find the peak-to-peak difference.

```
n = 0:99;
x = exp(1j*pi/4*n);

y = peak2peak(x)
```

```
y = 0.0000e+00 + 1.1034e-15i
```

Peak-to-Peak Differences of 2-D Matrix

Create a matrix in which 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)'*(1:4);
```

```
y = peak2peak(x)
```

```
y = 1×4

     2     4     6     8
```

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 = (1:4)'*cos(2*pi*100*t);

y = peak2peak(x,2)
```

```
y = 4×1

     2
     4
     6
     8
```

Input Arguments

x — Input array

vector | matrix | *N*-D array | `gpuArray` object

Input signal, specified as a vector, matrix, *N*-D array, or `gpuArray` object. For complex-valued inputs, `peak2peak` identifies the maximum and minimum in complex magnitude. `peak2peak` then subtracts the complex number with the minimum modulus from the complex number with the maximum modulus.

See “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox) and “GPU Support by Release” (Parallel Computing Toolbox) for details on `gpuArray` objects.

Example: `sin(2*pi*(0:255)/4)` specifies a sinusoid as a row vector.

Example: `sin(2*pi*[0.1;0.3]*(0:39))'` specifies a two-channel sinusoid.

Data Types: `double` | `single`
Complex Number Support: Yes

dim — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. By default, `peak2peak` operates along the first array dimension of `x` with size greater than 1. For example, if `x` is a row or column

vector, `y` is a real-valued scalar. If `x` 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`.

Data Types: `double` | `single`

Output Arguments

y — Maximum-to-minimum difference

scalar | vector | matrix | *N*-D array | `gpuArray` object

Maximum-to-minimum difference, returned as a real-valued scalar, vector, matrix, *N*-D array, or `gpuArray` object.

References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Thread-Based Environment

Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

This function fully supports thread-based environments. For more information, see “Run MATLAB Functions in Thread-Based Environment”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`min` | `max` | `peak2rms` | `rms` | `rssq`

Introduced in R2012a

peak2rms

Peak-magnitude-to-RMS ratio

Syntax

```
y = peak2rms(x)
y = peak2rms(x,dim)
```

Description

`y = peak2rms(x)` returns the ratio of the largest absolute value in `x` to the root-mean-square (RMS) value of `x`.

`y = peak2rms(x,dim)` computes the peak-magnitude-to-RMS ratio of `x` along dimension `dim`.

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)
```

```
y = 1.4142
```

Peak-Magnitude-to-RMS Ratio of Complex Exponential

Create a complex exponential with a frequency of $\pi/4$ rad/sample. Find the peak-magnitude-to-RMS ratio.

```
n = 0:99;
x = exp(1j*pi/4*n);

y = peak2rms(x)
```

```
y = 1
```

Peak-Magnitude-to-RMS Ratios of 2-D Matrix

Create a matrix in which 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 ratios of the columns.

```
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)'*(1:4);

y = peak2rms(x)
```

```
y = 1×4

    1.4142    1.4142    1.4142    1.4142
```

Peak-Magnitude-to-RMS Ratios of 2-D Matrix Along Specified Dimension

Create a matrix in which 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 = (1:4)'*cos(2*pi*100*t);

y = peak2rms(x,2)
```

```
y = 4×1

    1.4142
    1.4142
    1.4142
    1.4142
```

Input Arguments

`x` — Input array

vector | matrix | *N*-D array | `gpuArray` object

Input signal, specified as a vector, matrix, *N*-D array, or `gpuArray` object.

See “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox) and “GPU Support by Release” (Parallel Computing Toolbox) for details on `gpuArray` objects.

Data Types: `double` | `single`
Complex Number Support: Yes

`dim` — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. By default, `peak2rms` operates along the first array dimension of `x` with size greater than 1. For example, if `x` is a row or column vector, `y` is a real-valued scalar. If `x` 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`.

Data Types: `double` | `single`

Output Arguments

y — Peak-magnitude-to-RMS-ratio

scalar | matrix | *N*-D array | `gpuArray` object

Peak-magnitude-to-RMS ratio, specified as a real-valued scalar, matrix, *N*-D array, or `gpuArray` object.

More About

Peak-Magnitude-to-RMS Ratio

The peak-magnitude-to-RMS ratio is

$$\frac{\|X\|_\infty}{\sqrt{\frac{1}{N}\sum_{n=1}^{N}|X_n|^2}},$$

where the 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.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Thread-Based Environment

Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

This function fully supports thread-based environments. For more information, see “Run MATLAB Functions in Thread-Based Environment”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`min` | `max` | `mean` | `peak2peak` | `rms` | `rssq`

Introduced in R2012a

peig

Pseudospectrum using eigenvector method

Syntax

```
[S,wo] = peig(x,p)
[S,wo] = peig(x,p,wi)
[S,wo] = peig(___,nfft)
[S,wo] = peig(___,'corr')

[S,fo] = peig(x,p,nfft,fs)
[S,fo] = peig(x,p,fi,fs)
[S,fo] = peig(x,p,nfft,fs,nwin,noverlap)

[___] = peig(___,freqrange)
[___,v,e] = peig(___)
peig(___)
```

Description

`[S,wo] = peig(x,p)` implements the eigenvector spectral estimation method and returns `S`, the pseudospectrum estimate of the input signal `x`, and a vector `wo` 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`. You can specify the signal subspace dimension using the input argument `p`.

`[S,wo] = peig(x,p,wi)` returns the pseudospectrum computed at the normalized frequencies specified in vector `wi`. The vector `wi` must have two or more elements, because otherwise the function interprets it as `nfft`.

`[S,wo] = peig(___,nfft)` specifies the integer length of the FFT, `nfft`, to use to estimate the pseudospectrum. This syntax can include any combination of input arguments from previous syntaxes.

`[S,wo] = peig(___,'corr')` forces the input argument `x` to be interpreted as a correlation matrix rather than a matrix of signal data. For this syntax, `x` must be a square matrix, and all of its eigenvalues must be nonnegative.

`[S,fo] = peig(x,p,nfft,fs)` returns the pseudospectrum computed at the frequencies specified in vector `fo` (in Hz). Supply the sample rate `fs` in Hz.

`[S,fo] = peig(x,p,fi,fs)` returns the pseudospectrum computed at the frequencies specified in the vector `fi`. The vector `fi` must have two or more elements, because otherwise the function interprets it as `nfft`.

`[S,fo] = peig(x,p,nfft,fs,nwin,noverlap)` returns the pseudospectrum `S` by segmenting the input data `x` using the window `nwin` and overlap length `noverlap`.

`[___] = peig(___,freqrange)` specifies the range of frequency values to include in `fo` or `wo`.

`[___,v,e] = peig(___)` returns the matrix `v` of noise eigenvectors, along with the associated eigenvalues in the vector `e`.

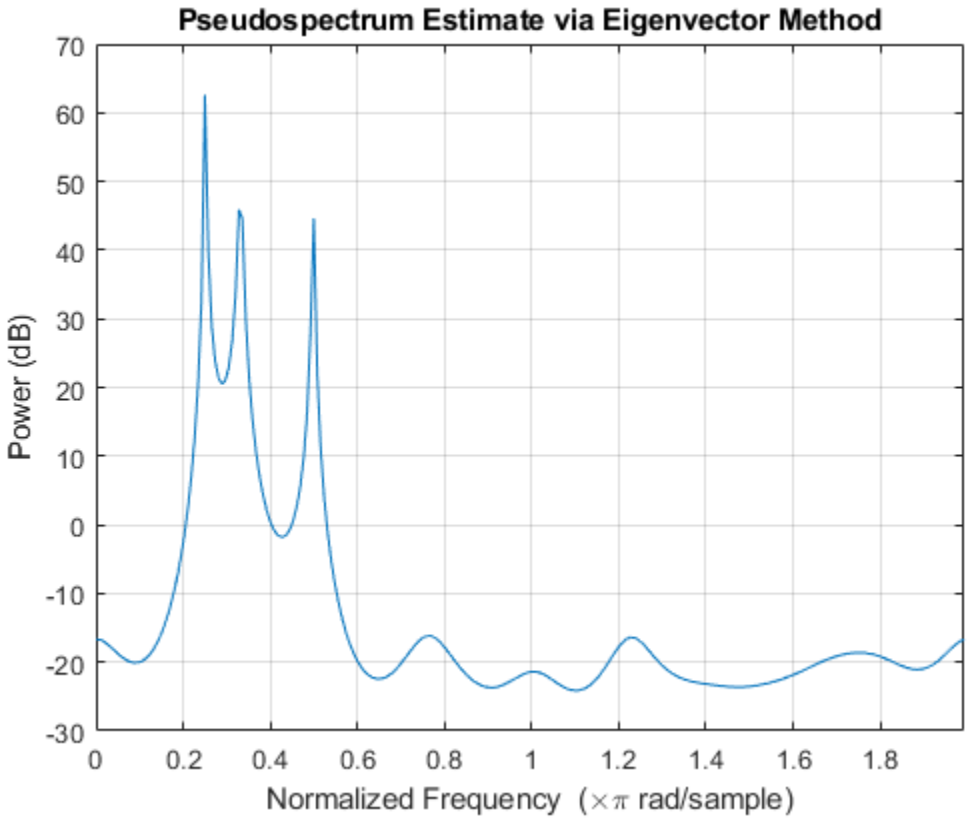
`peig(___)` with no output arguments plots the pseudospectrum in the current figure window.

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 = corrmtx(s,12,'mod');
peig(X,3,'whole')
```



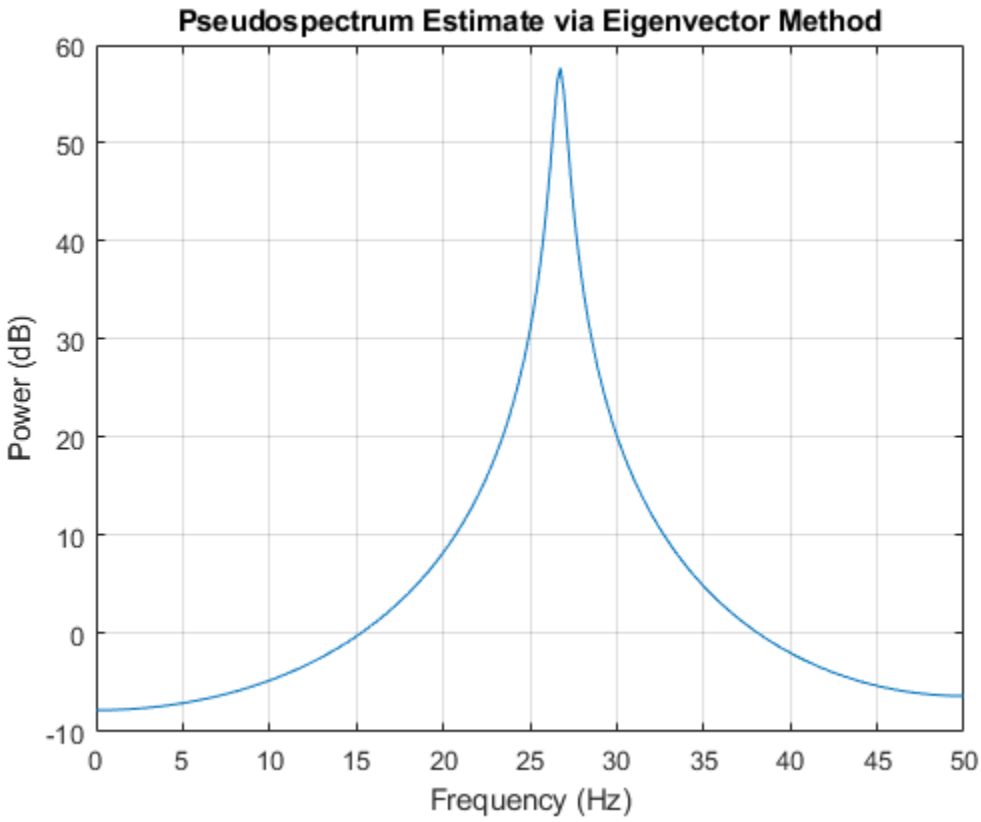
Pseudospectrum of Real Signal

Generate a real signal that consists of the sum of two sinusoids embedded in white Gaussian noise of unit variance. The signal is sampled at 100 Hz for 1 second. The sinusoids have frequencies of 25 Hz and 35 Hz. The lower-frequency sinusoid has twice the amplitude of the other.

```
fs = 100;
t = 0:1/fs:1-1/fs;

s = 2*sin(2*pi*25*t)+sin(2*pi*35*t)+randn(1,100);
```

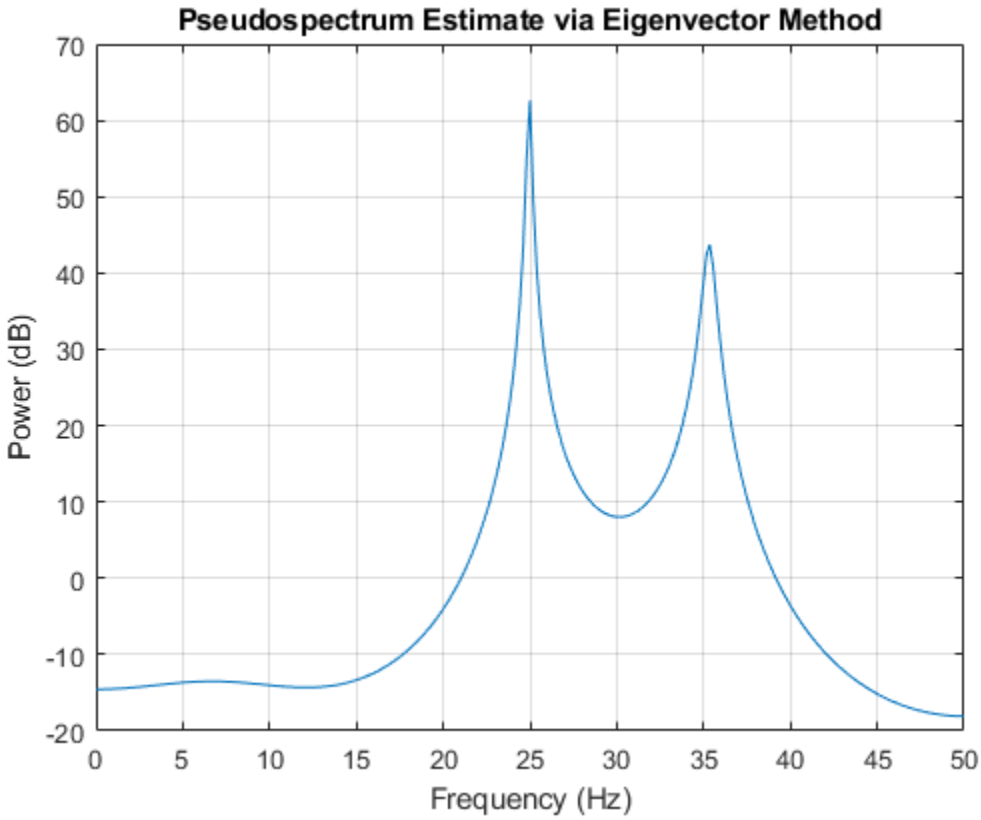
Use the eigenvector method to compute the pseudospectrum of the signal between 0 and the Nyquist frequency. Specify a signal subspace dimension of 2 and a DFT length of 512.

```
peig(s,2,512,fs,'half')
```



It is not possible to resolve the two sinusoids because the signal is real. Repeat the computation using a signal subspace of dimension 4.

```
peig(s,4,512,fs,'half')
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If `x` is a vector, then it is treated as one observation of the signal. If `x` is a matrix, 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 `x`.

p — Subspace dimension

real positive integer | two-element vector

Subspace dimension, specified as a real positive integer or a two-element vector. If `p` is a real positive integer, then it is treated as the subspace dimension. If `p` is a two-element vector, the second element of `p` represents a threshold that is multiplied by λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold λ_{min}`*p(2)` are assigned to the noise subspace. In this case, `p(1)` specifies the maximum dimension of the signal subspace. The extra threshold parameter in the second entry in `p` provides you more flexibility and control in assigning the noise and signal subspaces.

Note If the inputs to `peig` are real sinusoids, set the value of `p` to two times the number of sinusoids. If the inputs are complex sinusoids, set `p` equal to the number of sinusoids.

`wi` — Input normalized frequencies

vector

Input normalized frequencies, specified as a vector.

Data Types: `double`

`nfft` — Number of DFT points

`256` (default) | integer | []

Number of DFT points, specified as a positive integer. If `nfft` is specified as empty, the default `nfft` is used.

`fs` — Sample rate

`1` (default) | positive scalar | []

Sample rate, specified as a positive scalar in Hz. in Hz. If you specify `fs` with the empty vector [], the sample rate defaults to `1` Hz.

`fi` — Input frequency

vector

Input frequencies, specified as a vector. The pseudospectrum is computed at the frequencies specified in the vector.

`nwin` — Length of rectangular window

`2*p(1)` (default) | nonnegative integer

Length of rectangular window, specified as a nonnegative integer.

`noverlap` — Number of overlapped samples

`nwin-1` (default) | nonnegative integer

Number of overlapped samples, specified as a nonnegative integer smaller than the length of window.

Note The arguments `nwin` and `noverlap` are ignored when you include `'corr'` in the syntax.

`freqrange` — Frequency range of pseudospectrum estimates

`'half'` | `'whole'` | `'centered'`

Frequency range of pseudospectrum estimates,specified as one of `'half'`, `whole`, or `'centered'`.

- `'half'` — Returns half the spectrum for a real input signal `x`. If `nfft` is even, then `S` has length `nfft`/2 + 1 and is computed over the interval [0, π]. If `nfft` is odd, the length of `S` is (`nfft` + 1)/2 and the frequency interval is [0,π). When your specify `fs`, the intervals are [0, `fs`/2) and [0, `fs`/2] for even and odd `nfft`, respectively.
- `'whole'` — Returns the whole spectrum for either real or complex input `x`. In this case, `S` has length `nfft` and is computed over the interval [0, 2π). When you specify `fs`, the frequency interval is [0, `fs`).

- `'centered'` — Returns the centered whole spectrum for either real or complex input `x`. In this case, `S` has length `nfft` and is computed over the interval (–π, π] for even `nfft` and (–π, π) for odd `nfft`. When you specify `fs`, the frequency intervals are (–`fs`/2, `fs`/2] and (–`fs`/2, `fs`/2) for even and odd `nfft`, respectively.

Note You can put the arguments `freqrange` or `'corr'` anywhere in the input argument list after `p`.

Output Arguments

S — Pseudospectrum estimate

vector

Pseudospectrum estimate, returned as a vector. The pseudospectrum is calculated using estimates of the eigenvectors of a correlation matrix associated with the input data `x`.

wo — Output normalized frequencies

vector

Output normalized frequencies, specified as a vector. `S` and `wo` 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 table indicates the length of `S` (and `wo`) and the range of the corresponding normalized frequencies for the first syntax.

S Characteristics for an FFT Length of 256 (Default)

Input Data Type	Length of S and w0	Range of the Corresponding Normalized Frequencies
Real	129	[0, π]
Complex	256	[0, 2π)

If `nfft` is specified, the following table indicates the length of `S` and `wo` and the frequency range for `wo`.

S and Frequency Vector Characteristics

Input Data Type	nfft Even or Odd	Length of S and w	Range of w
Real	Even	`(nfft/2 )+ 1`	[0, π]
Real	Odd	`(nfft + 1)/2`	[0, π)
Complex	Even or odd	`nfft`	[0, 2π)

fo — Output frequency

vector

Output frequency, returned as a vector. The frequency range for `fo` depends on `nfft`, `fs`, and the values of the input `x`. The length of `S` (and `fo`) is the same as in the S and Frequency Vector Characteristics above. The following table indicates the frequency range for `fo` if `nfft` and `fs` are specified.

S and Frequency Vector Characteristics with fs Specified

Input Data Type	nfft Even/Odd	Range of f
Real	Even	[0, `fs`/2]
Real	Odd	[0, `fs`/2)
Complex	Even or odd	[0, `fs`)

Additionally, if `nwin` and `noverlap` are also specified, 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

form of 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 for related information on this syntax.

v — Noise eigenvector
matrix

Noise eigenvectors, returned as a matrix. 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)`.

e — Estimated eigenvalues
vector

Estimated eigenvalues of the correlation matrix, returned as a vector.

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 to estimate the signal's frequency content. If you do not supply the correlation matrix, the eigenvalues and eigenvectors of the signal's correlation matrix are estimated using `svd`. 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

$$P_{\mathrm{ev}}(f) = \frac{1}{\sum_{k=p+1}^{N} \left| \mathrm{v}_k^H e(f) \right|^2 / \lambda_k}$$

where N is the dimension of the eigenvectors and $v_k$is the kth eigenvector of the correlation matrix of the input signal. The integer p is the dimension of the signal subspace, so the eigenvectors v_k used in the sum correspond to the smallest eigenvalues λ_k of the correlation matrix. The eigenvectors used span the noise subspace. The vector $e(f)$ consists of complex exponentials, so the inner product $v_k^H e(f)$ 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If `nfft` or `nwin` is variable-size during code generation, then it must not reduce to a scalar or an empty array at runtime.

See Also

`corrmtx` | `pburg` | `periodogram` | `pmtm` | `pmusic` | `prony` | `pwelch` | `rooteig` | `rootmusic`

Introduced before R2006a

pentropy

Spectral entropy of signal

Syntax

```
se = pentropy(xt)
se = pentropy(x,sampx)

se = pentropy(p,fp,tp)
se = pentropy( ___ ,Name=Value)

[se,t] = pentropy( ___ )

pentropy( ___ )
```

Description

`se = pentropy(xt)` returns the “Spectral Entropy” on page 1-1566 of single-variable, single-column `timetable` `xt` as the `timetable` `se`. `pentropy` computes the spectrogram of `xt` using the default options of `pspectrum`.

`se = pentropy(x,sampx)` returns the spectral entropy of vector `x`, sampled at rate or time interval `sampx`, as a vector.

`se = pentropy(p,fp,tp)` returns the spectral entropy using the power spectrogram `p`, along with spectrogram frequency and time vectors `fp` and `tp`.

Use this syntax when you want to customize the options for `pspectrum`, rather than accept the default `pspectrum` options that `pentropy` applies.

`se = pentropy( ___ ,Name=Value)` specifies additional properties using name-value arguments. Options include instantaneous or whole-signal entropy, scaling by white noise entropy, frequency limits, and time limits. You can use `Name=Value` with any of the input arguments in previous syntaxes.

`[se,t] = pentropy( ___ )` returns the spectral entropy `se` along with the time vector or `timetable` `t`. If `se` is a `timetable`, then `t` is equal to the row times of `timetable` `se`. This syntax does not apply if `Instantaneous` is set to `false`.

`pentropy( ___ )` with no output arguments plots the spectral entropy against time. If `Instantaneous` is set to `false`, the function outputs the scalar value of the spectral entropy.

Examples

Plot Spectral Entropy of Signal

Plot the spectral entropy of a signal expressed as a timetable and as a time series.

Generate a random series with normal distribution (white noise).

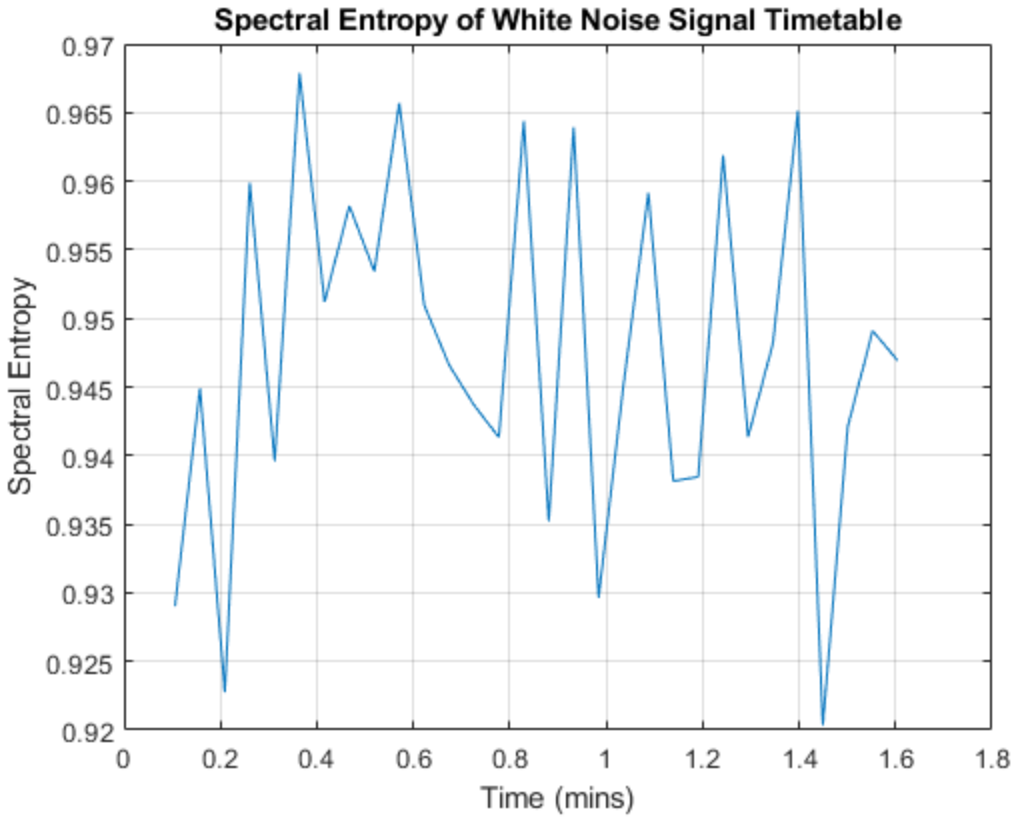
```
xn = randn(1000,1);
```

Create time vector `t` and convert to `duration` vector `tdur`. Combine `tdur` and `xn` in a timetable.

```
fs = 10;
ts = 1/fs;
t = 0.1:ts:100;
tdur = seconds(t);
xt = timetable(tdur',xn);
```

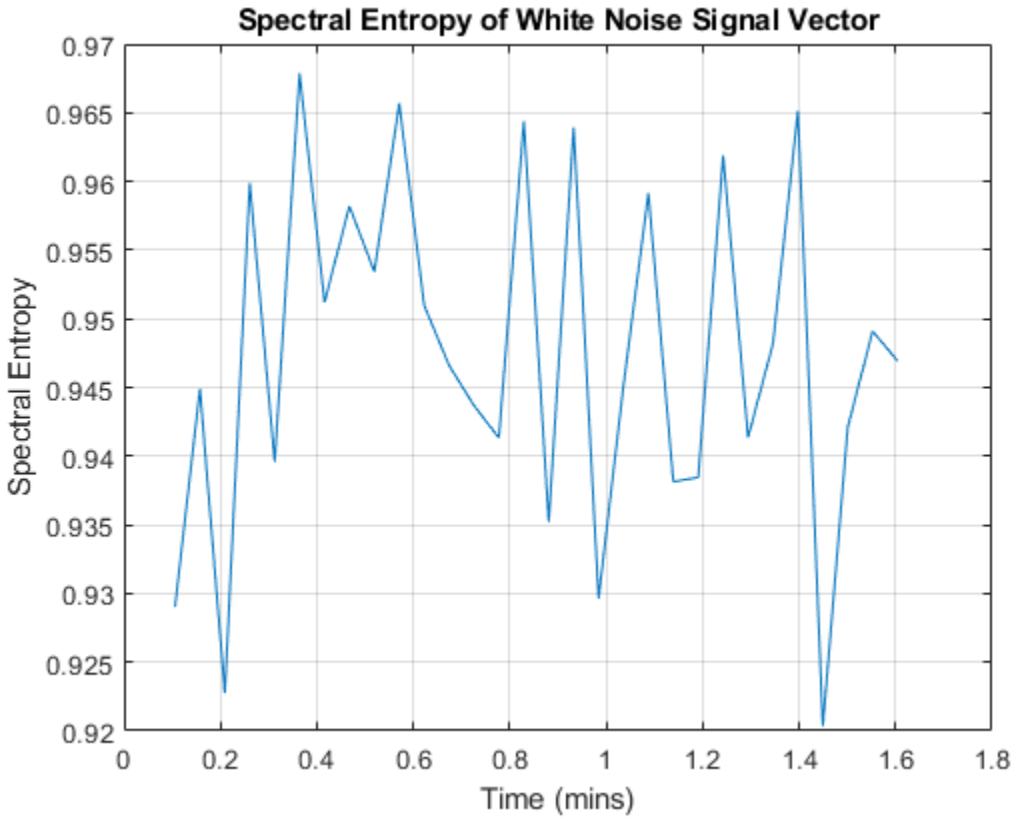
Plot the spectral entropy of the timetable `xt`.

```
pentropy(xt)
title('Spectral Entropy of White Noise Signal Timetable')
```



Plot the spectral entropy of the signal, using time-point vector `t` and the form which returns `se` and associated time `te`. Match the *x*-axis units and grid to the `pentropy`-generated plots for comparison.

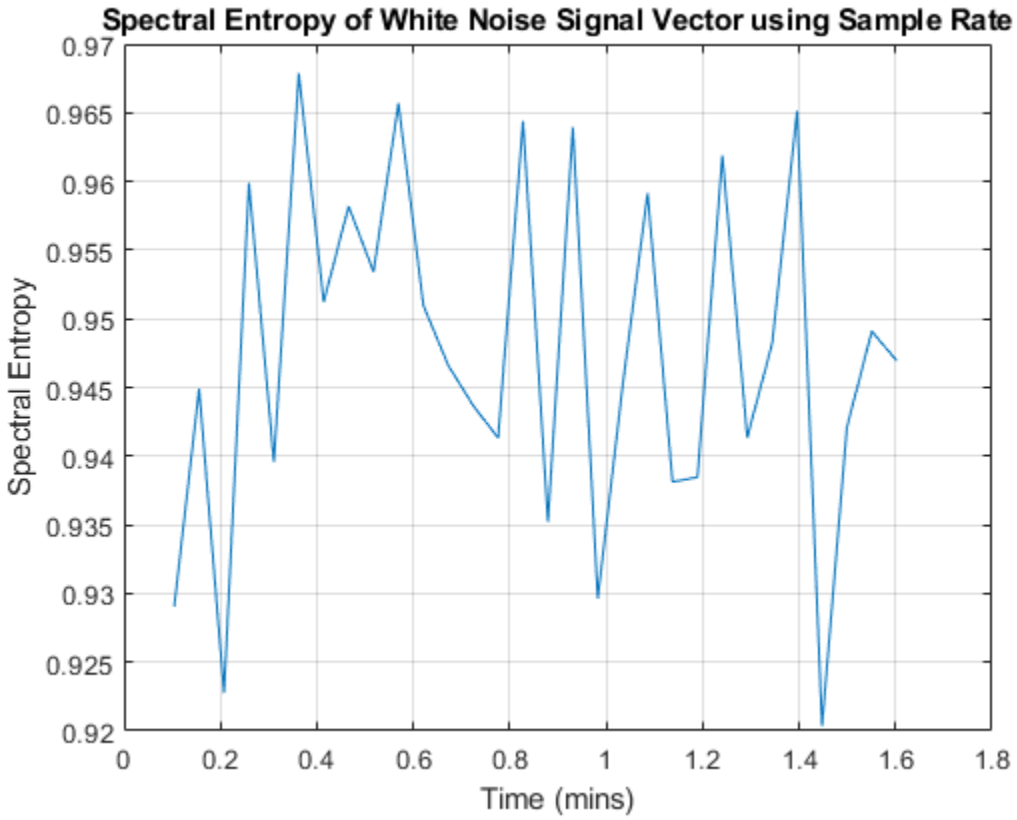
```
[se,te] = pentropy(xn,t');
te_min = te/60;
plot(te_min,se)
title('Spectral Entropy of White Noise Signal Vector')
xlabel('Time (mins)')
ylabel('Spectral Entropy')
grid on
```



Both yield the same result.

The second input argument for `pentropy` can represent either frequency or time. The software interprets according to the data type of the argument. Plot the spectral entropy of the signal, using sample rate scalar `fs` instead of time vector `t`.

```
pentropy(xn,fs)
title('Spectral Entropy of White Noise Signal Vector using Sample Rate')
```



This plot matches the previous plots.

Plot Spectral Entropy of Speech Signal

Plot the spectral entropy of a speech signal and compare it to the original signal. Visualize the spectral entropy on a color map by first creating a power spectrogram, and then taking the spectral entropy of frequency bins within the bandwidth of speech.

Load the data `x`, which contains a two-channel recording of the word "Hello" embedded by low-level white noise. `x` consists of two columns representing the two channels. Use only the first channel.

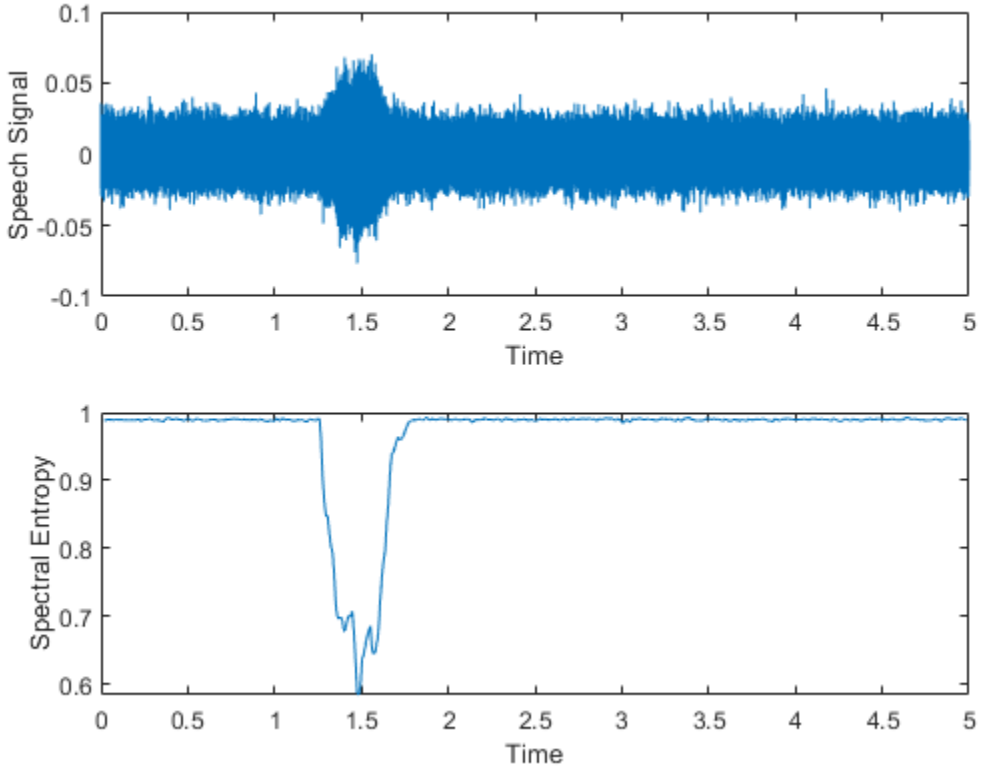
Define the sample rate and the time vector. Augment the first channel of `x` with white noise to achieve a signal-to-noise ratio of about 5 to 1.

```
load Hello x
fs = 44100;
t = 1/fs*(0:length(x)-1);
x1 = x(:,1) + 0.01*randn(length(x),1);
```

Find the spectral entropy. Visualize the data for the original signal and for the spectral entropy.

```
[se,te] = pentropy(x1,fs);

subplot(2,1,1)
```

```
plot(t,x1)
ylabel("Speech Signal")
xlabel("Time")

subplot(2,1,2)
plot(te,se)
ylabel("Spectral Entropy")
xlabel("Time")
```



The spectral entropy drops when "Hello" is spoken. This is because the signal spectrum has changed from almost a constant (white noise) to the distribution of a human voice. The human-voice distribution contains more information and has lower spectral entropy.

Compute the power spectrogram `p` of the original signal, returning frequency vector `fp` and time vector `tp` as well. For this case, specifying a frequency resolution of 20 Hz provides acceptable clarity in the result.

```
[p,fp,tp] = pspectrum(x1,fs,"spectrogram",FrequencyResolution=20);
```

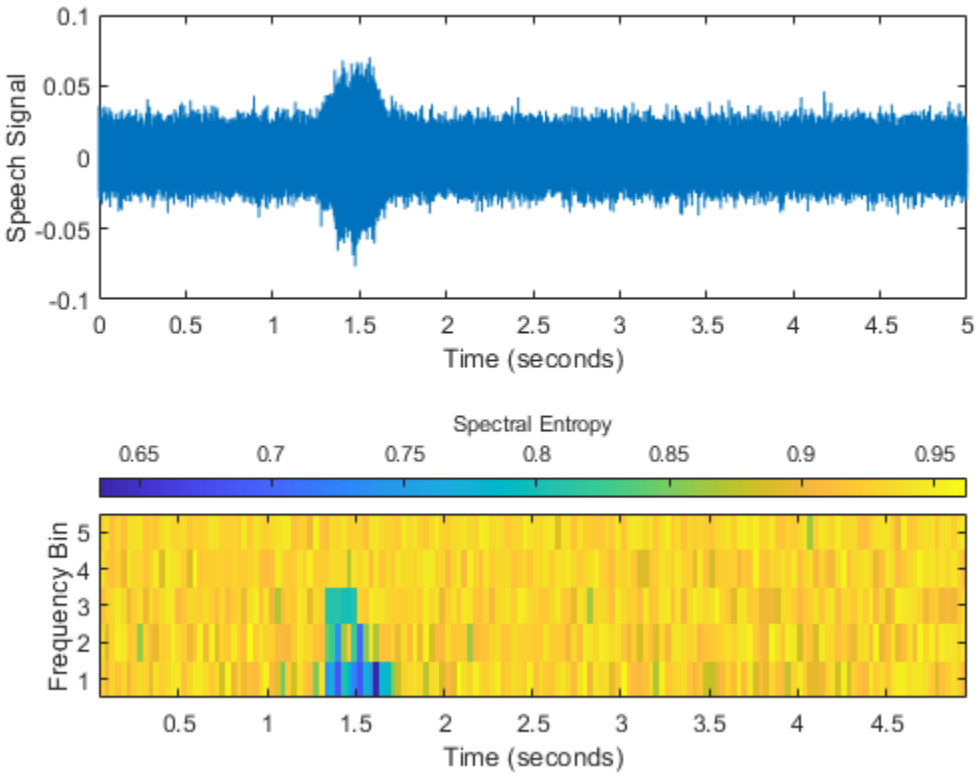
The frequency vector of the power spectrogram goes to 22,050 Hz, but the range of interest with respect to speech is limited to the telephony bandwidth of 300–3400 Hz. Divide the data into five frequency bins by defining start and end points, and compute the spectral entropy for each bin.

```
flow = [300 628 1064 1634 2394];
fup = [627 1060 1633 2393 3400];

se2 = zeros(length(flow),size(p,2));
```

```
for i = 1:length(flow)
    se2(i,:) = pentropy(p,fp,tp,FrequencyLimits=[flow(i) fup(i)]);
end
```

Visualize the data in a color map that shows ascending frequency bins, and compare with the original signal.

```
figure
subplot(2,1,1)
plot(t,x1)
xlabel("Time (seconds)")
ylabel("Speech Signal")

subplot(2,1,2)
imagesc(tp,[],flip(se2))                % Flip se2 so its plot corresponds to the
                                        % ascending frequency bins.
h = colorbar(gca,"NorthOutside");
ylabel(h,"Spectral Entropy")
yticks(1:5)
set(gca,YTickLabel=num2str((5:-1:1).')) % Label the ticks for the ascending bins.
xlabel("Time (seconds)")
ylabel("Frequency Bin")
```

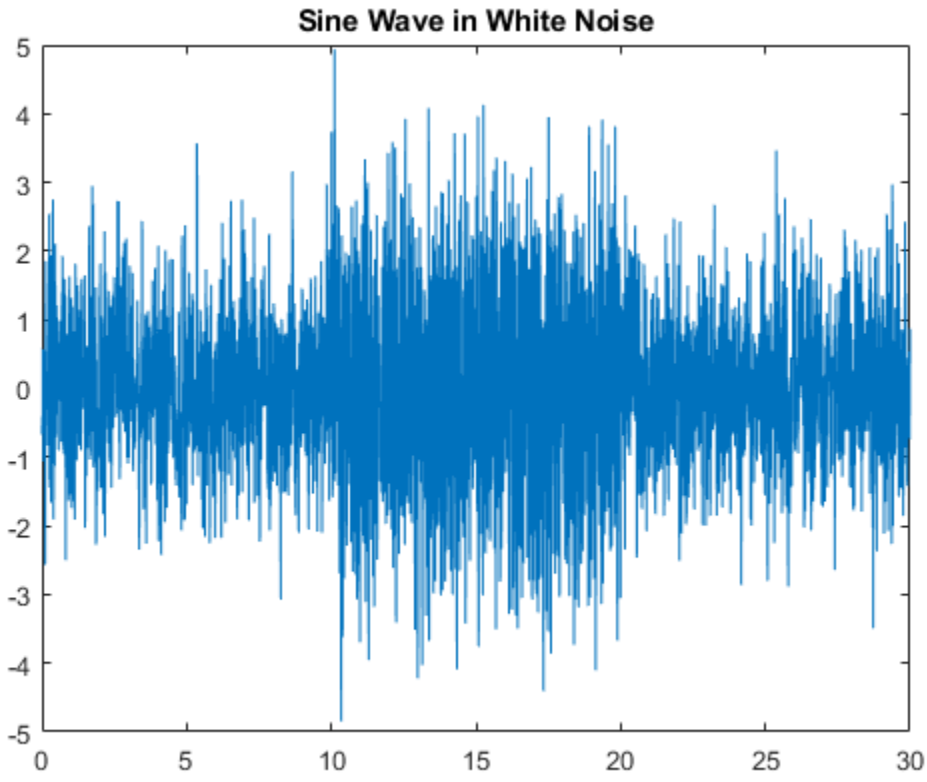


Use Spectral Entropy to Detect Sine Wave in White Noise

Create a signal that combines white noise with a segment that consists of a sine wave. Use spectral entropy to detect the existence and position of the sine wave.

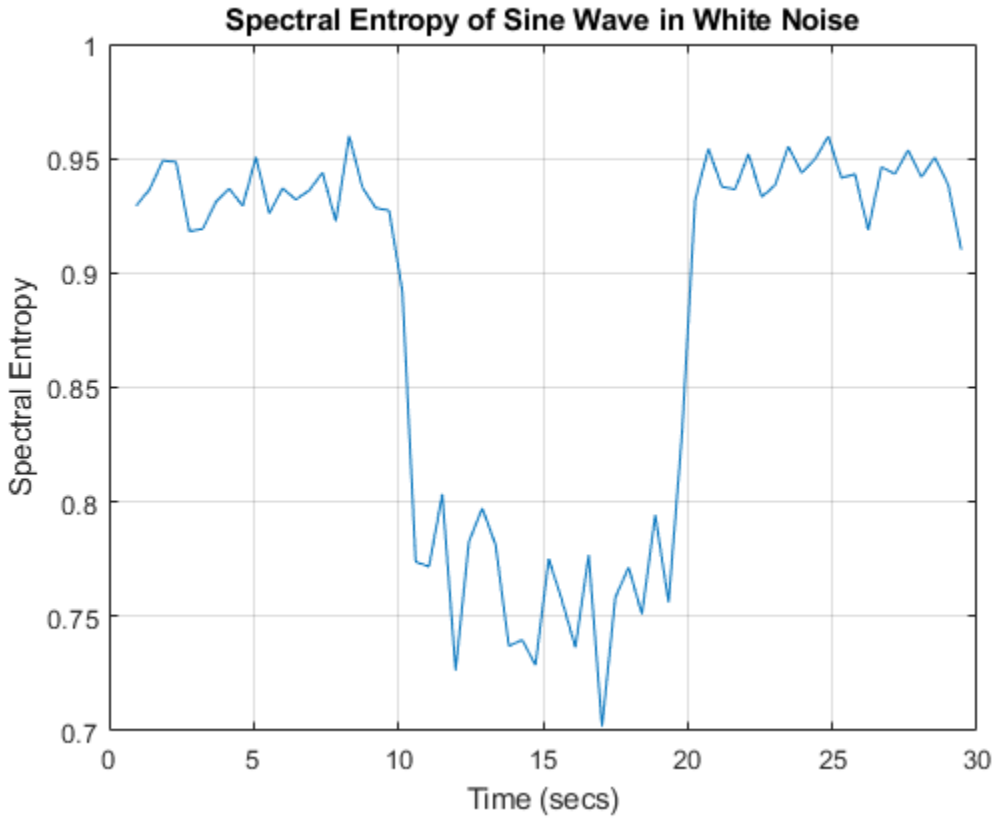
Generate and plot the signal, which contains three segments. The middle segment contains the sine wave along with white noise. The other two segments are pure white noise.

```
fs = 100;
t = 0:1/fs:10;
sin_wave = 2*sin(2*pi*20*t')+randn(length(t),1);
x = [randn(1000,1);sin_wave;randn(1000,1)];
t3 = 0:1/fs:30;

plot(t3,x)
title("Sine Wave in White Noise")
```



Plot the spectral entropy.

```
pentropy(x,fs)
title("Spectral Entropy of Sine Wave in White Noise")
```



The plot clearly differentiates the segment with the sine wave from the white-noise segments. This is because the sine wave contains information. Pure white noise has the highest spectral entropy.

The default for `pentropy` is to return or plot the instantaneous spectral entropy for each time point, as the previous plot displays. You can also distill the spectral entropy information into a single number that represents the entire signal by setting `Instantaneous` to `false`. Use the form that returns the spectral entropy value if you want to directly use the result in other calculations. Otherwise, `pentropy` returns the spectral entropy in `ans`.

```
se = pentropy(x,fs,Instantaneous=false)
```

```
se = 0.9033
```

A single number characterizes the spectral entropy, and therefore the information content, of the signal. You can use this number to efficiently compare this signal with other signals.

Input Arguments

`xt` — Signal timetable

timetable

Signal timetable from which `pentropy` returns the spectral entropy `se`, specified as a `timetable` that contains a single variable with a single column. `xt` must contain increasing, finite row times. If the `xt` `timetable` has missing or duplicate time points, you can fix it using the tips in “Clean

Timetable with Missing, Duplicate, or Nonuniform Times". `xt` can be nonuniformly sampled, with the `pspectrum` constraint that the median time interval and the mean time interval must obey:

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

For an example, see "Plot Spectral Entropy of Signal" on page 1-1556.

x — Time-series signal
vector

Time-series signal from which `pentropy` returns the spectral entropy `se`, specified as a vector.

sampx — Sample rate or sample time of signal
normalized frequency (default) | positive numeric scalar | `duration` scalar | numeric vector in seconds | `duration` array | `datetime` array

Sample rate or sample time, specified as one of the following:

- Positive numeric scalar — Sample rate in hertz
- `duration` scalar — Time interval between consecutive samples of `X`
- Vector, `duration` array, or `datetime` array — Time instant or duration corresponding to each element of `x`

When `sampx` represents a time vector, time samples can be nonuniform, with the `pspectrum` constraint that the median time interval and the mean time interval must obey:

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

For an example, see "Plot Spectral Entropy of Signal" on page 1-1556.

p — Power spectrogram or spectrum of signal
real nonnegative matrix

Power spectrogram or spectrum of a signal, specified as a matrix (spectrogram) or a column vector (spectrum). If you specify `p`, then `pentropy` uses `p` rather than generate its own spectrogram or power spectrogram. `fp` and `tp`, which provide the frequency and time information, must accompany `p`. Each element of `p` at the *i*'th row and the *j*'th column represents the signal power at the frequency bin centered at `fp`(*i*) and the time instance `tp`(*j*).

For an example, see "Plot Spectral Entropy of Speech Signal" on page 1-1559.

fp — Frequencies for spectrogram p
vector

Frequencies for spectrogram or power spectrogram `p` when `p` is supplied explicitly to `pentropy`, specified as a vector in hertz. The length of `fp` must be equal to the number of rows in `s`.

tp — Time information for spectrogram p
vector | `duration` array | `datetime` array

Time information for power spectrogram or spectrum `p` when `p` is supplied explicitly to `pentropy`, specified as one of the following:

- Vector of time points, whose data type can be numeric, `duration`, or `datetime`. The length of vector `tp` must be equal to the number of columns in `p`.
- `duration` scalar that represents the time interval in `p`. The scalar form of `tp` can be used only when `p` is a power spectrogram matrix.
- For the special case where `p` is a column vector (power spectrum), `tp` can be a numeric, `duration`, or `datetime` scalar representing the time point of the spectrum.

For the special case where `p` is a column vector (power spectrum), `tp` can be a single/double/`duration`/`datetime` scalar representing the time point of the spectrum.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `"Instantaneous",false,"FrequencyLimits",[25 50]` computes the scalar spectral entropy representing the portion of the signal ranging from 25 Hz to 50 Hz.

`Instantaneous` — Instantaneous time series option

`true` (default) | `false`

Instantaneous time series option, specified as a logical.

- If `Instantaneous` is `true`, then `pentropy` returns the instantaneous spectral entropy as a time-series vector.
- If `Instantaneous` is `false`, then `pentropy` returns the spectral entropy value of the whole signal or spectrum as a scalar.

For an example, see “Use Spectral Entropy to Detect Sine Wave in White Noise” on page 1-1561.

`Scaled` — Scale by white noise option

`true` (default) | `false`

Scale by white noise option, specified as a logical. Scaling by white noise — or $\log_2 n$, where n is the number of frequency points — is equivalent to normalizing in “Spectral Entropy” on page 1-1566. It allows you to perform a direct comparison on signals of different length.

- If `Scaled` is `true`, then `pentropy` returns the spectral entropy scaled by the spectral entropy of the corresponding white noise.
- If `Scaled` is `false`, then `pentropy` does not scale the spectral entropy.

`FrequencyLimits` — Frequency limits

[0 `sampfreq`/2] (default) | [f1 f2]

Frequency limits to use, specified as a two-element vector containing lower and upper bounds f1 and f2 in hertz. The default is [0 `sampfreq`/2], where `sampfreq` is the sample rate in hertz that `pentropy` derives from `sampx`.

This specification allows you to exclude a band of data at either end of the spectral range.

For an example, see “Plot Spectral Entropy of Speech Signal” on page 1-1559.

`TimeLimits` — Time Limits
full timespan (default) | [t1 t2]

Time limits, specified as a two-element vector containing lower and upper bounds t1 and t2 in the same units as the sample time provided in `sampx`, and of the data types:

- Numeric or `duration` when `sampx` is numeric or duration
- Numeric, `duration`, or `datetime` when `sampx` is `datetime`

This specification allows you to extract a time segment of data from the full timespan.

Output Arguments

`se` — Spectral entropy
timetable | double vector

"Spectral Entropy" on page 1-1566, returned as a `timetable` if the input signal is `timetable` `xt`, and as a double vector if the input signal is time series `x`.

`t` — time values corresponding to `se`
timetable | double vector

Time values associated with `se`, returned in the same form as the time in `se`. This argument does not apply if `Instantaneous` is set to `false`.

For an example, see "Plot Spectral Entropy of Signal" on page 1-1556.

More About

Spectral Entropy

The spectral entropy (SE) of a signal is a measure of its spectral power distribution. The concept is based on the Shannon entropy, or information entropy, in information theory. The SE treats the signal's normalized power distribution in the frequency domain as a probability distribution, and calculates the Shannon entropy of it. The Shannon entropy in this context is the spectral entropy of the signal. This property can be useful for feature extraction in fault detection and diagnosis [2], [1]. SE is also widely used as a feature in speech recognition [3] and biomedical signal processing [4].

The equations for spectral entropy arise from the equations for the power spectrum and probability distribution for a signal. For a signal $x(n)$, the power spectrum is $S(m) = |X(m)|^2$, where $X(m)$ is the discrete Fourier transform of $x(n)$. The probability distribution $P(m)$ is then:

$$P(m) = \frac{S(m)}{\sum_i S(i)}.$$

The spectral entropy H follows as:

$$H = -\sum_{m=1}^{N} P(m)\log_2 P(m).$$

Normalizing:

$$H_n = -\frac{\sum_{m=1}^{N} P(m)\log_2 P(m)}{\log_2 N},$$

where N is the total frequency points. The denominator, $\log_2 N$ represents the maximal spectral entropy of white noise, uniformly distributed in the frequency domain.

If a time-frequency power spectrogram $S(t,f)$ is known, then the probability distribution becomes:

$$P(m) = \frac{\sum_t S(t,m)}{\sum_f \sum_t S(t,f)}.$$

Spectral entropy is still:

$$H = -\sum_{m=1}^{N} P(m)\log_2 P(m).$$

To compute the instantaneous spectral entropy given a time-frequency power spectrogram $S(t,f)$, the probability distribution at time t is:

$$P(t,m) = \frac{S(t,m)}{\sum_f S(t,f)}.$$

Then the spectral entropy at time t is:

$$H(t) = -\sum_{m=1}^{N} P(t,m)\log_2 P(t,m).$$

References

[1] Pan, Y. N., J. Chen, and X. L. Li. "Spectral Entropy: A Complementary Index for Rolling Element Bearing Performance Degradation Assessment." *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*. Vol. 223, Issue 5, 2009, pp. 1223–1231.

[2] Sharma, V., and A. Parey. "A Review of Gear Fault Diagnosis Using Various Condition Indicators." *Procedia Engineering*. Vol. 144, 2016, pp. 253–263.

[3] Shen, J., J. Hung, and L. Lee. "Robust Entropy-Based Endpoint Detection for Speech Recognition in Noisy Environments." *ICSLP*. Vol. 98, November 1998.

[4] Vakkuri, A., A. Yli-Hankala, P. Talja, S. Mustola, H. Tolvanen-Laakso, T. Sampson, and H. Viertiö-Oja. "Time-Frequency Balanced Spectral Entropy as a Measure of Anesthetic Drug Effect in Central Nervous System during Sevoflurane, Propofol, and Thiopental Anesthesia." *Acta Anaesthesiologica Scandinavica*. Vol. 48, Number 2, 2004, pp. 145–153.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Arguments specified using name-value pairs must be compile-time constants.
- `datetime` and duration arrays are not supported for code generation.
- Timetables are not supported for code generation.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also
`kurtogram` | `pkurtosis` | `pspectrum`

Introduced in R2018a

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)

[ ___ ,pxxc] = periodogram( ___ ,'ConfidenceLevel',probability)

[rpxx,f] = periodogram( ___ ,'reassigned')
[rpxx,f,pxx,fc] = periodogram( ___ ,'reassigned')

[ ___ ] = periodogram( ___ ,spectrumtype)

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 sample rate, `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 $[0,\texttt{fs}/2]$ when `nfft` is even and

[0,fs/2) when nfft is odd. For complex-valued signals, f spans the interval [0,fs). fs must be the fourth input to periodogram. To input a sample rate and still use the default values of the preceding optional arguments, specify these arguments as empty, [].

[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, because otherwise the function interprets it as nfft.

[pxx,f] = periodogram(x,window,f,fs) returns the two-sided periodogram estimates at the frequencies specified in the vector. The vector f must contain at least two elements, because otherwise the function interprets it as nfft. The frequencies in f are in cycles per unit time. The sample rate, 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'.

[___ ,pxxc] = periodogram(___ ,'ConfidenceLevel',probability) returns the probability × 100% confidence intervals for the PSD estimate in pxxc.

[rpxx,f] = periodogram(___ ,'reassigned') reassigns each PSD estimate to the frequency closest to its center of energy. rpxx contains the sum of the estimates reassigned to each element of f.

[rpxx,f,pxx,fc] = periodogram(___ ,'reassigned') also returns the nonreassigned PSD estimates, pxx, and the center-of-energy frequencies, fc. If you use the 'reassigned' flag, then you cannot specify a probability confidence interval.

[___] = periodogram(___ ,spectrumtype) returns the PSD estimate if spectrumtype is specified as 'psd' and returns the power spectrum if spectrumtype is specified as 'power'.

periodogram(___) with no output arguments plots the periodogram PSD estimate in dB per unit frequency in the current figure window.

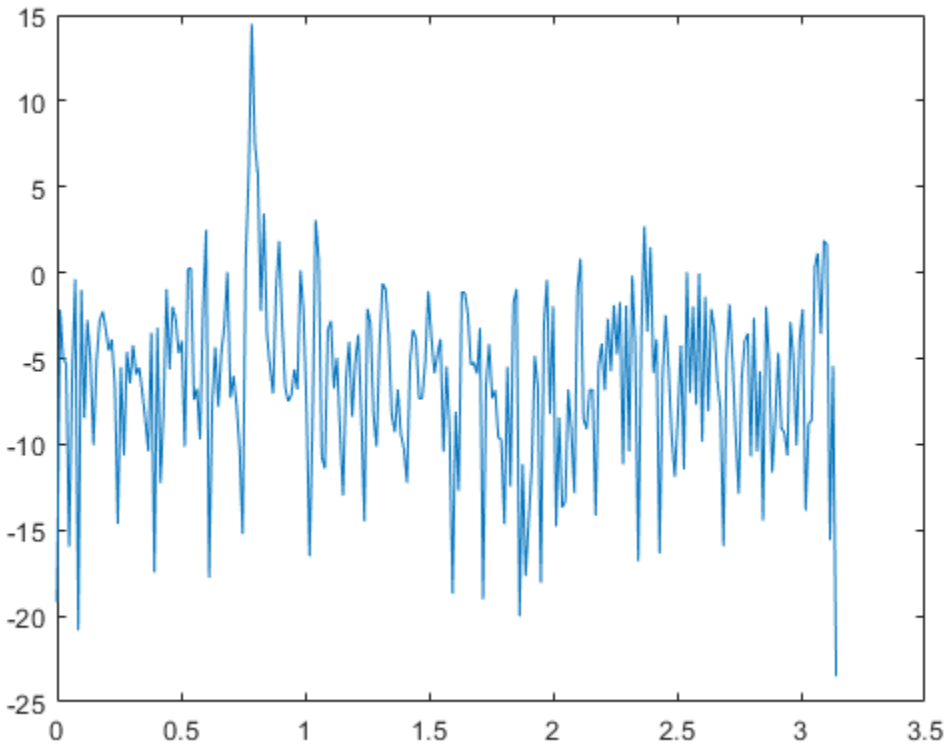
Examples

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.

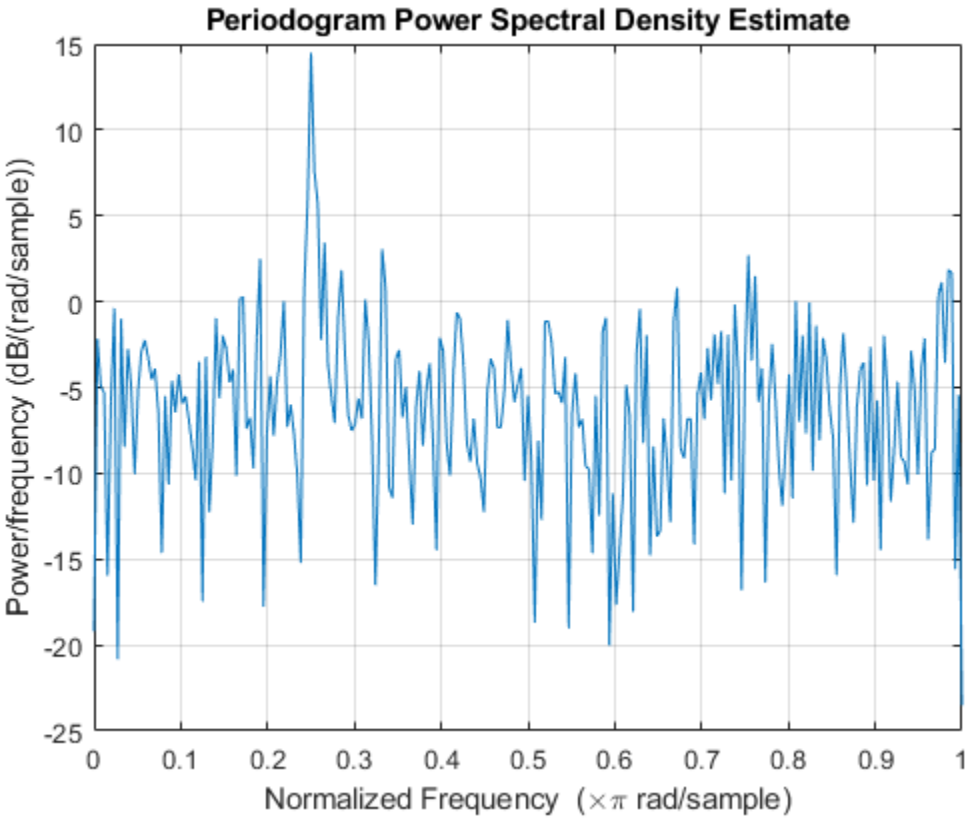
Create a sine wave with an angular frequency of $\pi/4$ rad/sample with additive $N(0, 1)$ white noise. The signal is 320 samples in length. Obtain the periodogram using the default rectangular window and DFT length. The DFT length is the next power of two greater than the signal length, or 512 points. 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)
```

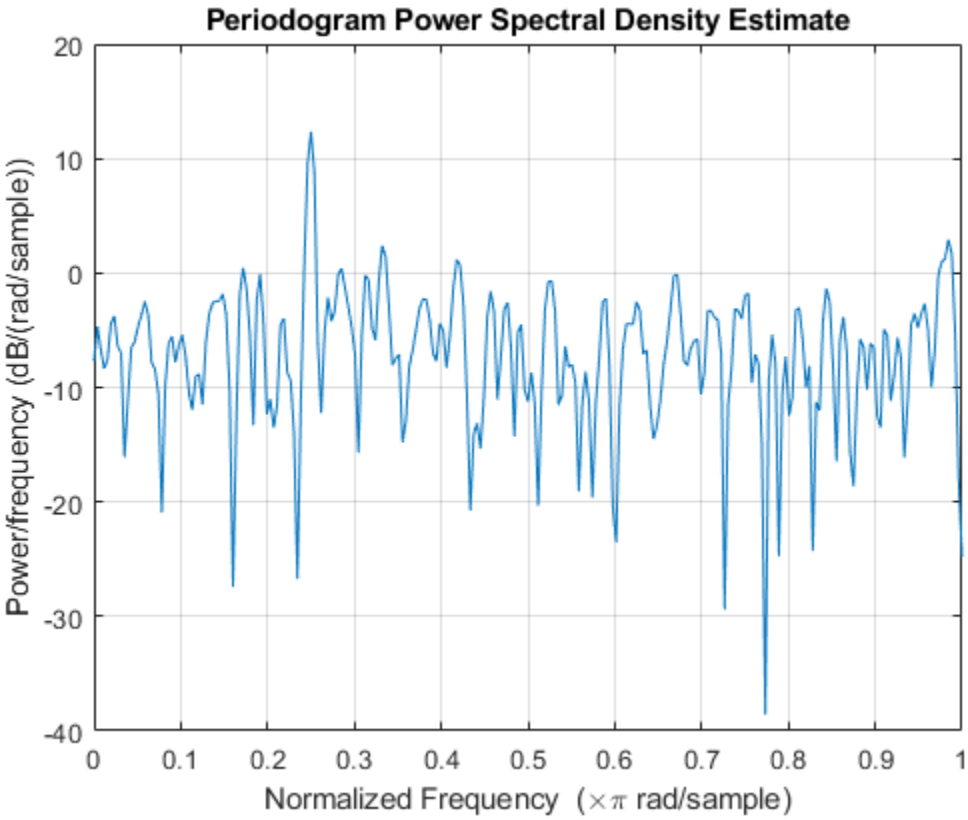


Modified Periodogram with Hamming Window

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)))
```

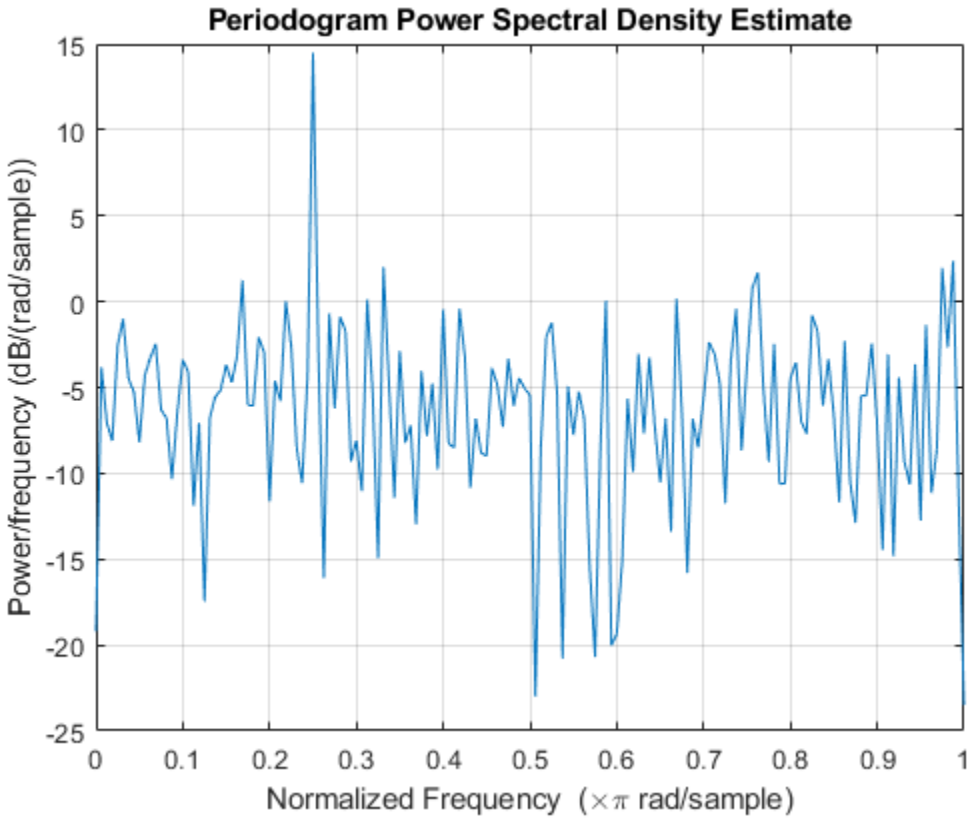


DFT Length Equal to Signal Length

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)
```

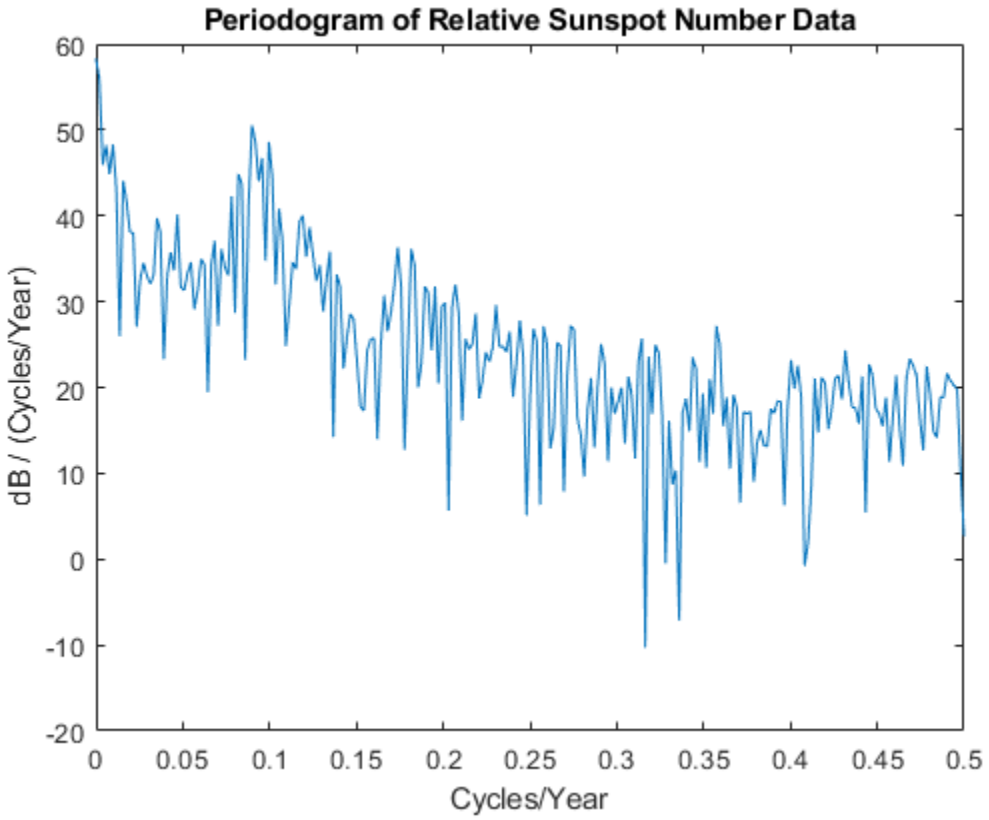


Periodogram of Relative Sunspot Numbers

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 / (Cycles/Year)')
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.

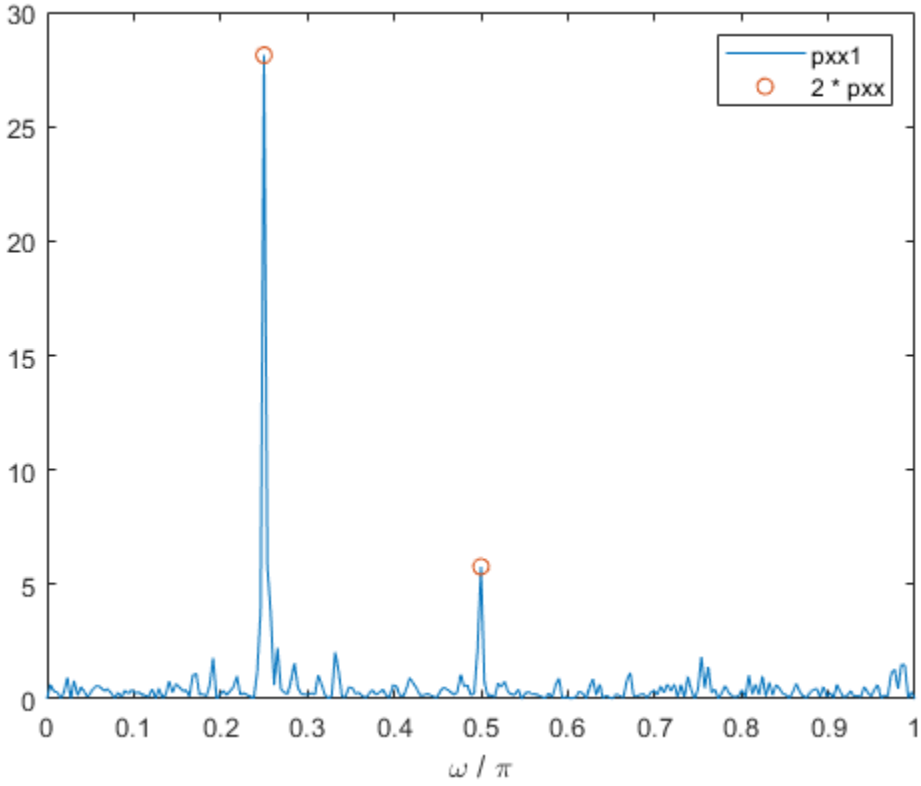
Periodogram at a Given Set of Normalized Frequencies

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
```

```
pxx = 1×2

   14.0589    2.8872
```

```
[pxx1,w1] = periodogram(x);
plot(w1/pi,pxx1,w/pi,2*pxx,'o')
legend('pxx1','2 * pxx')
xlabel('\omega / \pi')
```



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 = 1×2

    0.2647    0.2313
```

Upper and Lower 95%-Confidence Bounds

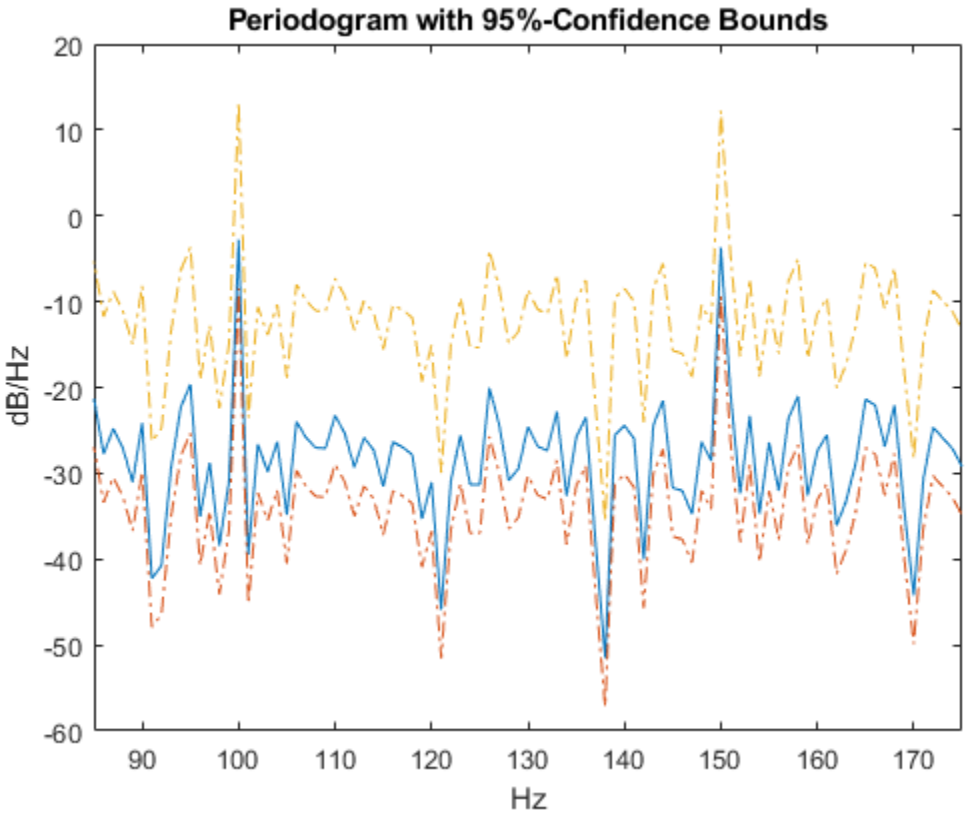
The following example illustrates the use of confidence bounds with the periodogram. While not a necessary condition for statistical significance, frequencies in the periodogram where the lower confidence bound exceeds the upper confidence bound for surrounding PSD estimates clearly indicate significant 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 = 0:1/fs:1-1/fs;
x = cos(2*pi*100*t) + sin(2*pi*150*t) + randn(size(t));
```

Obtain the periodogram PSD estimate 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),'-.')

xlim([85 175])
xlabel('Hz')
ylabel('dB/Hz')
title('Periodogram with 95%-Confidence Bounds')
```

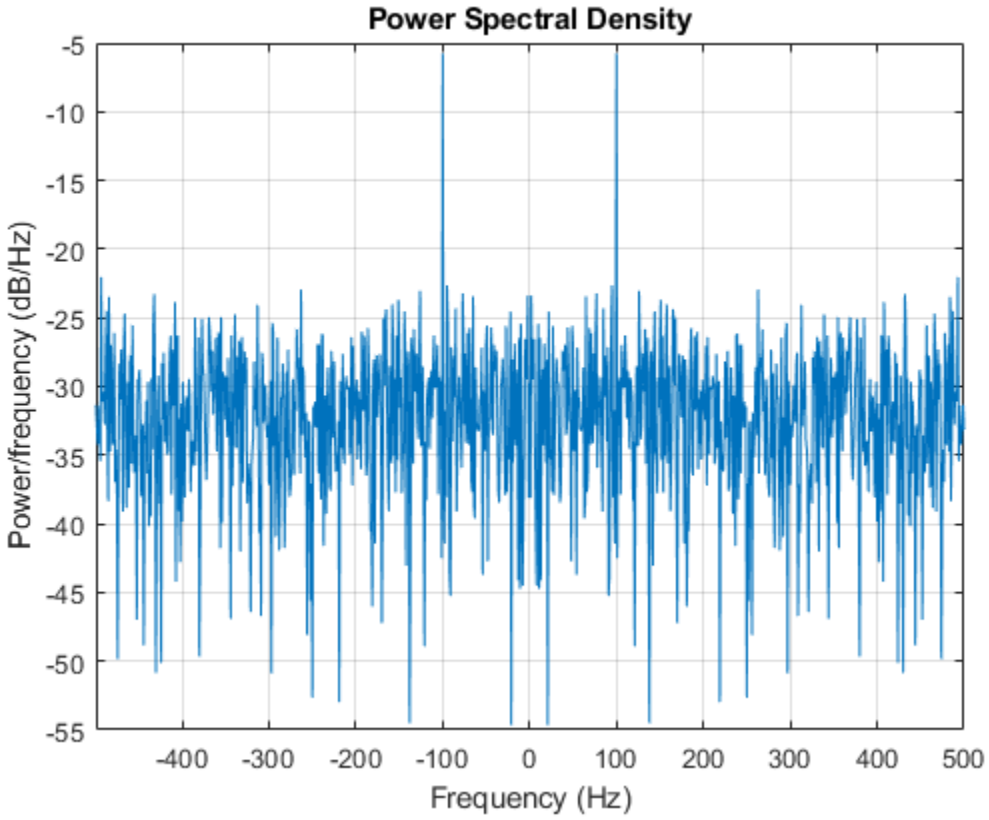


The lower confidence bound in the immediate vicinity of 100 and 150 Hz is significantly above the upper confidence bound outside the vicinity of 100 and 150 Hz.

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')
```



Reassigned Periodogram

Generate a signal that consists of a 200 Hz sinusoid embedded in white Gaussian noise. The signal is sampled at 1 kHz for 1 second. The noise has a variance of 0.01^2. Reset the random number generator for reproducible results.

```
rng('default')

Fs = 1000;
```

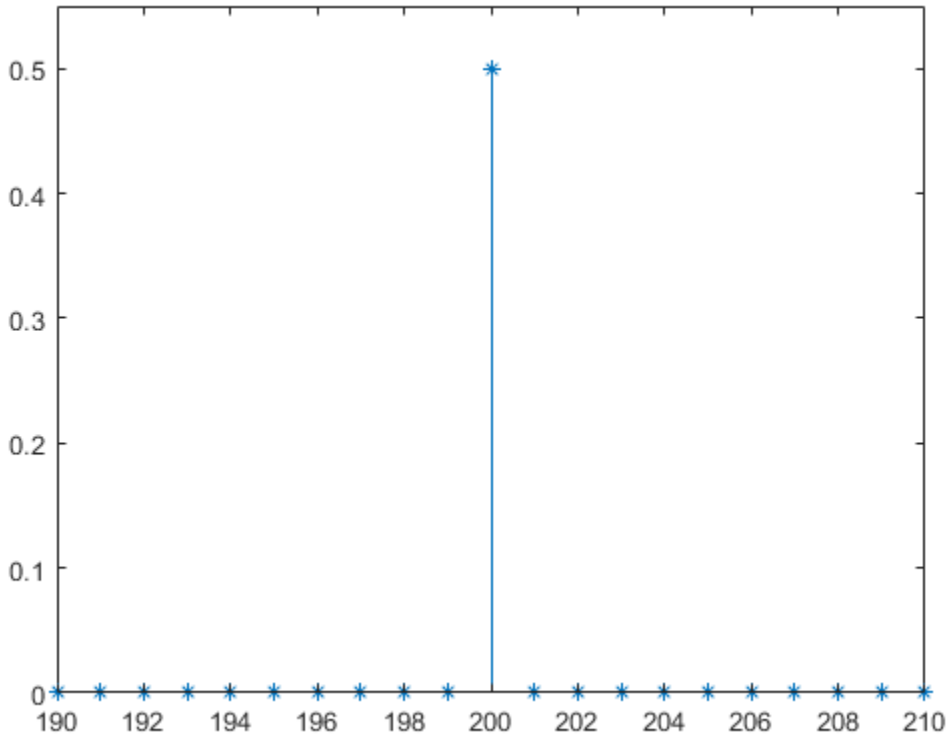
```
t = 0:1/Fs:1-1/Fs;
N = length(t);
x = sin(2*pi*t*200)+0.01*randn(size(t));
```

Use the FFT to compute the power spectrum of the signal, normalized by the signal length. The sinusoid is in-bin, so all the power is concentrated in a single frequency sample. Plot the one-sided spectrum. Zoom in to the vicinity of the peak.

```
q = fft(x,N);
ff = 0:Fs/N:Fs-Fs/N;

ffts = q*q'/N^2
```

```
ffts = 0.4997
```

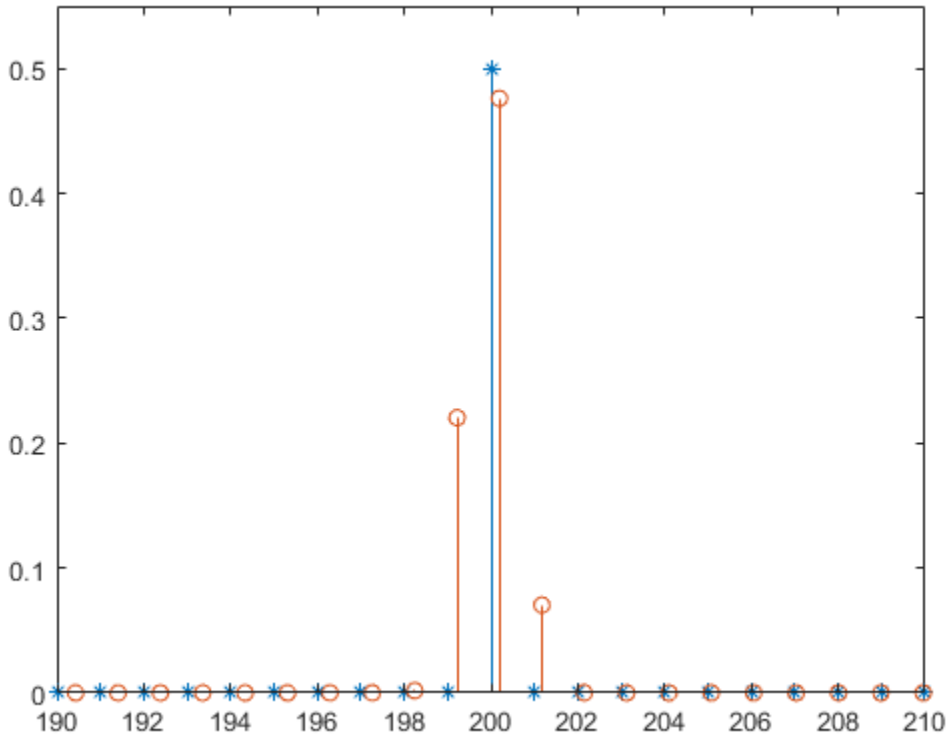
```
ff = ff(1:floor(N/2)+1);
q = q(1:floor(N/2)+1);

stem(ff,abs(q)/N,'*')
axis([190 210 0 0.55])
```



Use `periodogram` to compute the power spectrum of the signal. Specify a Hann window and an FFT length of 1024. Find the percentage difference between the estimated power at 200 Hz and the actual value.

```
wind = hann(N);
```

```
[pun,fr] = periodogram(x,wind,1024,Fs,'power');

hold on
stem(fr,pun)
```

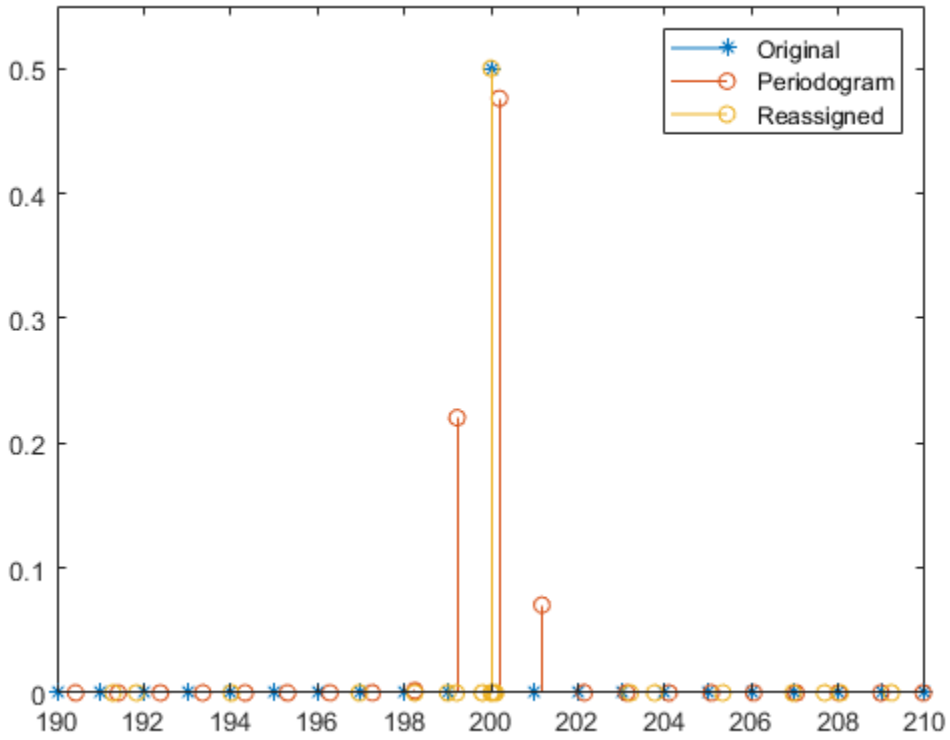



```
periodogErr = abs(max(pun)-ffts)/ffts*100
```

```
periodogErr = 4.7349
```

Recompute the power spectrum, but this time use reassignment. Plot the new estimate and compare its maximum with the FFT value.

```
[pre,ft,pxx,fx] = periodogram(x,wind,1024,Fs,'power','reassigned');

stem(fx,pre)
hold off
legend('Original','Periodogram','Reassigned')
```




```
reassignErr = abs(max(pre)-ffts)/ffts*100
```

```
reassignErr = 0.0779
```

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))
```

```
The maximum power occurs at 100.0 Hz
```

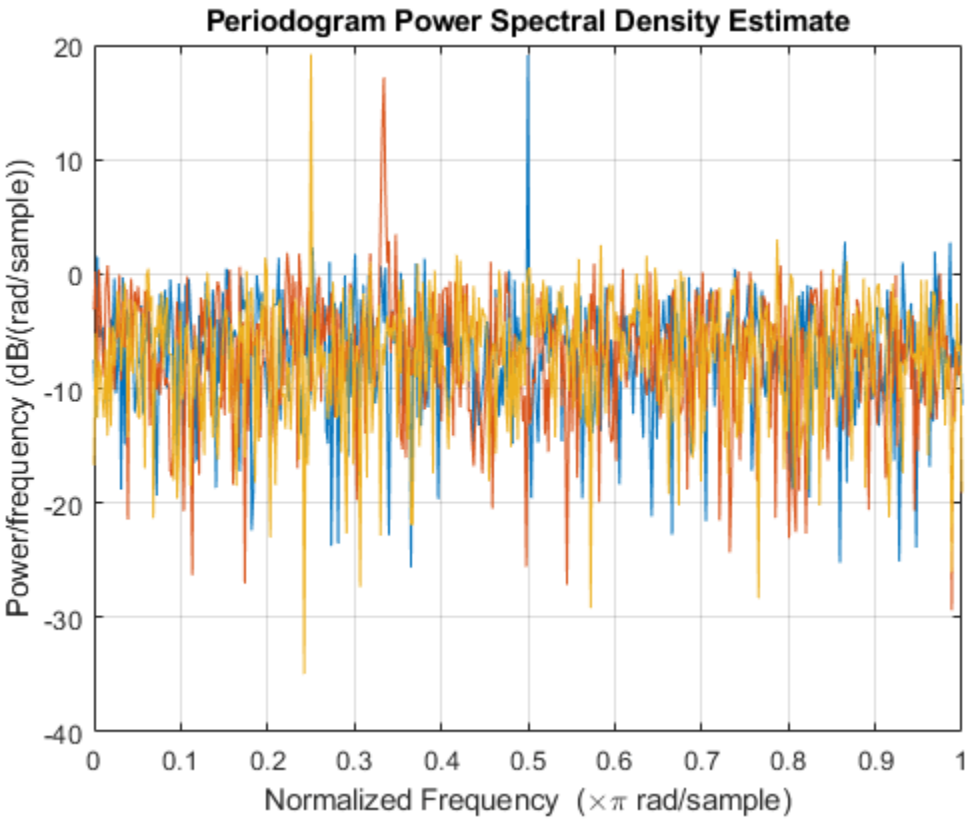
```
fprintf('The power estimate is %2.2f\n',pwrest)
```

```
The power estimate is 1.62
```

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)
```



Compute Modified Periodogram Using Generated C Code

Create a function `periodogram_data.m` that returns the modified periodogram power spectral density (PSD) estimate of an input signal using a window. The function specifies a number of discrete Fourier transform points equal to the length of the input signal.

```
type periodogram_data
```

```
function [pxx,f] = periodogram_data(inputData,window)
%#codegen
```

```
nfft = length(inputData);
[pxx,f] = periodogram(inputData,window,nfft);
end
```

Use `codegen` (MATLAB Coder) to generate a MEX function.

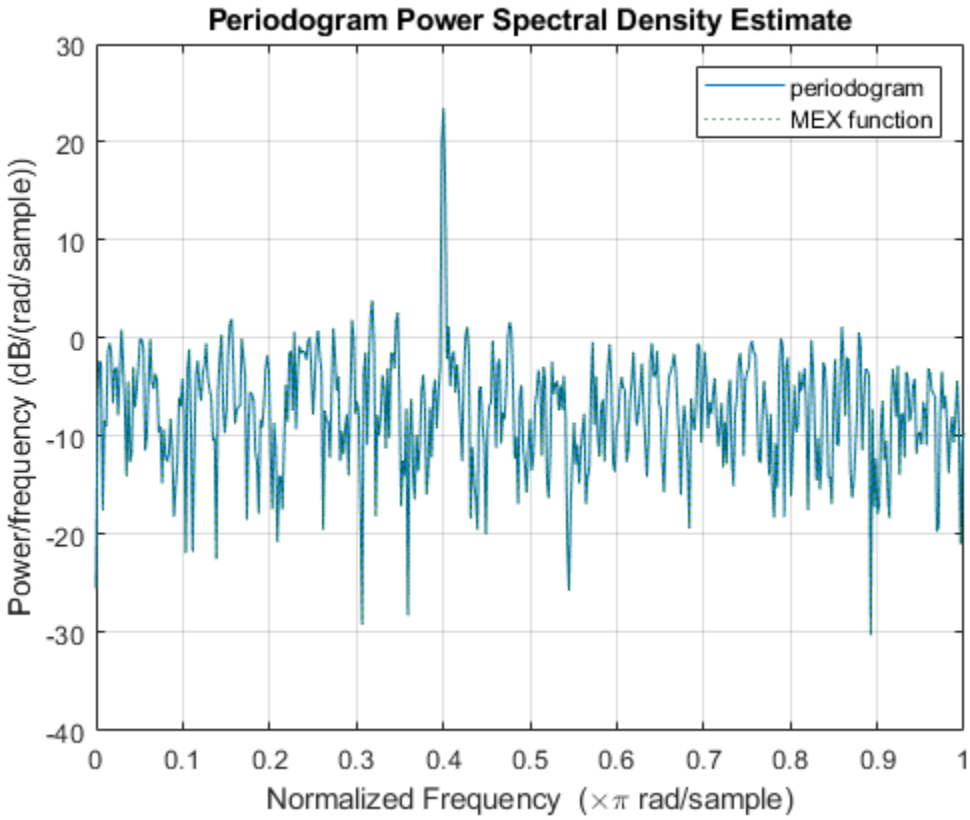
- The `%#codegen` directive in the function indicates that the MATLAB® code is intended for code generation.
- The `-args` option specifies example arguments that define the size, class, and complexity of the inputs to the MEX-file. For this example, specify `inputData` as a 1024-by-1 double precision random vector and `window` as a Hamming window of length 1024. In subsequent calls to the MEX function, use 1024-sample input signals and windows.
- If you want the MEX function to have a different name, use the `-o` option.
- If you want to view a code generation report, add the `-report` option at the end of the `codegen` command.

```
codegen periodogram_data -args {randn(1024,1),hamming(1024)}
```

```
Code generation successful.
```

Compute the PSD estimate of a 1024-sample noisy sinusoid using the periodogram function and the MEX function you generated. Specify a sinusoid normalized frequency of $2\pi/5$ rad/sample and a Hann window. Plot the two estimates to verify they coincide.

```
N = 1024;
x = 2*cos(2*pi/5*(0:N-1)') + randn(N,1);
periodogram(x,hann(N))
[pxMex,fMex] = periodogram_data(x,hann(N));
hold on
plot(fMex/pi,pow2db(pxMex),':','Color',[0 0.4 0])
hold off
grid on
legend('periodogram','MEX function')
```



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))` | `kaiser(length(x),38)` | vector | `[]`

Window, specified as a row or column vector with the same length as the input signal. If you specify `window` as empty, then `periodogram` uses a rectangular window. If you specify the `'reassigned'` flag and an empty `window`, then the function uses a Kaiser window with $\beta = 38$.

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` — 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 has units of Hz.

`w` — Normalized frequencies

vector

Normalized frequencies, specified as a row or column vector with at least two elements. Normalized frequencies are in rad/sample.

Example: `w = [pi/4 pi/2]`

Data Types: `double`

`f` — Frequencies

vector

Frequencies, specified as a row or column vector with at least two elements. The frequencies are in cycles per unit time. The unit time is specified by the sample rate, `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,π] rad/sample. If `nfft` is odd, the length of `pxx` is (`nfft` + 1)/2 and the interval is [0,π) 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.

  The function multiplies the power by 2 at all frequencies except 0 and the Nyquist frequency to conserve the total power.
- `'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 [0,2π) 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 (–π,π] rad/sample for even length `nfft` and (–π,π) 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.

`spectrumtype` — Power spectrum scaling
`'psd'` (default) | `'power'`

Power spectrum scaling, specified as `'psd'` or `'power'`. To return the power spectral density, omit `spectrumtype` or specify `'psd'`. To obtain an estimate of the power at each frequency, use `'power'` instead. Specifying `'power'` scales each estimate of the PSD by the equivalent noise bandwidth of the window, except when the `'reassigned'` flag is used.

`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` × 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 Ω and specify the sample rate in hertz, the PSD estimate is in watts per hertz.

Data Types: `single` | `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` | `single`

`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,π] if `nfft` is even and [0,π) if `nfft` is odd. If `pxx` is a two-sided PSD estimate, `w` spans the interval [0,2π). For a DC-centered PSD estimate, `w` spans the interval (–π,π] for even `nfft` and (–π,π) for odd `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`

`rpxx` — Reassigned PSD estimate
vector | matrix

Reassigned PSD estimate, returned as a real-valued, nonnegative column vector or matrix. Each column of `rpxx` is the reassigned PSD estimate of the corresponding column of `x`.

`fc` — Center-of-energy frequencies
vector | matrix

Center-of-energy frequencies, specified as a vector or matrix.

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

$$\widehat{P}(f) = \frac{\Delta t}{N}\left|\sum_{n=0}^{N-1} x_n e^{-j2\pi f \Delta t\, n}\right|^2, \quad -1/2\Delta t < f \le 1/2\Delta t,$$

where Δt 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

$$\widehat{P}(\omega) = \frac{1}{2\pi N}\left|\sum_{n=0}^{N-1} x_n e^{-j\omega n}\right|^2, \quad -\pi < \omega \le \pi\,.$$

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 1-1584.

The integral of the true PSD, $P(f)$, over one period, $1/\Delta t$ for cyclical frequency and 2π 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

$$\widehat{P}(f) = \frac{\Delta t}{N}\left|\sum_{n=0}^{N-1} h_n x_n e^{-j2\pi f \Delta t\, n}\right|^2, \quad -1/2\Delta t < f \le 1/2\Delta t,$$

where Δt is the sampling interval.

If the frequencies are in radians/sample, the modified periodogram is defined as

$$\widehat{P}(\omega) = \frac{1}{2\pi N}\left|\sum_{n=0}^{N-1} h_n x_n e^{-j\omega n}\right|^2, \quad -\pi < \omega \le \pi\,.$$

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 1-1584.

Reassigned Periodogram

The reassignment technique sharpens the localization of spectral estimates and produces periodograms that are easier to read and interpret. This technique reassigns each PSD estimate to the center of energy of its bin, away from the bin’s geometric center. It provides exact localization for chirps and impulses.

References

[1] Auger, François, and Patrick Flandrin. "Improving the Readability of Time-Frequency and Time-Scale Representations by the Reassignment Method." *IEEE Transactions on Signal Processing*. Vol. 43, May 1995, pp. 1068–1089.

[2] Fulop, Sean A., and Kelly Fitz. "Algorithms for computing the time-corrected instantaneous frequency (reassigned) spectrogram, with applications." *Journal of the Acoustical Society of America*. Vol. 119, January 2006, pp. 360–371.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Apps
Signal Analyzer

Functions
`bandpower` | `pcov` | `pburg` | `plomb` | `pmcov` | `pmtm` | `pspectrum` | `pwelch` | `sfdr`

Topics
“Bias and Variability in the Periodogram”
“Power Spectral Density Estimates Using FFT”
“Nonparametric Methods”

Introduced before R2006a

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,f] = phasedelay(___,n,fs)
[phi,f] = phasedelay(___,n,'whole',fs)

phi = phasedelay(___,w)
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 corresponding `n`-point angular frequency vector `w` for the digital filter with transfer function coefficients stored in `b` and `a`.

`[phi,w] = phasedelay(sos,n)` returns the `n`-point phase delay response corresponding to the second-order sections `sos`.

`[phi,w] = phasedelay(d,n)` returns the `n`-point phase delay response of the digital filter `d`.

`[phi,w] = phasedelay(___,n,'whole')` returns the phase delay response at `n` equally spaced points around the whole unit circle.

`[phi,f] = phasedelay(___,n,fs)` returns the phase delay response and the corresponding `n`-point frequency vector `f` for a digital filter designed to filter signals sampled at a rate `fs`.

`[phi,f] = phasedelay(___,n,'whole',fs)` returns the frequency vector `f` at `n` points ranging between `0` and `fs`.

`phi = phasedelay(___,w)` returns the phase delay response evaluated at the angular frequencies specified in `w`.

`phi = phasedelay(___,f,fs)` returns the phase delay response evaluated at the frequencies specified in `f`.

`[phi,w,s] = phasedelay(___)` returns plotting information, where `s` is a structure with fields that you can change to display different frequency response plots.

`[phi,f,s] = phasedelay( ___ )` returns plotting information, where `s` is a structure with fields that you can change to display different frequency response plots.

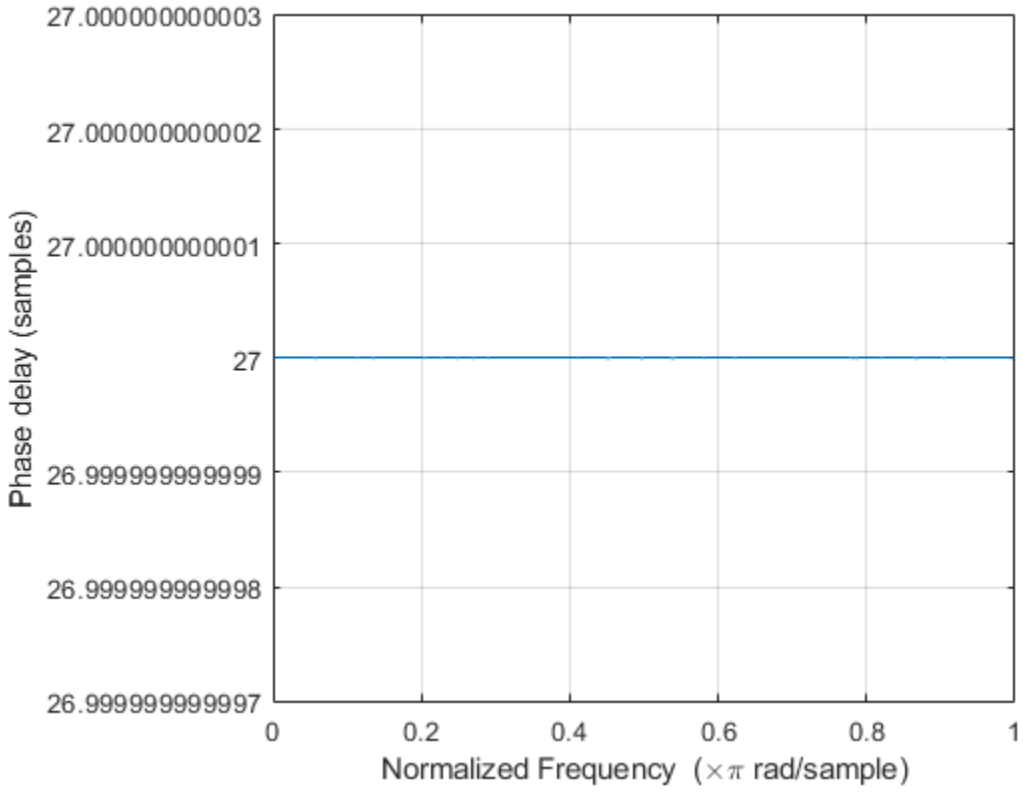
`phasedelay( ___ )` plots the phase delay response versus frequency.

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.

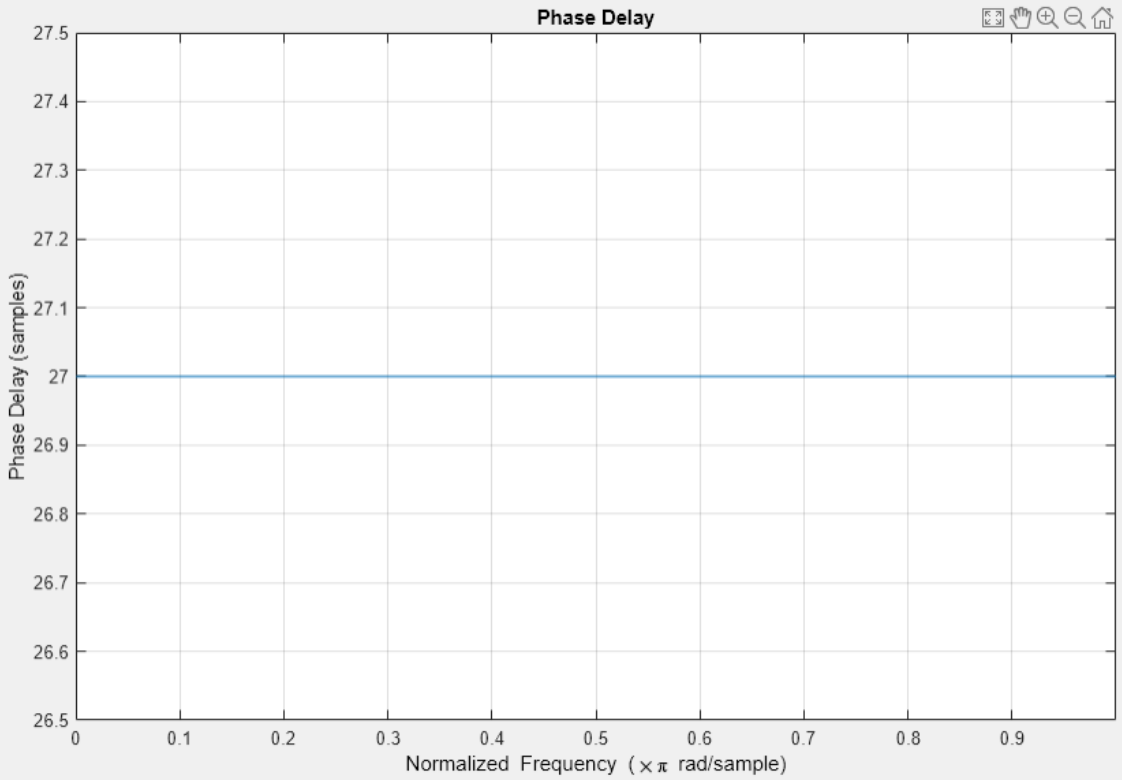
```
Ap = 0.02;
As = 0.008;

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)
```



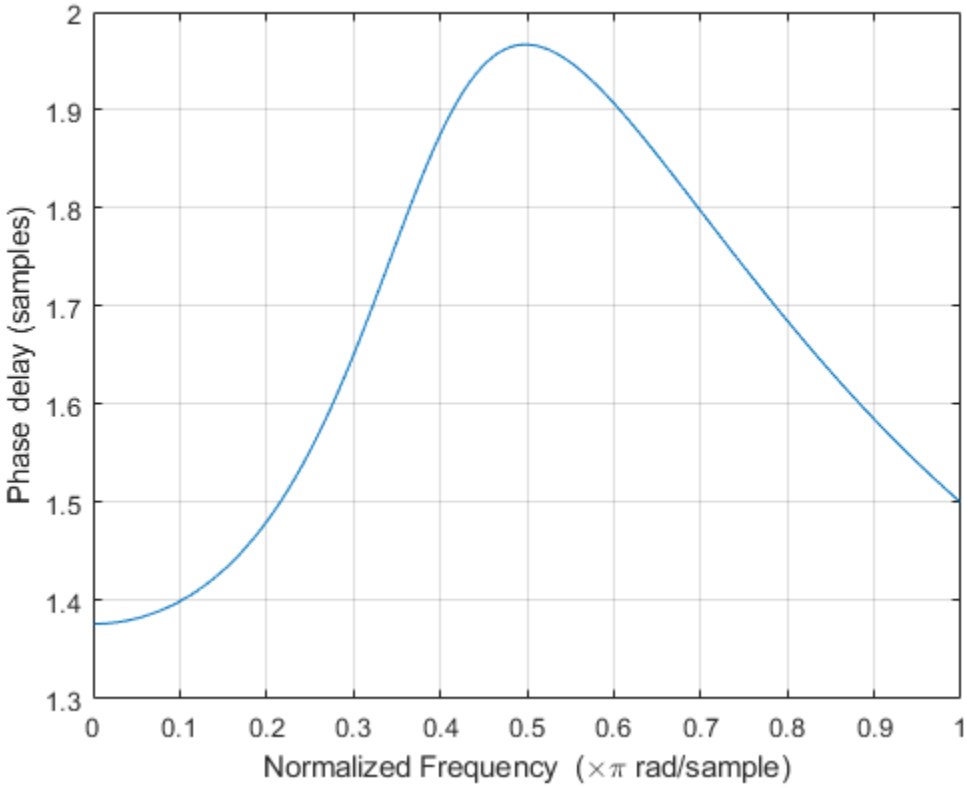
Phase Delay Response from Second-Order Sections

Design a third-order lowpass Butterworth filter with a cutoff frequency of 200 Hz. The sample rate is 1000 Hz.

```
fc = 200;
fs = 1000;

[z,p,k] = butter(3,fc/(fs/2),'low');
```

Use the `zp2sos` function to convert the zeros, poles, and gain to second-order sections. Compute the phase delay response of the filter and set the number of evaluation points to `1024`. Display the result.

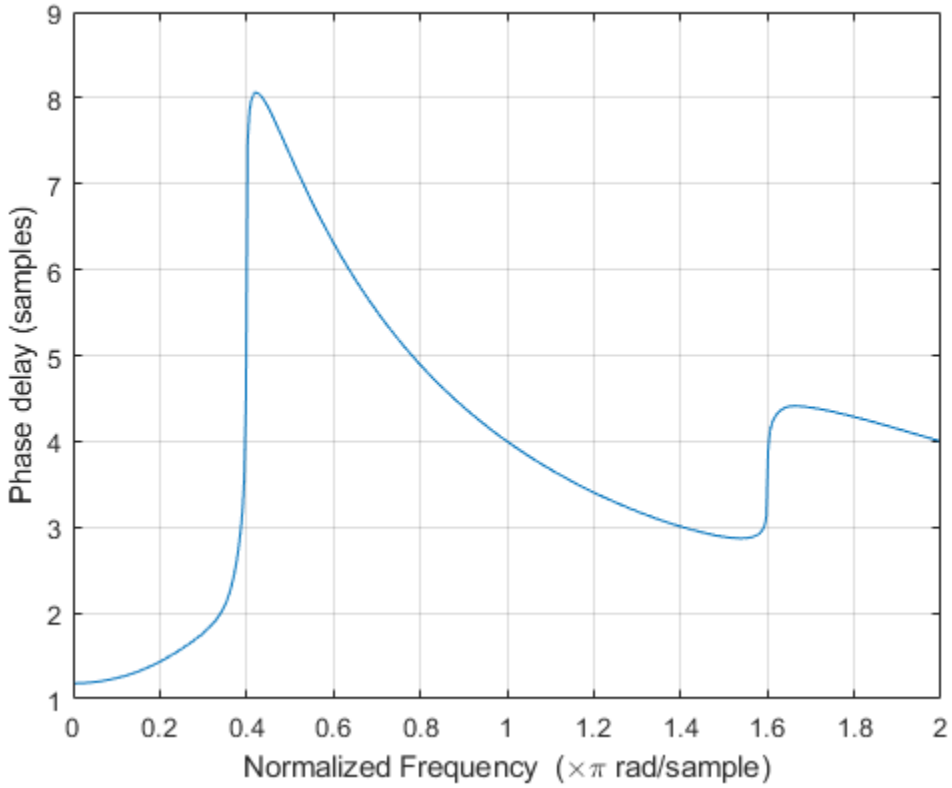
```
sos = zp2sos(z,p,k);
phasedelay(sos,1024)
```



Phase Delay Response of an Elliptic Filter

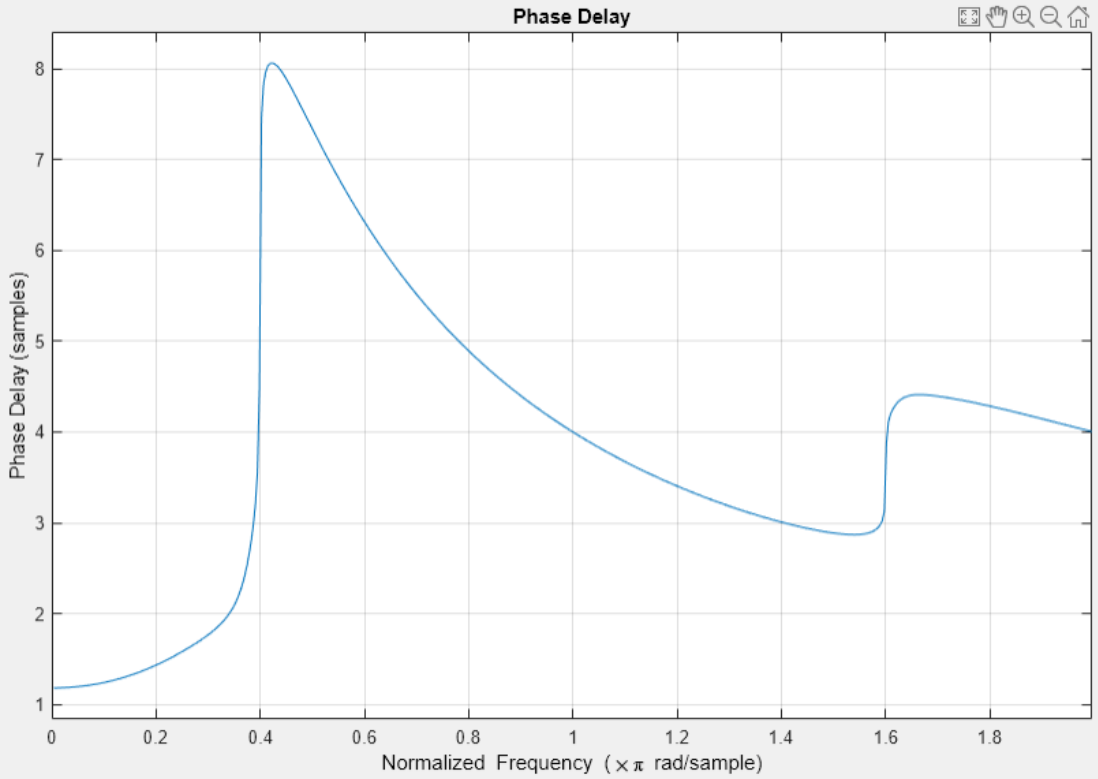
Design an elliptic filter of order 10 and normalized passband frequency 0.4. Specify a passband ripple of 0.5 dB and a stopband attenuation of 20 dB. Display the phase delay response of the filter over the complete unit circle.

```
[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')
```



Input Arguments

b, a — Transfer function coefficients

vectors

Transfer function coefficients, specified as vectors.

Data Types: `single` | `double`

n — Number of evaluation points

512 (default) | positive integer

Number of evaluation points, specified as a positive integer. Set `n` to a value greater than the filter order.

Data Types: `single` | `double`

sos — Second-order section coefficients

matrix

Second-order section coefficients, specified as a matrix. `sos` is a *K*-by-6 matrix, where *K* is the number of sections and must be greater than or equal to 2. If the number of sections is less than 2, the function 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 `[`b_i`(1) `b_i`(2) `b_i`(3) `a_i`(1) `a_i`(2) `a_i`(3)]`.

Data Types: `single` | `double`

`d` — Digital filter
`digitalFilter` object

Digital filter, specified as a `digitalFilter` object. To generate `d` based on frequency-response specifications, use the `designfilt` function.

`w` — Angular frequencies
vector

Angular frequencies at which the function evaluates the phase delay response, specified as a vector and expressed in rad/sample. The frequencies are normally between 0 and π. `w` must contain at least two elements.

`fs` — Sample rate
real-valued scalar

Sample rate, specified as a real-valued scalar and expressed in hertz.

Data Types: `double`

`f` — Frequencies
vector

Frequencies at which the function evaluates the phase delay response, specified as a vector and expressed in hertz. `f` must contain at least two elements.

Output Arguments

`phi` — Phase delay response
vector

Phase delay response, returned as a vector of length `n`. The phase delay response is evaluated at `n` equally spaced points around the upper half of the unit circle.

Note If the input to `phasedelay` is single precision, the function calculates the phase delay response using single-precision arithmetic. The output `phi` is single precision.

`w` — Angular frequencies
vector

Angular frequencies in rad/sample, returned as a vector. If you specify `n`, `w` has length `n`. If you do not specify `n` or you specify `n` as an empty vector, then `w` has length `512`.

`f` — Frequencies
vector

Frequencies in hertz, returned as a vector. If you specify `n`, `f` has length `n`. If you do not specify `n` or you specify `n` as an empty vector, then `f` has length `512`.

`s` — Plotting information
structure

Plotting information, returned as a structure. You can modify the fields in `s` to display different frequency response plots.

Algorithms

The *phase delay response* of a filter corresponds to the time delay that each frequency component experiences as the input signal passes through the system. The `phasedelay` function returns the phase delay response and the frequency vector of the filter

$$H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})} = \frac{\mathrm{b}(1) + \mathrm{b}(2)e^{-j\omega} + \ldots + \mathrm{b}(\mathrm{m}+1)e^{-jm\omega}}{\mathrm{a}(1) + \mathrm{a}(2)e^{-j\omega} + \ldots + \mathrm{a}(\mathrm{n}+1)e^{-jn\omega}}$$

given numerator and denominator coefficients in inputs `b` and `a`.

See Also

`designfilt` | `digitalFilter` | `freqz` | **FVTool** | `phasez` | `grpdelay`

Introduced before R2006a

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,f] = phasez(___,n,fs)
phi = phasez(___,f,fs)

phi = phasez(___,w)
phasez(___)
```

Description

`[phi,w] = phasez(b,a,n)` returns the `n`-point phase response vector `phi` and the corresponding angular frequency vector `w` for the digital filter with the transfer function coefficients stored in `b` and `a`.

`[phi,w] = phasez(sos,n)` returns the `n`-point phase response corresponding to the second-order sections matrix `sos`.

`[phi,w] = phasez(d,n)` returns the `n`-point phase response for the digital filter `d`.

`[phi,w] = phasez(___,n,'whole')` returns the phase response at `n` sample points around the entire unit circle. This syntax can include any combination of input arguments from the previous syntaxes.

`[phi,f] = phasez(___,n,fs)` returns the frequency vector.

`phi = phasez(___,f,fs)` returns the phase response vector `phi` evaluated at the physical frequencies supplied in `f`. This syntax can include any combination of input arguments from the previous syntaxes.

`phi = phasez(___,w)` returns the unwrapped phase response in radians at frequencies specified in `w`.

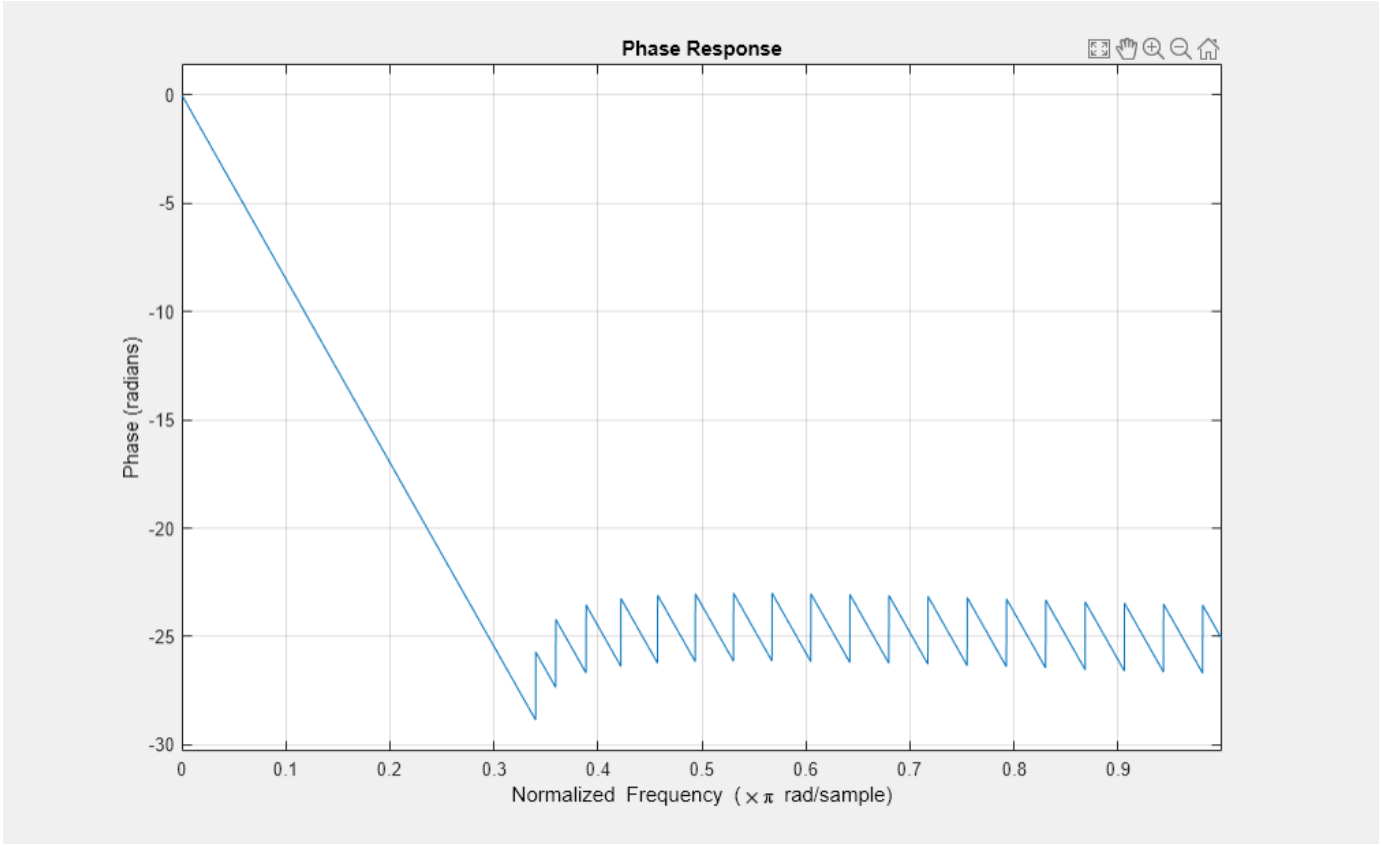
`phasez(___)` with no output arguments plots the phase response of the filter.

Examples

Phase Response of an FIR Filter

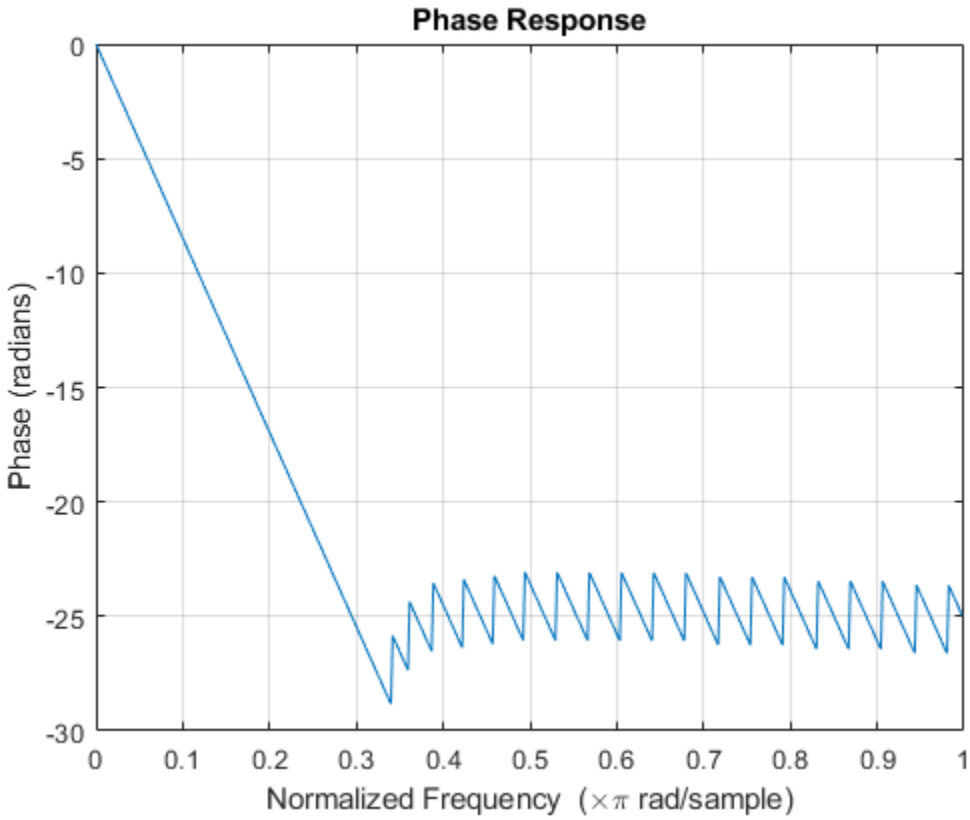
Use `designfilt` to design an FIR filter of order 54, normalized cutoff frequency 0.3π rad/s, passband ripple 0.7 dB, and stopband attenuation 42 dB. Use the method of constrained least squares. Display the phase response of the filter.

```
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.

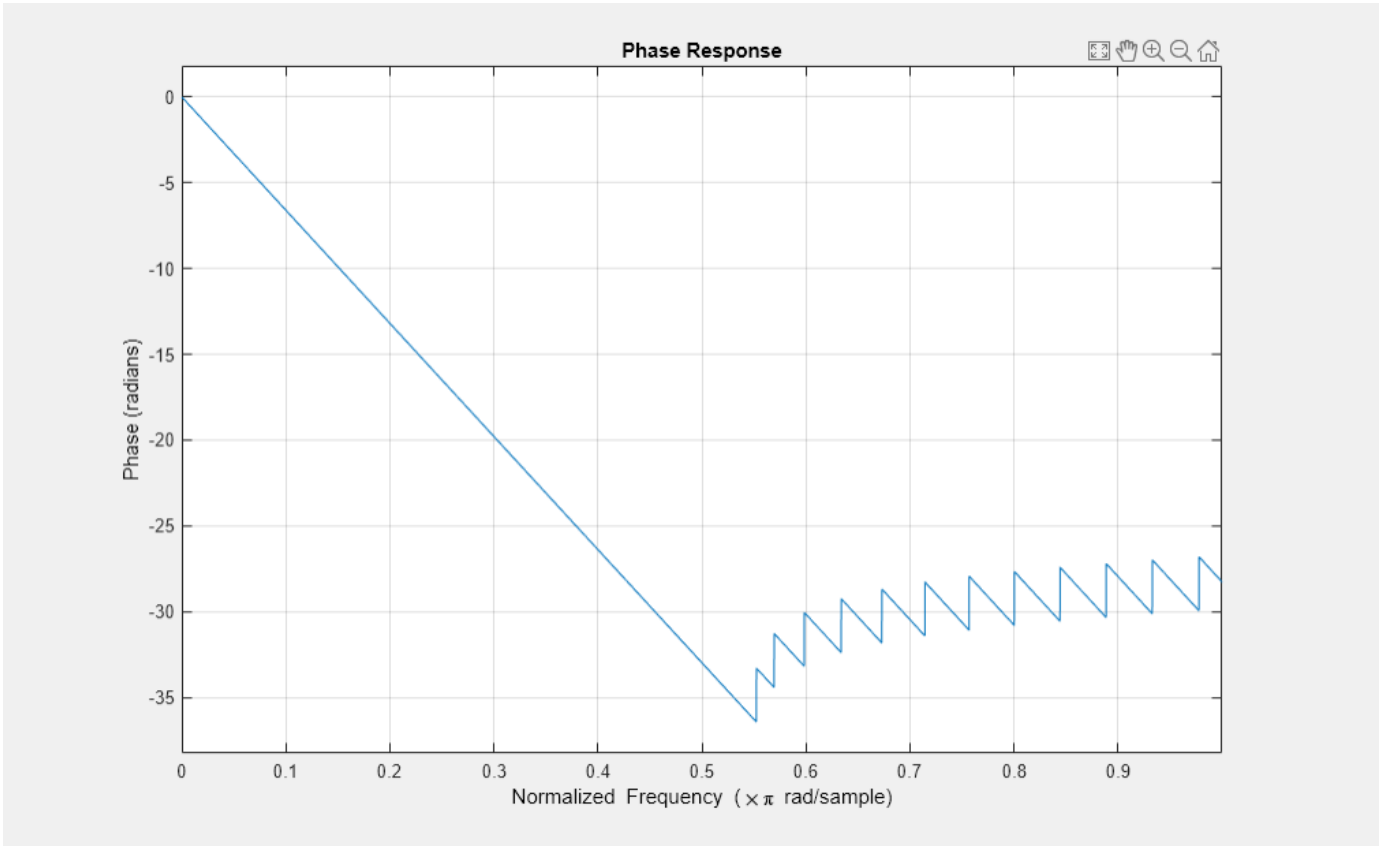
```
pAp = 10^(Ap/40);
Apl = (pAp-1)/(pAp+1);

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 0.45π rad/s, normalized stopband frequency 0.55π rad/s, passband ripple 1 dB, and stopband attenuation 60 dB. Display the phase response of the filter.

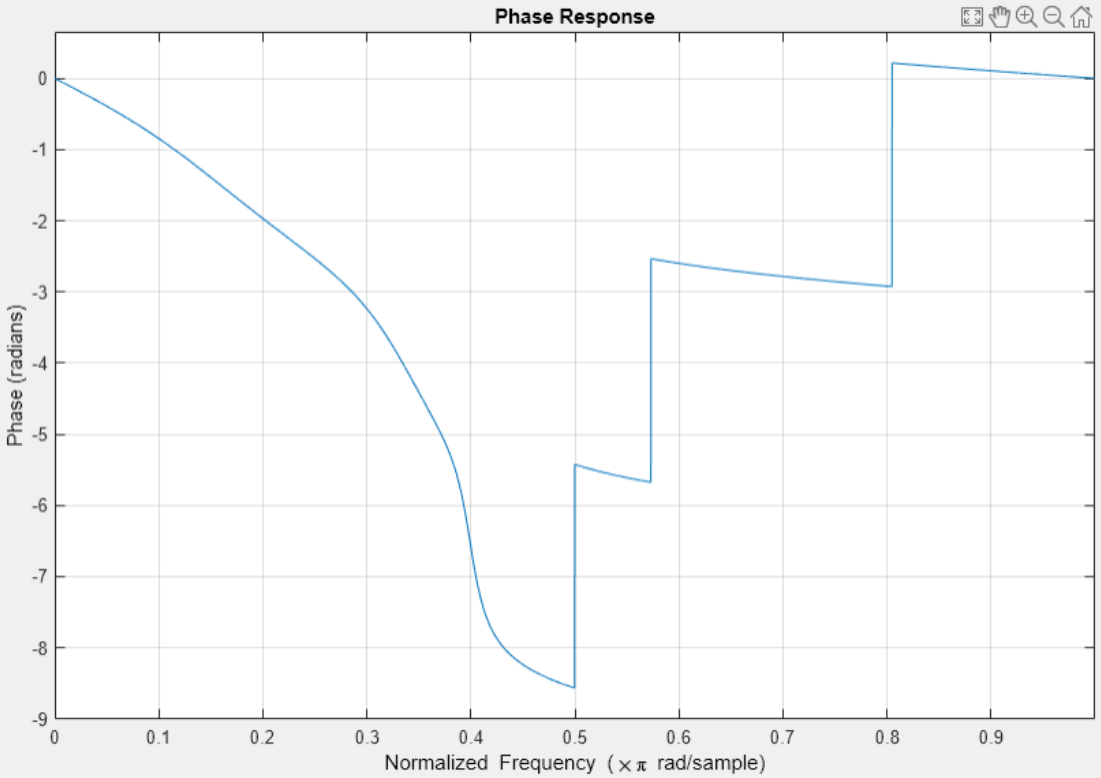
```
d = designfilt('lowpassfir', ...
               'PassbandFrequency',0.45,'StopbandFrequency',0.55, ...
               'PassbandRipple',1,'StopbandAttenuation',60, ...
               'DesignMethod','equiripple');
phasez(d)
```



Phase Response of an Elliptic Filter

Design an elliptic lowpass IIR filter with normalized passband frequency 0.4π rad/s, normalized stopband frequency 0.5π 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)
```



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

$$H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})} = \frac{\mathrm{b(1)}+\mathrm{b(2)}\,e^{-j\omega}+\mathrm{b(3)}\,e^{-j2\omega}+\cdots+\mathrm{b(M)}\,e^{-j(M-1)\omega}}{\mathrm{a(1)}+\mathrm{a(2)}\,e^{-j\omega}+\mathrm{a(3)}\,e^{-j2\omega}+\cdots+\mathrm{a(N)}\,e^{-j(N-1)\omega}}.$$

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 0.5π 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.

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, the function treats the input as 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 0.5π 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 0.5π rad/sample.

`fs` — Sample rate
positive scalar

Sample 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 rad/sample. `w` must have at least two elements, because otherwise the function interprets it as `n`. `w` = π corresponds to the Nyquist frequency.

`f` — Frequencies
vector

Frequencies, specified as a vector. `f` must have at least two elements, because otherwise the function interprets it as `n`. When the unit of time is seconds, `f` is expressed in hertz.

Data Types: `double`

Output Arguments

`phi` — Phase response
vector

Phase response, returned as a vector. If you specify `n`, then `phi` has length `n`. If you do not specify `n`, or specify `n` as an empty vector, then `phi` has length 512.

If the input to `phasez` is single precision, the function computes the phase response using single-precision arithmetic. The output `phi` is single precision.

`w` — Angular frequencies
vector

Angular frequencies, returned as a vector. `w` has values ranging from 0 to π. If you specify `'whole'` in your input, the values in `w` range from 0 to 2π. 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 `fs`/2 Hz. If you specify `'whole'` in your input, the values in `f` range from 0 to `fs` Hz. If you specify `n`, `f` has length `n`. If you do not specify `n`, or specify `n` as an empty vector, `f` has length 512.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If input `b` is a variable-sized matrix during code generation, then it must not reduce to a vector at runtime.
- If input `n` is a variable-sized vector during code generation, then it must not reduce to a scalar at runtime.
- If there is a discontinuity in the phase response, the result returned by the generated code might differ from the result returned by MATLAB by 2π.

See Also
`designfilt` | `digitalFilter` | `freqz` | **FVTool** | `phasedelay` | `grpdelay`

Introduced before R2006a

pkurtosis

Spectral kurtosis from signal or spectrogram

Syntax

```
sk = pkurtosis(x)
sk = pkurtosis(x,sampx)
sk = pkurtosis(xt)
sk = pkurtosis( ___ ,window)

sk = pkurtosis(s,sampx,f,window)

[sk,fout] = pkurtosis( ___ )
[ ___ ,thresh] = pkurtosis( ___ ,'ConfidenceLevel',p)

pkurtosis( ___ )
```

Description

`sk = pkurtosis(x)` returns the spectral kurtosis on page 1-1616 of vector `x` as the vector `sk`. `pkurtosis` uses normalized frequency (evenly spaced frequency vector spanning [0 π]) to compute the time values. `pkurtosis` computes the spectrogram of `x` using `pspectrum` with default window size (time resolution in samples), and 80% window overlap.

`sk = pkurtosis(x,sampx)` returns the spectral kurtosis of vector `x` sampled at rate or time interval `sampx`.

`sk = pkurtosis(xt)` returns the spectral kurtosis of single-variable `timetable` `xt` in the vector `sk`. `xt` must contain increasing finite time samples.

`sk = pkurtosis( ___ ,window)` returns the spectral kurtosis using the time resolution specified in `window` for the `pspectrum` spectrogram computation. You can use `window` with any of the input arguments in previous syntaxes.

`sk = pkurtosis(s,sampx,f,window)` returns the spectral kurtosis using the spectrogram or power spectrogram `s`, along with:

- Sample rate or time, `sampx`, of the original time-series signal that was transformed to produce `s`
- Spectrogram frequency vector `f`
- Spectrogram time resolution `window`

Use this syntax when you want to customize the options for `pspectrum`, rather than accept the default `pspectrum` options that `pkurtosis` applies. You can specify `sampx` as empty to default to normalized frequency. Although `window` is optional for previous syntaxes, you must supply a value for `window` when using this syntax.

`[sk,fout] = pkurtosis( ___ )` returns the spectral kurtosis `sk` along with the frequency vector `fout`. You can use these output arguments with any of the input arguments in previous syntaxes.

`[ ___ ,thresh] = pkurtosis( ___ ,'ConfidenceLevel',p)` returns the spectral kurtosis threshold `thresh` using the confidence level `p`. `thresh` represents the range within which the

spectral kurtosis indicates a Gaussian stationary signal, at the optional confidence level `p` that you either specify or accept as default. Specifying `p` allows you to tune the sensitivity of the spectral kurtosis `thresh` results to behavior that is non-Gaussian or nonstationary. You can use the `thresh` output argument with any of the input arguments in previous syntaxes. You can also set the confidence level in previous syntaxes, but it has no effect unless you are returning or plotting `thresh`.

`pkurtosis( ___ )` plots the spectral kurtosis, along with confidence level and thresholds, without returning any data. You can use this syntax with any of the input arguments in previous syntaxes.

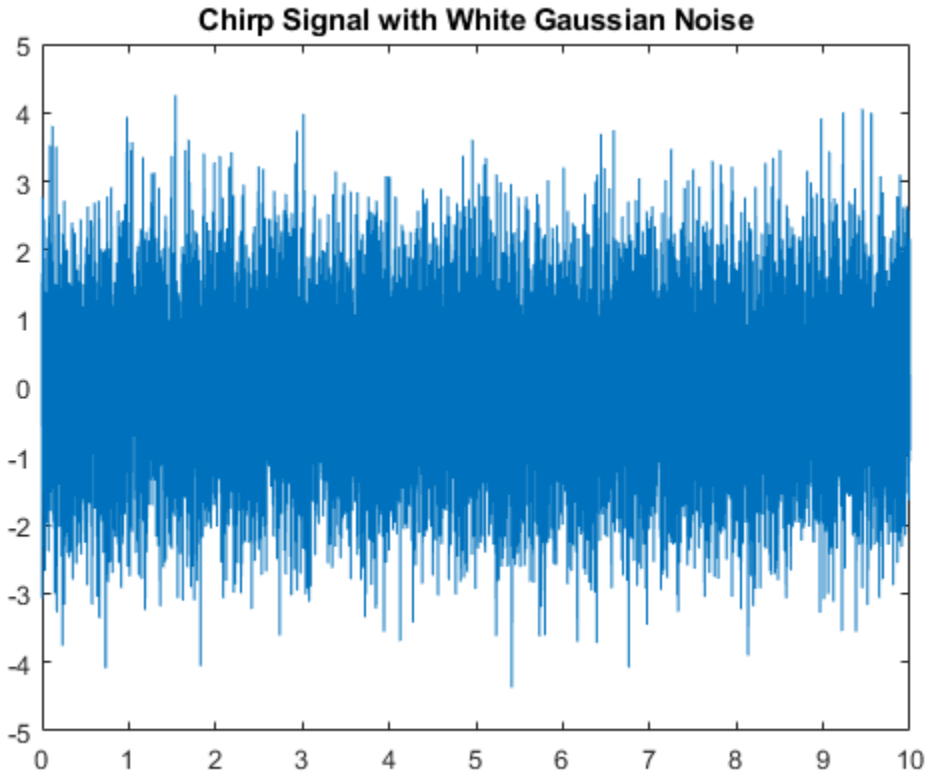
Examples

Plot Spectral Kurtosis of Nonstationary Signal Using Different Confidence Levels

Plot the spectral kurtosis of a chirp signal in white noise, and see how the nonstationary non-Gaussian regime can be detected. Explore the effects of changing the confidence level, and of invoking normalized frequency.

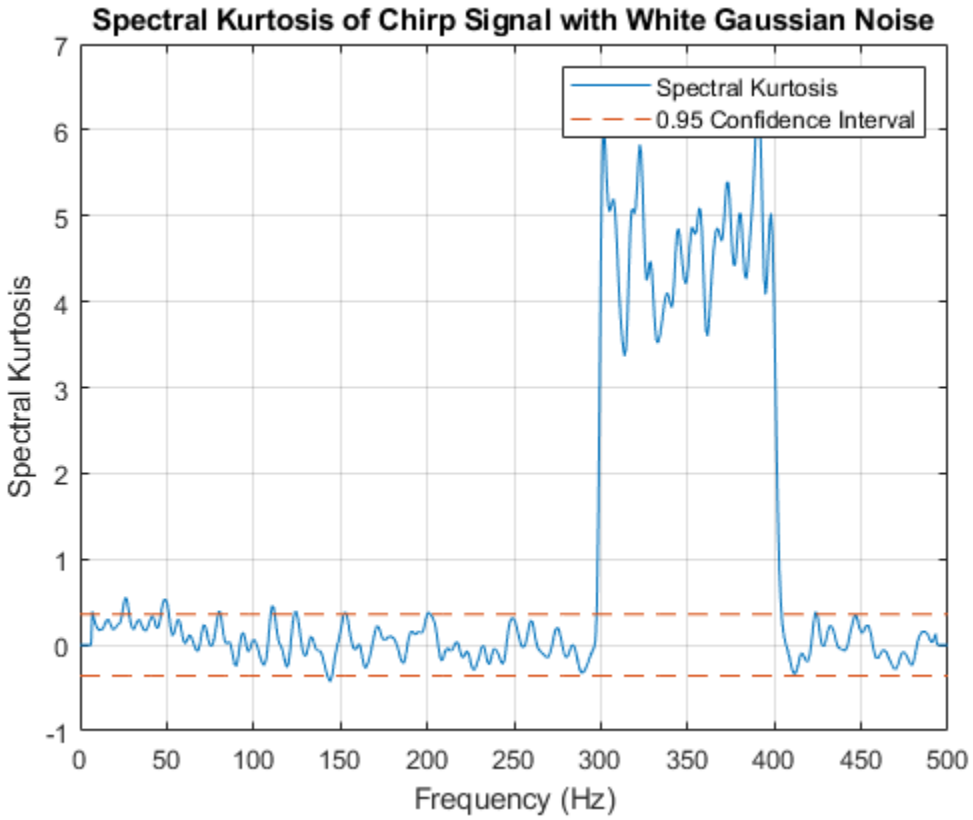
Create a chirp signal, add white Gaussian noise, and plot.

```
fs = 1000;
t = 0:1/fs:10;
f1 = 300;
f2 = 400;

xc = chirp(t,f1,10,f2);
x = xc + randn(1,length(t));

plot(t,x)
title('Chirp Signal with White Gaussian Noise')
```



Plot the spectral kurtosis of the signal.

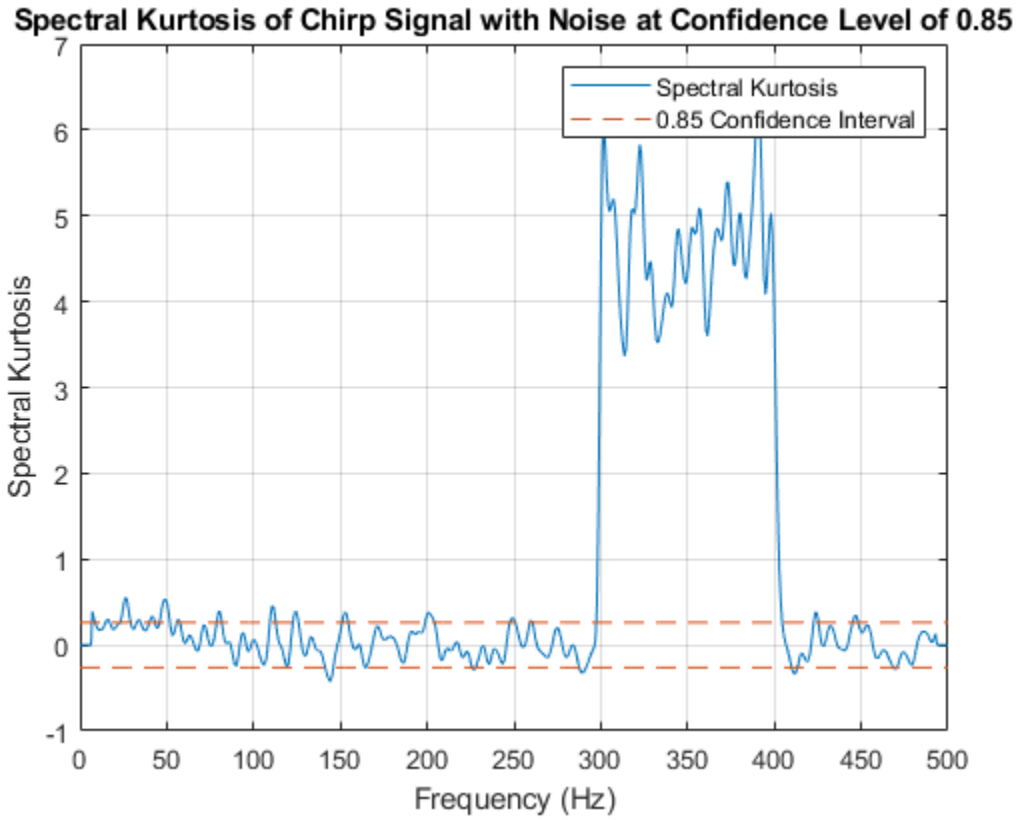
```
pkurtosis(x,fs)
title('Spectral Kurtosis of Chirp Signal with White Gaussian Noise')
```



The plot shows a clear extended excursion from 300–400 Hz. This excursion corresponds to the signal component which represents the nonstationary chirp. The area between the two horizontal red-dashed lines represents the zone of probable stationary and Gaussian behavior, as defined by the 0.95 confidence interval. Any kurtosis points falling within this zone are likely to be stationary and Gaussian. Outside of the zone, kurtosis points are flagged as nonstationary or non-Gaussian. Below 300 Hz, there are a few additional excursions slightly above the above the zone threshold. These excursions represent false positives, where the signal is stationary and Gaussian, but because of the noise, has exceeded the threshold.

Investigate the impact of the confidence level by changing it from the default 0.95 to 0.85.

```
pkurtosis(x,fs,'ConfidenceLevel',0.85)
title('Spectral Kurtosis of Chirp Signal with Noise at Confidence Level of 0.85')
```



The lower confidence level implies more sensitive detection of nonstationary or non-Gaussian frequency components. Reducing the confidence level shrinks the `thresh`-delimited zone. Now the low-level excursions — false alarms — have increased in both number and amount. Setting the confidence level is a balancing act between achieving effective detection and limiting the number of false positives.

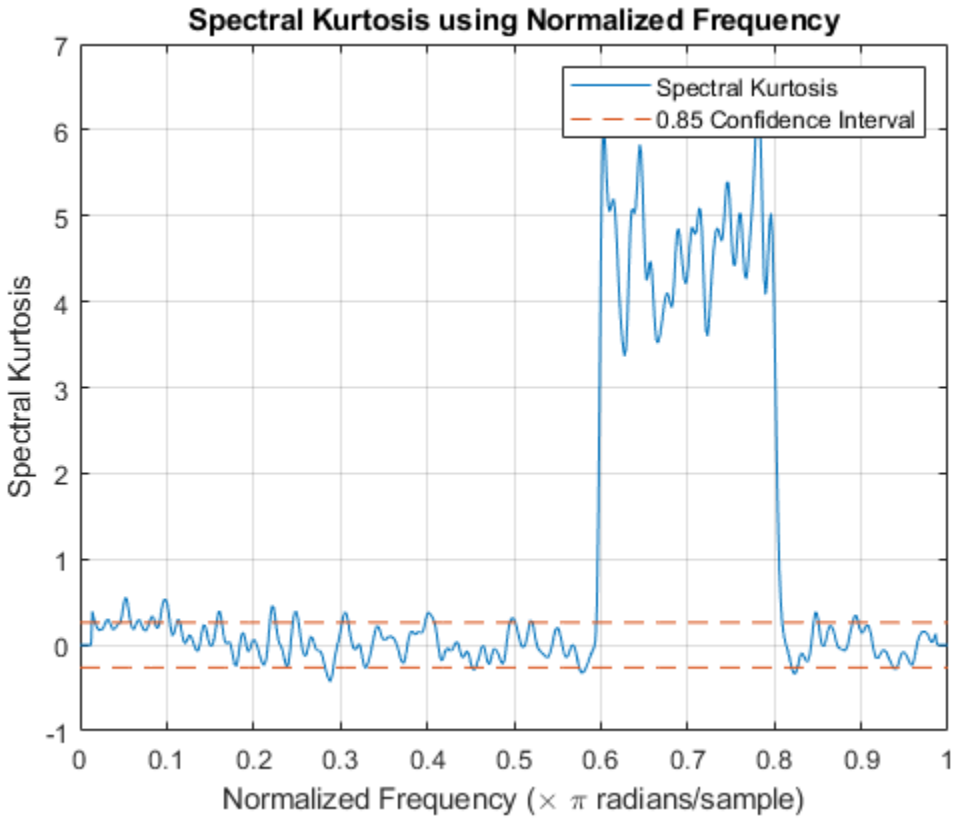
You can accurately determine and compare the zone width for the two cases by using the `pkurtosis` form that returns it.

```
[sk1,~,thresh95] = pkurtosis(x);
[sk2,~,thresh85] = pkurtosis(x,'ConfidenceLevel',0.85);
thresh = [thresh95 thresh85]
```

```
thresh = 1×2

    0.3578    0.2628
```

Plot the spectral kurtosis again, but this time, omit the sample time information so that `pkurtosis` plots normalized frequency.

```
pkurtosis(x,'ConfidenceLevel',0.85)
title('Spectral Kurtosis using Normalized Frequency')
```



The frequency axis has changed from Hz to a scale from 0 to π rad/sample.

Plot Spectral Kurtosis Using a Customized Window Size

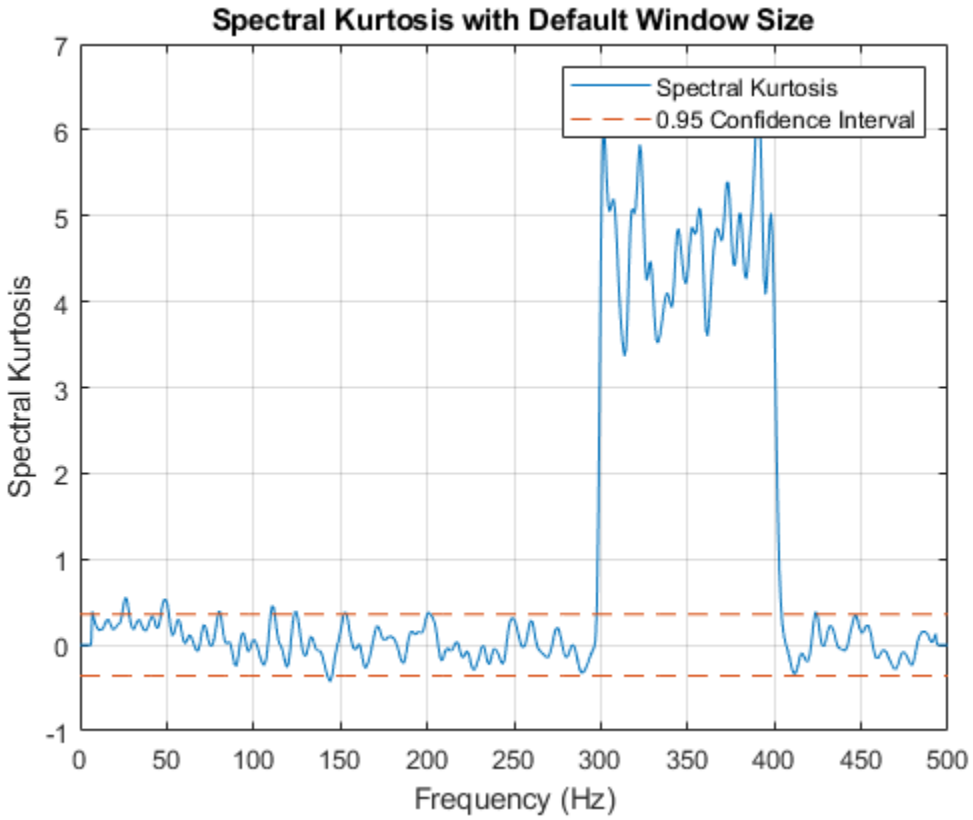
The `pkurtosis` function uses the default `pspectrum` window size (time resolution). You can specify the window size to use instead. In this example, use the function `kurtogram` to return an optimal window size and use that result for `pkurtosis`.

Create a chirp signal with white Gaussian noise.

```
fs = 1000;
t = 0:1/fs:10;
f1 = 300;
f2 = 400;
x = chirp(t,f1,10,f2)+randn(1,length(t));
```

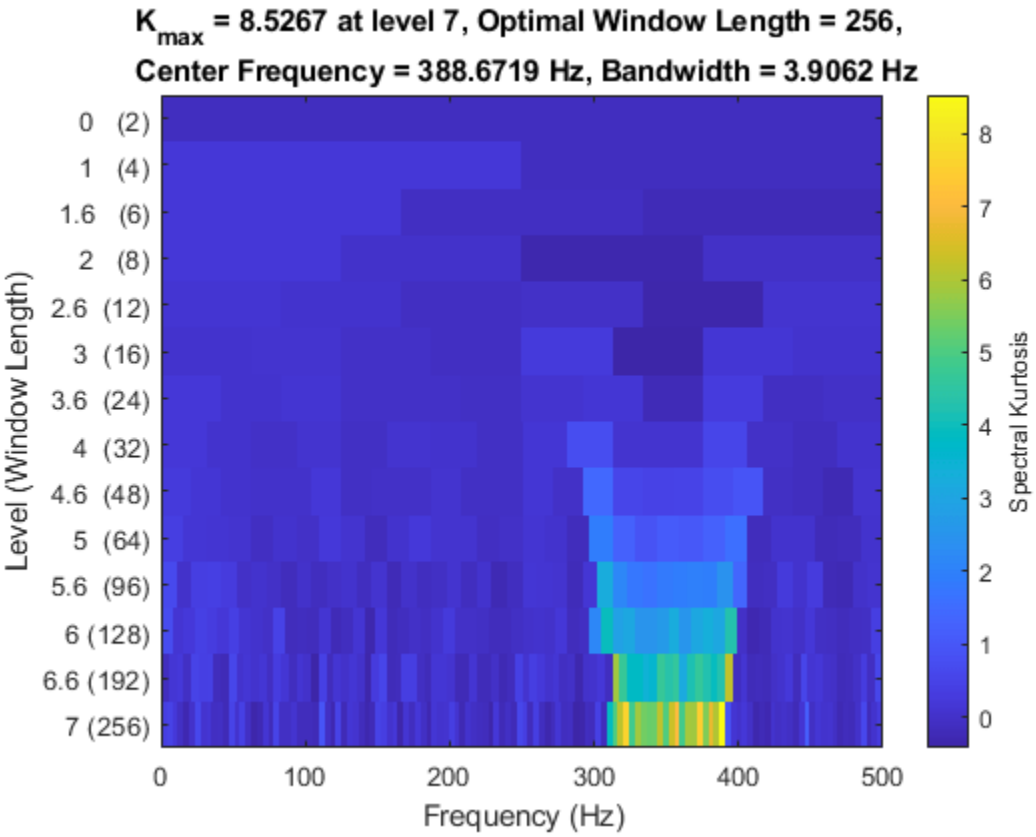
Plot the spectral kurtosis with the default window size.

```
pkurtosis(x,fs)
title('Spectral Kurtosis with Default Window Size')
```



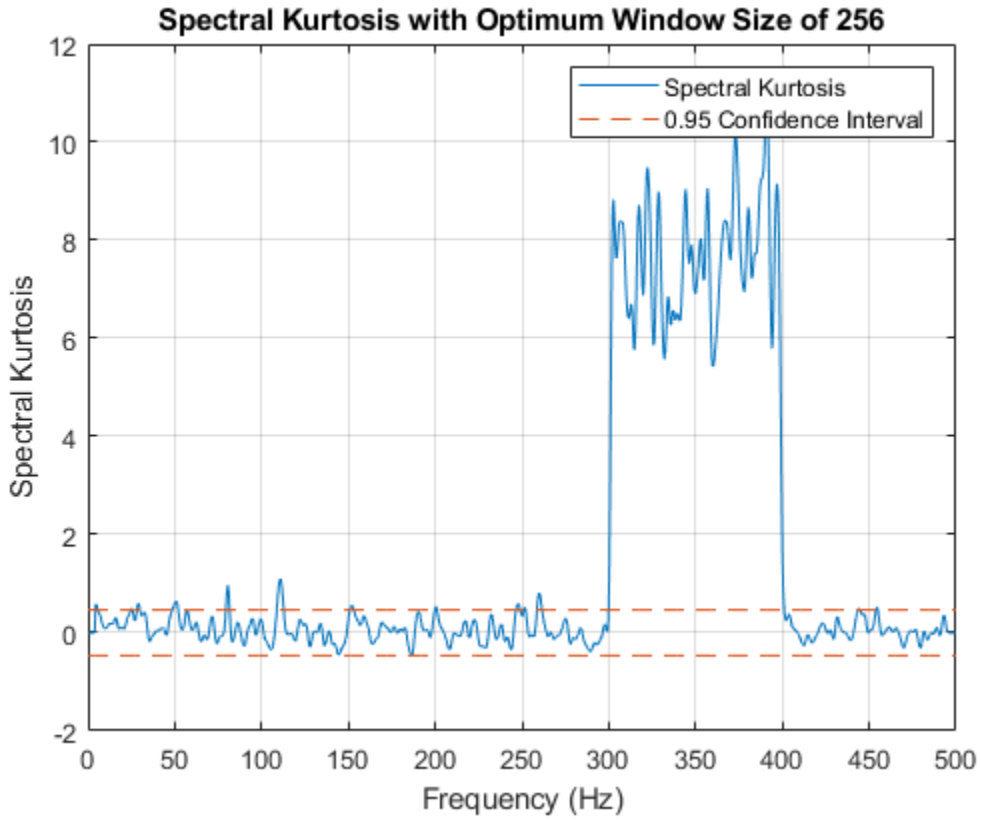
Now compute the optimal window size using `kurtogram`.

```
kurtogram(x,fs)
```



The kurtogram plot also illustrates the chirp between 300 and 400 Hz, and shows that the optimum window size is 256. Feed `w0` into `pkurtosis`.

```
w0 = 256;
pkurtosis(x,fs,w0)
title('Spectral Kurtosis with Optimum Window Size of 256')
```



The main excursion has higher kurtosis values. The higher values improve the differentiation between stationary and nonstationary components, and enhance your ability to extract the nonstationary component as a feature.

Plot Spectral Kurtosis Using a Customized Spectrogram

When using signal input data, `pkurtosis` generates a spectrogram by using `pspectrum` with default options. You can also create the spectrogram yourself if you want to customize the options.

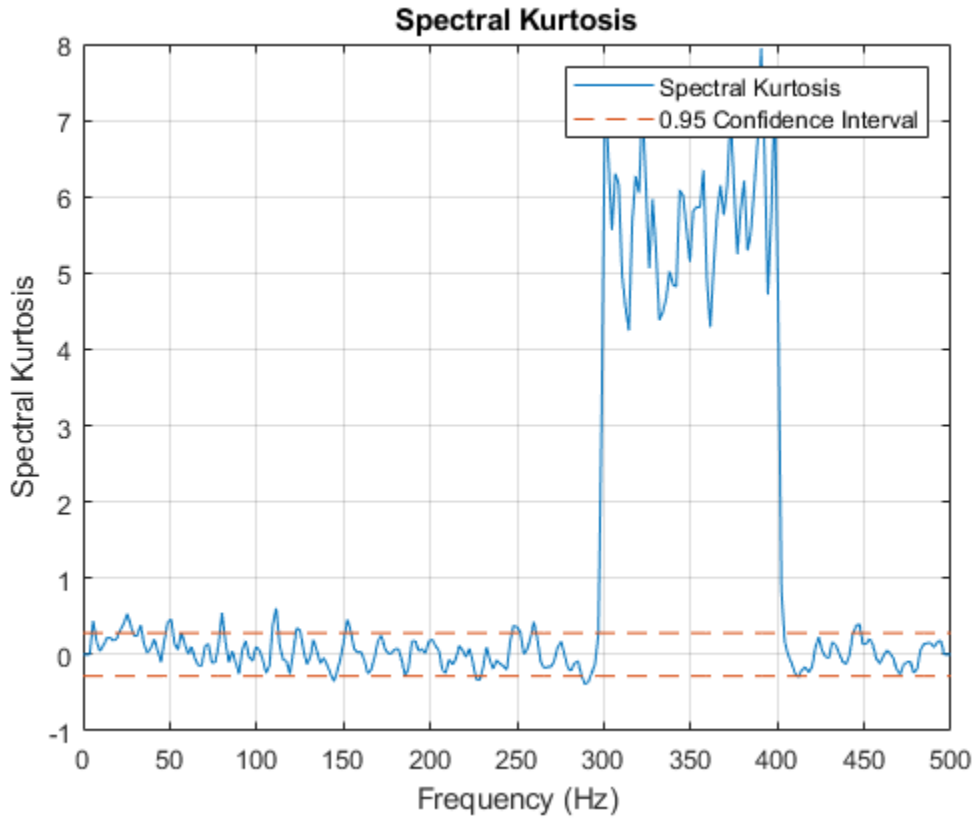
Create a chirp signal with white Gaussian noise.

```
fs = 1000;
t = 0:1/fs:10;
f1 = 300;
f2 = 400;
x = chirp(t,f1,10,f2)+randn(1,length(t));
```

Generate a spectrogram that uses your specification for window, overlap, and number of FFT points. Then use that spectrogram in `pkurtosis`.

```
window = 256;
overlap = round(window*0.8);
nfft = 2*window;
[s,f,t] = spectrogram(x,window,overlap,nfft,fs);
```

```
figure
pkurtosis(s,fs,f,window)
```



The magnitude of the excursion is higher, and therefore better differentiated, than with default inputs in previous examples. However, the excursion magnitude here is not as high as it is in the kurtogram-optimized window example.

Input Arguments

x — Time-series signal

vector

Time-series signal from which `pkurtosis` returns the spectral kurtosis, specified as a vector.

sampx — Sample rate or sample time of signal

normalized frequency (default) | positive numeric scalar | `duration` scalar | numeric vector in seconds | `duration` array | `datetime` array

Sample rate or sample time, specified as one of the following::

- Positive numeric scalar — frequency in hertz
- `duration` scalar — time interval between consecutive samples of `X`
- Vector, `duration` array, or `datetime` array — time instant or duration corresponding to each element of `x`

When `sampx` represents a time vector, time samples can be nonuniform, with the `pspectrum` constraint that the median time interval and the mean time interval must obey:

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

If you specify `sampx` as empty, then `pkurtosis` uses normalized frequency. In other words, it assumes an evenly spaced frequency vector spanning [0 π].

`xt` — Signal timetable
timetable

Signal timetable from which `pkurtosis` returns the spectral kurtosis, specified as a `timetable` that contains a single variable with a single column. `xt` must contain increasing, finite row times. If the `timetable` has missing or duplicate time points, you can fix it using the tips in "Clean Timetable with Missing, Duplicate, or Nonuniform Times". `xt` can be nonuniformly sampled, with the `pspectrum` constraint that the median time interval and the mean time interval must obey:

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

`window` — Window time resolution
positive scalar

Window time resolution to use for the internal `pspectrum` spectrogram computation, specified as a positive scalar in samples. `window` is required for syntaxes that use an existing spectrogram as input, and optional for the rest. You can use the function `kurtogram` to determine the optimal window size to use. `pspectrum` uses 80% overlap by default.

`s` — Spectrogram or power spectrogram of signal
complex matrix | real nonnegative matrix

Power spectrogram or spectrum of a signal, specified as a matrix (spectrogram) or a column vector (spectrum).

- If `s` is complex, then `pkurtosis` treats `s` as a short-time Fourier transform (STFT) of the original signal (spectrogram).
- If `s` is real, then `pkurtosis` treats `s` as the square of the absolute values of the STFT of the original signal (power spectrogram). Thus, every element of `s` must be nonnegative.

If you specify `s`, `pkurtosis` uses `s` rather than generate its own spectrogram or power spectrogram. For an example, see "Plot Spectral Kurtosis Using a Customized Spectrogram" on page 1-1613.

Data Types: `single` | `double`
Complex Number Support: Yes

`f` — Frequencies for `s`
vector

Frequencies for spectrogram or power spectrogram `s` when `s` is supplied explicitly to `pkurtosis`, specified as a vector in hertz. The length of `f` must be equal to the number of rows in `s`.

`'ConfidenceLevel',p` — Confidence level
0.95 (default) | [0 to 1]

Confidence level used to determine whether signal is likely to be Gaussian and stationary, specified as a numeric scalar value from 0 to 1. `p` influences the `thresh` range where the spectral kurtosis value indicates a Gaussian and stationary signal. The confidence level therefore provides a detection-sensitivity tuning parameter. Kurtosis values outside of this range indicate, with a probability of (1-`p`), non-Gaussian or nonstationary behavior. For an example, see “Plot Spectral Kurtosis of Nonstationary Signal Using Different Confidence Levels” on page 1-1606.

Output Arguments

`sk` — Spectral kurtosis

double vector

“Spectral Kurtosis” on page 1-1616, returned as a double vector. The spectral kurtosis is a statistical quantity that contains low values where data is stationary and Gaussian, and high values where transients occur. One use of the spectral kurtosis is to detect and locate nonstationary or non-Gaussian behavior that could result from faults or degradation. The high-valued kurtosis data reveals such signal components.

`fout` — frequencies for `sk`

double vector

Frequencies associated with `sk` values, returned as a vector in hertz.

`thresh` — Spectral kurtosis band size for stationary Gaussian behavior

scalar

Spectral kurtosis band size for stationary Gaussian behavior, specified as a numeric scalar representing the thickness of the band centered at the `sk = 0` line, given confidence level `p`. Excursions outside the `thresh`-delimited band indicate possible nonstationary or non-Gaussian behavior. Confidence level `p` directly influences the thickness of the band and the sensitivity of the results. For an example, see “Plot Spectral Kurtosis of Nonstationary Signal Using Different Confidence Levels” on page 1-1606.

More About

Spectral Kurtosis

Spectral kurtosis (SK) is a statistical tool that can indicate and pinpoint nonstationary or non-Gaussian behavior in the frequency domain, by taking:

- Small values at frequencies where stationary Gaussian noise only is present
- High positive values at frequencies where transients occur

This capability makes SK a powerful tool for detecting and extracting signals associated with faults in rotating mechanical systems. On its own, SK can identify features or conditional indicators for fault detection and classification. As preprocessing for other tools such as envelope analysis, SK can supply key inputs such as optimal band [2], [1].

The spectral kurtosis, or $K(f)$, of a signal $x(t)$ can be computed based on the short-time Fourier transform (STFT) of the signal, $S(t,f)$:

$$S(t,f) = \int_{-\infty}^{+\infty} x(t)w(t-\tau)e^{-2\pi ft}dt,$$

where $w(t)$ is the window function used in STFT. $K(f)$ is calculated as:

$$K(f) = \frac{\left\langle |S(t,f)|^4 \right\rangle}{\left\langle |S(t,f)|^2 \right\rangle^2} - 2, f \neq 0,$$

where $\langle \cdot \rangle$ is the time-average operator.

If the signal $x(t)$ contains only stationary Gaussian noise, then $K(f)$ at each frequency f has an asymptotic normal distribution with 0 mean and variance $4/M$, where M is the number of elements along the time axis in $S(t,f)$. Hence, a statistical threshold s_α given a confidence level α is:

$$s_\alpha = \Phi^{-1}(\alpha)\frac{2}{\sqrt{M}},$$

where Φ^{-1} is the quantile function of the standard normal distribution.

It is important to note that the STFT window length N_w directly drives frequency resolution, which is f_s/N_w, where f_s is the sample rate. The window size must be shorter than the spacing between transient impulses, but longer than the individual transient impulses.

References

[1] Antoni, J. "The Spectral Kurtosis: A Useful Tool for Characterising Non-Stationary Signals." *Mechanical Systems and Signal Processing*. Vol. 20, Issue 2, 2006, pp. 282–307.

[2] Antoni, J., and R. B. Randall. "The Spectral Kurtosis: Application to the Vibratory Surveillance and Diagnostics of Rotating Machines." *Mechanical Systems and Signal Processing*. Vol. 20, Issue 2, 2006, pp. 308–331.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `datetime` and duration arrays are not supported for code generation.
- Timetables are not supported for code generation.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If the input is a pure tone signal, the result may show small numerical differences with respect to in-memory computation of spectral kurtosis.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`kurtogram` | `pentropy` | `pspectrum`

Topics
“Rolling Element Bearing Fault Diagnosis” (Predictive Maintenance Toolbox)

Introduced in R2018a

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 `NaN`s or `NaT`s. These values are treated as missing data and 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 `NaN`s.

`[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 Δt is the time difference between the first and the last of them, then `pxx` is returned at `round(fmax / `$f_{\min}$`)` 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/`$f_{\min}$`)` frequency points, but the minimum frequency considered in this case is $1/(\texttt{ofac} \times N \times t_s)$. `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 `NaN`s 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

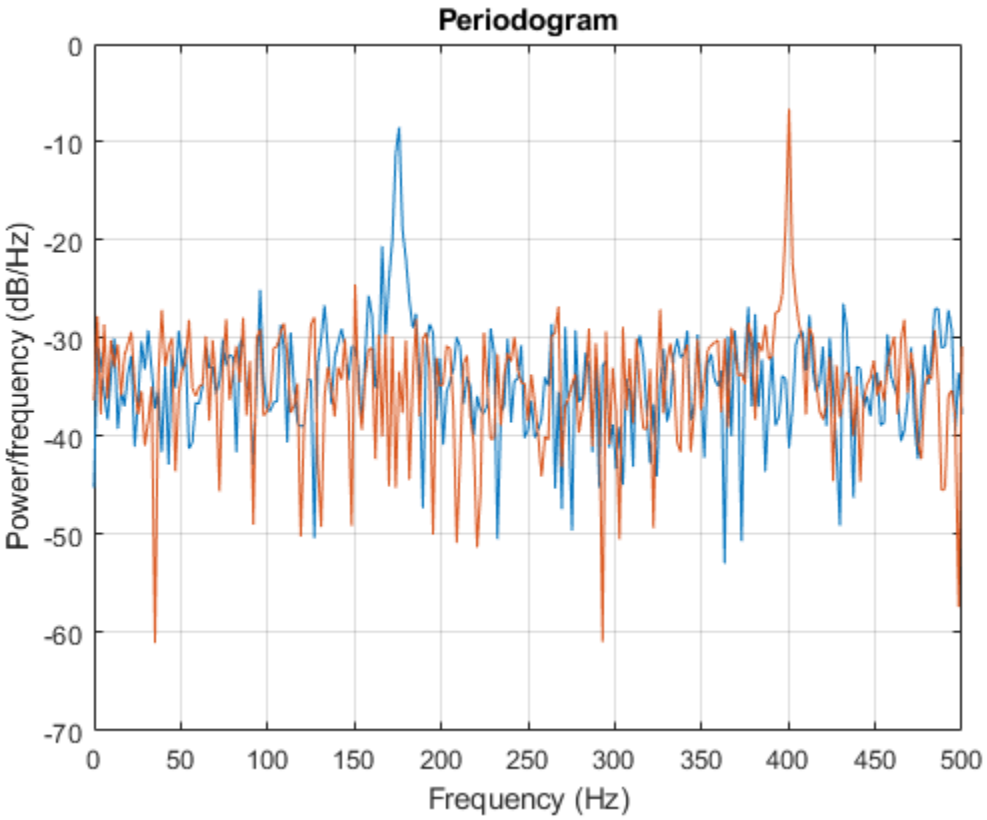
The Lomb-Scargle method can handle signals that have been sampled unevenly or have missing samples.

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;
```

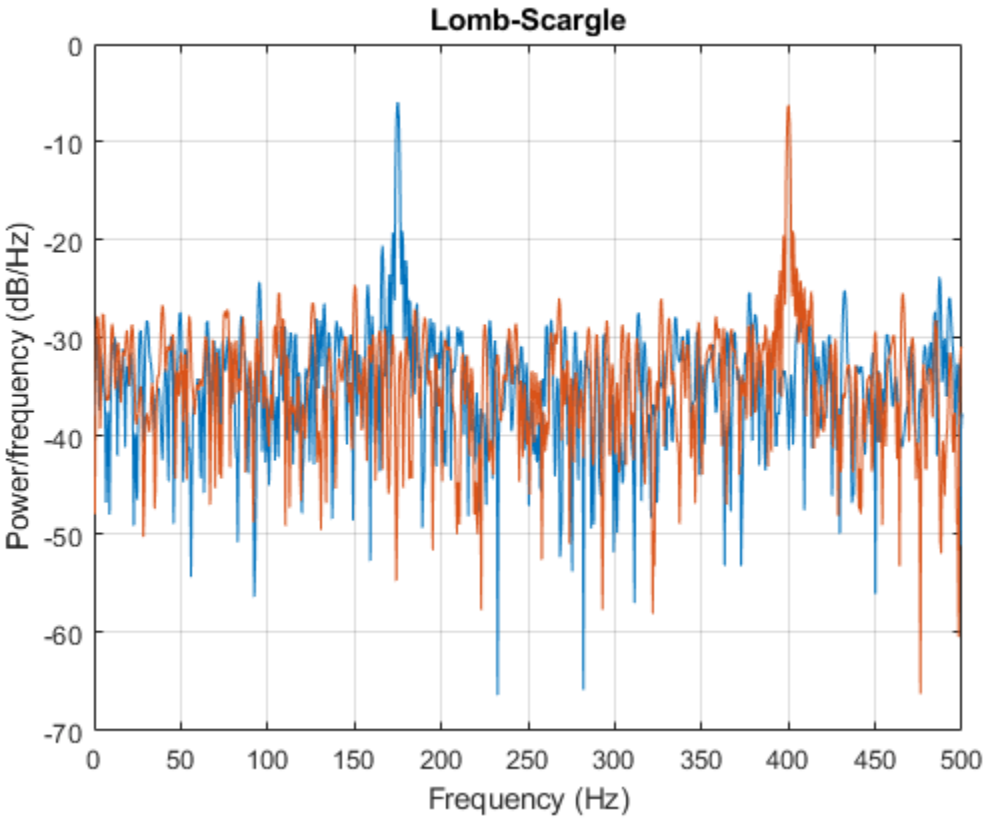
Compute and plot the periodogram of the signal. Use `periodogram` with the default settings.

```
periodogram(sigOrig,[],[],Fs)

axisLim = axis;
title('Periodogram')
```



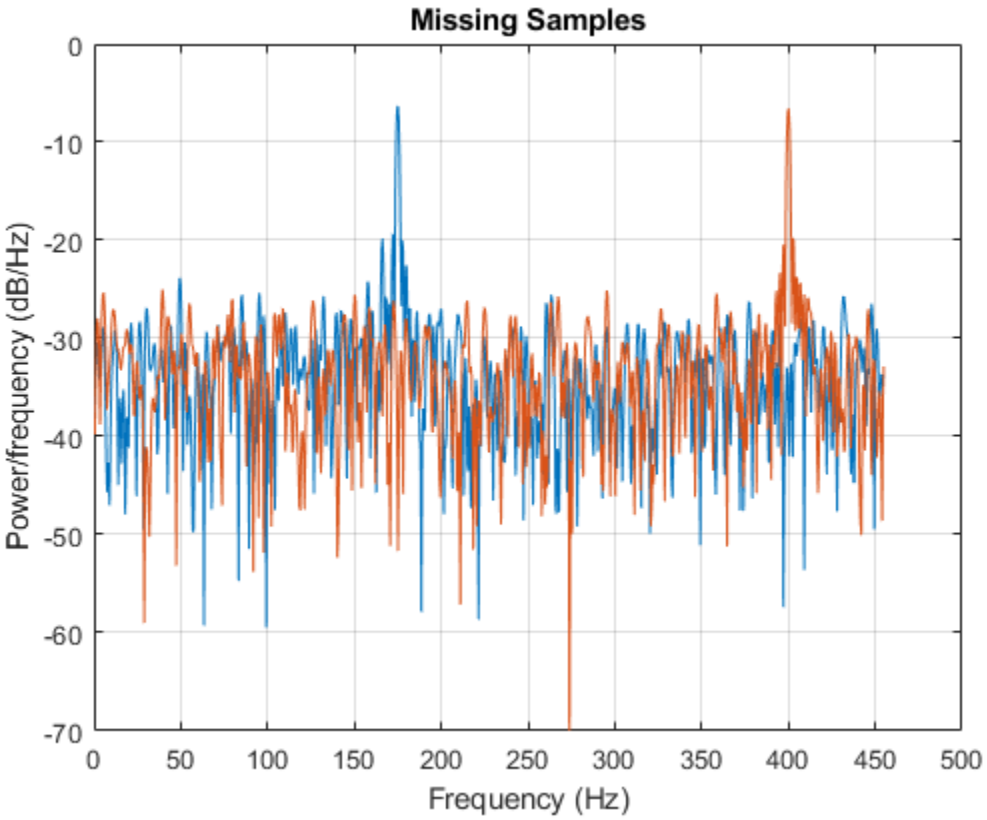
Use `plomb` with the default settings to estimate and plot the PSD of the signal. Use the axis limits from the previous plot.

```
plomb(sigOrig,t)

axis(axisLim)
title('Lomb-Scargle')
```



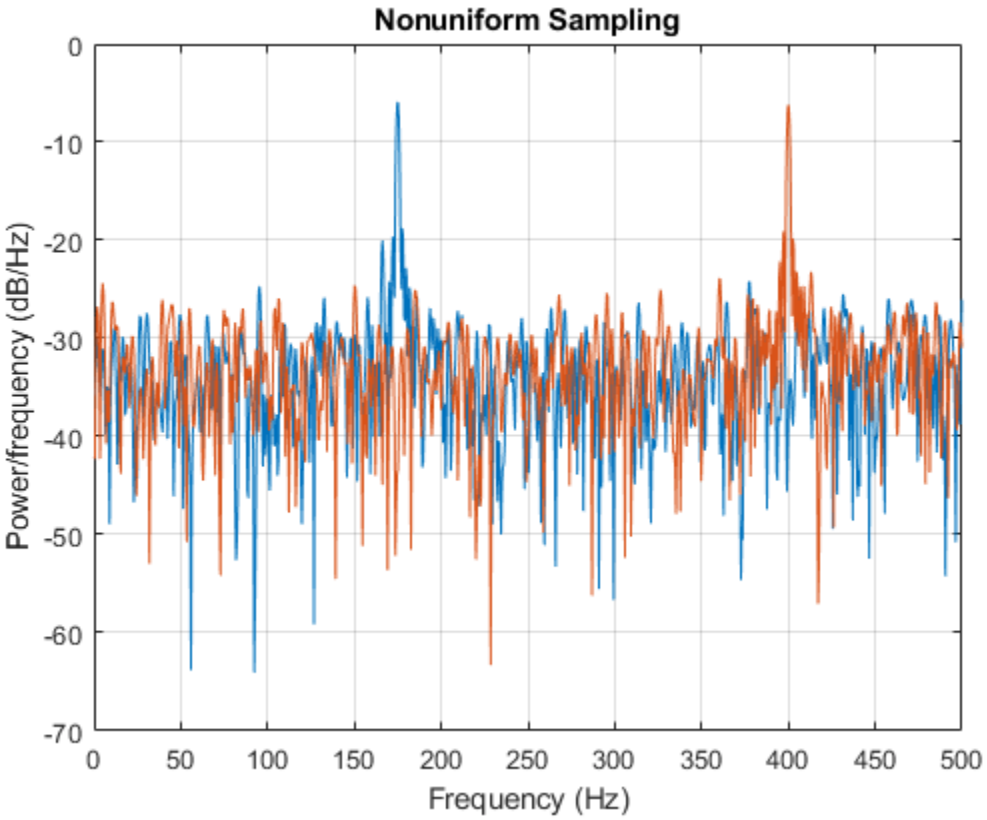
Suppose the signal is missing 10% of the original samples. Place `NaN`s in random locations to simulate the missing data points. Use `plomb` to estimate and plot the PSD of the signal with missing samples.

```
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')
```

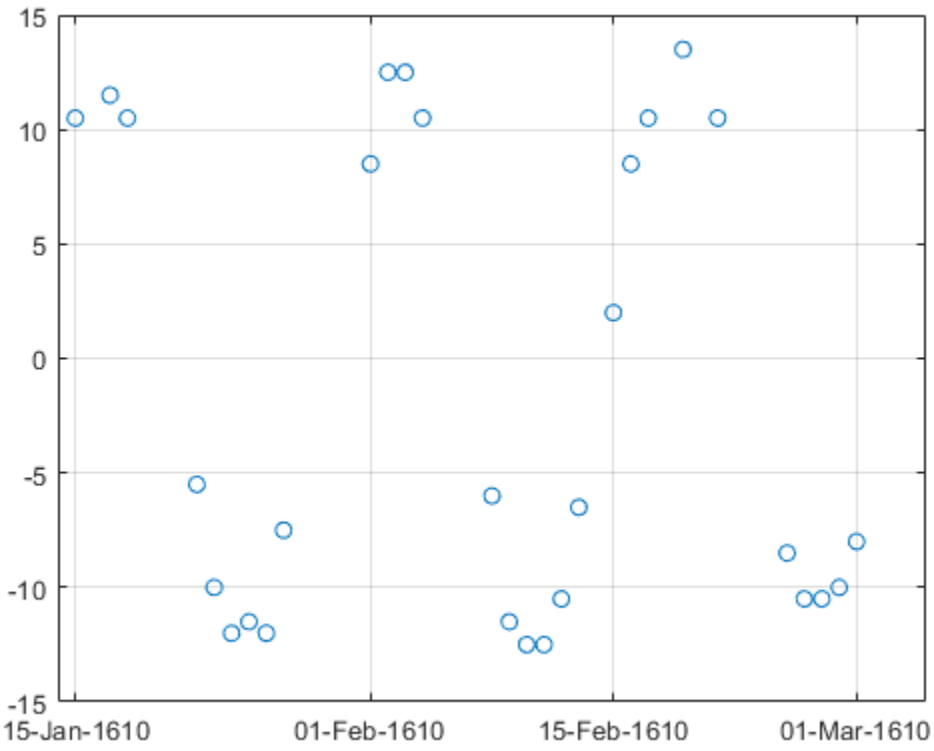


Periodogram of Data Set with Missing Samples

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 NaNs. The first observation is dated January 15. Generate a `datetime` array of observation times.

```
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]';

obsv = datetime(1610,1,14+(1:length(yg)));

plot(yg,'o')

ax = gca;
nights = [1 18 32 46];
ax.XTick = nights;
ax.XTickLabel = char(obsv(nights));
grid
```

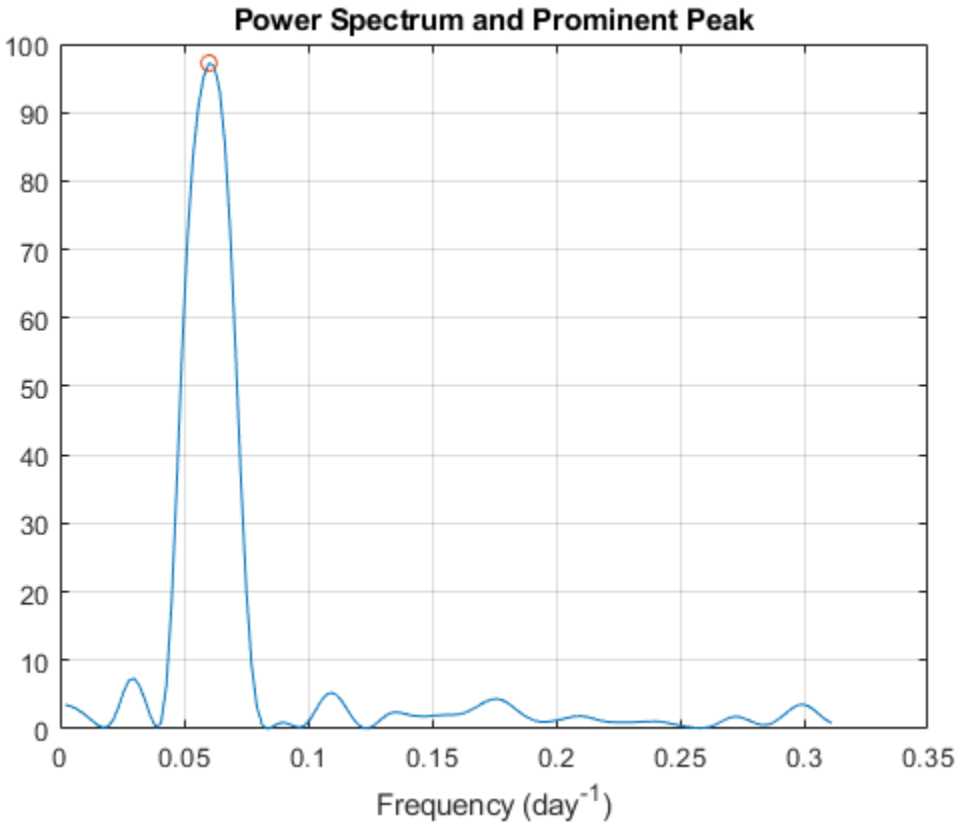


Estimate the power spectrum of the data using `plomb`. Specify an oversampling factor of 10. Express the resulting frequencies in inverse days.

```
[pxx,f] = plomb(yg,obsv,[],10,'power');
f = f*86400;
```

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})')
title('Power Spectrum and Prominent Peak')
grid
```



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
```

```
Period = 16.6454
```

```
NASA = 16.6890184;
PercentDiscrep = (Period-NASA)/NASA*100
```

```
PercentDiscrep = -0.2613
```

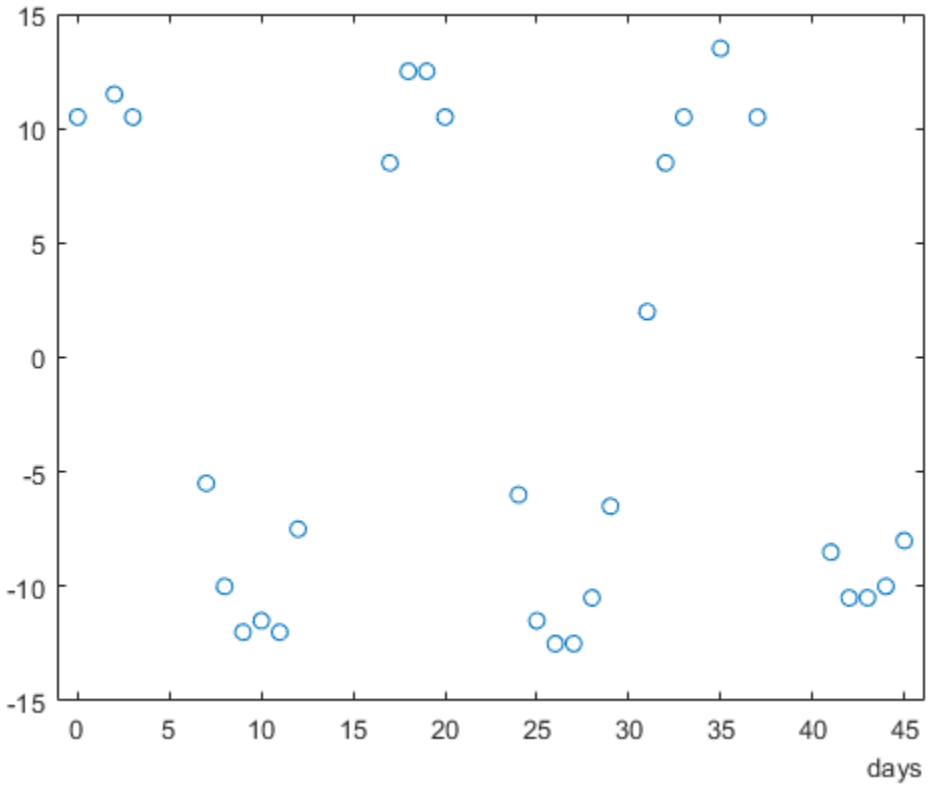
Periodogram of Data Set with Irregular Sampling

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. Generate a `duration` array of observation times.

```
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]';
td = days(t);
```

```
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(td,yg,'o')
```

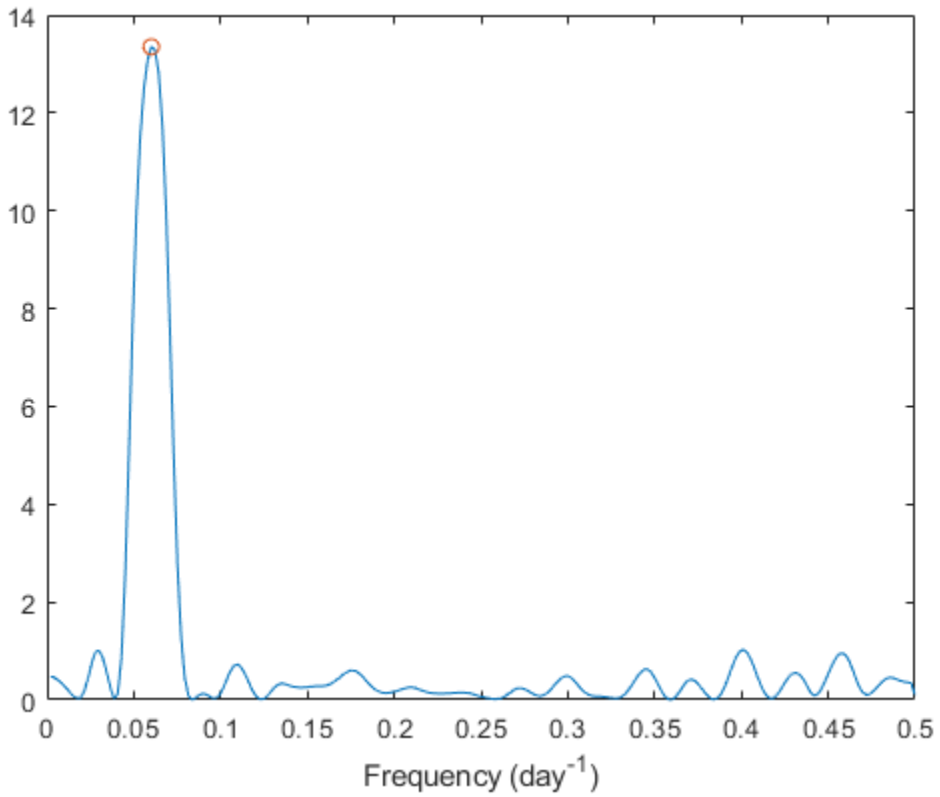


Use `plomb` to compute the periodogram of the data. Estimate the power spectrum up to a frequency of 0.5 day^{-1}. Specify an oversampling factor of 10. Choose the standard Lomb-Scargle normalization.

```
oneday = seconds(days(1));

[pxx,f] = plomb(yg,td,0.5/oneday,10,'normalized');

f = f*oneday;
```

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

$$y(t) = A + B\cos 2\pi f_0 t + C\sin 2\pi f_0 t\,.$$

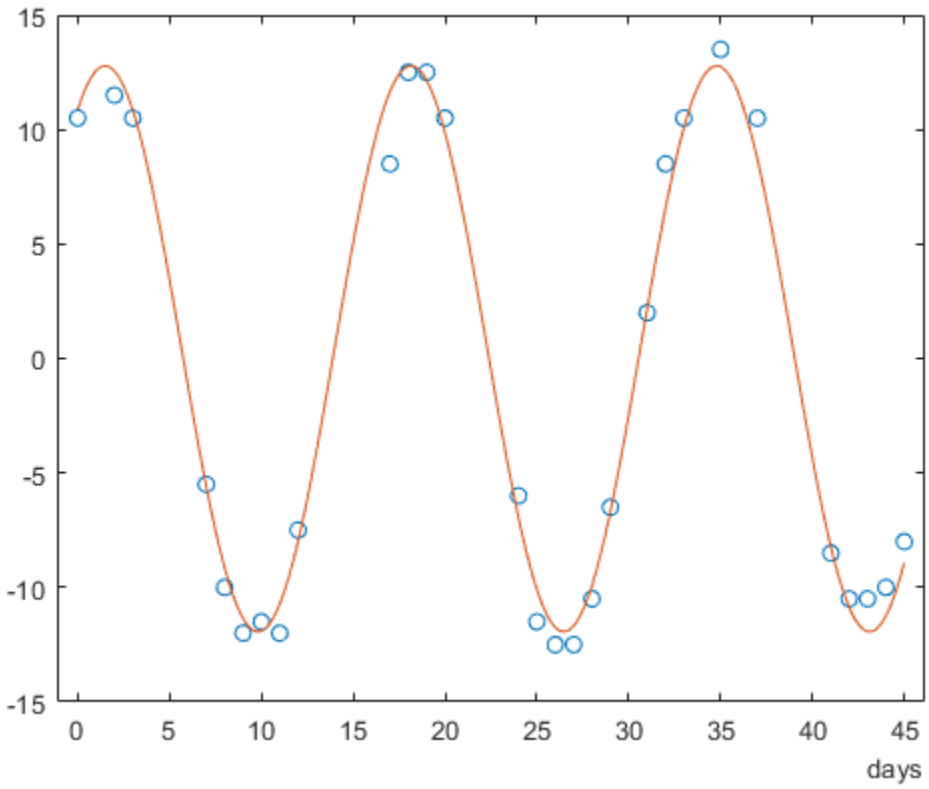
The fitting parameters are the amplitudes A, B, and C.

```
ft = 2*pi*f0*t;

ABC = [ones(size(ft)) cos(ft) sin(ft)] \ yg
```

```
ABC = 3×1

    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(td,yg,'o',days(x),y)
```



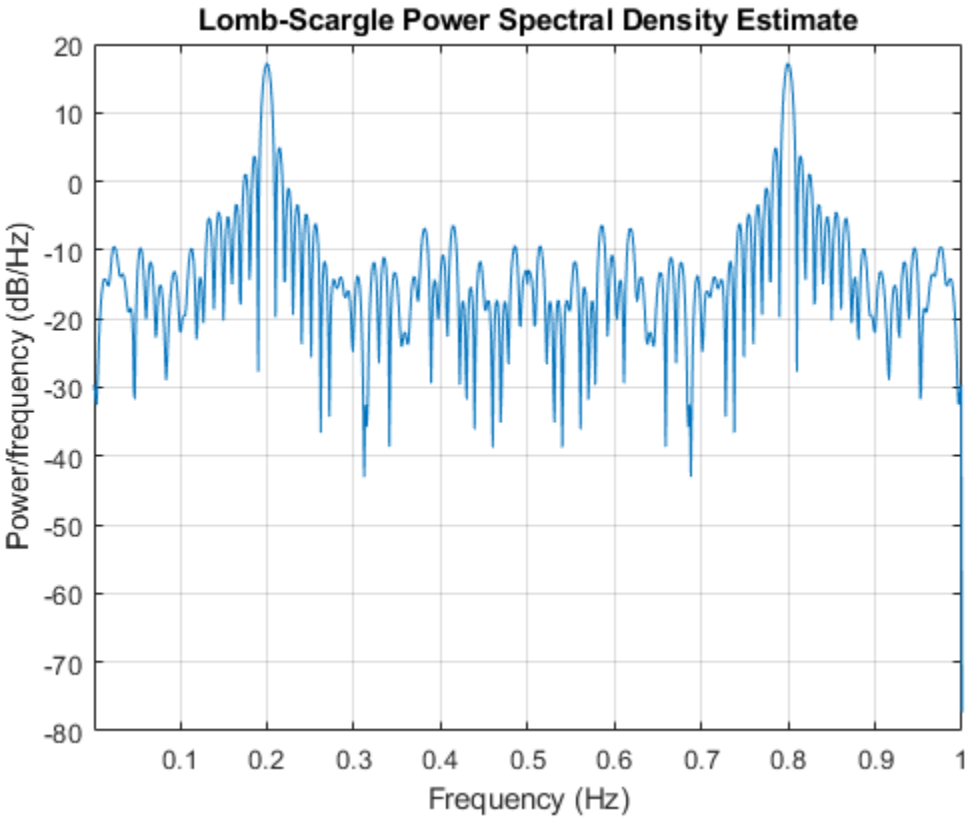
Irregular Sampling and Aliasing

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.

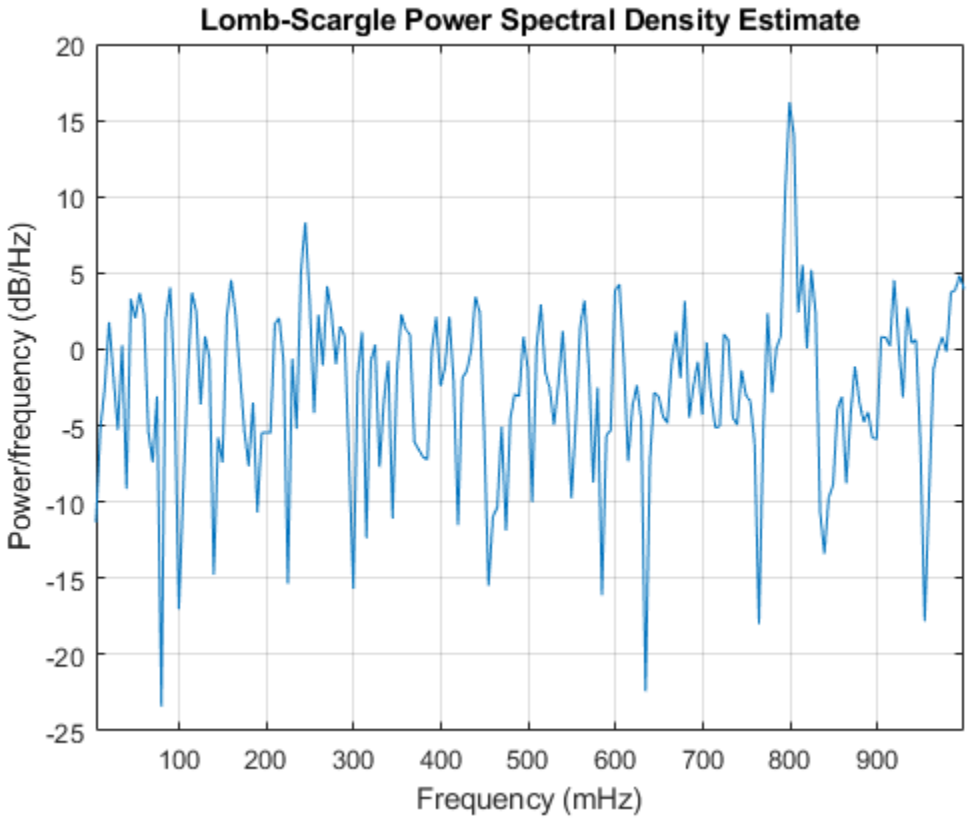
```
plomb(s,ts,1,10)
```



There is aliasing because 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.

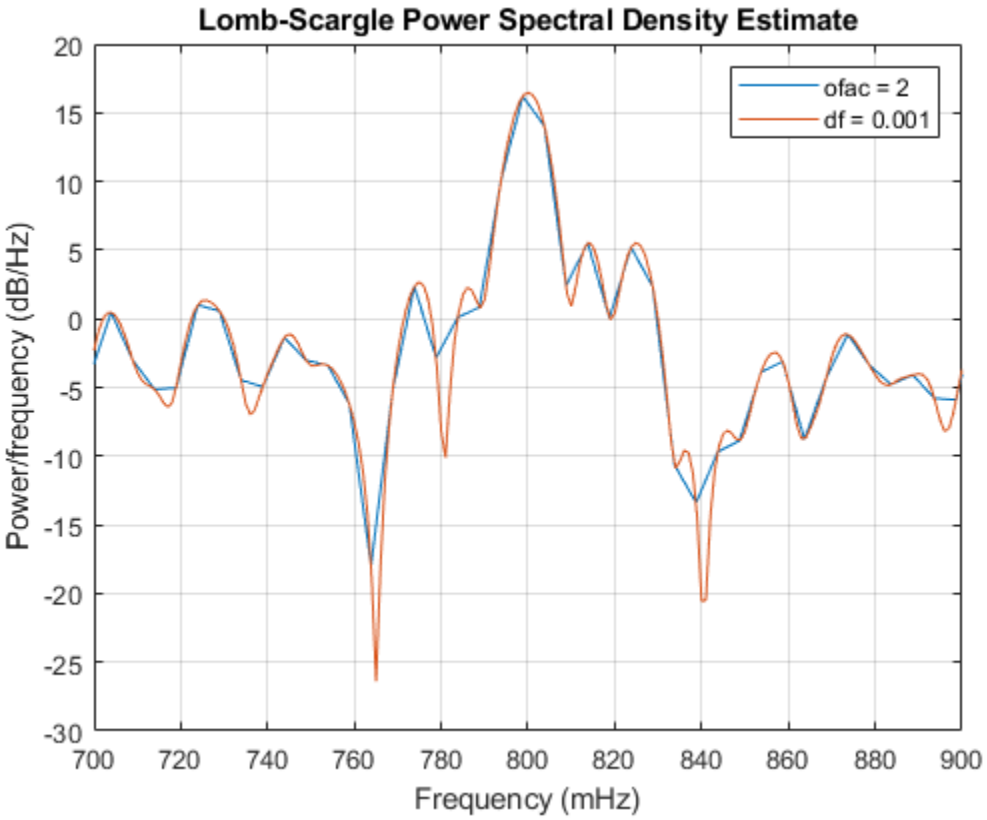
```
tn = sort(100*rand(1,100));
n = sin(2*pi*f0*tn) + wgn;

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

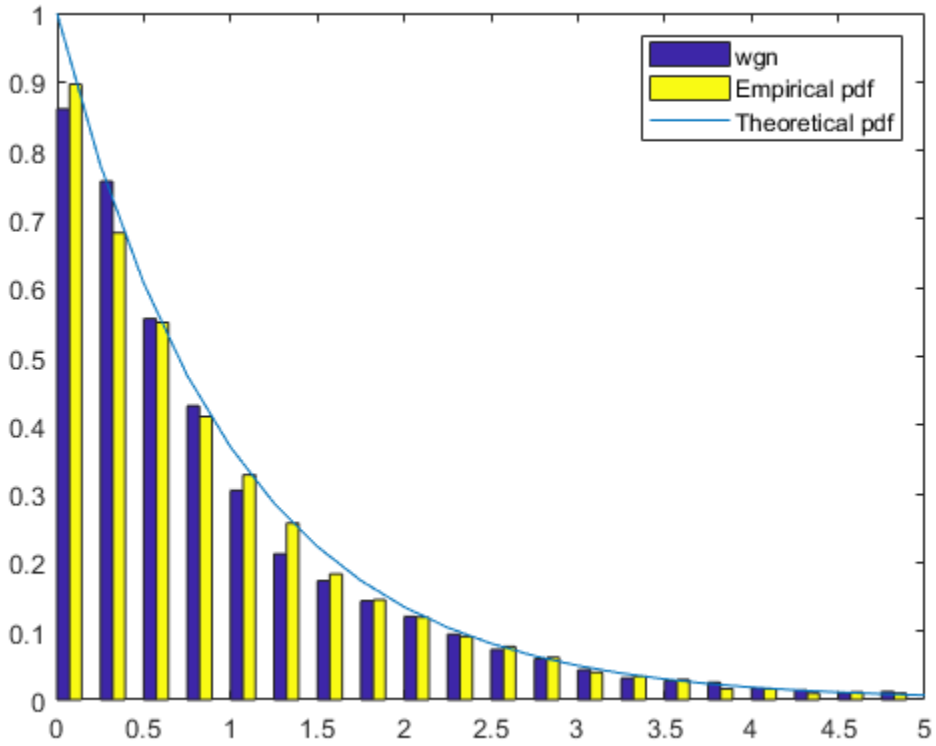
Generate $N = 1024$ samples of white noise with variance $\sigma = 1$, given a sample rate of 1 Hz. Compute the power spectrum of the white noise. Choose the Lomb-Scargle normalization and specify an oversampling factor ofac = 15. Reset the random number generator for repeatable results.

```
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 $N \times$ ofac/2. Specify a bin width of 0.25. Overlay a plot of the theoretical distribution function.

```
dx = 0.25;
br = 0:dx:5;
```

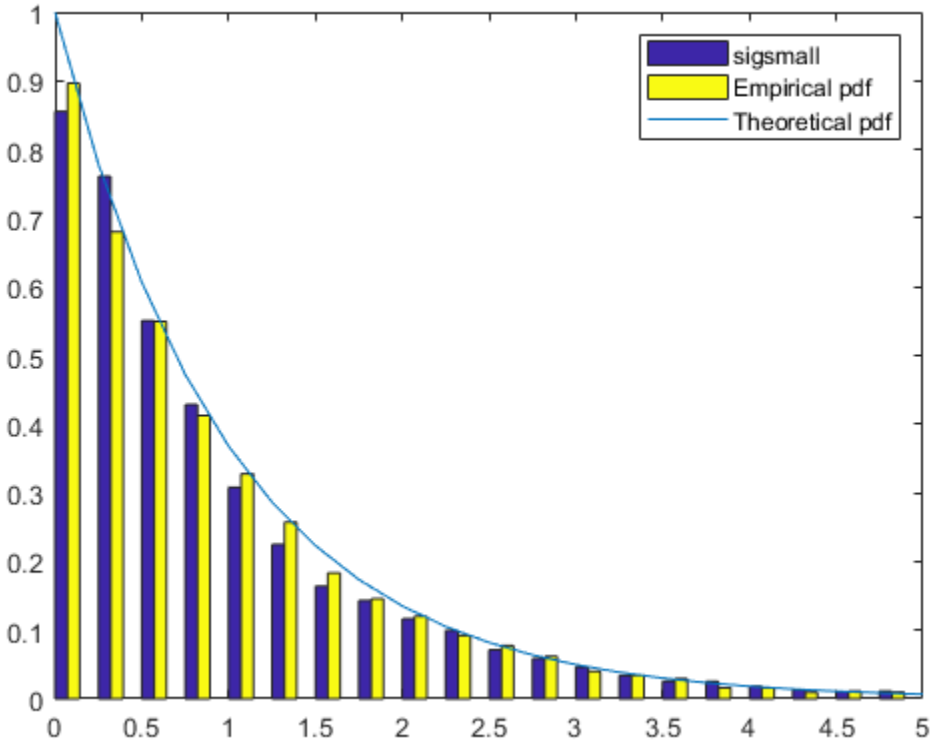
```
Nf = N*ofac/2;

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.

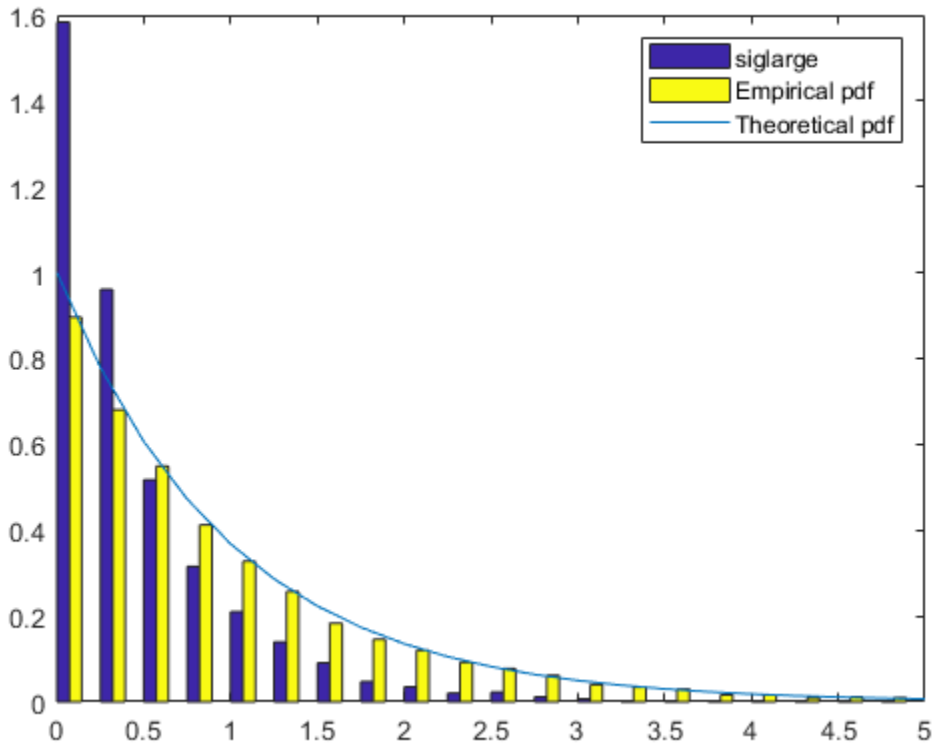
```
SNR = 0.01;
x0 = sqrt(2*SNR);
sigsmall = wgn + x0*sin(2*pi/10*t);

[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
```



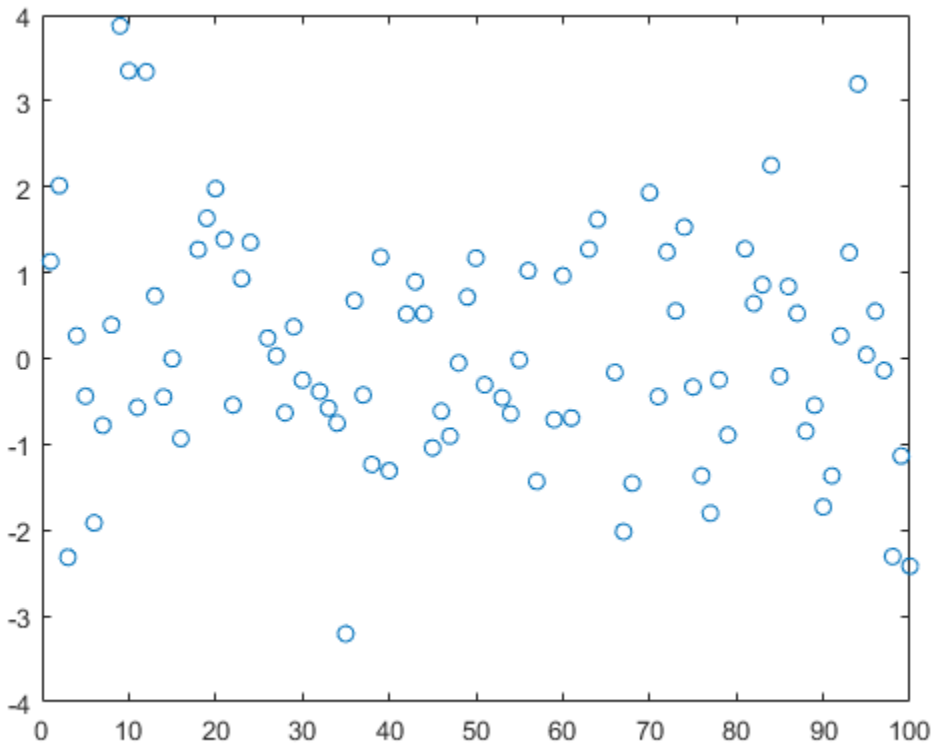
False-Alarm Probabilities

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);
```

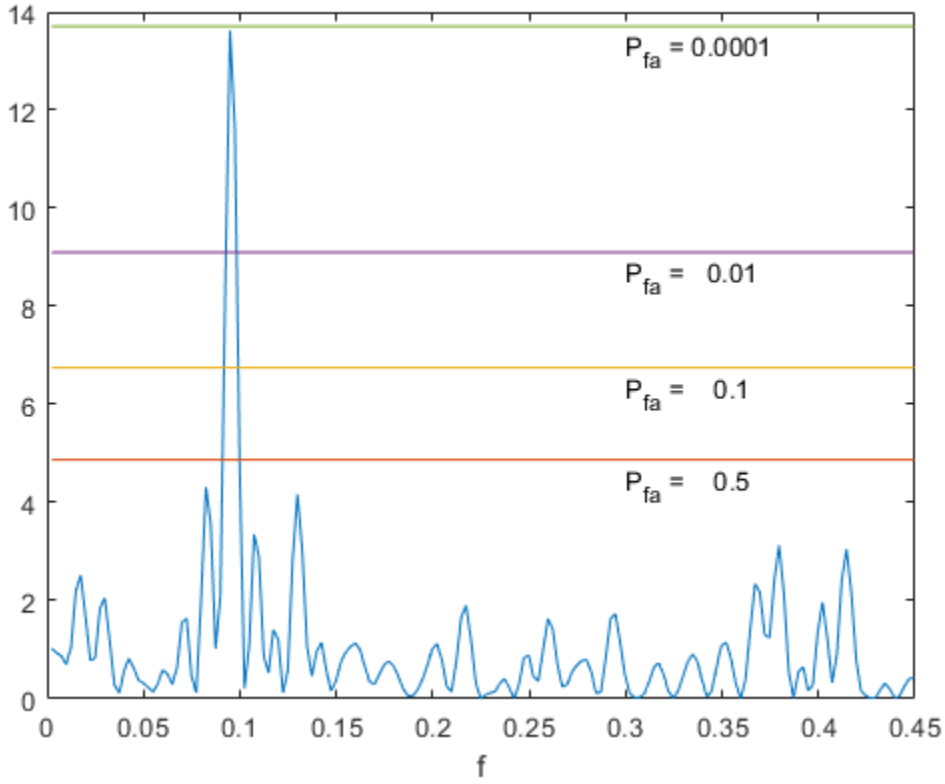
Discard 10% of the samples at random. Plot the signal.

```
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.

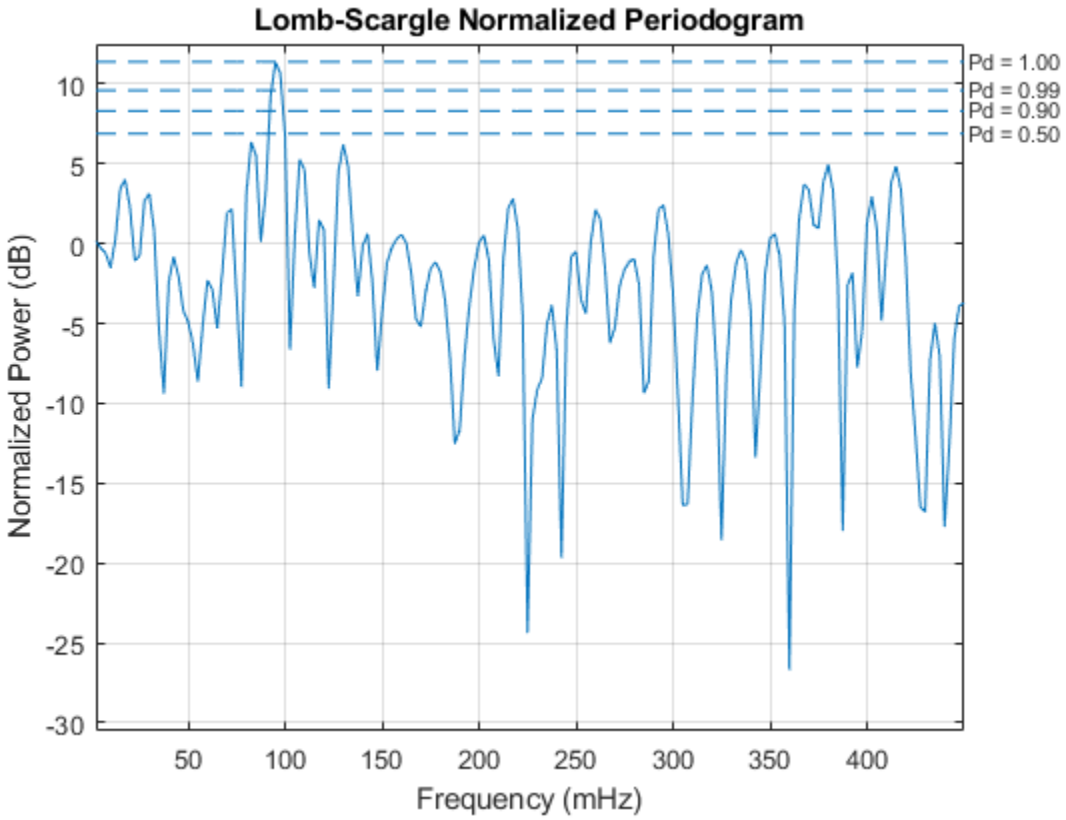
```
Pfa = [50 10 1 0.01]/100;
Pd = 1-Pfa;

[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;
```

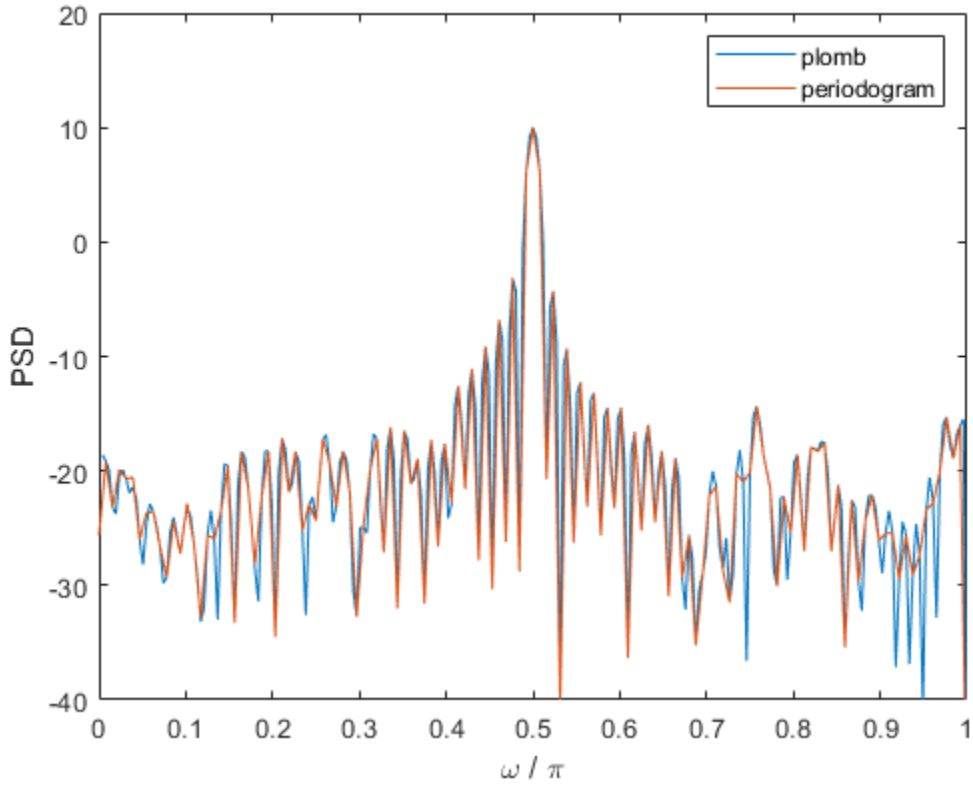
Estimate the PSD using the Lomb-Scargle procedure. Repeat the calculation with `periodogram`.

```
[p,f] = plomb(x);
[pper,fper] = periodogram(x);
```

Plot the PSD estimates in decibels. Verify that the results are equivalent.

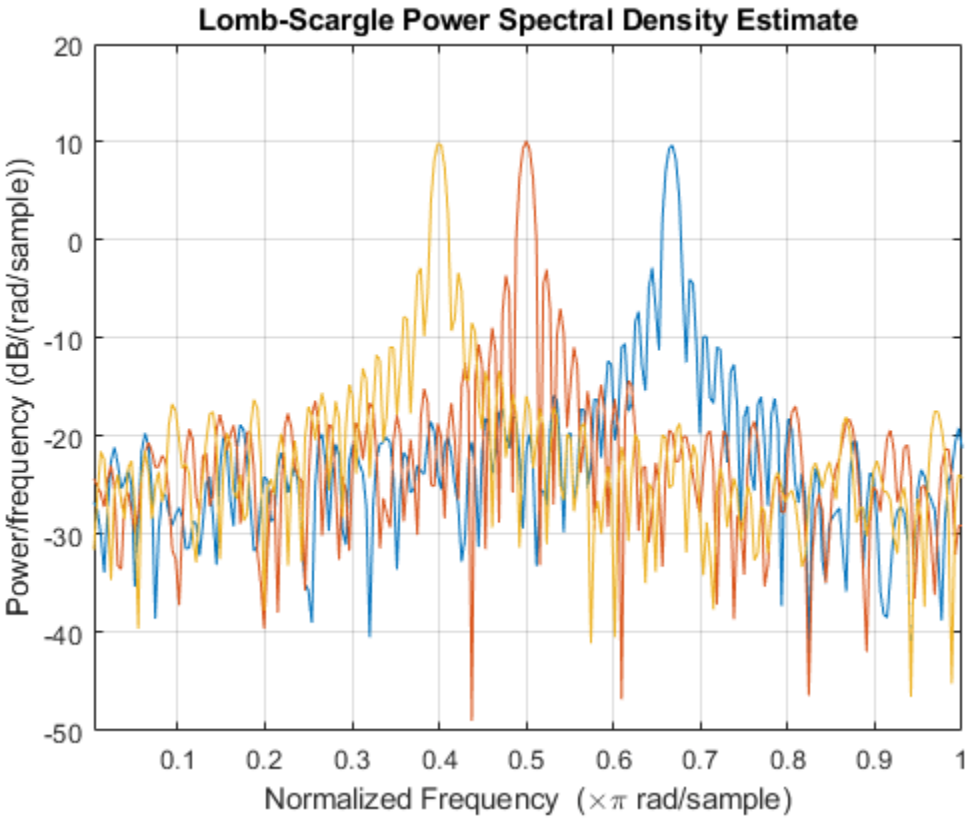
```
plot(f/pi,pow2db(p))
hold on
plot(fper/pi,pow2db(pper))

axis([0 1 -40 20])
xlabel('\omega / \pi')
```

```
ylabel('PSD')
legend('plomb','periodogram')
```



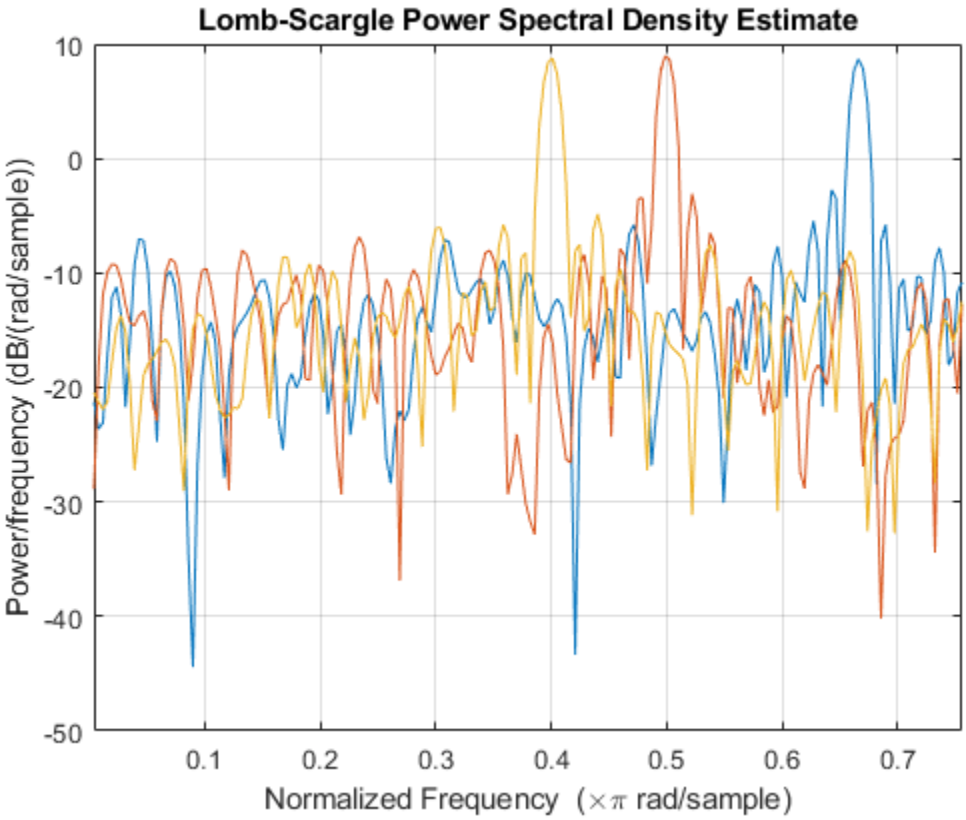
Estimate the Lomb-Scargle PSD of a three-channel signal composed of sinusoids. Specify the frequencies as $2\pi/3$ rad/sample, $\pi/2$ rad/sample, and $2\pi/5$ rad/sample. Add white Gaussian noise of variance 1/100. Use `plomb` with no output arguments to compute and plot the PSD estimate in decibels.

```
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)
```

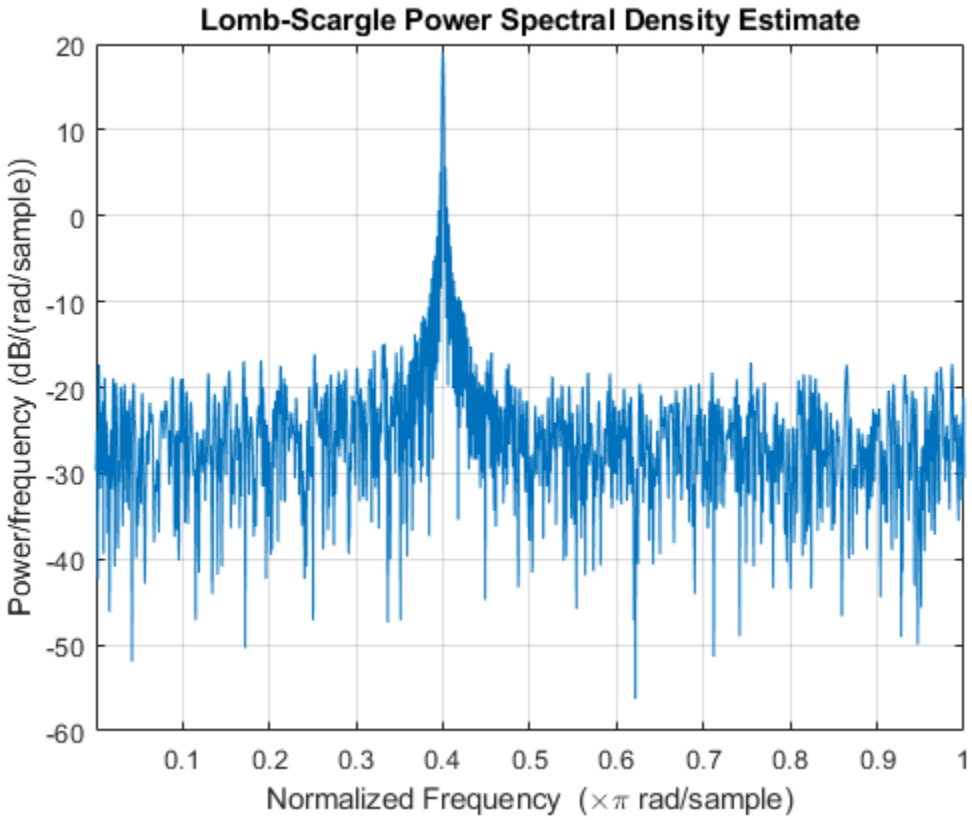


Power Spectral Density of Signal with Missing Samples

If you do not have a time vector, use `NaN`'s to specify missing samples in a signal.

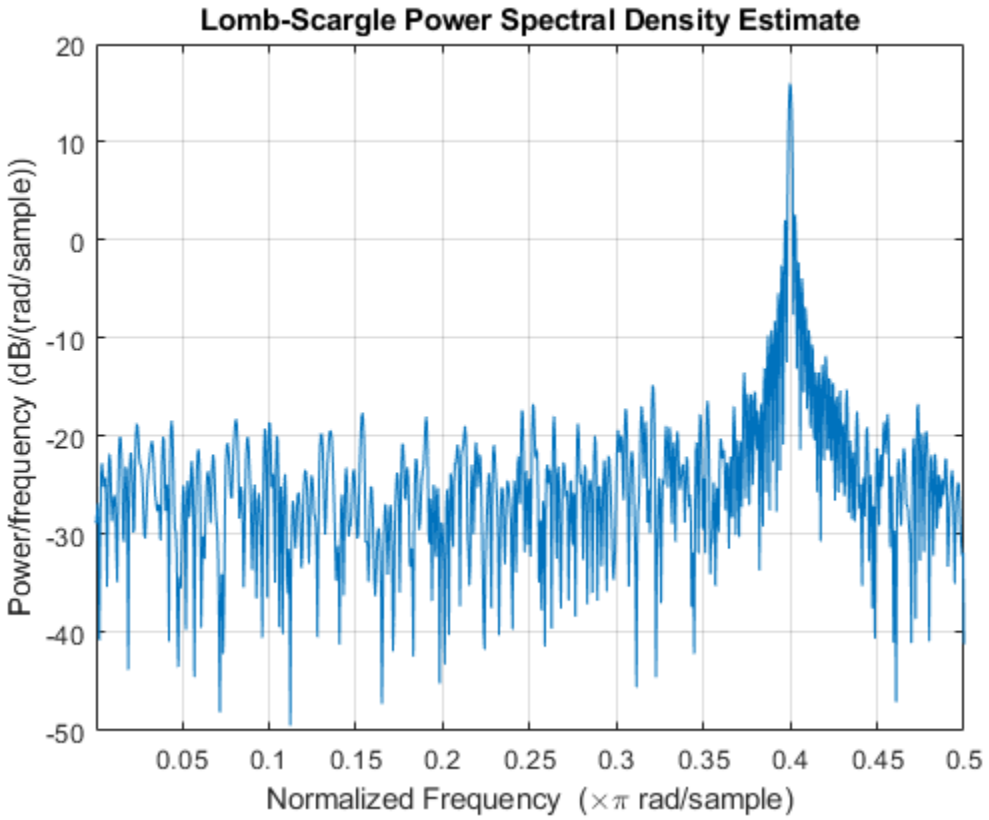
Generate 1024 samples of a sinusoid of normalized frequency $2\pi/5$ rad/sample embedded in white noise of variance 1/100. Estimate the power spectral density using the Lomb-Scargle procedure. Use `plomb` with no output arguments to plot the estimate.

```
t = (0:1023)';
x = sin(2*pi*t/5);
x = x + randn(size(x))/10;

[pxx,f] = plomb(x);

plomb(x)
```



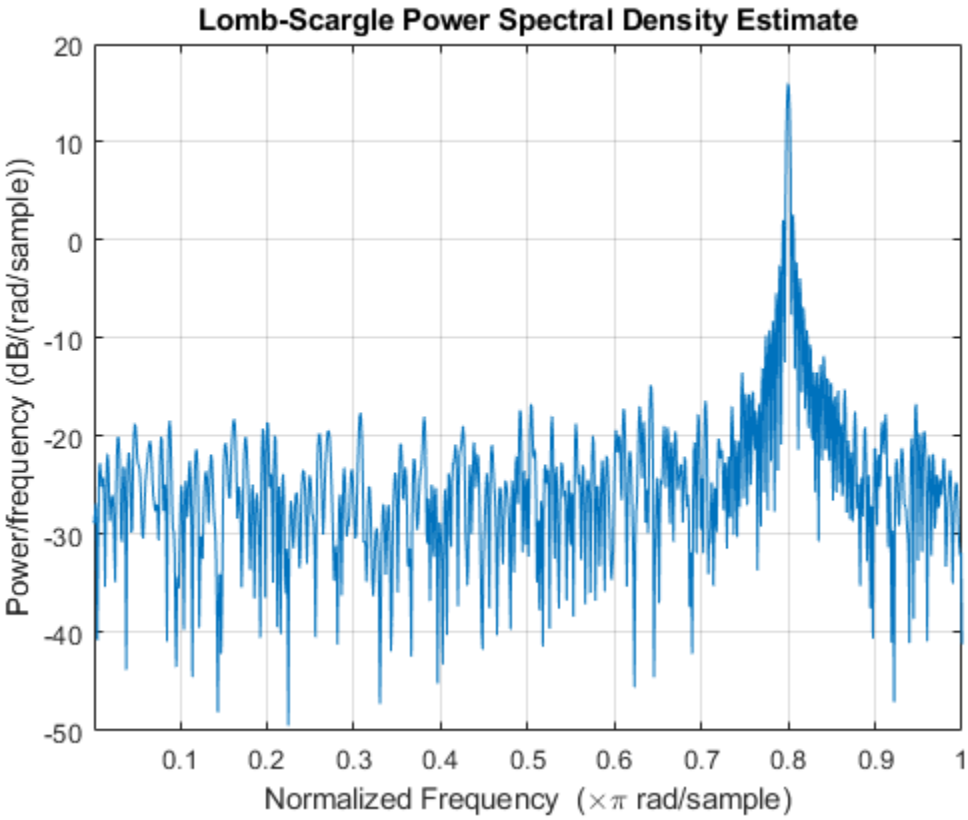
Remove every other sample by assigning `NaN`'s. Use `plomb` to compute and plot the PSD. The periodogram peaks at the same frequency because the time axis is unchanged.

```
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)
```



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 `NaN`s. `NaN`s are treated as missing data and are excluded from the spectrum computation.

Data Types: `single` | `double`

t — Time instants

nonnegative real vector | `datetime` array | `duration` array

Time instants, specified as a nonnegative real vector, a `datetime` array, or a `duration` array. `t` must increase monotonically but need not be uniformly spaced. `t` can contain `NaN`s or `NaT`s. These values are treated as missing data and excluded from the spectrum computation.

Data Types: `single` | `double` | `datetime` | `duration`

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`.

`pdvec` — 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 $k = 1, ..., N$. The Lomb-Scargle periodogram is defined by [2]

$$P_{\mathrm{LS}}(f) = \frac{1}{2\sigma^2}\left\{\frac{\left[\sum_{k=1}^{N}(x_k-\bar{x})\cos(2\pi f(t_k-\tau))\right]^2}{\sum_{k=1}^{N}\cos^2(2\pi f(t_k-\tau))} + \frac{\left[\sum_{k=1}^{N}(x_k-\bar{x})\sin(2\pi f(t_k-\tau))\right]^2}{\sum_{k=1}^{N}\sin^2(2\pi f(t_k-\tau))}\right\},$$

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, τ, 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 $t_k \to t_k + T$ in the time measurements results in an identical shift in the offset: $\tau \to \tau + T$. 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." [4] 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] 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.

[2] Lomb, Nicholas R. "Least-Squares Frequency Analysis of Unequally Spaced Data." *Astrophysics and Space Science.* Vol. 39, 1976, pp. 447–462.

[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] 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Timetables are not supported for code generation.

See Also

`bandpower` | `pburg` | `pcov` | `peig` | `periodogram` | `pmcov` | `pmtm` | `pmusic` | `pwelch` | `pyulear` | `spectrogram`

Topics

“Detect Periodicity in a Signal with Missing Samples”
“Spectral Analysis of Nonuniformly Sampled Signals”

Introduced in R2014b

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`. If you omit `nfft`, or specify it as empty, then `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, because otherwise the function interprets it as `nfft`.

`[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, because otherwise the function interprets it as `nfft`. 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.

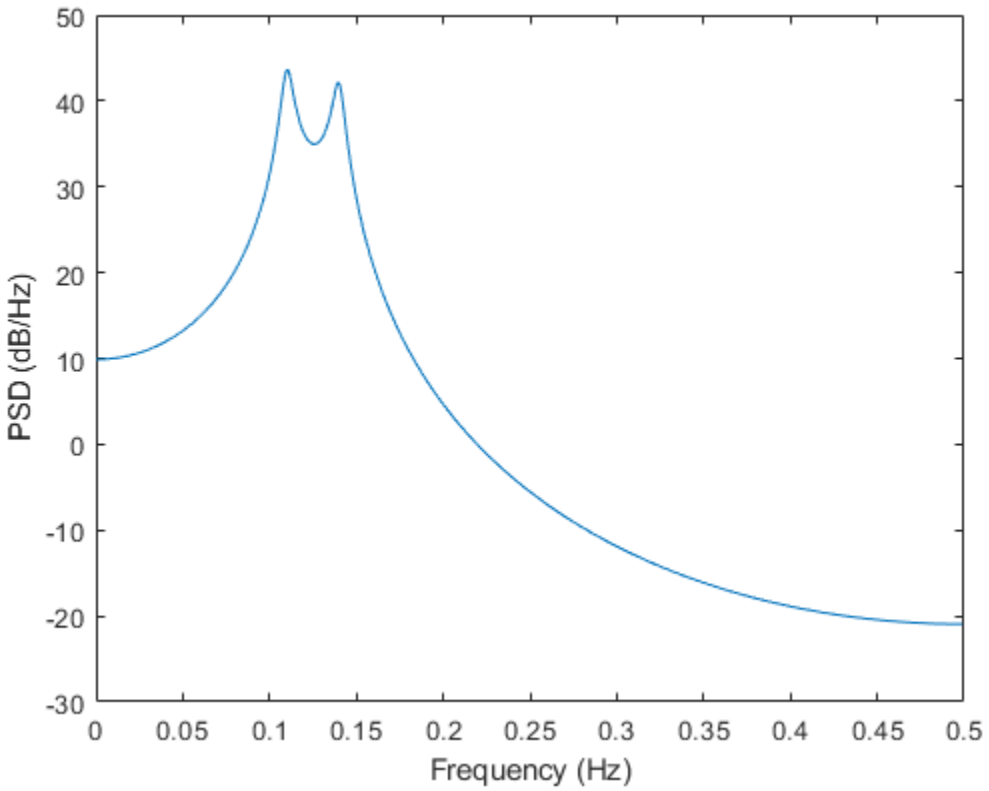
Examples

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.

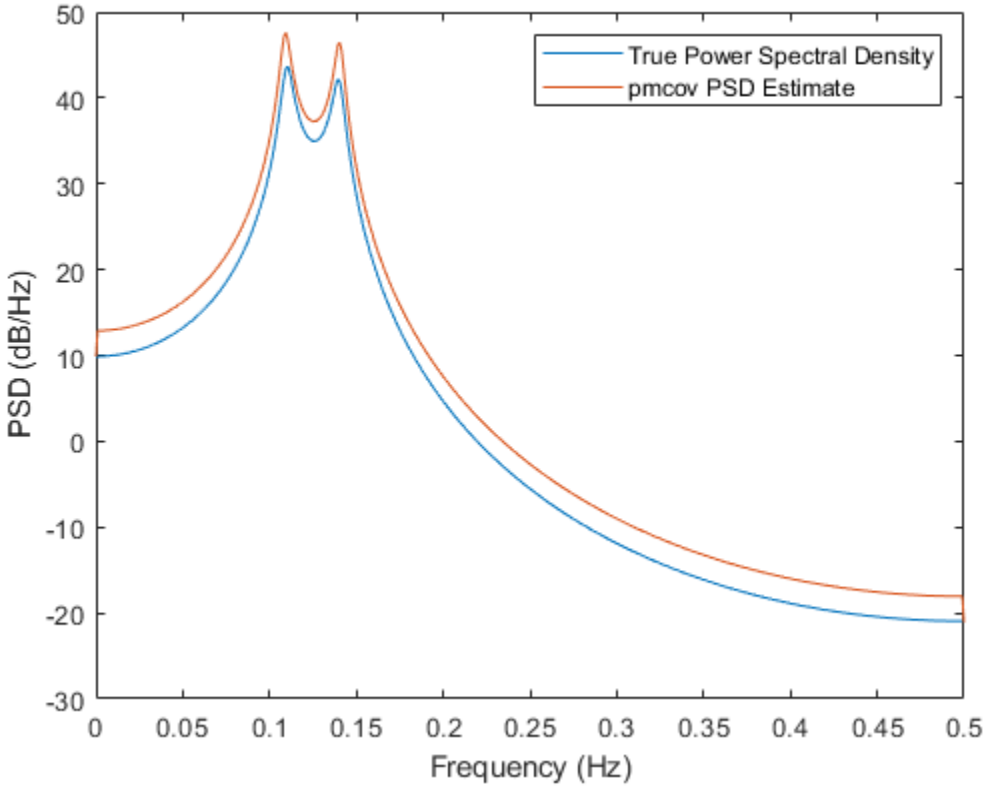
Create an AR(4) system function. Obtain the frequency response and plot the PSD of the system.

```
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')
```



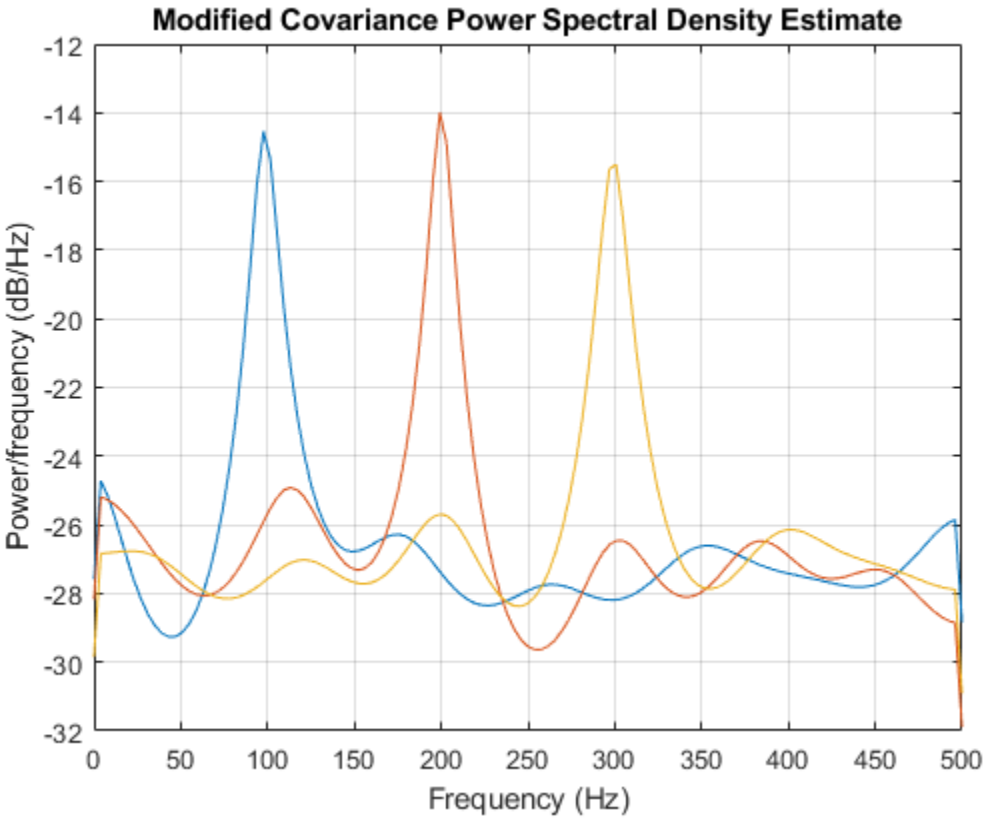
Modified-Covariance PSD Estimate of a Multichannel Signal

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;

pmcov(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` — 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 has units of Hz.

`w` — Normalized frequencies

vector

Normalized frequencies, specified as a row or column vector with at least two elements. Normalized frequencies are in rad/sample.

Example: `w = [pi/4 pi/2]`

Data Types: `double`

`f` — Frequencies

vector

Frequencies, specified as a row or column vector with at least two elements. The frequencies are in cycles per unit time. The unit time is specified by the sample rate, `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,π] rad/sample. If `nfft` is odd, the length of `pxx` is (`nfft` + 1)/2 and the interval is [0,π) 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.

  The function multiplies the power by 2 at all frequencies except 0 and the Nyquist frequency to conserve the total power.
- `'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 [0,2π) 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 (–π,π] rad/sample for even length `nfft` and (–π,π) 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.

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 × 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 Ω and specify the sample rate 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,π] if nfft is even and [0,π) if nfft is odd. If pxx is a two-sided PSD estimate, w spans the interval [0,2π). For a DC-centered PSD estimate, w spans the interval (–π,π] for even nfft and (–π,π) for odd 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 | single

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If the `nfft` argument is variable-size at compile time, then it must not become a scalar or an empty array at runtime.

See Also

`pburg` | `pcov` | `pyulear`

Introduced before R2006a

pmtm

Multitaper power spectral density estimate

Syntax

```
pxx = pmtm(x)
pxx = pmtm(x,'Tapers',tapertype)

pxx = pmtm(x,nw)
pxx = pmtm(x,m,'Tapers','sine')

pxx = pmtm(___,nfft)

[pxx,w] = pmtm(___)
[pxx,f] = pmtm(___,fs)

[pxx,w] = pmtm(x,nw,w)
[pxx,w] = pmtm(x,m,'Tapers','sine',w)
[pxx,f] = pmtm(___,f,fs)

[___] = pmtm(___,freqrange)
[___,pxxc] = pmtm(___,'ConfidenceLevel',probability)

[___] = pmtm(___,'DropLastTaper',dropflag)
[___] = pmtm(___,method)
[___] = pmtm(x,e,v,___)
[___] = pmtm(x,dpss_params,___)

pmtm(___)
```

Description

`pxx = pmtm(x)` returns Thomson's multitaper power spectral density (PSD) estimate, `pxx`, of the input signal `x` using "Discrete Prolate Spheroidal (Slepian) Sequences" on page 1-1677 as tapers.

`pxx = pmtm(x,'Tapers',tapertype)` specifies the type of tapers to use when computing the multitaper PSD estimate. You can specify the `'Tapers'`,`tapertype` name-value pair anywhere after `x` in the function call.

`pxx = pmtm(x,nw)` uses the time-halfbandwidth product `nw` to control the frequency resolution when computing a PSD estimate using Slepian tapers.

`pxx = pmtm(x,m,'Tapers','sine')` specifies the number of tapers or the averaging weights to apply when computing a PSD estimate using "Sine Tapers" on page 1-1677.

`pxx = pmtm(___,nfft)` uses `nfft` discrete Fourier transform (DFT) points in combination with any of the previous syntaxes. 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 a vector with the normalized frequencies at which `pxx` is computed.

`[pxx,f] = pmtm( ___ ,fs)` returns a frequency vector, `f`, in cycles per unit time. `fs` must follow `x`, `nw` (or `m` for sine tapers), and `nfft` in the function call. To input a sample rate and still use the default values of the preceding arguments, specify these arguments as empty, `[]`.

`[pxx,w] = pmtm(x,nw,w)` returns the multitaper PSD estimate computed using Slepian sequences at the normalized frequencies specified in `w`. The vector `w` must contain at least two elements.

`[pxx,w] = pmtm(x,m,'Tapers','sine',w)` returns the multitaper PSD estimate computed using sine tapers at the normalized frequencies specified in `w`. The vector `w` must contain at least two elements.

`[pxx,f] = pmtm( ___ ,f,fs)` computes the multitaper PSD estimate at the frequencies specified in `f`. The vector `f` must contain at least two elements in the same units as the sample rate `fs`.

`[ ___ ] = pmtm( ___ ,freqrange)` returns the multitaper PSD estimate over the frequency range specified by `freqrange`.

`[ ___ ,pxxc] = pmtm( ___ ,'ConfidenceLevel',probability)` returns the `probability` × 100% confidence intervals for the PSD estimate in `pxxc`.

`[ ___ ] = pmtm( ___ ,'DropLastTaper',dropflag)` specifies whether `pmtm` drops the last Slepian taper when computing the multitaper PSD estimate.

`[ ___ ] = pmtm( ___ ,method)` combines the individual tapered PSD estimates using the method specified in `method`. This syntax applies only to Slepian tapers.

`[ ___ ] = pmtm(x,e,v, ___ )` uses the Slepian tapers in `e` and the eigenvalues in `v` to compute the PSD. Use `dpss` to obtain `e` and `v`.

`[ ___ ] = pmtm(x,dpss_params, ___ )` uses the cell array `dpss_params` to pass input arguments to `dpss`. This syntax applies only to Slepian tapers.

`pmtm( ___ )` with no output arguments plots the multitaper PSD estimate in the current figure window.

Examples

Multitaper Estimate Using Default Inputs

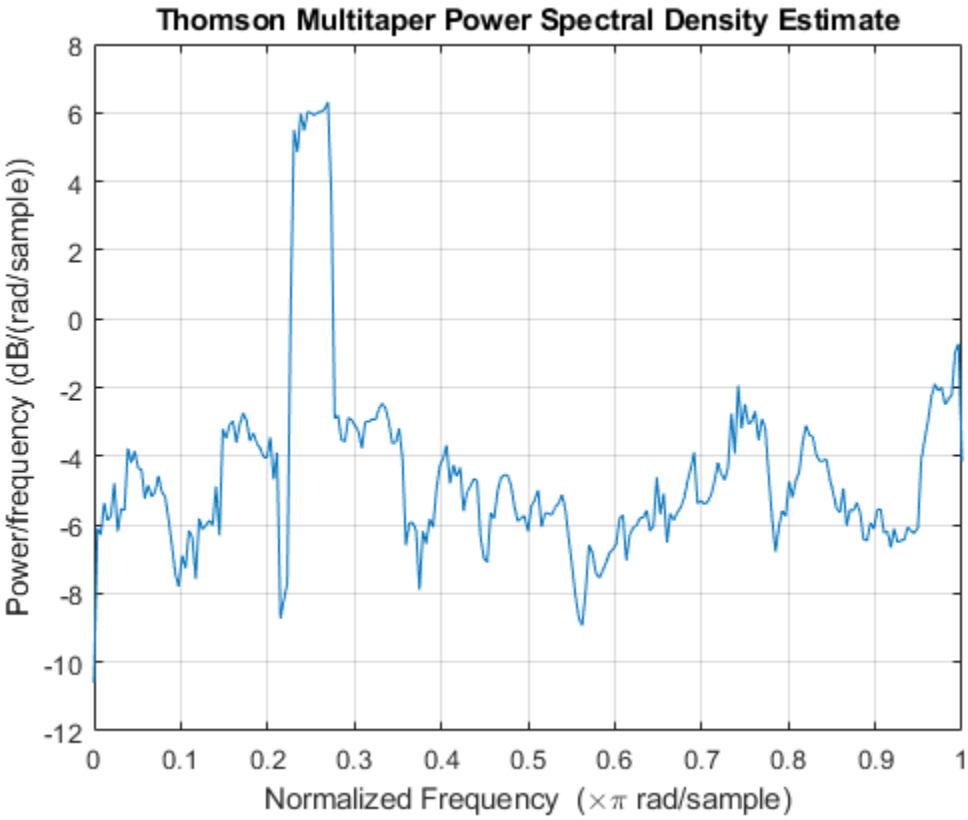
Obtain the multitaper PSD estimate of an input signal consisting of a discrete-time sinusoid with an angular frequency of $\pi/4$ rad/sample with additive $N(0,1)$ white noise.

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)
```



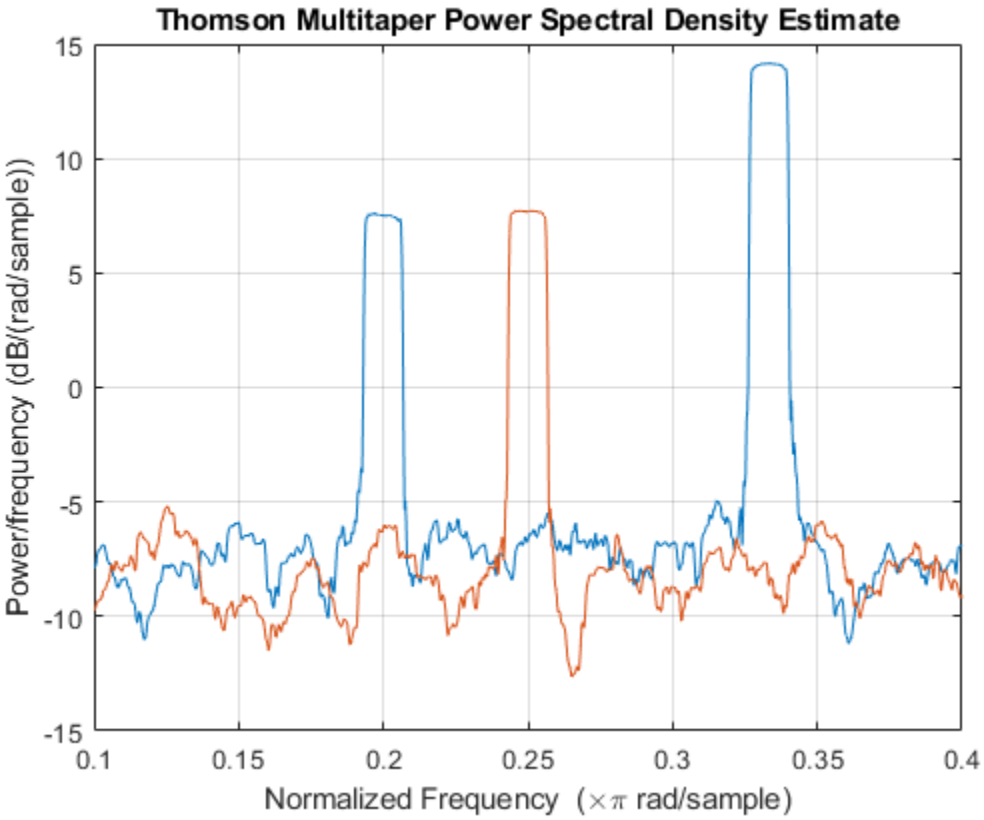
Sine and Slepian Tapers

Generate 2048 samples of a two-channel signal embedded in additive $N(0,1)$ white Gaussian noise.

- The first channel consists of two sinusoids with normalized frequencies of $\pi/3$ and $\pi/5$ rad/sample. The first sinusoid has twice the amplitude of the second.
- The second channel has a normalized frequency of $\pi/4$ rad/sample.

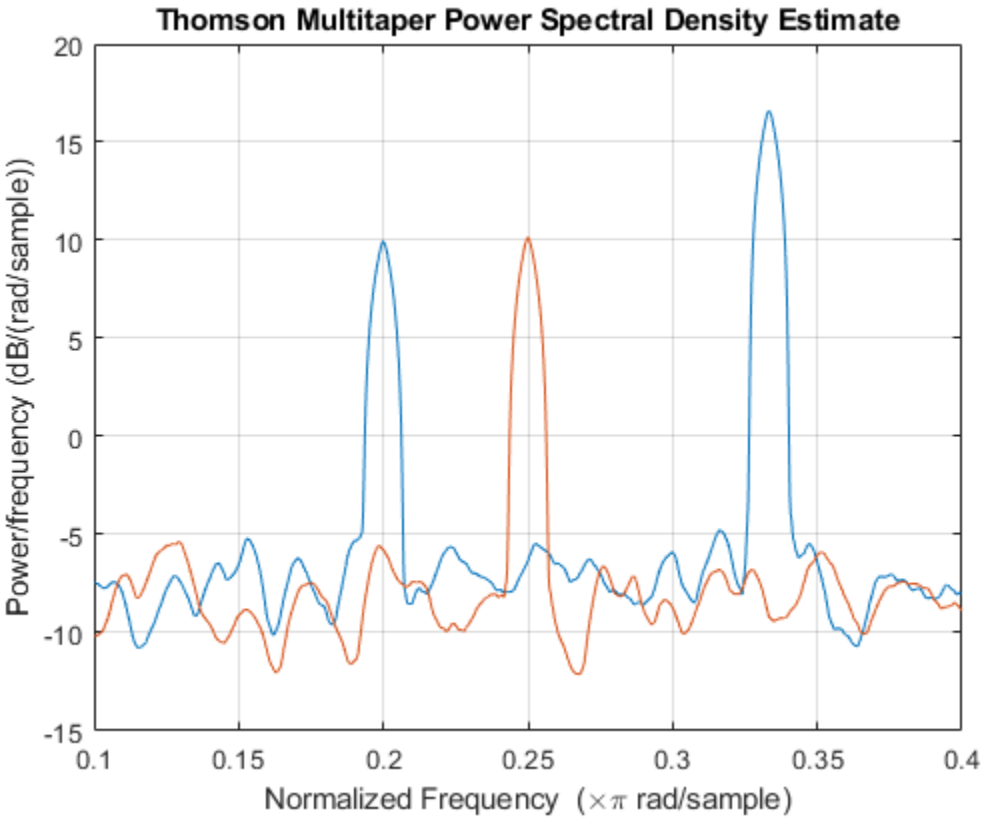
Use the multitaper method to estimate the PSD of the signal over a 1024-sample interval from 0.1π rad/sample to 0.4π rad/sample. Use 13 sine tapers weighted equally.

```
n = (0:2047)';

x = [sin(pi./[3 5].*n)*[2 1]' sin(pi/4*n)] + randn(length(n),2);

w = linspace(0.1,0.4,1024);

ntp = 13;
pmtm(x,ntp,'Tapers','sine',w*pi)
```



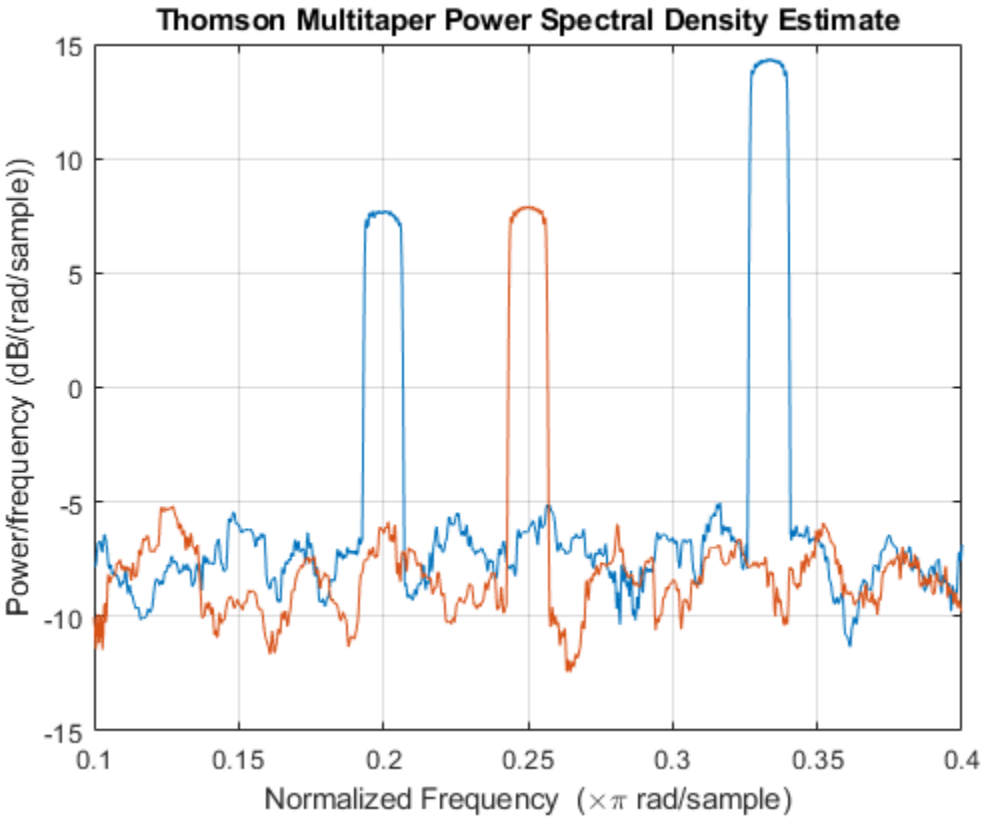
Repeat the computation, but now weight the 13 tapers in linear descending order. You can place the `'Tapers','sine'` name-value pair anywhere after `x` in the function call.

```
pmtm(x,(ntp:-1:1)/sum(1:ntp),w*pi,'Tapers','sine')
```



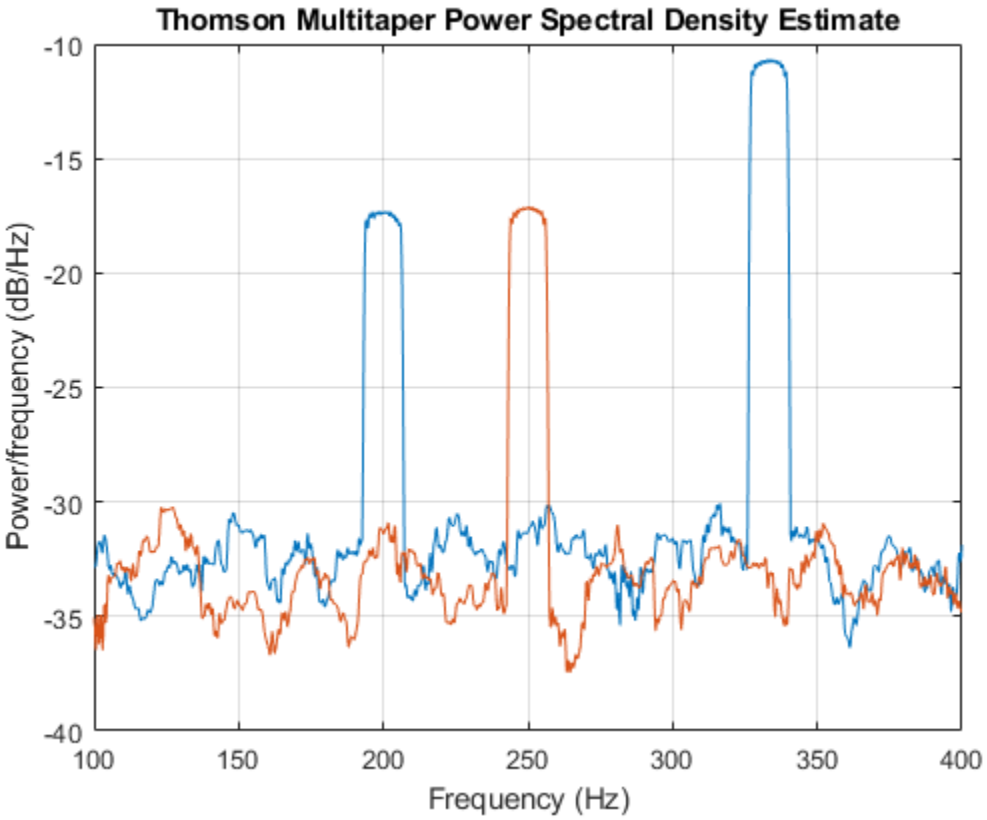
Repeat the computation, but now use 13 Slepian tapers and specify a time-halfbandwidth product of 7.5.

```
nw = 7.5;

pmtm(x,{nw,ntp},w*pi)
```



Repeat the computation, but now specify a sample rate of 2 kHz.

```
fs = 2e3;

pmtm(x,{nw,ntp},w*(fs/2),fs)
```

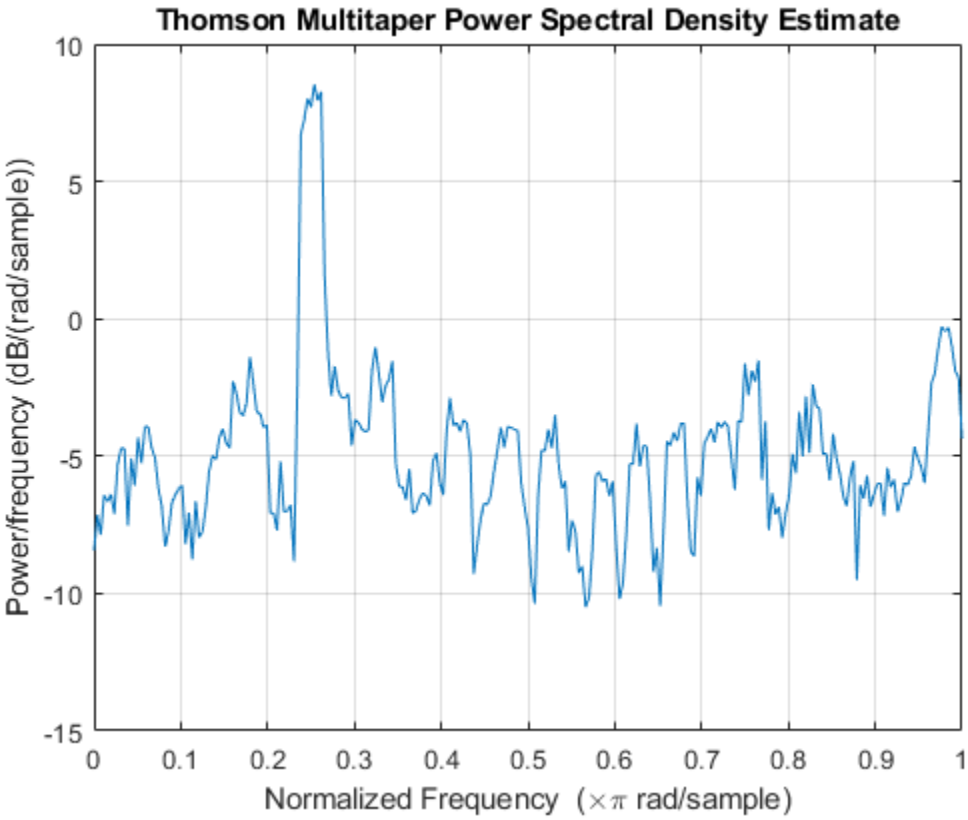


Specify Time-Halfbandwidth Product

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)
```

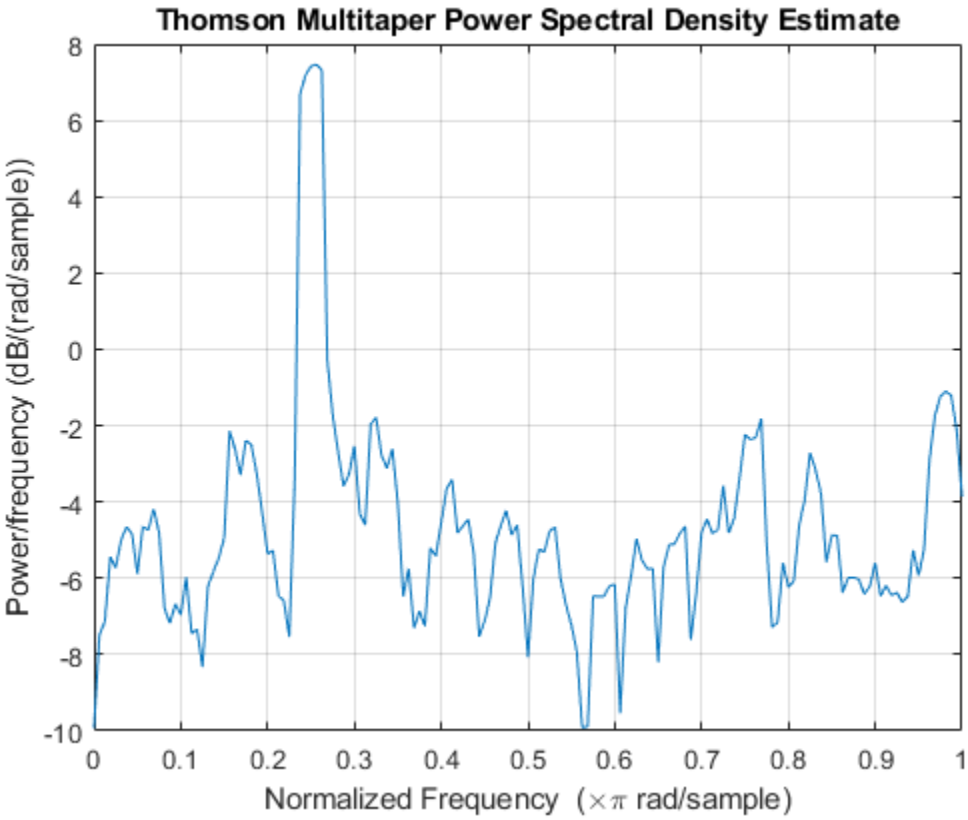


DFT Length Equal to Signal Length

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))
```



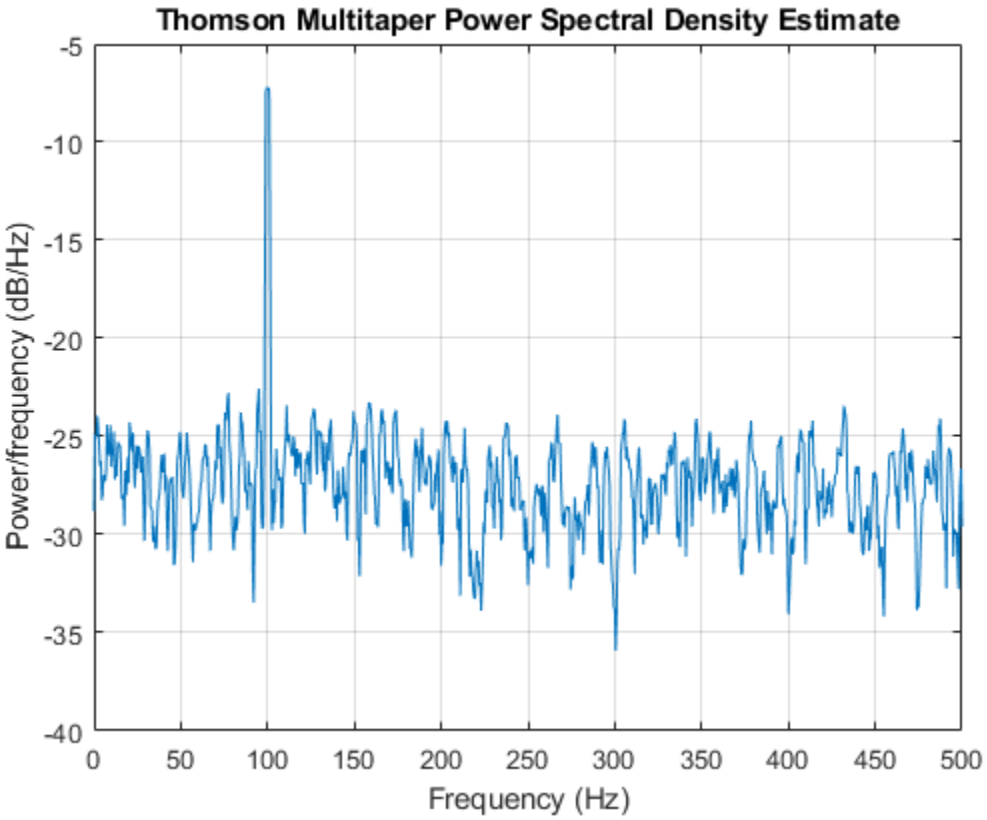
Multitaper Estimate with Sample Rate

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

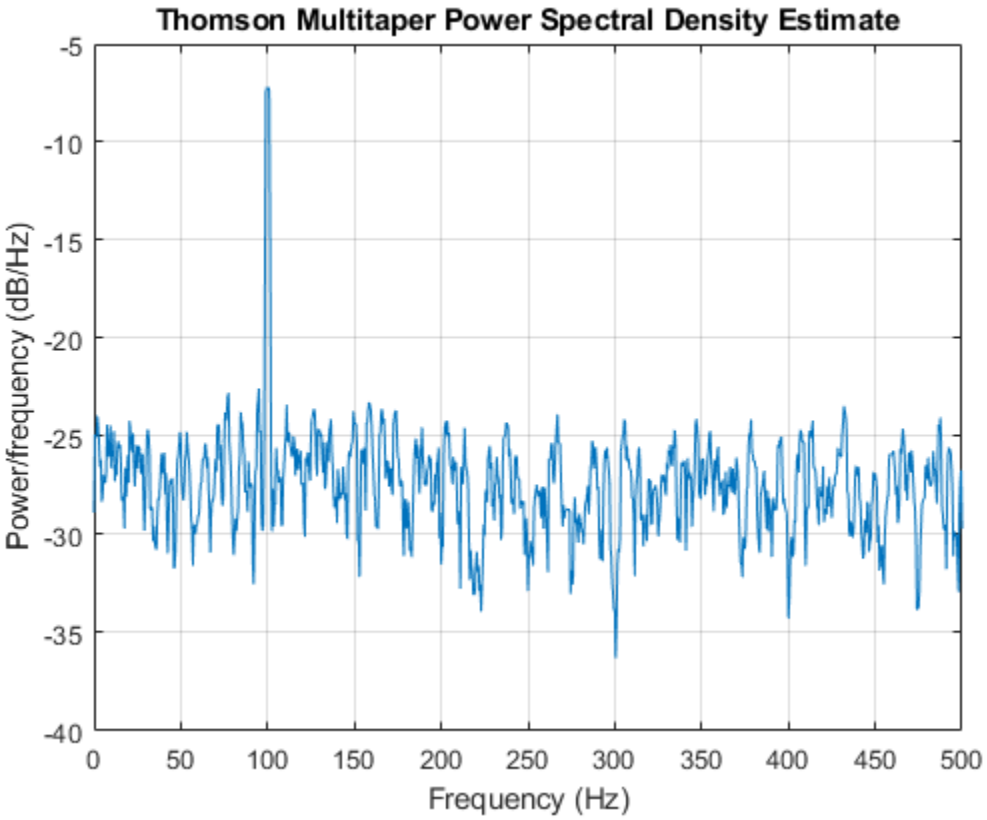
Obtain a multitaper PSD estimate where the individual tapered direct spectral estimates are given equal weight in the average.

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

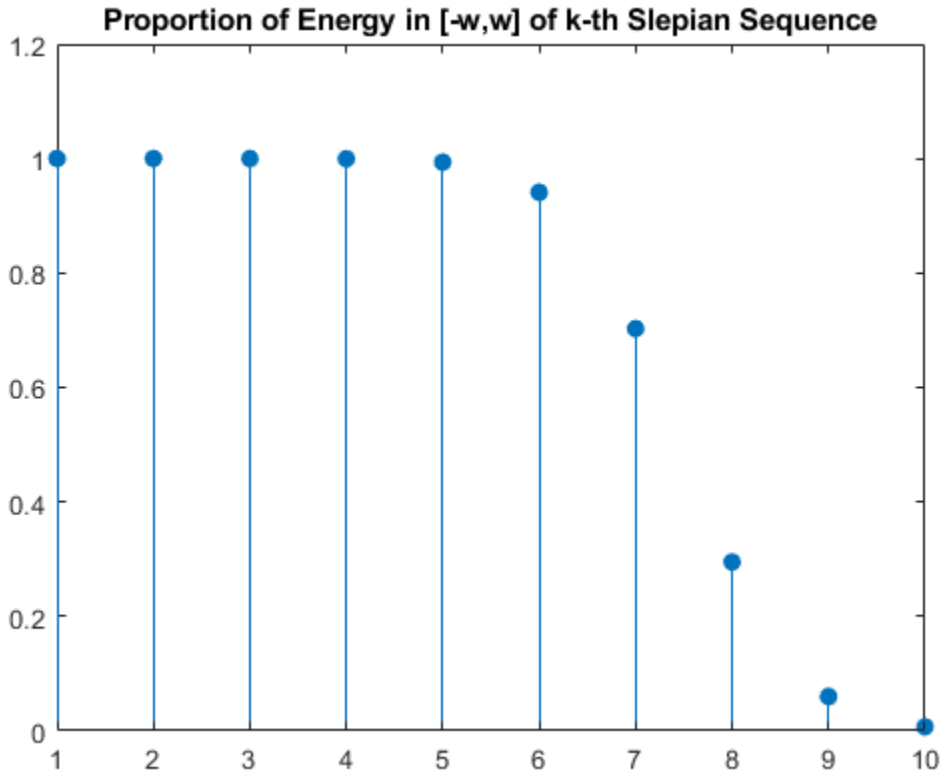
This example examines the frequency-domain concentrations of the DPSS sequences. The example produces a multitaper PSD estimate of an input signal by precomputing the Slepian sequences and selecting only those with more than 99% of their energy concentrated in the resolution bandwidth.

The signal is a 100 Hz sine wave in additive $N(0,1)$ white noise. The signal duration is 2 seconds.

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

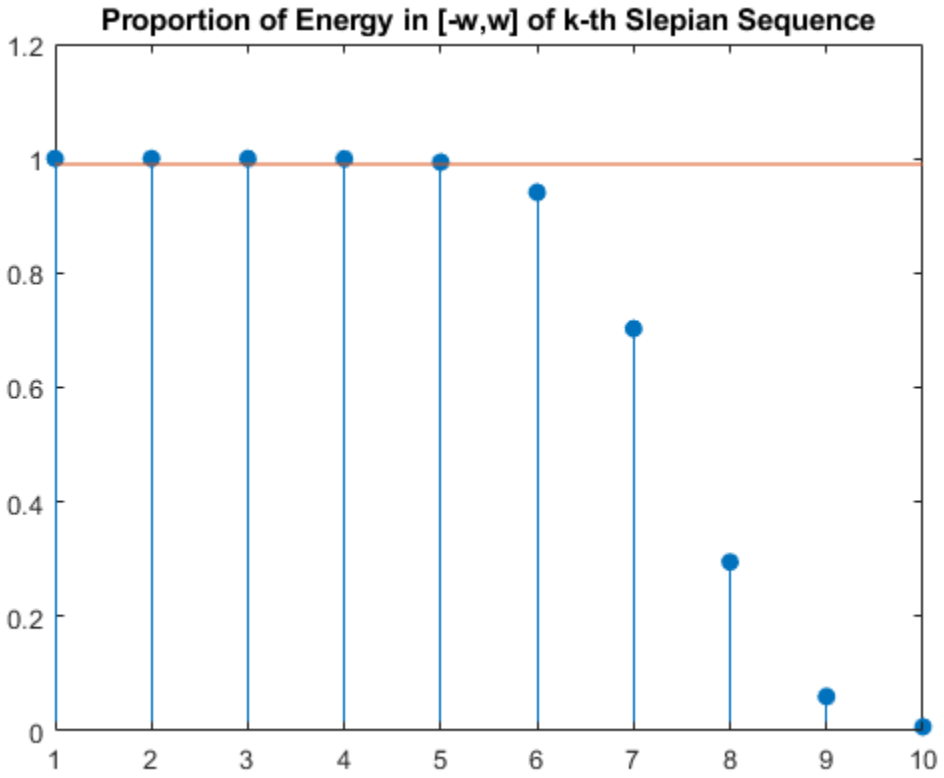
Set the time-halfbandwidth product to 3.5. For the signal length of 2000 samples and a sampling interval of 0.001 seconds, this results in a resolution bandwidth of [–1.75,1.75] Hz. Calculate the first 10 Slepian sequences and examine their frequency concentrations in the specified resolution bandwidth.

```
[e,v] = dpss(length(x),3.5,10);
lv = length(v);

stem(1:lv,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 lv],0.99*[1 1])
hold off
```



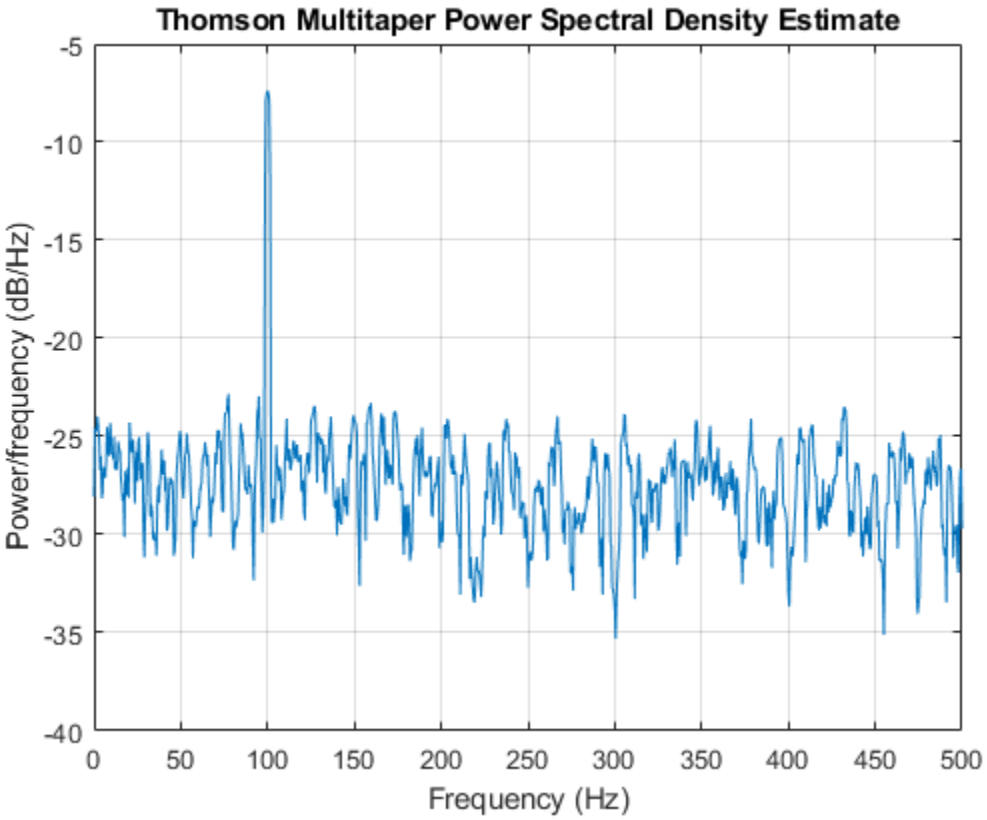

```
idx = find(v>0.99,1,'last')
```

```
idx = 5
```

```
[pxx,f] = pmtm(x,e(:,1:idx),v(1:idx),length(x),fs,'DropLastTaper',false);
```

Plot the multitaper PSD estimate.

```
pmtm(x,e(:,1:idx),v(1:idx),length(x),fs,'DropLastTaper',false)
```



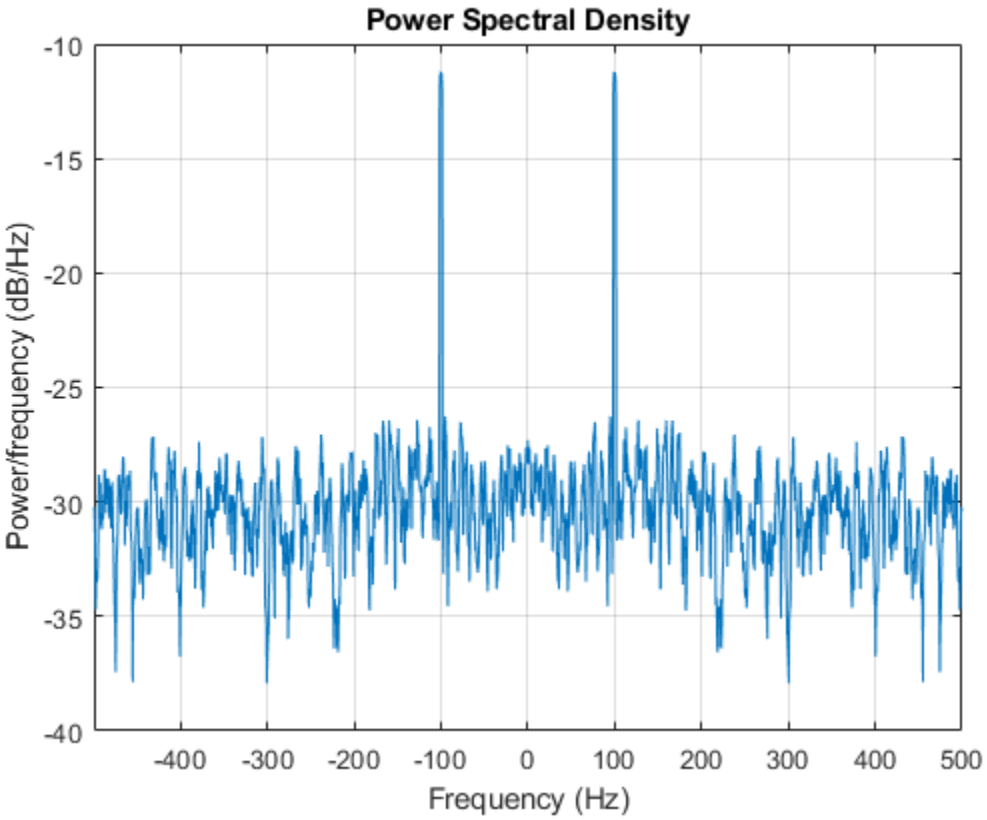
DC-Centered Multitaper PSD Estimate

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.

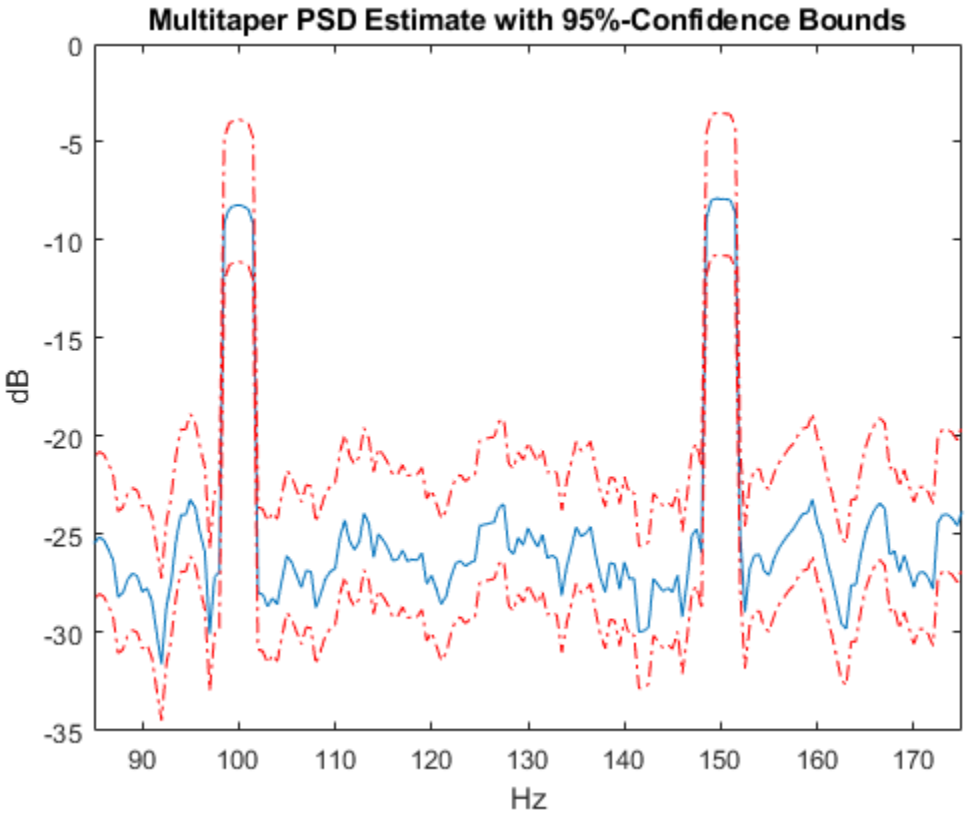
Create a signal consisting of the superposition of 100-Hz and 150-Hz sine waves in additive white $N(0,1)$ noise. The amplitude of the two sine waves is 1. The sampling frequency is 1 kHz. The signal is 2 s in duration.

```
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')
```

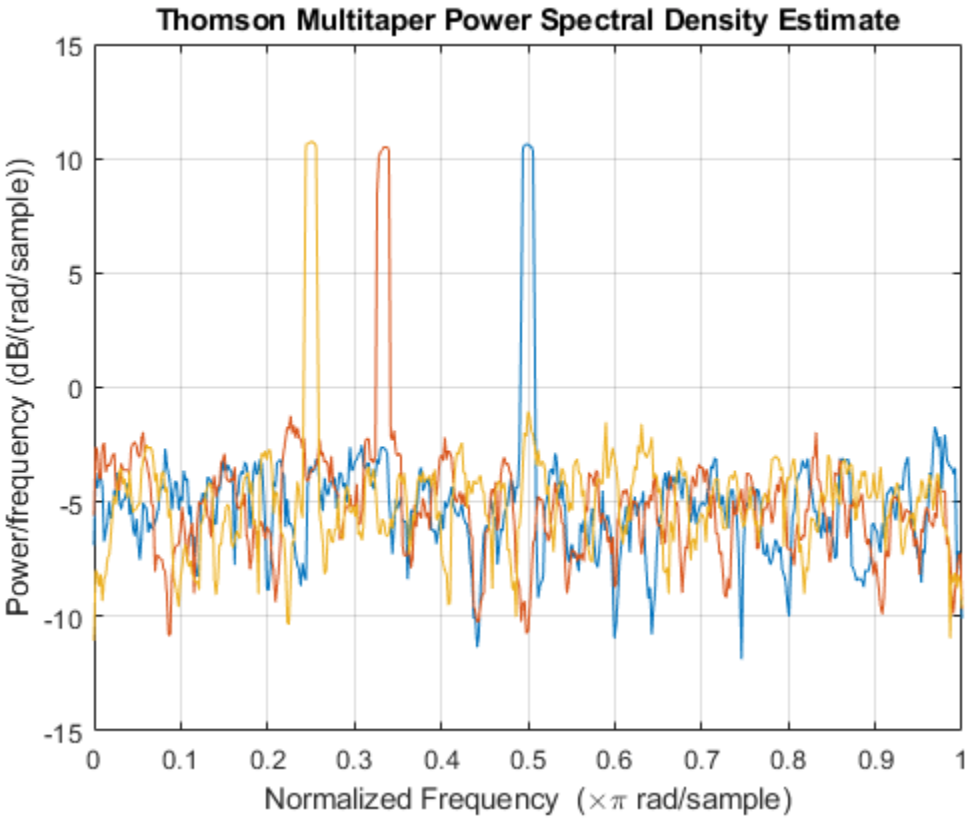


The lower confidence bound in the immediate vicinity of 100 and 150 Hz is significantly above the upper confidence bound outside the vicinity of 100 and 150 Hz.

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

tapertype — Taper type

`'slepian'` (default) | `'sine'`

Taper type, specified as `'slepian'` or `'sine'`.

- `'slepian'` — Use “Discrete Prolate Spheroidal (Slepian) Sequences” on page 1-1677 as tapers.
- `'sine'` — Use “Sine Tapers” on page 1-1677.

You can specify the `'Tapers',tapertype` name-value pair anywhere after `x` in the function call.

Data Types: `char` | `string`

`nw` — Time-halfbandwidth product

4 (default) | positive scalar

Time-halfbandwidth product, specified as a positive scalar. `pmtm` uses 2 × `nw` – 1 Slepian tapers in the PSD estimate. Typical choices for `nw` are `2`, `5/2`, `3`, `7/2`, or `4`.

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 $2NW - 1$.

`m` — Sine taper number or averaging weights

7 (default) | integer scalar | vector

Sine taper number or averaging weights, specified as an integer scalar or a vector.

- If `m` is a scalar, it denotes the number of sine tapers used as data windows when computing the PSD estimate. The sine tapers are weighted uniformly.
- If `m` is a vector, it denotes the weights used to average the sine tapers when computing the PSD estimate. The length of `m` indicates the number of tapers to use. The elements of `m` must add to 1.

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` — 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 has units of Hz.

`w` — Normalized frequencies

vector

Normalized frequencies, specified as a row or column vector with at least two elements. Normalized frequencies are in rad/sample.

Example: `w = [pi/4 pi/2]`

Data Types: `double`

`f` — Frequencies

vector

Frequencies, specified as a row or column vector with at least two elements. The frequencies are in cycles per unit time. The unit time is specified by the sample rate, `fs`. If `fs` has units of samples/second, then `f` has units of Hz.

Example: `fs = 1000; f = [100 200]`

Data Types: `double`

`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 2*NW* – 1 DPSS sequences have eigenvalues close to unity. If you use less than 2*NW* – 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.

`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 [2]. 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 matrix. If `x` has length *N*, then `e` has *N* rows. The matrix `e` is an 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 2*NW* – 1 eigenvalues are close to 1 and then decrease toward 0. The vector `v` is an output of `dpss`.

`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` because it is obtained from the length of `x`.

Example: `pmtm(randn(1000,1),{2.5,3})` computes the PSD of a random sequence using the first 3 Slepian sequences with time-halfbandwidth product 2.5.

`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,π] rad/sample. If `nfft` is odd, the length of `pxx` is (`nfft` + 1)/2 and the interval is [0,π) 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.

The function multiplies the power by 2 at all frequencies except 0 and the Nyquist frequency to conserve the total power.

- `'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.

`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` × 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 Ω and specify the sample rate 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 $(-\pi,\pi]$ for even `nfft` and $(-\pi,\pi)$ for odd `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` | `single`

`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

Thomson's Multitaper Spectral Estimation

The periodogram is not a consistent estimator of the true power spectral density (PSD) of a wide-sense stationary process. To reduce the variability in the periodogram — and thus produce a consistent estimate of the PSD — the multitaper method averages modified periodograms obtained using a family of mutually orthogonal windows or tapers. In addition to mutual orthogonality, the tapers also have optimal time-frequency concentration properties. Both the orthogonality and time-frequency concentration of the tapers are critical to the success of the multitaper technique. See “Discrete Prolate Spheroidal (Slepian) Sequences” on page 1-1677 for a brief description of the Slepian sequences used in Thomson’s multitaper method.

The multitaper method uses K modified periodograms, each one obtained using a different Slepian sequence as the window. Let

$$S_k(f) = \Delta t \left| \sum_{n=0}^{N-1} g_k(n)\, x(n)\, e^{-j2\pi f n \Delta t} \right|^2$$

denote the modified periodogram obtained with the kth Slepian sequence, $g_k(n)$. In its simplest form, the multitaper method simply averages the K modified periodograms to produce the multitaper PSD estimate:

$$S^{(\mathrm{MT})}(f) = \frac{1}{K} \sum_{k=0}^{K-1} S_k(f)\,.$$

Thomson's multitaper approach, introduced in [4], resembles Welch’s overlapped segment averaging method, in that both average 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 approach uses segments of the signal in each modified periodogram, and the segmenting decorrelates the different modified periodograms.

The equation for $S^{(\mathrm{MT})}(f)$ corresponds to the `'unity'` option in `pmtm`. However, as explained in “Discrete Prolate Spheroidal (Slepian) Sequences” on page 1-1677, 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 a 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.

Discrete Prolate Spheroidal (Slepian) Sequences

The derivation of the Slepian sequences proceeds from the discrete-time/continuous-frequency concentration problem. For all ℓ^2 sequences index-limited to 0, 1, ..., N – 1, the problem seeks the sequence having the maximal concentration of its energy in a frequency band [–W, W] with |W| < 1/2Δt.

This amounts to finding the eigenvalues and corresponding eigenvectors of an N-by-N self-adjoint positive semidefinite 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

$$\sum_{m=0}^{N-1} \frac{\sin(2\pi W(n-m))}{\pi(n-m)}\, g_k(m) = \lambda_k(N,W)\, g_k(n), \quad n,k = 0,\ 1,\ 2,\ \ldots,\ N-1.$$

The zeroth-order DPSS sequence, g_0, is the eigenvector corresponding to the largest eigenvalue. The first-order DPSS sequence, g_1, is the eigenvector corresponding to the next largest eigenvalue and is orthogonal to the zeroth-order sequence. The second-order DPSS sequence, g_2, is the eigenvector corresponding to the third-largest eigenvalue and is orthogonal to the two lower-order DPSS sequences. Because the operator is N-by-N, there are N eigenvectors. However, 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. Use `nw` to specify NW.

Sine Tapers

Sine tapers, an alternative to Slepian sequences proposed in [3], are defined by

$$g_k(n) = \sqrt{\frac{2}{N+1}} \sin\frac{\pi k n}{N+1}, \quad n,k = 1,2,\ldots,N\,.$$

Unlike Slepian sequences, sine tapers can be computed directly, with no need to set up and solve an eigenvalue equation. This makes sine tapers much faster to compute. Sine tapers have a spectral concentration close to that of Slepian sequences but do not need additional parameters to specify the spectral bandwidth. The bandwidth of the PSD estimate computed using sine tapers can be adjusted locally by changing the number of tapers using `m`.

Compare Slepian and Sine Tapers

Generate the first five Slepian tapers corresponding to a time-halfbandwidth product of 3. Specify a taper length of 1000.

```
N = 1000;
nw = 3;
ns = 2*(nw)-1;
```

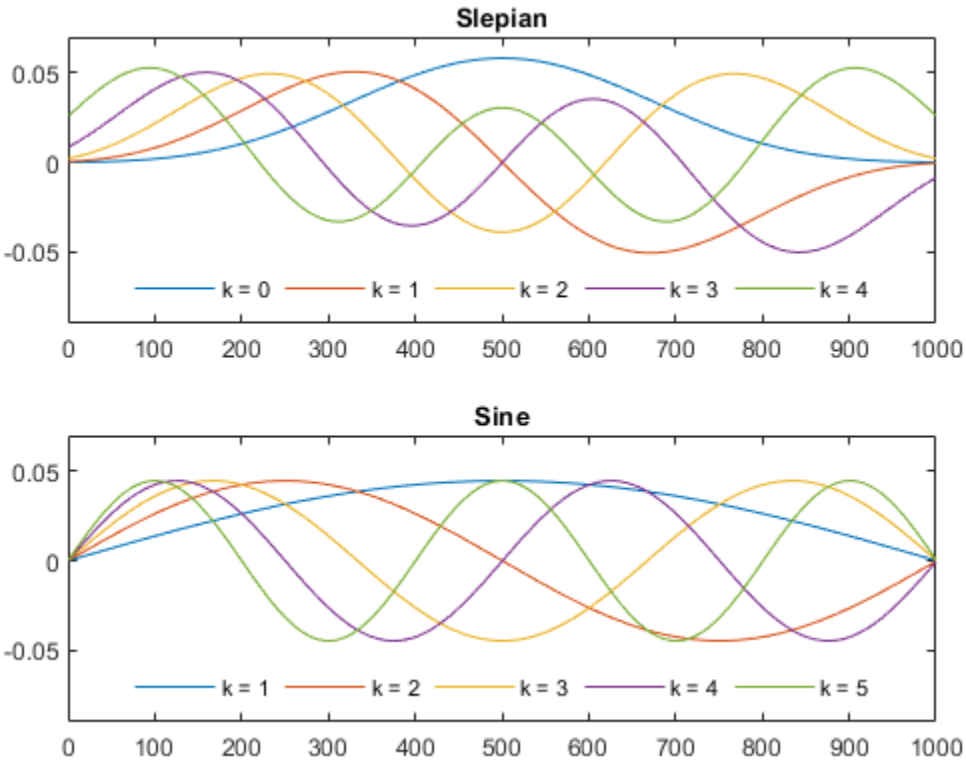
```
tprs = dpss(N,nw,ns);
lbs = "Slepian";
```

Generate the first five sine tapers.

```
n = 1:N;
k = 1:ns;

tprs(:,:,2) = sqrt(2/(N+1))*sin(pi*n'*k/(N+1));
lbs(2) = "Sine";
```

Plot the two sets of tapers.

```
for kj = 1:2
    subplot(2,1,kj)
    plot(tprs(:,:,kj))
    title(lbs(kj))
    legend(append('k = ',string(k+kj-2)), ...
        'Orientation','horizontal','Location','south')
    legend('boxoff')
    ylim([-0.09 0.07])
end
```



References

[1] McCoy, Emma J., Andrew T. Walden, and Donald B. Percival. "Multitaper Spectral Estimation of Power Law Processes." *IEEE Transactions on Signal Processing* 46, no. 3 (March 1998): 655–68. https://doi.org/10.1109/78.661333.

[2] Percival, Donald B., and Andrew T. Walden. *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques*. Cambridge; New York, NY, USA: Cambridge University Press, 1993.

[3] Riedel, Kurt S., and Alexander Sidorenko. “Minimum Bias Multiple Taper Spectral Estimation.” *IEEE Transactions on Signal Processing* 43, no. 1 (January 1995): 188–95. https://doi.org/10.1109/78.365298.

[4] Thomson, David J. "Spectrum estimation and harmonic analysis." *Proceedings of the IEEE* 70, no. 9 (1982): 1055–96. https://doi.org/10.1109/PROC.1982.12433.

See Also

`dpss` | `periodogram` | `pwelch`

Topics

“Bias and Variability in the Periodogram”

Introduced before R2006a

pmusic

Pseudospectrum using MUSIC algorithm

Syntax

```
[S,wo] = pmusic(x,p)
[S,wo] = pmusic(x,p,wi)
[S,wo] = pmusic(___,nfft)
[S,wo] = pmusic(___,'corr')

[S,fo] = pmusic(x,p,nfft,fs)
[S,fo] = pmusic(x,p,fi,fs)
[S,fo] = pmusic(x,p,nfft,fs,nwin,noverlap)

[___] = pmusic(___,freqrange)
[___,v,e] = pmusic(___)
pmusic(___)
```

Description

`[S,wo] = pmusic(x,p)` implements the multiple signal classification (MUSIC) algorithm and returns `S`, the pseudospectrum estimate of the input signal `x`, and a vector `wo` of normalized frequencies (in rad/sample) at which the pseudospectrum is evaluated. You can specify the signal subspace dimension using the input argument `p`.

`[S,wo] = pmusic(x,p,wi)` returns the pseudospectrum computed at the normalized frequencies specified in vector `wi`. The vector `wi` must have two or more elements, because otherwise the function interprets it as `nfft`.

`[S,wo] = pmusic(___,nfft)` specifies the integer length of the FFT, `nfft`, used to estimate the pseudospectrum. This syntax can include any combination of input arguments from previous syntaxes.

`[S,wo] = pmusic(___,'corr')` forces the input argument `x` to be interpreted as a correlation matrix rather than matrix of signal data. For this syntax, `x` must be a square matrix, and all of its eigenvalues must be nonnegative.

`[S,fo] = pmusic(x,p,nfft,fs)` returns the pseudospectrum computed at the frequencies specified in vector `fo` (in Hz). Supply the sample rate `fs` in Hz.

`[S,fo] = pmusic(x,p,fi,fs)` returns the pseudospectrum computed at the frequencies specified in the vector `fi`. The vector `fi` must have two or more elements, because otherwise the function interprets it as `nfft`.

`[S,fo] = pmusic(x,p,nfft,fs,nwin,noverlap)` returns the pseudospectrum `S` by segmenting the input data `x` using the window `nwin` and overlap length `noverlap`.

`[___] = pmusic(___,freqrange)` specifies the range of frequency values to include in `fo` or `wo`.

`[___,v,e] = pmusic(___)` returns the matrix `v` of noise eigenvectors along with the associated eigenvalues in the vector `e`.

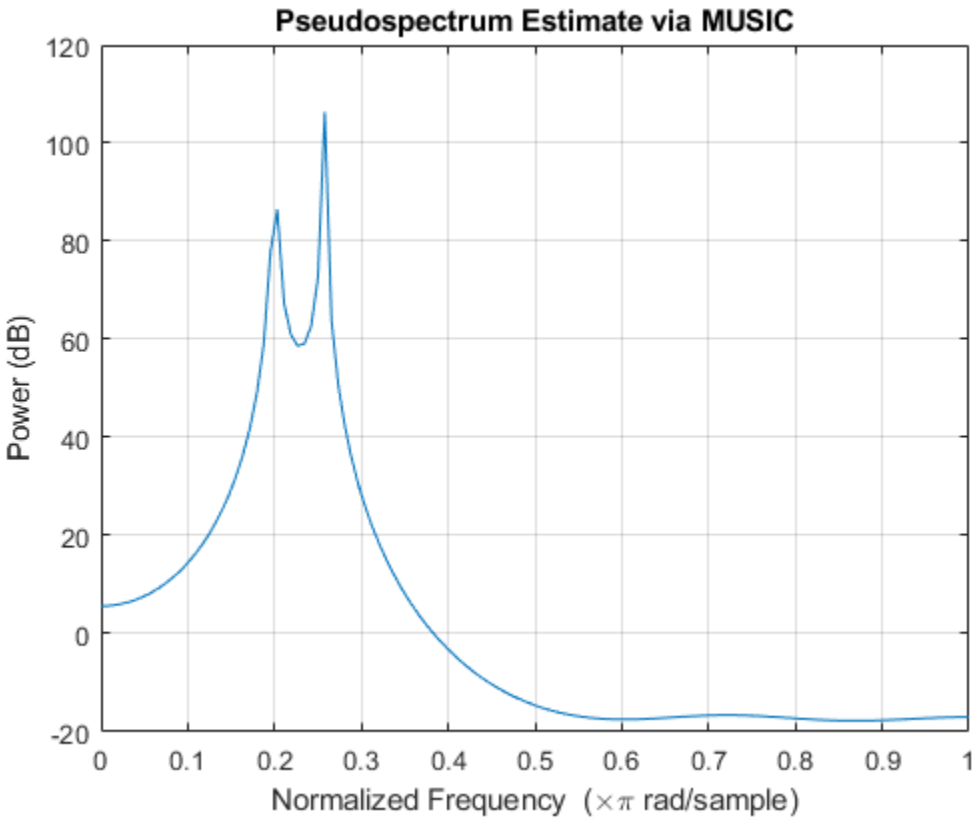
`pmusic(___)` with no output arguments plots the pseudospectrum in the current figure window.

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.

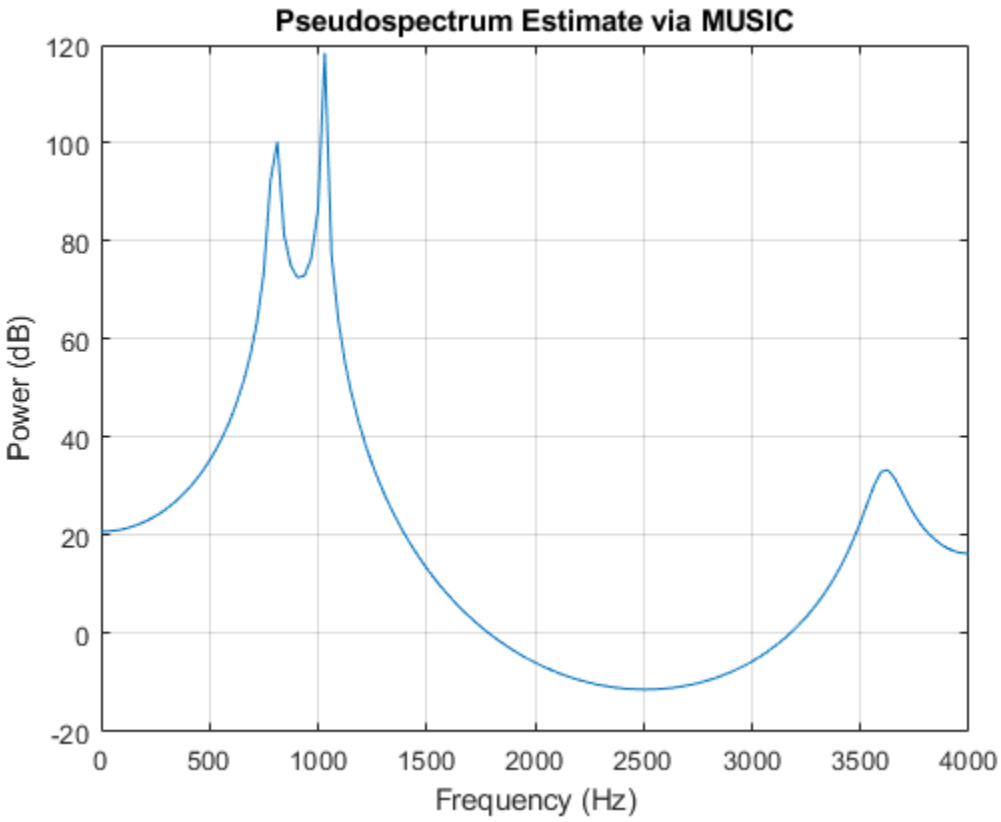
```
n = 0:199;
x = cos(0.257*pi*n) + sin(0.2*pi*n) + 0.01*randn(size(n));
pmusic(x,4)      % Set p to 4 because there are two real inputs
```



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) = 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:

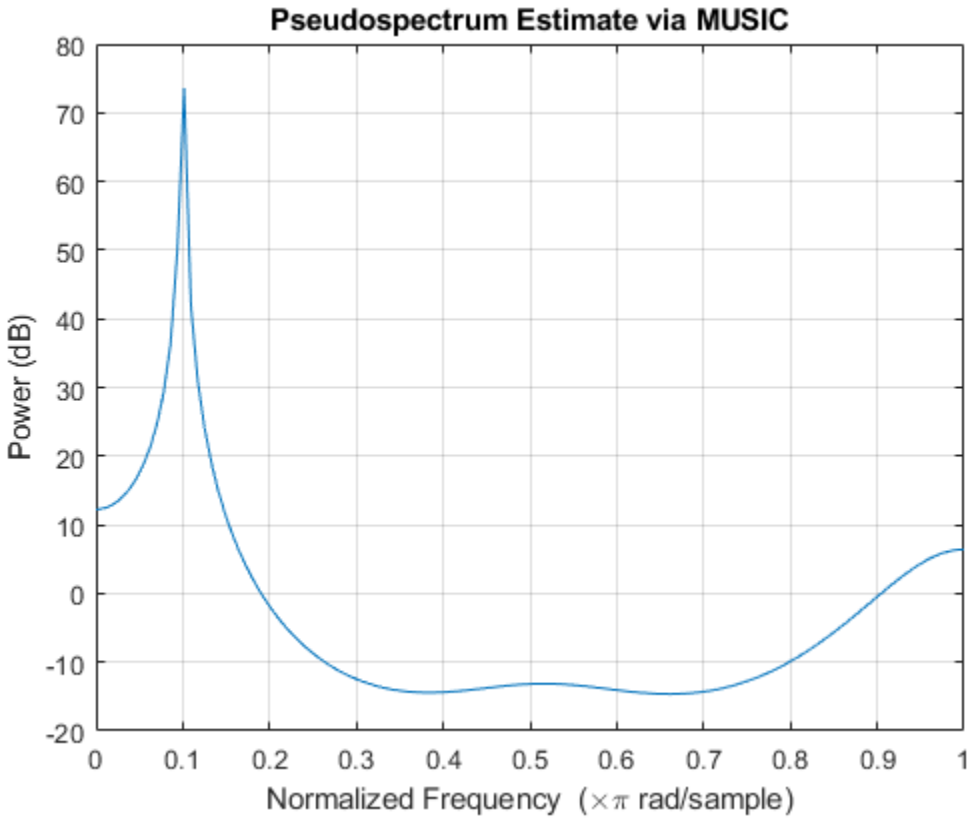
```
rng default
n = 0:199;
x = cos(0.257*pi*n) + sin(0.2*pi*n) + 0.01*randn(size(n));
[P,f] = pmusic(x,[Inf,1.1],[],8000,7); % Window length = 7
plot(f,20*log10(abs(P)))
xlabel 'Frequency (Hz)', ylabel 'Power (dB)'
title 'Pseudospectrum Estimate via MUSIC', grid on
```



Entering a Correlation Matrix

Supply a positive definite correlation matrix, R, for estimating the spectral density. Use the default 256 samples.

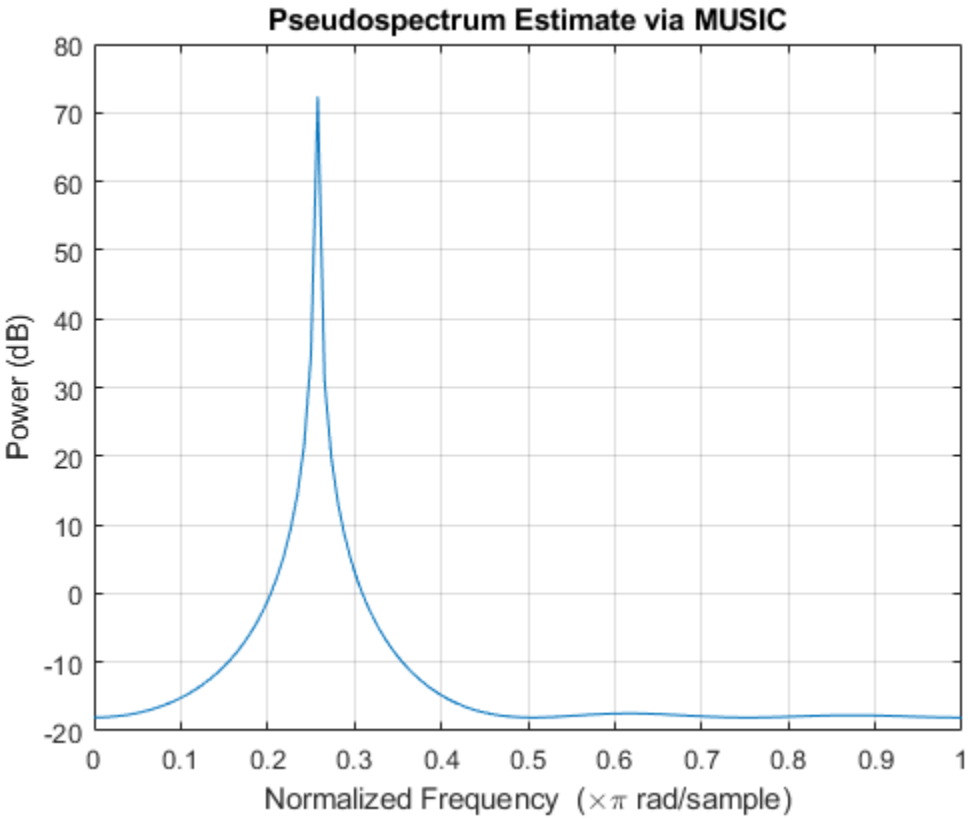
```
R = toeplitz(cos(0.1*pi*(0:6))) + 0.1*eye(7);
pmusic(R,4,'corr')
```



Enter Signal Data Matrix Generated by `corrmtx`

Enter a signal data matrix, `Xm`, generated from data using `corrmtx`.

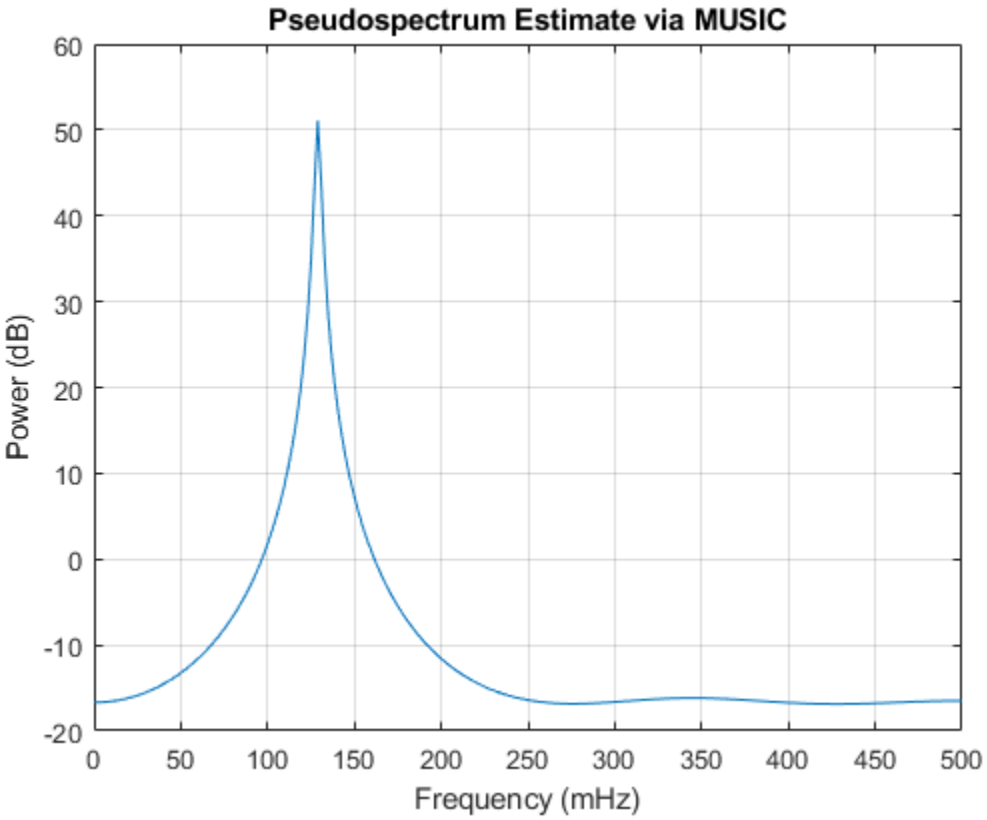
```
n = 0:699;
x = cos(0.257*pi*(n)) + 0.1*randn(size(n));
Xm = corrmtx(x,7,'modified');
pmusic(Xm,2)
```



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)
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If `x` is a vector, then it is treated as one observation of the signal. If `x` is a matrix, 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 `x`.

p — Subspace dimension

real positive integer | two-element vector

Subspace dimension, specified as a real positive integer or a two-element vector. If `p` is a real positive integer, then it is treated as the subspace dimension. If `p` is a two-element vector, the second element of `p` represents a threshold that is multiplied by λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold λ_{min}`*p(2)` are assigned to the noise subspace. In this case, `p(1)` specifies the maximum dimension of the signal subspace. The extra threshold parameter in the second entry in `p` provides you more flexibility and control in assigning the noise and signal subspaces.

Note If the inputs to `pmusic` are real sinusoids, set the value of `p` to two times the number of sinusoids. If the inputs are complex sinusoids, set `p` equal to the number of sinusoids.

`wi` — Input normalized frequencies

vector

Input normalized frequencies, specified as a vector.

Data Types: `double`

`nfft` — Number of DFT points

`256` (default) | integer | []

Number of DFT points, specified as a positive integer. If `nfft` is specified as empty, the default `nfft` is used.

`fs` — Sample rate

`1` (default) | positive scalar | []

Sample rate, specified as a positive scalar in Hz. in Hz. If you specify `fs` with the empty vector [], the sample rate defaults to `1` Hz.

`fi` — Input frequency

vector

Input frequencies, specified as a vector. The pseudospectrum is computed at the frequencies specified in the vector.

`nwin` — Length of rectangular window

`2*p(1)` (default) | nonnegative integer

Length of rectangular window, specified as a nonnegative integer.

`noverlap` — Number of overlapped samples

`nwin-1` (default) | nonnegative integer

Number of overlapped samples, specified as a nonnegative integer smaller than the length of window.

Note The arguments `nwin` and `noverlap` are ignored when you include `'corr'` in the syntax.

`freqrange` — Frequency range of pseudospectrum estimates

`'half'` | `'whole'` | `'centered'`

Frequency range of pseudospectrum estimates,specified as one of `'half'`, `whole`, or `'centered'`.

- `'half'` — Returns half the spectrum for a real input signal `x`. If `nfft` is even, then `S` has length `nfft`/2 + 1 and is computed over the interval [0, π]. If `nfft` is odd, the length of `S` is (`nfft` + 1)/2 and the frequency interval is [0,π). When your specify `fs`, the intervals are [0, `fs`/2) and [0, `fs`/2] for even and odd `nfft`, respectively.
- `'whole'` — Returns the whole spectrum for either real or complex input `x`. In this case, `S` has length `nfft` and is computed over the interval [0, 2π). When you specify `fs`, the frequency interval is [0, `fs`).

- `'centered'` — Returns the centered whole spectrum for either real or complex input `x`. In this case, `S` has length `nfft` and is computed over the interval (–π, π] for even `nfft` and (–π, π) for odd `nfft`. When you specify `fs`, the frequency intervals are (–`fs`/2, `fs`/2] and (–`fs`/2, `fs`/2) for even and odd `nfft`, respectively.

Note You can put the arguments `freqrange` or `'corr'` anywhere in the input argument list after `p`.

Output Arguments

S — Pseudospectrum estimate

vector

Pseudospectrum estimate, returned as a vector. The pseudospectrum is calculated using estimates of the eigenvectors of a correlation matrix associated with the input data `x`.

wo — Output normalized frequencies

vector

Output normalized frequencies, specified as a vector. `S` and `wo` 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 table indicates the length of `S` (and `wo`) and the range of the corresponding normalized frequencies for the first syntax.

S Characteristics for an FFT Length of 256 (Default)

Input Data Type	Length of S and w0	Range of the Corresponding Normalized Frequencies
Real	129	[0, π]
Complex	256	[0, 2π)

If `nfft` is specified, the following table indicates the length of `S` and `wo` and the frequency range for `wo`.

S and Frequency Vector Characteristics

Input Data Type	nfft Even or Odd	Length of S and w	Range of w
Real	Even	`(nfft/2 )+ 1`	[0, π]
Real	Odd	`(nfft + 1)/2`	[0, π)
Complex	Even or odd	`nfft`	[0, 2π)

fo — Output frequency

vector

Output frequency, returned as a vector. The frequency range for `fo` depends on `nfft`, `fs`, and the values of the input `x`. The length of `S` (and `fo`) is the same as in the S and Frequency Vector Characteristics above. The following table indicates the frequency range for `fo` if `nfft` and `fs` are specified.

S and Frequency Vector Characteristics with fs Specified

Input Data Type	nfft Even/Odd	Range of f
Real	Even	[0, `fs`/2]
Real	Odd	[0, `fs`/2)
Complex	Even or odd	[0, `fs`)

Additionally, if `nwin` and `noverlap` are also specified, 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

form of 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 for related information on this syntax.

v — Noise eigenvector
matrix

Noise eigenvectors, returned as a matrix. 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)`.

e — Estimated eigenvalues
vector

Estimated eigenvalues of the correlation matrix, returned as a vector.

Tips

In the process of estimating the pseudospectrum, `pmusic` computes the noise and signal subspaces from the estimated eigenvectors v_j and eigenvalues λ_j of the signal's correlation matrix. The smallest of these eigenvalues is used in conjunction with the threshold parameter `p(2)` to affect the dimension of the noise subspace in some cases.

The length n of the eigenvectors computed by `pmusic` is the sum of the dimensions of the signal and noise subspaces. This eigenvector length depends on your input (signal data or correlation matrix) and the syntax you use.

The following table summarizes the dependency of the eigenvector length on the input argument.

Eigenvector Length Depending on Input Data and Syntax

Form of Input Data x	Comments on the Syntax	Length n of Eigenvectors
Row or column vector	`nwin` is specified as a scalar integer.	`nwin`
Row or column vector	`nwin` is specified as a vector.	`length(nwin)`
Row or column vector	`nwin` is not specified.	2 × `p(1)`
l-by-*m* matrix	If `nwin` is specified as a scalar, it is not used. If `nwin` is specified as a vector, `length(nwin)` must equal *m*.	*m*
m-by-*m* nonnegative definite matrix	`'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 multiple signal classification (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 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 correlation matrix of the signal are estimated if you do not supply the correlation matrix.

The algorithm used to compute the MUSIC pseudospectrum estimate takes into account the relative sizes of the signal and noise subspaces.

- If the dimension of the signal subspace is smaller than the dimension of the noise subspace, the MUSIC pseudospectrum estimate is given by

  $$P_{\mathrm{MUSIC}}(f) = \frac{1}{\sum_{k=1}^{p}\left(1 - \left|v_k^H e(f)\right|^2\right)},$$

  where p is the dimension of the signal subspace and v_k is the kth eigenvector of the correlation matrix. The eigenvectors used in the sum correspond to the largest eigenvalues and span the signal subspace.

- If the dimension of the noise subspace is smaller than the dimension of the signal subspace, the MUSIC pseudospectrum estimate is given by

  $$P_{\mathrm{MUSIC}}(f) = \frac{1}{e^H(f)\left(\sum_{k=p+1}^{N} v_k v_k^H\right)e(f)} = \frac{1}{\sum_{k=p+1}^{N}\left|v_k^H e(f)\right|^2},$$

  where N is the dimension of the signal space plus the dimension of the noise space and v_k is the kth eigenvector of the correlation matrix. The eigenvectors used in the sum correspond to the smallest eigenvalues and span the noise subspace.

In each case, 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If `nfft` or `nwin` is variable-size during code generation, then it must not reduce to a scalar or an empty array at runtime.

See Also

`corrmtx` | `pburg` | `peig` | `periodogram` | `pmtm` | `prony` | `pwelch` | `rooteig` | `rootmusic`

Introduced before R2006a

poctave

Generate octave spectrum

Syntax

```
p = poctave(x,fs)
p = poctave(xt)

p = poctave(pxx,fs,f)

p = poctave( ___ ,type)

p = poctave( ___ ,Name,Value)

[p,cf] = poctave( ___ )

[p,cf,t] = poctave( ___ )

poctave( ___ )
```

Description

`p = poctave(x,fs)` returns the octave spectrum of a signal `x` sampled at a rate `fs`. The octave spectrum is the average power over octave bands as defined by the ANSI S1.11 standard [2]. If `x` is a matrix, then the function estimates the octave spectrum independently for each column and returns the result in the corresponding column of `p`.

`p = poctave(xt)` returns the octave spectrum of a signal stored in the MATLAB timetable `xt`.

`p = poctave(pxx,fs,f)` performs octave smoothing by converting a power spectral density, `pxx`, to a 1/*b* octave power spectrum, where *b* is the number of subbands in the octave band. The frequencies in `f` correspond to the PSD estimates in `pxx`.

`p = poctave( ___ ,type)` specifies the kind of spectral analysis performed by the function. Specify `type` as `'power'` or `'spectrogram'`.

`p = poctave( ___ ,Name,Value)` specifies additional options for any of the previous syntaxes using name-value arguments.

`[p,cf] = poctave( ___ )` also returns the center frequencies of the octave bands over which the octave spectrum is computed.

`[p,cf,t] = poctave( ___ )` additionally returns a time vector, `t`, corresponding to the center times of the segments used to compute the power spectrum estimates when `type` is `'spectrogram'`.

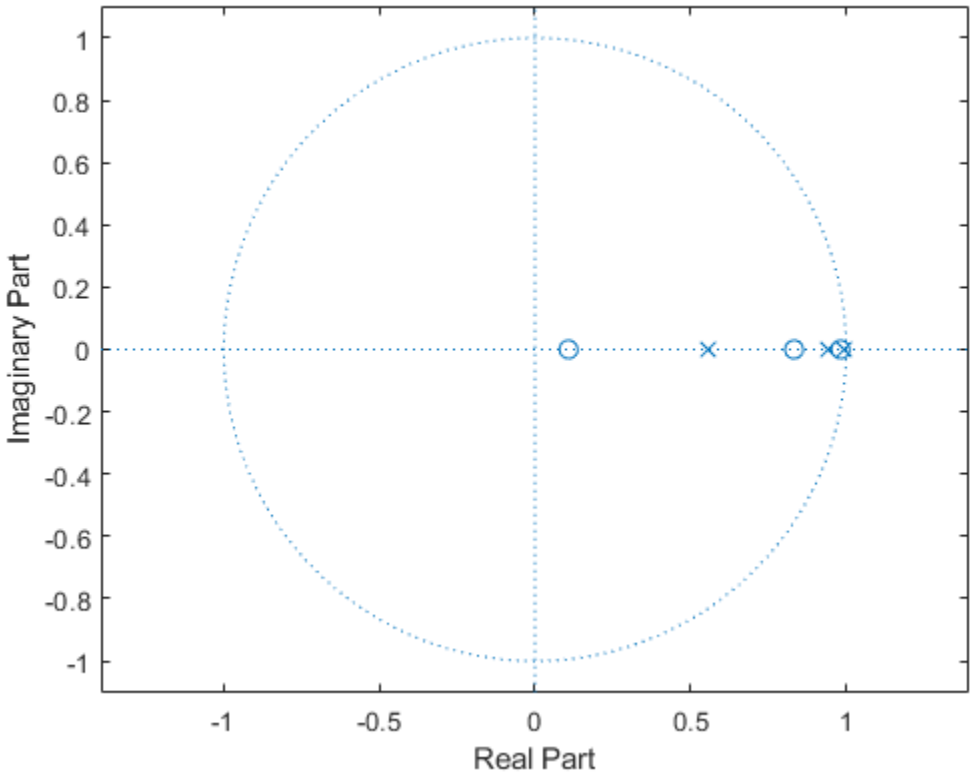
`poctave( ___ )` with no output arguments plots the octave spectrum or spectrogram in the current figure. If `type` is specified as `'spectrogram'`, then this function is supported only for single-channel input.

Examples

Octave Spectra of White and Pink Noise

Generate 10^5 samples of white Gaussian noise. Create a signal of pseudopink noise by filtering the white noise with a filter whose zeros and poles are all on the positive *x*-axis. Visualize the zeros and poles.

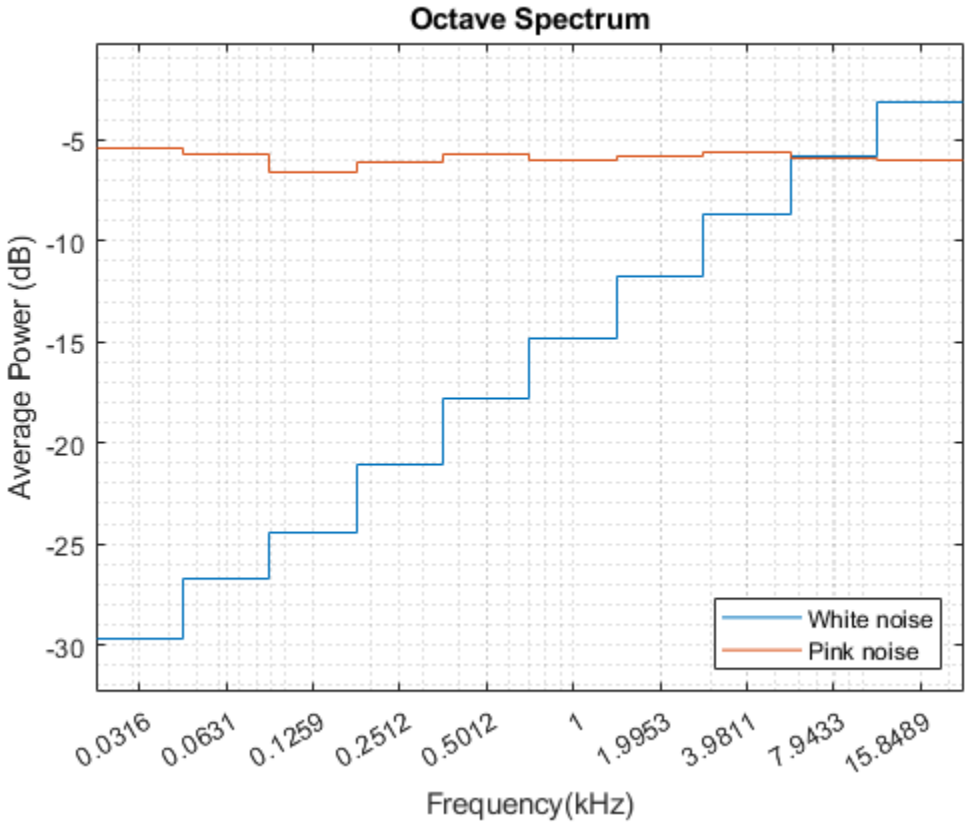
```
N = 1e5;
wn = randn(N,1);

z = [0.982231570015379 0.832656605953720 0.107980893771348]';
p = [0.995168968915815 0.943841773712820 0.555945259371364]';

[b,a] = zp2tf(z,p,1);
pn = filter(b,a,wn);

zplane(z,p)
```



Create a two-channel signal consisting of white and pink noise. Compute the octave spectrum. Assume a sample rate of 44.1 kHz. Set the frequency band from 30 Hz to the Nyquist frequency.

```
sg = [wn pn];

fs = 44100;
```

```
poctave(sg,fs,'FrequencyLimits',[30 fs/2])
legend('White noise','Pink noise','Location','SouthEast')
```



The white noise has an octave spectrum that increases with frequency. The octave spectrum of the pink noise is approximately constant throughout the frequency range. The octave spectrum of a signal illustrates how the human ear perceives the signal.

Octave Smoothing of White and Pink Noise

Generate 10^5 samples of white Gaussian noise sampled at 44.1 kHz. Create a signal of pink noise by filtering the white noise with a filter whose zeros and poles are all on the positive *x*-axis.

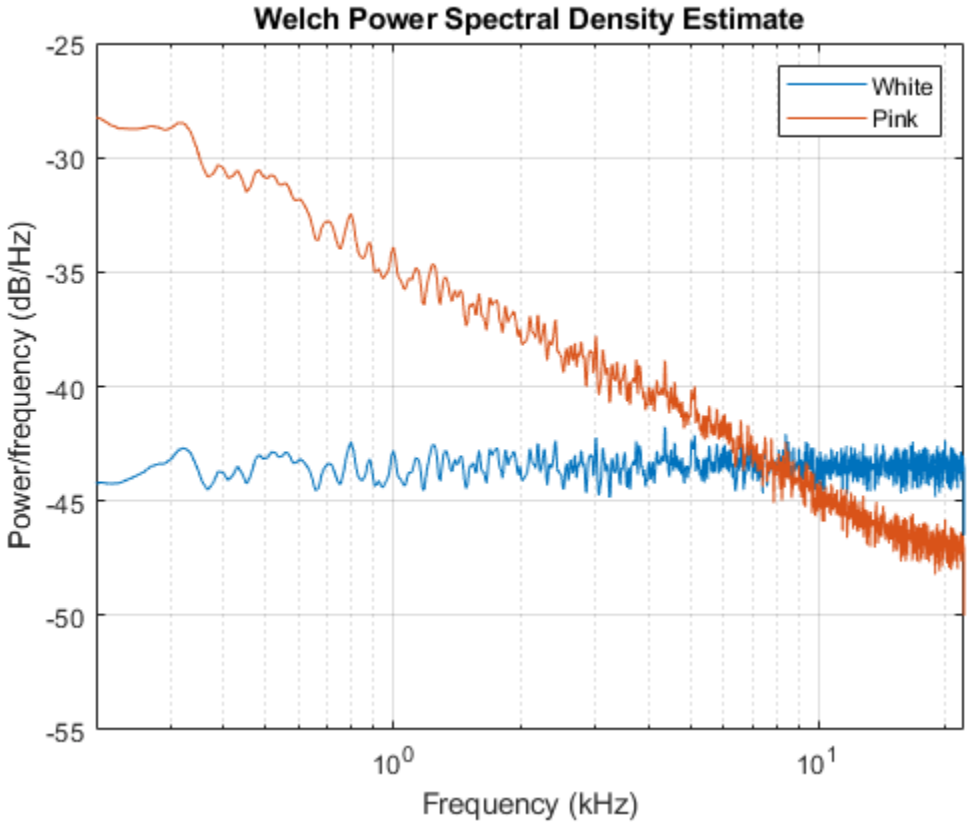
```
N = 1e5;
fs = 44.1e3;
wn = randn(N,1);

z = [0.982231570015379 0.832656605953720 0.107980893771348]';
p = [0.995168968915815 0.943841773712820 0.555945259371364]';
[b,a] = zp2tf(z,p,1);

pn = filter(b,a,wn);
```

Compute the Welch estimate of the power spectral density for both signals. Divide the signals into 2048-sample segments, specify 50% overlap between adjoining segments, window each segment with a Hamming window, and use 4096 DFT points.

```
[pxx,f] = pwelch([wn pn],hamming(2048),1024,4096,fs);
```

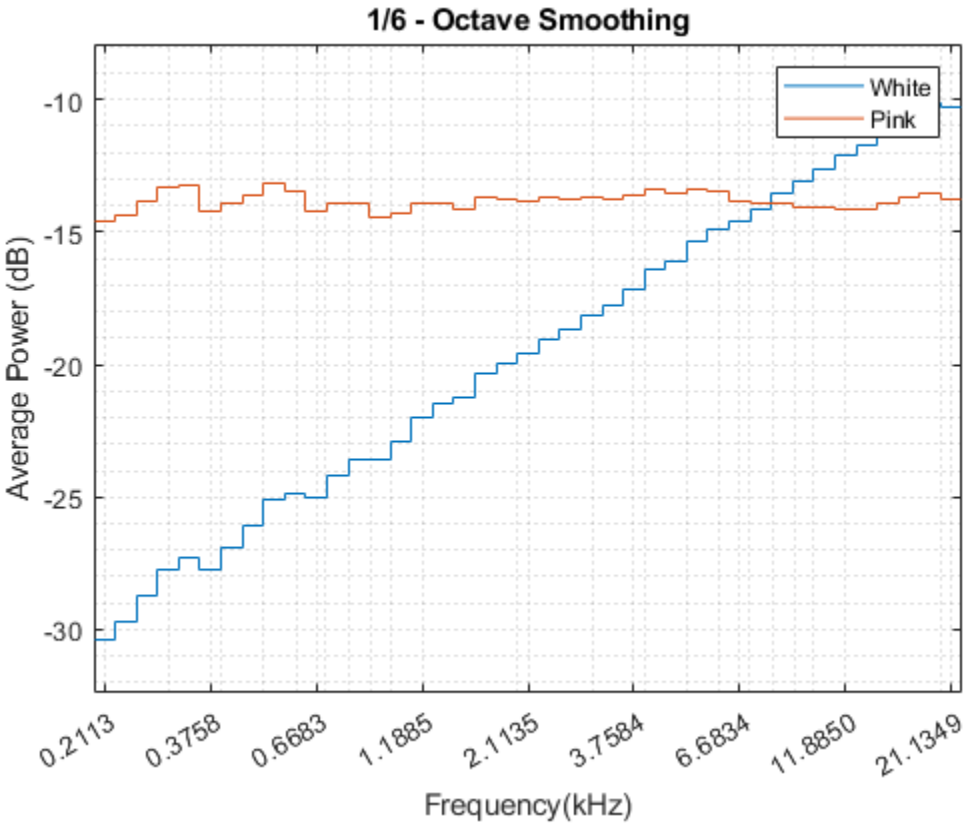
Display the spectral densities over a frequency band ranging from 200 Hz to the Nyquist frequency. Use a logarithmic scale for the frequency axis.

```
pwelch([wn pn],hamming(2048),1024,4096,fs)
ax = gca;
ax.XScale = 'log';
xlim([200 fs/2]/1000)
legend('White','Pink')
```



Compute and display the octave spectra of the signals. Use the same frequency range as in the previous plot. Specify six bands per octave and compute the spectra using 8th-order filters.

```
poctave(pxx,fs,f,'BandsPerOctave',6,'FilterOrder',8,'FrequencyLimits',[200 fs/2],'psd')
legend('White','Pink')
```



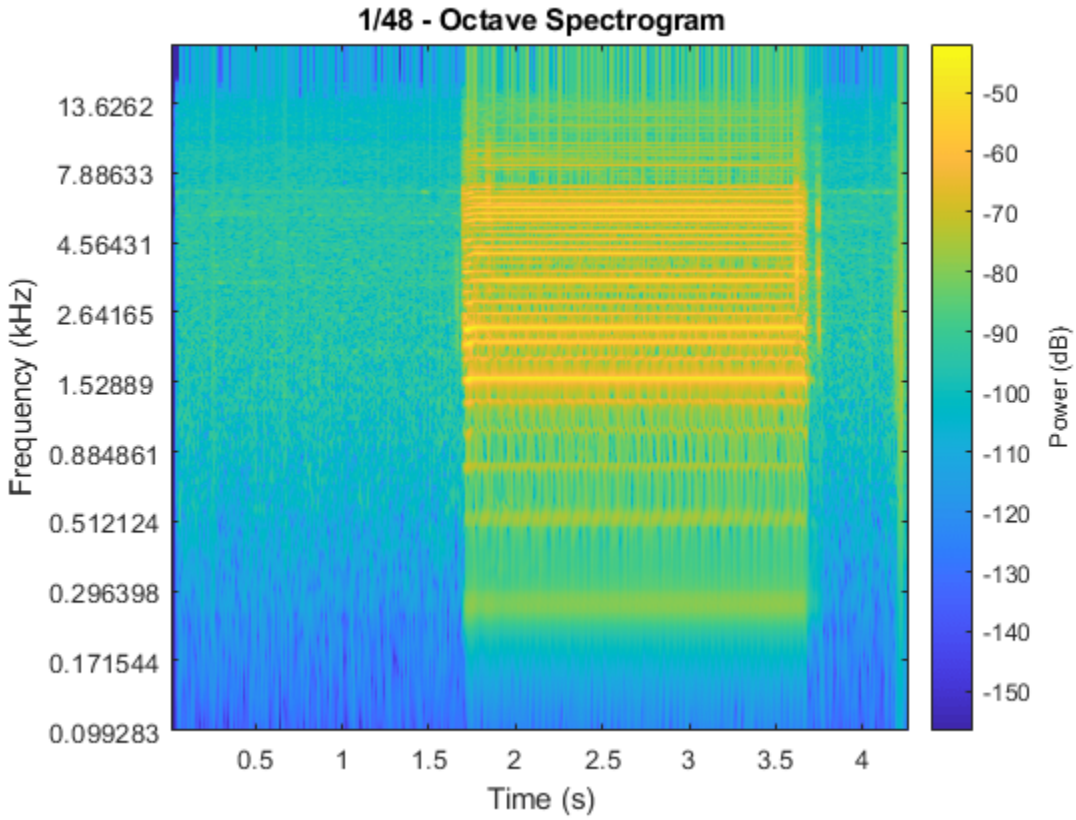
Octave Spectrogram of Audio Signal

Read an audio recording of an electronic toothbrush into MATLAB®. The toothbrush turns on at about 1.75 seconds and stays on for approximately 2 seconds.

```
[y,fs] = audioread('toothbrush.m4a');
```

Compute the octave spectrogram of the audio signal. Specify 48 bands per octave and 82% overlap. Restrict the total frequency range from 100 Hz to `fs`/2 Hz and use C-weighting.

```
poctave(y,fs,'spectrogram','BandsPerOctave',48,'OverlapPercent',82,'FrequencyLimits',[100 fs/2],
```



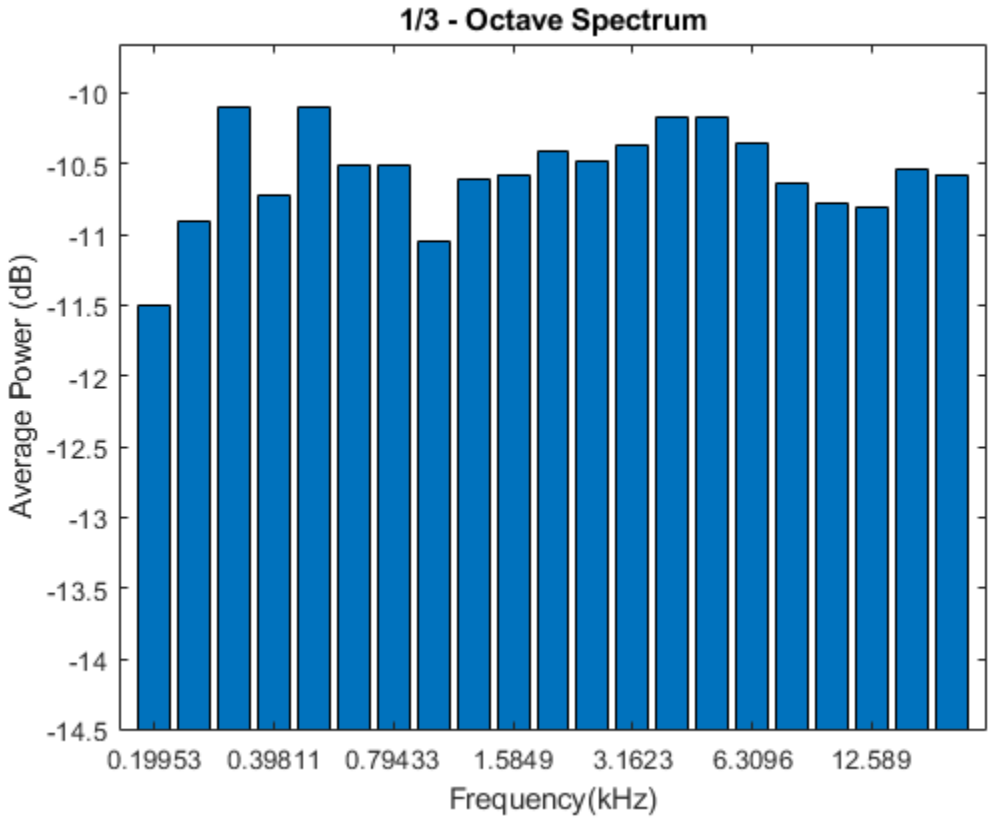
Octave Spectrum Weighting

Generate 10^5 samples of white Gaussian noise sampled at 44.1 kHz. Create a signal of pink noise by filtering the white noise with a filter whose zeros and poles are all on the positive *x*-axis.

```
N = 1e5;
fs = 44.1e3;
wn = randn(N,1);

z = [0.982231570015379 0.832656605953720 0.107980893771348]';
p = [0.995168968915815 0.943841773712820 0.555945259371364]';
[b,a] = zp2tf(z,p,1);

pn = filter(b,a,wn);
```

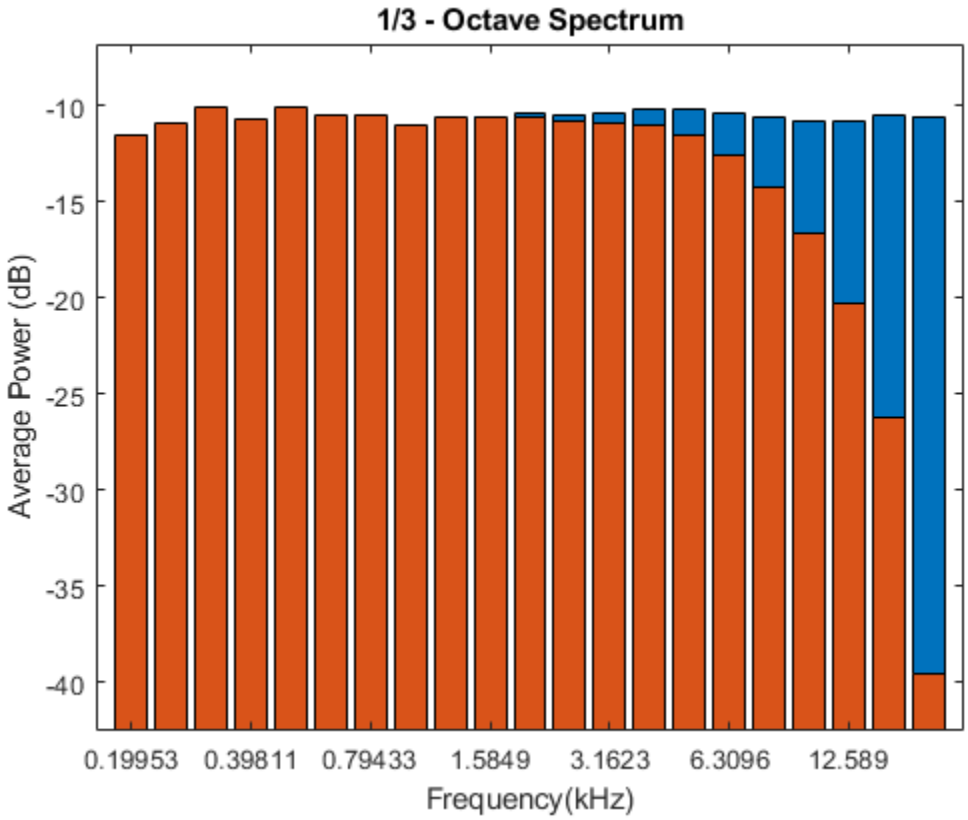
Compute the octave spectrum of the signal. Specify three bands per octave and restrict the total frequency range from 200 Hz to 20 kHz. Store the name-value pairs in a cell array for later use. Display the spectrum.

```
flims = [200 20e3];
bpo = 3;
opts = {'FrequencyLimits',flims,'BandsPerOctave',bpo};

poctave(pn,fs,opts{:});
```



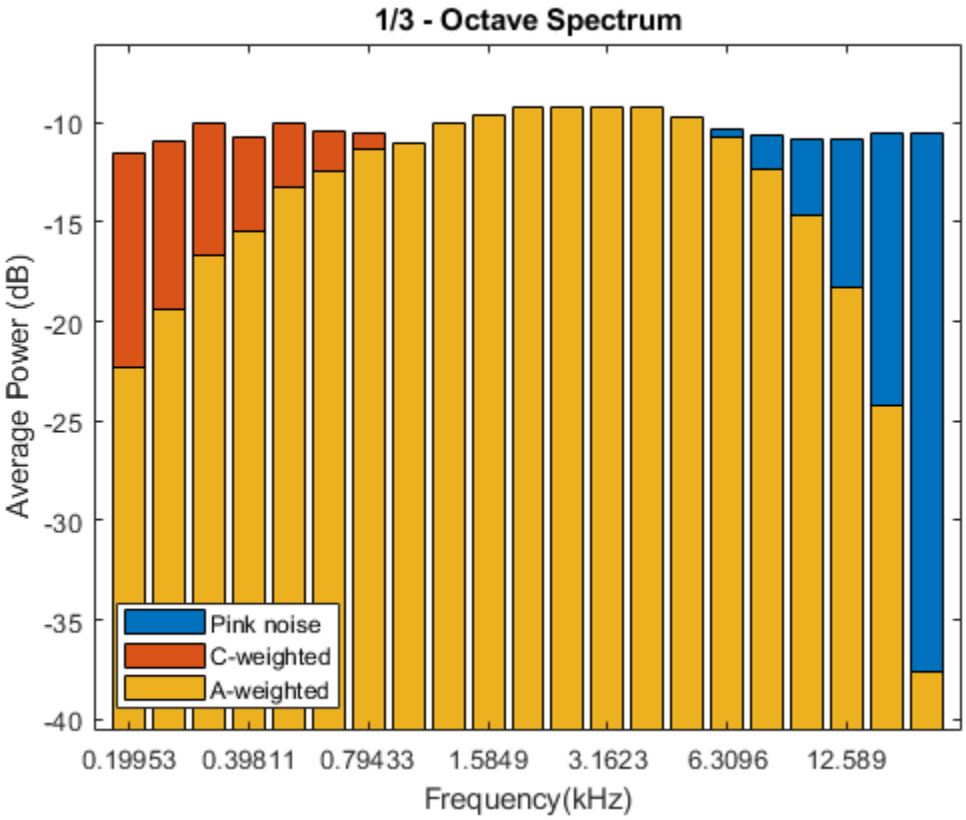
Compute the octave spectrum of the signal with the same settings, but use C-weighting. The C-weighted spectrum falls off at frequencies above 6 kHz.

```
hold on
poctave(pn,fs,opts{:},'Weighting','C')
```



Compute the octave spectrum again, but now use A-weighting. The A-weighted spectrum peaks at about 3 kHz and falls off above 6 kHz and at the lower end of the frequency band.

```
poctave(pn,fs,opts{:},'Weighting','A')
hold off
legend('Pink noise','C-weighted','A-weighted','Location','SouthWest')
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If `x` is a vector, then `poctave` treats it as a single channel. If `x` is a matrix, then `poctave` computes the octave spectrum or spectrogram independently for each column and returns the result in the corresponding column of `p`. If `type` is set to `'spectrogram'`, the function concatenates the spectrograms along the third dimension of `p`.

Example: `sin(2*pi*(0:127)/16)+randn(1,128)/100` specifies a noisy sinusoid.

Example: `[2 1].*sin(2*pi*(0:127)'./[16 64])` specifies a two-channel sinusoid.

Data Types: `single` | `double`

fs — Sample rate

positive scalar

Sample rate, specified as a positive scalar expressed in hertz. The sample rate cannot be lower than 7 Hz.

xt — Input timetable

timetable

Input timetable. `xt` must contain increasing, finite, uniformly spaced row times. If `xt` represents a multichannel signal, then it must have either a single variable containing a matrix or multiple variables consisting of vectors.

If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

Example: `timetable(seconds(0:4)',randn(5,1))` specifies a random process sampled at 1 Hz for 4 seconds.

`pxx` — Power spectral density
vector | matrix

Power spectral density (PSD), specified as a vector or matrix with real nonnegative elements. The power spectral density must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values. If `type` is `'spectrogram'`, then each column in `pxx` is considered to be the PSD for a particular time window or sample.

Example: `[pxx,f] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` specifies the periodogram PSD estimate of a noisy two-channel sinusoid sampled at 2π Hz and the frequencies at which it is computed.

`f` — PSD frequencies
vector

PSD frequencies, specified as a vector. `f` must be finite, strictly increasing, and uniformly spaced in the linear scale.

Example: `[pxx,f] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` specifies the periodogram PSD estimate of a noisy two-channel sinusoid sampled at 2π Hz and the frequencies at which it is computed.

`type` — Type of spectrum to compute
`'power'` (default) | `'spectrogram'`

Type of spectrum to compute, specified as `'power'` or `'spectrogram'`.

- `'power'` — Compute the octave power spectrum of the input.
- `'spectrogram'` — Compute the octave spectrogram of the input. The function divides the input into segments and returns the short-time octave power spectrum of each segment.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Weighting','A','FilterOrder',8` computes the octave spectrum using A-weighting and 8th-order filters.

`BandsPerOctave` — Number of subbands in octave band
`1` (default) | `3/2` | `2` | `3` | `6` | `12` | `24` | `48` | `96`

Number of subbands in the octave band, specified as `1`, `3/2`, `2`, `3`, `6`, `12`, `24`, `48`, or `96`. This parameter dictates the width of a fractional-octave band. In such a frequency band, the upper edge frequency is the lower edge frequency times $2^{1/b}$, where b is the number of subbands.

Data Types: `single` | `double`

`FilterOrder` — Order of bandpass filters
`6` (default) | positive even integer

Order of bandpass filters, specified as a positive even integer.

Data Types: `single` | `double`

`FrequencyLimits` — Frequency band
`[max(3,3*fs/48e3) fs/2]` (default) | two-element vector

Frequency band, specified as an increasing two-element vector expressed in hertz. The lower value of the vector must be at least 3 Hz. The upper value of the vector must be smaller than or equal to the Nyquist frequency. If the vector does not contain an octave center, `poctave` may return a center frequency outside the specified limits. To ensure a stable filter design, the actual minimum achievable frequency limit increases to `3*fs/48e3` if the sample rate exceeds 48 kHz. If this argument is not specified, `poctave` uses the interval `[max(3,3*fs/48e3) fs/2]`.

Data Types: `single` | `double`

`Weighting` — Frequency weighting
`'none'` (default) | `'A'` | `'C'` | vector | matrix | 1-by-2 cell array | `digitalFilter` object

Frequency weighting, specified as one of these:

- `'none'` — `poctave` does not perform any frequency weighting on the input.
- `'A'` — `poctave` performs A-weighting on the input. The ANSI S1.42 standard defines the A-weighting curve. The IEC 61672-1 standard defines the minimum and maximum attenuation limits for an A-weighting filter. The ANSI S1.42.2001 standard defines the weighting curve by specifying analog poles and zeros.
- `'C'` — `poctave` performs C-weighting on the input. The ANSI S1.42 standard defines the C-weighting curve. The IEC 61672-1 standard defines the minimum and maximum attenuation limits for a C-weighting filter. The ANSI S1.42.2001 standard defines the weighting curve by specifying analog poles and zeros.
- Vector — `poctave` treats the input as a vector of coefficients that specify a finite impulse response (FIR) filter.
- Matrix — `poctave` treats the input as a matrix of second-order section coefficients that specify an infinite impulse response (IIR) filter. The matrix must have at least two rows and exactly six columns.
- 1-by-2 cell array — `poctave` treats the input as the numerator and denominator coefficients, in that order, that specify the transfer function of an IIR filter.
- `digitalFilter` object — `poctave` treats the input as a filter that was designed using `designfilt`.

This argument is supported only when the input is a signal. Octave smoothing does not support frequency weighting.

Example: `'Weighting',fir1(30,0.5)` specifies a 30th-order FIR filter with a normalized cutoff frequency of 0.5π rad/sample.

Example: `'Weighting',[2 4 2 6 0 2;3 3 0 6 0 0]` specifies a third-order Butterworth filter with a normalized 3-dB frequency of 0.5π rad/sample.

Example: `'Weighting',{[1 3 3 1]/6 [3 0 1]/3}` specifies a third-order Butterworth filter with a normalized 3-dB frequency of 0.5π rad/sample.

Example: `'Weighting',designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)` specifies a third-order Butterworth filter with a normalized 3-dB frequency of 0.5π rad/sample.

Data Types: `single` | `double` | `char` | `string` | `cell`

`MinThreshold` — Lower bound for nonzero values
`-Inf` (default) | real scalar

Lower bound for nonzero values, specified as a real scalar. The function sets those elements of `p` such that $10\log_{10}(p) \le$ `'MinThreshold'` to zero. Specify `'MinThreshold'` in decibels.

Data Types: `single` | `double`

`WindowLength` — Length of data segments
nonnegative integer

Length of data segments, specified as a nonnegative integer. `'WindowLength'` must be less than or equal to the length of the input signal. If not specified, the length of data segments is calculated based on the size of the input signal. This input is valid only when `type` is `'spectrogram'`.

Data Types: `single` | `double`

`OverlapPercent` — Overlap percent between adjoining segments
real scalar in the interval [0, 100)

Overlap percent between adjoining segments, specified as a real scalar in the interval [0, 100). If not specified, `'OverlapPercent'` is zero. This input is valid only when `type` is `'spectrogram'`.

Data Types: `single` | `double`

Output Arguments

`p` — Octave spectrum or spectrogram
vector | matrix | 3-D array

Octave spectrum or spectrogram, returned as a vector, matrix, or 3-D array. The third dimension, if present, corresponds to the input channels.

`cf` — Center frequencies
vector

Center frequencies, returned as a vector. `cf` contains a list of center frequencies of the octave bands over which `poctave` estimated the octave spectrum. `cf` has units of hertz.

`t` — Center times
vector

Center times, returned as a vector. If the input is a PSD, then `t` represents the sample indices corresponding to the columns of `pxx`. This argument applies only when `type` is `'spectrogram'`.

Algorithms

Octave analysis is used to identify sound or vibration levels across a broad frequency range in a process that resembles how a human ear perceives sound. The signal spectrum is split into octave or fractional-octave bands. The frequency limit of each band is twice the lower frequency limit, thus the bandwidth increases at higher frequencies.

Using Octave Filters

To perform octave analysis, the `poctave` function creates a filter bank of parallel bandpass filters. Each digital bandpass filter is mapped to an equivalent Butterworth lowpass analog filter [3]. The analog filter is mapped back to a digital bandpass filter using a bandpass version of the `bilinear` transformation, and the result is returned as a cascade of fourth-order sections.

The lower and upper edge frequencies of each octave band are given by

$$f_l = cf \cdot (G^{-1/2b})$$

$$f_u = cf \cdot (G^{1/2b})$$

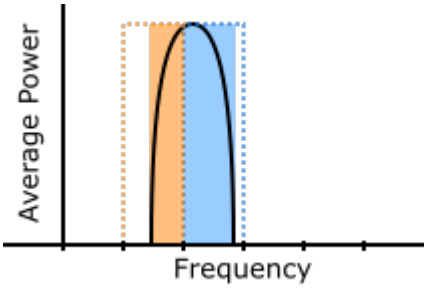
where f_c is the center frequency of each band defined by the ANSI S1.11-2004 standard [2] and returned in `cf`, G is a conversion constant ($10^{3/10}$), and b is the number of bands per octave.

For more information on the design and implementation of octave filters, see “Digital Filter Design” (Audio Toolbox).

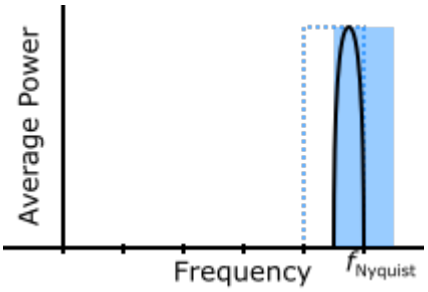
Using Octave Smoothing

The `poctave` function calculates the average power over each octave band by integrating the power spectral density (PSD) of the signal within the band using the rectangle method. The average power of an octave band represents the signal level at the band center frequency.

- When a band edge falls within a bin, the function assigns to the band only the fraction of power corresponding to the percentage of the frequency bin that the band occupies. For example, this diagram shows an octave band whose edges fall within two different frequency bins, represented by orange and blue dashed rectangles. The power within the shaded regions is computed for the given octave band.



- When a band edge falls at 0 or at the Nyquist frequency, f_{Nyquist}, the function assigns to the band two times the fraction of power corresponding to the percentage of the frequency bin that the band occupies. This duplication accounts for the half bin power that is present in the range [–w/2, `0`] and [f_{Nyquist}, f_{Nyquist} + w/2], where w is the bin width. For example, this diagram shows an octave band whose right edge falls at the Nyquist frequency. The power within the shaded region is computed for the given octave band.



References

[1] Smith, Julius Orion, III. "Example: Synthesis of 1/F Noise (Pink Noise)." In *Spectral Audio Signal Processing*. https://ccrma.stanford.edu/~jos/sasp/.

[2] *Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters*. ANSI Standard S1.11-2004. Melville, NY: Acoustical Society of America, 2004.

[3] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 2010.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If input weighting is specified as a variable-sized matrix during code generation, then it must not reduce to a vector at runtime.

See Also

`pspectrum`

Introduced in R2018a

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 = 6×1

    5.5917
   -1.7277
   -4.4231
    4.3985
    1.6426
   -5.3126
```

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

See Also

`ac2poly` | `poly2rc` | `rc2ac`

Introduced before R2006a

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 = 5×1

    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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

- Poorly conditioned prediction polynomials generate line spectral frequencies that do not always match MATLAB.

See Also

`lsf2poly`

Introduced before R2006a

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 = 5×1

    0.3090
    0.9801
    0.0031
    0.0081
   -0.0082


r0 = 5.6032
```

Limitations

If `abs(k(i)) == 1` for any `i`, finding the reflection coefficients is an ill-conditioned problem. `poly2rc` returns some `NaN`s and provides a warning message in those cases.

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:

$$k(n) = a_n(n)$$

$$a_{n-1}(m) = \frac{a_n(m) - k(n)a_n(n-m)}{1 - k(n)^2}, \quad m = 1, 2, \cdots, n-1$$

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "Variable-Sizing Restrictions for Code Generation of Toolbox Functions" (MATLAB Coder).

See Also

`ac2rc` | `latc2tf` | `latcfilt` | `poly2ac` | `rc2poly` | `tf2latc`

Introduced before R2006a

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.

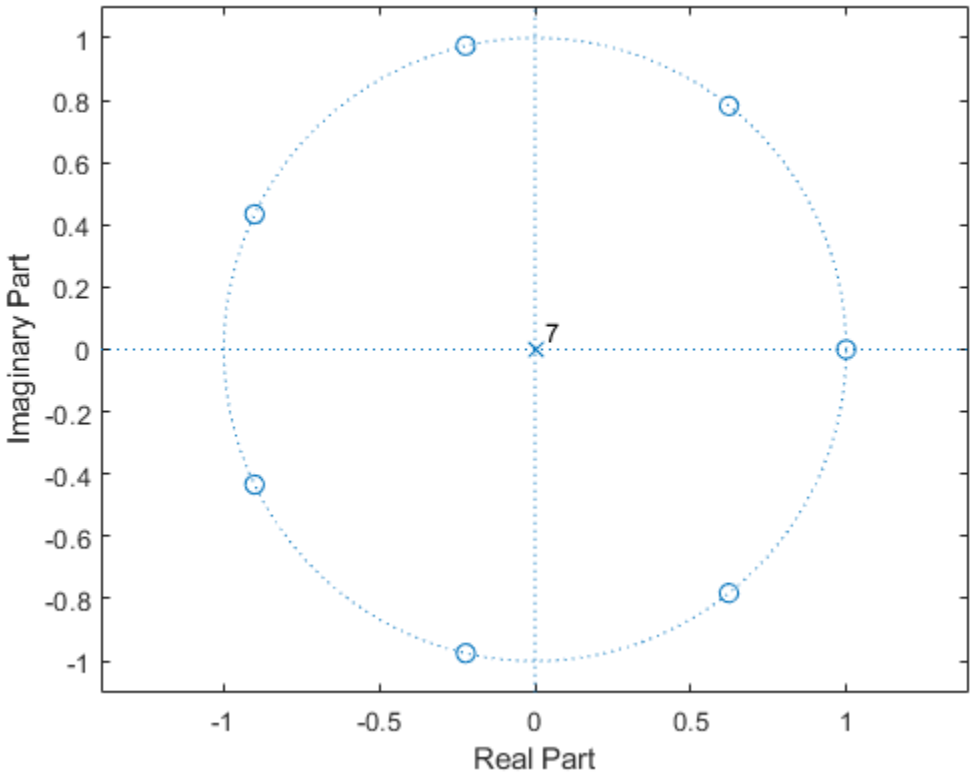
If `alpha` is a real value in the range `[0 1]`, then the roots of `a` are radially scaled toward the origin in the *z*-plane. Complex values for `alpha` allow arbitrary changes to the root locations.

Examples

Roots of Unity

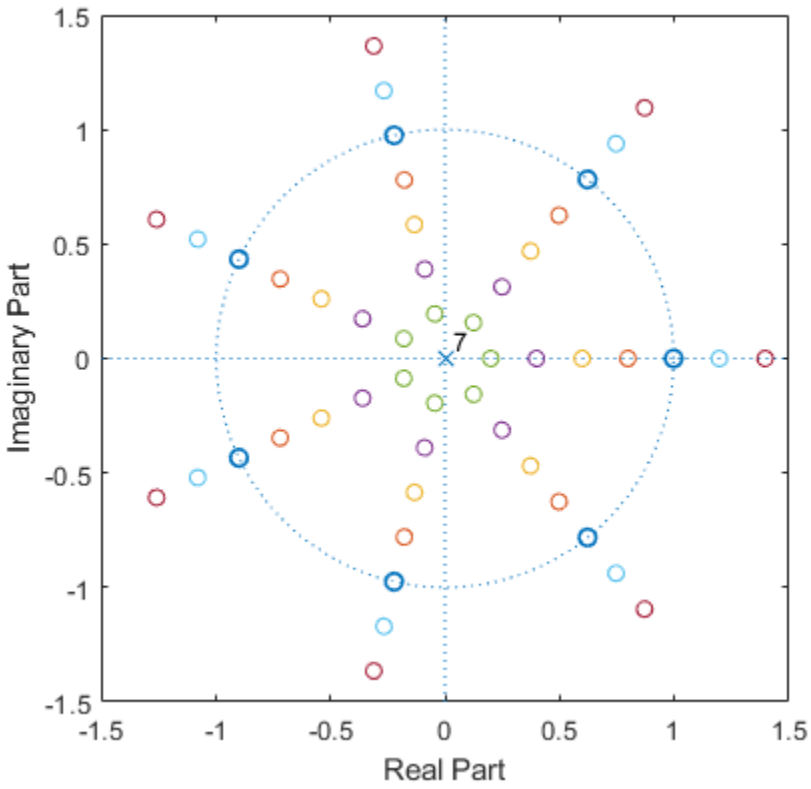
Express the solutions to the equation $x^7 = 1$ as the roots of a polynomial. Plot the roots in the complex plane.

```
pp = [1 0 0 0 0 0 0 -1];
zplane(pp,1)
```



Scale the roots of p in and out of the unit circle. Plot the results.

```
hold on

for sc = [1:-0.2:0.2 1.2 1.4];
    b = polyscale(pp,sc);
    plot(roots(b),'o')
end

axis([-1 1 -1 1]*1.5)

hold off
```



Bandwidth Expansion of LPC Speech Spectrum

Load a speech signal sampled at F_s = 7418 Hz. The file contains a recording of a female voice saying the word "MATLAB®."

```
load mtlb
```

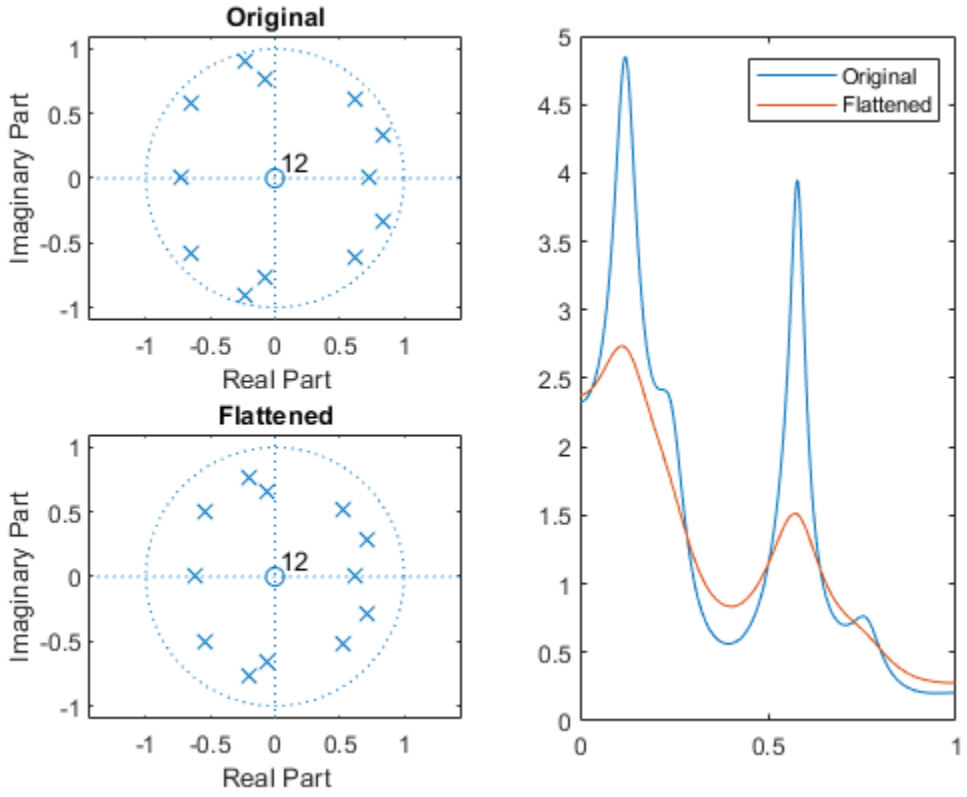
Model a 100-sample section of the signal using a 12th-order autoregressive polynomial. Perform bandwidth expansion of the signal by scaling the roots of the autoregressive polynomial by 0.85.

```
Ao = lpc(mtlb(1000:1100),12);
Ax = polyscale(Ao,0.85);
```

Plot the zeros, poles, and frequency responses of the models.

```
subplot(2,2,1)
zplane(1,Ao)
title('Original')

subplot(2,2,3)
zplane(1,Ax)
title('Flattened')

subplot(1,2,2)
[ho,w] = freqz(1,Ao);
```

```
[hx,w] = freqz(1,Ax);
plot(w/pi,abs([ho hx]))
legend('Original','Flattened')
```



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`

Introduced before R2006a

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:

$$A(z) = a(1) + a(2)z^{-1} + \ldots + a(m+1)z^{-m}.$$

Examples

Convert Linear-Phase Filter to Minimum-Phase

Use the window method to design a 25th-oder FIR filter with normalized cutoff frequency 0.4π rad/sample. Verify that it has linear phase but not minimum phase.

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

h_linphase = islinphase(h)
```

```
h_linphase = logical
   1
```

```
h_minphase = isminphase(h)
```

```
h_minphase = logical
   0
```

Use `polystab` to convert the linear-phase filter into a minimum-phase filter. Plot the phase responses of the filters.

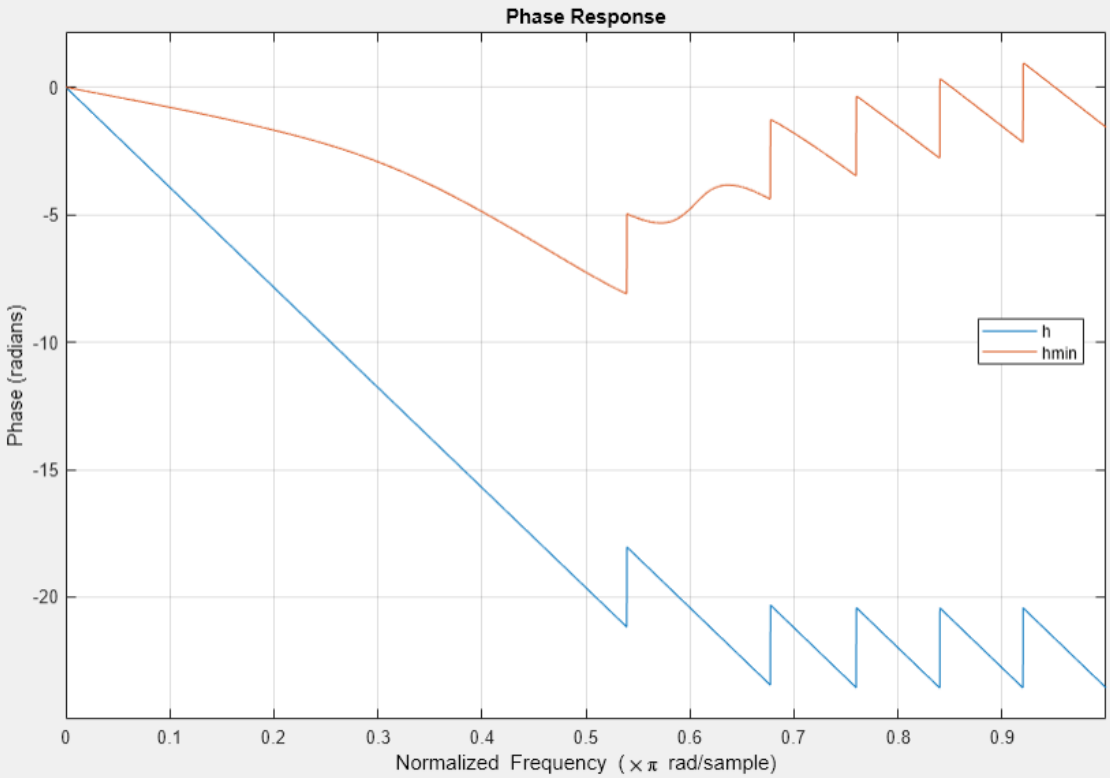
```
hmin = polystab(h)*norm(h)/norm(polystab(h));

hmin_linphase = islinphase(hmin)
```

```
hmin_linphase = logical
   0
```

```
hmin_minphase = isminphase(hmin)
```

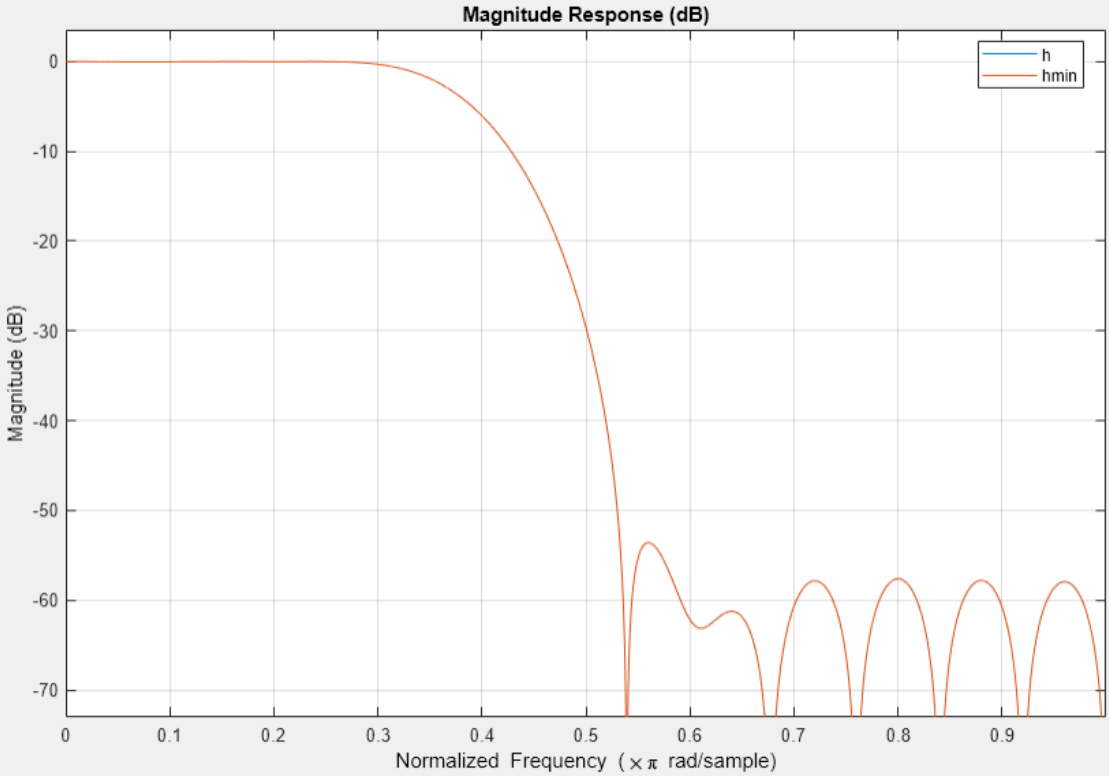
```
hmin_minphase = logical
   1
```

```
hfvt = fvtool(h,1,hmin,1,'Analysis','phase');
legend(hfvt,'h','hmin')
```



Verify that the two filters have identical magnitude responses.

```
hfvt = fvtool(h,1,hmin,1);
legend(hfvt,'h','hmin')
```



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`

Introduced before R2006a

pow2db

Convert power to decibels

Syntax

```
ydb = pow2db(y)
```

Description

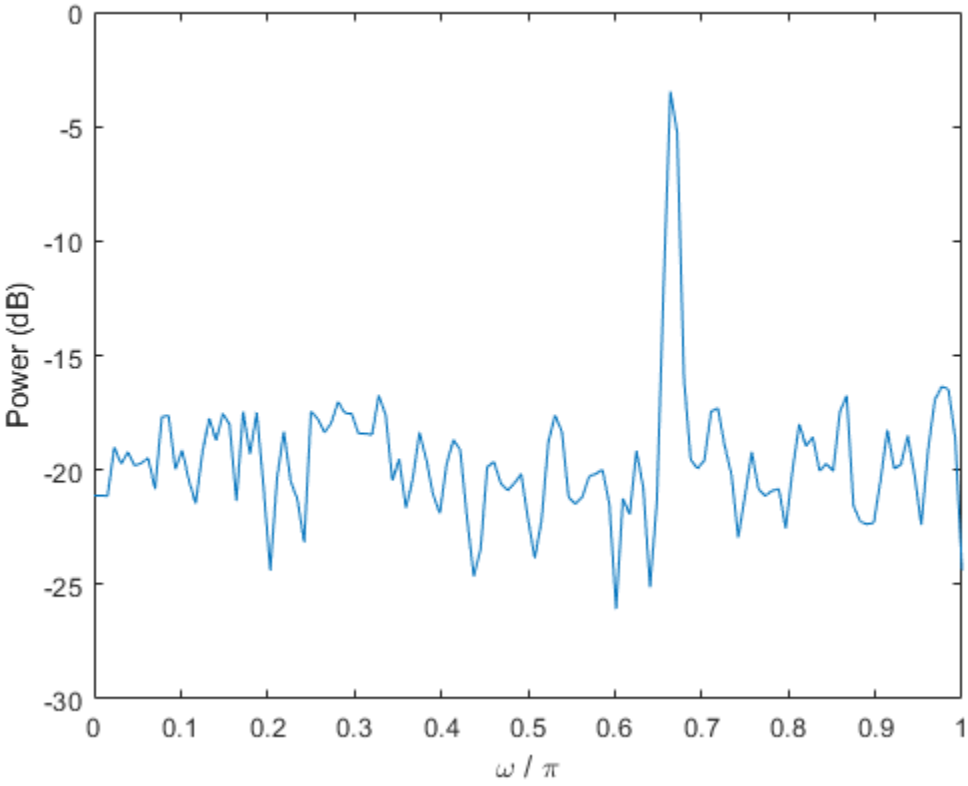
`ydb = pow2db(y)` expresses in decibels (dB) the power measurements specified in `y`. The relationship between power and decibels is `ydb` = 10 $\log_{10}$(`y`).

Examples

Power Spectrum of a Noisy Sinusoid

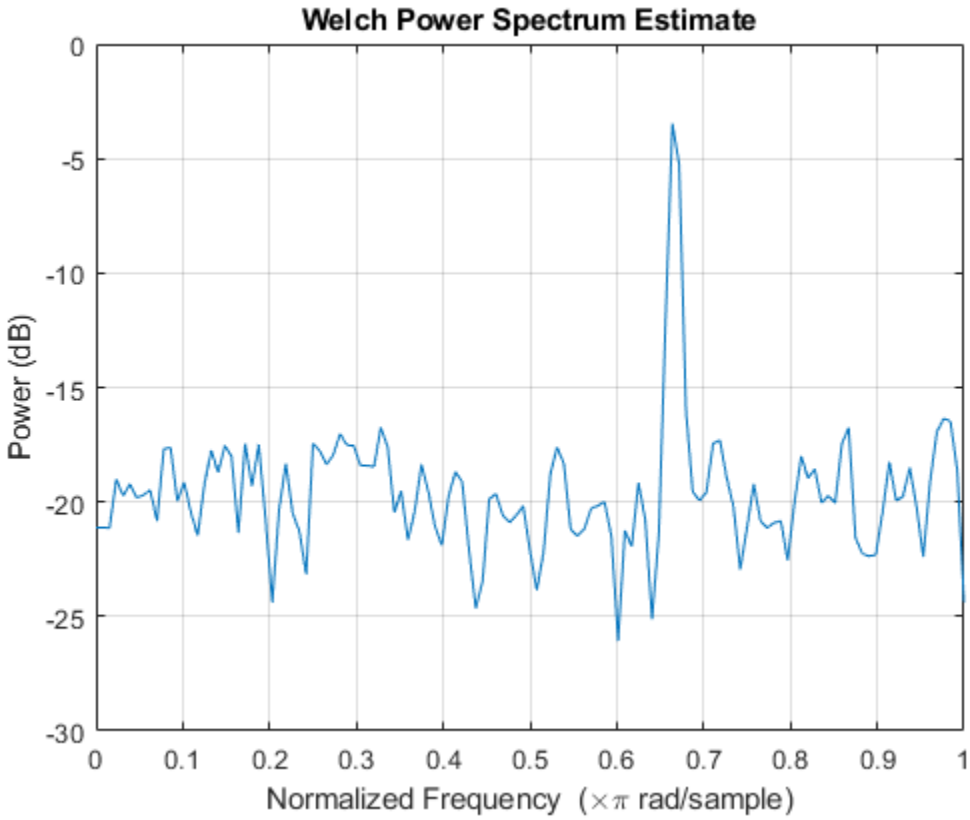
Generate 1024 samples of a noisy sinusoid having a normalized frequency of $2\pi/3$ rad/sample. Estimate the power spectrum of the signal using `pwelch`. Express the estimate in decibels and plot it.

```
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`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`db` | `db2mag` | `db2pow` | `mag2db`

Introduced in R2007b

powerbw

Power bandwidth

Syntax

```
bw = powerbw(x)
bw = powerbw(x,fs)

bw = powerbw(pxx,f)
bw = powerbw(sxx,f,rbw)

bw = powerbw( ___ ,freqlims,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( ___ ,freqlims,r)` specifies the frequency interval over which to compute the reference level. This syntax can include any combination of input arguments from previous syntaxes, as long as the second input argument is either `fs` or `f`. If the second input is passed as empty, normalized frequency will be assumed. `freqlims` 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

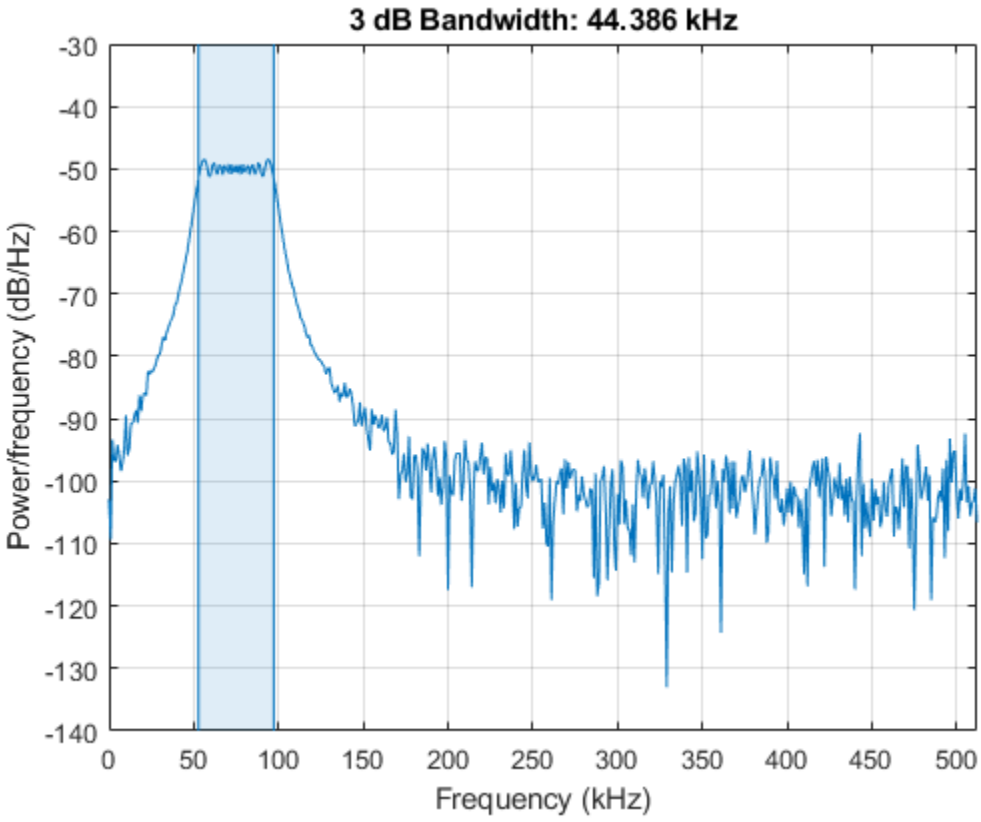
3-dB Bandwidth of Chirps

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.

```
nSamp = 1024;
Fs = 1024e3;
SNR = 40;

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);
```

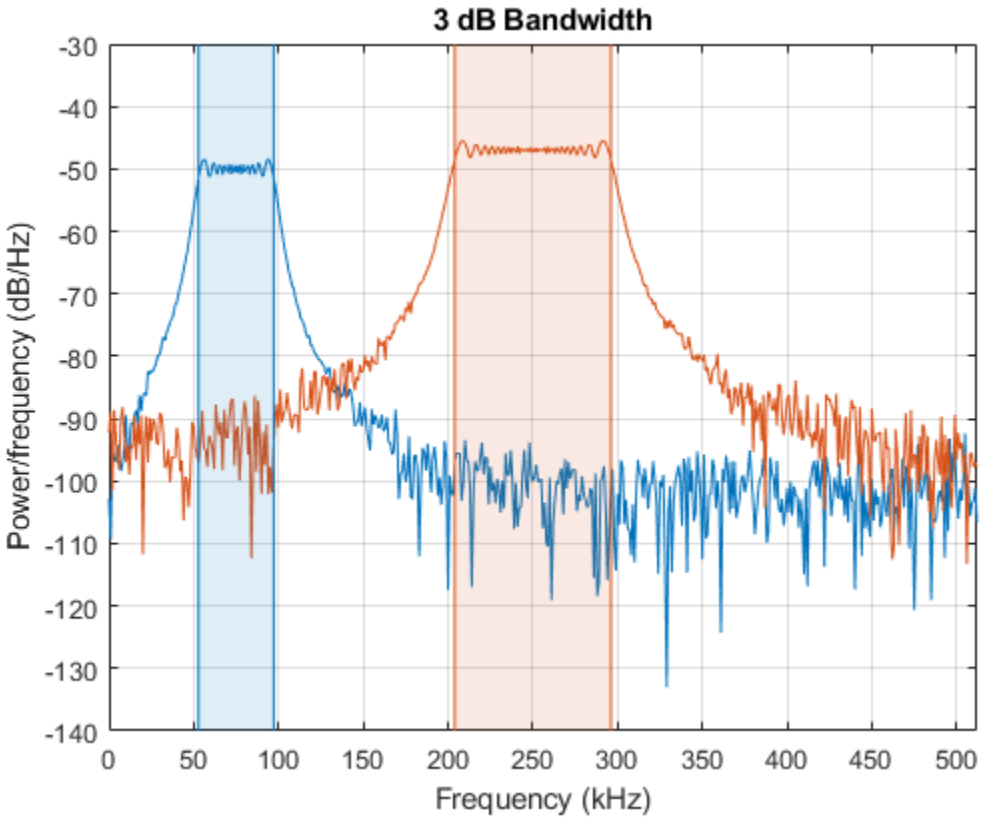
Concatenate the chirps to produce a two-channel signal. Estimate the 3-dB bandwidth of each channel.

```
y = powerbw([x x2],Fs)
```

```
y = 1×2
10^4 ×

    4.4386    9.2208
```

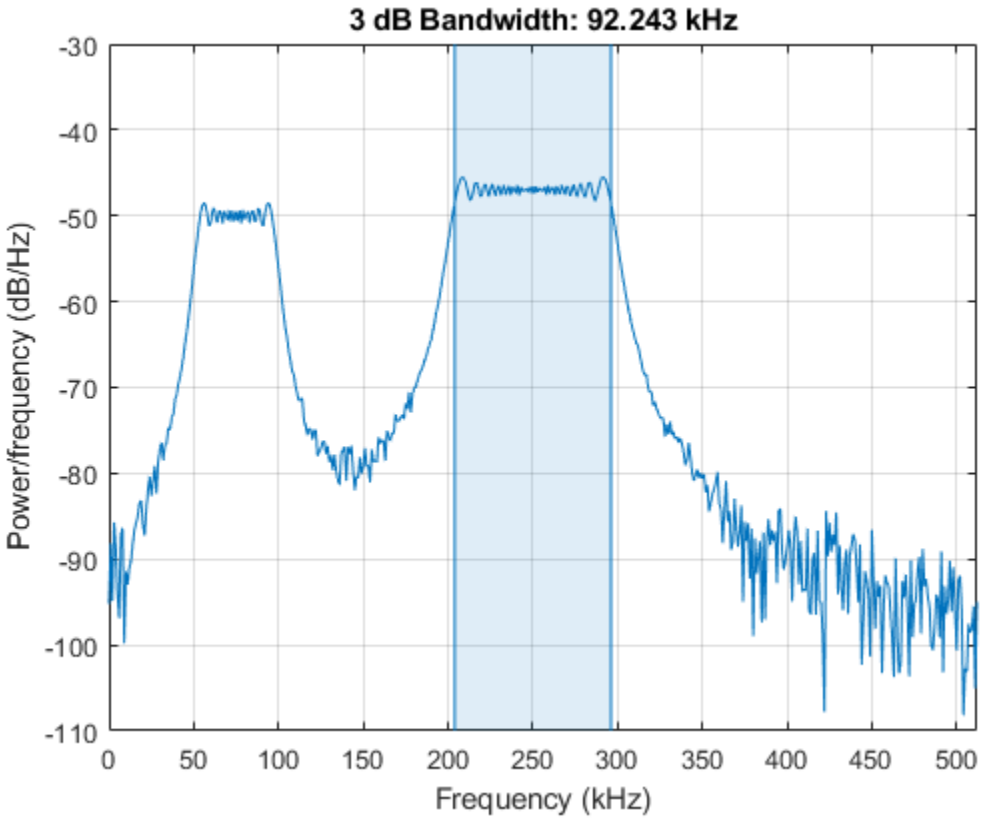
Annotate the 3-dB bandwidths of the two channels on a plot of the PSDs.

```
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
```

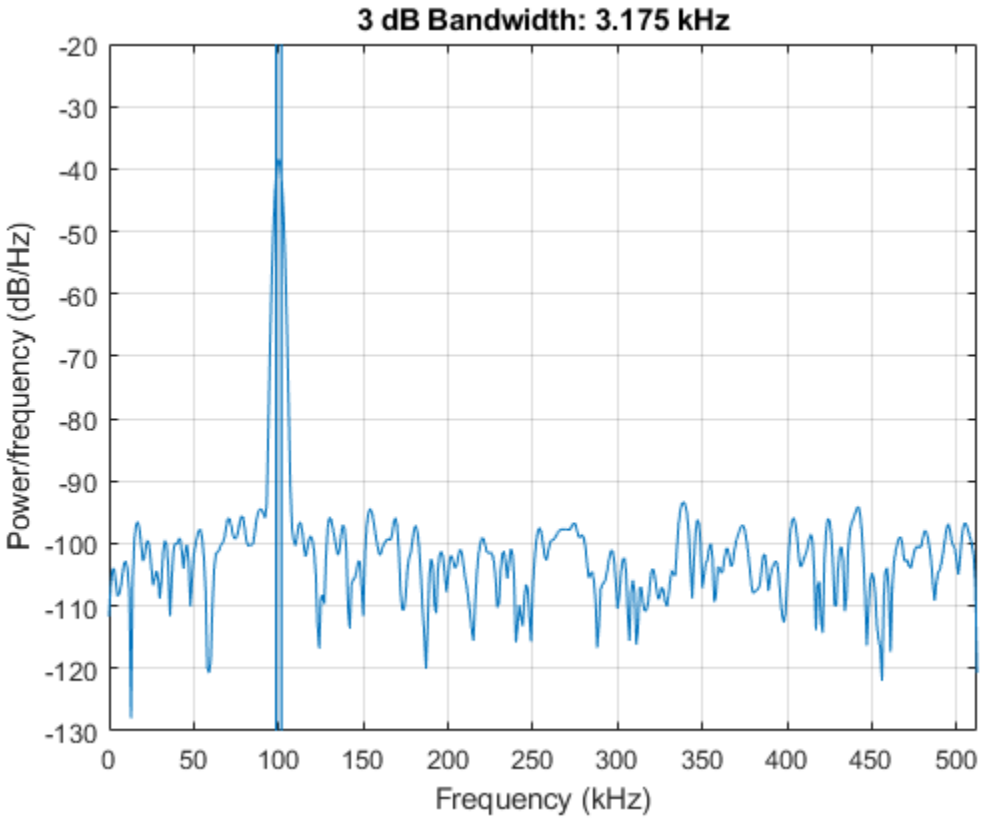
3-dB Bandwidth of Sinusoids

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.

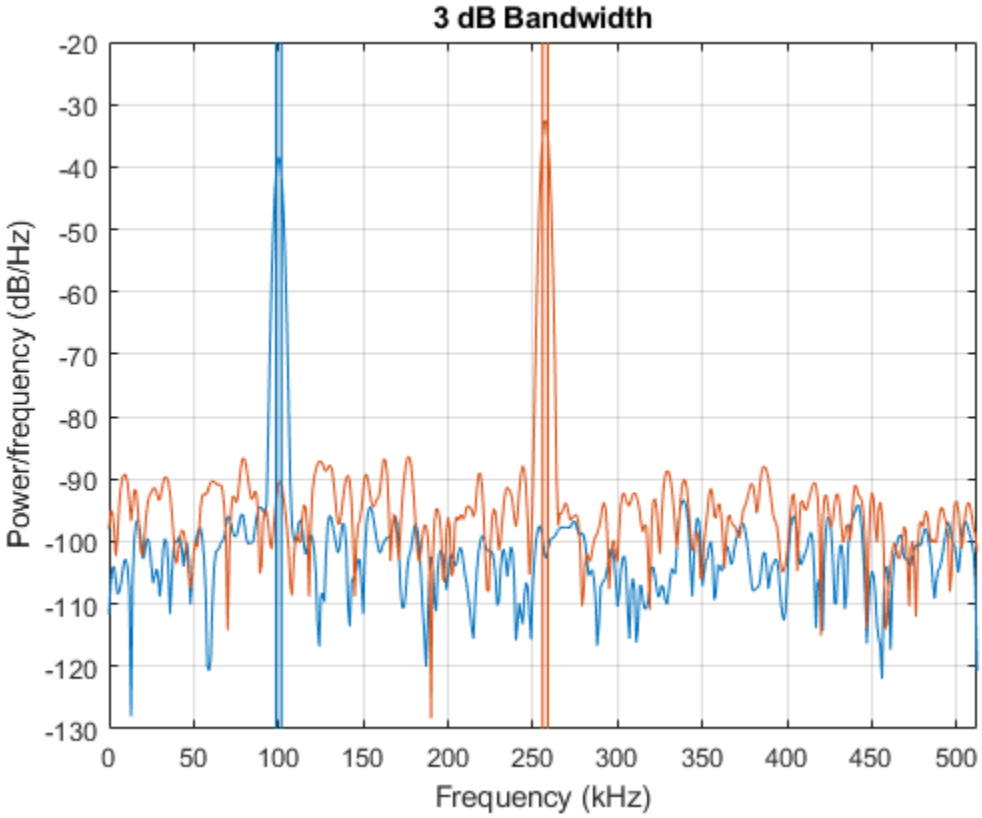
```
[Pyy,f] = periodogram([x x2],kaiser(nSamp,38),[],Fs);
```

```
y = powerbw(Pyy,f)
```

```
y = 1×2
10^3 ×

    3.1753    3.3015
```

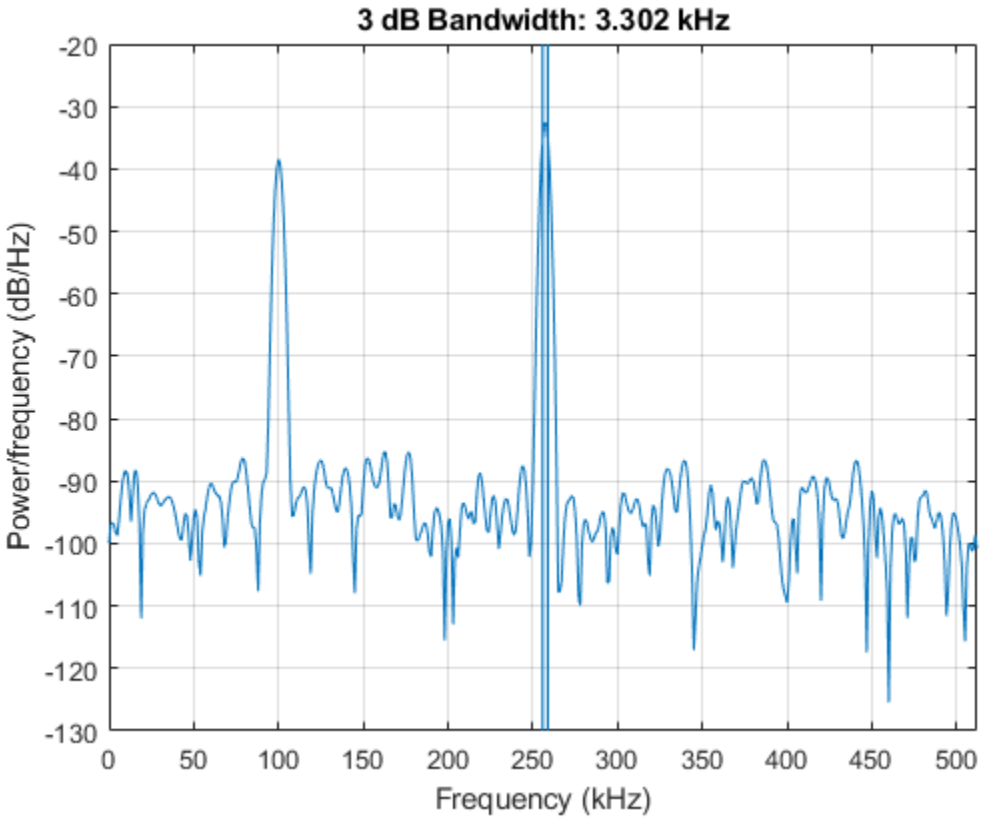
Annotate the 3-dB bandwidths of the two channels on a plot of the PSDs.

```
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);
```



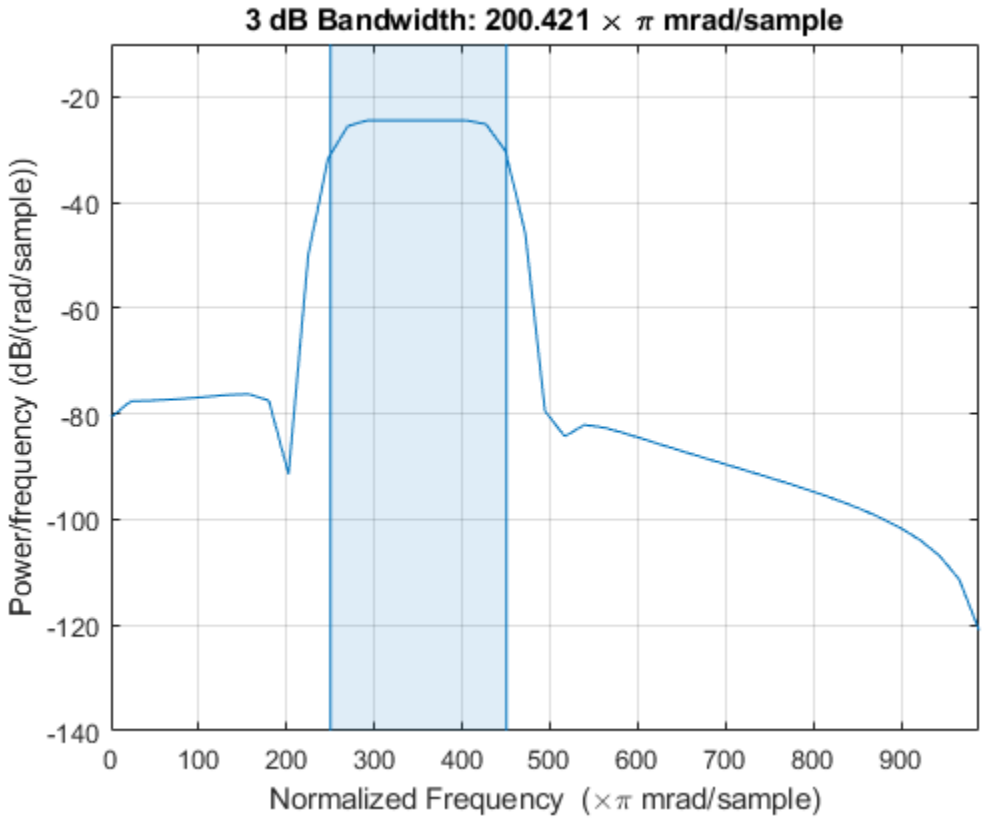
Bandwidth of Bandlimited Signals

Generate a signal whose PSD resembles the frequency response of an 88th-order bandpass FIR filter with normalized cutoff frequencies 0.25π rad/sample and 0.45π 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 0.2π rad/sample and 0.6π 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 2π 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)
```

```
bw = 0.200*pi, flo = 0.250*pi, fhi = 0.450*pi
```

```
fprintf('power = %.1f%% of total',power/bandpower(d)*100)
```

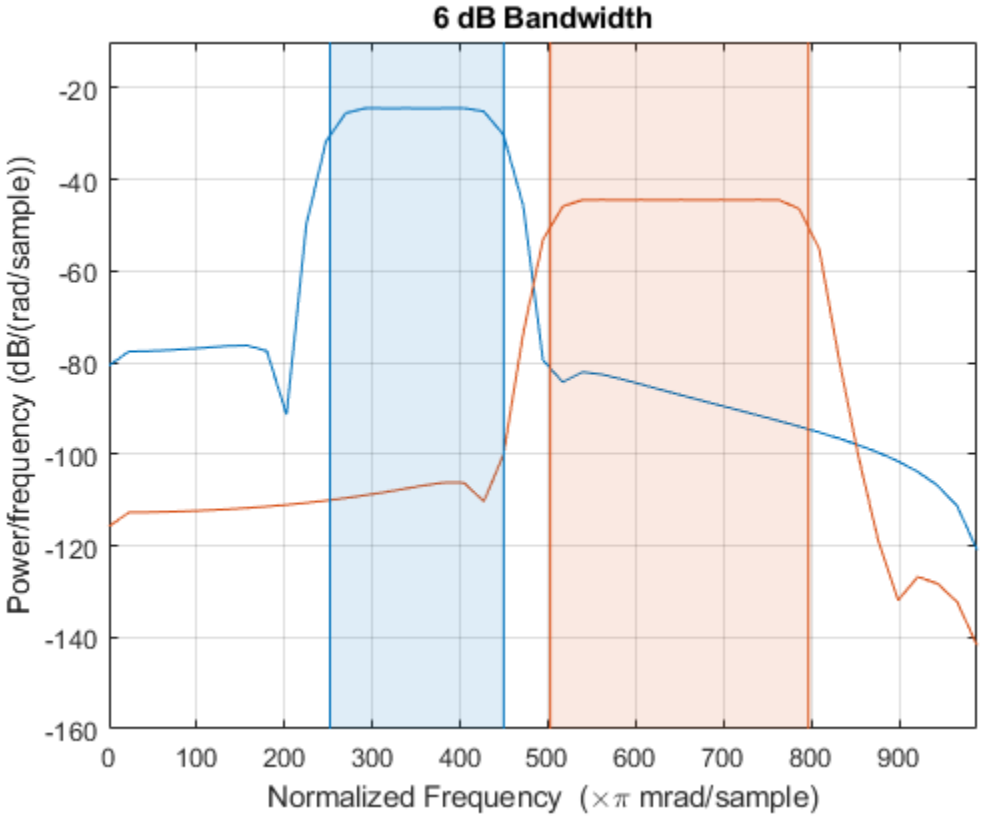
```
power = 96.9% of total
```

Add a second channel with normalized cutoff frequencies 0.5π rad/sample and 0.8π 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)
```

```
One: bw = 0.198*pi, flo = 0.252*pi, fhi = 0.450*pi
```

```
fprintf('Two: bw = %.3f*pi, flo = %.3f*pi, fhi = %.3f*pi \n',bds(:,2)/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.

The power spectral density must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[pxx,f] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` specifies the periodogram PSD estimate of a noisy two-channel sinusoid sampled at 2π Hz and the frequencies at which it is computed.

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.

The power spectrum must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[sxx,w] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2),'power')` specifies the periodogram power spectrum estimate of a two-channel sinusoid embedded in white Gaussian noise and the normalized frequencies at which it is computed.

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`

`freqlims` — Frequency limits
two-element vector

Frequency limits, specified as a two-element vector of real values. If you specify `freqlims`, then the reference level is the average power level in the reference band. If you do not specify `freqlims`, 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.

`flo`, `fhi` — 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.

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`bandpower` | `obw` | `periodogram` | `plomb` | `pwelch`

Introduced in R2015a

prony

Prony method for filter design

Syntax

```
[b,a] = prony(h,bord,aord)
```

Description

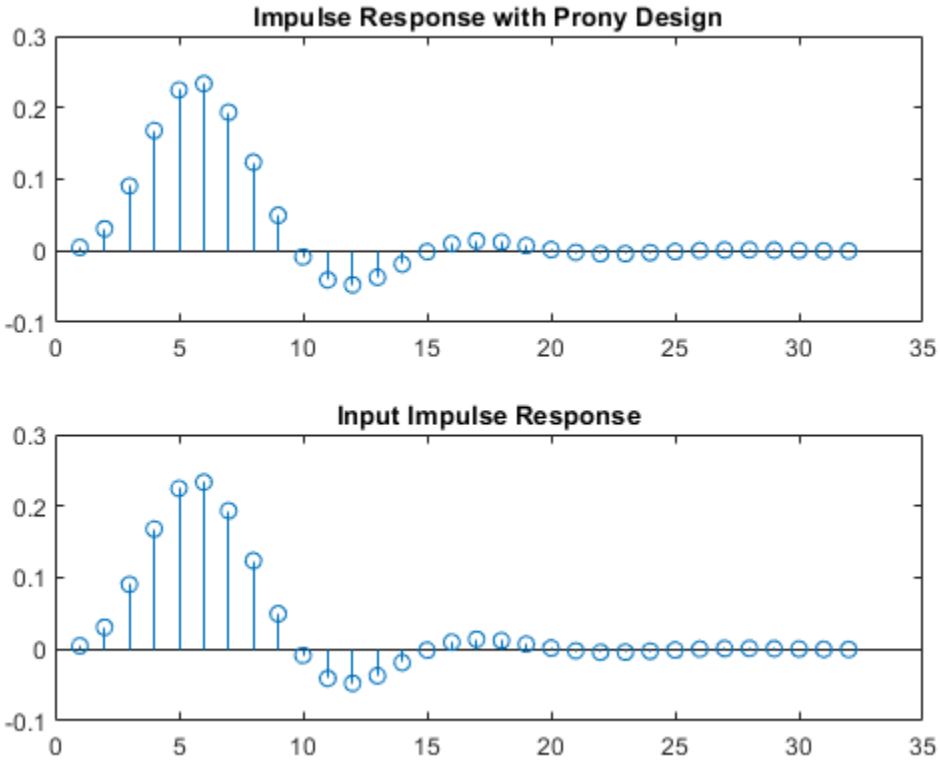
`[b,a] = prony(h,bord,aord)` returns the numerator and denominator coefficients for a causal rational transfer function with impulse response `h`, numerator order `bord`, and denominator order `aord`.

Examples

Filter Responses Using the Prony Method

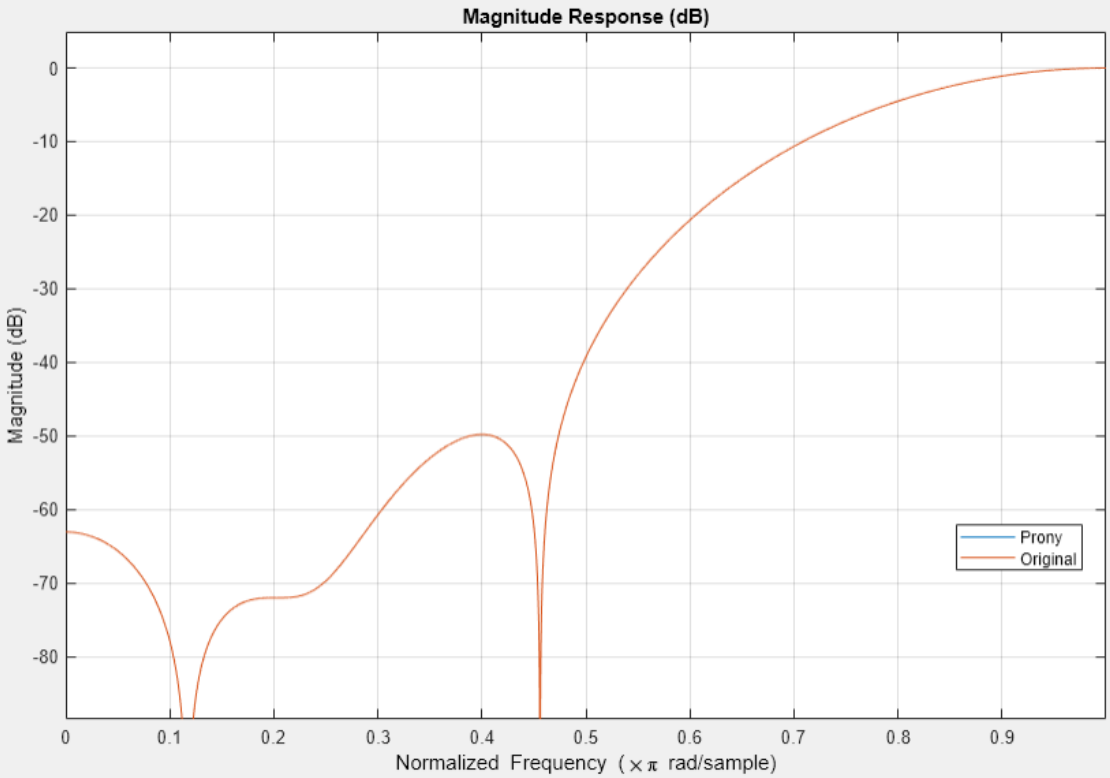
Fit a 4th-order IIR model to the impulse response of a lowpass filter. Plot the original and Prony-designed impulse responses.

```
d = designfilt('lowpassiir','NumeratorOrder',4,'DenominatorOrder',4, ...
    'HalfPowerFrequency',0.2,'DesignMethod','butter');

h = filter(d,[1 zeros(1,31)]);
bord = 4;
aord = 4;
[b,a] = prony(h,bord,aord);

subplot(2,1,1)
stem(impz(b,a,length(h)))
title 'Impulse Response with Prony Design'

subplot(2,1,2)
stem(h)
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. The responses match to high precision.

```
d = designfilt('highpassfir','FilterOrder',10,'CutoffFrequency',0.8);

h = filter(d,[1 zeros(1,31)]);
bord = 10;
aord = 0;
[b,a] = prony(h,bord,aord);

fvt = fvtool(b,a,d);
legend(fvt,'Prony','Original')
```



Input Arguments

h — Impulse response

vector

Impulse response, specified as a vector.

Example: `impz(fir1(20,0.5))` specifies the impulse response of a 20th-order FIR filter with normalized cutoff frequency $\pi/2$ rad/sample.

Data Types: `single` | `double`
Complex Number Support: Yes

bord, aord — Numerator and denominator orders

positive integer scalars

Numerator and denominator orders, specified as positive integer scalars. If the length of `h` is less than max(`bord`,`aord`), the function pads the impulse response with zeros.

- If you want an all-pole transfer function, specify `bord` as `0`.
- If you want an all-zero transfer function, specify `aord` as `0`.

Data Types: `single` | `double`

Output Arguments

b, a — Transfer function coefficients

vectors

Transfer function coefficients, returned as vectors. `b` has length `bord` + 1 and `a` has length `aord` + 1.

More About

Transfer Function

The transfer function is the Z-transform of the impulse response $h[n]$:

$$H(z) = \sum_{n=-\infty}^{\infty} h(n)\, z^{-n}.$$

A rational transfer function is a ratio of polynomials in z^{-1}. This equation describes a causal rational transfer function of numerator order q and denominator order p:

$$H(z) = \frac{B(z)}{A(z)} = \frac{\sum_{k=0}^{q} b(k)\, z^{-k}}{1 + \sum_{l=1}^{p} a(l)\, z^{-l}},$$

where $a[0] = 1$.

References

[1] Parks, Thomas W., and C. Sidney Burrus. *Digital Filter Design*. New York, NY, USA: Wiley-Interscience, 1987.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`designfilt` | `impz` | `levinson` | `lpc`

Topics

“Parametric Modeling”

Introduced before R2006a

pspectrum

Analyze signals in the frequency and time-frequency domains

Syntax

```
p = pspectrum(x)
p = pspectrum(x,fs)
p = pspectrum(x,t)

p = pspectrum( ___ ,type)

p = pspectrum( ___ ,Name,Value)

[p,f] = pspectrum( ___ )

[p,f,t] = pspectrum( ___ ,'spectrogram')
[p,f,pwr] = pspectrum( ___ ,'persistence')

pspectrum( ___ )
```

Description

`p = pspectrum(x)` returns the power spectrum of `x`.

- If `x` is a vector or a timetable with a vector of data, then it is treated as a single channel.
- If `x` is a matrix, a timetable with a matrix variable, or a timetable with multiple vector variables, then the spectrum is computed independently for each channel and stored in a separate column of `p`.

`p = pspectrum(x,fs)` returns the power spectrum of a vector or matrix signal sampled at a rate `fs`.

`p = pspectrum(x,t)` returns the power spectrum of a vector or matrix signal sampled at the time instants specified in `t`.

`p = pspectrum( ___ ,type)` specifies the kind of spectral analysis performed by the function. Specify `type` as `'power'`, `'spectrogram'`, or `'persistence'`. This syntax can include any combination of input arguments from previous syntaxes.

`p = pspectrum( ___ ,Name,Value)` specifies additional options using name-value pair arguments. Options include the frequency resolution bandwidth and the percent overlap between adjoining segments.

`[p,f] = pspectrum( ___ )` returns the frequencies corresponding to the spectral estimates contained in `p`.

`[p,f,t] = pspectrum( ___ ,'spectrogram')` also returns a vector of time instants corresponding to the centers of the windowed segments used to compute short-time power spectrum estimates.

`[p,f,pwr] = pspectrum( ___ ,'persistence')` also returns a vector of power values corresponding to the estimates contained in a persistence spectrum.

`pspectrum( ___ )` with no output arguments plots the spectral estimate in the current figure window. For the plot, the function converts `p` to dB using $10\log_{10}(p)$.

Examples

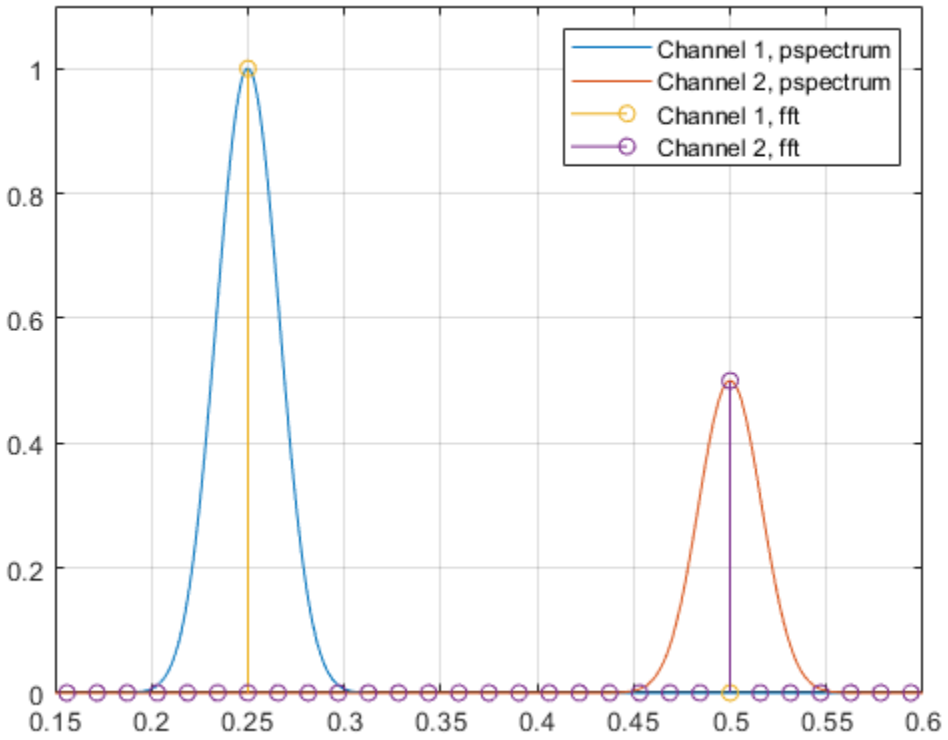
Power Spectra of Sinusoids

Generate 128 samples of a two-channel complex sinusoid.

- The first channel has unit amplitude and a normalized sinusoid frequency of $\pi/4$ rad/sample
- The second channel has an amplitude of $1/\sqrt{2}$ and a normalized frequency of $\pi/2$ rad/sample.

Compute the power spectrum of each channel and plot its absolute value. Zoom in on the frequency range from 0.15π rad/sample to 0.6π rad/sample. `pspectrum` scales the spectrum so that, if the frequency content of a signal falls exactly within a bin, its amplitude in that bin is the true average power of the signal. For a complex exponential, the average power is the square of the amplitude. Verify by computing the discrete Fourier transform of the signal. For more details, see “Measure Power of Deterministic Periodic Signals”.

```
N = 128;
x = [1 1/sqrt(2)].*exp(1j*pi./[4;2]*(0:N-1)).';

[p,f] = pspectrum(x);

plot(f/pi,abs(p))
hold on
stem(0:2/N:2-1/N,abs(fft(x)/N).^2)
hold off
axis([0.15 0.6 0 1.1])
legend("Channel 1, pspectrum","Channel 2, pspectrum", ...
    "Channel 1, fft","Channel 2, fft")
grid
```

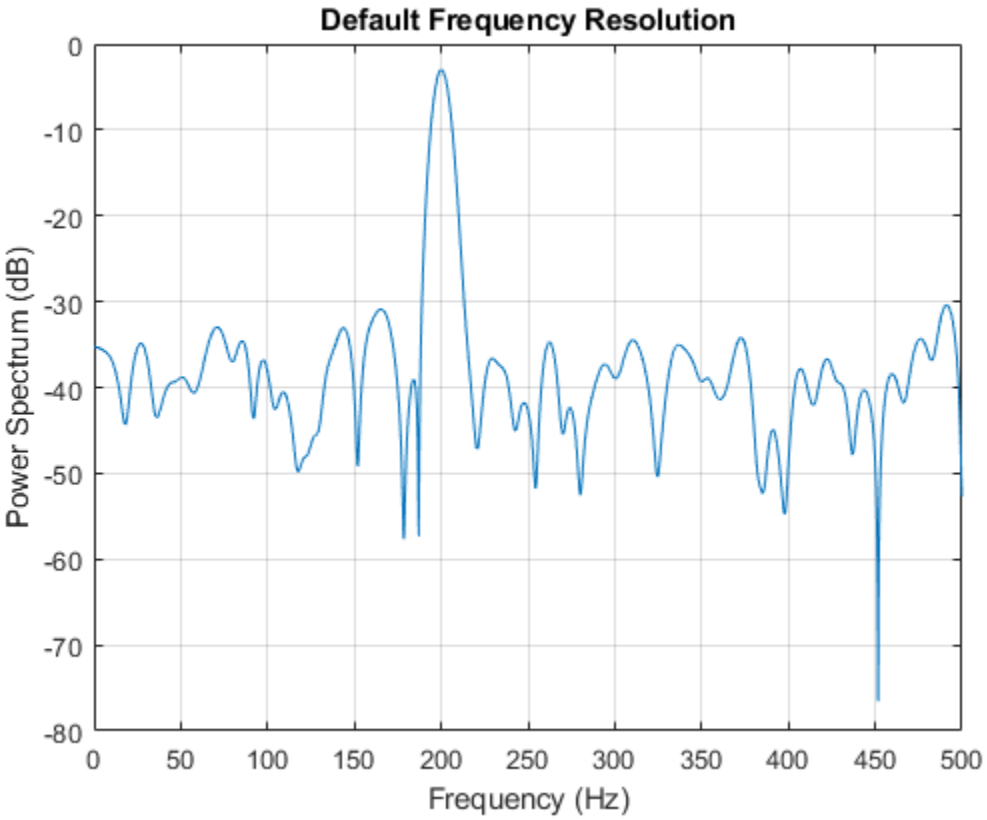


Generate a sinusoidal signal sampled at 1 kHz for 296 milliseconds and embedded in white Gaussian noise. Specify a sinusoid frequency of 200 Hz and a noise variance of 0.1^2. Store the signal and its time information in a MATLAB® timetable.

```
Fs = 1000;
t = (0:1/Fs:0.296)';
x = cos(2*pi*t*200)+0.1*randn(size(t));
xTable = timetable(seconds(t),x);
```

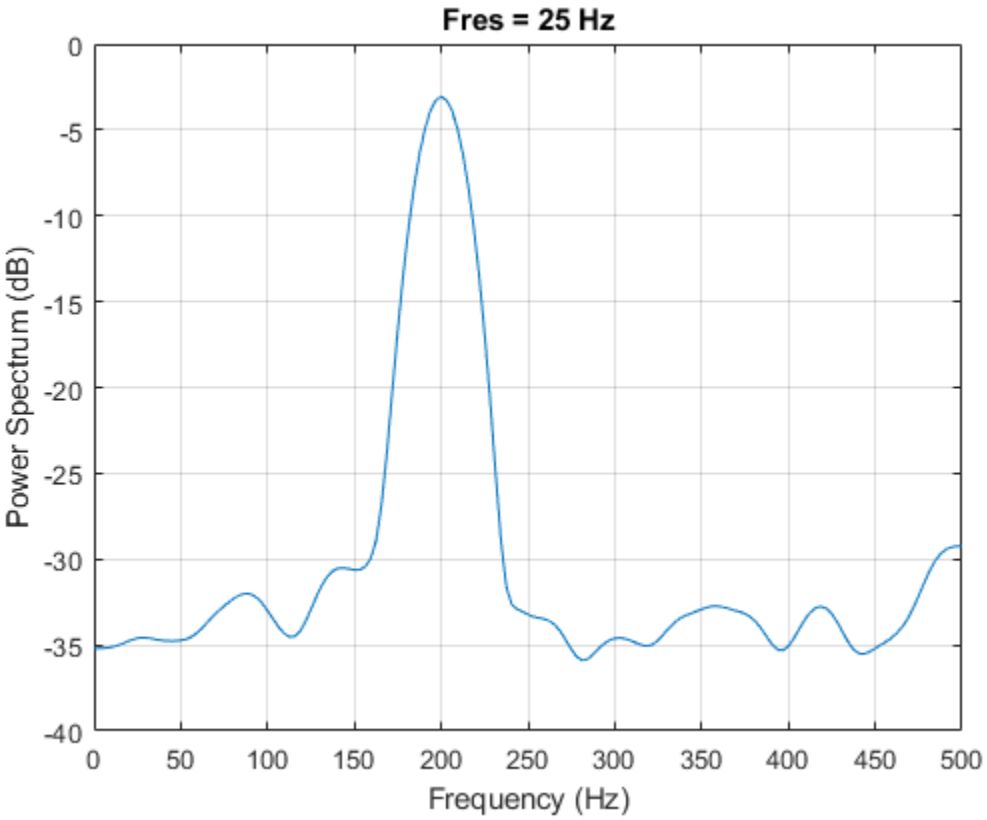
Compute the power spectrum of the signal. Express the spectrum in decibels and plot it.

```
[pxx,f] = pspectrum(xTable);

plot(f,pow2db(pxx))
grid on
xlabel('Frequency (Hz)')
ylabel('Power Spectrum (dB)')
title('Default Frequency Resolution')
```



Recompute the power spectrum of the sinusoid, but now use a coarser frequency resolution of 25 Hz. Plot the spectrum using the `pspectrum` function with no output arguments.

```
pspectrum(xTable,'FrequencyResolution',25)
```



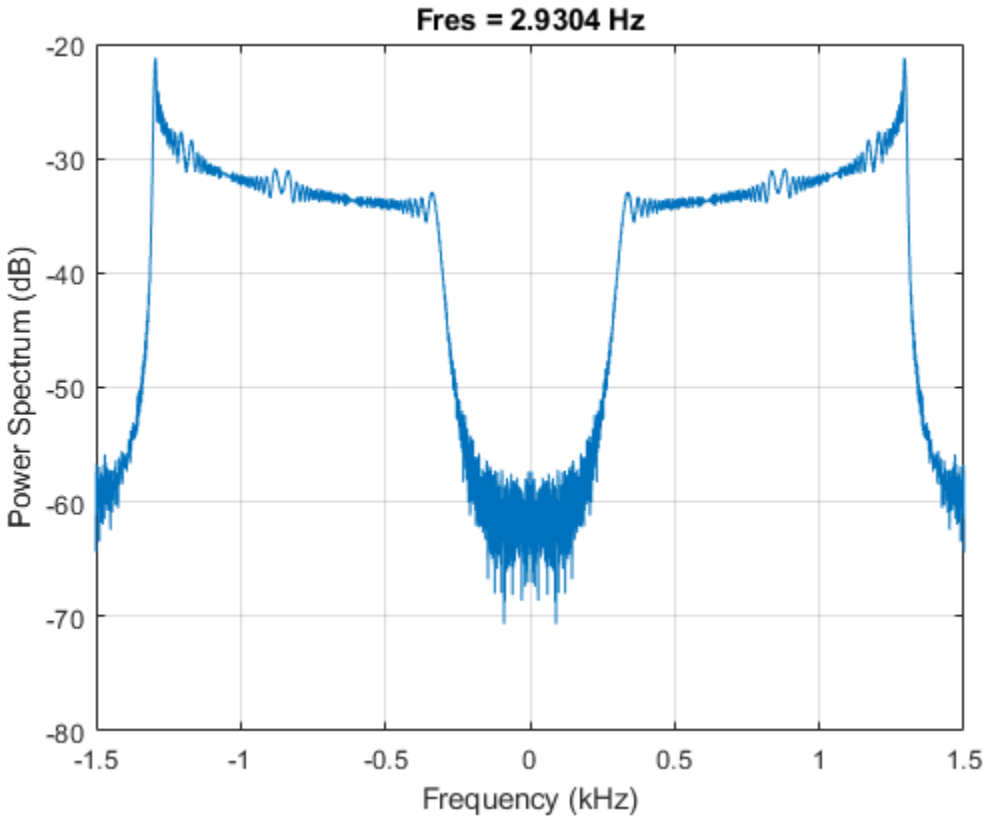
Two-Sided Spectra

Generate a signal sampled at 3 kHz for 1 second. The signal is a convex quadratic chirp whose frequency increases from 300 Hz to 1300 Hz during the measurement. The chirp is embedded in white Gaussian noise.

```
fs = 3000;
t = 0:1/fs:1-1/fs;

x1 = chirp(t,300,t(end),1300,'quadratic',0,'convex') + ...
    randn(size(t))/100;
```

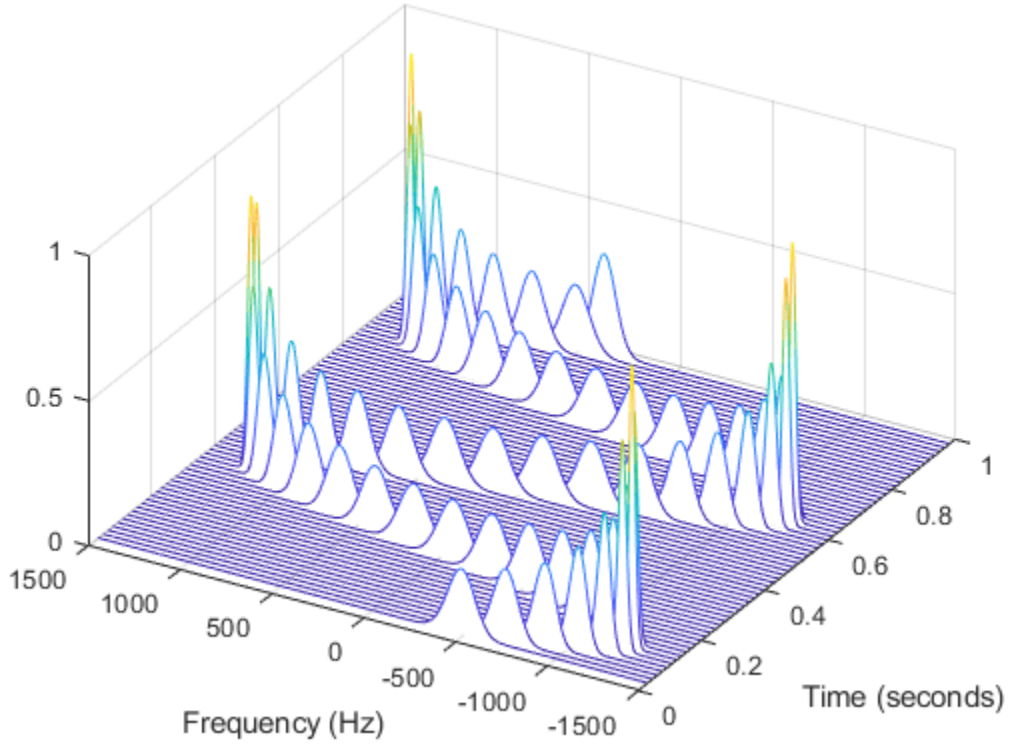
Compute and plot the two-sided power spectrum of the signal using a rectangular window. For real signals, `pspectrum` plots a one-sided spectrum by default. To plot a two-sided spectrum, set `TwoSided` to true.

```
pspectrum(x1,fs,'Leakage',1,'TwoSided',true)
```



Generate a complex-valued signal with the same duration and sample rate. The signal is a chirp with sinusoidally varying frequency content and embedded in white noise. Compute the spectrogram of the signal and display it as a waterfall plot. For complex-valued signals, the spectrogram is two-sided by default.

```
x2 = exp(2j*pi*100*cos(2*pi*2*t)) + randn(size(t))/100;

[p,f,t] = pspectrum(x2,fs,'spectrogram');

waterfall(f,t,p')
xlabel('Frequency (Hz)')
ylabel('Time (seconds)')
wtf = gca;
wtf.XDir = 'reverse';
view([30 45])
```



Window Leakage and Tone Resolution

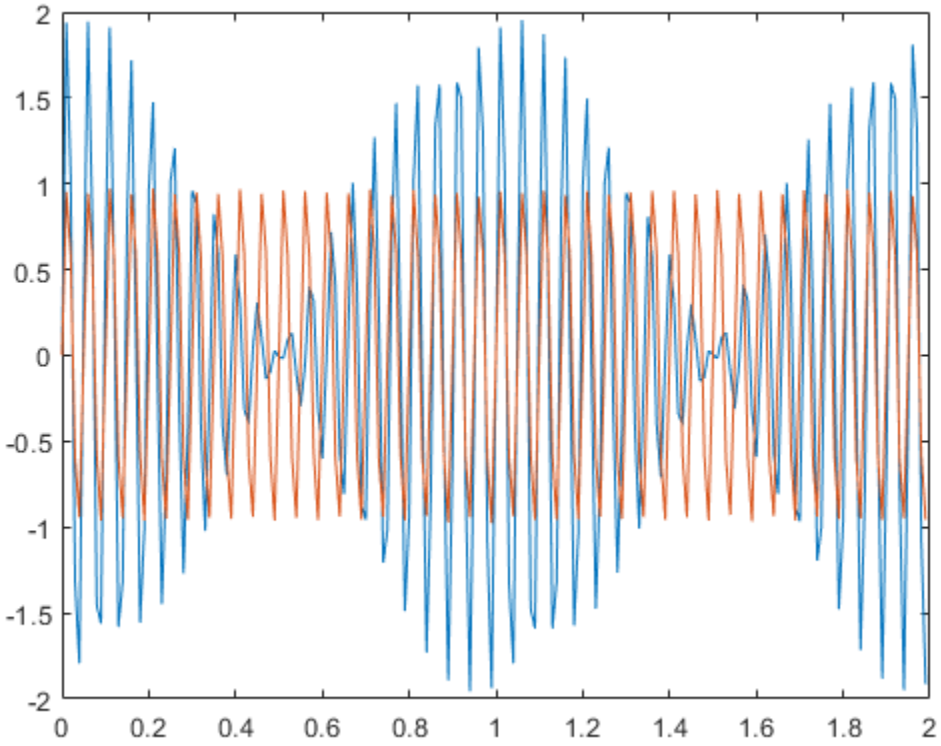
Generate a two-channel signal sampled at 100 Hz for 2 seconds.

1. The first channel consists of a 20 Hz tone and a 21 Hz tone. Both tones have unit amplitude.
2. The second channel also has two tones. One tone has unit amplitude and a frequency of 20 Hz. The other tone has an amplitude of 1/100 and a frequency of 30 Hz.

```
fs = 100;
t = (0:1/fs:2-1/fs)';

x = sin(2*pi*[20 20].*t) + [1 1/100].*sin(2*pi*[21 30].*t);
```

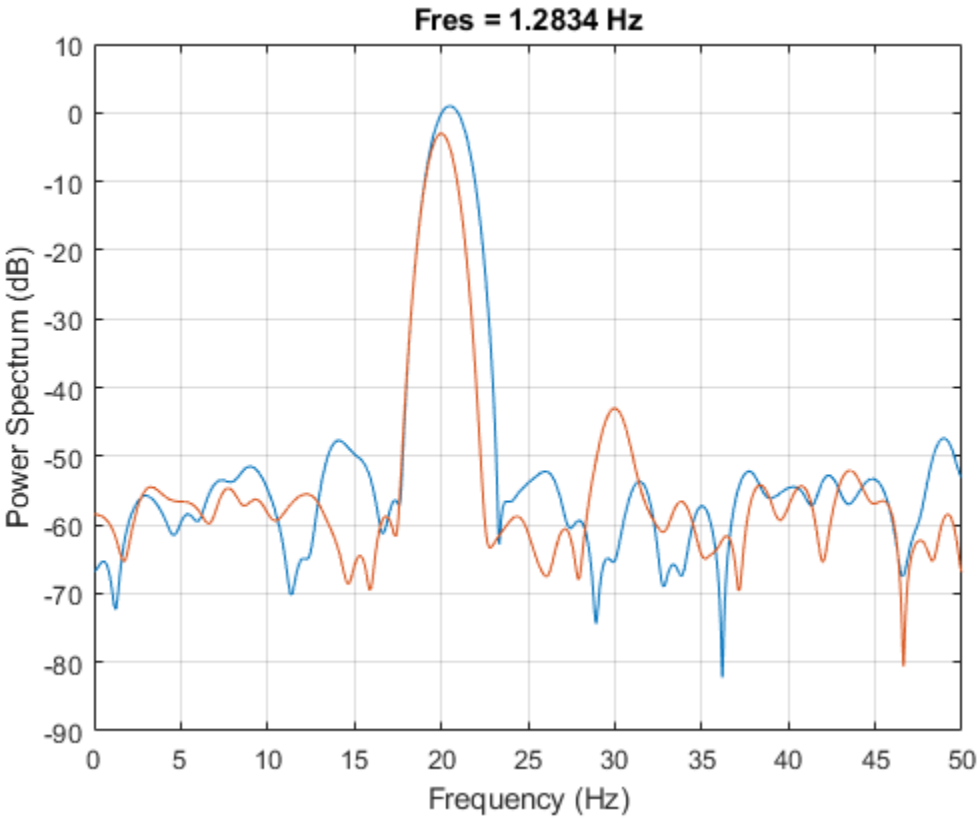
Embed the signal in white noise. Specify a signal-to-noise ratio of 40 dB. Plot the signals.

```
x = x + randn(size(x)).*std(x)/db2mag(40);

plot(t,x)
```



Compute the spectra of the two channels and display them.

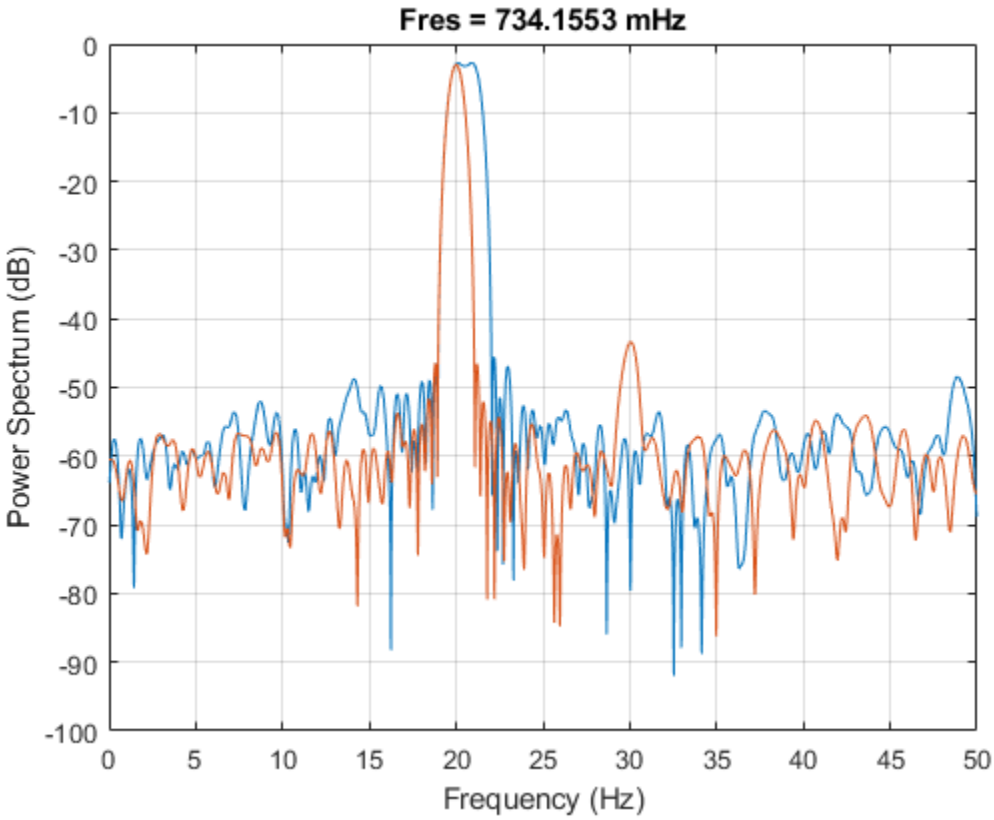
```
pspectrum(x,t)
```



The default value for the spectral leakage, 0.5, corresponds to a resolution bandwidth of about 1.29 Hz. The two tones in the first channel are not resolved. The 30 Hz tone in the second channel is visible, despite being much weaker than the other one.

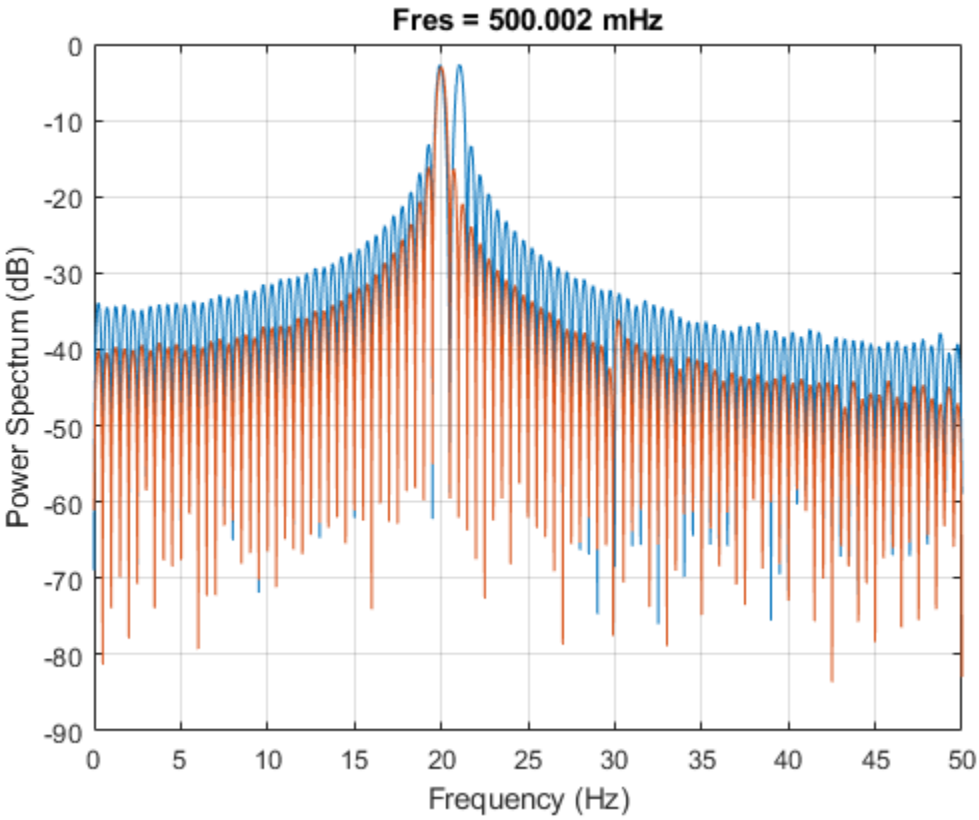
Increase the leakage to 0.85, equivalent to a resolution of about 0.74 Hz. The weak tone in the second channel is clearly visible.

```
pspectrum(x,t,'Leakage',0.85)
```



Increase the leakage to the maximum value. The resolution bandwidth is approximately 0.5 Hz. The two tones in the first channel are resolved. The weak tone in the second channel is masked by the large window sidelobes.

```
pspectrum(x,t,'Leakage',1)
```



Persistence Spectrum of Transient Signal

Visualize an interference narrowband signal embedded within a broadband signal.

Generate a chirp sampled at 1 kHz for 500 seconds. The frequency of the chirp increases from 180 Hz to 220 Hz during the measurement.

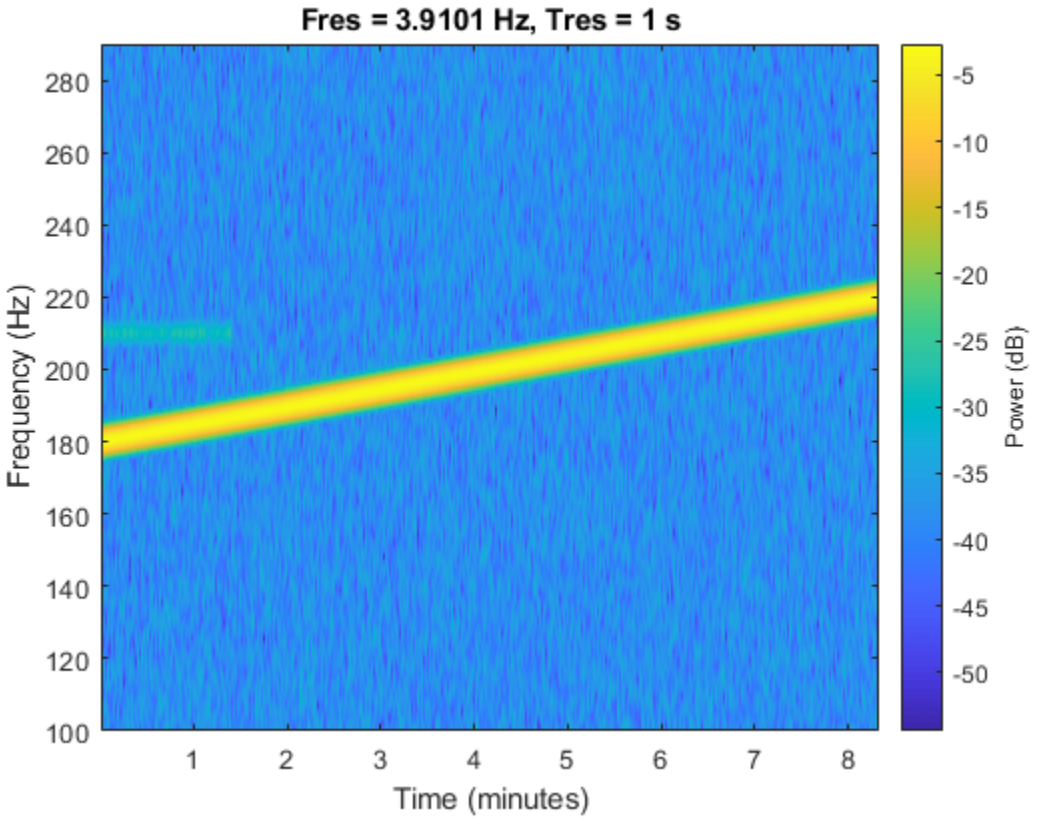
```
fs = 1000;
t = (0:1/fs:500)';

x = chirp(t,180,t(end),220) + 0.15*randn(size(t));
```

The signal also contains a 210 Hz sinusoid. The sinusoid has an amplitude of 0.05 and is present only for 1/6 of the total signal duration.

```
idx = floor(length(x)/6);
x(1:idx) = x(1:idx) + 0.05*cos(2*pi*t(1:idx)*210);
```

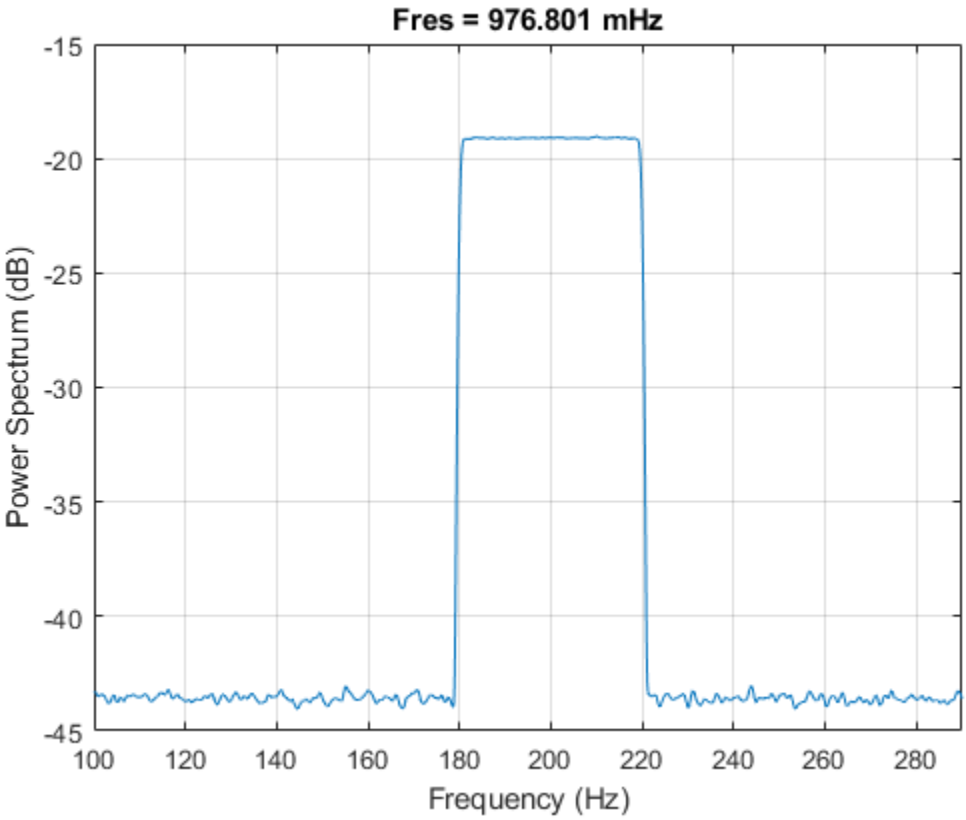
Compute the spectrogram of the signal. Restrict the frequency range from 100 Hz to 290 Hz. Specify a time resolution of 1 second. Both signal components are visible.

```
pspectrum(x,fs,'spectrogram', ...
    'FrequencyLimits',[100 290],'TimeResolution',1)
```



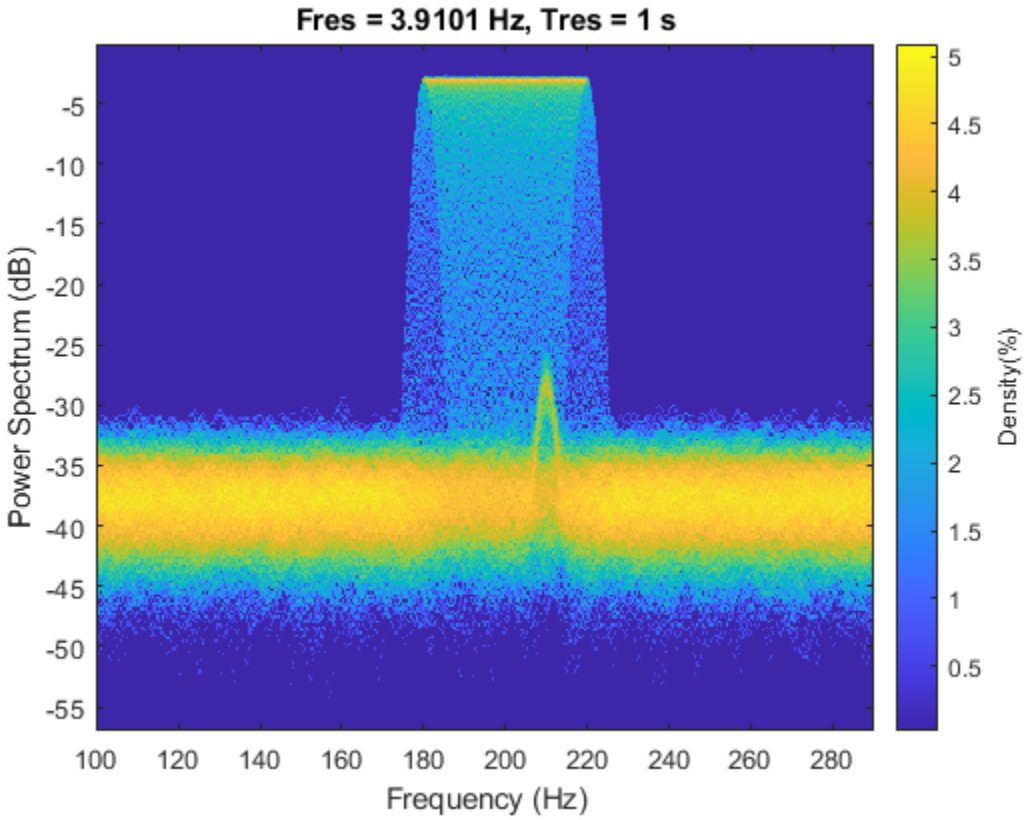
Compute the power spectrum of the signal. The weak sinusoid is obscured by the chirp.

```
pspectrum(x,fs,'FrequencyLimits',[100 290])
```



Compute the persistence spectrum of the signal. Now both signal components are clearly visible.

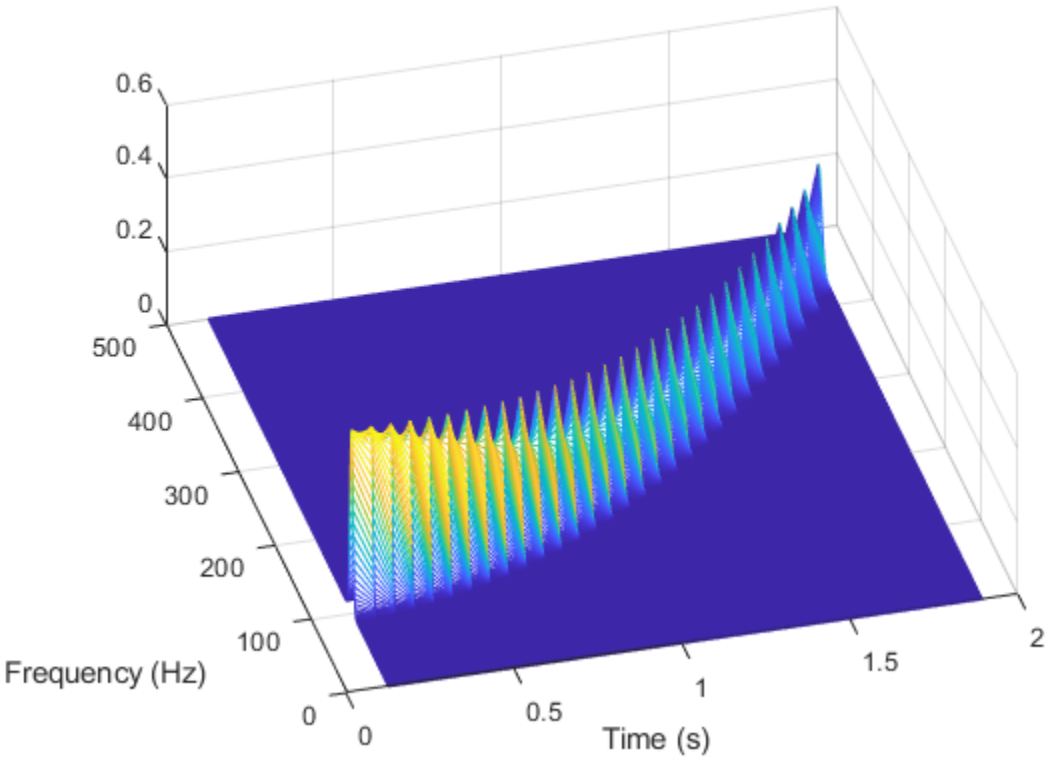
```
pspectrum(x,fs,'persistence', ...
    'FrequencyLimits',[100 290],'TimeResolution',1)
```



Spectrogram and Reassigned Spectrogram of Chirp

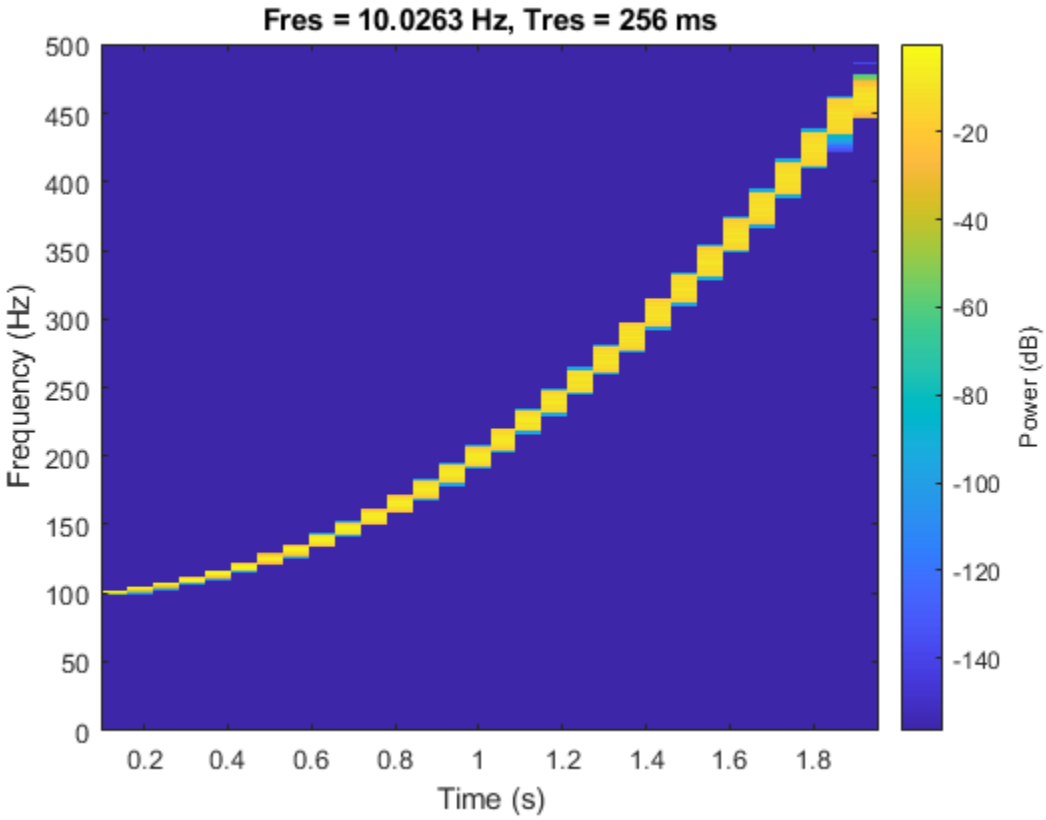
Generate a quadratic chirp sampled at 1 kHz for 2 seconds. The chirp has an initial frequency of 100 Hz that increases to 200 Hz at $t = 1$ second. Compute the spectrogram using the default settings of the `pspectrum` function.

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

[sp,fp,tp] = pspectrum(y,fs,'spectrogram');

mesh(tp,fp,sp)
view(-15,60)
xlabel('Time (s)')
ylabel('Frequency (Hz)')
```



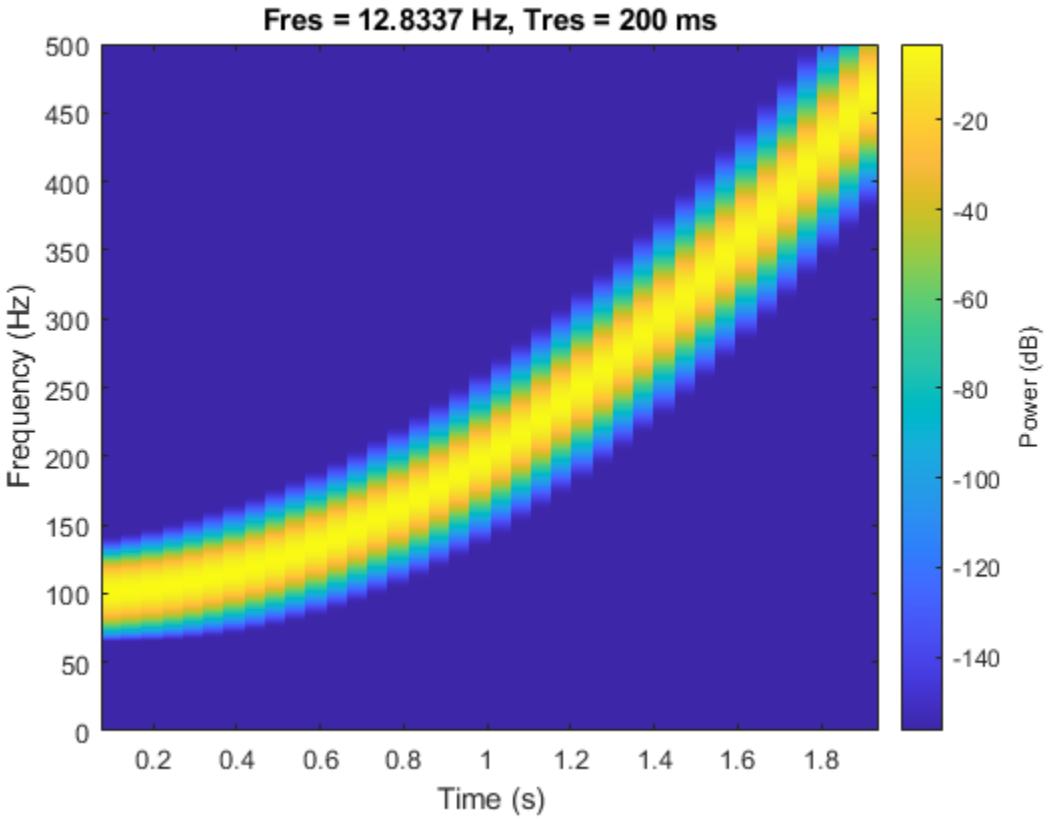
Compute the reassigned spectrogram. Specify a frequency resolution of 10 Hz. Visualize the result using the `pspectrum` function with no output arguments.

```
pspectrum(y,fs,'spectrogram','FrequencyResolution',10,'Reassign',true)
```



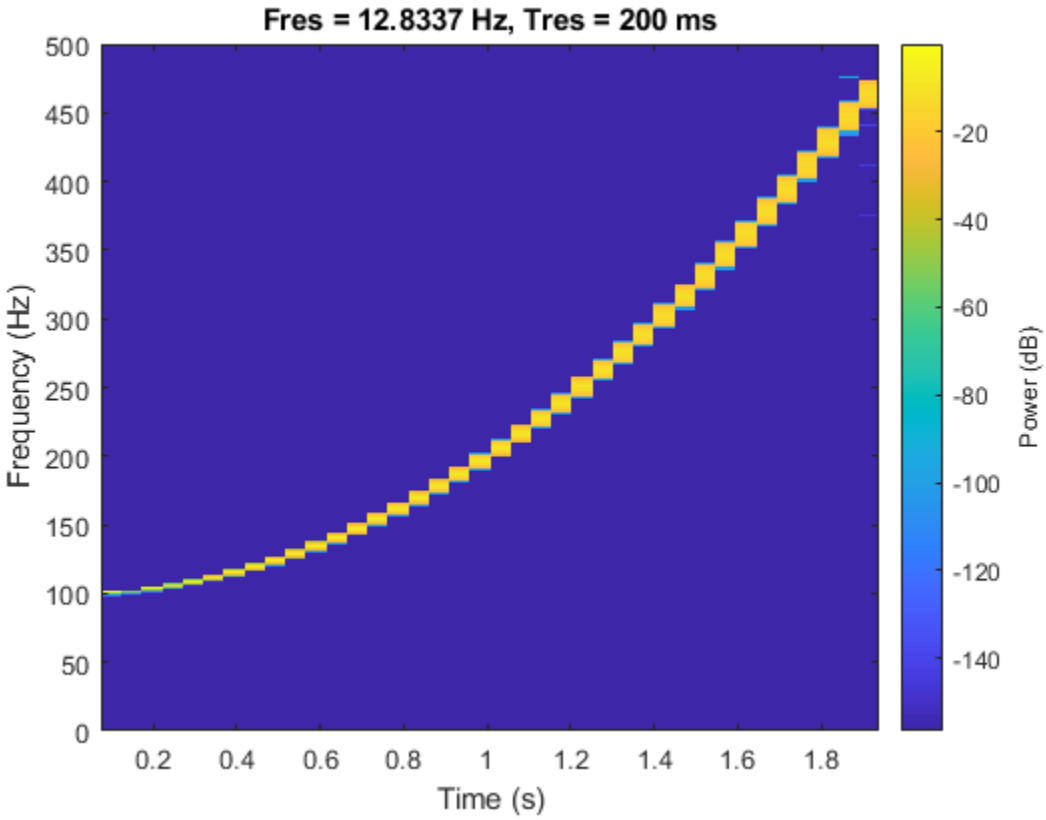
Recompute the spectrogram using a time resolution of 0.2 second.

```
pspectrum(y,fs,'spectrogram','TimeResolution',0.2)
```



Compute the reassigned spectrogram using the same time resolution.

```
pspectrum(y,fs,'spectrogram','TimeResolution',0.2,'Reassign',true)
```



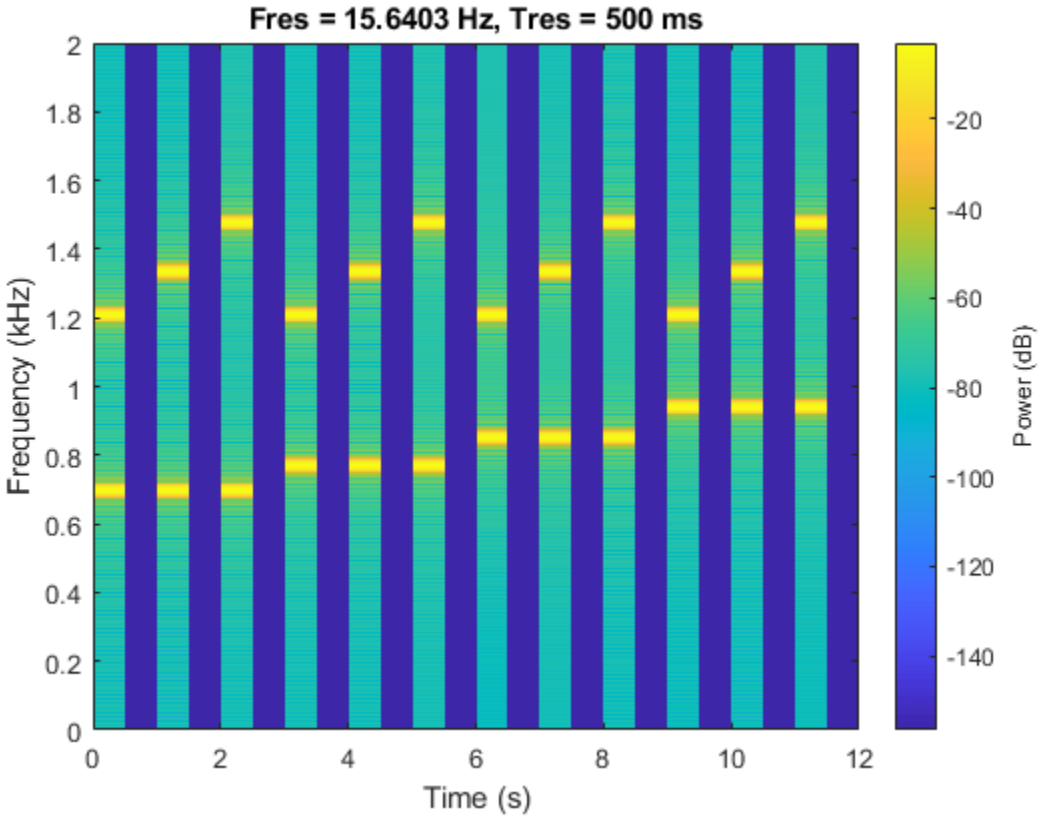
Spectrogram of Dial Tone Signal

Create a signal, sampled at 4 kHz, that resembles pressing all the keys of a digital telephone. Save the signal as a MATLAB® timetable.

```
fs = 4e3;
t = 0:1/fs:0.5-1/fs;

ver = [697 770 852 941];
hor = [1209 1336 1477];

tones = [];

for k = 1:length(ver)
    for l = 1:length(hor)
        tone = sum(sin(2*pi*[ver(k);hor(l)].*t))';
        tones = [tones;tone;zeros(size(tone))];
    end
end

% To hear, type soundsc(tones,fs)

S = timetable(seconds(0:length(tones)-1)'/fs,tones);
```

Compute the spectrogram of the signal. Specify a time resolution of 0.5 second and zero overlap between adjoining segments. Specify the leakage as 0.85, which is approximately equivalent to windowing the data with a Hann window.

```
pspectrum(S,'spectrogram', ...
    'TimeResolution',0.5,'OverlapPercent',0,'Leakage',0.85)
```



The spectrogram shows that each key is pressed for half a second, with half-second silent pauses between keys. The first tone has a frequency content concentrated around 697 Hz and 1209 Hz, corresponding to the digit `'1'` in the DTMF standard.

Input Arguments

x — Input signal

vector | matrix | timetable

Input signal, specified as a vector, a matrix, or a MATLAB `timetable`.

- If `x` is a timetable, then it must contain increasing finite row times.

  Note If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

- If `x` is a timetable representing a multichannel signal, then it must have either a single variable containing a matrix or multiple variables consisting of vectors.

If `x` is nonuniformly sampled, then `pspectrum` interpolates the signal to a uniform grid to compute spectral estimates. The function uses linear interpolation and assumes a sample time equal to the median of the differences between adjacent time points. For a nonuniformly sampled signal to be supported, the median time interval and the mean time interval must obey

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

Example: `cos(pi./[4;2]*(0:159))'+randn(160,2)` is a two-channel signal consisting of sinusoids embedded in white noise.

Example: `timetable(seconds(0:4)',rand(5,2))` specifies a two-channel random variable sampled at 1 Hz for 4 seconds.

Example: `timetable(seconds(0:4)',rand(5,1),rand(5,1))` specifies a two-channel random variable sampled at 1 Hz for 4 seconds.

Data Types: `single` | `double`
Complex Number Support: Yes

`fs` — Sample rate

2π (default) | positive numeric scalar

Sample rate, specified as a positive numeric scalar.

`t` — Time values

vector | `datetime` array | `duration` array | `duration` scalar

Time values, specified as a vector, a `datetime` or `duration` array, or a `duration` scalar representing the time interval between samples.

Example: `seconds(0:1/100:1)` is a `duration` array representing 1 second of sampling at 100 Hz.

Example: `seconds(1)` is a `duration` scalar representing a 1-second time difference between consecutive signal samples.

`type` — Type of spectrum to compute

`'power'` (default) | `'spectrogram'` | `'persistence'`

Type of spectrum to compute, specified as `'power'`, `'spectrogram'`, or `'persistence'`:

- `'power'` — Compute the power spectrum of the input. Use this option to analyze the frequency content of a stationary signal. For more information, “Spectrum Computation” on page 1-1761.
- `'spectrogram'` — Compute the spectrogram of the input. Use this option to analyze how the frequency content of a signal changes over time. For more information, see “Spectrogram Computation” on page 1-1763.
- `'persistence'` — Compute the persistence power spectrum of the input. Use this option to visualize the fraction of time that a particular frequency component is present in a signal. For more information, see “Persistence Spectrum Computation” on page 1-1765.

Note The `'spectrogram'` and `'persistence'` options do not support multichannel input.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Leakage',1,'Reassign',true,'MinThreshold',-35` windows the data using a rectangular window, computes a reassigned spectrum estimate, and sets all values smaller than –35 dB to zero.

`FrequencyLimits` — Frequency band limits

`[0 fs/2]` (default) | two-element numeric vector

Frequency band limits, specified as the comma-separated pair consisting of `'FrequencyLimits'` and a two-element numeric vector:

- If the input contains time information, then the frequency band is expressed in Hz.
- If the input does not contain time information, then the frequency band is expressed in normalized units of rad/sample.

By default, `pspectrum` computes the spectrum over the whole Nyquist range:

- If the specified frequency band contains a region that falls outside the Nyquist range, then `pspectrum` truncates the frequency band.
- If the specified frequency band lies completely outside of the Nyquist range, then `pspectrum` throws an error.

See “Spectrum Computation” on page 1-1761 for more information about the Nyquist range.

If `x` is nonuniformly sampled, then `pspectrum` linearly interpolates the signal to a uniform grid and defines an effective sample rate equal to the inverse of the median of the differences between adjacent time points. Express `'FrequencyLimits'` in terms of the effective sample rate.

Example: `[0.2*pi 0.7*pi]` computes the spectrum of a signal with no time information from 0.2π to 0.7π rad/sample.

`FrequencyResolution` — Frequency resolution bandwidth

real numeric scalar

Frequency resolution bandwidth, specified as the comma-separated pair consisting of `'FrequencyResolution'` and a real numeric scalar, expressed in Hz if the input contains time information, or in normalized units of rad/sample if not. This argument cannot be specified simultaneously with `'TimeResolution'`. The default value of this argument depends on the size of the input data. See “Spectrogram Computation” on page 1-1763 for details.

Example: `pi/100` computes the spectrum of a signal with no time information with a frequency resolution of $\pi/100$ rad/sample.

`Leakage` — Spectral leakage

`0.5` (default) | real numeric scalar between 0 and 1

Spectral leakage, specified as the comma-separated pair consisting of `'Leakage'` and a real numeric scalar between 0 and 1. `'Leakage'` controls the Kaiser window sidelobe attenuation relative to the mainlobe width, compromising between improving resolution and decreasing leakage:

- A large leakage value resolves closely spaced tones, but masks nearby weak tones.
- A small leakage value finds small tones in the vicinity of larger tones, but smears close frequencies together.

Example: `'Leakage',0` reduces leakage to a minimum at the expense of spectral resolution.

Example: `'Leakage',0.85` approximates windowing the data with a Hann window.

Example: `'Leakage',1` is equivalent to windowing the data with a rectangular window, maximizing leakage but improving spectral resolution.

`MinThreshold` — Lower bound for nonzero values
`-Inf` (default) | real scalar

Lower bound for nonzero values, specified as the comma-separated pair consisting of `'MinThreshold'` and a real scalar. `pspectrum` implements `'MinThreshold'` differently based on the value of the `type` argument:

- `'power'` or `'spectrogram'` — `pspectrum` sets those elements of `p` such that $10 \log_{10}(p) \le$ `'MinThreshold'` to zero. Specify `'MinThreshold'` in decibels.
- `'persistence'` — `pspectrum` sets those elements of `p` smaller than `'MinThreshold'` to zero. Specify `'MinThreshold'` between 0 and 100%.

`NumPowerBins` — Number of power bins for persistence spectrum
`256` (default) | integer between 20 and 1024

Number of power bins for persistence spectrum, specified as the comma-separated pair consisting of `'NumPowerBins'` and an integer between 20 and 1024.

`OverlapPercent` — Overlap between adjoining segments
real scalar in the interval [0, 100)

Overlap between adjoining segments for spectrogram or persistence spectrum, specified as the comma-separated pair consisting of `'OverlapPercent'` and a real scalar in the interval [0, 100). The default value of this argument depends on the spectral window. See “Spectrogram Computation” on page 1-1763 for details.

`Reassign` — Reassignment option
`false` (default) | `true`

Reassignment option, specified as the comma-separated pair consisting of `'Reassign'` and a logical value. If this option is set to `true`, then `pspectrum` sharpens the localization of spectral estimates by performing time and frequency reassignment. The reassignment technique produces periodograms and spectrograms that are easier to read and interpret. This technique reassigns each spectral estimate to the center of energy of its bin instead of the bin's geometric center. The technique provides exact localization for chirps and impulses.

`TimeResolution` — Time resolution of spectrogram or persistence spectrum
real scalar

Time resolution of spectrogram or persistence spectrum, specified as the comma-separated pair consisting of `'TimeResolution'` and a real scalar, expressed in seconds if the input contains time information, or as an integer number of samples if not. This argument controls the duration of the segments used to compute the short-time power spectra that form spectrogram or persistence spectrum estimates. `'TimeResolution'` cannot be specified simultaneously with

`'FrequencyResolution'`. The default value of this argument depends on the size of the input data and, if it was specified, the frequency resolution. See “Spectrogram Computation” on page 1-1763 for details.

`TwoSided` — Two-sided spectral estimate
false | true

Two-sided spectral estimate, specified as the comma-separated pair consisting of `'TwoSided'` and a logical value.

- If this option is `true`, the function computes centered, two-sided spectrum estimates over $[-\pi, \pi]$. If the input has time information, the estimates are computed over $[-f_s/2, f_s/2]$, where f_s is the effective sample rate.
- If this option is `false`, the function computes one-sided spectrum estimates over the Nyquist range $[0, \pi]$. If the input has time information, the estimates are computed over $[0, f_s/2]$, where f_s is the effective sample rate. To conserve the total power, the function multiplies the power by 2 at all frequencies except 0 and the Nyquist frequency. This option is valid only for real signals.

If not specified, `'TwoSided'` defaults to `false` for real input signals and to `true` for complex input signals.

Output Arguments

`p` — Spectrum
vector | matrix

Spectrum, returned as a vector or a matrix. The type and size of the spectrum depends on the value of the `type` argument:

- `'power'` — `p` contains the power spectrum estimate of each channel of `x`. In this case, `p` is of size $N_f \times N_{ch}$, where N_f is the length of `f` and N_{ch} is the number of channels of `x`. `pspectrum` scales the spectrum so that, if the frequency content of a signal falls exactly within a bin, its amplitude in that bin is the true average power of the signal. For example, the average power of a sinusoid is one-half the square of the sinusoid amplitude. For more details, see “Measure Power of Deterministic Periodic Signals”.
- `'spectrogram'` — `p` contains an estimate of the short-term, time-localized power spectrum of `x`. In this case, `p` is of size $N_f \times N_t$, where N_f is the length of `f` and N_t is the length of `t`.
- `'persistence'` — `p` contains, expressed as percentages, the probabilities that the signal has components of a given power level at a given time and frequency location. In this case, `p` is of size $N_{pwr} \times N_f$, where N_{pwr} is the length of `pwr` and N_f is the length of `f`.

`f` — Spectrum frequencies
vector

Spectrum frequencies, returned as a vector. If the input signal contains time information, then `f` contains frequencies expressed in Hz. If the input signal does not contain time information, then the frequencies are in normalized units of rad/sample.

`t` — Time values of spectrogram
vector | `datetime` array | `duration` array

Time values of spectrogram, returned as a vector of time values in seconds or a `duration` array. If the input does not have time information, then `t` contains sample numbers. `t` contains the time values

corresponding to the centers of the data segments used to compute short-time power spectrum estimates.

- If the input to `pspectrum` is a timetable, then `t` has the same format as the time values of the input timetable.
- If the input to `pspectrum` is a numeric vector sampled at a set of time instants specified by a numeric, `duration`, or `datetime` array, then `t` has the same type and format as the input time values.
- If the input to `pspectrum` is a numeric vector with a specified time difference between consecutive samples, then `t` is a `duration` array.

`pwr` — Power values of persistence spectrum
vector

Power values of persistence spectrum, returned as a vector.

More About

Spectrum Computation

To compute signal spectra, `pspectrum` finds a compromise between the spectral resolution achievable with the entire length of the signal and the performance limitations that result from computing large FFTs:

- If possible, the function computes a single modified periodogram of the whole signal using a Kaiser window.
- If it is not possible to compute a single modified periodogram in a reasonable amount of time, the function computes a Welch periodogram: It divides the signal into overlapping segments, windows each segment using a Kaiser window, and averages the periodograms of the segments.

Spectral Windowing

Any real-world signal is measurable only for a finite length of time. This fact introduces nonnegligible effects into Fourier analysis, which assumes that signals are either periodic or infinitely long. Spectral windowing, which assigns different weights to different signal samples, deals systematically with finite-size effects.

The simplest way to window a signal is to assume that it is identically zero outside of the measurement interval and that all samples are equally significant. This "rectangular window" has discontinuous jumps at both ends that result in spectral ringing. All other spectral windows taper at both ends to lessen this effect by assigning smaller weights to samples close to the signal edges.

The windowing process always involves a compromise between conflicting aims: improving resolution and decreasing leakage:

- Resolution is the ability to know precisely how the signal energy is distributed in the frequency space. A spectrum analyzer with ideal resolution can distinguish two different tones (pure sinusoids) present in the signal, no matter how close in frequency. Quantitatively, this ability relates to the mainlobe width of the transform of the window.
- Leakage is the fact that, in a finite signal, every frequency component projects energy content throughout the complete frequency span. The amount of leakage in a spectrum can be measured by the ability to detect a weak tone from noise in the presence of a neighboring strong tone. Quantitatively, this ability relates to the sidelobe level of the frequency transform of the window.

- The spectrum is normalized so that a pure tone within that bandwidth, if perfectly centered, has the correct amplitude.

The better the resolution, the higher the leakage, and vice versa. At one end of the range, a rectangular window has the narrowest possible mainlobe and the highest sidelobes. This window can resolve closely spaced tones if they have similar energy content, but it fails to find the weaker one if they do not. At the other end, a window with high sidelobe suppression has a wide mainlobe in which close frequencies are smeared together.

`pspectrum` uses Kaiser windows to carry out windowing. For Kaiser windows, the fraction of the signal energy captured by the mainlobe depends most importantly on an adjustable shape factor, β. `pspectrum` uses shape factors ranging from $\beta = 0$, which corresponds to a rectangular window, to $\beta = 40$, where a wide mainlobe captures essentially all the spectral energy representable in double precision. An intermediate value of $\beta \approx 6$ approximates a Hann window quite closely. To control β, use the `'Leakage'` name-value pair. If you set `'Leakage'` to ℓ, then ℓ and β are related by $\beta = 40(1 - \ell)$. See `kaiser` for more details.

Parameter and Algorithm Selection

To compute signal spectra, `pspectrum` initially determines the resolution bandwidth, which measures how close two tones can be and still be resolved. The resolution bandwidth has a theoretical value of

$$\mathrm{RBW}_{\mathrm{theory}} = \frac{\mathrm{ENBW}}{t_{\max} - t_{\min}}.$$

- $t_{\max} - t_{\min}$, the record length, is the time-domain duration of the selected signal region.
- ENBW is the equivalent noise bandwidth of the spectral window. See `enbw` for more details.

  Use the `'Leakage'` name-value pair to control the ENBW. The minimum value of the argument corresponds to a Kaiser window with $\beta = 40$. The maximum value corresponds to a Kaiser window with $\beta = 0$.

In practice, however, `pspectrum` might lower the resolution. Lowering the resolution makes it possible to compute the spectrum in a reasonable amount of time and to display it with a finite number of pixels. For these practical reasons, the lowest resolution bandwidth `pspectrum` can use is

$$\mathrm{RBW}_{\mathrm{performance}} = 4 \times \frac{f_{\mathrm{span}}}{4096 - 1},$$

where f_{span} is the width of the frequency band specified using `'FrequencyLimits'`. If `'FrequencyLimits'` is not specified, then `pspectrum` uses the sample rate as f_{span}. $\mathrm{RBW}_{\mathrm{performance}}$ cannot be adjusted.

To compute the spectrum of a signal, the function chooses the larger of the two values, called the target resolution bandwidth:

$$\mathrm{RBW} = \max(\mathrm{RBW}_{\mathrm{theory}}, \mathrm{RBW}_{\mathrm{performance}}).$$

- If the resolution bandwidth is $\mathrm{RBW}_{\mathrm{theory}}$, then `pspectrum` computes a single modified periodogram for the whole signal. The function uses a Kaiser window with shape factor controlled by the `'Leakage'` name-value pair. See `periodogram` for more details.
- If the resolution bandwidth is $\mathrm{RBW}_{\mathrm{performance}}$, then `pspectrum` computes a Welch periodogram for the signal. The function:

1. Divides the signals into overlapping segments.
2. Windows each segment separately using a Kaiser window with the specified shape factor.
3. Averages the periodograms of all the segments.

Welch's procedure is designed to reduce the variance of the spectrum estimate by averaging different "realizations" of the signals, given by the overlapping sections, and using the window to remove redundant data. See `pwelch` for more details.

- The length of each segment (or, equivalently, of the window) is computed using

$$\text{Segment length} = \frac{f_{\text{Nyquist}} \times \text{ENBW}}{\text{RBW}},$$

  where f_{Nyquist} is the Nyquist frequency. (If there is no aliasing, the Nyquist frequency is one-half the effective sample rate, defined as the inverse of the median of the differences between adjacent time points. The Nyquist range is $[0, f_{\text{Nyquist}}]$ for real signals and $[-f_{\text{Nyquist}}, f_{\text{Nyquist}}]$ for complex signals.)

- The stride length is found by adjusting an initial estimate,

$$\text{Stride length} \equiv \text{Segment length} - \text{Overlap} = \frac{\text{Segment length}}{2 \times \text{ENBW} - 1},$$

  so that the first window starts exactly on the first sample of the first segment and the last window ends exactly on the last sample of the last segment.

Spectrogram Computation

To compute the time-dependent spectrum of a nonstationary signal, `pspectrum` divides the signal into overlapping segments, windows each segment with a Kaiser window, computes the short-time Fourier transform, and then concatenates the transforms to form a matrix.

A nonstationary signal is a signal whose frequency content changes with time. The spectrogram of a nonstationary signal is an estimate of the time evolution of its frequency content. To construct the spectrogram of a nonstationary signal, `pspectrum` follows these steps:

1. Divide the signal into equal-length segments. The segments must be short enough that the frequency content of the signal does not change appreciably within a segment. The segments may or may not overlap.
2. Window each segment and compute its spectrum to get the short-time Fourier transform.
3. Use the segment spectra to construct the spectrogram:
   - If called with output arguments, concatenate the spectra to form a matrix.
   - If called with no output arguments, display the power of each spectrum in decibels segment by segment. Depict the magnitudes side-by-side as an image with magnitude-dependent colormap.

The function can compute the spectrogram only for single-channel signals.

Divide Signal into Segments

To construct a spectrogram, first divide the signal into possibly overlapping segments. With the `pspectrum` function, you can control the length of the segments and the amount of overlap between adjoining segments using the `'TimeResolution'` and `'OverlapPercent'` name-value pair

arguments. If you do not specify the length and overlap, the function chooses a length based on the entire length of the signal and an overlap percentage given by

$$\left(1-\frac{1}{2\times \mathrm{ENBW}-1}\right)\times 100,$$

where ENBW is the equivalent noise bandwidth of the spectral window. See `enbw` and “Spectrum Computation” on page 1-1761 for more information.

Specified Time Resolution

- If the signal does not have time information, specify the time resolution (segment length) in samples. The time resolution must be an integer greater than or equal to 1 and smaller than or equal to the signal length.

  If the signal has time information, specify the time resolution in seconds. The function converts the result into a number of samples and rounds it to the nearest integer that is less than or equal to the number but not smaller than 1. The time resolution must be smaller than or equal to the signal duration.
- Specify the overlap as a percentage of the segment length. The function converts the result into a number of samples and rounds it to the nearest integer that is less than or equal to the number.

Default Time Resolution

If you do not specify a time resolution, then `pspectrum` uses the length of the entire signal to choose the length of the segments. The function sets the time resolution as $\lceil N/d \rceil$ samples, where the $\lceil\rceil$ symbols denote the ceiling function, N is the length of the signal, and d is a divisor that depends on N:

Signal Length (*N*)	**Divisor (*d*)**	**Segment Length**
2 samples – 63 samples	2	1 sample – 32 samples
64 samples – 255 samples	8	8 samples – 32 samples
256 samples – 2047 samples	8	32 samples – 256 samples
2048 samples – 4095 samples	16	128 samples – 256 samples
4096 samples – 8191 samples	32	128 samples – 256 samples
8192 samples – 16383 samples	64	128 samples – 256 samples
16384 samples – *N* samples	128	128 samples – $\lceil N$ / 128$\rceil$ samples

You can still specify the overlap between adjoining segments. Specifying the overlap changes the number of segments. Segments that extend beyond the signal endpoint are zero-padded.

Consider the seven-sample signal `[s0 s1 s2 s3 s4 s5 s6]`. Because $\lceil 7/2 \rceil = \lceil 3.5 \rceil = 4$, the function divides the signal into two segments of length four when there is no overlap. The number of segments changes as the overlap increases.

Number of Overlapping Samples	**Resulting Segments**
0	`s0 s1 s2 s3` `s4 s5 s6 0`
1	`s0 s1 s2 s3` `s3 s4 s5 s6`

Number of Overlapping Samples	Resulting Segments
2	s0 s1 s2 s3 s2 s3 s4 s5 s4 s5 s6 0
3	s0 s1 s2 s3 s1 s2 s3 s4 s2 s3 s4 s5 s3 s4 s5 s6

`pspectrum` zero-pads the signal if the last segment extends beyond the signal endpoint. The function returns `t`, a vector of time instants corresponding to the centers of the segments.

Window the Segments and Compute Spectra

After `pspectrum` divides the signal into overlapping segments, the function windows each segment with a Kaiser window. The shape factor β of the window, and therefore the leakage, can be adjusted using the `'Leakage'` name-value pair. The function then computes the spectrum of each segment and concatenates the spectra to form the spectrogram matrix. To compute the segment spectra, `pspectrum` follows the procedure described in “Spectrum Computation” on page 1-1761, except that the lower limit of the resolution bandwidth is

$$\mathrm{RBW}_{\mathrm{performance}} = 4 \times \frac{f_{\mathrm{span}}}{1024 - 1}.$$

Display Spectrum Power

If called with no output arguments, the function displays the power of the short-time Fourier transform in decibels, using a color bar with the default MATLAB colormap. The color bar comprises the full power range of the spectrogram.

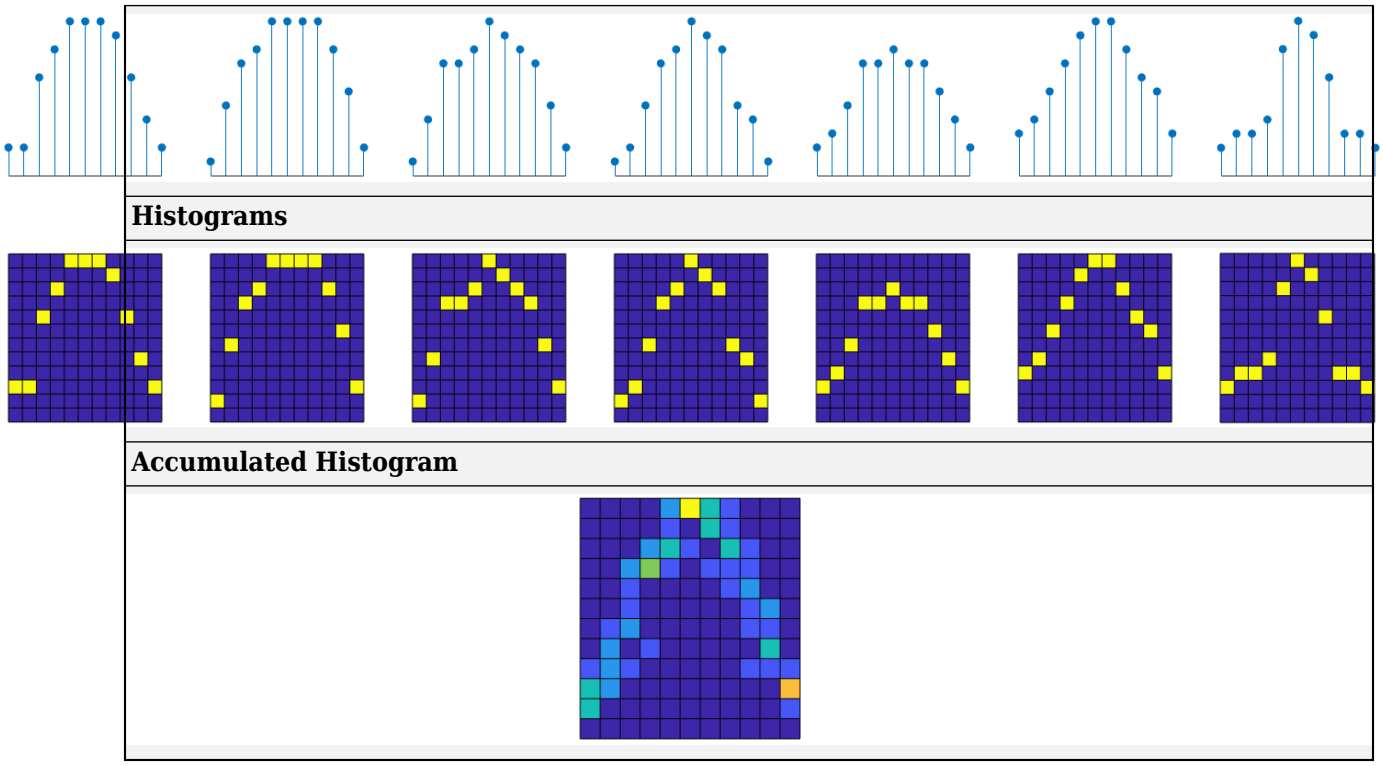
Persistence Spectrum Computation

The persistence spectrum is a time-frequency view that shows the percentage of the time that a given frequency is present in a signal. The persistence spectrum is a histogram in power-frequency space. The longer a particular frequency persists in a signal as the signal evolves, the higher its time percentage and thus the brighter or "hotter" its color in the display. Use the persistence spectrum to identify signals hidden in other signals.

To compute the persistence spectrum, `pspectrum` performs these steps:

1. Compute the spectrogram using the specified leakage, time resolution, and overlap. See “Spectrogram Computation” on page 1-1763 for more details.
2. Partition the power and frequency values into 2-D bins. (Use the `'NumPowerBins'` name-value pair to specify the number of power bins.)
3. For each time value, compute a bivariate histogram of the logarithm of the power spectrum. For every power-frequency bin where there is signal energy at that instant, increase the corresponding matrix element by 1. Sum the histograms for all the time values.
4. Plot the accumulated histogram against the power and the frequency, with the color proportional to the logarithm of the histogram counts expressed as normalized percentages. To represent zero values, use one-half of the smallest possible magnitude.

Power Spectra



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] 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. 15, June 1967, pp. 70–73.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Timetables are not supported for code generation.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Persistence spectrum is not supported.
- Reassigned spectrum or spectrogram is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Apps
Signal Analyzer

Functions
`periodogram` | `pwelch` | `spectrogram`

Topics
“Time-Frequency Gallery”

Introduced in R2017b

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 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`. To determine the transitions for each pulse, the `pulseperiod` function 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.

`p = pulseperiod(x,Fs)` specifies the sample rate `Fs`.

`P = pulseperiod(x,t)` specifies the sampling instants `t`.

`[p,initcross] = pulseperiod( ___ )` returns the mid-reference level instants `initcross` of the first transition of each pulse.

`[p,initcross,finalcross] = pulseperiod( ___ )` returns the mid-reference level instants `finalcross` of the final transition of each pulse. You can specify an input combination from any of the previous syntaxes.

`[p,initcross,finalcross,nextcross] = pulseperiod( ___ )` returns the mid-reference level instants `nextcross` of the 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 arguments.

`pulseperiod( ___ )` plots the signal and darkens every other identified pulse. The function marks the location of the mid crossings and their associated reference level. The function also plots the state levels and their associated lower and upper boundaries. You can adjust the boundaries using the `'Tolerance'` name-value argument.

Examples

Pulse Period of Bilevel Waveform

Compute the pulse period of a bilevel waveform with two positive-polarity transitions. The sample rate is 4 MHz.

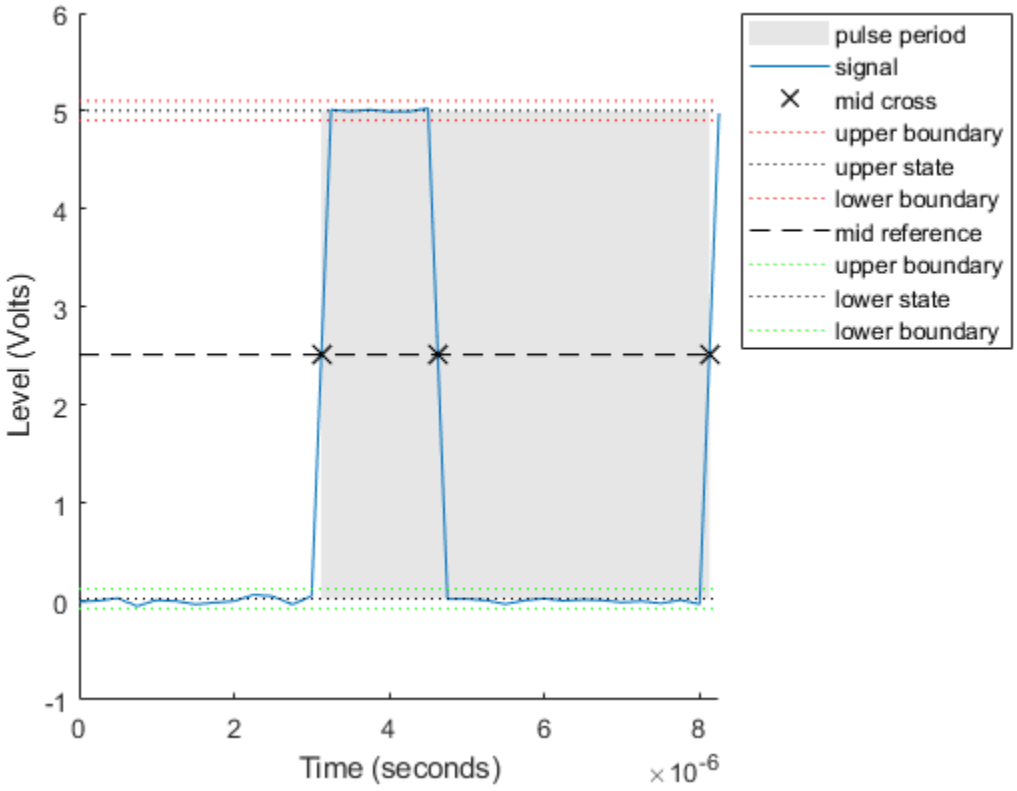
```
load('pulseex.mat','x','t')
```

```
p = pulseperiod(x,t)
```

```
p = 5.0030e-06
```

Annotate the pulse period on a plot of the waveform.

```
pulseperiod(x,t);
```



Mid-Reference Level Instants of Pulse Period

Determine the mid-reference level instants that define the pulse period for a bilevel waveform.

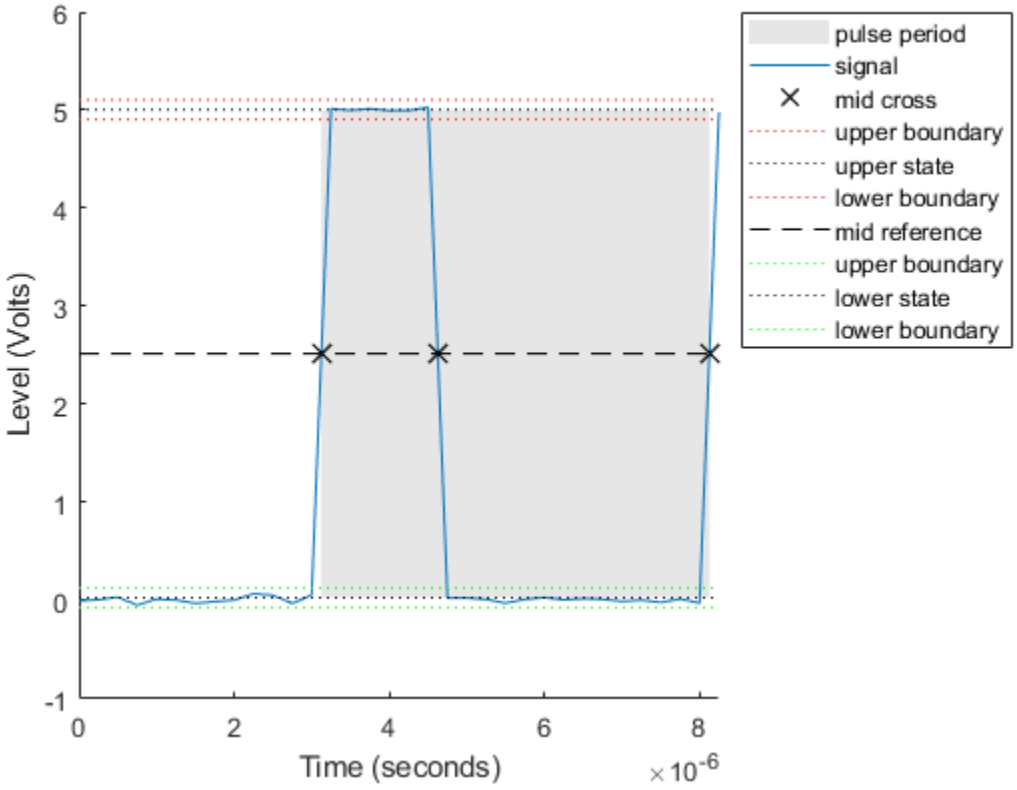
```
load('pulseex.mat','x','t')
[~,initcross,~,nextcross] = pulseperiod(x,t)
```

```
initcross = 3.1240e-06
```

```
nextcross = 8.1270e-06
```

Output the pulse period. Mark the mid-reference level instants on a plot of the data.

```
pulseperiod(x,t)
```




```
ans = 5.0030e-06
```

Input Arguments

x — Bilevel waveform

real-valued vector

Bilevel waveform, specified as a real-valued vector. If `x` does not contain at least two transitions, the function outputs an empty matrix. The first sample instant in `x` corresponds to `t=0`.

Fs — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar in hertz.

t — Sample instants

vector

Sample instants, specified as a vector. The length of `t` must equal the length of the input bilevel waveform `x`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `p = pulseperiod(x,Fs,StateLevels=[0 5])`

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `p = pulseperiod(x,Fs,'StateLevels',[0 5])`

`MidPercentReferenceLevel` — Mid-reference level
50 (default) | scalar

Mid-reference level as a percentage of the waveform amplitude, specified as a scalar. For more information, see “Mid-Reference Level” on page 1-1772.

`Polarity` — Pulse polarity
`"positive"` (default) | `"negative"`

Pulse polarity, specified as `"positive"` or `"negative"`. If you specify `"positive"`, the function looks for pulses whose initial transition is positive-going (positive polarity). If you specify `"negative"`, the function looks for pulses whose initial transition is negative-going (negative polarity). For more information, see “Pulse Polarity” on page 1-1772

`StateLevels` — Low- and high-state levels
1-by-2 real-valued vector

Low- and high-state levels, specified as a 1-by-2 real-valued vector. The first element is the low-state level and the second element is the high-state level. If you do not specify `'StateLevels'`, the function estimates the state levels from the input waveform using the histogram method.

`Tolerance` — Tolerance levels
2 (default) | scalar

Tolerance levels (lower- and upper-state boundaries), specified as a scalar and expressed as a percentage. 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. For more information, see “State-Level Tolerances” on page 1-1773.

Output Arguments

`p` — Pulse period
vector

Pulse period in seconds, returned as a vector. The pulse period is defined as the time between the mid-reference level instants of two consecutive transitions. If the function does not find two positive-polarity transitions, `p` is empty.

Note 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`.

`initcross` — Mid-reference level instants of initial transition
vector

Mid-reference level instants of the initial transition, returned as a vector.

`finalcross` — Mid-reference level instants of final transition
vector

Mid-reference level instants of the final transition, returned as a vector.

`nextcross` — Mid-reference level instants of initial transition after final transition of preceding pulse
vector

Mid-reference level instants of the first pulse transition after the final transition of the preceding pulse, returned as a vector.

`midlev` — Waveform value
scalar

Waveform value that corresponds to the mid-reference level, returned as a scalar.

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

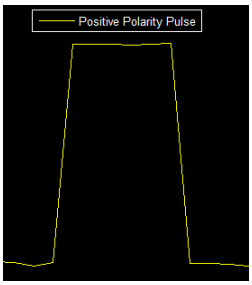
- Let $y_{50\%}$ denote the mid-reference level.
- Let $t_{50\%_-}$ and $t_{50\%_+}$ denote the two consecutive sampling instants corresponding to the waveform values nearest in value to $y_{50\%}$.
- Let $y_{50\%_-}$ and $y_{50\%_+}$ denote the waveform values at $t_{50\%_-}$ and $t_{50\%_+}$.

The mid-reference level instant is

$$t_{50\%} = t_{50\%} + (\frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}})(y_{50\%_+} - y_{50\%_-})$$

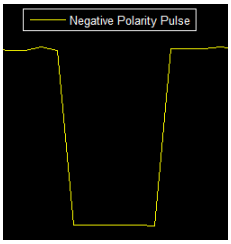
Pulse Polarity

If the initial transition of a pulse is positive-going, the pulse has positive polarity. This 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. This figure shows a negative-polarity pulse.



Equivalently, a negative-polarity (negative-going) pulse has a originating state more positive than the terminating state.

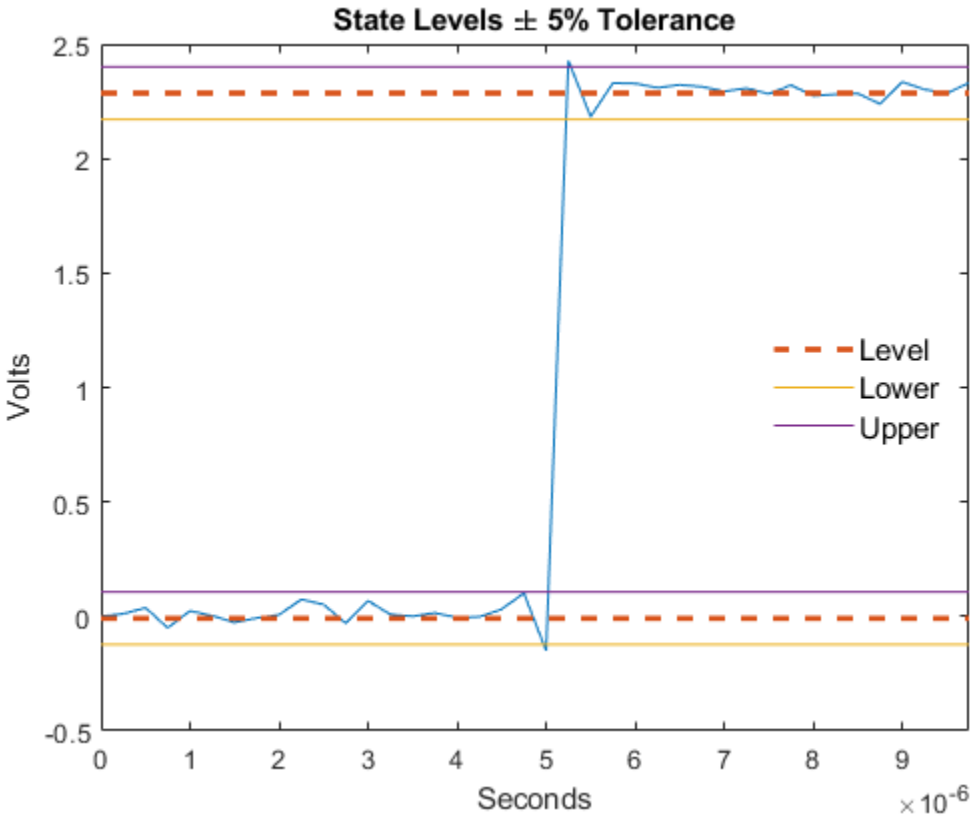
State-Level Tolerances

You can specify lower- and upper-state boundaries for each state level. Define the boundaries as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, specify the scalar as a small number such as 2/100 or 3/100. In general, the $\alpha\%$ 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.

This figure shows lower and upper 5% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The thick dashed lines indicate the estimated state levels.



References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`dutycycle` | `pulsesep` | `pulsewidth` | `statelevels`

Introduced in R2012a

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 between the mid-reference level instants of the final negative-going transitions of every positive-polarity pulse and the next positive-going transition. To determine the transitions that compose each pulse, the `pulsesep` function estimates the state levels of the input waveform by a histogram method. The function identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state.

`S = pulsesep(x,Fs)` specifies the sample rate `Fs`.

`S = pulsesep(x,t)` specifies the sampling instants `t`.

`[s,initcross] = pulsesep( ___ )` returns the mid-reference level instants `initcross` of the first positive-polarity transitions. You can specify an input combination from any of the previous syntaxes.

`[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 arguments.

`pulsesep( ___ )` plots the signal and darkens the regions between each pulse where pulse separation is computed. The function marks the location of the mid crossings and their associated reference level. The function also plots the state levels and their associated lower and upper boundaries. You can adjust the boundaries using the `'Tolerance'` name-value argument.

Examples

Pulse Separation in Bilevel Waveform

Compute the pulse separation in a bilevel waveform with two positive-polarity transitions. The sample rate is 4 MHz.

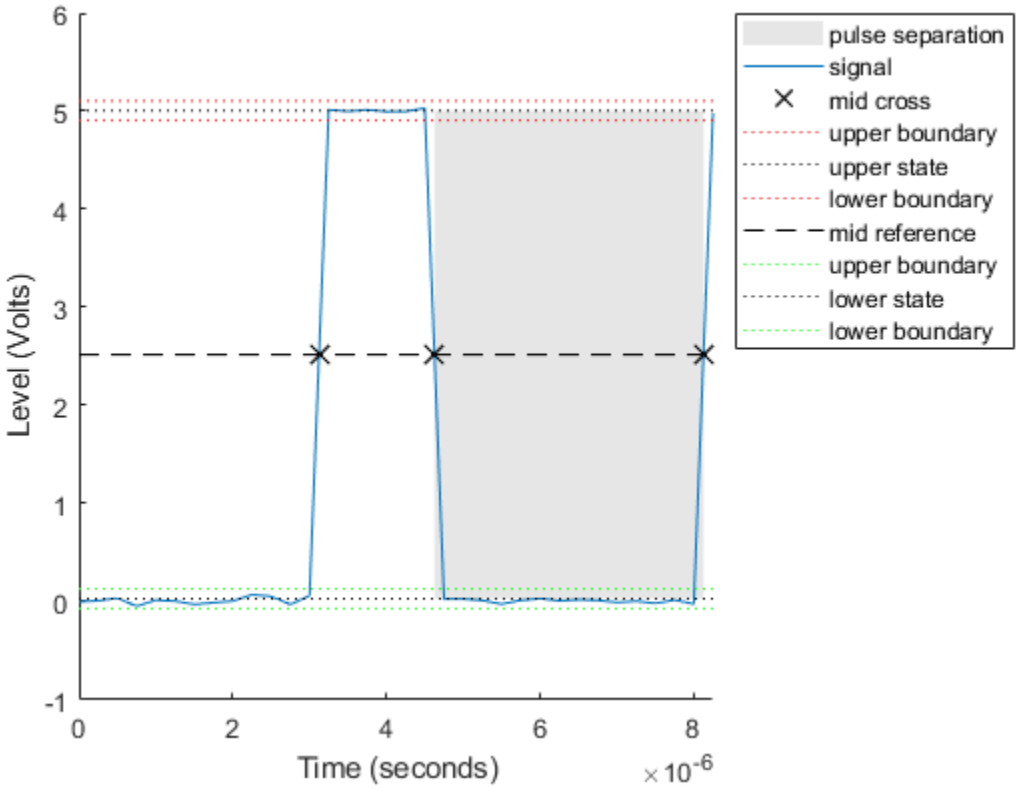
```
load('pulseex.mat','x','t')
```

```
s = pulsesep(x,t)
```

```
s = 3.5014e-06
```

Plot the waveform and annotate the pulse separation.

```
pulsesep(x,t);
```



Mid-Reference Level Instants Defining Pulse Separation

Determine the mid-reference level instants that define the pulse separation for a bilevel waveform.

```
load('pulseex.mat','x','t')
```

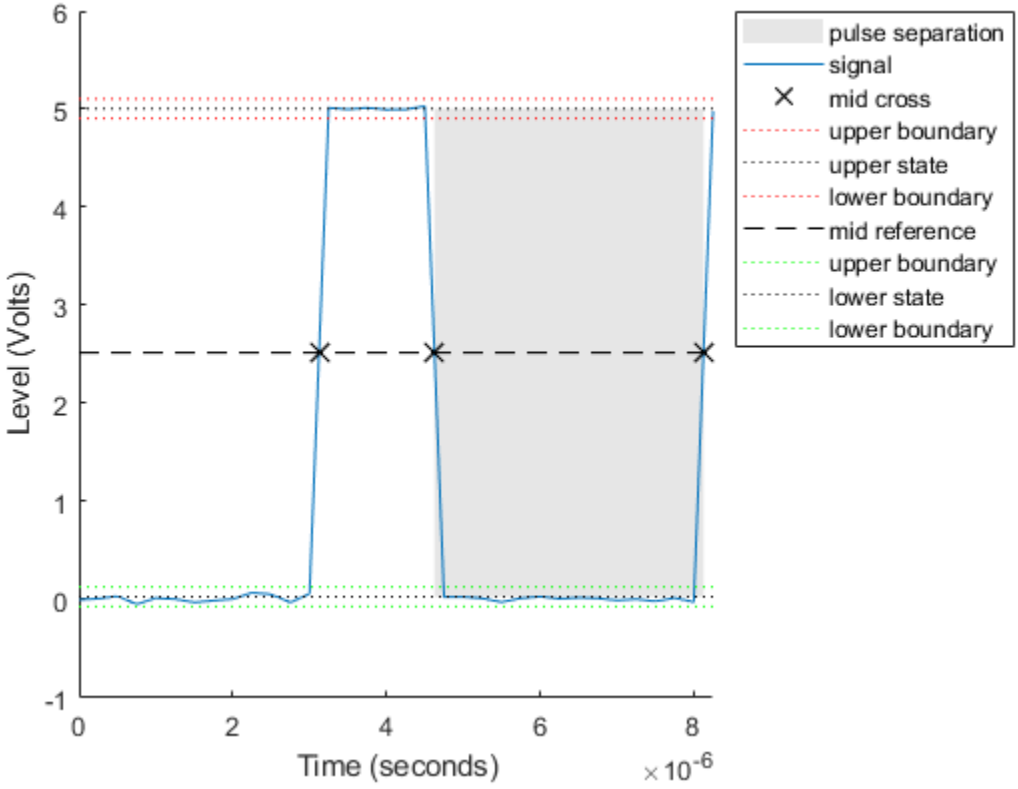
```
[~,~,finalcross,nextcross] = pulsesep(x,t)
```

```
finalcross = 4.6256e-06
```

```
nextcross = 8.1270e-06
```

Return the pulse separation. Annotate the mid-reference level instants on a plot of the data.

```
pulsesep(x,t)
```




```
ans = 3.5014e-06
```

Input Arguments

x — Bilevel waveform

real-valued vector

Bilevel waveform, specified as a real-valued vector. If the waveform does not contain at least two transitions, the function outputs an empty matrix. The first time instant in `x` corresponds to t=0.

Fs — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar in hertz.

t — Sample instants

vector

Sample instants, specified as a vector. The length of `t` must equal the length of the input bilevel waveform `x`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `p = pulsesep(x,t,Polarity="negative")`

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `p = pulsesep(x,t,'Polarity',"negative")`

`MidPercentReferenceLevel` — Mid-reference level
`50` (default) | scalar

Mid-reference level as a percentage of the waveform amplitude, specified as scalar. For more information, see “Mid-Reference Level” on page 1-1779.

`Polarity` — Pulse polarity
`"positive"` (default) | `"negative"`

Pulse polarity, specified as `"positive"` or `"negative"`. If you specify `"positive"`, the function looks for pulses with positive-going (positive polarity) initial transitions. If you specify `"negative"`, the function looks for pulses with negative-going (negative polarity) initial transitions. For more information, see “Pulse Polarity” on page 1-1779.

`StateLevels` — Low- and high-state levels
1-by-2 real-valued vector

Low- and high-state levels, specified as a 1-by-2 real-valued vector. The first element is the low-state level and the second element is the high-state level. If you do not specify `'StateLevels'`, the function estimates the state levels from the input waveform using the histogram method.

`Tolerance` — Tolerance levels
2 (default) | scalar

Tolerance levels (lower- and upper-state boundaries), specified as a scalar and expressed as a percentage. 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. For more information, see “State-Level Tolerances” on page 1-1780.

Output Arguments

`s` — Pulse separations
vector

Pulse separations in seconds, returned as a vector. 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. For more information, see “Pulse Separation” on page 1-1781.

Note 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`.

`initcross` — Mid-reference level instants of initial transition
vector

Mid-reference level instants of the initial transition, returned as a vector.

`finalcross` — Mid-reference level instants of final transition
vector

Mid-reference level instants of the final transition, returned as a vector.

`nextcross` — Mid-reference level instants of initial transition after final transition of preceding pulse
vector

Mid-reference level instants of the initial transition after the final transition of the preceding pulse, returned as a vector.

`midlev` — Waveform value
scalar

Waveform value that corresponds to the mid-reference level, returned as a scalar.

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

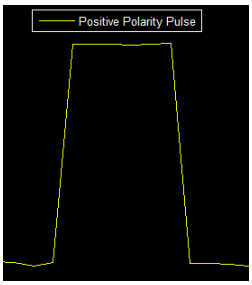
- Let $y_{50\%}$ denote the mid-reference level.
- Let $t_{50\%_-}$ and $t_{50\%_+}$ denote the two consecutive sampling instants corresponding to the waveform values nearest in value to $y_{50\%}$.
- Let $y_{50\%_-}$ and $y_{50\%_+}$ denote the waveform values at $t_{50\%_-}$ and $t_{50\%_+}$.

The mid-reference level instant is

$$t_{50\%} = t_{50\%} + (\frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}})(y_{50\%_+} - y_{50\%_-})$$

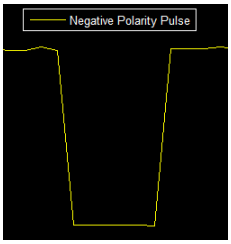
Pulse Polarity

If the pulse has an initial positive-going transition, the pulse has positive polarity. This 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. This figure shows a negative-polarity pulse.



Equivalently, a negative-polarity (negative-going) pulse has a originating state more positive than the terminating state.

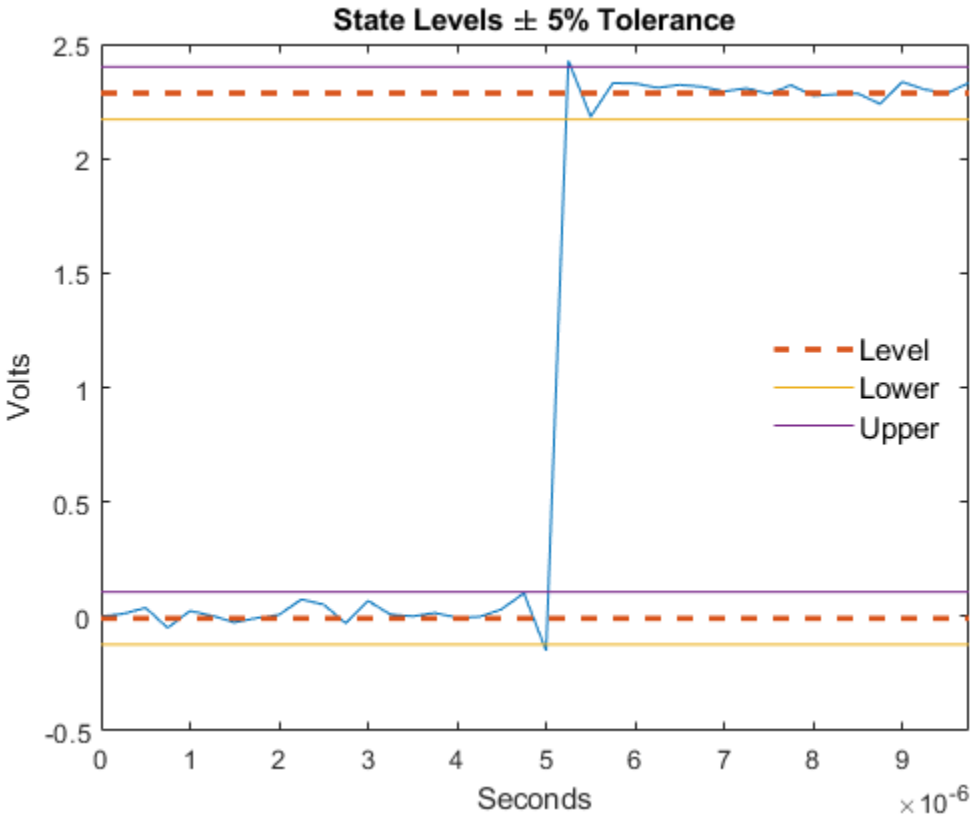
State-Level Tolerances

You can specify lower- and upper-state boundaries for each state level. Define the boundaries as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, specify the scalar as a small number such as 2/100 or 3/100. In general, the $\alpha\%$ region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1),$$

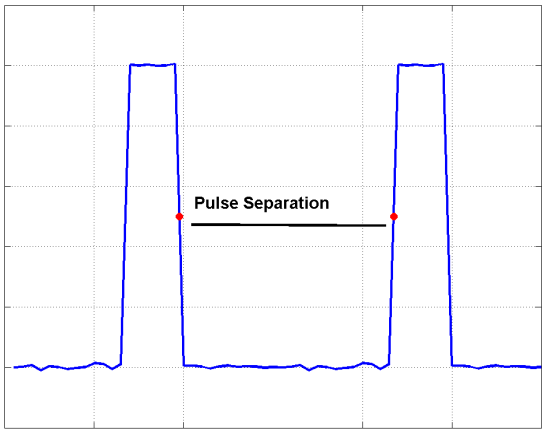
where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the $\alpha\%$ tolerance region for the high state.

This figure shows lower and upper 5% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The thick 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. This figure illustrates pulse separation.



References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`dutycycle` | `pulseperiod` | `pulsewidth` | `statelevels`

Introduced in R2012a

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 the time differences between the midreference level instants of the initial and final transitions of each positive-polarity pulse in the input bilevel waveform.

`w = pulsewidth(x,fs)` specifies the sample rate `fs` in hertz. The first sample in the waveform corresponds to t = 0.

`w = pulsewidth(x,t)` specifies the sample instants `t`.

`[w,initcross] = pulsewidth( ___ )` returns `initcross`, the midreference level instants of the initial transition of each pulse. You can specify an input combination from any of the previous syntaxes.

`[w,initcross,finalcross] = pulsewidth( ___ )` returns `finalcross`, the midreference level instants of the final transition of each pulse.

`[w,initcross,finalcross,midlev] = pulsewidth( ___ )` returns the waveform value `midlev` that corresponds to the midreference level.

`W = pulsewidth( ___ ,Name,Value)` specifies additional options using one or more `Name,Value` arguments.

`pulsewidth( ___ )` plots the signal and darkens the regions of each pulse where the function computes the pulse width. The function marks the location of the midcrossings and their associated reference level. The function also plots the state levels and their associated lower and upper boundaries.

Examples

Pulse Width of Bilevel Waveform

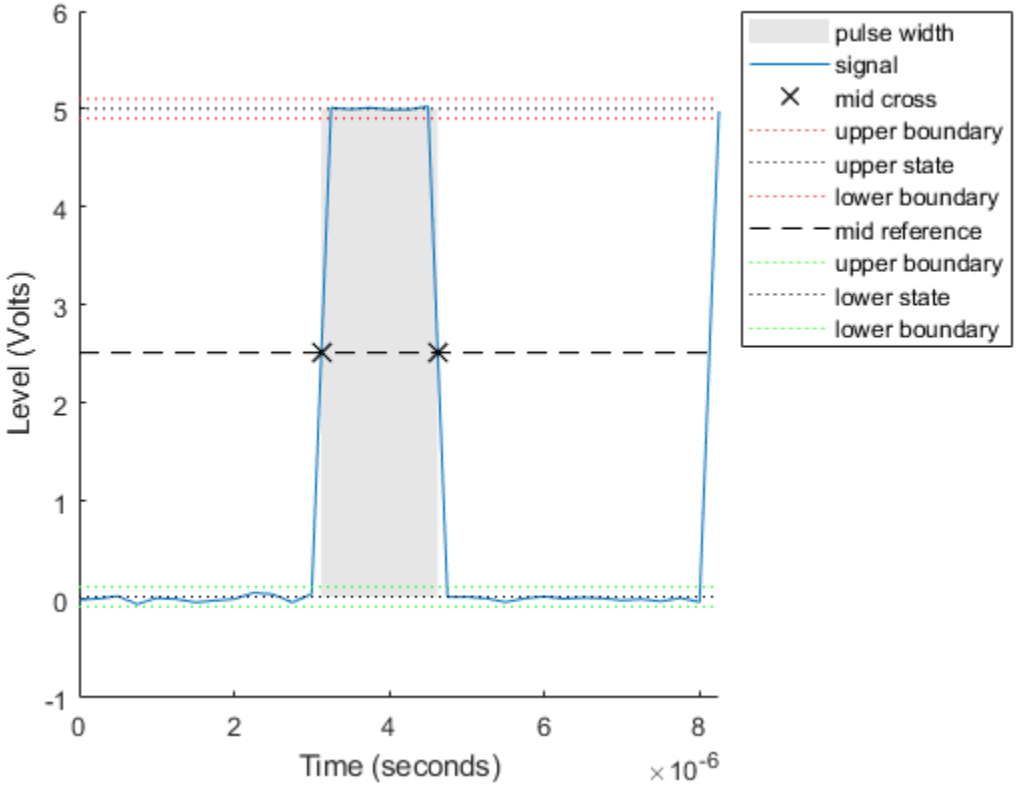
Compute the pulse width of a bilevel waveform sampled at 4 MHz.

```
load('pulseex.mat','x','t')
w = pulsewidth(x,t)
```

```
w = 1.5016e-06
```

Plot the waveform and annotate the pulse width.

```
pulsewidth(x,t);
```



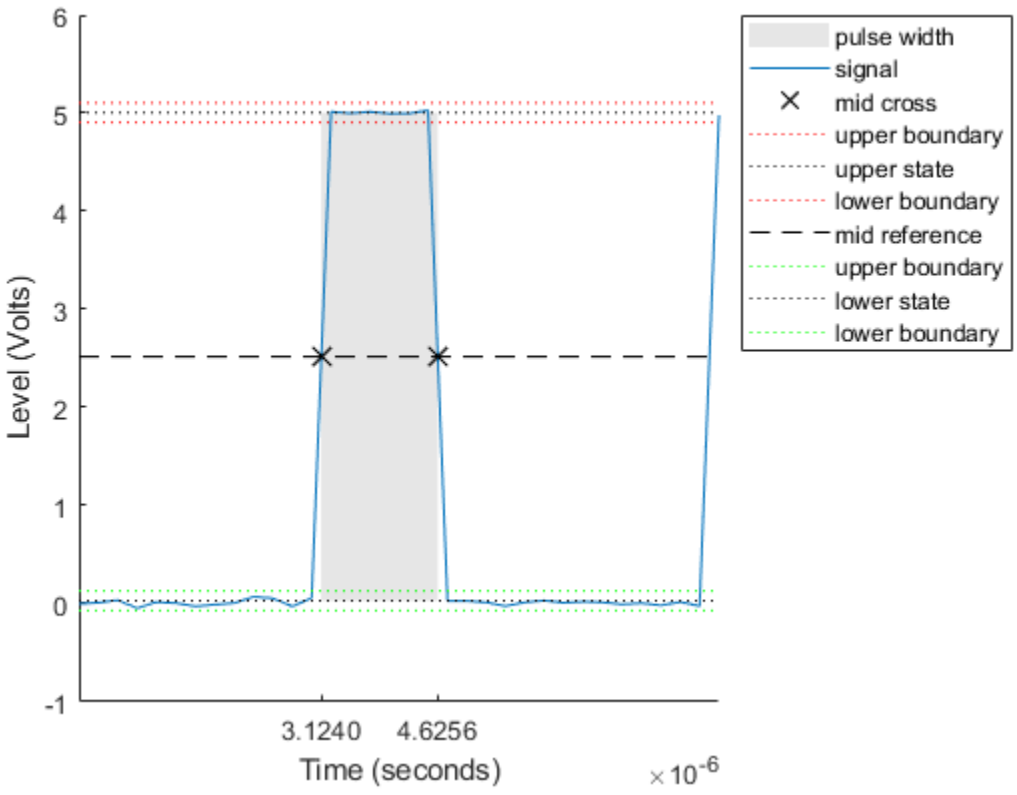
Compute First and Second Transition Times for Bilevel Waveform

Compute the initial and final transition occurrences for a bilevel waveform sampled at 4 MHz.

```
load('pulseex.mat','x','t');
fs = 4e6;

[w,initcross,finalcross] = pulsewidth(x,fs);
```

Plot the result, annotated with the transition occurrences.

```
pulsewidth(x,fs);
ax = gca;
ax.XTick = [initcross finalcross];
```



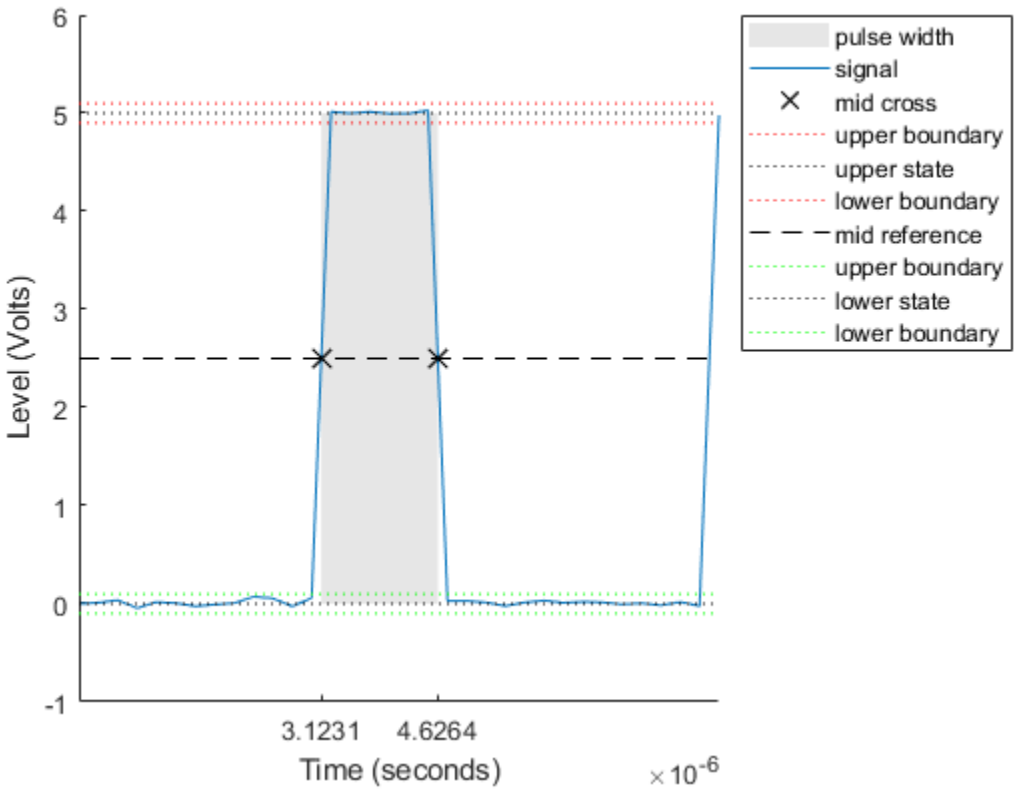
Specify State Levels for Bilevel Waveform

Specify the state levels for the bilevel waveform instead of estimating the levels from the data. Specify the low-state level as 0 and the high-state level as 5.

```
load('pulseex.mat','x','t')
fs = 4e6;
[w,initcross,finalcross] = pulsewidth(x,fs,'StateLevels',[0 5]);
```

Plot the result annotated with the transition occurrences.

```
pulsewidth(x,fs,'StateLevels',[0 5]);
ax = gca;
ax.XTick = [initcross finalcross];
```



Input Arguments

x — Bilevel waveform

real-valued vector

Bilevel waveform, specified as a real-valued vector.

fs — Sample rate

real positive scalar

Sample rate in hertz, specified as a real positive scalar.

t — Sample instants

vector

Sample instants, specified as a vector. The length of T must equal the length of the bilevel waveform x.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'StateLevels',[0 5]` specifies a low-state level of 0 and a high-state level of 5.

`MidPercentReferenceLevel` — Midreference level
`50` (default) | real-valued scalar

Midreference level as a percentage of the waveform amplitude, specified as a real-valued scalar. For more information, see “Midreference Level” on page 1-1788.

`Polarity` — Pulse polarity
`'positive'` (default) | `'negative'`

Pulse polarity, specified as `'positive'` or `'negative'`. If you specify `'positive'`, the function looks for pulses with positive-going (positive polarity) initial transitions. If you specify `'negative'`, the `pulsewidth` function looks for pulses with negative-going (negative polarity) initial transitions. For more information, see “Pulse Polarity” on page 1-1788.

`StateLevels` — Low- and high-state levels
1-by-2 real-valued vector

Low- and high-state levels, specified as a 1-by-2 real-valued vector. The first element is the low-state level, and the second element is the high-state level. If you do not specify low- and high-state levels, the `pulsewidth` function estimates the state levels from the input waveform using the histogram method. For a detailed description of the histogram method, see “State-Level Estimation” on page 1-1788.

`Tolerance` — Tolerance levels
2 (default) | real-valued scalar

Tolerance levels (lower- and upper-state boundaries) expressed as a percentage, specified as a real-valued scalar. For more information, see “State-Level Tolerances” on page 1-1789.

Output Arguments

`w` — Pulse widths
vector

Pulse widths in seconds, returned as a vector. 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] on page 1-1790.

Note Because the `pulsewidth` function uses interpolation to determine the midreference level instants, `w` might contain values that do not correspond to sampling instants of the bilevel waveform `x`.

`initcross` — Midreference level instants of initial transition
column vector

Midreference level instants of the initial transition of each pulse, returned as a column vector.

`finalcross` — Midreference level instants of final transition
column vector

Midreference level instants of the final transition of each pulse, returned as a column vector.

`midlev` — Waveform value
scalar

Waveform value corresponding to the midreference level, returned as a scalar.

More About

State-Level Estimation

To determine the transitions, the `pulsewidth` function estimates the low- and high-state levels of input `x` by using a histogram method with these steps.

1. Determine the minimum and maximum amplitudes of the data.
2. For the specified number of histogram bins, determine the bin width, which is the ratio of the amplitude range to the number of bins.
3. Sort the data values into the histogram bins.
4. Identify the lowest and highest indexed histogram bins with nonzero counts.
5. Divide the histogram into two subhistograms.
6. Compute the state levels by determining the mode or mean of the upper and lower histograms.

The function 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.

Midreference Level

The midreference level in a bilevel waveform with low-state level, S_1, and high-state level, S_2, is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

Midreference Level Instant

The midreference level instant is

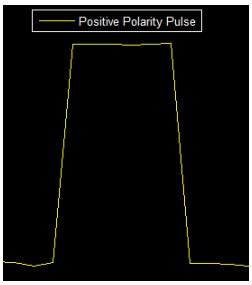
$$t_{50\%} = t_{50\%} + (\frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}})(y_{50\%_+} - y_{50\%_-})$$

where:

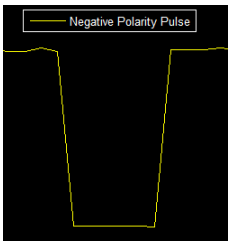
- $y_{50\%}$ denotes the midreference level.
- $t_{50\%_-}$ and $t_{50\%_+}$ denote the two consecutive sampling instants corresponding to the waveform values that are nearest in value to $y_{50\%}$.
- $y_{50\%_-}$ and $y_{50\%_+}$ denote the waveform values at $t_{50\%_-}$ and $t_{50\%_+}$.

Pulse Polarity

If the pulse has a positive-going initial transition, the pulse has positive polarity. Equivalently, a positive-polarity (positive-going) pulse has a terminating state that is more positive than the originating state. This figure shows a positive-polarity pulse.



If the pulse has a negative-going initial transition, the pulse has negative polarity. Equivalently, a negative-polarity (negative-going) pulse has an originating state that is more positive than the terminating state. This figure shows a negative-polarity pulse.



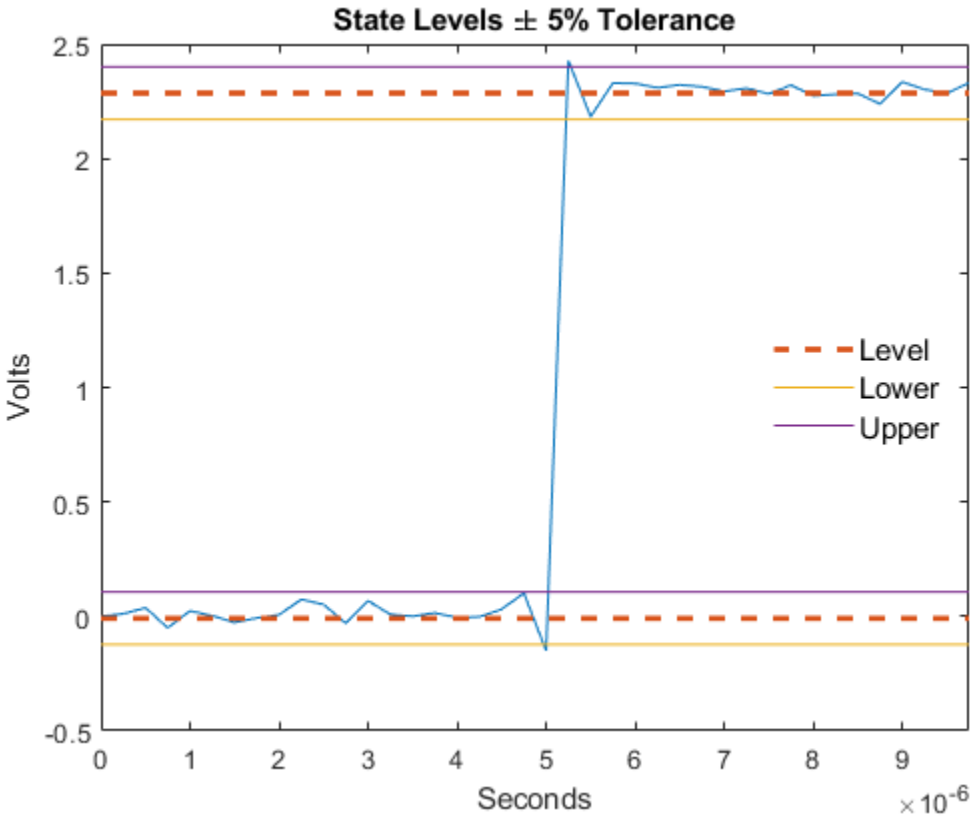
State-Level Tolerances

You can specify lower- and upper-state boundaries for each state level. Define the boundaries as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, specify the scalar as a small number such as 2/100 or 3/100. In general, the $\alpha\%$ 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.

This figure shows lower and upper 5% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The thick dashed lines indicate the estimated state levels.



References

[1] IEEE Standard 181. *IEEE Standard on Transitions, Pulses, and Related Waveforms* (2003).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`dutycycle` | `pulseperiod` | `pulsesep` | `statelevels`

Introduced in R2012a

pulstran

Pulse train

Syntax

```
y = pulstran(t,d,func)
y = pulstran(t,d,func,fs)
y = pulstran(t,d,p)
y = pulstran( ___ ,intfunc)
```

Description

`y = pulstran(t,d,func)` generates a pulse train based on samples of a continuous function, `func`.

`y = pulstran(t,d,func,fs)` uses a sample rate of `fs`.

`y = pulstran(t,d,p)` generates a pulse train that is the sum of multiple delayed interpolations of the prototype pulse in vector `p`.

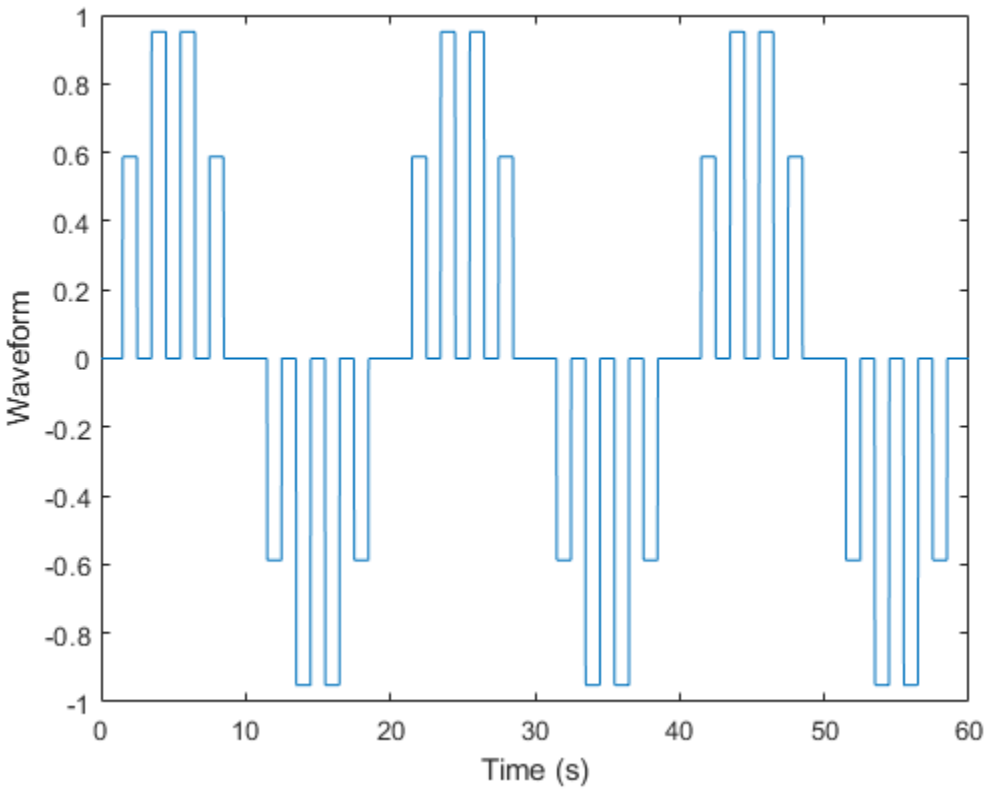
`y = pulstran( ___ ,intfunc)` specifies alternative interpolation methods. See `interp1` for a list of available methods. You can use this parameter with any of the previous input syntaxes.

Examples

Periodic Rectangular Pulse

This example generates a pulse train using the default rectangular pulse of unit width. The repetition frequency is 0.5 Hz, the signal length is 60 s, and the sample rate is 1 kHz. The gain factor is a sinusoid of frequency 0.05 Hz.

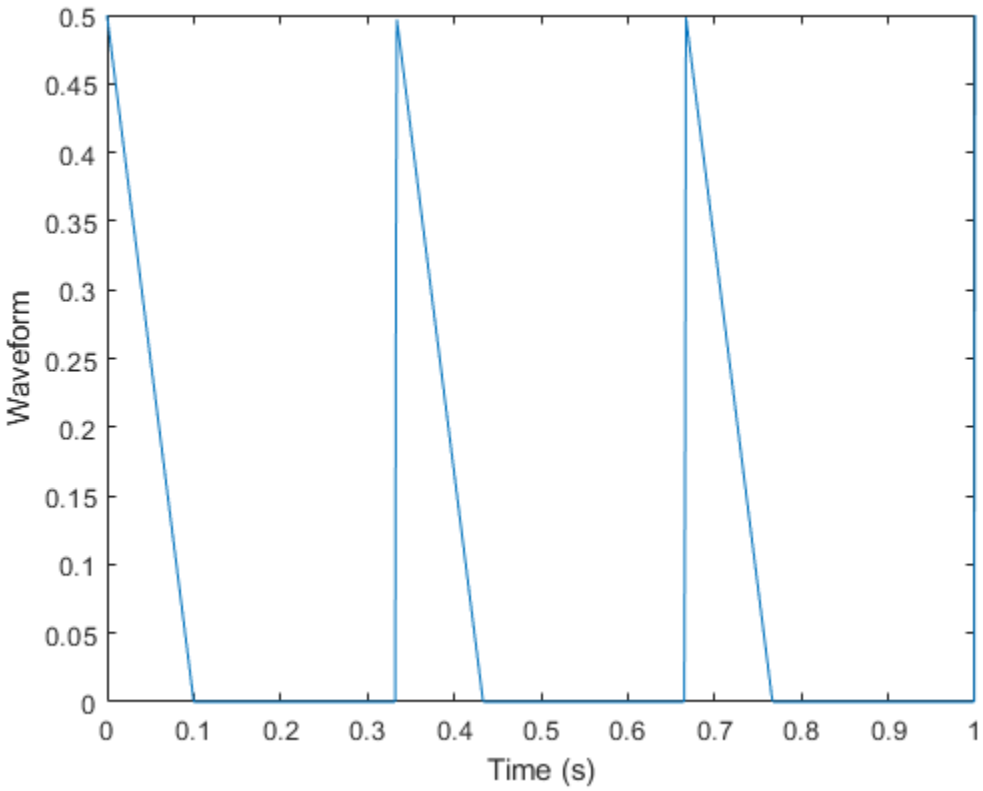
```
t = 0:1/1e3:60;
d = [0:2:60;sin(2*pi*0.05*(0:2:60))]';
x = @rectpuls;
y = pulstran(t,d,x);

plot(t,y)
hold off
xlabel('Time (s)')
ylabel('Waveform')
```



Asymmetric Sawtooth Waveform

This example generates an asymmetric sawtooth waveform with a repetition frequency of 3 Hz. The sawtooth has width 0.2 s and skew factor –1. The signal length is 1 s, and the sample rate is 1 kHz. Plot the pulse train.

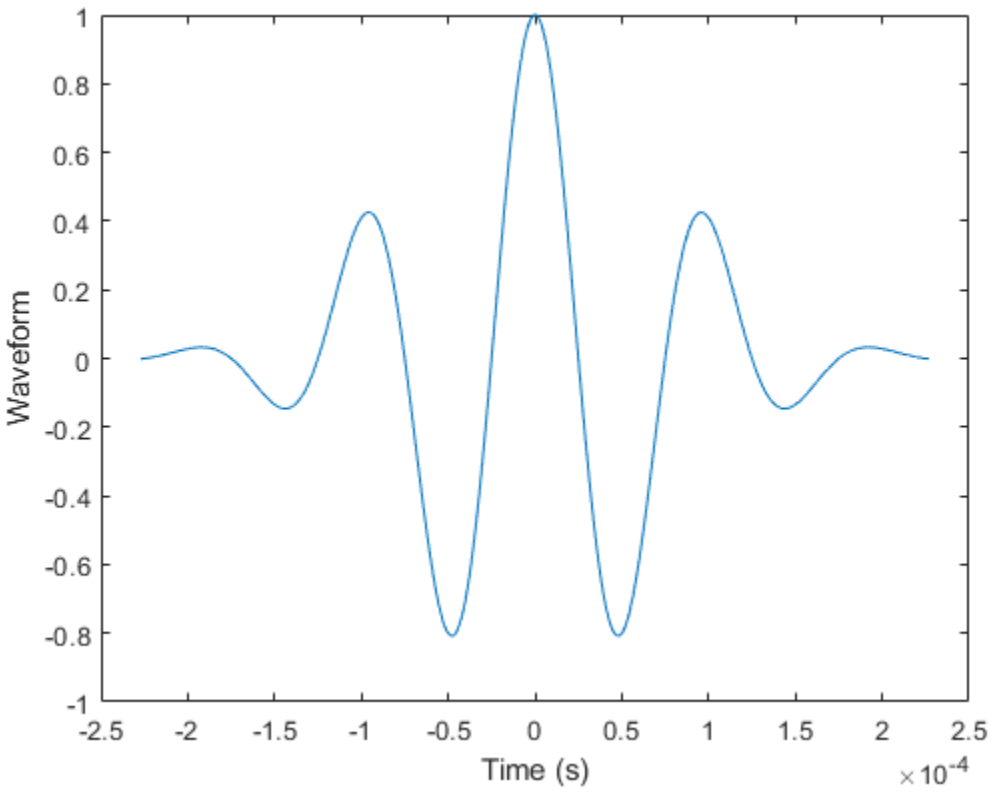
```
fs = 1e3;
t = 0:1/1e3:1;
d = 0:1/3:1;
x = tripuls(t,0.2,-1);
y = pulstran(t,d,x,fs);

plot(t,y)
hold off
xlabel('Time (s)')
ylabel('Waveform')
```



Periodic Gaussian Pulse

Plot a 10 kHz Gaussian RF pulse with 50% bandwidth, sampled at a rate of 10 MHz. Truncate the pulse where the envelope falls 40 dB below the peak.

```
fs = 1e7;
tc = gauspuls('cutoff',10e3,0.5,[],-40);
t = -tc:1/fs:tc;
x = gauspuls(t,10e3,0.5);

plot(t,x)
xlabel('Time (s)')
ylabel('Waveform')
```

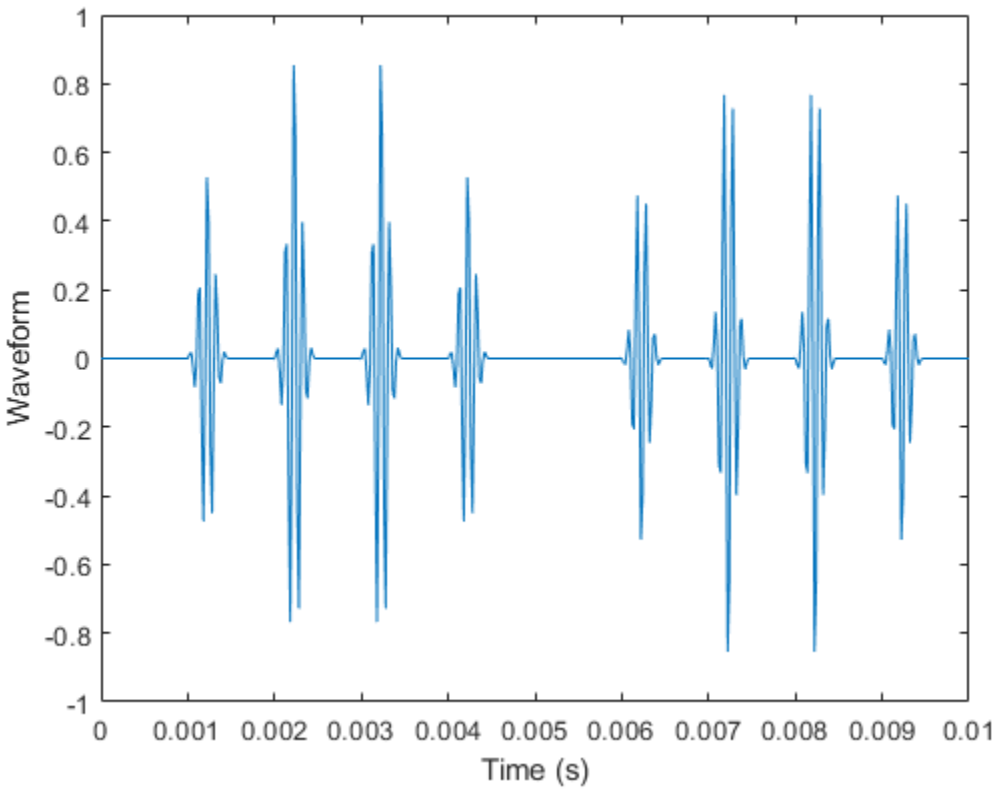


The pulse repetition frequency is 1 kHz, the sample rate is 50 kHz, and the pulse train length is 25 ms. The gain factor is a sinusoid of frequency 0.1 Hz.

```
ts = 0:1/50e3:0.025;
d = [0:1/1e3:0.025;sin(2*pi*0.1*(0:25))]';
y = pulstran(ts,d,x,fs);
```

Plot the periodic Gaussian pulse train.

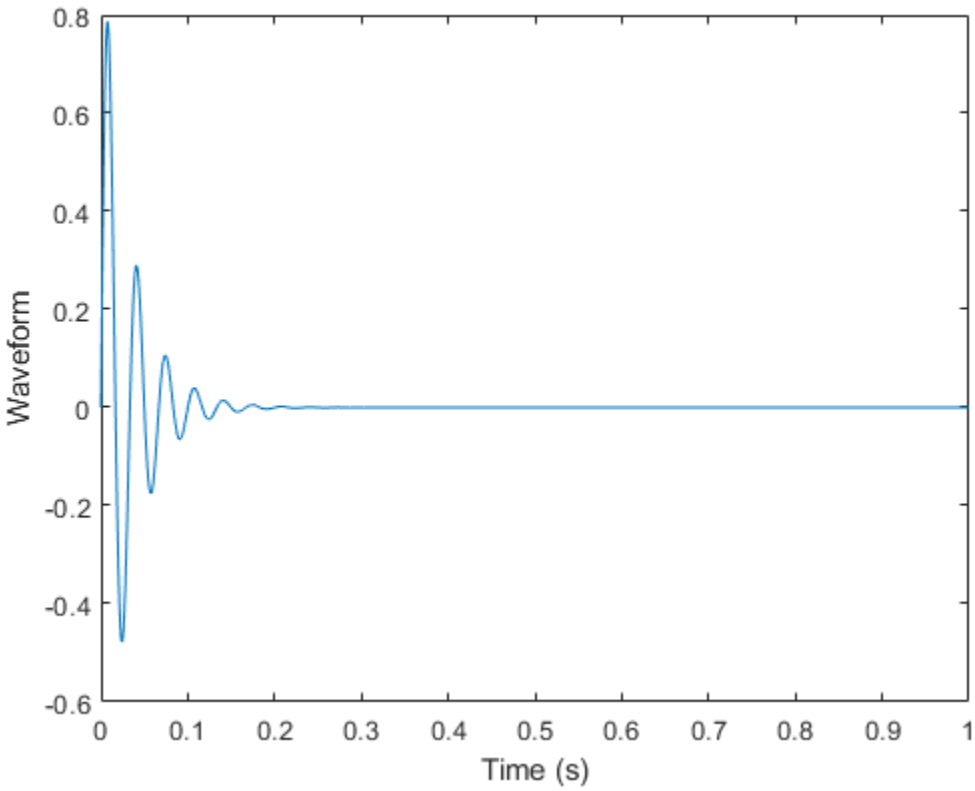
```
plot(ts,y)
xlim([0 0.01])
xlabel('Time (s)')
ylabel('Waveform')
```



Custom Pulse Trains

Write a function that generates custom pulses consisting of a sinusoid damped by an exponential. The pulse is an odd function of time. The generating function has a second input argument that specifies a single value for the sinusoid frequency and the damping factor. Display a generated pulse, sampled at 1 kHz for 1 second, with a frequency and damping value, both equal to 30.

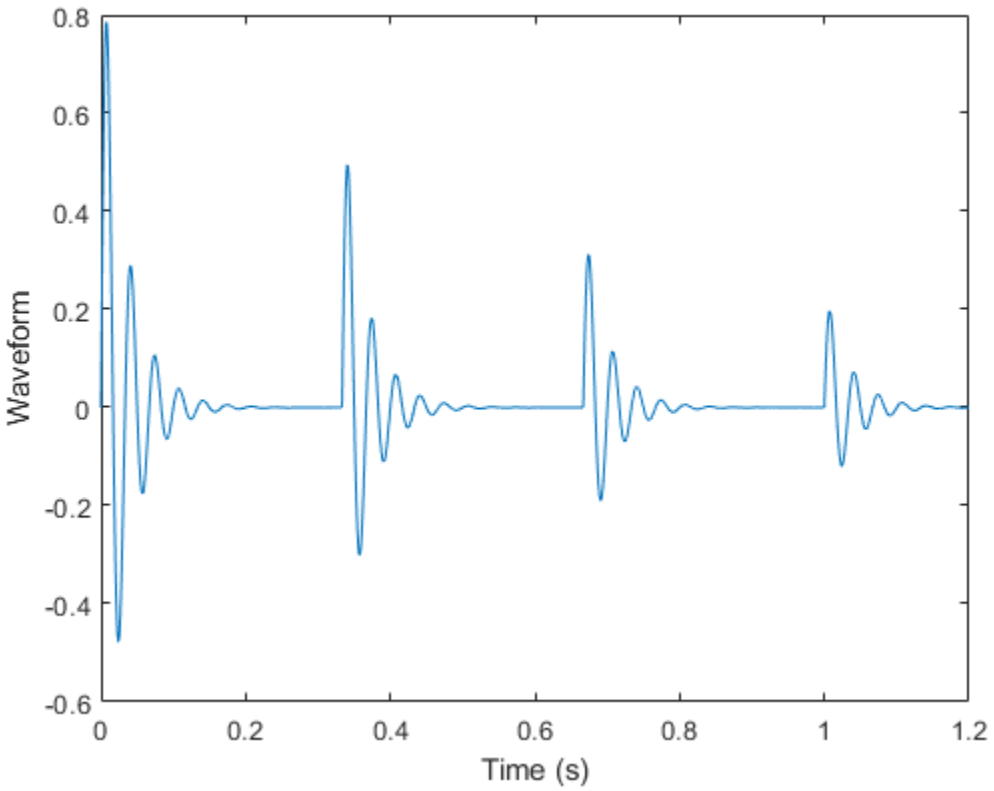
```
fnx = @(x,fn) sin(2*pi*fn*x).*exp(-fn*abs(x));

ffs = 1000;
tp = 0:1/ffs:1;

pp = fnx(tp,30);

plot(tp,pp)
xlabel('Time (s)')
ylabel('Waveform')
```



Use the `pulstran` function to generate a train of custom pulses. The train is sampled at 2 kHz for 1.2 seconds. The pulses occur every third of a second and have exponentially decreasing amplitudes.

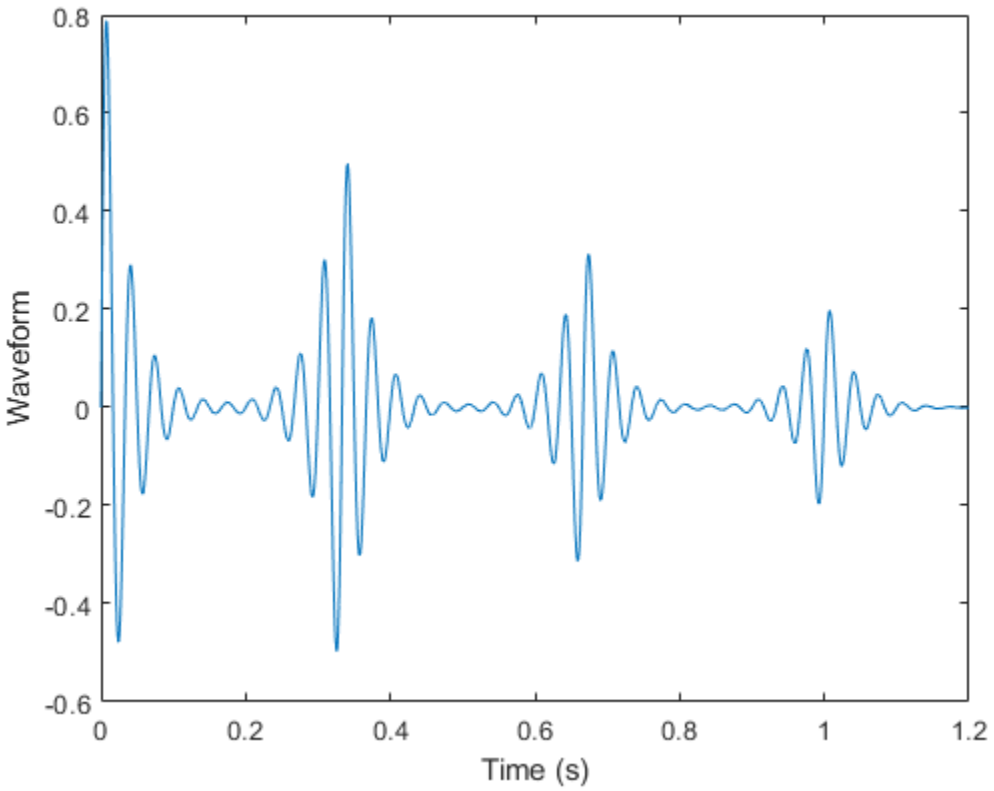
Initially specify the generated pulse as a prototype. Include the prototype sample rate in the function call. In this case, `pulstran` replicates the pulses at the specified locations.

```
fs = 2e3;
t = 0:1/fs:1.2;

d = 0:1/3:1;
dd = [d;4.^-d]';

z = pulstran(t,dd,pp,ffs);

plot(t,z)
xlabel('Time (s)')
ylabel('Waveform')
```



Generate the pulse train again, but now use the generating function as an input argument. Include the frequency and damping parameter in the function call. In this case, `pulstran` generates the pulse so that it is centered about zero.

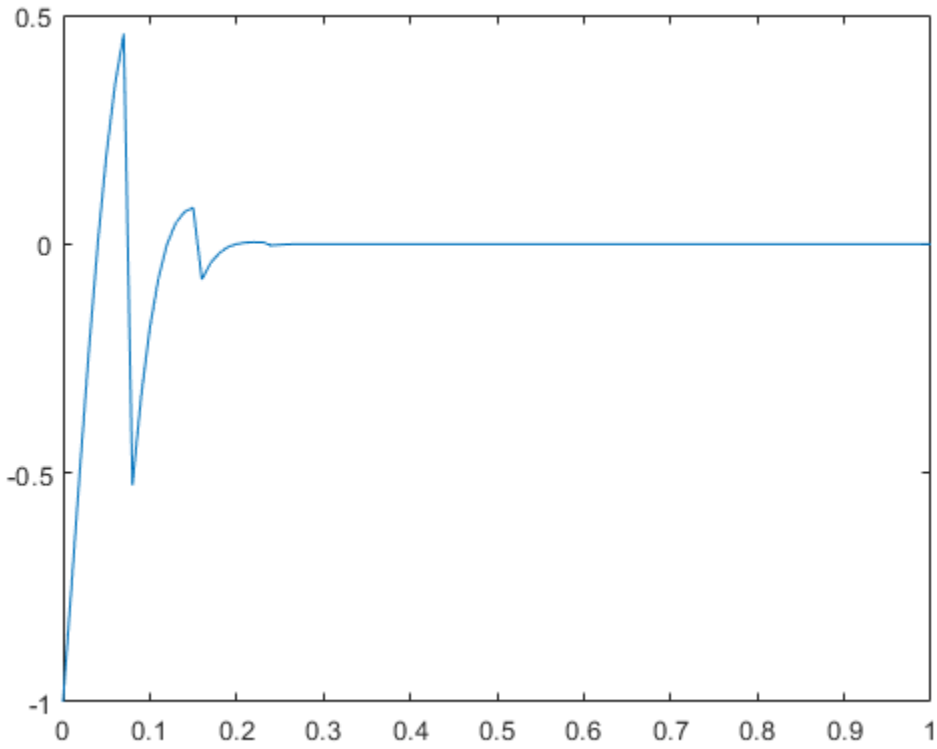
```
y = pulstran(t,dd,fnx,30);

plot(t,y)
xlabel('Time (s)')
ylabel('Waveform')
```



Change Interpolation Method with Custom Pulse

Write a function that generates a custom exponentially decaying sawtooth waveform of frequency 0.25 Hz. The generating function has a second input argument that specifies a single value for the sawtooth frequency and the damping factor. Display a generated pulse, sampled at 0.1 kHz for 1 second, with a frequency and damping value equal to 50.

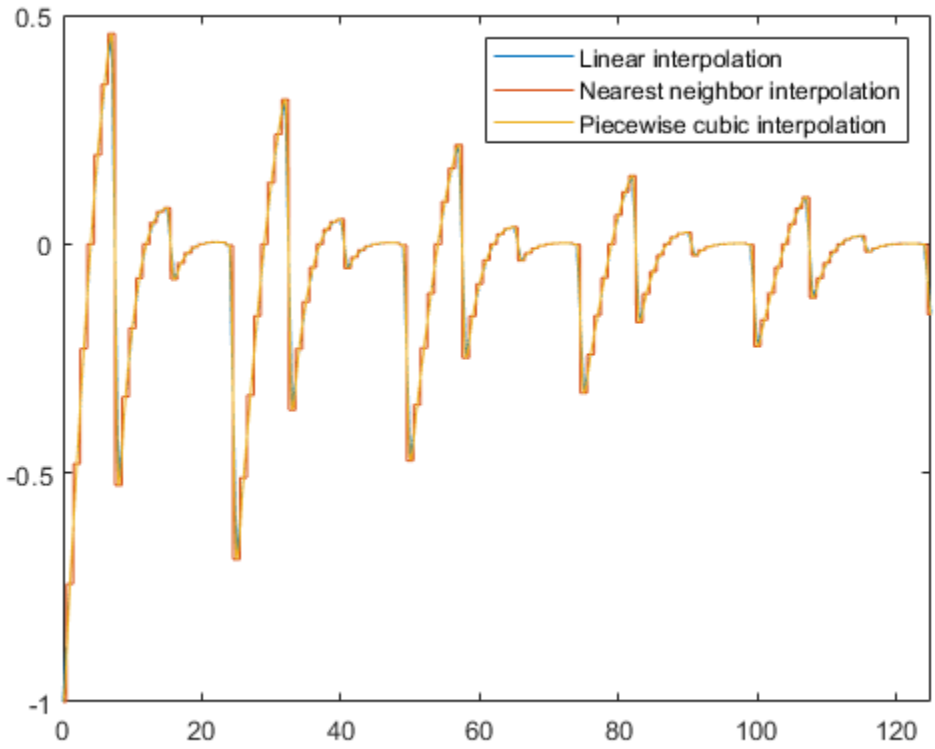
```
fnx = @(x,fn) sawtooth(2*pi*fn*0.25*x).*exp(-2*fn*x.^2);

fs = 100;
t = 0:1/fs:1;

pp = fnx(t,50);

plot(t,pp)
```



Use the `pulstran` function to generate a train of custom pulses. The train is sampled at 0.1 kHz for 125 seconds. The pulses occur every 25 seconds and have exponentially decreasing amplitudes.

Specify the generated pulse as a prototype. Generate three pulse trains using the default linear interpolation method, nearest neighbor interpolation and piecewise cubic interpolation. Compare the pulse trains on a single plot.

```
d = [0:25:125; exp(-0.015*(0:25:125))]';
ffs = 100;
tp = 0:1/ffs:125;

r = pulstran(tp,d,pp);
y = pulstran(tp,d,pp,'nearest');
q = pulstran(tp,d,pp,'pchip');


plot(tp,r)
hold on
plot(tp,y)
plot(tp,q)
xlim([0 125])
legend('Linear interpolation','Nearest neighbor interpolation','Piecewise cubic interpolation')
hold off
```



Input Arguments

`t` — Time values

vector

Time values at which `func` is evaluated, specified as a vector.

`d` — Offset

row vector | two-column matrix

Offset removed from the values of the array `t`, specified as a real vector. You can apply an optional gain factor to each delayed evaluation by specifying `d` as a two-column matrix, with offset defined in column 1 and associated gain in column 2. If you specify `d` as a row vector, the values are interpreted as delays only.

`func` — Continuous function

`'rectpuls'` | `'gauspuls'` | `'tripuls'` | function handle

Continuous function used to generate a pulse train based on its samples, specified as `'rectpuls'`, `'gauspuls'`, `'tripuls'`, or a function handle.

If you use `func` as a function handle, you can pass the function parameters as follows:

```
y = pulstran(t,d,'gauspuls',10e3,0.5);
```

This creates a pulse train using a 10 kHz Gaussian pulse with 50% bandwidth.

`p` — Prototype pulse

vector

Prototype function, specified as a vector. The interval of `p` is given by `[0,(length(p)-1)/fs]`, and its samples are identically zero outside this interval. By default, linear interpolation is used for generating delays.

`fs` — Sample rate

1 (default) | real scalar

Sample rate in Hz, specified as a real scalar.

`intfunc` — Interpolation method

`'linear'` (default) | `'nearest'` | `'next'` | `'previous'` | `'pchip'` | `'cubic'` | `'v5cubic'` | `'makima'` | `'spline'`

Interpolation method, specified as one of the options in this table.

Method	Description	Continuity	Comments
`'linear'`	Linear interpolation. The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension. This is the default interpolation method.	C^0	• Requires at least 2 points • Requires more memory and computation time than nearest neighbor
`'nearest'`	Nearest neighbor interpolation. The interpolated value at a query point is the value at the nearest sample grid point.	Discontinuous	• Requires at least 2 points • Modest memory requirements • Fastest computation time
`'next'`	Next neighbor interpolation. The interpolated value at a query point is the value at the next sample grid point.	Discontinuous	• Requires at least 2 points • Similar memory requirements and computation time as `'nearest'`
`'previous'`	Previous neighbor interpolation. The interpolated value at a query point is the value at the previous sample grid point.	Discontinuous	• Requires at least 2 points • Similar memory requirements and computation time as `'nearest'`

Method	Description	Continuity	Comments
`'pchip'` or `'cubic'`	Shape-preserving piecewise cubic interpolation. The interpolated value at a query point is based on a shape-preserving piecewise cubic interpolation of the values at neighboring grid points.	C^1	• Requires at least 4 points • Requires more memory and computation time than `'linear'`
`'v5cubic'`	Cubic convolution used in MATLAB 5.	C^1	Points must be uniformly spaced.
`'makima'`	Modified Akima cubic Hermite interpolation. The interpolated value at a query point is based on a piecewise function of polynomials with a degree of at most three. The Akima formula is modified to avoid overshoots.	C^1	• Requires at least 2 points • Produces fewer undulations than `'spline'`, but does not flatten as aggressively as `'pchip'` • Computation is more expensive than `'pchip'`, but typically less than `'spline'` • Memory requirements are similar to those of `'spline'`
`'spline'`	Spline interpolation using not-a-knot end conditions. The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension.	C^2	• Requires at least 4 points • Requires more memory and computation time than `'pchip'`

Output Arguments

y — Pulse train

vector

Pulse train generated by the function, returned as a vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The pulse function can only be `tripuls`, `rectpuls`, or `gauspuls`.
- If you specify a custom pulse instead of a pulse function, then the limitations for `interp1` apply.

See Also

`chirp` | `cos` | `diric` | `gauspuls` | `rectpuls` | `sawtooth` | `sin` | `sinc` | `square` | `tripuls`

Introduced before R2006a

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, ___ ,trace)

[ ___ ,pxxc] = pwelch( ___ ,'ConfidenceLevel',probability)

[ ___ ] = pwelch( ___ ,spectrumtype)

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 segments to obtain as close to but not exceed 8 segments with 50% overlap. Each segment 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 segments with 50% overlap, `x` is truncated accordingly.

`pxx = pwelch(x,window)` uses the input vector or integer, `window`, to divide the signal into segments. If `window` is a vector, `pwelch` divides the signal into segments equal in length to the length of `window`. The modified periodograms are computed using the signal segments multiplied by the vector, `window`. If `window` is an integer, the signal is divided into segments 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 segment to segment. `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,π] if `nfft` is even and [0,π) if `nfft` is odd. If `pxx` is a two-sided PSD estimate, `w` spans the interval [0,2π).

`[pxx,f] = pwelch( ___ ,fs)` returns a frequency vector, `f`, in cycles per unit time. The sample rate, `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`). `fs` must be the fifth input to `pwelch`. To input a sample rate and still use the default values of the preceding optional arguments, specify these arguments as empty, `[]`.

`[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 two elements, because otherwise the function interprets it as `nfft`.

`[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 two elements, because otherwise the function interprets it as `nfft`. The frequencies in `f` are in cycles per unit time. The sample rate, `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, ___ ,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` × 100% confidence intervals for the PSD estimate in `pxxc`.

`[ ___ ] = pwelch( ___ ,spectrumtype)` returns the PSD estimate if `spectrumtype` is specified as `'psd'` and returns the power spectrum if `spectrumtype` is specified as `'power'`.

`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 has a length $N_x = 320$ samples.

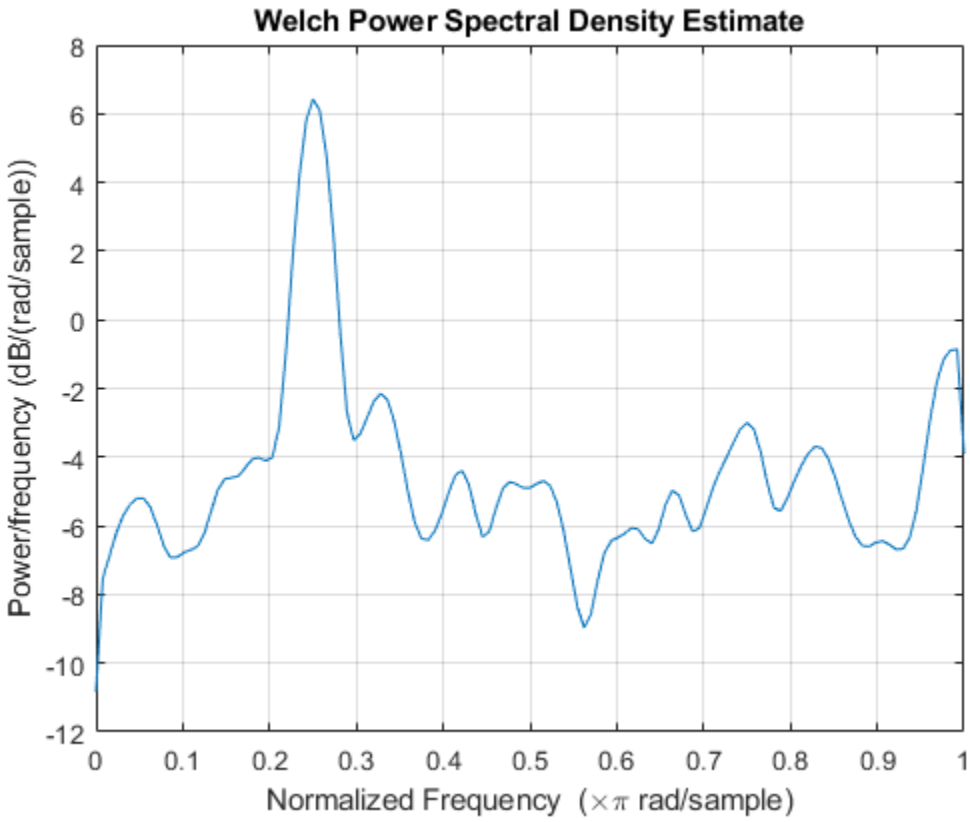
```
rng default
```

```
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. Plot the Welch PSD estimate.

```
pxx = pwelch(x);
```

```
pwelch(x)
```



Repeat the computation.

- Divide the signal into sections of length nsc = $\lfloor N_x/4.5 \rfloor$. This action is equivalent to dividing the signal into the longest possible segments to obtain as close to but not exceed 8 segments with 50% overlap.
- Window the sections using a Hamming window.
- Specify 50% overlap between contiguous sections
- To compute the FFT, use $\max(256, 2^p)$ points, where $p = \lceil \log_2 \text{nsc} \rceil$.

Verify that the two approaches give identical results.

```
Nx = length(x);
nsc = floor(Nx/4.5);
```

```
nov = floor(nsc/2);
nff = max(256,2^nextpow2(nsc));

t = pwelch(x,hamming(nsc),nov,nff);

maxerr = max(abs(abs(t(:))-abs(pxx(:))))
```

```
maxerr = 0
```

Divide the signal into 8 sections of equal length, with 50% overlap between sections. Specify the same FFT length as in the preceding step. Compute the Welch PSD estimate and verify that it gives the same result as the previous two procedures.

```
ns = 8;
ov = 0.5;
lsc = floor(Nx/(ns-(ns-1)*ov));

t = pwelch(x,lsc,floor(ov*lsc),nff);

maxerr = max(abs(abs(t(:))-abs(pxx(:))))
```

```
maxerr = 0
```

Welch Estimate Using Specified Segment Length

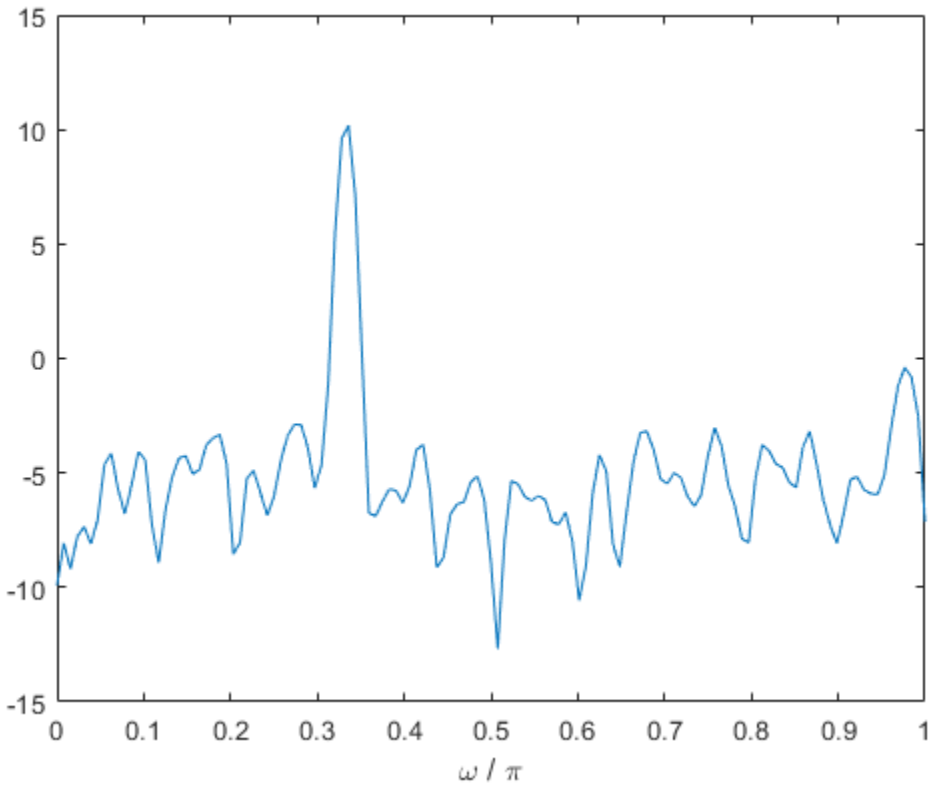
Obtain the Welch PSD estimate of an input signal consisting of a discrete-time sinusoid with an angular frequency of $\pi/3$ rad/sample with additive $N(0, 1)$ white noise.

Create a sine wave with an angular frequency of $\pi/3$ rad/sample with additive $N(0, 1)$ white noise. Reset the random number generator for reproducible results. The signal has 512 samples.

```
rng default

n = 0:511;
x = cos(pi/3*n)+randn(size(n));
```

Obtain the Welch PSD estimate dividing the signal into segments 132 samples in length. The signal segments are multiplied by a Hamming window 132 samples in length. The number of overlapped samples is not specified, so it is set to 132/2 = 66. 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 = 129 points. Plot the PSD as a function of normalized frequency.

```
segmentLength = 132;
[pxx,w] = pwelch(x,segmentLength);

plot(w/pi,10*log10(pxx))
xlabel('\omega / \pi')
```



Welch Estimate Specifying Segment Overlap

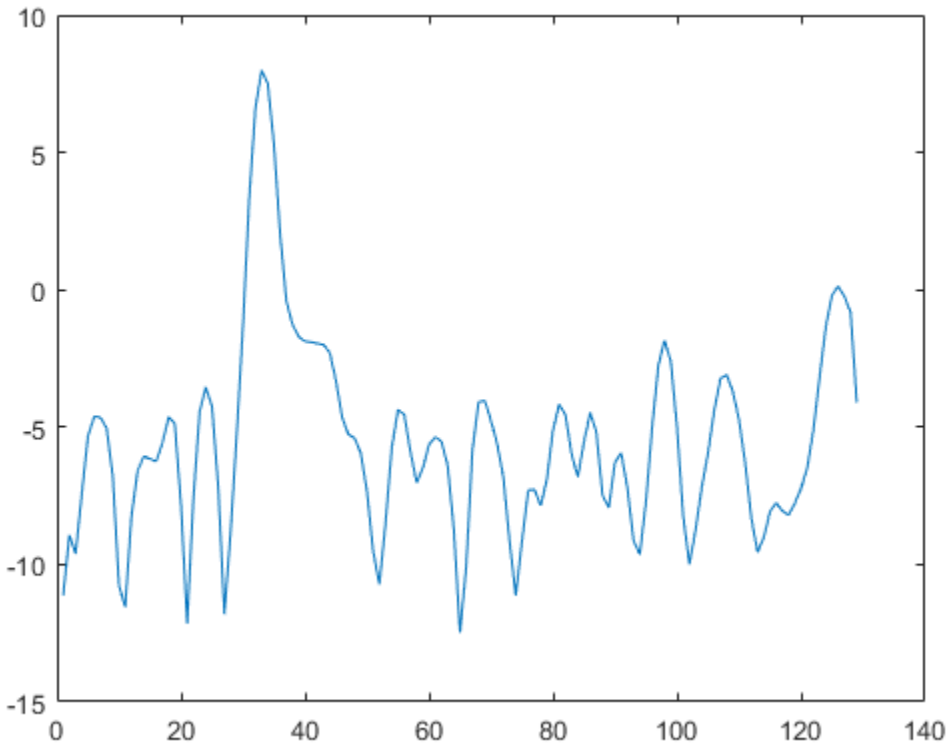
Obtain the Welch PSD estimate of an input signal consisting of a discrete-time sinusoid with an angular frequency of $\pi/4$ rad/sample with additive $N(0, 1)$ white noise.

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

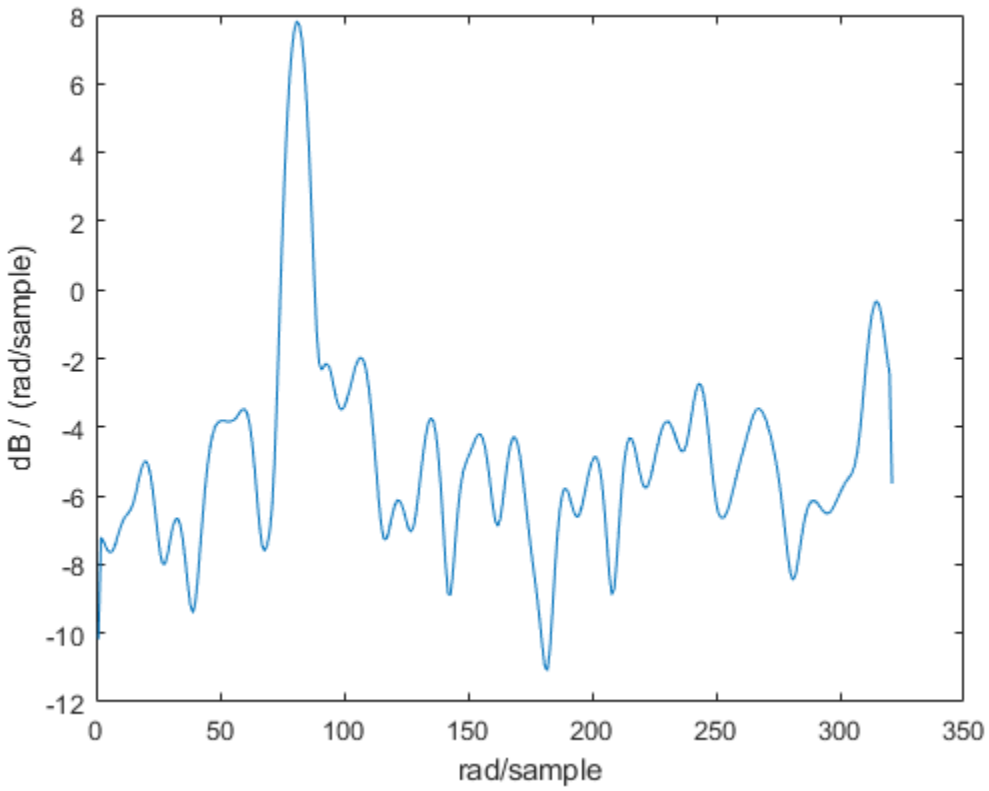
Obtain the Welch PSD estimate of an input signal consisting of a discrete-time sinusoid with an angular frequency of $\pi/4$ rad/sample with additive $N(0, 1)$ white noise.

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('rad/sample')
ylabel('dB / (rad/sample)')
```



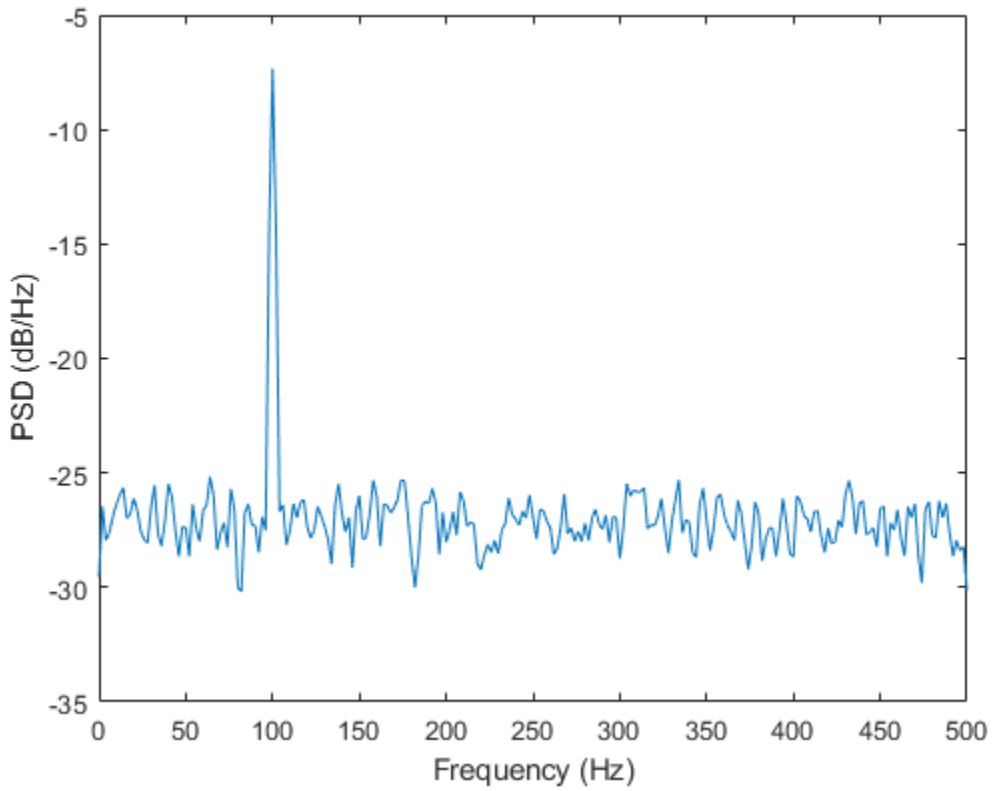
Welch PSD Estimate of Signal with Frequency in Hertz

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('PSD (dB/Hz)')
```



Maximum-Hold and Minimum-Hold Spectra

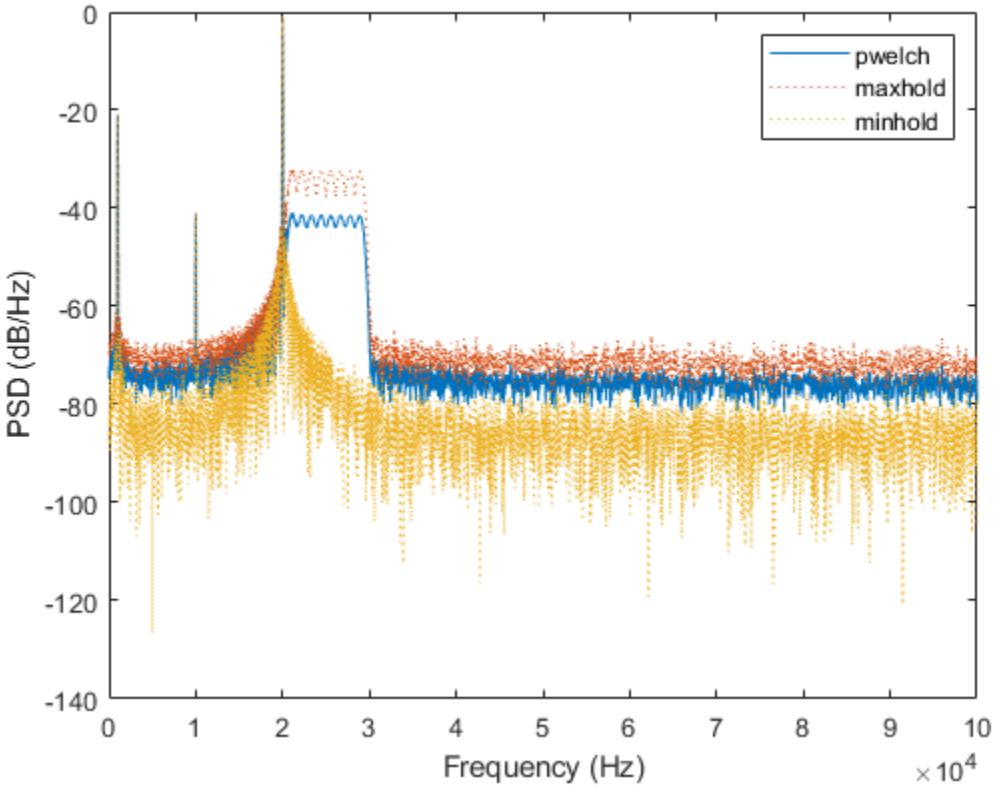
Create a signal consisting of three noisy sinusoids and a chirp, sampled at 200 kHz for 0.1 second. The frequencies of the sinusoids are 1 kHz, 10 kHz, and 20 kHz. The sinusoids have different amplitudes and noise levels. The noiseless chirp has a frequency that starts at 20 kHz and increases linearly 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.

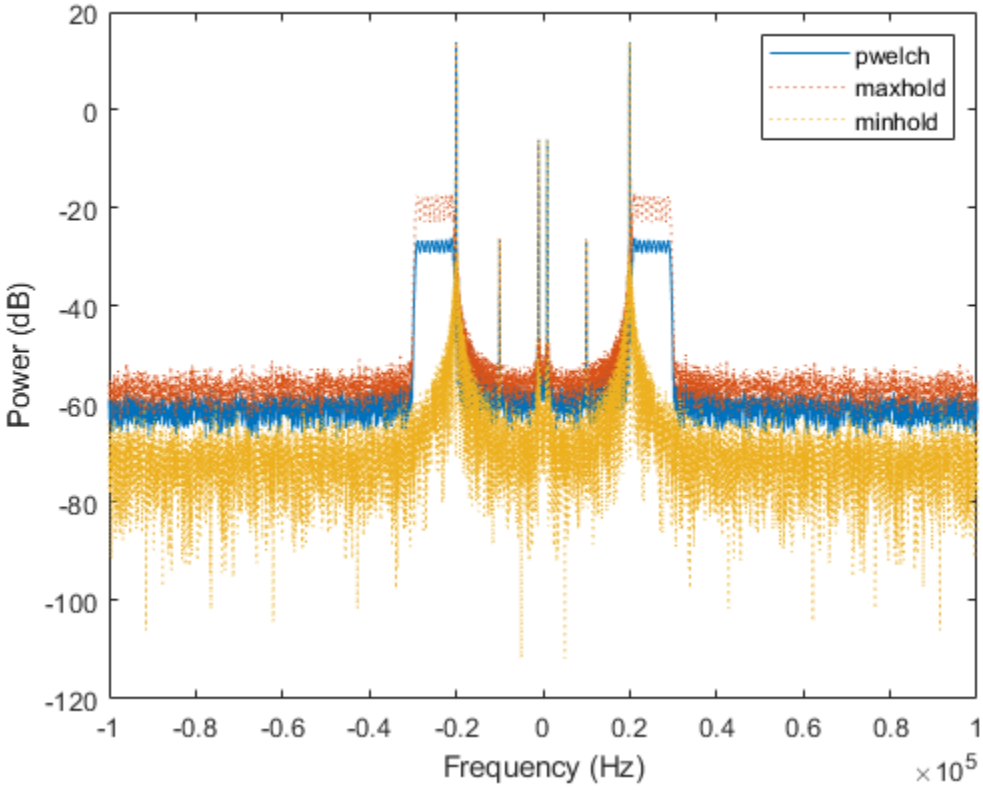
```
[pxx,f] = pwelch(x,[],[],[],Fs);
pmax = pwelch(x,[],[],[],Fs,'maxhold');
pmin = pwelch(x,[],[],[],Fs,'minhold');

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.

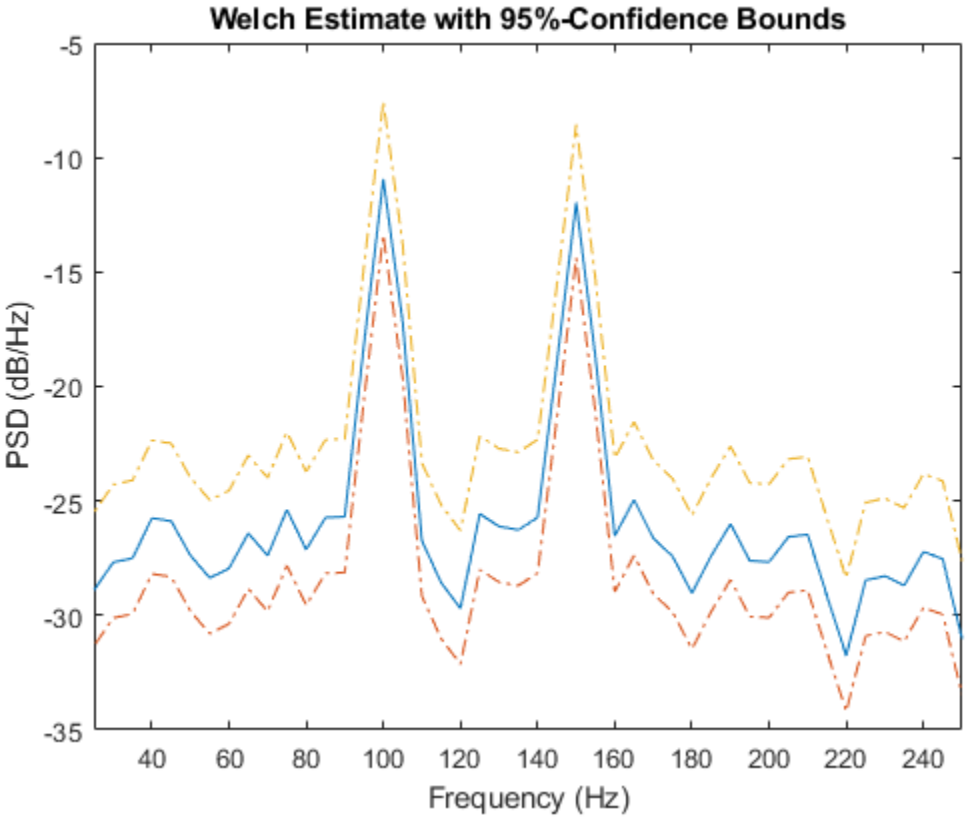
Create a signal consisting of the superposition of 100 Hz and 150 Hz sine waves in additive white $N(0,1)$ noise. The amplitude of the two sine waves is 1. The sample rate is 1 kHz. Reset the random number generator for reproducible results.

```
rng default
fs = 1000;
t = 0:1/fs:1-1/fs;
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),'-.')
hold off

xlim([25 250])
xlabel('Frequency (Hz)')
ylabel('PSD (dB/Hz)')
title('Welch Estimate with 95%-Confidence Bounds')
```



The lower confidence bound in the immediate vicinity of 100 and 150 Hz is significantly above the upper confidence bound outside the vicinity of 100 and 150 Hz.

DC-Centered Power Spectrum

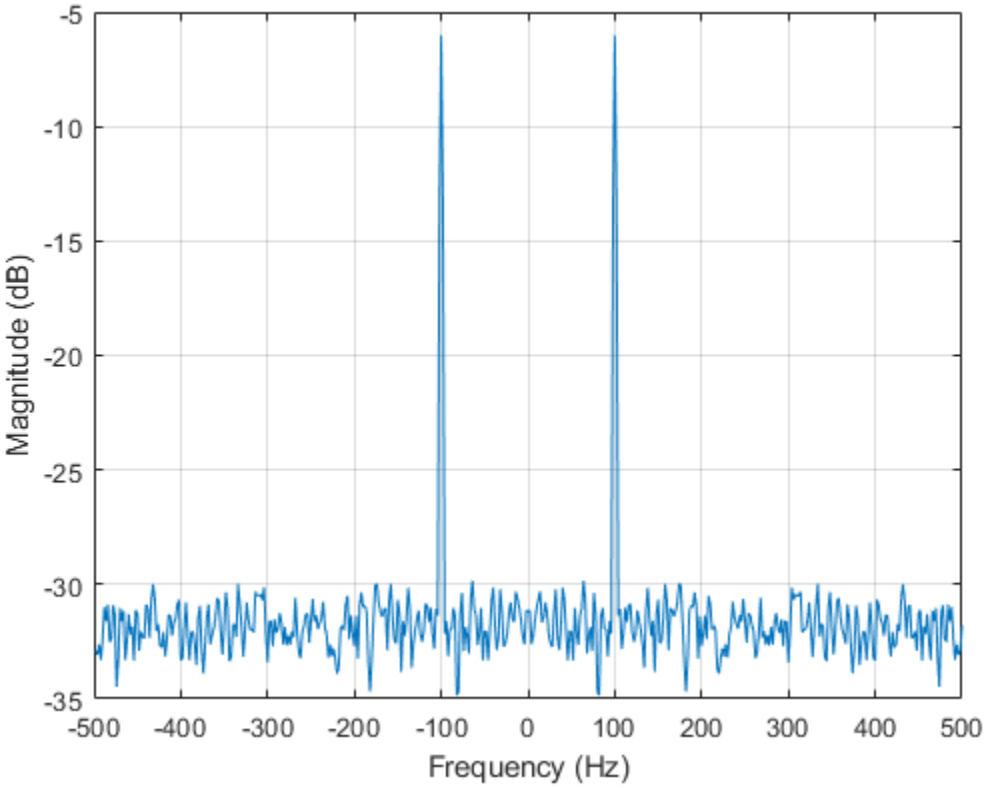
Create a signal consisting of a 100 Hz sinusoid in additive $N(0, 1/4)$ white noise. Reset the random number generator for reproducible results. The sample rate is 1 kHz and the signal is 5 seconds in duration.

```
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
```



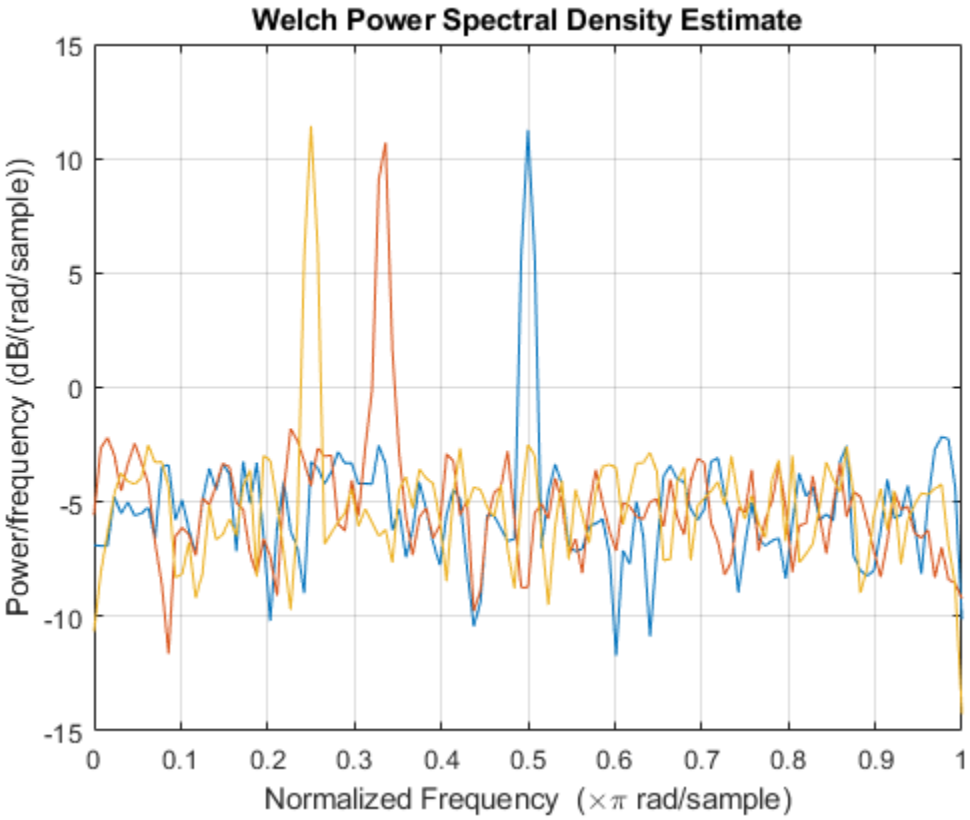
You see that the power at -100 and 100 Hz is close to the expected power of 1/4 for a real-valued sine wave with an amplitude of 1. The deviation from 1/4 is due to the effect of the additive noise.

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)
```



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 segments of length equal to the length of `window`, and then multiplies each signal segment with the vector specified in `window`. If `window` is an integer, `pwelch` is divided into

segments 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 segments 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 segments 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` — 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 has units of Hz.

`w` — Normalized frequencies
vector

Normalized frequencies, specified as a row or column vector with at least two elements. Normalized frequencies are in rad/sample.

Example: `w = [pi/4 pi/2]`

Data Types: `double`

`f` — Frequencies
vector

Frequencies, specified as a row or column vector with at least two elements. The frequencies are in cycles per unit time. The unit time is specified by the sample rate, `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,π] rad/sample. If `nfft` is odd, the length of `pxx` is (`nfft` + 1)/2 and the interval is [0,π) 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.

  The function multiplies the power by 2 at all frequencies except 0 and the Nyquist frequency to conserve the total power.
- `'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 [0,2π) 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 (–π,π] rad/sample for even length `nfft` and (–π,π) 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.

`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.

`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 `probability` × 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 Ω and specify the sample rate 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 $(-\pi,\pi]$ for even `nfft` and $(-\pi,\pi)$ for odd `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` | `single`

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

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.

References

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

[2] Stoica, Petre, and Randolph Moses. *Spectral Analysis of Signals*. Upper Saddle River, NJ: Prentice Hall, 2005.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The input `x` must not be a tall row vector
- The `window` argument must always be specified.

For more information, see "Tall Arrays".

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Thread-Based Environment

Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

Usage notes and limitations:

- The syntax with no output arguments is not supported.

For more information, see "Run MATLAB Functions in Thread-Based Environment".

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

See Also

Apps

Signal Analyzer

Functions
`periodogram` | `pmtm` | `pspectrum`

Topics
“Bias and Variability in the Periodogram”
“Spectral Analysis”

Introduced before R2006a

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`. If you omit `nfft`, or specify it as empty, then `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 sample rate, `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, because otherwise the function interprets it as `nfft`.

`[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, because otherwise the

function interprets it as `nfft`. The frequencies in `f` are in cycles per unit time. The sample rate, `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.

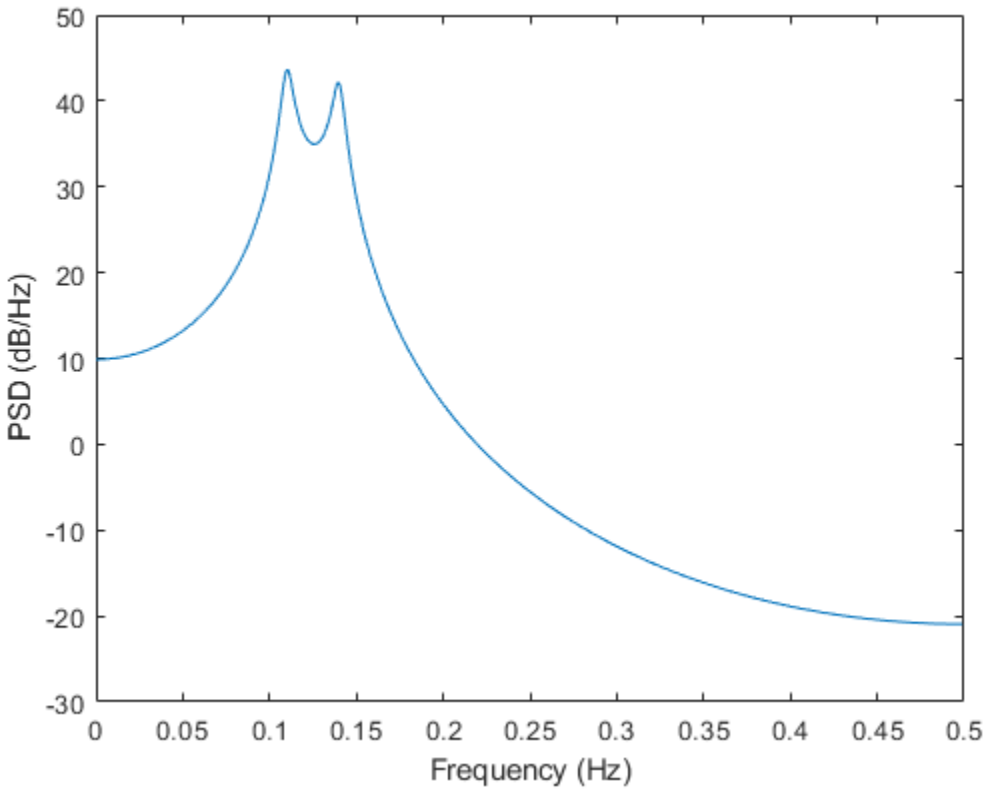
Examples

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.

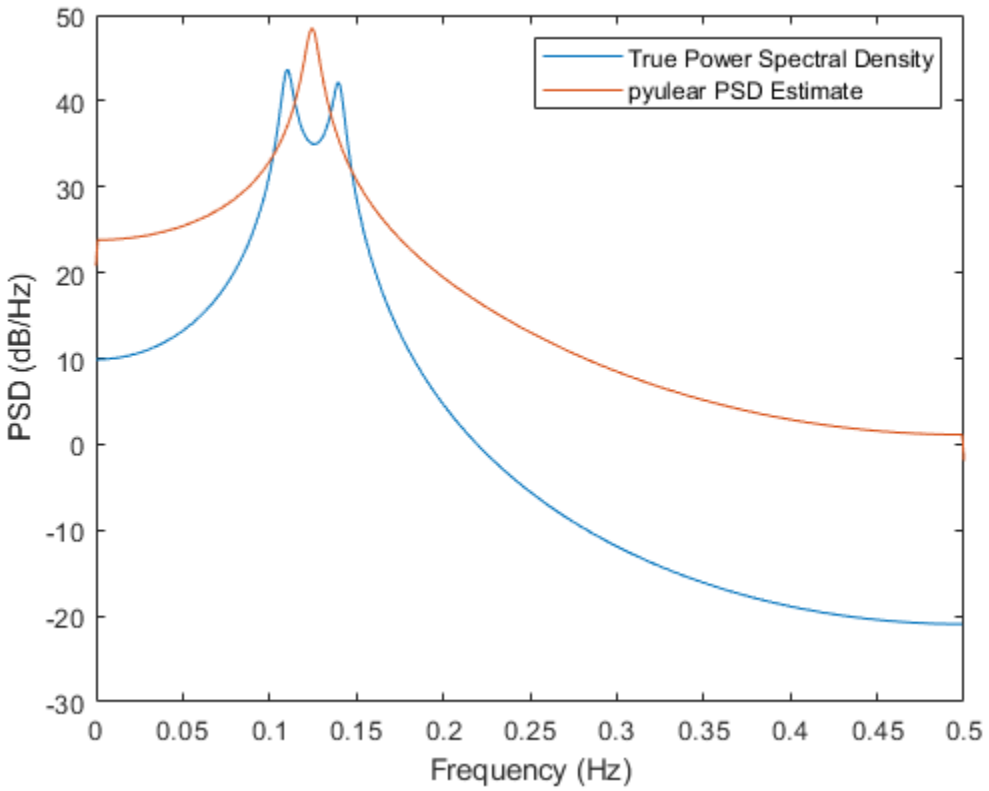
Create an AR(4) system function. Obtain the frequency response and plot the PSD of the system.

```
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')
```



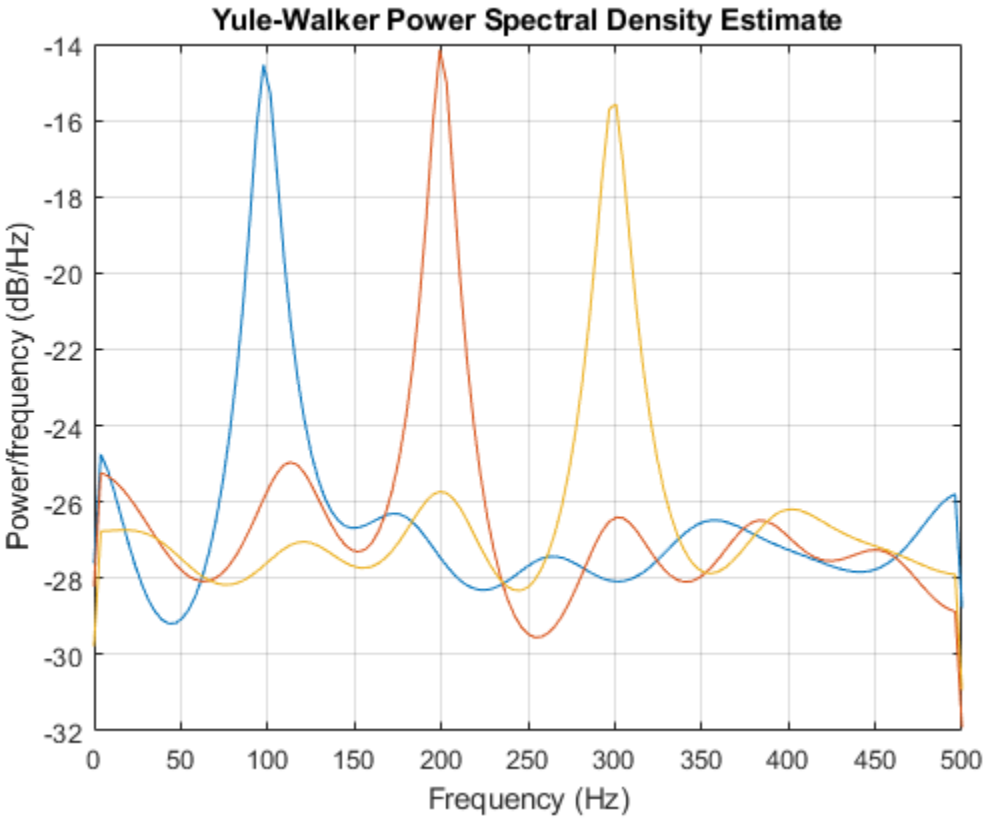
Yule-Walker PSD Estimate of a Multichannel Signal

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` — 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 has units of Hz.

`w` — Normalized frequencies

vector

Normalized frequencies, specified as a row or column vector with at least two elements. Normalized frequencies are in rad/sample.

Example: `w = [pi/4 pi/2]`

Data Types: `double`

`f` — Frequencies

vector

Frequencies, specified as a row or column vector with at least two elements. The frequencies are in cycles per unit time. The unit time is specified by the sample rate, `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,π] rad/sample. If `nfft` is odd, the length of `pxx` is (`nfft` + 1)/2 and the interval is [0,π) 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.

  The function multiplies the power by 2 at all frequencies except 0 and the Nyquist frequency to conserve the total power.
- `'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 [0,2π) 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 (–π,π] rad/sample for even length `nfft` and (–π,π) 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.

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 × 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 Ω and specify the sample rate 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,π] if nfft is even and [0,π) if nfft is odd. If pxx is a two-sided PSD estimate, w spans the interval [0,2π). For a DC-centered PSD estimate, w spans the interval (–π,π] for even nfft and (–π,π) for odd 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 | single

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If the `nfft` argument is variable-size at compile time, then it must not become a scalar or an empty array at runtime.

See Also

`pburg` | `pcov` | `pmcov`

Introduced before R2006a

rainflow

Rainflow counts for fatigue analysis

Syntax

```
c = rainflow(x)
c = rainflow(x,fs)
c = rainflow(x,t)
c = rainflow(xt)

c = rainflow( ___ ,'ext')

[c,rm,rmr,rmm] = rainflow( ___ )
[c,rm,rmr,rmm,idx] = rainflow( ___ )

rainflow( ___ )
```

Description

`c = rainflow(x)` returns cycle counts for the load time history, `x`, according to the ASTM E 1049 standard. See “Algorithms” on page 1-1840 for more information.

`c = rainflow(x,fs)` returns cycle counts for `x` sampled at a rate `fs`.

`c = rainflow(x,t)` returns cycle counts for `x` sampled at the time values stored in `t`.

`c = rainflow(xt)` returns cycle counts for the time history stored in the MATLAB timetable `xt`.

`c = rainflow( ___ ,'ext')` specifies the time history as a vector of identified *reversals* (peaks and valleys). `'ext'` can be used with any of the previous syntaxes.

`[c,rm,rmr,rmm] = rainflow( ___ )` outputs a rainflow matrix, `rm`, and two vectors, `rmr` and `rmm`, containing histogram bin edges for the rows and columns of `rm`, respectively.

`[c,rm,rmr,rmm,idx] = rainflow( ___ )` also returns the linear indices of the reversals identified in the input.

`rainflow( ___ )` with no output arguments plots load reversals and a rainflow matrix histogram in the current figure.

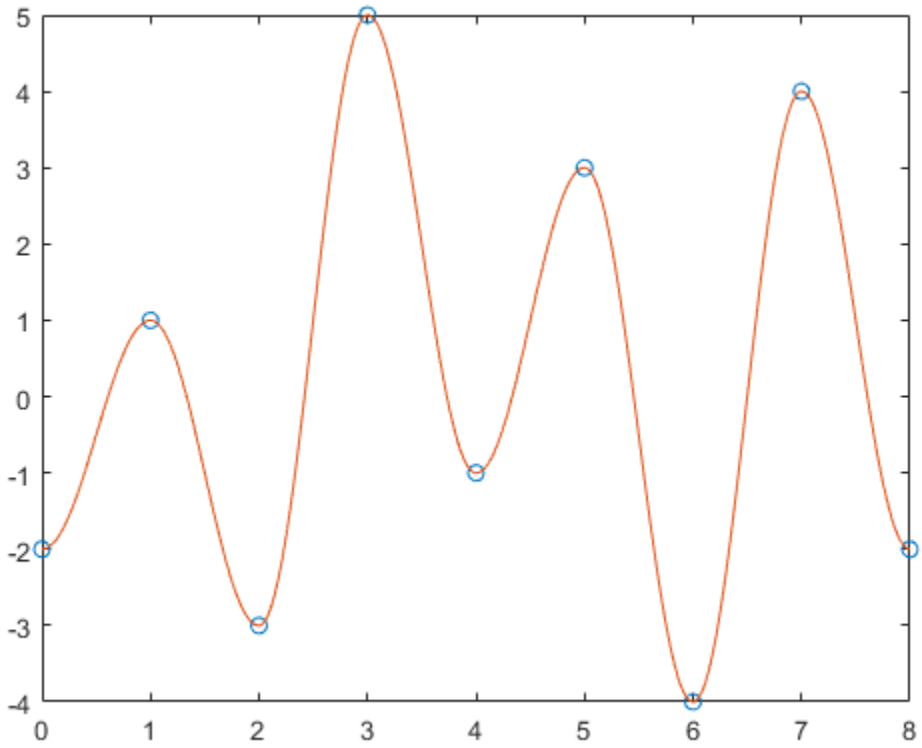
Examples

Cycle Counts with Known Sample Rate

Generate a signal that resembles a load history, consisting of sinusoid half-periods connecting known, equispaced reversals. The signal is sampled at 512 Hz for 8 seconds. Plot the extrema and the signal.

```
fs = 512;

X = [-2 1 -3 5 -1 3 -4 4 -2];
```

```
lX = length(X)-1;

Y = -diff(X)/2.*cos(pi*(0:1/fs:1-1/fs)') + (X(1:lX)+X(2:lX+1))/2;
Y = [Y(:);X(end)];

plot(0:lX,X,'o',0:1/fs:lX,Y)
```



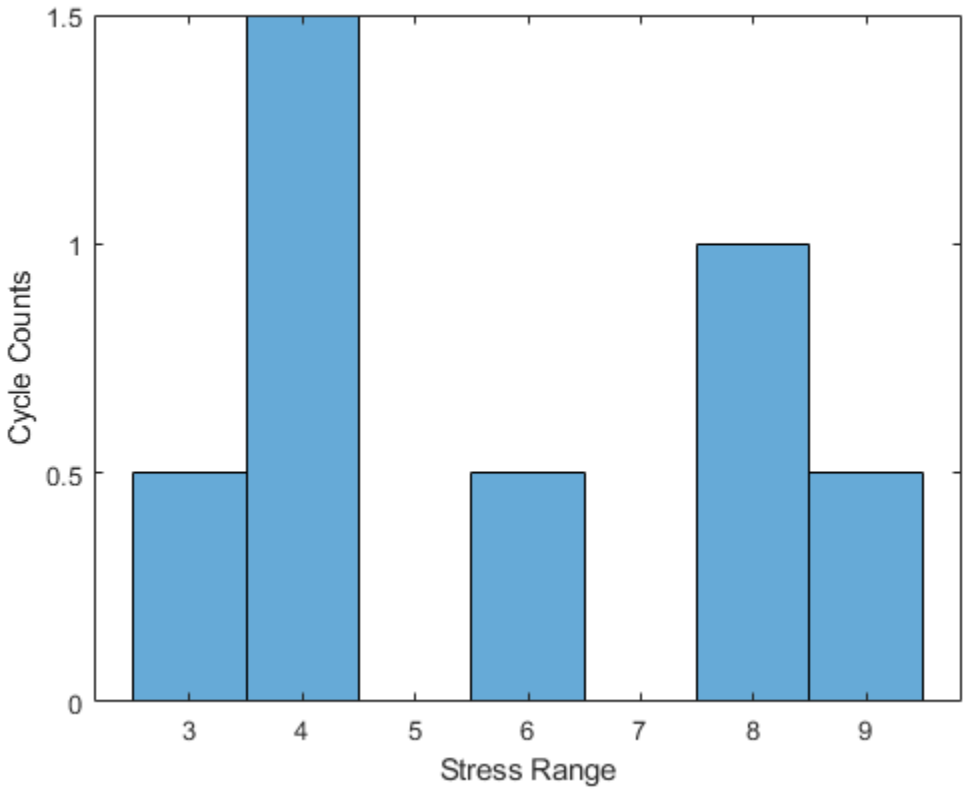
Compute cycle counts for the data. Display the matrix of cycle counts.

```
[c,hist,edges,rmm,idx] = rainflow(Y,fs);

T = array2table(c,'VariableNames',{'Count','Range','Mean','Start','End'})
```

```
T=7×5 table
    Count    Range    Mean    Start    End
    _____    _____    ____    _____    ___

     0.5        3    -0.5        0      1
     0.5        4      -1        1      2
       1        4       1        4      5
     0.5        8       1        2      3
     0.5        9     0.5        3      6
     0.5        8       0        6      7
     0.5        6       1        7      8
```

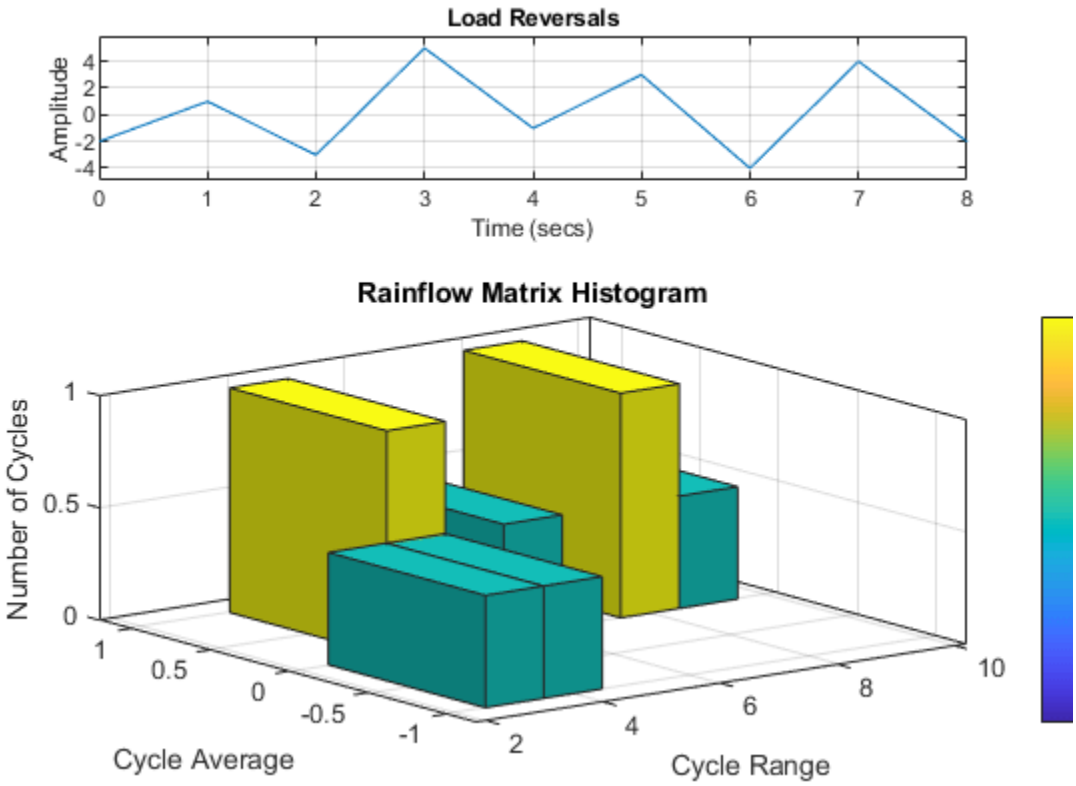
Display a histogram of cycle counts as a function of stress range.

```
histogram('BinEdges',edges','BinCounts',sum(hist,2))
xlabel('Stress Range')
ylabel('Cycle Counts')
```



Use `rainflow` without output arguments to display a histogram of cycles as a function of cycle average and cycle range.

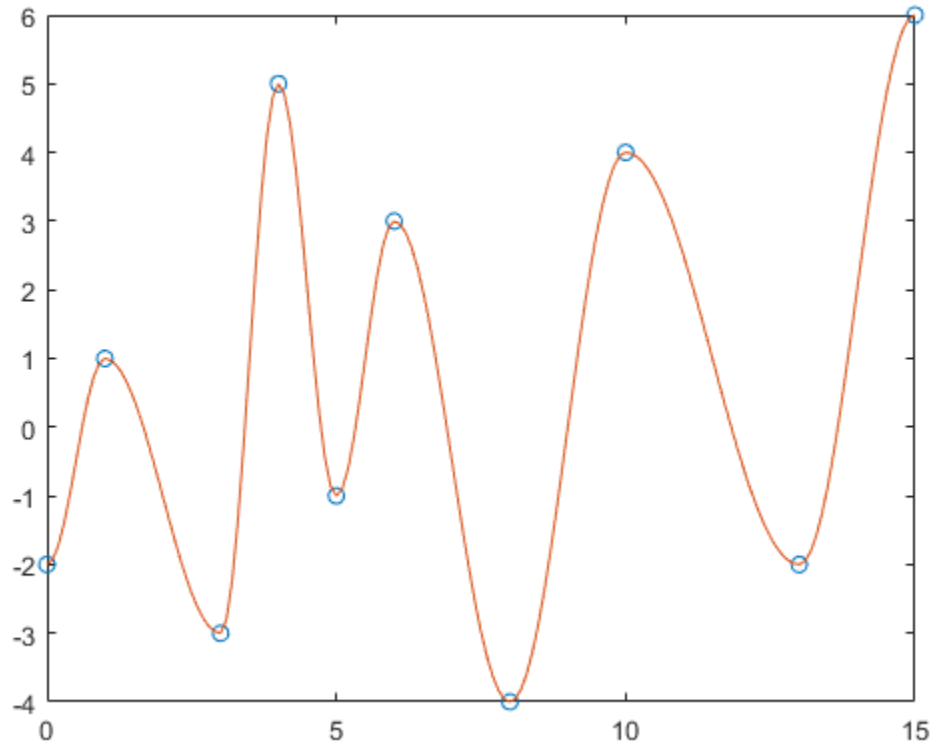
```
rainflow(Y,fs)
```



Cycle Counts with Known Time Values

Generate a signal that resembles a load history, consisting of sinusoid half-periods connecting known, unevenly spaced reversals. The signal is sampled at 10 Hz for 15 seconds. Plot the extrema and the signal.

```
fs = 10;

X = [0 1 3 4 5 6 8 10 13 15];
Y = [-2 1 -3 5 -1 3 -4 4 -2 6];

Z = [];
for k = 1:length(Y)-1
    x = X(k+1)-X(k);
    z = -(Y(k+1)-Y(k))*cos(pi*(0:1/fs:x-1/fs)/x)+Y(k+1)+Y(k);
    Z = [Z z/2];
end
Z = [Z Y(end)];

t = linspace(X(1),X(end),length(Z));
plot(X,Y,'o',t,Z)
```



Compute cycle counts for the data. Display the matrix of cycle counts.

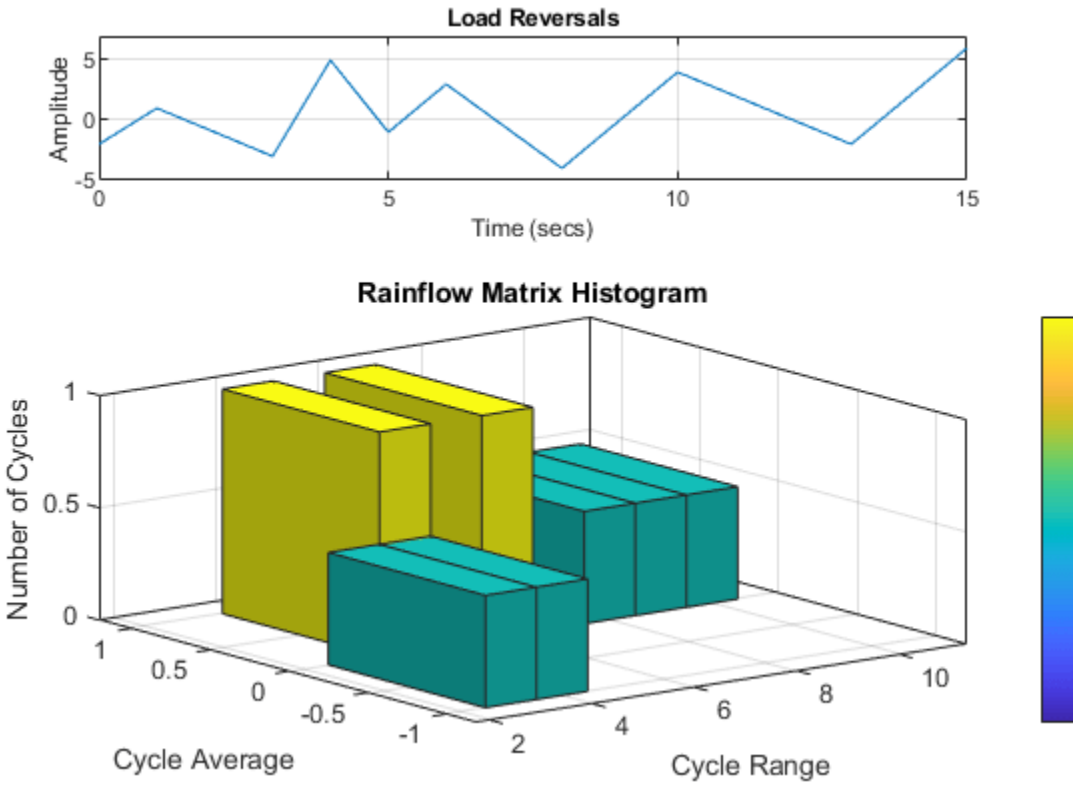
```
[c,hist,edges,rmm,idx] = rainflow(Z,t);
```

```
TT = array2table(c,'VariableNames',{'Count','Range','Mean','Start','End'})
```

```
TT=7×5 table
    Count    Range    Mean    Start    End
    _____    _____    ____    _____    ___

     0.5       3      -0.5      0       1
     0.5       4        -1      1       3
       1       4         1      5       6
     0.5       8         1      3       4
       1       6         1     10      13
     0.5       9       0.5      4       8
     0.5      10         1      8      15
```

Use `rainflow` without output arguments to display a histogram of cycles as a function of cycle average and cycle range.

```
rainflow(Z,t)
```



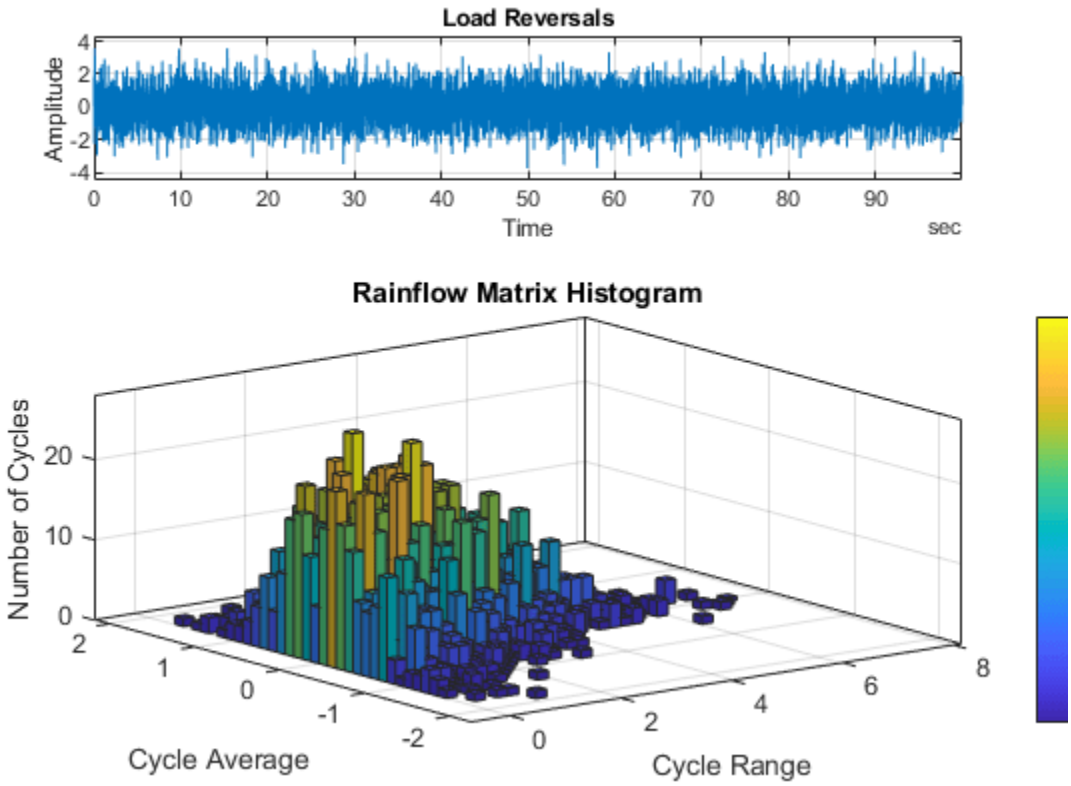
Cycle Counts of Timetable

Generate a random signal sampled at 100 Hz for 100 seconds. Store the signal and its time information in a timetable.

```
fs = 100;
t = seconds(0:1/fs:100-1/fs)';

x = randn(size(t));
TT = timetable(t,x);
```

Display the reversals and the rainflow matrix of the signal.

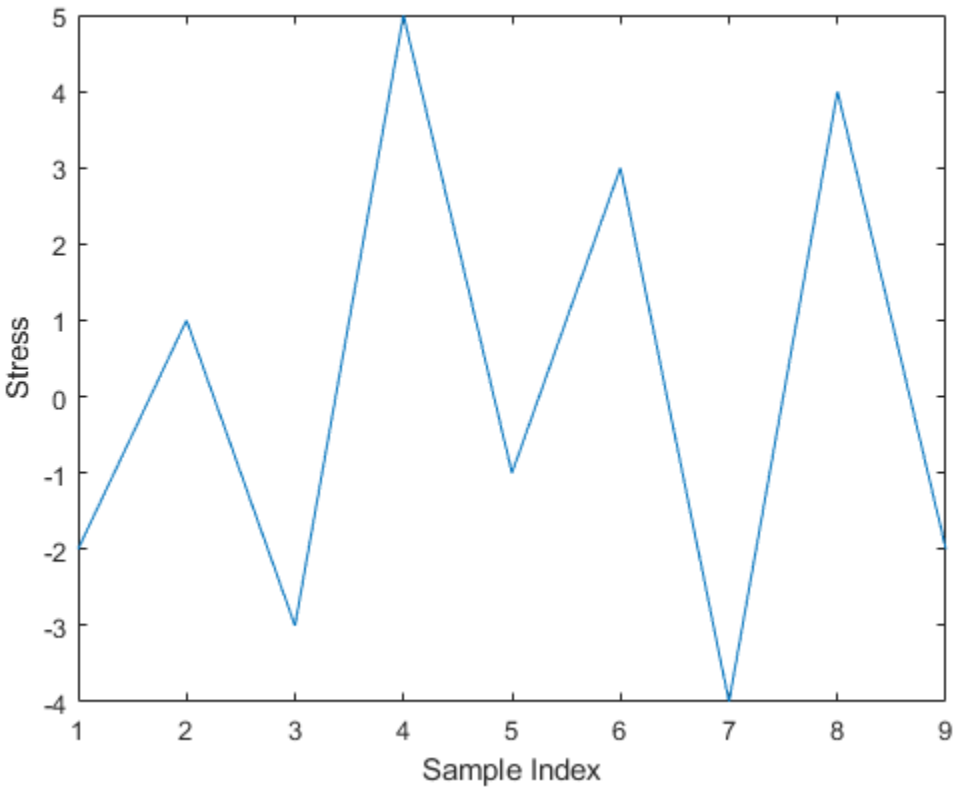
```
rainflow(TT)
```



Cycle Counts of Identified Reversals

Generate a set of extrema resembling load reversals. Plot the data.

```
X = [-2 1 -3 5 -1 3 -4 4 -2]';

plot(X)
xlabel('Sample Index')
ylabel('Stress')
```

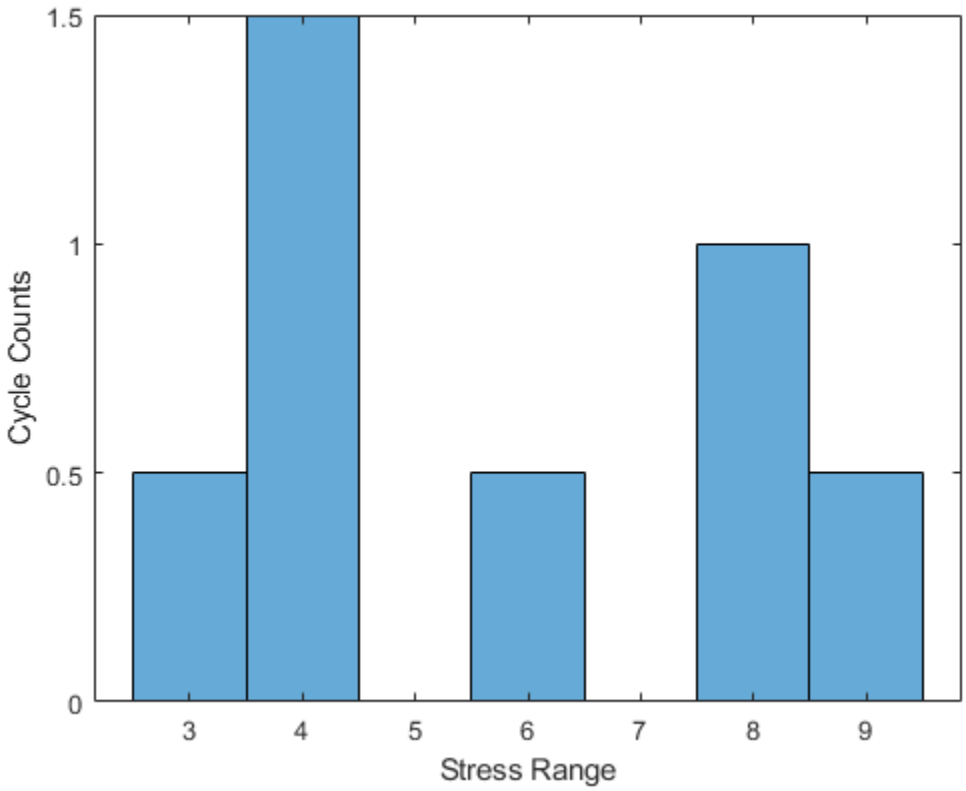


Compute cycle counts for the data. Specify that the input consists of already identified extrema.

```
[C,hist,edges] = rainflow(X,'ext');
```

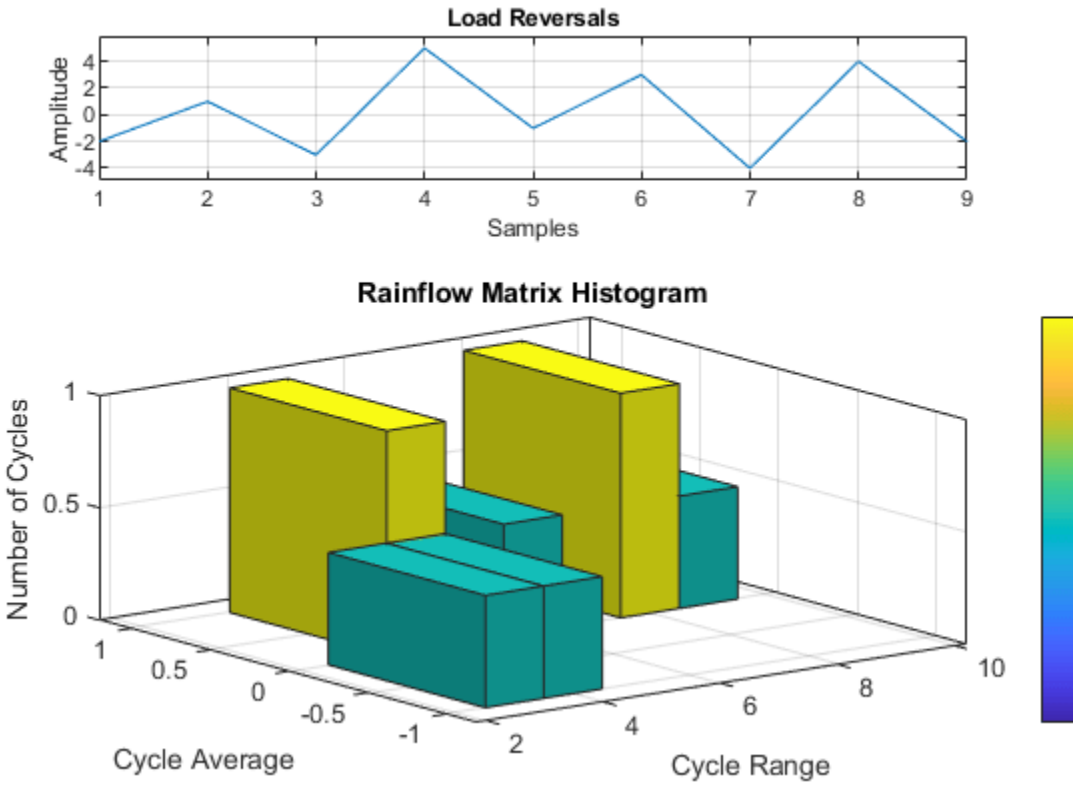
Display a histogram of cycle counts as a function of stress range.

```
histogram('BinEdges',edges','BinCounts',sum(hist,2))
xlabel('Stress Range')
ylabel('Cycle Counts')
```



Use `rainflow` without output arguments to display a histogram of cycles as a function of cycle average and cycle range.

```
rainflow(X,'ext')
```



Input Arguments

x — Load time history

vector

Load time history, specified as a vector. `x` must have finite values.

Data Types: `single` | `double`

fs — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar.

Data Types: `single` | `double`

t — Time values

vector | `duration` array | `duration` scalar

Time values, specified as a vector, a `duration` array, or a `duration` scalar representing the time interval between samples.

Example: `seconds(0:1/100:1)` is a `duration` array representing 1 second of sampling at 100 Hz.

Data Types: `single` | `double` | `duration`

`xt` — Load time history
timetable

Load time history, specified as a timetable. `xt` must contain increasing, finite row times. The timetable must contain only one numeric data vector with finite load values.

If a timetable has missing or duplicate time points, you can fix it using the tips in "Clean Timetable with Missing, Duplicate, or Nonuniform Times".

Example: `timetable(seconds(0:4)',rand(5,1))` specifies a random variable sampled at 1 Hz for 4 seconds.

Data Types: `single` | `double`

Output Arguments

`c` — Cycle counts
matrix

Cycle counts, returned as a matrix. `c` contains cycle information in its columns in this order: counts, ranges, mean values, initial sample indices, and final sample indices. See "Algorithms" on page 1-1840 for an example. If you specify a sample rate, a time interval, or a vector of time values, then the last two columns of `c` contain initial and final cycle times. If you call `rainflow` with a timetable as input, then the last two columns express the initial and final cycle times in seconds.

`rm` — Rainflow matrix
matrix

Rainflow matrix. The rows of `rm` correspond to cycle range, and the columns correspond to cycle mean.

`rmr`, `rmm` — Histogram bin edges
vectors

Histogram bin edges, returned as vectors. `rmr` and `rmm` contain the bin edges of the rows and columns of `rm`, respectively.

`idx` — Linear indices of reversals
vector

Linear indices of reversals, returned as a vector.

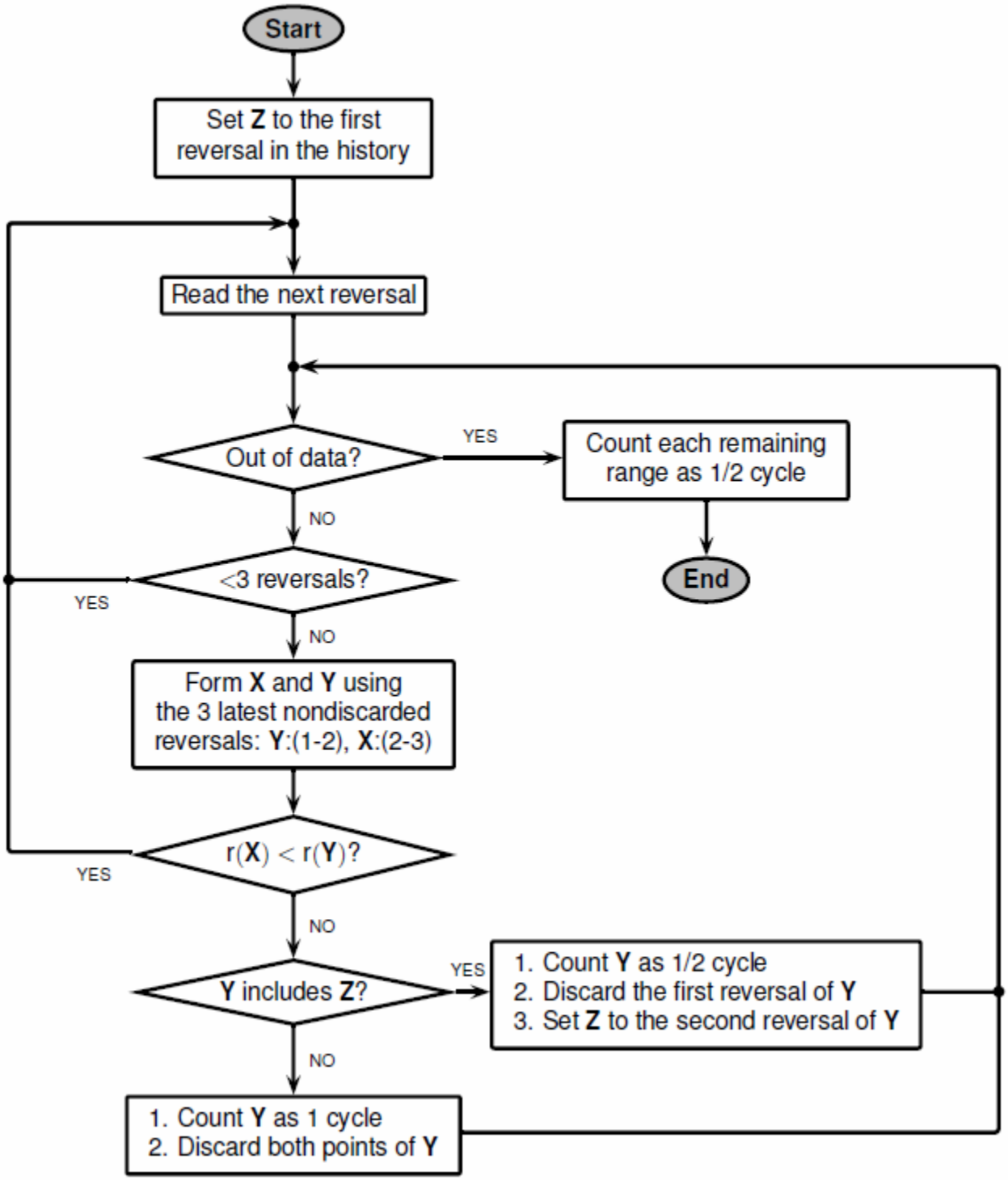
Algorithms

Fatigue analysis studies how damage accumulates in an object subjected to cyclical changes in stress. The number of cycles necessary to break the object depends on the cycle amplitude. Broadband input excitation contains cycles of diverse amplitude, and the presence of hysteresis in the object has the effect of nesting some cycles within others, either completely or partially. *Rainflow counting* estimates the number of load change cycles as a function of cycle amplitude.

Initially, `rainflow` turns the load history into a sequence of *reversals*. Reversals are the local minima and maxima where the load changes sign. The function counts cycles by considering a moving reference point of the sequence, **Z**, and a moving ordered three-point subset with these characteristics:

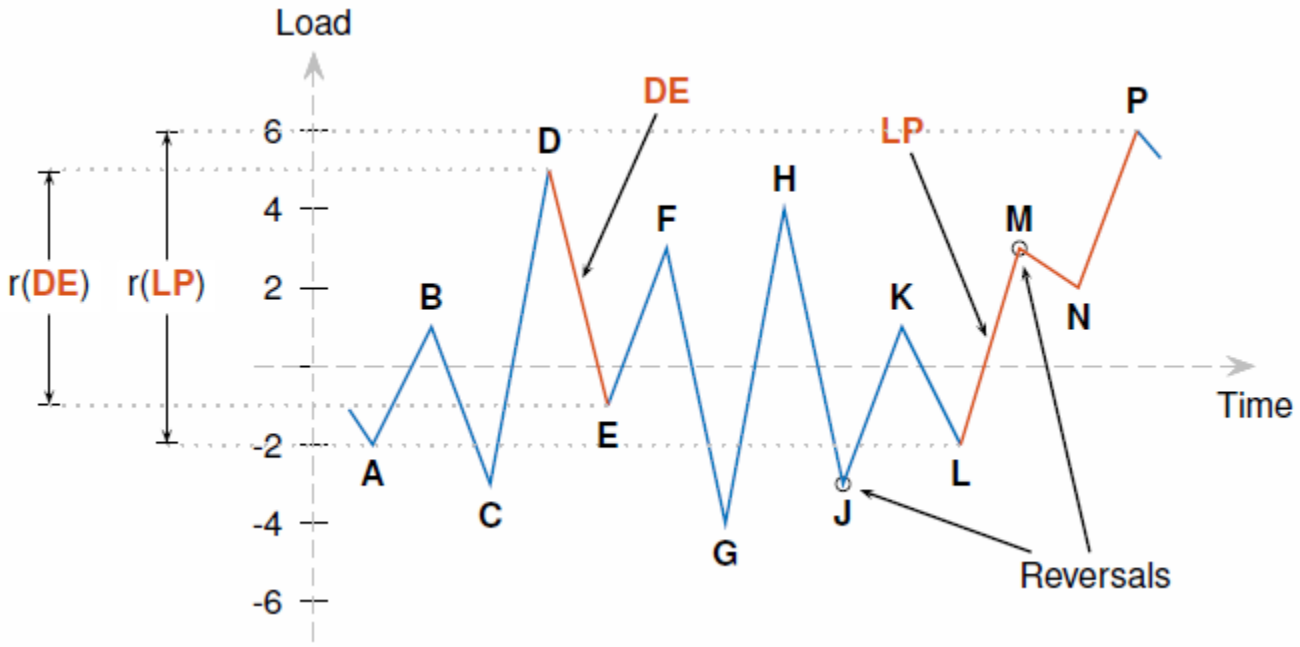
1. The first and second points are collectively called **Y**.
2. The second and third points are collectively called **X**.
3. In both **X** and **Y**, the points are sorted from earlier to later in time, but are not necessarily consecutive in the reversal sequence.
4. The *range* of **X**, denoted by **r**(**X**), is the absolute value of the difference between the amplitude of the first point and the amplitude of the second point. The definition of **r**(**Y**) is analogous.

The `rainflow` algorithm is as follows:



At the end, the function collects the different cycles and half-cycles and tabulates their ranges, their means, and the points at which they start and end. This information can then be used to produce a histogram of cycles.

Consider this reversal sequence:



Step	Z	Reversals	Three Reversals?	Y	r(Y)	X	r(X)	r(X) < r(Y)?	Z in Y?	Actions
1	**A**	**A**, **B**, **C**	Yes	**AB**	3	**BC**	4	No	Yes	**1** Count **AB** as ½ cycle. **2** Discard **A**. **3** Set **Z** to **B**.
2	**B**	**B**, **C**	No	—	—	—	—	—	—	Read **D**.
3	**B**	**B**, **C**, **D**	Yes	**BC**	4	**CD**	8	No	Yes	**1** Count **BC** as ½ cycle. **2** Discard **B**. **3** Set **Z** to **C**.
4	**C**	**C**, **D**	No	—	—	—	—	—	—	Read **E**.
5	**C**	**C**, **D**, **E**	Yes	**CD**	8	**DE**	6	Yes	—	Read **F**.
6	**C**	**C**, **D**, **E**, **F**	Yes	**DE**	6	**EF**	4	Yes	—	Read **G**.
7	**C**	**C**, **D**, **E**, **F**, **G**	Yes	**EF**	4	**FG**	7	No	No	**1** Count **EF** as 1 cycle. **2** Discard **E** and **F**.
8	**C**	**C**, **D**, **G**	Yes	**CD**	8	**DG**	9	No	Yes	**1** Count **CD** as ½ cycle. **2** Discard **C**. **3** Set **Z** to **D**.
9	**D**	**D**, **G**	No	—	—	—	—	—	—	Read **H**.
10	**D**	**D**, **G**, **H**	Yes	**DG**	9	**GH**	8	Yes	—	Read **J**.

Step	Z	Reversals	Three Reversals?	Y	r(Y)	X	r(X)	r(X) < r(Y)?	Z in Y?	Actions
11	**D**	**D**, **G**, **H**, **J**	Yes	**GH**	8	**HJ**	7	Yes	—	Read **K**.
12	**D**	**D**, **G**, **H**, **J**, **K**	Yes	**HJ**	7	**JK**	4	Yes	—	Read **L**.
13	**D**	**D**, **G**, **H**, **J**, **K**, **L**	Yes	**JK**	4	**KL**	3	Yes	—	Read **M**.
14	**D**	**D**, **G**, **H**, **J**, **K**, **L**, **M**	Yes	**KL**	3	**LM**	5	No	No	**1** Count **KL** as 1 cycle. **2** Discard **K** and **L**.
15	**D**	**D**, **G**, **H**, **J**, **M**	Yes	**HJ**	7	**JM**	5	Yes	—	Read **N**.
16	**D**	**D**, **G**, **H**, **J**, **M**, **N**	Yes	**JM**	5	**MN**	1	Yes	—	Read **P**.
17	**D**	**D**, **G**, **H**, **J**, **M**, **N**, **P**	Yes	**MN**	1	**NP**	4	No	No	**1** Count **MN** as 1 cycle. **2** Discard **M** and **N**.
18	**D**	**D**, **G**, **H**, **J**, **P**	Yes	**HJ**	7	**JP**	9	No	No	**1** Count **HJ** as 1 cycle. **2** Discard **H** and **J**.
19	**D**	**D**, **G**, **P**	Yes	**DG**	9	**GP**	10	No	Yes	**1** Count **DG** as ½ cycle. **2** Discard **D**. **3** Set **Z** to **G**.
20	**G**	**G**, **P**	Out of data	—	—	—	—	—	—	Count **GP** as ½ cycle.

Now collect the results.

Cycle Count	Range	Mean	Start	End
½	3	–0.5	**A**	**B**
½	4	–1	**B**	**C**
1	4	1	**E**	**F**
½	8	1	**C**	**D**
1	3	–0.5	**K**	**L**
1	1	2.5	**M**	**N**
1	7	0.5	**H**	**J**
½	9	0.5	**D**	**G**
½	10	1	**G**	**P**

Compare this to the result of running `rainflow` on the sequence:

```
q = rainflow([-2 1 -3 5 -1 3 -4 4 -3 1 -2 3 2 6])
```

```
q =

    0.5000    3.0000   -0.5000    1.0000    2.0000
```

```
    0.5000    4.0000   -1.0000    2.0000    3.0000
    1.0000    4.0000    1.0000    5.0000    6.0000
    0.5000    8.0000    1.0000    3.0000    4.0000
    1.0000    3.0000   -0.5000   10.0000   11.0000
    1.0000    1.0000    2.5000   12.0000   13.0000
    1.0000    7.0000    0.5000    8.0000    9.0000
    0.5000    9.0000    0.5000    4.0000    7.0000
    0.5000   10.0000    1.0000    7.0000   14.0000
```

References

[1] ASTM E1049-85(2017), "Standard Practices for Cycle Counting in Fatigue Analysis." West Conshohocken, PA: ASTM International, 2011, https://www.astm.org/cgi-bin/resolver.cgi?E1049.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Timetables are not supported for code generation.

See Also

`findpeaks` | `histcounts` | `histcounts2`

Introduced in R2017b

realizemdl

Simulink subsystem block for filter

Syntax

```
realizemdl(FiltObject)
realizemdl(FiltObject,Name,Value)
```

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,Name,Value)` generates the model for`FiltObject` with the associated `Name,Value` 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), `'new'` or *`SubsystemName`*	Specify whether to add the block to your current Simulink model or create a new model to contain the block. If you provide the name of a current subsystem in *`SubsystemName`*, `realizemdl` 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, set `propertyvalue` to the name of your choice, enclosed in single quotes.

Property Name	Property Values	Description
`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`	`'on'` (default) or `'off'`	Specify whether to remove zero-gain blocks.
`OptimizeOnes`	`'on'` (default) or `'off'`	Specify whether to replace unity-gain blocks with direct connections.
`OptimizeNegOnes`	`'on'` (default) or `'off'`	Specify whether to replace negative unity-gain blocks with a sign change at the nearest sum block.
`OptimizeDelayChains`	`'on'` (default) or `'off'`	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 a cell array.`MapCoeffsToPorts` 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. The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release.

Property Name	Property Values	Description
`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 `InputProcessing` is `'columnsaschannels'`.

Note `OptimizeZeros`, `OptimizeOnes`, and `OptimizeNegOnes` are `'on'` by default. If you want to map all your coefficients to a port, do one of these:

- Turn off these optimization properties.
- Do not initialize the input filter with zeros, ones, or negative ones.

Examples

Model of Lowpass Butterworth Filter

Create a lowpass Butterworth filter and realize its Simulink® model with coefficients mapped to ports. Call the filter `NewFilter`.

```
[b,a] = butter(4,.25);
d = dfilt.df1(b,a);
realizemdl(d,'MapCoeffsToPorts','on','BlockName','NewFilter')
```

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

Apps
Filter Designer

Functions
`designfilt` | `fdesign` | `filt2block`

Introduced in R2009b

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 = 6×1

    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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

See Also

ac2rc | poly2ac | rc2poly

Introduced before R2006a

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 = 1×5

    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`

Introduced before R2006a

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 = 1×5

    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`

Introduced before R2006a

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 = 1×6

    1.0000    0.6148    0.9899    0.0000    0.0032   -0.0082
```

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`, `a = [a + a(i-1:-1:1)*k(i) k(i)]`.
3. Implements `a = [1 a]`.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

See Also

`ac2poly` | `latc2tf` | `latcfilt` | `poly2rc` | `rc2ac` | `rc2is` | `rc2lar` | `tf2latc`

Introduced before R2006a

rceps

Real cepstrum and minimum-phase reconstruction

Syntax

```
[y,ym] = rceps(x)
```

Description

`[y,ym] = rceps(x)` returns both the real cepstrum `y` and a minimum phase reconstructed version `ym` of the input sequence.

Examples

Echo Cancelation Using the Real Cepstrum

A speech recording includes an echo caused by reflection off a wall. Use the real cepstrum to filter it out.

In the recording, a person says the word MATLAB®. Load the data and the sample rate, F_s = 7418 Hz.

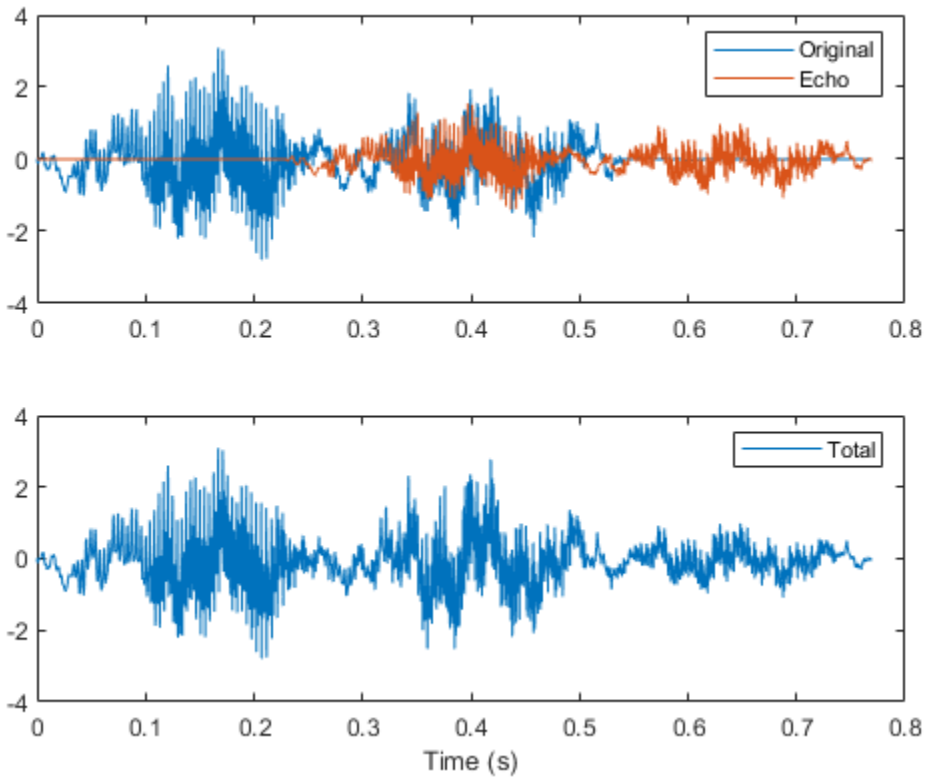
```
load mtlb

% To hear, type soundsc(mtlb,Fs)
```

Model the echo by adding to the recording a copy of the signal delayed by Δ samples and attenuated by a known factor α: $y(n) = x(n) + \alpha x(n - \Delta)$. 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.

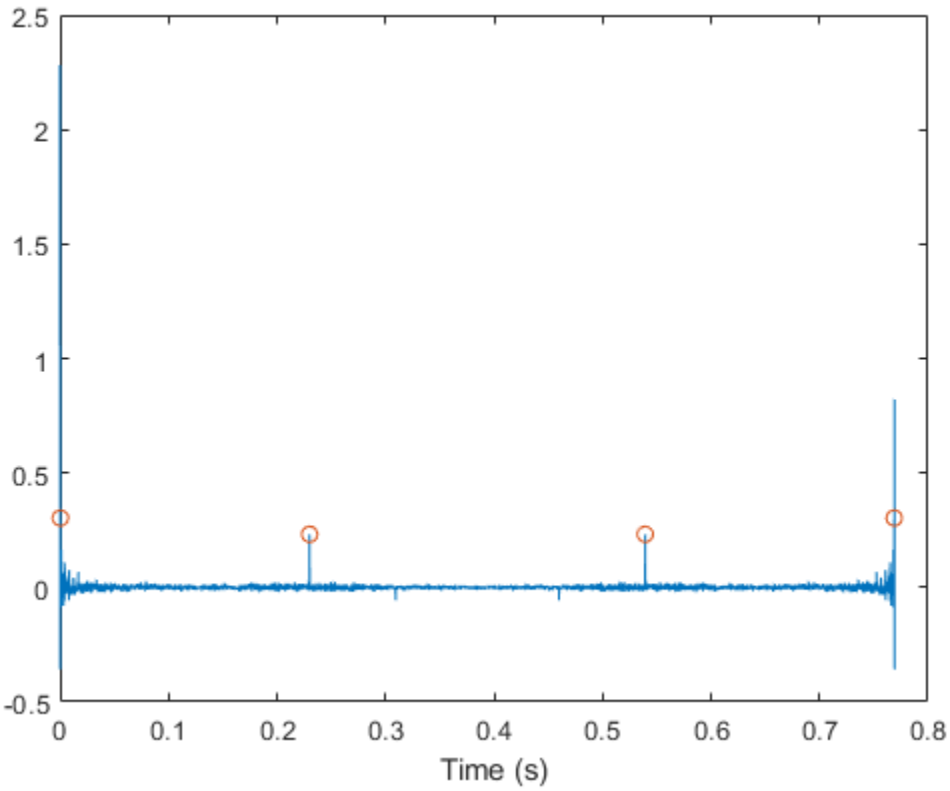
```
t = (0:length(mtEcho)-1)/Fs;

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)
```

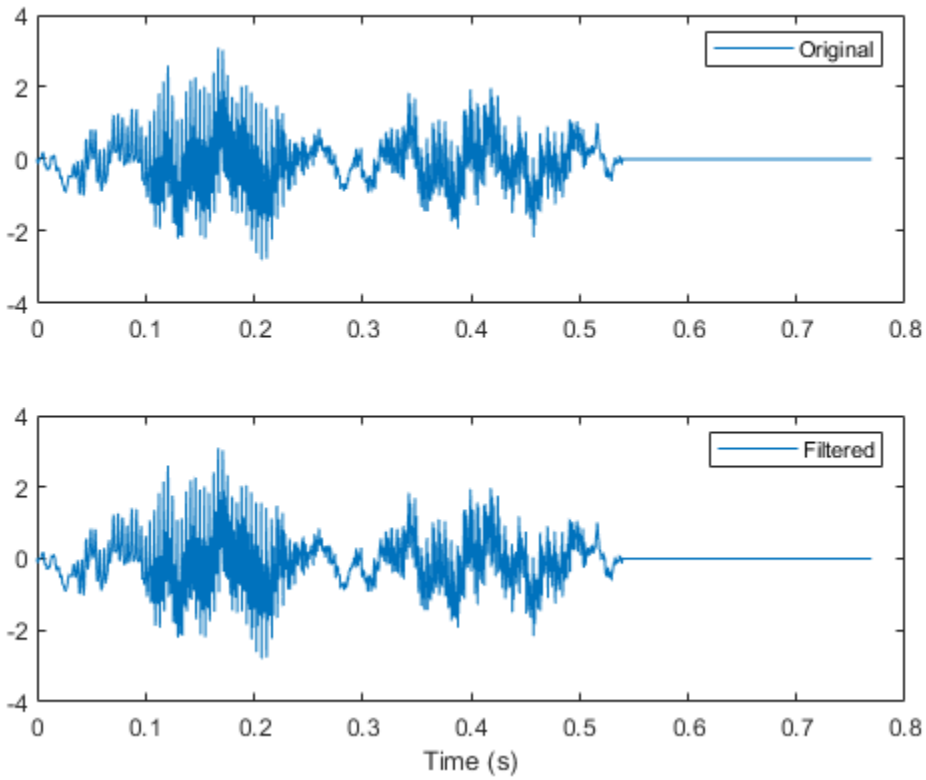
Compute the real cepstrum of the signal. Plot the cepstrum and annotate its maxima. The cepstrum has a sharp peak at the time at which the echo starts to arrive.

```
c = rceps(mtEcho);

[px,locs] = findpeaks(c,'Threshold',0.2,'MinPeakDistance',0.2);

clf
plot(t,c,t(locs),px,'o')
xlabel('Time (s)')
```



Cancel the echo by filtering the signal through an IIR system whose output, w, obeys $w(n) + \alpha w(n - \Delta) = y(n)$. Plot the filtered signal and compare it to the original.

```
dl = locs(2)-1;

mtNew = filter(1,[1 zeros(1,dl-1) alpha],mtEcho);

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)
```

Input Arguments

x — Input signal

real vector

Input signal, specified as a real vector.

Output Arguments

y — Real cepstrum

vector

Real cepstrum, returned as a vector.

ym — Minimum phase real cepstrum

vector

Minimum phase real cepstrum, returned as a vector.

Algorithms

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` 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, Alan V., and Ronald W. Schafer. *Digital Signal Processing*, Englewood Cliffs, NJ, Prentice-Hall, 1975.

[2] *Programs for Digital Signal Processing*, IEEE Press, New York, 1979.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`cceps` | `fft` | `hilbert` | `icceps` | `unwrap`

Introduced before R2006a

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

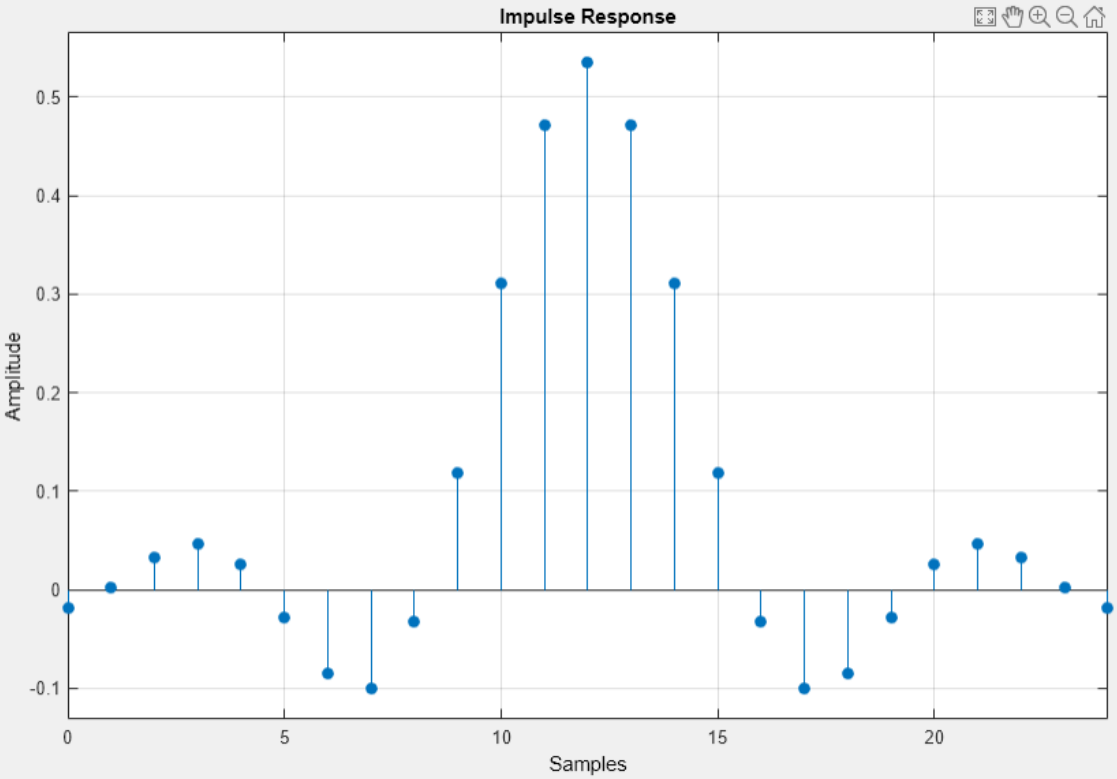
Design a Square-Root Raised Cosine Filter

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')))
```

```
mx = 0
```

```
fvtool(h,'Analysis','impulse')
```

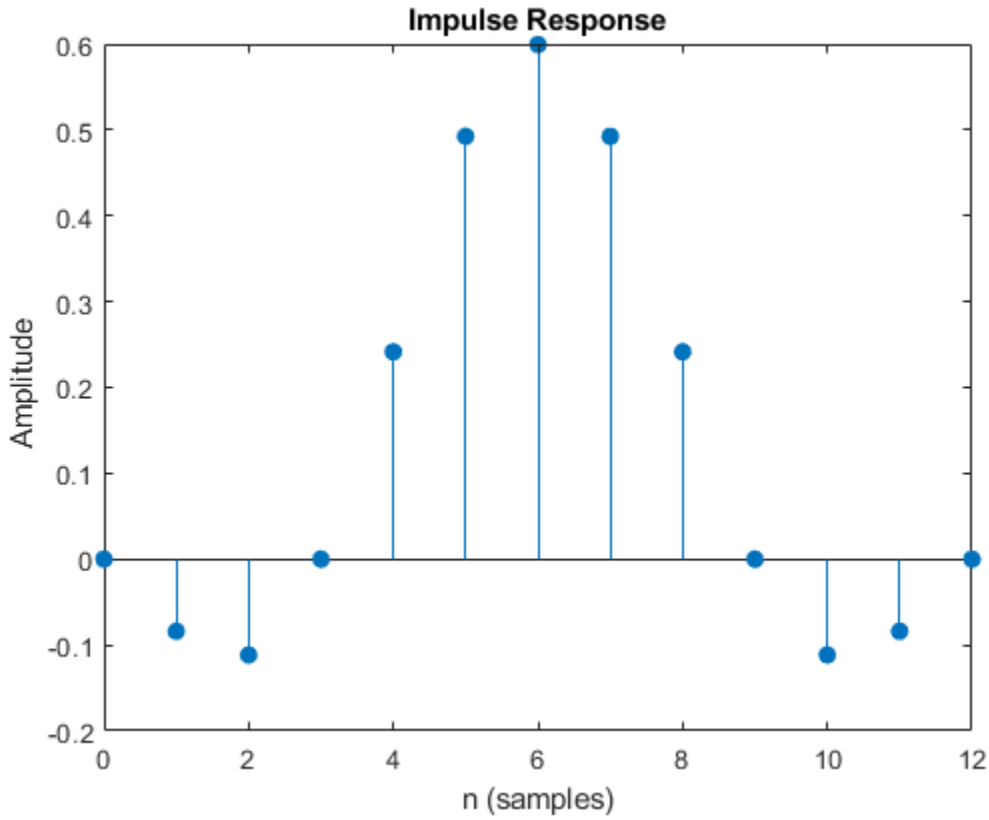


Impulse Responses of Normal and Square-Root Raised Cosine Filters

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.

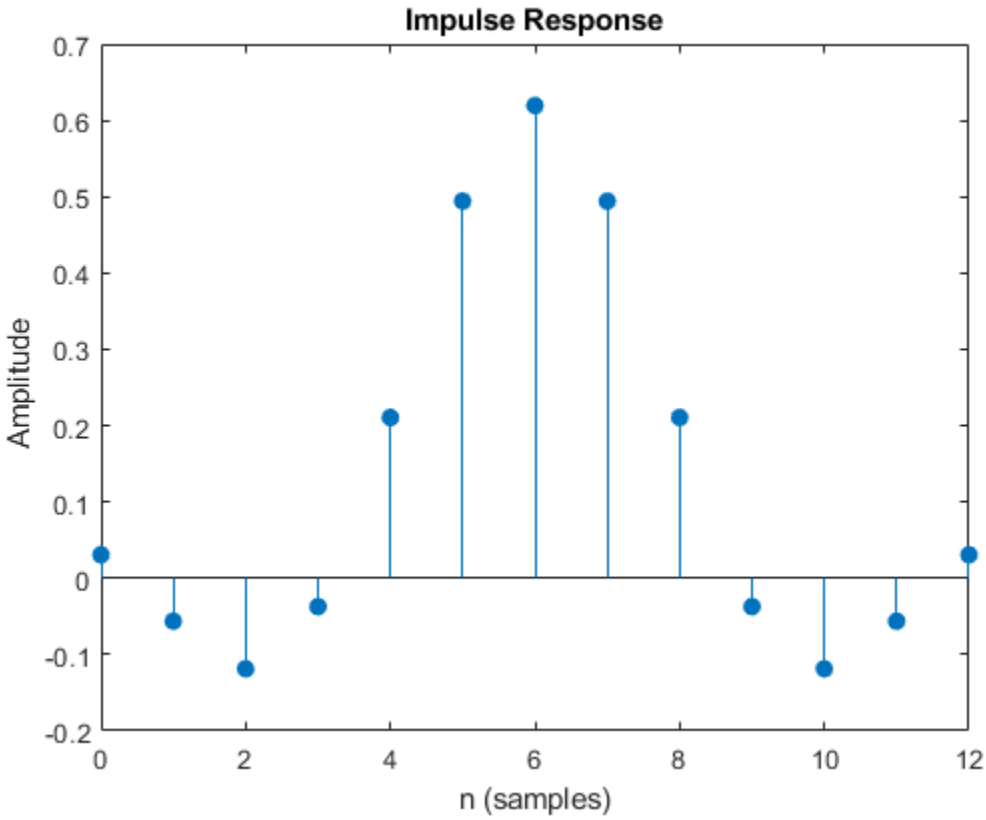
Create a normal raised cosine filter with rolloff 0.25. Specify that this filter span 4 symbols with 3 samples per symbol.

```
rf = 0.25;
span = 4;
sps = 3;

h1 = rcosdesign(rf,span,sps,"normal");
impz(h1)
```



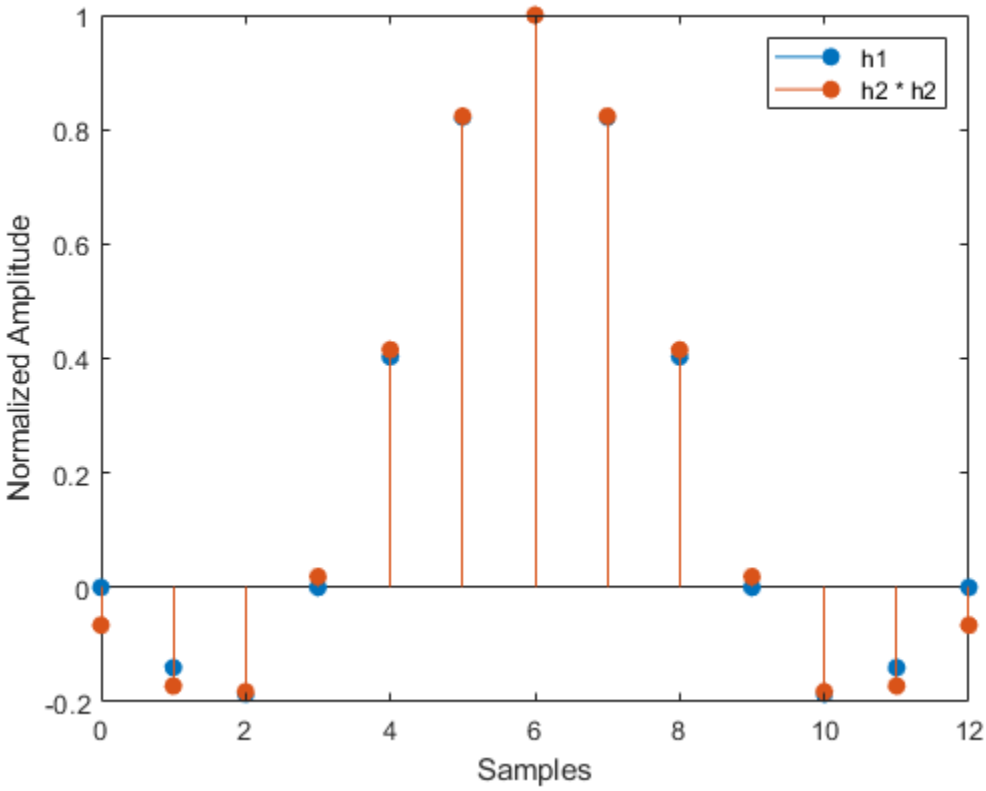
The normal filter has zero crossings at integer multiples of `sps`. It thus satisfies Nyquist's criterion for zero intersymbol interference. The square-root filter, however, does not:

```
h2 = rcosdesign(rf,span,sps,"sqrt");
impz(h2)
```



Convolve the square-root filter with itself. Truncate the impulse response outward from the maximum so it has the same length as `h1`. Normalize the response using the maximum. Compare the convolved square-root filter to the normal filter.

```
h3 = conv(h2,h2,"same");

stem(0:span*sps,[h1/max(abs(h1));h3/max(abs(h3))]',"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);
```

For information on how to use square-root, raised cosine filters to interpolate and decimate signals, see “Interpolate and Decimate Using RRC Filter” (Communications Toolbox).

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 either `'normal'` or `'sqrt'`.

Output Arguments

`b` — FIR filter coefficients

row vector

Raised cosine filter coefficients, returned as a row vector.

Data Types: `double` | `single`

Tips

- If you have a license for Communications 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

`gaussdesign`

Topics

“Interpolate and Decimate Using RRC Filter” (Communications Toolbox)

Introduced in R2013b

rectpuls

Sampled aperiodic rectangle

Syntax

```
y = rectpuls(t)
y = rectpuls(t,w)
```

Description

`y = rectpuls(t)` returns a continuous, aperiodic, unit-height rectangular pulse at the sample times indicated in array `t`, centered about `t = 0`.

`y = rectpuls(t,w)` generates a rectangle of width `w`.

Examples

Generate and Displace Rectangular Pulse

Generate 200 ms of a rectangular pulse with a sample rate of 10 kHz and a width of 20 ms.

```
fs = 10e3;
t = -0.1:1/fs:0.1;

w = 20e-3;

x = rectpuls(t,w);
```

Generate two copies of the same pulse:

- One displaced 45 ms into the past.

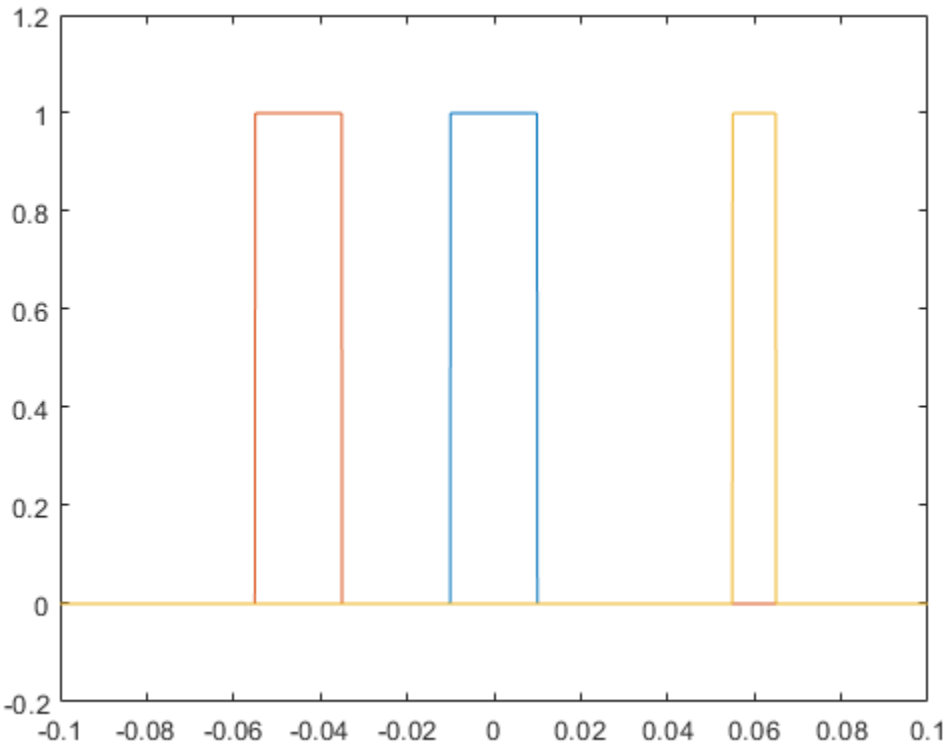
```
tpast = -45e-3;
xpast = rectpuls(t-tpast,w);
```

- One displaced 60 ms into the future and half as wide.

```
tfutr = 60e-3;
xfutr = rectpuls(t-tfutr,w/2);
```

Plot the original pulse and the two copies on the same axes.

```
plot(t,x,t,xpast,t,xfutr)
ylim([-0.2 1.2])
```



Input Arguments

`t` — Sample times

vector

Sample times of unit rectangular pulse, specified as a vector.

Data Types: `single` | `double`

`w` — Rectangle width

1 (default) | positive number

Rectangle width, specified as a positive number.

Output Arguments

`y` — Rectangular pulse

vector

Rectangular pulse of unit amplitude, returned as a vector.

Note The interval of nonzero amplitude is defined to be open on the right, that is, `rectpuls(-0.5) = 1` while `rectpuls(0.5) = 0`.

Tips

`rectpuls` can be used in conjunction with the pulse train generating function `pulstran`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`chirp` | `cos` | `diric` | `gauspuls` | `pulstran` | `sawtooth` | `sin` | `sinc` | `square` | `tripuls`

Introduced before R2006a

rectwin

Rectangular window

Syntax

```
w = rectwin(L)
```

Description

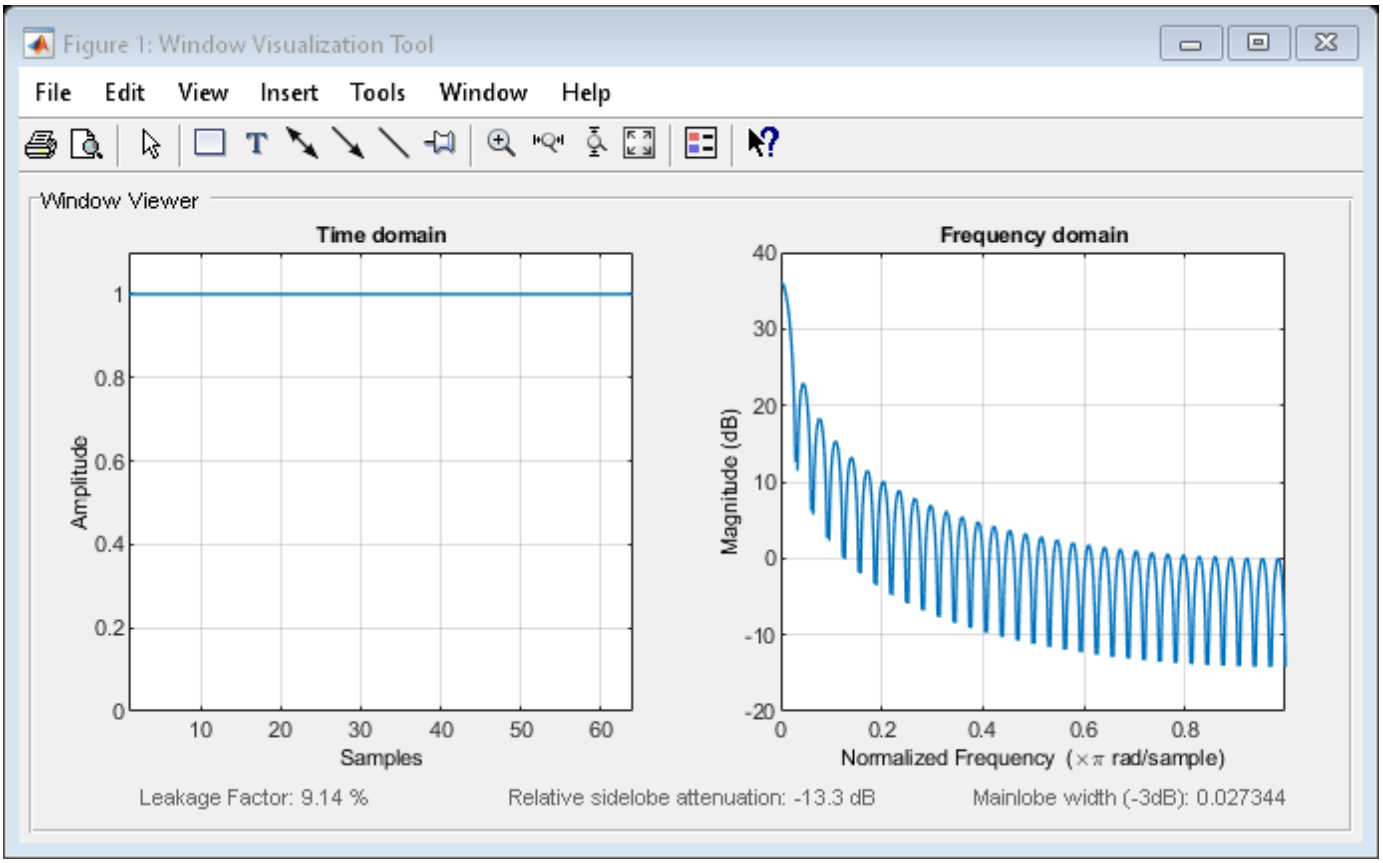
`w = rectwin(L)` returns a rectangular window of length `L`.

Examples

Rectangular Window

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

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



Input Arguments

`L` — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

Output Arguments

`w` — Rectangular window

column vector

Rectangular window, returned as a column vector.

Algorithms

The output of the `rectwin` function with input `L` can also be created using the `ones` function.

```
w = ones(L,1);
```

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps

Window Designer

Functions

`flattopwin` | `hamming` | `hann` | **WVTool** | `barthannwin` | `blackmanharris` | `blackman`

Introduced before R2006a

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,___)

[yTT,b] = resample(xTT,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,___)

yTT = resample(xTT)
[yTT,b] = resample(xTT,___)

[___] = resample(___,'Dimension',dim)
```

Description

`y = resample(x,p,q)` resamples the input sequence, `x`, at `p/q` times the original sample rate. `resample` applies an FIR “Antialiasing Lowpass Filter” on page 1-1897 to `x` and compensates for the delay introduced by the filter. The function operates along the first array dimension with size greater than 1.

`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.

`[yTT,b] = resample(xTT,p,q,___)` resamples the uniformly sampled data in the MATLAB timetable `xTT` at `p/q` times the original sample rate and returns a timetable `yTT`. You can specify additional arguments `n`, `beta`, or `b`.

`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`. `NaN`s 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 an intermediate uniform grid with a sample spacing of (`p`/`q`)/`fs`. The function then filters the result to upsample it by `p` and downsample it by `q`, resulting in a final sample rate of `fs`. For best results, ensure that `fs` × `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.

`yTT = resample(xTT)` resamples the nonuniformly sampled data in `xTT` and returns uniformly sampled data. `yTT` has the same endpoints and number of samples as `xTT`.

`[yTT,b] = resample(xTT, ___ )` resamples the nonuniformly sampled data in `xTT` and also returns the coefficients of the antialiasing filter in `b`. You can specify the same argument options available for input `x`,`tx`.

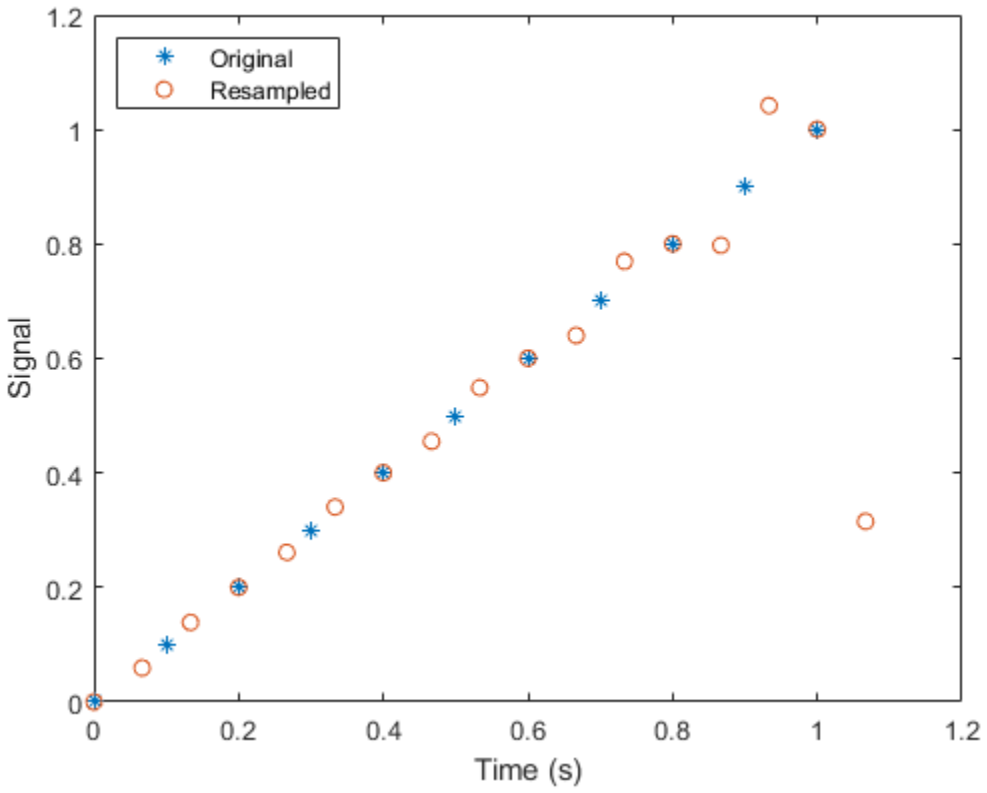
`[ ___ ] = resample( ___ ,'Dimension',dim)` resamples the input along dimension `dim`.

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.

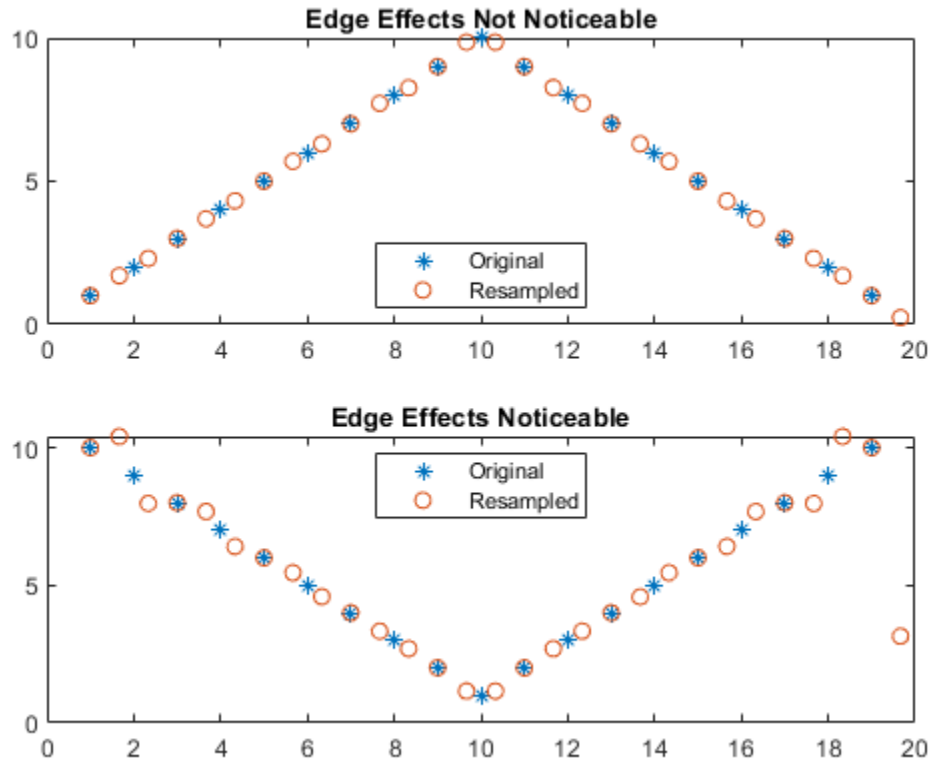
```
fs = 10;
t1 = 0:1/fs:1;
x = t1;
y = resample(x,3,2);
t2 = (0:(length(y)-1))*2/(3*fs);

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.

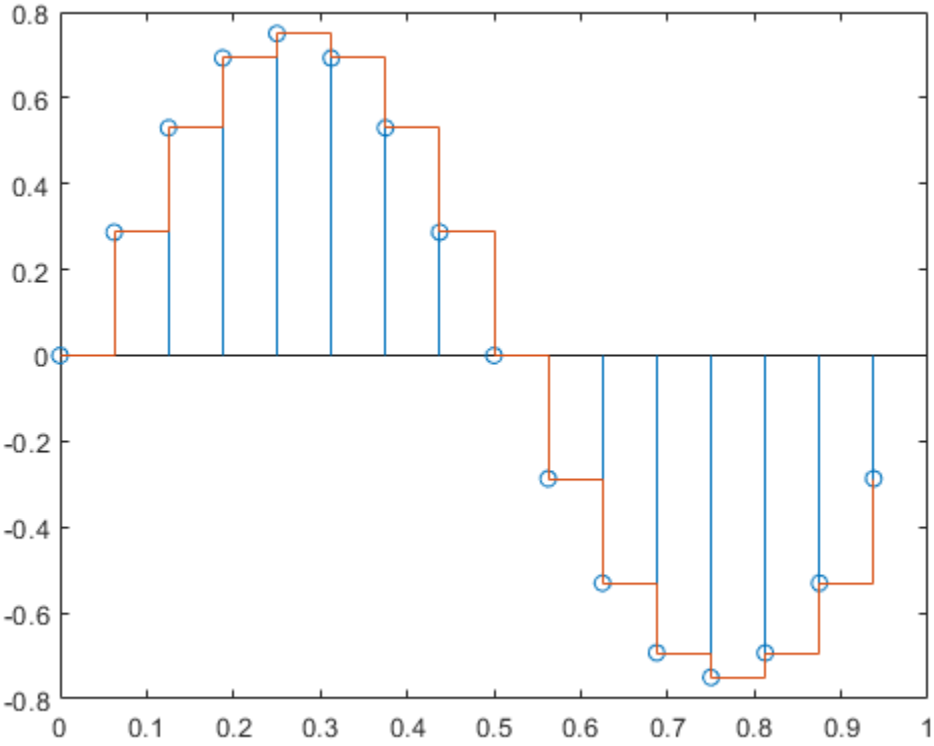
```
x = [1:10 9:-1:1;
    10:-1:1 2:10]';
y = resample(x,3,2);

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

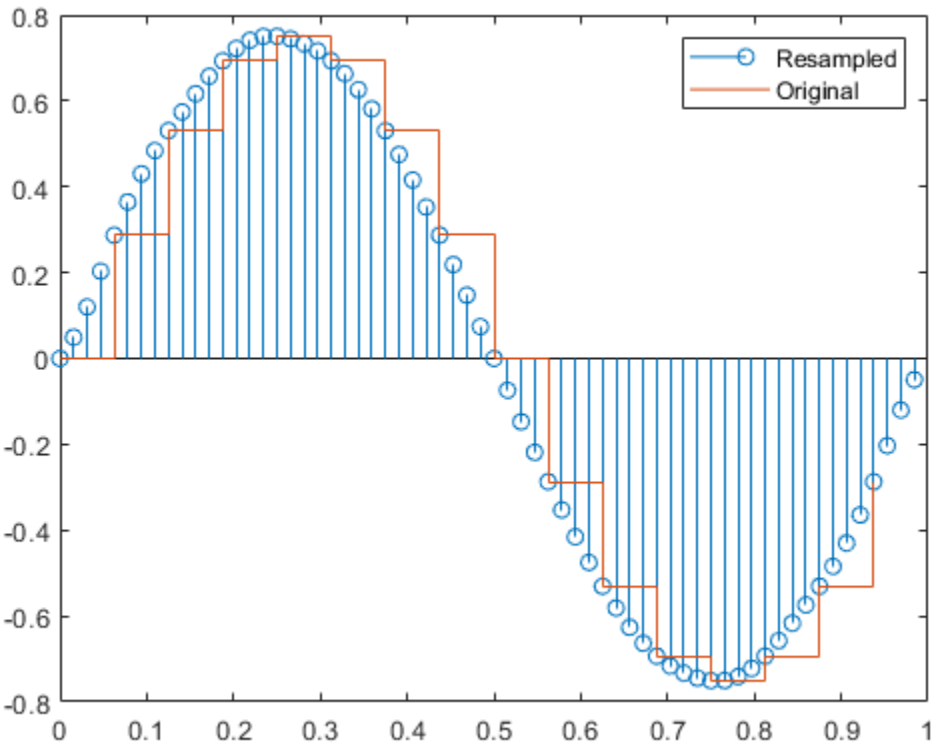
Construct a sinusoidal signal. Specify a sample rate such that 16 samples correspond to exactly one signal period. Draw a stem plot of the signal. Overlay a stairstep graph for sample-and-hold visualization.

```
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')
```

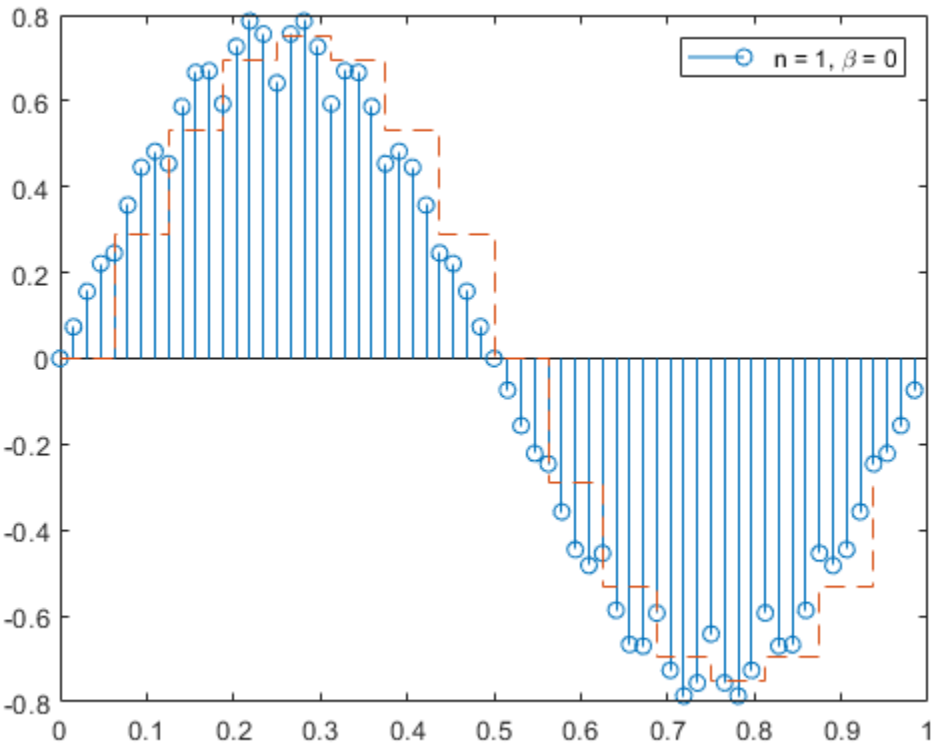


Repeat the calculation. Specify `n = 1` so that the antialiasing filter is of order $2 \times 1 \times 4 = 8$. Specify a shape parameter $\beta = 0$ for the Kaiser window. Output the filter as well as the resampled signal.

```
n = 1;
beta = 0;

[y,b] = resample(x,ups,dns,n,beta);

fo = filtord(b)
```

```
fo = 8
```

```
stem(tu,y)
hold on
stairs(t,x,'--')
hold off
legend('n = 1, \beta = 0')
```



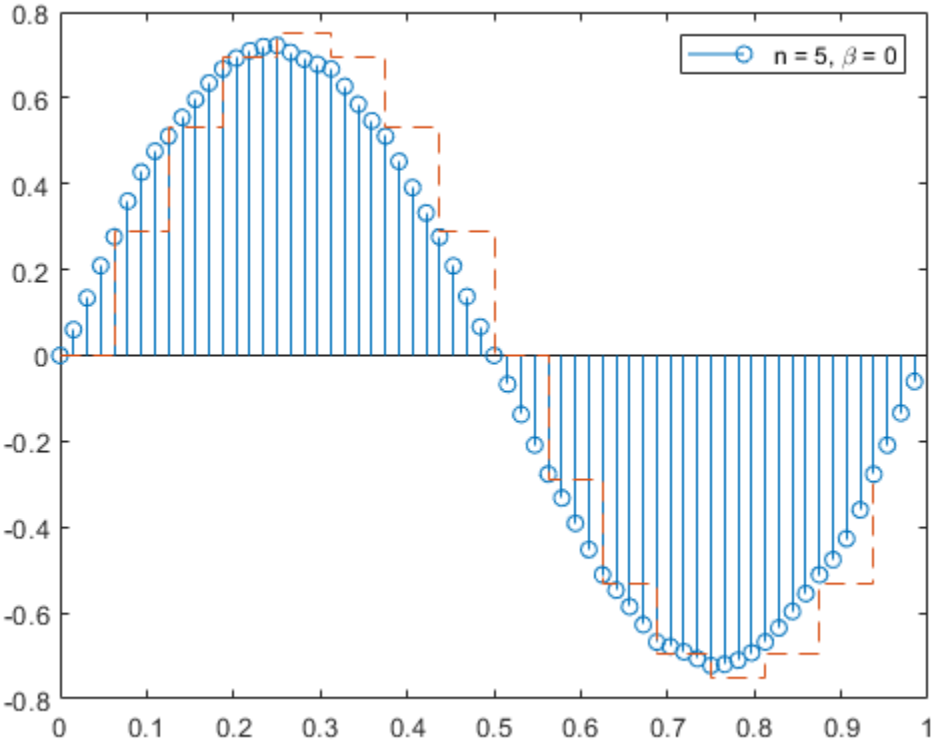
The resampled signal shows aliasing effects that result from the relatively wide mainlobe and low sidelobe attenuation of the window.

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)
```

```
fo = 40
```

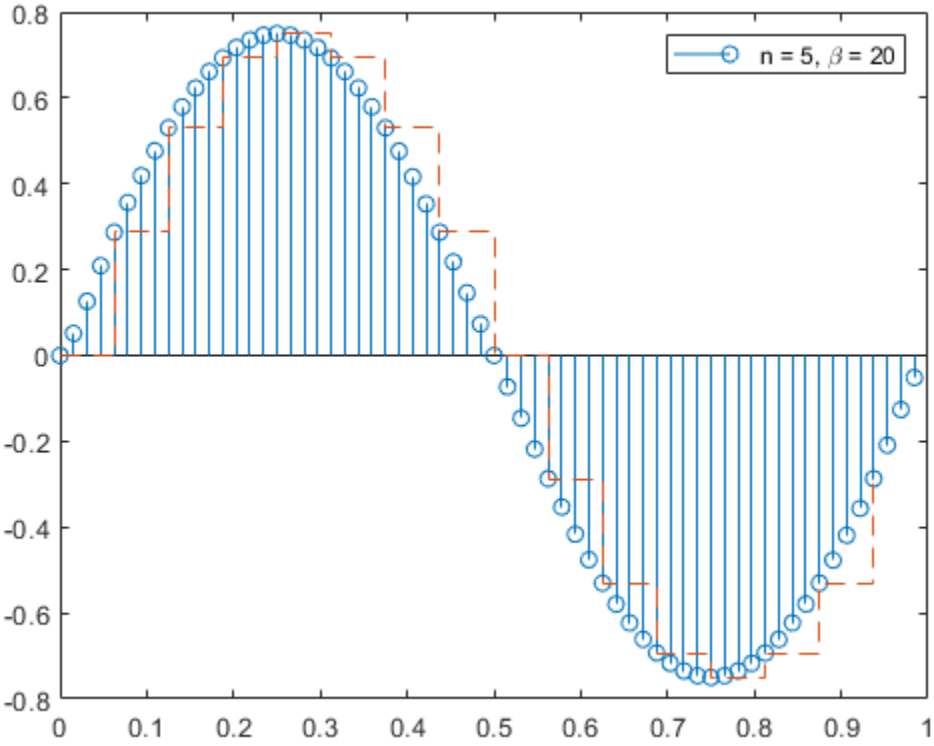
```
stem(tu,y)
hold on
stairs(t,x,'--')
hold off
legend('n = 5, \beta = 0')
```



The longer window has a narrower mainlobe and attenuates aliasing effects better. It also attenuates the signal.

Leave the filter order at $2 \times 5 \times 4 = 40$ and increase the shape parameter to $\beta = 20$.

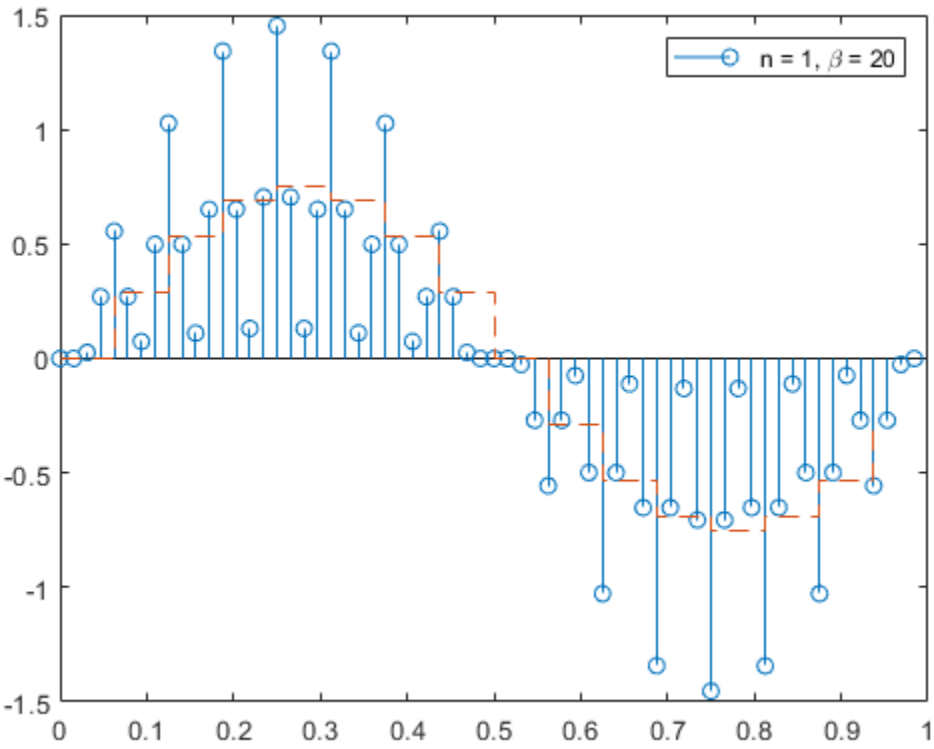
```
beta = 20;

y = resample(x,ups,dns,n,beta);

stem(tu,y)
hold on
stairs(t,x,'--')
hold off
legend('n = 5, \beta = 20')
```



The high sidelobe attenuation results in good resampling.

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')
```

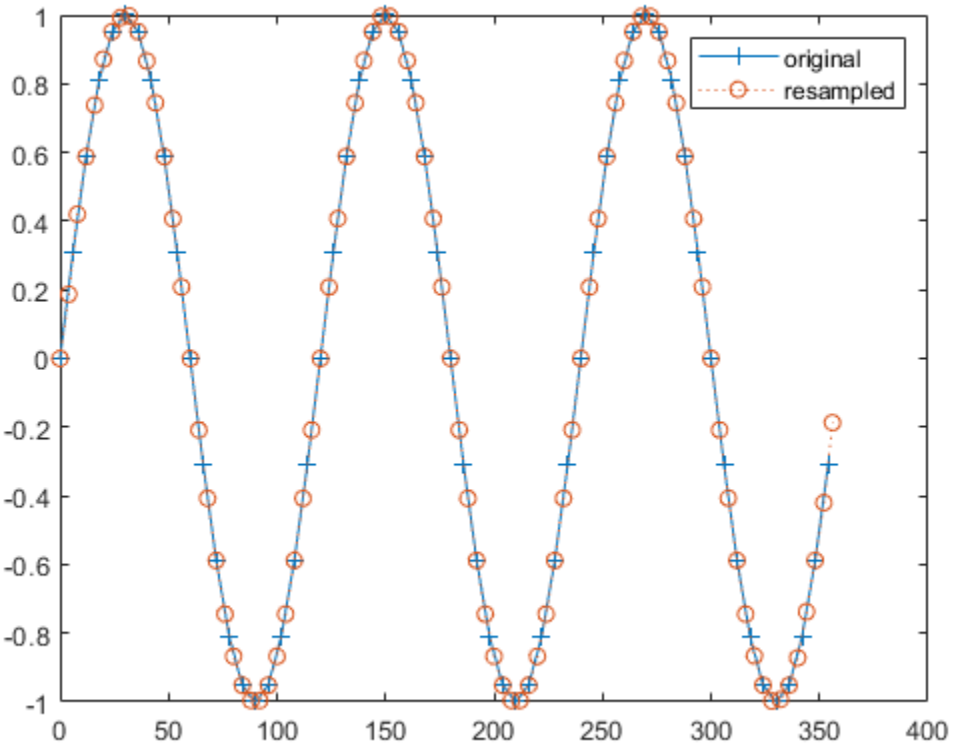


The wider mainlobe generates considerable artifacts upon resampling.

Resample a Sinusoid

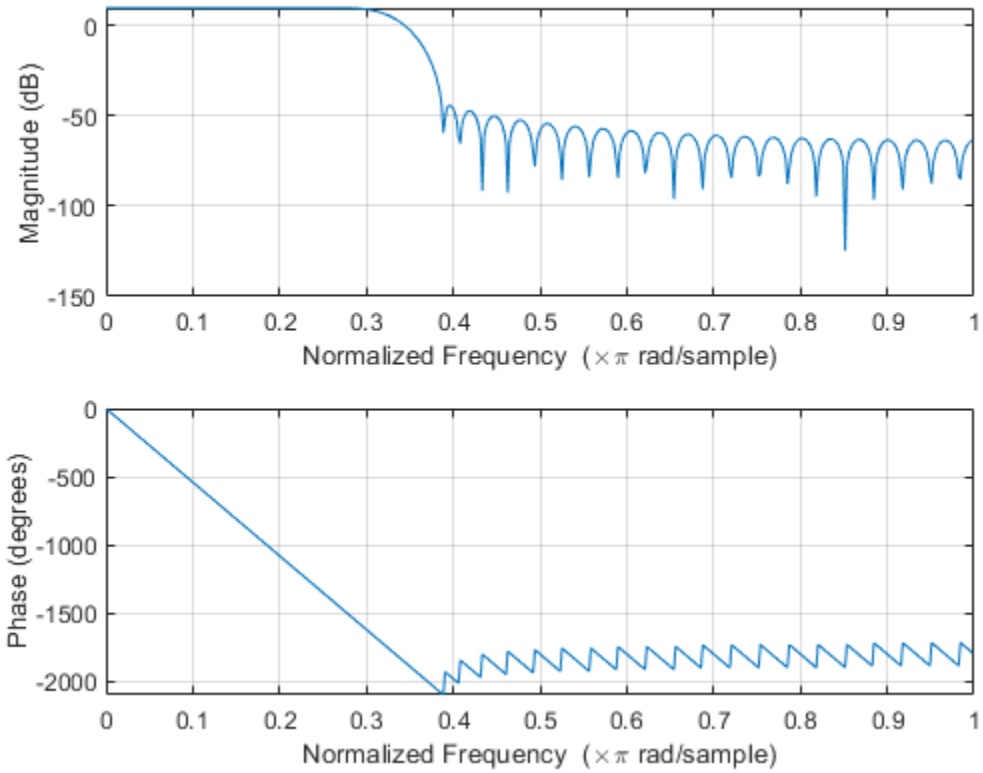
Generate 60 samples of a sinusoid and resample it at 3/2 the original rate. Display the original and resampled signals.

```
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')
```



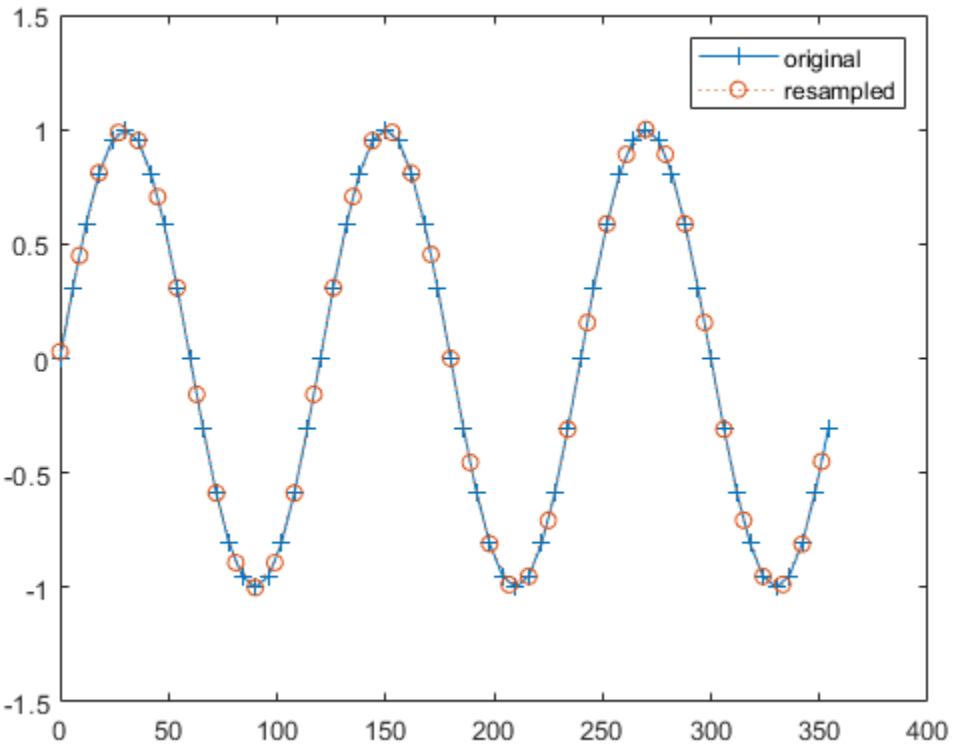
Plot the frequency response of the anti-aliasing filter.

```
freqz(by)
```



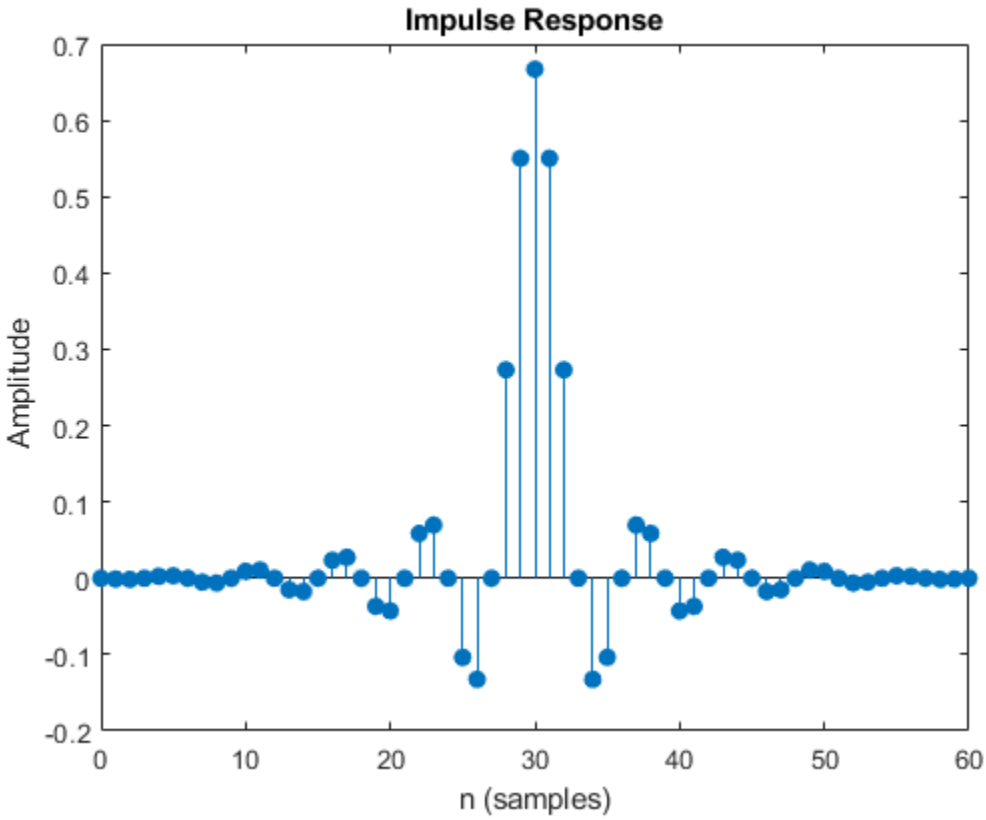
Resample the signal at 2/3 the original rate. Display the original signal and its resampling.

```
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)
```



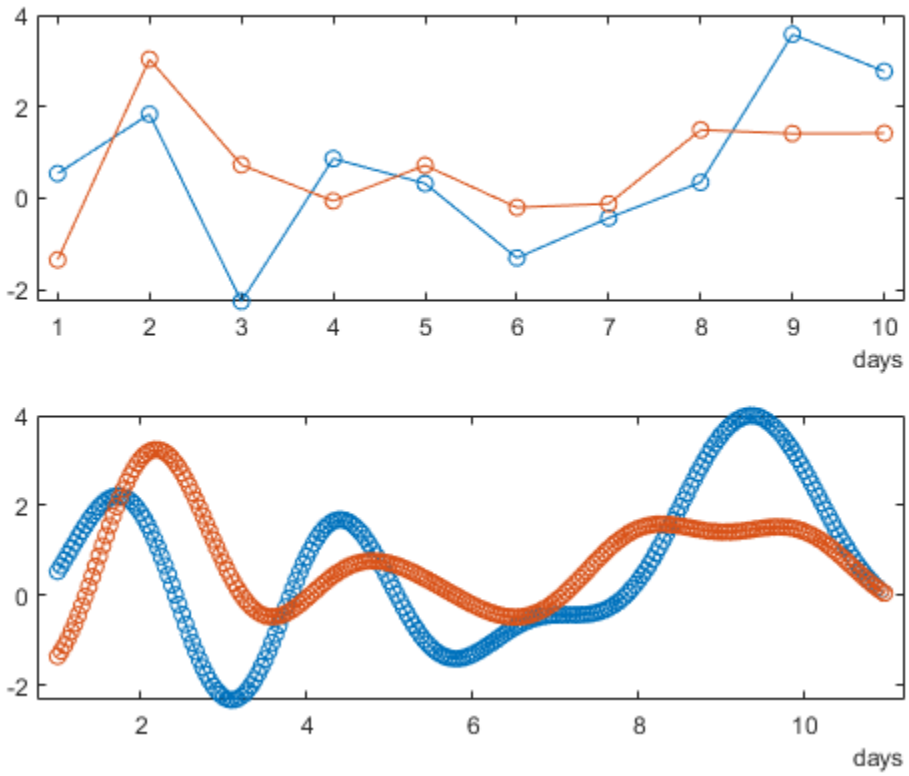
Resample Data in Timetable

Create two vectors of ten randomly generated numbers. Assume one number for each vector was recorded daily for a total of ten days. Store the data in a MATLAB timetable.

```
a = randn(10,1);
b = randn(10,1);

t = days(1:10);

xTT = timetable(t',[a b]);
```

Use the `resample` function to increase the sample rate from once daily to once hourly. Plot both data sets.

```
yTT = resample(xTT,24,1);

subplot(2,1,1)
plot(xTT.Time,xTT.Var1,'-o')
subplot(2,1,2)
plot(yTT.Time,yTT.Var1,'-o')
```



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, starting on 15 January. The observations have several gaps because Jupiter was not visible on cloudy nights. Generate a `datetime` array of observation times.

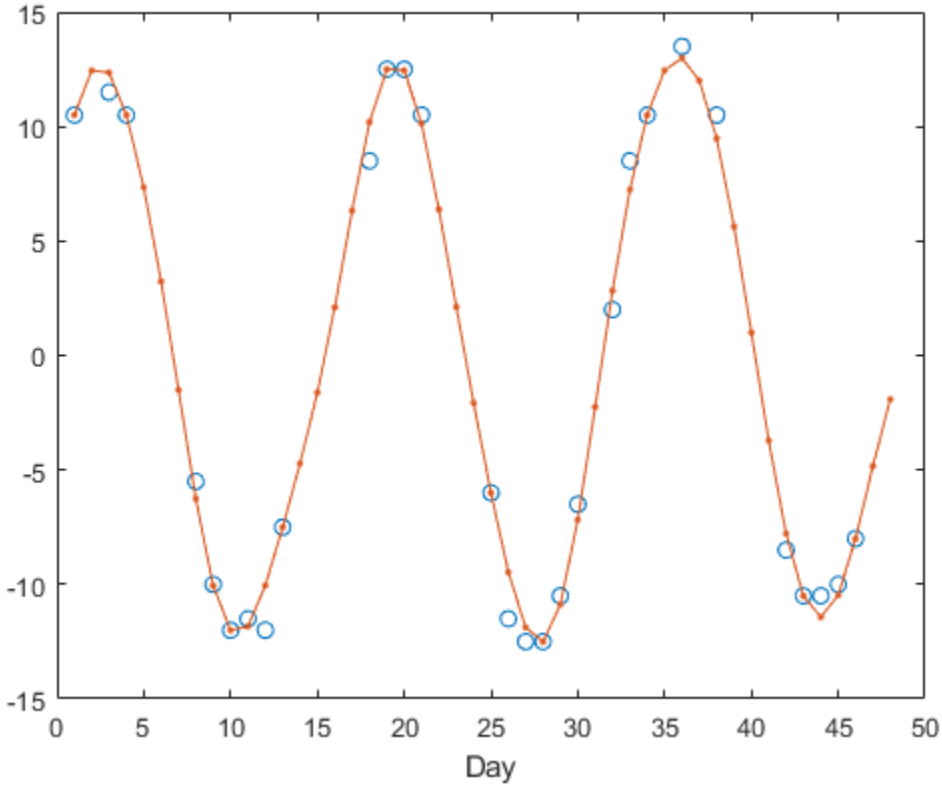
```
t = [0 2 3 7 8 9 10 11 12 17 18 19 20 24 25 26 27 28 29 31 32 33 35 37 ...
    41 42 43 44 45]'+1;

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]';

obsv = datetime(1610,1,15+t);
```

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);
```

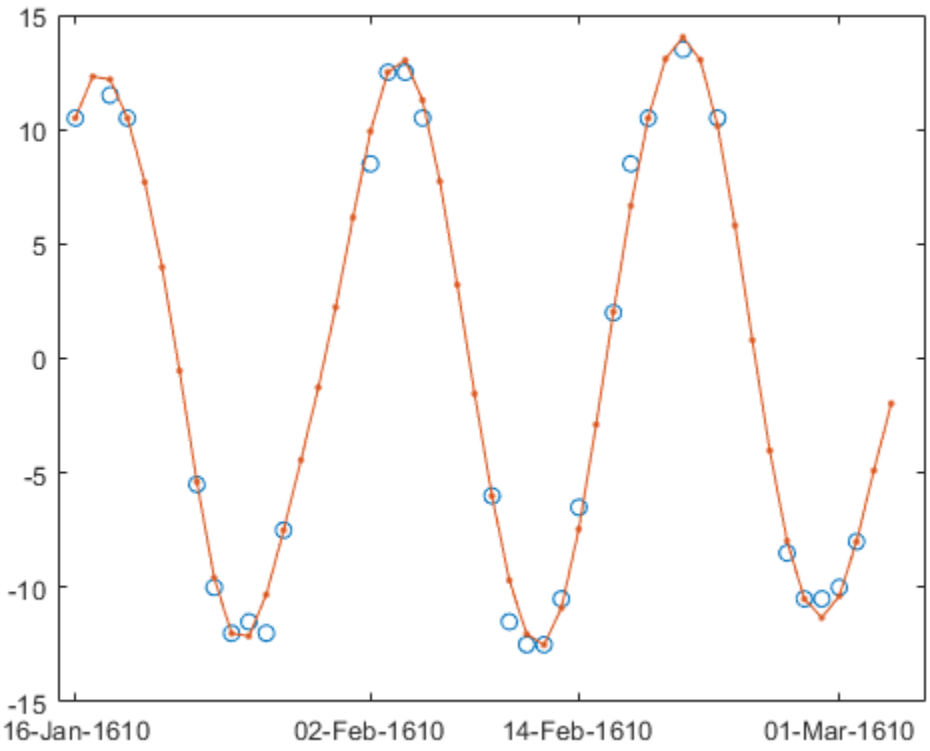
Plot the data and the resampled signal.

```
plot(t,yg,'o',ty,y,'.-')
xlabel('Day')
```



Repeat the procedure using spline interpolation and displaying the observation dates. Express the sample rate in inverse days.

```
fs = 1/86400;

[ys,tys] = resample(yg,obsv,fs,3,1,'spline');

plot(t,yg,'o')
hold on
plot(ys,'.-')
hold off

ax = gca;
ax.XTick = t(1:9:end);
ax.XTickLabel = char(obsv(1:9:end));
```

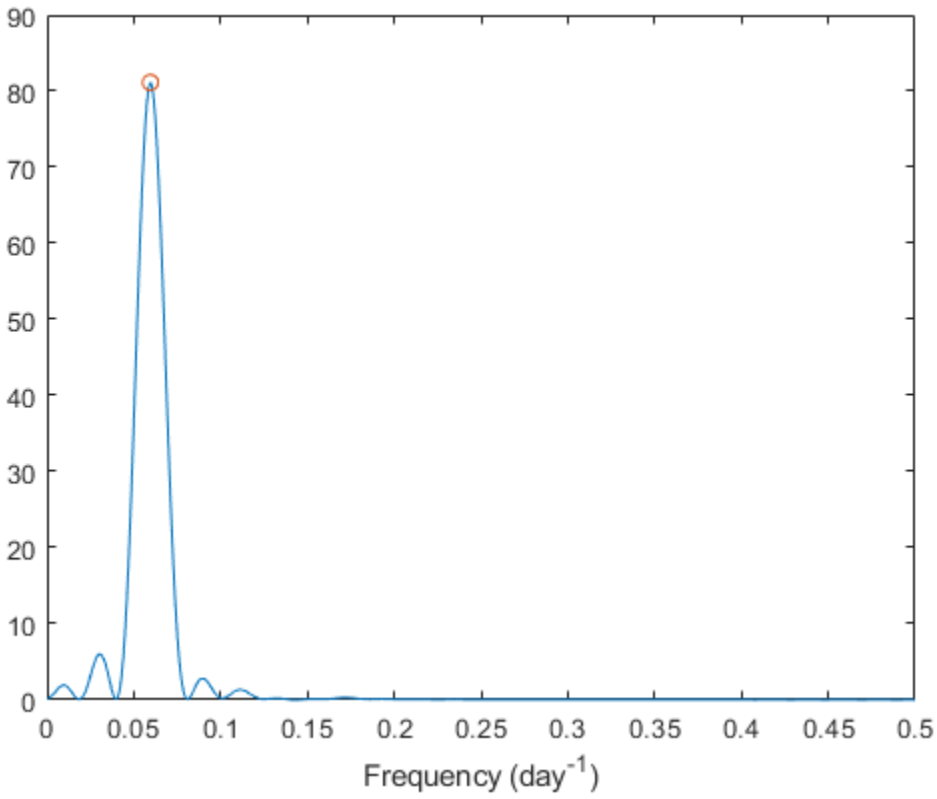


Compute the periodogram power spectrum estimate of the uniformly spaced, linearly interpolated data. Choose a DFT length of 1024. The signal peaks at the inverse of the orbital period.

```
[pxx,f] = periodogram(ys,[],1024,1,'power');
[pk,i0] = max(pxx);

f0 = f(i0);
T0 = 1/f0
```

```
T0 = 16.7869
```

```
plot(f,pxx,f0,pk,'o')
xlabel('Frequency (day^{-1})')
```



Resample Nonuniformly Sampled Data in Timetable

A person recorded their weight in pounds during the leap year 2012. The person did not record their weight every day, so the data are nonuniform. Load the data and store the measurements in a MATLAB timetable. Use a datetime vector to specify “RowTimes”.

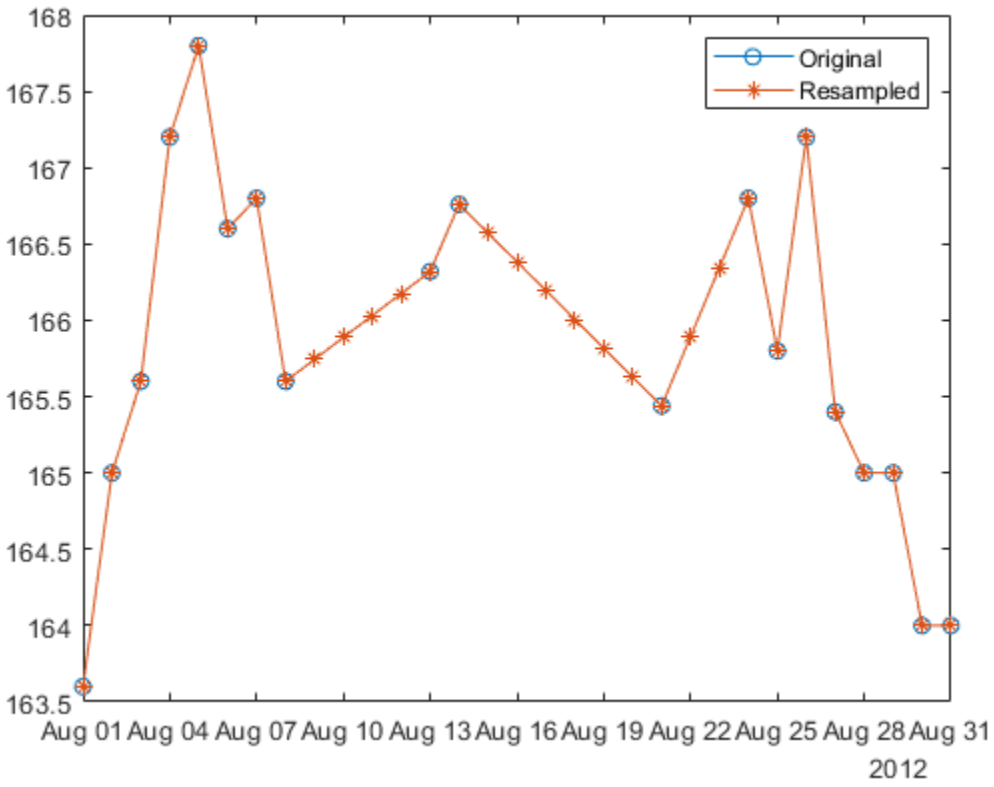
```
load weight2012.dat

rowTimes = datetime(2012,1,1:366)';
wt = weight2012(:,2);
xTT = timetable(rowTimes,wt);
```

Resample the data. The result is a timetable containing uniformly sampled data with the same endpoints and number of samples as `wt`.

```
yTT = resample(xTT);
```

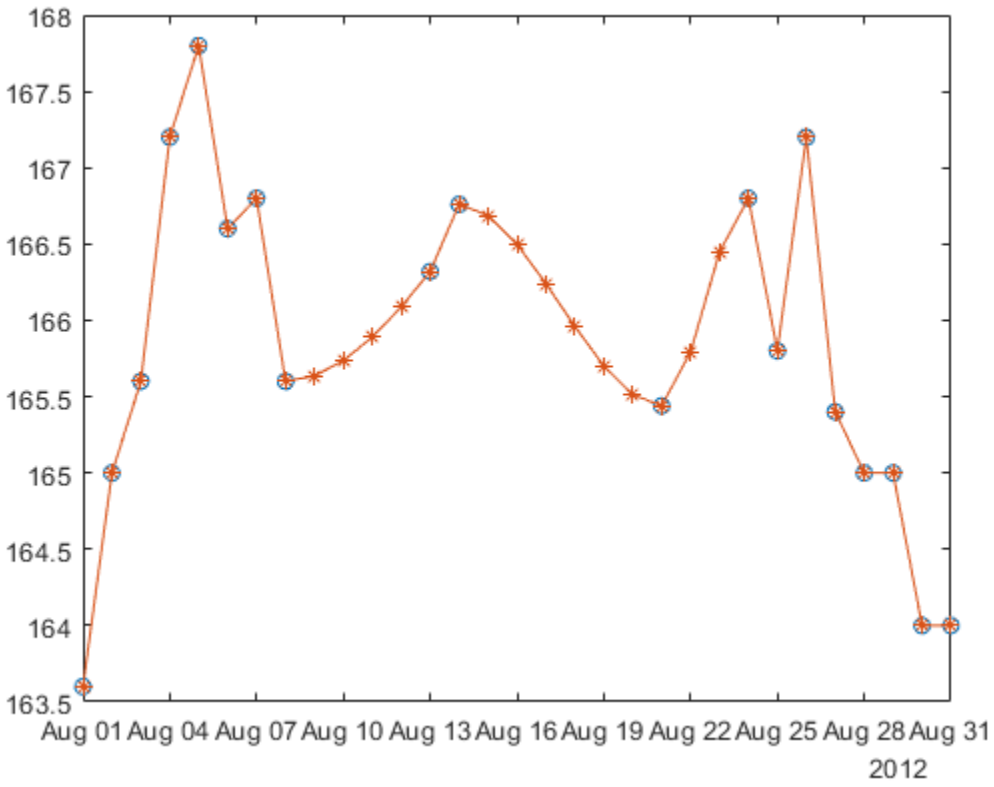
Plot both the original and the resampled data for comparison. Adjust the *x*-axis limits to display only the measurements in the month of August.

```
plot(xTT.rowTimes,xTT.wt,'-o',yTT.Time,yTT.wt,'-*')
xlim(datetime([2012 08 01;2012 08 31]))
legend('Original','Resampled')
```



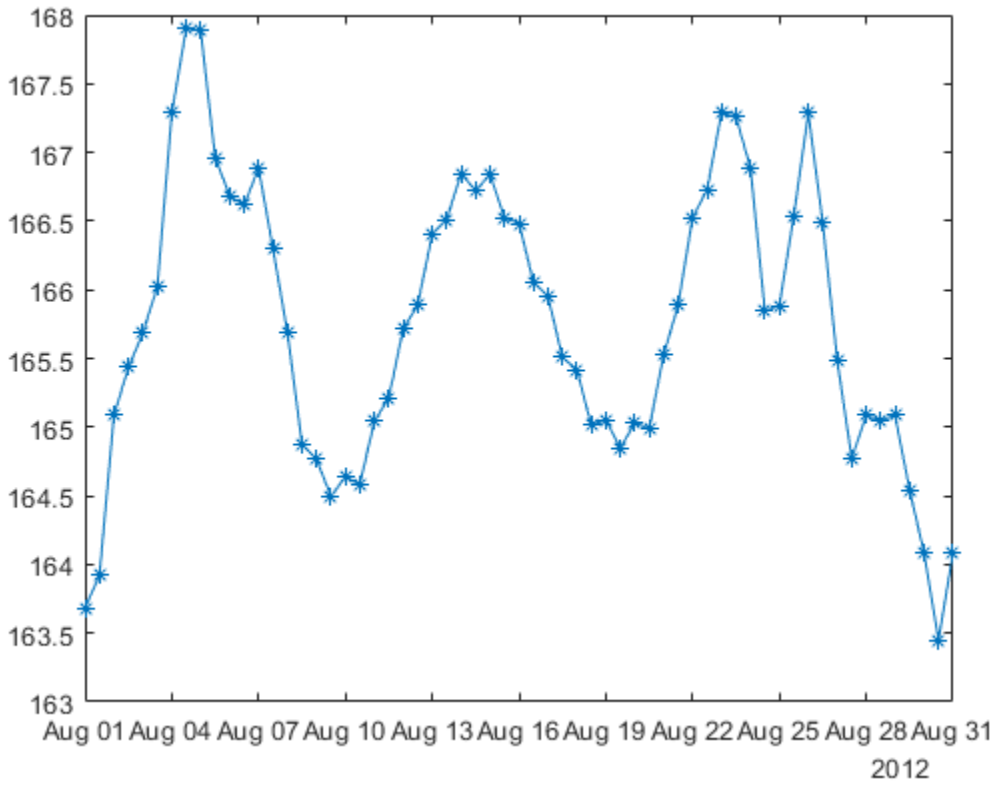
Resample the data again using cubic interpolation.

```
yTTs = resample(xTT,'pchip');

plot(xTT.rowTimes,xTT.wt,'o',yTTs.Time,yTTs.wt,'-*')
xlim(datetime([2012 08 01;2012 08 31]))
```



Now increase the sample rate to two measurements per day and use spline interpolation. Plot the result.

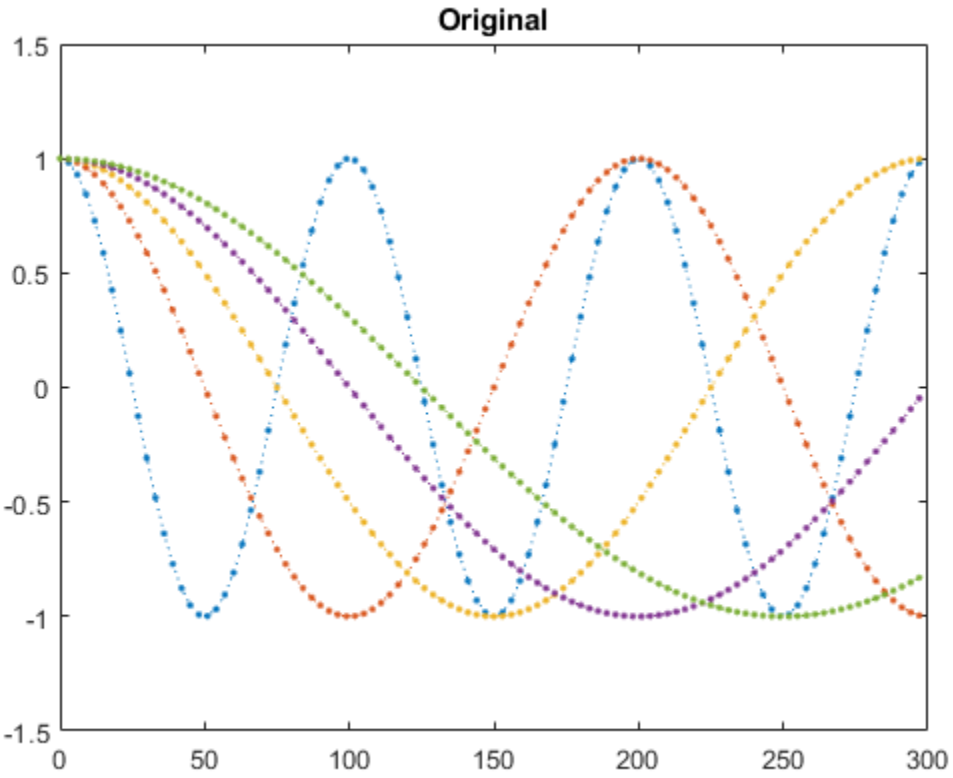
```
fs = 1/86400;
yTTf = resample(xTT,2*fs,'spline');

plot(yTTf.Time,yTTf.wt,'-*')
xlim(datetime([2012 08 01;2012 08 31]))
```



Resample Multichannel Signal

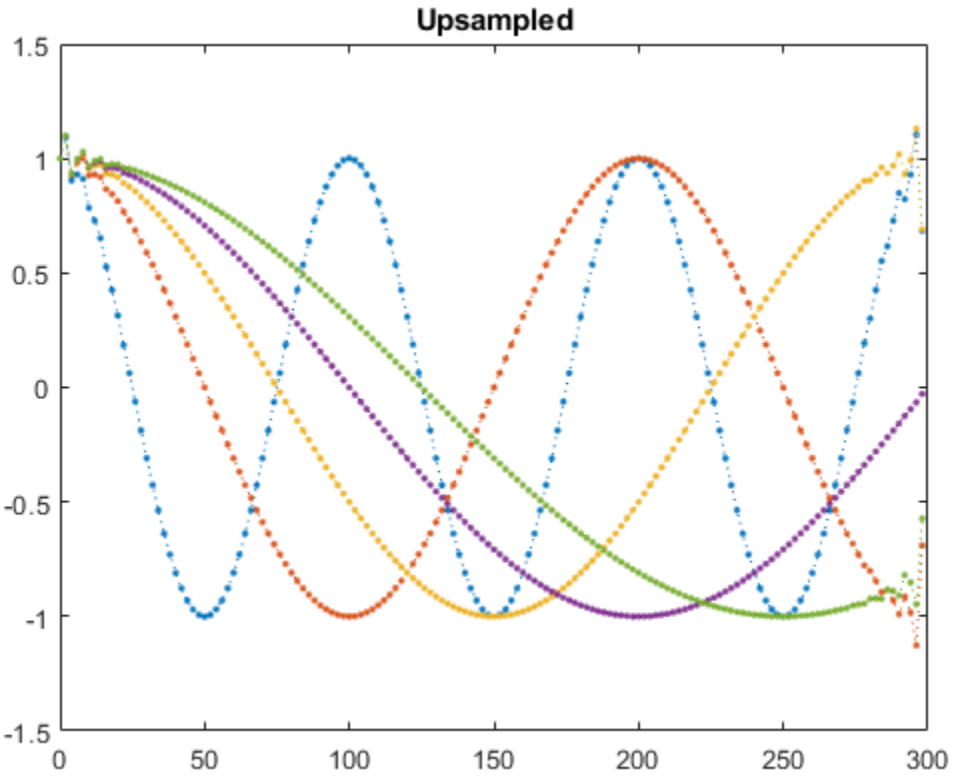
Generate a five-channel, 100-sample sinusoidal signal. Time increases across the columns and frequency increases down the rows. Plot the signal.

```
p = 3;
q = 2;

tx = 0:p:300-p;

x = cos(2*pi*tx./(1:5)'/100);

plot(tx,x,'.:')
title('Original')
ylim([-1.5 1.5])
```



Upsample the sinusoid by 3/2 along its second dimension. Overlay the resampled signal on the plot.

```
ty = 0:q:300-q;

y = resample(x,p,q,'Dimension',2);

plot(ty,y,'.:')
title('Upsampled')
```



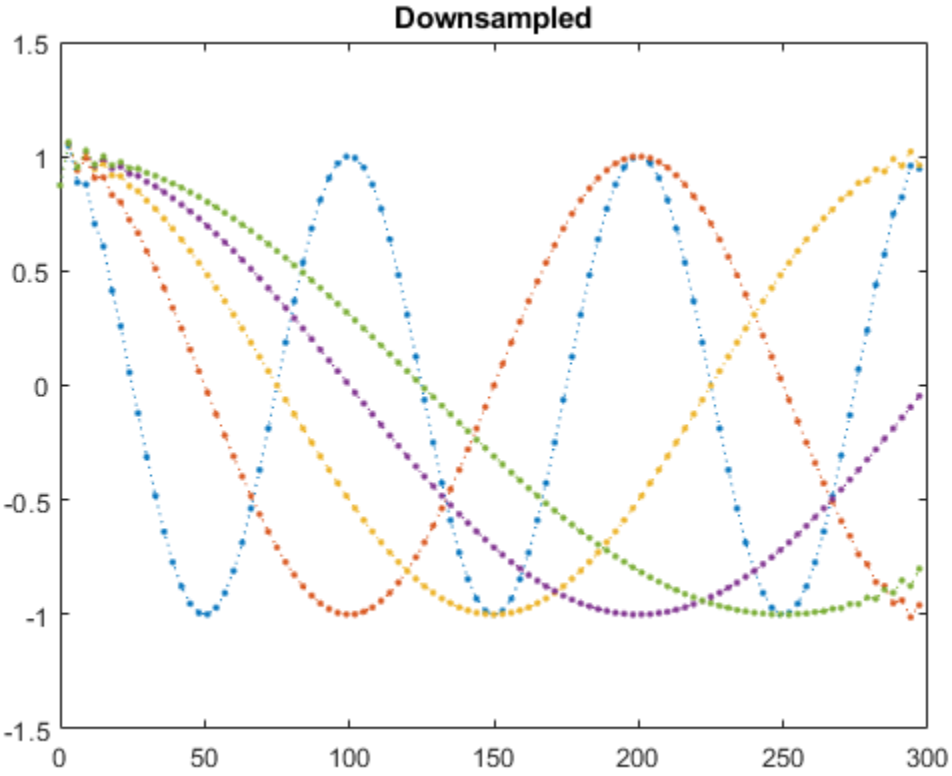
Reshape the resampled signal so that time runs along a third dimension.

```
y = permute(y,[1 3 2]);
size(y)
```

```
ans = 1×3

     5     1   150
```

Downsample the signal back to its original rate and plot it.

```
z = resample(y,q,p,'Dimension',3);

plot(tx,squeeze(z),'.:')
title('Downsampled')
```



Input Arguments

x — Input signal

vector | matrix | *N*-D array

Input signal, specified as a vector, matrix, or *N*-D array. `x` can contain `NaN`s when time information is provided. `NaN`s 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: `double`

p, q — Resampling factors

positive integers

Resampling factors, specified as positive integers.

Data Types: `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: 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: 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. See "Antialiasing Lowpass Filter" on page 1-1897 for more information.

Data Types: double

xTT — Input timetable
timetable

Input timetable with at least two rows, specified as a timetable. Each variable in xTT is treated as an independent signal. If a variable in the timetable is an *N*-D array, then resample operates along the first dimension.

Note

- RowTimes must be either a duration vector or a datetime object with unique and finite values. Nonfinite time values are treated as missing data and are ignored.
- If unsorted, resample sorts RowTimes in ascending order.

See timetable for more information.

Data Types: double

tx — Time instants
nonnegative real vector | datetime array

Time instants, specified as a nonnegative real vector or a datetime array. tx must increase monotonically but need not be uniformly spaced. tx can contain NaNs or NaTs. These values are treated as missing data and excluded from the resampling. tx is valid only for input x.

Data Types: double | datetime

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: 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.

Note If `x` is not slowly varying, consider using `interp1` with the `'pchip'` interpolation method.

`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If `dim` is not specified, `resample` operates along the first array dimension with size greater than 1. If the input is a timetable, then `dim` must be 1.

Data Types: `single` | `double`

Output Arguments

`y` — Resampled signal
vector | matrix | *N*-D array

Resampled signal, returned as a vector, matrix, or *N*-D array. If `x` is of length *N* along dimension `dim` and you specify `p` and `q`, then `y` is of length $\lceil N \times p/q \rceil$ along `dim`.

`b` — FIR filter coefficients
vector

FIR filter coefficients, returned as a vector.

`ty` — Output instants
nonnegative real vector

Output instants, returned as a nonnegative real vector. `ty` applies only for input `x`.

`yTT` — Resampled timetable
`timetable`

Resampled timetable, returned as a `timetable`.

More About

Antialiasing Lowpass Filter

To resample a signal by a rational factor `p`/`q`, `resample` calls `upfirdn`, which conceptually performs these steps:

1. Insert zeros to upsample the signal by `p`.
2. Apply an FIR antialiasing filter to the upsampled signal.
3. Discard samples to downsample the filtered signal by `q`.

The ideal antialiasing filter has normalized cutoff frequency $f_c = \pi/\max(p,q)$ rad/sample and gain `p`. To approximate the antialiasing filter, `resample` uses the Kaiser window method.

- The filter order is $2 \times n \times \max(p,q)$. The default value of `n` is 50.
- The Kaiser window has a shape parameter `beta` that controls the tradeoff between transition width and stopband attenuation. The default value of `beta` is 5.
- The filter coefficients are normalized to account for the processing gain of the window.

As an example, design an antialiasing filter to resample a signal to 3/2 times its original sample rate:

```
p = 3;
q = 2;
maxpq = max(p,q);

fc = 1/maxpq;
n = 50;
order = 2*n*maxpq;
beta = 5;

b = fir1(order,fc,kaiser(order+1,beta));
b = p*b/sum(b);
```

See “Resampling Uniformly Sampled Signals” for more information.

Tips

- Use the `isregular` function to determine if a timetable is uniformly sampled.

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`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

`datetime` and duration arrays are not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

`datetime` and duration arrays are not supported for code generation.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'pchip'` interpolation method is not supported.
- Input timetable containing a `gpuArray` is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`decimate` | `downsample` | `fillgaps` | `firls` | `interp` | `interp1` | `intfilt` | `kaiser` | `spline` | `upfirdn` | `upsample`

Topics

“Resampling Uniformly Sampled Signals”
“Resampling Nonuniformly Sampled Signals”
“Reconstructing Missing Data”
“Reconstruct a Signal from Irregularly Sampled Data”

Introduced before R2006a

residuez

Z-transform partial-fraction expansion

Syntax

```
[ro,po,ko] = residuez(bi,ai)
[bo,ao] = residuez(ri,pi,ki)
```

Description

`[ro,po,ko] = residuez(bi,ai)` finds the residues, poles, and direct terms of a partial fraction expansion of the ratio of numerator and denominator polynomials, `b` and `a`.

`[bo,ao] = residuez(ri,pi,ki)` with three input arguments and two output arguments, converts the partial fraction expansion back to polynomials with coefficients in row vectors `b` and `a`.

Examples

Partial-Fraction Expansion of IIR Lowpass Filter

Compute the partial-fraction expansion corresponding to the third-order IIR lowpass filter described by the 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.

```
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);
```

Compute the residues, poles, and direct terms of the partial-fraction expansion.

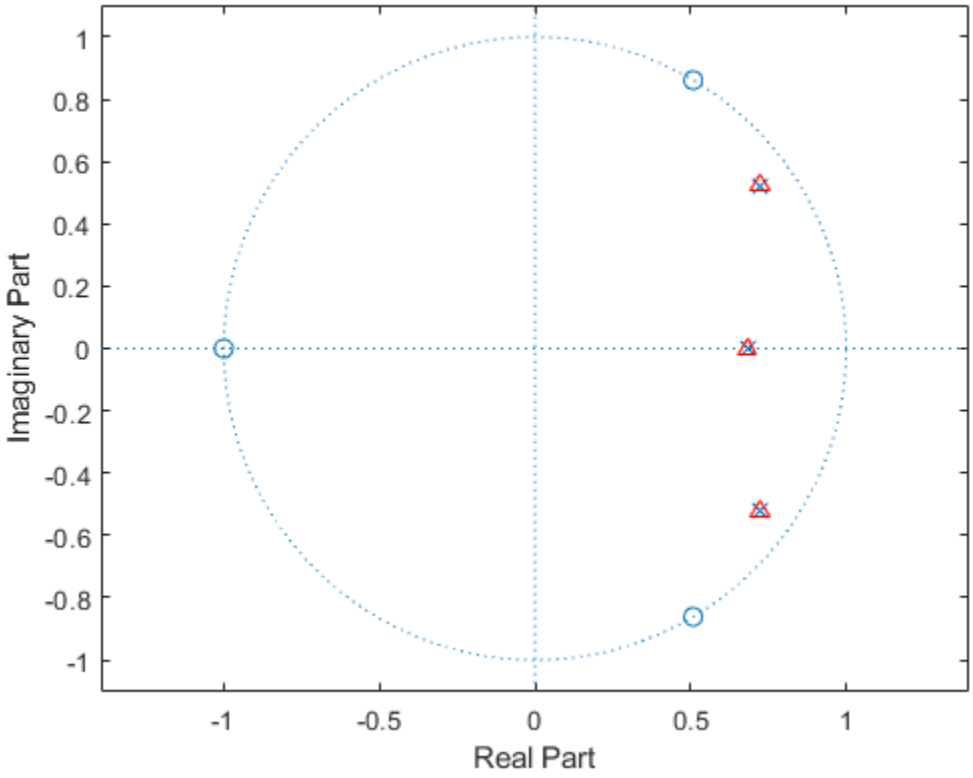
```
[r,p,k] = residuez(b,a)
```

```
r = 3×1 complex

  -0.1153 - 0.0182i
  -0.1153 + 0.0182i
   0.3905 + 0.0000i


p = 3×1 complex
```

```
   0.7230 + 0.5224i
   0.7230 - 0.5224i
   0.6830 + 0.0000i


k = -0.1037
```

Plot the poles and zeros of the transfer function and overlay the poles you just found.

```
zplane(b,a)
hold on
plot(p,'^r')
hold off
```



Use `residuez` again to reconstruct the transfer function.

```
[bn,an] = residuez(r,p,k)

bn = 1×4

    0.0563   -0.0009   -0.0009    0.0563


an = 1×4

    1.0000   -2.1291    1.7834   -0.5435
```

Input Arguments

bi, ai — Polynomial coefficients

vector

Polynomial coefficients, specified as vectors. Vectors `b` and `a` specify the coefficients of the polynomials of the discrete-time system $b(z)/a(z)$ in descending powers of z.

$$B(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_m z^{-m}$$

$$A(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_n z^{-n}$$

If there are multiple roots and `a > n-1`,

$$\frac{B(z)}{A(z)} = \frac{r(1)}{1 - p(1)z^{-1}} + \cdots + \frac{r(n)}{1 - p(n)z^{-1}} + k(1) + k(2)z^{-1} + \cdots + k(m - n + 1)z^{-(m - n)}$$

ri — Residues

column vector

Residues of the partial fraction, specified as a vector.

pi — Poles

column vector

Poles of the partial fraction, specified as a vector.

ki — Direct terms

row vector

Direct terms, specified as a row vector.

Output Arguments

ro — Residues

column vector

Residues of the partial fraction, returned as a vector.

po — Poles

column vector

Pole of the partial fraction, returned as a vector. The number of poles is

```
n = length(a)-1 = length(r) = length(p)
```

If `p(j) = ... = p(j+s-1)` is a pole of multiplicity `s`, then the expansion includes terms of the form

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

ko — Direct term

row vector

Direct terms, returned as a row vector. The direct term coefficient vector `k` is empty if `length(b)` is less than `length(a)`; otherwise:

```
length(k) = length(b) - length(a) + 1
```

bo, ao — Polynomial coefficients
vector

Polynomial coefficients, returned as vectors.

Algorithms

`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 round-off errors, can cause arbitrarily large changes in the resulting poles and residues. You should use state-space or pole-zero representations instead.

`residuez` applies standard MATLAB functions and partial fraction techniques to find `r`, `p`, and `k` from `b` and `a`. It finds

- The direct terms `a` using `deconv` (polynomial long division) when `length(b) > length(a)-1`.
- The poles using `p = roots(a)`.
- Any repeated poles, reordering the poles according to their multiplicities.
- The residue for each nonrepeating pole p_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.

  $$\frac{1}{1 - p_j z^{-1}}, \frac{1}{(1 - p_j z^{-1})^2}, \cdots, \frac{1}{(1 - p_j z^{-1})^{s_j}}$$

  The vector `h` and matrices `S1` and `S2` have `n + xtra` rows, where `n` is the total number of roots and the internal parameter `xtra`, set to 1 by default, determines the degree of over-determination of the system of equations.

Note 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`.

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

`convmtx` | `deconv` | `poly` | `prony` | `residue` | `roots` | `ss2tf` | `tf2ss` | `tf2zp` | `tf2zpk` | `zp2ss`

Introduced before R2006a

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 1-1914. Because `risetime` uses interpolation, `r` can contain values that do not correspond to sampling instants of the bilevel waveform, `x`.

`r = risetime(x,fs)` specifies the sample rate in hertz. The sample rate 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` can 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.

Examples

Rise Time in Bilevel Waveform

Determine the rise time in samples for a 2.3 V clock waveform.

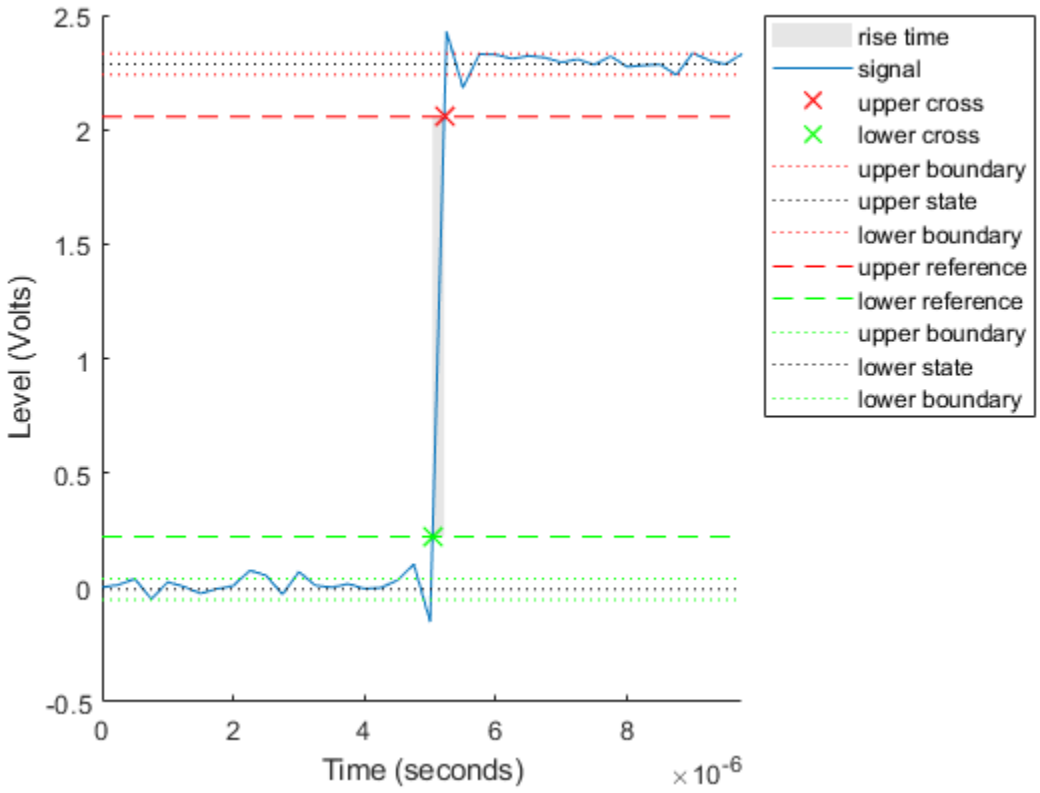
Load the 2.3 V clock data. Determine the rise time in samples. Use the default 10% and 90% percent reference levels.

```
load('transitionex.mat','x','t')
R = risetime(x)
```

```
R = 0.7120
```

The rise time is less than 1, indicating that the transition occurred in a fraction of a sample. Plot the data, including the time information, and annotate the rise time.

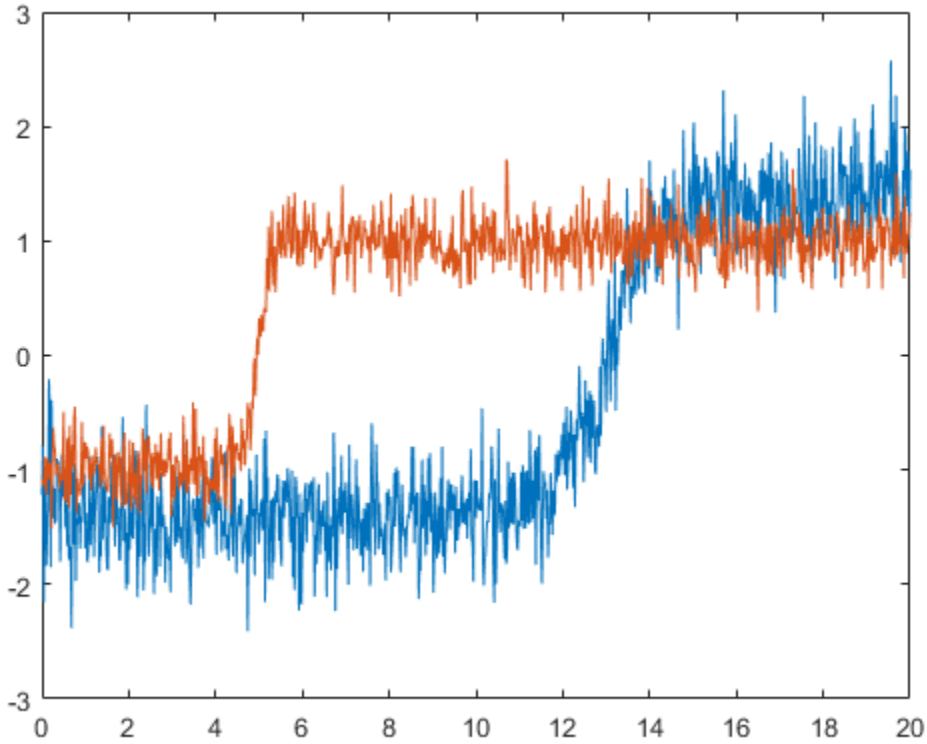
```
risetime(x,t);
```



Align Two Bilevel Waveforms

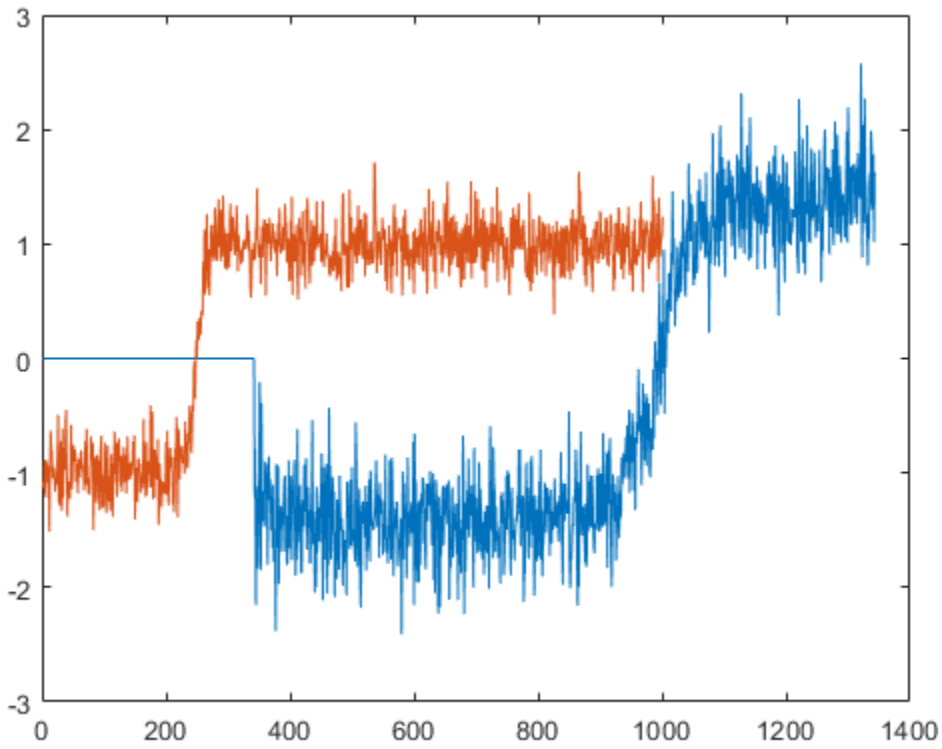
Generate two signals that represent bilevel waveforms. The signals are sampled at 50 Hz for 20 seconds. For the first signal, the transition occurs 13 seconds after the start of the measurement. For the second signal, the transition occurs 5 seconds after the start of the measurement. The signals have different amplitudes and are embedded in white Gaussian noise of different variances. Plot the signals.

```
tt = linspace(0,20,1001)';
e1 = 1.4*tanh(tt-13)+randn(size(tt))/3;
e2 = tanh(3*(tt-5))+randn(size(tt))/5;

plot(tt,e1,tt,e2)
```



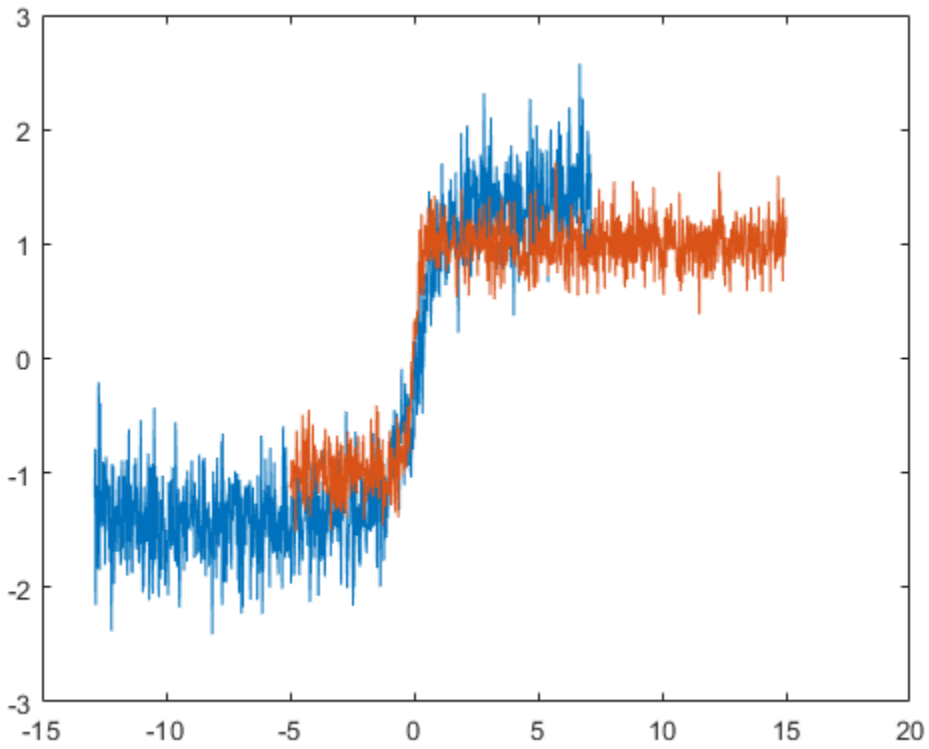
Align the signals so their transition times coincide. Correlation-based methods cannot align this type of signals adequately.

```
[y1,y2,D] = alignsignals(e1,e2);

plot(y1)
hold on
plot(y2)
hold off
```



Use `risetime` to align the signals. For each signal, find the transition time as the average of the instant at which the signal crosses the lower reference level and the instant at which it crosses the upper reference level. Plot the aligned waveforms.

```
[~,l1,u1] = risetime(e1,tt);
[~,l2,u2] = risetime(e2,tt);

t1 = tt-(l1+u1)/2;
t2 = tt-(l2+u2)/2;

plot(t1,e1,t2,e2)
```

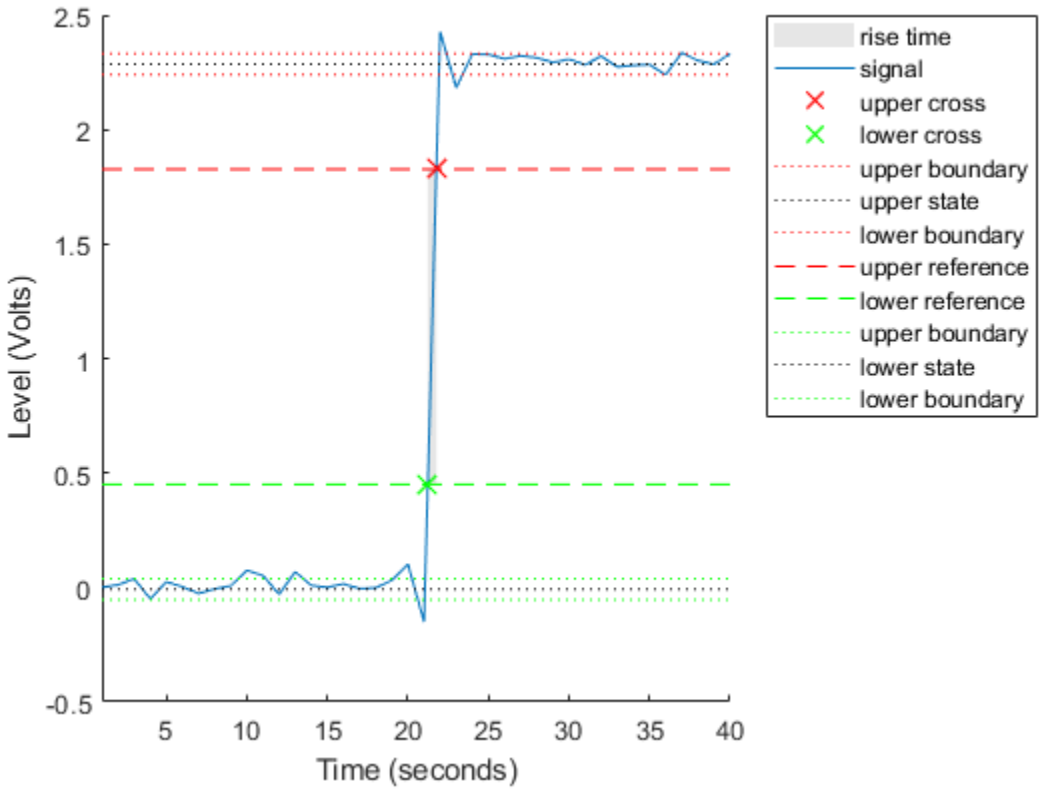


Rise Time with Nondefault Reference Levels

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. Compute the sample rate as the inverse of the time difference between consecutive samples. Determine the rise time using the 20% and 80% reference levels. Plot the annotated waveform.

```
load('transitionex.mat','x','t')

fs = 1/(t(2)-t(1));

risetime(x,'PercentReferenceLevels',[20 80])
```




```
ans = 0.5340
```

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.

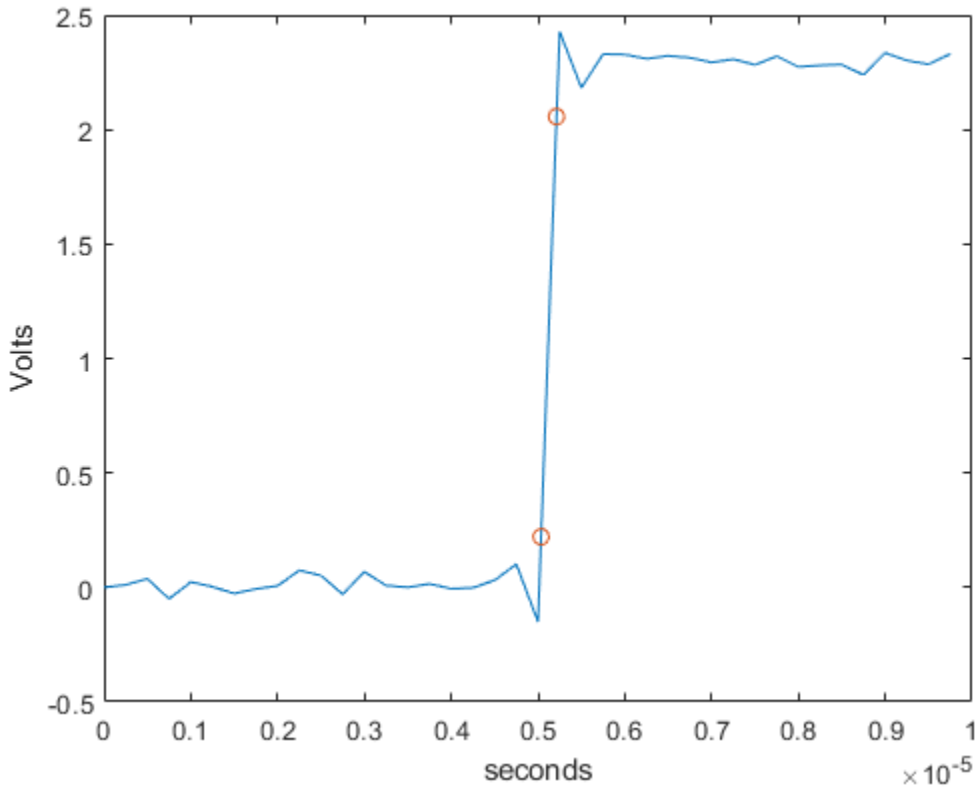
```
load('transitionex.mat','x','t')
```

Determine the rise time, reference-level instants, and reference levels.

```
[R,lt,ut,ll,ul] = risetime(x,t);
```

Plot the waveform 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,x)
xlabel('seconds')
ylabel('Volts')

hold on
plot([lt ut],[ll ul],'o')
hold off
```




```
fprintf('Rise time is %g seconds.',ut-lt)
```

```
Rise time is 1.78e-07 seconds.
```

Input Arguments

x — Bilevel waveform

real vector

Bilevel waveform, specified as a real-valued vector.

fs — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar in hertz.

t — sample instants

real vector

Sample instants, specified as a vector. The length of `t` must equal the length of the bilevel waveform `x`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'StateLevels',[0,0.8],'Tolerance',10,'PercentReferenceLevels',[20 50]` specifies that the low and high levels are 0 ± 10% and 0.8 ± 10%, respectively, and that a transition occurs when the signal crosses from 0.8 × 0.2 to 0.8 × 0.5.

`PercentReferenceLevels` — Reference levels as a percentage of the waveform amplitude
`[10 90]` (default) | two-element positive row vector

Reference levels as a percentage of the waveform amplitude, specified as the comma-separated pair consisting of `'PercentReferenceLevels'`and a two-element positive row vector. The elements of the row vector correspond to the lower and upper percent reference levels. The high state level is defined to be 100 percent and the low state level is defined to be 0 percent. See “Percent Reference Levels” on page 1-628 for more details.

`StateLevels` — Low and high state levels
two-element positive row vector

Low and high state levels, specified as the comma-separated pair consisting of `'StateLevels'` and a two-element positive row vector. The first and second elements of the vector correspond to the low and high state levels.

`Tolerance` — Lower- and upper-state boundaries
2 (default) | real positive scalar

Lower- and upper-state boundaries, specified as the comma-separated pair consisting of `'Tolerance'` and a real positive scalar as a percentage value. See “State-Level Tolerances” on page 1-628 for more information on this name-value pair.

Output Arguments

`r` — Duration of positive-going transition
vector

Duration of positive-going transition, returned as a vector. If you specify the sample rate `Fs` or the sample instants `t`, the rise times are in seconds. If you do not specify a sample rate or sample instants, the rise times are in samples.

`lt` — Lower reference-level crossing instants
vector

Lower reference-level crossing instants, returned as a vector. The vector `lt` contains the time instants when the positive-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` — Upper reference-level crossing instants
vector

Upper reference-level crossing instants, returned as a vector. The vector `ut` contains the time instants when the positive-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
real numeric scalar

Lower reference level in waveform amplitude units, returned as a real numeric scalar. `ll` is a vector containing the waveform values 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.

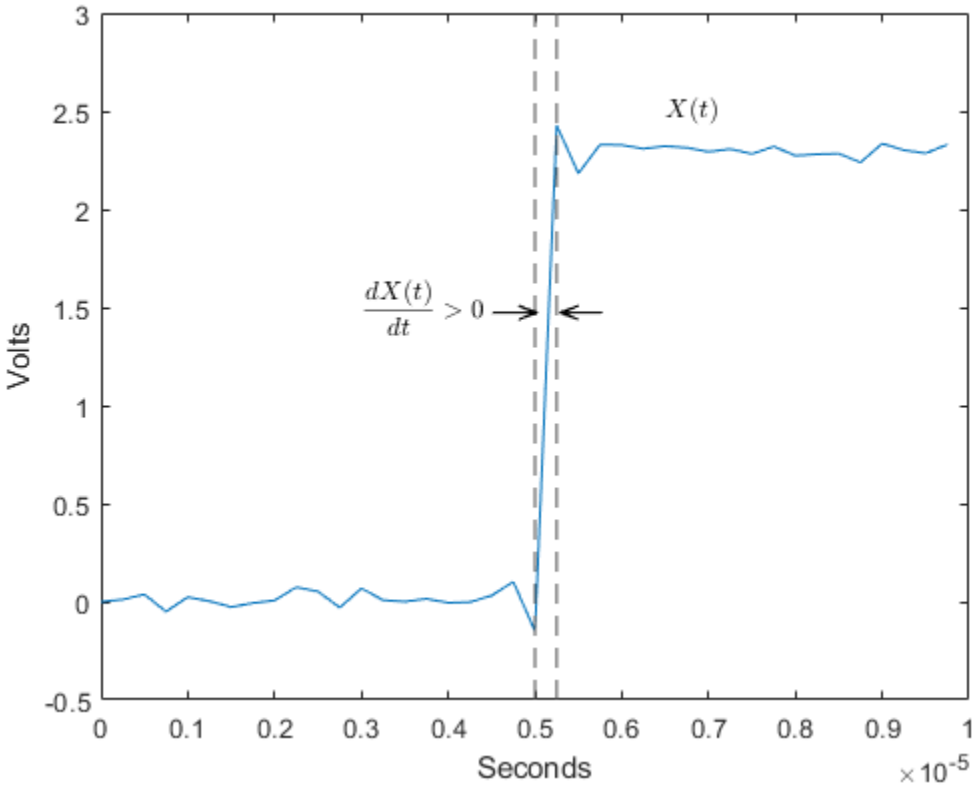
`ul` — Upper reference level
real numeric scalar

Upper reference level in waveform amplitude units, returned as a real numeric scalar. `ul` is a vector containing the waveform values 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.

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. This figure shows a positive-going transition.



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

$$S_1 + \frac{L}{100}(S_2 - S_1)\,.$$

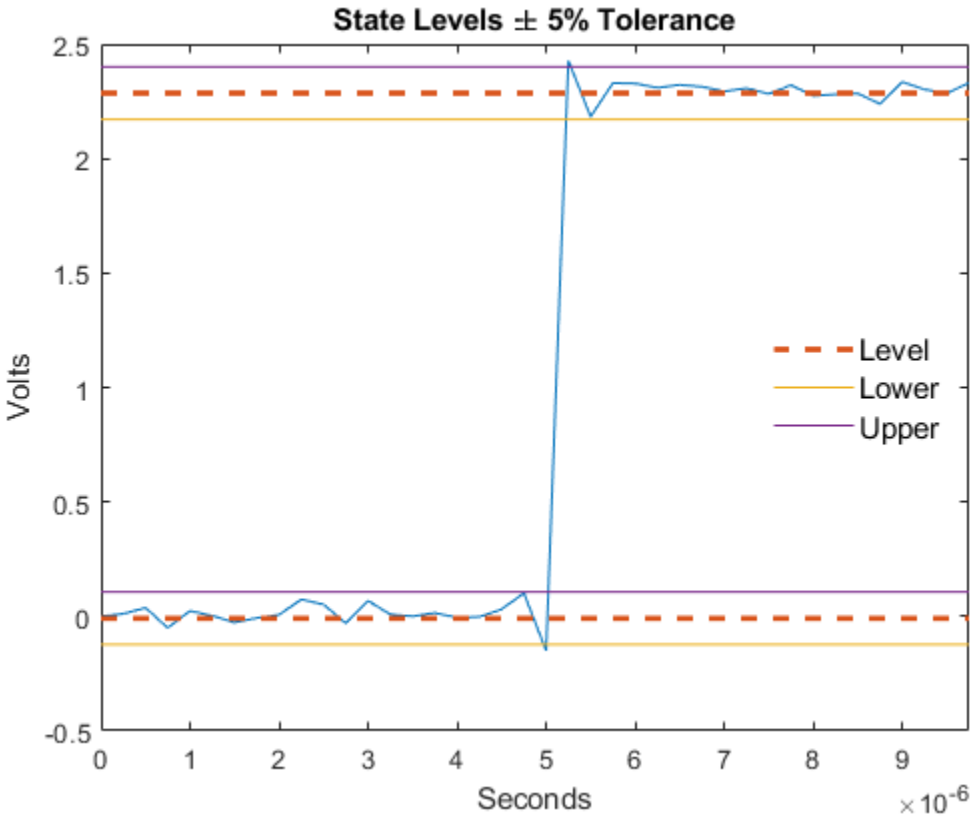
State-Level Tolerances

You can specify lower- and upper-state boundaries for each state level. Define the boundaries as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, specify the scalar as a small number such as 2/100 or 3/100. In general, the $\alpha\%$ 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.

This figure shows lower and upper 5% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The thick dashed lines indicate the estimated state levels.



References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`falltime` | `slewrate` | `statelevels`

Introduced in R2012a

rlevinson

Reverse Levinson-Durbin recursion

Syntax

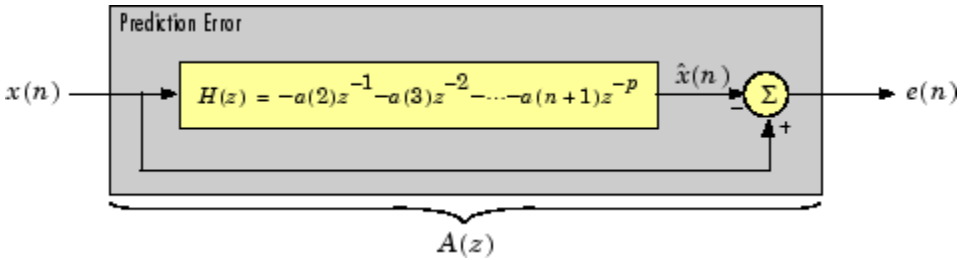
```
r = rlevinson(a,efinal)
[r,u] = rlevinson(a,efinal)
[r,u,k] = rlevinson(a,efinal)
[r,u,k,e] = rlevinson(a,efinal)
```

Description

The reverse Levinson-Durbin recursion implements the step-down algorithm for solving the following symmetric Toeplitz system of linear equations for r, where $r = [r(1) \ldots r(p+1)]$ and $r(i)^*$ denotes the complex conjugate of $r(i)$.

$$\begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \cdots & r(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(p+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(p+1) \end{bmatrix}$$

`r = rlevinson(a,efinal)` solves the above system of equations for r given vector a, where $a = [1\ a(2) \ldots a(p+1)]$. In linear prediction applications, `r` represents the autocorrelation sequence of the input to the prediction error filter, where $r(1)$ is the zero-lag element. The figure below shows the typical filter of this type, where $H(z)$ is the optimal linear predictor, $x(n)$ is the input signal, $\hat{x}(n)$ is the predicted signal, and $e(n)$ is the prediction error.



Input vector a represents the polynomial coefficients of this prediction error filter in descending powers of z.

$$A(z) = 1 + a(2)z^{-1} + \cdots + a(n+1)z^{-p}$$

The filter must be minimum-phase to generate a valid autocorrelation sequence. `efinal` is the scalar prediction error power, which is equal to the variance of the prediction error signal, $\sigma^2(e)$.

`[r,u] = rlevinson(a,efinal)` returns upper triangular matrix U from the UDU^* decomposition

$$R^{-1} = UE^{-1}U^*$$

where

$$R = \begin{bmatrix} r(1) & r(2)^* & \cdots & r(p)^* \\ r(2) & r(1) & \cdots & r(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ r(p) & \cdots & r(2) & r(1) \end{bmatrix}$$

and E is a diagonal matrix with elements returned in output `e` (see below). This decomposition permits the efficient evaluation of the inverse of the autocorrelation matrix, R^{-1}.

Output matrix `u` contains the prediction filter polynomial, `a`, from each iteration of the reverse Levinson-Durbin recursion

$$U = \begin{bmatrix} a_1(1)^* & a_2(2)^* & \cdots & a_{p+1}(p+1)^* \\ 0 & a_2(1)^* & \ddots & a_{p+1}(p)^* \\ 0 & 0 & \ddots & a_{p+1}(p-1)^* \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{p+1}(1)^* \end{bmatrix}$$

where $a_i(j)$ is the jth coefficient of the ith order prediction filter polynomial (i.e., step i in the recursion). For example, the 5th order prediction filter polynomial is

```
a5 = u(5:-1:1,5)'
```

Note that `u(p+1:-1:1,p+1)'` is the input polynomial coefficient vector `a`.

`[r,u,k] = rlevinson(a,efinal)` returns a vector `k` of length $p + 1$ containing the reflection coefficients. The reflection coefficients are the conjugates of the values in the first row of `u`.

```
k = conj(u(1,2:end))
```

`[r,u,k,e] = rlevinson(a,efinal)` returns a vector of length $p + 1$ containing the prediction errors from each iteration of the reverse Levinson-Durbin recursion: `e(1)` is the prediction error from the first-order model, `e(2)` is the prediction error from the second-order model, and so on.

These prediction error values form the diagonal of the matrix E in the UDU^* decomposition of R^{-1}.

$$R^{-1} = UE^{-1}U^*$$

Examples

Optimum Autoregressive Model Order

Estimate the spectrum of two sine waves in noise using an autoregressive model. Choose the best model order from a group of models returned by the reverse Levinson-Durbin recursion.

Generate the signal. Specify a sample rate of 1 kHz and a signal duration of 50 seconds. The sinusoids have frequencies of 50 Hz and 55 Hz. The noise has a variance of 0.2^2.

```
Fs = 1000;
t = (0:50e3-1)'/Fs;
x = sin(2*pi*50*t) + sin(2*pi*55*t) + 0.2*randn(50e3,1);
```

Estimate the autoregressive model parameters.

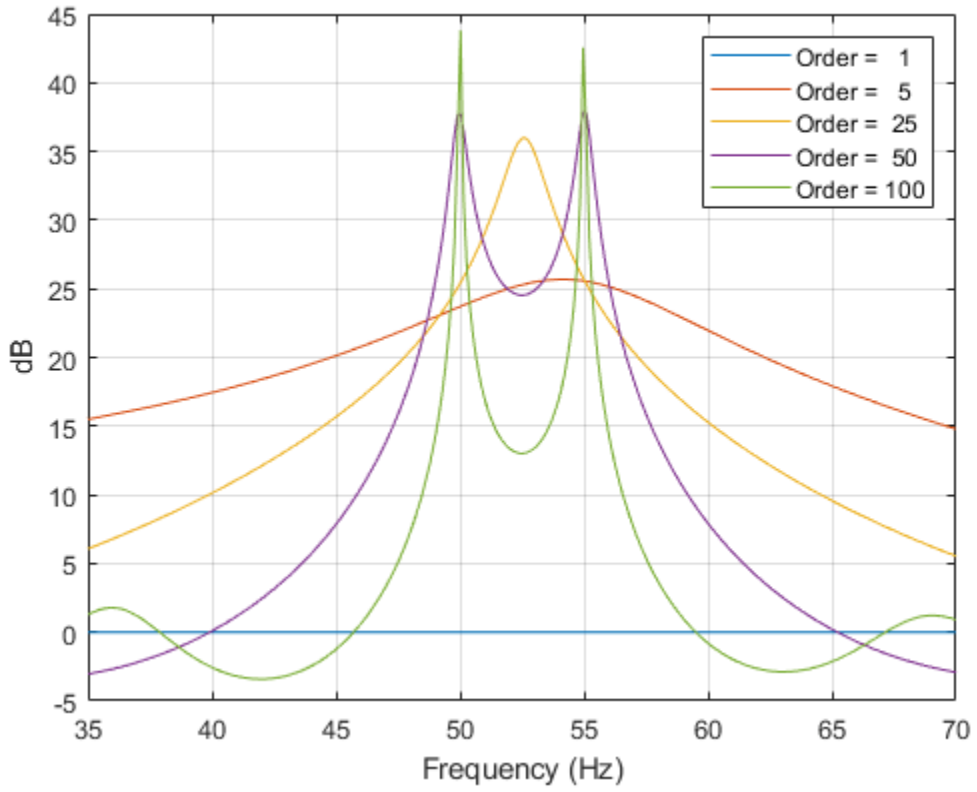
```
[a,e] = arcov(x,100);
[r,u,k] = rlevinson(a,e);
```

Estimate the power spectral density for orders 1, 5, 25, 50, and 100.

```
N = [1 5 25 50 100];
nFFT = 8096;
P = zeros(nFFT,5);

for idx = 1:numel(N)
    order = N(idx);
    ARtest = flipud(u(:,order));
    P(:,idx) = 1./abs(fft(ARtest,nFFT)).^2;
end
```

Plot the PSD estimates.

```
F = (0:1/nFFT:1/2-1/nFFT)*Fs;
plot(F, 10*log10(P(1:length(P)/2,:)))
grid

legend([repmat('Order = ',[5 1]) num2str(N')])
xlabel('Frequency (Hz)')
ylabel('dB')
xlim([35 70])
```



References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

See Also

`levinson` | `lpc` | `prony` | `stmcb`

Introduced before R2006a

rooteig

Frequency and power content using eigenvector method

Syntax

```
[w,pow] = rooteig(x,p)
[w,pow] = rooteig( ___ ,'corr')
[f,pow] = rooteig( ___ ,fs)
```

Description

`[w,pow] = rooteig(x,p)` estimates the frequency content in the input signal `x` and returns `w`, a vector of frequencies in rad/sample, and the corresponding signal power in the vector `pow`. You can specify the signal subspace dimension using the input argument `p`.

The extra threshold parameter in the second entry in `p` provides you more flexibility and control in assigning the noise and signal subspaces.

`[w,pow] = rooteig( ___ ,'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. This syntax can include the input arguments from the previous syntax.

Note You can place `'corr'` anywhere after `p`.

`[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` as the empty vector [], the sampling frequency defaults to `1` Hz.

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 = corrmtx(s,12,'mod');
[W,P] = rooteig(X,3)
```

```
W = 3×1

    0.7883
    1.5674
```

```
    1.0429


P = 3×1

    4.1748
    1.0572
    1.2419
```

Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If `x` is a vector, then it is treated as one observation of the signal. If `x` is a matrix, 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.

For complex-valued input data `x`, `pow` and `w` have the same length. For real-valued input data `x`, the length of the corresponding power vector `pow` is `0.5*length(w)`.

Note You can use the output of `corrmtx` to generate such an array `x`.

p — Subspace dimension

real positive integer | two-element vector

Subspace dimension, specified as a real positive integer or a two-element vector. If `p` is a real positive integer, then it is treated as the subspace dimension. If `p` is a two-element vector, the second element of `p` represents a threshold that is multiplied by λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold λ_{min}*`p(2)` are assigned to the noise subspace. In this case, `p(1)` specifies the maximum dimension of the signal subspace. The extra threshold parameter in the second entry in `p` provides you more flexibility and control in assigning the noise and signal subspaces.

fs — Sample rate

1 (default) | positive scalar | []

Sample rate, specified as a positive scalar. You can supply the sample rate `fs`in Hz. If you specify `fs` as the empty vector [], the sample rate defaults to `1` Hz.

Output Arguments

w — Output frequencies in rad/sample

vector

Output frequencies in rad/sample, returned as a vector. The length of the vector `w` is the computed dimension of the signal subspace.

pow — Signal power

vector

Signal power, returned as a vector.

`f` — Output frequencies in Hz
vector

Output frequencies in Hz, returned as a vector. You supply the sampling frequency `fs` in Hz. If you specify `fs` with the empty vector [], the sampling frequency defaults to `1` Hz.

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 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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Generated code might return outputs in a different sorted order compared to MATLAB.

See Also
`corrmtx` | `peig` | `pmusic` | `rootmusic`

Introduced before R2006a

rootmusic

Root MUSIC algorithm

Syntax

```
w = rootmusic(x,p)
[w,pow] = rootmusic(x,p)
[w,pow] = rootmusic( ___ ,'corr')
[f,pow] = rootmusic( ___ ,fs)
```

Description

`w = rootmusic(x,p)` estimates the frequency content in the input signal `x` and returns `w`, a vector of frequencies in rad/sample. You can specify the signal subspace dimension using the input argument `p`.

The extra threshold parameter in the second entry in `p` provides you more flexibility and control in assigning the noise and signal subspaces.

`[w,pow] = rootmusic(x,p)` returns the vector of frequencies `w` and the corresponding signal power in the vector `pow`.

`[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, `x` must be a square matrix, and all of its eigenvalues must be nonnegative. This syntax can include the input arguments from the previous syntax.

Note You can place `'corr'` anywhere after `p`.

`[f,pow] = rootmusic( ___ ,fs)` returns the vector of frequencies `f` calculated in Hz. You supply the sampling frequency `fs` in Hz.

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)';
frqs = [pi/4 pi/4+0.06];

s = 2*exp(1j*frqs(1)*n)+1.5*exp(1j*frqs(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 = 2×1

    0.7946
    0.8917


P = 2×1

    4.1535
    0.7797
```

Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If `x` is a vector, then it is treated as one observation of the signal. If `x` is a matrix, 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.

For complex-valued input data `x`, `pow` and `w` have the same length. For real-valued input data `x`, the length of the corresponding power vector `pow` is `0.5*length(w)`.

Note You can use the output of `corrmtx` to generate such an array `x`.

p — Subspace dimension

real positive integer | two-element vector

Subspace dimension, specified as a real positive integer or a two-element vector. If `p` is a real positive integer, then it is treated as the subspace dimension. If `p` is a two-element vector, the second element of `p` represents a threshold that is multiplied by λ_{min}, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold λ_{min}`*p(2)` are assigned to the noise subspace. In this case, `p(1)` specifies the maximum dimension of the signal subspace. The extra threshold parameter in the second entry in `p` provides you more flexibility and control in assigning the noise and signal subspaces.

fs — Sample rate

1 (default) | positive scalar | []

Sample rate, specified as a positive scalar. You can supply the sample rate `fs`in Hz. If you specify `fs` as the empty vector [], the sample rate defaults to `1` Hz.

Output Arguments

w — Output frequencies in rad/sample

vector

Output frequencies in rad/sample, returned as a vector. The length of the vector `w` is the computed dimension of the signal subspace.

`pow` — Signal power
vector

Signal power, returned as a vector.

`f` — Output frequencies in Hz
vector

Output frequencies in Hz, returned as a vector. You supply the sampling frequency `fs` in Hz. If you specify `fs` with the empty vector [], the sampling frequency defaults to `1` Hz.

Diagnostics

If the input signal `x` is real, and an odd number of sinusoids is specified by `p`, an error message is displayed:

```
Real signals require an even number p of complex sinusoids.
```

Algorithms

The multiple signal classification (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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Generated code might return outputs in a different sorted order compared to MATLAB.

See Also

`corrmtx` | `peig` | `pmusic` | `rooteig`

Introduced before R2006a

rpmfreqmap

Frequency-RPM map for order analysis

Syntax

```
map = rpmfreqmap(x,fs,rpm)
map = rpmfreqmap(x,fs,rpm,res)

map = rpmfreqmap(___,Name,Value)

[map,freq,rpm,time,res] = rpmfreqmap(___)

rpmfreqmap(___)
```

Description

`map = rpmfreqmap(x,fs,rpm)` returns the frequency-RPM map matrix, `map`, that results from performing frequency analysis on the input vector, `x`. `x` is measured at a set `rpm` of rotational speeds expressed in revolutions per minute. `fs` is the sample rate in Hz. Each column of `map` contains root-mean-square (RMS) amplitude estimates of the spectral content present at each value of `rpm`. `rpmfreqmap` uses the short-time Fourier transform to analyze the spectral content of `x`.

`map = rpmfreqmap(x,fs,rpm,res)` specifies the resolution bandwidth of the map in Hz.

`map = rpmfreqmap(___,Name,Value)` specifies options using `Name,Value` pairs in addition to the input arguments in previous syntaxes.

`[map,freq,rpm,time,res] = rpmfreqmap(___)` returns vectors with the frequencies, rotational speeds, and time instants at which the frequency map is computed. It also returns the resolution bandwidth used.

`rpmfreqmap(___)` with no output arguments plots the frequency map as a function of rotational speed and time on an interactive figure. The plot is also known as a *Campbell diagram*.

Examples

Frequency-RPM Map of Chirp with 4 Orders

Create a simulated signal sampled at 600 Hz for 5 seconds. The system that is being tested increases its rotational speed from 10 to 40 revolutions per second during the observation period.

Generate the tachometer readings.

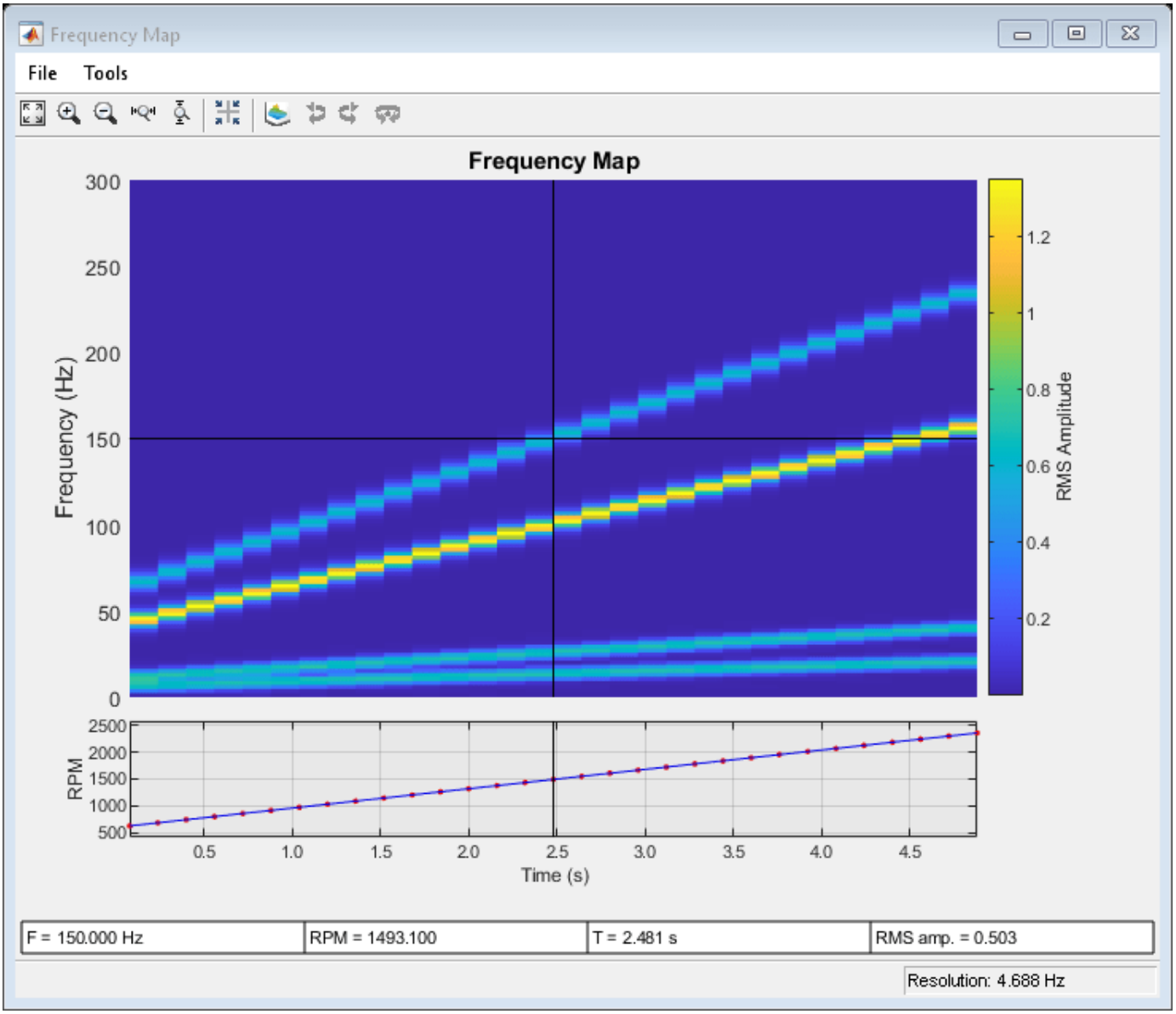
```
fs = 600;
t1 = 5;
t = 0:1/fs:t1;

f0 = 10;
f1 = 40;
rpm = 60*linspace(f0,f1,length(t));
```

The signal consists of four harmonically related chirps with orders 1, 0.5, 4, and 6. The order-4 chirp has twice the amplitude of the others. To generate the chirps, use the trapezoidal rule to express the phase as the integral of the rotational speed.

```
o1 = 1;
o2 = 0.5;
o3 = 4;
o4 = 6;

ph = 2*pi*cumtrapz(rpm/60)/fs;

x = [1 1 2 1]*cos([o1 o2 o3 o4]'*ph);
```

Visualize the frequency-RPM map of the signal.

```
rpmfreqmap(x,fs,rpm)
```



Frequency-RPM Map of Helicopter Vibration Data

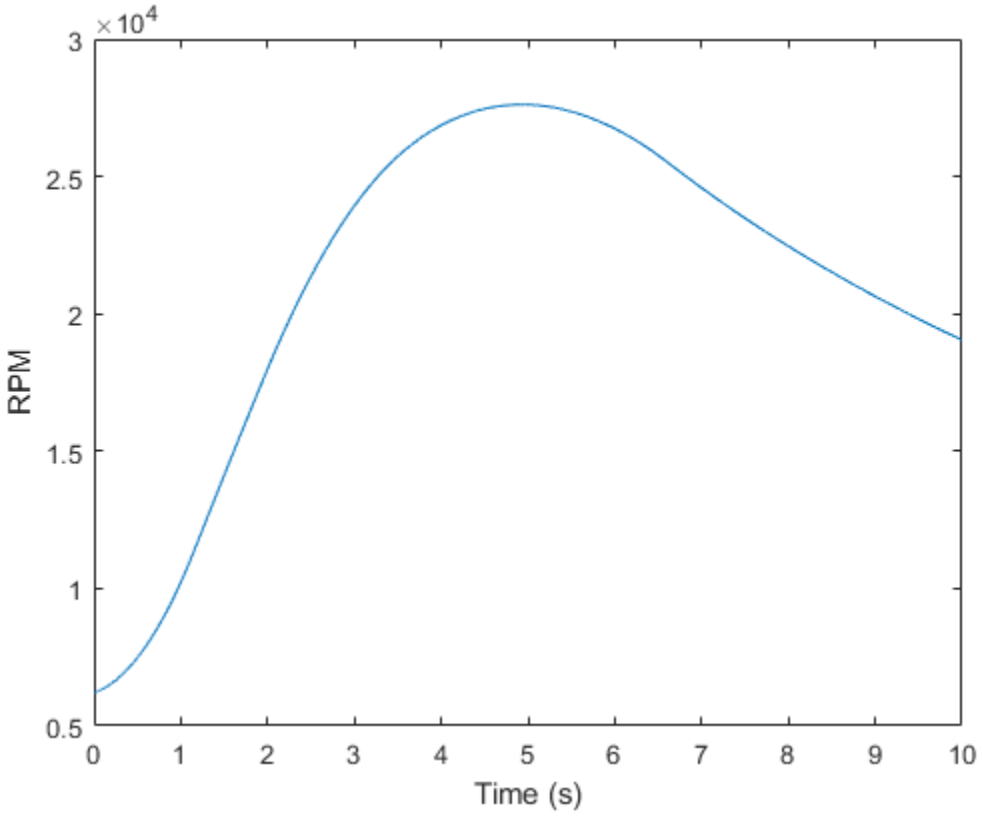
Analyze simulated data from an accelerometer placed in the cockpit of a helicopter.

Load the helicopter data. The vibrational measurements, `vib`, are sampled at a rate of 500 Hz for 10 seconds. Inspection of the data reveals that it has a linear trend. Remove the trend to prevent it from degrading the quality of the frequency estimation.

```
load('helidata.mat')

vib = detrend(vib);
```

Plot the nonlinear RPM profile. The rotor runs up until it reaches a maximum rotational speed of about 27,600 revolutions per minute and then coasts down.

```
plot(t,rpm)
xlabel('Time (s)')
ylabel('RPM')
```

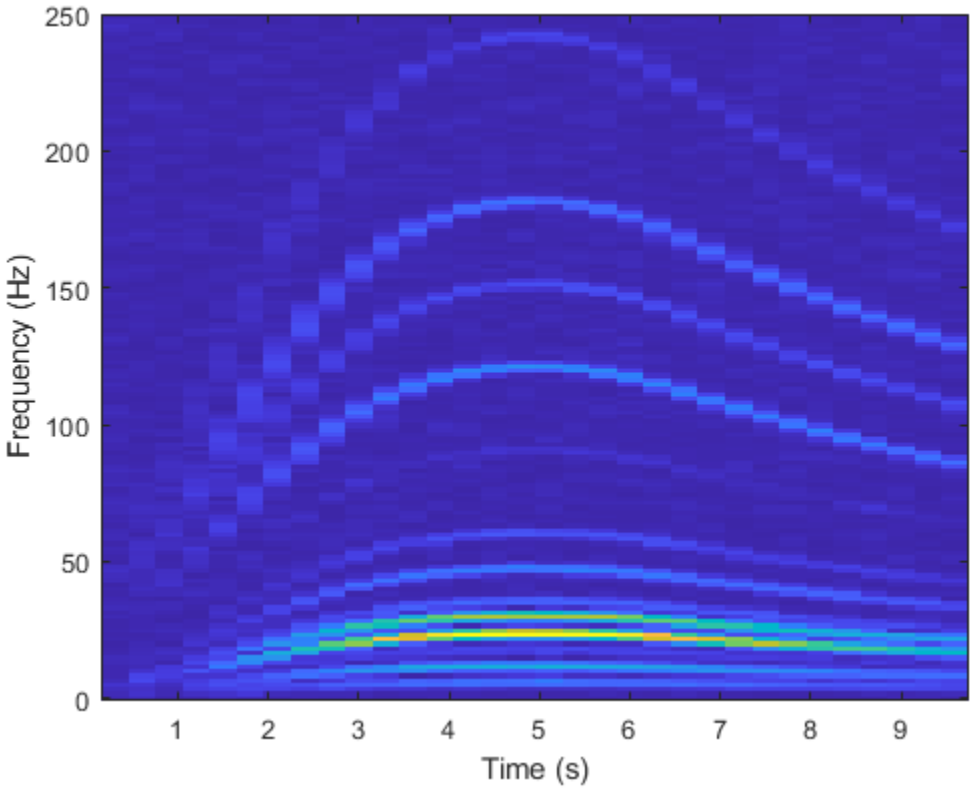


Compute the frequency-RPM map. Specify a resolution bandwidth of 2.5 Hz.

```
[map,freq,rpmOut,time] = rpmfreqmap(vib,fs,rpm,2.5);
```

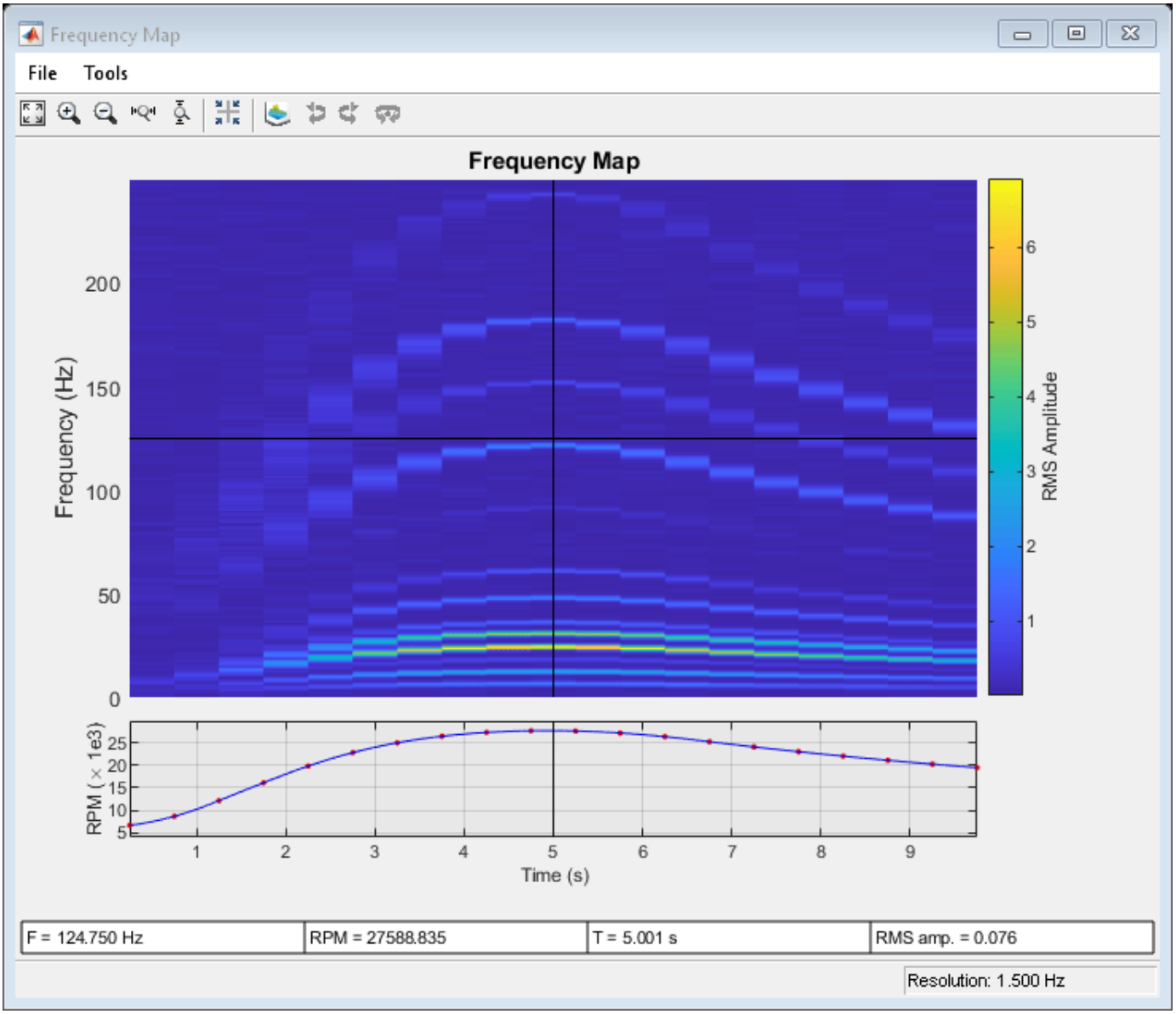
Visualize the map.

```
imagesc(time,freq,map)
ax = gca;
ax.YDir = 'normal';
xlabel('Time (s)')
ylabel('Frequency (Hz)')
```



Repeat the computation using a finer resolution bandwidth. Plot the map using the built-in functionality of `rpmfreqmap`. The gain in frequency resolution comes at the expense of time resolution.

```
rpmfreqmap(vib,fs,rpm,1.5);
```



Waterfall Plot of Frequency-RPM Map

Generate a signal that consists of two linear chirps and a quadratic chirp, all sampled at 600 Hz for 15 seconds. The system that produces the signal increases its rotational speed from 10 to 40 revolutions per second during the testing period.

Generate the tachometer readings.

```
fs = 600;
t1 = 15;
t = 0:1/fs:t1;

f0 = 10;
```

```
f1 = 40;
rpm = 60*linspace(f0,f1,length(t));
```

The linear chirps have orders 1 and 2.5. The component with order 1 has half the amplitude of the other. The quadratic chirp starts at order 6 and returns to this order at the end of the measurement. Its amplitude is 0.8. Create the signal using this information.

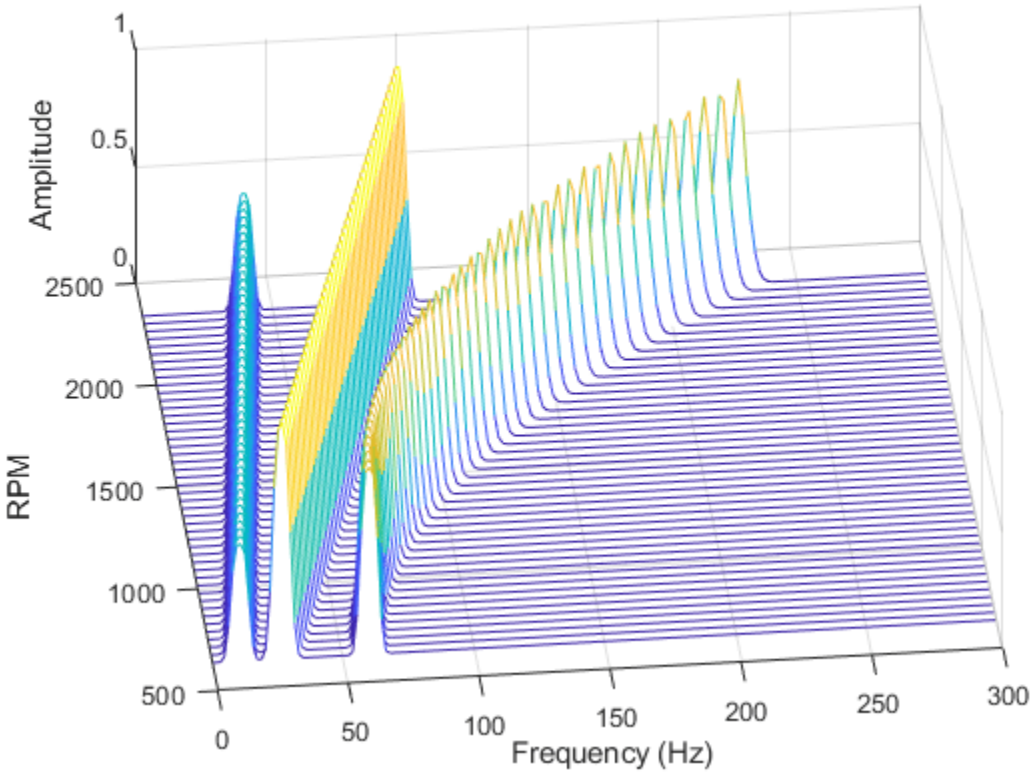
```
o1 = 1;
o2 = 2.5;
o6 = 6;

x = 0.5*chirp(t,o1*f0,t1,o1*f1)+chirp(t,o2*f0,t1,o2*f1) + ...
    0.8*chirp(t,o6*f0,t1,o6*f1,'quadratic');
```

Compute the frequency-RPM map of the signal. Use the peak amplitude at each measurement cell. Specify a resolution of 6 Hz. Window the data with a flat top window.

```
[map,fr,rp] = rpmfreqmap(x,fs,rpm,6, ...
    'Amplitude','peak','Window','flattopwin');
```

Draw the frequency-RPM map as a waterfall plot.

```
[FR,RP] = meshgrid(fr,rp);
waterfall(FR,RP,map')

view(-6,60)
xlabel('Frequency (Hz)')
ylabel('RPM')
zlabel('Amplitude')
```

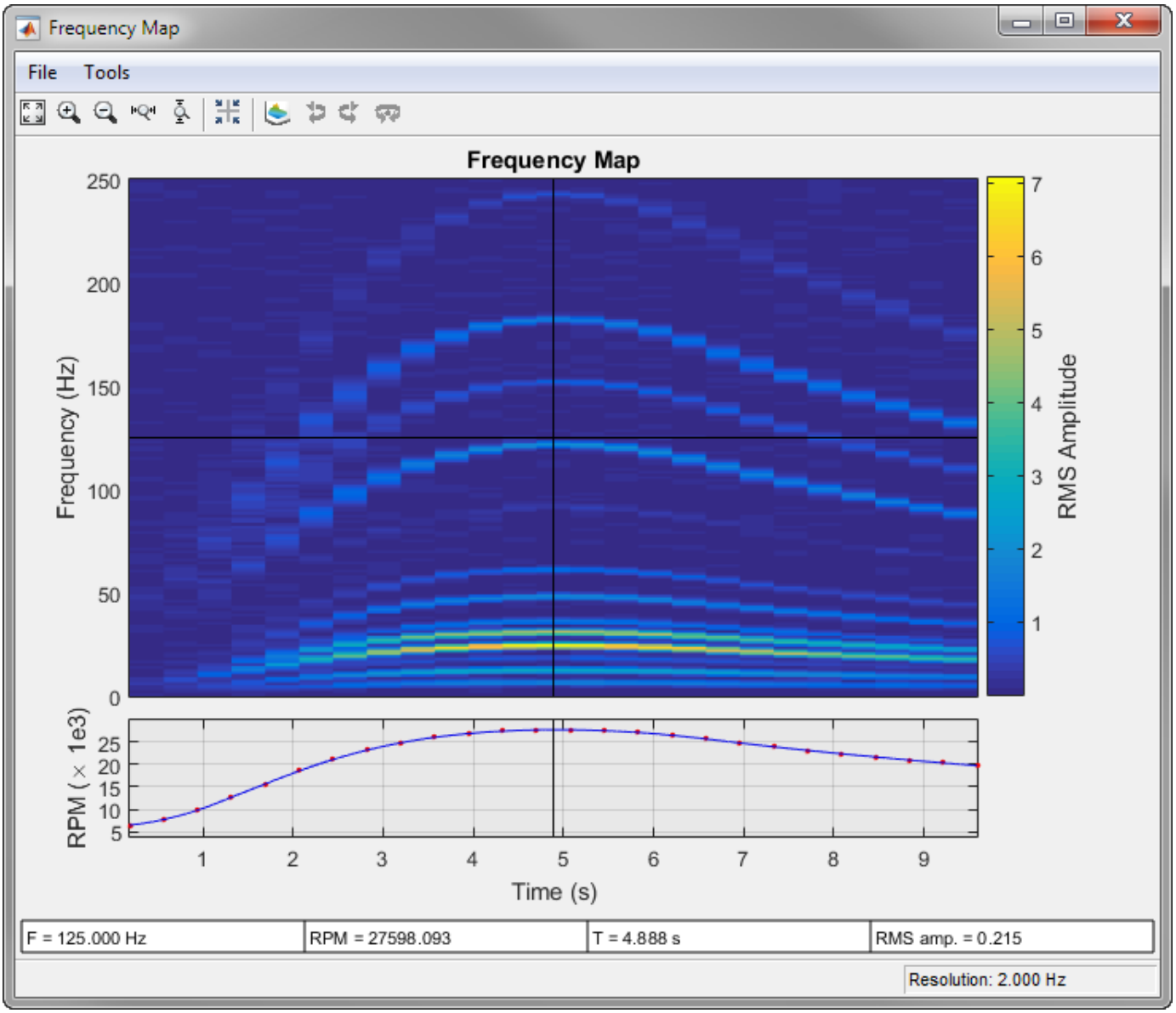


Interactive Frequency-RPM Map

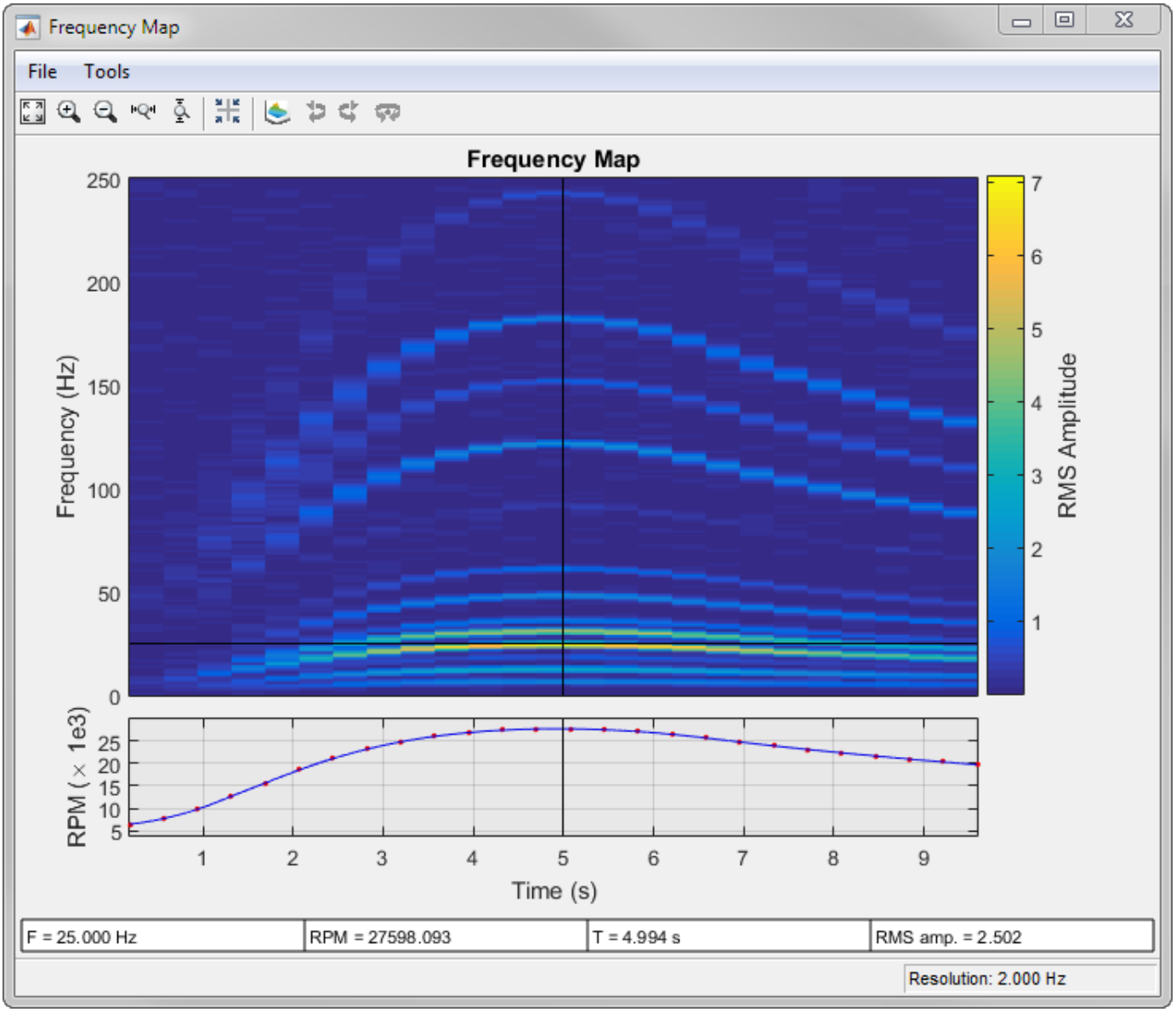
Plot an interactive frequency-RPM map by calling `rpmfreqmap` without output arguments.

Load a file containing simulated vibrational data from an accelerometer placed in the cockpit of a helicopter. The data is sampled at a rate of 500 Hz for 10 seconds. Remove the linear trend in the data. Call `rpmfreqmap` to generate an interactive plot of the frequency-RPM map. Specify a frequency resolution of 2 Hz.

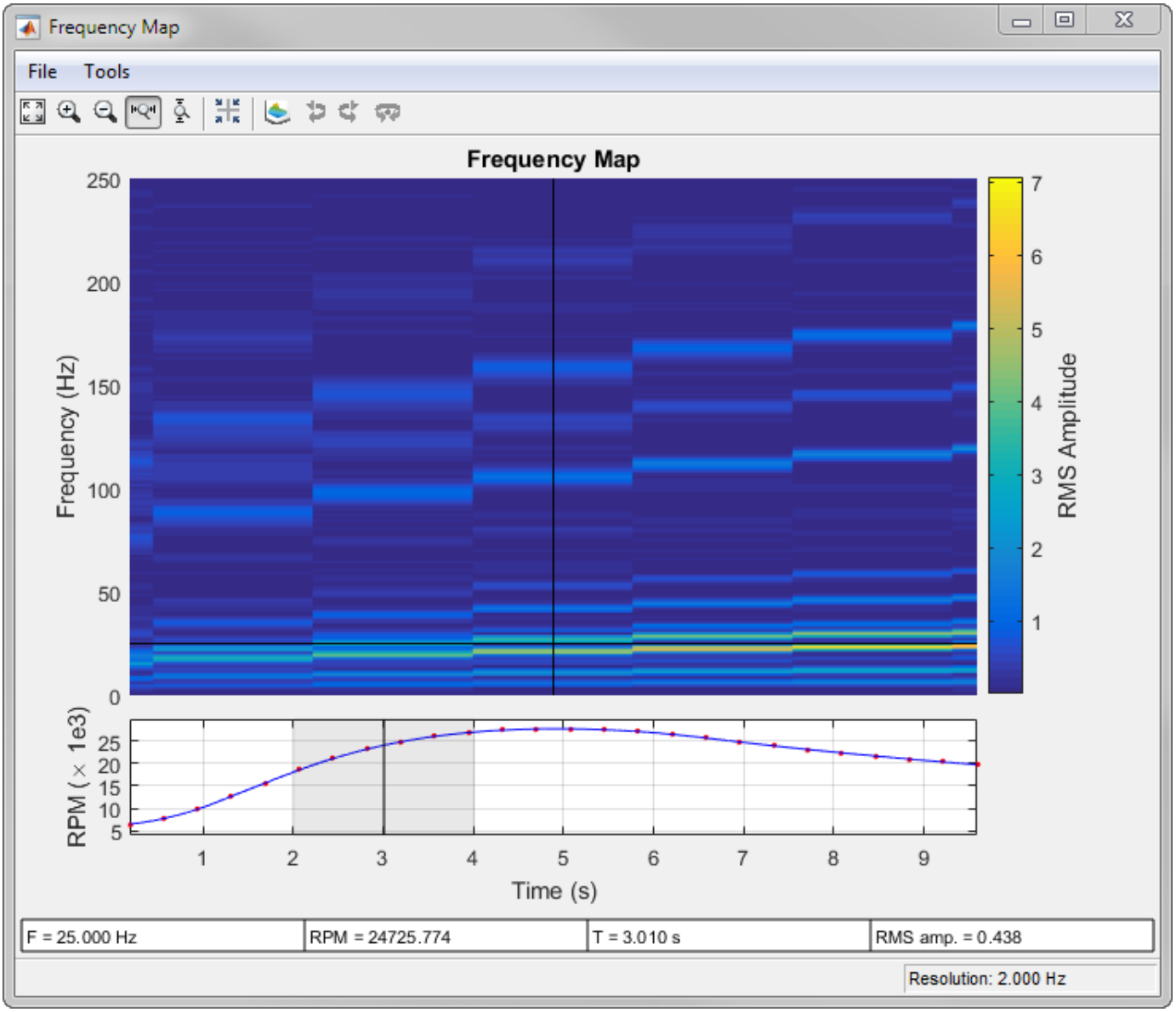
```
load helidata.mat
rpmfreqmap(detrend(vib),fs,rpm,2)
```



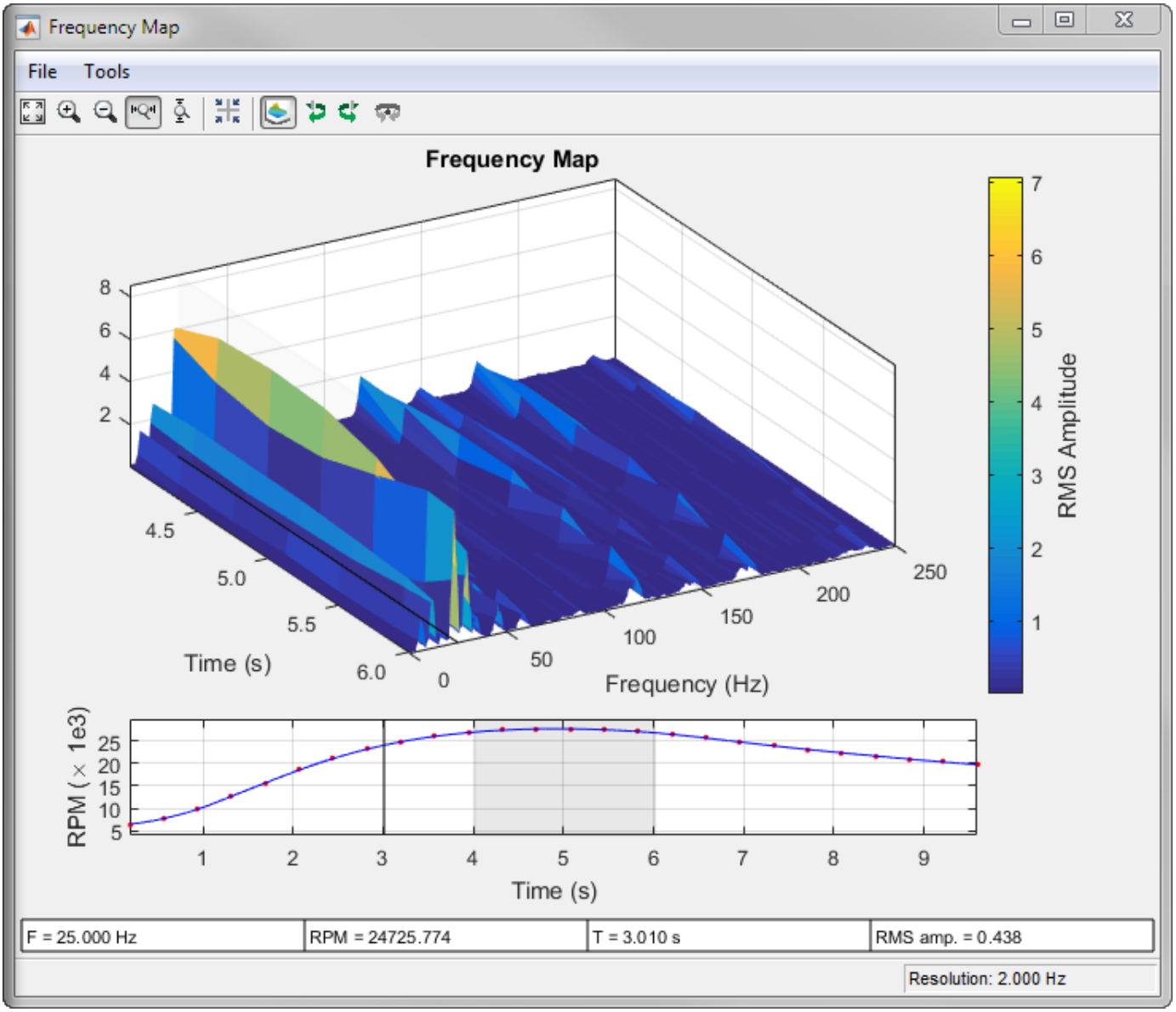
Move the crosshair cursors in the figure to determine the RPM and the RMS amplitude at a frequency of 25 Hz after 5 seconds.



Click the **Zoom X** button in the toolbar to zoom into the time region between 2 and 4 seconds. A panner appears in the bottom plot.



Click the **Waterfall Plot** button in the toolbar to display the frequency-RPM map as a waterfall plot. For improved visibility, rotate the plot clockwise using the **Rotate Left** button three times. Move the panner to the interval between 4 and 6 seconds.



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.

fs — Sample rate

positive scalar

Sample rate, specified as a positive scalar expressed in Hz.

`rpm` — Rotational speeds

vector of positive values

Rotational speeds, specified as a vector of positive values expressed in revolutions per minute. `rpm` must have the same length as `x`.

- If you have a tachometer pulse signal, use `tachorpm` to extract `rpm` directly.
- If you do not have a tachometer pulse signal, use `rpmtrack` to extract `rpm` from a vibration signal.

Example: `100:10:3000` specifies that a system rotates initially at 100 revolutions per minute and runs up to 3000 revolutions per minute in increments of 10.

`res` — Resolution bandwidth

`fs`/128 (default) | positive scalar

Resolution bandwidth of the frequency-RPM map, specified as a positive scalar. If `res` is not specified, then `rpmfreqmap` sets it to the sample rate divided by 128. If the signal is not long enough, then the function uses the entire signal length to compute a single frequency estimate.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Scale','dB','Window','hann'` specifies that the frequency map estimates are to be scaled in decibels and determined using a Hann window.

`Amplitude` — Frequency-RPM map amplitudes

`'rms'` (default) | `'peak'` | `'power'`

Frequency-RPM map amplitudes, specified as the comma-separated pair consisting of `'Amplitude'` and one of `'rms'`, `'peak'`, or `'power'`.

- `'rms'` — Returns the root-mean-square amplitude for each estimated frequency.
- `'peak'` — Returns the peak amplitude for each estimated frequency.
- `'power'` — Returns the power level for each estimated frequency.

`OverlapPercent` — Overlap percentage between adjoining segments

50 (default) | scalar from 0 to 100

Overlap percentage between adjoining segments, specified as the comma-separated pair consisting of `'OverlapPercent'` and a scalar from 0 to 100. A value of 0 means that adjoining segments do not overlap. A value of 100 means that adjoining segments are shifted by one sample. A larger overlap percentage produces a smoother map but increases the computation time. See `rpmordermap` for more information.

Data Types: `double` | `single`

`Scale` — Frequency-RPM map scaling

`'linear'` (default) | `'dB'`

Frequency-RPM map scaling, specified as the comma-separated pair consisting of `'Scale'` and either `'linear'` or `'dB'`.

- `'linear'` — Returns a linearly scaled map.
- `'dB'` — Returns a logarithmic map with values expressed in decibels.

`Window` — Analysis window
`'hann'` (default) | `'chebwin'` | `'flattopwin'` | `'hamming'` | `'kaiser'` | `'rectwin'`

Analysis window, specified as the comma-separated pair consisting of `'Window'` and one of these values:

- `'hann'` specifies a Hann window. See `hann` for more details.
- `'chebwin'` specifies a Chebyshev window. Use a cell array to specify a sidelobe attenuation in decibels. The sidelobe attenuation must be greater than 45 dB. If not specified, it defaults to 100 dB. See `chebwin` for more details.
- `'flattopwin'` specifies a flat top window. See `flattopwin` for more details.
- `'hamming'` specifies a Hamming window. See `hamming` for more details.
- `'kaiser'` specifies a Kaiser window. Use a cell array to specify a shape parameter, β. The shape parameter must be a positive scalar. If not specified, it defaults to 0.5. See `kaiser` for more details.
- `'rectwin'` specifies a rectangular window. See `rectwin` for more details.

Example: `'Window','chebwin'` specifies a Chebyshev window with a sidelobe attenuation of 100 dB.

Example: `'Window',{'chebwin',60}` specifies a Chebyshev window with a sidelobe attenuation of 60 dB.

Example: `'Window','kaiser'` specifies a Kaiser window with a shape parameter of 0.5.

Example: `'Window',{'kaiser',1}` specifies a Kaiser window with a shape parameter of 1.

Data Types: `char` | `string` | `cell`

Output Arguments

`map` — Frequency-RPM map
matrix

Frequency-RPM map, returned as a matrix.

`freq` — Frequencies
vector

Frequencies, returned as a vector.

`rpm` — Rotational speeds
vector

Rotational speeds, returned as a vector.

`time` — Time instants
vector

Time instants, returned as a vector.

`res` — Resolution bandwidth
scalar

Resolution bandwidth, returned as a scalar.

References

[1] Brandt, Anders. *Noise and Vibration Analysis: Signal Analysis and Experimental Procedures*. Chichester, UK: John Wiley & Sons, 2011.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`orderspectrum` | `ordertrack` | `orderwaveform` | `rpmordermap` | `rpmtrack` | `spectrogram` | `tachorpm`

Introduced in R2015b

rpmordermap

Order-RPM map for order analysis

Syntax

```
map = rpmordermap(x,fs,rpm)
map = rpmordermap(x,fs,rpm,res)

map = rpmordermap( ___ ,Name,Value)

[map,order,rpm,time,res] = rpmordermap( ___ )

rpmordermap( ___ )
```

Description

`map = rpmordermap(x,fs,rpm)` returns the order-RPM map matrix, `map`, that results from performing order analysis on the input vector, `x`. `x` is measured at a set `rpm` of rotational speeds expressed in revolutions per minute. `fs` is the measurement sample rate in Hz. Each column of `map` contains root-mean-square (RMS) amplitude estimates of the orders present at each `rpm` value. `rpmordermap` resamples `x` to a constant samples-per-cycle rate and uses the short-time Fourier transform to analyze the spectral content of the resampled signal.

`map = rpmordermap(x,fs,rpm,res)` specifies the order resolution of the map in units of orders.

`map = rpmordermap( ___ ,Name,Value)` specifies options using `Name,Value` pairs in addition to the input arguments in previous syntaxes.

`[map,order,rpm,time,res] = rpmordermap( ___ )` returns vectors with the orders, rotational speeds, and time instants at which the order map is computed. It also returns the order resolution used.

`rpmordermap( ___ )` with no output arguments plots the order map as a function of rotational speed and time on an interactive figure.

Examples

Order-RPM Map of Chirp with 4 Orders

Create a simulated signal sampled at 600 Hz for 5 seconds. The system that is being tested increases its rotational speed from 10 to 40 revolutions per second during the observation period.

Generate the tachometer readings.

```
fs = 600;
t1 = 5;
t = 0:1/fs:t1;

f0 = 10;
```

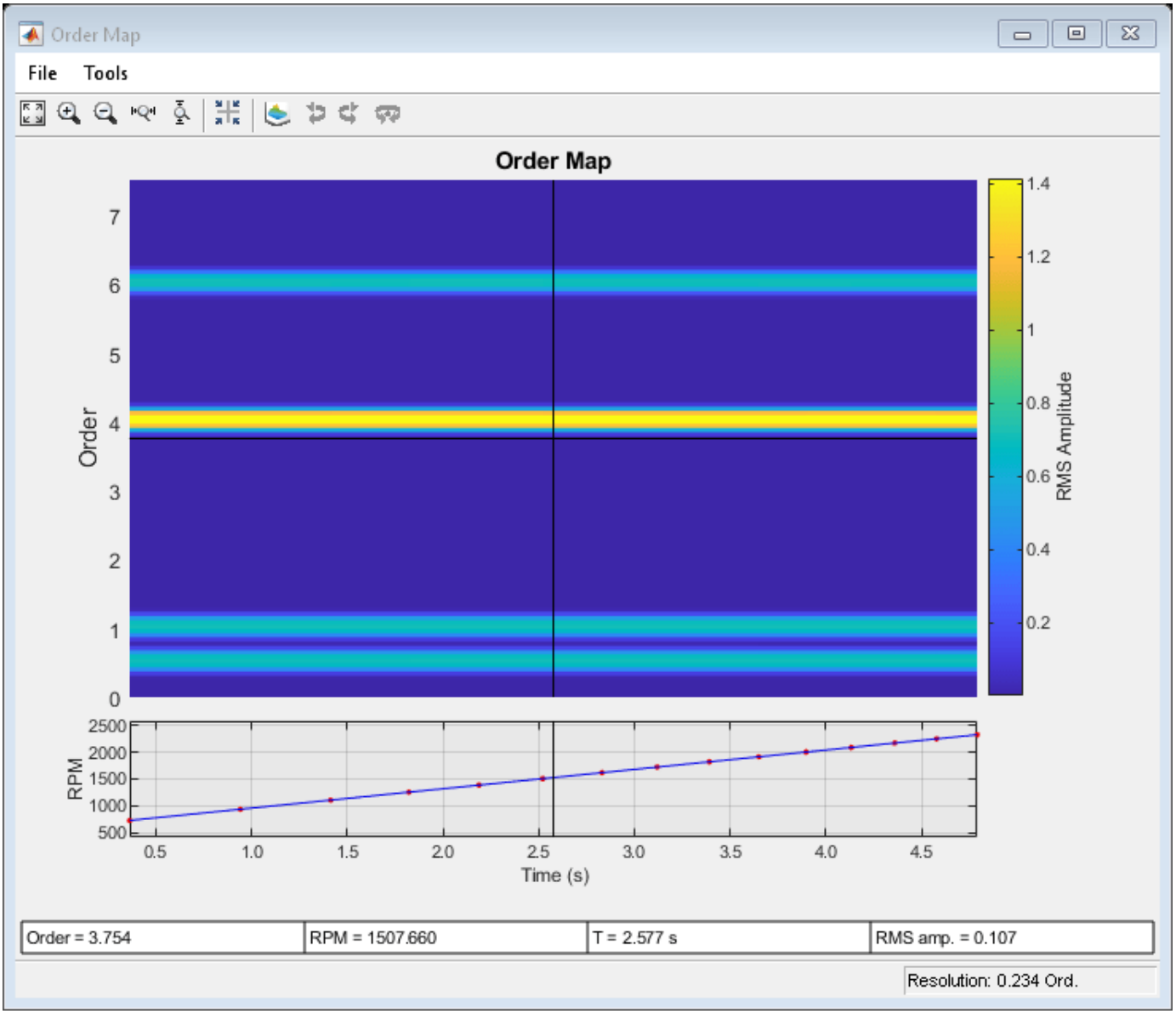
```
f1 = 40;
rpm = 60*linspace(f0,f1,length(t));
```

The signal consists of four harmonically related chirps with orders 1, 0.5, 4, and 6. The order-4 chirp has twice the amplitude of the others. To generate the chirps, use the trapezoidal rule to express the phase as the integral of the rotational speed.

```
o1 = 1;
o2 = 0.5;
o3 = 4;
o4 = 6;

ph = 2*pi*cumtrapz(rpm/60)/fs;

x = [1 1 2 1]*cos([o1 o2 o3 o4]'*ph);
```

Visualize the order-RPM map of the signal.

```
rpmordermap(x,fs,rpm)
```



Order-RPM Map of Helicopter Vibration Data

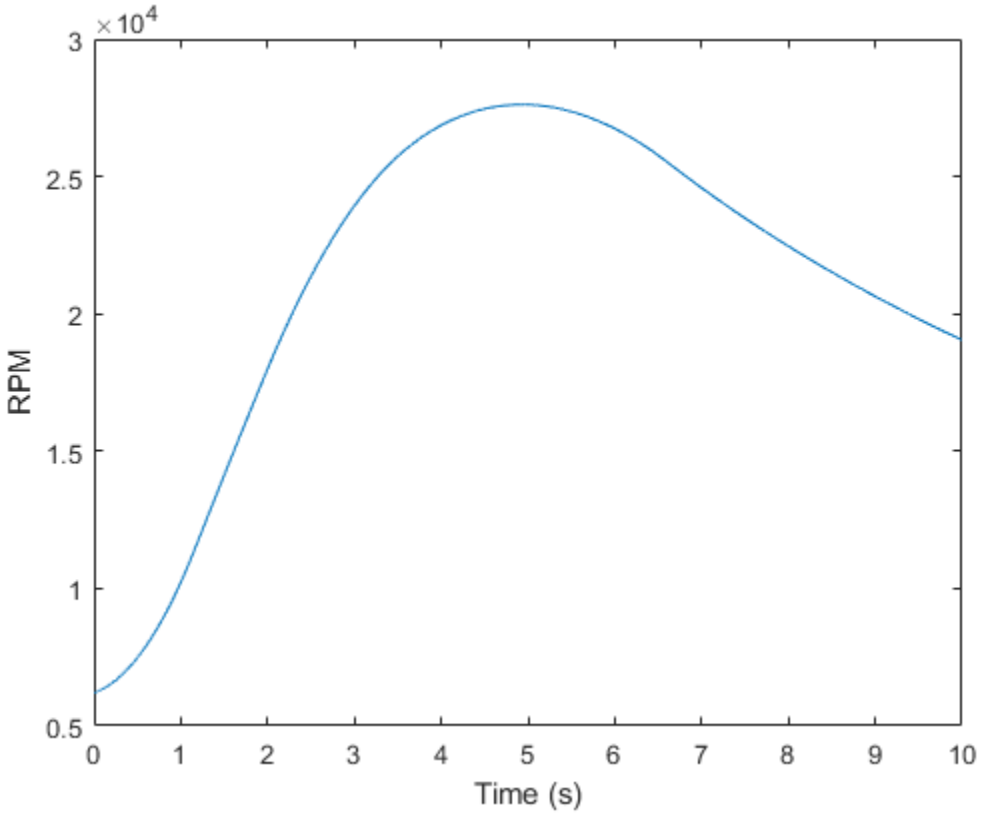
Analyze simulated data from an accelerometer placed in the cockpit of a helicopter.

Load the helicopter data. The vibrational measurements, `vib`, are sampled at a rate of 500 Hz for 10 seconds. Inspection of the data reveals that it has a linear trend. Remove the trend to prevent it from degrading the quality of the order estimation.

```
load('helidata.mat')

vib = detrend(vib);
```

Plot the nonlinear RPM profile. The rotor runs up until it reaches a maximum rotational speed of about 27,600 revolutions per minute and then coasts down.

```
plot(t,rpm)
xlabel('Time (s)')
ylabel('RPM')
```

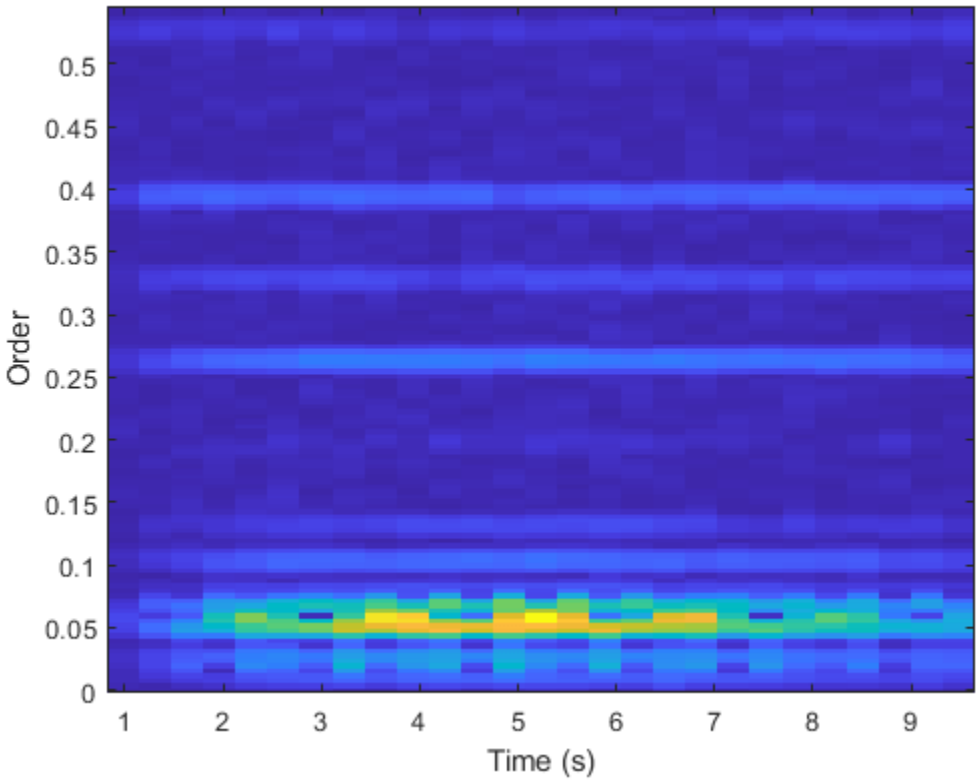


Compute the order-RPM map. Specify an order resolution of 0.015.

```
[map,order,rpmOut,time] = rpmordermap(vib,fs,rpm,0.015);
```

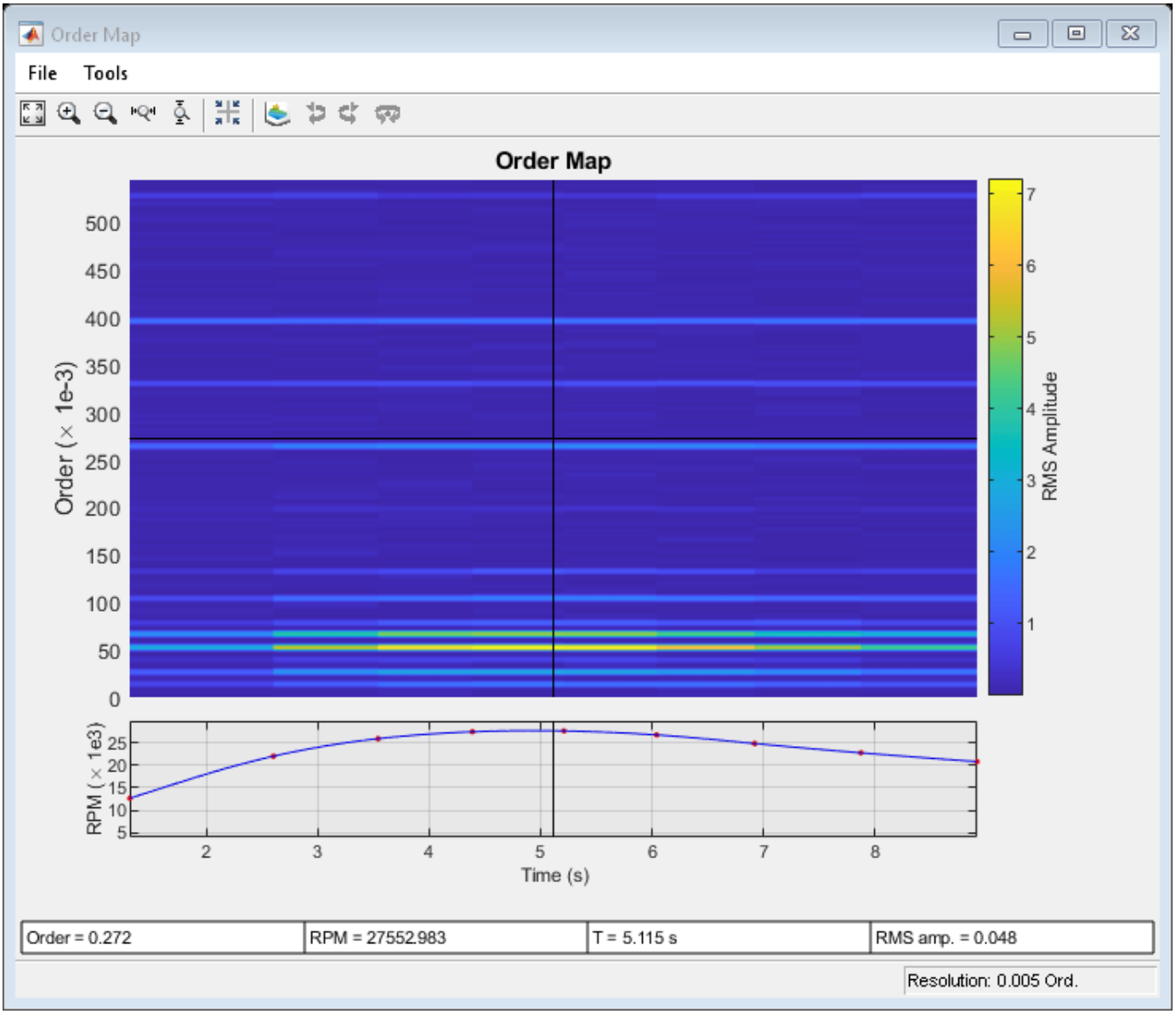
Visualize the map.

```
imagesc(time,order,map)
ax = gca;
ax.YDir = 'normal';
xlabel('Time (s)')
ylabel('Order')
```



Repeat the computation using a finer order resolution. Plot the map using the built-in functionality of `rpmordermap`. The lower orders are resolved more clearly.

```
rpmordermap(vib,fs,rpm,0.005)
```



Waterfall Plot of Order-RPM Map

Generate a signal that consists of two linear chirps and a quadratic chirp, all sampled at 600 Hz for 5 seconds. The system that produces the signal increases its rotational speed from 10 to 40 revolutions per second during the testing period.

Generate the tachometer readings.

```
fs = 600;
t1 = 5;
t = 0:1/fs:t1;

f0 = 10;
```

```
f1 = 40;
rpm = 60*linspace(f0,f1,length(t));
```

The linear chirps have orders 1 and 2.5. The component with order 1 has twice the amplitude of the other. The quadratic chirp starts at order 6 and returns to this order at the end of the measurement. Its amplitude is 0.8. Create the signal using this information.

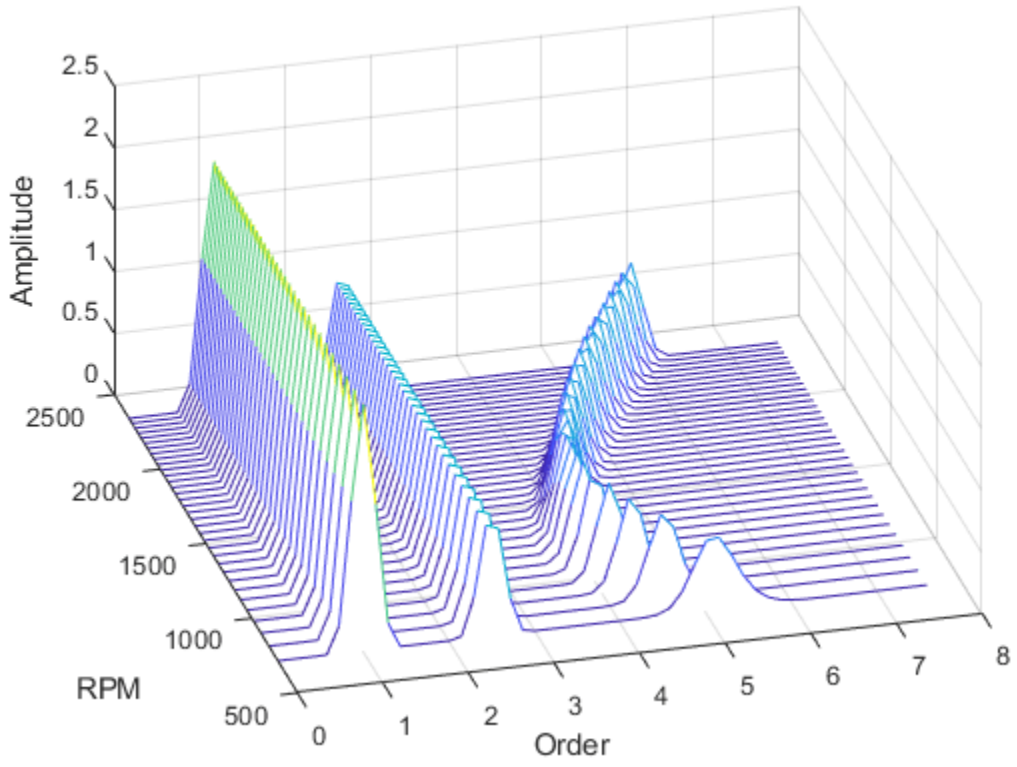
```
o1 = 1;
o2 = 2.5;
o6 = 6;

x = 2*chirp(t,o1*f0,t1,o1*f1)+chirp(t,o2*f0,t1,o2*f1) + ...
    0.8*chirp(t,o6*f0,t1,o6*f1,'quadratic');
```

Compute the order-RPM map of the signal. Use the peak amplitude at each measurement cell. Specify a resolution of 0.25 orders. Window the data with a Chebyshev window whose sidelobe attenuation is 80 dB.

```
[map,or,rp] = rpmordermap(x,fs,rpm,0.25, ...
    'Amplitude','peak','Window',{'chebwin',80});
```

Draw the order-RPM map as a waterfall plot.

```
[OR,RP] = meshgrid(or,rp);
waterfall(OR,RP,map')

view(-15,45)
xlabel('Order')
ylabel('RPM')
zlabel('Amplitude')
```

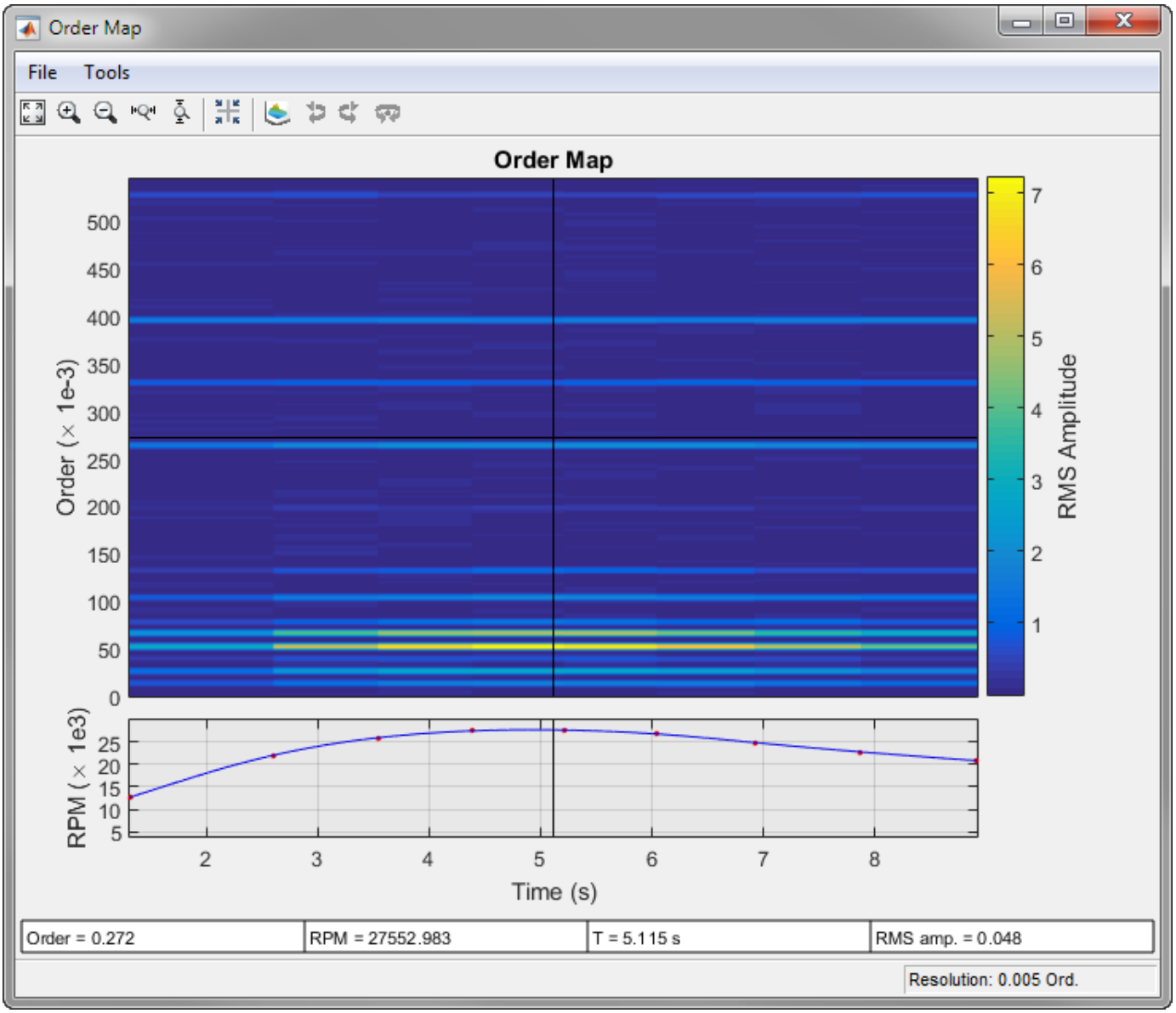


Interactive Order-RPM Map

Plot an interactive order-RPM map by calling `rpmordermap` without output arguments.

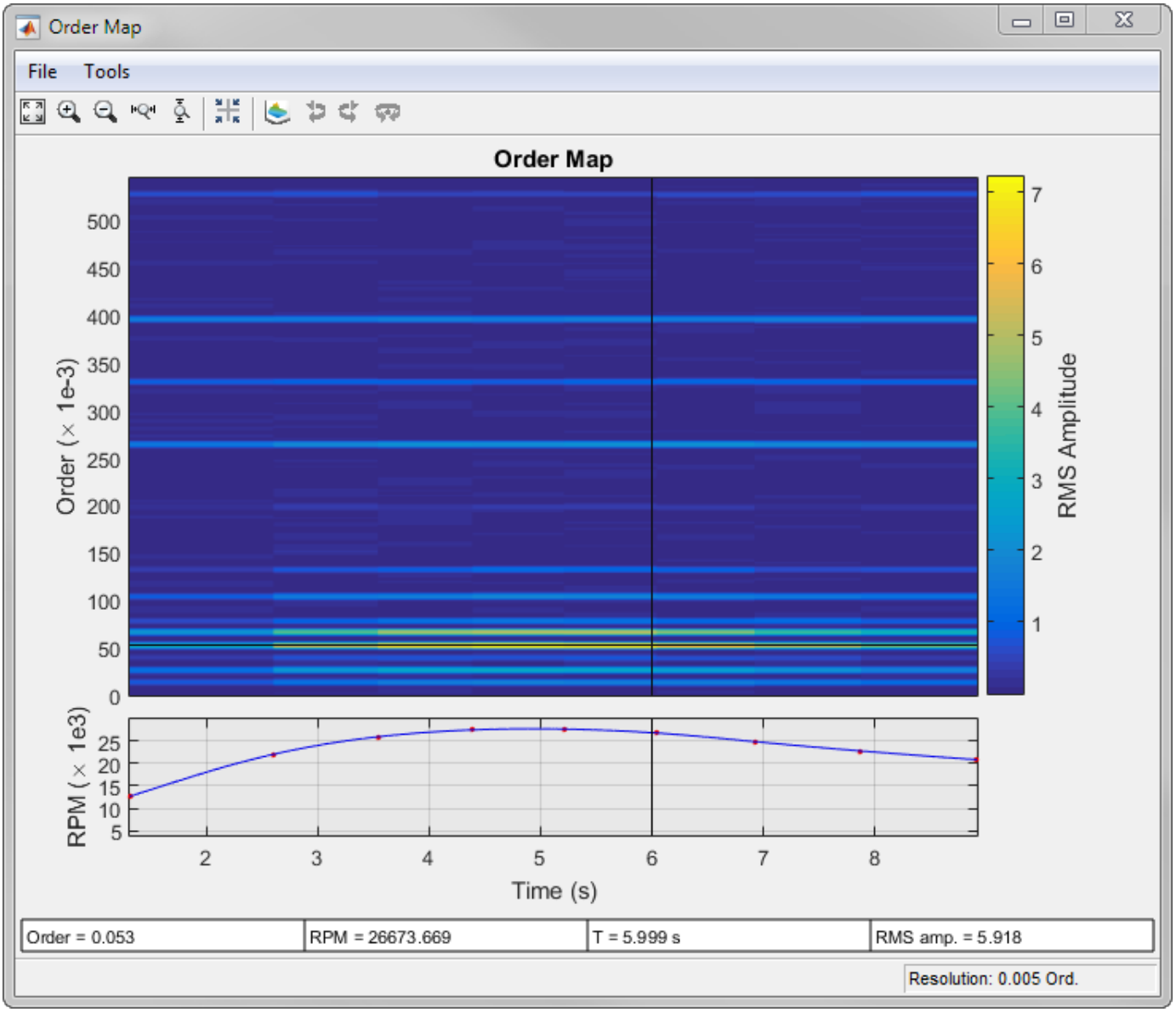
Load the file `helidata.mat`, which contains simulated vibrational data from an accelerometer placed in the cockpit of a helicopter. The data is sampled at a rate of 500 Hz for 10 seconds. Remove the linear trend in the data. Call `rpmordermap` to generate an interactive plot of the order-RPM map. Specify an order resolution of 0.005 orders.

```
load helidata.mat
rpmordermap(detrend(vib),fs,rpm,0.005)
```

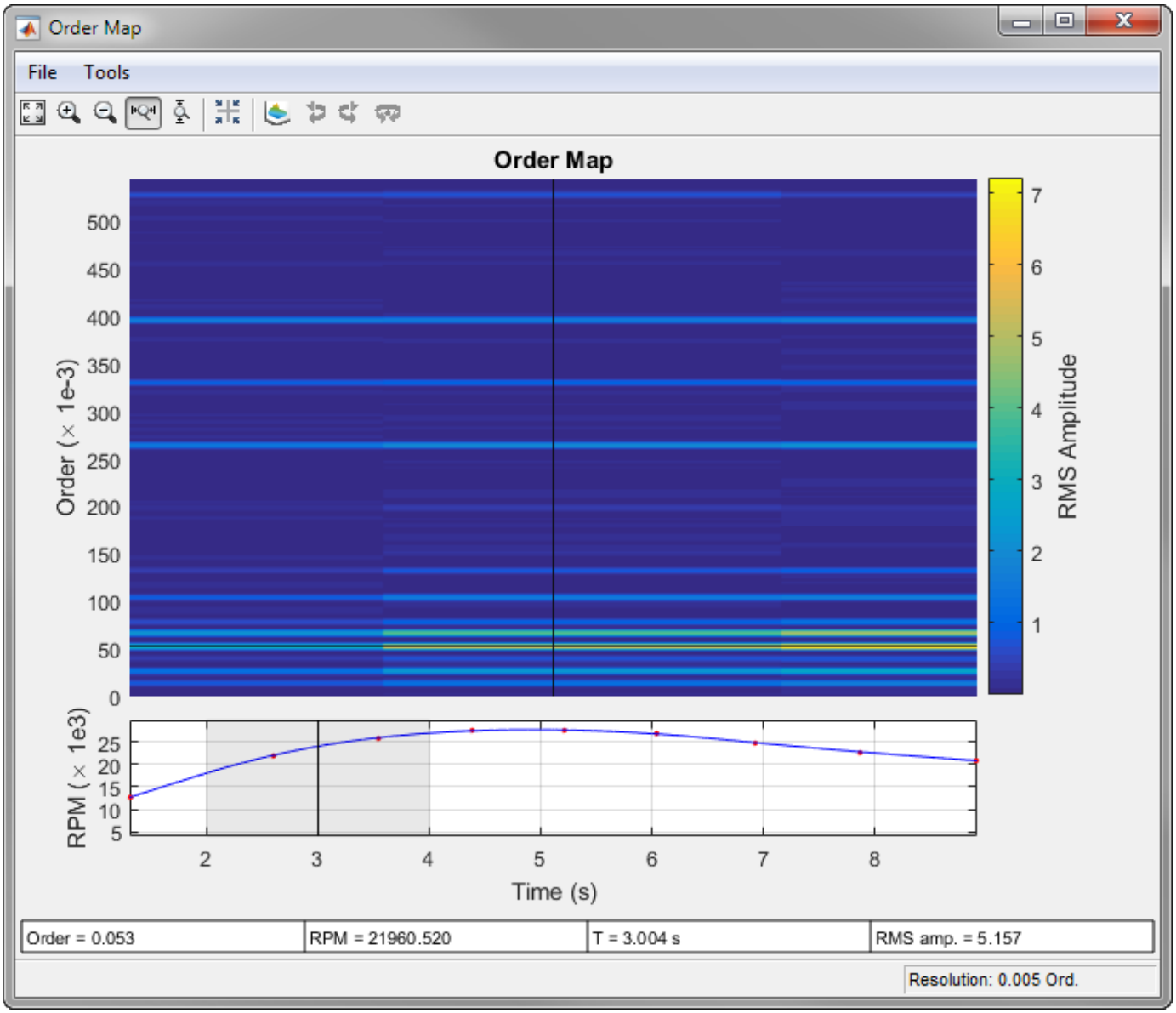


See “Algorithms” on page 1-1954 for a more detailed description of the RPM-vs.-time plot at the bottom of the figure.

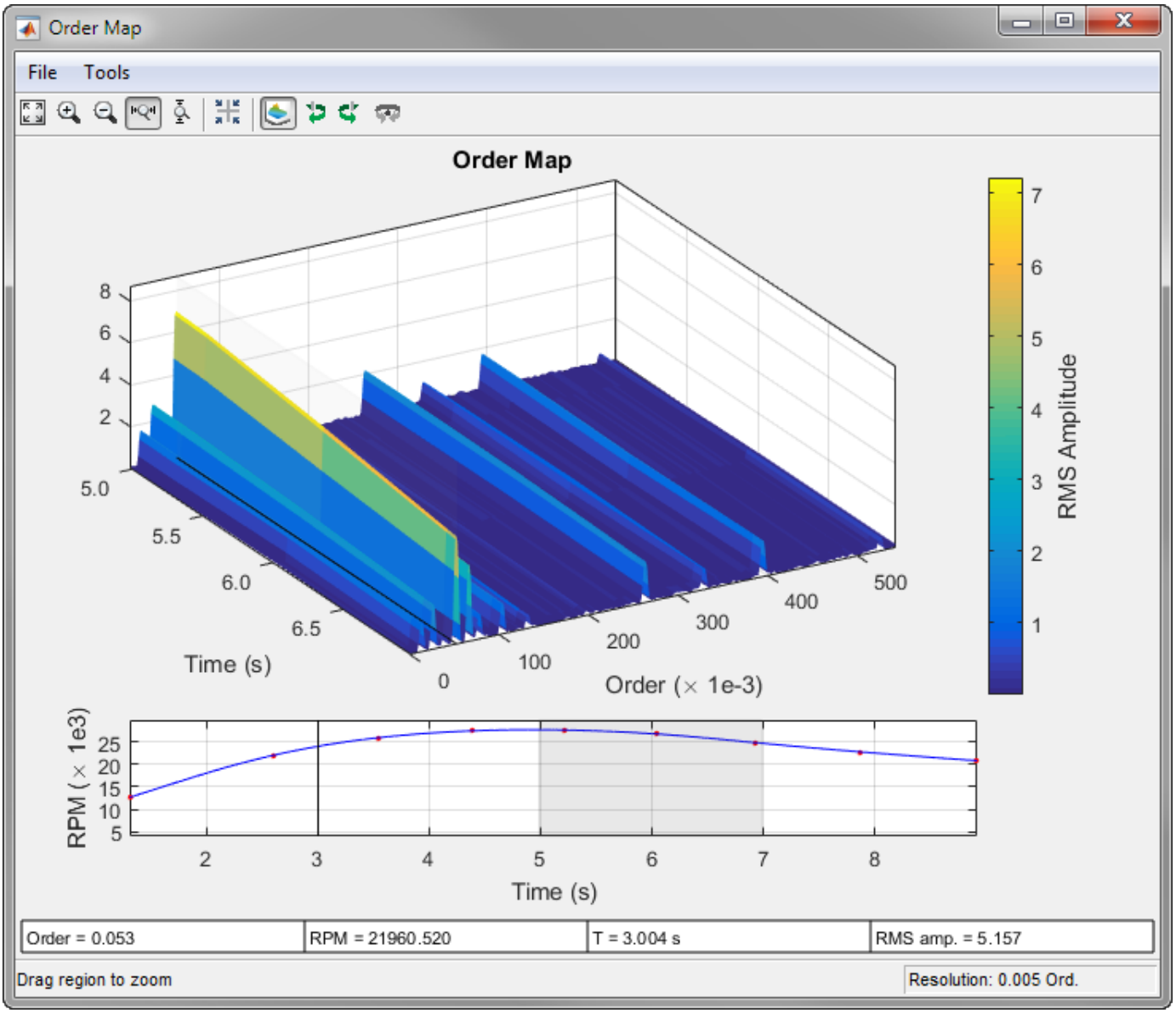
Move the crosshair cursors in the figure to determine the RPM and the RMS amplitude at order 0.053 after 6 seconds.



Click the **Zoom X** button in the toolbar to zoom into the time region between 2 and 4 seconds. The gray rectangle in the RPM-vs.-time plot shows the region of interest. You can slide this region to pan through time.



Click the **Waterfall Plot** button to display the order-RPM map as a waterfall plot. For improved visibility, rotate the plot clockwise using the **Rotate Left** button three times. Move the panner to the interval between 5 and 7 seconds.



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.

fs — Sample rate

positive scalar

Sample rate, specified as a positive scalar expressed in Hz.

`rpm` — Rotational speeds
vector of positive values

Rotational speeds, specified as a vector of positive values expressed in revolutions per minute. `rpm` must have the same length as `x`.

- If you have a tachometer pulse signal, use `tachorpm` to extract `rpm` directly.
- If you do not have a tachometer pulse signal, use `rpmtrack` to extract `rpm` from a vibration signal.

Example: `100:10:3000` specifies that a system rotates initially at 100 revolutions per minute and runs up to 3000 revolutions per minute in increments of 10.

`res` — Order resolution
(15 × `fs`)/(16 × max(`rpm`)) (default) | positive scalar

Order resolution of the order-RPM map, specified as a positive scalar. If `res` is not specified, then `rpmordermap` sets it to the sample rate of the constant-samples-per-cycle signal divided by 256. If the resampled input signal is not long enough, then the function uses the entire resampled signal length to compute a single order estimate.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Scale','dB','Window','hann'` specifies that the order map estimates are to be scaled in decibels and determined using a Hann window.

`Amplitude` — Order-RPM map amplitudes
`'rms'` (default) | `'peak'` | `'power'`

Order-RPM map amplitudes, specified as the comma-separated pair consisting of `'Amplitude'` and one of `'rms'`, `'peak'`, or `'power'`.

- `'rms'` — Returns the root-mean-square amplitude for each estimated order.
- `'peak'` — Returns the peak amplitude for each estimated order.
- `'power'` — Returns the power level for each estimated order.

`OverlapPercent` — Overlap percentage between adjoining segments
50 (default) | scalar from 0 to 100

Overlap percentage between adjoining segments, specified as the comma-separated pair consisting of `'OverlapPercent'` and a scalar from 0 to 100. A value of 0 means that adjoining segments do not overlap. A value of 100 means that adjoining segments are shifted by one sample. A larger overlap percentage produces a smoother map but increases the computation time. See “Algorithms” on page 1-1954 for more information.

Data Types: `double` | `single`

`Scale` — Order-RPM map scaling
`'linear'` (default) | `'dB'`

Order-RPM map scaling, specified as the comma-separated pair consisting of `'Scale'` and either `'linear'` or `'dB'`.

- `'linear'` — Returns a linearly scaled map.
- `'dB'` — Returns a logarithmic map with values expressed in decibels.

`Window` — Analysis window
`'flattopwin'` (default) | `'chebwin'` | `'hamming'` | `'hann'` | `'kaiser'` | `'rectwin'`

Analysis window, specified as the comma-separated pair consisting of `'Window'` and one of these values:

- `'flattopwin'` specifies a flat top window. See `flattopwin` for more details.
- `'chebwin'` specifies a Chebyshev window. Use a cell array to specify a sidelobe attenuation in decibels. The sidelobe attenuation must be greater than 45 dB. If not specified, it defaults to 100 dB. See `chebwin` for more details.
- `'hamming'` specifies a Hamming window. See `hamming` for more details.
- `'hann'` specifies a Hann window. See `hann` for more details.
- `'kaiser'` specifies a Kaiser window. Use a cell array to specify a shape parameter, β. The shape parameter must be a positive scalar. If not specified, it defaults to 0.5. See `kaiser` for more details.
- `'rectwin'` specifies a rectangular window. See `rectwin` for more details.

Example: `'Window','chebwin'` specifies a Chebyshev window with a sidelobe attenuation of 100 dB.

Example: `'Window',{'chebwin',60}` specifies a Chebyshev window with a sidelobe attenuation of 60 dB.

Example: `'Window','kaiser'` specifies a Kaiser window with a shape parameter of 0.5.

Example: `'Window',{'kaiser',1}` specifies a Kaiser window with a shape parameter of 1.

Data Types: `char` | `string` | `cell`

Output Arguments

`map` — Order-RPM map
matrix

Order-RPM map, returned as a matrix.

`order` — Orders
vector

Orders, returned as a vector.

`rpm` — Rotational speeds
vector

Rotational speeds, returned as a vector.

`time` — Time instants
vector

Time instants, returned as a vector.

`res` — Order resolution
scalar

Order resolution, returned as a scalar.

Algorithms

Order analysis is the study of vibrations in rotating systems that result from the rotation itself. The frequencies of these vibrations are often proportional to the rotational speed. The constants of proportionality are the *orders*.

The rotational speed is usually measured independently and changes with time under most experimental conditions. Proper analysis of rotation-induced vibrations requires resampling and interpolating the measured signal to achieve a constant number of samples per cycle. Through this process, the signal components whose frequencies are constant multiples of the rotational speed transform into constant tones. The transformation reduces the smearing of spectral components that occurs when frequency changes rapidly with time.

The `rpmordermap` function performs these steps:

1. Uses `cumtrapz` to estimate the phase angle as the time integral of the rotational speed:

   $$\phi(t) = \int_0^t \frac{\mathrm{RPM}(\tau)}{60}\, d\tau\,.$$

2. Uses `resample` to upsample and lowpass-filter the signal. This step enables the function to interpolate the signal at nonsampled time points without aliasing of the high-frequency components. `rpmordermap` upsamples the signal by a factor of 15.
3. Uses `interp1` to interpolate the upsampled signal linearly onto a uniform grid in the phase domain. The highest accessible order in a measurement is fixed by the sample rate and the highest rotational speed reached by the system:

   $$O_{\max} = \frac{f_s/2}{\max\left(\frac{\mathrm{RPM}}{60}\right)}\,.$$

   To capture this highest order accurately, it is necessary to sample the signal at twice $O_{\max}$ at least. For better results, `rpmordermap` oversamples by an extra factor of 4. The resulting phase-domain sample rate, f_p, is

   $$f_p = 4 \times 2O_{\max} = 4 \times 2\frac{f_s/2}{\max\left(\frac{\mathrm{RPM}}{60}\right)}\,.$$

   The default order resolution, r, is

   $$r = \frac{f_p}{256} = \frac{4 \times 60}{256}\frac{2 \times f_s/2}{\max(\mathrm{RPM})} = \frac{15}{16}\frac{f_s}{\max(\mathrm{RPM})}\,.$$

4. Uses `spectrogram` to compute the short-time Fourier transform (STFT) of the interpolated signal. By default, the function divides the signal into L-sample segments and windows each of them with a flat top window. There are

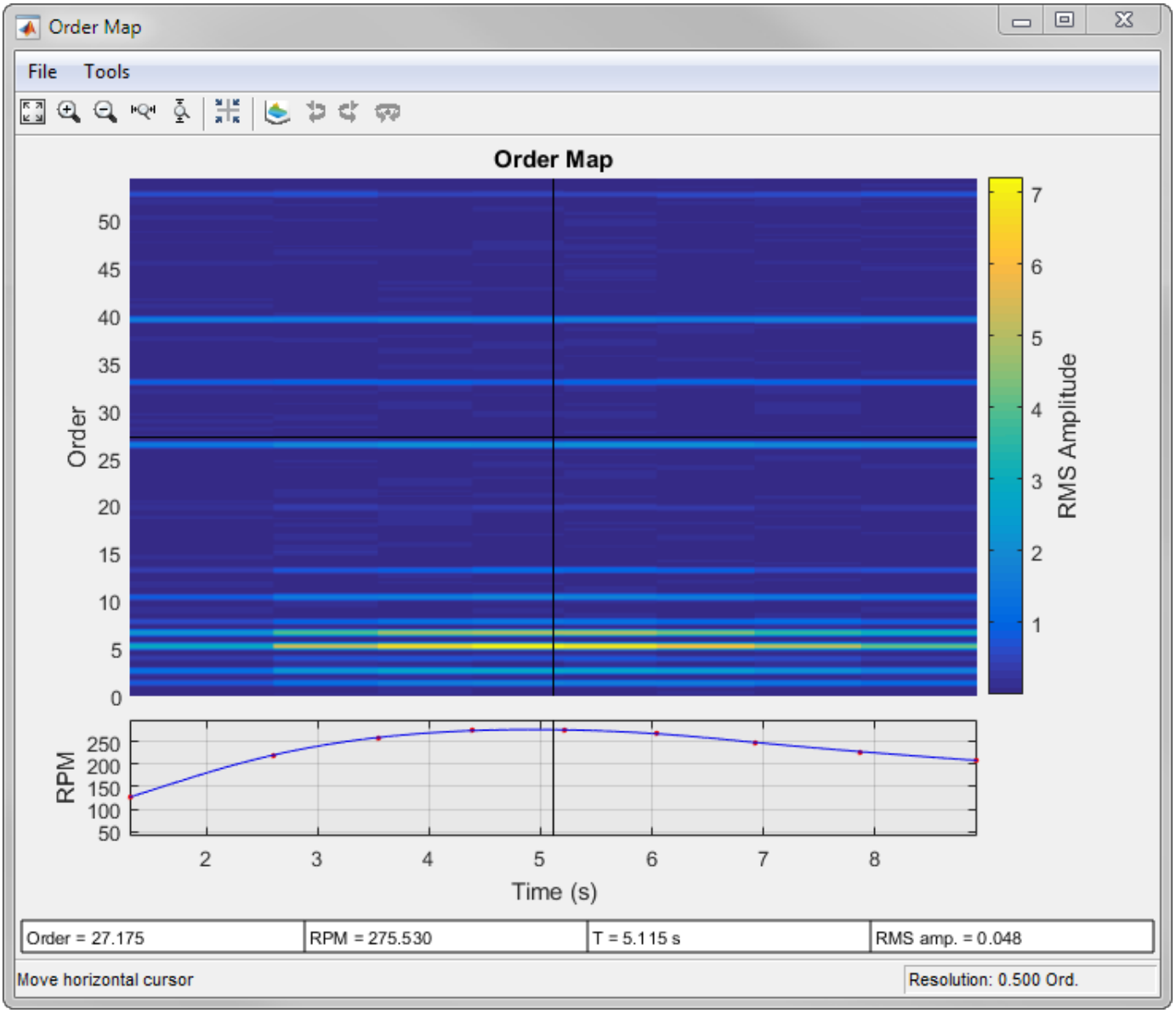
   $$N_{\mathrm{overlap}} = \min\left(\left\lfloor \frac{p_{\mathrm{overlap}}}{100} \times L \right\rfloor, L - 1\right)$$

samples of overlap between adjoining segments, where p_{overlap} is specified using the `'OverlapPercent'` name-value pair and defaults to 50%. The DFT length is set to L. The resolution is related to the sample rate and segment length through

$$r = \frac{kf_p}{L},$$

where k is the equivalent noise bandwidth of the window, as implemented in `enbw`. Adjust the resolution to differentiate closely spaced orders. Smaller r values require larger segment lengths. If you need to attain a given resolution, make sure that your signal has enough samples.

The red dots in the RPM-vs.-time plot at the bottom of the interactive `rpmordermap` window correspond to the right edge of each windowed segment. The blue line in the plot is the RPM signal drawn as a function of time:



References

[1] Brandt, Anders. *Noise and Vibration Analysis: Signal Analysis and Experimental Procedures*. Chichester, UK: John Wiley & Sons, 2011.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`orderspectrum` | `ordertrack` | `orderwaveform` | `rpmfreqmap` | `rpmtrack` | `spectrogram` | `tachorpm`

Introduced in R2015b

rpmtrack

Track and extract RPM profile from vibration signal

Syntax

```
rpm = rpmtrack(x,fs,order,p)
rpm = rpmtrack(xt,order,p)

rpm = rpmtrack(___,Name=Value)

[rpm,tout] = rpmtrack(___)

rpmtrack(___)
```

Description

`rpm = rpmtrack(x,fs,order,p)` returns a time-dependent estimate of the rotational speed `rpm` from a vibration signal `x` sampled at a rate `fs`.

The two-column matrix `p` contains a set of points that lie on a time-frequency ridge corresponding to a given `order`. Each row of `p` specifies one coordinate pair. If you call `rpmtrack` without specifying both `order` and `p`, the function opens an interactive plot that displays the time-frequency map and enables you to select the points.

If you have a tachometer pulse signal, use `tachorpm` to extract `rpm` directly.

`rpm = rpmtrack(xt,order,p)` returns a time-dependent estimate of the rotational speed from a signal stored in the MATLAB timetable `xt`.

`rpm = rpmtrack(___,Name=Value)` specifies additional options for any of the previous syntaxes using name-value arguments. Options include the method used to estimate the time-frequency map and the starting time for the RPM profile.

`[rpm,tout] = rpmtrack(___)` also returns the time vector at which the RPM profile is computed.

`rpmtrack(___)` with no output arguments plots the power time-frequency map and the estimated RPM profile on an interactive figure.

Examples

RPM Profile of Vibration Signal

Generate a vibration signal with three harmonic components. The signal is sampled at 1 kHz for 16 seconds. The signal's instantaneous frequency resembles the runup and coastdown of an engine. Compute the instantaneous phase by integrating the frequency using the trapezoidal rule.

```
fs = 1000;
t = 0:1/fs:16;
```

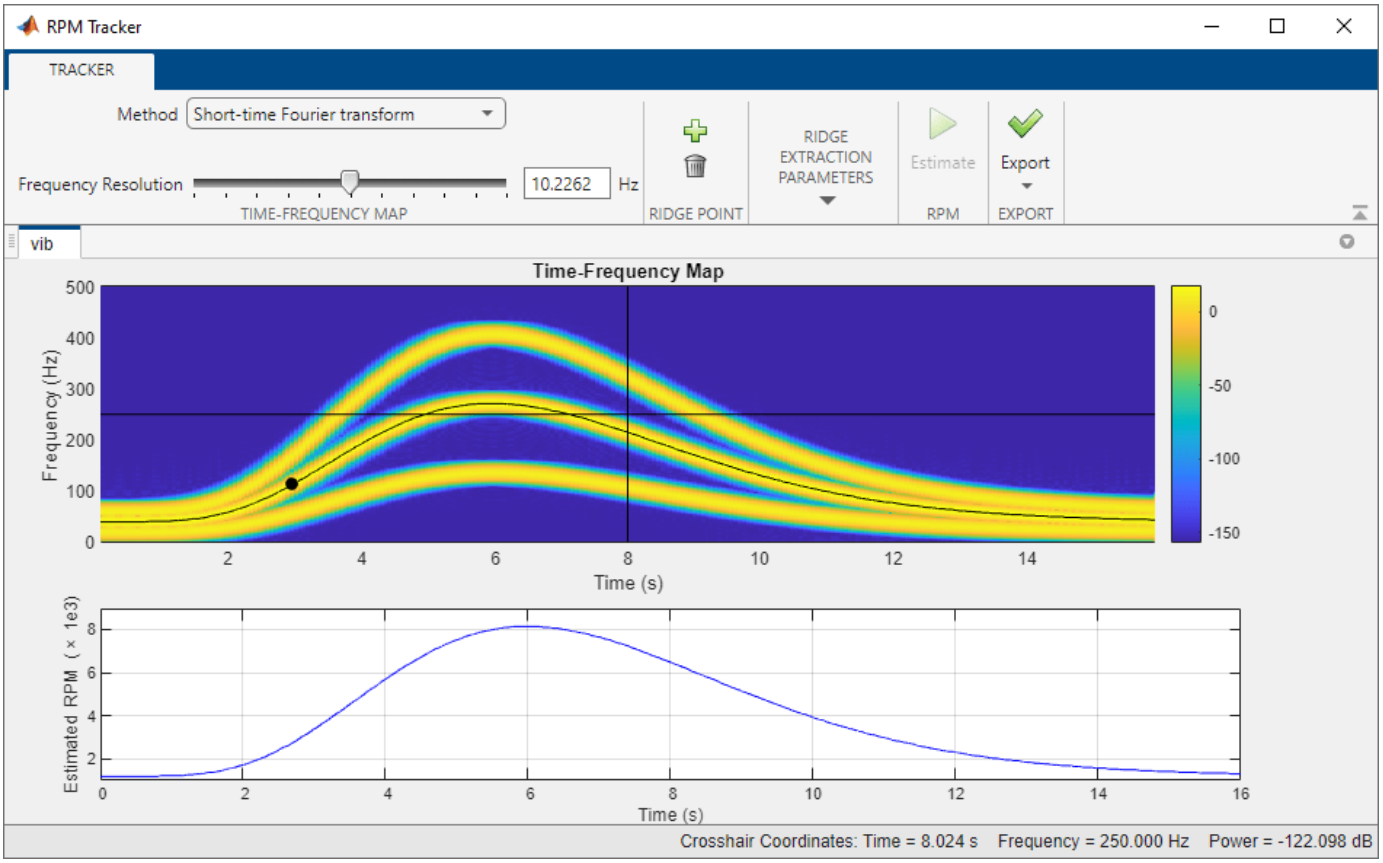
```
ifq = 20 + t.^6.*exp(-t);
phi = 2*pi*cumtrapz(t,ifq);
```

The harmonic components of the signal correspond to orders 1, 2, and 3. The order-2 sinusoid has twice the amplitude of the others.

```
ol = [1 2 3];
amp = [5 10 5];

vib = amp*cos(ol'.*phi);
```

Extract and visualize the RPM profile of the signal using a point on the order-2 ridge.

```
time = 3;
order = 2;
p = [time order*ifq(t==time)];

rpmtrack(vib,fs,order,p)
```

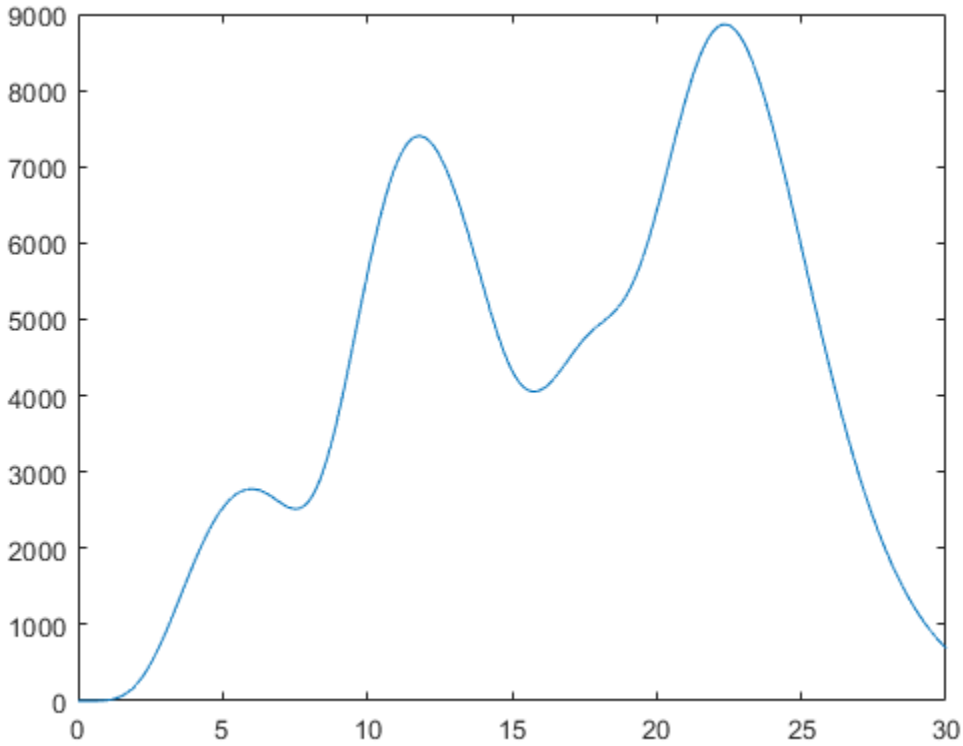


RPM Profile of Revving Engine

Generate a signal that resembles the vibrations caused by revving a car engine. The signal is sampled at 1 kHz for 30 seconds and contains three harmonic components of orders 1, 2.4, and 3, with amplitudes 5, 4, and 0.5, respectively. Embed the signal in unit-variance white Gaussian noise and

store it in a MATLAB® timetable. Multiply the instantaneous frequency by 60 to obtain an RPM profile. Plot the RPM profile.

```
fs = 1000;
t = (0:1/fs:30)';

fit = @(a,x) (t-x).^6.*exp(-(t-x)).*((t-x)>=0)*a';

fis = fit([0.4 1 0.6 1],[0 6 13 17]);
phi = 2*pi*cumtrapz(t,fis);

ol = [1 2.4 3];
amp = [5 4 0.5]';
vib = cos(phi.*ol)*amp + randn(size(t));

xt = timetable(seconds(t),vib);

plot(t,fis*60)
```



Derive the RPM profile from the vibration signal. Use four points at 5 second intervals to specify the ridge corresponding to order 2.4. Display a summary of the output timetable.

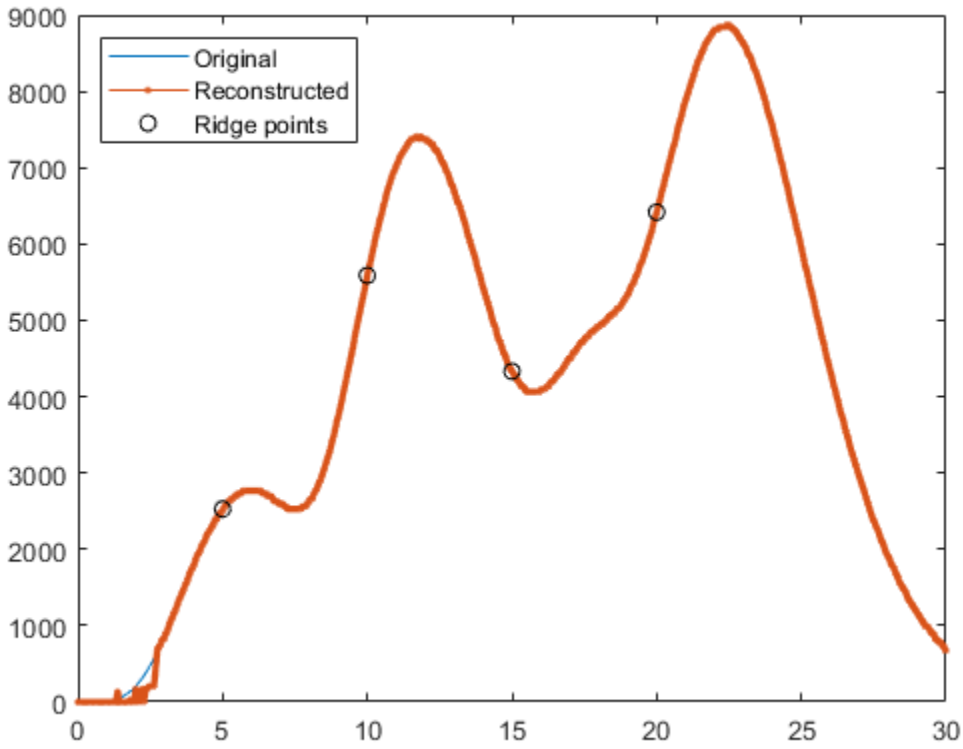
```
ndx = (5:5:20)*fs;
order = ol(2);

p = [t(ndx) order*fis(ndx)];

rpmest = rpmtrack(xt,order,p);
```

```
summary(rpmest)
```

```
RowTimes:

    tout: 30001x1 duration
        Values:
            Min           0 sec
            Median        15 sec
            Max           30 sec
            TimeStep      0.001 sec

Variables:

    rpm: 30001x1 double

        Values:

            Min       2.2204e-16
            Median        4327.2
            Max           8879.8
```

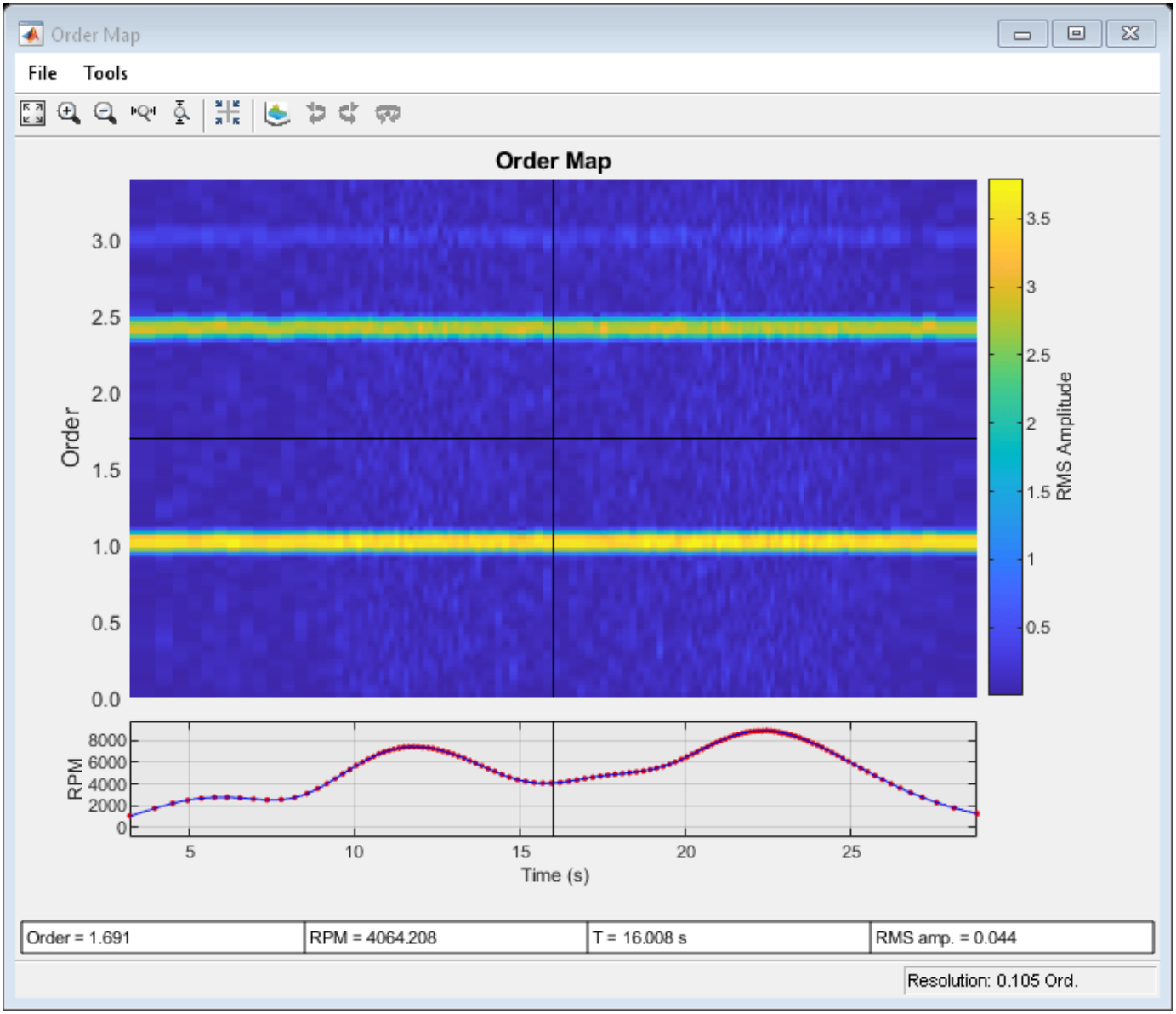
Plot the reconstructed RPM profile and the points used in the reconstruction.

```
hold on
plot(seconds(rpmest.tout),rpmest.rpm,'.-')
plot(t(ndx),fis(ndx)*60,'ok')
hold off
legend('Original','Reconstructed','Ridge points','Location','northwest')
```



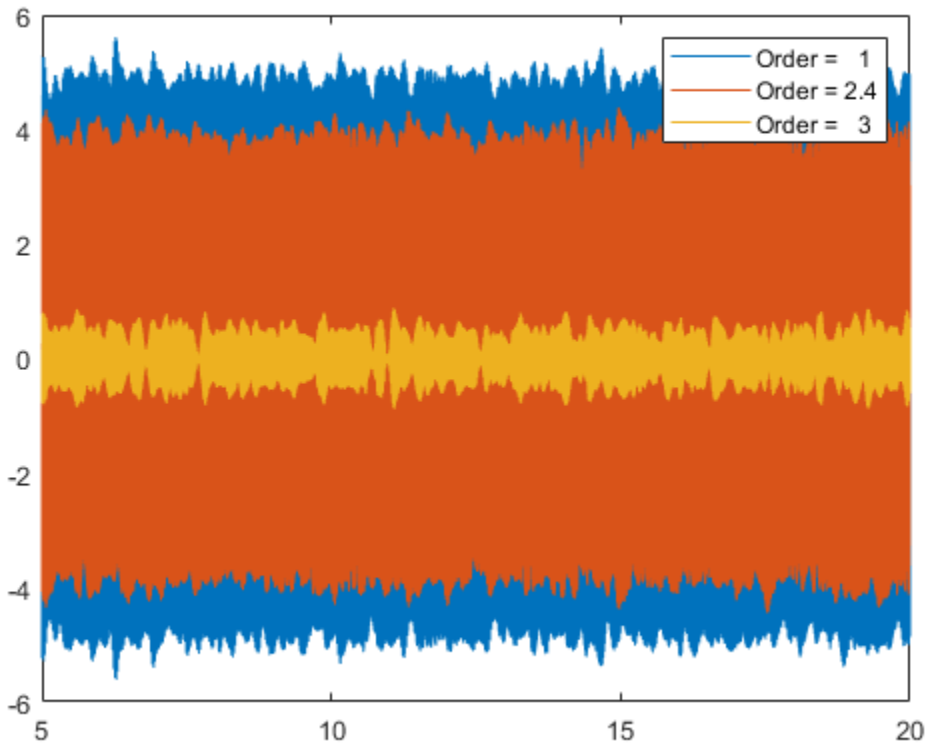
Use the extracted RPM profile to generate the order-RPM map of the signal.

```
rpmordermap(vib,fs,rpmest.rpm)
```



Reconstruct and plot the time-domain waveforms that compose the signal. Zoom in on a time interval occurring after the transients have decayed.

```
xrc = orderwaveform(vib,fs,rpmest.rpm,ol);

figure
plot(t,xrc)
legend([repmat('Order = ',[3 1]) num2str(ol')])
xlim([5 20])
```



Fan Switchoff RPM Profile

Estimate the RPM profile of a fan blade as it slows down after switchoff.

An industrial roof fan spinning at 20,000 rpm is turned off. Air resistance (with a negligible contribution from bearing friction) causes the fan rotor to stop in approximately 6 seconds. A high-speed camera measures the *x*-coordinate of one of the fan blades at a rate of 1 kHz.

```
fs = 1000;
t = 0:1/fs:6-1/fs;

rpm0 = 20000;
```

Idealize the fan blade as a point mass circling the rotor center at a radius of 50 cm. The blade experiences a drag force proportional to speed, resulting in the following expression for the phase angle

$$\phi = 2\pi f_0 T(1 - e^{-t/T}),$$

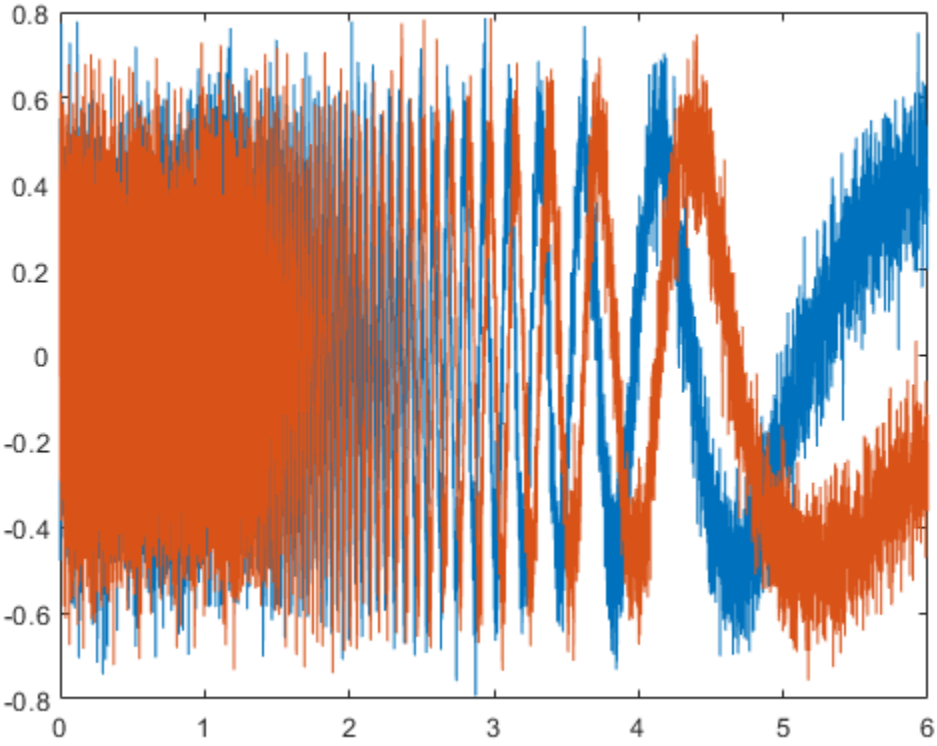
where f_0 is the initial frequency and $T = 0.75$ second is the decay time.

```
a = 0.5;
f0 = rpm0/60;
T = 0.75;
```

```
phi = 2*pi*f0*T*(1-exp(-t/T));
```

Compute and plot the *x*- and *y*-coordinates of the blade. Add white Gaussian noise of variance 0.1^2.

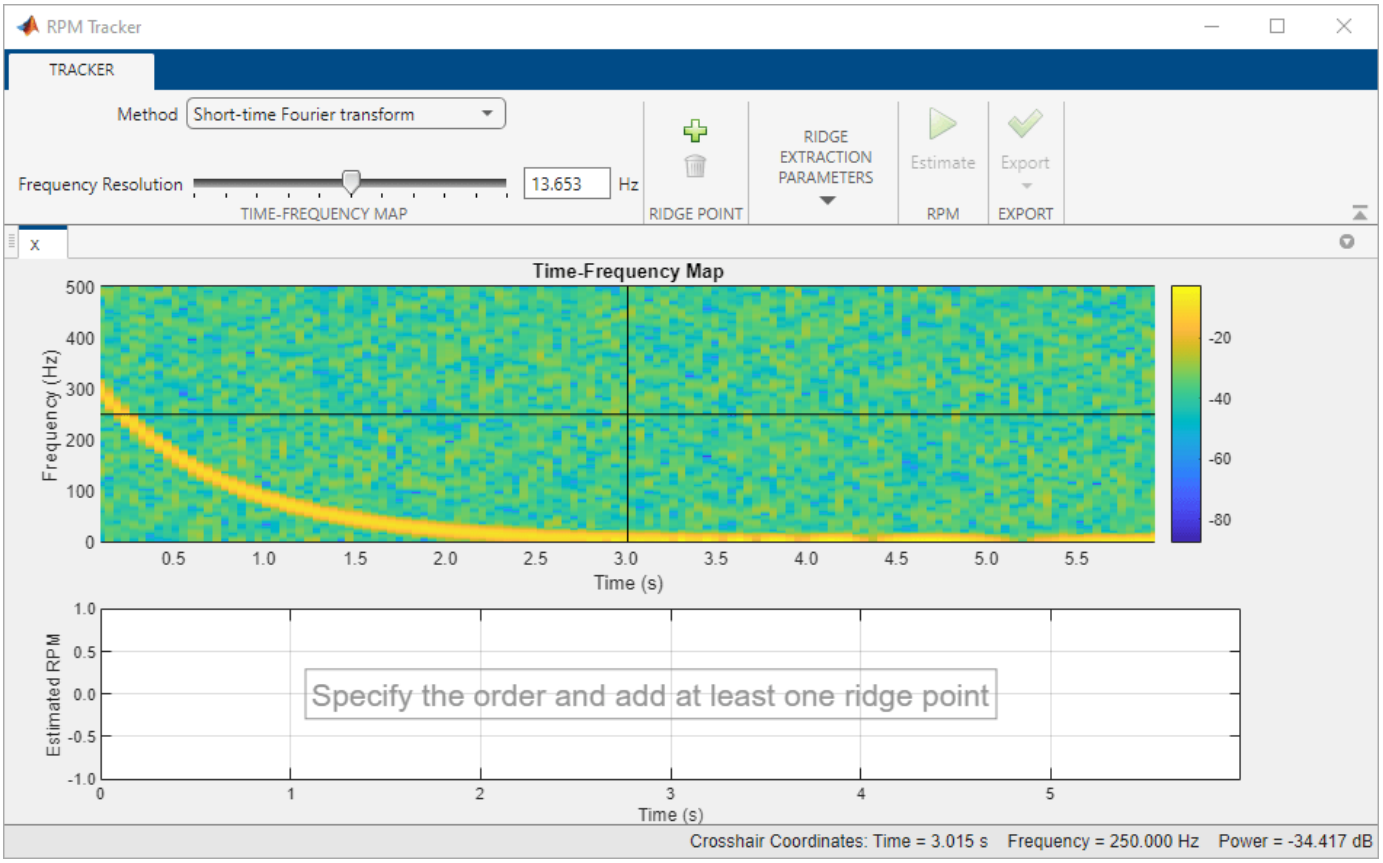
```
x = a*cos(phi) + randn(size(phi))/10;
y = a*sin(phi) + randn(size(phi))/10;

plot(t,x,t,y)
```



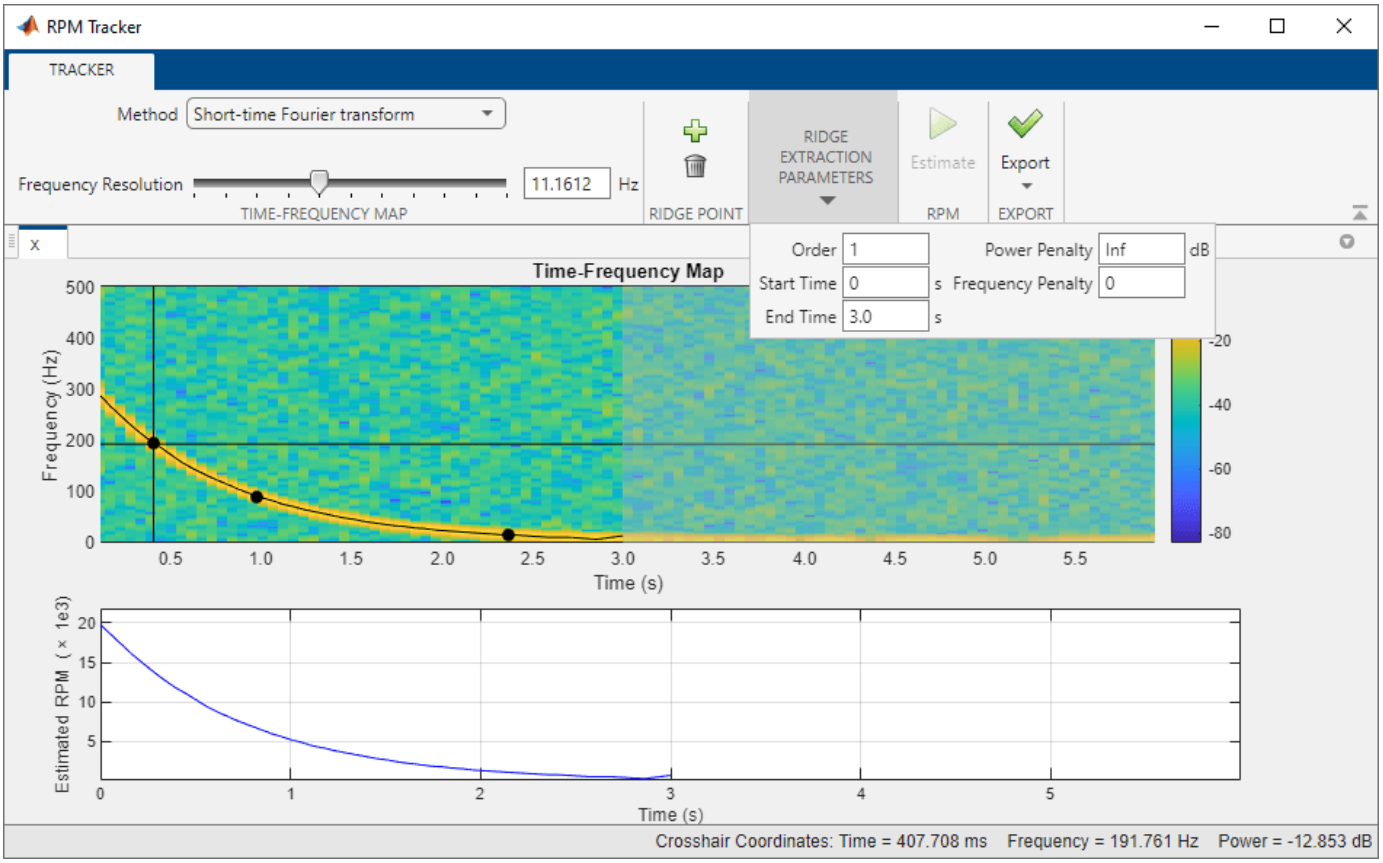
Use the `rpmtrack` function to determine the RPM profile. Type

```
rpmtrack(x,fs)
```

at the command line to open the interactive figure.



Use the slider to adjust the frequency resolution of the time-frequency map to about 11 Hz. Assume that the signal component corresponds to order 1 and set the end time for ridge extraction to 3.0 seconds. Use the crosshair cursor in the time-frequency map and the **Add** button to add three points lying on the ridge. Alternatively, double-click the cursor to add the points at the locations you choose. Click **Estimate** to track and extract the RPM profile.

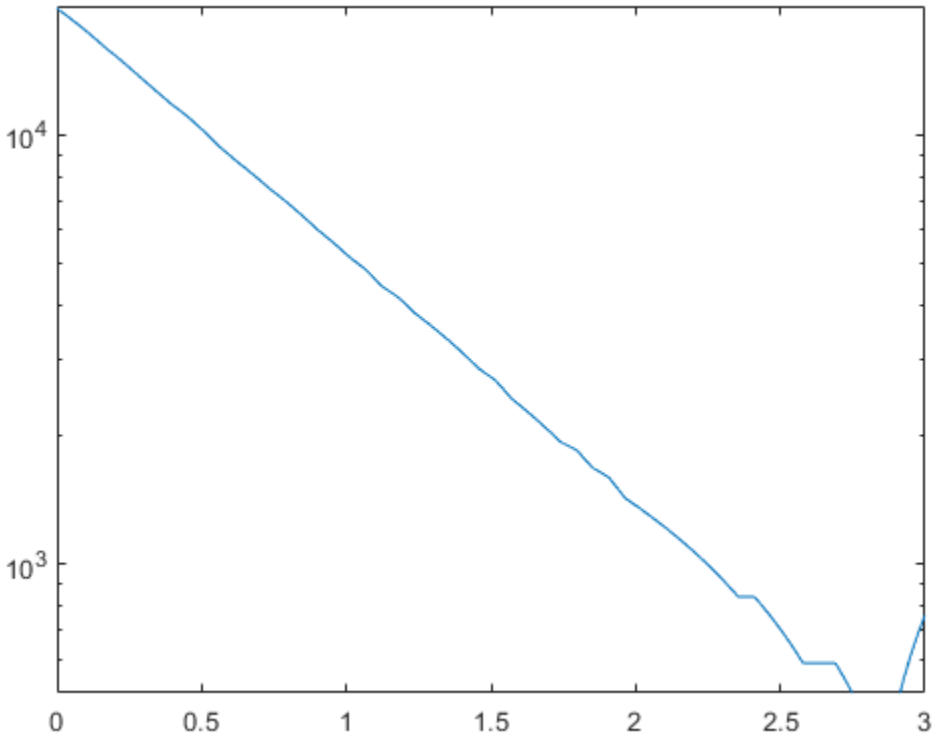


Verify that the RPM profile decays exponentially. On the **Export** tab, click **Export** and select `Generate MATLAB Script`. The script appears in the Editor.

```
% MATLAB Code from rpmtrack GUI

% Generated by MATLAB 9.12 and Signal Processing Toolbox 8.7

% Generated on 12-Oct-2021 09:36:49

% Set sample rate
fs = 1000.0000;
% Set order of ridge of interest
order = 1.0000;
% Set ridge points on ridge of interest
ridgePoints = [...
    0.4077 194.6023;...
    0.9781 89.4886;...
    2.3678 15.6250];
% Estimate RPM
[rpmOut,tOut] = rpmtrack(x,fs,order,ridgePoints,...
    'Method','stft',...
    'FrequencyResolution',11.1612,...
    'PowerPenalty',Inf,...
    'FrequencyPenalty',0.0000,...
    'StartTime',0.0000,...
    'EndTime',3.0000);
```

Run the script. Display the RPM profile in a semilogarithmic plot.

```
semilogy(tOut,rpmOut)
ylim([500 20000])
```



Input Arguments

x — Input signal

vector

Input signal, specified as a vector.

Example: `cos(pi/4*(0:159))+randn(1,160)` specifies a noisy sinusoid sampled at 2π Hz.

Data Types: `single` | `double`

fs — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar.

Data Types: `single` | `double`

order — Ridge order

positive real scalar

Ridge order, specified as a positive real scalar.

Data Types: `single` | `double`

`p` — Ridge points
two-column matrix

Ridge points, specified as a two-column matrix containing one time-frequency coordinate on each row. The coordinates describe points on the time-frequency map belonging to the order ridge of interest.

Data Types: `single` | `double`

`xt` — Input timetable
timetable

Input timetable. `xt` must contain increasing, finite, and equally spaced row times of type `duration`. The timetable must contain only one numeric data vector with signal values.

If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

Example: `timetable(seconds(0:4)',randn(5,1))` specifies a random variable sampled at 1 Hz for 4 seconds.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `"Method","fsst","PowerPenalty",10` specifies that the time-frequency map is estimated using the synchrosqueezed Fourier transform, allowing up to 10 decibels of power difference between adjacent points on a ridge.

`Method` — Type of time-frequency map
`"stft"` (default) | `"fsst"`

Type of time-frequency map used in the estimation process, specified as either `"stft"` or `"fsst"`.

- `"stft"` — Use the short-time Fourier transform to compute a power spectrogram time-frequency map. For more details about the short-time Fourier transform, see `pspectrum`.
- `"fsst"` — Use the synchrosqueezed Fourier transform to compute a time-frequency map. For more details about the synchrosqueezed Fourier transform, see `fsst`.

`FrequencyResolution` — Frequency resolution bandwidth
numeric scalar

Frequency resolution bandwidth used to compute the time-frequency map, specified as a numeric scalar expressed in hertz.

Data Types: `single` | `double`

`PowerPenalty` — Maximum difference in power between adjacent ridge points
`Inf` (default) | numeric scalar in dB

Maximum difference in power between adjacent ridge points, specified as a numeric scalar expressed in decibels.

Use this parameter to ensure that the ridge-extraction algorithm of `rpmtrack` finds the correct ridge for the corresponding order. `PowerPenalty` is useful when the order ridge of interest crosses other ridges or is very close in frequency to other ridges, but has a different power level.

Data Types: `single` | `double`

`FrequencyPenalty` — Penalty in coarse ridge extraction
`0` (default) | nonnegative scalar

Penalty in coarse ridge extraction, specified as a nonnegative scalar.

Use this parameter to ensure that the ridge-extraction algorithm of `rpmtrack` avoids large jumps that could make the ridge estimate move to an incorrect time-frequency location. `FrequencyPenalty` is useful when you want to differentiate order ridges that cross or are closely spaced in frequency.

Data Types: `single` | `double`

`StartTime` — Start time for RPM profile estimation
input signal start time (default) | scalar in seconds | `duration` scalar

Start time for RPM profile estimation, specified as a numeric or `duration` scalar.

Data Types: `single` | `double` | `duration`

`EndTime` — End time for RPM profile estimation
input signal end time (default) | scalar in seconds | `duration` scalar

End time for RPM profile estimation, specified as a numeric or `duration` scalar.

Data Types: `single` | `double` | `duration`

Output Arguments

`rpm` — Rotational speed estimate
vector | timetable

Rotational speed estimate, returned as a vector expressed in revolutions per minute.

If the input to `rpmtrack` is a timetable, then `rpm` is also a timetable with a single variable labeled `rpm`. The row times of the timetable are labeled `tout` and are of type `duration`.

`tout` — Time values
vector

Time values at which the RPM profile is estimated, returned as a vector.

Algorithms

`rpmtrack` uses a two-step (coarse-fine) estimation method:

1. Compute a time-frequency map of `x` and extract a time-frequency ridge based on a specified set of points on the ridge, `p`, the `order` corresponding to that ridge, and the optional penalty

parameters `PowerPenalty` and `FrequencyPenalty`. The extracted ridge provides a coarse estimate of the RPM profile.

2. Compute the order waveform corresponding to the extracted ridge using a Vold-Kalman filter and calculate a new time-frequency map from this waveform. The isolated order ridge from the new time-frequency map provides a fine estimate of the RPM profile.

References

[1] Urbanek, Jacek, Tomasz Barszcz, and Jerome Antoni. "A Two-Step Procedure for Estimation of Instantaneous Rotational Speed with Large Fluctuations." *Mechanical Systems and Signal Processing*. Vol. 38, 2013, pp. 96–102.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Any name-value argument character or string input must be a constant at compile time.

See Also

Functions

`ordertrack` | `rpmfreqmap` | `rpmordermap` | `tachorpm`

Introduced in R2018a

rssq

Root-sum-of-squares level

Syntax

```
y = rssq(x)
y = rssq(x,dim)
```

Description

`y = rssq(x)` returns the root-sum-of-squares (RSS) level, `y`, of the input array `x`. If `x` is a row or column vector, `y` is a real-valued scalar. If `x` has more than one dimension, then `rssq` operates along the first array dimension with size greater than 1.

`y = rssq(x,dim)` computes the RSS level of `x` along dimension `dim`.

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)
```

```
y = 22.3607
```

RSS 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 RSS levels of the columns.

```
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)'*(1:4);

y = rssq(x)
```

```
y = 1×4

   22.3607   44.7214   67.0820   89.4427
```

RSS 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 RSS levels of the rows by specifying the dimension with the `dim` argument.

```
t = 0:0.001:1-0.001;
x = (1:4)'*cos(2*pi*100*t);

y = rssq(x,2)
```

```
y = 4×1

   22.3607
   44.7214
   67.0820
   89.4427
```

Input Arguments

x — Input array

vector | matrix | *N*-D array

Input array, specified as a vector, matrix, or *N*-D array.

Example: `cos(2*pi*100*(0:0.001:1-0.001))` specifies a sinusoid sampled at 1 kHz for 1 second.

Data Types: `single` | `double`
Complex Number Support: Yes

dim — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar.

Data Types: `single` | `double`

Output Arguments

y — Root-sum-of-squares level

scalar | vector | matrix | *N*-D array

Root-sum-of-squares level, returned as a scalar, vector, matrix, or *N*-D array.

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 level is also referred to as the 2-norm.

References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you supply `dim`, then it must be constant.
- For limitations related to variable-size inputs, see “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).
- Code generation does not support sparse matrix inputs for this function.

Thread-Based Environment

Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

This function fully supports thread-based environments. For more information, see “Run MATLAB Functions in Thread-Based Environment”.

See Also

`mean` | `peak2peak` | `peak2rms` | `rms` | `std`

Introduced in R2012a

sawtooth

Sawtooth or triangle wave

Syntax

```
x = sawtooth(t)
x = sawtooth(t,xmax)
```

Description

`x = sawtooth(t)` generates a sawtooth wave with period 2π for the elements of the time array `t`. `sawtooth` is similar to the sine function but creates a sawtooth wave with peaks of –1 and 1. The sawtooth wave is defined to be –1 at multiples of 2π and to increase linearly with time with a slope of $1/\pi$ at all other times.

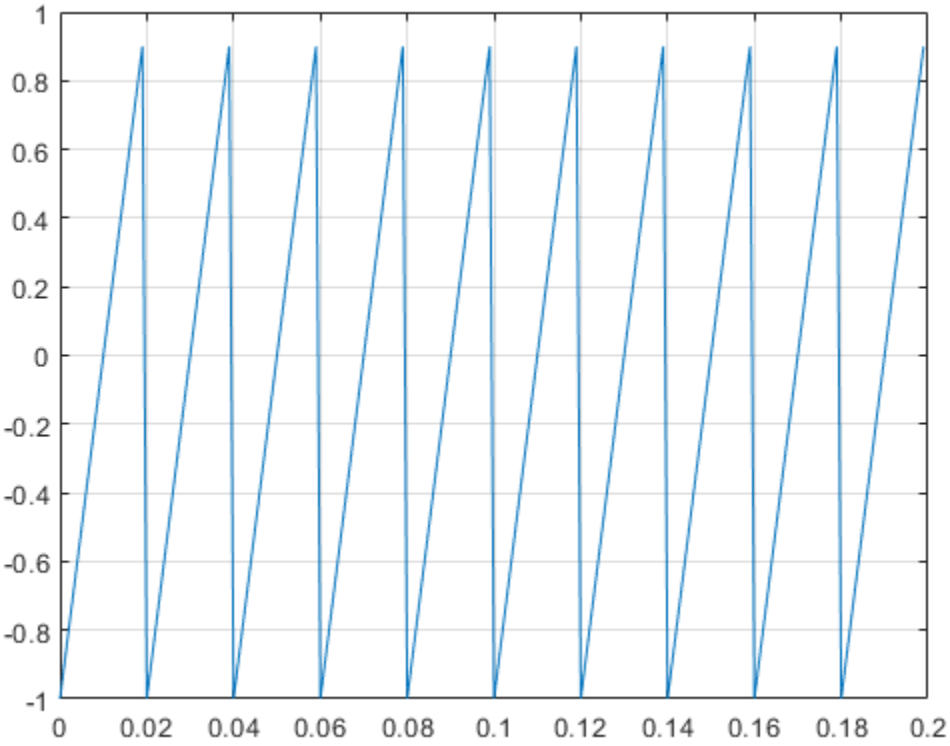
`x = sawtooth(t,xmax)` generates a modified triangle wave with the maximum location at each period controlled by `xmax`. Set `xmax` to `0.5` to generate a standard triangle wave.

Examples

50 Hz Sawtooth Wave

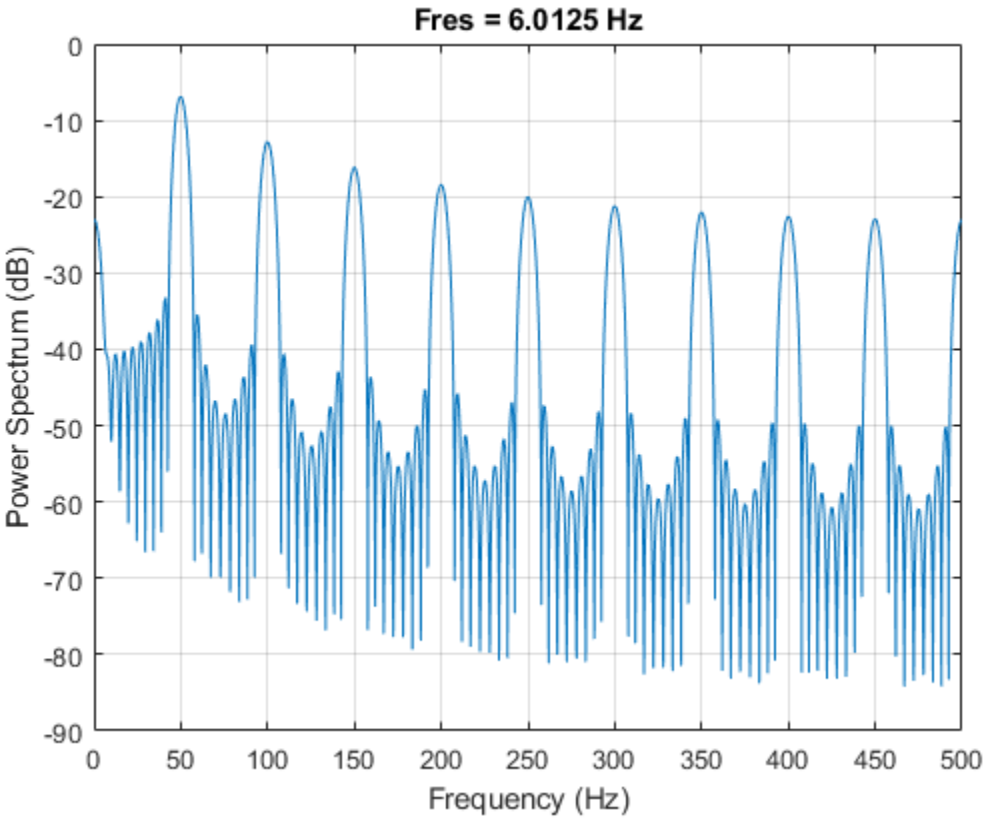
Generate 10 periods of a sawtooth wave with a fundamental frequency of 50 Hz. The sample rate is 1 kHz.

```
T = 10*(1/50);

fs = 1000;
t = 0:1/fs:T-1/fs;

x = sawtooth(2*pi*50*t);

plot(t,x)
grid on
```



Plot the power spectrum of the wave.

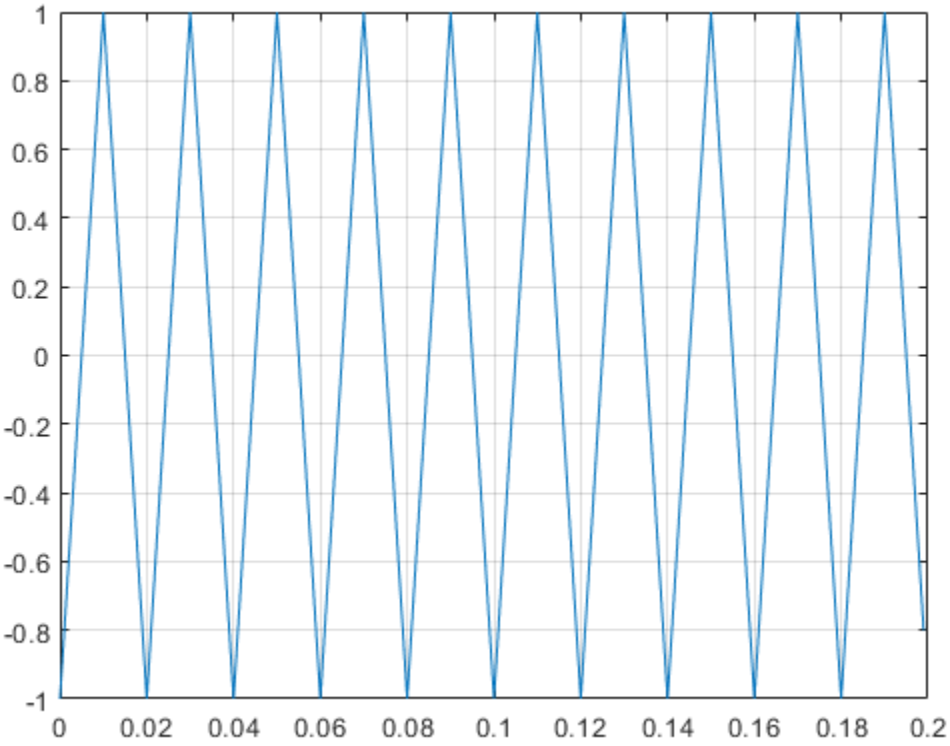
```
pspectrum(x,fs,'Leakage',0.91)
```



50 Hz Triangle Wave

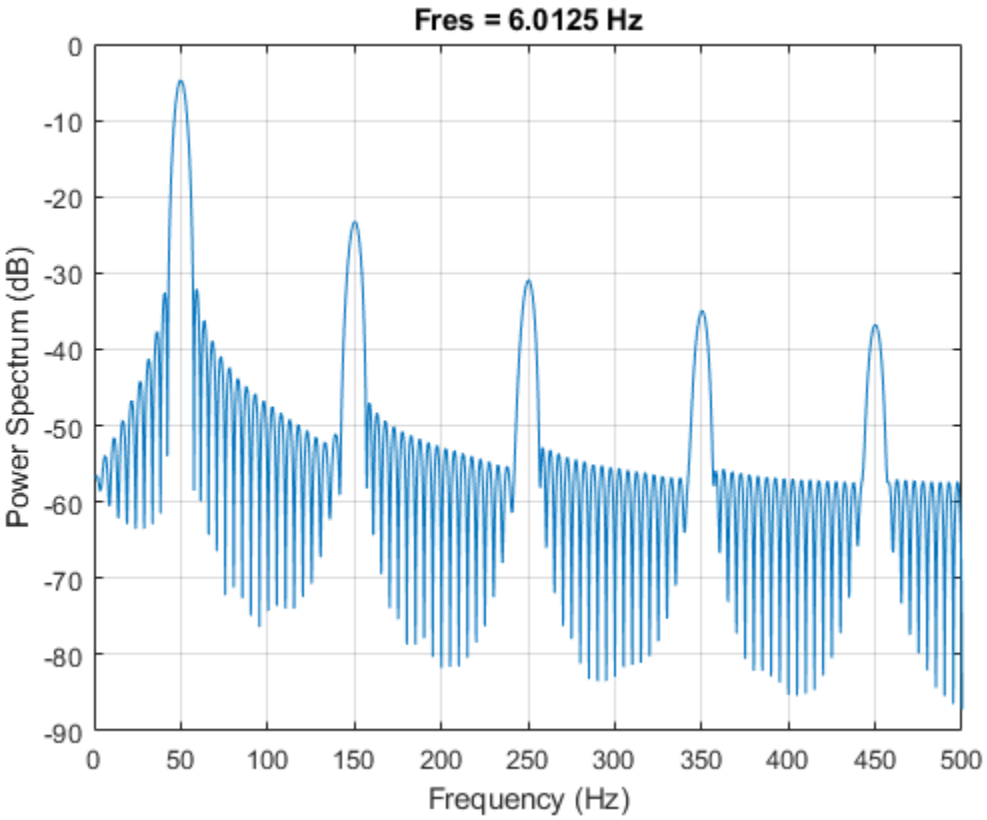
Generate 10 periods of a triangle wave with a fundamental frequency of 50 Hz. The sample rate is 1 kHz.

```
T = 10*(1/50);

fs = 1000;
t = 0:1/fs:T-1/fs;

x = sawtooth(2*pi*50*t,1/2);

plot(t,x)
grid on
```



Plot the power spectrum of the wave.

```
pspectrum(x,fs,'Leakage',0.91)
```



Input Arguments

`t` — Time array

vector | matrix | *N*-D array

Time array, specified as a vector, matrix, or *N*-D array. `sawtooth` operates along the first array dimension of `t` with size greater than 1.

Data Types: `double`

`xmax` — Wave maximum location

1 (default) | scalar between 0 and 1

Wave maximum location, specified as a scalar between 0 and 1. `xmax` determines the point between 0 and 2π at which the wave reaches its maximum. The function increases from –1 to 1 on the interval 0 to $2\pi \times$ `xmax`, then decreases linearly from 1 to –1 on the interval $2\pi \times$ `xmax` to 2π. The shape then repeats with a period of 2π.

Example: `xmax = 0.5` specifies a standard triangle wave, symmetric about time π with a peak-to-peak amplitude of 1.

Data Types: `double`

Output Arguments

x — Sawtooth wave
vector | matrix | *N*-D array

Sawtooth wave, returned as a vector, matrix, or *N*-D array.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`chirp` | `cos` | `diric` | `gauspuls` | `pulstran` | `rectpuls` | `sin` | `square` | `tripuls`

Introduced before R2006a

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 = 4×1

   -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`

Introduced before R2006a

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 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`. To determine the transitions, the `settlingtime` function estimates the state levels of the input waveform by a histogram method and identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state.

Note 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`. For cases in which `settlingtime` returns a `NaN`, see “Settle Seek Duration” on page 1-1987.

`s = settlingtime(x,Fs,d)` specifies the sample rate `Fs`.

`s = settlingtime(x,t,d)` specifies the sample instants `t`.

`[s,slev,sinst] = settlingtime( ___ )` returns the levels `slev` and sample instants `sinst` 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 arguments. You can specify an input combination from any of the previous syntaxes.

`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.

Examples

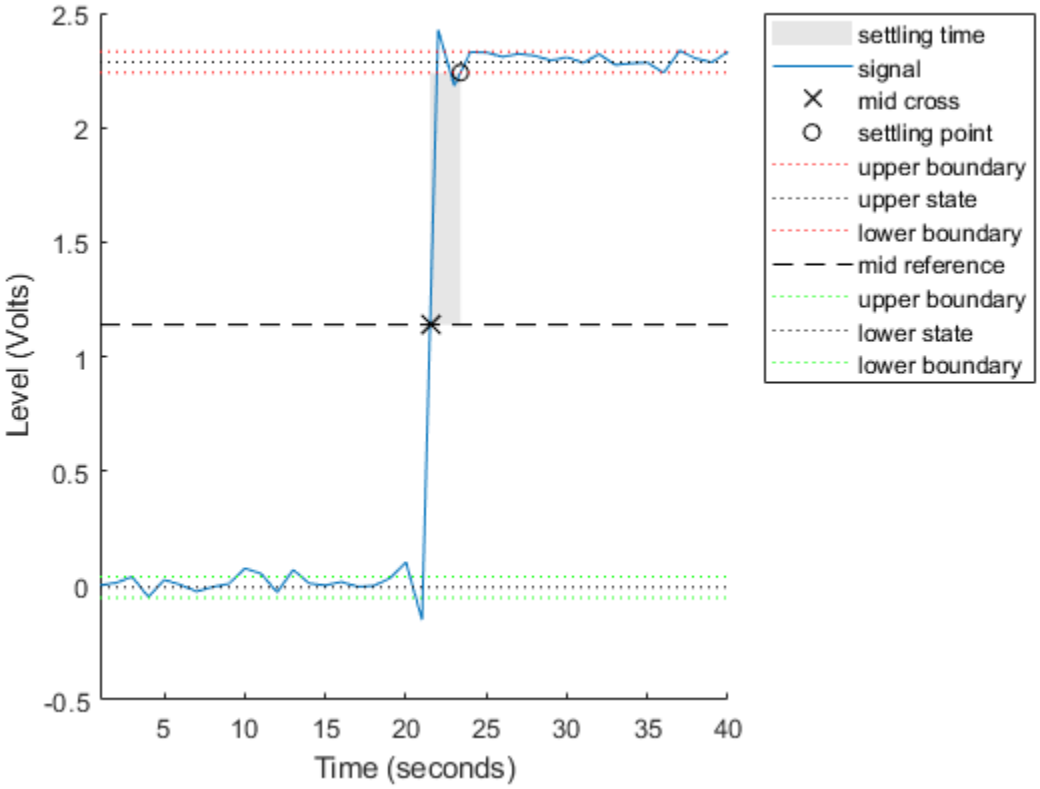
Determine Settling Point and Settling Level

Determine the settling point and corresponding waveform value for a bilevel waveform. Specify a settle-seek duration of 10 seconds.

```
load('transitionex.mat', 'x')
[s,slev,sinst] = settlingtime(x,10);
```

Plot the waveform and annotate the settling point.

```
settlingtime(x,10)
```




```
ans = 1.8901
```

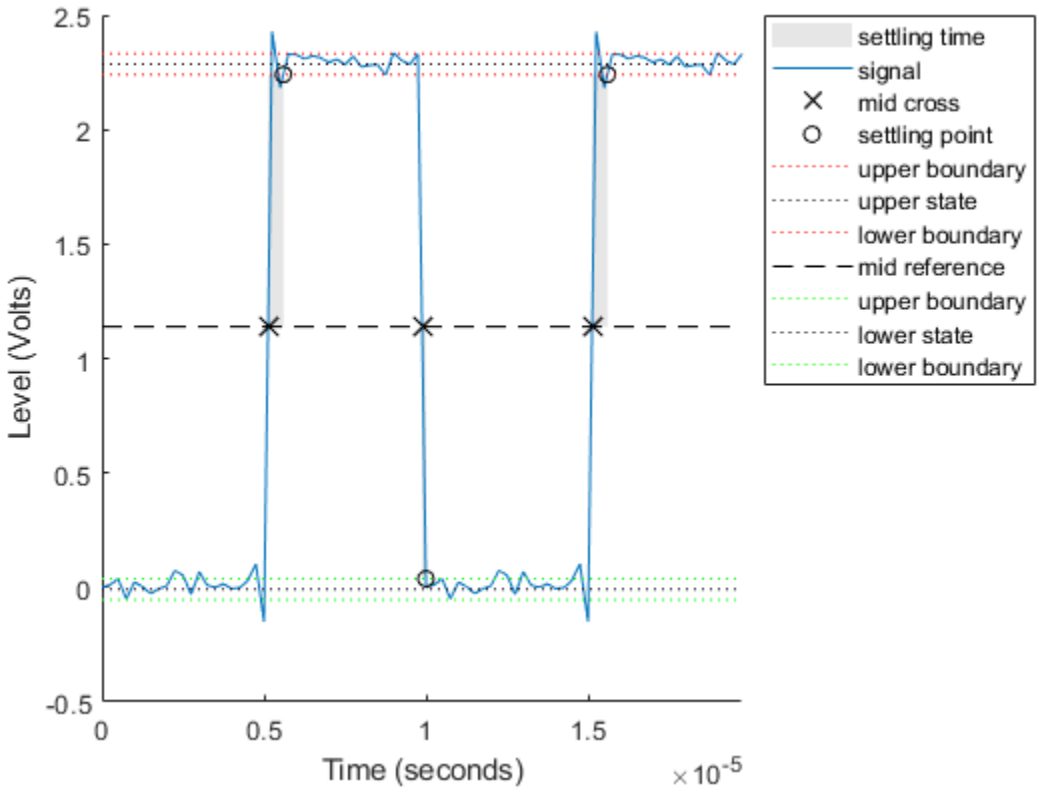
Determine Settling Points of Three-Transition Bilevel Waveform

Determine the settling points for a three-transition bilevel waveform. The data are sampled at 4 MHz. Specify a settle-seek duration of one microsecond.

```
load('transitionex.mat','x')
y = [x; fliplr(x)];
fs = 4e6;
t = 0:1/fs:(length(y)*1/fs)-1/fs;

[s,slev,sinst] = settlingtime(y,fs,1e-6);
```

Plot the waveform and annotate the settling points.

```
settlingtime(y,fs,1e-6)
```




```
ans = 3×1
10^-6 ×

    0.4725
    0.1181
    0.4725
```

Input Arguments

x — Bilevel waveform

real-valued vector

Bilevel waveform, specified as a real-valued vector. The first sample instant in `x` is equal to t=0.

d — Settle-seek duration

positive scalar

Settle-seek duration in seconds, specified as a positive scalar. This argument 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, the function returns a `NaN`. For more information, see “Settling Time” on page 1-1986 and “Settle Seek Duration” on page 1-1987.

Fs — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar in hertz.

`t` — Sample instants
vector

Sample instants, specified as a vector. The length of `t` must equal the length of the input bilevel waveform `x`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `s = settlingtime(x,Fs,Tolerance=5)`

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `s = settlingtime(x,Fs,'Tolerance',5)`

`MidPercentReferenceLevel` — Mid-reference level
`50` (default) | scalar

Mid-reference level as a percentage of the waveform amplitude, specified as a scalar. For more information, see “Mid-Reference Level” on page 1-1986.

`StateLevels` — Low- and high-state levels
1-by-2 real-valued vector

Low- and high-state levels, specified as a 1-by-2 real-valued vector. The first element is the low-state level and the second element is the high-state level. If you do not specify `'StateLevels'`, the function estimates the state levels from the input waveform using the histogram method.

`Tolerance` — Tolerance levels
2 (default) | scalar

Tolerance levels (lower and upper state boundaries), specified as a scalar and expressed as a percentage. 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. For more information, see “State-Level Tolerances” on page 1-1987.

Output Arguments

`s` — Settling time
vector

Settling time, returned as a vector. The settling time corresponds to 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`. The length of `s` is equal to the number of detected transitions in the input bilevel waveform `x`.

Note Because `settlingtime` uses interpolation to determine the mid-reference level instant, `s` may contain values that do not correspond to sampling instants.

`slev` — Waveform values at settling points

vector

Waveform values at the settling points, returned as a vector.

`sinst` — Time instants of settling points

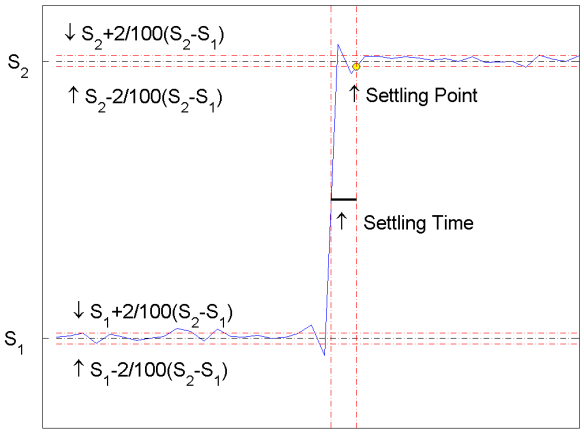
vector

Time instants of the settling points, returned as a vector.

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 this figure, where 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

$$t_{50\%} = t_{50\%} + (\frac{t_{50\%_+} - t_{50\%_-}}{y_{50\%_+} - y_{50\%_-}})(y_{50\%_+} - y_{50\%_-})$$

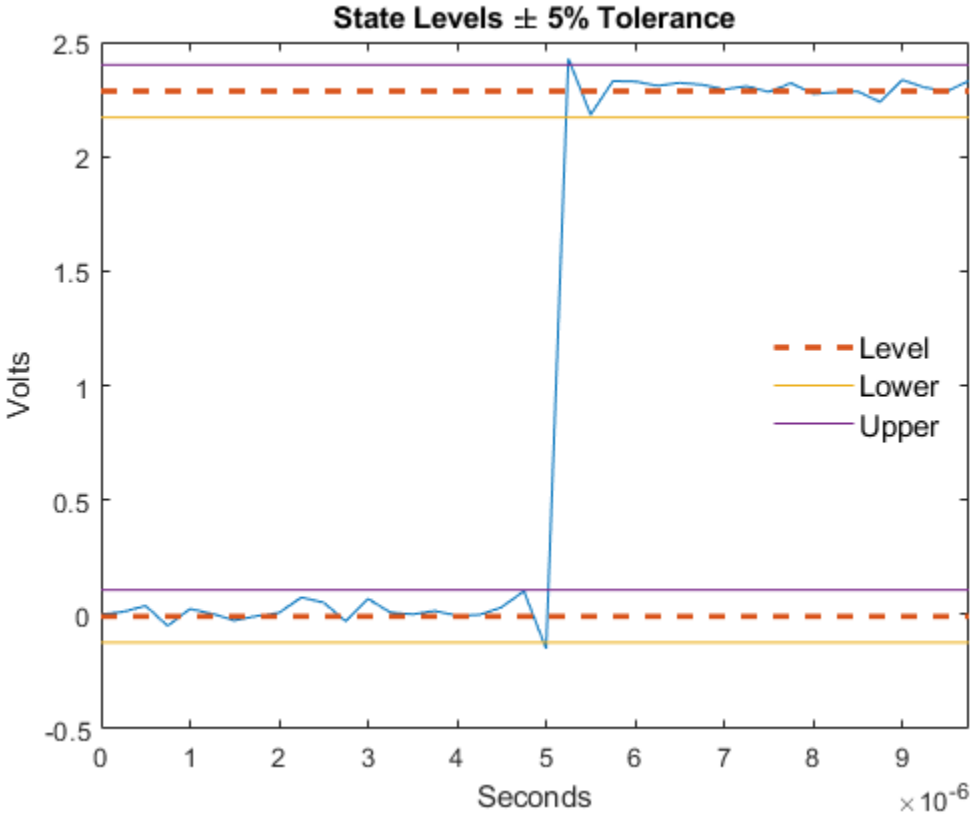
State-Level Tolerances

You can specify lower- and upper-state boundaries for each state level. Define the boundaries as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, specify the scalar as a small number such as 2/100 or 3/100. In general, the $\alpha\%$ region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1),$$

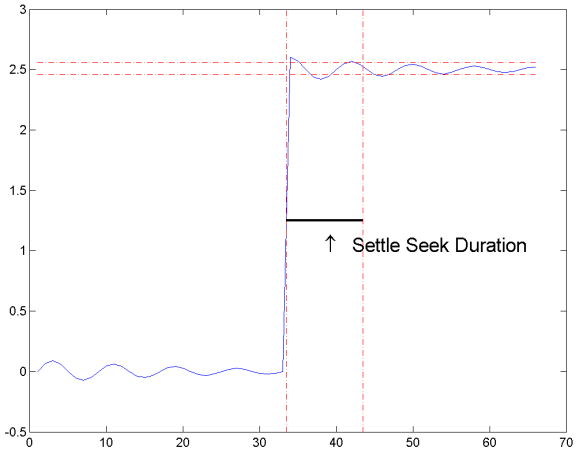
where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the $\alpha\%$ tolerance region for the high state.

This figure shows lower and upper 5% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The thick dashed lines indicate the estimated state levels.



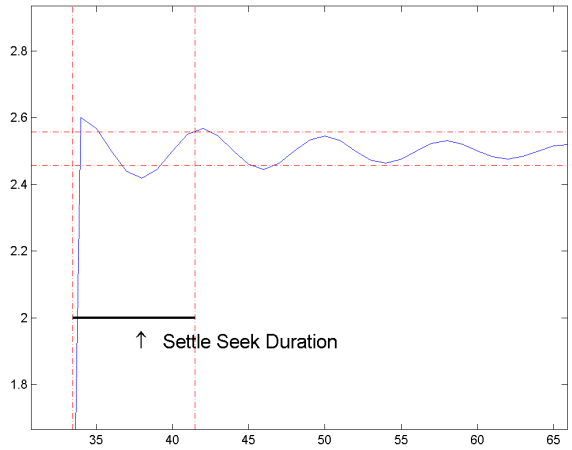
Settle Seek Duration

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. This figure illustrates a settle seek duration of 10 samples.

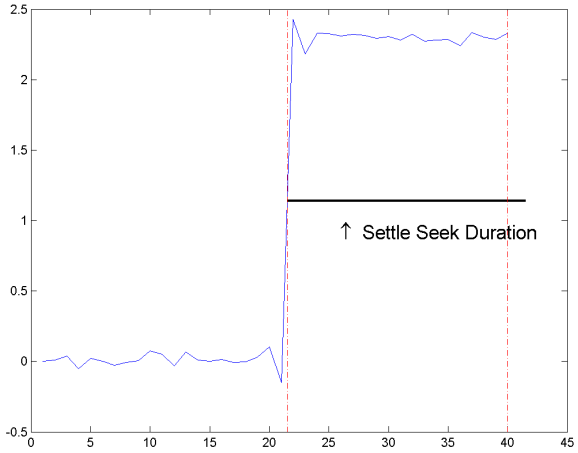


The `settlingtime` function may fail to find a settling point in the specified settle seek duration if any one of these conditions occurs:

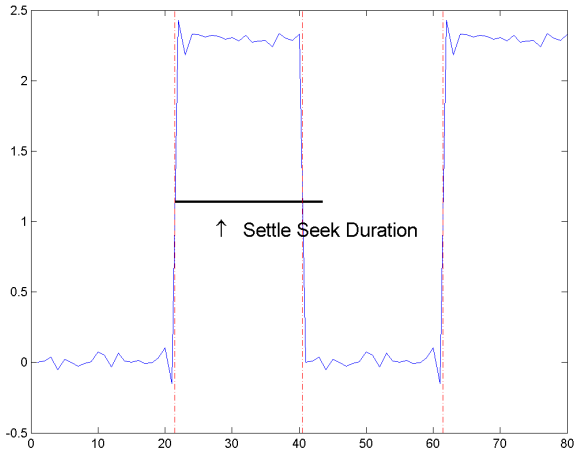
- The last waveform value in the settle seek interval is not within the upper- and lower-state boundaries determined by the specified tolerance. This figure illustrates this condition for a settle seek duration of 8 samples and a 2% tolerance region. The last sample in the settle seek interval exceeds 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. This 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. This 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`falltime` | `midcross` | `pulsewidth` | `risetime` | `statelevels`

Introduced in R2012a

seqperiod

Compute period of sequence

Syntax

```
p = seqperiod(x)
p = seqperiod(x,tol)
[p,nr] = seqperiod(x)
```

Description

`p = seqperiod(x)` returns the integers that correspond to the periods of the sequences in `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.

`p = seqperiod(x,tol)` specifies `tol` as the absolute tolerance to determine when two numbers are close enough to be treated as equal.

`[p,nr] = seqperiod(x)` also returns the number of repetitions of `x(1:p)` in `x`.

Examples

Multichannel Signal Periods

Generate a multichannel signal and determine the period of each column.

```
x = [4 0 1 6;
     2 0 2 7;
     4 0 1 5;
     2 0 5 6];

p = seqperiod(x)
```

```
p = 1×4

     2     1     4     3
```

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 second repetition of the periodic sequence is incomplete.

Compute the number of times that each periodic sequence is repeated.

```
[~,nr] = seqperiod(x)
```

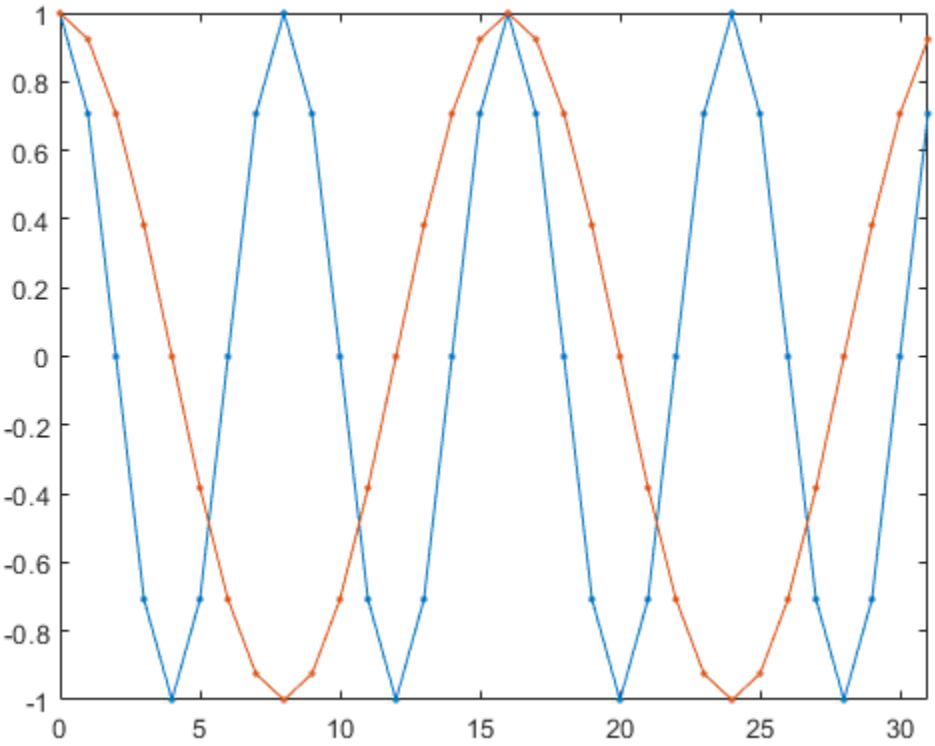
```
nr = 1×4

    2.0000    4.0000    1.0000    1.3333
```

In the first column of `x`, the periodic sequence appears twice. In the second column, the one-sample sequence is repeated as many times as there are samples. In the third column, there is no repetition. The number of repetitions in the fourth column is one plus the fraction of the sequence length represented by the remaining sample.

Periods of Two-Channel Sinusoid

Generate a two-channel sinusoid such that one channel has four periods in the sampling interval and the other channel has two periods. Plot the sinusoid.

```
n = 0:31;
x = cos(2*pi./[8;16].*n)';

plot(n,x,'.-')
axis tight
```



Compute the lengths of the repeated subsequences and the number of repetitions. Specify an absolute tolerance of `1e-5`.

```
[p,nr] = seqperiod(x,1e-5)
```

```
p = 1×2

     8    16
```

```
nr = 1×2

     4     2
```

Sequence Period Along Higher Dimension

Create an array whose first two dimensions have size 1. Along the third dimension, the array has a repeating sequence.

```
a = permute([5 4 3 5 4 3 5 4],[3 1 2])
```

```
a =
a(:,:,1) =

     5


a(:,:,2) =

     4


a(:,:,3) =

     3


a(:,:,4) =

     5


a(:,:,5) =

     4


a(:,:,6) =

     3


a(:,:,7) =

     5


a(:,:,8) =

     4
```

Compute the period of the repeating sequence and the number of repetitions contained in the array. The function works along the third dimension, as expected.

```
[p,nr] = seqperiod(a)

p = 3

nr = 2.6667
```

Input Arguments

x — Input array

vector | matrix | *N*-D array

Input array, specified as a vector, matrix, or *N*-D array.

- If x is a matrix, then `seqperiod` checks for periodicity along each column of x.
- If x is a multidimensional array, then `seqperiod` checks for periodicity along the first array dimension of x with size greater than 1.

The length of x does not have to be a multiple of p, so that incomplete repetitions are permitted at the end of x.

Example: `sin(pi./[4;2]*(0:159))'` specifies a two-channel sinusoid. The second channel has twice the frequency of the first channel.

Data Types: `double`

tol — Absolute tolerance

`1e-10` (default) | positive real scalar

Absolute tolerance to determine when two numbers are close enough to be treated as equal, specified as a positive real scalar.

Data Types: `double`

Output Arguments

p — Sequence period

scalar | vector | matrix | *N*-D array

Sequence period, returned as a scalar, vector, matrix, or *N*-D array. If a sequence is not periodic, then p equals the length of x along the chosen dimension.

- If x is a matrix, then p is a row vector with the same number of columns as x.
- If x is a multidimensional array, then p is a multidimensional array of integers whose first dimension is of size 1. The remaining dimensions of p correspond to the remaining dimensions of x with sizes larger than 1.

nr — Number of sequence repetitions

vector | matrix | *N*-D array

Number of sequence repetitions, returned as a scalar, vector, matrix, or *N*-D array. `nr` has the same dimensions as p. The elements of `nr` are not necessarily integers.

See Also

`findsignal` | `pulseperiod`

Topics
“Measuring Signal Similarities”

Introduced before R2006a

sfdr

Spurious free dynamic range

Syntax

```
r = sfdr(x)
r = sfdr(x,fs)
r = sfdr(x,fs,msd)

r = sfdr(sxx,f,'power')
r = sfdr(sxx,f,msd,'power')

[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 β = 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 sample 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 sample rate is `fs`. If the carrier frequency is `Fc`, then all spurs in the interval (`Fc-msd`, `Fc+msd`) are ignored.

`r = sfdr(sxx,f,'power')` 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,'power')` 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 (`Fc-msd`, `Fc+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

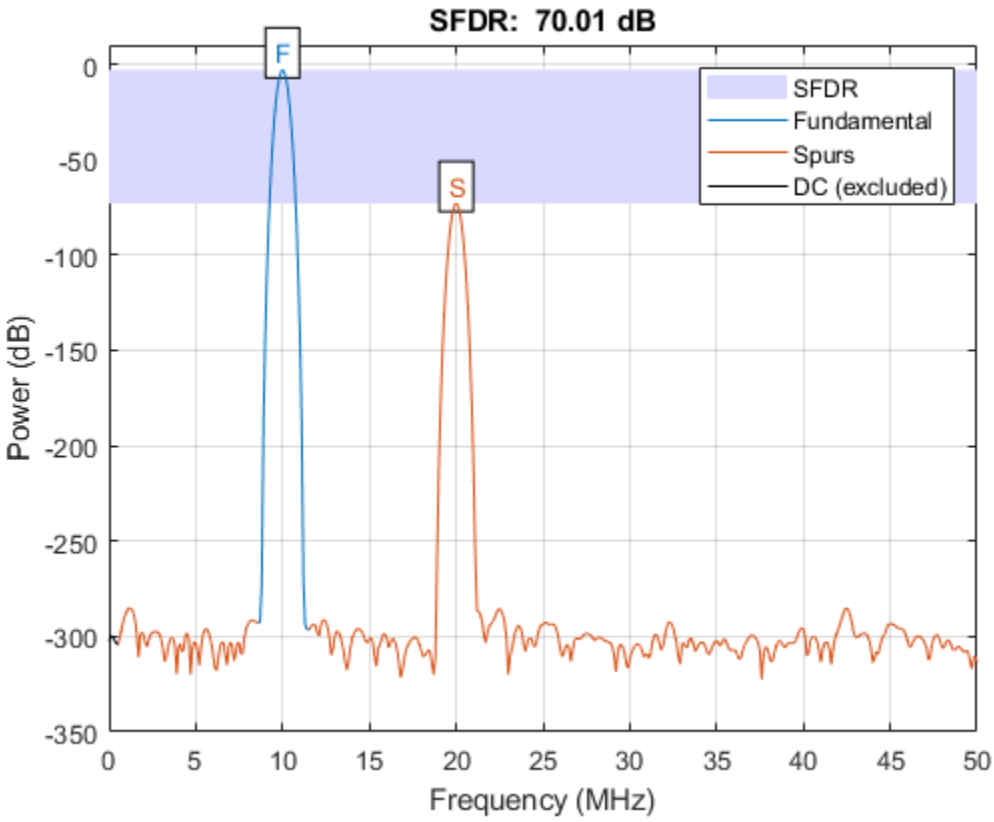
Obtain the SFDR for a 10 MHz tone with amplitude 1 sampled at 100 MHz. There is a spur at the 1st harmonic (20 MHz) with an amplitude of 3.16×10^{-4}.

```
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

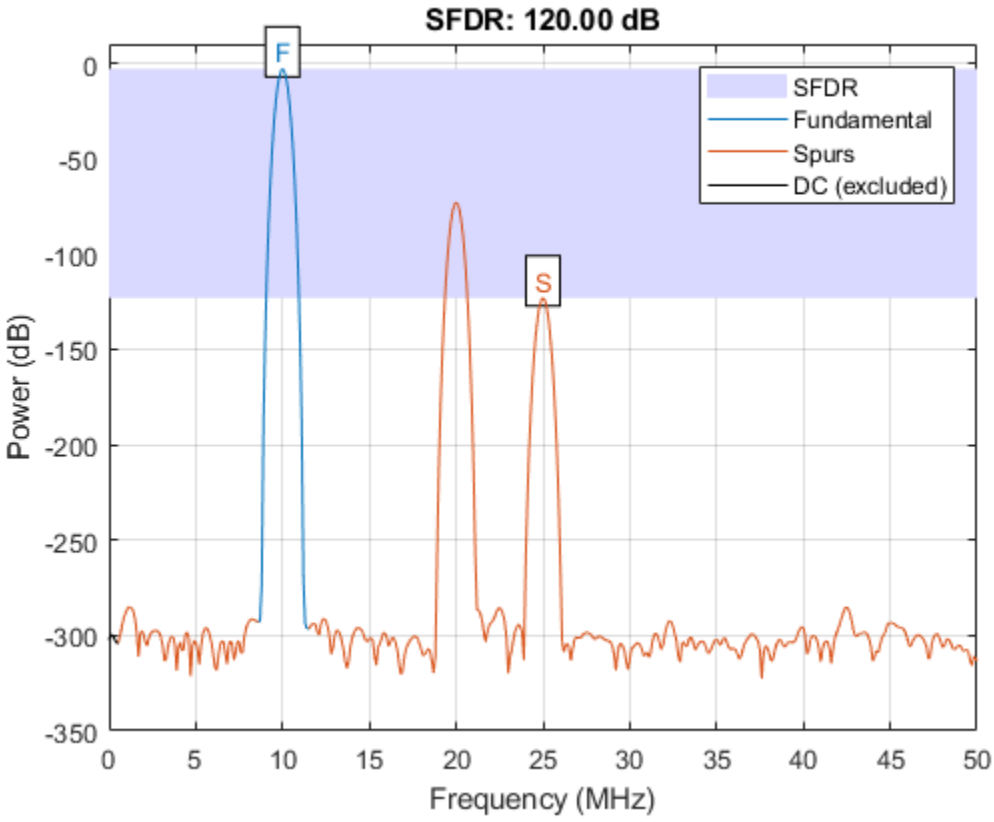
Obtain the SFDR for a 10 MHz tone with amplitude 1 sampled at 100 MHz. There is a spur at the 1st harmonic (20 MHz) with an amplitude of 3.16×10^{-4} and another spur at 25 MHz with an amplitude of 10^{-5}. Skip the first harmonic by using a minimum spur distance of 11 MHz.

```
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);
```



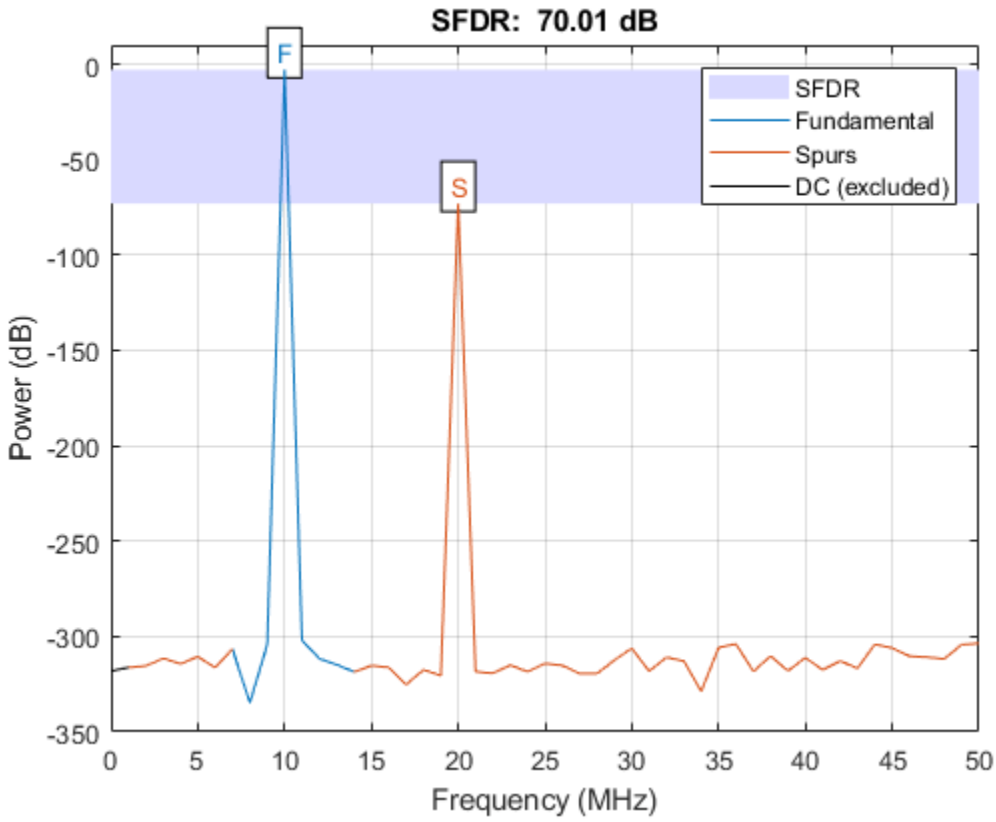
SFDR from Periodogram

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 3.16×10^{-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 3.16×10^{-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

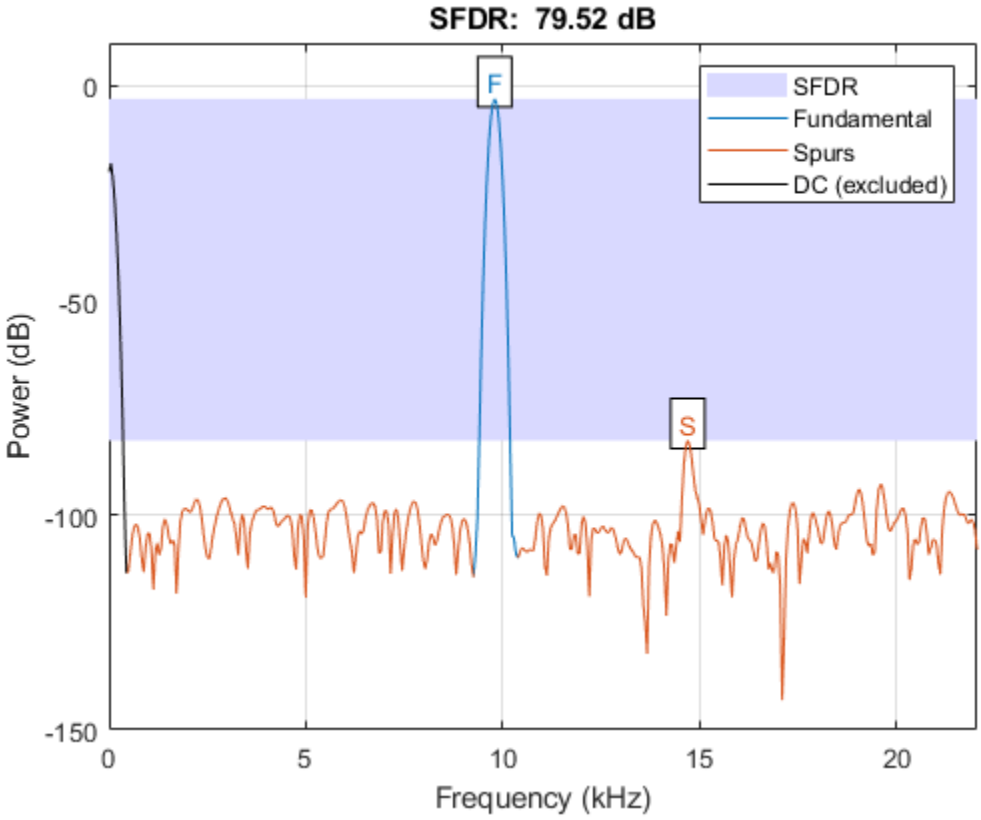
Create a superposition of three sinusoids, with frequencies of 9.8, 14.7, and 19.6 kHz, in white Gaussian additive noise. The signal is sampled at 44.1 kHz. The 9.8 kHz sine wave has an amplitude

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);
```



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` — Sample rate

1 (default) | positive scalar

Sample 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 `Fc`, then all spurs in the range `(Fc-msd, Fc+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.

The power spectrum must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[sxx,f] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2),'power')` specifies the periodogram power spectrum estimate of a noisy two-channel sinusoid sampled at 2π Hz and the frequencies at which it is computed.

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.

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 sample rate as an input argument, `sfdr` assumes a sample rate 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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If supplied, input argument `'power'` must be a compile-time constant.

See Also
`bandpower` | `enbw` | `periodogram`

Topics
“Spurious-Free Dynamic Range (SFDR) Measurement”

Introduced in R2013a

sgolay

Savitzky-Golay filter design

Syntax

```
b = sgolay(order,framelen)
b = sgolay(order,framelen,weights)
[b,g] = sgolay(___)
```

Description

`b = sgolay(order,framelen)` designs a Savitzky-Golay FIR smoothing filter with polynomial order `order` and frame length `framelen`.

`b = sgolay(order,framelen,weights)` specifies a weighting vector, `weights`, 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. You can use these output arguments with any of the previous input syntaxes.

Examples

Savitzky-Golay Smoothing of Noisy Sinusoid

Generate a signal that consists of a 0.2 Hz sinusoid embedded in white Gaussian noise and sampled five times a second for 200 seconds.

```
dt = 1/5;
t = (0:dt:200-dt)';

x = 5*sin(2*pi*0.2*t) + randn(size(t));
```

Use `sgolay` to smooth the signal. Use 21-sample frames and fourth order polynomials.

```
order = 4;
framelen = 21;

b = sgolay(order,framelen);
```

Compute the steady-state portion of the signal by convolving it with the center row of `b`.

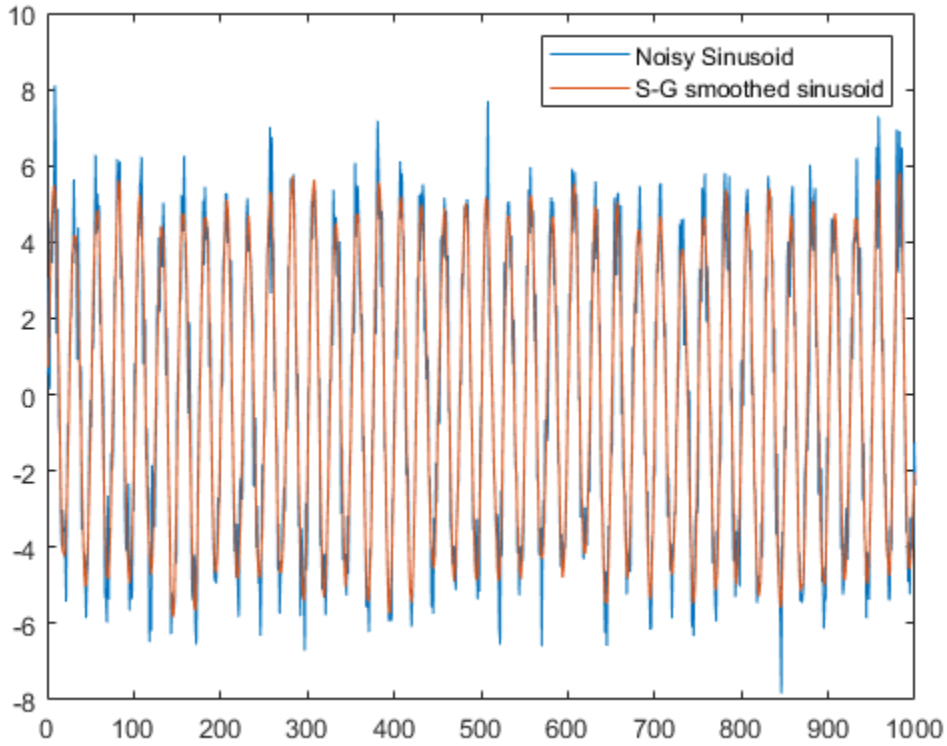
```
ycenter = conv(x,b((framelen+1)/2,:),'valid');
```

Compute the transients. Use the last rows of `b` for the startup and the first rows of `b` for the terminal.

```
ybegin = b(end:-1:(framelen+3)/2,:) * x(framelen:-1:1);
yend = b((framelen-1)/2:-1:1,:) * x(end:-1:end-(framelen-1));
```

Concatenate the transients and the steady-state portion to generate the complete smoothed signal. Plot the original signal and the Savitzky-Golay estimate.

```
y = [ybegin; ycenter; yend];
plot([x y])
legend('Noisy Sinusoid','S-G smoothed sinusoid')
```



Savitzky-Golay Differentiation

Generate a signal that consists of a 0.2 Hz sinusoid embedded in white Gaussian noise and sampled four times a second for 20 seconds.

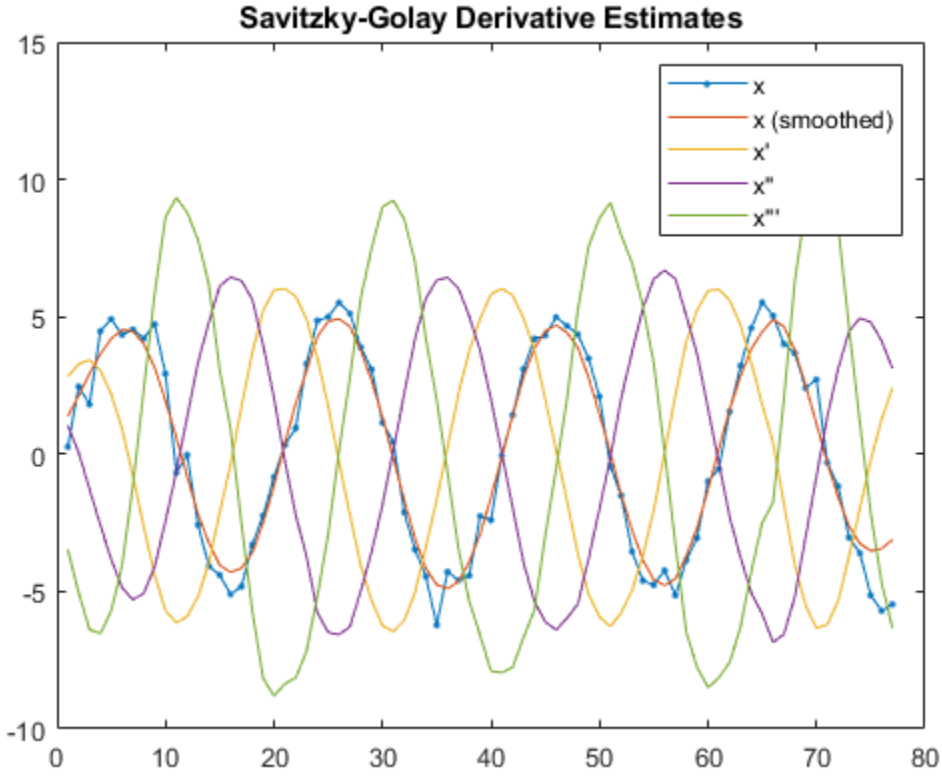
```
dt = 0.25;
t = (0:dt:20-1)';

x = 5*sin(2*pi*0.2*t)+0.5*randn(size(t));
```

Estimate the first three derivatives of the sinusoid using the Savitzky-Golay method. Use 25-sample frames and fifth order polynomials. Divide the columns by powers of `dt` to scale the derivatives correctly.

```
[b,g] = sgolay(5,25);

dx = zeros(length(x),4);
for p = 0:3
  dx(:,p+1) = conv(x, factorial(p)/(-dt)^p * g(:,p+1), 'same');
end
```

Plot the original signal, the smoothed sequence, and the derivative estimates.

```
plot(x,'.-')
hold on
plot(dx)
hold off

legend('x','x (smoothed)','x''','x''''', 'x''''''')
title('Savitzky-Golay Derivative Estimates')
```



Input Arguments

`order` — Polynomial order
positive integer

Polynomial order, specified as a positive integer. The value of `order` must be smaller than `framelen`. If `order` = `framelen` – 1, then the designed filter produces no smoothing.

`framelen` — Frame length
positive odd integer

Frame length, specified as a positive odd integer. The value of `framelen` must be greater than `order`.

`weights` — Weighting vector
real positive vector

Weighting vector, specified as a real positive vector. The weighting vector has the same length as `framelen` and is used to perform least-squares minimization.

Output Arguments

b — Time-varying FIR filter coefficients

matrix

Time-varying FIR filter coefficients, specified as a `framelen`-by-`framelen` matrix. In a smoothing filter implementation (for example, `sgolayfilt`), the last `(framelen-1)/2` rows (each an FIR filter) are applied to the signal during the startup transient, and the first `(framelen-1)/2` rows are applied to the signal during the terminal transient. The center row is applied to the signal in the steady state.

g — Matrix of differentiation filters

matrix

Matrix of differentiation filters, specified as a matrix. 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 `framelen`, you can find an estimate of the pth order derivative, `xp`, of its middle value from `xp((framelen+1)/2) = (factorial(p)) * g(:,p+1)' * x`.

Algorithms

Savitzky-Golay filters are used to smooth out noisy signals with a large frequency span. Savitzky-Golay smoothing filters tend to filter out less of the signal's high-frequency content than standard averaging FIR filters. However, they are less successful at rejecting noise when noise levels are particularly high.

In general, filtering consists of replacing each point of a signal by some combination of the signal values contained in a moving window centered at the point, on the assumption that nearby points measure nearly the same underlying value. For example, moving average filters replace each data point with the local average of the surrounding data points. If a given data point has k points to the left and k points to the right, for a total window length of $L = 2k + 1$, the moving average filter makes the replacement

$$x_s \to \widehat{x}_s = \frac{1}{L} \sum_{r=-k}^{k} x_{s+r}.$$

Savitzky-Golay filters generalize this idea by least-squares fitting an nth-order polynomial through the signal values in the window and taking the calculated central point of the fitted polynomial curve as the new smoothed data point. For a given point, x_s,

$$\begin{bmatrix} x_{s-k} \\ \vdots \\ x_{s-1} \\ x_s \\ x_{s+1} \\ \vdots \\ x_{s+k} \end{bmatrix} = \begin{bmatrix} b_0 + b_1(t_s - k\Delta t) + b_2(t_s - k\Delta t)^2 + \cdots + b_n(t_s - k\Delta t)^n \\ \vdots \\ b_0 + b_1(t_s - 1\Delta t) + b_2(t_s - 1\Delta t)^2 + \cdots + b_n(t_s - 1\Delta t)^n \\ b_0 + b_1(t_s - 0\Delta t) + b_2(t_s - 0\Delta t)^2 + \cdots + b_n(t_s - 0\Delta t)^n \\ b_0 + b_1(t_s + 1\Delta t) + b_2(t_s + 1\Delta t)^2 + \cdots + b_n(t_s + 1\Delta t)^n \\ \vdots \\ b_0 + b_1(t_s + k\Delta t) + b_2(t_s + k\Delta t)^2 + \cdots + b_n(t_s + k\Delta t)^n \end{bmatrix}$$

$$= \begin{bmatrix} a_0 + a_1(-k) + a_2(-k)^2 + \cdots + a_n(-k)^n \\ \vdots \\ a_0 + a_1(-1) + a_2(-1)^2 + \cdots + a_n(-1)^n \\ a_0 + a_1(0) + a_2(0)^2 + \cdots + a_n(0)^n \\ a_0 + a_1(1) + a_2(1)^2 + \cdots + a_n(1)^n \\ \vdots \\ a_0 + a_1(k) + a_2(k)^2 + \cdots + a_n(k)^n \end{bmatrix}$$

or, in terms of matrices,

$$\mathbf{x} = \begin{bmatrix} 1 & -k & (-k)^2 & \cdots & (-k)^n \\ 1 & \vdots & \vdots & \iddots & \vdots \\ 1 & -2 & (-2)^2 & \cdots & (-2)^n \\ 1 & -1 & (-1)^2 & \cdots & (-1)^n \\ 1 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 1^2 & \cdots & 1^n \\ 1 & 2 & 2^2 & \cdots & 2^n \\ 1 & \vdots & \vdots & \ddots & \vdots \\ 1 & k & k^2 & \cdots & k^n \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix} \equiv \mathbf{Ha}.$$

To find the Savitzky-Golay estimates, use the pseudoinverse of **H** to compute **a** and then premultiply by **H**:

$$\widehat{\mathbf{x}} = \mathbf{H}\left(\mathbf{H}^T\mathbf{H}\right)^{-1}\mathbf{H}^T\mathbf{x} = \mathbf{Bx}.$$

To avoid ill-conditioning, `sgolay` uses the `qr` function to compute an economy-size decomposition of **H** as **H** = **QR**, in terms of which **B** = **QQ**^{T}.

It is necessary to compute **B** only once. The Savitzky-Golay estimates for most signal points result from convolving the signal with the center row of **B**. The result is the steady-state portion of the filtered signal. The first k rows of **B** yield the initial transient, and the final k rows of **B** yield the final transient. See `sgolayfilt` for an example. It is possible to improve noise suppression by increasing the window length, but this introduces ringing analogous to the Gibbs phenomenon near any transients.

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1996.

[2] Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. New York: Cambridge University Press, 1992.

[3] Schafer, Ronald W. "What *Is* a Savitzky-Golay Filter? [Lecture Notes]." *IEEE Signal Processing Magazine* Vol. 28, Number 4, July 2011, pp. 111–117. https://doi.org/10.1109/MSP.2011.941097.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`fir1` | `firls` | `filter` | `sgolayfilt`

Introduced before R2006a

sgolayfilt

Savitzky-Golay filtering

Syntax

```
y = sgolayfilt(x,order,framelen)
y = sgolayfilt(x,order,framelen,weights)
y = sgolayfilt(x,order,framelen,weights,dim)
```

Description

`y = sgolayfilt(x,order,framelen)` applies a Savitzky-Golay finite impulse response (FIR) smoothing filter of polynomial order `order` and frame length `framelen` to the data in vector `x`. If `x` is a matrix, then `sgolayfilt` operates on each column.

`y = sgolayfilt(x,order,framelen,weights)` specifies a weighting vector to use during the least-squares minimization.

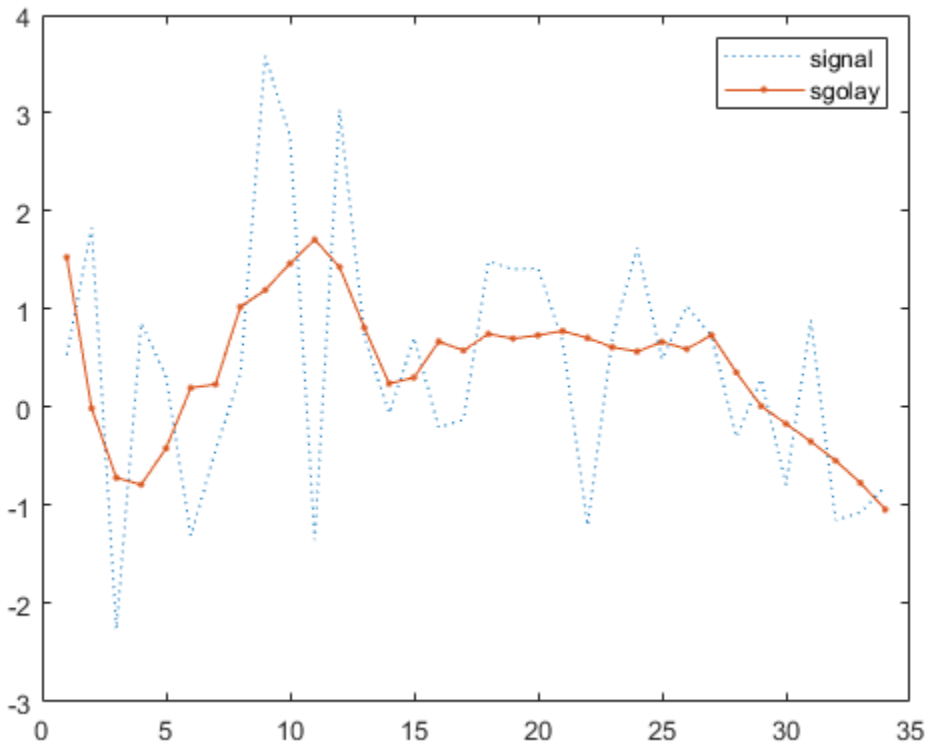
`y = sgolayfilt(x,order,framelen,weights,dim)` specifies the dimension along which the filter operates.

Examples

Steady-State and Transient Savitzky-Golay Filters

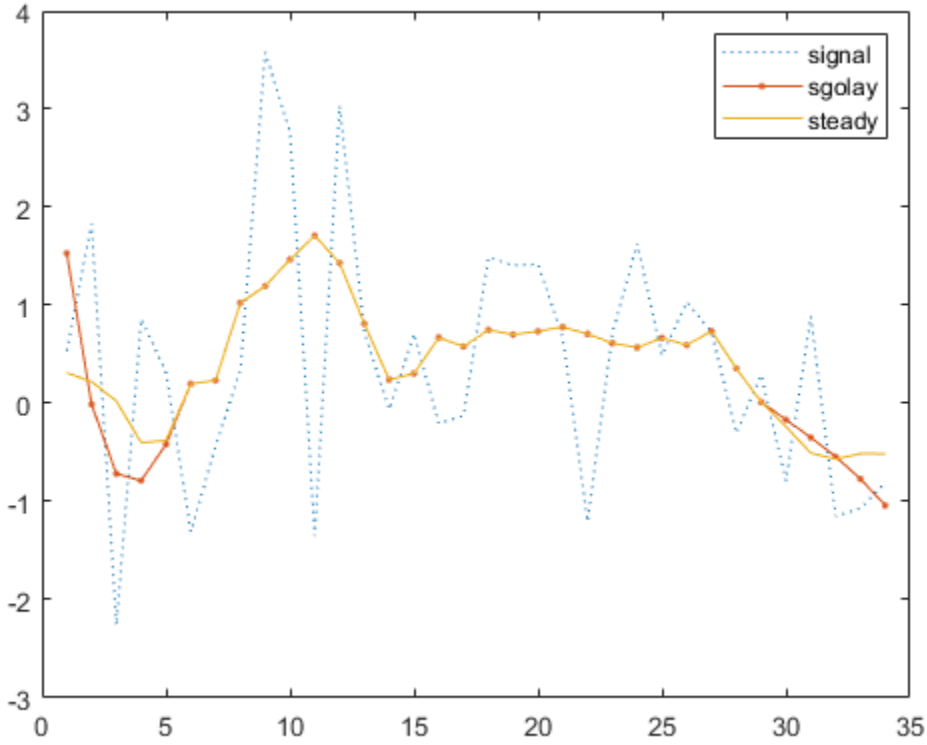
Generate a random signal and smooth it using `sgolayfilt`. Specify a polynomial order of 3 and a frame length of 11. Plot the original and smoothed signals.

```
order = 3;
framelen = 11;

lx = 34;
x = randn(lx,1);

sgf = sgolayfilt(x,order,framelen);

plot(x,':')
hold on
plot(sgf,'.-')
legend('signal','sgolay')
```



The `sgolayfilt` function performs most of the filtering by convolving the signal with the center row of `B`, the output of `sgolay`. The result is the steady-state portion of the filtered signal. Generate and plot this portion.

```
m = (framelen-1)/2;

B = sgolay(order,framelen);

steady = conv(x,B(m+1,:),'same');

plot(steady)
legend('signal','sgolay','steady')
```



Samples close to the signal edges cannot be placed at the center of a symmetric window and have to be treated differently.

To determine the startup transient, matrix multiply the first `(framelen-1)/2` rows of `B` by the first `framelen` samples of the signal.

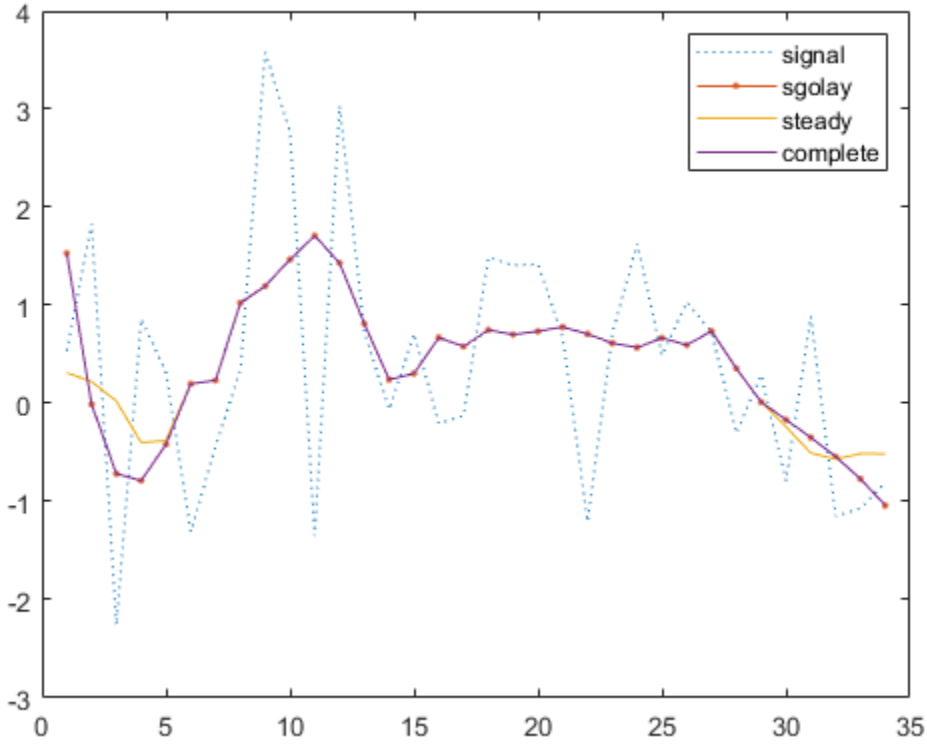
```
ybeg = B(1:m,:)*x(1:framelen);
```

To determine the terminal transient, matrix multiply the final `(framelen-1)/2` rows of `B` by the final `framelen` samples of the signal.

```
yend = B(framelen-m+1:framelen,:)*x(lx-framelen+1:lx);
```

Concatenate the transients and the steady-state portion to generate the complete signal.

```
cmplt = steady;
cmplt(1:m) = ybeg;
cmplt(lx-m+1:lx) = yend;

plot(cmplt)
legend('signal','sgolay','steady','complete')
hold off
```



Adding weights to the minimization breaks the symmetry of B and requires extra steps for a proper solution.

Savitzky-Golay Filtering of Speech Signal

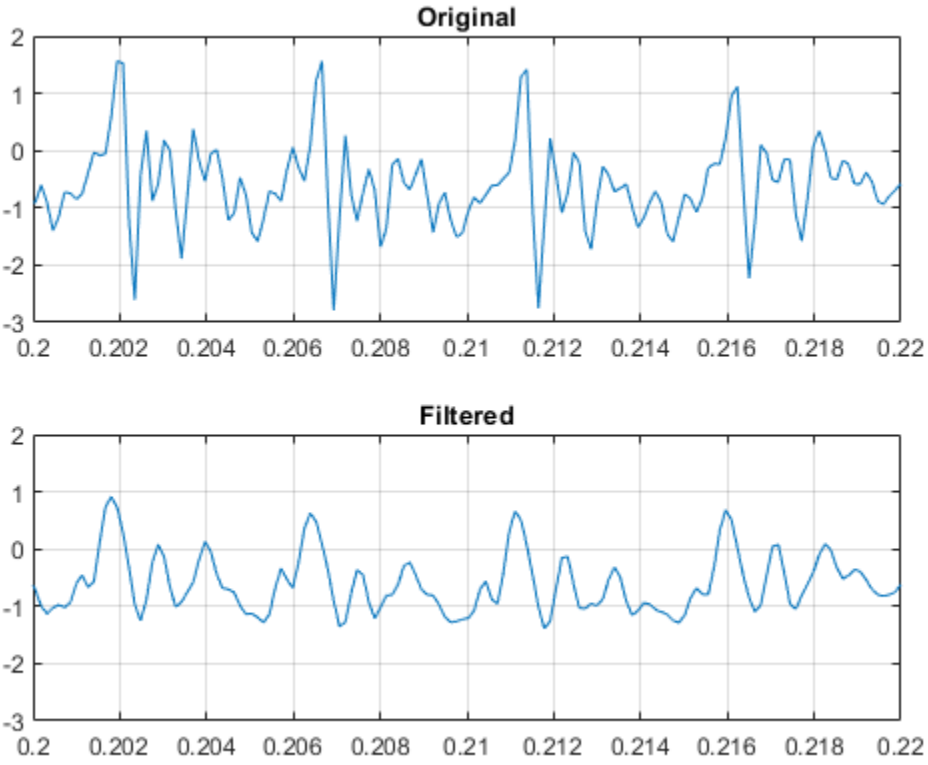
Load a speech signal sampled at F_s = 7418 Hz. The file contains a recording of a female voice saying the word "MATLAB®."

```
load mtlb
t = (0:length(mtlb)-1)/Fs;
```

Smooth the signal by applying a Savitzky-Golay filter of polynomial order 9 to data frames of length 21. Plot the original and filtered signals. Zoom in on a 0.02-second interval.

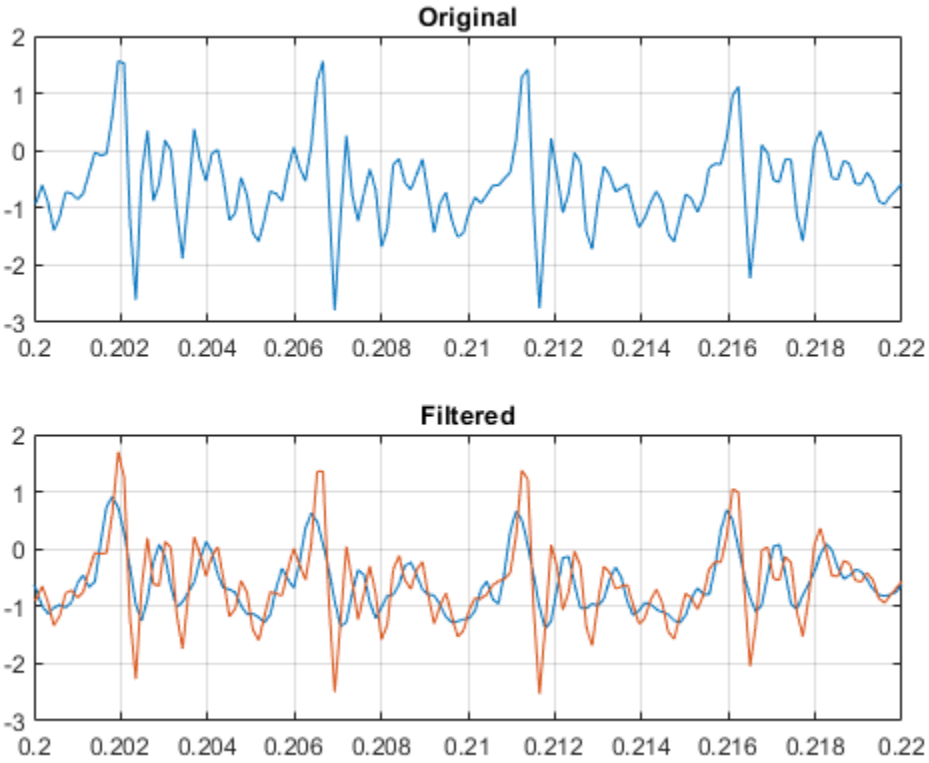
```
rd = 9;
fl = 21;

smtlb = sgolayfilt(mtlb,rd,fl);

subplot(2,1,1)
plot(t,mtlb)
axis([0.2 0.22 -3 2])
title('Original')
grid
```

```
subplot(2,1,2)
plot(t,smtlb)
axis([0.2 0.22 -3 2])
title('Filtered')
grid
```



Repeat the calculation, but now use a Kaiser window as a weighting vector. Specify a shape factor $\beta = 38$. Plot the new filtered signal.

```
kmtlb = sgolayfilt(mtlb,rd,fl,kaiser(fl,38));

subplot(2,1,2)
hold on
plot(t,kmtlb)
axis([0.2 0.22 -3 2])
hold off
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix.

Data Types: `single` | `double`

`order` — Polynomial order

positive integer

Polynomial order, specified as a positive integer. `order` must be smaller than `framelen`. If `order = framelen` – 1, the filter produces no smoothing.

Data Types: `single` | `double`

`framelen` — Frame length

positive odd integer

Frame length, specified as a positive odd integer.

Data Types: `single` | `double`

`weights` — Weighting array

real positive vector | real positive matrix

Weighting array, specified as a real positive vector or matrix of length `framelen`.

Data Types: `single` | `double`

`dim` — Dimension to filter along
positive integer scalar

Dimension to filter along, specified as a positive integer scalar. By default, `sgolayfilt` operates along the first dimension of `x` whose size is greater than 1.

Data Types: `single` | `double`

Output Arguments

y — Filtered signal
vector | matrix

Filtered signal, returned as a vector or matrix.

Tips

Savitzky-Golay smoothing filters are typically used to "smooth out" a noisy signal whose frequency span (without noise) is large. They are also called digital smoothing polynomial filters or least-squares smoothing filters. Savitzky-Golay filters perform better in some applications than standard averaging FIR filters, which tend to filter high-frequency content along with the noise. Savitzky-Golay filters are more effective at preserving high frequency signal components but less successful 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. See `sgolay` for more information about the Savitzky-Golay algorithm.

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1996.

[2] Schafer, Ronald. “What Is a Savitzky-Golay Filter? [Lecture Notes].” *IEEE Signal Processing Magazine* 28, no. 4 (July 2011): 111–17. https://doi.org/10.1109/MSP.2011.941097.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`medfilt1` | `filter` | `sgolay` | `sosfilt`

Introduced before R2006a

shiftdata

Shift data to operate on specified dimension

Syntax

```
[y,perm,nshifts] = shiftdata(x,dim)
```

Description

`[y,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. `perm` is the permutation that the function uses.

Note Use the `shiftdata` function in tandem with the `unshiftdata` function, 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 Magic Square

Shift a 3-by-3 magic square, permuting the second dimension to the first column. Shift the matrix back to its original shape.

Create a 3-by-3 magic square.

```
x = magic(3)
```

```
x = 3×3

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

Shift the matrix to work along the second dimension. Return the permutation vector, the number of shifts, and the shifted matrix.

```
[x,perm,nshifts] = shiftdata(x,2)
```

```
x = 3×3

     8     3     4
     1     5     9
     6     7     2


perm = 1×2
```

```
     2     1


nshifts =

     []
```

Restore the matrix back to its original shape.

```
y = unshiftdata(x,perm,nshifts)
```

```
y = 3×3

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

Rearrange Array to Operate on First Nonsingleton Dimension

Define the data to shift as a row vector.

```
x = 1:5
```

```
x = 1×5

     1     2     3     4     5
```

Define `dim` as empty to shift the first nonsingleton dimension of the data to the first column. `shiftdata` returns the data as a column vector, the permutation vector, and the number of shifts.

```
dim = [];
[x,perm,nshifts] = shiftdata(x,dim)
```

```
x = 5×1

     1
     2
     3
     4
     5


perm =

     []

nshifts = 1
```

Restore the shifted data to its original shape.

```
y = unshiftdata(x,perm,nshifts)
```

```
y = 1×5
```

```
     1     2     3     4     5
```

Input Arguments

`x` — Data

vector | matrix

Data, specified as a vector or matrix.

Data Types: `single` | `double`

`dim` — Dimension to operate along

`[]` | positive integer

Dimension to operate along, specified as a positive integer or `[]`. If `dim` is `[]`, then the function shifts the first nonsingleton dimension to the first column and returns the number of shifts in `nshifts`.

Data Types: `single` | `double`

Output Arguments

`y` — Shifted data

vector | matrix

Shifted data, returned as a vector or matrix.

`perm` — Permutation

vector

Permutation used to shift the data, returned as a vector.

`nshifts` — Number of shifts

scalar

Number of shifts, returned as a scalar.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`permute` | `shiftdim` | `unshiftdata`

Introduced in R2012b

Signal Analyzer

Visualize and compare multiple signals and spectra

Description

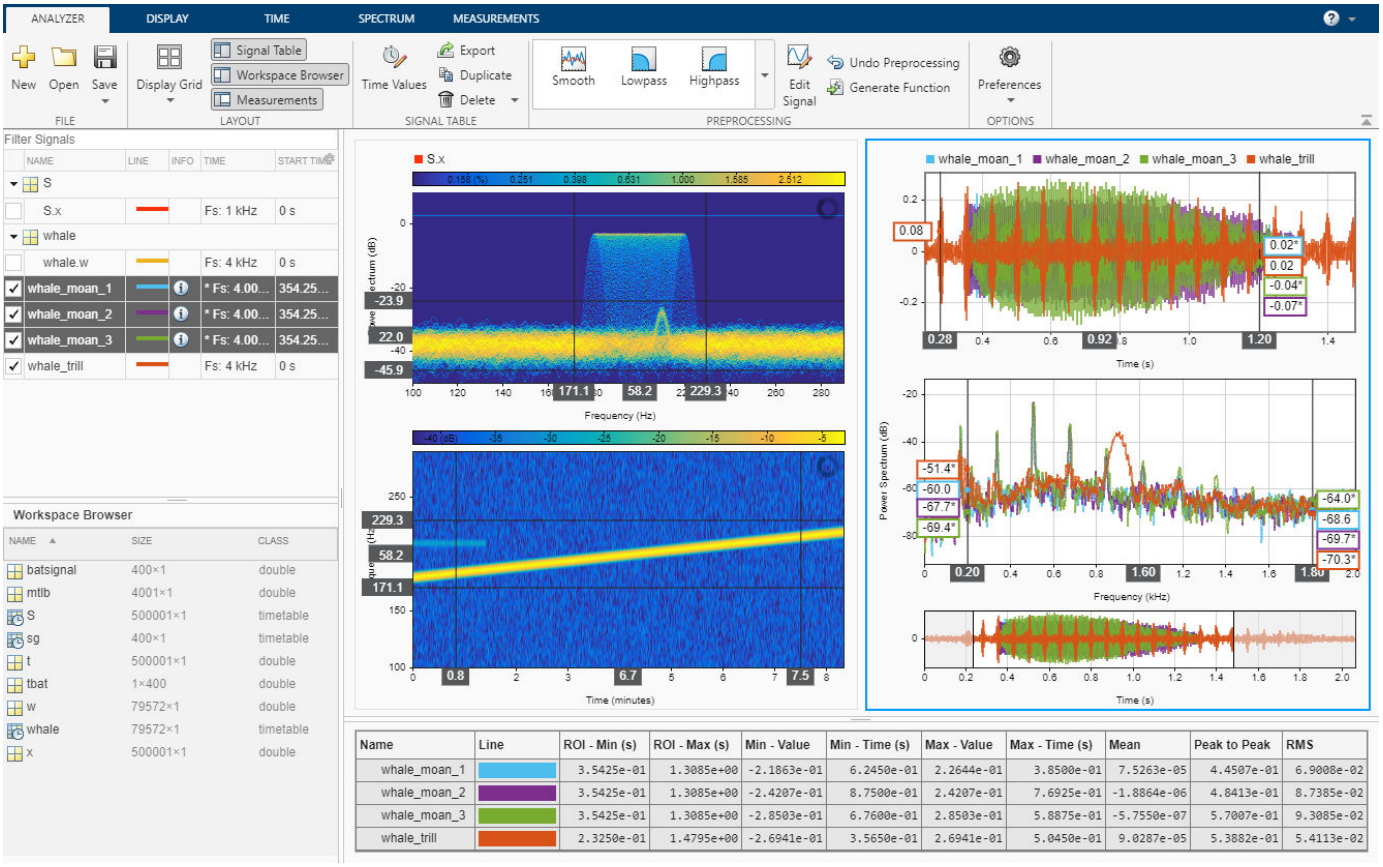
The **Signal Analyzer** app is an interactive tool for visualizing, preprocessing, measuring, analyzing, and comparing signals in the time domain, in the frequency domain, and in the time-frequency domain. Using the app, you can:

- Easily access all the signals in the MATLAB workspace
- Smooth, filter, resample, detrend, denoise, duplicate, extract, rename, and edit signals without leaving the app
- Add and apply custom preprocessing functions
- Visualize and compare multiple waveform, spectrum, persistence, spectrogram, and scalogram representations of signals simultaneously
- Measure data and signal statistics

The **Signal Analyzer** app provides a way to work with many signals of varying durations at the same time and in the same view.

For more information, see **Using Signal Analyzer App.**

- **Signal Analyzer** no longer opens **Signal Labeler**, which is now available as an app. If you want to label signals, open **Signal Labeler** from the MATLAB Toolstrip or the Command Window.
- You need a Wavelet Toolbox™ license to use the scalogram view and to apply wavelet denoising to signals.



Open the Signal Analyzer App

- MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- MATLAB command prompt: Enter `signalAnalyzer`.

Examples

- “Extract Voices from Music Signal”
- “Modulation and Demodulation Using Complex Envelope”
- “Find and Track Ridges Using Reassigned Spectrogram”
- “Declip Saturated Signals Using Your Own Function”
- “Compute Envelope Spectrum of Vibration Signal”
- “Find Delay Between Correlated Signals”
- “Resolve Tones by Varying Window Leakage”
- “Find Interference Using Persistence Spectrum”
- “Denoise Noisy Doppler Signal”
- “Resample and Filter a Nonuniformly Sampled Signal”

- “Extract Regions of Interest from Whale Song”

Programmatic Use

`signalAnalyzer` opens the **Signal Analyzer** app.

`signalAnalyzer(sig)` opens the **Signal Analyzer** app and imports and plots the signal `sig`. If the app is already open, then it plots `sig` in the current display. If `sig` is already plotted but has changed, then the function call updates the plot.

`sig` can be a variable in the workspace or a MATLAB expression. `sig` can be:

- A vector or a matrix with independent signals in each column.
- A `timetable` with time values specified as durations.
- A `timeseries` object.

See “Data Types Supported by Signal Analyzer” for more details.

By default, the app plots the signal as a function of sample index. If you provide time information, or if the signal has inherent time information, then the app plots the signal as a function of time.

`signalAnalyzer(sig1,...,sigN)` imports *N* signal vectors or matrices and plots them in the current display. The app does not support importing signals with inherent time information and signals without inherent time information in the same function call.

`signalAnalyzer(___,'SampleRate',fs)` specifies a sample rate, `fs`, as a positive scalar expressed in Hz. The app uses the sample rate to plot one or more signals against time, assuming a start time of zero. You can specify a sample rate for signals with no inherent time information.

`signalAnalyzer(___,'SampleTime',ts)` specifies a sample time, `ts`, as a positive scalar expressed in seconds. The app uses the sample time to plot one or more signals against time, assuming a start time of zero. You can specify a sample time for signals with no inherent time information.

`signalAnalyzer(___,'StartTime',st)` specifies a signal start time, `st`, as a scalar expressed in seconds. If you do not specify a sample rate or sample time, then the app assumes a sample rate of 1 Hz. You can specify a start time for signals with no inherent time information.

`signalAnalyzer(___,'TimeValues',tv)` specifies a vector, `tv`, with time values corresponding to the data points. `tv` can be a real numeric vector with values expressed in seconds. `tv` can also be a `duration` array. The values in `tv` must be unique and cannot be `NaN`, but they need not be uniformly spaced. All input signals must have the same length as `tv`. You can specify a vector of time values for signals with no inherent time information.

Filtering and scalogram view do not support nonuniformly sampled signals.

Compatibility Considerations

Label button removed from Signal Analyzer
Behavior changed in R2020a

Signal Analyzer no longer opens **Signal Labeler**, which is now available as an app. If you want to label signals, open **Signal Labeler** from the MATLAB Toolstrip or the Command Window.

See Also

Apps
Filter Designer | **Signal Labeler**

Functions
`periodogram` | `pspectrum` | `pwelch` | `spectrogram`

Topics
“Extract Voices from Music Signal”
“Modulation and Demodulation Using Complex Envelope”
“Find and Track Ridges Using Reassigned Spectrogram”
“Declip Saturated Signals Using Your Own Function”
“Compute Envelope Spectrum of Vibration Signal”
“Find Delay Between Correlated Signals”
“Resolve Tones by Varying Window Leakage”
“Find Interference Using Persistence Spectrum”
“Denoise Noisy Doppler Signal”
“Resample and Filter a Nonuniformly Sampled Signal”
“Extract Regions of Interest from Whale Song”
“Time-Frequency Gallery”
“Using Signal Analyzer App”
“Edit Sample Rate and Other Time Information”
“Data Types Supported by Signal Analyzer”
“Spectrum Computation in Signal Analyzer”
“Persistence Spectrum in Signal Analyzer”
“Spectrogram Computation in Signal Analyzer”
“Scalogram Computation in Signal Analyzer”
“Keyboard Shortcuts for Signal Analyzer”
“Signal Analyzer Tips and Limitations”

Introduced in R2016a

signalLabelDefinition

Create signal label definition

Description

Use `signalLabelDefinition` to create signal label definitions for data sets. The labels can correspond to attributes, regions, or points of interest. Use a vector of `signalLabelDefinition` objects to create a `labeledSignalSet`.

Creation

Syntax

```
sld = signalLabelDefinition(name)
sld = signalLabelDefinition(name,Name=Value)
```

Description

`sld = signalLabelDefinition(name)` creates a signal label definition object, `sld`, with the "Name" on page 1-0 property set to `name` and other properties set to default values.

`sld = signalLabelDefinition(name,Name=Value)` sets "Properties" on page 1-2024 using name-value arguments. You can specify multiple name-value arguments. Enclose each property name in quotes.

Input Arguments

`name` — Label name

character vector | string scalar

Label name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Properties

`Name` — Name of label

character vector | string scalar

Name of label, specified as a character vector or string scalar.

Data Types: `char` | `string`

`LabelType` — Type of label

`"attribute"` (default) | `"roi"` | `"point"` | `"attributeFeature"` | `"roiFeature"`

Type of label, specified as one of these:

- `"attribute"` — Define signal characteristics.
- `"roi"` — Define signal characteristics over regions of interest.
- `"point"` — Define signal characteristics over points of interest.
- `"attributeFeature"` — Define signal characteristics that correspond to features.
- `"roiFeature"` — Define signal characteristics over regions of interest that correspond to features.

Data Types: `char` | `string`

`LabelDataType` — Data type of label
`"logical"` (default) | `"categorical"` | `"numeric"` | `"string"` | `"table"` | `"timetable"`

Data type of label, specified as `"logical"`, `"categorical"`, `"numeric"`, `"string"`, `"table"`, or `"timetable"`. When you set this property to `"categorical"`, use the "Categories" on page 1-0 property to specify the array of categories. The object does not support timetable and table data types for `attributeFeature` and `roiFeature` labels.

Data Types: `char` | `string`

`Categories` — Label category names
string array | cell array of character vectors

Label category names, specified as a string array or a cell array of character vectors. The array must have unique elements. This property applies only when the "LabelDataType" on page 1-0 property is set to `"categorical"`.

Example: `LabelDataType="categorical",Categories=["apple","orange"]`

Data Types: `char` | `string`

`ROILimitsDataType` — Data type of ROI limits
`"double"` (default) | `"duration"`

Data type of ROI limits, specified as either `"double"` or `"duration"`. This property applies only when "LabelType" on page 1-0 is set to `"roi"`.

Data Types: `char` | `string`

`PointLocationsDataType` — Data type of point locations
`"double"` (default) | `"duration"`

Data type of point locations, specified as either `"double"` or `"duration"`. This property applies only when "LabelType" on page 1-0 is set to `"point"`.

Data Types: `char` | `string`

`ValidationFunction` — Validation function
function handle

Validation function, specified as a function handle and used when setting label values in a `labeledSignalSet` object. This property applies only when "LabelDataType" on page 1-0 is set to `"categorical"`, `"logical"`, `"numeric"`, `"table"`, or `"timetable"`. If not specified, the function checks only that its input values are of the correct data type. If "LabelDataType" on page 1-0 is set to `"categorical"`, the function checks that the input is one of the values specified using "Categories" on page 1-0 . The function takes an input value and returns `true` if the value is valid and `false` if the value is invalid.

Example: `LabelDataType="numeric",DefaultValue=1,ValidationFunction=@(x)x<2`

Data Types: `function_handle`

`DefaultValue` — Default value of label
`[]` (default) | `LabelDataType` value

Default value of label, specified as a value of the type specified using "LabelDataType" on page 1-0 . If "LabelDataType" on page 1-0 is set to `"categorical"`, then "DefaultValue" on page 1-0 must be one of the values specified using "Categories" on page 1-0 .

Example: `LabelDataType="categorical",Categories=["apple","orange"],DefaultValue="apple"`

Data Types: `char` | `double` | `logical` | `string` | `table`

`Description` — Label description
character vector | string scalar

Label description, specified as a character vector or string scalar.

Example: `Description="Patient is asleep"`

Data Types: `char` | `string`

`Tag` — Label tag identifier
character vector | string scalar

Label tag identifier, specified as a character vector or string scalar. Use this property to identify the same label in a larger labeling scheme or public labeling set.

Example: `Tag="Peak1"`

Data Types: `char` | `string`

`Sublabels` — Array of sublabels
signal label definition object

Array of sublabels, specified as a signal label definition object. To specify more than one sublabel, set this property to a vector of signal label definition objects. Use this property to create a relationship between a parent label and its children. If "LabelType" on page 1-0 is set to `"attributeFeature"` or `"roiFeature"`, then this property does not apply.

Note Sublabels cannot have sublabels.

Example: `Sublabels=[signalLabelDefinition("negative"),signalLabelDefinition("positive")]`

`FrameSize` — Frame size
numeric scalar

Frame size, specified as a numeric scalar. You must specify `FrameSize` when "LabelType" on page 1-0 is set to `"roiFeature"`.

Example: `FrameSize=50`

Data Types: `double`

`FrameOverlapLength` — Overlap length of adjacent frames
`0` (default) | numeric scalar

Overlap length of adjacent frames, specified as a numeric scalar. To enable this property, set "LabelType" on page 1-0 to `"roiFeature"`. You cannot specify `FrameOverlapLength` and `FrameRate` simultaneously. If you do not specify `FramerOverlapLength`, then the object assumes the overlap length to be zero.

Example: `FrameSize=50,FrameOverlapLength=5`

Data Types: `double`

`FrameRate` — Frame rate
`0` (default) | numeric scalar

Frame rate, specified as a numeric scalar. To enable this property, set "LabelType" on page 1-0 to `"roiFeature"`. You cannot specify `FrameRate` and `FrameOverlapLength` simultaneously. If you do not specify `FrameRate`, then the object assumes no overlap between frames.

Example: `FrameSize=50,FrameRate=45`

Data Types: `double`

Object Functions

labelDefinitionsHierarchy	Get hierarchical list of label and sublabel names
labelDefinitionsSummary	Get summary table of signal label definitions

Examples

Label Definitions for Whale Songs

Consider a set of whale sound recordings. The recorded whale sounds consist of trills and moans. *Trills* sound like series of clicks. *Moans* are low-frequency cries similar to the sound made by a ship's horn. You want to look at each signal and label it to identify the whale type, the trill regions, and the moan regions. For each trill region, you also want to label the signal peaks higher than a certain threshold.

Signal Label Definitions

Define an attribute label to store whale types. The possible categories are blue whale, humpback whale, and white whale.

```
dWhaleType = signalLabelDefinition('WhaleType',...
   'LabelType','attribute',...
   'LabelDataType','categorical',...
   'Categories',{'blue','humpback','white'},...
   'Description','Whale type');
```

Define a region-of-interest (ROI) label to capture moan regions. Define another ROI label to capture trill regions.

```
dMoans = signalLabelDefinition('MoanRegions',...
   'LabelType','roi',...
```

```
   'LabelDataType','logical',...
   'Description','Regions where moans occur');

dTrills = signalLabelDefinition('TrillRegions',...
   'LabelType','roi',...
   'LabelDataType','logical',...
   'Description','Regions where trills occur');
```

Finally, define a point label to capture the trill peaks. Set this label as a sublabel of the `dTrills` definition.

```
dTrillPeaks = signalLabelDefinition('TrillPeaks',...
   'LabelType','point',...
   'LabelDataType','numeric',...
   'Description','Trill peaks');

dTrills.Sublabels = dTrillPeaks;
```

Labeled Signal Set

Create a `labeledSignalSet` with the whale signals and the label definitions. Add label values to identify the whale type, the moan and trill regions, and the peaks of the trills.

```
load labelwhalesignals
lbldefs = [dWhaleType dMoans dTrills];

lss = labeledSignalSet({whale1 whale2},lbldefs,'MemberNames',{'Whale1','Whale2'}, ...
   'SampleRate',Fs,'Description','Characterize whale song regions');
```

Visualize the label hierarchy and label properties using `labelDefinitionsHierarchy` and `labelDefinitionsSummary`.

```
labelDefinitionsHierarchy(lss)
```

```
ans =
    'WhaleType
       Sublabels: []
     MoanRegions
       Sublabels: []
     TrillRegions
       Sublabels: TrillPeaks
     '
```

```
labelDefinitionsSummary(lss)
```

```
ans=3×9 table
      LabelName        LabelType     LabelDataType     Categories     ValidationFunction    Defau
    _____________    ___________    _____________    ____________    __________________    _____

    "WhaleType"      "attribute"    "categorical"    {3x1 string}       {["N/A"   ]}       {0x0
    "MoanRegions"    "roi"          "logical"        {["N/A"   ]}       {0x0 double}      {0x0
    "TrillRegions"   "roi"          "logical"        {["N/A"   ]}       {0x0 double}      {0x0
```

The signals in the loaded data correspond to songs of two blue whales. Set the `'WhaleType'` values for both signals.

```
setLabelValue(lss,1,'WhaleType','blue');
setLabelValue(lss,2,'WhaleType','blue');
```

Visualize the `'Labels'` property. The table has the newly added `'WhaleType'` values for both signals.

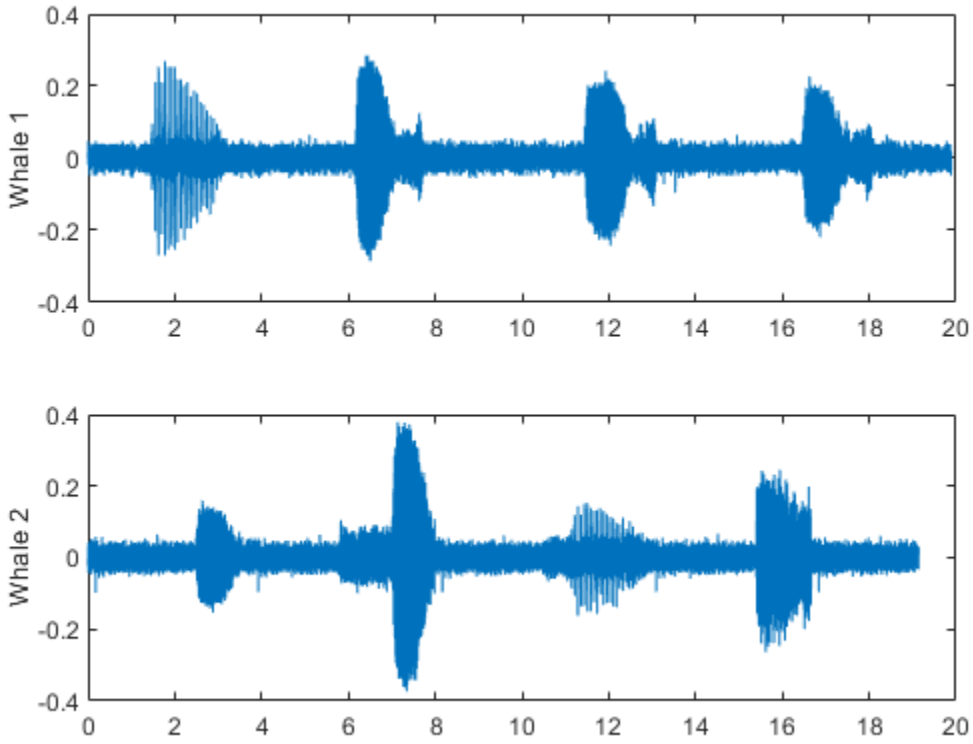
```
lss.Labels
```

```
ans=2×3 table
              WhaleType    MoanRegions    TrillRegions
              _________    ___________    ____________

    Whale1      blue       {0x2 table}    {0x3 table}
    Whale2      blue       {0x2 table}    {0x3 table}
```

Visualize Region Labels

Visualize the whale songs to identify the trill and moan regions.

```
subplot(2,1,1)
plot((0:length(whale1)-1)/Fs,whale1)
ylabel('Whale 1')

subplot(2,1,2)
plot((0:length(whale2)-1)/Fs,whale2)
ylabel('Whale 2')
```



Moan regions are sustained low-frequency wails.

- `whale1` has moans centered at about 7 seconds, 12 seconds, and 17 seconds.
- `whale2` has moans centered at about 3 seconds, 7 seconds, and 16 seconds.

Add the moan regions to the labeled set. Specify the ROI limits in seconds and the label values.

```
moanRegionsWhale1 = [6.1 7.7; 11.4 13.1; 16.5 18.1];
mrsz1 = [size(moanRegionsWhale1,1) 1];
setLabelValue(lss,1,'MoanRegions',moanRegionsWhale1,true(mrsz1));

moanRegionsWhale2 = [2.5 3.5; 5.8 8; 15.4 16.7];
mrsz2 = [size(moanRegionsWhale2,1) 1];
setLabelValue(lss,2,'MoanRegions',moanRegionsWhale2,true(mrsz2));
```

Trill regions have distinct bursts of sound punctuated by silence.

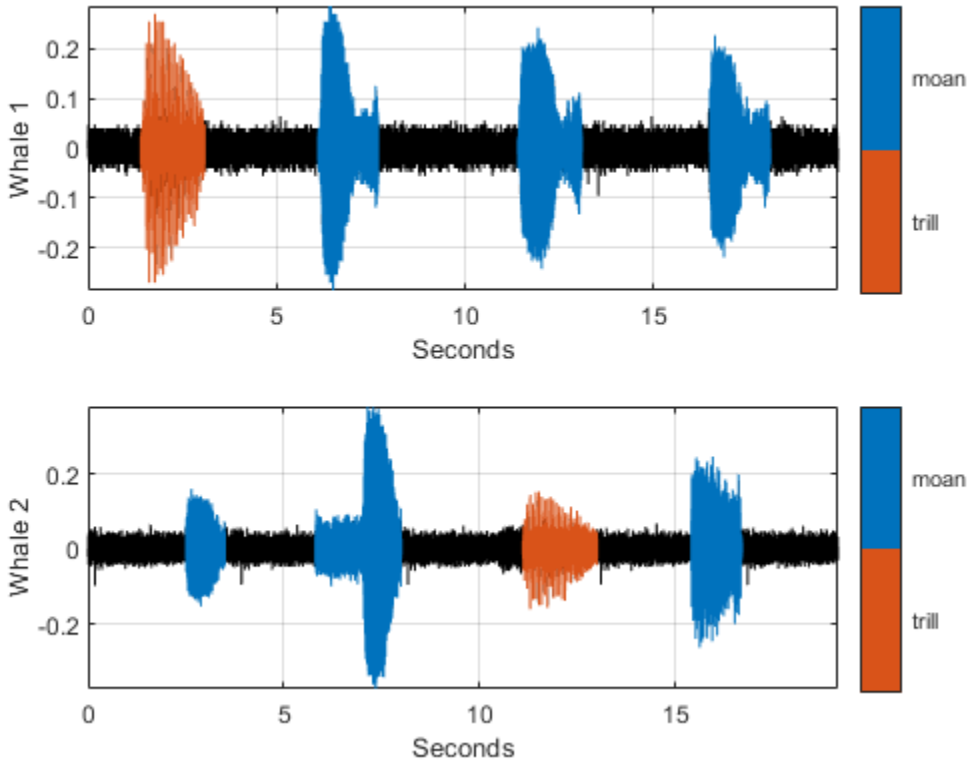
- `whale1` has a trill centered at about 2 seconds.
- `whale2` has a trill centered at about 12 seconds.

Add the trill regions to the labeled set.

```
trillRegionWhale1 = [1.4 3.1];
trsz1 = [size(trillRegionWhale1,1) 1];
setLabelValue(lss,1,'TrillRegions',trillRegionWhale1,true(trsz1));

trillRegionWhale2 = [11.1 13];
trsz2 = [size(trillRegionWhale1,1) 1];
setLabelValue(lss,2,'TrillRegions',trillRegionWhale2,true(trsz2));
```

Create a `signalMask` object for each whale song and use it to visualize and label the different regions. For better visualization, change the label values from logical to categorical.

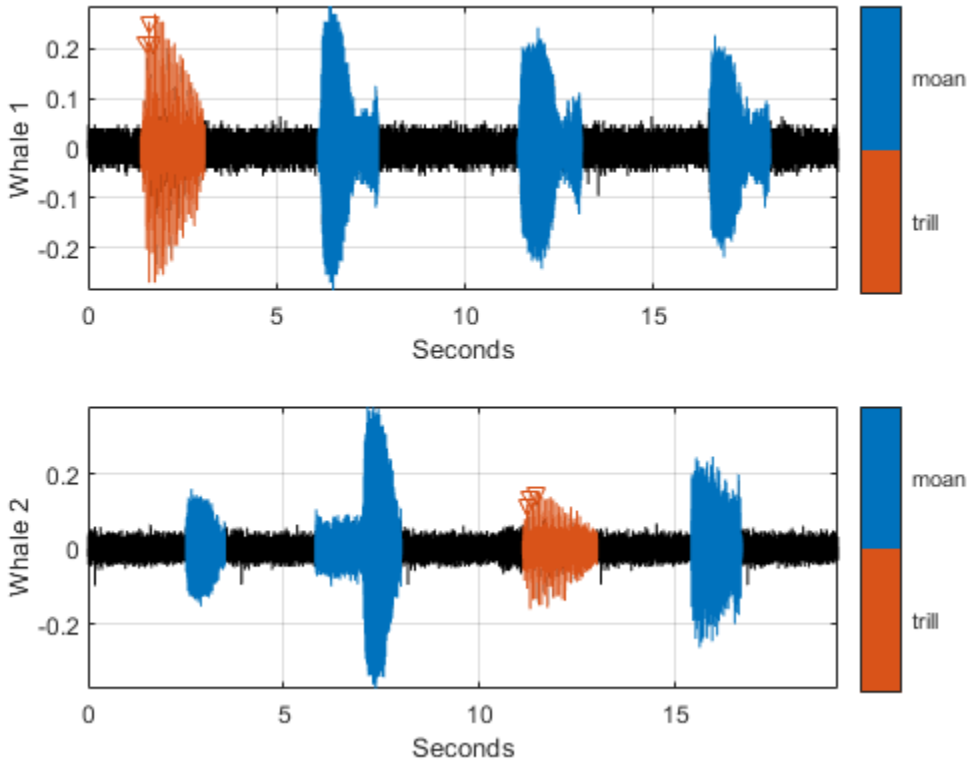
```
mr1 = getLabelValues(lss,1,'MoanRegions');
mr1.Value = categorical(repmat("moan",mrsz1));
tr1 = getLabelValues(lss,1,'TrillRegions');
tr1.Value = categorical(repmat("trill",trsz1));

msk1 = signalMask([mr1;tr1],'SampleRate',Fs);

subplot(2,1,1)
plotsigroi(msk1,whale1)
ylabel('Whale 1')
hold on

mr2 = getLabelValues(lss,2,'MoanRegions');
mr2.Value = categorical(repmat("moan",mrsz2));
tr2 = getLabelValues(lss,2,'TrillRegions');
tr2.Value = categorical(repmat("trill",trsz2));

msk2 = signalMask([mr2;tr2],'SampleRate',Fs);

subplot(2,1,2)
plotsigroi(msk2,whale2)
ylabel('Whale 2')
hold on
```



Visualize Point Labels

Label three peaks for each trill region. For point labels, you specify the point locations and the label values. In this example, the point locations are in seconds.

```
peakLocsWhale1 = [1.553 1.626 1.7];
peakValsWhale1 = [0.211 0.254 0.211];

setLabelValue(lss,1,{'TrillRegions','TrillPeaks'}, ...
   peakLocsWhale1,peakValsWhale1,'LabelRowIndex',1);

subplot(2,1,1)
plot(peakLocsWhale1,peakValsWhale1,'v')
hold off

peakLocsWhale2 = [11.214 11.288 11.437];
peakValsWhale2 = [0.119 0.14 0.15];

setLabelValue(lss,2,{'TrillRegions','TrillPeaks'}, ...
   peakLocsWhale2,peakValsWhale2,'LabelRowIndex',1);

subplot(2,1,2)
plot(peakLocsWhale2,peakValsWhale2,'v')
hold off
```



Explore Label Values

Explore the label values using `getLabelValues`.

```
getLabelValues(lss)
```

```
ans=2×3 table
              WhaleType    MoanRegions    TrillRegions
              _________    ___________    ____________

    Whale1      blue       {3x2 table}    {1x3 table}
    Whale2      blue       {3x2 table}    {1x3 table}
```

Retrieve the moan regions for the first member of the labeled set.

```
getLabelValues(lss,1,'MoanRegions')
```

```
ans=3×2 table
     ROILimits      Value
    ____________    _____

     6.1     7.7    {[1]}
    11.4    13.1    {[1]}
    16.5    18.1    {[1]}
```

Use a second output argument to list the sublabels of a label.

```
[value,valueWithSublabel] = getLabelValues(lss,1,'TrillRegions')
```

```
value=1×2 table
    ROILimits     Value
    __________    _____

    1.4    3.1    {[1]}


valueWithSublabel=1×3 table
    ROILimits     Value     Sublabels
                           TrillPeaks
    __________    _____    ___________

    1.4    3.1    {[1]}    {3x2 table}
```

To retrieve the values in a sublabel, express the label name as a two-element array.

```
getLabelValues(lss,1,{'TrillRegions','TrillPeaks'})
```

```
ans=3×2 table
    Location      Value
    ________    __________

     1.553      {[0.2110]}
     1.626      {[0.2540]}
       1.7      {[0.2110]}
```

Find the value of the third trill peak corresponding to the second member of the set.

```
getLabelValues(lss,2,{'TrillRegions','TrillPeaks'}, ...
    'LabelRowIndex',1,'SublabelRowIndex',3)
```

```
ans=1×2 table
    Location      Value
    ________    __________

     11.437     {[0.1500]}
```

Count Label Values and Create Datastores

Specify the path to a set of audio signals included as MAT-files with MATLAB®. Each file contains a signal variable and a sample rate. List the names of the files.

```
folder = fullfile(matlabroot,"toolbox","matlab","audiovideo");
lst = dir(append(folder,"/*.mat"));
nms = {lst(:).name}'
```

```
nms = 7x1 cell
    {'chirp.mat'   }
    {'gong.mat'    }
    {'handel.mat'  }
    {'laughter.mat'}
    {'mtlb.mat'    }
```

```
    {'splat.mat'   }
    {'train.mat'   }
```

Create a signal datastore that points to the specified folder. Set the sample rate variable name to `Fs`, which is common to all files. Generate a subset of the datastore that excludes the file `mtlb.mat`. Use the subset datastore as the source for a `labeledSignalSet` object.

```
sds = signalDatastore(folder,"SampleRateVariableName","Fs");
sds = subset(sds,~strcmp(nms,"mtlb.mat"));
lss = labeledSignalSet(sds);
```

Create three label definitions to label the signals:

- Define a logical attribute label that is true for signals that contain human voices.
- Define a numeric point label that marks the location and amplitude of the maximum of each signal.
- Define a categorical region-of-interest (ROI) label to pick out nonoverlapping, uniform-length random regions of each signal.

Add the signal label definitions to the labeled signal set.

```
vc = signalLabelDefinition("Voice",'LabelType','attribute', ...
    'LabelDataType','logical','DefaultValue',false);
mx = signalLabelDefinition("Maximum",'LabelType','point', ...
    'LabelDataType','numeric');
rs = signalLabelDefinition("RanROI",'LabelType','ROI', ...
    'LabelDataType','categorical','Categories',["ROI" "other"]);
addLabelDefinitions(lss,[vc mx rs])
```

Label the signals:

- Label `'handel.mat'` and `'laughter.mat'` as having human voices.
- Use the `islocalmax` function to find the maximum of each signal. Label its location and value.
- Use the `randROI` on page 1-0 function to generate as many regions of length $N/10$ samples as can fit in a signal of length N given a minimum separation of $N/6$ samples between regions. Label their locations and assign them to the `ROI` category.

When labeling points and regions, convert sample values to time values. Subtract 1 to account for MATLAB® array indexing and divide by the sample rate.

```
kj = 1;
while hasdata(sds)

    [sig,info] = read(sds);
    fs = info.SampleRate;

    [~,fn] = fileparts(info.FileName);
    if fn=="handel" || fn=="laughter"
        setLabelValue(lss,kj,"Voice",true)
    end

    xm = find(islocalmax(sig,'MaxNumExtrema',1));
    setLabelValue(lss,kj,"Maximum",(xm-1)/fs,sig(xm))

    N = length(sig);
```

```
    rois = randROI(N,round(N/10),round(N/6));
    setLabelValue(lss,kj,"RanROI",(rois-1)/fs,repelem("ROI",size(rois,1)))

    kj = kj+1;

end
```

Verify that only two signals contain voices.

```
countLabelValues(lss,"Voice")
```

```
ans=2×3 table
    Voice    Count    Percent
    _____    _____    _______

    false      4      66.667
    true       2      33.333
```

Verify that two signals have a maximum amplitude of 1.

```
countLabelValues(lss,"Maximum")
```

```
ans=5×4 table
           Maximum             Count    Percent    MemberCount
    ___________________________    _____    _______    ___________

    0.80000000000000004441         1      16.667           1
    0.89113331915798421612         1      16.667           1
    0.94730769230769229505         1      16.667           1
    1                              2      33.333           2
    1.0575668990330560071          1      16.667           1
```

Verify that each signal has four nonoverlapping random regions of interest.

```
countLabelValues(lss,"RanROI")
```

```
ans=2×4 table
    RanROI    Count    Percent    MemberCount
    ______    _____    _______    ___________

    ROI        24        100           6
    other       0          0           0
```

Create two datastores with the data in the labeled signal set:

- The `signalDatastore` object `sd` contains the signal data.
- The `arrayDatastore` object `ld` contains the labeling information. Specify that you want to include the information corresponding to all the labels you created.

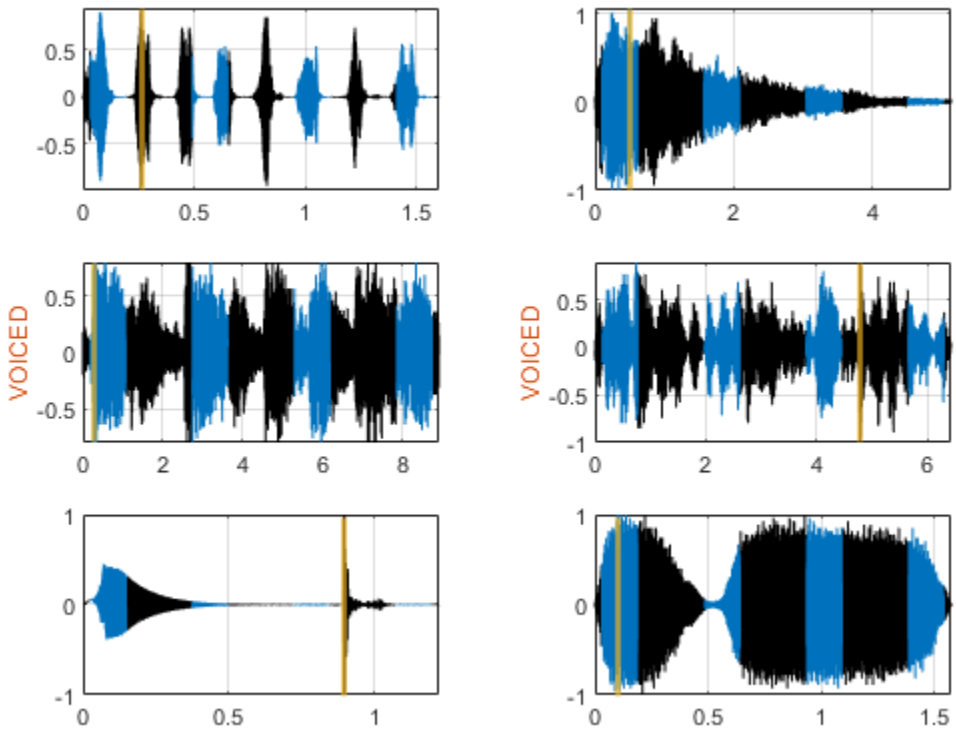
```
[sd,ld] = createDatastores(lss,["Voice" "RanROI" "Maximum"]);
```

Use the information in the datastores to plot the signals and display their labels.

- Use a `signalMask` object to highlight the regions of interest in blue.
- Plot yellow lines to mark the locations of the maxima.

- Add a red axis label to the signals that contain human voices.

```
tiledlayout flow

while hasdata(sd)

    [sg,nf] = read(sd);

    lbls = read(ld);

    nexttile

    msk = signalMask(lbls{:}.RanROI{:},'SampleRate',nf.SampleRate);
    plotsigroi(msk,sg)
    colorbar off
    xlabel('')

    xline(lbls{:}.Maximum{:}.Location, ...
        'LineWidth',2,'Color','#EDB120')

    if lbls{:}.Voice{:}
        ylabel('VOICED','Color','#D95319')
    end

end
```




```
function roilims = randROI(N,wid,sep)
```

```
num = floor((N+sep)/(wid+sep));
hq = histcounts(randi(num+1,1,N-num*wid-(num-1)*sep),(1:num+2)-1/2);
roilims = (1 + (0:num-1)*(wid+sep) + cumsum(hq(1:num)))' + [0 wid-1];

end
```

See Also

Apps
Signal Labeler

Objects
`labeledSignalSet` | `signalMask`

Introduced in R2018b

Signal Labeler

Label signal attributes, regions, and points of interest, and extract features

Description

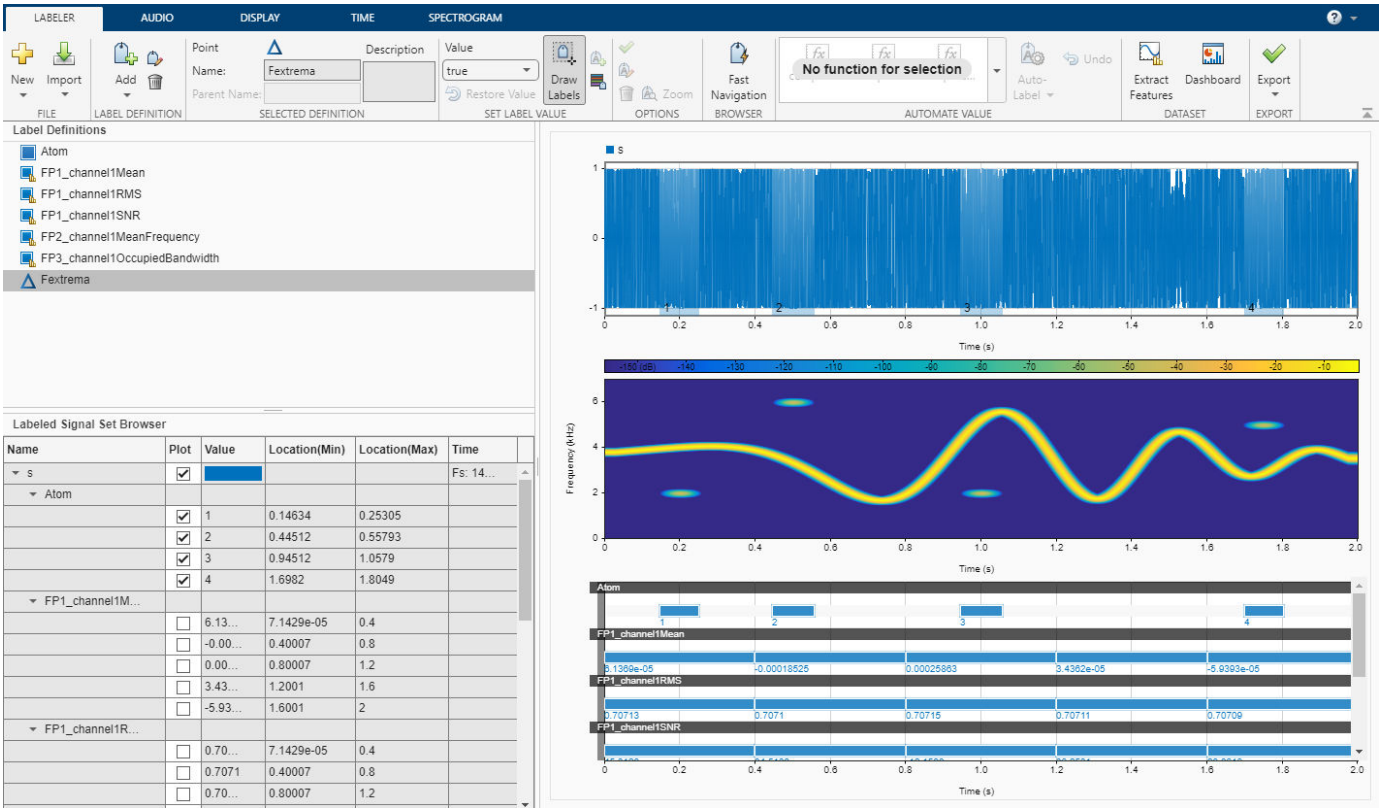
The **Signal Labeler** app is an interactive tool that enables you to label signals for analysis or for use in machine learning and deep learning applications. Using **Signal Labeler**, you can:

- Label signal attributes, regions, and points of interest
- Use logical, categorical, numerical, or string-valued labels
- Automatically label signal peaks or apply custom labeling functions
- Import, label, and play audio signals
- Use frequency and time-frequency views to aid labeling
- Generate attribute and region-of-interest (ROI) labels from signal features
- Add, edit, and delete labels or sublabels
- Display selected subsets of signals and labels

Signal Labeler saves data as `labeledSignalSet` objects. You can use `labeledSignalSet` objects to train a network, classifier, or analyze data and report statistics.

For more information, see **“Using Signal Labeler App”.**

- You need an Audio Toolbox license to label audio signals.



Open the Signal Labeler App

- MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- MATLAB command prompt: Enter `signalLabeler`.

Examples

- “Label Signal Attributes, Regions of Interest, and Points”
- “Label ECG Signals and Track Progress”
- “Examine Labeled Signal Set”
- “Automate Signal Labeling with Custom Functions”
- “Label Spoken Words in Audio Signals”
- “Label Radar Signals with Signal Labeler” (Radar Toolbox)

Programmatic Use

`signalLabeler` opens the **Signal Labeler** app.

See Also

Apps
Signal Analyzer

Functions
`labeledSignalSet` | `signalLabelDefinition`

Topics
“Label Signal Attributes, Regions of Interest, and Points”
“Label ECG Signals and Track Progress”
“Examine Labeled Signal Set”
“Automate Signal Labeling with Custom Functions”
“Label Spoken Words in Audio Signals”
“Label Radar Signals with Signal Labeler” (Radar Toolbox)
“Using Signal Labeler App”
“Import Data into Signal Labeler”
“Import and Play Audio File Data in Signal Labeler”
“Create or Import Signal Label Definitions”
“Label Signals Interactively or Automatically”
“Custom Labeling Functions”
“Customize Labeling View”
“Feature Extraction Using Signal Labeler”
“Dashboard”
“Export Labeled Signal Sets and Signal Label Definitions”
“Signal Labeler Usage Tips”

Introduced in R2019a

signalDatastore

Datastore for collection of signals

Description

Use a `signalDatastore` object to manage a collection of in-memory data or signal files, where each individual file fits in memory, but the entire collection does not necessarily fit.

Creation

Syntax

```
sds = signalDatastore(data)
sds = signalDatastore(location)
sds = signalDatastore( ___ ,Name,Value)
```

Description

`sds = signalDatastore(data)` creates a signal datastore with in-memory input signals contained in `data`.

`sds = signalDatastore(location)` creates a signal datastore based on a collection of either MAT-files or CSV files in `location`. If `location` contains a mixture of MAT-files and CSV files, then `sds` contains MAT-files.

`sds = signalDatastore( ___ ,Name,Value)` specifies additional properties using one or more name-value arguments.

Input Arguments

`data` — In-memory input data

cell array of vectors | cell array of matrices | cell array of timetables | cell array of cell arrays

In-memory input data, specified as vectors, matrices, timetables, or cell arrays. Each element of `data` is a member that is output by the datastore on each call to `read`.

Example: `{randn(100,1); randn(120,3); randn(135,2); randn(100,1)}`

`location` — Files or folders to include in datastore

`FileSet` object | path | `DsFileSet` object

Files or folders included in the datastore, specified as a `FileSet` object, as file paths, or as a `DsFileSet` object.

- `FileSet` object — You can specify `location` as a `FileSet` object. Specifying the location as a `FileSet` object leads to a faster construction time for datastores compared to specifying a path or `DsFileSet` object. For more information, see `matlab.io.datastore.FileSet`.
- File path — You can specify a single file path as a character vector or string scalar. You can specify multiple file paths as a cell array of character vectors or a string array.

- `DsFileSet` object — You can specify a `DsFileSet` object. For more information, see `matlab.io.datastore.DsFileSet`.

Files or folders may be local or remote:

- Local files or folders — Specify local paths to files or folders. If the files are not in the current folder, then specify full or relative paths. Files within subfolders of the specified folder are not automatically included in the datastore. You can use the wildcard character (*) when specifying the local path. This character specifies that the datastore include all matching files or all files in the matching folders.
- Remote files or folders — Specify full paths to remote files or folders as a uniform resource locator (URL) of the form `hdfs:///path_to_file`. For more information, see “Work with Remote Data”.

When you specify a folder, the datastore includes only files with supported file formats and ignores files with any other format. To specify a custom list of file extensions to include in your datastore, see the `FileExtensions` property.

Example: `'whale.mat'`

Example: `'../dir/data/signal.mat'`

Data Types: `char` | `string` | `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `sds = signalDatastore('C:\dir\signaldata','FileExtensions','.csv')`

`IncludeSubfolders` — Subfolder inclusion flag
`false` or `0` (default) | `true` or `1`

Subfolder inclusion flag, specified as `true` or `false`. Specify `true` to include all files and subfolders within each folder or `false` to include only the files within each folder.

Example: `'IncludeSubfolders',true`

Data Types: `logical` | `double`

`FileExtensions` — Signal file extensions
character vector | cell array of character vectors | string scalar | string array

Signal file extensions, specified as a string scalar, string array, character vector, or cell array of character vectors.

If no read function is specified, `'FileExtensions'` can only be set to `.mat` to read MAT-files, or to `.csv` to read CSV files. If `'FileExtensions'` is omitted, it defaults to `.mat` if there are MAT-files in the specified location, otherwise `'FileExtensions'` defaults to `.csv` if there are CSV files in the specified location.

If the specified location contains both MAT-files and CSV files, `signalDatastore` defaults to reading the MAT-files. If neither MAT-files nor CSV files are present, `signalDatastore` errors out with the default `read` function. Specify a custom `read` using `ReadFcn` function to read files of any other type.

When you do not specify a file extension, the `signalDatastore` needs to parse the files to decide the default extension to read. Specify an extension to avoid the parsing time.

Example: `'FileExtensions','.csv'`

Data Types: `string` | `char` | `cell`

In addition to these name-value arguments, you also can specify any of the properties on this page as name-value pairs, except for the `Files` property.

Properties

In-Memory Data

`Members` — Member names

cell array

Member names, specified as a cell array. The length of the member names for the input data should equal the length of the `data` cell array. This property applies only when the datastore contains in-memory data.

`MemberNames` — Signal member data

`["Member1"..."MemberN"]` (default) | string scalar | string array

Signal member data, specified as a string scalar or a string array. The length of the member names for the input data should equal the length of the `data` cell array. This property applies only when the datastore contains in-memory data.

File Data

`Files` — Files included in datastore

cell array of strings | cell array of character vectors

Files included in the datastore, specified as a cell array of strings or character vectors. Each character vector in the cell array represents the full path to a file. The `location` argument in the `signalDatastore` defines `Files` when the datastore is created. This property applies only when the datastore contains file data.

Data Types: `string` | `char` | `cell`

`ReadFcn` — Custom read function

`read` (default) | function handle

Function that reads data, specified as a function handle. The function must take a file name as input, and then it outputs the corresponding data. For example, if `customreader` is the specified function to read the data, then it must have one of these templates:

```
function data = customreader(filename)
...
end

function [data,info] = customreader(filename)
...
end
```

The signal data is output in the `data` variable. The `info` variable must be a user-defined structure containing user-defined information from the file. If you need extra arguments, you can include them

after the `filename` argument. `signalDatastore` appends to the `info` structure a field containing the name of the file.

Example: `@customreader`

Data Types: `function_handle`

`AlternateFileSystemRoots` — Alternate file system root paths
string vector | cell array

Alternate file system root paths, specified as the name-value argument consisting of `"AlternateFileSystemRoots"` and a string vector or a cell array. Use `"AlternateFileSystemRoots"` when you create a datastore on a local machine, but need to access and process the data on another machine (possibly of a different operating system). Also, when processing data using the Parallel Computing Toolbox and the MATLAB Parallel Server™, and the data is stored on your local machines with a copy of the data available on different platform cloud or cluster machines, you must use `"AlternateFileSystemRoots"` to associate the root paths.

- To associate a set of root paths that are equivalent to one another, specify `"AlternateFileSystemRoots"` as a string vector. For example,

  ```
  ["Z:\datasets","/mynetwork/datasets"]
  ```

- To associate multiple sets of root paths that are equivalent for the datastore, specify `"AlternateFileSystemRoots"` as a cell array containing multiple rows where each row represents a set of equivalent root paths. Specify each row in the cell array as either a string vector or a cell array of character vectors. For example:

  - Specify `"AlternateFileSystemRoots"` as a cell array of string vectors.

    ```
    {["Z:\datasets", "/mynetwork/datasets"];...
     ["Y:\datasets", "/mynetwork2/datasets","S:\datasets"]}
    ```

  - Alternatively, specify `"AlternateFileSystemRoots"` as a cell array of cell array of character vectors.

    ```
    {{'Z:\datasets','/mynetwork/datasets'};...
     {'Y:\datasets', '/mynetwork2/datasets','S:\datasets'}}
    ```

The value of `"AlternateFileSystemRoots"` must satisfy these conditions:

- Contains one or more rows, where each row specifies a set of equivalent root paths.
- Each row specifies multiple root paths and each root path must contain at least two characters.
- Root paths are unique and are not subfolders of one another.
- Contains at least one root path entry that points to the location of the files.

For more information, see “Set Up Datastore for Processing on Different Machines or Clusters”.

Example: `["Z:\datasets","/mynetwork/datasets"]`

Data Types: `string` | `cell`

`SignalVariableNames` — Names of variables in signal files
first variable name (default) | string scalar | string vector

Names of variables in signal files, specified as a string scalar or vector of unique names. Use this property when your files contain more than one variable and you want to specify the names of the variables that hold the signal data you want to read.

- When the property value is a string scalar, `signalDatastore` returns data contained in the specified variable.
- When the property value is a string vector, `signalDatastore` returns a cell array with the data contained in the specified variables. In this case, you can use the `ReadOutputOrientation` property to specify the orientation of the output cell array as a column or a row.

If this property is not specified, `signalDatastore` reads the first variable in the variable list of each file.

Note To determine the name of the first variable in a file, `signalDatastore` follows these steps:

- For MAT-files:

```
s = load(fileName);
varNames = fieldnames(s);
firstVar = s.(varNames{1});
```

- For CSV files:

```
opts = detectImportOptions(fileName,'PreserveVariableNames',true);
varNames = opts.VariableNames;
firstVar = string(varNames{1});
```

This property applies only when the datastore contains file data and the default read function is used.

`ReadOutputOrientation` — Output signal data cell array orientation
`'column'` (default) | `'row'`

Output signal data cell array orientation, specified as `'column'` or `'row'`. This property specifies how to orient the output signal data cell array after a call to the `read` function when `SignalVariableNames` contains more than one signal name. `ReadOutputOrientation` has no effect when `SignalVariableNames` contains only one element and does not apply if `SignalVariableNames` has not been specified.

This property applies only when the datastore contains file data and the default read function is used.

Example: Output Cell Array Orientation

In the “Read Multiple Variables from Files in Signal Datastore” on page 1-2053 example, `data` has the default output orientation and is a 2-by-1 column array:

```
    {1×4941 double}
    {1×4941 double}
```

If you specify `ReadOutputOrientation` as `'row'`, then `data` is a 1-by-2 row array:

```
    {1×4941 double}    {1×4941 double}
```

`SampleRateVariableName` — Name of variable holding sample rate
string scalar

Name of the variable holding the sample rate, specified as a string scalar. This property applies only when the datastore contains file data.

SampleTimeVariableName — Name of variable holding sample time value
string scalar

Name of the variable holding the sample time value, specified as a string scalar. This property applies only when the datastore contains file data.

TimeValuesVariableName — Name of variable holding time values vector
string scalar

Name of the variable holding the time values vector, specified as a string scalar. This property applies only when the datastore contains file data.

Note `'SampleRateVariableName'`, `'SampleTimeVariableName'`, and `'TimeValuesVariableName'` are mutually exclusive. Use these properties when your files contain a variable that holds the time information of the signal data. If not specified, `signalDatastore` assumes that signal data has no time information. These properties are not valid if a custom `read` function is specified.

In-Memory and File Data

SampleRate — Sample rate values
positive scalar | positive vector

Sample rate values, specified as a positive real scalar or vector.

- Set the value of `SampleRate` to a scalar to specify the same sample rate for all signals in the `signalDatastore`.
- Set the value of `SampleRate` to a vector to specify a different sample rate for each signal in the `signalDatastore`.

The number of elements in the vector must equal the number of elements in the `signalDatastore`.

SampleTime — Sample time values
positive scalar | vector | `duration` scalar | `duration` vector

Sample time values, specified as a positive scalar, a vector, a `duration` scalar, or a `duration` vector.

- Set the value of `SampleTime` to a scalar to specify the same sample time for all signals in the `signalDatastore`.
- Set the value of `SampleTime` to a vector to specify a different sample time for each signal in the `signalDatastore`.

The number of elements in the vector must equal the number of elements in the `signalDatastore`.

TimeValues — Time values
vector | `duration` vector | matrix | cell array

Time values, specified as a vector, a `duration` vector, a matrix, or a cell array.

- Set `TimeValues` to a numeric or `duration` vector to specify the same time values for all signals in the `signalDatastore`. The vector must have the same length as all the signals in the set.

- Set `TimeValues` to a numeric or `duration` matrix or cell array to specify that each signal of the `signalDatastore` has signals with the same time values, but the time values differ from signal to signal.
  - If `TimeValues` is a matrix, then the number of columns equal the number of members of the `signalDatastore`. All signals in the datastore must have a length equal to the number of rows of the matrix.
  - If `TimeValues` is a cell array, then the number of vectors equal the number of members of the `signalDatastore`. All signals in a member must have a length equal to the number of elements of the corresponding vector in the cell array.

`ReadSize` — Maximum number of signal files returned by `read`
1 (default) | positive real scalar

Maximum number of signal files returned by `read`, specified as a positive real scalar. If you set the `ReadSize` property to *n*, such that $n > 1$, each time you call the `read` function, the function reads:

- The first variable of the first *n* files, if `sds` contains file data.
- The first *n* members, if `sds` contains in-memory data.

The output of `read` is a cell array of signal data when `ReadSize` > 1.

Object Functions

read	Read next consecutive signal observation
readall	Read all signals from datastore
writeall	Write datastore to files
preview	Read first signal observation from datastore for preview
shuffle	Shuffle signals in signal datastore
subset	Create datastore with subset of signals
partition	Partition signal datastore and return partitioned portion
numpartitions	Return estimate for reasonable number of partitions for parallel processing
reset	Reset datastore to initial state
progress	Determine how much data has been read
hasdata	Determine if data is available to read
transform	Transform datastore
combine	Combine data from multiple datastores
isPartitionable	Determine whether datastore is partitionable
isShuffleable	Determine whether datastore is shuffleable

Note `isPartitionable` and `isShuffleable` return `true` by default for `signalDatastore`. You can test if the output of `combine` and `transform` are partitionable or shuffleable using the two functions.

Examples

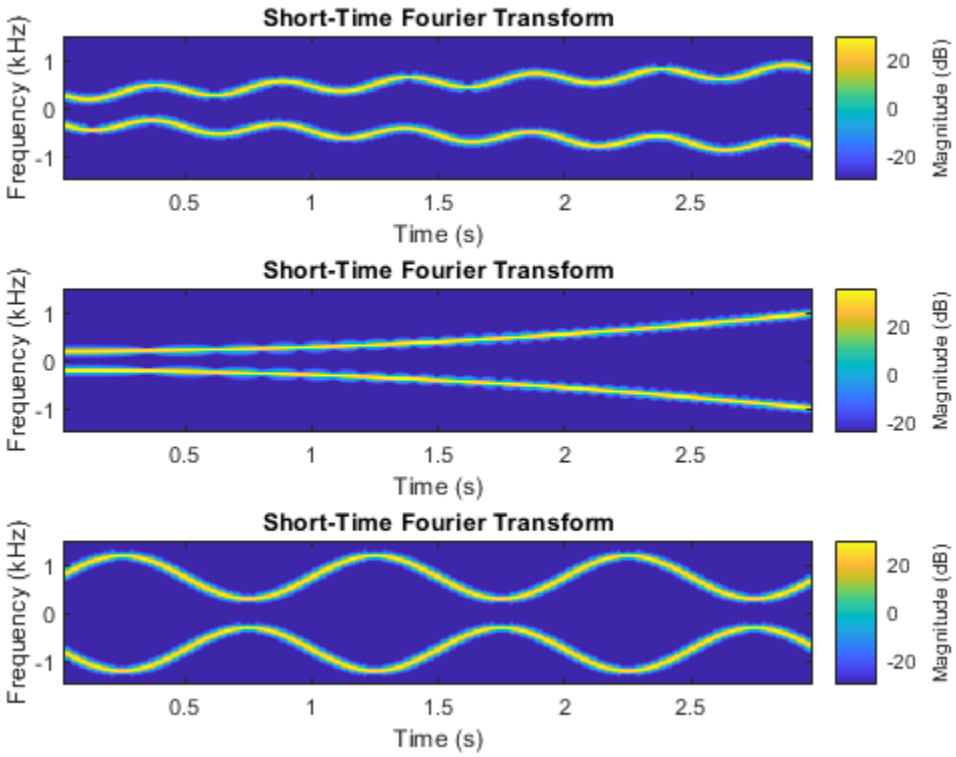
Signal Datastore with In-Memory Data

Create a signal datastore to iterate through the elements of an in-memory cell array of signal data. The data consists of a sinusoidally modulated linear chirp, a concave quadratic chirp, and a voltage controlled oscillator. The signals are sampled at 3000 Hz.

```
fs = 3000;
t = 0:1/fs:3-1/fs;
data = {chirp(t,300,t(end),800).*exp(2j*pi*10*cos(2*pi*2*t)); ...
        2*chirp(t,200,t(end),1000,'quadratic',[],'concave'); ...
        vco(sin(2*pi*t),[0.1 0.4]*fs,fs)};
sds = signalDatastore(data,'SampleRate',fs);
```

While the datastore has data, read each observation from the signal datastore and plot the short-time Fourier transform.

```
plotID = 1;
while hasdata(sds)
    [dataOut,info] = read(sds);
    subplot(3,1,plotID)
    stft(dataOut,info.SampleRate)
    plotID = plotID + 1;
end
```



Create Signal Datastore

Specify the path to the sample signals included with Signal Processing Toolbox™.

```
folder = fullfile(matlabroot,'examples','signal','data');
```

Create and display a signal datastore that points to the specified folder.

```
sds = signalDatastore(folder)

sds =
  signalDatastore with properties:

                      Files:{
                             'B:\matlab\examples\signal\data\BufferedHumanActivity.mat';
                             'B:\matlab\examples\signal\data\BufferedHumanactivity.mat';
                             'B:\matlab\examples\signal\data\EMGdata.mat'
                              ... and 48 more
                            }
                    Folders: {'B:\matlab\examples\signal\data'}
    AlternateFileSystemRoots: [0x0 string]
                   ReadSize: 1
```

Specify File Extensions to Include in Signal Datastore

Specify the file path to the signal samples included with Signal Processing Toolbox™.

```
folder = fullfile(matlabroot,'examples','signal','data');
```

Create a signal datastore that points to the `.csv` files in the specified folder.

```
sds = signalDatastore(folder,'FileExtensions','.csv')

sds =
  signalDatastore with properties:

                      Files:{
                             'B:\matlab\examples\signal\data\tremor.csv'
                            }
                    Folders: {'B:\matlab\examples\signal\data'}
    AlternateFileSystemRoots: [0x0 string]
                   ReadSize: 1
```

Read Multiple Files with Signal Datastore

Specify the path to four example files included with Signal Processing Toolbox™.

```
folder = fullfile(matlabroot,'examples','signal','data', ...
         ["INR.mat","relatedsig.mat","spots_num.mat","voice.mat"]);
```

Set the `ReadSize` property to `2` to read data from two files at a time. Each `read` returns a cell array where the first cell contains the first variable of the first file read, and the second cell contains the first variable from the second file. While the datastore has data, display the names of the variables read in each `read`.

```
sds = signalDatastore(folder,'ReadSize',2);
while hasdata(sds)
    [data,info] = read(sds);
```

```
    fprintf('Variable Name:\t%s\n',info.SignalVariableNames)
end
```

```
Variable Name:    Date
Variable Name:    s1
Variable Name:    year
Variable Name:    fs
```

Custom Read Data From Signal Datastore

Specify the path to three signals included with Signal Processing Toolbox™.

- The `strong.mat` file contains three variables: her, `him` and `fs`.
- The `slogan.mat` file contains three variables: `hotword`, `phrase` and `fs`.
- The `Ring.mat` file contains two variables: y and `Fs`.

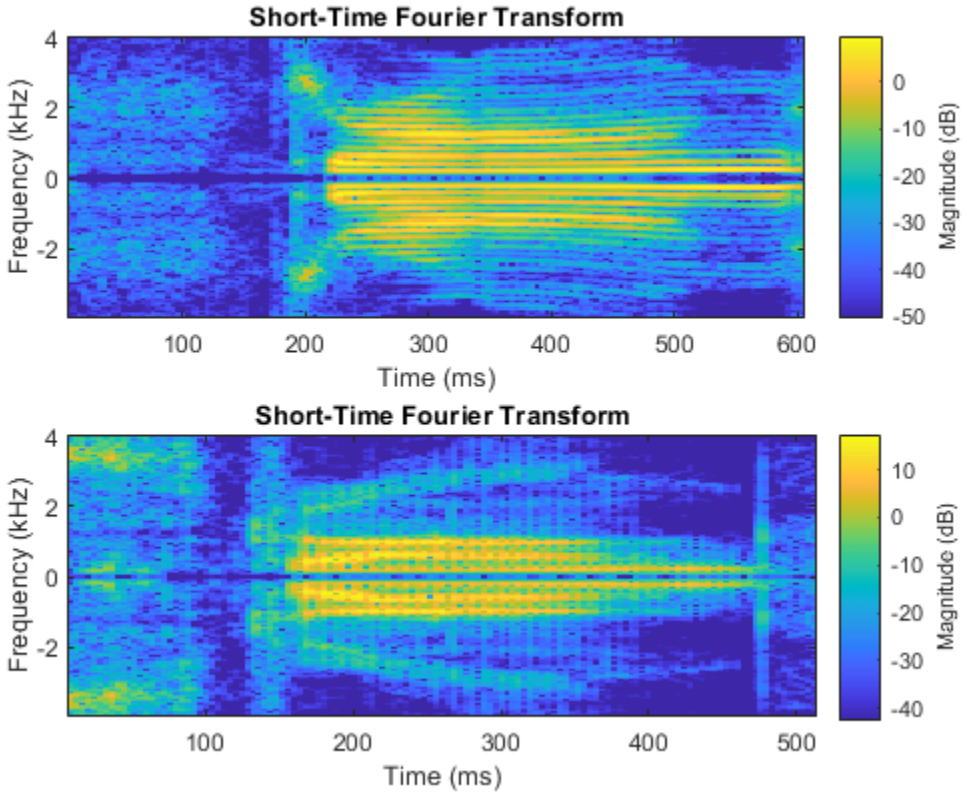
```
fld = ["strong.mat","slogan.mat","Ring.mat"];
folder = fullfile(matlabroot,'examples','signal','data',fld);
```

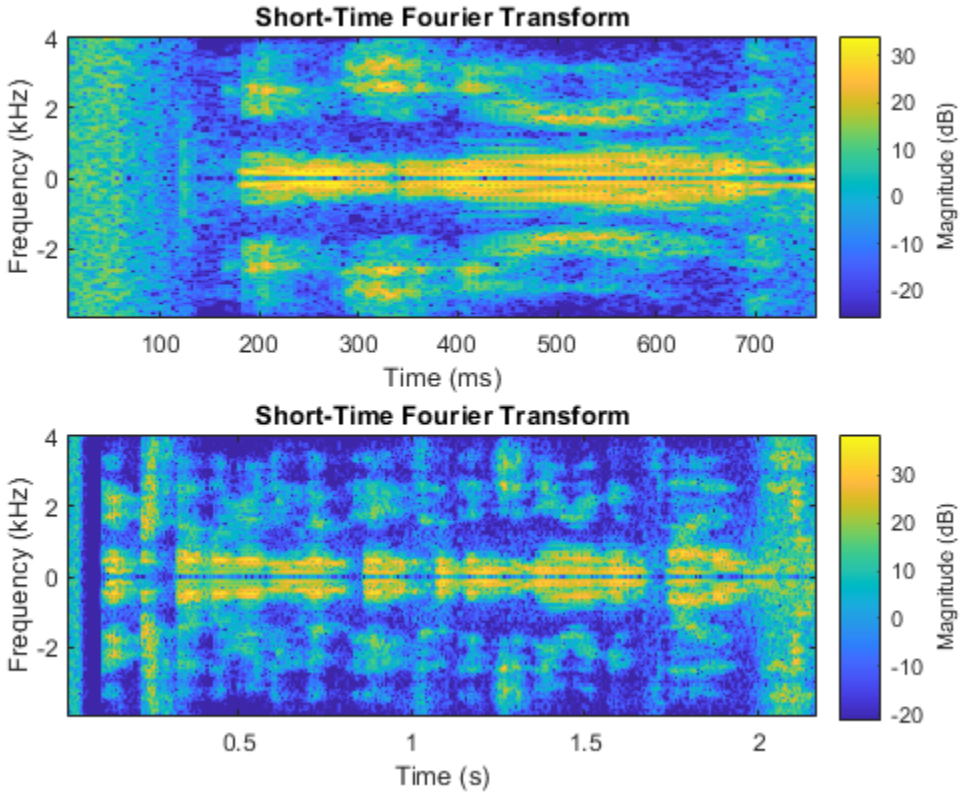
Create a signal datastore that points to the specified folder. Each file contains multiple variables of different names. The scalar in each file represents a sample rate. Define a custom read function that reads all the variables in the file as a structure and returns the variable in `dataOut` and information about the variables in `infoOut`. The `SampleRate` field of `infoOut` contains the scalar contained in each file, and `dataOut` contains the variables read from each file.

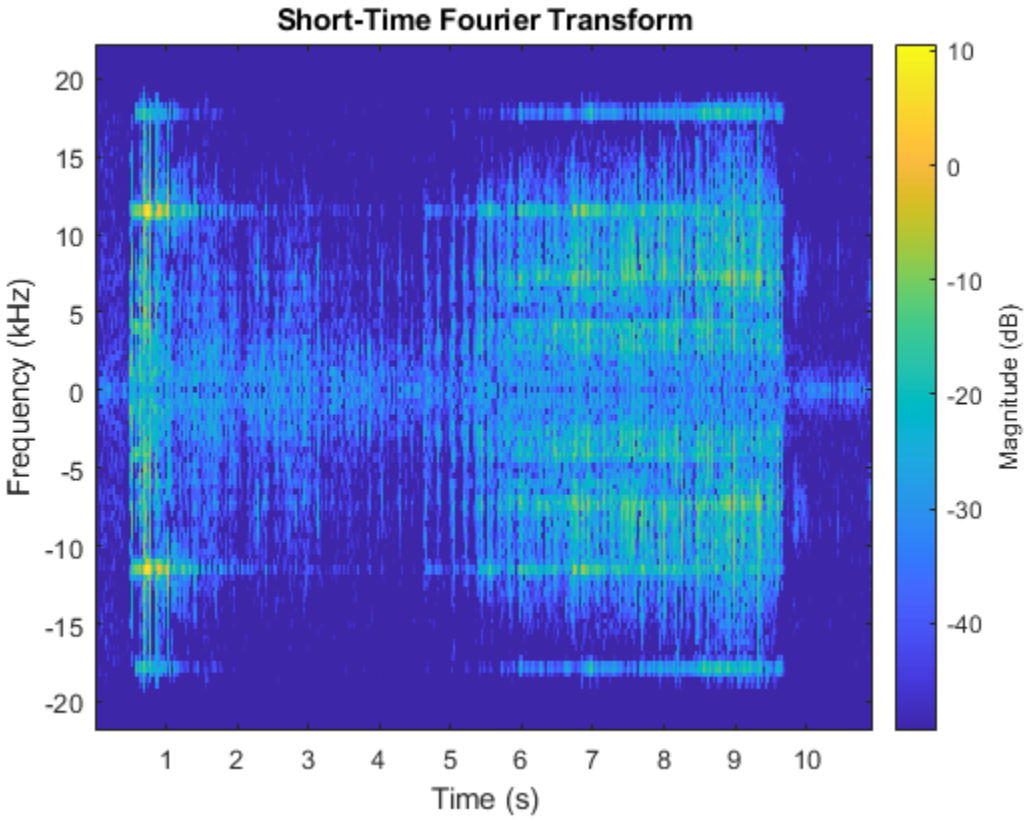
```
function [dataOut,infoOut] =   MyCustomRead(filename)
    fText = importdata(filename);
    value = struct2cell(fText);
    dataOut = {};
    for i = 1:length(value)
        if isscalar(value{i}) == 1
            infoOut.SampleRate = value{i};
        else
            dataOut{end+1} = value{i};
        end
    end
end

sds = signalDatastore(folder,'ReadFcn',@MyCustomRead);
```

While the datastore has unread files, read from the datastore and compute the short-time Fourier transforms of the signals.

```
while hasdata(sds)
    [data,infoOut] = read(sds);
    fs = infoOut.SampleRate;
    figure
    for i = 1:length(data)
        if length(data)>1
        subplot(2,1,i)
        end
        stft(data{i},fs)
    end
end
```


Short-Time Fourier Transform
Frequency (kHz)
4
2
0
-2
100
200
300
400
500
600
Time (ms)
0
-10
-20
-30
-40
-50
Magnitude (dB)
Short-Time Fourier Transform
Frequency (kHz)
4
2
0
-2
100
200
300
400
500
Time (ms)
10
0
-10
-20
-30
-40
Magnitude (dB)


Short-Time Fourier Transform
Frequency (kHz)
4
2
0
-2
100
200
300
400
500
600
700
Time (ms)
30
20
10
0
-10
-20
Magnitude (dB)
Short-Time Fourier Transform
Frequency (kHz)
4
2
0
-2
0.5
1
1.5
2
Time (s)
30
20
10
0
-10
-20
Magnitude (dB)



Read Multiple Variables from Files in Signal Datastore

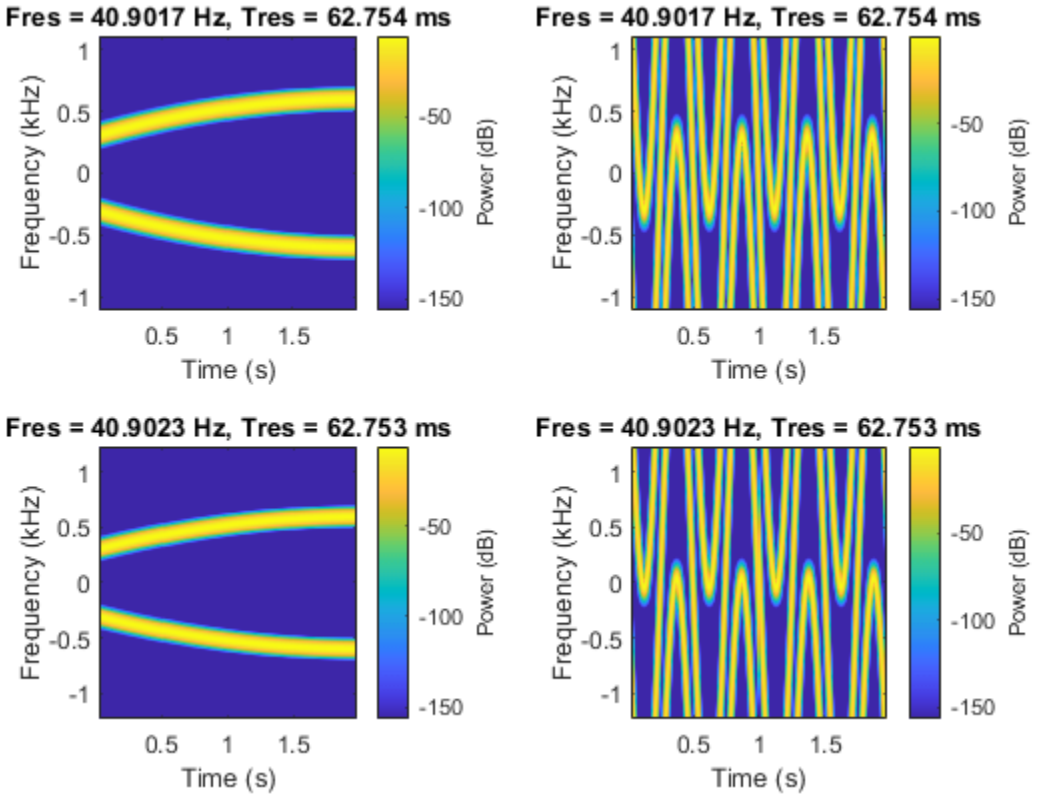
Specify the path to example files included with Signal Processing Toolbox™. Each file contains two signals and a random sample rate `fs` ranging from 3000 to 4000 Hz.

- The first signal, `x1`, is a convex quadratic chirp.
- The second signal, `x2`, is a chirp with sinusoidally varying frequency content.

```
folder = fullfile(matlabroot,'examples','signal','data','dataset');
```

Create a signal datastore that points to the specified folder and set the names of the signal variables and sample rate. While the datastore has data, read each observation and visualize the spectrogram of each signal.

```
sds = signalDatastore(folder,'SignalVariableNames',['x1';'x2'],'SampleRateVariableName','fs');

tiledlayout flow
while hasdata(sds)
    [data,info] = read(sds);
    nexttile
    pspectrum(data{1},info.SampleRate,'spectrogram','TwoSided',true)
    nexttile
    pspectrum(data{2},info.SampleRate,'spectrogram','TwoSided',true)
end
```



See Also

`datastore` | `mapreduce` | `tall`

Topics

“Waveform Segmentation Using Deep Learning”
“Preprocess Data for Domain-Specific Deep Learning Applications” (Deep Learning Toolbox)

Introduced in R2020a

combine

Combine data from multiple datastores

Syntax

```
sdsnew = combine(sds1,sds2,...,sdsn)
```

Description

`sdsnew = combine(sds1,sds2,...,sdsn)` combines two or more datastores by horizontally concatenating the data returned by the `read` function called on the input datastores.

Examples

Compute Envelopes of Signals

Specify the path to four signals included with MATLAB®. The signals are recordings of a bird chirp, a gong, a train, and a splat. All signals are sampled at 8192 Hz.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo', ...
         ["chirp.mat","gong.mat","train.mat","splat.mat"]);
```

Create a signal datastore that points to the specified files. Each file contains the variable `Fs` that denotes the sample rate.

```
sds1 = signalDatastore(folder,'SampleRateVariableName','Fs');
```

Define a function that takes the output of the `read` function and calculates the upper and lower envelopes of the signals using spline interpolation over local maxima separated by at least 80 samples. The function also returns the sample times for each signal.

```
function [dataOut,infoOut] = signalEnvelope(dataIn,info)
    [dataOut(:,1),dataOut(:,2)] = envelope(dataIn,80,'peak');
    infoOut = info;
    infoOut.TimeInstants = (0:length(dataOut)-1)/info.SampleRate;
end
```

Call the `transform` function to create a second datastore, `sds2`, that computes the envelopes of the signals using the function you defined.

```
sds2 = transform(sds1,@signalEnvelope,"IncludeInfo",true);
```

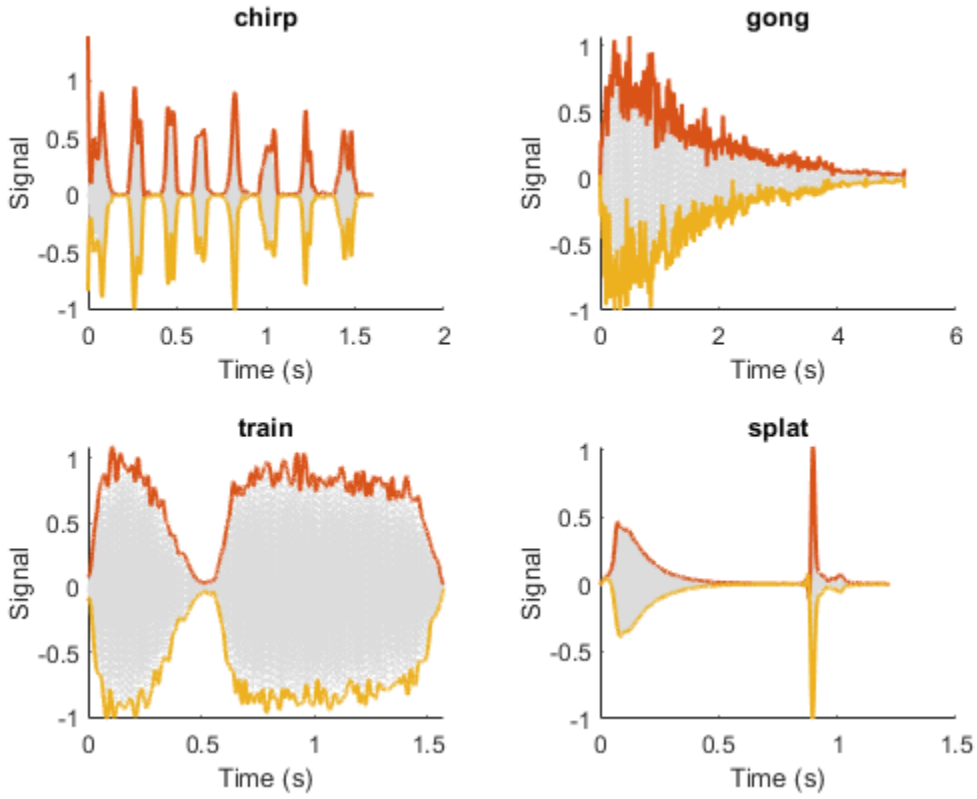
Combine `sds1` and `sds2` create a third datastore. Each call to the `read` function from the combined datastore returns a matrix with three columns:

- The first column corresponds to the original signal.
- The second and third columns correspond to the upper and lower envelopes, respectively.

```
sdsCombined = combine(sds1,sds2);
```

Read and display the original data and the upper and lower envelopes from the combined datastore. Use the `extractBetween` function to extract the file name from the file path.

```
tiledlayout('flow')
while hasdata(sdsCombined)
    [dataOut,infoOut] = read(sdsCombined);
    ts = infoOut{2}.TimeInstants;
    nexttile
    hold on
    plot(ts,dataOut(:,1),'Color','#DCDCDC','LineStyle',':')
    plot(ts,dataOut(:,2:3),'Linewidth',1.5)
    hold off
    xlabel('Time (s)')
    ylabel('Signal')
    title(extractBetween(infoOut{:,2}.FileName,'audiovideo\','.mat'))
end
```



Calculate Instantaneous Frequencies of Chirps

Specify the path to example files included with Signal Processing Toolbox™. Each file contains a chirp and a random sample rate ranging from 100 to 150 Hz.

```
folder = fullfile(matlabroot,'examples','signal','data','sample_chirps');
```

Create a signal datastore that points to the specified folder and set the names of the sample rate variables.

```
sds = signalDatastore(folder,'SampleRateVariableName','fs');
```

Define a function that takes the output of the `read` function and uses the `pspectrum` function to estimate the power spectrum of the signal. Use the estimate to compute the instantaneous frequency. The function also returns the vector of time instants corresponding to the centers of the windowed segments and the frequencies corresponding to the spectral estimates contained in the spectrograms of the signals.

```
function [dataOut,infoOut] = extractinstfreq(dataIn,info)
    [P,F,T] = pspectrum(dataIn,info.SampleRate,'spectrogram',...
              'TimeResolution',0.1,'OverlapPercent',40,'Leakage',0.8);
    dataOut = {instfreq(P,F,T)'};
    infoOut = info;
    infoOut.CenterFrequencies = F;
    infoOut.TimeInstants = T;
end
```

Call the `transform` function to create a new datastore that computes the instantaneous frequencies.

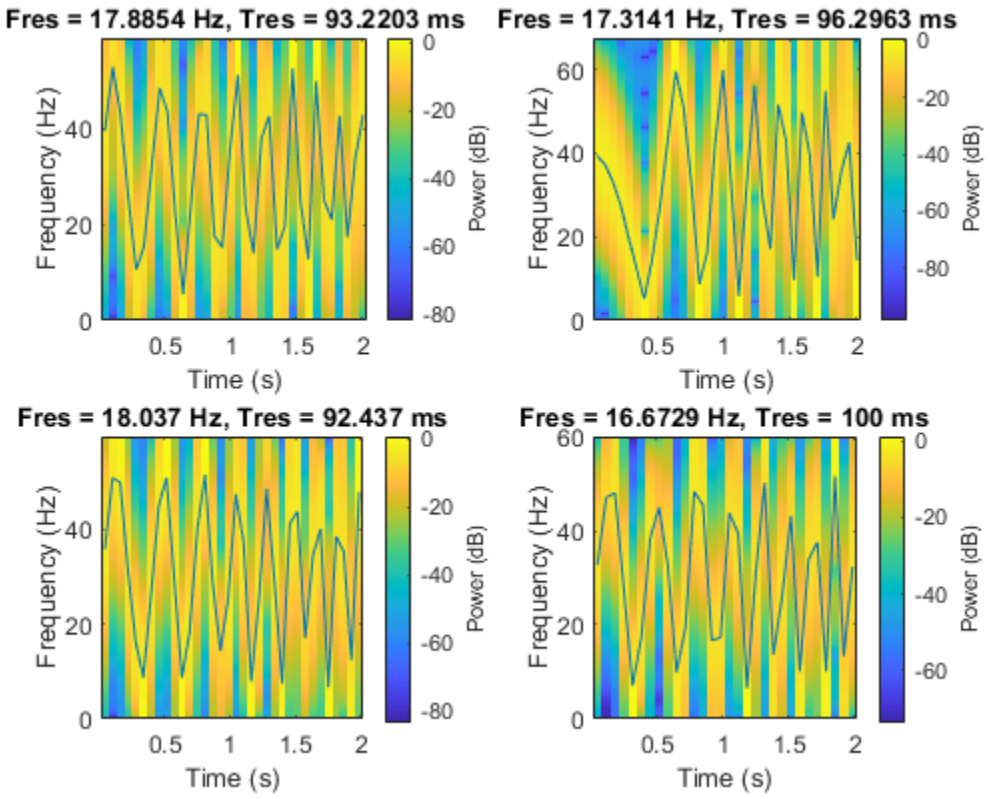
```
sds2 = transform(sds,@extractinstfreq,'IncludeInfo',true);
```

Because the data in `sds2` is not horizontally concatenable with the data in `sds`, transform the data in `sds` into cell arrays.

```
sds1 = transform(sds,@(x) {x});
```

Combine `sds1` and `sds2`. While the combined datastore has unread files, read from the new datastore and visualize the spectrograms. Overlay the instantaneous frequencies on the spectrograms.

```
sdsCombined = combine(sds1,sds2);
sdsSubset = subset(sdsCombined,[1,4,9,10]);
plotID = 1;
while hasdata(sdsSubset)
    subplot(2,2,plotID)
    [sig,info] = read(sdsSubset);
    pspectrum(sig{:,1},info{:,2}.SampleRate,'spectrogram', ...
              'TimeResolution',0.1,'OverlapPercent',40,'Leakage',0.8)
    hold on
    plot(info{:,2}.TimeInstants',sig{:,2})
    plotID = plotID + 1;
end
```



Input Arguments

`sds1,sds2,...,sdsn` — Signal datastores to combine

`signalDatastore` objects

Signal datastores to combine, specified as two or more comma-separated `signalDatastore` objects.

Output Arguments

`sdsnew` — New signal datastore with combined data

`CombinedDatastore` object

New datastore with the combined data, returned as a `CombinedDatastore` object.

Calling `read` on the combined datastore, horizontally concatenates the data by calling `read` on each input datastore.

See Also

`signalDatastore` | `hasdata`

Topics

“Waveform Segmentation Using Deep Learning”

Introduced in R2020a

hasdata

Return true if there is more data in datastore

Syntax

```
tf = hasdata(sds)
```

Description

`tf = hasdata(sds)` returns logical `1` (`true`) if there is data available to read from the datastore specified by `sds`. Otherwise, it returns logical `0` (`false`).

Examples

Read File Data in Signal Datastore

Specify the path to a set of audio signals included as MAT-files with MATLAB®.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo');
```

Create a signal datastore that points to the specified folder and set sample rate variable name to `Fs`. List the names of the MAT-files in the datastore.

```
sds = signalDatastore(folder,'FileExtension','.mat','SampleRateVariableName','Fs');
[~,c] = fileparts(sds.Files)
```

```
c = 7x1 cell
    {'chirp'   }
    {'gong'    }
    {'handel'  }
    {'laughter'}
    {'mtlb'    }
    {'splat'   }
    {'train'   }
```

While the signal datastore has unread files, read consecutive files from the datastore. Use the `progress` function to monitor the fraction of files read.

```
while hasdata(sds)
    [data,info] = read(sds);
    fprintf('Fraction of files read: %.2f\n',progress(sds))
end
```

```
Fraction of files read: 0.14
Fraction of files read: 0.29
Fraction of files read: 0.43
Fraction of files read: 0.57
Fraction of files read: 0.71
Fraction of files read: 0.86
Fraction of files read: 1.00
```

Print and inspect the `info` structure returned by the last call to the `read` function.

```
info
```

```
info = struct with fields:
             SampleRate: 8192
       TimeVariableName: "Fs"
    SignalVariableNames: "y"
               FileName: "B:\matlab\toolbox\matlab\audiovideo\train.mat"
```

Find Spectrogram of Chirps

Specify the path to four files included with Signal Processing Toolbox™. Each file contains a chirp and a random sample rate, `fs`, ranging from 100 to 150 Hz. Create a signal datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'examples','signal','data','sample_chirps', ...
    ["chirp_1.mat","chirp_4.mat","chirp_9.mat","chirp_10.mat"]);
sds = signalDatastore(folder,'SampleRateVariableName','fs');
```

Define a function that takes the output of the `read` function and computes and returns:

- The spectrograms of the chirps.
- The vector of time instants corresponding to the centers of the windowed segments.
- The frequencies corresponding to the estimates.

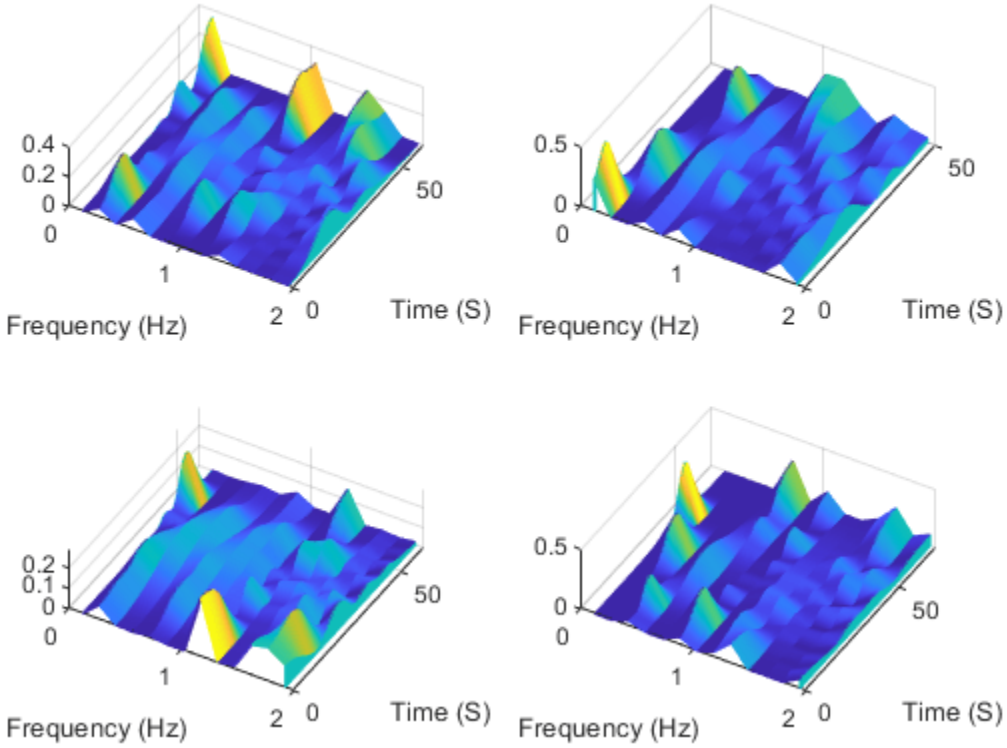
```
function [dataOut,infoOut] = extractSpectrogram(dataIn,info)
    [dataOut,F,T] = pspectrum(dataIn,info.SampleRate,'spectrogram',...
                              'TimeResolution',0.25,...
                              'OverlapPercent',40,'Leakage',0.8);
    infoOut = info;
    infoOut.CenterFrequencies = F;
    infoOut.TimeInstants = T;
end
```

Call the `transform` function to create a datastore that computes the spectrogram of each chirp using the function you defined.

```
sdsNew = transform(sds,@extractSpectrogram,'IncludeInfo',true);
```

While the transformed datastore has unread files, read from the new datastore and visualize the spectrograms in three-dimensional space.

```
t = tiledlayout('flow');
while hasdata(sdsNew)
    nexttile
    [sig,infoOut] = read(sdsNew);
    waterfall(infoOut.TimeInstants,infoOut.CenterFrequencies,sig)
    xlabel('Frequency (Hz)')
    ylabel('Time (S)')
    view([30 70])
end
```



Compute Envelopes of Signals

Specify the path to four signals included with MATLAB®. The signals are recordings of a bird chirp, a gong, a train, and a splat. All signals are sampled at 8192 Hz.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo', ...
         ["chirp.mat","gong.mat","train.mat","splat.mat"]);
```

Create a signal datastore that points to the specified files. Each file contains the variable `Fs` that denotes the sample rate.

```
sds1 = signalDatastore(folder,'SampleRateVariableName','Fs');
```

Define a function that takes the output of the `read` function and calculates the upper and lower envelopes of the signals using spline interpolation over local maxima separated by at least 80 samples. The function also returns the sample times for each signal.

```
function [dataOut,infoOut] = signalEnvelope(dataIn,info)
    [dataOut(:,1),dataOut(:,2)] = envelope(dataIn,80,'peak');
    infoOut = info;
    infoOut.TimeInstants = (0:length(dataOut)-1)/info.SampleRate;
end
```

Call the `transform` function to create a second datastore, `sds2`, that computes the envelopes of the signals using the function you defined.

```
sds2 = transform(sds1,@signalEnvelope,"IncludeInfo",true);
```

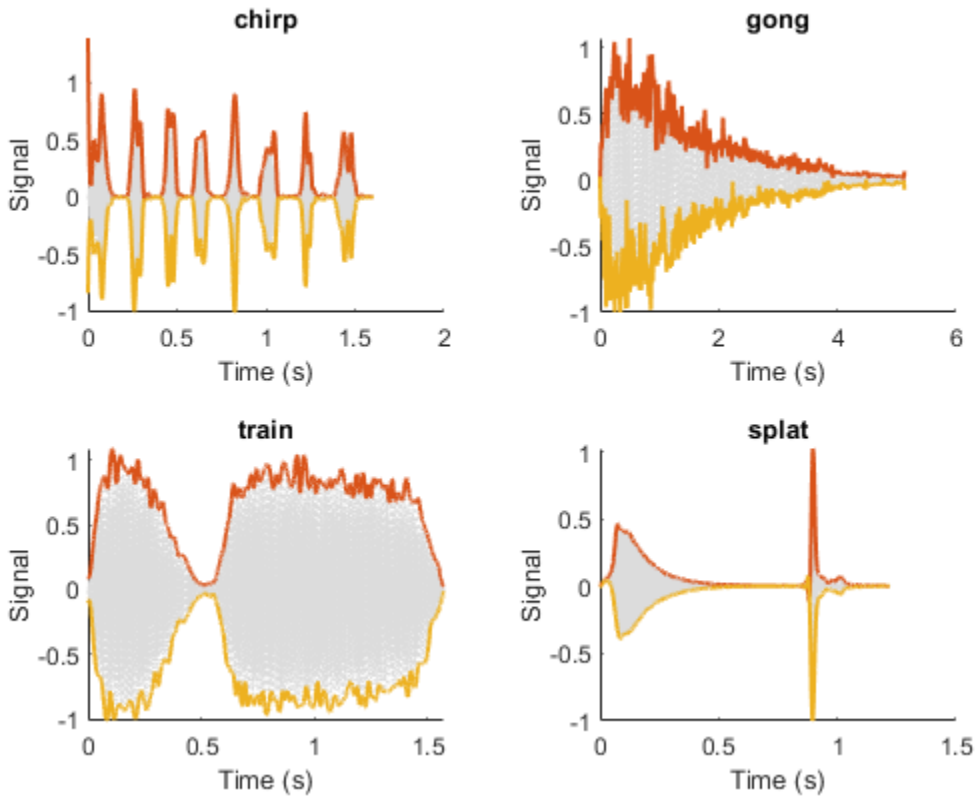
Combine `sds1` and `sds2` create a third datastore. Each call to the `read` function from the combined datastore returns a matrix with three columns:

- The first column corresponds to the original signal.
- The second and third columns correspond to the upper and lower envelopes, respectively.

```
sdsCombined = combine(sds1,sds2);
```

Read and display the original data and the upper and lower envelopes from the combined datastore. Use the `extractBetween` function to extract the file name from the file path.

```
tiledlayout('flow')
while hasdata(sdsCombined)
    [dataOut,infoOut] = read(sdsCombined);
    ts = infoOut{2}.TimeInstants;
    nexttile
    hold on
    plot(ts,dataOut(:,1),'Color','#DCDCDC','LineStyle',':')
    plot(ts,dataOut(:,2:3),'Linewidth',1.5)
    hold off
    xlabel('Time (s)')
    ylabel('Signal')
    title(extractBetween(infoOut{:,2}.FileName,'audiovideo\','.mat'))
end
```



Input Arguments

sds — Signal datastore

signalDatastore object

Specify sds as an signalDatastore object.

Output Arguments

tf — Indication if data is available to read

true | false

Indication if data is available to read from the datastore, returned as true or false.

Data Types: logical

See Also

signalDatastore | progress | subset | hasdata

Introduced in R2020a

read

Read next consecutive signal observation

Syntax

```
sig = read(sds)
[sig,info] = read(sds)
```

Description

`sig = read(sds)` returns signal data extracted from the datastore. Each subsequent call to `read` returns data from the next file in the datastore (if `sds` contains file data) or the next member (if `sds` contains in-memory data).

`[sig,info] = read(sds)` also returns information about the extracted signal data.

Examples

Read File Data in Signal Datastore

Specify the path to a set of audio signals included as MAT-files with MATLAB®.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo');
```

Create a signal datastore that points to the specified folder and set sample rate variable name to `Fs`. List the names of the MAT-files in the datastore.

```
sds = signalDatastore(folder,'FileExtension','.mat','SampleRateVariableName','Fs');
[~,c] = fileparts(sds.Files)
```

```
c = 7x1 cell
    {'chirp'   }
    {'gong'    }
    {'handel'  }
    {'laughter'}
    {'mtlb'    }
    {'splat'   }
    {'train'   }
```

While the signal datastore has unread files, read consecutive files from the datastore. Use the `progress` function to monitor the fraction of files read.

```
while hasdata(sds)
    [data,info] = read(sds);
    fprintf('Fraction of files read: %.2f\n',progress(sds))
end
```

```
Fraction of files read: 0.14
Fraction of files read: 0.29
Fraction of files read: 0.43
```

```
Fraction of files read: 0.57
Fraction of files read: 0.71
Fraction of files read: 0.86
Fraction of files read: 1.00
```

Print and inspect the `info` structure returned by the last call to the `read` function.

```
info
```

```
info = struct with fields:
             SampleRate: 8192
       TimeVariableName: "Fs"
    SignalVariableNames: "y"
               FileName: "B:\matlab\toolbox\matlab\audiovideo\train.mat"
```

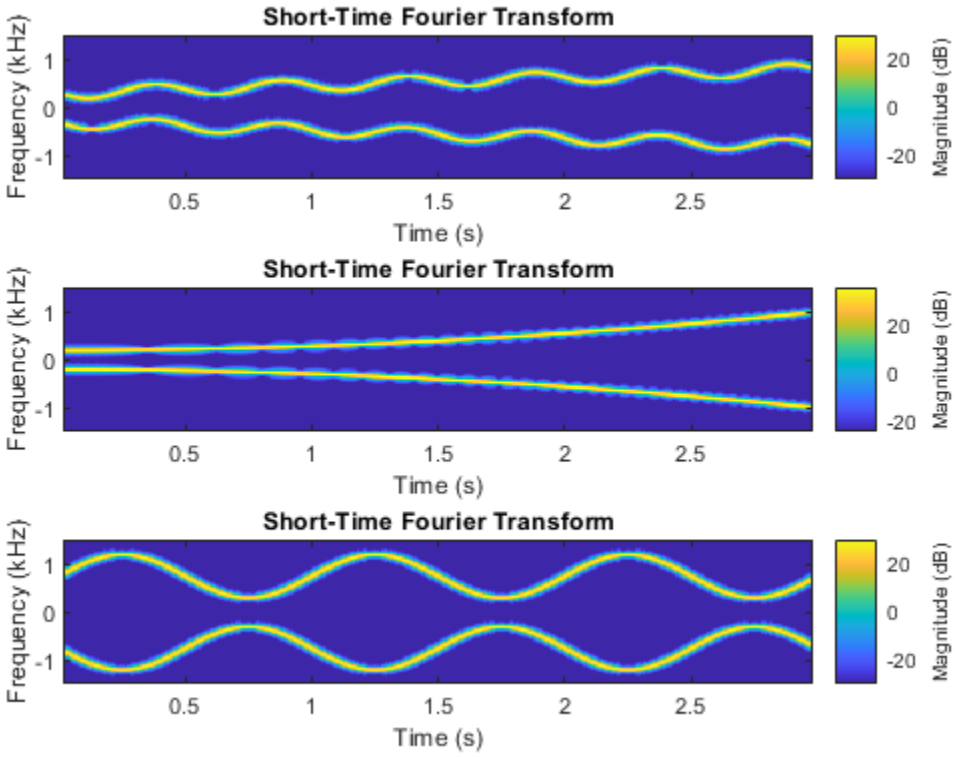
Signal Datastore with In-Memory Data

Create a signal datastore to iterate through the elements of an in-memory cell array of signal data. The data consists of a sinusoidally modulated linear chirp, a concave quadratic chirp, and a voltage controlled oscillator. The signals are sampled at 3000 Hz.

```
fs = 3000;
t = 0:1/fs:3-1/fs;
data = {chirp(t,300,t(end),800).*exp(2j*pi*10*cos(2*pi*2*t)); ...
        2*chirp(t,200,t(end),1000,'quadratic',[],'concave'); ...
        vco(sin(2*pi*t),[0.1 0.4]*fs,fs)};
sds = signalDatastore(data,'SampleRate',fs);
```

While the datastore has data, read each observation from the signal datastore and plot the short-time Fourier transform.

```
plotID = 1;
while hasdata(sds)
    [dataOut,info] = read(sds);
    subplot(3,1,plotID)
    stft(dataOut,info.SampleRate)
    plotID = plotID + 1;
end
```



Compute Envelopes of Signals

Specify the path to four signals included with MATLAB®. The signals are recordings of a bird chirp, a gong, a train, and a splat. All signals are sampled at 8192 Hz.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo', ...
         ["chirp.mat","gong.mat","train.mat","splat.mat"]);
```

Create a signal datastore that points to the specified files. Each file contains the variable `Fs` that denotes the sample rate.

```
sds1 = signalDatastore(folder,'SampleRateVariableName','Fs');
```

Define a function that takes the output of the `read` function and calculates the upper and lower envelopes of the signals using spline interpolation over local maxima separated by at least 80 samples. The function also returns the sample times for each signal.

```
function [dataOut,infoOut] = signalEnvelope(dataIn,info)
    [dataOut(:,1),dataOut(:,2)] = envelope(dataIn,80,'peak');
    infoOut = info;
    infoOut.TimeInstants = (0:length(dataOut)-1)/info.SampleRate;
end
```

Call the `transform` function to create a second datastore, `sds2`, that computes the envelopes of the signals using the function you defined.

```
sds2 = transform(sds1,@signalEnvelope,"IncludeInfo",true);
```

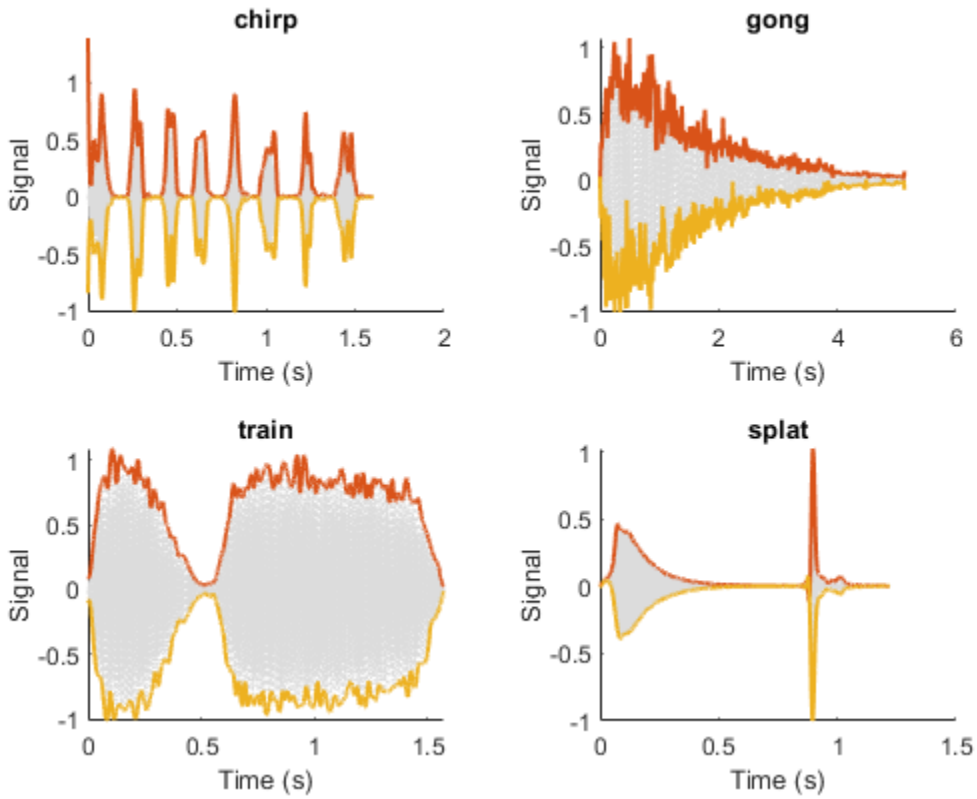
Combine `sds1` and `sds2` create a third datastore. Each call to the `read` function from the combined datastore returns a matrix with three columns:

- The first column corresponds to the original signal.
- The second and third columns correspond to the upper and lower envelopes, respectively.

```
sdsCombined = combine(sds1,sds2);
```

Read and display the original data and the upper and lower envelopes from the combined datastore. Use the `extractBetween` function to extract the file name from the file path.

```
tiledlayout('flow')
while hasdata(sdsCombined)
    [dataOut,infoOut] = read(sdsCombined);
    ts = infoOut{2}.TimeInstants;
    nexttile
    hold on
    plot(ts,dataOut(:,1),'Color','#DCDCDC','LineStyle',':')
    plot(ts,dataOut(:,2:3),'Linewidth',1.5)
    hold off
    xlabel('Time (s)')
    ylabel('Signal')
    title(extractBetween(infoOut{:,2}.FileName,'audiovideo\','.mat'))
end
```



Find Spectrogram of Chirps

Specify the path to four files included with Signal Processing Toolbox™. Each file contains a chirp and a random sample rate, `fs`, ranging from 100 to 150 Hz. Create a signal datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'examples','signal','data','sample_chirps', ...
    ["chirp_1.mat","chirp_4.mat","chirp_9.mat","chirp_10.mat"]);
sds = signalDatastore(folder,'SampleRateVariableName','fs');
```

Define a function that takes the output of the `read` function and computes and returns:

- The spectrograms of the chirps.
- The vector of time instants corresponding to the centers of the windowed segments.
- The frequencies corresponding to the estimates.

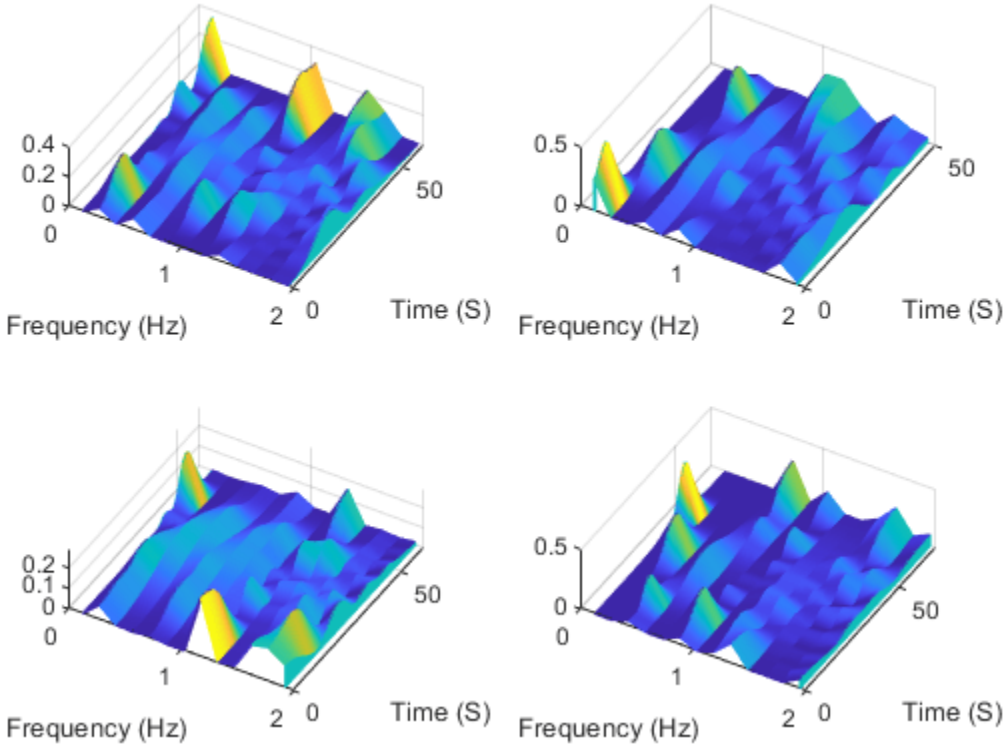
```
function [dataOut,infoOut] = extractSpectrogram(dataIn,info)
    [dataOut,F,T] = pspectrum(dataIn,info.SampleRate,'spectrogram',...
                              'TimeResolution',0.25,...
                              'OverlapPercent',40,'Leakage',0.8);
    infoOut = info;
    infoOut.CenterFrequencies = F;
    infoOut.TimeInstants = T;
end
```

Call the `transform` function to create a datastore that computes the spectrogram of each chirp using the function you defined.

```
sdsNew = transform(sds,@extractSpectrogram,'IncludeInfo',true);
```

While the transformed datastore has unread files, read from the new datastore and visualize the spectrograms in three-dimensional space.

```
t = tiledlayout('flow');
while hasdata(sdsNew)
    nexttile
    [sig,infoOut] = read(sdsNew);
    waterfall(infoOut.TimeInstants,infoOut.CenterFrequencies,sig)
    xlabel('Frequency (Hz)')
    ylabel('Time (S)')
    view([30 70])
end
```



Partition Signal Datastore into Default Number of Parts

Specify the file path to the example signals included with MATLAB®. Create a signal datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo');
sds = signalDatastore(folder,'SampleRateVariableName','Fs');
```

Get the default number of partitions for the signal datastore.

```
n = numpartitions(sds)
```

```
n = 7
```

Partition the datastore into the default number of partitions and return the datastore corresponding to the fourth partition.

```
subsds = partition(sds,n,4);
```

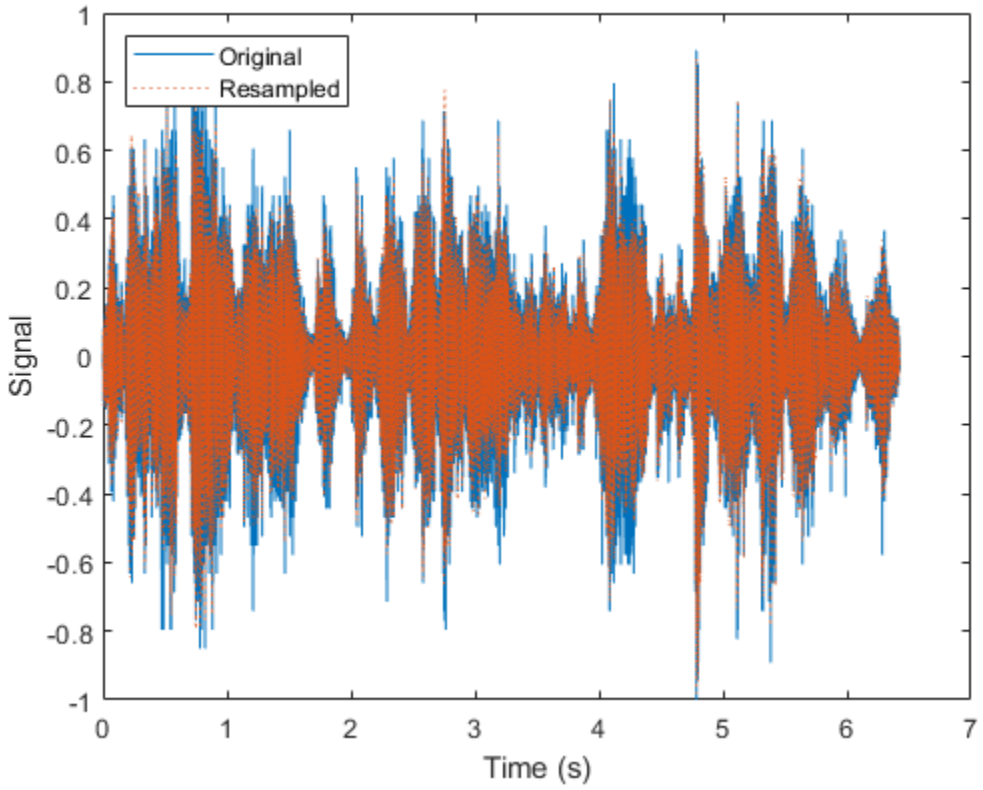
Use the `extractAfter` function to display the name of the file contained in the datastore corresponding to the fourth partition.

```
fName = extractAfter(subsds.Files,'audiovideo\')
```

```
fName = 1x1 cell array
    {'laughter.mat'}
```

Read the data and information about the signal in the datastore corresponding to the fourth partition. Extract the sample rate from `info` and resample the signal to half the original sample rate. Plot the original and resampled signals.

```
while hasdata(subsds)
    [data,info] = read(subsds);
    fs = info.SampleRate;
    f_res = 0.5*fs;
    ts = (0:length(data)-1)/fs;
    data_res = resample(data,1,2);
    t_res = (0:length(data_res)-1)/f_res;
    plot(ts,data,t_res,data_res,':')
    xlabel('Time (s)')
    ylabel('Signal')
    legend('Original','Resampled','Location','NorthWest')
end
```



Input Arguments

sds — Signal datastore

`signalDatastore` object

Signal datastore, specified as a `signalDatastore` object.

Output Arguments

`sig` — Signal data

array

Signal data, returned as an array. By default, calling `read` once returns the first variable of one file at a time. If you set the ReadSize property to *n*, such that $n > 1$, each time you call the `read` function, the function reads:

- The first variable of the first *n* files, if `sds` contains file data.
- The first *n* members, if `sds` contains in-memory data.

Note To determine the name of the first variable in a file, `read` follows these steps:

- For MAT-files:

```
s = load(fileName);
varNames = fieldnames(s);
firstVar = s.(varNames{1});
```

- For CSV files:

```
opts = detectImportOptions(fileName,'PreserveVariableNames',true);
varNames = opts.VariableNames;
firstVar = string(varNames{1});
```

If the `SignalVariableNames` property of the datastore contains more than one signal name, then `sig` is a cell array. Use the `ReadOutputOrientation` property of the datastore to control the orientation of `sig` as either a column array or a row array.

`info` — Information about signal data

structure

Information about signal data, returned as a structure.

- In case of file data, `info` contains the time information (if specified), file names, and the variable names used to read signal and time data, if this information was specified in the `signalDatastore`.
- If the datastore contains in-memory data, `info` contains time information (if specified) and member names.

See Also

`signalDatastore` | `readall` | `preview`

Topics

“Waveform Segmentation Using Deep Learning”

Introduced in R2020a

reset

Reset datastore read pointer to start of data

Syntax

```
reset(sds)
```

Description

`reset(sds)` resets the datastore read pointer to the start of the data. Resetting allows re-reading from the same datastore.

Examples

Reset Signal Datastore to Initial State

Specify the path to the sample signals included with Signal Processing Toolbox™.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo');
```

Create a signal datastore that points to the specified folder.

```
sds = signalDatastore(folder);
```

While the datastore has unread files, call `read` in a loop to read files sequentially.

```
while hasdata(sds)
    data = read(sds);
end
```

Reset the datastore to the state where no data has been read from it. Read the first file from the datastore.

```
reset(sds)
data = read(sds);
```

Input Arguments

sds — Signal datastore

`signalDatastore` object

Specify `sds` as an `signalDatastore` object.

See Also

`signalDatastore` | `read` | `readall`

Introduced in R2020a

readall

Read all signals from datastore

Syntax

```
data = readall(sds)
```

Description

`data = readall(sds)` reads all signal data from the datastore `sds`.

Examples

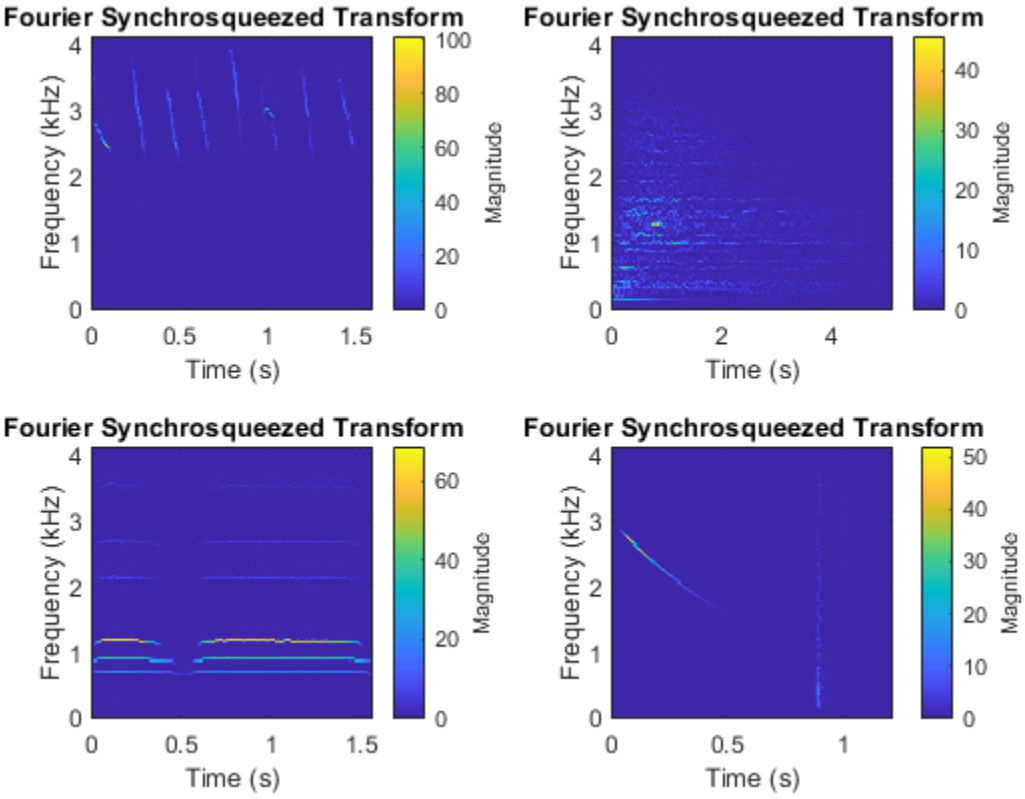
Read and Analyze All Signals in Signal Datastore

Specify the path to four signals included with MATLAB®. The signals are recordings of a bird chirp, a gong, a train, and a splat. All signals are sampled at 8192 Hz.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo', ...
         ["chirp.mat","gong.mat","train.mat","splat.mat"]);
fs = 8192;
sds = signalDatastore(folder,'SampleRate',fs);
```

Read the data in the first variables of all the files in the datastore and plot the Fourier synchrosqueezed transform of each signal.

```
data = readall(sds);

tiledlayout('flow')
for i = 1:length(data)
    nexttile
    fsst(data{i},fs,'yaxis')
end
```



Input Arguments

sds — Signal datastore

`signalDatastore` object

Signal datastore, specified as a `signalDatastore` object.

Output Arguments

data — All signals in signal datastore

cell array

All signals in the signal datastore, returned as a cell array. Each cell of `data` contains signals from a file or a member. Use the `ReadOutputOrientation` property of the datastore to control the orientation of `data` as either a column array or a row array.

Tips

When reading file data, because this function reads all the data in the files at once, you may run out of memory if your dataset is large.

See Also

`signalDatastore` | `read` | `preview`

Topics
“Waveform Segmentation Using Deep Learning”

Introduced in R2020a

numpartitions

Return estimate for reasonable number of partitions for parallel processing

Syntax

```
n = numpartitions(sds)
n = numpartitions(sds,pool)
```

Description

`n = numpartitions(sds)` returns the default number of partitions for the signal datastore `sds`.

`n = numpartitions(sds,pool)` returns a reasonable number of partitions to parallelize `sds` over the parallel pool.

- If `sds` contains file data, the number of partitions depends on the number of workers in the pool and the total number of files.
- If `sds` contains in-memory data, the number of partitions depends on the number of workers in the pool and the total number of members.

To parallelize datastore access, you must have Parallel Computing Toolbox installed.

Examples

Partition Signal Datastore into Default Number of Parts

Specify the file path to the example signals included with MATLAB®. Create a signal datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo');
sds = signalDatastore(folder,'SampleRateVariableName','Fs');
```

Get the default number of partitions for the signal datastore.

```
n = numpartitions(sds)
```

```
n = 7
```

Partition the datastore into the default number of partitions and return the datastore corresponding to the fourth partition.

```
subsds = partition(sds,n,4);
```

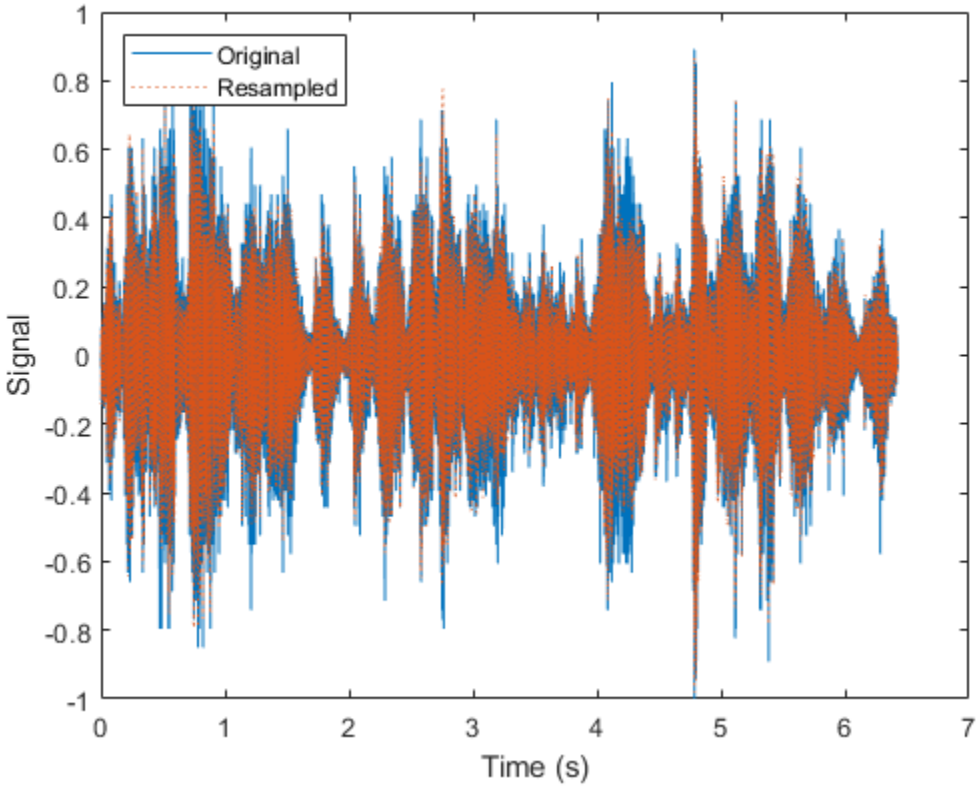
Use the `extractAfter` function to display the name of the file contained in the datastore corresponding to the fourth partition.

```
fName = extractAfter(subsds.Files,'audiovideo\')
```

```
fName = 1x1 cell array
    {'laughter.mat'}
```

Read the data and information about the signal in the datastore corresponding to the fourth partition. Extract the sample rate from `info` and resample the signal to half the original sample rate. Plot the original and resampled signals.

```
while hasdata(subsds)
    [data,info] = read(subsds);
    fs = info.SampleRate;
    f_res = 0.5*fs;
    ts = (0:length(data)-1)/fs;
    data_res = resample(data,1,2);
    t_res = (0:length(data_res)-1)/f_res;
    plot(ts,data,t_res,data_res,':')
    xlabel('Time (s)')
    ylabel('Signal')
    legend('Original','Resampled','Location','NorthWest')
end
```



Partition Signal Datastore for Parallel Access

Specify the path to a directory containing example signals included with MATLAB®.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo');
```

Create a signal datastore that points to the specified folder.

```
sds = signalDatastore(folder);
```

Return an estimate for a reasonable number of partitions for parallel processing, given the current parallel pool.

```
pool = gcp;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
n = numpartitions(sds,pool)
```

```
n = 7
```

Partition the signal datastore and read the signal data in each part.

```
parfor ii = 1:n
    subds = partition(sds,n,ii);
    while hasdata(subds)
        data = read(subds);
    end
end
```

Input Arguments

`sds` — Signal datastore

`signalDatastore` object

Signal datastore, specified as a `signalDatastore` object.

`pool` — Parallel pool

parallel pool object

Parallel pool, specified as a parallel pool object.

Output Arguments

`n` — Number of partitions

positive integer

Number of partitions over which datastore access is parallelized. By default, the number of partitions is $\min(N_{\text{observations}}, 3N_{\text{workers}})$, where:

- $N_{\text{observations}}$ is the number of files in the datastore (in case of file data) or number of members in the datastore (in case of in-memory data).
- N_{workers} is the number of workers in the pool.

See Also

`signalDatastore` | `partition`

Topics

“Waveform Segmentation Using Deep Learning”

Introduced in R2020a

preview

Read first signal observation from datastore for preview

Syntax

```
data = preview(sds)
```

Description

`data = preview(sds)` always reads the first file from the signal datastore in the case of file data, or the first member in the case of in-memory data. A call to `preview` does not affect the state of the signal datastore object.

Examples

Preview Data in Signal Datastore

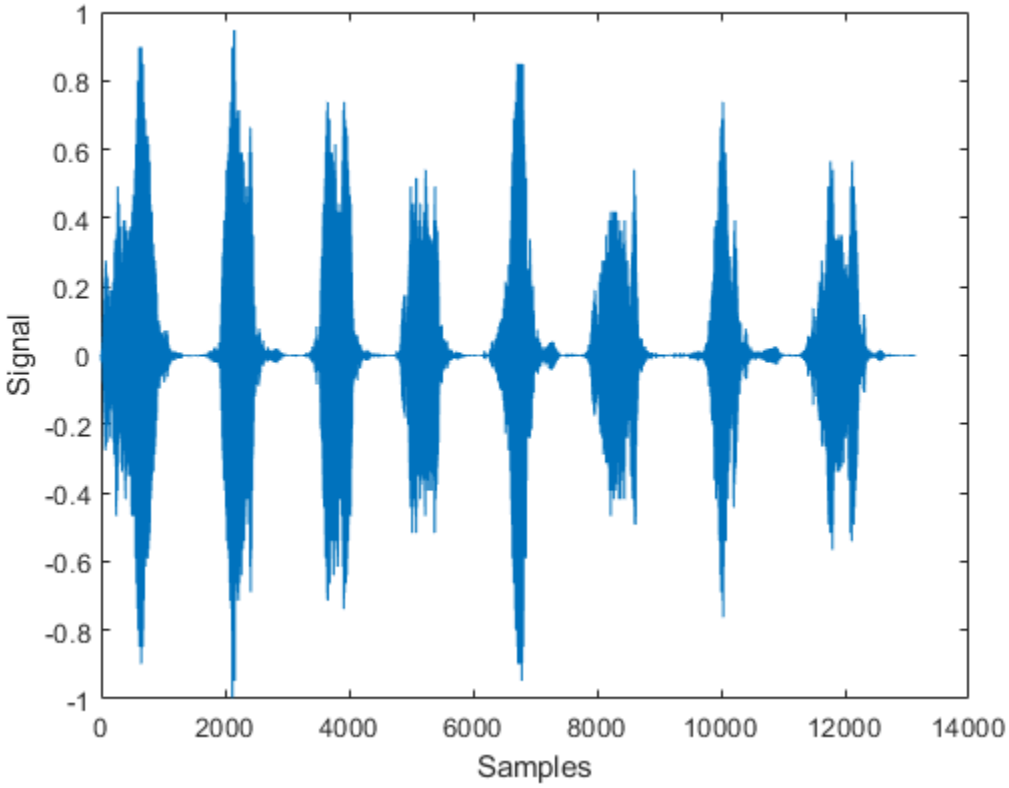
Specify the path to four signals included with MATLAB®. Create a signal datastore that points to the specified folder and display the name of the first file.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo');
sds = signalDatastore(folder);
[~,c] = fileparts(sds.Files(1))
```

```
c =
'chirp'
```

Plot the previewed data from the first file of `sds`.

```
data = preview(sds);
plot(data)
xlabel('Samples')
ylabel('Signal')
```



Input Arguments

`sds` — Signal datastore

`signalDatastore` object

Signal datastore, specified as a `signalDatastore` object.

Output Arguments

`data` — First observation

signal data

First observation from `sds`, returned as signal data.

See Also

`signalDatastore` | `read` | `readall`

Topics

"Waveform Segmentation Using Deep Learning"

Introduced in R2020a

partition

Partition signal datastore and return partitioned portion

Syntax

```
subsds = partition(sds,numPartitions,index)
subsds = partition(sds,'Observations',index)
subsds = partition(sds,'Observations',obsname)
```

Description

`subsds = partition(sds,numPartitions,index)` partitions the signal datastore into the number of parts specified by `numPartitions` and returns the partition corresponding to `index`.

`subsds = partition(sds,'Observations',index)` partitions the signal datastore and returns the partition corresponding to the `index` in the `Observations` property.

- If `sds` contains file data, the function partitions the signal datastore by files.
- If `sds` contains in-memory data, the function partitions the signal datastore by members.

`subsds = partition(sds,'Observations',obsname)` partitions the signal datastore and returns the partition corresponding to the observation name `obsname`.

- If `sds` contains file data, the function partitions the datastore by files.
- If `sds` contains in-memory data, the function partitions the datastore by members.

Examples

Partition Signal Datastore into Default Number of Parts

Specify the file path to the example signals included with MATLAB®. Create a signal datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo');
sds = signalDatastore(folder,'SampleRateVariableName','Fs');
```

Get the default number of partitions for the signal datastore.

```
n = numpartitions(sds)
```

```
n = 7
```

Partition the datastore into the default number of partitions and return the datastore corresponding to the fourth partition.

```
subsds = partition(sds,n,4);
```

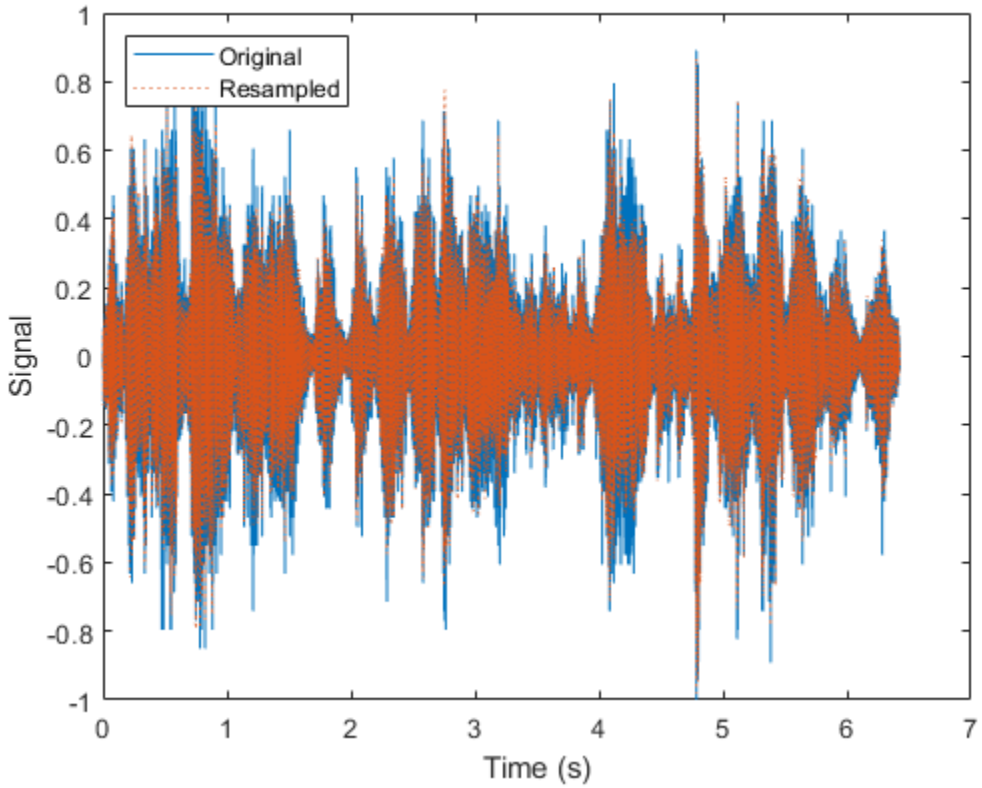
Use the `extractAfter` function to display the name of the file contained in the datastore corresponding to the fourth partition.

```
fName = extractAfter(subsds.Files,'audiovideo\')
```

```
fName = 1x1 cell array
    {'laughter.mat'}
```

Read the data and information about the signal in the datastore corresponding to the fourth partition. Extract the sample rate from `info` and resample the signal to half the original sample rate. Plot the original and resampled signals.

```
while hasdata(subsds)
    [data,info] = read(subsds);
    fs = info.SampleRate;
    f_res = 0.5*fs;
    ts = (0:length(data)-1)/fs;
    data_res = resample(data,1,2);
    t_res = (0:length(data_res)-1)/f_res;
    plot(ts,data,t_res,data_res,':')
    xlabel('Time (s)')
    ylabel('Signal')
    legend('Original','Resampled','Location','NorthWest')
end
```



Partition Signal Datastore for Parallel Access

Specify the path to a directory containing example signals included with MATLAB®.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo');
```

Create a signal datastore that points to the specified folder.

```
sds = signalDatastore(folder);
```

Return an estimate for a reasonable number of partitions for parallel processing, given the current parallel pool.

```
pool = gcp;
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
n = numpartitions(sds,pool)
```

```
n = 7
```

Partition the signal datastore and read the signal data in each part.

```
parfor ii = 1:n
    subds = partition(sds,n,ii);
    while hasdata(subds)
        data = read(subds);
    end
end
```

Input Arguments

`sds` — Signal datastore

`signalDatastore` object

Signal datastore, specified as a `signalDatastore` object.

`numPartitions` — Number of partitions

positive integer

Number of partitions, specified as a positive integer. Use the `numpartitions` function to estimate a reasonable value for `numPartitions`.

Data Types: `single` | `double`

`index` — Index of sub-datastore

positive integer

Index of sub-datastore, specified as a positive integer in the range `[1,numPartitions]`.

Data Types: `single` | `double`

`obsname` — Observation name

string scalar | character vector

Observation name, specified as a string scalar or a character vector.

The value of `obsname` is:

- A file name in the case of file data.
- A member name in the case of in-memory data.

Data Types: `char` | `string`

Output Arguments

subsds — Output signal datastore

`signalDatastore` object

Output signal datastore, returned as a `signalDatastore` object.

See Also

`signalDatastore` | `numpartitions`

Topics

"Waveform Segmentation Using Deep Learning"

Introduced in R2020a

progress

Fraction of files read

Syntax

```
fractionRead = progress(sds)
```

Description

`fractionRead = progress(sds)` returns the fraction of files read in the datastore as a normalized value in the range [0,1].

Examples

Read File Data in Signal Datastore

Specify the path to a set of audio signals included as MAT-files with MATLAB®.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo');
```

Create a signal datastore that points to the specified folder and set sample rate variable name to `Fs`. List the names of the MAT-files in the datastore.

```
sds = signalDatastore(folder,'FileExtension','.mat','SampleRateVariableName','Fs');
[~,c] = fileparts(sds.Files)
```

```
c = 7x1 cell
    {'chirp'   }
    {'gong'    }
    {'handel'  }
    {'laughter'}
    {'mtlb'    }
    {'splat'   }
    {'train'   }
```

While the signal datastore has unread files, read consecutive files from the datastore. Use the `progress` function to monitor the fraction of files read.

```
while hasdata(sds)
    [data,info] = read(sds);
    fprintf('Fraction of files read: %.2f\n',progress(sds))
end
```

```
Fraction of files read: 0.14
Fraction of files read: 0.29
Fraction of files read: 0.43
Fraction of files read: 0.57
Fraction of files read: 0.71
Fraction of files read: 0.86
Fraction of files read: 1.00
```

Print and inspect the `info` structure returned by the last call to the `read` function.

```
info
```

```
info = struct with fields:
             SampleRate: 8192
       TimeVariableName: "Fs"
    SignalVariableNames: "y"
               FileName: "B:\matlab\toolbox\matlab\audiovideo\train.mat"
```

Input Arguments

`sds` — Signal datastore

`signalDatastore` object

Specify `sds` as an `signalDatastore` object.

Output Arguments

`fractionRead` — Fraction of files read

normalized value in the range [0,1]

Fraction of files read, returned as a normalized value in the range [0,1].

Data Types: `double`

See Also

`signalDatastore` | `preview` | `readall` | `read` | `hasdata`

Introduced in R2020a

shuffle

Shuffle signals in signal datastore

Syntax

```
shuffledsds = shuffle(sds)
```

Description

`shuffledsds = shuffle(sds)` creates a “Deep Copy” on page 1-2090 of the input datastore `sds` and shuffles the signals using the `randperm` function.

Examples

Shuffle Files in Signal Datastore

Specify the path to the example signals included with MATLAB®. Create a signal datastore that points to the specified folder and display the names of the files in the datastore.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo');
sds = signalDatastore(folder);
[~,c1] = fileparts(sds.Files)
```

```
c1 = 7x1 cell
    {'chirp'   }
    {'gong'    }
    {'handel'  }
    {'laughter'}
    {'mtlb'    }
    {'splat'   }
    {'train'   }
```

Shuffle the files to create a new datastore containing the same files in random order. Display the names of the files in the shuffled datastore.

```
sdsshuffled = shuffle(sds);
[~,c2] = fileparts(sdsshuffled.Files)
```

```
c2 = 7x1 cell
    {'splat'   }
    {'handel'  }
    {'train'   }
    {'mtlb'    }
    {'chirp'   }
    {'gong'    }
    {'laughter'}
```

Input Arguments

sds — Signal datastore

`signalDatastore` object

Signal datastore, specified as a `signalDatastore` object.

Output Arguments

shuffledsds — Shuffled signal datastore

`signalDatastore` object

Shuffled signal datastore, returned as a `signalDatastore` object containing randomly ordered files or members from `sds`.

More About

Deep Copy

A deep copy refers to a copy in which all levels of data are copied. For example, a deep copy of a structure copies each field and the contents of each field, if any.

See Also

`signalDatastore` | `subset`

Topics

“Waveform Segmentation Using Deep Learning”

Introduced in R2020a

subset

Create datastore with subset of signals

Syntax

```
sdssubset = subset(sds,indices)
```

Description

`sdssubset = subset(sds,indices)` returns a signal datastore `sdssubset` containing a subset of the signals in `sds`.

Examples

Create Signal Datastore with Subset Based on File Name

Specify the file path to the example signals included with Signal Processing Toolbox™.

```
folder = fullfile(matlabroot,'toolbox','matlab','demos');
```

Create a signal datastore that points to the specified folder. List the names of the first ten files in the datastore.

```
sds = signalDatastore(folder);
[~,c] = fileparts(sds.Files(1:10))
```

```
c = 10x1 cell
    {'accidents'     }
    {'airfoil'       }
    {'airlineResults'}
    {'cape'          }
    {'census'        }
    {'clown'         }
    {'detail'        }
    {'dmbanner'      }
    {'durer'         }
    {'earth'         }
```

Create a logical vector indicating whether the file names in the signal datastore start with `'air'`.

```
fileContainsAir = cellfun(@(c)startsWith(c,'air'),c);
```

Call the `subset` function on the signal datastore and the indices corresponding to the files starting with `'air'`.

```
sdssubset = subset(sds,fileContainsAir)
```

```
sdssubset =
  signalDatastore with properties:
```

```
                    Files:{
                          'B:\matlab\toolbox\matlab\demos\airfoil.mat';
                          'B:\matlab\toolbox\matlab\demos\airlineResults.mat'
                          }
                  Folders: {'B:\matlab\toolbox\matlab\demos'}
AlternateFileSystemRoots: [0x0 string]
                 ReadSize: 1
```

Input Arguments

sds — Signal datastore

`signalDatastore` object

Signal datastore, specified as a `signalDatastore` object.

indices — Indices of files in subset

vector of indices | logical vector

Specify `indices` as:

- A vector containing the indices of files or members to be included in `sdssubset`. The `subset` function accepts nonunique indices.
- A logical vector the same length as the number of files or members in `sds`. If indices are specified as a logical vector, `true` indicates that the corresponding files or members are to be included in `sdssubset`.

Data Types: `double` | `logical`

Output Arguments

sdssubset — Subset of signal datastore

`signalDatastore` object

Subset of signal datastore, returned as a `signalDatastore` object.

See Also

`signalDatastore` | `shuffle` | `numpartitions` | `partition`

Topics

“Waveform Segmentation Using Deep Learning”

Introduced in R2020a

transform

Transform signal datastore

Syntax

```
transformDatastore = transform(sds,@fcn)
transformDatastore = transform(sds,@fcn,'IncludeInfo',infoIn)
```

Description

`transformDatastore = transform(sds,@fcn)` creates a new datastore that transforms output from the `read` function.

`transformDatastore = transform(sds,@fcn,'IncludeInfo',infoIn)` includes the info returned by the read of `sds`.

Examples

Find Spectrogram of Chirps

Specify the path to four files included with Signal Processing Toolbox™. Each file contains a chirp and a random sample rate, `fs`, ranging from 100 to 150 Hz. Create a signal datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'examples','signal','data','sample_chirps', ...
    ["chirp_1.mat","chirp_4.mat","chirp_9.mat","chirp_10.mat"]);
sds = signalDatastore(folder,'SampleRateVariableName','fs');
```

Define a function that takes the output of the `read` function and computes and returns:

- The spectrograms of the chirps.
- The vector of time instants corresponding to the centers of the windowed segments.
- The frequencies corresponding to the estimates.

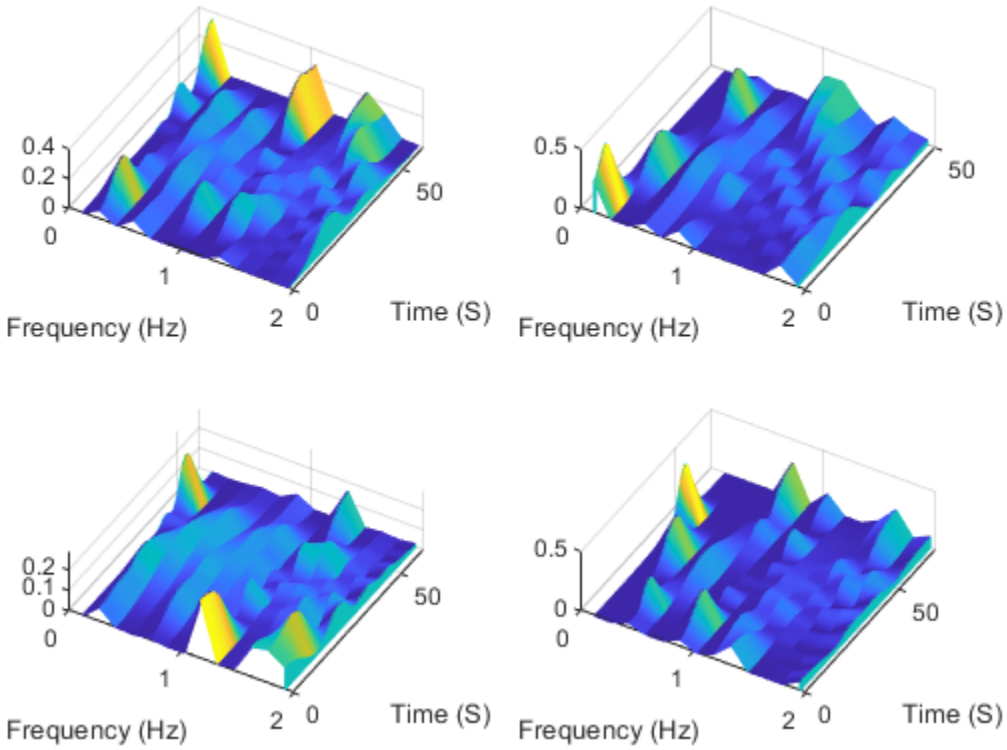
```
function [dataOut,infoOut] = extractSpectrogram(dataIn,info)
    [dataOut,F,T] = pspectrum(dataIn,info.SampleRate,'spectrogram',...
                              'TimeResolution',0.25,...
                              'OverlapPercent',40,'Leakage',0.8);
    infoOut = info;
    infoOut.CenterFrequencies = F;
    infoOut.TimeInstants = T;
end
```

Call the `transform` function to create a datastore that computes the spectrogram of each chirp using the function you defined.

```
sdsNew = transform(sds,@extractSpectrogram,'IncludeInfo',true);
```

While the transformed datastore has unread files, read from the new datastore and visualize the spectrograms in three-dimensional space.

```
t = tiledlayout('flow');
while hasdata(sdsNew)
    nexttile
    [sig,infoOut] = read(sdsNew);
    waterfall(infoOut.TimeInstants,infoOut.CenterFrequencies,sig)
    xlabel('Frequency (Hz)')
    ylabel('Time (S)')
    view([30 70])
end
```



Compute Envelopes of Signals

Specify the path to four signals included with MATLAB®. The signals are recordings of a bird chirp, a gong, a train, and a splat. All signals are sampled at 8192 Hz.

```
folder = fullfile(matlabroot,'toolbox','matlab','audiovideo', ...
         ["chirp.mat","gong.mat","train.mat","splat.mat"]);
```

Create a signal datastore that points to the specified files. Each file contains the variable `Fs` that denotes the sample rate.

```
sds1 = signalDatastore(folder,'SampleRateVariableName','Fs');
```

Define a function that takes the output of the `read` function and calculates the upper and lower envelopes of the signals using spline interpolation over local maxima separated by at least 80 samples. The function also returns the sample times for each signal.

```
function [dataOut,infoOut] = signalEnvelope(dataIn,info)
    [dataOut(:,1),dataOut(:,2)] = envelope(dataIn,80,'peak');
    infoOut = info;
    infoOut.TimeInstants = (0:length(dataOut)-1)/info.SampleRate;
end
```

Call the `transform` function to create a second datastore, `sds2`, that computes the envelopes of the signals using the function you defined.

```
sds2 = transform(sds1,@signalEnvelope,"IncludeInfo",true);
```

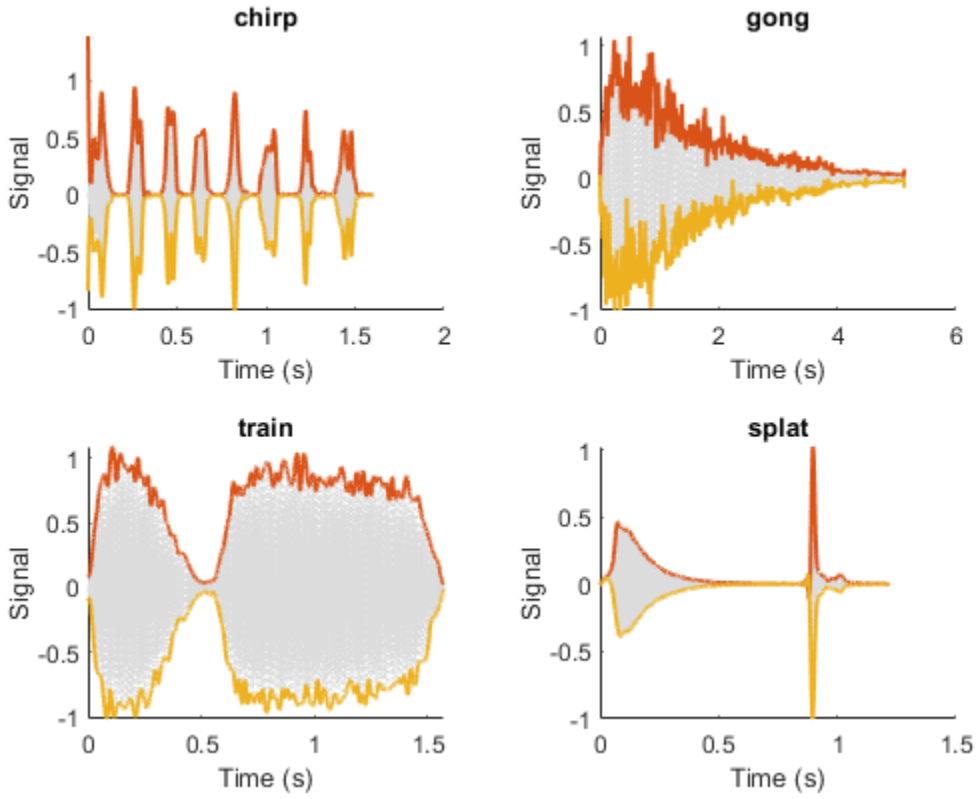
Combine `sds1` and `sds2` create a third datastore. Each call to the `read` function from the combined datastore returns a matrix with three columns:

- The first column corresponds to the original signal.
- The second and third columns correspond to the upper and lower envelopes, respectively.

```
sdsCombined = combine(sds1,sds2);
```

Read and display the original data and the upper and lower envelopes from the combined datastore. Use the `extractBetween` function to extract the file name from the file path.

```
tiledlayout('flow')
while hasdata(sdsCombined)
    [dataOut,infoOut] = read(sdsCombined);
    ts = infoOut{2}.TimeInstants;
    nexttile
    hold on
    plot(ts,dataOut(:,1),'Color','#DCDCDC','LineStyle',':')
    plot(ts,dataOut(:,2:3),'Linewidth',1.5)
    hold off
    xlabel('Time (s)')
    ylabel('Signal')
    title(extractBetween(infoOut{:,2}.FileName,'audiovideo\','.mat'))
end
```



Input Arguments

`sds` — Signal datastore

`signalDatastore` object

Specify `sds` as a `signalDatastore` object.

`@fcn` — Function that transforms data

function handle

Function that transforms data, specified as a function handle. The signature of the function depends on the `IncludeInfo` parameter.

- If `IncludeInfo` is set to `false` (default), the function transforms the signal output from `read`. The info output from `read` is unaltered.

  The transform function must have this signature:

  ```
  function dataOut = fcn(dataIn)
  ...
  end
  ```

- If `IncludeInfo` is set to `true`, the function transforms the signal output from `read`, and can use or modify the information returned from `read`.

  The transform function must have this signature:

```
function [dataOut,infoOut] = fcn(signal,infoIn)
...
end
```

`infoIn` — Pass info through customized `read` function
`false` (default) | true

Pass info through the customized `read` function, specified as `true` or `false`. If `true`, the transform function can use or modify the information it gets from `read`. If unspecified, `IncludeInfo` defaults to `false`.

Data Types: `logical`

Output Arguments

`transformDatastore` — New datastore with customized `read`
`TransformedDatastore`

New datastore with customized `read`, returned as a `TransformedDatastore` with UnderlyingDatastore set to `sds`, TransformSet set to `fcn`, and IncludeInfo set to `true` or `false`.

See Also

`signalDatastore` | `subset` | `hasdata` | `progress`

Introduced in R2020a

writeall

Write datastore to files

Syntax

```
writeall(sds,outputLocation)
writeall(sds,outputLocation,Name,Value)
```

Description

`writeall(sds,outputLocation)` writes the data from the input datastore `sds` to output files at the location specified in `outputLocation`. The number of output files is the same as the number of files referenced by the datastore.

`writeall(sds,outputLocation,Name,Value)` writes data with additional options specified by one or more name-value arguments. For example, `'FilenameSuffix','norm'` adds the descriptive text `'norm'` at the end of all output files.

Examples

Write In-Memory Signal Datastore to Files

Create a signal datastore to iterate through the elements of an in-memory cell array of signal data. The array contains:

- A sinusoidally modulated linear chirp
- A concave quadratic chirp
- A voltage controlled oscillator
- A set of pulses of decreasing duration separated by regions of oscillating amplitude and fluctuating frequency with an increasing trend

The signals are sampled at 3000 Hz.

```
fs = 3000;
t = 0:1/fs:3-1/fs;
data = {chirp(t,300,t(end),800).*exp(2j*pi*10*cos(2*pi*2*t)); ...
        2*chirp(t,200,t(end),1000,'quadratic',[],'concave'); ...
        vco(sin(2*pi*t),[0.1 0.4]*fs,fs);
        besselj(0,600*(sin(2*pi*(t+1).^3/30).^5));};
sds = signalDatastore(data,'SampleRate',fs);
```

Create a folder called `Files` in the current folder. Write the contents of the datastore to files. List the contents of the folder. The `writeall` function uses the `MemberNames` property of `signalDatastore` to name the files and the signals in the files.

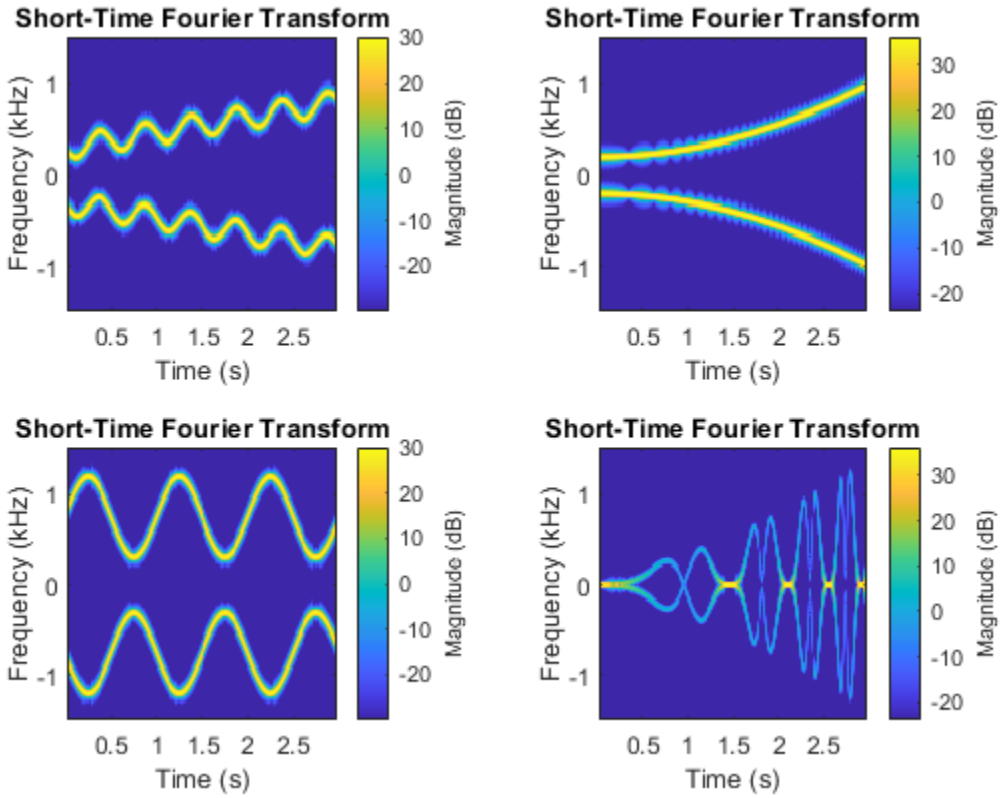
```
fname = 'Files';
mkdir(fname)

writeall(sds,fname)
```

```
dir(fname)
```

```
.            Member1.mat  Member3.mat
..           Member2.mat  Member4.mat
```

Create a datastore that points to the files in `Files`. Read the data one file at a time. Compute and display the short-time Fourier transform of each signal.

```
sdfs = signalDatastore(fname,'SampleRate',fs);
tiledlayout flow
while hasdata(sdfs)
    nexttile
    [sg,nf] = read(sdfs);
    stft(sg,nf.SampleRate)
end
```



Remove the `Files` directory you created earlier in the example.

```
rmdir(fname,'s')
```

Write Datastore Spectrograms to Files

Specify the path to four signals included with MATLAB®. The signals are recordings of a bird chirping, a train, a splat, and a female voice saying the word "MATLAB." The first three signals are

sampled at 8192 Hz and the fourth at 7418 Hz. Create a signal datastore that points to the specified files.

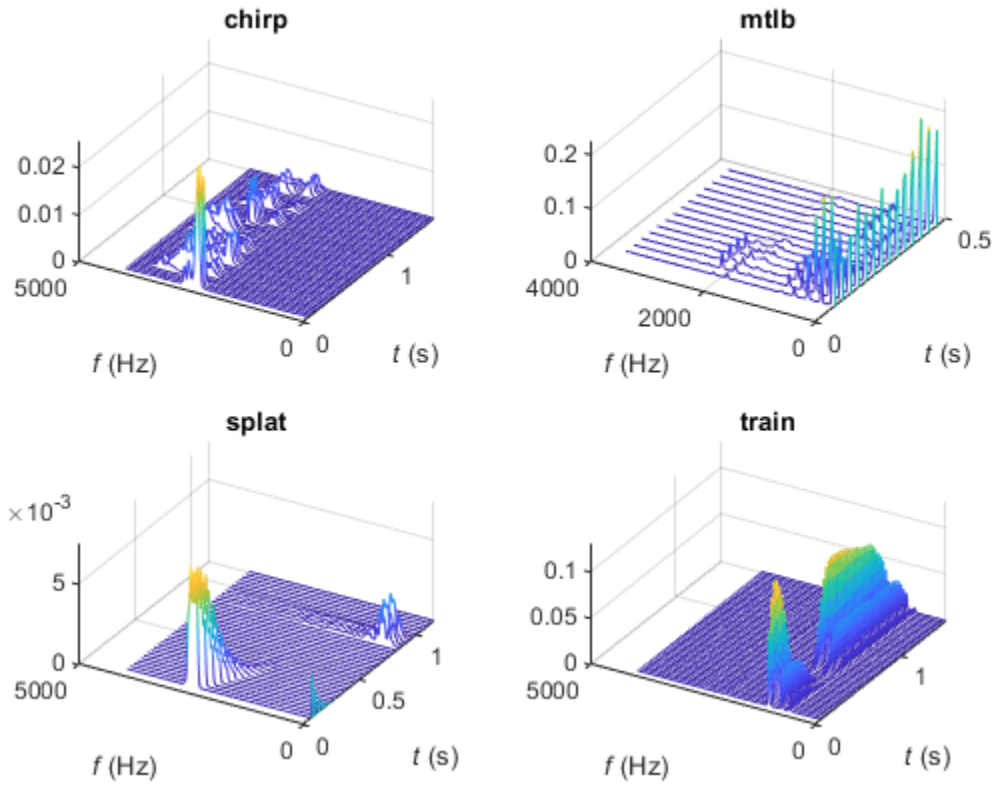
```
fls = ["chirp" "train" "splat" "mtlb"];
folder = fullfile(matlabroot,"toolbox","matlab","audiovideo",append(fls,".mat"));

sds = signalDatastore(folder,'SampleRateVariableName','Fs');
```

Write the spectrograms of the signals to text files in the current folder using the `writeall` and `writeSpectrogram` on page 1-0 functions. `writeall` uses the `MemberNames` property of `signalDatastore` to name the files and the signals in the files. Create a datastore that points to the files in the current folder.

```
writeall(sds,'.','WriteFcn',@writeSpectrogram)

sdfs = signalDatastore('.');
```

Read the data one file at a time. Display the spectrogram of each signal.

```
tiledlayout flow
while hasdata(sdfs)
    nexttile

    [d,info] = read(sdfs);
    waterfall(d(2:end,1),d(1,2:end),d(2:end,2:end)')

    wtf = gca;
    wtf.XDir = 'reverse';
    view(30,45)
    xlabel("{\it f} (Hz)")
    ylabel("{\it t} (s)")

    [~,k] = fileparts(info.FileName);
    title(k)
end
```



The `writeSpectrogram` function computes the spectrogram of the input signal using `pspectrum` and writes it to a MAT-file in the current folder. The function specifies 80% of overlap between adjoining segments, a time resolution of 0.15 second, and a spectral leakage of 0.8.

```
function writeSpectrogram(data,info,~)

    [s,f,t] = pspectrum(data,info.ReadInfo.SampleRate,'spectrogram', ...
        'TimeResolution',0.15,'OverlapPercent',80,'Leakage',0.8);
    d = [NaN t'; f s];

    [~,q] = fileparts(info.SuggestedOutputName);
    save(q,"d")

end
```

Input Arguments

sds — Signal datastore

`signalDatastore` object

Signal datastore, specified as a `signalDatastore` object. By default, when `sds` contains in-memory data, the `writeall` function writes the input data to MAT-files.

Example: `signalDatastore({randn(100,1)},'SampleRate',100)` specifies a signal datastore containing one member, a random signal, sampled at 100 Hz.

`outputLocation` — Folder location to write data

character vector | string scalar

Folder location to write data, specified as a character vector or string scalar. `outputLocation` can specify a full or relative path.

Example: `outputLocation = '../../dir/data'`

Example: `outputLocation = "C:\Users\MyName\Desktop"`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `writeall(sds,outputLocation,'FolderLayout','flatten')`

`FolderLayout` — Layout of files in output folder

`'duplicate'` (default) | `'flatten'`

Layout of files in the output folder, specified as either `'duplicate'` or `'flatten'`.

- `'duplicate'` — Replicate the folder structure of the data that the signal datastore points to. Specify the `'FolderLayout'` as `'duplicate'` to maintain correspondence between the input and output datasets.
- `'flatten'` — Write all the files from the input to the specified output folder without any subfolders.

`'FolderLayout'` does not apply when `sds` contains in-memory data.

Data Types: `char` | `string`

`FilenamePrefix` — Prefix to file name

character vector | string scalar

Prefix to file name, specified as a character vector or string scalar.

The `writeall` function adds the specified prefix to the output file names. For example, this code adds today's date to the beginning of all output file names from the datastore.

```
prefixText = string(datetime('today'))
writeall(imds,'C:\myFolder','FilenamePrefix',prefixText);
```

Data Types: `char` | `string`

`FilenameSuffix` — Suffix to file name

character vector | string scalar

Suffix to file name, specified as a character vector or string scalar.

The `writeall` function adds the specified suffix to the output file names. For example, this code adds the descriptive text `'jpeg_70per'` at the end of all output file names from the datastore.

```
writeall(imds,'C:\myFolder','FilenameSuffix','jpeg_70per');
```

Data Types: `char` | `string`

`UseParallel` — Indicator to write in parallel

`false` or `0` (default) | `true` or `1`

Indicator to write in parallel, specified as either `false` or `true`.

By default `writeall` writes in serial. If you set `UseParallel` to `true`, then `writeall` divides the writing operations into separate groups and runs the groups in parallel if:

- Parallel Computing Toolbox is installed.
- An open parallel pool exists or automatic pool creation is enabled in the Parallel Preferences.

Otherwise, `writeall` writes in serial regardless of the value for `UseParallel`.

Note Parallel writing is not supported for `CombinedDatastore` objects or datastores resulting from the `transform` applied to a `CombinedDatastore`.

Data Types: `logical`

`WriteFcn` — Custom writing function

function handle

Custom writing function, specified as a function handle. The specified function is responsible for creating the output files. You can use the `'WriteFcn'` name-value argument to transform data or write data to a file format different from the default, even if `writeall` does not directly support the output format.

Function Signature

The custom writing function must accept at least three input arguments, `data`, `writeInfo`, and `suggestedOutputType`.

```
function myWriteFcn(data,writeInfo,suggestedOutputType)
```

The function can also accept additional inputs, such as name-value arguments, after the first three required inputs.

- `data` contains the output of the `read` method operating on the datastore.
- `writeInfo` is an object of type `matlab.io.datastore.WriteInfo` with fields listed in the table.

Field	Description	Type
`ReadInfo`	The second output of the `read` method of the `signalDatastore`	`struct`
`SuggestedOutputName`	A fully qualified, globally unique file name that meets the location and naming requirements	`string`

Field	Description	Type
`Location`	The specified `outputLocation` passed to `writeall`	`string`

- `suggestedOutputType` is the suggested output file type.

Example Function

A simple write function that computes the spectrogram of the input signal using `pspectrum` and writes it to a text file in the current folder using the MATLAB function `writematrix`. The function specifies 80% of overlap between adjoining segments, a time resolution of 0.15 second, and a spectral leakage of 0.8.

```
function writeSpectrogram(data,info,~)

    [s,f,t] = pspectrum(data,info.ReadInfo.SampleRate,'spectrogram', ...
        'TimeResolution',0.15,'OverlapPercent',80,'Leakage',0.8);
    d = [NaN t'; f s];

    [~,q] = fileparts(info.SuggestedOutputName);
    writematrix(d,append(q,".txt"))

end
```

To use `writeSpectrogram` as the writing function for the `signalDatastore` object `sds`, use this command.

```
writeall(sds,'.','WriteFcn',@writeSpectrogram)
```

Data Types: `function_handle`

See Also

`signalDatastore` | `read` | `preview`

Introduced in R2021a

signalMask

Modify and convert signal masks and extract signal regions of interest

Description

Use `signalMask` to store the locations of regions of interest of a signal together with the label or category values for each region.

Using `signalMask`, you can:

- Represent a signal mask as a table, a categorical sequence, or a matrix of binary sequences.
- Modify regions of interest by extending or shortening their duration, merge same-category regions that are sufficiently close, or remove regions that are not long enough.
- Extract signal regions of interest from a signal vector.
- Plot a signal with color-coded regions of interest.

Creation

Syntax

```
msk = signalMask(src)
msk = signalMask(src,Name,Value)
```

Description

`msk = signalMask(src)` creates a signal mask for the input data source `src`. A mask defines the locations of regions of interest of a signal together with the label or category values for each region.

`msk = signalMask(src,Name,Value)` sets “Properties” on page 1-2106 using name-value arguments. You can specify multiple name-value arguments. Enclose each property name in quotes.

Input Arguments

`src` — Input data source

table | categorical vector sequence | matrix of binary sequences

Input data source, specified as a region-of-interest (ROI) table, a categorical vector sequence, or a matrix of binary sequences.

- When `src` is an ROI table, it must contain two variables:
  - The first variable is a two-column matrix. Each row of the matrix contains the beginning and end limits of a signal region of interest.
    - If `SampleRate` is specified, `signalMask` interprets the limits as time values expressed in seconds.

    - If `SampleRate` is not specified, `signalMask` interprets the limits as sample indices. If the matrix elements are not integers, `signalMask` rounds their values to the nearest integer greater than zero.
  - The second variable contains the region labels, specified as a categorical array or a string.
- When `src` is a categorical vector sequence, groups of contiguous same-value category elements indicate a signal region of interest labeled with that particular category. Elements that belong to no category (and hence have no label value) should be specified as the missing categorical, displayed as `<undefined>`. For more information, see `categorical`.
- When `src` is a *P*-column matrix of binary sequences, each column is interpreted as a signal mask with `true` elements marking regions of interest for each of *P* different categories, labeled with integers from 1 to *P*. If you prefer, you can specify a list of *P* category names using the `Categories` property.

Example: `signalMask(table([2 4;6 7],["male" "female"]'))` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Example: `signalMask(categorical(["" "male" "male" "male" "" "female" "female" ""]',["male" "female"]))` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Example: `signalMask([0 1 1 1 0 0 0 0;0 0 0 0 0 1 1 0]','Categories',["male" "female"])` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `table` | `categorical`

Properties

`SourceType` — Type of input source

`"roiTable"` | `"categoricalSequence"` | `"binarySequences"`

This property is read-only.

Type of input source, returned as `"roiTable"`, `"categoricalSequence"`, or `"binarySequences"`. This property is inferred from `src` and cannot be set.

Example: `signalMask(table([2 4;6 7],["male" "female"]'))` has `SourceType` returned as `"roiTable"`.

Example: `signalMask(categorical(["" "male" "male" "male" "" "female" "female" ""]',["male" "female"]))` has `SourceType` returned as `"categoricalSequence"`.

Example: `signalMask([0 1 1 1 0 0 0 0;0 0 0 0 0 1 1 0]','Categories',["male" "female"])` has `SourceType` returned as `"binarySequences"`.

Data Types: `string`

`SampleRate` — Sample rate value

positive numeric scalar

This property is read-only.

Sample rate value, specified as a positive numeric scalar. If `src` is specified as an ROI table, `signalMask` assumes that the table contains region limits expressed in seconds. If you omit this property, the object treats all region limits as sample indices.

Data Types: `single` | `double`

`Categories` — Category names
string vector | cell array of character vectors

Category names, specified as a string vector or a cell array of character vectors. This property can be set only when `src` is a matrix of binary sequences. For all other input `src` types, `signalMask` infers category names directly from `src` and this property is read-only. The vector has a number of elements equal to the number of columns of `src`, and its *i*th category corresponds to the *i*th column of `src`. If `src` has *P* columns and this property is not specified, `signalMask` sets the category names to `["1" "2" ... "P"]`.

Data Types: `string` | `char`

`SpecifySelectedCategories` — Option to select a subset of categories
`false` (default) | `true`

Option to select a subset of categories, specified as a logical value. If this property is set to `false` after creating the mask, then all categories in `Categories` are selected. `SpecifySelectedCategories` can only be used on an existing object and cannot be specified as a name-value argument.

Data Types: `logical`

`SelectedCategories` — Indices of selected categories
vector of integer index values

Indices of selected categories, specified as a vector of integer index values pointing to category elements in `Categories`. Categories not listed in this property are filtered out from the mask input when calling the object functions of `signalMask`. The category indices must be sorted in ascending order. This property applies only when `SpecifySelectedCategories` is `true`. `SelectedCategories` can only be used on an existing object and cannot be specified as a name-value argument.

Example: Given a set of categories `["woman" "girl" "man" "boy"]`, specifying `SelectedCategories` as `[1 2 4]` selects `["woman" "girl" "boy"]` and filters out the rest.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LeftExtension` — Number of samples to extend regions to the left
`0` (default) | positive integer

Number of samples to extend regions to the left, specified as a positive integer. The number of extended samples is truncated when the beginning of the sequence is reached. For more information, see “Region Limit Modification” on page 1-2111.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`RightExtension` — Number of samples to extend regions to the right
`0` (default) | positive integer

Number of samples to extend regions to the right, specified as a positive integer. For more information, see “Region Limit Modification” on page 1-2111.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LeftShortening` — Number of samples to shorten regions from the left
`0` (default) | positive integer

Number of samples to shorten regions from the left, specified as a positive integer. `signalMask` removes regions that are shortened by a number of samples equal to or greater than their length. For more information, see “Region Limit Modification” on page 1-2111.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`RightShortening` — Number of samples to shorten regions from the right
`0` (default) | positive integer

Number of samples to shorten regions from the right, specified as a positive integer. `signalMask` removes regions that are shortened by a number of samples equal to or greater than their length. For more information, see “Region Limit Modification” on page 1-2111.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MergeDistance` — Distance between regions to be merged
`0` (default) | positive integer

Distance between regions to be merged, specified as a positive integer. When this property is specified, `signalMask` merges regions of the same category that are separated by the specified number of samples or less. For more information, see “Region Limit Modification” on page 1-2111.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MinLength` — Minimum length of regions to keep
`1` (default) | positive integer

Minimum length of regions to keep, specified as a positive integer. When this property is specified, `signalMask` removes regions shorter than the specified number of samples. For more information, see “Region Limit Modification” on page 1-2111.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Object Functions

binmask	Get matrix of binary sequences mask
catmask	Get categorical sequence mask
extractsigroi	Extract regions of interest based on signal mask
plotsigroi	Plot signal regions based on signal mask
roimask	Get ROI table mask

Examples

Regions of Interest in Audio File

Load a speech signal sampled at F_s = 7418 Hz. The file contains a recording of a female voice saying the word "MATLAB®."

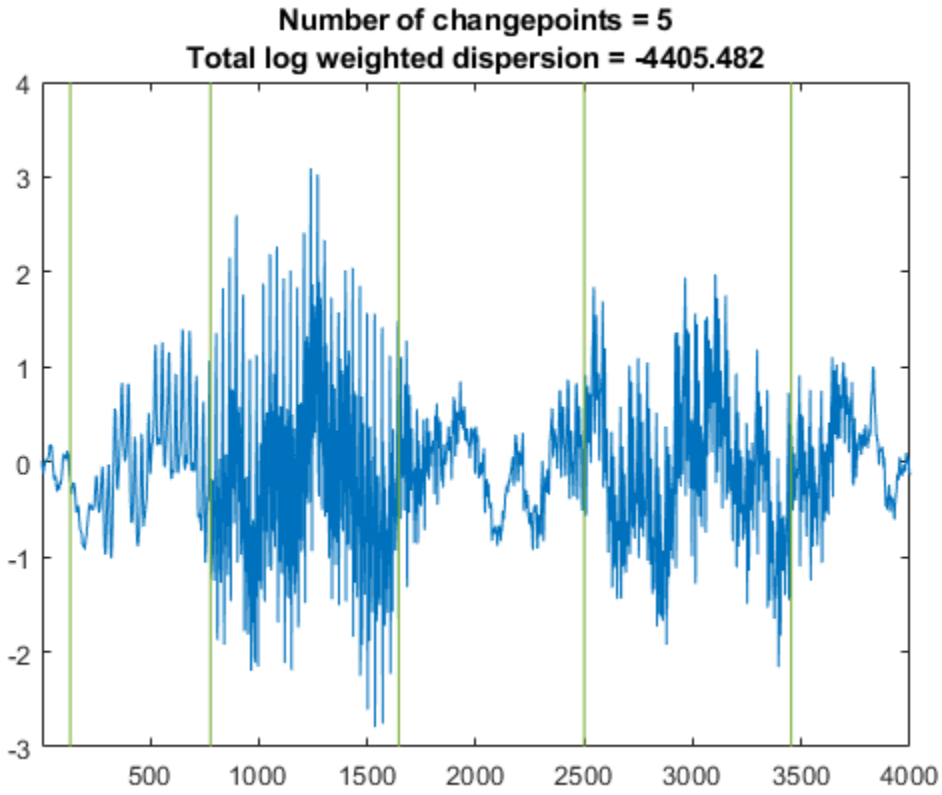
```
load mtlb
t = (0:length(mtlb)-1)/Fs;
```

Discern the vowels and consonants in the word by finding the points at which the variance of the signal changes significantly. Limit the number of changepoints to five.

```
q = findchangepts(mtlb,"Statistic","rms","MaxNumChanges",5);
```

Plot the signal and display the changepoints.

```
findchangepts(mtlb,"Statistic","rms","MaxNumChanges",5)
axis tight
```



Define regions of interest that correspond to each letter in the word.

```
roitable = t([[1;q] [q;length(mtlb)]]);
```

Assign region labels and preserve their order.

```
x = ["M" "A" "T" "L" "A" "B"]';
y = unique(x,"stable");
c = categorical(x,y);
```

Create a signal mask for the regions of interest and corresponding labels. Shorten each region by one sample from the right to avoid contiguity. Display the region-of-interest table mask.

```
src = table(roitable,c);
msk = signalMask(src,"SampleRate",Fs,"RightShortening",1);
roimask(msk)
```

```
ans=6×2 table
         roitable          c
    ____________________    _

          0    0.017525    M
    0.01766     0.10461    A
```

```
    0.10475     0.22162    T
    0.22176     0.33675    L
    0.33688     0.46535    A
    0.46549     0.53909    B
```

Introduce gaps in the signal where the letter "A" is vocalized.

```
m = mtlb;

seq = catmask(msk,length(mtlb));
m(seq == "A") = NaN;
```

Reconstruct the signal using an autoregressive process. Extract each region of interest from the reconstructed signal.

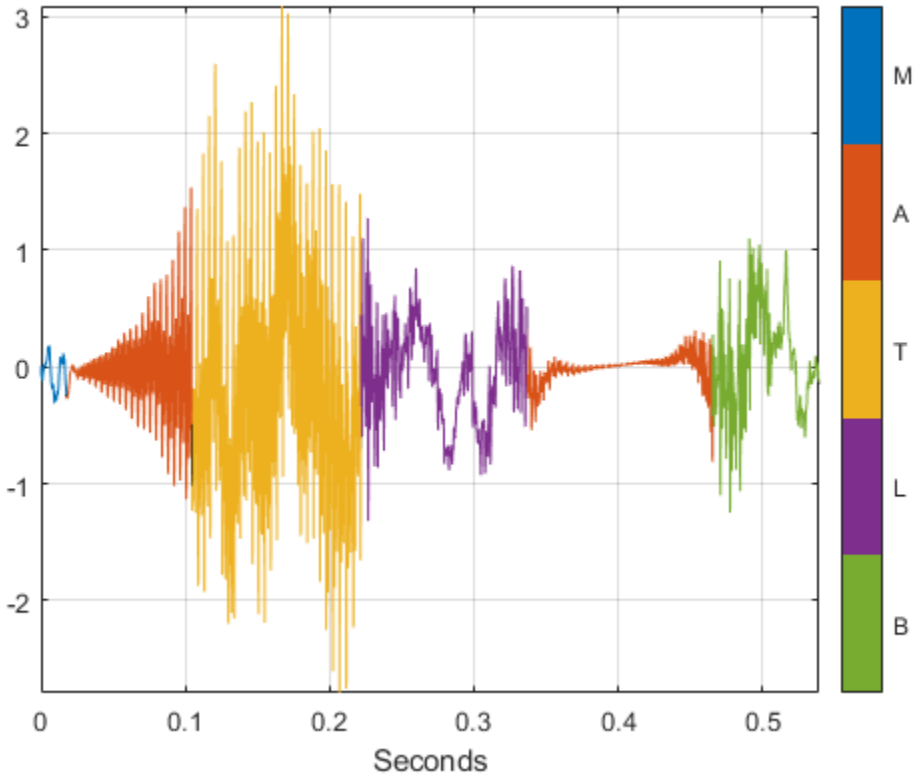
```
p = fillgaps(m);

w = extractsigroi(msk,p);
```

To play the sound with a pause after each region, uncomment these lines:

```
% for k = 1:length(w)
%     sound(cell2mat(w{k}),Fs)
%     pause(0.5)
% end
```

Plot the reconstructed signal and visualize the regions of interest.

```
figure
plotsigroi(msk,p)
```



More About

Region Limit Modification

You can use `signalMask` to modify regions of interest by extending or shortening their duration, merge same-category regions that are sufficiently close, or remove regions that are not long enough.

`signalMask` first converts the region limits of `src` to a matrix and, depending on the specified "Properties" on page 1-2106, modifies the limits in this order:

1. Extends regions to the left or right based on `LeftExtension` and `RightExtension`.
2. Shortens regions from the left or right based on `LeftShortening` and `RightShortening`.
3. Merges close-enough regions based on `MergeDistance`. `signalMask` always merges contiguous, overlapping, or repeated regions. Think of these regions as being separated by zero samples, or a negative number of samples.
4. Removes short regions based on `MinLength`.

See Also

Apps
Signal Labeler

Functions
`binmask2sigroi` | `extendsigroi` | `extractsigroi` | `mergesigroi` | `removesigroi` | `shortensigroi` | `sigroi2binmask` | `categorical`

Objects
`labeledSignalSet` | `signalLabelDefinition`

Topics
“Label Definitions for Whale Songs” on page 1-1203
“Automate Signal Labeling with Custom Functions”
“Label Spoken Words in Audio Signals”
“Label Signal Attributes, Regions of Interest, and Points”
“Examine Labeled Signal Set”

Introduced in R2020b

binmask

Get matrix of binary sequences mask

Syntax

```
seqs = binmask(msk)
seqs = binmask(msk,len)

[seqs,numroi,cats] = binmask( ___ )
```

Description

`seqs = binmask(msk)` returns a matrix of binary sequences mask, `seqs`, based on the source and properties in `msk`.

`seqs = binmask(msk,len)` specifies the lengths of the sequences in `seqs`.

`[seqs,numroi,cats] = binmask( ___ )` also returns `numroi`, a vector containing the number of regions found for each of the categories listed in `cats`.

Examples

Binary Sequences Mask from ROI Table

Consider a region-of-interest (ROI) table mask with four regions of interest spanning samples numbered from 2 to 19. Specify the category labels as `A`, `B`, and `C`. Use the mask to create a `signalMask` object.

```
roiTbl = table([2 5; 7 10; 15 18; 17 19],["A" "B" "C" "A"]');

m = signalMask(roiTbl);
```

Extract a binary sequences mask from the object. Specify a sequence length of 20 samples.

```
binSeqs = binmask(m,20)'
```

```
binSeqs = 3x20 logical array

   0   1   1   1   1   0   0   0   0   0   0   0   0   0   0   0   1   1   1   0
   0   0   0   0   0   0   1   1   1   1   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   1   1   1   0   0
```

Shorten the regions of interest by one sample from the right and extend them two samples to the left. Extract the modified mask, specifying a sequence length of 18 samples.

```
m.RightShortening = 1;
m.LeftExtension = 2;

binSeqs = binmask(m,18)'
```

```
binSeqs = 3x18 logical array

   1   1   1   1   0   0   0   0   0   0   0   0   0   0   1   1   1   1
   0   0   0   0   1   1   1   1   1   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   1   1   1   1   1   0
```

Input Arguments

`msk` — Signal mask

`signalMask` object

Signal mask, specified as a `signalMask` object.

Example: `signalMask(table([2 4;6 7],["male" "female"]'))` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Example: `signalMask(categorical(["" "male" "male" "male" "" "female" "female" ""]',["male" "female"]))` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Example: `signalMask([0 1 1 1 0 0 0 0;0 0 0 0 0 1 1 0]','Categories',["male" "female"])` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

`len` — Output sequence length

integer scalar

Output sequence length, specified as an integer scalar. Regions beyond `len` are ignored. The output matrix of binary sequences `seqs` is padded with `false` values in these cases:

- `SourceType` is `'categoricalSequence'` or `'binarySequences'` and `len` is greater than the length of the source sequence.
- `SourceType` is `'roiTable'` and `len` is greater than the maximum region index.

For more information about the length of the output, see “Region Limit Modification” on page 1-2111.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`seqs` — Matrix of binary sequences mask

matrix

Matrix of binary sequences mask, returned as a matrix. The *i*th column of `seqs` contains a binary mask sequence for the *i*th category listed in the `SelectedCategories` property of `msk`.

- If `SourceType` is `'categoricalSequence'` or `'binarySequences'` and `len` is not specified, then `seqs` has the same length as the source mask sequence.
- If `SourceType` is `'roiTable'`, then `len` must be specified.

When `RightExtension` is nonzero and `SourceType` is `'categoricalSequence'` or `'binarySequences'`, `binmask` extends regions possibly beyond the sequence length, applies all other modifications based on `LeftExtension`, `LeftShortening`, `RightShortening`,

`MergeDistance`, and `MinLength`, and then truncates the resulting sequence to the original sequence length, or to the specified length `len`.

For more information on how the properties of `msk` affect the length of `seqs`, see “Region Limit Modification” on page 1-2111.

`numroi` — Number of regions
vector of integers

Number of regions found for each of the categories in `cats`, returned as a vector of integers.

`cats` — Category list
vector of strings

Category list, returned as a vector of strings.

See Also

Apps
Signal Labeler

Objects
`labeledSignalSet` | `signalMask` | `signalLabelDefinition`

Topics
“Label Definitions for Whale Songs” on page 1-1203
“Automate Signal Labeling with Custom Functions”
“Label Spoken Words in Audio Signals”
“Label Signal Attributes, Regions of Interest, and Points”
“Examine Labeled Signal Set”

Introduced in R2020b

binmask2sigroi

Convert binary mask to matrix of ROI limits

Syntax

```
roilims = binmask2sigroi(mask)
```

Description

`roilims = binmask2sigroi(mask)` converts `mask`, a binary mask of signal region-of-interest (ROI) samples, to a matrix of ROI limits, `roilims`.

Examples

Convert Binary Sequence to ROI Limits

Consider a logical sequence that is true for samples belonging to four possible regions of interest of a signal. Convert the sequence to a two-column matrix of ROI limits.

```
mask = logical([0 0 1 1 1 1 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1 1 0 0]);

roilims = binmask2sigroi(mask)
```

```
roilims = 4×2

     3     6
    11    13
    19    27
    31    32
```

Input Arguments

`mask` — Binary mask

logical vector | numeric vector

Binary mask, specified as a logical vector. You can also specify `mask` as a numeric vector. In that case, any nonzero element of the vector is converted to logical `1` (`true`) and zeros are converted to logical `0` (`false`).

Example: `logical([0 0 1 1 1 0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 1 1 0])` specifies a binary mask containing four regions of interest.

Example: `[0 0 1 1 1 0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 1 1 0]` specifies a binary mask containing four regions of interest.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Output Arguments

`roilims` — Region-of-interest limits

two-column matrix of positive integers

Region-of-interest limits, returned as a two-column matrix of positive integers. The *i*th row of `roilims` contains nondecreasing indices corresponding to the beginning and end samples of the *i*th region of interest.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Objects

`signalMask`

Functions

`extendsigroi` | `extractsigroi` | `mergesigroi` | `removesigroi` | `shortensigroi` | `sigroi2binmask`

Introduced in R2020b

catmask

Get categorical sequence mask

Syntax

```
seq = catmask(msk)
seq = catmask(msk,len)

seq = catmask( ___ ,'OverlapAction',action)
seq = catmask( ___ ,'OverlapAction','prioritizeByList','PriorityList',idxlist)

[seq,numroi,cats] = catmask( ___ )
```

Description

`seq = catmask(msk)` returns a categorical sequence mask, `seq`, based on the source and properties in `msk`.

`seq = catmask(msk,len)` specifies the length of `seq`.

`seq = catmask( ___ ,'OverlapAction',action)` specifies how `signalMask` deals with regions having different category values that overlap.

`seq = catmask( ___ ,'OverlapAction','prioritizeByList','PriorityList',idxlist)` specifies the order in which `msk` categories are prioritized when regions with different category values overlap.

`[seq,numroi,cats] = catmask( ___ )` also returns `numroi`, a vector containing the number of regions found for each of the categories listed in `cats`.

Examples

Categorical Mask with Specified Length

Consider a region-of-interest (ROI) table mask with four regions of interest spanning samples numbered from 2 to 30. Specify the category labels as `A` and `B`. Use the mask to create a `signalMask` object.

```
roiTbl = table([2 5; 7 10; 15 25; 28 30],["A","B","B","A"]');

m = signalMask(roiTbl);
```

Extract a categorical mask from the object specifying a sequence length of 35. Samples beyond the 30th are returned as `<undefined>`.

```
catSeq = catmask(m,35);

catSeq(max(roiTbl.Var1(end)):end)
```

```
ans = 6x1 categorical
     A
```

```
     <undefined>
     <undefined>
     <undefined>
     <undefined>
     <undefined>
```

Extract the categorical mask again, but now specify a sequence length of 8. Samples beyond the eighth are removed.

```
catSeq = catmask(m,8)
```

```
catSeq = 8x1 categorical
     <undefined>
     A
     A
     A
     A
     <undefined>
     B
     B
```

Categorical Mask from Overlapping Binary Sequences

Consider an 18-by-2 mask of binary sequences. Use the mask to create a `signalMask` object. Label the categories with `A` and `B`, in that order.

```
binSeqs = logical([ ...
    0 0 1 1 1 0 0 1 1 0 0 0 1 1 1 1 0 1;
    1 1 0 0 0 1 1 1 1 0 1 1 0 1 0 1 1 0]');

m = signalMask(binSeqs);
m.Categories = ["A" "B"];
```

Extract a categorical mask from the object. To treat overlap between the categories, assign samples shared by the two categories to the first one in the list, `A`.

```
seqA = catmask(m,'OverlapAction','PrioritizeByList');
seqA(binSeqs(:,1) & binSeqs(:,2))
```

```
ans = 4x1 categorical
     A
     A
     A
     A
```

Extract the categorical mask again, but now treat overlap between the categories by assigning the shared samples to `B` with an explicit priority list.

```
seqB = catmask(m,'OverlapAction','PrioritizeByList', ...
    'PriorityList',[2 1]);
seqB(binSeqs(:,1) & binSeqs(:,2))
```

```
ans = 4x1 categorical
     B
```

```
     B
     B
     B
```

Treat the overlap by removing regions with fewer than three samples. Display the modified binary-sequence mask that results.

```
m.MinLength = 3;
binmask(m)'
```

```
ans = 2x18 logical array

   0   0   1   1   1   0   0   0   0   0   0   0   1   1   1   1   0   0
   0   0   0   0   0   1   1   1   1   0   0   0   0   0   0   0   0   0
```

The formerly shared samples are assigned to the categories of the regions that remain.

```
seqM = catmask(m);
seqM(binSeqs(:,1) & binSeqs(:,2))
```

```
ans = 4x1 categorical
     B
     B
     A
     A
```

Input Arguments

`msk` — Signal mask

`signalMask` object

Signal mask, specified as a `signalMask` object.

Example: `signalMask(table([2 4;6 7],["male" "female"]'))` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Example: `signalMask(categorical(["" "male" "male" "male" "" "female" "female" ""]',["male" "female"]))` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Example: `signalMask([0 1 1 1 0 0 0 0;0 0 0 0 0 1 1 0]','Categories',["male" "female"])` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

`len` — Output sequence length

integer scalar

Output sequence length, specified as an integer scalar. Regions beyond `len` are ignored. The output categorical sequence `seq` is padded with `<missing>` values in these cases:

- `SourceType` is `'categoricalSequence'` or `'binarySequences'` and `len` is greater than the length of the source sequence.
- `SourceType` is `'roiTable'` and `len` is greater than the maximum region index.

When `RightExtension` is nonzero and `SourceType` is `'categoricalSequence'` or `'binarySequences'`, `catmask` extends regions possibly beyond the sequence length, applies all other modifications based on `LeftExtension`, `LeftShortening`, `RightShortening`, `MergeDistance`, and `MinLength`, and then truncates the resulting sequence to the original sequence length, or to the specified length `len`.

As a last step, `catmask` manages overlap based on the value set for `'OverlapAction'`, if that argument is specified.

For more information about the length of the output, see “Region Limit Modification” on page 1-2111.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`action` — Way to deal with overlap
`'error'` (default) | `'prioritizeByList'`

Way to deal with overlap, specified as `'error'` or `'prioritizeByList'`.

- `'error'` — `catmask` throws an error if there are overlaps between regions with different categories.
- `'prioritizeByList'` — `catmask` uses the priority list specified in `idxlist` to deal with overlaps between regions with different categories. The first category in the list has the highest priority, and all its samples are kept in cases of overlap. The second category in the list follows, and its samples are kept in overlap cases not previously resolved.

  If `idxlist` is not specified, `catmask` prioritizes categories in the same order as they appear in the `Categories` property of `msk`.

Data Types: `char` | `string`

`idxlist` — Category priorities in cases of overlap
`msk.Categories` list (default) | vector of integers

Category priorities in cases of overlap, specified as a vector of integers. The indices in `idxlist` correspond to entries in the `Categories` of `msk` and are ordered by the priority with which they should be treated when regions with different category values overlap. `idxlist` must contain indices for all the elements in `Categories`. The first category in the list has the highest priority. This means that when regions with different categories overlap, all the values of the highest priority are kept. Then the values with the next highest priority are kept in the remaining nonoverlapping samples, and so on.

If `idxlist` is not specified, `catmask` prioritizes categories in the same order as they appear in the `Categories` property of `msk`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`seq` — Categorical sequence mask
categorical array

Categorical sequence mask, returned as a categorical array. Samples in `seq` that do not belong to a region of interest and have no label value are set to missing categorical values, displayed as `<undefined>`. For more information, see `categorical`.

- If `SourceType` is `'categoricalSequence'` or `'binarySequences'` and `len` is not specified, then `seqs` has the same length as the source mask sequence.
- If `SourceType` is `'roiTable'`, then `len` must be specified.

For more information on how the properties of `msk` affect the length of `seqs`, see “Region Limit Modification” on page 1-2111.

`numroi` — Number of regions
vector of integers

Number of regions found for each of the categories in `cats`, returned as a vector of integers.

`cats` — Category list
vector of strings

Category list, returned as a vector of strings.

See Also

Apps
Signal Labeler

Objects
`labeledSignalSet` | `signalMask` | `signalLabelDefinition`

Topics
“Label Definitions for Whale Songs” on page 1-1203
“Automate Signal Labeling with Custom Functions”
“Label Spoken Words in Audio Signals”
“Label Signal Attributes, Regions of Interest, and Points”
“Examine Labeled Signal Set”

Introduced in R2020b

extendsigroi

Extend signal regions of interest to left and right

Syntax

```
roilimsout = extendsigroi(roilims,sl,sr)
```

Description

`roilimsout = extendsigroi(roilims,sl,sr)` extends the signal regions of interest specified in `roilims` to the left by `sl` samples and to the right by `sr` samples.

Examples

Extend Regions of Interest

Consider a two-column matrix of integers that can represent regions of interest of a signal. Extend the regions of interest by two samples to the left and three samples to the right. `extendsigroi` does not extend regions to the left beyond the first sample.

```
rois = [1 8; 17 20; 27 31; 38 40];
xrois = extendsigroi(rois,2,3)
```

```
xrois = 4×2

     1    11
    15    23
    25    34
    36    43
```

Extend Overlapping Regions of Interest

Consider a two-column matrix of integers that can represent overlapping regions of interest of a signal. Extend the regions of interest by two samples to the left and three samples to the right. `extendsigroi` does not extend regions to the left beyond the first sample.

```
rois = [1 10; 17 26; 24 32; 38 40];

xrois = extendsigroi(rois,2,3)
```

```
xrois = 4×2

     1    13
    15    29
    22    35
    36    43
```

Input Arguments

`roilims` — Region-of-interest limits

two-column matrix of positive integers

Region-of-interest limits, specified as a two-column matrix of positive integers. The *i*th row of `roilims` contains nondecreasing indices corresponding to the beginning and end samples of the *i*th region of interest of a signal.

Example: `[5 8; 12 20; 18 25]` specifies a two-column region-of-interest matrix with three regions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sl` — Number of samples to extend to the left

nonnegative integer

Number of samples to extend to the left, specified as a nonnegative integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sr` — Number of samples to extend to the right

nonnegative integer

Number of samples to extend to the right, specified as a nonnegative integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`roilimsout` — Modified region-of-interest limits

two-column matrix of positive integers

Modified region-of-interest limits, returned as a two-column matrix of positive integers. Output limits are returned in sorted order using the `sortrows` function.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Objects

`signalMask`

Functions
`binmask2sigroi` | `extractsigroi` | `mergesigroi` | `removesigroi` | `shortensigroi` | `sigroi2binmask`

Introduced in R2020b

extractsigroi

Extract regions of interest based on signal mask

Syntax

```
sigroi = extractsigroi(msk,x)

sigroi = extractsigroi(msk,x,Name,Value)

[sigroi,limits] = extractsigroi( ___ )
[sigroi,limits,numroi,cats] = extractsigroi( ___ )
```

Description

`sigroi = extractsigroi(msk,x)` extracts regions of interest of the input signal vector `x` based on the source and properties in `msk`.

`sigroi = extractsigroi(msk,x,Name,Value)` specifies additional options using name-value arguments. You can choose to concatenate extracted regions and select the number of regions to extract per category.

`[sigroi,limits] = extractsigroi( ___ )` returns an array with the locations of the extracted region endpoints.

`[sigroi,limits,numroi,cats] = extractsigroi( ___ )` also returns `numroi`, a vector containing the number of regions found for each of the categories listed in `cats`.

Examples

Extract Regions from ROI Table Mask

Consider a region-of-interest (ROI) table with three regions labeled `A` and two regions labeled `B`. Use the table to create a `signalMask` object.

```
roiTbl = table([2 5; 7 10; 12 13; 15 25; 28 30],["A","B","A","B","A"]');

m = signalMask(roiTbl);
```

Generate an array with prime numbers smaller than 150. Use the `signalMask` object to extract the primes specified in the ROI table. Display the first set of `A` primes and the first set of `B` primes.

```
prm = primes(150);

rgs = extractsigroi(m,prm);

AB = [rgs{1}{1} rgs{2}{1}]
```

```
AB = 4×2

     3    17
```

```
     5    19
     7    23
    11    29
```

Repeat the operation, but now concatenate the regions of interest for each category. Display the first six A primes and the last six B primes.

```
rgs = extractsigroi(m,prm,'ConcatenateRegions',true);

AB = [rgs{1}(1:6) rgs{2}(end-5:end)]
```

```
AB = 6×2

     3    71
     5    73
     7    79
    11    83
    37    89
    41    97
```

Repeat the operation, but now ignore regions with two samples or less. Display the first six A primes and the last six B primes.

```
m.MinLength = 3;

rgs = extractsigroi(m,prm,'ConcatenateRegions',true);

AB = [rgs{1}(1:6) rgs{2}(end-5:end)]
```

```
AB = 6×2

     3    71
     5    73
     7    79
    11    83
   107    89
   109    97
```

Input Arguments

`msk` — Signal mask

`signalMask` object

Signal mask, specified as a `signalMask` object.

Example: `signalMask(table([2 4;6 7],["male" "female"]'))` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Example: `signalMask(categorical(["" "male" "male" "male" "" "female" "female" ""]',["male" "female"]))` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Example: `signalMask([0 1 1 1 0 0 0 0;0 0 0 0 0 1 1 0]','Categories',["male" "female"])` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

x — Input signal
vector

Input signal, specified as a vector.

Example: `chirp(0:1/1e3:1,25,1,50)` specifies a chirp sampled at 1 kHz.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ConcatenateRegions',true,'SelectedRegions',[2 4]` specifies that the function extract the second and fourth regions of each category and concatenate them.

`ConcatenateRegions` — Option to concatenate extracted signal regions
`false` (default) | `true`

Option to concatenate extracted signal regions, specified as a logical value.

- If this argument is set to `false`, then each cell of `sigroi` is a cell array corresponding to an individual signal region.
- If this argument is set to `true`, then each cell of `sigroi` is a vector of concatenated extracted signal regions for each category contained in `msk`.

Data Types: `logical`

`SelectedRegions` — Regions selected for extraction
vector of integers

Regions selected for extraction, specified as a vector of integers.

- If this argument is set to `1`, then `extractsigroi` extracts only the first region of each category and returns it in `sigroi`.
- If this argument is set to `[i j k ...]`, then `extractsigroi` extracts the `i`th, `j`th, `k`th, and successive regions of each category and returns them in `sigroi`.

The function ignores indices larger than the number of regions present for a given category.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`sigroi` — Signal regions of interest
cell array

Signal regions of interest, returned as a cell array. Each cell of `sigroi` contains a cell array with the signal regions extracted for each category in `msk`.

- If “`ConcatenateRegions`” on page 1-0 is set to `false`, then each cell of `sigroi` is a cell array corresponding to an individual signal region.

- If “ConcatenateRegions” on page 1-0 is set to `true`, then each cell of `sigroi` is a vector of concatenated extracted signal regions for each category contained in `msk`.

`limits` — Extracted region limits
cell array of two-column matrices

Extracted region limits, returned as a cell array of two-column matrices. If `msk` specifies a `SampleRate`, the region limits are expressed in seconds. If `msk` does not specify a sample rate, the region limits are integers corresponding to signal sample indices.

`numroi` — Number of regions
vector of integers

Number of regions found for each of the categories in `cats`, returned as a vector of integers.

`cats` — Category list
vector of strings

Category list, returned as a vector of strings.

See Also

Apps
Signal Labeler

Objects
`labeledSignalSet` | `signalMask` | `signalLabelDefinition`

Topics
“Label Definitions for Whale Songs” on page 1-1203
“Automate Signal Labeling with Custom Functions”
“Label Spoken Words in Audio Signals”
“Label Signal Attributes, Regions of Interest, and Points”
“Examine Labeled Signal Set”

Introduced in R2020b

extractsigroi

Extract signal regions of interest

Syntax

```
sigroi = extractsigroi(x,roilims)
sigroi = extractsigroi(x,roilims,concat)
```

Description

`sigroi = extractsigroi(x,roilims)` extracts regions of interest (ROIs) of the input signal vector `x` based on the ROI limits specified in `roilims`.

`sigroi = extractsigroi(x,roilims,concat)` with `concat` specified as `true` extracts regions of interest and concatenates them.

Examples

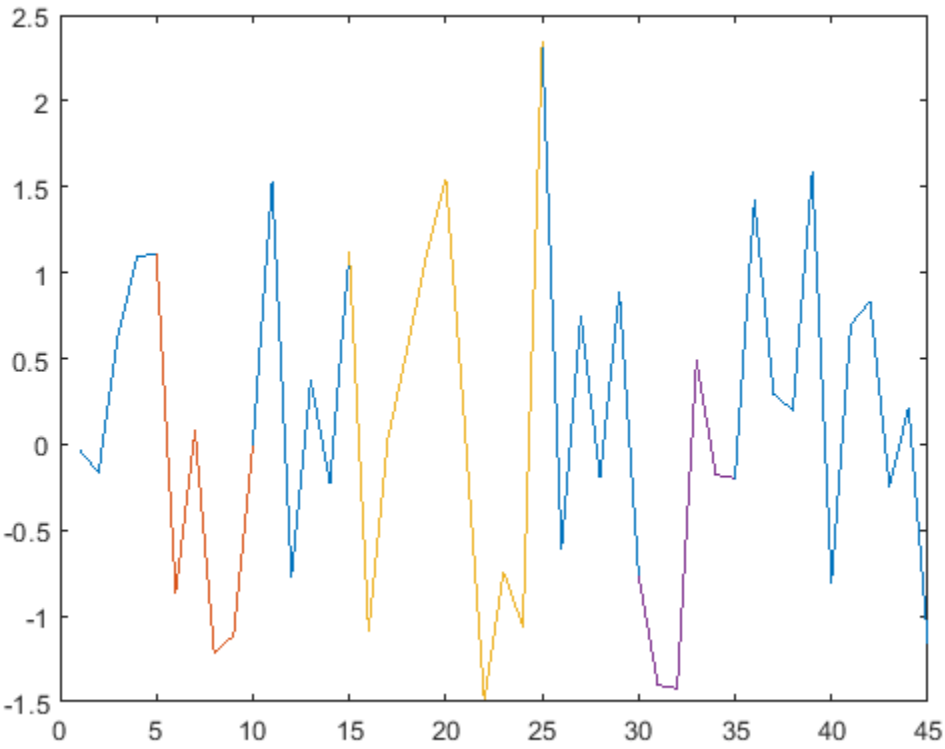
Extract Signal Regions of Interest

Consider a two-column matrix representing possible regions of interest of a 45-sample random signal. Extract the signal samples corresponding to the regions of interest.

```
x = randn(45,1);

roilims = [5 10; 15 25; 30 35];

sigroi = extractsigroi(x,roilims);
```

Plot the signal and highlight the regions of interest.

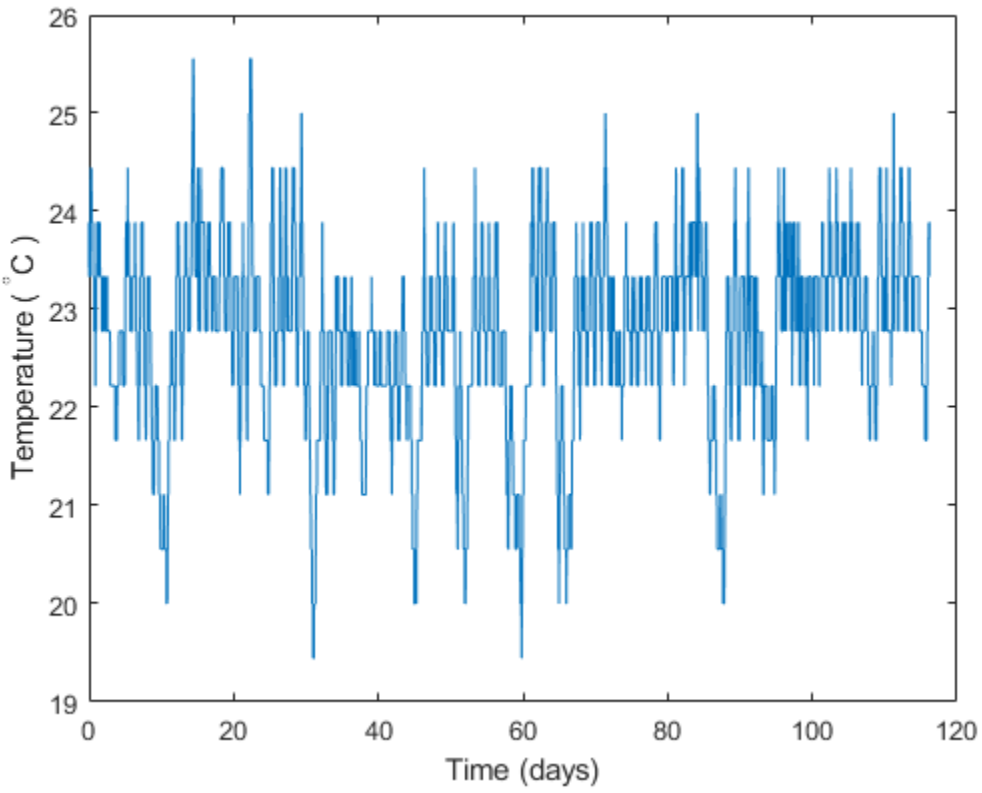
```
plot(x)

hold on
for kj = 1:length(sigroi)
    plot(roilims(kj,1):roilims(kj,2),sigroi{kj})
end
hold off
```



Extract Regions of Interest from Data Set

Consider a set of temperature data collected by a thermometer inside an office building for about four months. The device takes a reading every half hour. The sample rate is thus 48 measurements/day. Convert the temperature to degrees Celsius and plot the data.

```
load officetemp

tempC = (temp-32)*5/9;

fs = 48;
t = (0:length(tempC) - 1)/fs;

plot(t,tempC)
xlabel('Time (days)')
ylabel('Temperature ( {}^\circC )')
```



Create region-of-interest limits that separate the temperature data into 29-day periods.

```
roilims = [1 29; 30 58; 59 87; 88 116];
```

Extract the regions of interest. Compute the mean temperature of each period and display the values.

```
sigroi = extractsigroi(tempC,roilims*fs);
```

```
cellfun(@mean,sigroi)'
```

```
ans = 1×4

   22.8819   22.3073   22.7633   23.0066
```

Extract and Concatenate Signal Regions of Interest

Consider a two-column matrix representing possible regions of interest of a 45-sample random signal. Extract the signal samples corresponding to the regions of interest. Concatenate the samples into a single vector.

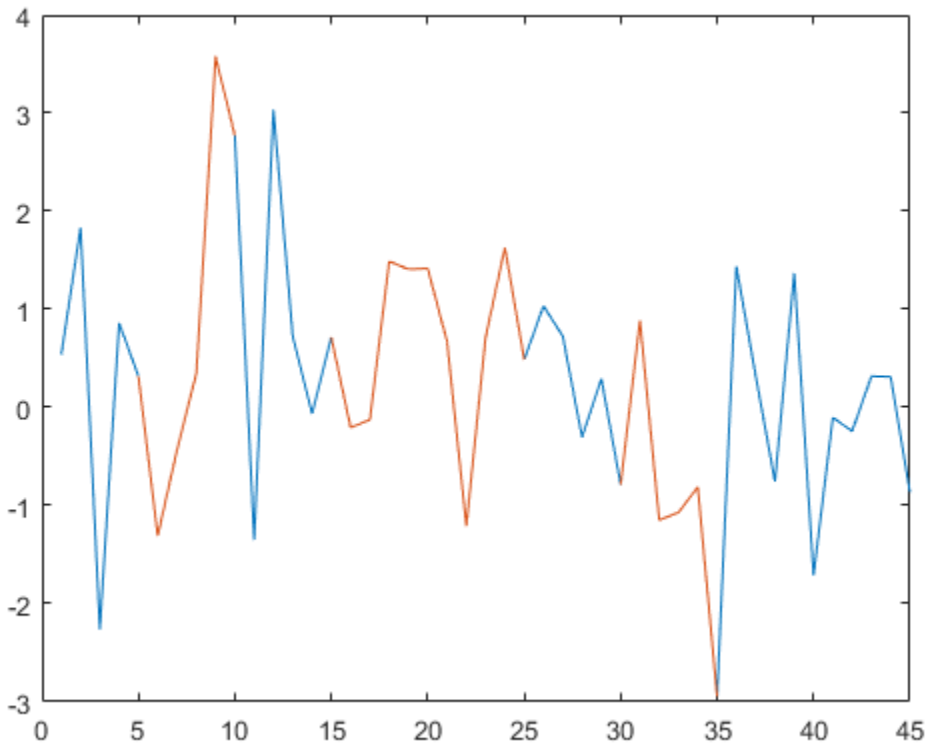
```
x = randn(45,1);
```

```
roilims = [5 10; 15 25; 30 35];
```

```
sigroi = extractsigroi(x,roilims,true);
```

Plot the signal and highlight the regions of interest.

```
plot(x)

y = NaN(size(x));

for kj = 1:size(roilims,1)
    roi = roilims(kj,1):roilims(kj,2);
    y(roi) = sigroi(1:length(roi));
    sigroi(1:length(roi)) = [];
end

hold on
plot(y)
hold off
```



Extract and Concatenate Regions of Interest From Data Set

Consider a set of temperature data collected by a thermometer inside an office building for four months. The device takes a reading every half hour. The sample rate is thus 48 measurements/day. Convert the temperature to degrees Celsius.

```
load officetemp

tempC = (temp-32)*5/9;

fs = 48;
```

Create region-of-interest (ROI) limits that correspond to five random two-week periods separated by at least 24 hours. Use the temperature readings from these days for an audit.

```
r = 5;
w = 14*fs;
s = 1*fs;

hq = histcounts(randi(r+1,1,length(tempC)-r*w-(r-1)*s),(1:r+2)-1/2);
t = (1 + (0:r-1)*(w+s) + cumsum(hq(1:r)))';

roilims = [t t+w-1];
```

Extract the regions of interest. Compute the mean temperature of each audited region of interest and display the values.

```
sigroi = extractsigroi(tempC,roilims);

cellfun(@mean,sigroi)'
```

```
ans = 1×5

   22.8075   22.2586   22.4256   22.9018   23.1457
```

Extract the regions of interest again, but now concatenate the samples into a single vector. Compute the mean temperature across the audited regions.

```
sigroic = extractsigroi(tempC,roilims,true);

avgTFc = mean(sigroic)
```

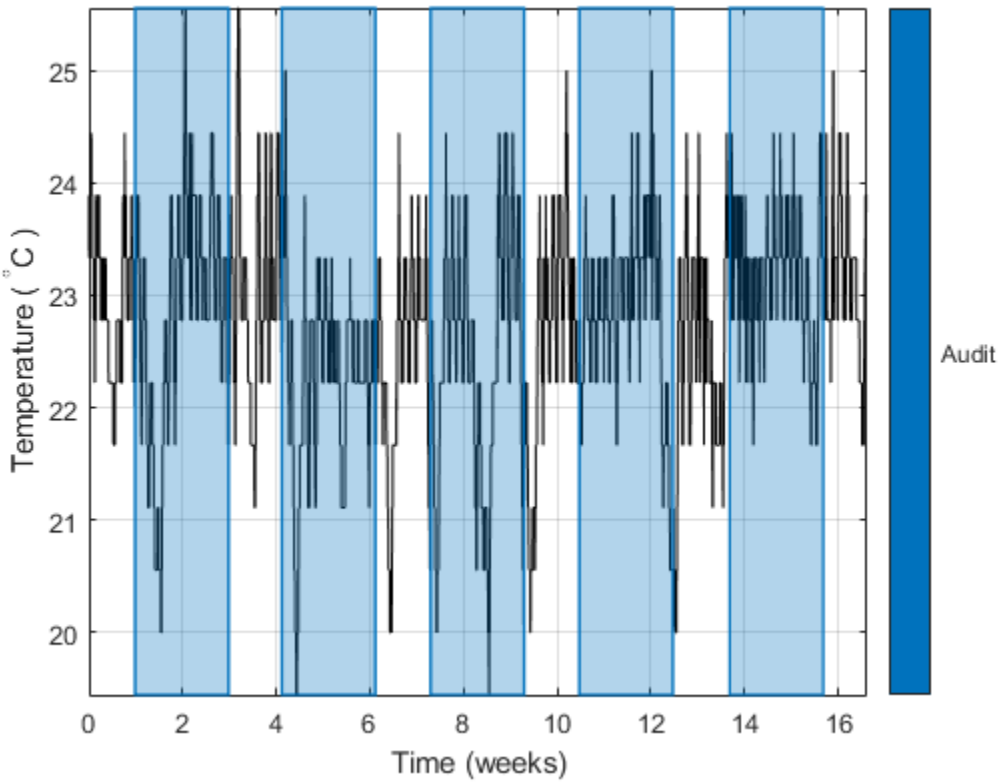
```
avgTFc = 22.7078
```

Convert the ROI limits to a binary sequence and create a mask. Express time in weeks.

```
m = sigroi2binmask(roilims,length(tempC));

msk = signalMask(m,'SampleRate',fs*7,'Categories',"Audit");
```

Plot the data and visualize the regions of interest with rectangular patches.

```
plotsigroi(msk,tempC,true)
xlabel('Time (weeks)')
ylabel('Temperature ( {}^\circC )')
```



Input Arguments

x — Input signal

vector

Input signal, specified as a vector.

Example: `chirp(0:1/1e3:1,25,1,50)` specifies a chirp sampled at 1 kHz.

Data Types: `single` | `double`

roilims — Region-of-interest limits

two-column matrix of positive integers

Region-of-interest limits, specified as a two-column matrix of positive integers. The *i*th row of `roilims` contains nondecreasing indices corresponding to the beginning and end samples of the *i*th region of interest of a signal.

Example: `[5 8; 12 20; 18 25]` specifies a two-column region-of-interest matrix with three regions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

concat — Option to concatenate extracted signal regions

`false` (default) | `true`

Option to concatenate extracted signal regions, specified as a logical value.

Data Types: `logical`

Output Arguments

`sigroi` — Signal regions of interest

cell array | vector

Signal regions of interest, returned as a cell array or a vector.

- If `concat` is set to `false`, `sigroi` is a cell array. The *i*th cell of `sigroi` contains the signal samples corresponding to the *i*th region of interest specified in `roilims`.
- If `concat` is set to `true`, `sigroi` is a vector that concatenates all extracted signal samples.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If supplied, `concat` must be a constant.

See Also

Objects

`signalMask`

Functions

`binmask2sigroi` | `extendsigroi` | `mergesigroi` | `removesigroi` | `shortensigroi` | `sigroi2binmask`

Introduced in R2020b

mergesigroi

Merge signal regions of interest

Syntax

```
roilimsout = mergesigroi(roilims,s)
```

Description

`roilimsout = mergesigroi(roilims,s)` merges the signal regions of interest specified in `roilims` if they are separated by `s` samples or less.

Examples

Merge Regions of Interest

Consider a two-column matrix of integers that can represent regions of interest of a signal. Merge any regions that are separated by four samples or less.

```
rois = [1 10; 17 26; 28 43; 47 57; 64 66];

xrois = mergesigroi(rois,4)
```

```
xrois = 3×2

     1    10
    17    57
    64    66
```

Specify the maximum separation as zero to merge contiguous or repeated regions.

```
nrois = [rois; 57 65; 1 10];

xrois = mergesigroi(rois,0)
```

```
xrois = 5×2

     1    10
    17    26
    28    43
    47    57
    64    66
```

Merge Overlapping Regions of Interest

Consider a two-column matrix of integers that can represent regions of interest of a signal. Merge any regions that are separated by four samples or less. The function merges overlapping regions in all cases.

```
rois = [1 10; 17 26; 24 32; 36 40];

xrois = mergesigroi(rois,4)
```

```
xrois = 2×2

     1    10
    17    40
```

Specify the maximum separation as zero to merge contiguous or repeated regions.

```
nrois = [rois; 41 45; 1 10];

xrois = mergesigroi(rois,0)
```

```
xrois = 3×2

     1    10
    17    32
    36    40
```

Input Arguments

`roilims` — Region-of-interest limits

two-column matrix of positive integers

Region-of-interest limits, specified as a two-column matrix of positive integers. The *i*th row of `roilims` contains nondecreasing indices corresponding to the beginning and end samples of the *i*th region of interest of a signal.

Example: `[5 8; 12 20; 18 25]` specifies a two-column region-of-interest matrix with three regions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`s` — Maximum separation between regions of interest to merge

nonnegative integer

Maximum separation between regions of interest to merge, specified as a nonnegative integer.

If you specify `s` as `0`, `mergesigroi` merges contiguous, overlapping, or repeated regions specified in `roilims`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`roilimsout` — Modified region-of-interest limits

two-column matrix of positive integers

Modified region-of-interest limits, returned as a two-column matrix of positive integers. Output limits are returned in sorted order using the `sortrows` function.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Objects

`signalMask`

Functions

`binmask2sigroi` | `extendsigroi` | `extractsigroi` | `removesigroi` | `shortensigroi` | `sigroi2binmask`

Introduced in R2020b

plotsigroi

Plot signal regions based on signal mask

Syntax

```
plotsigroi(msk,x)
plotsigroi(msk,x,patchflag)

h = plotsigroi(msk,x)
```

Description

`plotsigroi(msk,x)` plots signal `x` with color-coded regions based on the source and properties in `msk`. If `x` is complex-valued, the function plots its magnitude.

`plotsigroi(msk,x,patchflag)` plots regions of interest using rectangular patches if `patchflag` is `true`.

`h = plotsigroi(msk,x)` returns the figure handle of the color-coded plot. You can use the figure handle to query and modify figure properties.

Examples

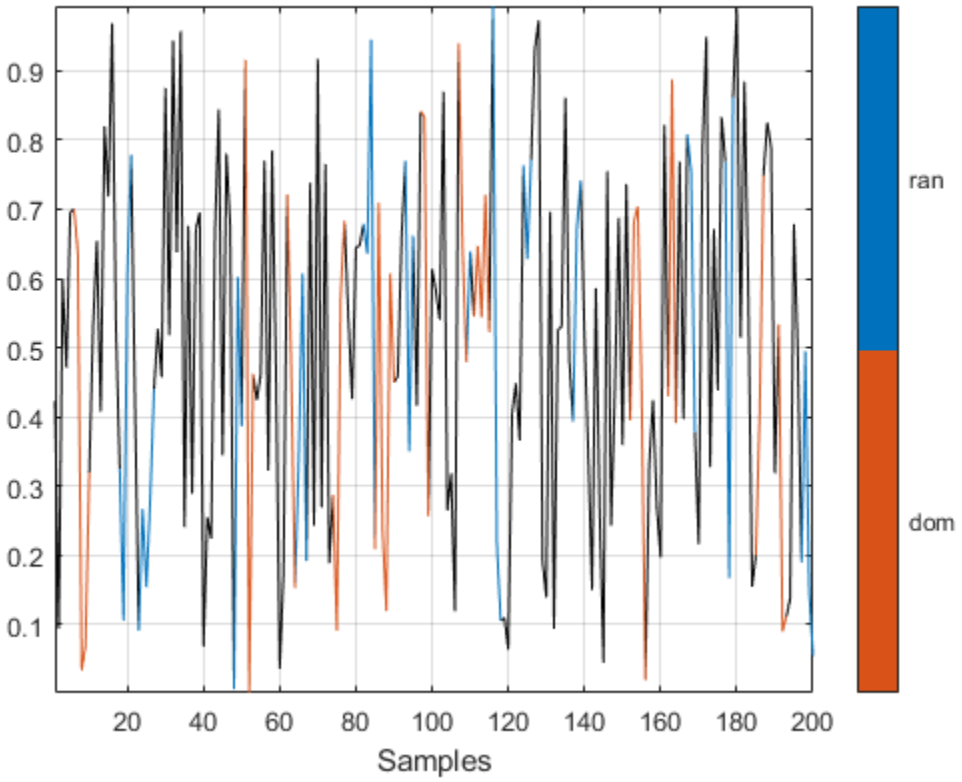
Plot Regions of Interest from Binary Sequences Mask

Consider a mask of binary sequences for two categories, `ran` and `dom`. Use the sequences to generate a `signalMask` object. Discard regions with fewer than 3 samples.

```
rng default

sq = randi(2,200,2)-1;

m = signalMask(sq,"MinLength",3,"Categories",["ran" "dom"]);
```

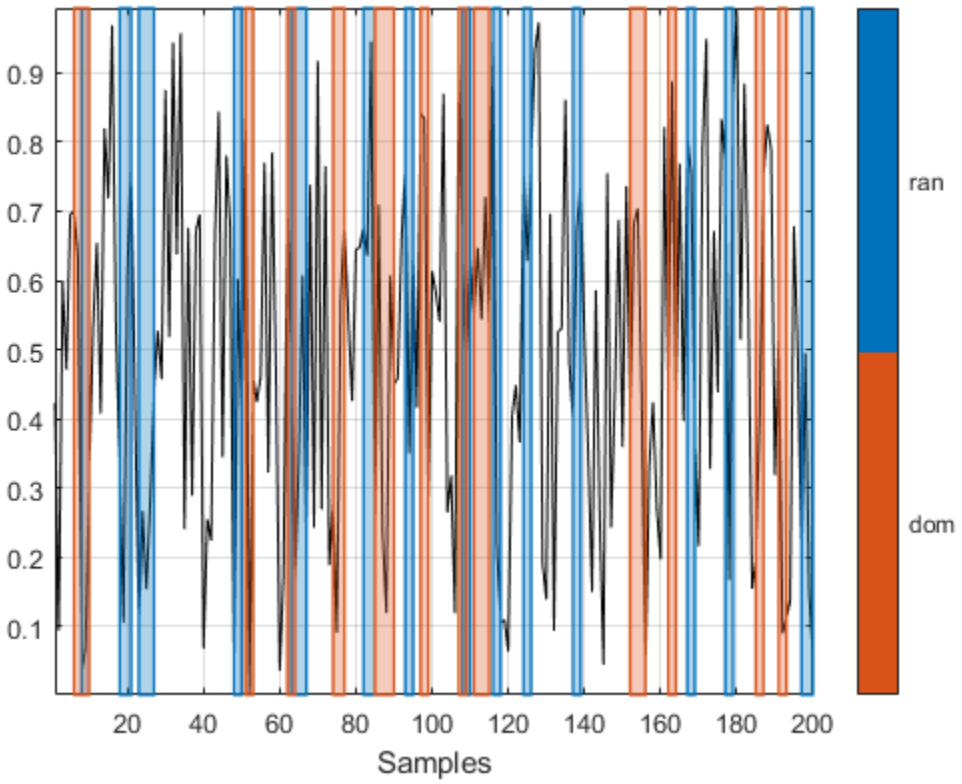
Generate a sequence of 200 random numbers. Plot the regions of interest.

```
x = rand(200,1);

plotsigroi(m,x)
```



Plot the regions of interest using rectangular patches.

```
plotsigroi(m,x,true)
```



Input Arguments

`msk` — Signal mask

`signalMask` object

Signal mask, specified as a `signalMask` object.

Example: `signalMask(table([2 4;6 7],["male" "female"]'))` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Example: `signalMask(categorical(["" "male" "male" "male" "" "female" "female" ""]',["male" "female"]))` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Example: `signalMask([0 1 1 1 0 0 0 0;0 0 0 0 0 1 1 0]','Categories',["male" "female"])` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

`x` — Input signal

vector

Input signal, specified as a vector.

Example: `chirp(0:1/1e3:1,25,1,50)` specifies a chirp sampled at 1 kHz.

Data Types: `single` | `double`

`patchflag` — Rectangular patch option

`false` (default) | `true`

Rectangular patch option, specified as a logical value. `plotsigroi` uses rectangular patches to plot regions of interest if `patchflag` is `true`.

Data Types: `logical`

Output Arguments

h — Figure handle

integer scalar

Figure handle, returned as an integer scalar.

See Also

Apps
Signal Labeler

Objects
`labeledSignalSet` | `signalMask` | `signalLabelDefinition`

Topics
“Label Definitions for Whale Songs” on page 1-1203
“Automate Signal Labeling with Custom Functions”
“Label Spoken Words in Audio Signals”
“Label Signal Attributes, Regions of Interest, and Points”
“Examine Labeled Signal Set”

Introduced in R2020b

removesigroi

Remove signal regions of interest

Syntax

```
roilimsout = removesigroi(roilims,s)
```

Description

`roilimsout = removesigroi(roilims,s)` removes signal regions of interest specified in `roilims` that have a length of `s` samples or less.

Examples

Remove Regions of Interest

Create a two-column matrix of integers that can represent regions of interest of a signal. Remove any regions that are six samples in length or shorter.

```
rois = [1 6; 17 26; 24 32; 36 40];

xrois = removesigroi(rois,6)
```

```
xrois = 2×2

    17    26
    24    32
```

Input Arguments

`roilims` — Region-of-interest limits

two-column matrix of positive integers

Region-of-interest limits, specified as a two-column matrix of positive integers. The *i*th row of `roilims` contains nondecreasing indices corresponding to the beginning and end samples of the *i*th region of interest of a signal.

Example: `[5 8; 12 20; 18 25]` specifies a two-column region-of-interest matrix with three regions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`s` — Maximum length of regions of interest to remove

nonnegative integer

Maximum length of regions of interest to remove, specified as a nonnegative integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`roilimsout` — Modified region-of-interest limits

two-column matrix of positive integers

Modified region-of-interest limits, returned as a two-column matrix of positive integers. Output limits are returned in sorted order using the `sortrows` function.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Objects

`signalMask`

Functions

`binmask2sigroi` | `extendsigroi` | `extractsigroi` | `mergesigroi` | `shortensigroi` | `sigroi2binmask`

Introduced in R2020b

roimask

Get ROI table mask

Syntax

```
tbl = roimask(msk)

[tbl,numroi,cats] = roimask(msk)
```

Description

`tbl = roimask(msk)` returns a region-of-interest (ROI) table mask, `tbl`, based on the source and properties in `msk`.

`[tbl,numroi,cats] = roimask(msk)` also returns `numroi`, a vector containing the number of regions found for each of the categories listed in `cats`.

Examples

ROI Table Mask from Binary Sequences

Generate an 18-by-2 mask of binary sequences. Use the mask to create a `signalMask` object and label the categories as `A` and `B`. Extract an ROI table mask from the object.

```
binSeqs = logical([ ...
    0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 1;
    1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0]');

m = signalMask(binSeqs,'Categories',["A" "B"]);

roiTbl = roimask(m)
```

```
roiTbl=4×2 table
    ROILimits    Value
    _________    _____

     1     2       B
     5     7       A
    10    12       B
    15    18       A
```

Extend the regions of interest by one sample to the left and right.

```
m.RightExtension = 1;
m.LeftExtension = 1;

roiTbl = roimask(m)
```

```
roiTbl=4×2 table
    ROILimits    Value
```

```
    _________    _____

     1     3       B
     4     8       A
     9    13       B
    14    19       A
```

ROI Table Mask from Categorical Sequence

Generate a 16-sample categorical sequence mask with two categories, `A` and `B`. Specify as `missing` those samples that do not belong to `A` or `B`. Use the mask to create a `signalMask` object. Extract an ROI table mask from the object.

```
catSeq = categorical(["A" "A" "A" missing missing "B" "B" ...
  missing missing "B" "B" missing "B" "A" "A" "A"]);

m = signalMask(catSeq);

roiTbl = roimask(m)
```

```
roiTbl=5×2 table
    ROILimits    Value
    _________    _____

     1     3       A
     6     7       B
    10    11       B
    13    13       B
    14    16       A
```

Output a list of categories and the number of regions belonging to each category.

```
[~,nroi,cats] = roimask(m)
```

```
nroi = 2×1

     2
     3


cats = 2x1 string
    "A"
    "B"
```

Merge same-category regions separated by only one sample.

```
m.MergeDistance = 1;

roiTbl = roimask(m)
```

```
roiTbl=4×2 table
    ROILimits    Value
    _________    _____
```

```
     1     3       A
     6     7       B
    10    13       B
    14    16       A
```

Input Arguments

`msk` — Signal mask

`signalMask` object

Signal mask, specified as a `signalMask` object.

Example: `signalMask(table([2 4;6 7],["male" "female"]'))` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Example: `signalMask(categorical(["" "male" "male" "male" "" "female" "female" ""]',["male" "female"]))` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Example: `signalMask([0 1 1 1 0 0 0 0;0 0 0 0 0 1 1 0]','Categories',["male" "female"])` specifies a signal mask with a three-sample `male` region and a two-sample `female` region.

Output Arguments

`tbl` — ROI table mask

table

ROI table mask, returned as a table.

- If `SampleRate` is specified, the region limits in `tbl` are expressed in seconds.
- If `RightExtension` is greater than zero and `SourceType` is specified as `'categoricalSequence'` or `'binarySequences'`, the region limits in `tbl` may go beyond the length of the sequence.

`numroi` — Number of regions

vector of integers

Number of regions found for each of the categories in `cats`, returned as a vector of integers.

`cats` — Category list

vector of strings

Category list, returned as a vector of strings.

See Also

Apps
Signal Labeler

Objects
`labeledSignalSet` | `signalMask` | `signalLabelDefinition`

Topics
“Label Definitions for Whale Songs” on page 1-1203
“Automate Signal Labeling with Custom Functions”
“Label Spoken Words in Audio Signals”
“Label Signal Attributes, Regions of Interest, and Points”
“Examine Labeled Signal Set”

Introduced in R2020b

shortensigroi

Shorten signal regions of interest from left and right

Syntax

```
roilimsout = shortensigroi(roilims,sl,sr)
```

Description

`roilimsout = shortensigroi(roilims,sl,sr)` shortens the signal regions of interest specified in `roilims` from the left by `sl` samples and from the right by `sr` samples. The function removes all regions of length `sl + sr` or less.

Examples

Shorten Regions of Interest

Create a two-column matrix of integers that can represent regions of interest of a signal. Shorten the regions of interest by three samples from the left and two samples from the right. `shortensigroi` removes regions of interest that are shortened by more than their length.

```
rois = [1 10; 17 26; 24 32; 38 40];

xrois = shortensigroi(rois,3,2)
```

```
xrois = 3×2

     4     8
    20    24
    27    30
```

Input Arguments

`roilims` — Region-of-interest limits

two-column matrix of positive integers

Region-of-interest limits, specified as a two-column matrix of positive integers. The *i*th row of `roilims` contains nondecreasing indices corresponding to the beginning and end samples of the *i*th region of interest of a signal.

Example: `[5 8; 12 20; 18 25]` specifies a two-column region-of-interest matrix with three regions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sl` — Number of samples to shorten from the left

nonnegative integer

Number of samples to shorten from the left, specified as a nonnegative integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sr` — Number of samples to shorten from the right
nonnegative integer

Number of samples to shorten from the right, specified as a nonnegative integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`roilimsout` — Modified region-of-interest limits
two-column matrix of positive integers

Modified region-of-interest limits, returned as a two-column matrix of positive integers. Output limits are returned in sorted order using the `sortrows` function.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Objects
`signalMask`

Functions
`binmask2sigroi` | `extendsigroi` | `extractsigroi` | `mergesigroi` | `removesigroi` | `sigroi2binmask`

Introduced in R2020b

sigroi2binmask

Convert matrix of ROI limits to binary mask

Syntax

```
mask = sigroi2binmask(roilims)
mask = sigroi2binmask(roilims,len)
```

Description

`mask = sigroi2binmask(roilims)` converts `roilims`, a matrix of signal region-of-interest (ROI) limits, to a binary sequence, `mask`, with `true` values indicating samples that belong to regions of interest.

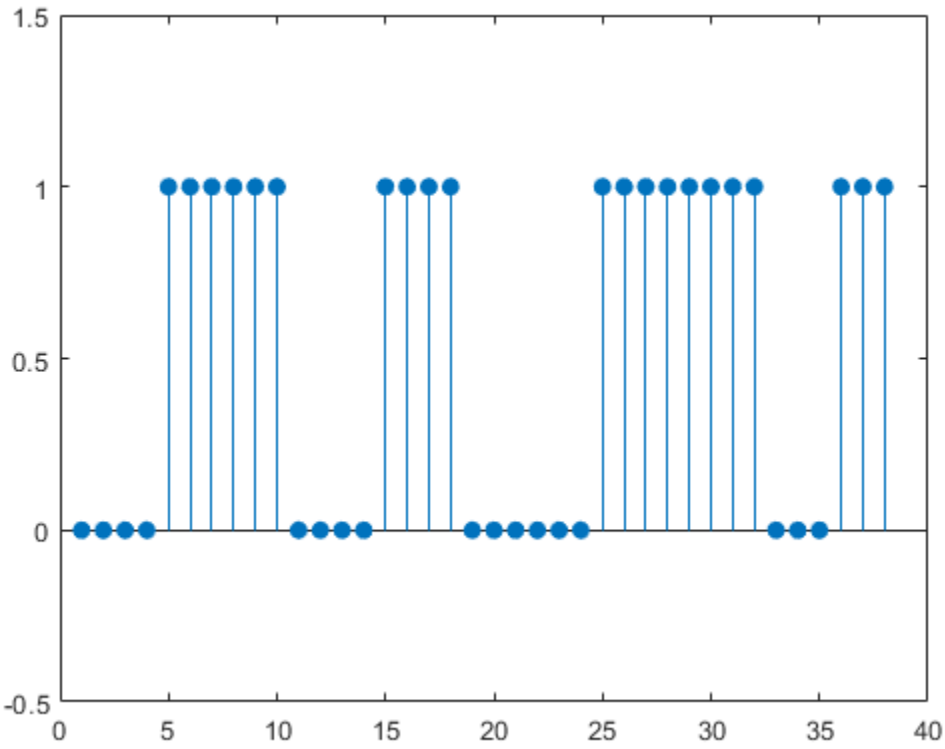
`mask = sigroi2binmask(roilims,len)` specifies the length of the output binary sequence.

Examples

Convert ROI Limits to Binary Sequence

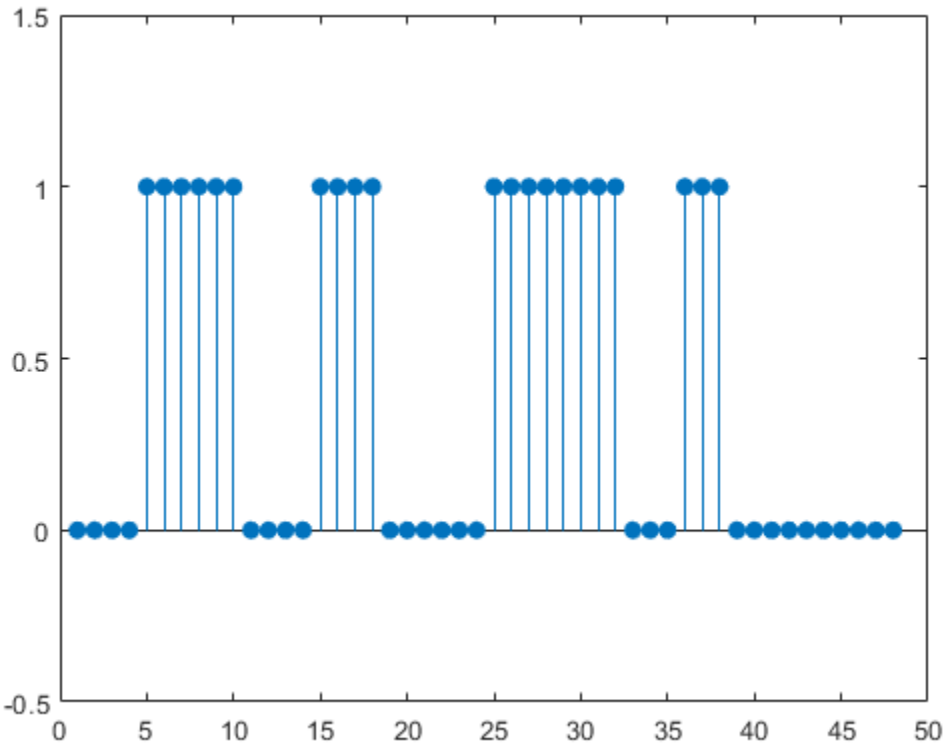
Consider a two-column matrix of beginning and end samples of four possible regions of interest of a signal. Convert the ROI limits to a logical sequence and display the sequence.

```
roilims = [5 10; 15 18; 25 32; 36 38];

mask = sigroi2binmask(roilims);

stem(mask,'filled')
ylim([0 2]-0.5)
```



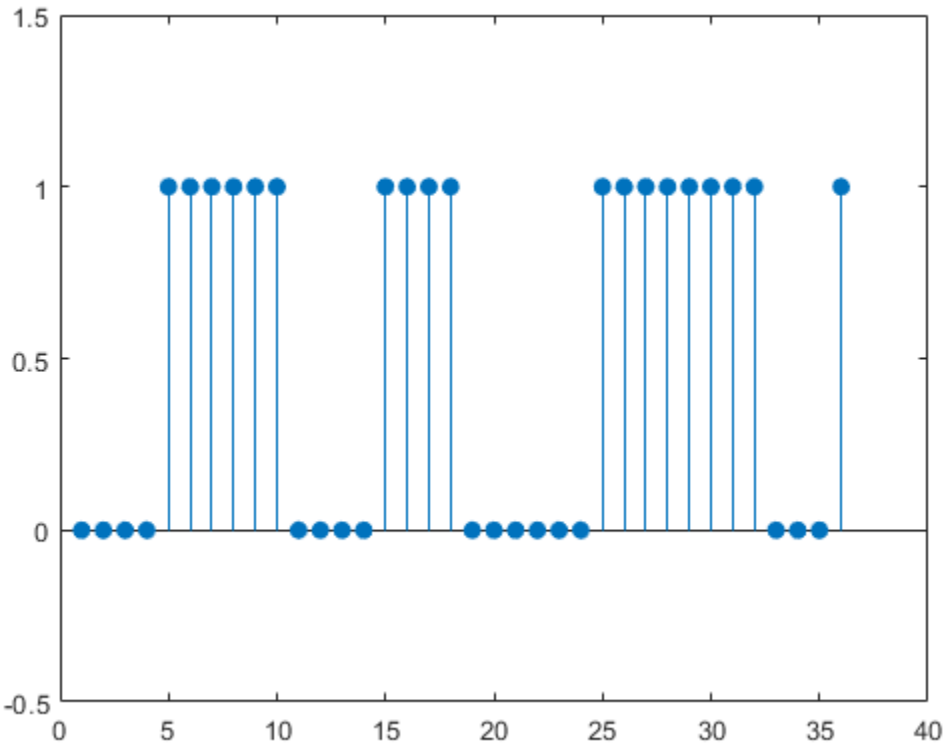
Specify the length of the output sequence as 48. `sigroi2binmask` pads the sequence with false values.

```
mask = sigroi2binmask(roilims,48);

stem(mask,'filled')
ylim([0 2]-0.5)
```



Specify the length of the output sequence as 36. `sigroi2binmask` ignores samples beyond the specified sequence length.

```
mask = sigroi2binmask(roilims,36);

stem(mask,'filled')
ylim([0 2]-0.5)
```



Input Arguments

`roilims` — Region-of-interest limits

two-column matrix of positive integers

Region-of-interest limits, specified as a two-column matrix of positive integers. The *i*th row of `roilims` contains nondecreasing indices corresponding to the beginning and end samples of the *i*th region of interest of a signal.

Example: `[5 8; 12 20; 18 25]` specifies a two-column region-of-interest matrix with three regions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`len` — Output sequence length

`max(roilims(:,2))` (default) | integer scalar

Output sequence length, specified as an integer scalar. Regions with indices larger than `len` are ignored or truncated. If `len` is greater than `max(roilims(:,2))`, then `sigroi2binmask` pads `mask` with `false` values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`mask` — Binary mask

logical vector

Binary mask, returned as a logical vector with `true` values indicating samples that belong to a region of interest.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Objects

`signalMask`

Functions

`binmask2sigroi` | `extendsigroi` | `extractsigroi` | `mergesigroi` | `removesigroi` | `shortensigroi`

Introduced in R2020b

signalFrequencyFeatureExtractor

Streamline signal frequency feature extraction

Description

Use `signalFrequencyFeatureExtractor` to extract frequency-domain features from a signal. You can use the extracted features to train a machine learning model or a deep learning network.

Creation

Syntax

```
sFE = signalFrequencyFeatureExtractor
sFE = signalFrequencyFeatureExtractor(Name,Value)
```

Description

`sFE = signalFrequencyFeatureExtractor` creates a `signalFrequencyFeatureExtractor` object with default property values.

`sFE = signalFrequencyFeatureExtractor(Name,Value)` specifies nondefault property values of the `signalFrequencyFeatureExtractor` object. For example, `signalFrequencyFeatureExtractor(FrameSize=30,FrameOverlapLength=6)` divides the signal into overlapping 30-sample frames and extracts features from each frame.

Properties

Main Properties

`FrameSize` — Number of samples in a frame
positive integer

Number of samples in a frame, specified as a positive integer. The object divides the signal into frames of the specified length and extracts features for each frame. If you do not specify `FrameSize`, or if you specify `FrameSize` as empty, the object extracts features for the whole signal.

Data Types: `single` | `double`

`FrameRate` — Number of samples between start of frames
positive integer

Number of samples between the start of frames, specified as a positive integer. The frame rate determines the distance in samples between the starting points of frames. If you specify `FrameRate`, then you must also specify `FrameSize`. If you do not specify `FrameRate` or `FrameOverlapLength`, then `FrameRate` is assumed to be equal to `FrameSize`. You cannot specify `FrameRate` and `FrameOverlapLength` simultaneously.

Data Types: `single` | `double`

`FrameOverlapLength` — Number of overlapping samples between consecutive frames
positive integer

Number of overlapping samples between consecutive frames, specified as a positive integer. `FrameOverlapLength` must be less than or equal to the frame size. If you specify `FrameOverlapLength`, then you must also specify `FrameSize`. You cannot specify `FrameOverlapLength` and `FrameRate` simultaneously.

Data Types: `single` | `double`

`SampleRate` — Sample rate
`[]` (default) | positive scalar

Input sample rate, specified as a positive scalar in hertz.

Data Types: `single` | `double`

`FeatureFormat` — Format of generated features
`"matrix"` (default) | `"table"`

Format of generated features, specified as `"matrix"` or `"table"`. The `extract` function returns features as one of these:

- `matrix` — Matrix with columns corresponding to feature values.
- `table` — Table with each table variable corresponding to a feature value.

Data Types: `char` | `string`

`IncompleteFrameRule` — Rule to handle incomplete frames
`"drop"` (default) | `"zeropad"`

Rule to handle incomplete frames, specified as `"drop"` or `"zeropad"`. This rule applies when the current frame size is less than the specified `FrameSize` property.

- `drop` — Drop the incomplete frame and do not use it to compute features.
- `zeropad` — Zero-pad the incomplete frame and use it to compute features.

Data Types: `char` | `string`

Features to Extract

`MeanFrequency` — Option to extract mean frequency
`false` (default) | `true`

Option to extract the mean frequency of the power spectrum, specified as `true` or `false`. If you specify `MeanFrequency` as true, the object extracts the mean frequency of the power spectrum and appends it to the features returned by the `extract` function.

Data Types: `logical`

`MedianFrequency` — Option to extract median frequency
`false` (default) | `true`

Option to extract the median frequency of the power spectrum, specified as `true` or `false`. If you specify `MedianFrequency` as true, the object extracts the median frequency of the power spectrum and appends it to the features returned by the `extract` function.

Data Types: `logical`

`BandPower` — Option to extract average band power
`false` (default) | `true`

Option to extract the average band power, specified as `true` or `false`. If you specify `BandPower` as true, the object extracts the band power and appends it to the features returned by the `extract` function.

Data Types: `logical`

`OccupiedBandwidth` — Option to extract occupied bandwidth
`false` (default) | `true`

Option to extract the 99% occupied bandwidth, specified as `true` or `false`. If you specify `OccupiedBandwidth` as true, the object extracts the 99% occupied bandwidth and appends it to the features returned by the `extract` function.

To set parameters of the occupied bandwidth extraction, use `setExtractorParameters`.

```
setExtractorParameters(sFE,"OccupiedBandwidth",Name=Value)
```

Settable parameters for the occupied bandwidth extraction are:

- `Percentage` — Power percentage, specified as a positive integer between `0` and `100`.

Data Types: `logical`

`PowerBandwidth` — Option to extract half-power bandwidth
`false` (default) | `true`

Option to extract the 3 dB (half-power) bandwidth, specified as `true` or `false`. If you specify `PowerBanwidth` as true, the object extracts the 3 dB bandwidth value and appends it to the features returned by the `extract` function.

To set parameters of the half-power bandwidth extraction, use `setExtractorParameters`.

```
setExtractorParameters(sFE,"PowerBandwidth",Name=Value)
```

Settable parameters for the half-power bandwidth extraction are:

- `RelativeAmplitude` — Relative amplitude, specified as an integer.

Data Types: `logical`

`WelchPSD` — Option to extract power spectral density estimate
`false` (default) | `true`

Option to extract the power spectral density (PSD) estimate, specified as `true` or `false`. If you specify `WelchPSD` as true, the object extracts the PSD estimate using Welch's method and appends it to the features returned by the `extract` function.

To set parameters of the Welch's PSD estimate, use `setExtractorParameters`.

```
setExtractorParameters(sFE,"WelchPSD",Name=Value)
```

Settable parameters for the Welch's PSD estimate extraction are:

- `FFTLength` — Number of DFT points, specified as a positive integer.
- `FrequencyVector` — Frequencies at which the PSD is estimated, specified as a vector with at least two elements. You can specify `FrequencyVector` only when `FFTLength` is not specified.
- `OverlapLength` — Number of overlapping samples, specified as a positive integer.
- `Window` — Window, specified as a scalar or vector.

Data Types: `logical`

`PeakAmplitude` — Option to extract peak amplitude
`false` (default) | `true`

Option to extract the peak spectral amplitudes, specified as `true` or `false`. If you specify `PeakAmplitude` as true, the object extracts the peak amplitudes of the computed Welch PSD estimate and appends them to the features returned by the `extract` function.

To set parameters of the peak amplitude extraction, use `setExtractorParameters`.

```
setExtractorParameters(sFE,"PeakAmplitude",Name=Value)
```

Settable parameters for the peak amplitude extraction are:

- `PeakType` — Type of peak, specified as `"minima"` or `"maxima"`.
- `MaxNumExtrema` — Maximum number of peaks, specified as a positive integer scalar.
- `MinProminence` — Minimum prominence, specified as a positive scalar. The object returns only peaks whose prominence is at least the value specified.
- `MinSeparation` — Minimum separation between peaks, specified as a positive scalar.
- `FlatSelection` — Flat region indicator, specified as one of these:
  - `"center"` — Indicate only the center element of a flat region as the peak.
  - `"first"` — Indicate only the first element of a flat region as the peak.
  - `"last"` — Indicate only the last element of a flat region as the peak.
  - `"all"` — Indicate all elements of a flat region as the peak.

Data Types: `logical`

`PeakLocation` — Option to extract peak location
`false` (default) | `true`

Option to extract the spectral peak locations, specified as `true` or `false`. If you specify `PeakLocation` as true, the object extracts the peak locations of the computed Welch PSD estimate and appends them to the features returned by the `extract` function.

To set parameters of the peak location extraction, use `setExtractorParameters`.

```
setExtractorParameters(sFE,"PeakLocation",Name=Value)
```

Settable parameters for the peak location extraction are:

- `PeakType` — Type of peak, specified as `"minima"` or `"maxima"`.
- `MaxNumExtrema` — Maximum number of peaks, specified as a positive integer scalar.

- `MinProminence` — Minimum prominence, specified as a positive scalar. The `setExtractorParameters` function returns only peaks whose prominence is at least the value specified.
- `MinSeparation` — Minimum separation between peaks, specified as a positive scalar.
- `FlatSelection` — Flat region indicator, specified as one of these:
  - `"center"` — Indicate only the center element of a flat region as the peak.
  - `"first"` — Indicate only the first element of a flat region as the peak.
  - `"last"` — Indicate only the last element of a flat region as the peak.
  - `"all"` — Indicate all elements of a flat region as the peak.

Data Types: `logical`

Note To compute frequency features, `signalFrequencyFeatureExtractor` first estimates the PSD of the input time-domain signal using Welch's method. The object uses the computed Welch PSD and corresponding frequency vector to compute the specified features. You can configure the computed Welch PSD estimate using the `setExtractorParameters` function.

Object Functions

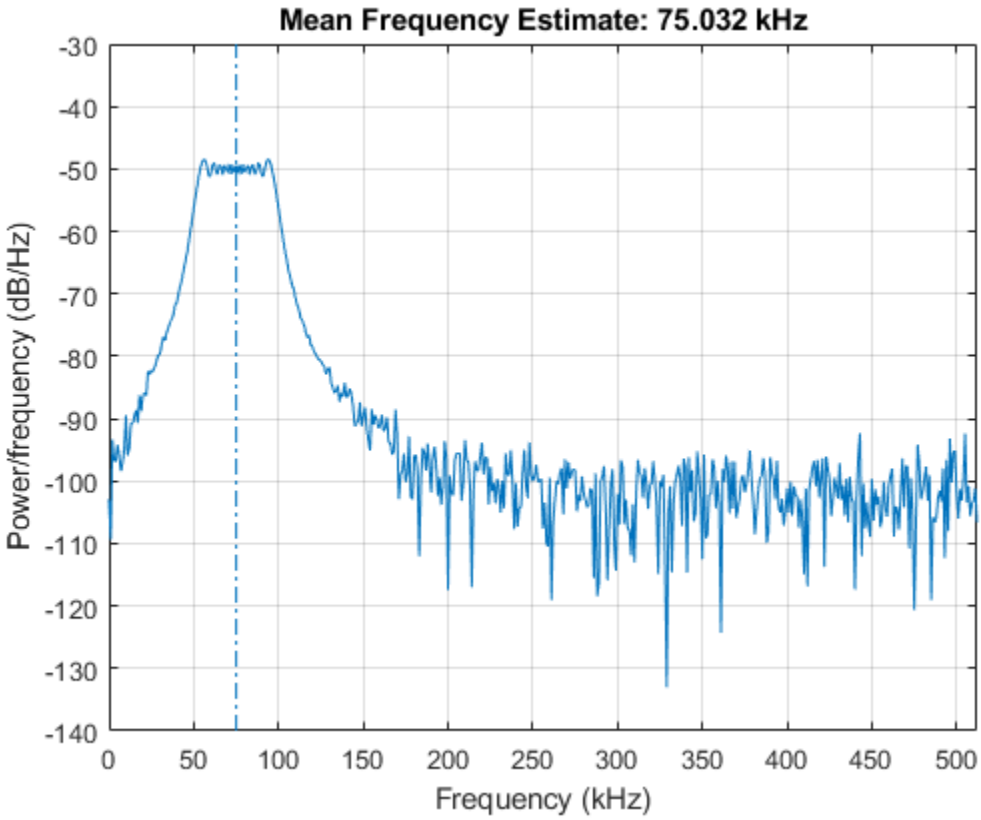
extract	Extract time-domain or frequency-domain features
generateMATLABFunction	Create MATLAB function compatible with C/C++ code generation
getExtractorParameters	Get current parameter values of feature extractor object
setExtractorParameters	Set nondefault values for feature extractor object

Examples

Extract Frequency-Domain Features From Signal

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. Plot the power spectral density (PSD) and annotate the mean frequency.

```
nSamp = 1024;
Fs = 1024e3;
SNR = 40;

t = (0:nSamp-1)'/Fs;

x = chirp(t,50e3,nSamp/Fs,100e3);
x = x+randn(size(x))*std(x)/db2mag(SNR);
meanfreq(x,Fs)
```




```
ans = 7.5032e+04
```

Create a `signalFrequencyFeatureExtractor` object to extract the mean frequency, 99% occupied bandwidth, and 3 dB bandwidth of the signal.

```
sFE = signalFrequencyFeatureExtractor(SampleRate=Fs,MeanFrequency=true,OccupiedBandwidth=true,Pow
```

```
sFE =
  signalFrequencyFeatureExtractor with properties:

   Properties
              FrameSize: []
              FrameRate: []
             SampleRate: 1024000
    IncompleteFrameRule: "drop"
          FeatureFormat: "matrix"

   Enabled Features
     MeanFrequency, OccupiedBandwidth, PowerBandwidth

   Disabled Features
     MedianFrequency, BandPower, WelchPSD, PeakAmplitude, PeakLocation
```

Call the `extract` function to extract the specified features.

```
[features,info] = extract(sFE,x)

features = 1×3
10^4 ×

    7.2252    4.3783    3.7773


info = struct with fields:
        MeanFrequency: 1
    OccupiedBandwidth: 2
       PowerBandwidth: 3
```

To view the extracted features in a table, modify the `FeatureFormat` property of the object.

```
sFE.FeatureFormat = "table";
features = extract(sFE,x)

features=1×5 table
    FrameStartTime    FrameEndTime    MeanFrequency    OccupiedBandwidth    PowerBandwidth
    ______________    ____________    _____________    _________________    ______________

          1               1024            72252              43783              37773
```

You can use the `getExtractorParameters` function to view parameters used to compute a specified feature. The occupied bandwidth measures the bandwidth containing 99% of the total power for the input signal by default. Use the `setExtractorParameters` function to change the percentage to 95% and extract the specified features again.

```
params = getExtractorParameters(sFE,'OccupiedBandwidth')

params = struct with fields:
    Percentage: []


params.Percentage = 95;
setExtractorParameters(sFE,'OccupiedBandwidth',params)
features2 = extract(sFE,x)

features2=1×5 table
    FrameStartTime    FrameEndTime    MeanFrequency    OccupiedBandwidth    PowerBandwidth
    ______________    ____________    _____________    _________________    ______________

          1               1024            72252              39840              37773
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`islocalmax` | `islocalmin` | `obw` | `powerbw` | `pwelch`

Objects
`signalTimeFeatureExtractor`

Introduced in R2021b

signalTimeFeatureExtractor

Streamline signal time feature extraction

Description

Use `signalTimeFeatureExtractor` to extract time-domain features from a signal. You can use the extracted features to train a machine learning model or a deep learning network.

Creation

Syntax

```
sFE = signalTimeFeatureExtractor
sFE = signalTimeFeatureExtractor(Name,Value)
```

Description

`sFE = signalTimeFeatureExtractor` creates a `signalTimeFeatureExtractor` object with default property values.

`sFE = signalTimeFeatureExtractor(Name,Value)` specifies nondefault property values of the `signalTimeFeatureExtractor` object. For example, `signalTimeFeatureExtractor(FeatureFormat="table")` sets the output format of the generated features to a table.

Properties

Main Properties

`FrameSize` — Number of samples in a frame
positive integer

Number of samples in a frame, specified as a positive integer. The object divides the signal into frames of the specified length and extracts features for each frame. If you do not specify `FrameSize`, or if you specify `FrameSize` as empty, the object extracts features for the whole signal.

Data Types: `single` | `double`

`FrameRate` — Number of samples between start of frames
positive integer

Number of samples between the start of frames, specified as a positive integer. The frame rate determines the distance in samples between the starting points of frames. If you specify `FrameRate`, then you must also specify `FrameSize`. If you do not specify `FrameRate` or `FrameOverlapLength`, then `FrameRate` is assumed to be equal to `FrameSize`. You cannot specify `FrameRate` and `FrameOverlapLength` simultaneously.

Data Types: `single` | `double`

`FrameOverlapLength` — Number of overlapping samples between consecutive frames
positive integer

Number of overlapping samples between consecutive frames, specified as a positive integer. `FrameOverlapLength` must be less than or equal to the frame size. If you specify `FrameOverlapLength`, then you must also specify `FrameSize`. You cannot specify `FrameOverlapLength` and `FrameRate` simultaneously.

Data Types: `single` | `double`

`SampleRate` — Sample rate
`[]` (default) | positive scalar

Input sample rate, specified as a positive scalar in hertz.

Data Types: `single` | `double`

`FeatureFormat` — Format of generated features
`"matrix"` (default) | `"table"`

Format of generated features, specified as `"matrix"` or `"table"`. The `extract` function returns features as one of these:

- `matrix` — Matrix with columns corresponding to feature values.
- `table` — Table with each table variable corresponding to a feature value.

Data Types: `char` | `string`

`IncompleteFrameRule` — Rule to handle incomplete frames
`"drop"` (default) | `"zeropad"`

Rule to handle incomplete frames, specified as `"drop"` or `"zeropad"`. This rule applies when the current frame size is less than the specified `FrameSize` property.

- `drop` — Drop the incomplete frame and do not use it to compute features.
- `zeropad` — Zero-pad the incomplete frame and use it to compute features.

Data Types: `char` | `string`

Features to Extract

`Mean` — Option to extract mean
`false` (default) | `true`

Option to extract the mean, specified as `true` or `false`. If you specify `Mean` as true, the object extracts the mean and appends the value to the features returned by the `extract` function.

Data Types: `logical`

`RMS` — Option to extract root mean square
`false` (default) | `true`

Option to extract the root mean square (RMS), specified as `true` or `false`. If you specify `RMS` as true, the object extracts the RMS and appends the value to the features returned by the `extract` function.

Data Types: `logical`

`StandardDeviation` — Option to extract standard deviation
`false` (default) | `true`

Option to extract the standard deviation, specified as `true` or `false`. If you specify `StandardDeviation` as true, the object extracts the standard deviation and appends the value to the features returned by the `extract` function.

Data Types: `logical`

`ShapeFactor` — Option to extract shape factor
`false` (default) | `true`

Option to extract the shape factor, specified as `true` or `false`. The shape factor is equal to the RMS value divided by the mean absolute value of the signal. If you specify `ShapeFactor` as true, the object extracts the shape factor and appends the value to the features returned by the `extract` function.

Data Types: `logical`

`SNR` — Option to extract signal-to-noise ratio
`false` (default) | `true`

Option to extract the signal-to-noise ratio (SNR), specified as `true` or `false`. If you specify `SNR` as true, the object extracts the SNR and appends the value to the features returned by the `extract` function.

Data Types: `logical`

`THD` — Option to extract total harmonic distortion
`false` (default) | `true`

Option to extract the total harmonic distortion (THD), specified as `true` or `false`. If you specify `THD` as true, the object extracts the THD and appends the value to the features returned by the `extract` function.

Data Types: `logical`

`SINAD` — Option to extract signal to noise and distortion ratio
`false` (default) | `true`

Option to extract the signal to noise and distortion ratio (SINAD) in decibels, specified as `true` or `false`. If you specify `Sinad` as true, the object extracts the SINAD and appends the value to the features returned by the `extract` function.

Data Types: `logical`

`PeakValue` — Option to extract peak value
`false` (default) | `true`

Option to extract the peak value, specified as `true` or `false`. The peak value corresponds to the maximum absolute value of the signal. If you specify `PeakValue` as true, the object extracts the peak and appends the value to the features returned by the `extract` function.

Data Types: `logical`

`CrestFactor` — Option to extract crest factor
`false` (default) | `true`

Option to extract the crest factor, specified as `true` or `false`. The crest factor is equal to the peak value divided by the RMS. If you specify `CrestFactor` as true, the object extracts the crest factor and appends the value to the features returned by the `extract` function.

Data Types: `logical`

`ClearanceFactor` — Option to extract clearance factor
`false` (default) | `true`

Option to extract the clearance factor, specified as `true` or `false`. The clearance factor is equal to the peak value divided by the squared mean of the square roots of the absolute amplitude. If you specify `ClearanceFactor` as true, the object extracts the clearance factor and appends the value to the features returned by the `extract` function.

Data Types: `logical`

`ImpulseFactor` — Option to extract impulse factor
`false` (default) | `true`

Option to extract the impulse factor, specified as `true` or `false`. The impulse factor is equal to the peak value divided by the mean of the absolute amplitude. If you specify `ImpulseFactor` as true, the object extracts the impulse factor and appends the value to the features returned by the `extract` function.

Data Types: `logical`

Object Functions

extract	Extract time-domain or frequency-domain features
generateMATLABFunction	Create MATLAB function compatible with C/C++ code generation

Examples

Extract Time-Domain Features from Data Set

Extract time-domain features from electromyographic (EMG) data for later use in a machine learning workflow to classify forearm motions. The files are available at this location: https://ssd.mathworks.com/supportfiles/SPT/data/MyoelectricData.zip.

This example uses EMG signals collected from the forearms of 30 subjects [1]. The data set consists of 720 files. Each subject participated in four testing sessions, and performed six trials of different forearm motions per session. Download and unzip the files into your temporary directory.

```
localfile = matlab.internal.examples.downloadSupportFile("SPT","data/MyoelectricData.zip");
datasetFolder = fullfile(tempdir,"MyoelectricData");
unzip(localfile,datasetFolder)
```

Each file contains an eight-channel EMG signal that represents the activation of eight forearm muscles during a series of motions. The sample rate is 1000 Hz. Create a `signalDatastore` that points to the data set folder.

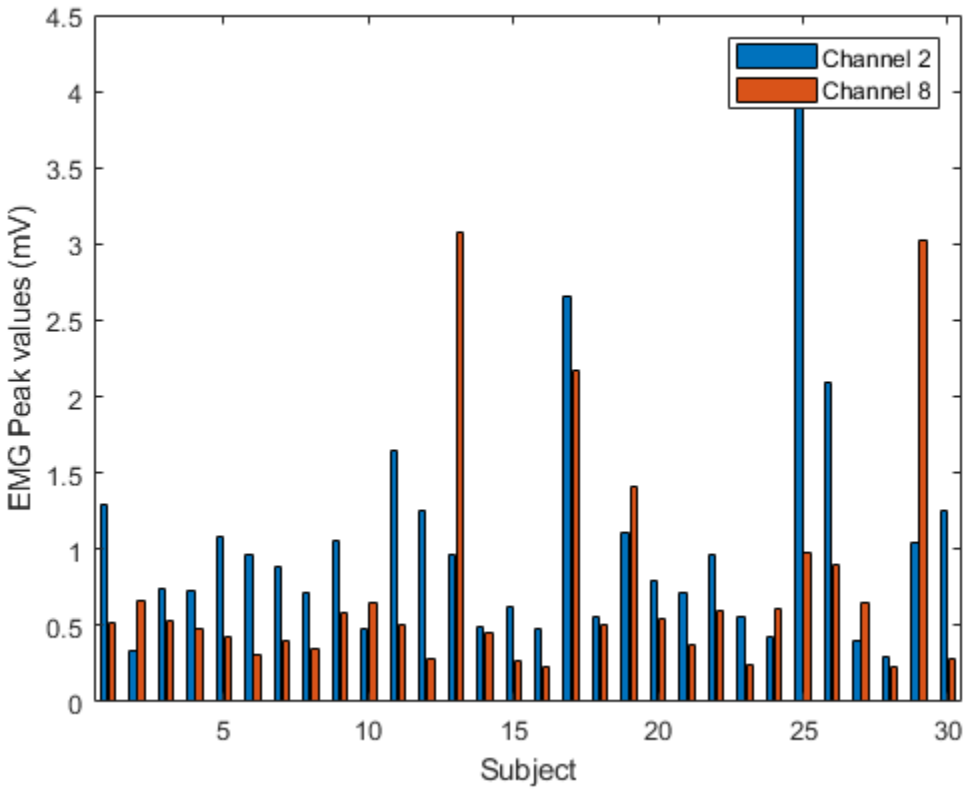
```
fs = 1000;
sds = signalDatastore(datasetFolder,IncludeSubfolders=true);
```

For this example, analyze only the last (sixth) trial of each session. Use the `endsWith` function to find the indices that correspond to these files. Create a new datastore that contains this subset of signals.

```
p = endsWith(sds.Files,"6d.mat");
sdssub = subset(sds,p);
data = readall(sdssub);
```

Create a `signalTimeFeatureExtractor` object to extract the mean, root mean square (RMS), and peak values from the EMG signals. Call the `extract` function to extract the specified features. Plot the peak values for the second and eighth EMG channels.

```
sFE = signalTimeFeatureExtractor(SampleRate=fs,Mean=true,RMS=true,PeakValue=true);

M = [];
F = [];

for i = 1:length(sdssub.Files)
    M{i,1} = extract(sFE,data{i});
end

for k = 1:30
    peak2 = M{k,1}(:,3,2);
    peak8 = M{k,1}(:,3,8);
    F(k,1) = peak2;
    F(k,2) = peak8;
end

bar(F)
xlabel("Subject")
ylabel("EMG Peak values (mV)")
legend(["Channel 2";"Channel 8"])
```



References

[1] Chan, Adrian D.C., and Geoffrey C. Green. 2007. "Myoelectric Control Development Toolbox." Paper presented at 30th Conference of the Canadian Medical & Biological Engineering Society, Toronto, Canada, 2007.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`signalFrequencyFeatureExtractor`

Introduced in R2021a

extract

Extract time-domain or frequency-domain features

Syntax

```
features = extract(sFE,x)
[features,info] = extract(sFE,x)
[features,info,framelimits] = extract(sFE,x)
```

Description

`features = extract(sFE,x)` returns a matrix or a table containing features extracted from input `x`. The output depends on the settings of the feature extractor object `sFE`.

`[features,info] = extract(sFE,x)` returns a structure `info` that maps a specific feature to its column location in the output feature matrix `features`. This syntax is valid only when you set the `FeatureFormat` property of the feature extractor object to `"matrix"`.

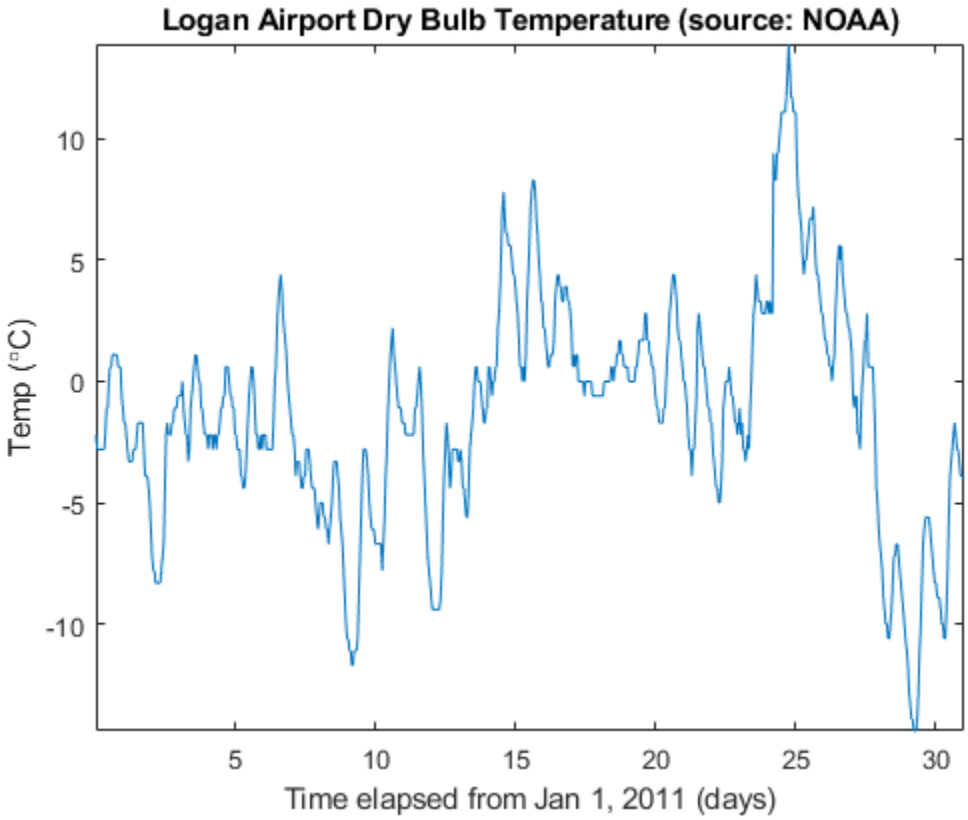
`[features,info,framelimits] = extract(sFE,x)` returns a matrix `framelimits` whose *i*-th row contains the beginning and end limits of the *i*-th frame. This syntax is valid only when you set the `FeatureFormat` property of the feature extractor object to `"matrix"`.

Examples

Extract Daily Peaks from Temperature Data

Load a set of temperature readings in Celsius taken every hour at Logan Airport in Boston for 31 days. Plot the data.

```
load bostemp
days = (1:31*24)/24;
plot(days, tempC)
axis tight
ylabel('Temp (\circC)')
xlabel('Time elapsed from Jan 1, 2011 (days)')
title('Logan Airport Dry Bulb Temperature (source: NOAA)')
```



Create a `signalTimeFeatureExtractor` object and enable the `PeakValue` feature. To obtain the maximum absolute temperature reading per day, set the frame size to `24` samples and the frame overlap to `0` samples.

```
sFE = signalTimeFeatureExtractor(FrameSize=24,FrameOverlapLength=0,PeakValue=true);
```

Call the `extract` function on the object to extract the daily absolute maximum temperatures in the data set.

```
peaktemps = extract(sFE,tempC)
```

```
peaktemps = 31×1

    2.8000
    6.1000
    8.3000
    3.3000
    2.8000
    4.4000
    4.4000
    6.1000
   10.6000
   11.7000
      ⋮
```

Confirm the extracted peak values. Divide the signal into 24-sample segments representing temperature readings per day and compute the maximum absolute value of each segment. Compare the resulting vector to `peaktemps`.

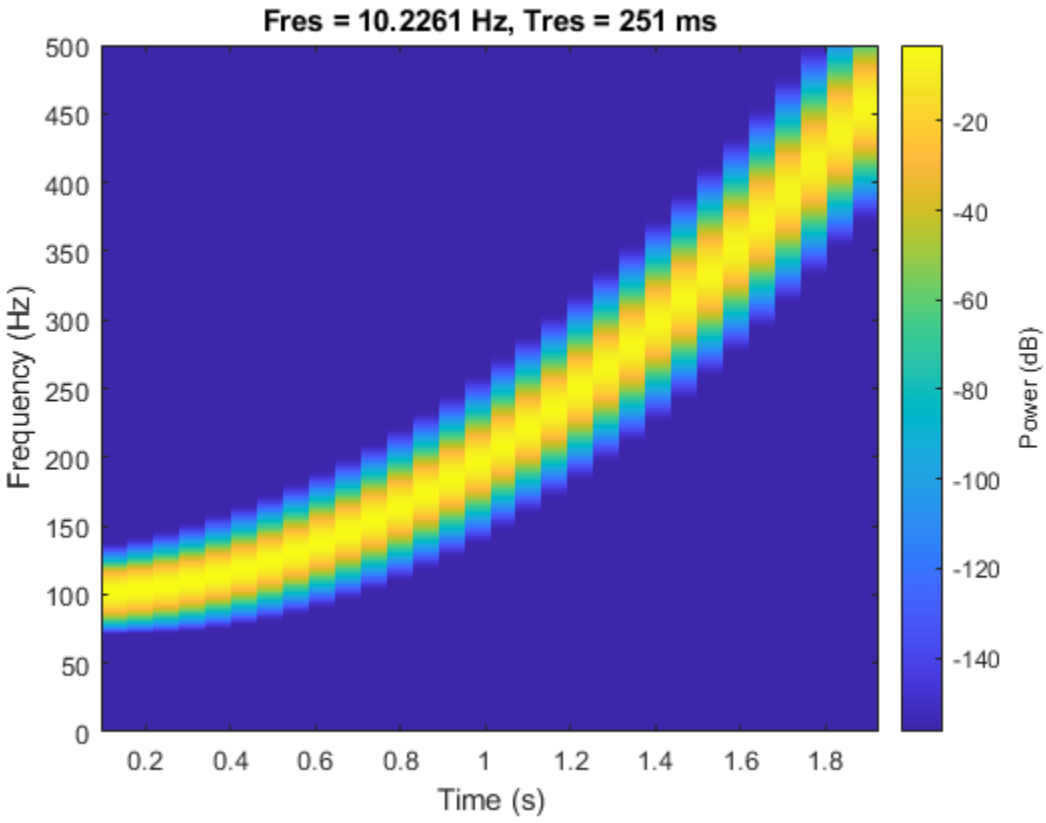
```
y = buffer(tempC,24);
[mx,idx] = max(abs(y));

tf = isequal(peaktemps,mx')
```

```
tf = logical
   1
```

Extract Features from Signal

Consider a quadratic chirp sampled at 1 kHz for 2 seconds. The chirp has an initial frequency of 100 Hz that increases to 200 Hz at $t = 1$ second. Compute and display the spectrogram.

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



Create a `signalFrequencyFeatureExtractor` object to obtain the mean and median frequencies from the signal. Specify the sample rate.

```
sFE = signalFrequencyFeatureExtractor(SampleRate=fs,MeanFrequency=true,MedianFrequency=true);
```

Extract the features. `info` returns the column index in `features` of each extracted feature.

```
[features,info] = extract(sFE,y)
```

```
features = 1×2

  226.0160  199.7034


info = struct with fields:
      MeanFrequency: 1
    MedianFrequency: 2
```

Set the `FrameSize` and `FrameRate` properties of the feature extractor object to divide the signal into two frames. The first frame represents the chirp oscillating at the initial frequency of 100 Hz and the second frame represents the chirp oscillating at 200 Hz. Extract the mean and median frequencies for each frame and include the frame limits in the output.

```
sFE.FrameSize = round(length(y)/2);
sFE.FrameRate = 1000;
[features,info,framelimits] = extract(sFE,y)
```

```
features = 2×2

  131.4921  124.9820
  331.2664  324.6992


info = struct with fields:
      MeanFrequency: 1
    MedianFrequency: 2


framelimits = 2×2

           1        1001
        1001        2001
```

Input Arguments

sFE — Feature extractor object

`signalFrequencyFeatureExtractor` object | `signalTimeFeatureExtractor` object

Feature extractor object, specified as a `signalFrequencyFeatureExtractor` object or a `signalTimeFeatureExtractor` object.

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix.

Data Types: `single` | `double`

Output Arguments

`features` — Extracted features

3-D array

Extracted features, returned as an *L*-by-*M*-by-*N* array:

- *L* — Number of frames
- *M* — Number of features extracted per frame
- *N* — Number of channels

When you set the `FeatureFormat` property of the input feature extractor object to `"table"`, the function returns the extracted features in a table with the frame limits listed in the first two table variables. When you set the `FeatureFormat` property of the input feature extractor object to `"matrix"`, the function returns the extracted features in a matrix.

`info` — Feature information

structure

Feature information, returned as a structure. The function maps each feature to its column location in the output matrix `features`. This argument applies only when you set the `FeatureFormat` property of the input feature extractor object to `"matrix"`.

`framelimits` — Frame limits

matrix

Frame limits, returned as a matrix. The *i*-th row in `framelimits` contains the beginning and end limits of the *i*-th frame. This argument applies only when you set the `FeatureFormat` property of the input feature extractor object to `"matrix"`.

See Also

Objects

`signalFrequencyFeatureExtractor` | `signalTimeFeatureExtractor`

Functions

`getExtractorParameters` | `setExtractorParameters`

Introduced in R2021b

generateMATLABFunction

Create MATLAB function compatible with C/C++ code generation

Syntax

```
generateMATLABFunction(sFE)
```

Description

`generateMATLABFunction(sFE)` generates code based on the input feature extractor object `sFE` and opens an untitled file containing the function `extractSignalFeatures`. The function signature depends on how you set the `FeatureFormat` property of the input feature extractor object.

- When you specify `FeatureFormat` as `"matrix"`, the generated MATLAB function has this signature:

  ```
  [features,info,framelimits] = extractSignalFeatures(x)
  ```

  The signature is equivalent to:

  ```
  [features,info,framelimits] = extract(sFE,x)
  ```

- When you specify `FeatureFormat` as `"table"`, the generated MATLAB function has this signature:

  ```
  features = extractSignalFeatures(x)
  ```

  The signature is equivalent to:

  ```
  features = extract(sFE,x)
  ```

Examples

Generate Equivalent MATLAB Function for Feature Extraction

Create a `signalTimeFeatureExtractor` object to extract the mean, standard deviation, and peak value of a random signal.

```
x = randn(1000,1);
sFE = signalTimeFeatureExtractor(FrameSize=100, ...
      FrameOverlapLength=10,Mean=true,StandardDeviation=true, ...
      PeakValue=true)
```

```
sFE =
   signalTimeFeatureExtractor with properties:

    Properties
              FrameSize: 100
      FrameOverlapLength: 10
              SampleRate: []
     IncompleteFrameRule: "drop"
           FeatureFormat: "matrix"
```

```
    Enabled Features
      Mean, StandardDeviation, PeakValue

    Disabled Features
      RMS, ShapeFactor, SNR, THD, SINAD, CrestFactor
      ClearanceFactor, ImpulseFactor
```

Call `generateMATLABFunction` on the object. The generated function `extractSignalFeatures` is equivalent to calling the `extract` function on `sFE`. Save the function to your current folder and view the function script.

```
generateMATLABFunction(sFE)
type extractSignalFeatures
```

```
function [features,info,frameLimits] = extractSignalFeatures(x)
% EXTRACTSIGNALFEATURES Extract signal features
% [FEATURES,INFO,FRAMELIMITS] = extractSignalFeatures(X) returns a matrix
% containing features extracted from input X, INFO, a structure that maps
% a specific feature to its column location in the output feature matrix
% and FRAMELIMITS, whose i-th row contains the beginning and end limits
% of the i-th frame.
%
% Parameters of the signalTimeFeatureExtractor used to generate this
% function must be honored when calling this function.

% Generated by MATLAB(R) 9.11 and Signal Processing Toolbox 8.7.
% Generated on: 10-Jun-2021 08:25:02.

%#codegen

if istimetable(x)
     xInTT = x{:,:};
else
     xInTT = x;
end
if isrow(xInTT)
     xIn = xInTT(:);
else
     xIn = xInTT;
end
dataType = class(xIn);
signalLength = size(xIn,1);
numChannels = size(xIn,2);
frameSize = 100;
frameOverlapLength = 10;
frameRate = frameSize - frameOverlapLength;
featureMatrix = zeros(0,1,dataType);
numFeatureCols = 0;
numFeatureRows = 0;
frameLimits = zeros(0,2,dataType);
info = struct('Mean',0,'StandardDeviation',0,'PeakValue',0);
for idx = 1:numChannels
    if numChannels == 1
         xChannel = xIn;
    else
         xChannel = xIn(:,idx);
    end
```

```
    startIdx = 1;
    endIdx = frameSize;
    while startIdx <= signalLength
        if endIdx > signalLength
            break;
        end
        featureIndex = 1;
        xFrame = xChannel(startIdx:endIdx,1);
        meanValue = mean(xFrame);
        numCurrentFeature = numel(meanValue);
        info.Mean = featureIndex;
        featureIndex = featureIndex+numCurrentFeature;

        standardDeviation = std(xFrame);
        numCurrentFeature = numel(standardDeviation);
        info.StandardDeviation = featureIndex;
        featureIndex = featureIndex+numCurrentFeature;

        peakValue = max(abs(xFrame));
        info.PeakValue = featureIndex;

        featureVector = [meanValue(:);standardDeviation(:);peakValue(:)];
        featureMatrix = [featureMatrix;featureVector];
        if startIdx == 1
            numFeatureCols = size(featureVector,1);
        end
        if idx == 1
            numFeatureRows = numFeatureRows+1;
            frameLimits = [frameLimits;[startIdx endIdx]];
        end
        startIdx = startIdx+frameRate;
        endIdx = startIdx+frameSize-1;
    end
end
tempFeatureMatrix = reshape(featureMatrix,numFeatureCols,numFeatureRows,numChannels);
features = permute(tempFeatureMatrix,[2,1,3]);
end
```

You can replace calls to `extract` with calls to the generated function in your code. The outputs are identical.

```
features1 = extract(sFE,x)
```

```
features1 = 11x3
      0.0842    1.0690    2.7526
      0.0500    1.0516    2.9491
      0.1901    1.0356    2.7304
      0.1209    0.9171    2.4366
      0.0443    0.9399    2.4247
     -0.1153    1.0490    3.5699
     -0.1001    0.9530    2.4124
      0.0616    0.9959    2.7485
     -0.0263    0.9482    2.4868
     -0.0234    0.9876    3.1585
         .
         .
         .
```

```
features2 = extractSignalFeatures(x)
```

```
features2 = 11x3
      0.0842    1.0690    2.7526
      0.0500    1.0516    2.9491
      0.1901    1.0356    2.7304
      0.1209    0.9171    2.4366
      0.0443    0.9399    2.4247
     -0.1153    1.0490    3.5699
     -0.1001    0.9530    2.4124
      0.0616    0.9959    2.7485
     -0.0263    0.9482    2.4868
     -0.0234    0.9876    3.1585
        .
        .
        .
```

Copyright 2020 The MathWorks, Inc.

Input Arguments

sFE — Feature extractor object

`signalFrequencyFeatureExtractor` object | `signalTimeFeatureExtractor` object

Feature extractor object, specified as a `signalFrequencyFeatureExtractor` object or a `signalTimeFeatureExtractor` object.

See Also

Objects

`signalFrequencyFeatureExtractor` | `signalTimeFeatureExtractor`

Functions

`extract`

Introduced in R2021b

getExtractorParameters

Get current parameter values of feature extractor object

Syntax

```
getExtractorParameters(sFE,featurename)
```

Description

`getExtractorParameters(sFE,featurename)` gets the parameters used to extract `featurename`.

Examples

Get Parameter Values for Feature Extraction

Create a `signalFrequencyFeatureExtractor` object and enable the mean frequency, band power, and peak amplitude features.

```
sFE = signalFrequencyFeatureExtractor(MeanFrequency=true,BandPower=true,PeakAmplitude=true)
```

```
sFE =
  signalFrequencyFeatureExtractor with properties:

   Properties
              FrameSize: []
              FrameRate: []
             SampleRate: []
    IncompleteFrameRule: "drop"
          FeatureFormat: "matrix"

   Enabled Features
     MeanFrequency, BandPower, PeakAmplitude

   Disabled Features
     MedianFrequency, OccupiedBandwidth, PowerBandwidth, WelchPSD, PeakLocation
```

Get the current parameters for the peak amplitude. Note that not all features have parameters for feature computation.

```
params = getExtractorParameters(sFE,'PeakAmplitude')
```

```
params = struct with fields:
          PeakType: "maxima"
    MaxNumExtrema: 1
    MinProminence: []
    MinSeparation: []
    FlatSelection: []
```

You can also modify feature parameters. Set the maximum number of peaks to 2 and the minimum peak separation to 3 samples.

```
params.MaxNumExtrema = 2;
params.MinSeparation = 3
```

```
params = struct with fields:
          PeakType: "maxima"
    MaxNumExtrema: 2
    MinProminence: []
    MinSeparation: 3
    FlatSelection: []
```

Input Arguments

sFE — Feature extractor object

`signalFrequencyFeatureExtractor` object

Feature extractor object, specified as a `signalFrequencyFeatureExtractor` object.

featurename — Extracted feature

string scalar | character vector

Extracted feature, specified as a string scalar or a character vector.

Data Types: `char` | `string`

See Also

Objects

`signalFrequencyFeatureExtractor`

Functions

`extract` | `setExtractorParameters`

Introduced in R2021b

setExtractorParameters

Set nondefault values for feature extractor object

Syntax

```
setExtractorParameters(sFE,featurename,params)
setExtractorParameters(sFE,featurename)
```

Description

`setExtractorParameters(sFE,featurename,params)` specifies the parameters used to extract `featurename`.

`setExtractorParameters(sFE,featurename)` sets the parameters used to extract `featurename` to their default values.

Examples

Set Parameter Values for Feature Extraction

Create a `signalFrequencyFeatureExtractor` object to extract the Welch power spectral density (PSD) estimate of a signal consisting of a 100 Hz sinusoid in additive $N(0,1)$ white noise. The sample rate is 1 kHz and the signal has a duration of 5 seconds.

```
fs = 1000;
t = 0:1/fs:5-1/fs;
x = cos(2*pi*100*t) + randn(size(t));

sFE = signalFrequencyFeatureExtractor(SampleRate=fs,WelchPSD=true);
```

For the PSD computation, set the `OverlapLength` to 25 samples and the `FFTLength` to 512 samples. Call the `getExtractorParameters` function on the object to view the PSD parameters.

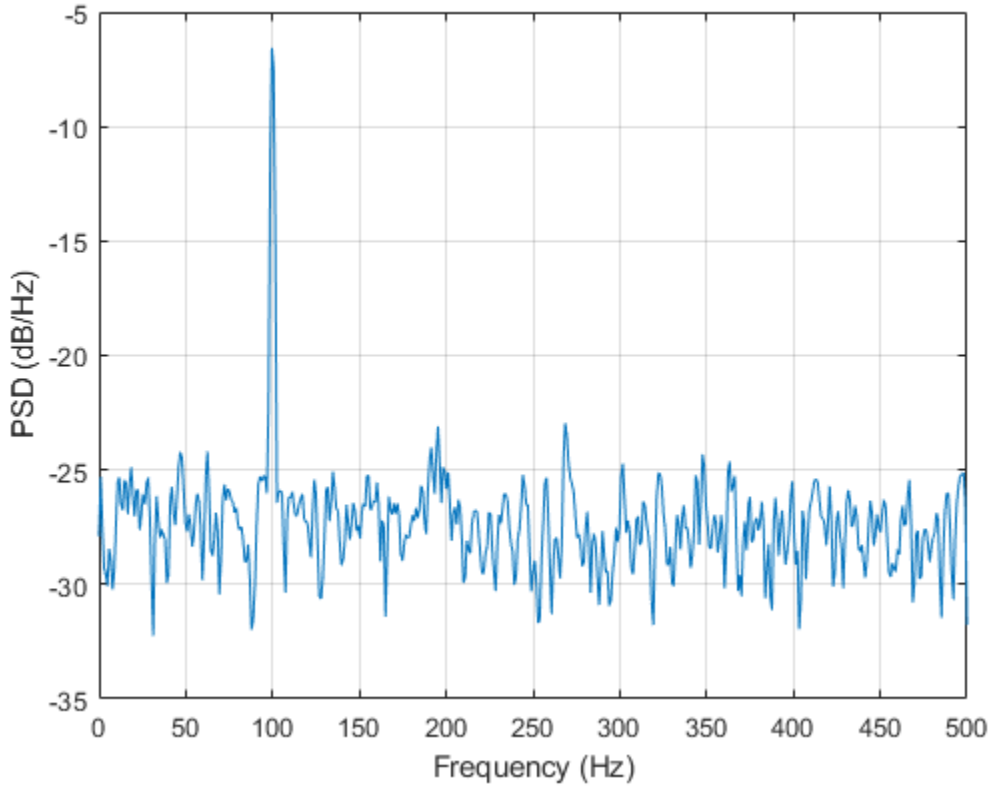
```
setExtractorParameters(sFE,"WelchPSD",OverlapLength=25,FFTLength=1024)
params = getExtractorParameters(sFE,"WelchPSD")
```

```
params = struct with fields:
          FFTLength: 1024
    FrequencyVector: []
      OverlapLength: 25
             Window: []
```

Use the `extract` function to extract the Welch's PSD estimate of the signal. Plot the result.

```
feature = extract(sFE,x);

fvec = linspace(0,fs/2,length(feature));
plot(fvec,pow2db(abs(feature)))

xlabel("Frequency (Hz)")
```

```
ylabel("PSD (dB/Hz)")
grid
```



Input Arguments

`sFE` — Feature extractor object

`signalFrequencyFeatureExtractor` object

Feature extractor object, specified as a `signalFrequencyFeatureExtractor` object.

`featurename` — Extracted feature

string scalar | character vector

Extracted feature, specified as a string scalar or a character vector.

Data Types: `char` | `string`

`params` — Parameters

name-value arguments | structure

Parameters used to extract `featurename`, specified as name-value arguments or a structure.

Data Types: `string` | `struct`

See Also

Objects
`signalFrequencyFeatureExtractor`

Functions
`extract` | `getExtractorParameters`

Introduced in R2021b

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 1-2185 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-Hann Window	`sigwin.barthannwin`	`barthannwin`
Bartlett Window	`sigwin.bartlett`	`bartlett`
Blackman Window	`sigwin.blackman`	`blackman`
Blackman-Harris Window	`sigwin.blackmanharris`	`blackmanharris`
Bohman Window	`sigwin.bohmanwin`	`bohmanwin`
Dolph-Chebyshev Window	`sigwin.chebwin`	`chebwin`
Flat Top Window	`sigwin.flattopwin`	`flattopwin`
Gaussian Window	`sigwin.gausswin`	`gausswin`
Hamming Window	`sigwin.hamming`	`hamming`
Hann (Hanning) Window	`sigwin.hann`	`hann`
Kaiser Window	`sigwin.kaiser`	`kaiser`
Nuttall defined 4-term Blackman-Harris Window	`sigwin.nuttallwin`	`nuttallwin`
Parzen Window	`sigwin.parzenwin`	`parzenwin`
Rectangular Window	`sigwin.rectwin`	`rectwin`
Taylor Window	`sigwin.taylorwin`	`taylorwin`

Window	Window Object	Corresponding Function
Triangular Window	`sigwin.triang`	`triang`
Tukey Window	`sigwin.tukeywin`	`tukeywin`

Methods

Methods provide ways of performing functions directly on your `sigwin` object without having to specify the window parameters again. You can apply this method directly on the variable you assigned to your `sigwin` object.

Method	Description
`generate`	Returns a column vector of values representing the window.
`info`	Returns information about the window object.
`winwrite`	Writes an ASCII file that contains window weights for a single window object or a vector of window objects. Default filename is `untitled.wf`. `winwrite(Hd,filename)` writes to a disk file named `filename` in the current working directory. The `.wf` extension is added automatically.

Viewing Object Parameters

As with any object, you can use `get` to view a `sigwin` object's parameters. To see a specific parameter,

```
get(w,'parameter')
```

or to see all parameters for an object,

```
get(w)
```

Changing Object Parameters

To set specific parameters,

```
set(w,'parameter1',value,'parameter2',value,...)
```

Note that you must use single quotation marks around the parameter name.

Examples

Bartlett window Object

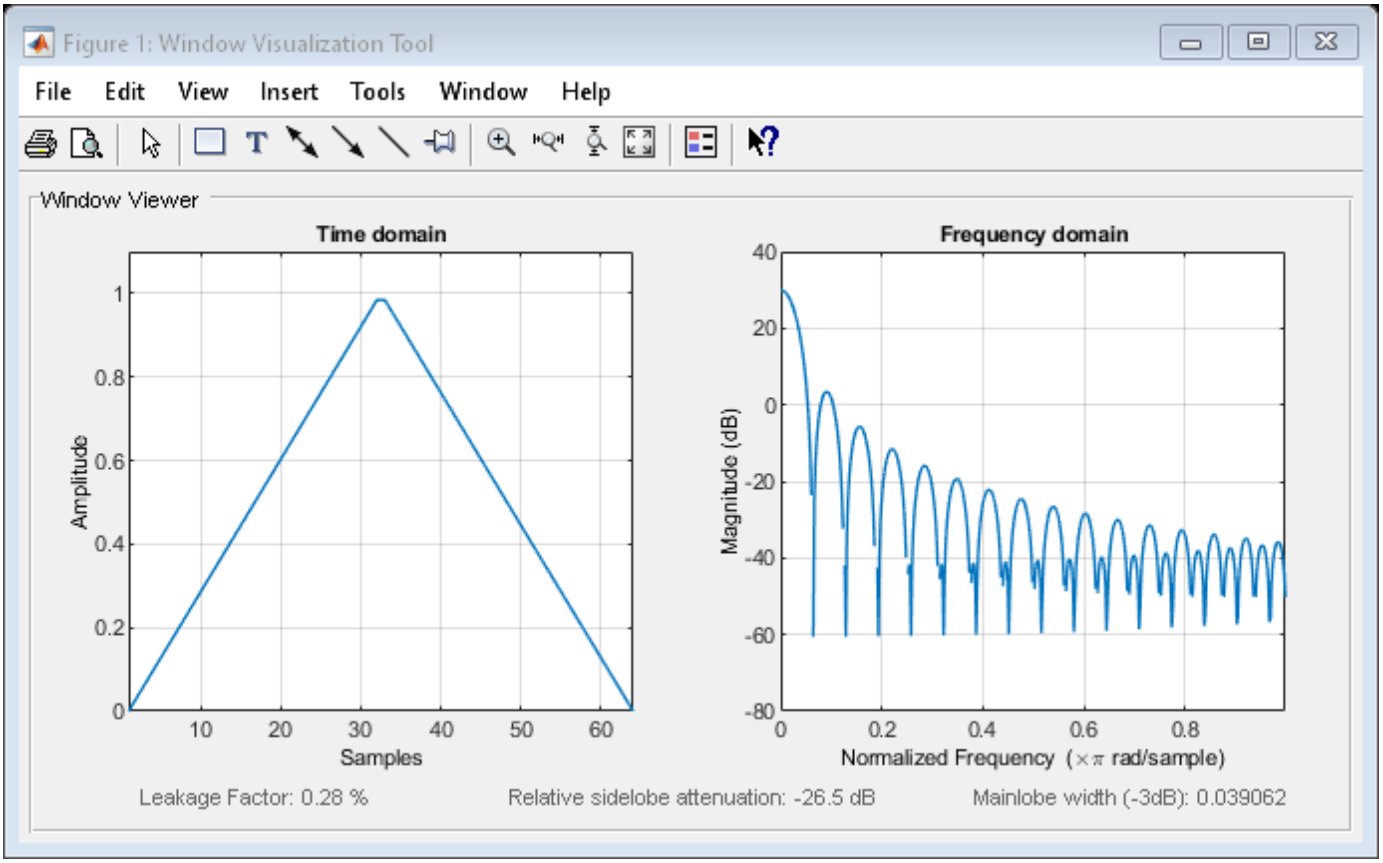
Create a default Bartlett window and view the result in `wvtool`. See `bartlett` for information on Bartlett windows.

```
w = sigwin.bartlett
```

```
w =
      Name: 'Bartlett'
```

```
    Length: 64
```

```
wvtool(w)
```



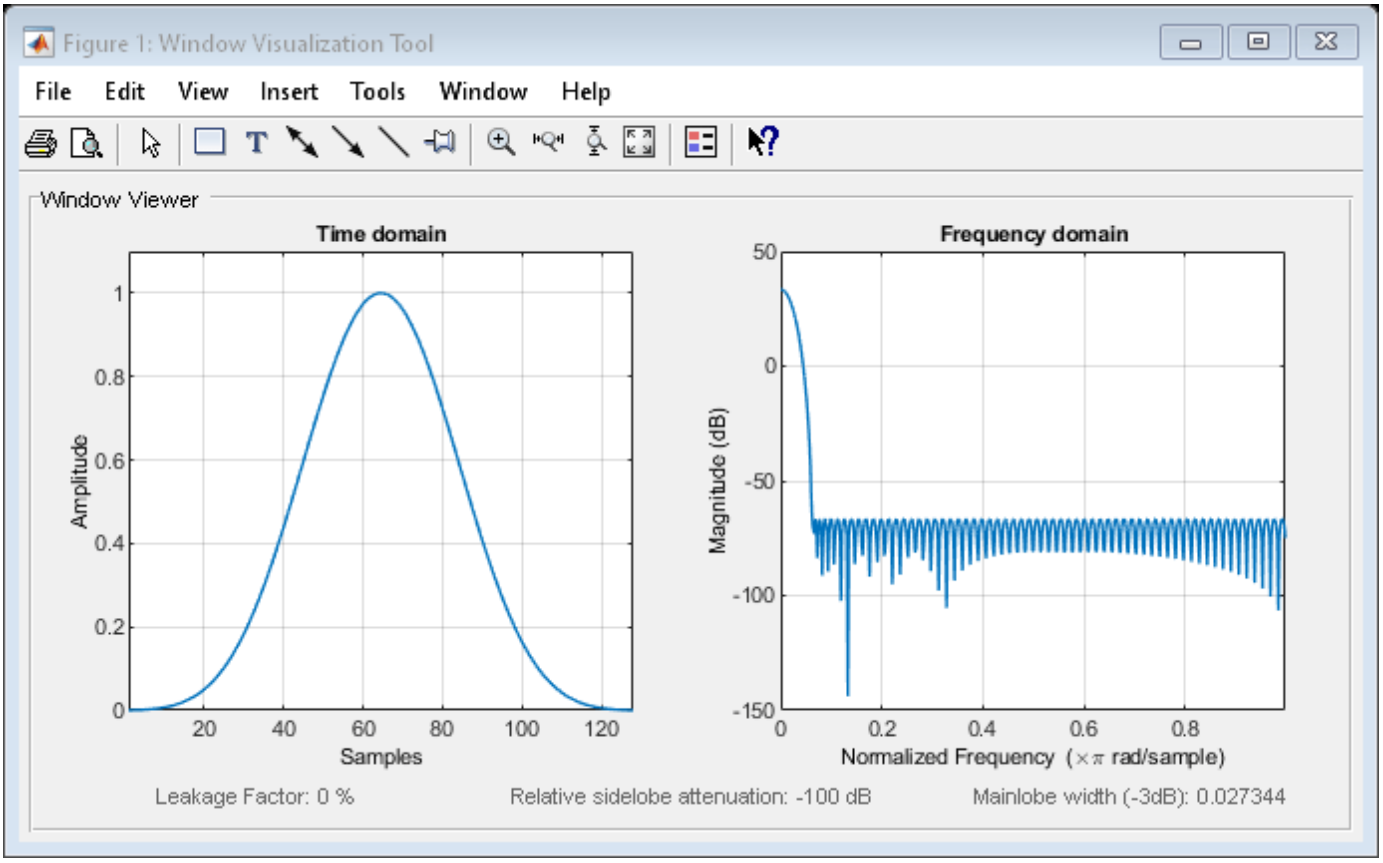
Chebyshev Window Object

Create a 128-point Chebyshev window with 100 dB of sidelobe attenuation. (See `chebwin` for information on Chebyshev windows.) View the result with `wvtool`.

```
w = sigwin.chebwin(128,100)
```

```
w =
             Name: 'Chebyshev'
           Length: 128
    SidelobeAtten: 100
```

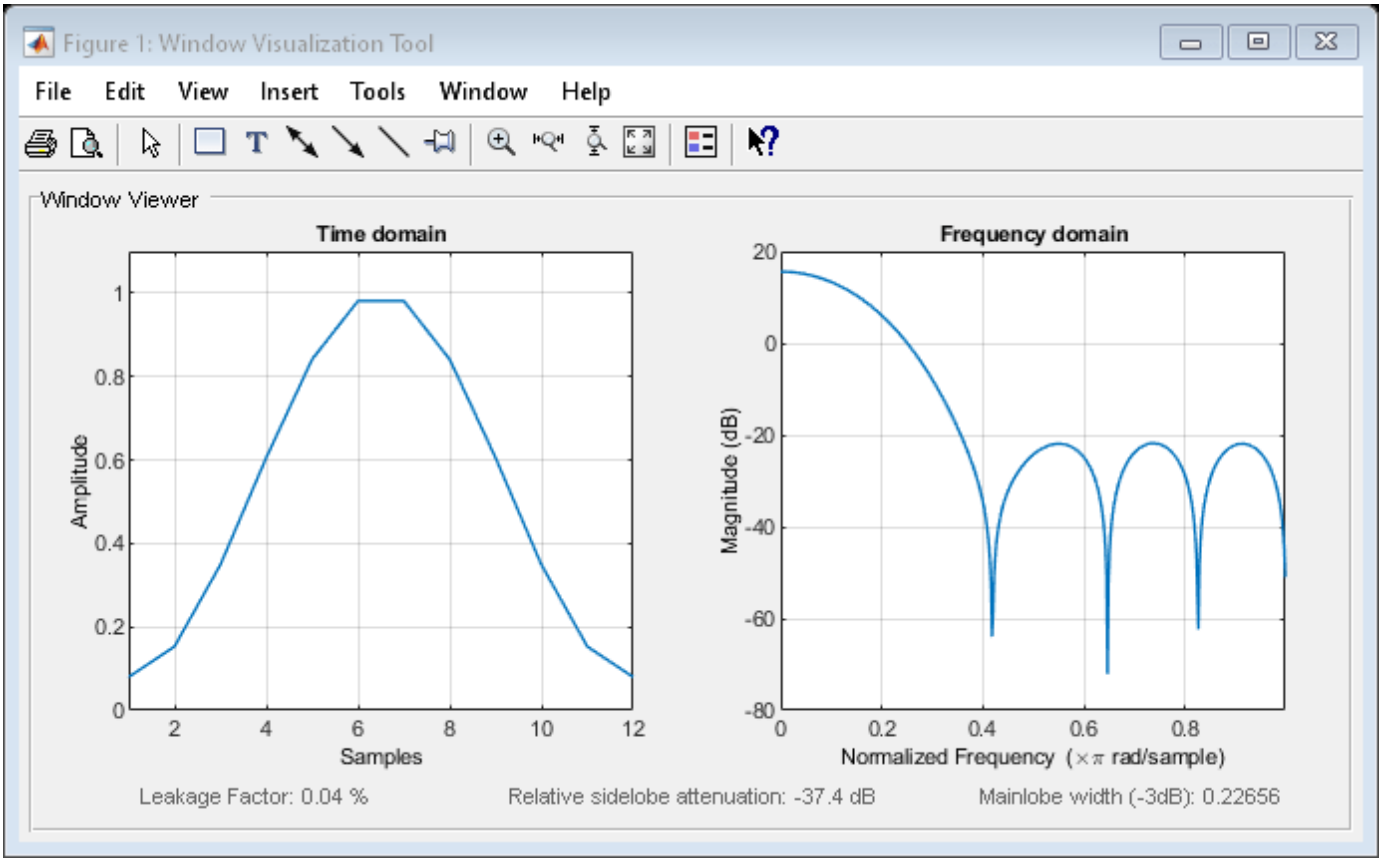
```
wvtool(w)
```



Save Window Object as Vector

Create a Hamming window of length 12. Visualize the result.

```
H = sigwin.hamming(12);
wvtool(H)
```



Save the window values in a column vector.

```
d = generate(H)'
```

```
d = 1×12

    0.0800    0.1530    0.3489    0.6055    0.8412    0.9814    0.9814    0.8412    0.6055    0.1
```

See Also

Apps
Window Designer

Functions
WVTool

Introduced before R2006a

sigwin.barthannwin class

Package: `sigwin`

Construct modified Bartlett-Hann window object

Description

Note The use of `sigwin.barthannwin` is not recommended. Use `barthannwin` instead.

`sigwin.barthannwin` creates a handle to a modified Bartlett-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 following equation defines a modified Bartlett-Hann window of length N:

$$w(x) = 0.62 - 0.48|x| + 0.38\cos 2\pi x, \quad -\frac{1}{2} \le x \le \frac{1}{2}$$

where x is an N-point linearly spaced vector over the interval [1/2, 1/2].

Construction

`H = sigwin.barthannwin` returns a modified Bartlett-Hann window object `H` of length 64.

`H = sigwin.barthannwin(Length)` returns a modified Bartlett-Hann 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-Hann 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-Hann window
info	Display information about modified Bartlett-Hann window object
winwrite	Save modified Bartlett-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

Modified Bartlett-Hann Window

Generate a modified Bartlett-Hann window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

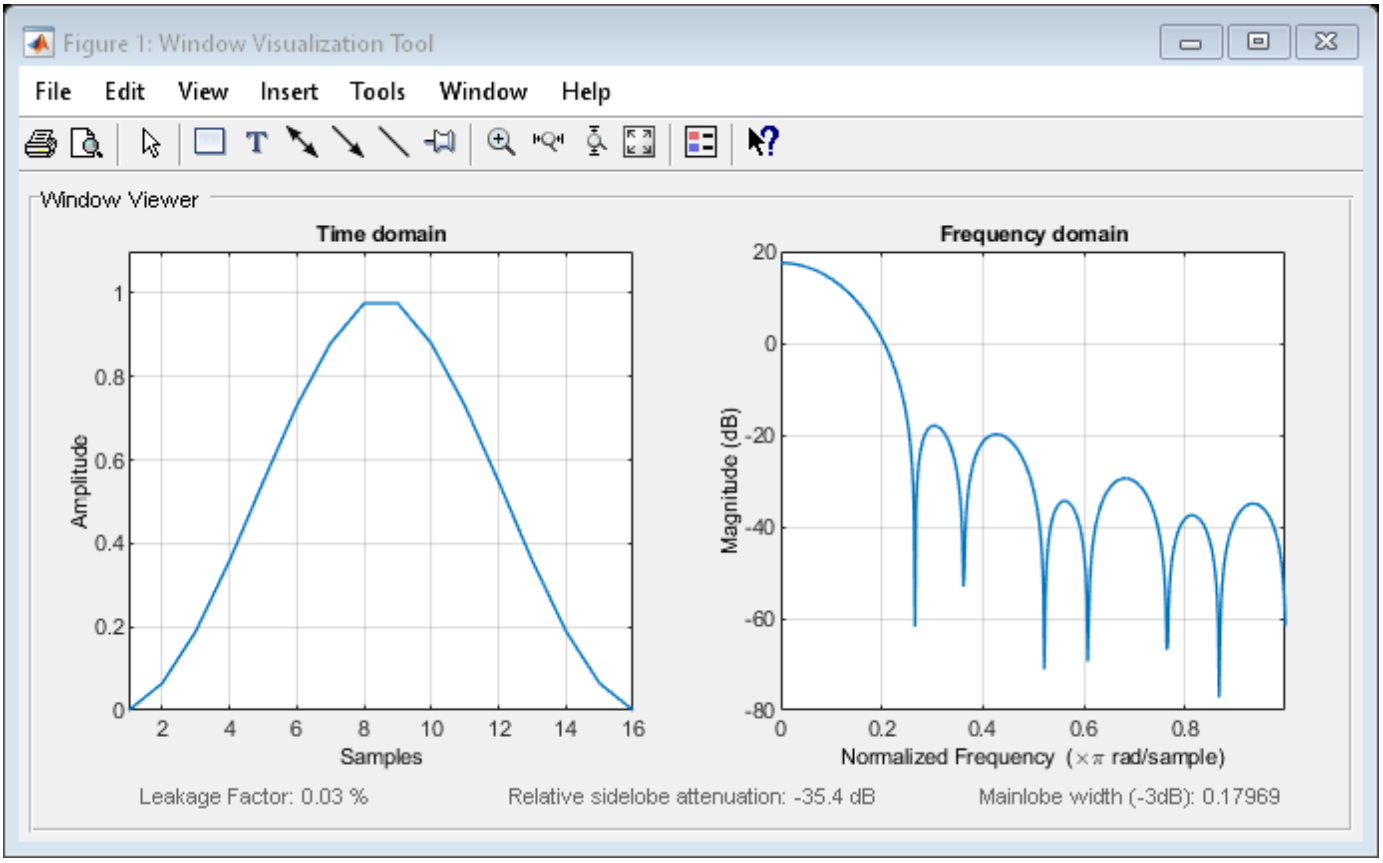
```
H = sigwin.barthannwin(16);

win = generate(H)
```

```
win = 16×1

         0
    0.0649
    0.1897
    0.3586
    0.5477
    0.7300
    0.8794
    0.9757
    0.9757
    0.8794
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x23 char array
    'Bartlett-Hanning Window'
    '-----------------------'
    'Length  : 16           '
```

```
wvtool(H)
```



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**

Topics

"Windows"
Class Attributes
Property Attributes

generate

Class: `sigwin.barthannwin`
Package: `sigwin`

Generates modified Bartlett-Hann window

Syntax

```
win = generate(H)
```

Description

`win = generate(H)` returns the values of the modified Bartlett-Hann window object `H` as a double-precision column vector.

Examples

Modified Bartlett-Hann Window

Generate a modified Bartlett-Hann window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

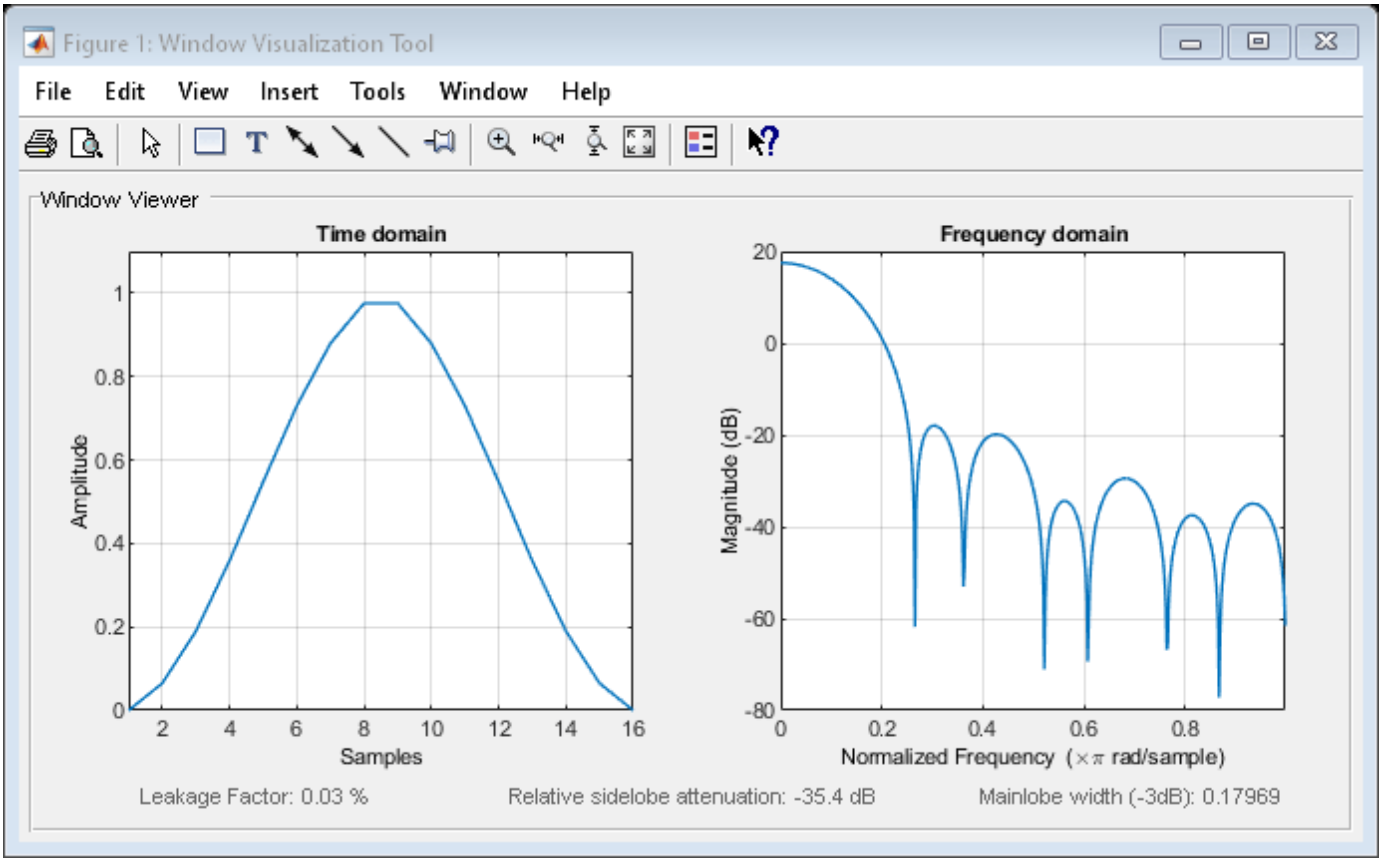
```
H = sigwin.barthannwin(16);

win = generate(H)
```

```
win = 16×1

         0
    0.0649
    0.1897
    0.3586
    0.5477
    0.7300
    0.8794
    0.9757
    0.9757
    0.8794
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x23 char array
    'Bartlett-Hanning Window'
    '-----------------------'
    'Length  : 16           '
```

```
wvtool(H)
```



See Also

`barthannwin` | `window` | **WVTool**

info

Class: `sigwin.barthannwin`
Package: `sigwin`

Display information about modified Bartlett-Hann window object

Syntax

```
info(H)
info_win = info(H)
```

Description

`info(H)` displays length information for the modified Bartlett-Hann window object `H`.

`info_win = info(H)` returns length information for the modified Bartlett-Hann window object `H` in the character array `info_win`.

Examples

Modified Bartlett-Hann Window

Generate a modified Bartlett-Hann window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

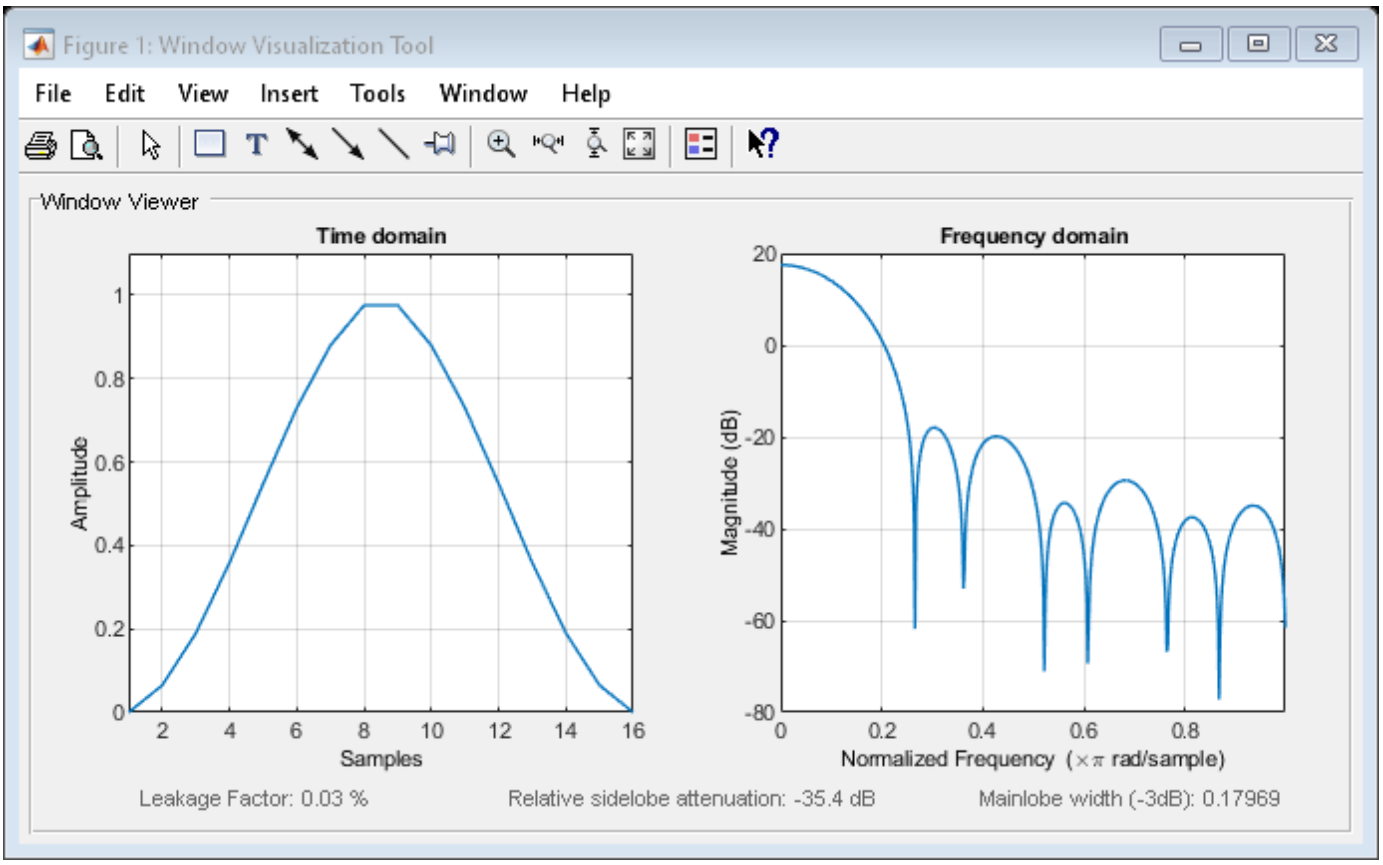
```
H = sigwin.barthannwin(16);

win = generate(H)
```

```
win = 16×1

         0
    0.0649
    0.1897
    0.3586
    0.5477
    0.7300
    0.8794
    0.9757
    0.9757
    0.8794
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x23 char array
    'Bartlett-Hanning Window'
    '------------------------'
    'Length  : 16           '
```

```
wvtool(H)
```



See Also

`barthannwin` | `window` | **WVTool**

winwrite

Class: `sigwin.barthannwin`
Package: `sigwin`

Save modified Bartlett-Hann 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-Hann window object `H` to an ASCII file with file name extension `wf`.

`winwrite(H,'filename')` saves the values of the modified Bartlett-Hann window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The filename extension is `wf`.

Examples

Modified Bartlett-Hann Window

Generate a modified Bartlett-Hann window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

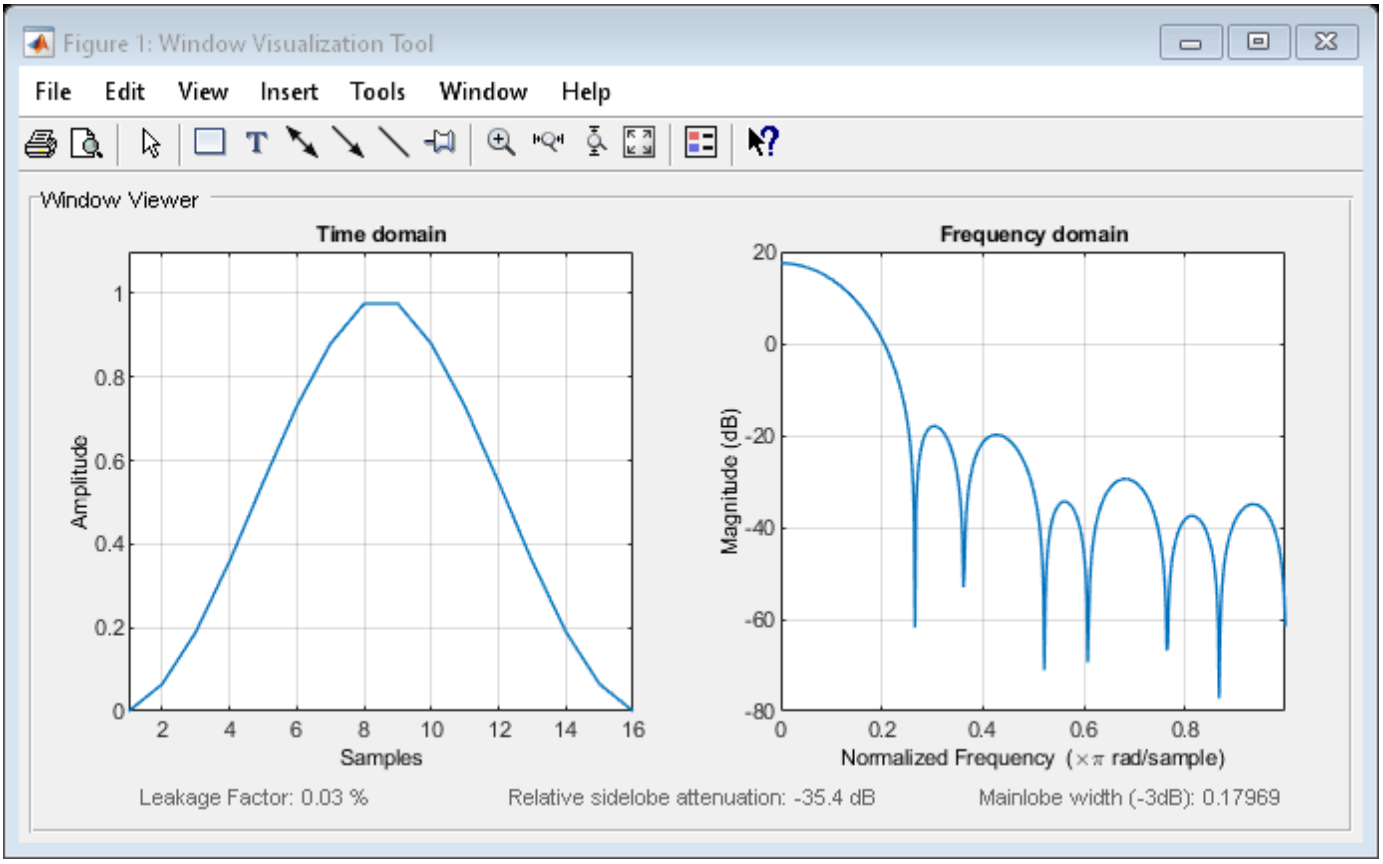
```
H = sigwin.barthannwin(16);

win = generate(H)
```

```
win = 16×1

         0
    0.0649
    0.1897
    0.3586
    0.5477
    0.7300
    0.8794
    0.9757
    0.9757
    0.8794
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x23 char array
    'Bartlett-Hanning Window'
    '------------------------'
```

```
        'Length  : 16          '

wvtool(H)
```



See Also

`barthannwin` | `window` | **WVTool**

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:

$$w(n) = \begin{cases} \dfrac{2n}{N-1} & 0 \le n \le N/2 - 1 \\ 2 - \dfrac{2n}{N-1} & N/2 \le n \le N - 1 \end{cases}$$

For *N* odd, the equation for the Bartlett window is:

$$w(n) = \begin{cases} \dfrac{2n}{N-1} & 0 \le n \le (N-1)/2 \\ 2 - \dfrac{2n}{N-1} & (N-1)/2 + 1 \le n \le N - 1 \end{cases}$$

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

Bartlett Window

Generate a Bartlett window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

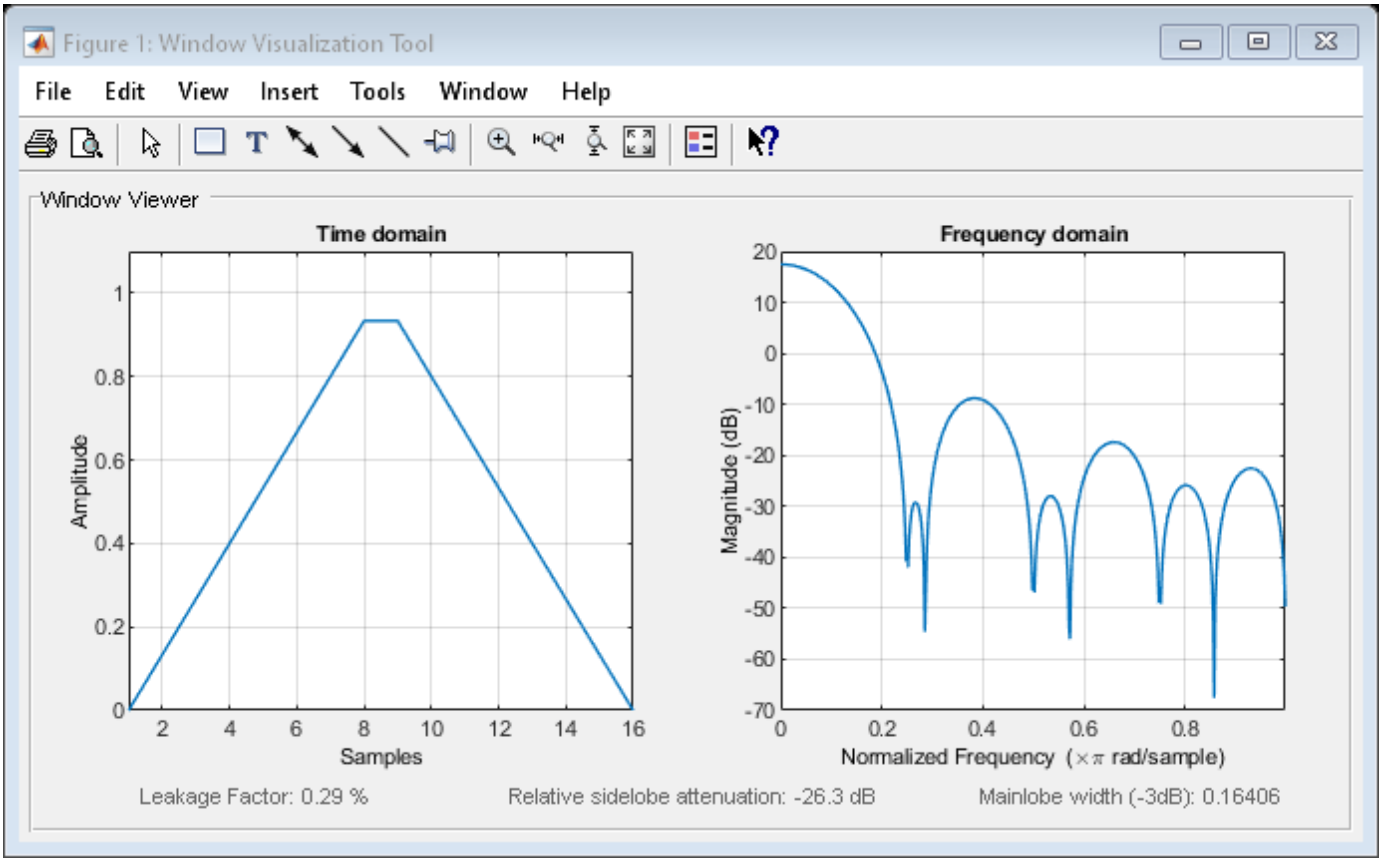
```
H = sigwin.bartlett(16);

win = generate(H)
```

```
win = 16×1

         0
    0.1333
    0.2667
    0.4000
    0.5333
    0.6667
    0.8000
    0.9333
    0.9333
    0.8000
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x15 char array
    'Bartlett Window'
    '---------------'
    'Length  : 16   '
```

```
wvtool(H)
```



References

Oppenheim, Alan V., and Ronald W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

See Also

`bartlett` | `window` | **WVTool**

Topics

“Windows”
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

Bartlett Window

Generate a Bartlett window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

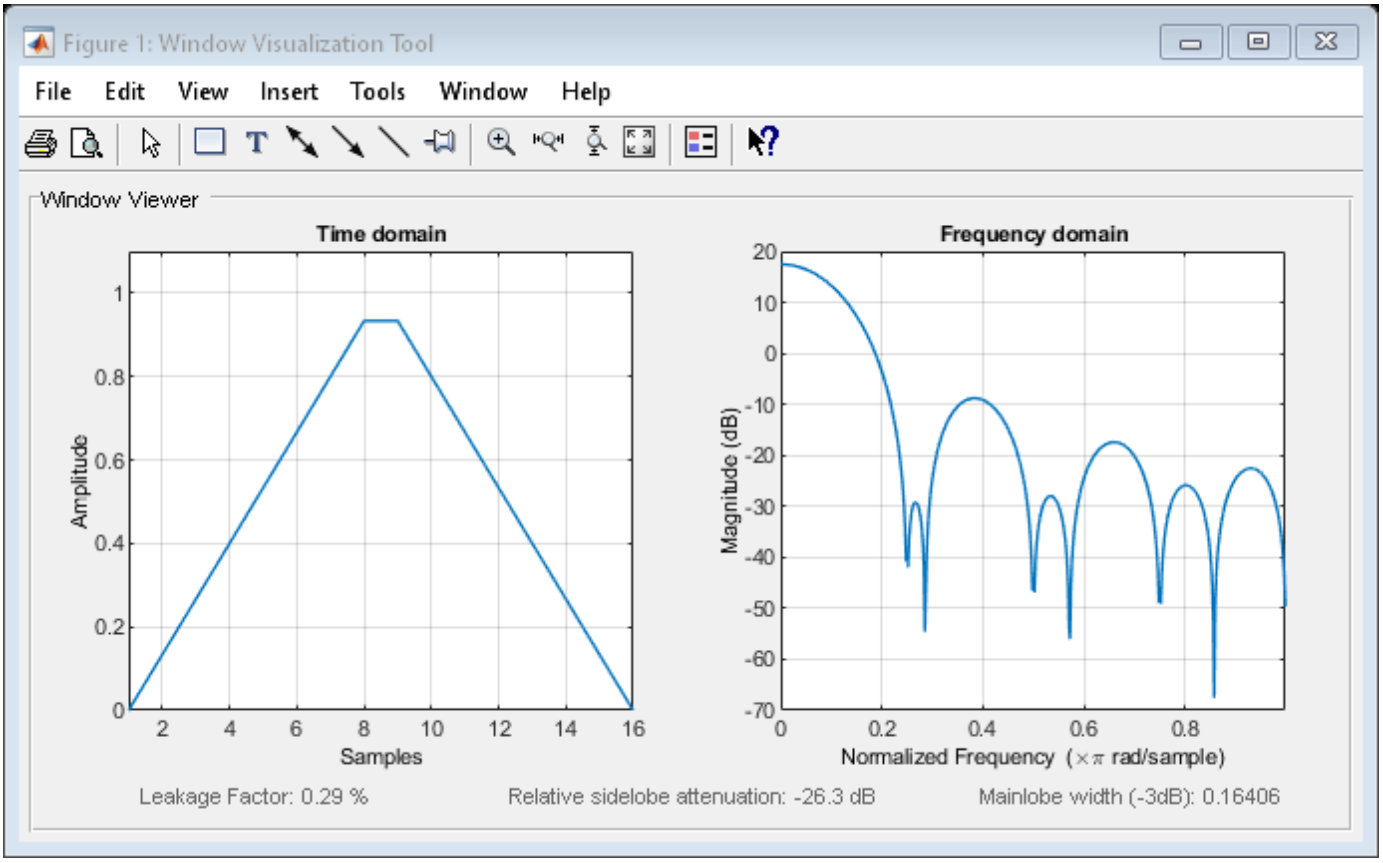
```
H = sigwin.bartlett(16);

win = generate(H)
```

```
win = 16×1

         0
    0.1333
    0.2667
    0.4000
    0.5333
    0.6667
    0.8000
    0.9333
    0.9333
    0.8000
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x15 char array
    'Bartlett Window'
    '---------------'
    'Length  : 16   '
```

```
wvtool(H)
```



See Also

`bartlett` | `window` | **WVTool**

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

Bartlett Window

Generate a Bartlett window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.bartlett(16);
```

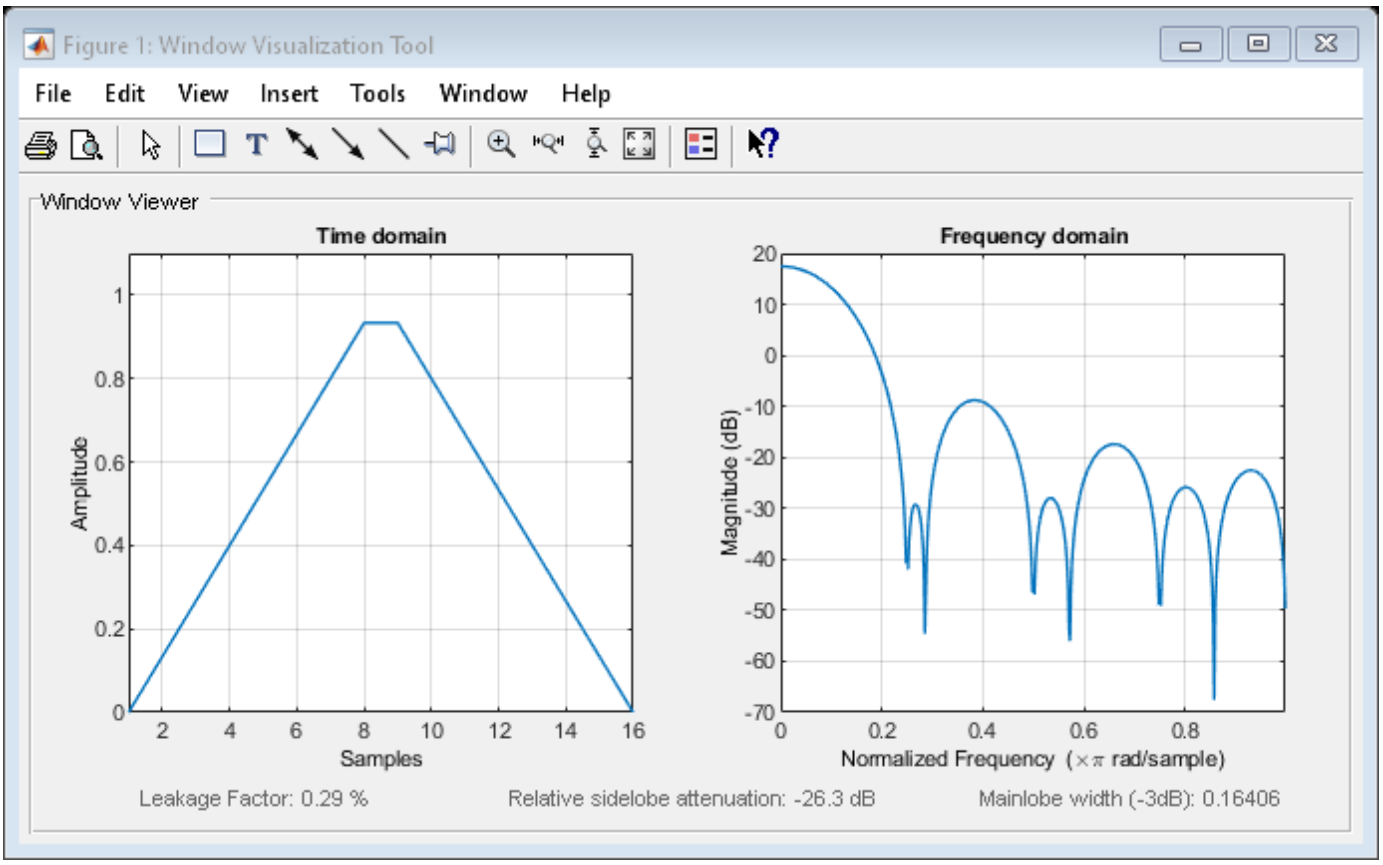
```
win = generate(H)
```

```
win = 16×1

         0
    0.1333
    0.2667
    0.4000
    0.5333
    0.6667
    0.8000
    0.9333
    0.9333
    0.8000
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x15 char array
    'Bartlett Window'
    '---------------'
    'Length  : 16   '
```

```
wvtool(H)
```



See Also

`bartlett` | `window` | **WVTool**

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

Bartlett Window

Generate a Bartlett window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

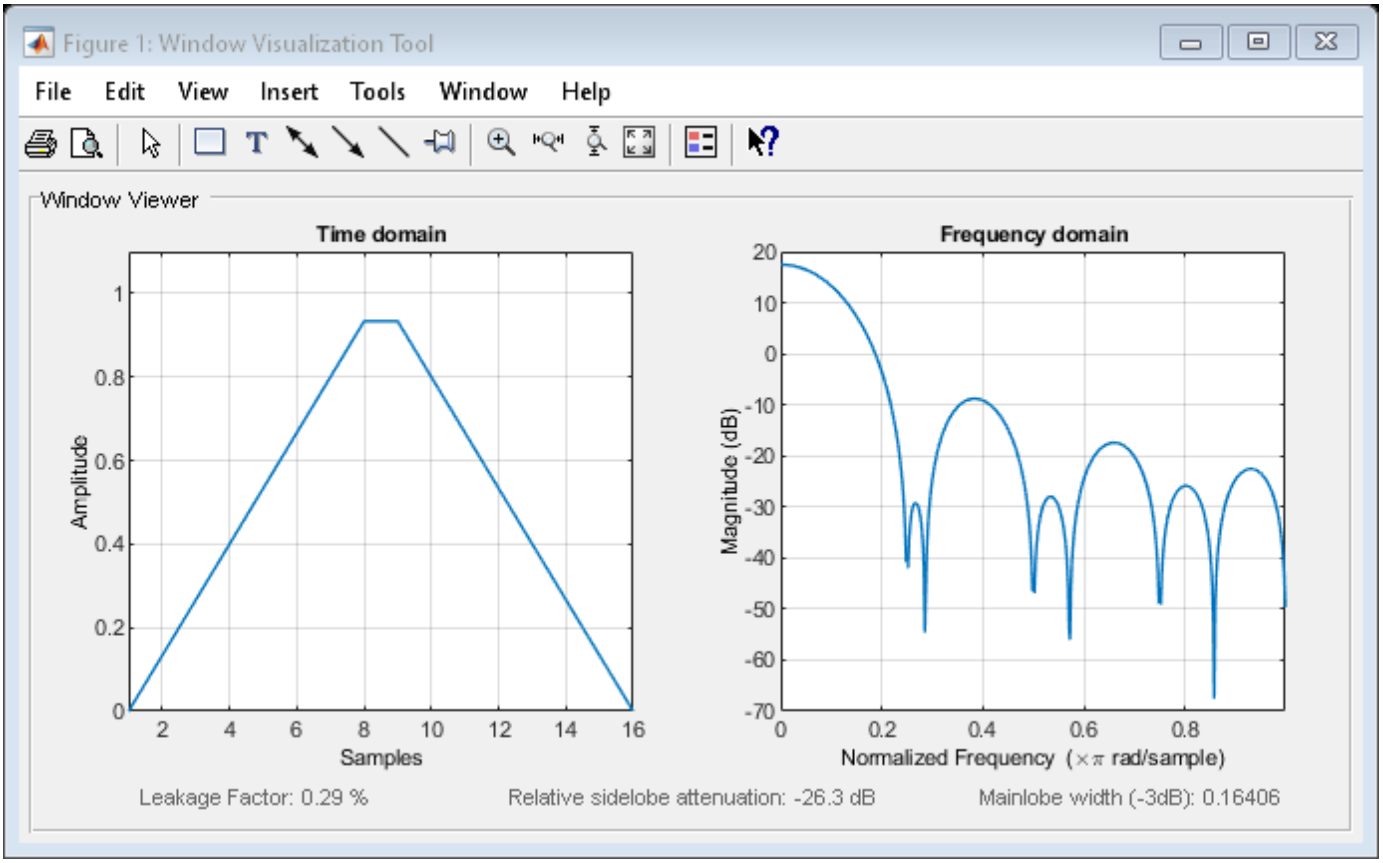
```
H = sigwin.bartlett(16);

win = generate(H)
```

```
win = 16×1

         0
    0.1333
    0.2667
    0.4000
    0.5333
    0.6667
    0.8000
    0.9333
    0.9333
    0.8000
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x15 char array
    'Bartlett Window'
    '---------------'
```

```
    'Length  : 16   '

wvtool(H)
```



See Also

`bartlett` | `window` | **WVTool**

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*:

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

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 $M \le n \le N-1$ 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

Blackman Window

Generate a Blackman window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.blackman(16);

win = generate(H)
```

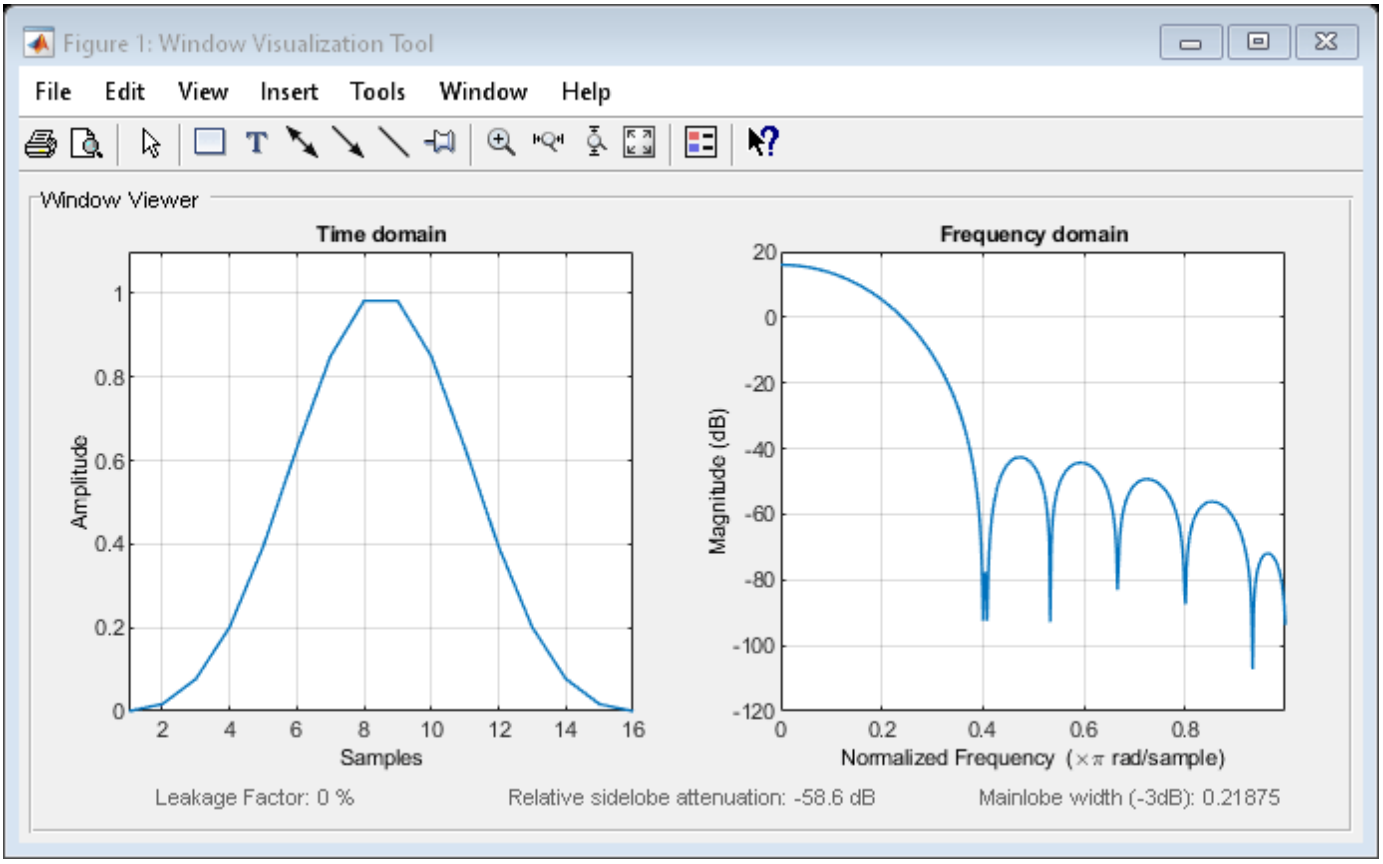
```
win = 16×1

         0
    0.0168
    0.0771
    0.2008
    0.3940
    0.6300
    0.8492
    0.9822
    0.9822
    0.8492
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Blackman Window           '
    '---------------           '
    'Length         : 16       '
```

```
'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



References

Oppenheim, Alan V., and Ronald W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

See Also

`blackman` | `window` | **WVTool**

Topics
“Windows”
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

Blackman Window

Generate a Blackman window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.blackman(16);
```

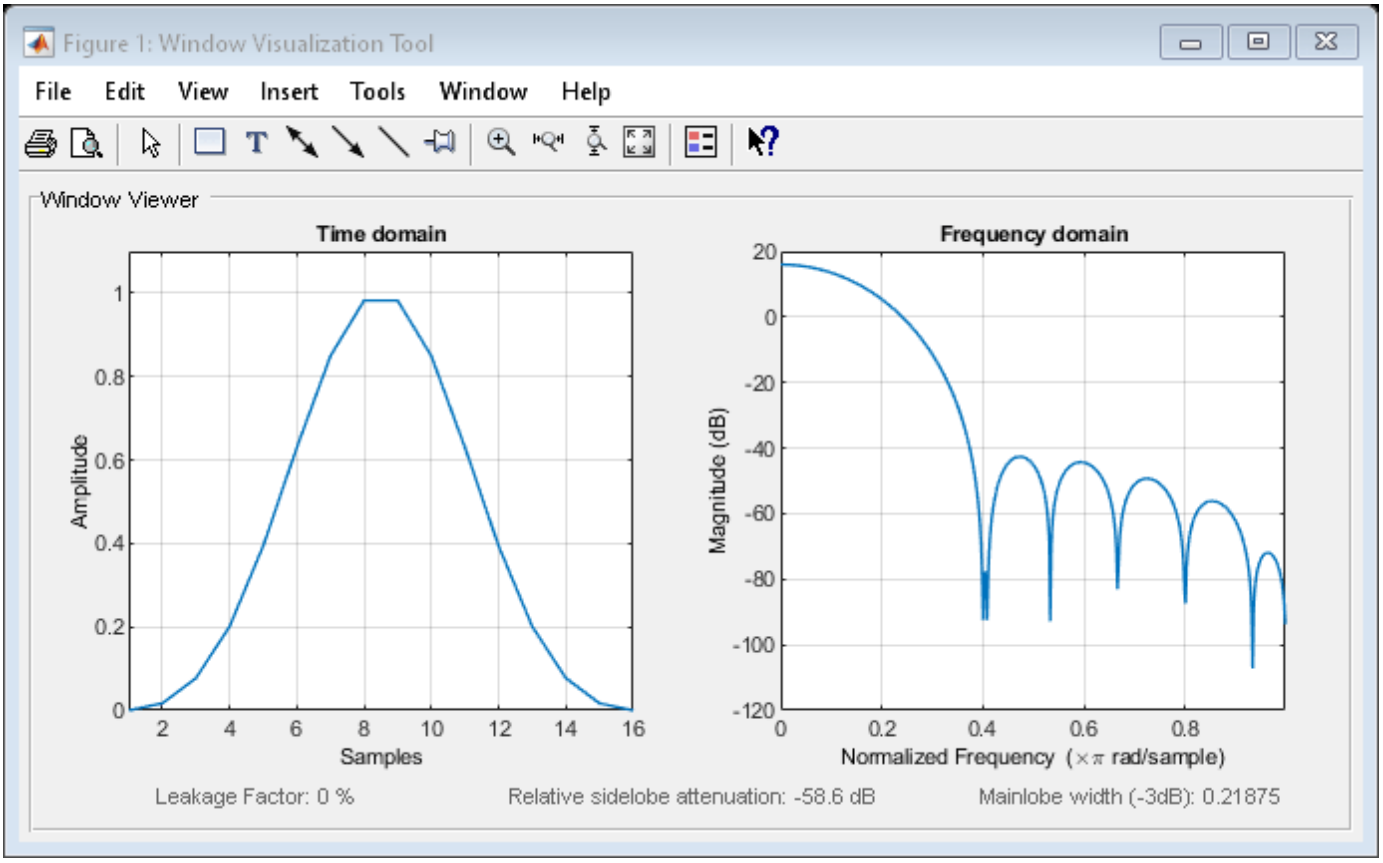
```
win = generate(H)
```

```
win = 16×1

         0
    0.0168
    0.0771
    0.2008
    0.3940
    0.6300
    0.8492
    0.9822
    0.9822
    0.8492
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Blackman Window           '
    '---------------           '
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

`blackman` | **WVTool**

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

Blackman Window

Generate a Blackman window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

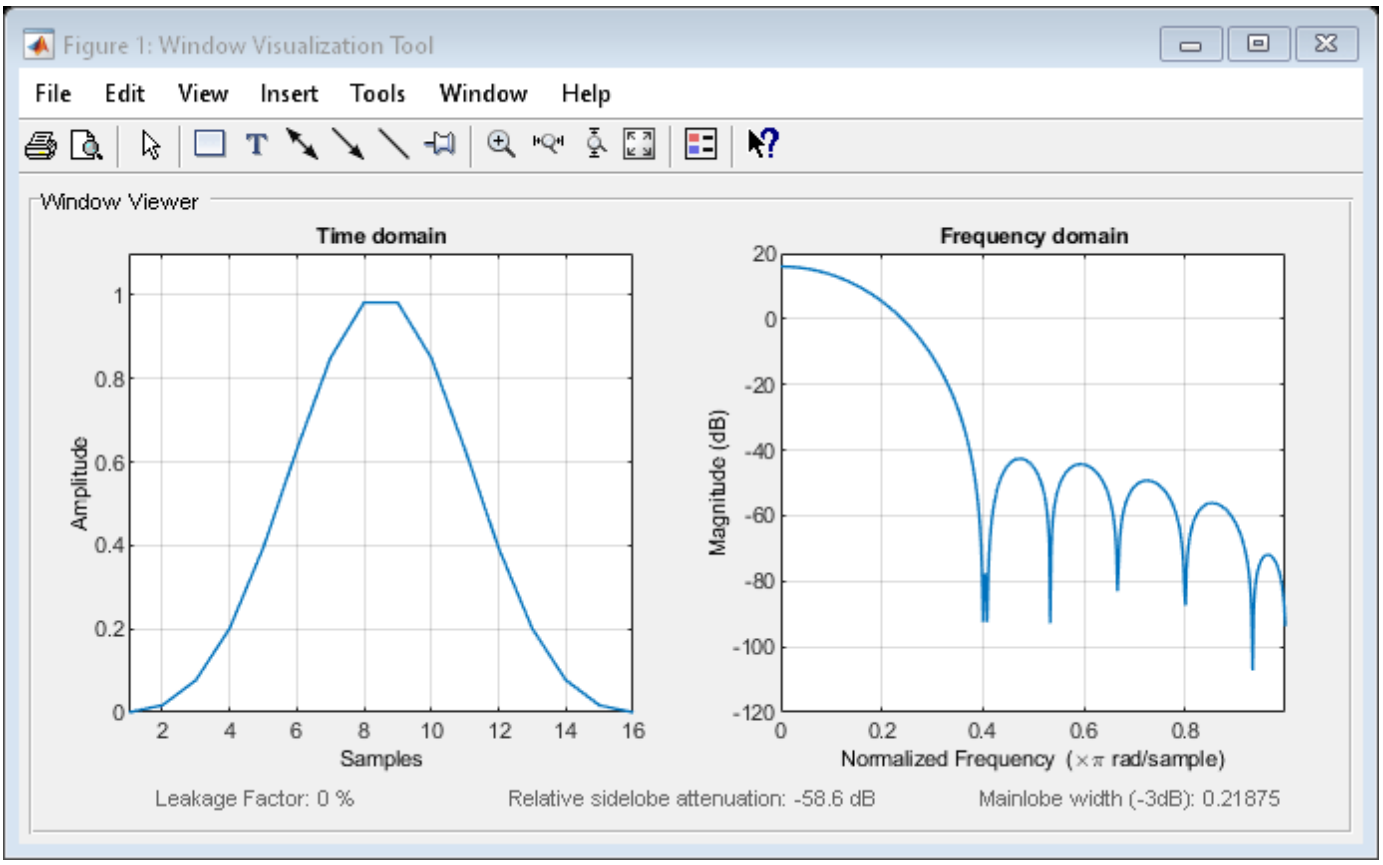
```
H = sigwin.blackman(16);

win = generate(H)
```

```
win = 16×1

         0
    0.0168
    0.0771
    0.2008
    0.3940
    0.6300
    0.8492
    0.9822
    0.9822
    0.8492
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Blackman Window           '
    '----------------          '
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

`blackman` | `window` | **WVTool**

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 file name 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

Blackman Window

Generate a Blackman window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.blackman(16);

win = generate(H)
```

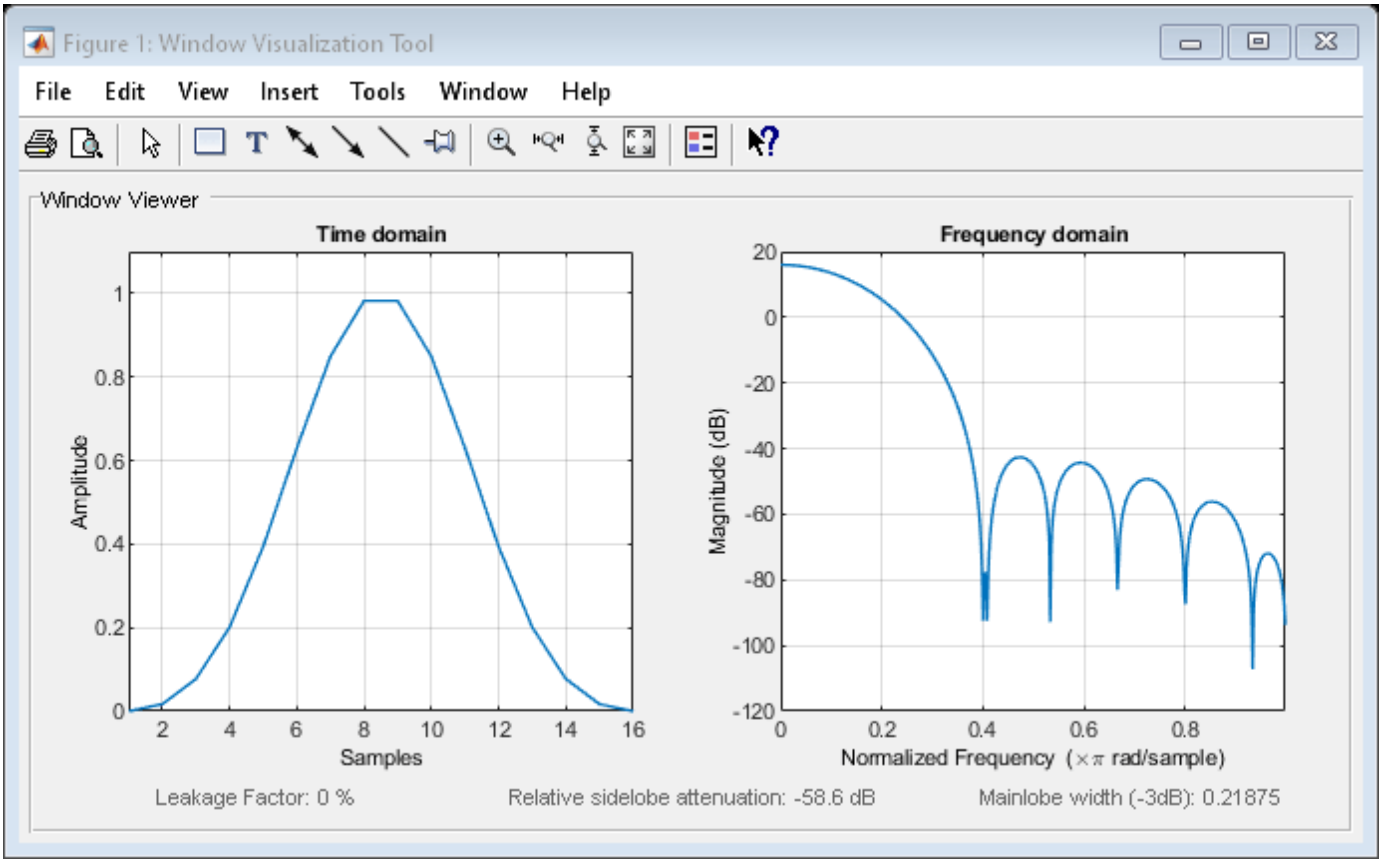
```
win = 16×1

         0
    0.0168
    0.0771
    0.2008
    0.3940
    0.6300
    0.8492
    0.9822
    0.9822
    0.8492
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Blackman Window           '
    '---------------           '
    'Length         : 16       '
```

```
'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

`blackman` | `window` | **WVTool**

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*:

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

The following equation defines the **periodic** Blackman-Harris window of length *N*:

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

The following table lists the coefficients:

Coefficient	Value
a0	0.35875
a1	0.48829
a2	0.14128
a3	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 1-2217 for details.

Methods

generate	Generates Blackman–Harris window
info	Display information about Blackman–Harris window object
winwrite	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

Blackman-Harris Window

Generate a Blackman-Harris window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

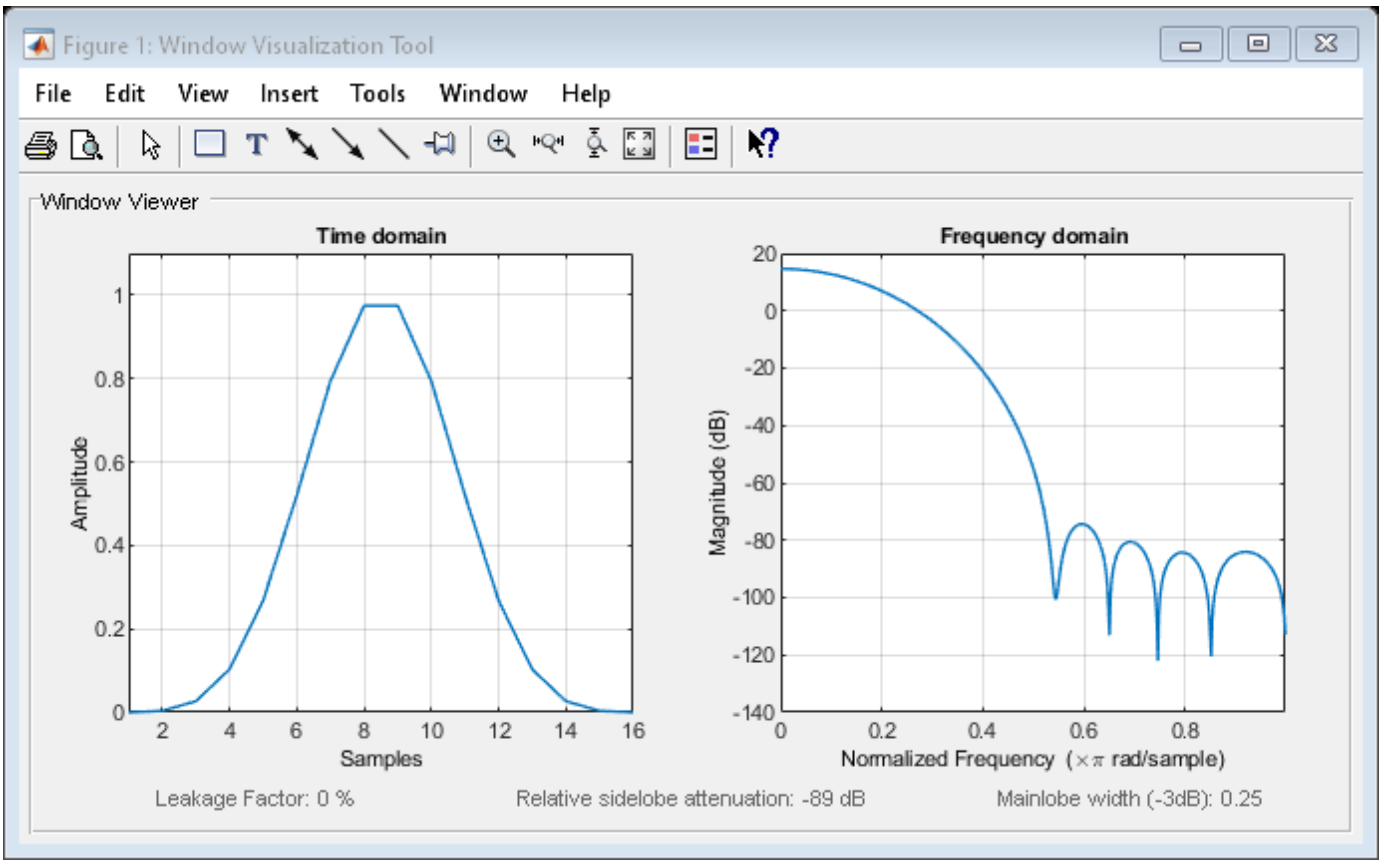
```
H = sigwin.blackmanharris(16);

win = generate(H)
```

```
win = 16×1

    0.0001
    0.0036
    0.0267
    0.1030
    0.2680
    0.5206
    0.7938
    0.9749
    0.9749
    0.7938
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Blackman-Harris Window    '
    '----------------------    '
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



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**

Topics
“Windows”
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

Blackman-Harris Window

Generate a Blackman-Harris window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.blackmanharris(16);
```

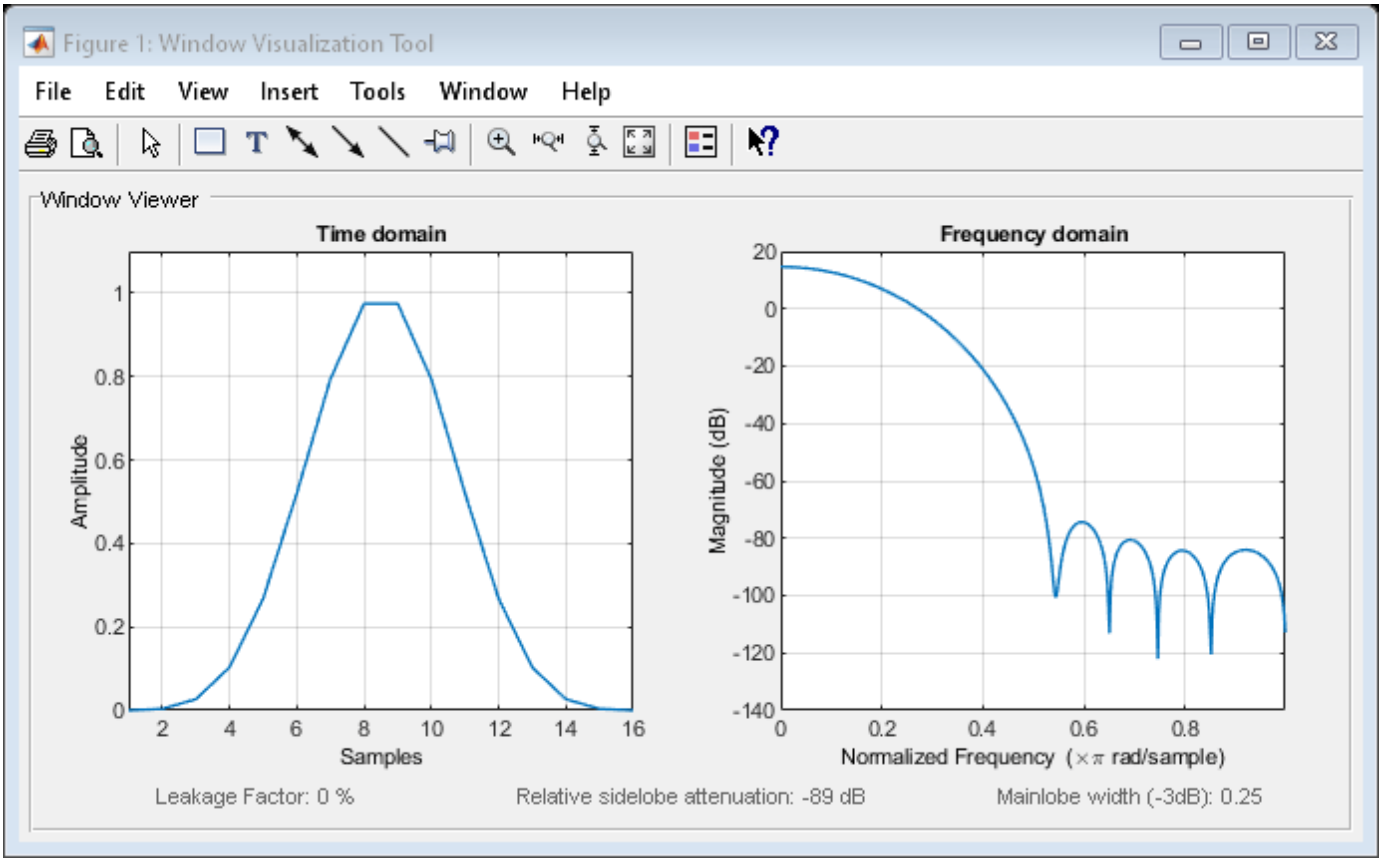
```
win = generate(H)
```

```
win = 16×1

    0.0001
    0.0036
    0.0267
    0.1030
    0.2680
    0.5206
    0.7938
    0.9749
    0.9749
    0.7938
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Blackman-Harris Window    '
    '----------------------    '
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

`blackmanharris` | `window` | **WVTool**

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

Blackman-Harris Window

Generate a Blackman-Harris window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

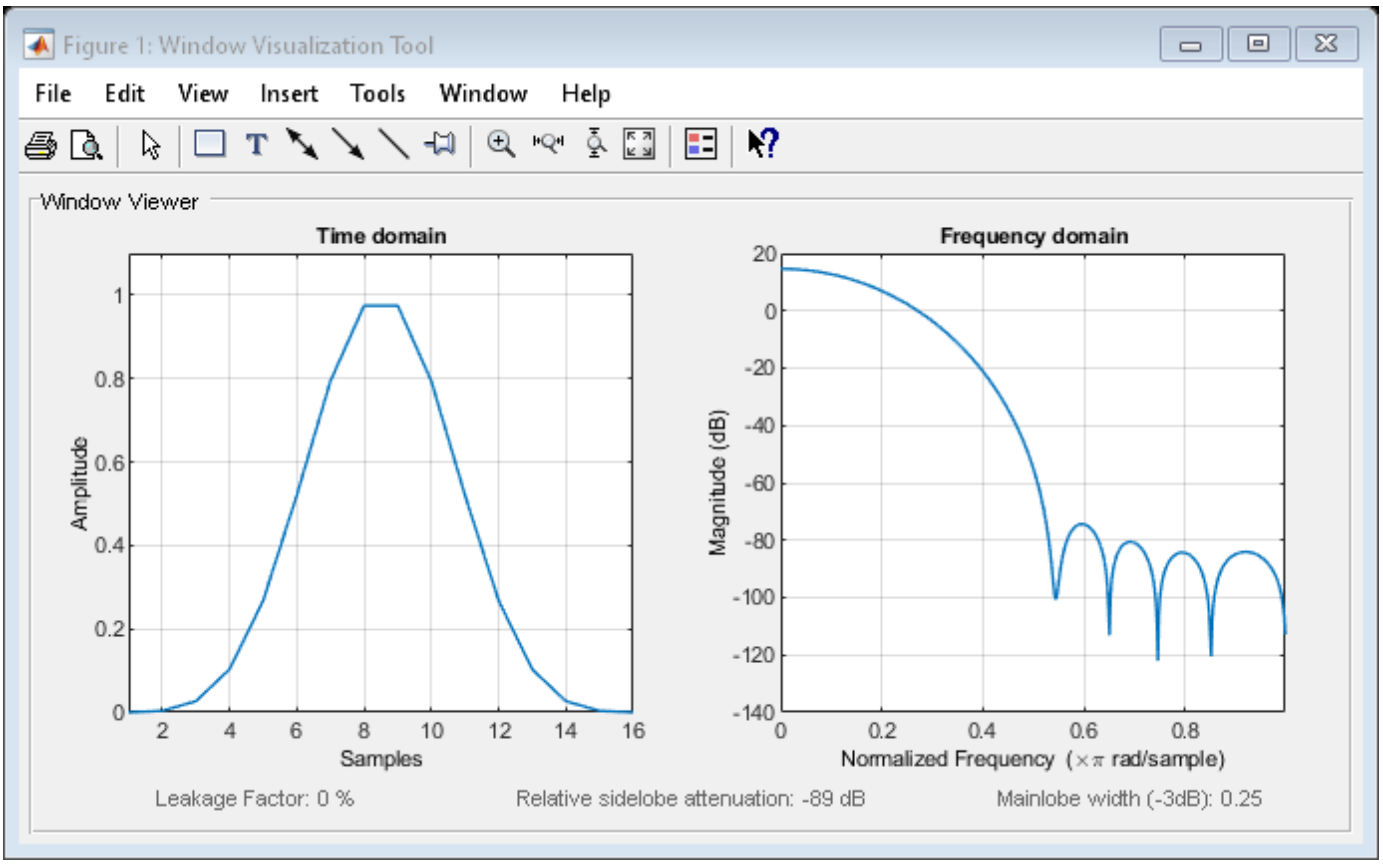
```
H = sigwin.blackmanharris(16);

win = generate(H)
```

```
win = 16×1

    0.0001
    0.0036
    0.0267
    0.1030
    0.2680
    0.5206
    0.7938
    0.9749
    0.9749
    0.7938
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Blackman-Harris Window    '
    '----------------------    '
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

`blackmanharris` | `window` | **WVTool**

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

Blackman-Harris Window

Generate a Blackman-Harris window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.blackmanharris(16);

win = generate(H)
```

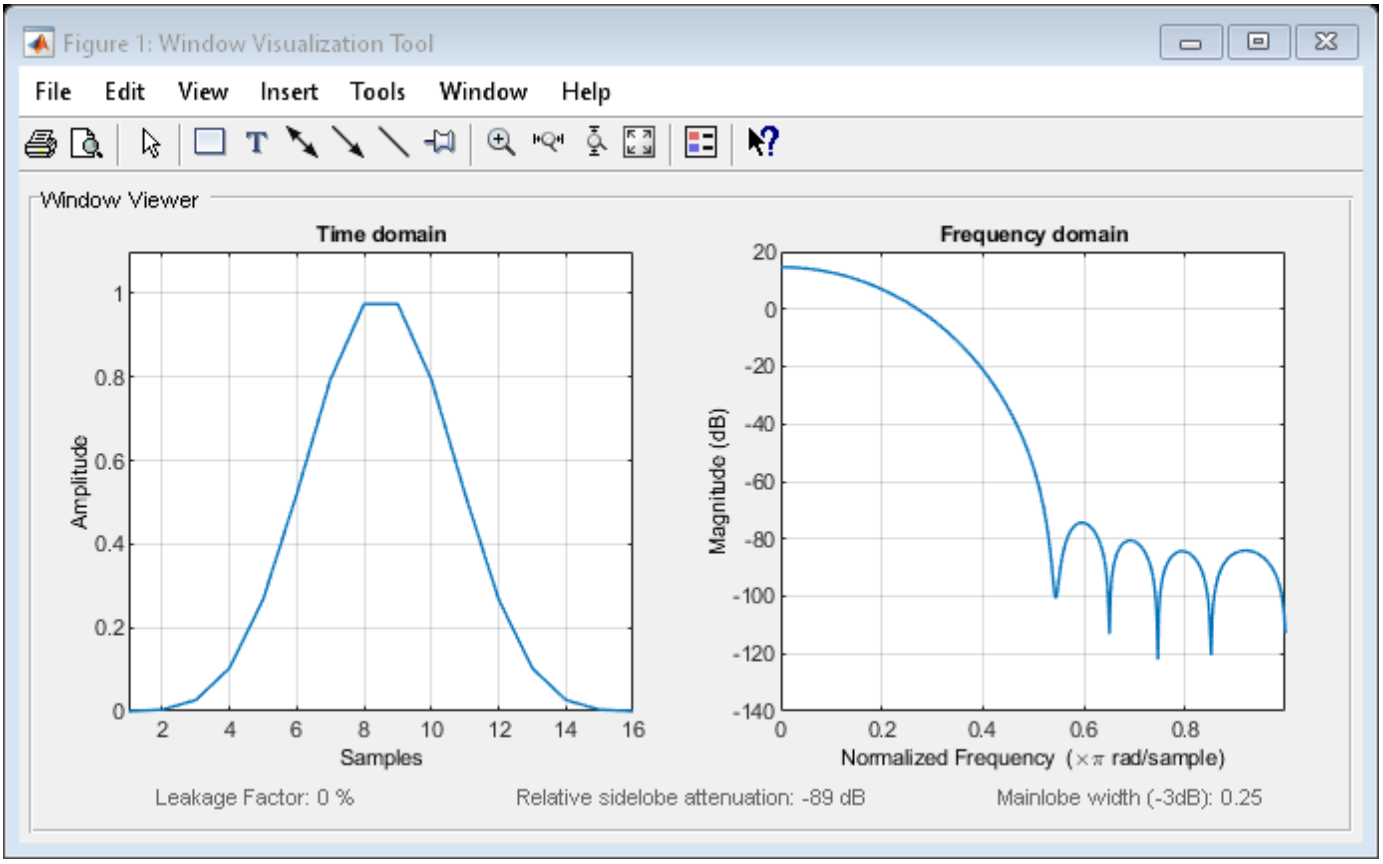
```
win = 16×1

    0.0001
    0.0036
    0.0267
    0.1030
    0.2680
    0.5206
    0.7938
    0.9749
    0.9749
    0.7938
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Blackman-Harris Window    '
    '----------------------    '
    'Length         : 16       '
```

```
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

`blackmanharris` | `window` | **WVTool**

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:

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

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

Bohman Window

Generate a Bohman window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

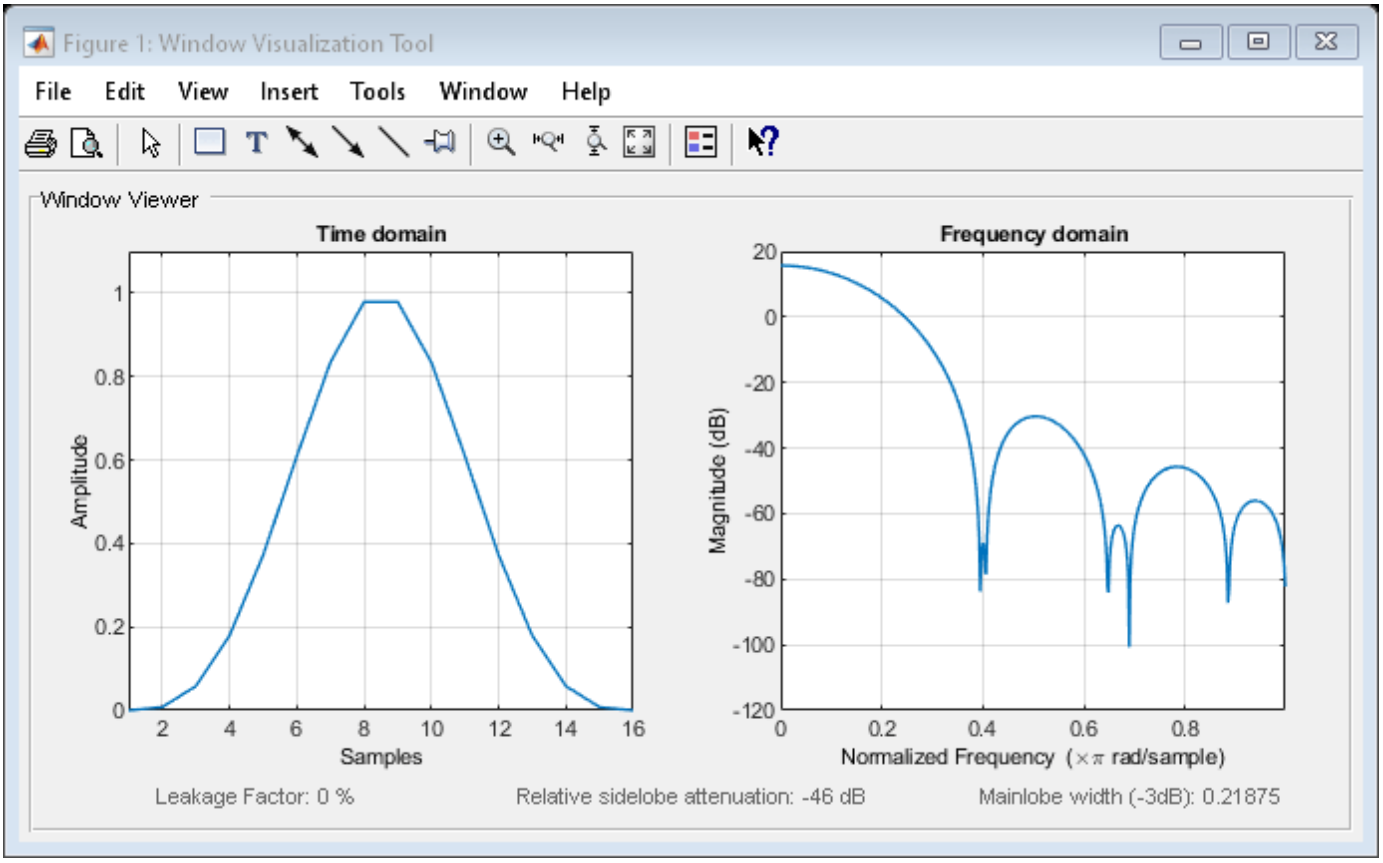
```
H = sigwin.bohmanwin(16);

win = generate(H)
```

```
win = 16×1

         0
    0.0077
    0.0581
    0.1791
    0.3723
    0.6090
    0.8343
    0.9791
    0.9791
    0.8343
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x13 char array
    'Bohman Window'
    '-------------'
    'Length  : 16 '
```

```
wvtool(H)
```



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**

Topics

“Windows”
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

Bohman Window

Generate a Bohman window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

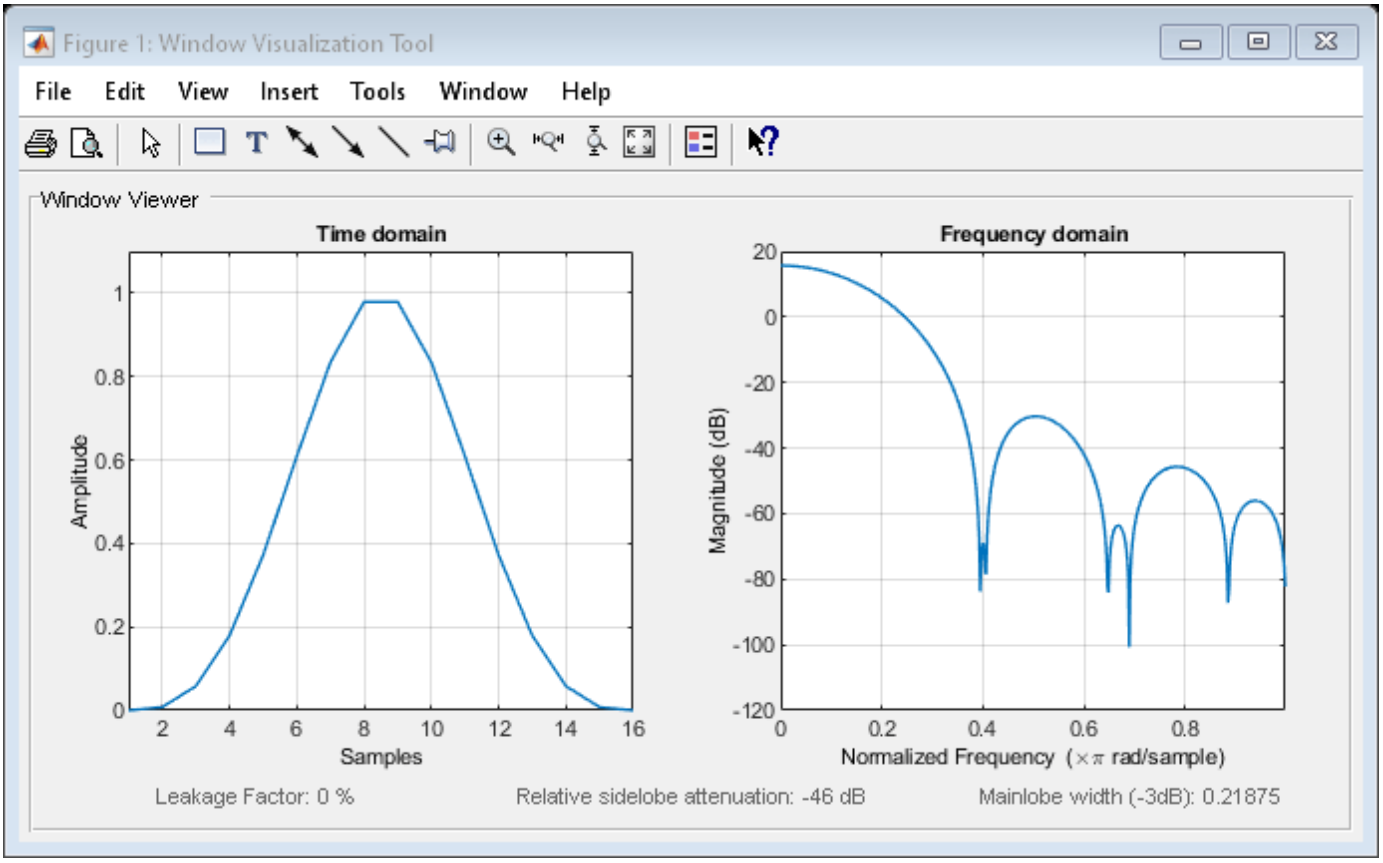
```
H = sigwin.bohmanwin(16);

win = generate(H)
```

```
win = 16×1

         0
    0.0077
    0.0581
    0.1791
    0.3723
    0.6090
    0.8343
    0.9791
    0.9791
    0.8343
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x13 char array
    'Bohman Window'
    '-------------'
    'Length  : 16 '
```

```
wvtool(H)
```



See Also

`bohmanwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Bohman Window

Generate a Bohman window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

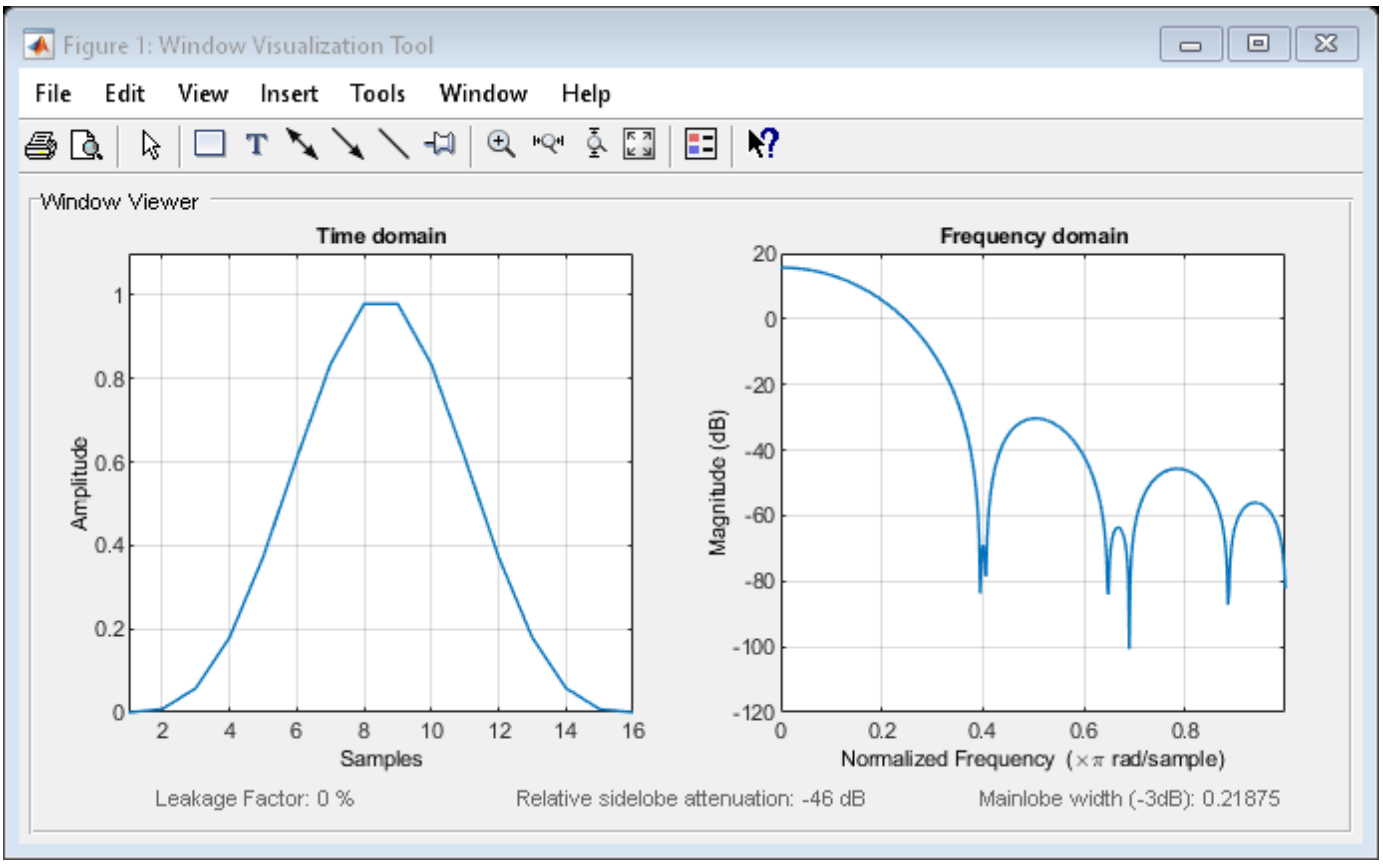
```
H = sigwin.bohmanwin(16);

win = generate(H)
```

```
win = 16×1

         0
    0.0077
    0.0581
    0.1791
    0.3723
    0.6090
    0.8343
    0.9791
    0.9791
    0.8343
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x13 char array
    'Bohman Window'
    '-------------'
    'Length  : 16 '
```

```
wvtool(H)
```



See Also

`bohmanwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Bohman Window

Generate a Bohman window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

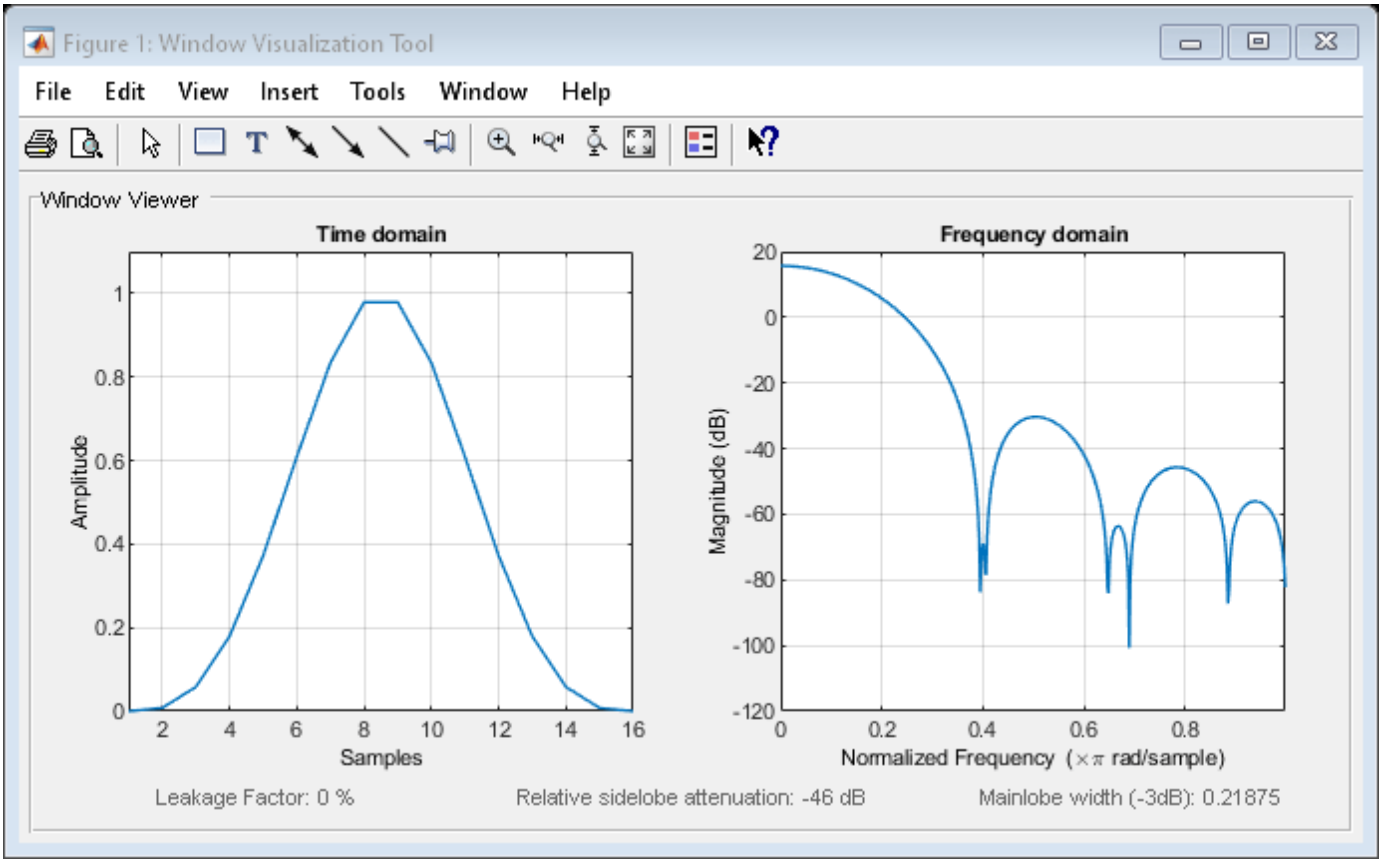
```
H = sigwin.bohmanwin(16);

win = generate(H)
```

```
win = 16×1

         0
    0.0077
    0.0581
    0.1791
    0.3723
    0.6090
    0.8343
    0.9791
    0.9791
    0.8343
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x13 char array
    'Bohman Window'
    '--------------'
```

```
        'Length  : 16 '

wvtool(H)
```



See Also

`bohmanwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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:

$$\widehat{W}(k) = (-1)^k \frac{\cos[N\cos^{-1}[\beta\cos(\pi k/N)]]}{\cosh[N\cosh^{-1}(\beta)]}, \quad 0 \le k \le N-1$$

where

$$\beta = \cos[1/N\cosh^{-1}(10^{\alpha})]$$

α determines the level of the sidelobe attenuation. The level of the sidelobe attenuation is equal to -20α. 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 $\widehat{W}(k)$ 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

generate	Generates Dolph-Chebyshev window
info	Display information about Dolph–Chebyshev window object
winwrite	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

Dolph-Chebyshev Window

Generate a Dolph-Chebyshev window of length $N = 16$. Specify a relative sidelobe attenuation of 40 dB. Return the window values as a column vector. Show information about the window object. Display the window.

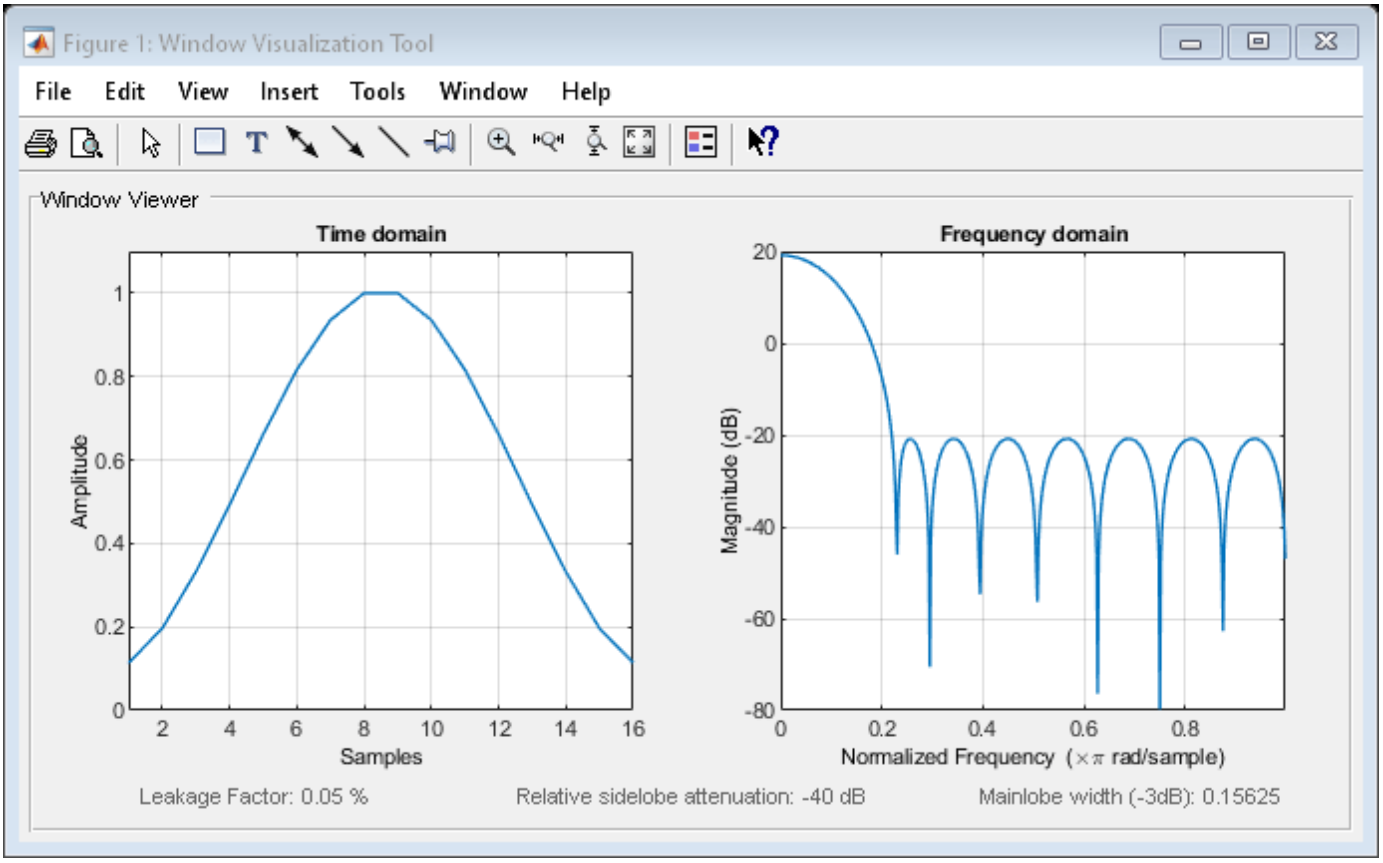
```
H = sigwin.chebwin(16,40);

win = generate(H)
```

```
win = 16×1

    0.1138
    0.1964
    0.3319
    0.4926
    0.6613
    0.8163
    0.9353
    1.0000
    1.0000
    0.9353
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Chebyshev Window          '
    '----------------          '
    'Length                : 16'
    'Sidelobe Attenuation  : 40'
```

```
wvtool(H)
```



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**

Topics

“Windows”
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

Dolph-Chebyshev Window

Generate a Dolph-Chebyshev window of length $N = 16$. Specify a relative sidelobe attenuation of 40 dB. Return the window values as a column vector. Show information about the window object. Display the window.

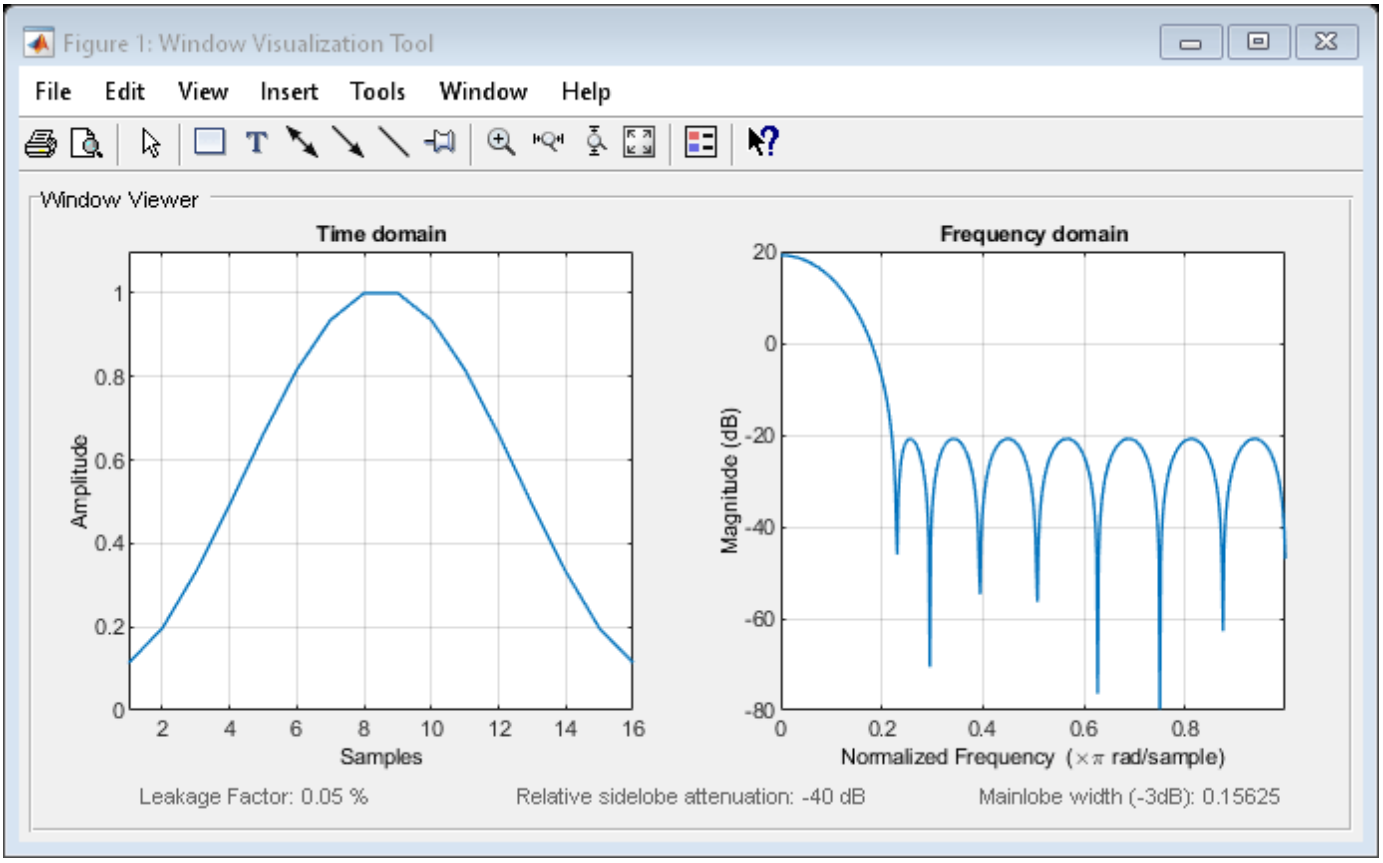
```
H = sigwin.chebwin(16,40);

win = generate(H)
```

```
win = 16×1

    0.1138
    0.1964
    0.3319
    0.4926
    0.6613
    0.8163
    0.9353
    1.0000
    1.0000
    0.9353
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Chebyshev Window          '
    '----------------          '
    'Length                : 16'
    'Sidelobe Attenuation  : 40'
```

```
wvtool(H)
```



See Also

`chebwin` | `window` | **WVTool**

Topics

"Windows"
Class Attributes
Property Attributes

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

Dolph-Chebyshev Window

Generate a Dolph-Chebyshev window of length $N = 16$. Specify a relative sidelobe attenuation of 40 dB. Return the window values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.chebwin(16,40);

win = generate(H)
```

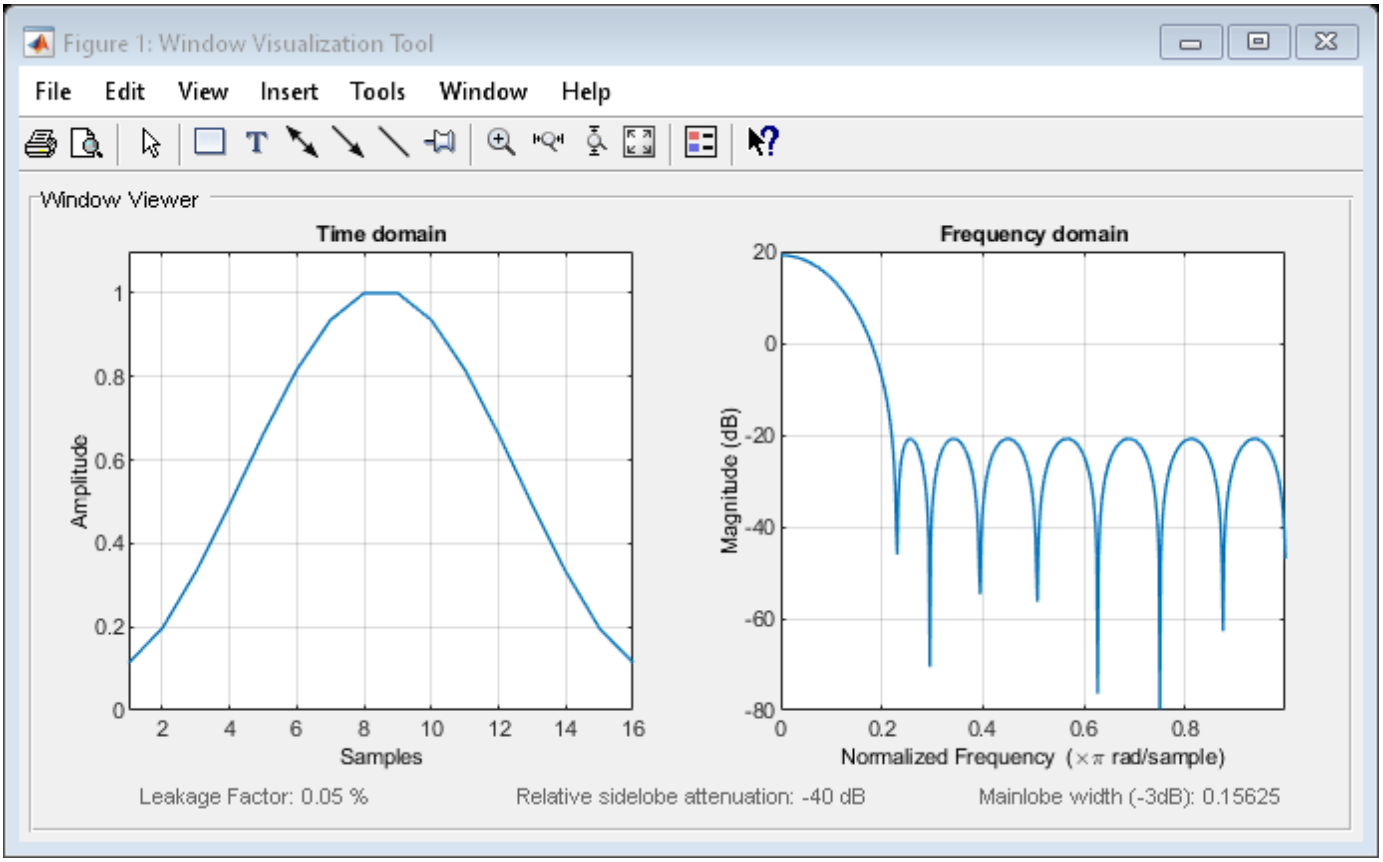
```
win = 16×1

    0.1138
    0.1964
    0.3319
    0.4926
    0.6613
    0.8163
    0.9353
    1.0000
    1.0000
    0.9353
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Chebyshev Window          '
    '----------------          '
```

```
    'Length                : 16'
    'Sidelobe Attenuation  : 40'
```

```
wvtool(H)
```



See Also

`chebwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Dolph-Chebyshev Window

Generate a Dolph-Chebyshev window of length $N = 16$. Specify a relative sidelobe attenuation of 40 dB. Return the window values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.chebwin(16,40);

win = generate(H)
```

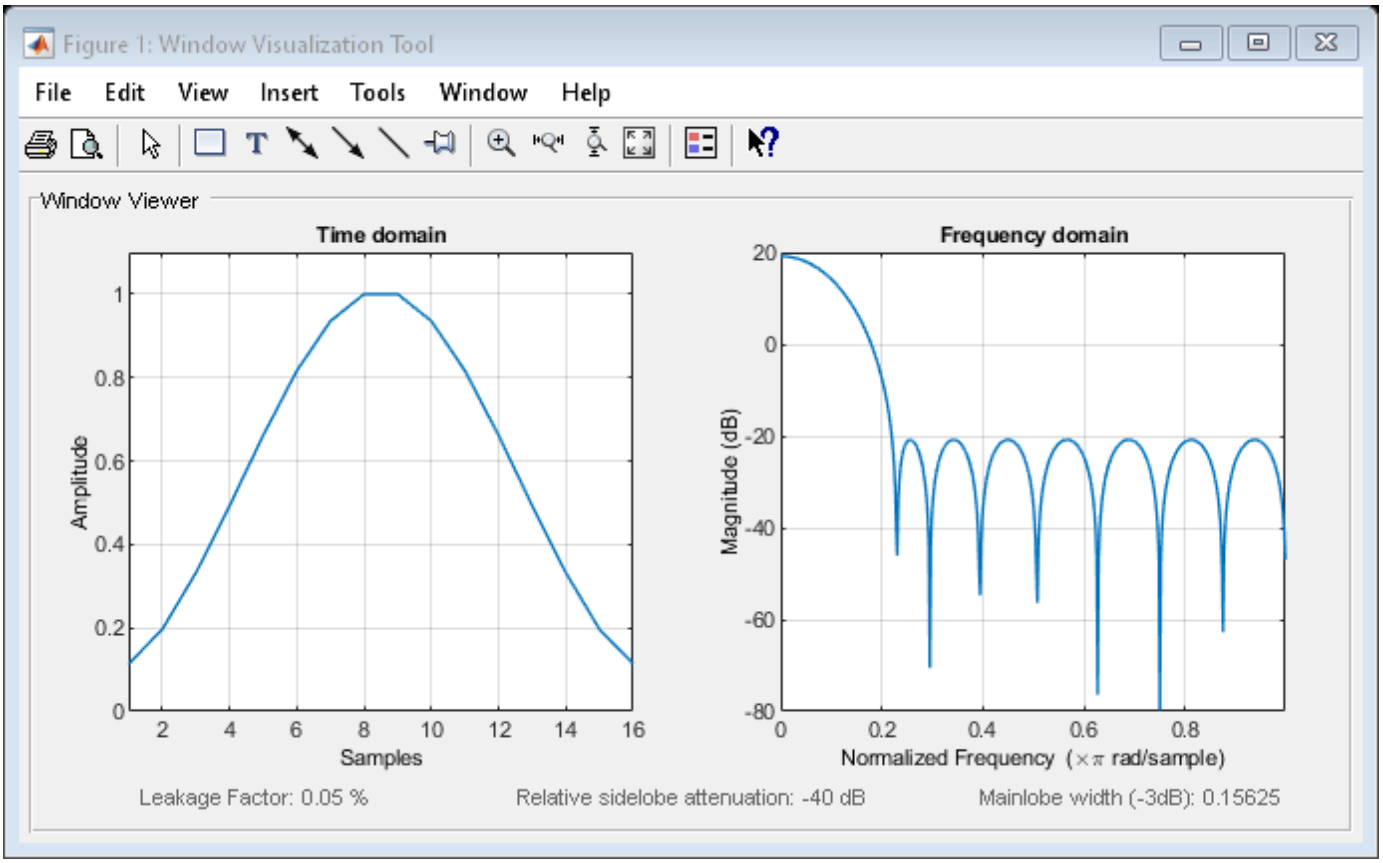
```
win = 16×1

    0.1138
    0.1964
    0.3319
    0.4926
    0.6613
    0.8163
    0.9353
    1.0000
    1.0000
    0.9353
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Chebyshev Window          '
```

```
'----------------           '
'Length                : 16'
'Sidelobe Attenuation  : 40'
```

```
wvtool(H)
```



See Also

`chebwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Flat Top Window

Generate a flat top window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

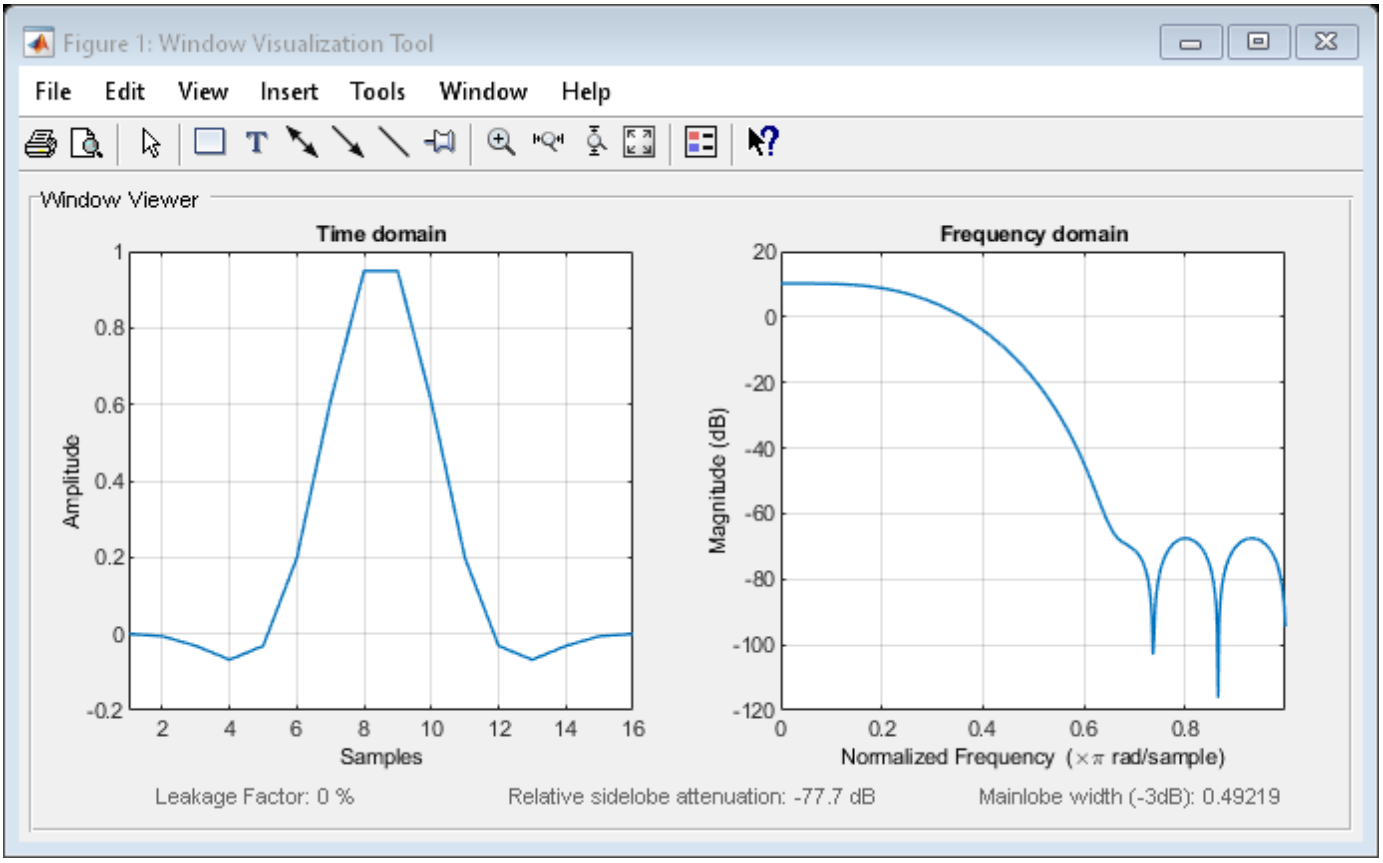
```
H = sigwin.flattopwin(16);

win = generate(H)
```

```
win = 16×1

   -0.0004
   -0.0061
   -0.0314
   -0.0677
   -0.0316
    0.1982
    0.6069
    0.9487
    0.9487
    0.6069
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Flat Top Window           '
    '---------------           '
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



Algorithms

The following equation defines the flat top window of length *N*:

$$w(n) = a_0 - a_1\cos\frac{2\pi n}{N-1} + a_2\cos\frac{4\pi n}{N-1} - a_3\cos\frac{6\pi n}{N-1} + a_4\cos\frac{8\pi n}{N-1}, \quad 0 \le n \le M-1,$$

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

The second half of the symmetric flat top window $M \le n \le N - 1$ 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
a_0	0.21557895
a_1	0.41663158

Coefficient	Value
a_2	0.277263158
a_3	0.083578947
a_4	0.006947368

References

Oppenheim, Alan V., and Ronald W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

See Also

`flattopwin` | `window` | **WVTool**

Topics

“Windows”
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

Flat Top Window

Generate a flat top window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

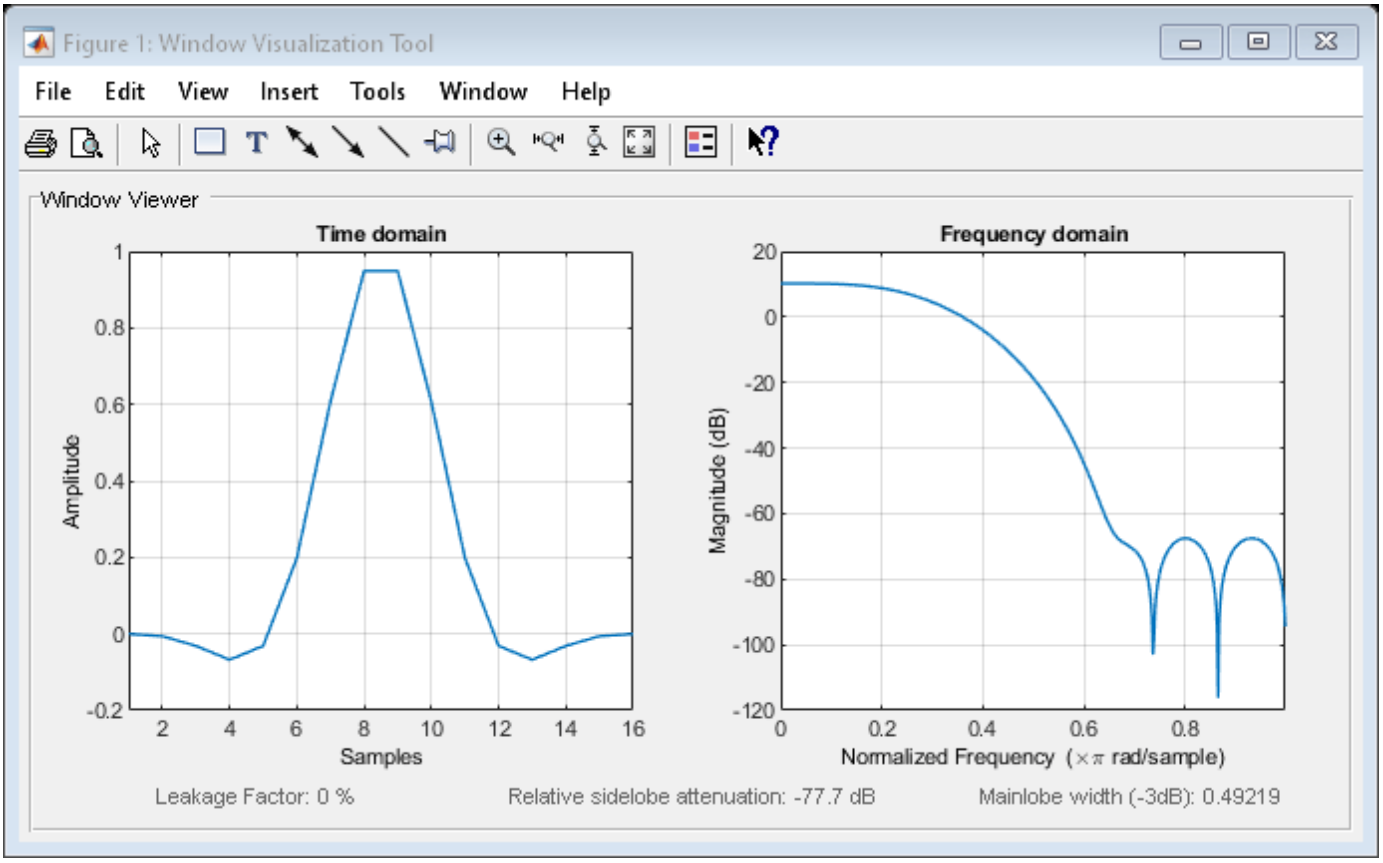
```
H = sigwin.flattopwin(16);

win = generate(H)
```

```
win = 16×1

   -0.0004
   -0.0061
   -0.0314
   -0.0677
   -0.0316
    0.1982
    0.6069
    0.9487
    0.9487
    0.6069
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Flat Top Window           '
    '---------------           '
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

`flattopwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Flat Top Window

Generate a flat top window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

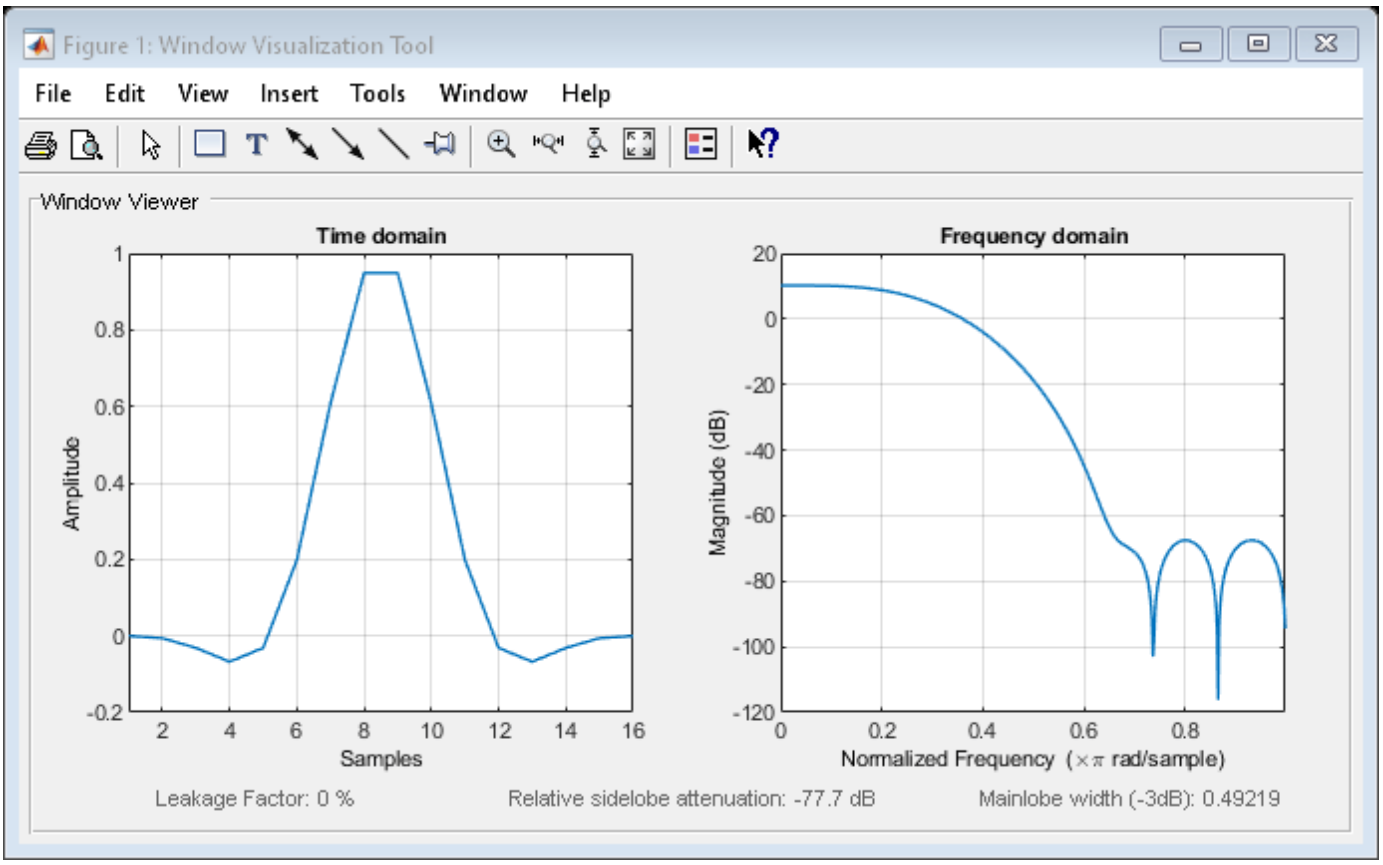
```
H = sigwin.flattopwin(16);

win = generate(H)
```

```
win = 16×1

   -0.0004
   -0.0061
   -0.0314
   -0.0677
   -0.0316
    0.1982
    0.6069
    0.9487
    0.9487
    0.6069
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Flat Top Window           '
    '----------------          '
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

`flattopwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Flat Top Window

Generate a flat top window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.flattopwin(16);

win = generate(H)
```

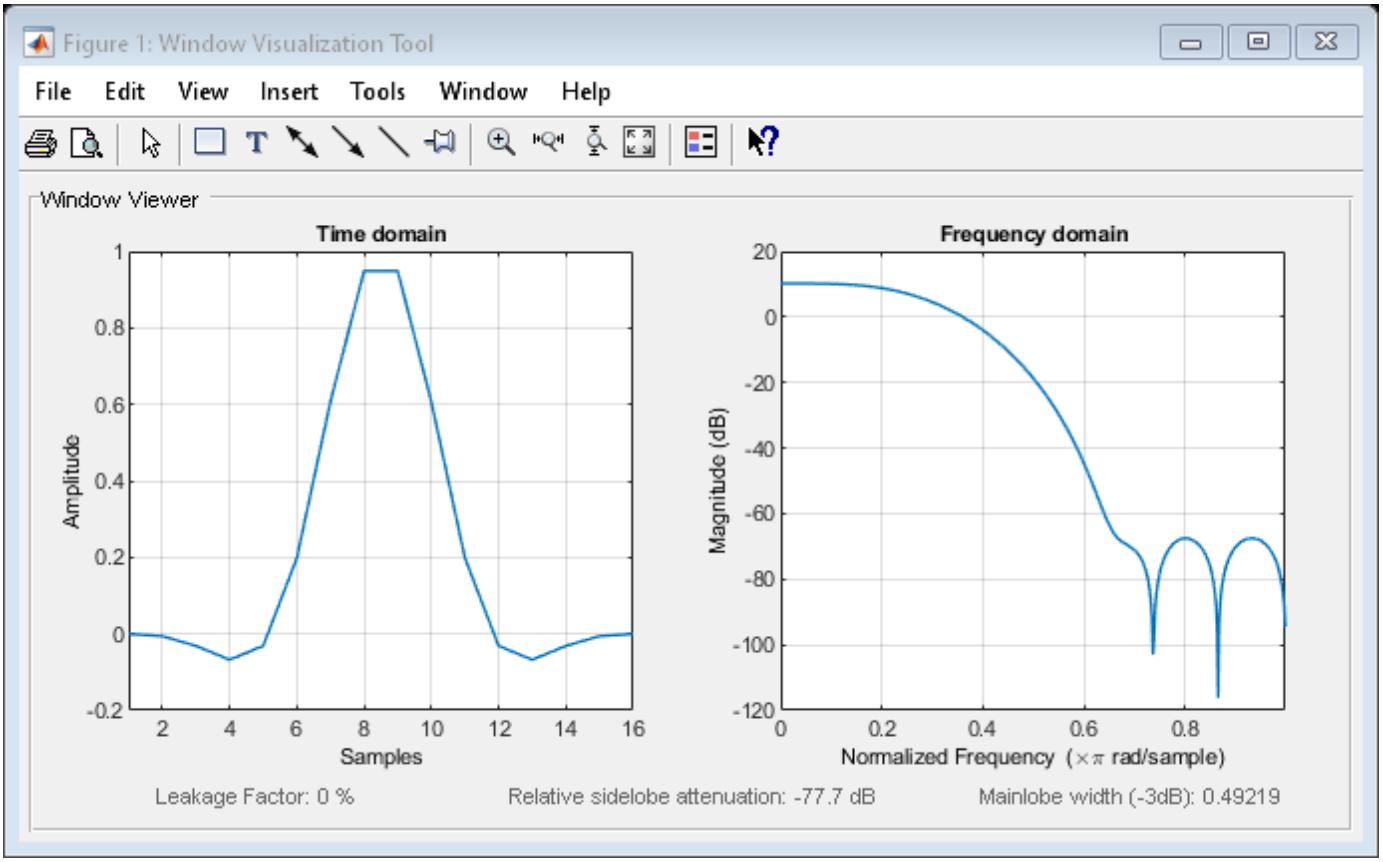
```
win = 16×1

   -0.0004
   -0.0061
   -0.0314
   -0.0677
   -0.0316
    0.1982
    0.6069
    0.9487
    0.9487
    0.6069
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Flat Top Window           '
    '----------------          '
```

```
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

`flattopwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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*:

$$w(x) = e^{-\frac{1}{2}(\alpha^2 x^2 / M^2)}, \quad -M \le x \le M$$

where *M=(N-1)/2* and *x* is a linearly spaced vector of length *N*.

Equating α with the usual standard deviation of a Gaussian value, σ, 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

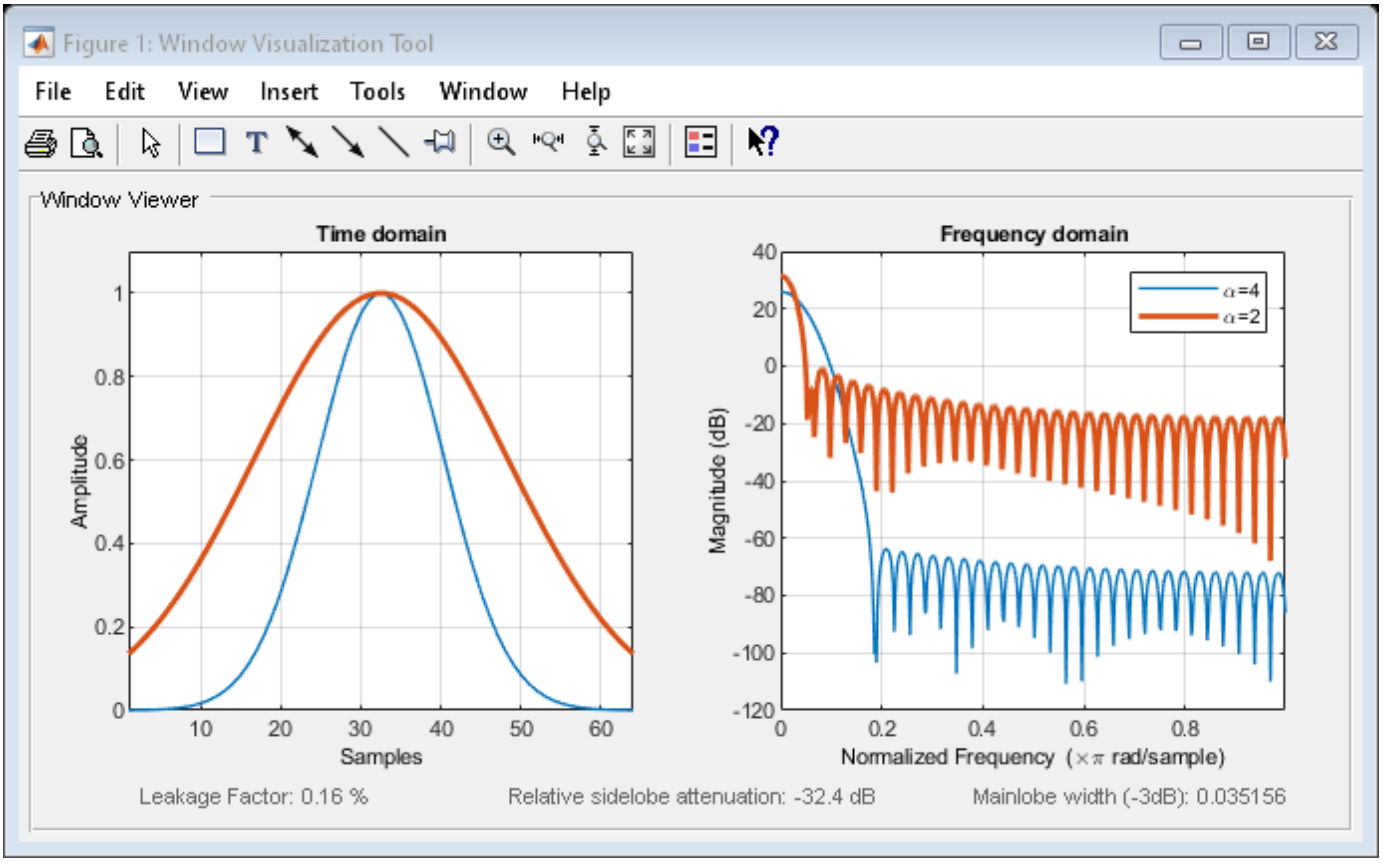
Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Gaussian Windows

Generate two Gaussian windows of length $N = 64$. Specify α values of 4 and 2. Show information about the window objects. Display the windows.

```
H4 = sigwin.gausswin(64,4);
H2 = sigwin.gausswin(64,2);

wvt = wvtool(H4,H2);
legend(wvt.CurrentAxes,'\alpha=4','\alpha=2')
```



The window with $\alpha = 4$ has a wider mainlobe and less sidelobe energy.

Generate a Gaussian window with $N = 16$ and $\alpha = 3$. Return the window values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.gausswin(16,3);

win = generate(H)
```

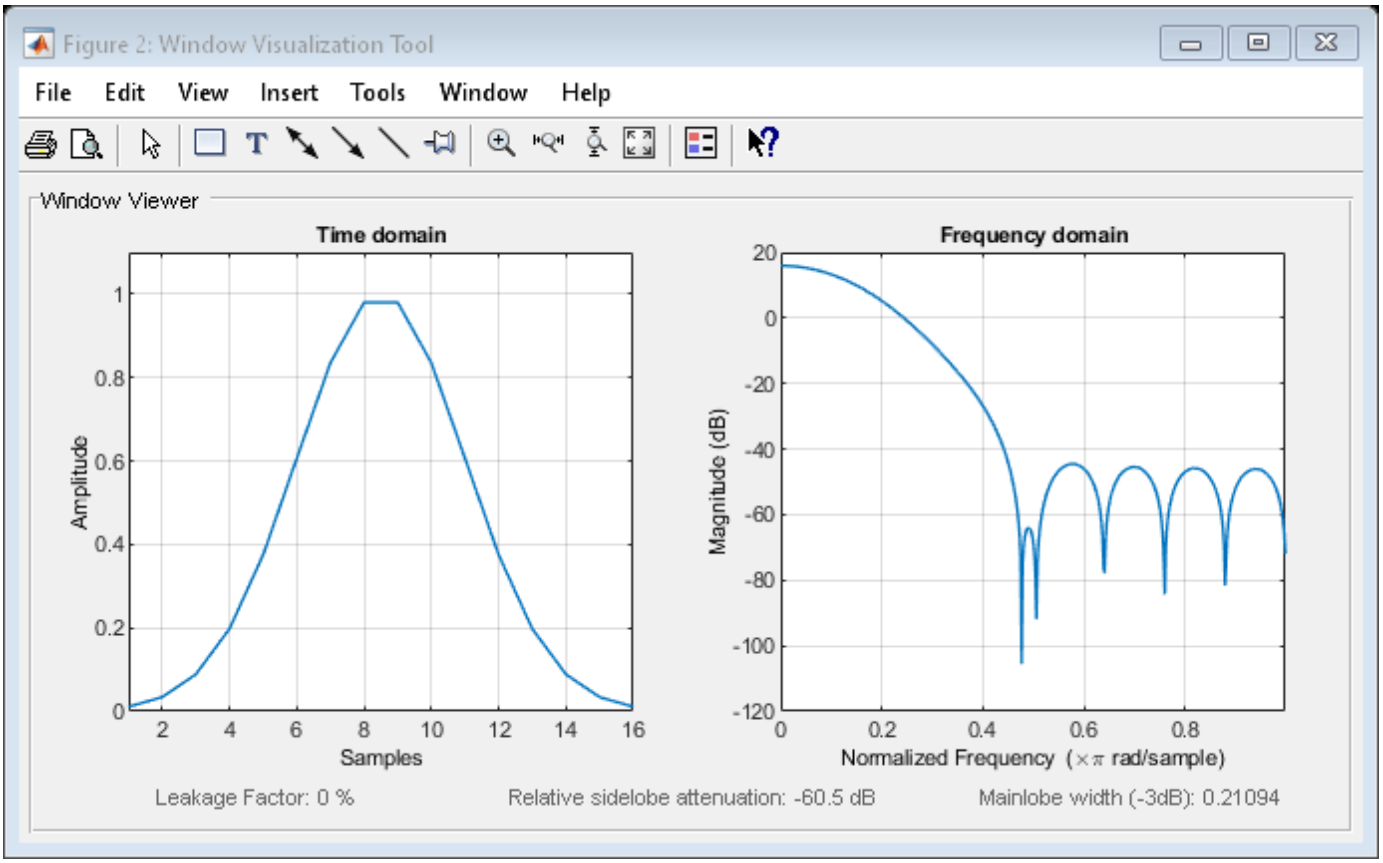
```
win = 16×1

    0.0111
    0.0340
    0.0889
    0.1979
    0.3753
    0.6065
    0.8353
    0.9802
    0.9802
    0.8353
       ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x15 char array
    'Gaussian Window'
```

```
'---------------'
'Length  : 16   '
'Alpha   : 3    '
```

```
wvtool(H)
```



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**

Topics
“Windows”
Class Attributes
Property Attributes

generate

Class: `sigwin.gausswin`
Package: `sigwin`

Generates Gaussian window

Syntax

```
win = generate(H)
```

Description

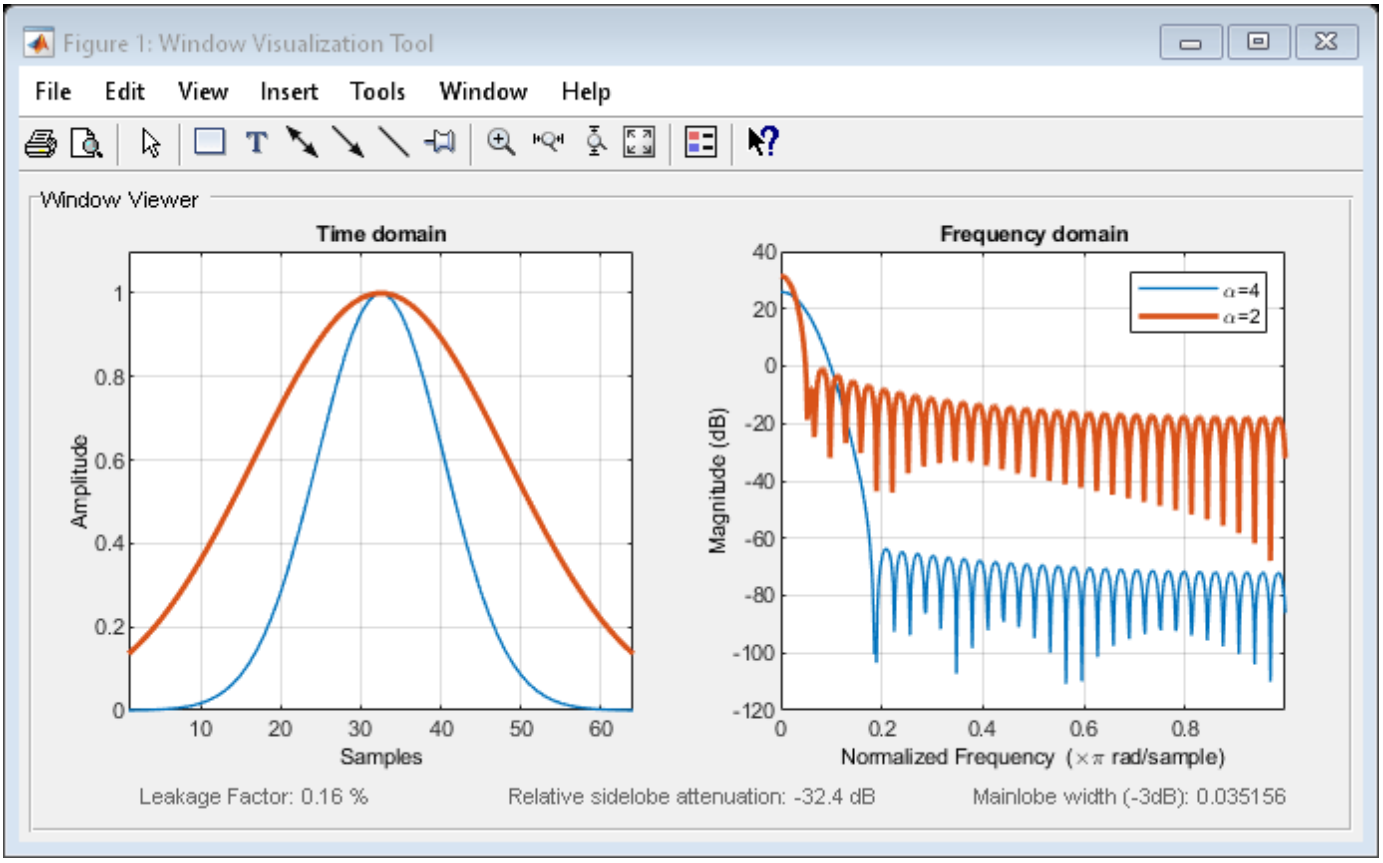
`win = generate(H)` returns the values of the Gaussian window object `H` as a double-precision column vector.

Examples

Gaussian Windows

Generate two Gaussian windows of length $N = 64$. Specify α values of 4 and 2. Show information about the window objects. Display the windows.

```
H4 = sigwin.gausswin(64,4);
H2 = sigwin.gausswin(64,2);

wvt = wvtool(H4,H2);
legend(wvt.CurrentAxes,'\alpha=4','\alpha=2')
```



The window with $\alpha = 4$ has a wider mainlobe and less sidelobe energy.

Generate a Gaussian window with $N = 16$ and $\alpha = 3$. Return the window values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.gausswin(16,3);

win = generate(H)
```

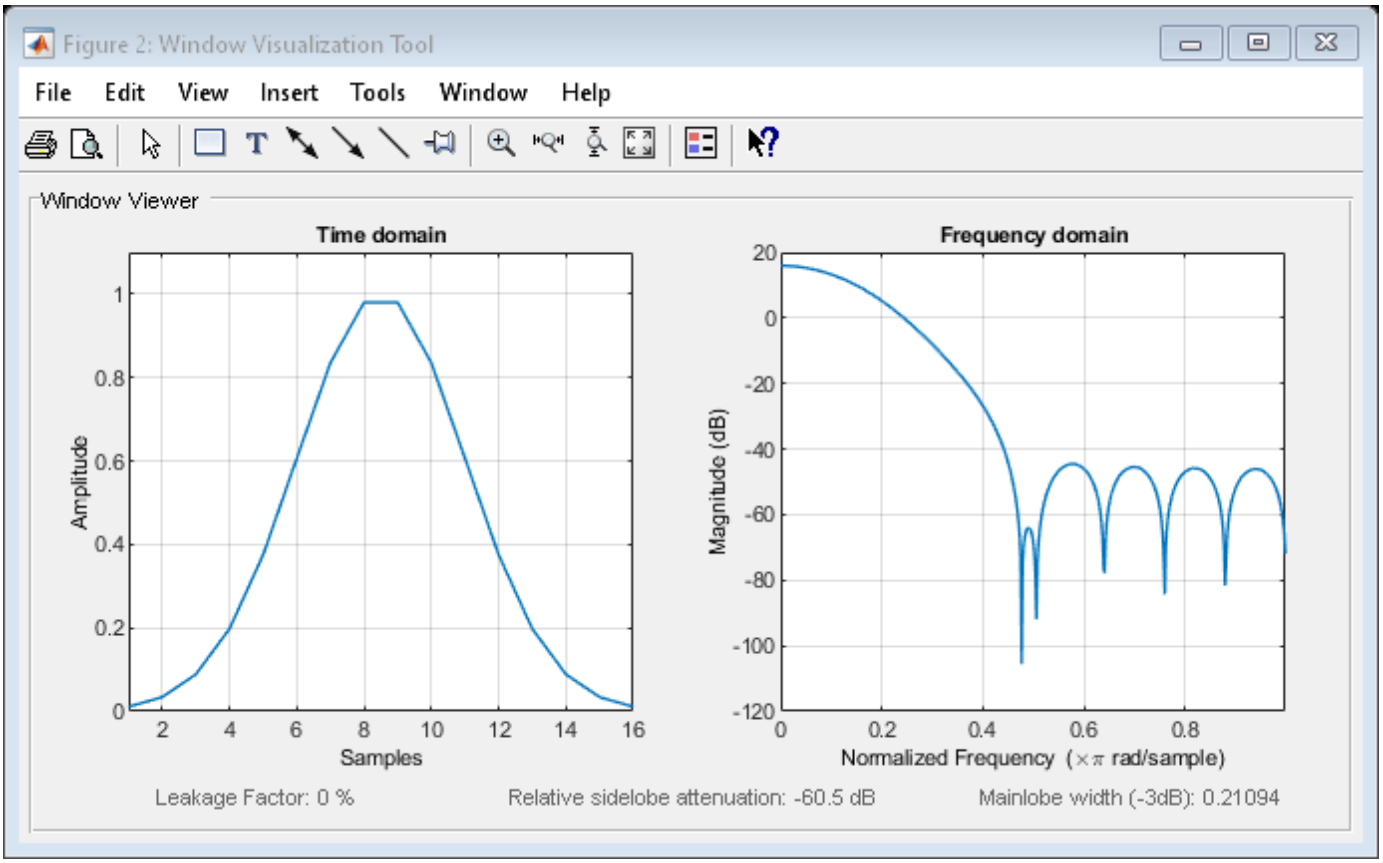
```
win = 16×1

    0.0111
    0.0340
    0.0889
    0.1979
    0.3753
    0.6065
    0.8353
    0.9802
    0.9802
    0.8353
       ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x15 char array
    'Gaussian Window'
```

```
'---------------'
'Length  : 16   '
'Alpha   : 3    '
```

```
wvtool(H)
```



See Also

`gausswin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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`.

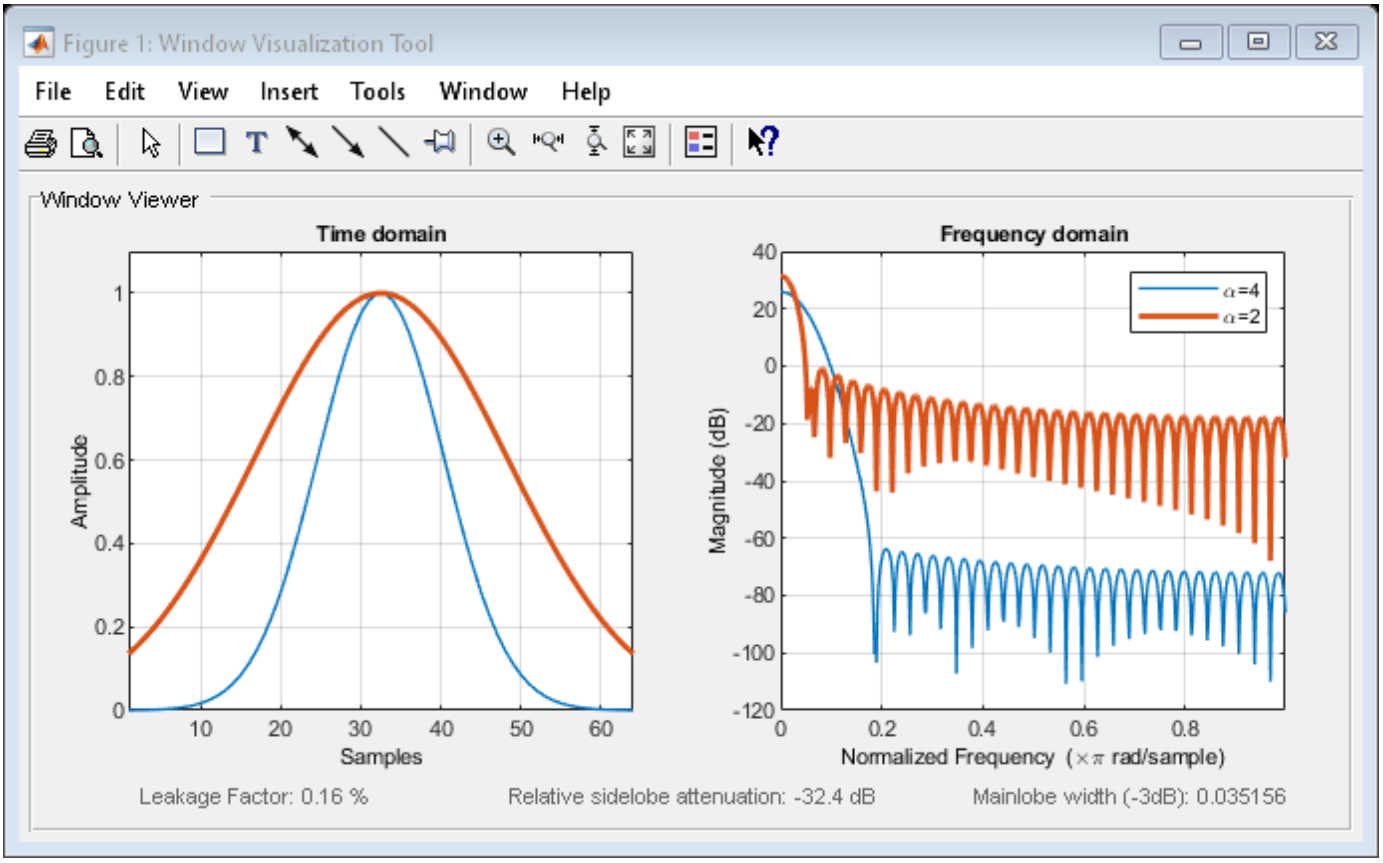
`info_win = info(H)` returns length and dispersion information for the Gaussian window object `H` in the character array `info_win`.

Examples

Gaussian Windows

Generate two Gaussian windows of length $N = 64$. Specify α values of 4 and 2. Show information about the window objects. Display the windows.

```
H4 = sigwin.gausswin(64,4);
H2 = sigwin.gausswin(64,2);

wvt = wvtool(H4,H2);
legend(wvt.CurrentAxes,'\alpha=4','\alpha=2')
```



The window with $\alpha = 4$ has a wider mainlobe and less sidelobe energy.

Generate a Gaussian window with $N = 16$ and $\alpha = 3$. Return the window values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.gausswin(16,3);

win = generate(H)
```

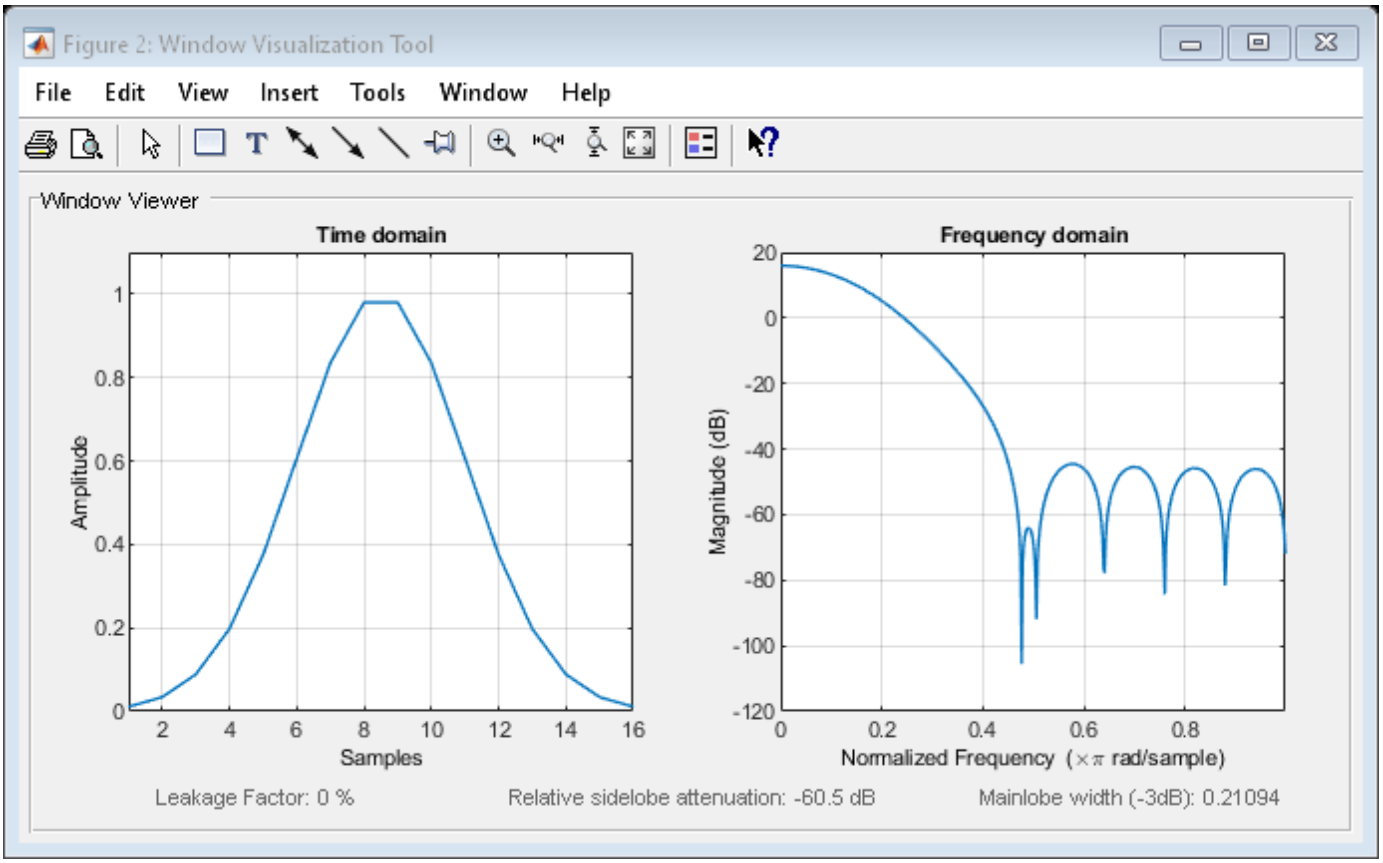
```
win = 16×1

    0.0111
    0.0340
    0.0889
    0.1979
    0.3753
    0.6065
    0.8353
    0.9802
    0.9802
    0.8353
       ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x15 char array
    'Gaussian Window'
```

```
'---------------'
'Length  : 16   '
'Alpha   : 3    '
```

```
wvtool(H)
```



See Also

`gausswin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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.

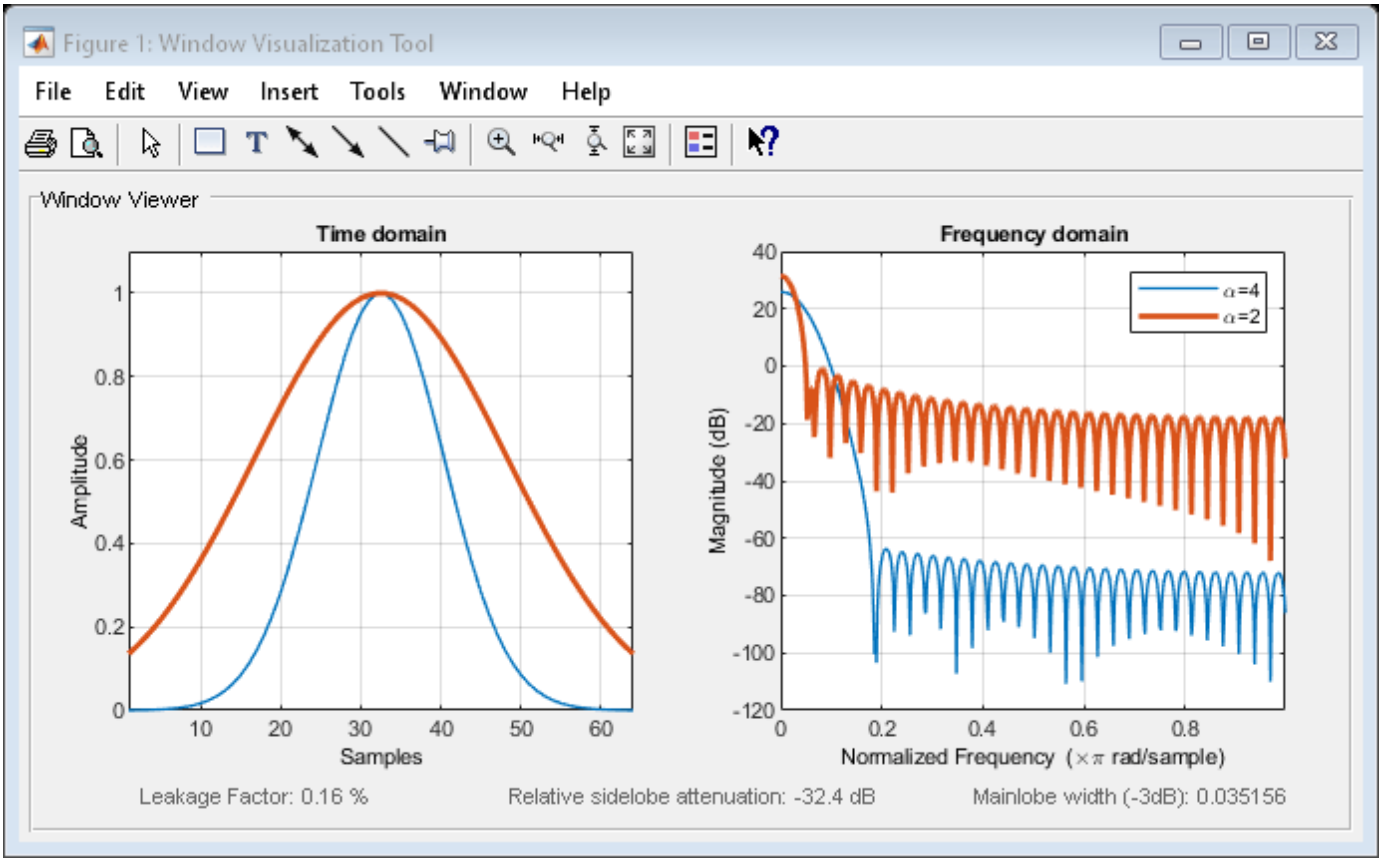
`winwrite(H,'filename')` saves the values of the Gaussian window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

Examples

Gaussian Windows

Generate two Gaussian windows of length $N = 64$. Specify α values of 4 and 2. Show information about the window objects. Display the windows.

```
H4 = sigwin.gausswin(64,4);
H2 = sigwin.gausswin(64,2);

wvt = wvtool(H4,H2);
legend(wvt.CurrentAxes,'\alpha=4','\alpha=2')
```



The window with $\alpha = 4$ has a wider mainlobe and less sidelobe energy.

Generate a Gaussian window with $N = 16$ and $\alpha = 3$. Return the window values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.gausswin(16,3);

win = generate(H)
```

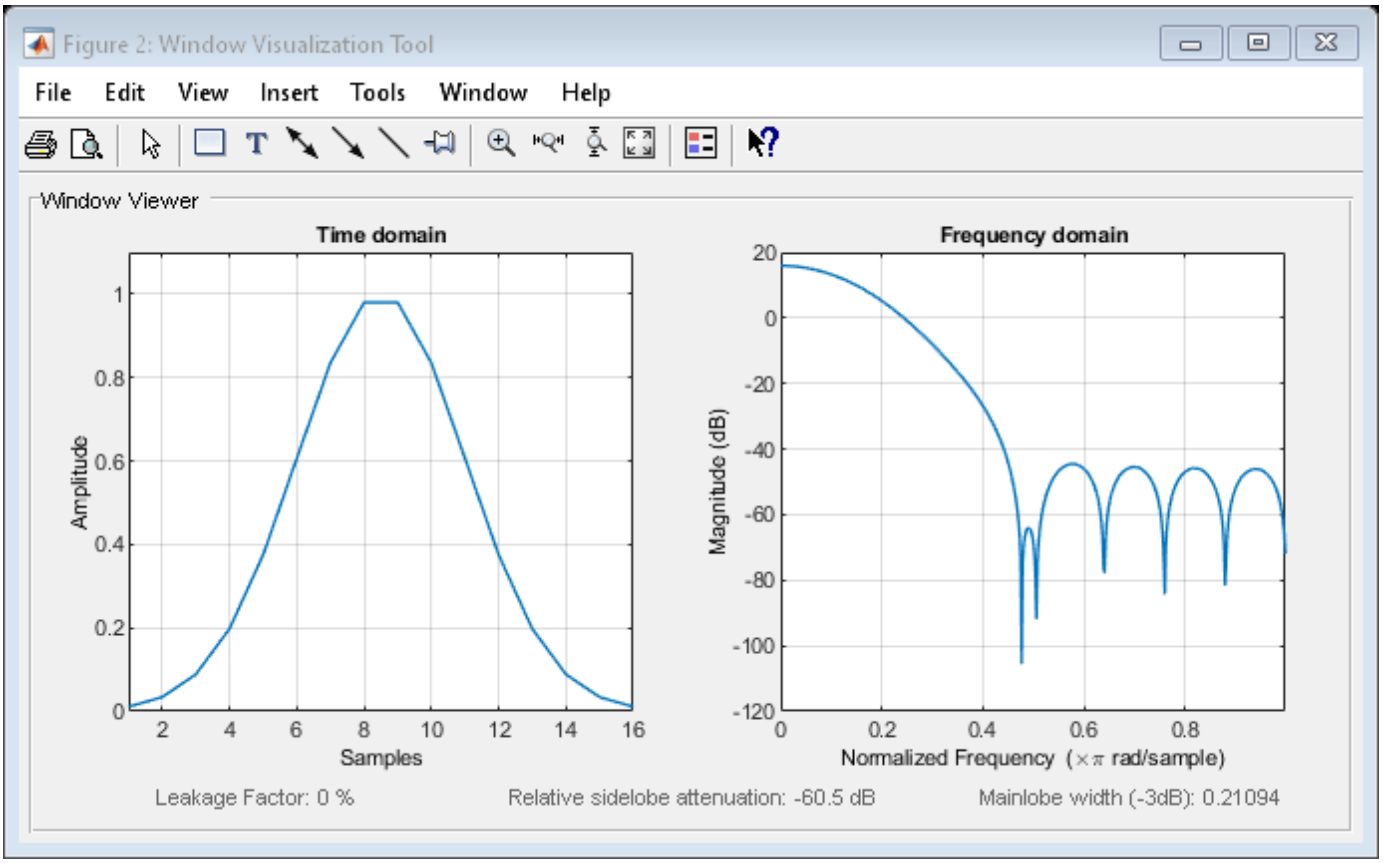
```
win = 16×1

    0.0111
    0.0340
    0.0889
    0.1979
    0.3753
    0.6065
    0.8353
    0.9802
    0.9802
    0.8353
       ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x15 char array
    'Gaussian Window'
```

```
'---------------'
'Length  : 16   '
'Alpha   : 3    '
```

```
wvtool(H)
```



See Also

`gausswin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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*:

$$w(n) = 0.54 - 0.46\cos\frac{2\pi n}{N-1}, \quad 0 \le n \le M-1$$

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

The second half of the symmetric Hamming window $M \le n \le N-1$ 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 *`Sampling_Flag`*. 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

Hamming Windows

Generate two Hamming windows:

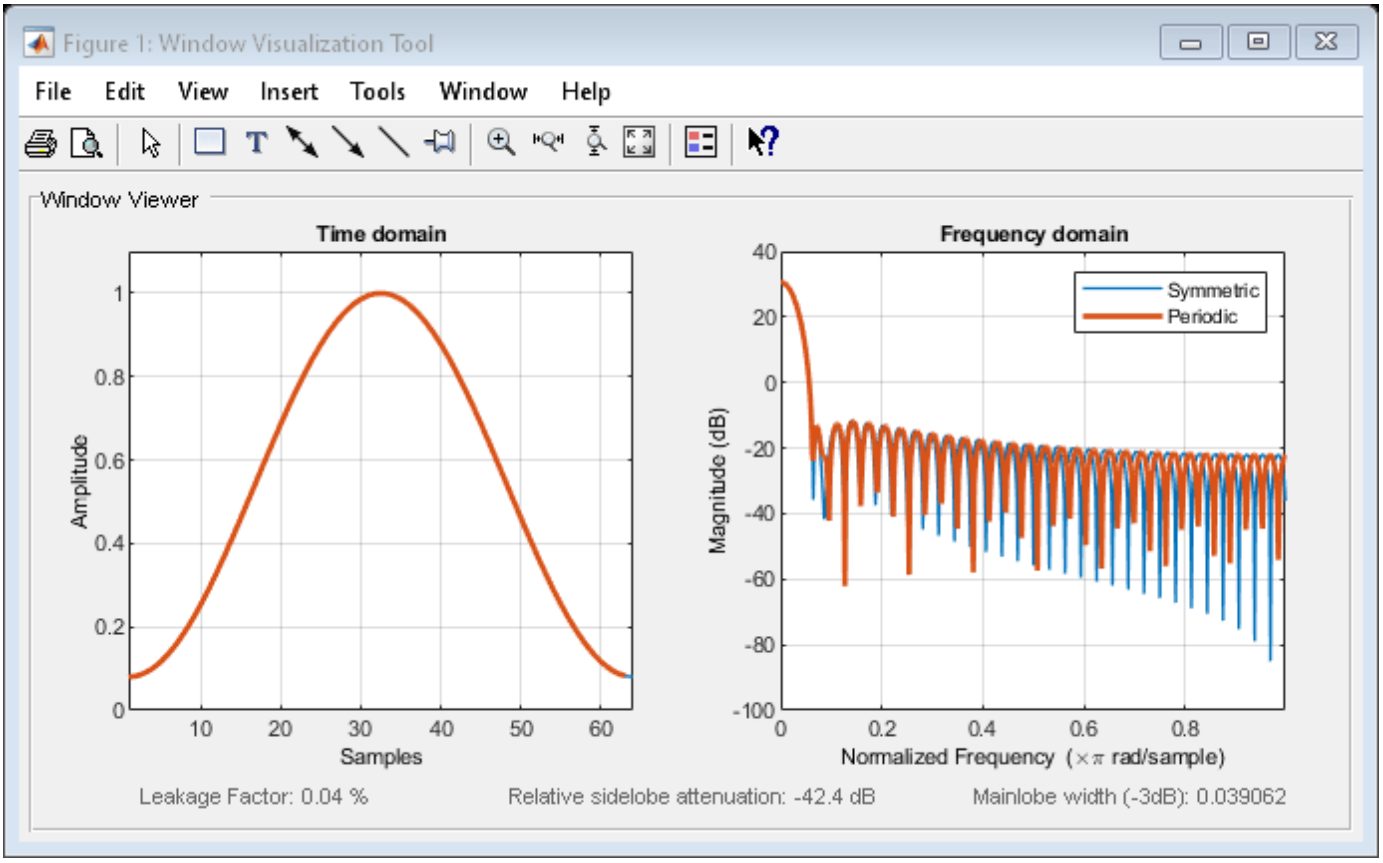
- The first window has $N = 64$ and is symmetric.
- The second window has $N = 63$ and is periodic.

Display the two windows.

```
Hs = sigwin.hamming(64,'symmetric');
Hp = sigwin.hamming(63,'periodic')
```

```
Hp =
            Name: 'Hamming'
    SamplingFlag: 'periodic'
          Length: 63
```

```
wvt = wvtool(Hs,Hp);
legend(wvt.CurrentAxes,'Symmetric','Periodic')
```



Generate a symmetric Hamming window with $N = 16$. Return the window values as a column vector. Show information about the window object. Display the window.

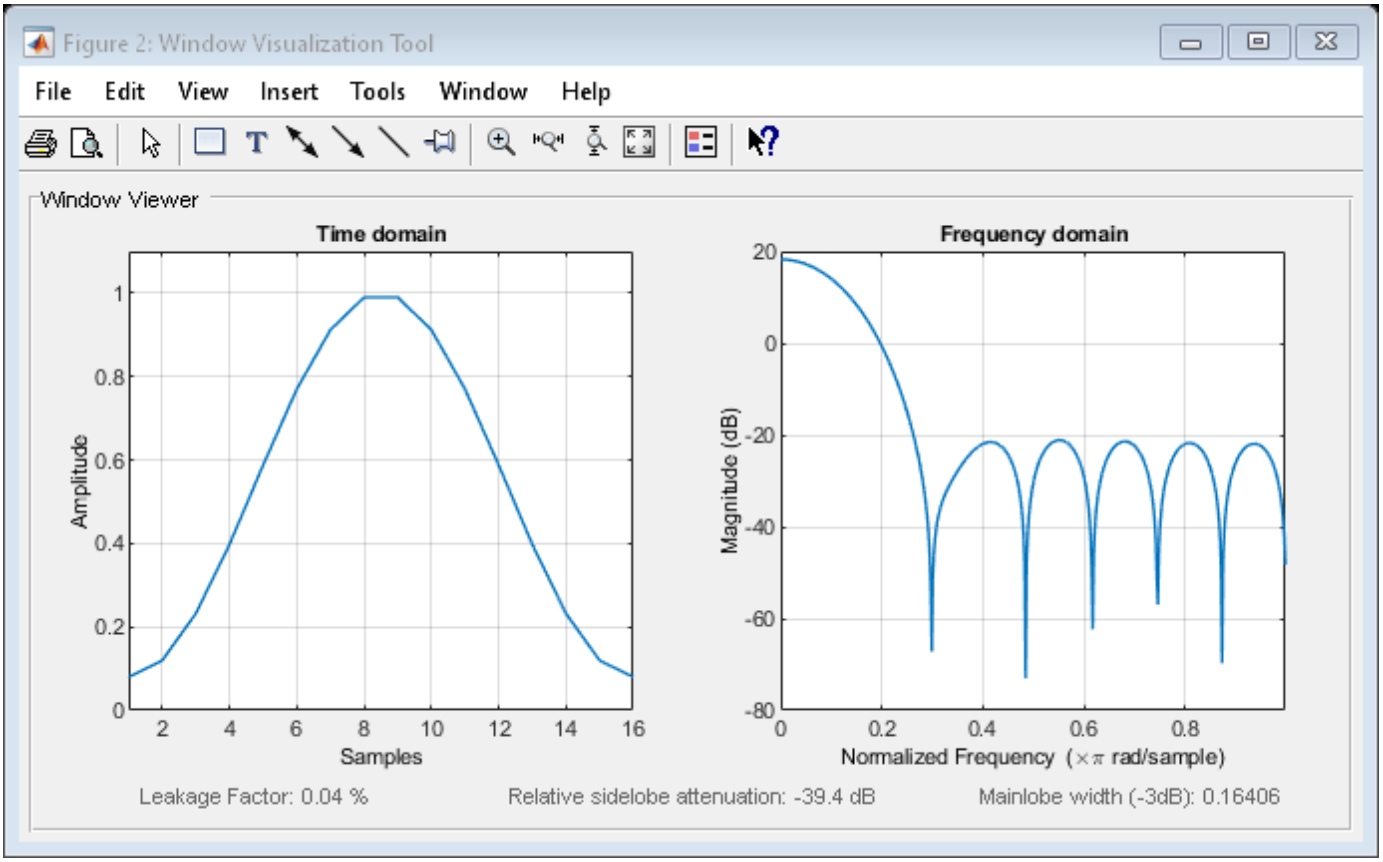
```
H = sigwin.hamming(16);

win = generate(H)
```

```
win = 16×1

    0.0800
    0.1198
    0.2322
    0.3979
    0.5881
    0.7700
    0.9121
    0.9899
    0.9899
    0.9121
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Hamming Window            '
    '--------------            '
```

```
    'Length         : 16       '
    'Sampling Flag  : symmetric'

wvtool(H)
```



References

Oppenheim, Alan V., and Ronald W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

See Also

`hamming` | `window` | **WVTool**

Topics
“Windows”
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

Hamming Windows

Generate two Hamming windows:

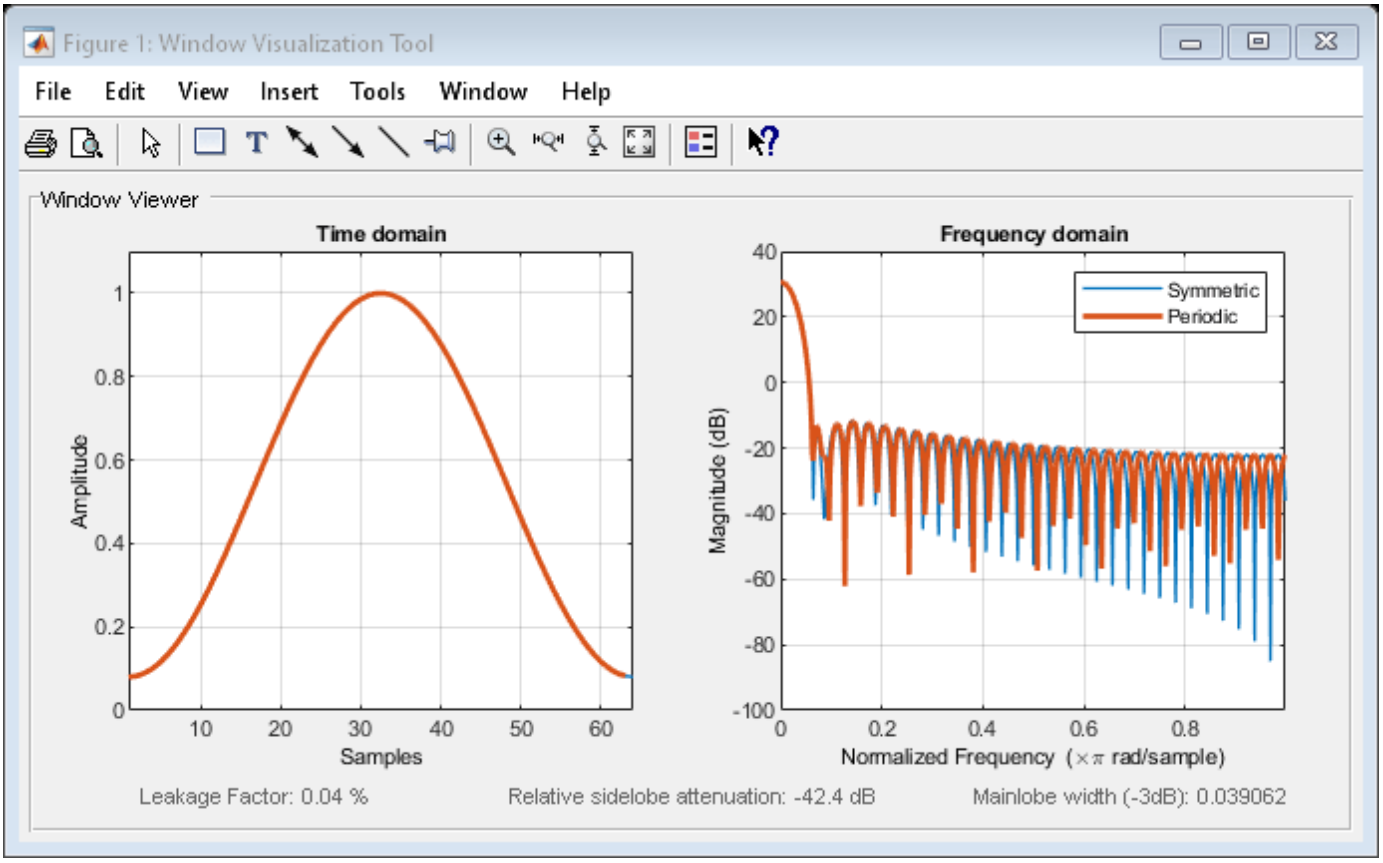
- The first window has $N = 64$ and is symmetric.
- The second window has $N = 63$ and is periodic.

Display the two windows.

```
Hs = sigwin.hamming(64,'symmetric');
Hp = sigwin.hamming(63,'periodic')
```

```
Hp =
            Name: 'Hamming'
    SamplingFlag: 'periodic'
          Length: 63
```

```
wvt = wvtool(Hs,Hp);
legend(wvt.CurrentAxes,'Symmetric','Periodic')
```



Generate a symmetric Hamming window with $N = 16$. Return the window values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.hamming(16);

win = generate(H)
```

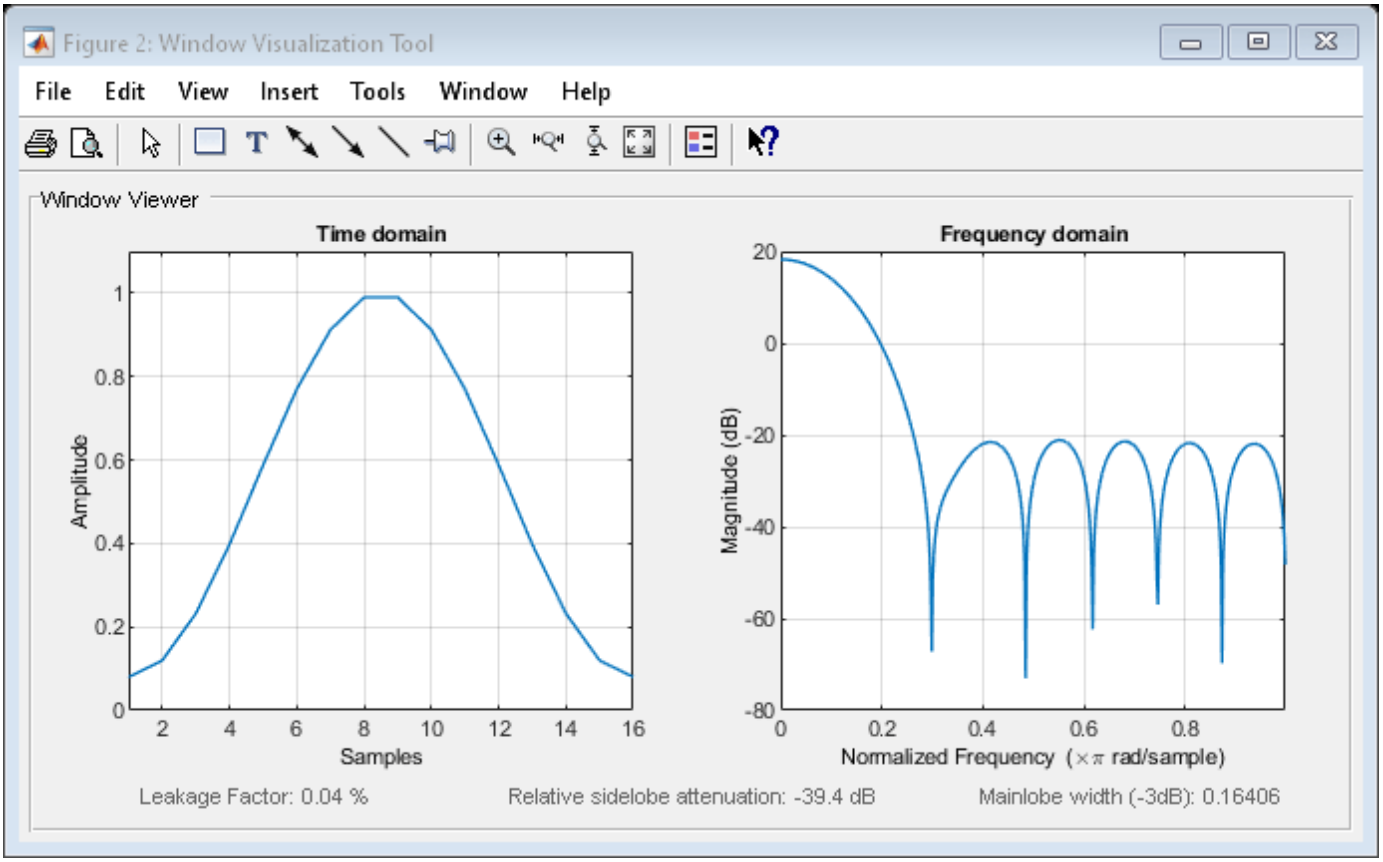
```
win = 16×1

    0.0800
    0.1198
    0.2322
    0.3979
    0.5881
    0.7700
    0.9121
    0.9899
    0.9899
    0.9121
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Hamming Window            '
    '--------------            '
```

```
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

`hamming` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Hamming Windows

Generate two Hamming windows:

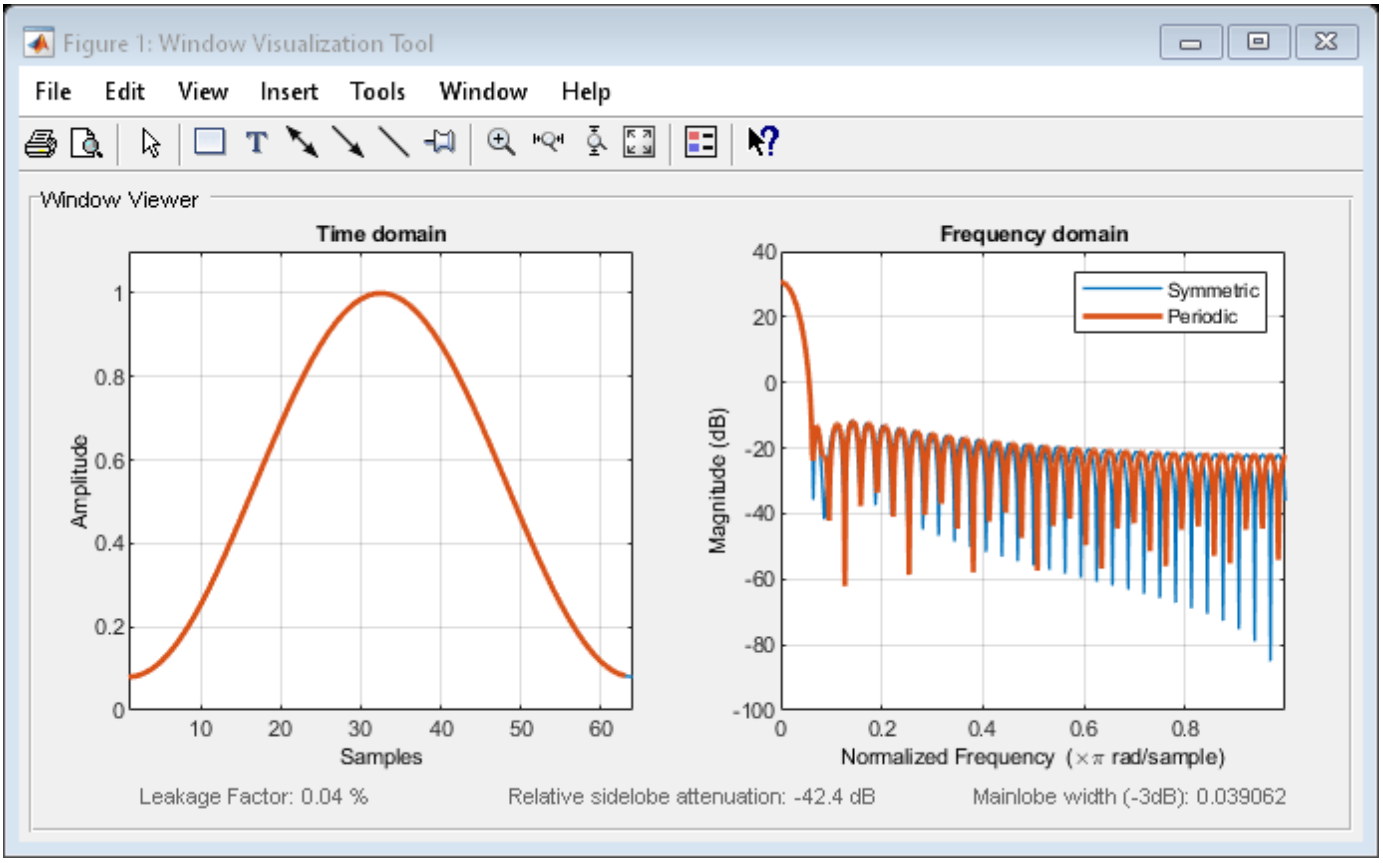
- The first window has $N = 64$ and is symmetric.
- The second window has $N = 63$ and is periodic.

Display the two windows.

```
Hs = sigwin.hamming(64,'symmetric');
Hp = sigwin.hamming(63,'periodic')
```

```
Hp =
            Name: 'Hamming'
    SamplingFlag: 'periodic'
          Length: 63
```

```
wvt = wvtool(Hs,Hp);
legend(wvt.CurrentAxes,'Symmetric','Periodic')
```



Generate a symmetric Hamming window with $N = 16$. Return the window values as a column vector. Show information about the window object. Display the window.

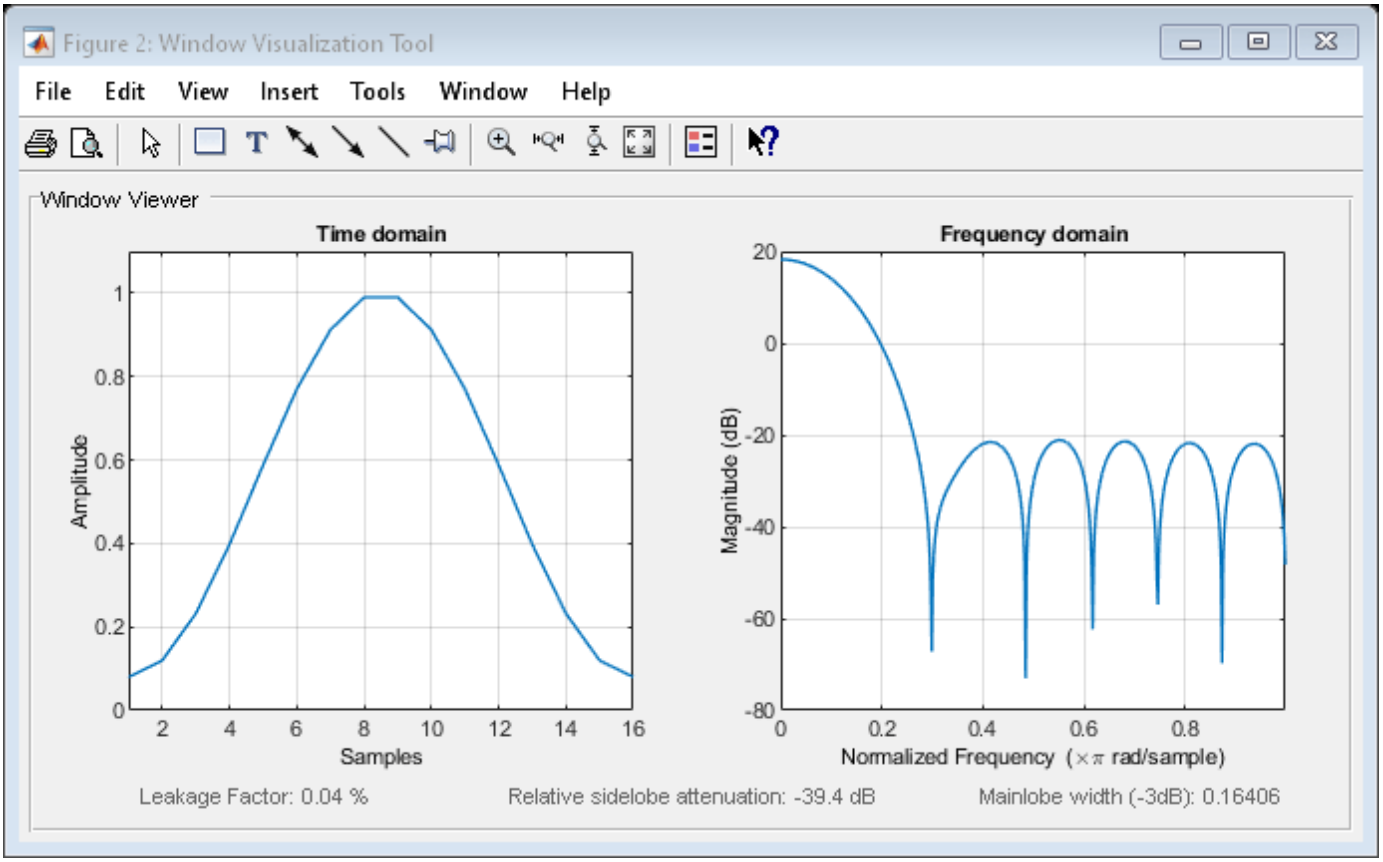
```
H = sigwin.hamming(16);

win = generate(H)
```

```
win = 16×1

    0.0800
    0.1198
    0.2322
    0.3979
    0.5881
    0.7700
    0.9121
    0.9899
    0.9899
    0.9121
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Hamming Window            '
    '--------------            '
```

```
    'Length         : 16       '
    'Sampling Flag  : symmetric'

wvtool(H)
```



See Also

`hamming` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Hamming Windows

Generate two Hamming windows:

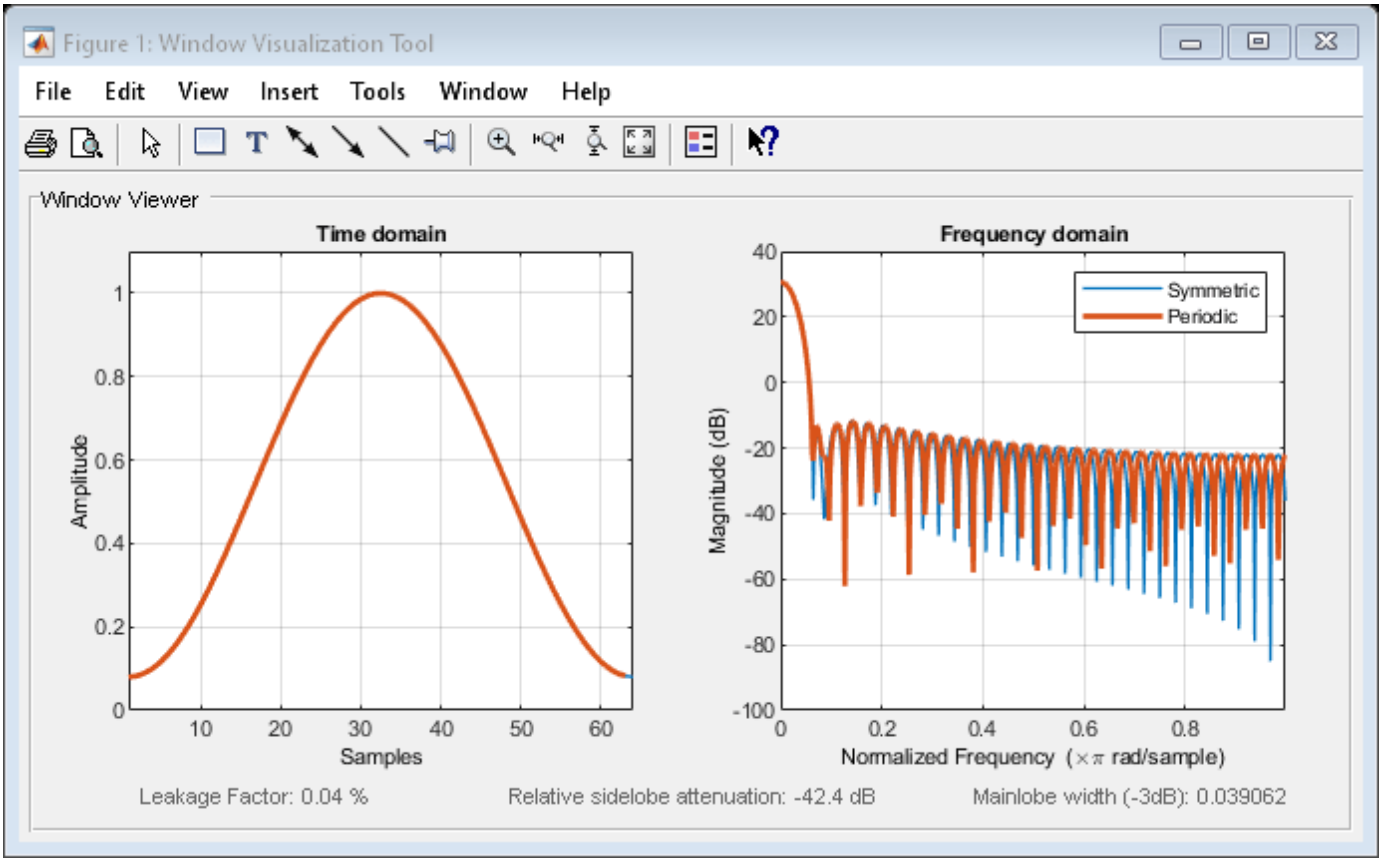
- The first window has $N = 64$ and is symmetric.
- The second window has $N = 63$ and is periodic.

Display the two windows.

```
Hs = sigwin.hamming(64,'symmetric');
Hp = sigwin.hamming(63,'periodic')
```

```
Hp =
            Name: 'Hamming'
    SamplingFlag: 'periodic'
          Length: 63
```

```
wvt = wvtool(Hs,Hp);
legend(wvt.CurrentAxes,'Symmetric','Periodic')
```



Generate a symmetric Hamming window with $N = 16$. Return the window values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.hamming(16);

win = generate(H)
```

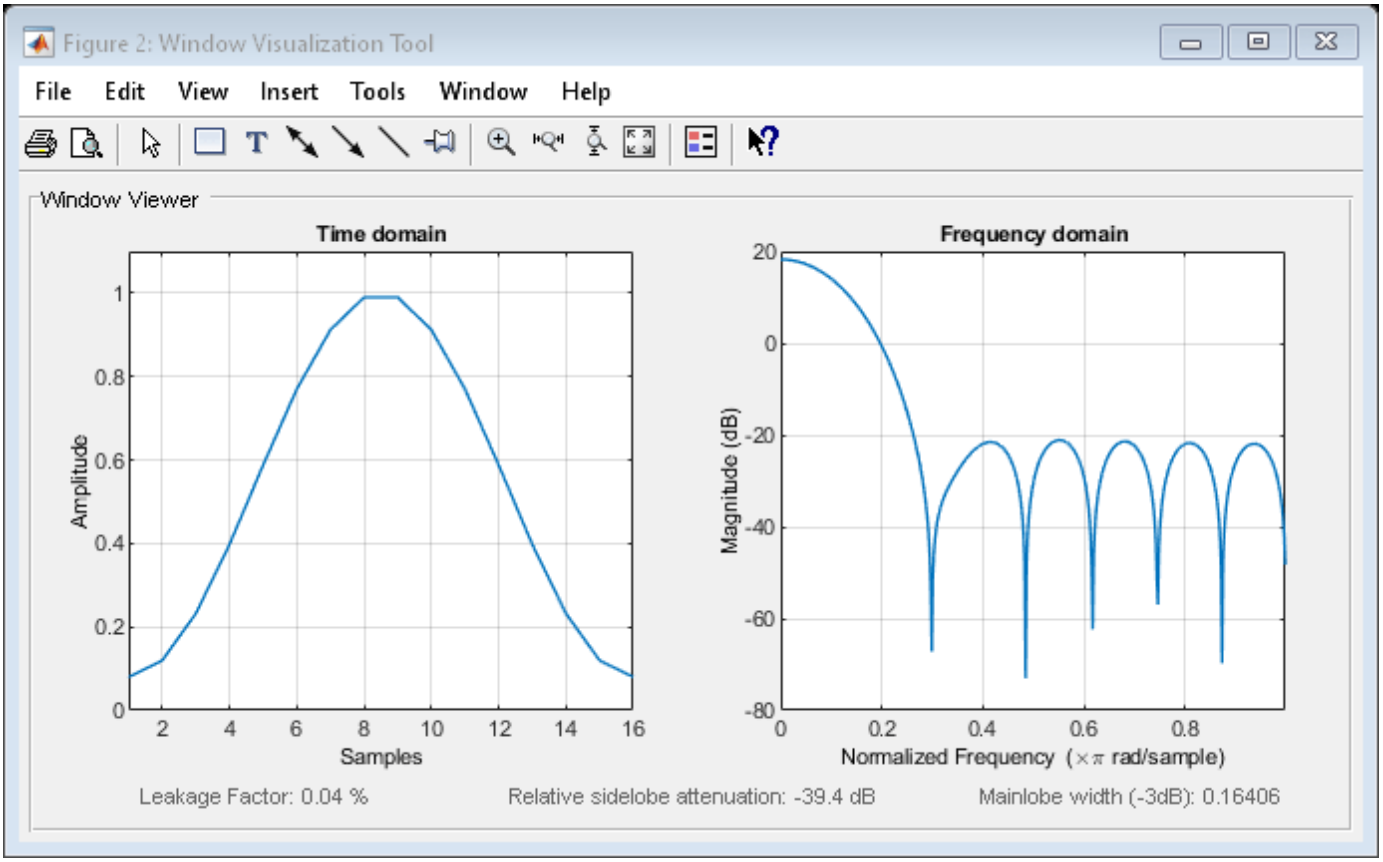
```
win = 16×1

    0.0800
    0.1198
    0.2322
    0.3979
    0.5881
    0.7700
    0.9121
    0.9899
    0.9899
    0.9121
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Hamming Window            '
    '--------------            '
```

```
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

`hamming` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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:

$$w(n) = \frac{1}{2}\left(1 - \cos\frac{2\pi n}{N-1}\right), \quad 0 \le n \le M-1$$

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

The second half of the symmetric Hann window $M \le n \le N-1$ 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 *`Sampling_Flag`*. 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

Hann Windows

Generate two Hann windows:

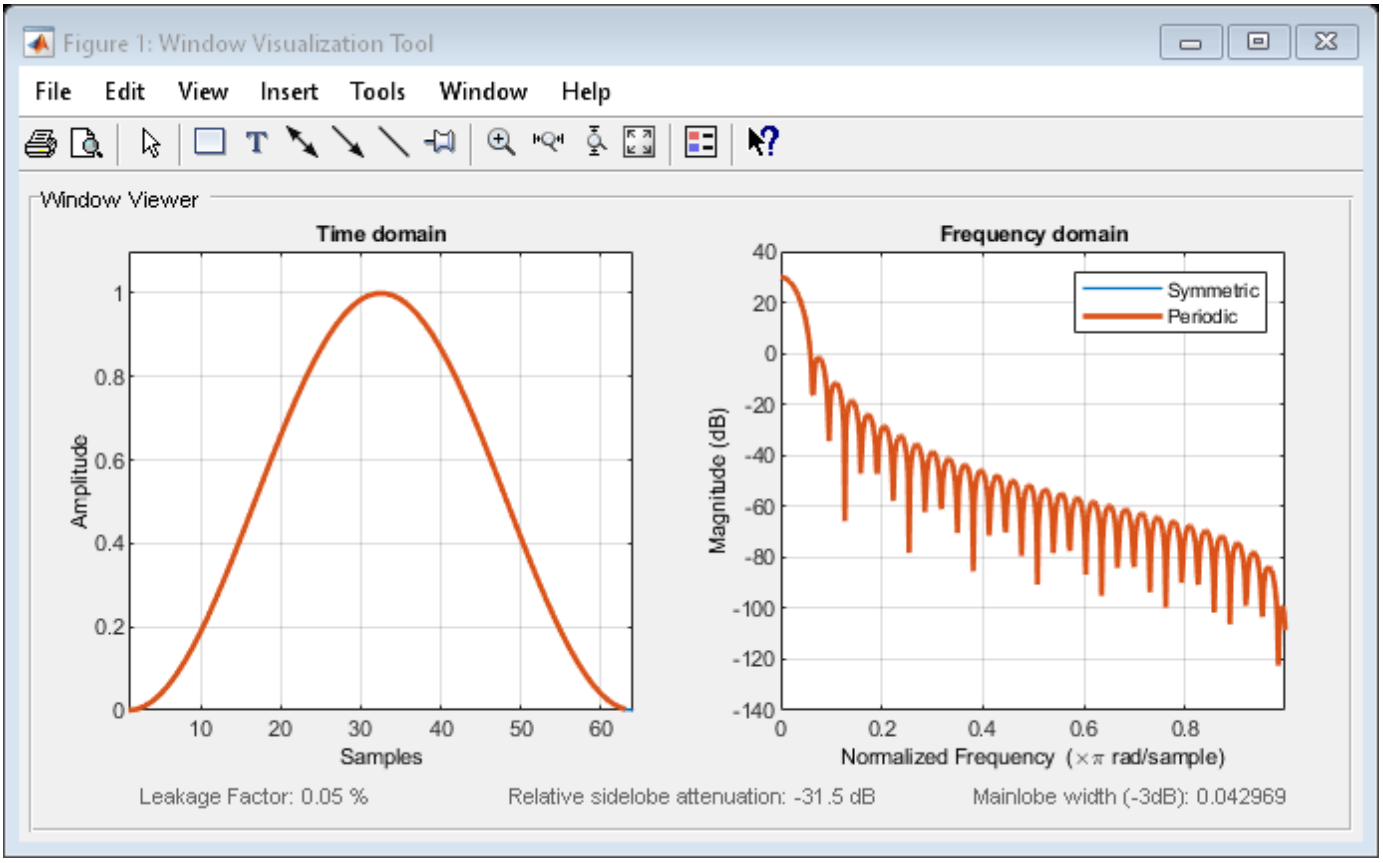
- The first window has $N = 64$ and is symmetric.
- The second window has $N = 63$ and is periodic.

Display the two windows.

```
Hs = sigwin.hann(64,'symmetric');
Hp = sigwin.hann(63,'periodic')
```

```
Hp =
            Name: 'Hann'
    SamplingFlag: 'periodic'
          Length: 63
```

```
wvt = wvtool(Hs,Hp);
legend(wvt.CurrentAxes,'Symmetric','Periodic')
```



Generate a symmetric Hann window with $N = 16$. Return the window values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.hann(16);

win = generate(H)
```

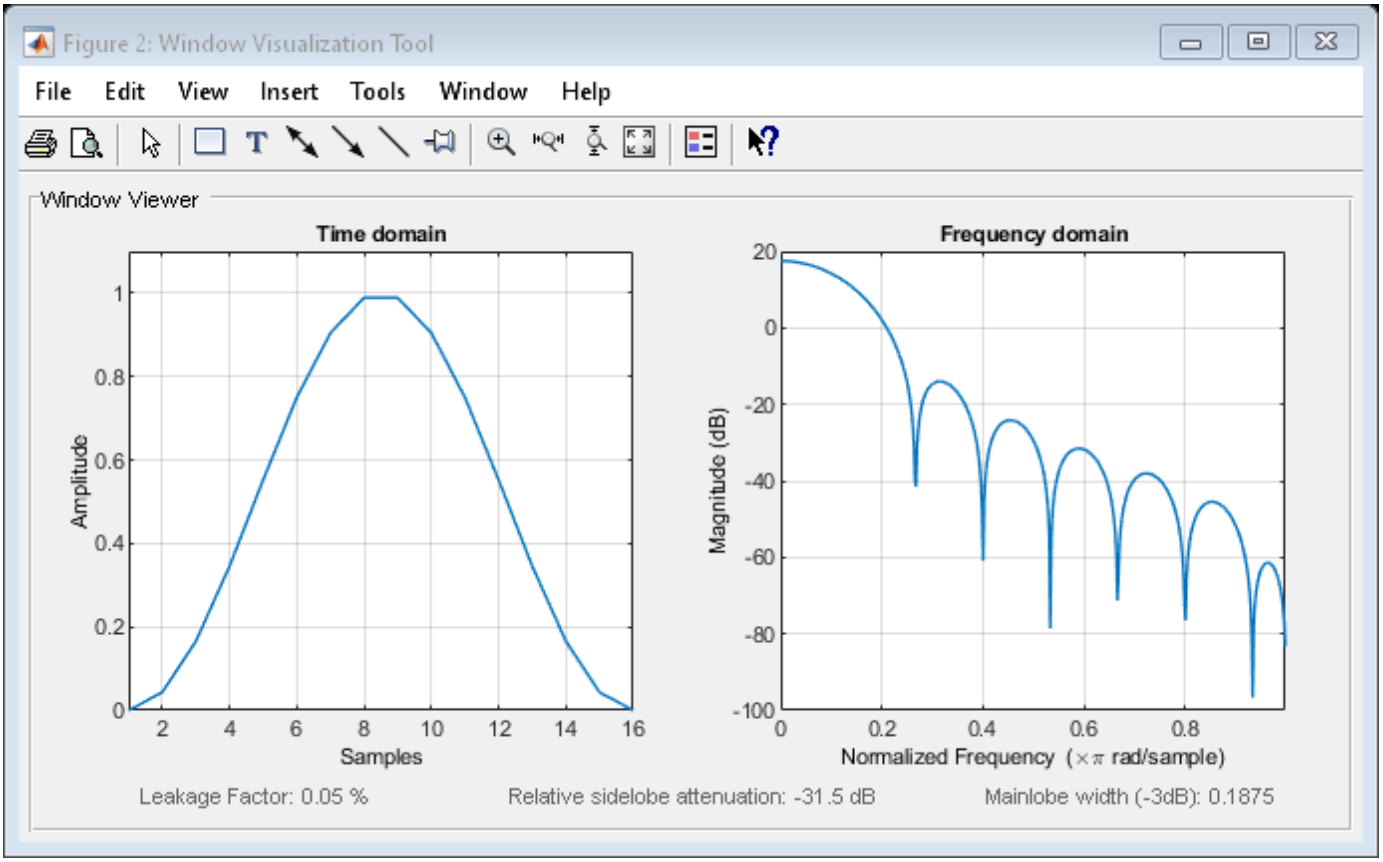
```
win = 16×1

         0
    0.0432
    0.1654
    0.3455
    0.5523
    0.7500
    0.9045
    0.9891
    0.9891
    0.9045
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Hann Window               '
    '------------              '
```

```
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



References

Oppenheim, Alan V., and Ronald W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

See Also

`hann` | `window` | **WVTool**

Topics

“Windows”
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

Hann Windows

Generate two Hann windows:

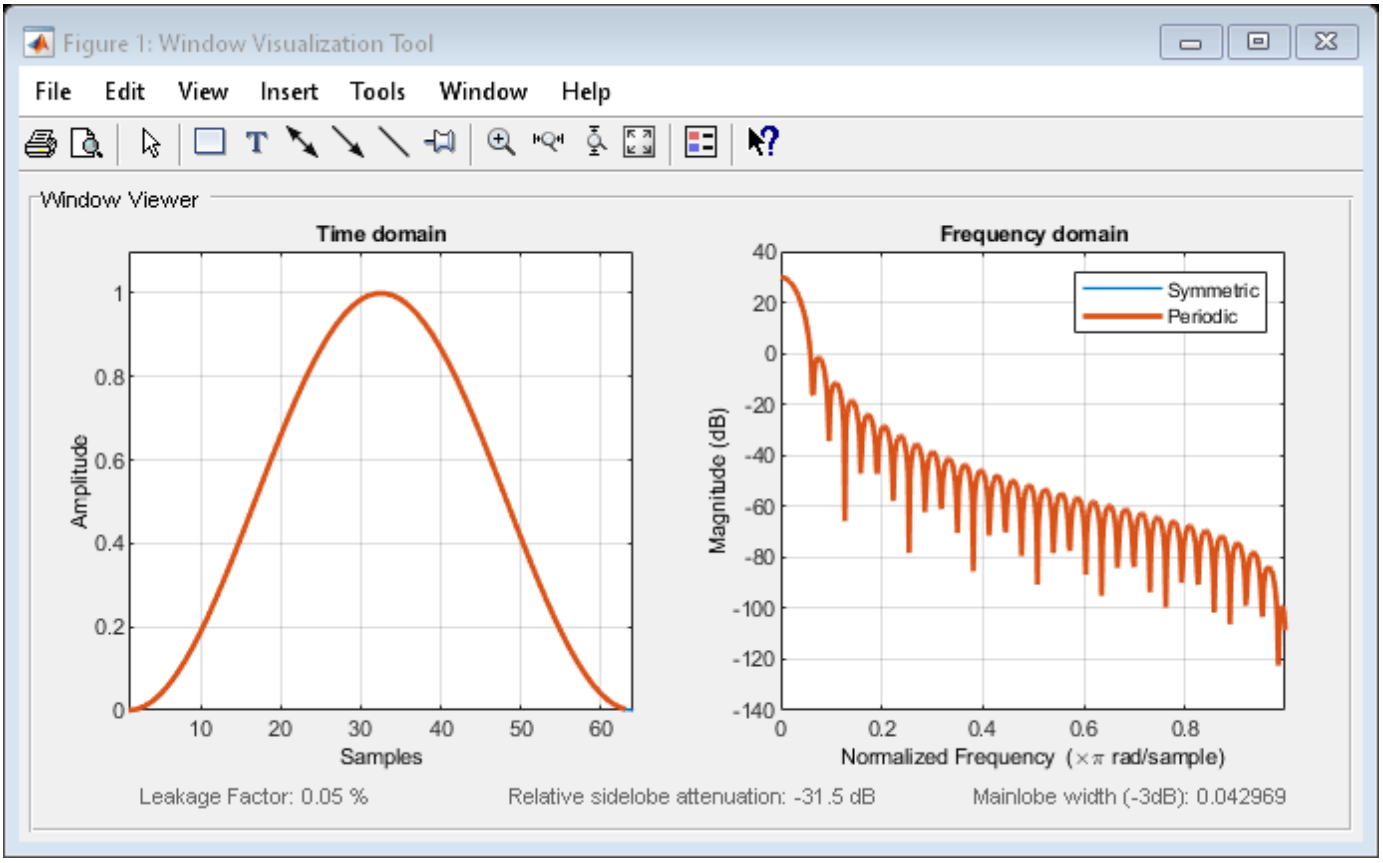
- The first window has $N = 64$ and is symmetric.
- The second window has $N = 63$ and is periodic.

Display the two windows.

```
Hs = sigwin.hann(64,'symmetric');
Hp = sigwin.hann(63,'periodic')
```

```
Hp =
            Name: 'Hann'
    SamplingFlag: 'periodic'
          Length: 63
```

```
wvt = wvtool(Hs,Hp);
legend(wvt.CurrentAxes,'Symmetric','Periodic')
```



Generate a symmetric Hann window with $N = 16$. Return the window values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.hann(16);

win = generate(H)
```

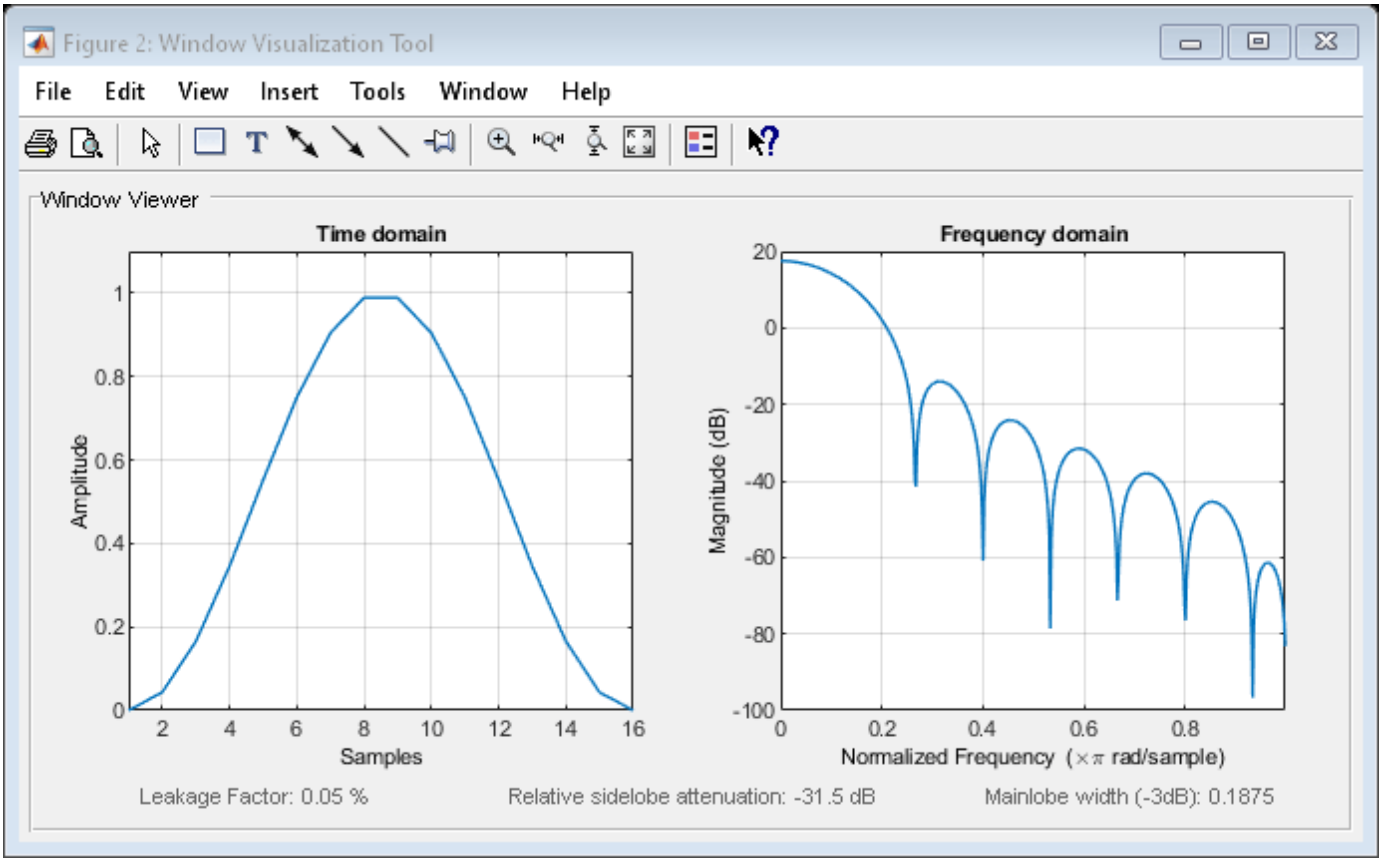
```
win = 16×1

         0
    0.0432
    0.1654
    0.3455
    0.5523
    0.7500
    0.9045
    0.9891
    0.9891
    0.9045
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Hann Window               '
    '------------              '
```

```
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

hann | window | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Hann Windows

Generate two Hann windows:

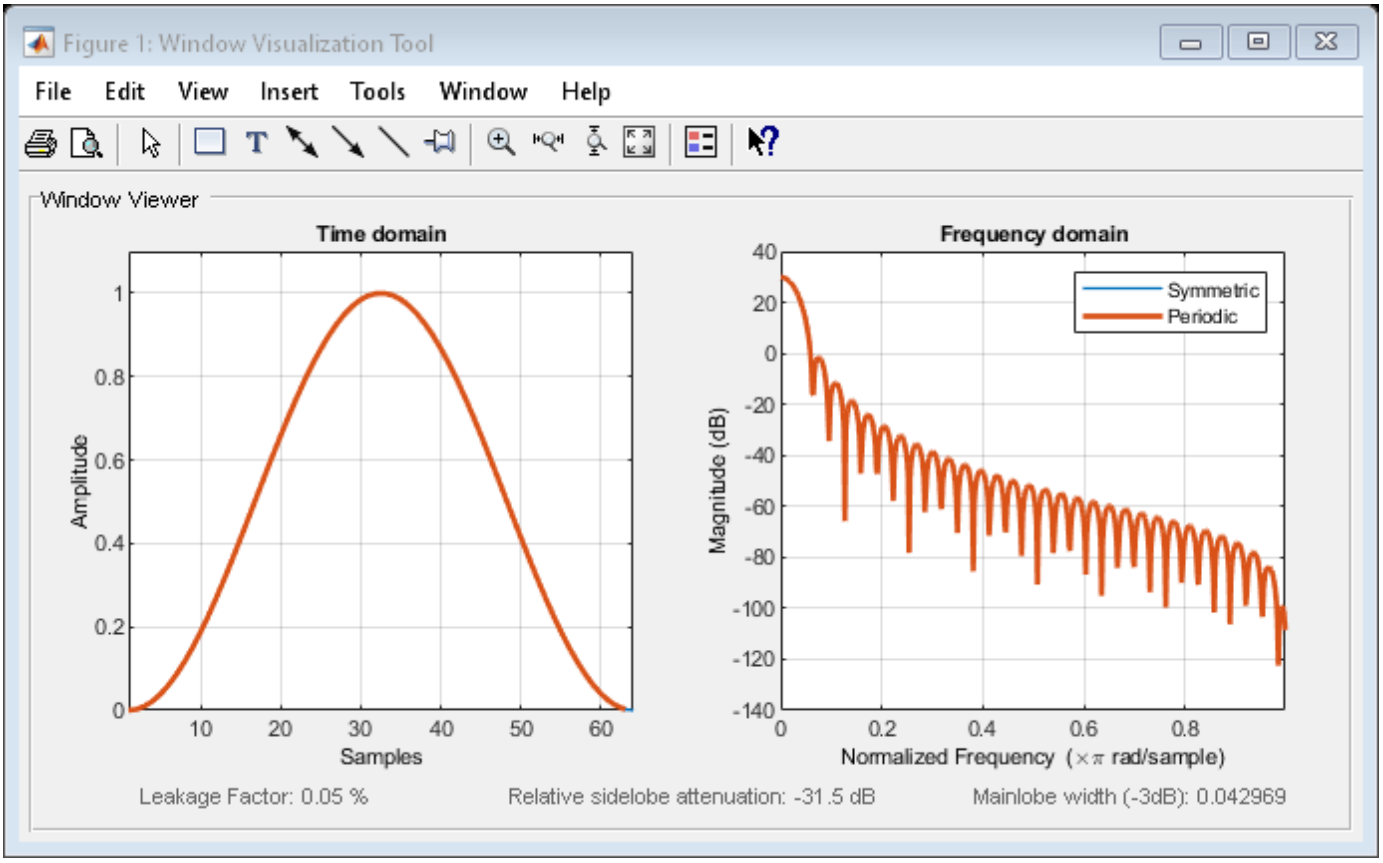
- The first window has $N = 64$ and is symmetric.
- The second window has $N = 63$ and is periodic.

Display the two windows.

```
Hs = sigwin.hann(64,'symmetric');
Hp = sigwin.hann(63,'periodic')
```

```
Hp =
            Name: 'Hann'
    SamplingFlag: 'periodic'
          Length: 63
```

```
wvt = wvtool(Hs,Hp);
legend(wvt.CurrentAxes,'Symmetric','Periodic')
```



Generate a symmetric Hann window with $N = 16$. Return the window values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.hann(16);

win = generate(H)
```

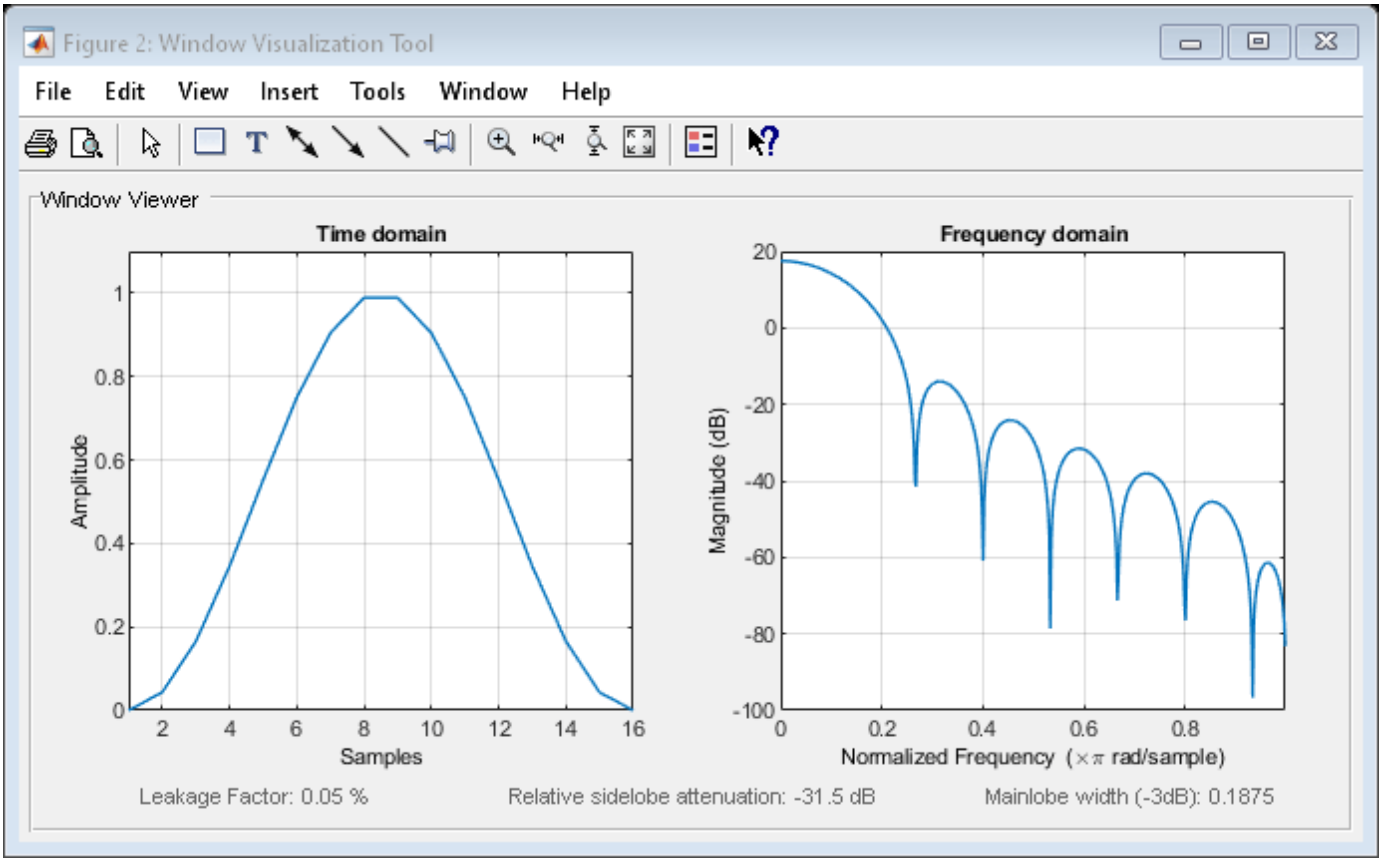
```
win = 16×1

         0
    0.0432
    0.1654
    0.3455
    0.5523
    0.7500
    0.9045
    0.9891
    0.9891
    0.9045
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Hann Window               '
    '------------              '
```

```
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

hann | window | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Hann Windows

Generate two Hann windows:

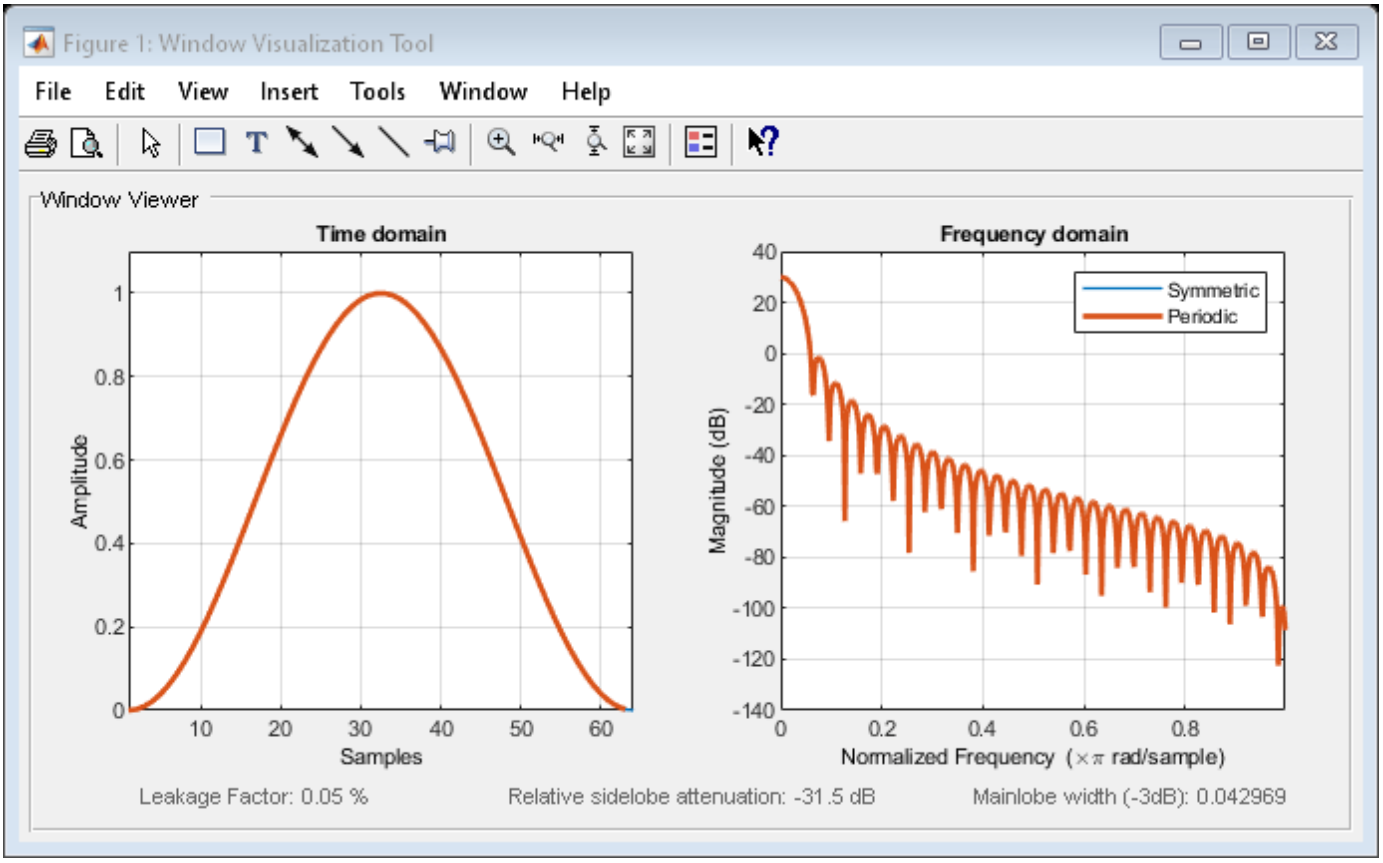
- The first window has $N = 64$ and is symmetric.
- The second window has $N = 63$ and is periodic.

Display the two windows.

```
Hs = sigwin.hann(64,'symmetric');
Hp = sigwin.hann(63,'periodic')
```

```
Hp =
            Name: 'Hann'
    SamplingFlag: 'periodic'
          Length: 63
```

```
wvt = wvtool(Hs,Hp);
legend(wvt.CurrentAxes,'Symmetric','Periodic')
```



Generate a symmetric Hann window with $N = 16$. Return the window values as a column vector. Show information about the window object. Display the window.

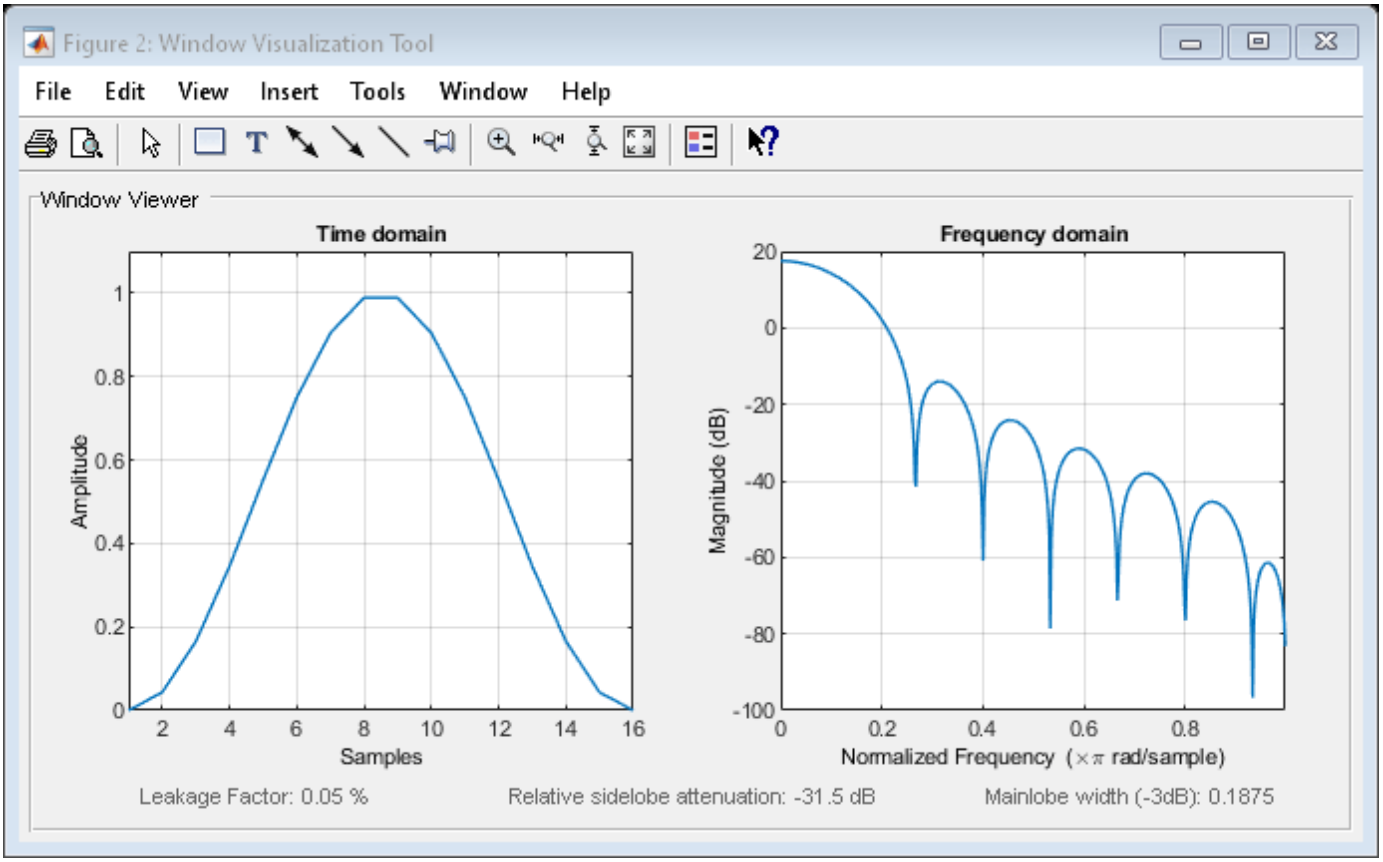
```
H = sigwin.hann(16);

win = generate(H)
```

```
win = 16×1

         0
    0.0432
    0.1654
    0.3455
    0.5523
    0.7500
    0.9045
    0.9891
    0.9891
    0.9045
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Hann Window               '
    '------------              '
```

```
    'Length         : 16       '
    'Sampling Flag  : symmetric'

wvtool(H)
```



See Also

hann | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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:

$$w(x) = I_0\left(\beta\sqrt{1-\frac{4x^2}{(N-1)^2}}\right)/I_0(\beta), \quad -(N-1)/2 \le x \le (N-1)/2$$

where x is linearly spaced N-point vector and $I_0()$ is the modified zeroth-order Bessel function of the first kind. β 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

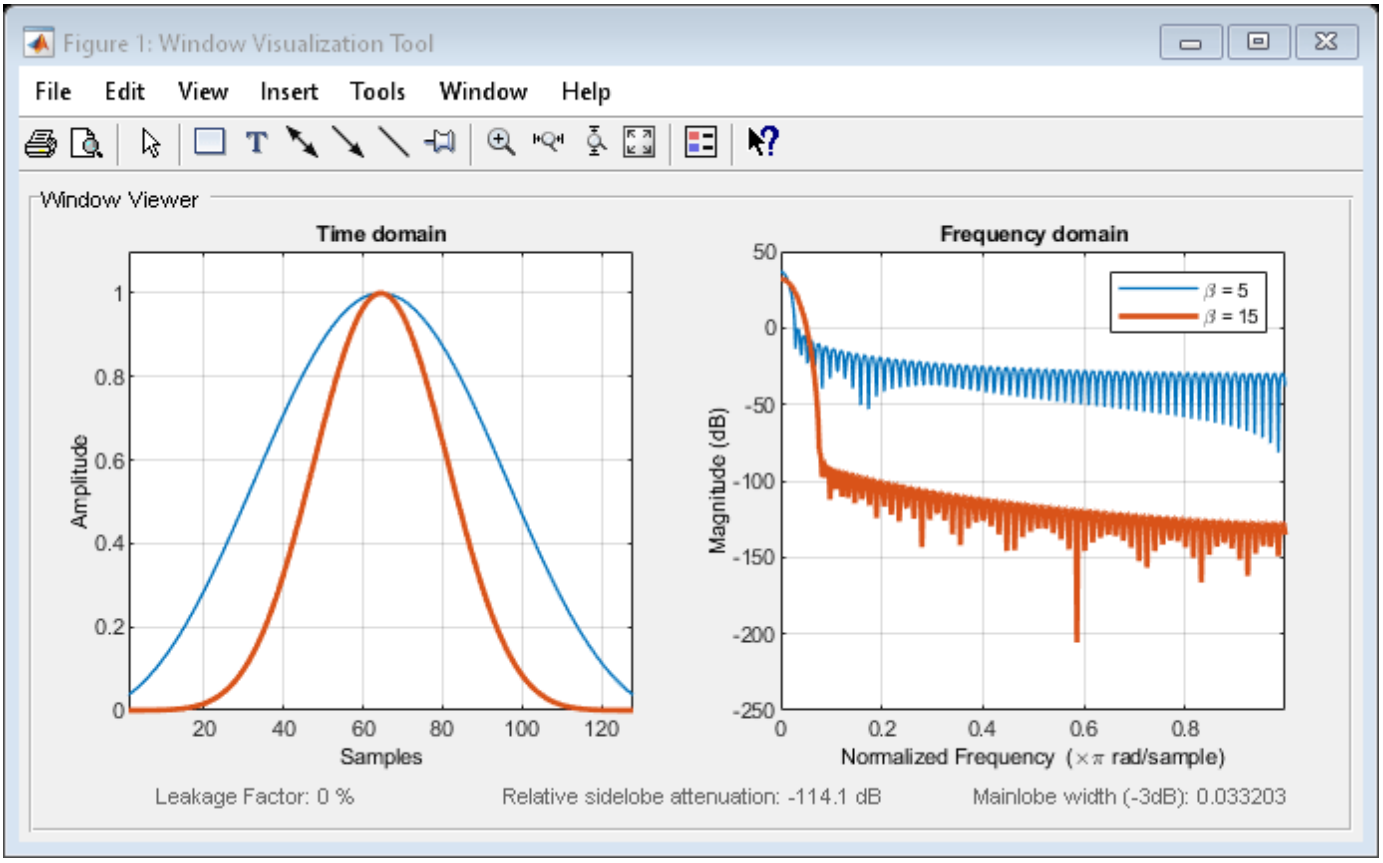
Kaiser Windows

Generate two Kaiser windows of length $N = 64$:

- The first window has an attenuation parameter $\beta = 5$.
- The second window has $\beta = 15$.

Display the two windows.

```
H05 = sigwin.kaiser(128,5);
H15 = sigwin.kaiser(128,15);

wvt = wvtool(H05,H15);
legend(wvt.CurrentAxes,'\beta = 5','\beta = 15')
```



Generate a Kaiser window with length $N = 16$ and the default $\beta = 1/2$. Return its values as a column vector. Show information about the window object. Display the window.

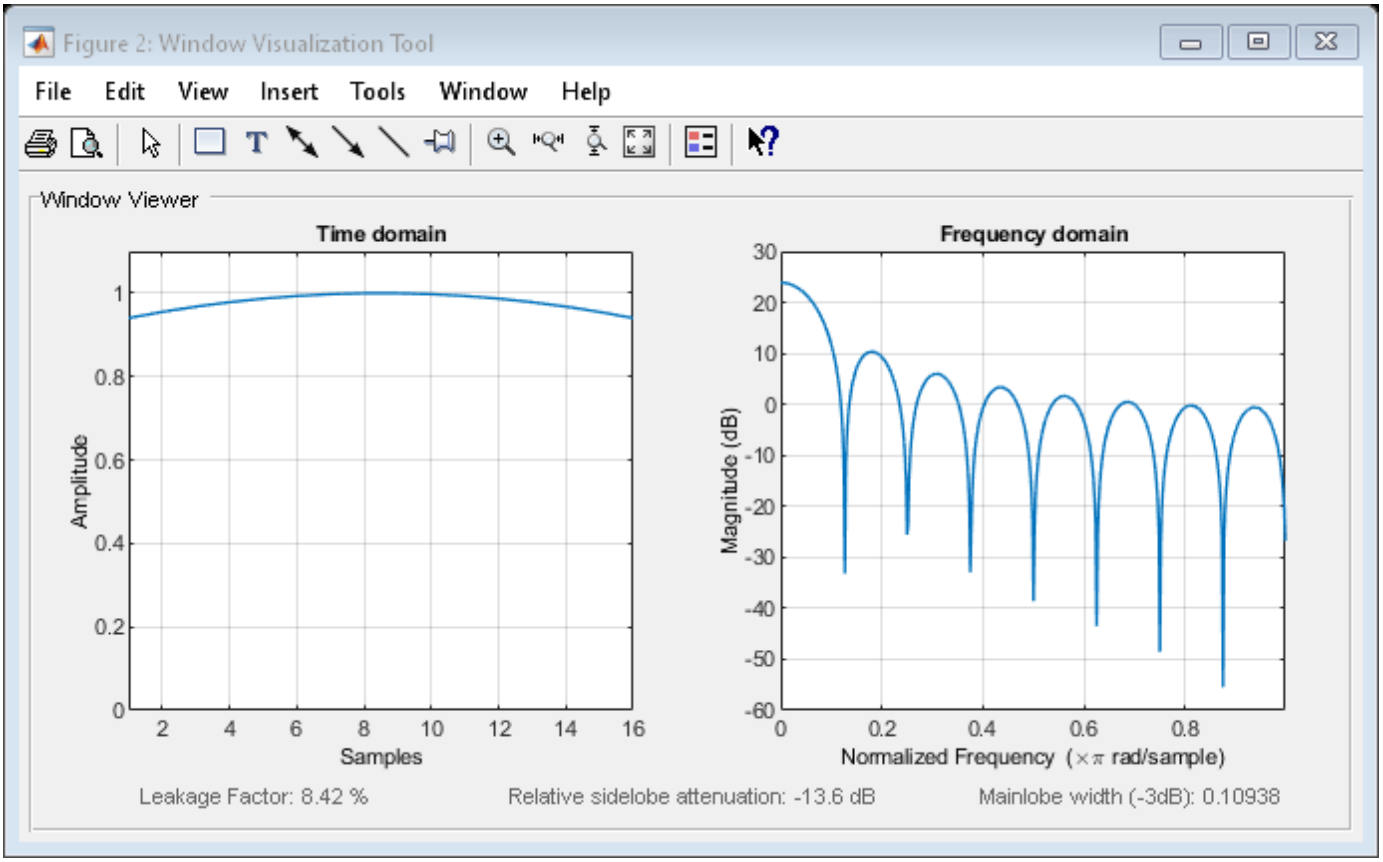
```
H = sigwin.kaiser(16);

win = generate(H)
```

```
win = 16×1

    0.9403
    0.9550
    0.9677
    0.9783
    0.9868
    0.9933
    0.9976
    0.9997
    0.9997
    0.9976
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x13 char array
    'Kaiser Window'
    '--------------'
```

```
    'Length  : 16 '
    'Beta    : 0.5'

wvtool(H)
```



References

Oppenheim, Alan V., and Ronald W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

See Also

`kaiser` | `window` | **WVTool**

Topics
“Windows”
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

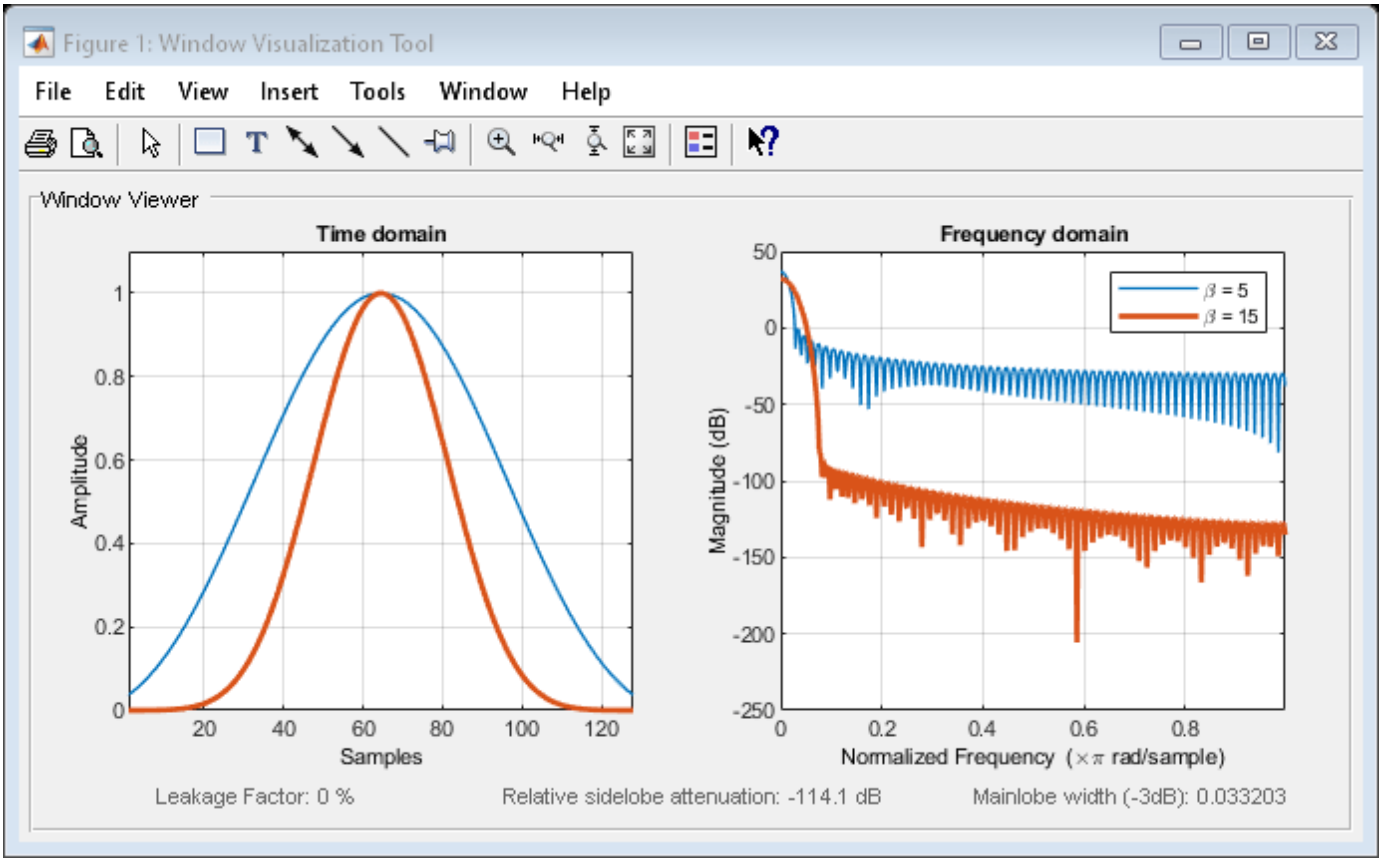
Kaiser Windows

Generate two Kaiser windows of length $N = 64$:

- The first window has an attenuation parameter $\beta = 5$.
- The second window has $\beta = 15$.

Display the two windows.

```
H05 = sigwin.kaiser(128,5);
H15 = sigwin.kaiser(128,15);

wvt = wvtool(H05,H15);
legend(wvt.CurrentAxes,'\beta = 5','\beta = 15')
```



Generate a Kaiser window with length $N = 16$ and the default $\beta = 1/2$. Return its values as a column vector. Show information about the window object. Display the window.

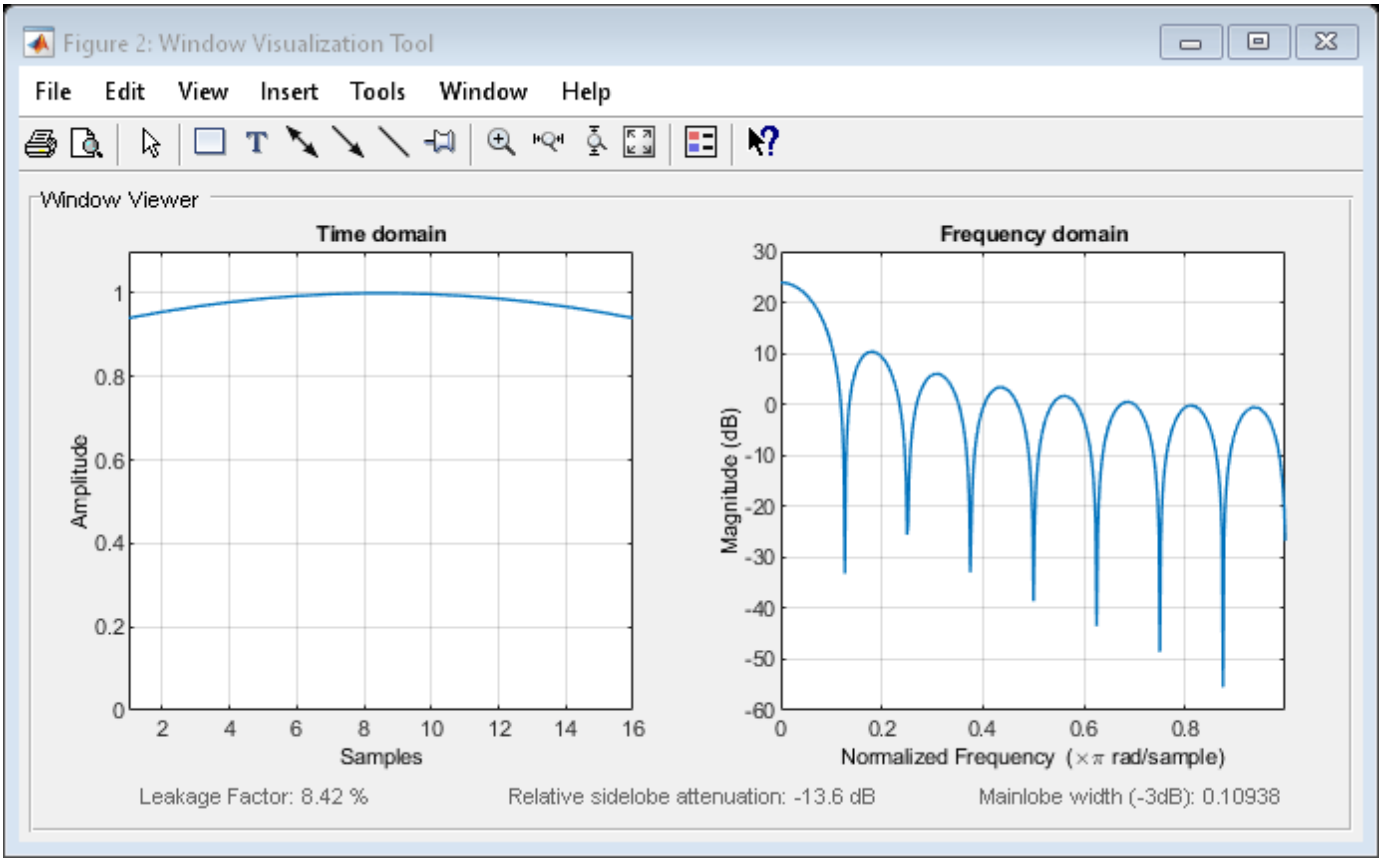
```
H = sigwin.kaiser(16);

win = generate(H)
```

```
win = 16×1

    0.9403
    0.9550
    0.9677
    0.9783
    0.9868
    0.9933
    0.9976
    0.9997
    0.9997
    0.9976
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x13 char array
    'Kaiser Window'
    '--------------'
```

```
    'Length  : 16 '
    'Beta    : 0.5'

wvtool(H)
```



See Also

`kaiser` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

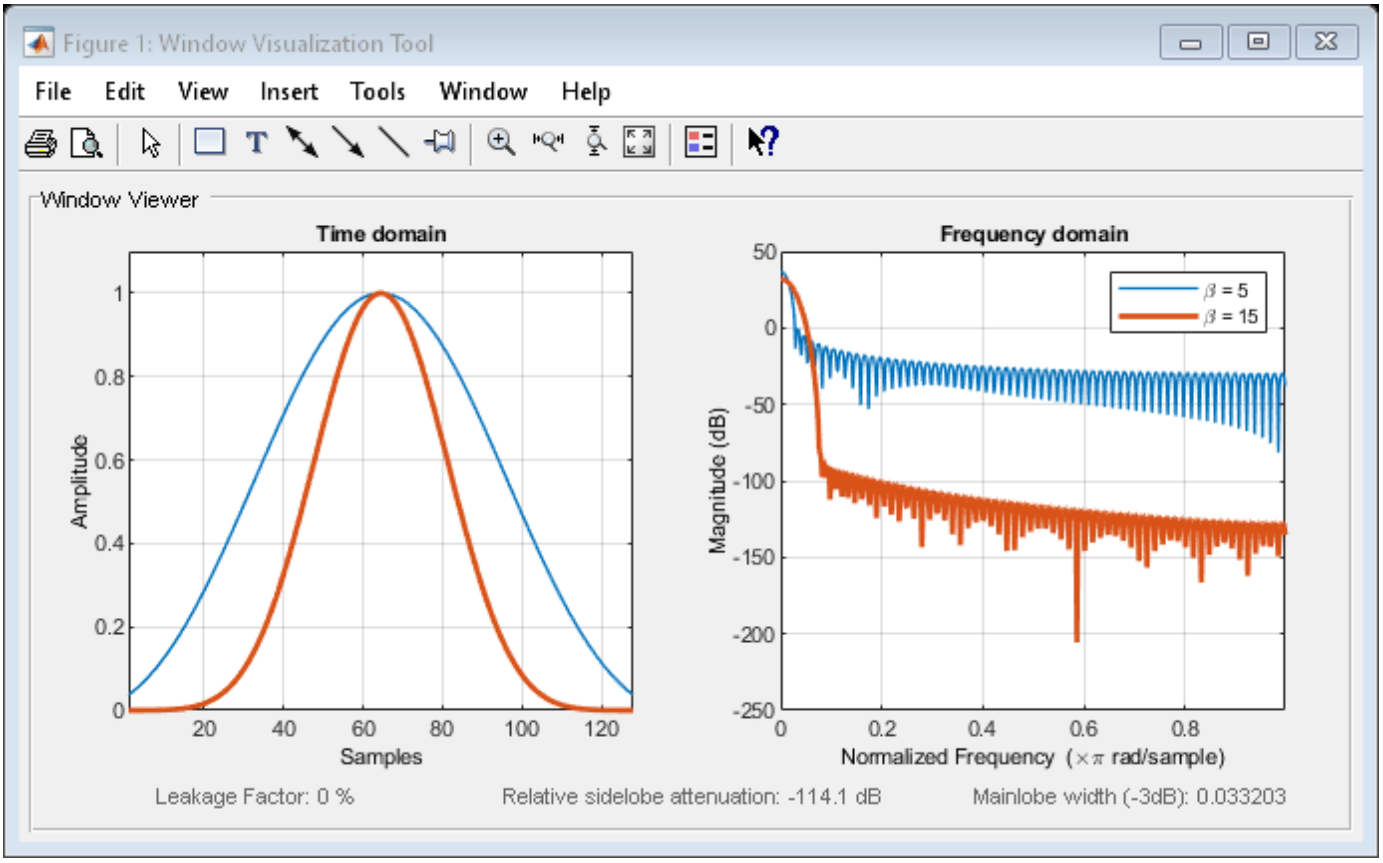
Kaiser Windows

Generate two Kaiser windows of length $N = 64$:

- The first window has an attenuation parameter $\beta = 5$.
- The second window has $\beta = 15$.

Display the two windows.

```
H05 = sigwin.kaiser(128,5);
H15 = sigwin.kaiser(128,15);

wvt = wvtool(H05,H15);
legend(wvt.CurrentAxes,'\beta = 5','\beta = 15')
```



Generate a Kaiser window with length $N = 16$ and the default $\beta = 1/2$. Return its values as a column vector. Show information about the window object. Display the window.

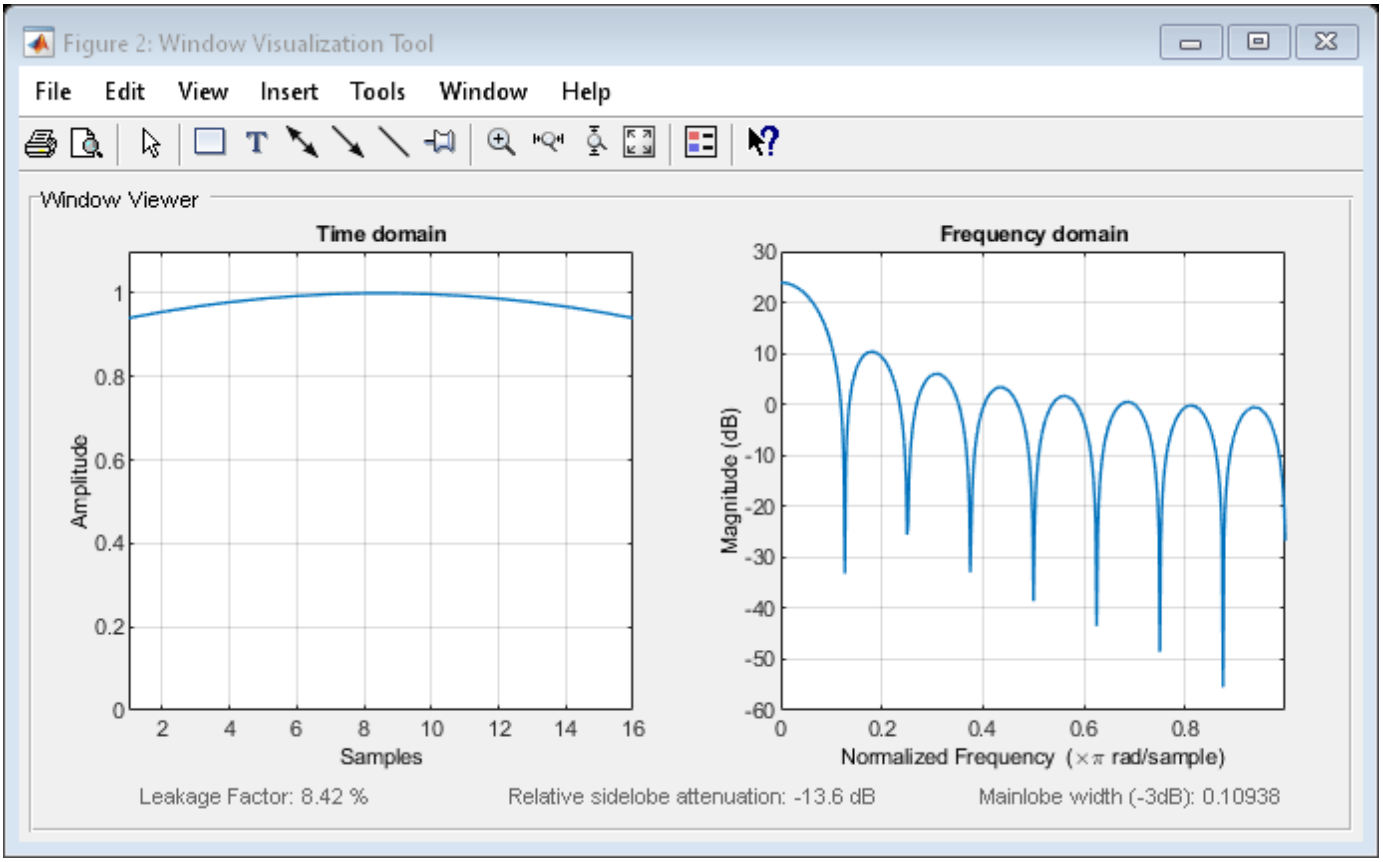
```
H = sigwin.kaiser(16);

win = generate(H)
```

```
win = 16×1

    0.9403
    0.9550
    0.9677
    0.9783
    0.9868
    0.9933
    0.9976
    0.9997
    0.9997
    0.9976
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x13 char array
    'Kaiser Window'
    '--------------'
```

```
    'Length  : 16 '
    'Beta    : 0.5'

wvtool(H)
```



See Also

`kaiser` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

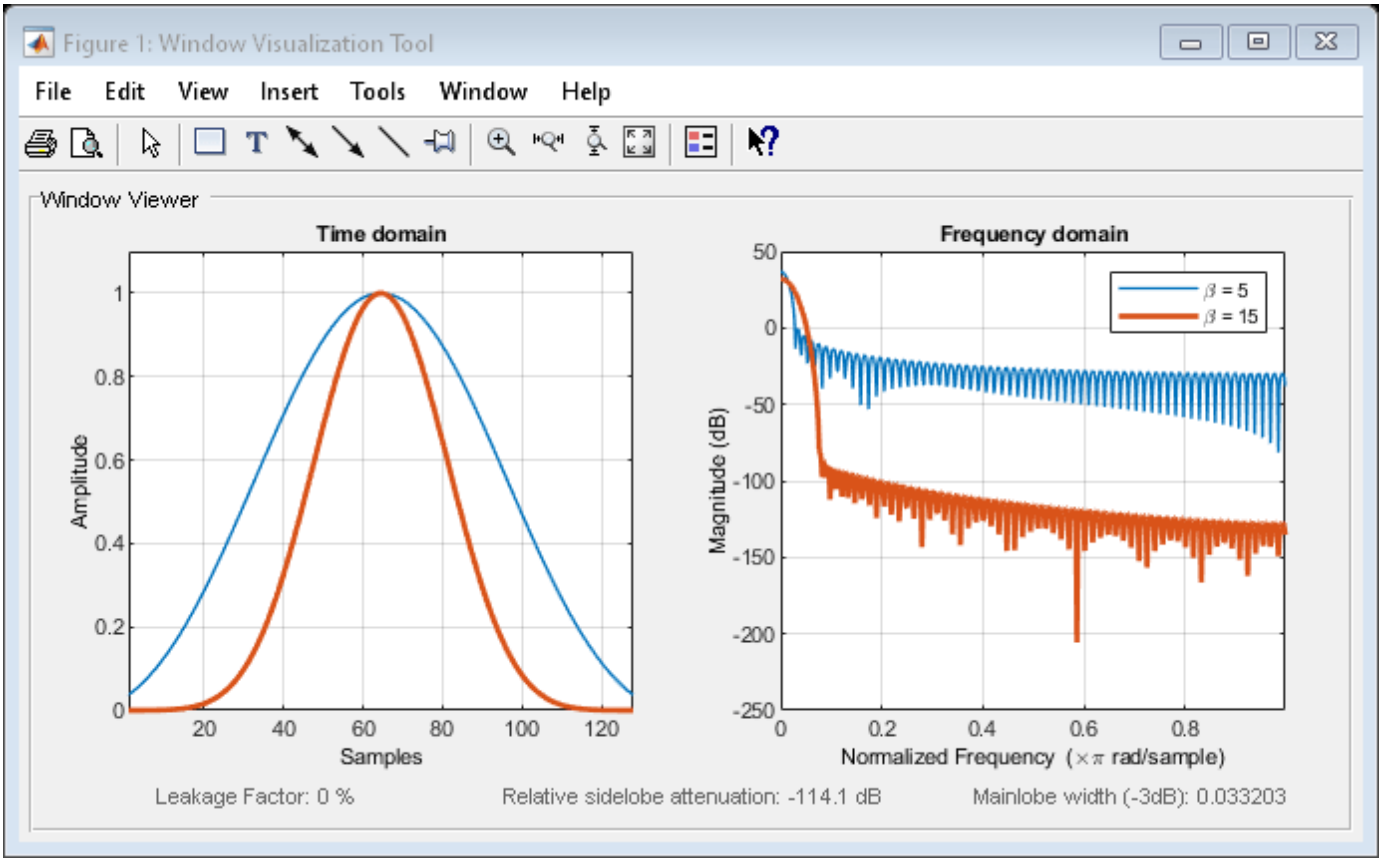
Kaiser Windows

Generate two Kaiser windows of length $N = 64$:

- The first window has an attenuation parameter $\beta = 5$.
- The second window has $\beta = 15$.

Display the two windows.

```
H05 = sigwin.kaiser(128,5);
H15 = sigwin.kaiser(128,15);

wvt = wvtool(H05,H15);
legend(wvt.CurrentAxes,'\beta = 5','\beta = 15')
```



Generate a Kaiser window with length $N = 16$ and the default $\beta = 1/2$. Return its values as a column vector. Show information about the window object. Display the window.

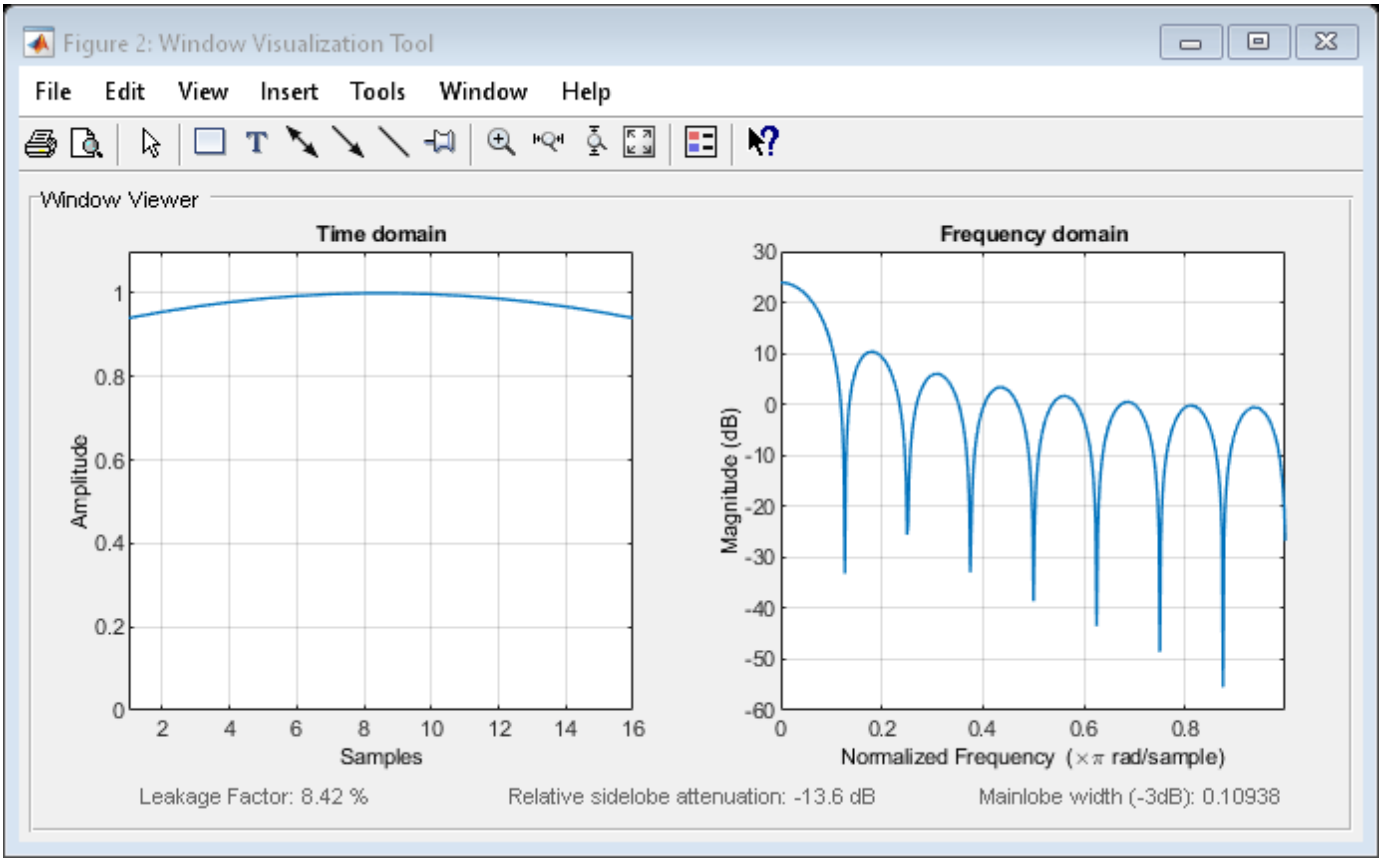
```
H = sigwin.kaiser(16);

win = generate(H)
```

```
win = 16×1

    0.9403
    0.9550
    0.9677
    0.9783
    0.9868
    0.9933
    0.9976
    0.9997
    0.9997
    0.9976
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x13 char array
    'Kaiser Window'
    '--------------'
```

```
    'Length  : 16 '
    'Beta    : 0.5'

wvtool(H)
```



See Also

`kaiser` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

sigwin.nuttallwin class

Package: `sigwin`

Construct Nuttall defined four-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 four-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 four-term Blackman-Harris window object window object `H` of length 64.

`H = sigwin.nuttallwin(Length)` returns a Nuttall defined four-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 four-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 “Algorithms” on page 1-2309 for details.

Methods

generate	Generates Nuttall defined four-term Blackman-Harris window
info	Display information about Nuttall defined four-term Blackman-Harris window object
winwrite	Save Nuttall defined four-term Blackman-Harris 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

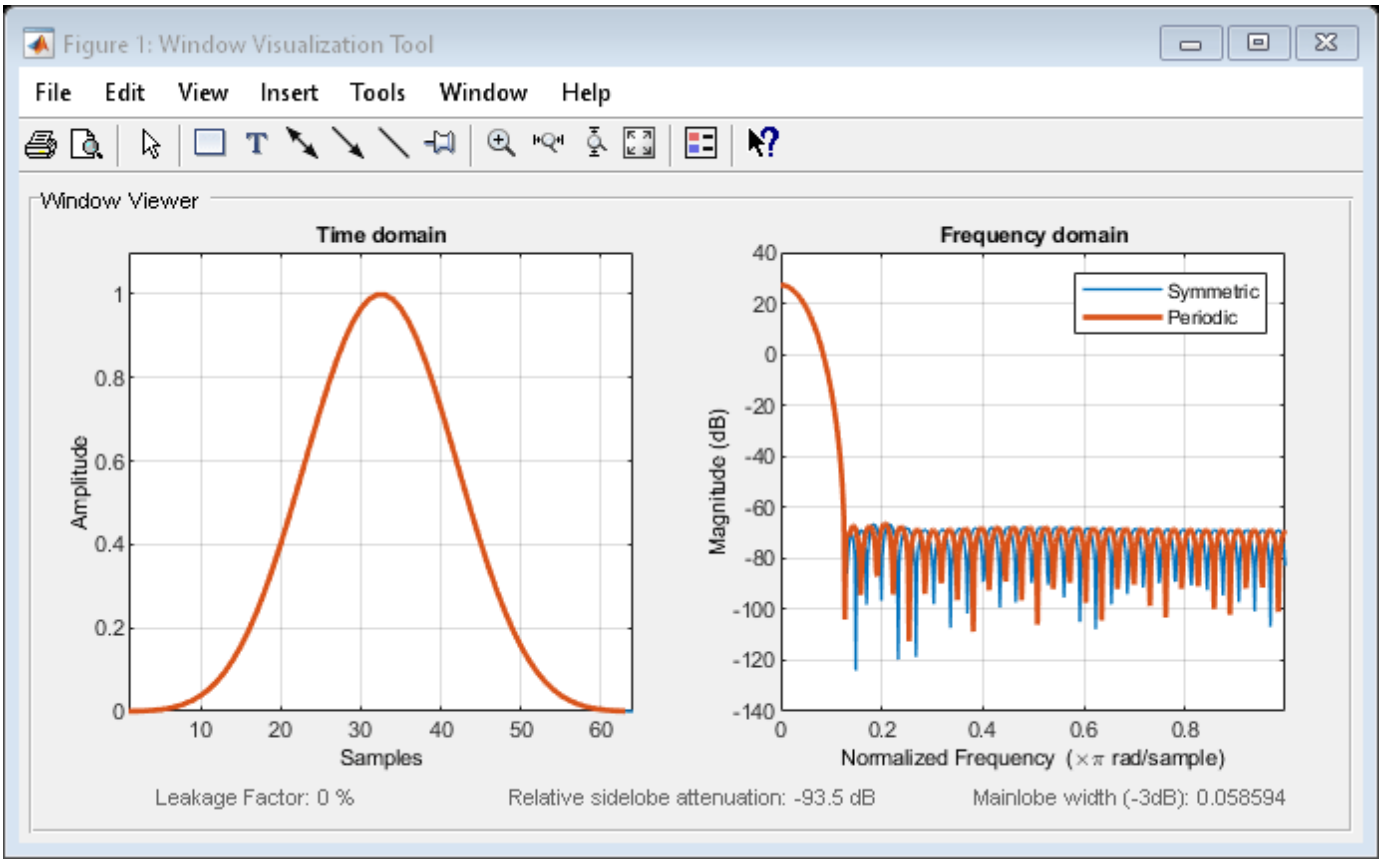
Nuttall-Defined Four-Term Blackman-Harris Windows

Generate two Nuttall-defined four-term Blackman-Harris windows:

- The first window has $N = 64$ and is symmetric.
- The second window has $N = 63$ and is periodic.

Display the two windows.

```
Hs = sigwin.nuttallwin(64);
Hp = sigwin.nuttallwin(63);
Hp.SamplingFlag = 'periodic';

wvt = wvtool(Hs,Hp);
legend(wvt.CurrentAxes,'Symmetric','Periodic')
```



Generate a symmetric Nuttall-defined four-term Blackman-Harris window with $N = 16$. Return the window values as a column vector. Show information about the window object. Display the window.

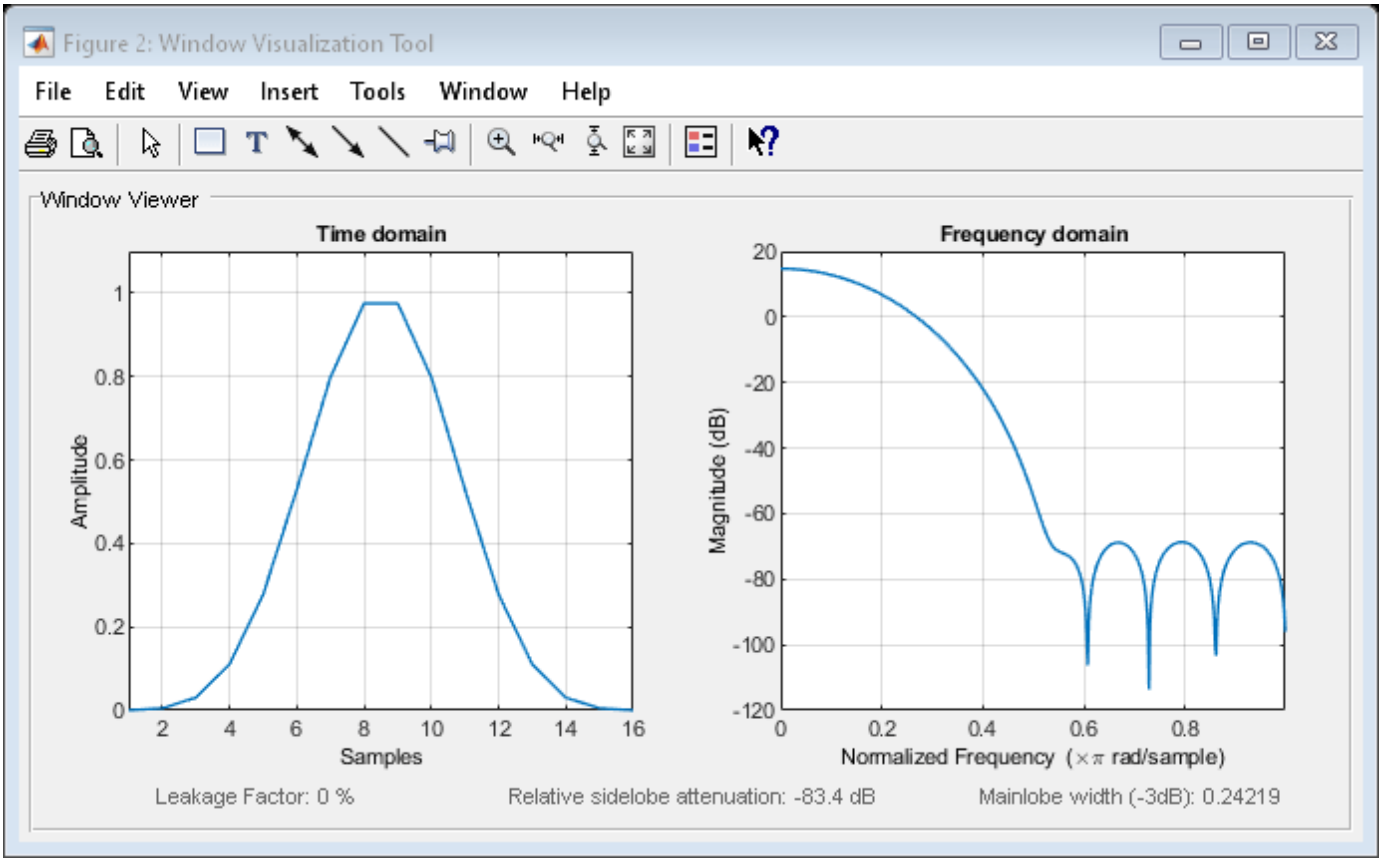
```
H = sigwin.nuttallwin(16);

win = generate(H)
```

```
win = 16×1

    0.0004
    0.0048
    0.0306
    0.1105
    0.2778
    0.5292
    0.7983
    0.9755
    0.9755
    0.7983
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Nuttall Window            '
    '--------------            '
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



Algorithms

The following equation defines the symmetric Nuttall defined four-term Blackman-Harris window of length N.

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

The following equation defines the periodic Nuttall defined four-term Blackman-Harris window of length N.

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

The following table lists the coefficients:

Coefficient	Value
a_0	0.3635819
a_1	0.4891775
a_2	0.1365995
a_3	0.0106411

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**

Topics
“Windows”
Class Attributes
Property Attributes

generate

Class: `sigwin.nuttallwin`
Package: `sigwin`

Generates Nuttall defined four-term Blackman-Harris window

Syntax

```
win = generate(H)
```

Description

`win = generate(H)` returns the values of the Nuttall defined four-term Blackman-Harris window object as a double-precision column vector.

Examples

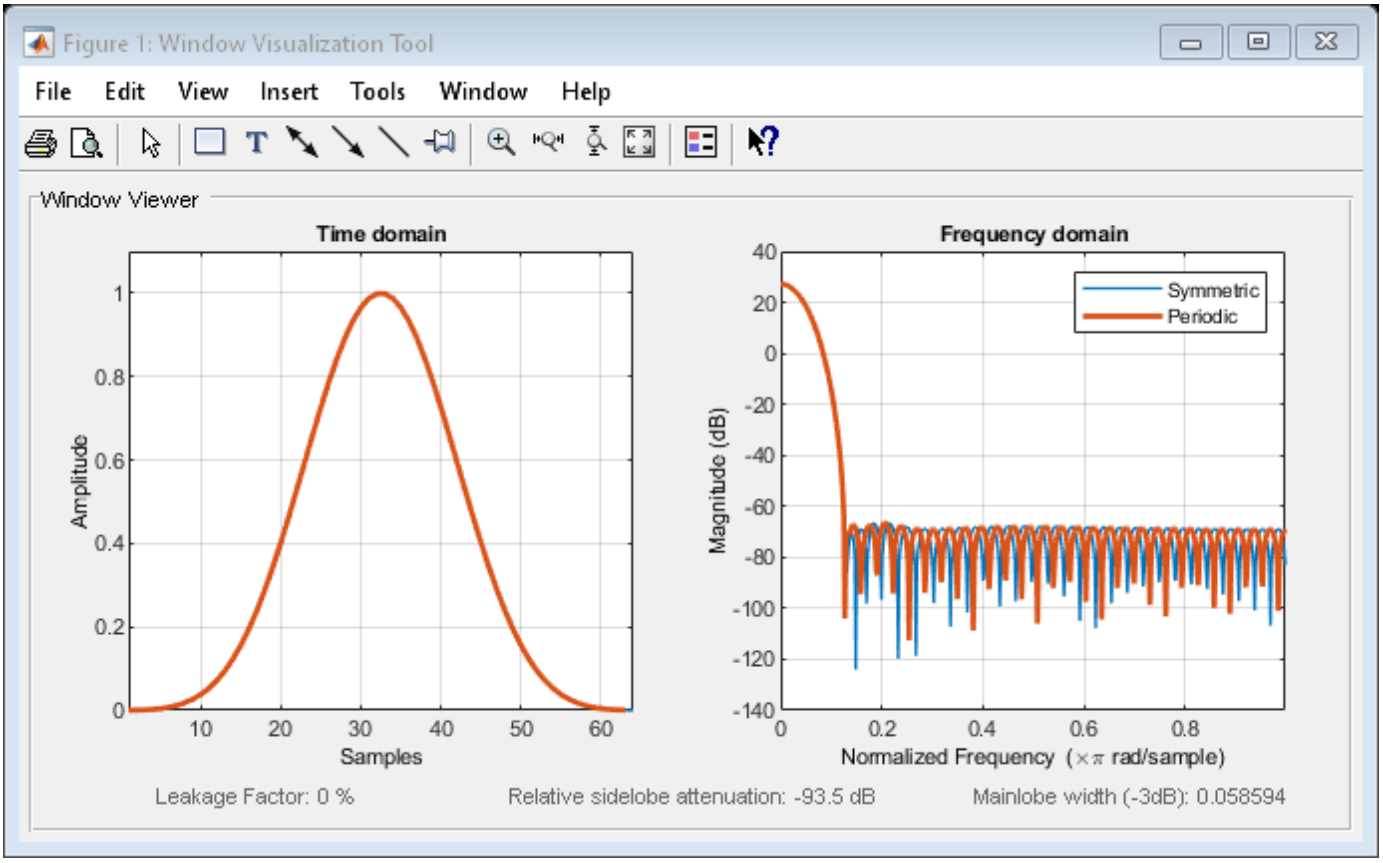
Nuttall-Defined Four-Term Blackman-Harris Windows

Generate two Nuttall-defined four-term Blackman-Harris windows:

- The first window has $N = 64$ and is symmetric.
- The second window has $N = 63$ and is periodic.

Display the two windows.

```
Hs = sigwin.nuttallwin(64);
Hp = sigwin.nuttallwin(63);
Hp.SamplingFlag = 'periodic';

wvt = wvtool(Hs,Hp);
legend(wvt.CurrentAxes,'Symmetric','Periodic')
```



Generate a symmetric Nuttall-defined four-term Blackman-Harris window with $N = 16$. Return the window values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.nuttallwin(16);

win = generate(H)
```

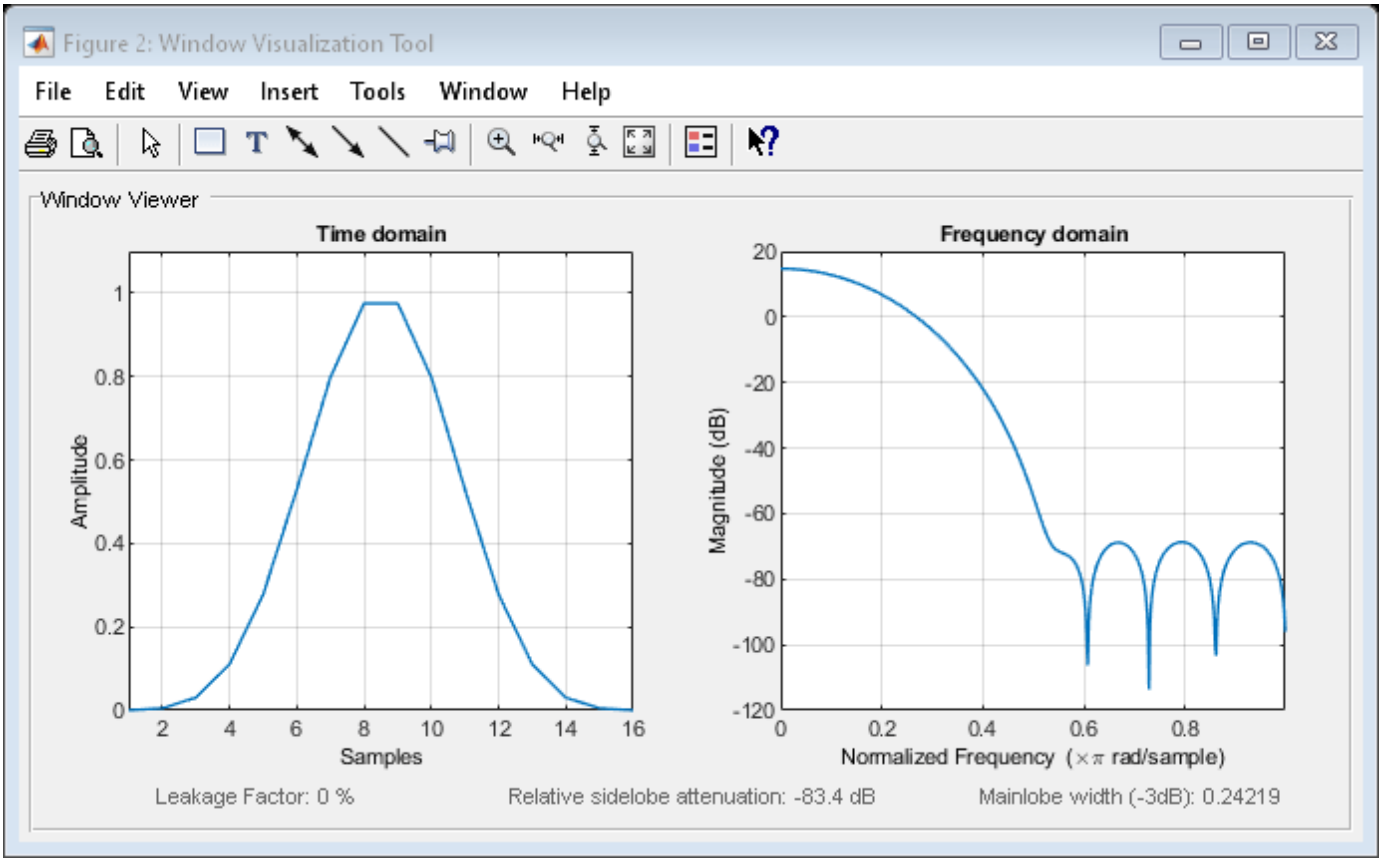
```
win = 16×1

    0.0004
    0.0048
    0.0306
    0.1105
    0.2778
    0.5292
    0.7983
    0.9755
    0.9755
    0.7983
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Nuttall Window            '
    '--------------            '
```

```
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

`nuttallwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

info

Class: `sigwin.nuttallwin`
Package: `sigwin`

Display information about Nuttall defined four-term Blackman-Harris window object

Syntax

```
info(H)
info_win = info(H)
```

Description

`info(H)` displays length information about the Nuttall defined four-term Blackman-Harris window object `H`.

`info_win = info(H)` returns length information about the Nuttall defined four-term Blackman-Harris window object `H` in the character array `info_win`.

Examples

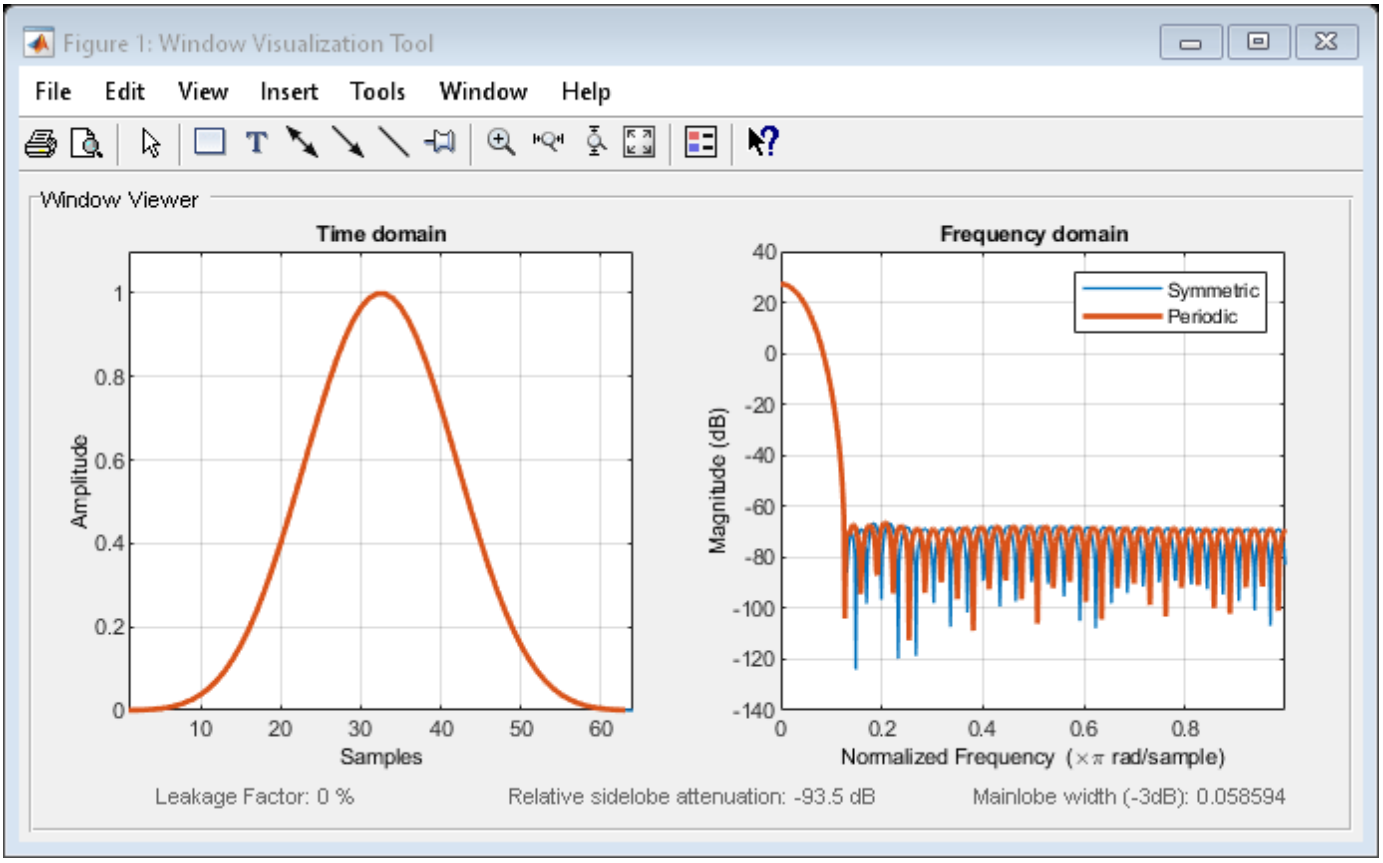
Nuttall-Defined Four-Term Blackman-Harris Windows

Generate two Nuttall-defined four-term Blackman-Harris windows:

- The first window has $N = 64$ and is symmetric.
- The second window has $N = 63$ and is periodic.

Display the two windows.

```
Hs = sigwin.nuttallwin(64);
Hp = sigwin.nuttallwin(63);
Hp.SamplingFlag = 'periodic';

wvt = wvtool(Hs,Hp);
legend(wvt.CurrentAxes,'Symmetric','Periodic')
```



Generate a symmetric Nuttall-defined four-term Blackman-Harris window with $N = 16$. Return the window values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.nuttallwin(16);

win = generate(H)
```

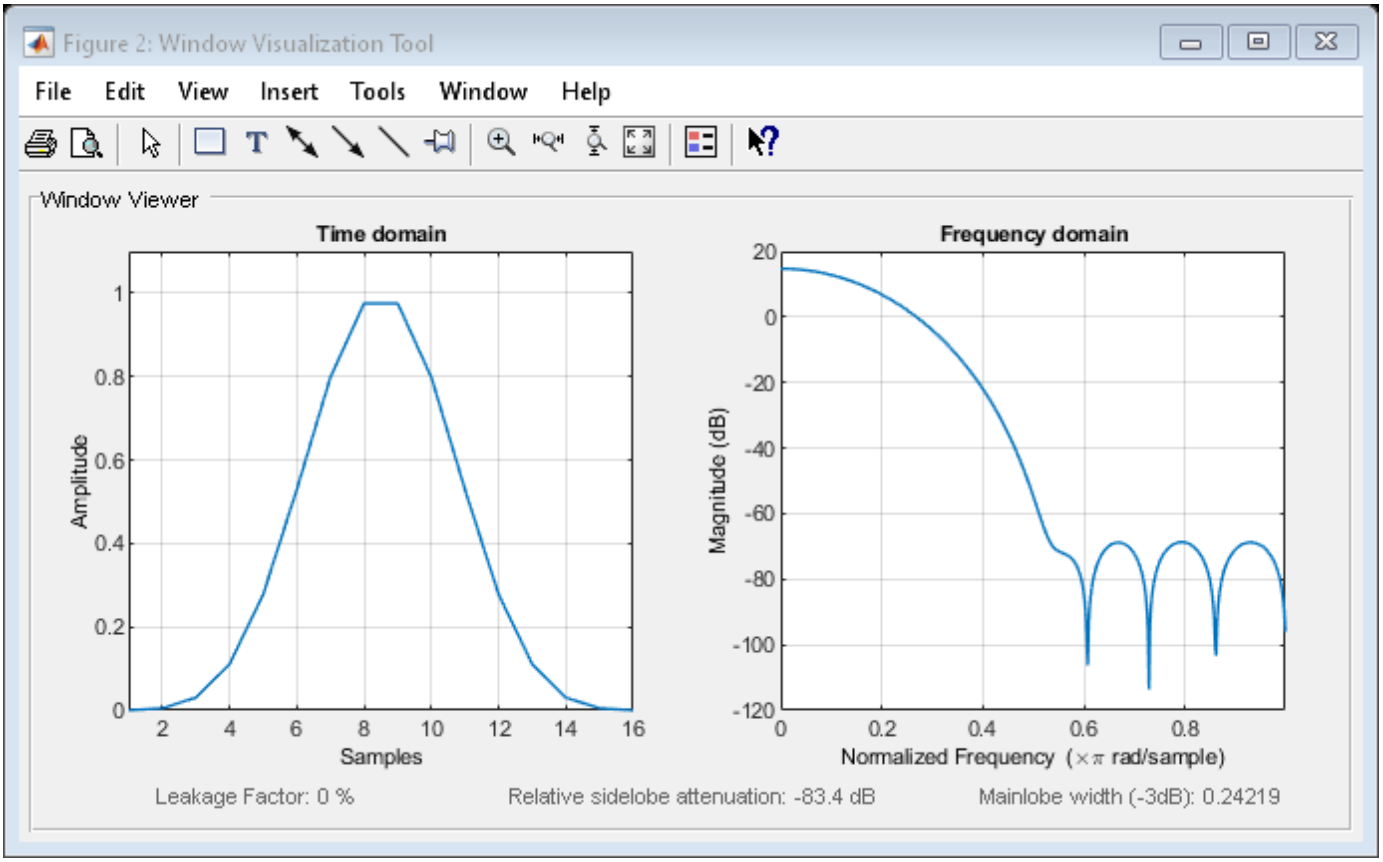
```
win = 16×1

    0.0004
    0.0048
    0.0306
    0.1105
    0.2778
    0.5292
    0.7983
    0.9755
    0.9755
    0.7983
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Nuttall Window            '
    '--------------            '
```

```
    'Length         : 16       '
    'Sampling Flag  : symmetric'
```

```
wvtool(H)
```



See Also

`nuttallwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

winwrite

Class: `sigwin.nuttallwin`
Package: `sigwin`

Save Nuttall defined four-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 four-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 four-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

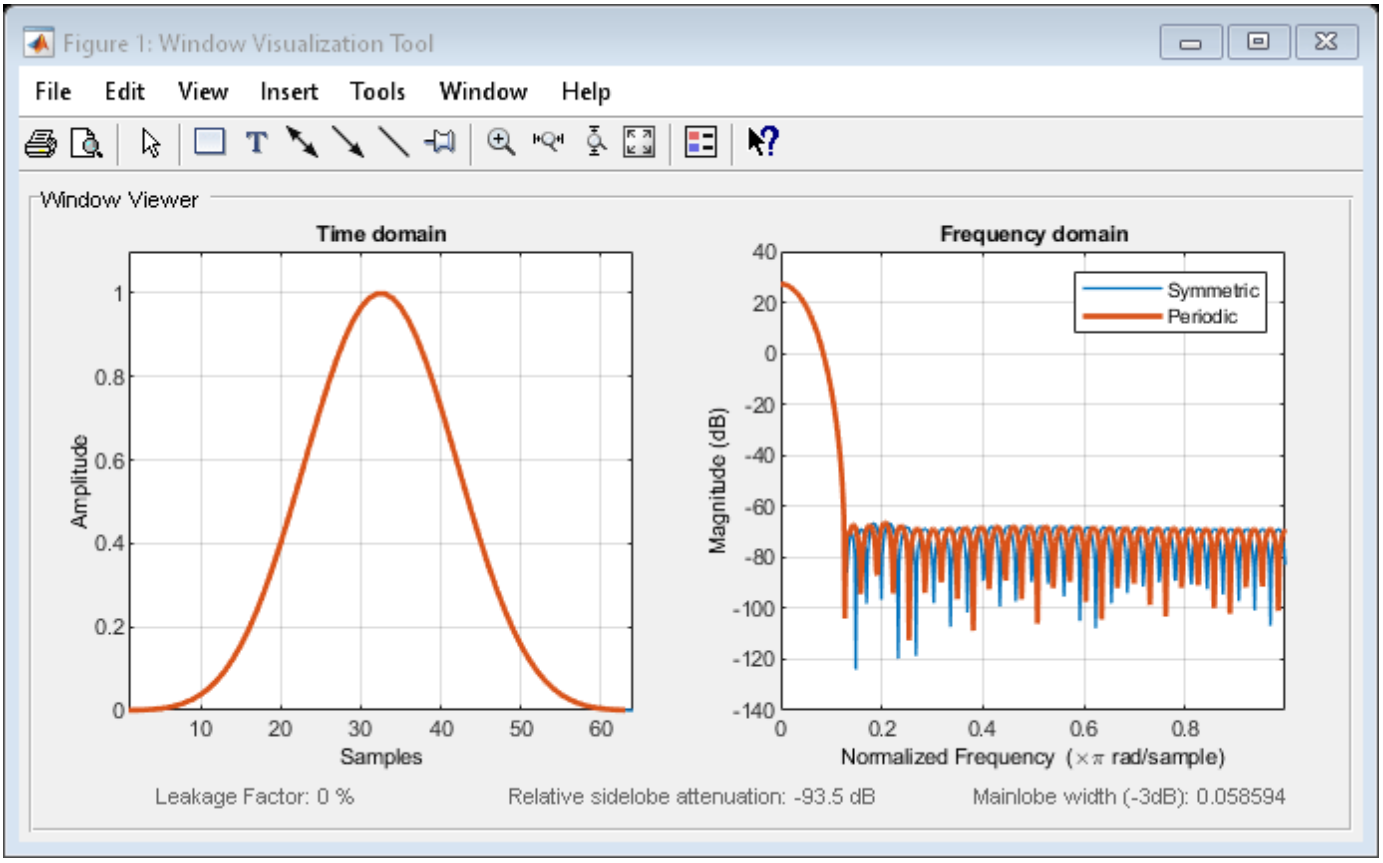
Nuttall-Defined Four-Term Blackman-Harris Windows

Generate two Nuttall-defined four-term Blackman-Harris windows:

- The first window has $N = 64$ and is symmetric.
- The second window has $N = 63$ and is periodic.

Display the two windows.

```
Hs = sigwin.nuttallwin(64);
Hp = sigwin.nuttallwin(63);
Hp.SamplingFlag = 'periodic';

wvt = wvtool(Hs,Hp);
legend(wvt.CurrentAxes,'Symmetric','Periodic')
```



Generate a symmetric Nuttall-defined four-term Blackman-Harris window with $N = 16$. Return the window values as a column vector. Show information about the window object. Display the window.

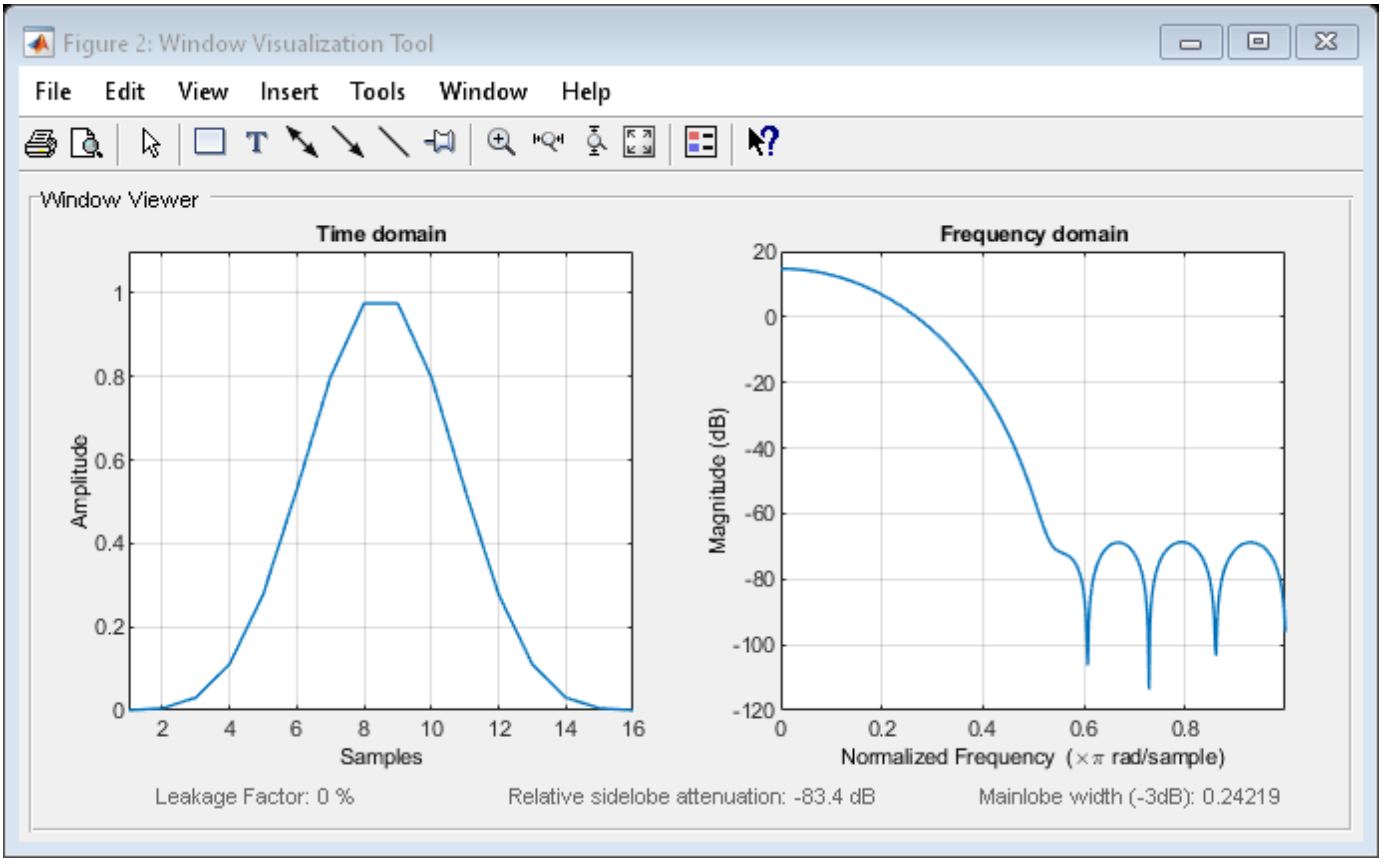
```
H = sigwin.nuttallwin(16);

win = generate(H)
```

```
win = 16×1

    0.0004
    0.0048
    0.0306
    0.1105
    0.2778
    0.5292
    0.7983
    0.9755
    0.9755
    0.7983
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x26 char array
    'Nuttall Window            '
    '--------------            '
```

```
    'Length         : 16       '
    'Sampling Flag  : symmetric'

wvtool(H)
```



See Also

`nuttallwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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 $-\frac{(N-1)}{2} \le n \le \frac{(N-1)}{2}$:

$$w(n) = \begin{cases} 1 - 6\left(\frac{|n|}{N/2}\right)^2 + 6\left(\frac{|n|}{N/2}\right)^3, & 0 \le |n| \le (N-1)/4 \\ 2\left(1 - \frac{|n|}{N/2}\right)^3, & (N-1)/4 < |n| \le (N-1)/2 \end{cases}$$

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

Parzen Window

Generate a Parzen window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

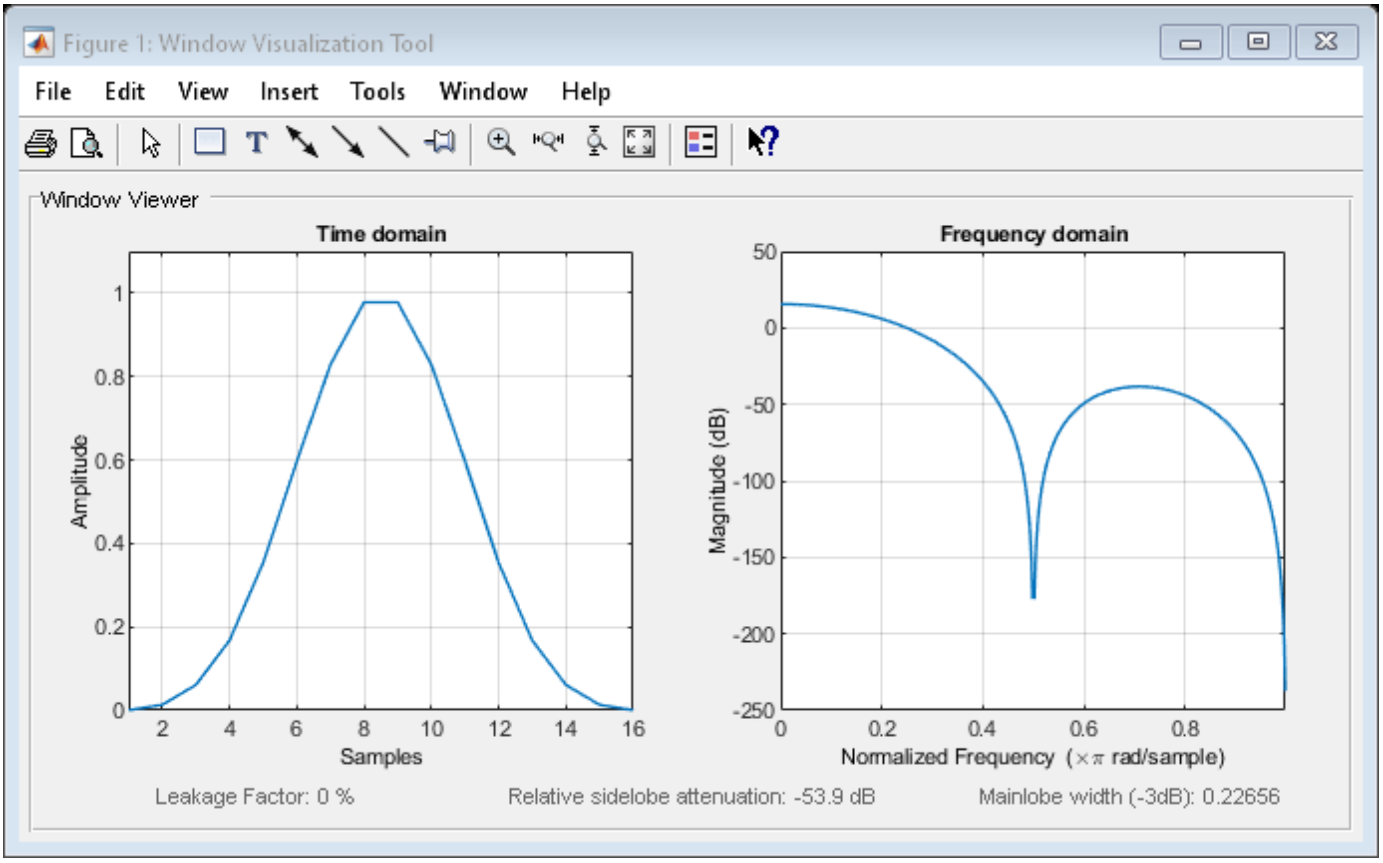
```
H = sigwin.parzenwin(16);

win = generate(H)
```

```
win = 16×1

    0.0005
    0.0132
    0.0610
    0.1675
    0.3540
    0.5972
    0.8286
    0.9780
    0.9780
    0.8286
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x13 char array
    'Parzen Window'
    '-------------'
    'Length  : 16 '
```

```
wvtool(H)
```



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**

Topics
“Windows”
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

Parzen Window

Generate a Parzen window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

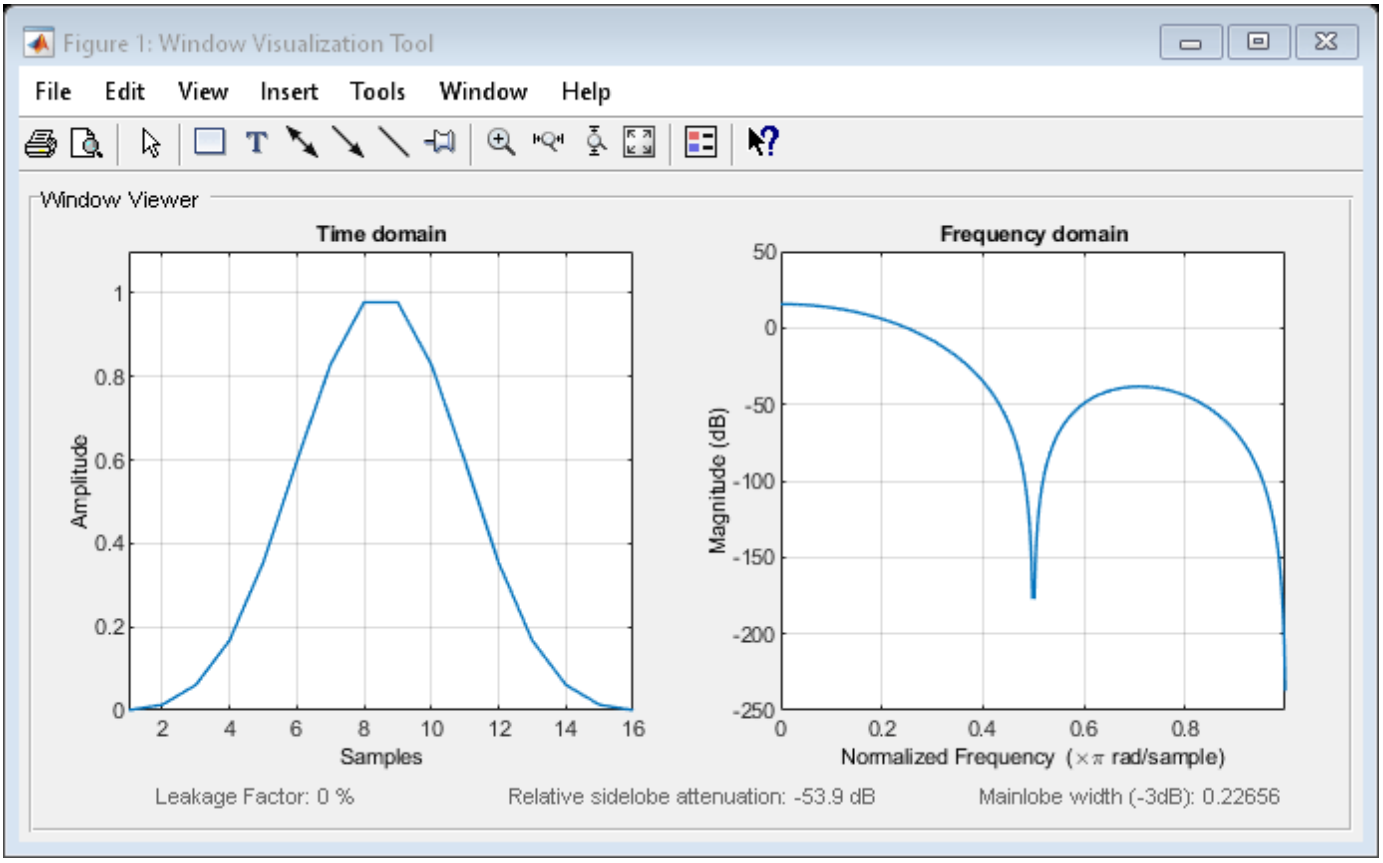
```
H = sigwin.parzenwin(16);

win = generate(H)
```

```
win = 16×1

    0.0005
    0.0132
    0.0610
    0.1675
    0.3540
    0.5972
    0.8286
    0.9780
    0.9780
    0.8286
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x13 char array
    'Parzen Window'
    '-------------'
    'Length  : 16 '
```

```
wvtool(H)
```



See Also

`parzenwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Parzen Window

Generate a Parzen window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

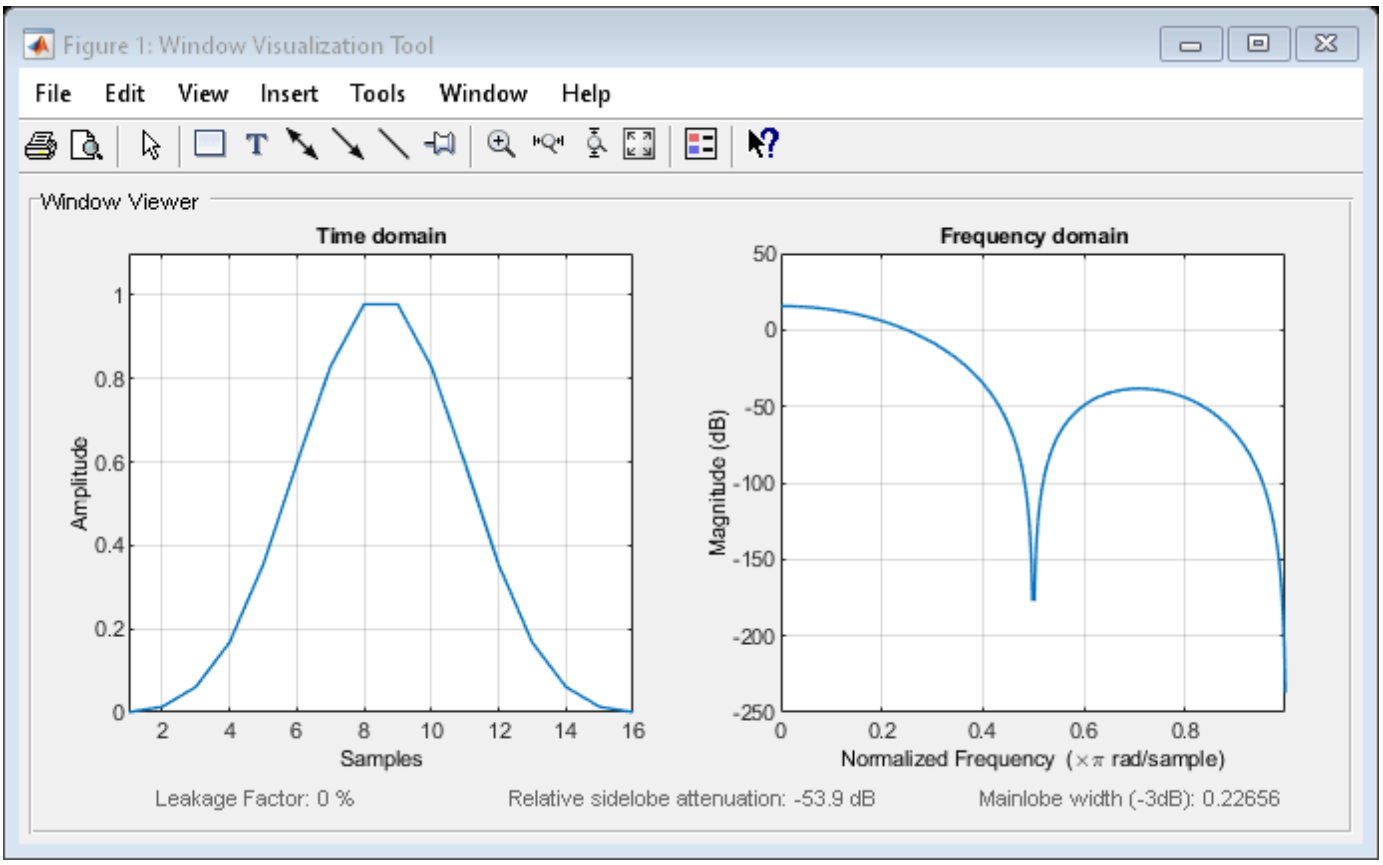
```
H = sigwin.parzenwin(16);

win = generate(H)
```

```
win = 16×1

    0.0005
    0.0132
    0.0610
    0.1675
    0.3540
    0.5972
    0.8286
    0.9780
    0.9780
    0.8286
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x13 char array
    'Parzen Window'
    '-------------'
    'Length  : 16 '
```

```
wvtool(H)
```



See Also

`parzenwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Parzen Window

Generate a Parzen window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

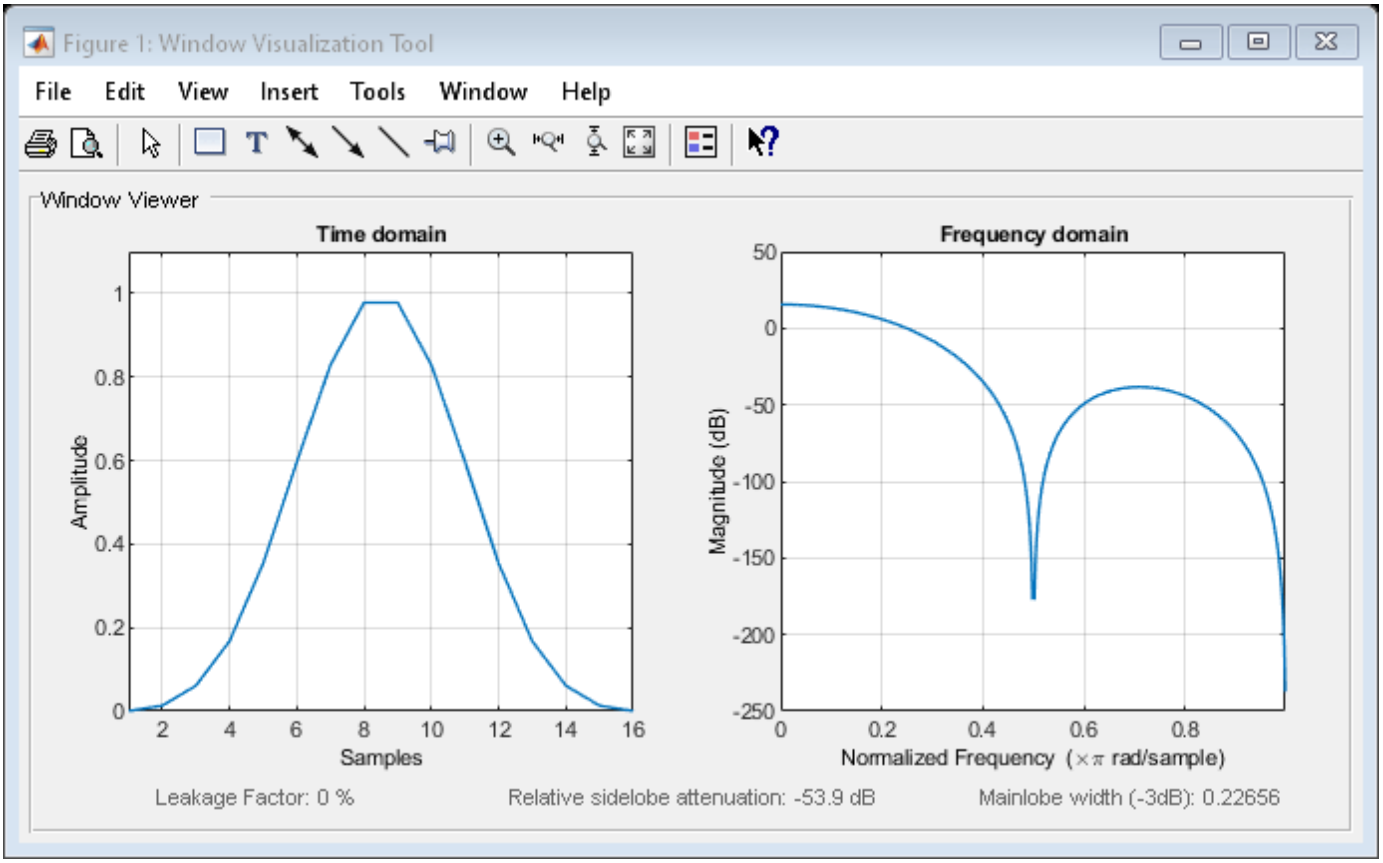
```
H = sigwin.parzenwin(16);

win = generate(H)
```

```
win = 16×1

    0.0005
    0.0132
    0.0610
    0.1675
    0.3540
    0.5972
    0.8286
    0.9780
    0.9780
    0.8286
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x13 char array
    'Parzen Window'
    '-------------'
```

```
'Length  : 16 '

wvtool(H)
```



See Also

`parzenwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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:

$$w(n) = 1, \quad 0 \le n \le N - 1$$

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

Rectangular Window

Generate a rectangular window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

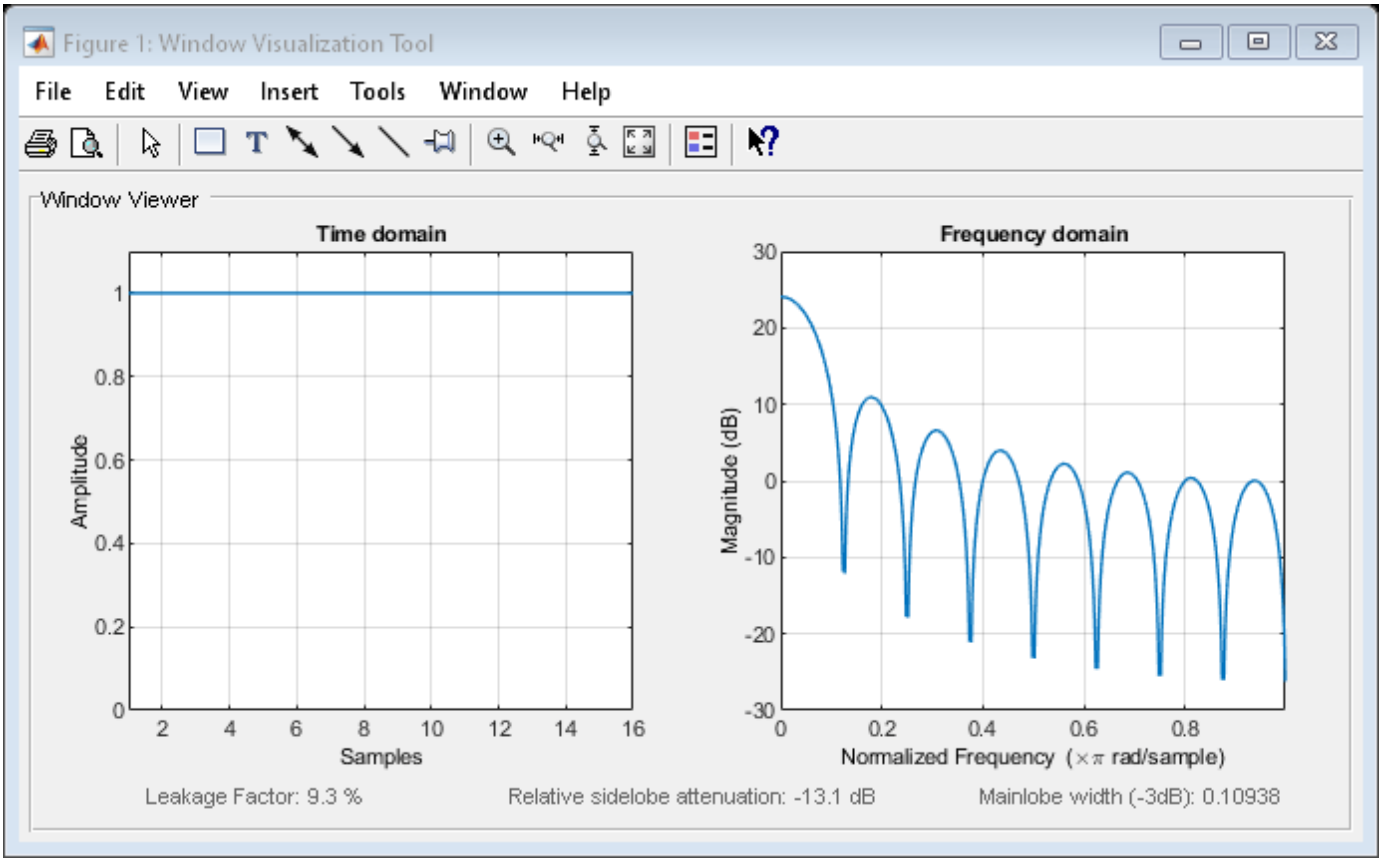
```
H = sigwin.rectwin(16);

win = generate(H)
```

```
win = 16×1

     1
     1
     1
     1
     1
     1
     1
     1
     1
     1
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x18 char array
    'Rectangular Window'
    '------------------'
    'Length  : 16      '
```

```
wvtool(H)
```



References

Oppenheim, Alan V., and Ronald W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

See Also

`rectwin` | `window` | **WVTool**

Topics

“Windows”
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

Rectangular Window

Generate a rectangular window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

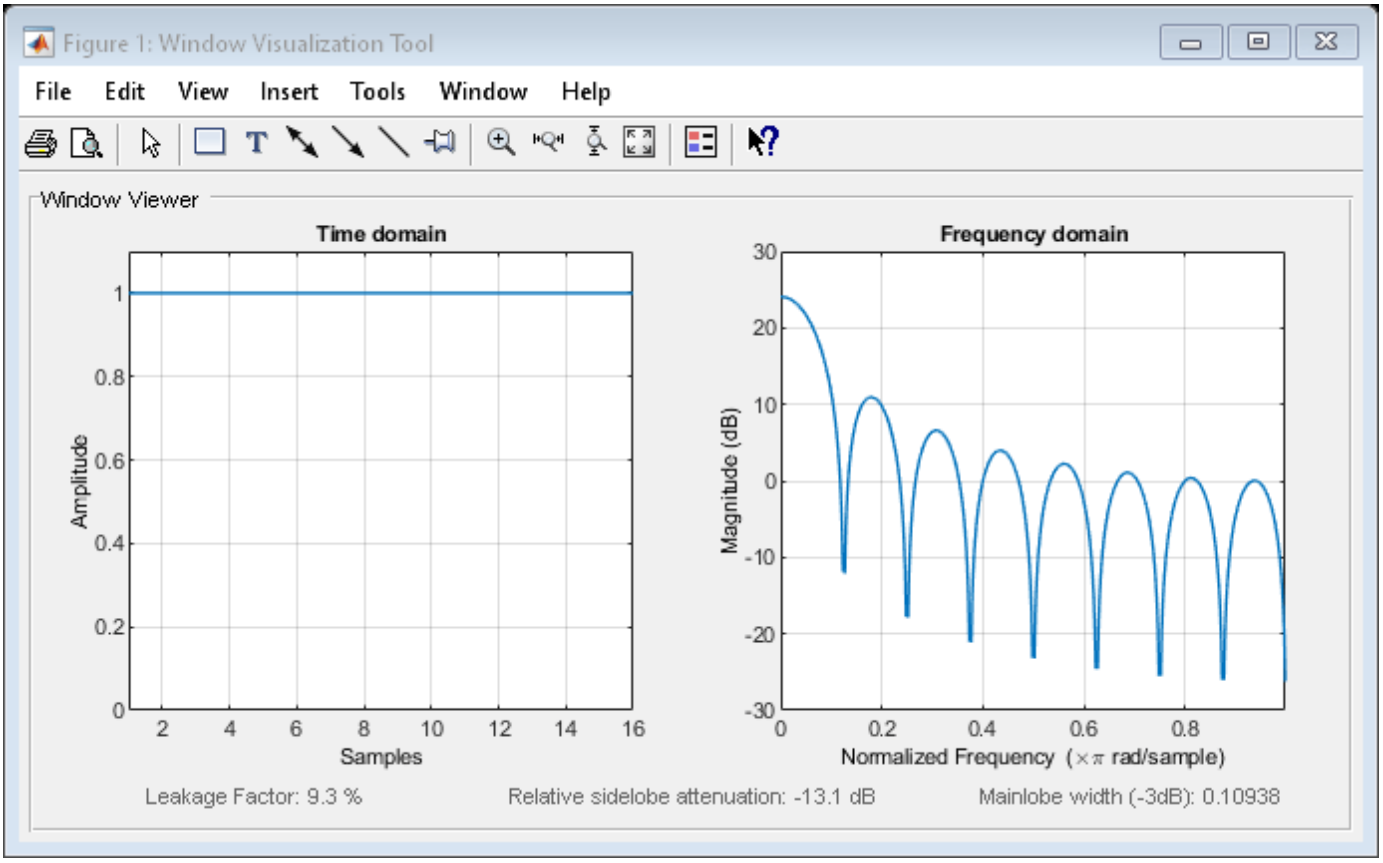
```
H = sigwin.rectwin(16);

win = generate(H)
```

```
win = 16×1

     1
     1
     1
     1
     1
     1
     1
     1
     1
     1
     ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x18 char array
    'Rectangular Window'
    '------------------'
    'Length  : 16      '
```

```
wvtool(H)
```



See Also

`rectwin` | `window` | **WVTool**

Topics

"Windows"
Class Attributes
Property Attributes

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

Rectangular Window

Generate a rectangular window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.rectwin(16);
```

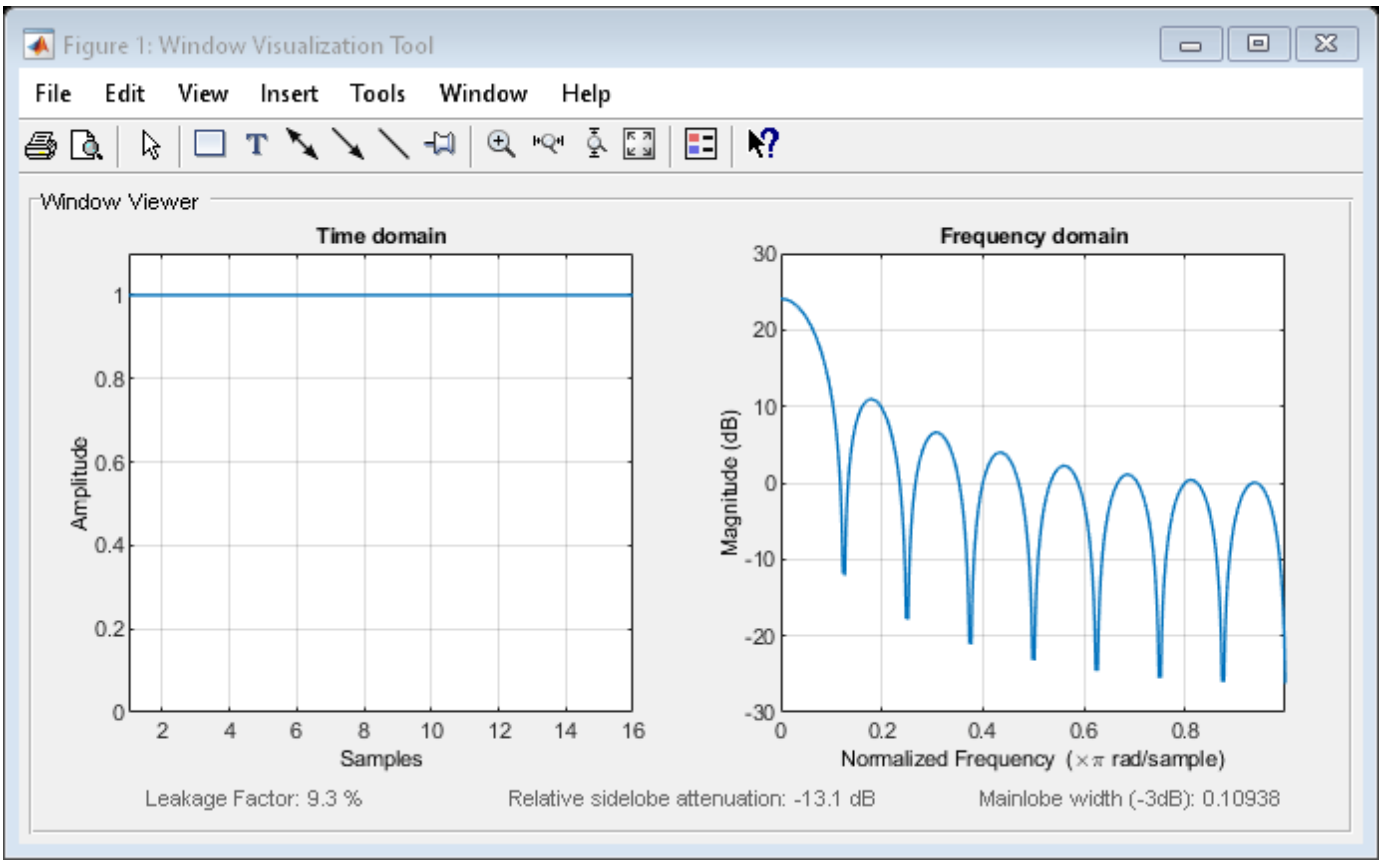
```
win = generate(H)
```

```
win = 16×1

     1
     1
     1
     1
     1
     1
     1
     1
     1
     1
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x18 char array
    'Rectangular Window'
    '------------------'
    'Length  : 16      '
```

```
wvtool(H)
```



See Also

`rectwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Rectangular Window

Generate a rectangular window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.rectwin(16);
```

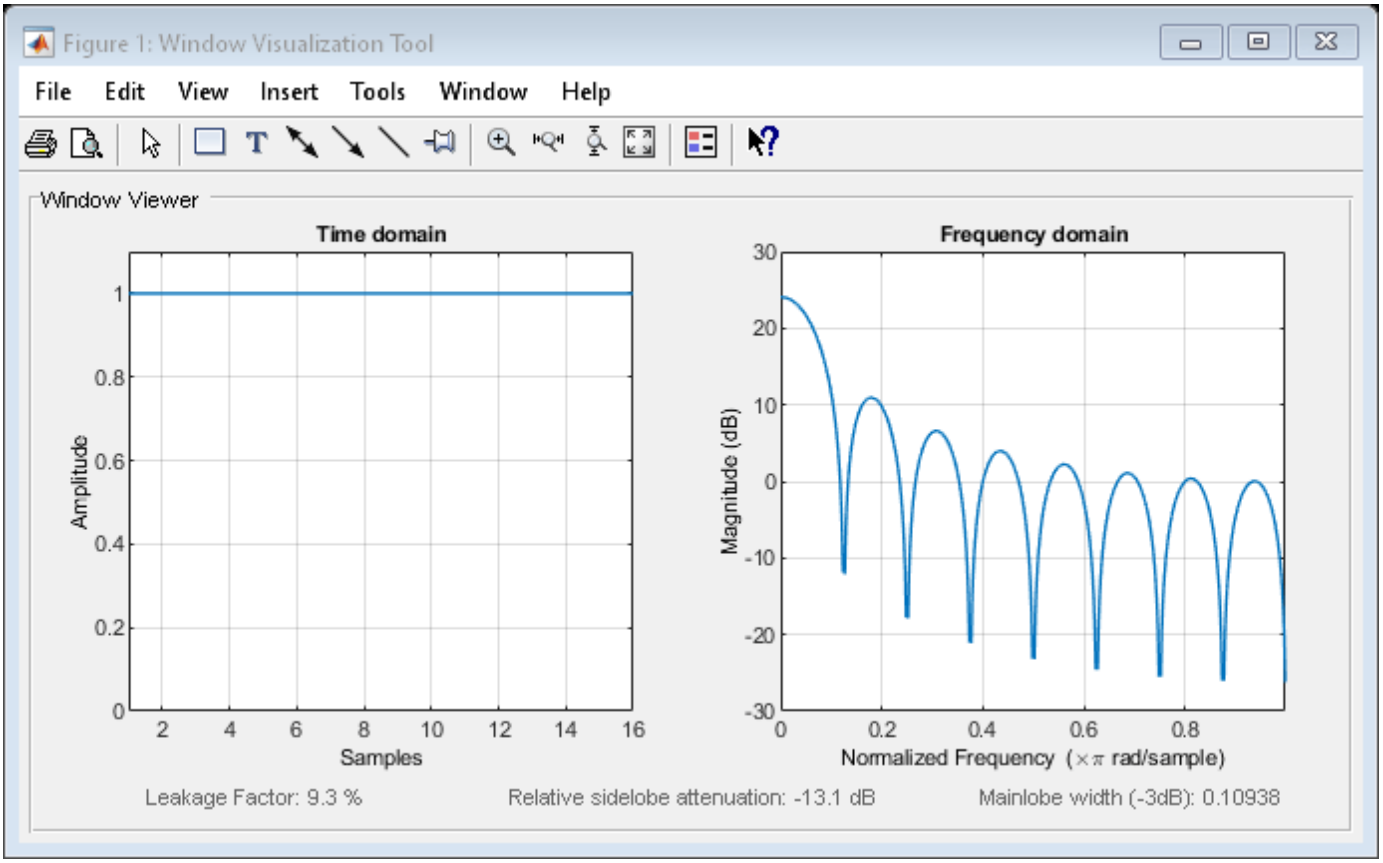
```
win = generate(H)
```

```
win = 16×1

     1
     1
     1
     1
     1
     1
     1
     1
     1
     1
     ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x18 char array
    'Rectangular Window'
    '------------------'
```

```
    'Length  : 16      '

wvtool(H)
```



See Also

`rectwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

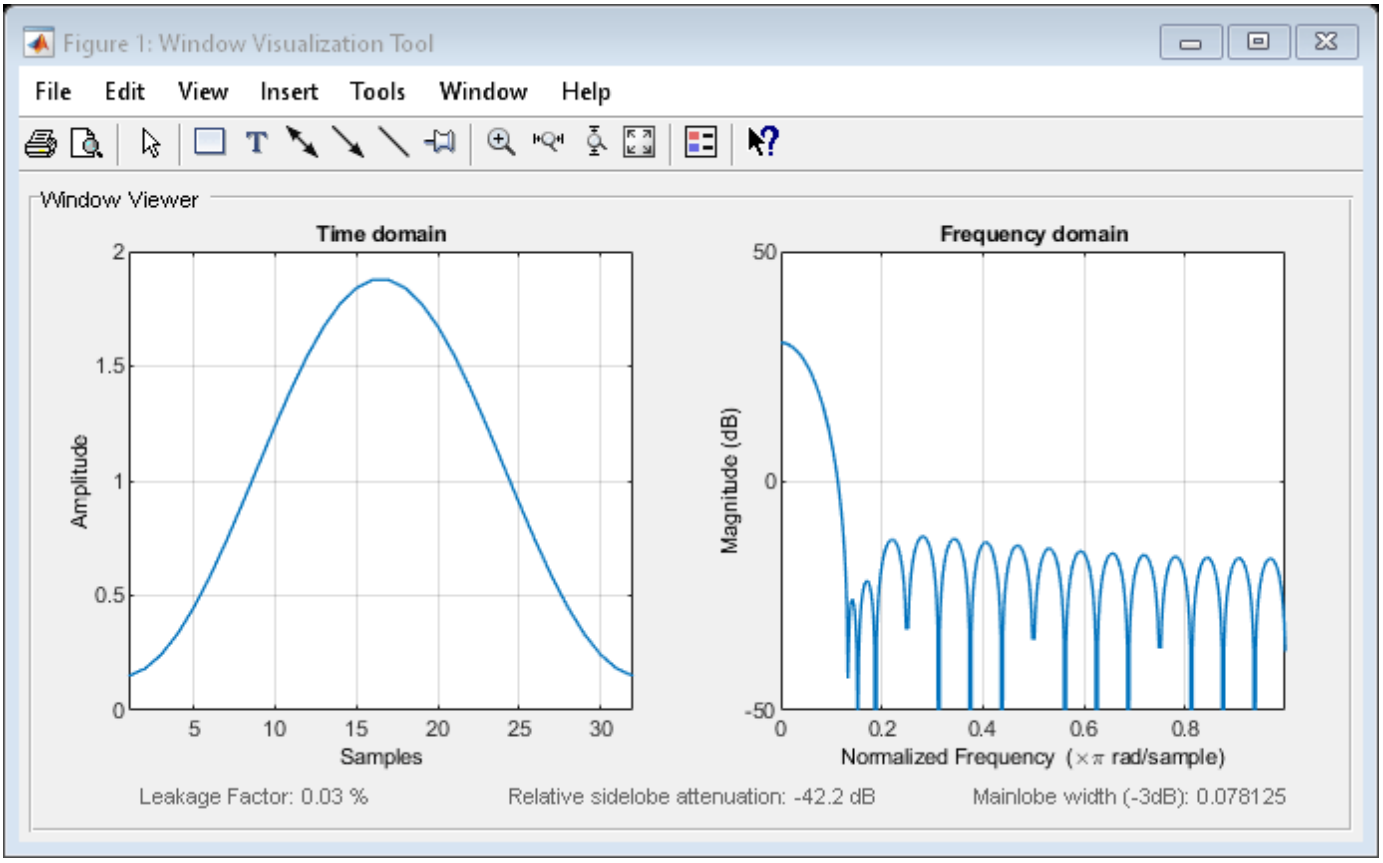
Handle. To learn how copy semantics affect your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Taylor Windows

Generate a Taylor window of length $N = 32$ with a maximum sidelobe level of 60 dB and two constant-level sidelobes adjacent to the mainlobe. Display the window.

```
H = sigwin.taylorwin(32,3,60);

wvt = wvtool(H);
ax = wvt.CurrentAxes;
ax.YLim = [-50 50];
```



Generate a Taylor window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.taylorwin(16);

win = generate(H)
```

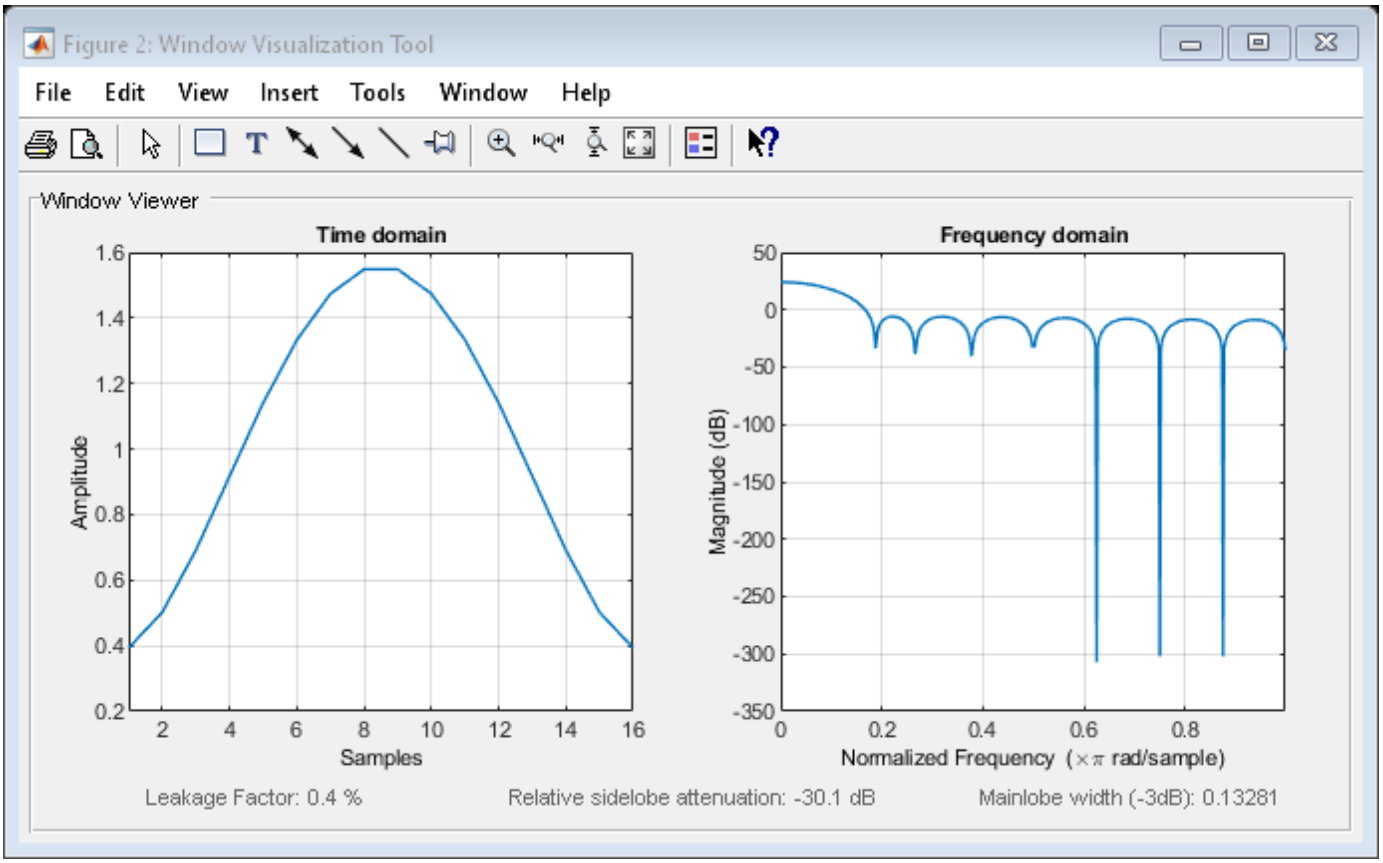
```
win = 16×1

    0.3931
    0.5021
    0.6912
    0.9174
    1.1409
    1.3330
    1.4737
    1.5485
    1.5485
    1.4737
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 5x47 char array
    'Taylor Window                                '
    '-------------                                '
```

```
    'Length                                        : 16'
    'Number of nearly constant-level sidelobes     : 4 '
    'Maximum sidelobe level                        : 30'
```

```
wvtool(H)
```



References

Carrara, W. G., R. M. Majewski, and R. S. Goodman. *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms*. Boston: Artech House Publishers, 1995. Appendix D.2.

See Also

`taylorwin` | `window` | **WVTool**

Topics
“Windows”
Class Attributes
Property Attributes

generate

Class: `sigwin.taylorwin`
Package: `sigwin`

Generates Taylor window

Syntax

```
win = generate(H)
```

Description

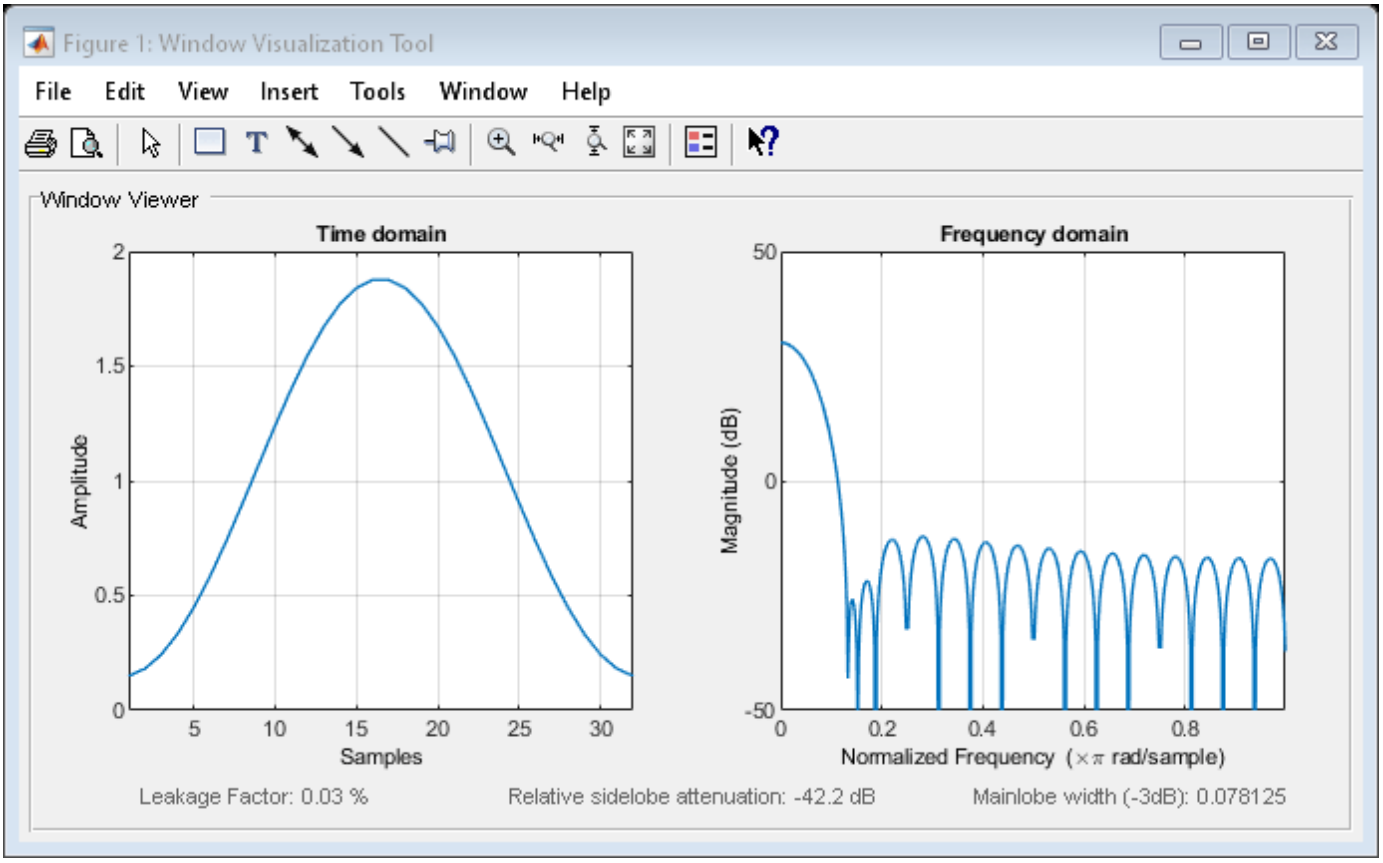
`win = generate(H)` returns the values of the Taylor window object `H` as a double-precision column vector.

Examples

Taylor Windows

Generate a Taylor window of length $N = 32$ with a maximum sidelobe level of 60 dB and two constant-level sidelobes adjacent to the mainlobe. Display the window.

```
H = sigwin.taylorwin(32,3,60);

wvt = wvtool(H);
ax = wvt.CurrentAxes;
ax.YLim = [-50 50];
```



Generate a Taylor window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.taylorwin(16);

win = generate(H)
```

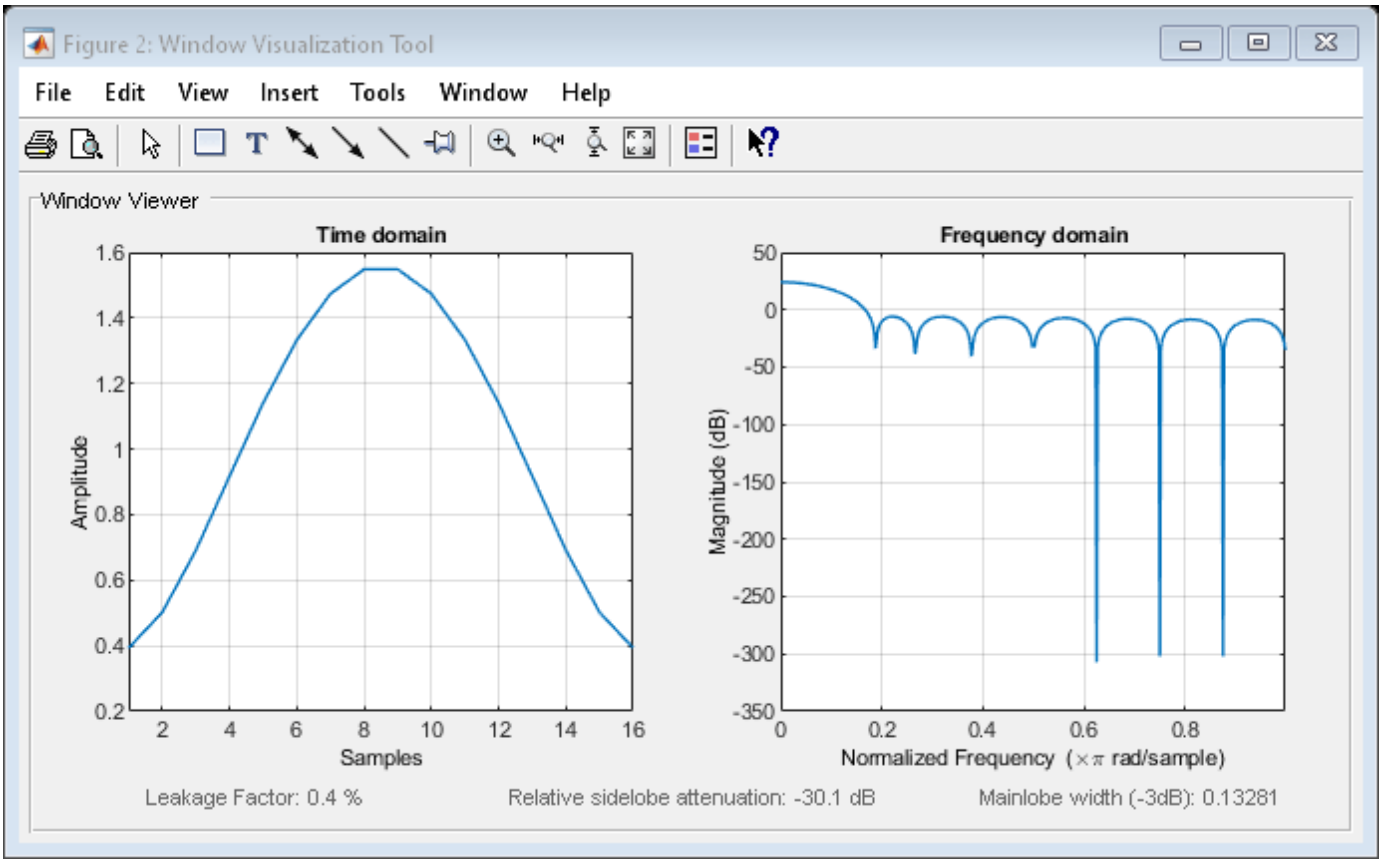
```
win = 16×1

    0.3931
    0.5021
    0.6912
    0.9174
    1.1409
    1.3330
    1.4737
    1.5485
    1.5485
    1.4737
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 5x47 char array
    'Taylor Window                                '
    '-------------                                '
```

```
    'Length                                        : 16'
    'Number of nearly constant-level sidelobes  : 4 '
    'Maximum sidelobe level                        : 30'
```

```
wvtool(H)
```



See Also

`taylorwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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`.

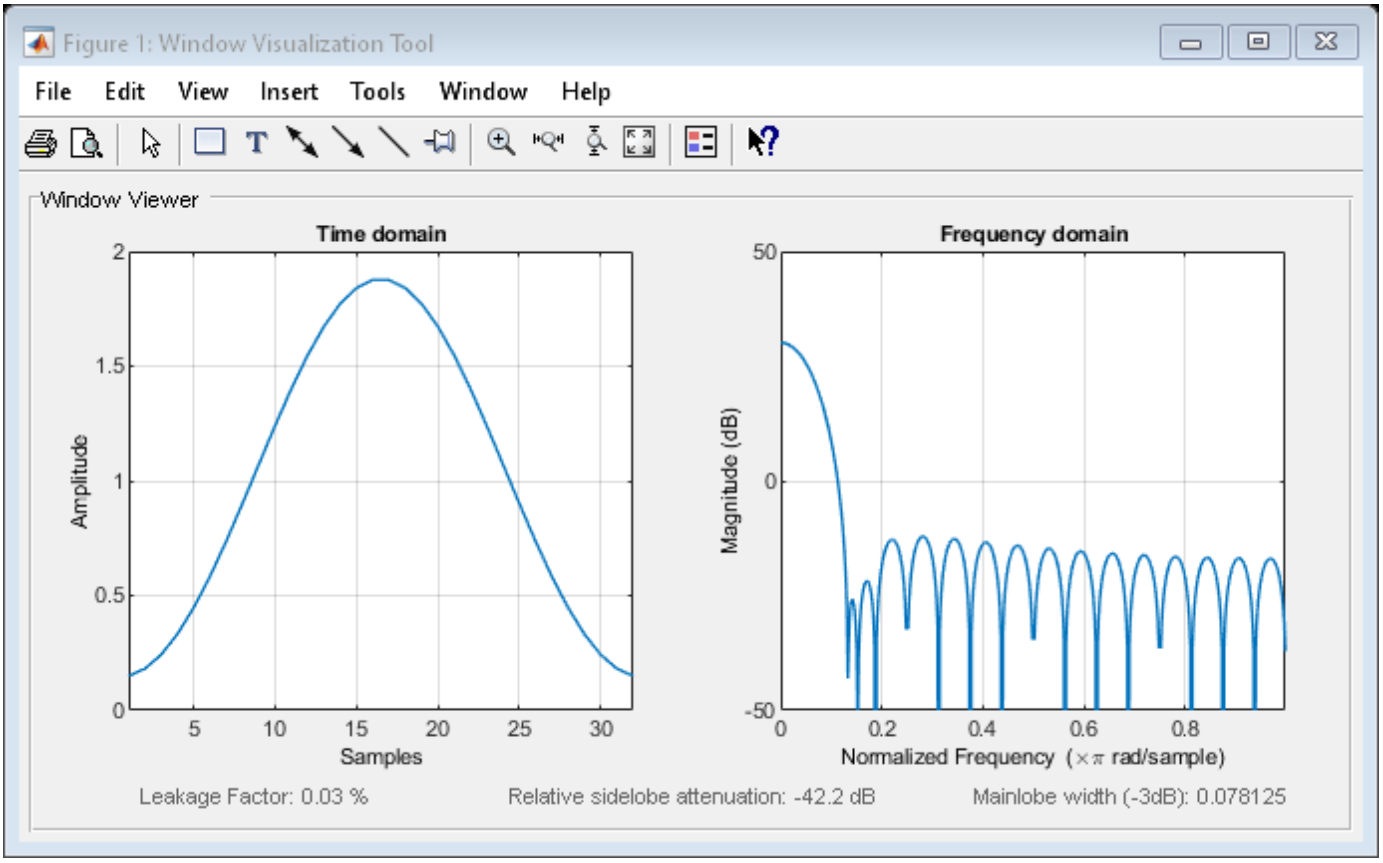
`info_win = info(H)` returns length and sidelobe information for the Taylor window object `H` in the character array `info_win`.

Examples

Taylor Windows

Generate a Taylor window of length $N = 32$ with a maximum sidelobe level of 60 dB and two constant-level sidelobes adjacent to the mainlobe. Display the window.

```
H = sigwin.taylorwin(32,3,60);

wvt = wvtool(H);
ax = wvt.CurrentAxes;
ax.YLim = [-50 50];
```



Generate a Taylor window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.taylorwin(16);

win = generate(H)
```

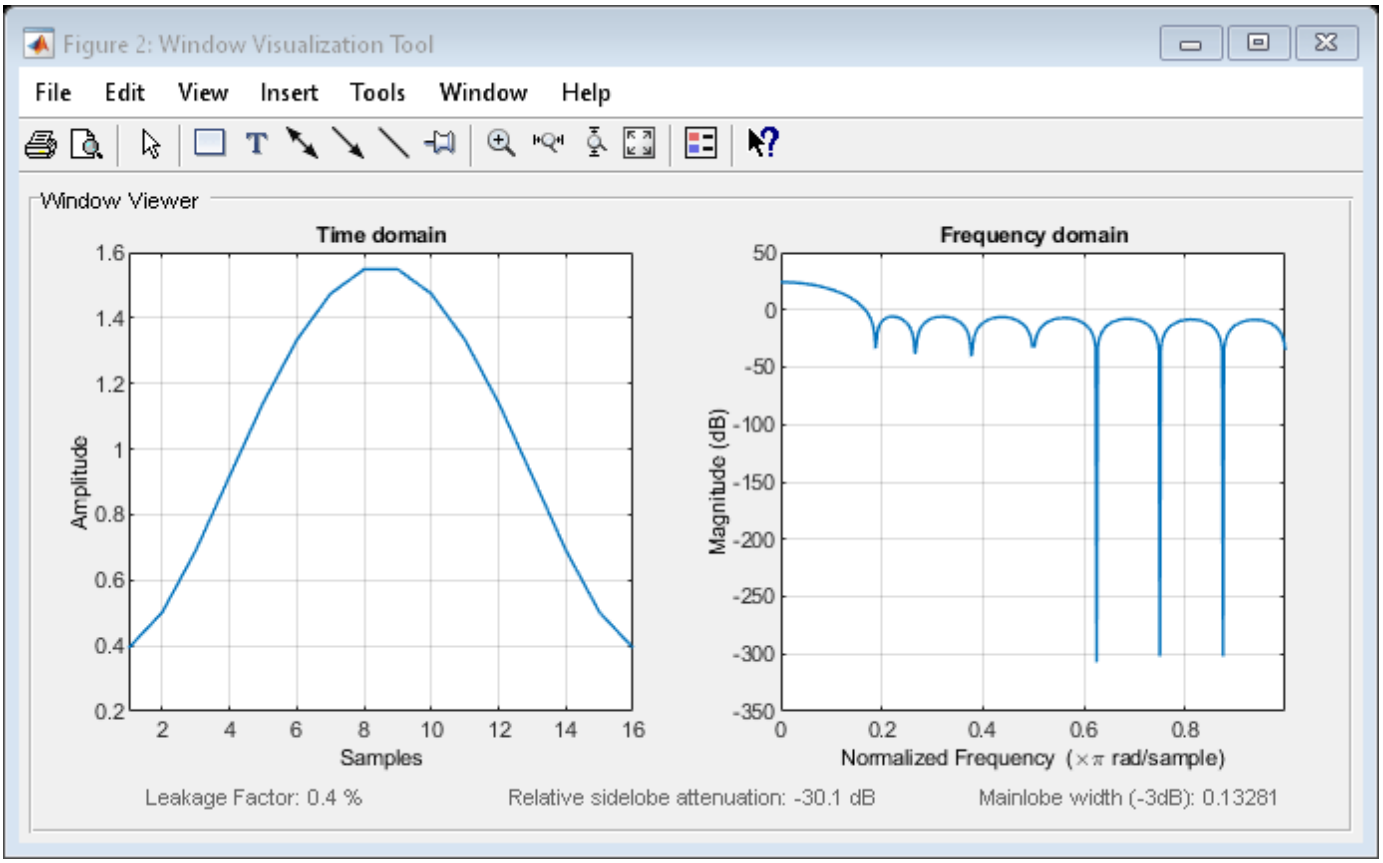
```
win = 16×1

    0.3931
    0.5021
    0.6912
    0.9174
    1.1409
    1.3330
    1.4737
    1.5485
    1.5485
    1.4737
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 5x47 char array
    'Taylor Window                                '
    '-------------                                '
```

```
    'Length                                        : 16'
    'Number of nearly constant-level sidelobes     : 4 '
    'Maximum sidelobe level                        : 30'
```

```
wvtool(H)
```



See Also

`taylorwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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.

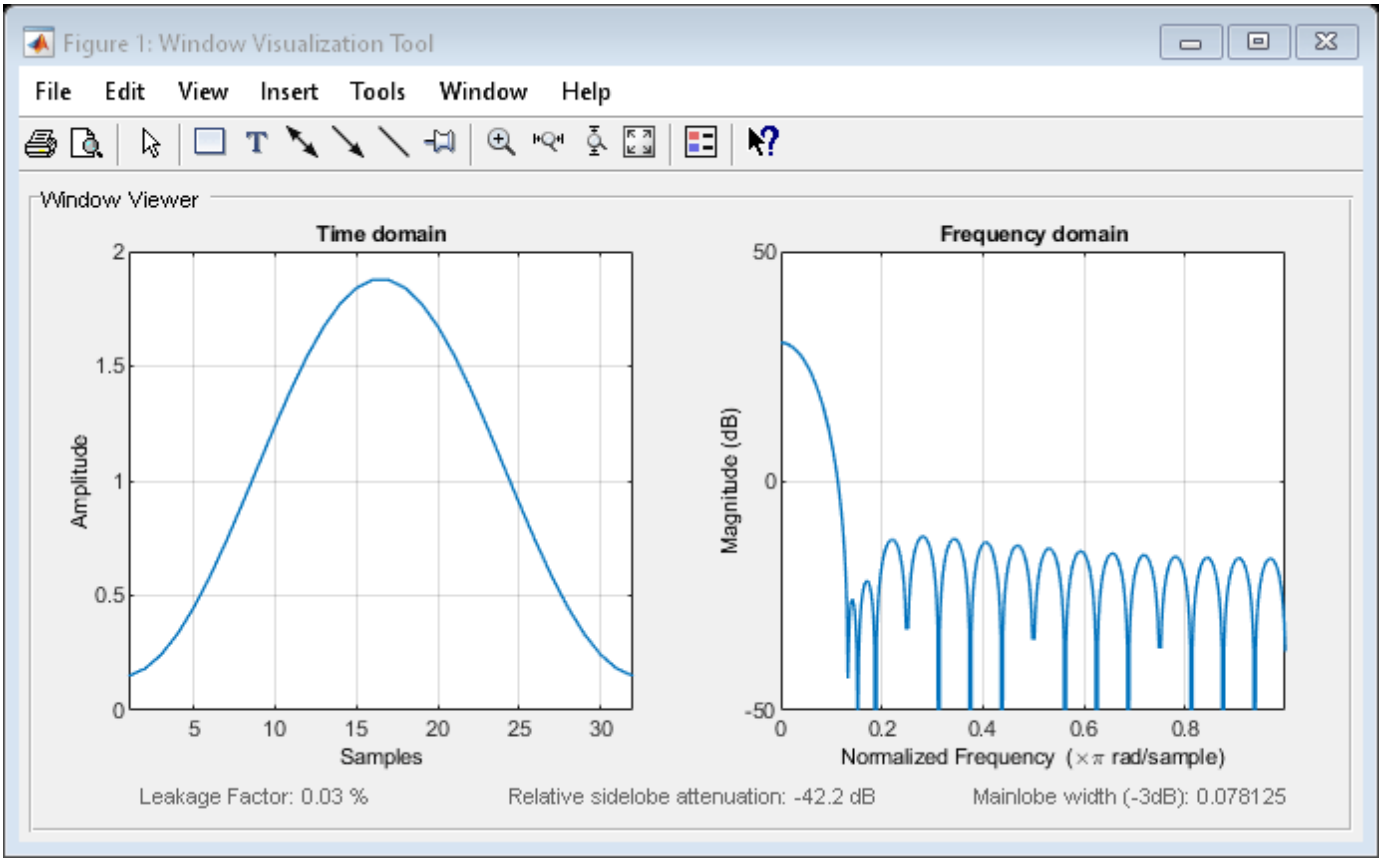
`winwrite(H,'filename')` saves the values of the Taylor window object `H` in the current folder as a column vector in the ASCII file `'filename'`. The file extension `.wf` is automatically appended to `filename`.

Examples

Taylor Windows

Generate a Taylor window of length $N = 32$ with a maximum sidelobe level of 60 dB and two constant-level sidelobes adjacent to the mainlobe. Display the window.

```
H = sigwin.taylorwin(32,3,60);

wvt = wvtool(H);
ax = wvt.CurrentAxes;
ax.YLim = [-50 50];
```



Generate a Taylor window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.taylorwin(16);

win = generate(H)
```

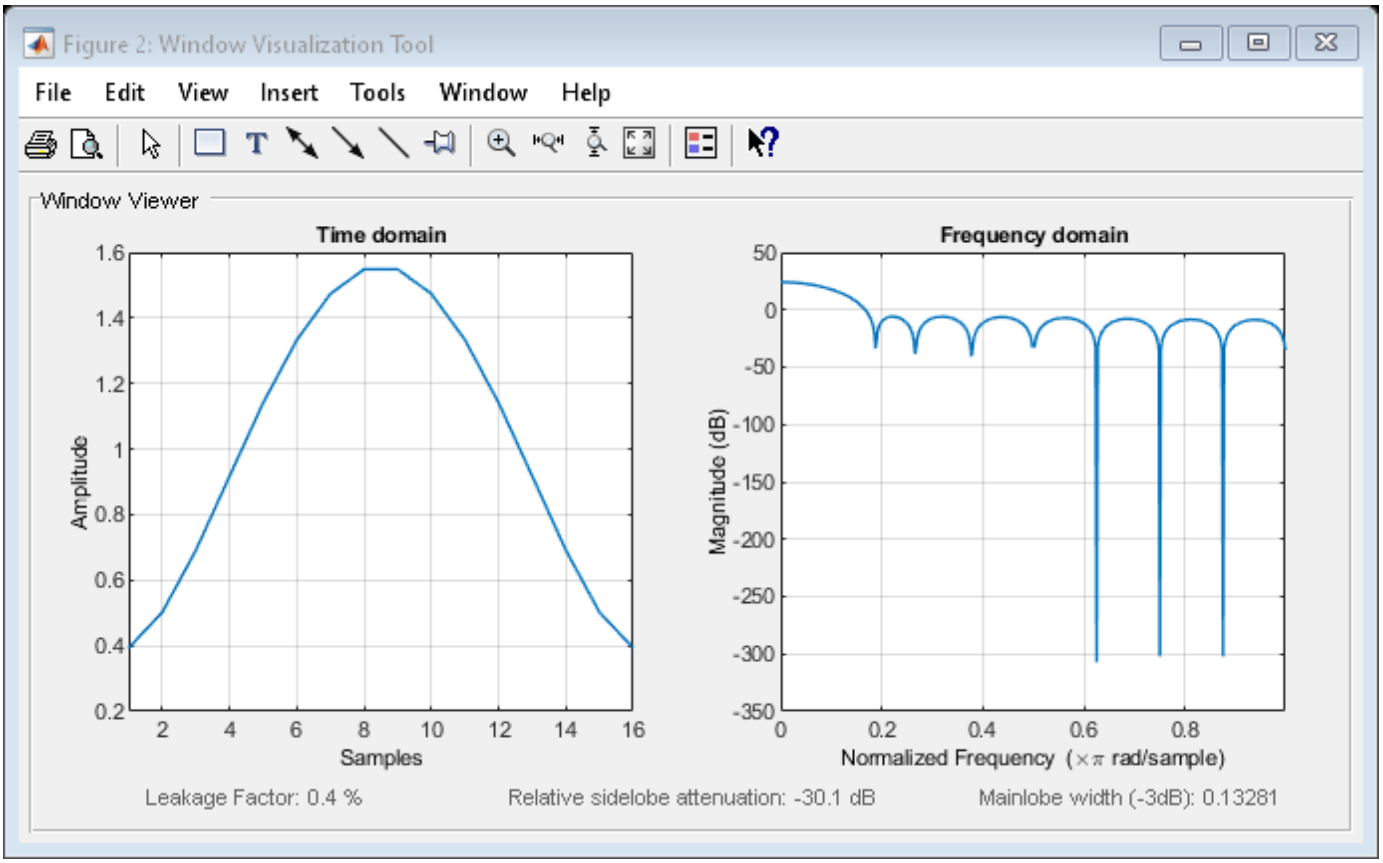
```
win = 16×1

    0.3931
    0.5021
    0.6912
    0.9174
    1.1409
    1.3330
    1.4737
    1.5485
    1.5485
    1.4737
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 5x47 char array
    'Taylor Window                                '
    '-------------                                '
```

```
'Length                                        : 16'
'Number of nearly constant-level sidelobes  : 4 '
'Maximum sidelobe level                        : 30'
```

```
wvtool(H)
```



See Also

`taylorwin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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:

$$w(n) = \begin{cases} \dfrac{2n}{L+1} & 1 \le n \le \dfrac{L+1}{2} \\ 2 - \dfrac{2n}{L+1} & \dfrac{L+1}{2} + 1 \le n \le L \end{cases}$$

For L even, the triangular window is defined as:

$$w(n) = \begin{cases} \dfrac{(2n-1)}{L} & 1 \le n \le \dfrac{L}{2} \\ 2 - \dfrac{(2n-1)}{L} & \dfrac{L}{2} + 1 \le n \le L \end{cases}$$

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

generate	Generates triangular window
info	Display information about triangular window
winwrite	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

Triangular Window

Generate a triangular window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

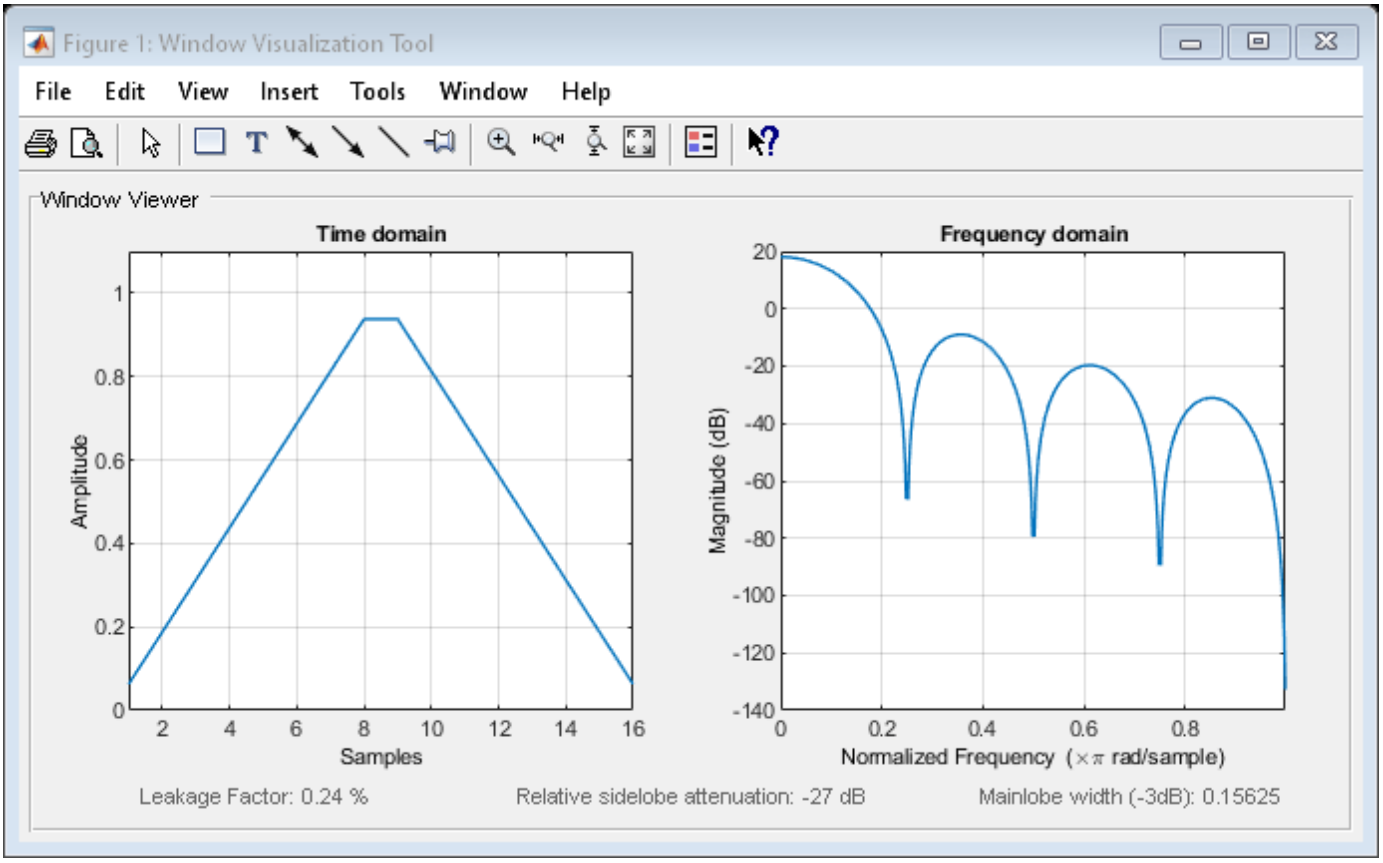
```
H = sigwin.triang(16);

win = generate(H)
```

```
win = 16×1

    0.0625
    0.1875
    0.3125
    0.4375
    0.5625
    0.6875
    0.8125
    0.9375
    0.9375
    0.8125
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x17 char array
    'Triangular Window'
    '------------------'
    'Length  : 16     '
```

```
wvtool(H)
```



References

Oppenheim, Alan V., and Ronald W. Schafer. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1989.

See Also

`triang` | `window` | **WVTool**

Topics

“Windows”
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

Triangular Window

Generate a triangular window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

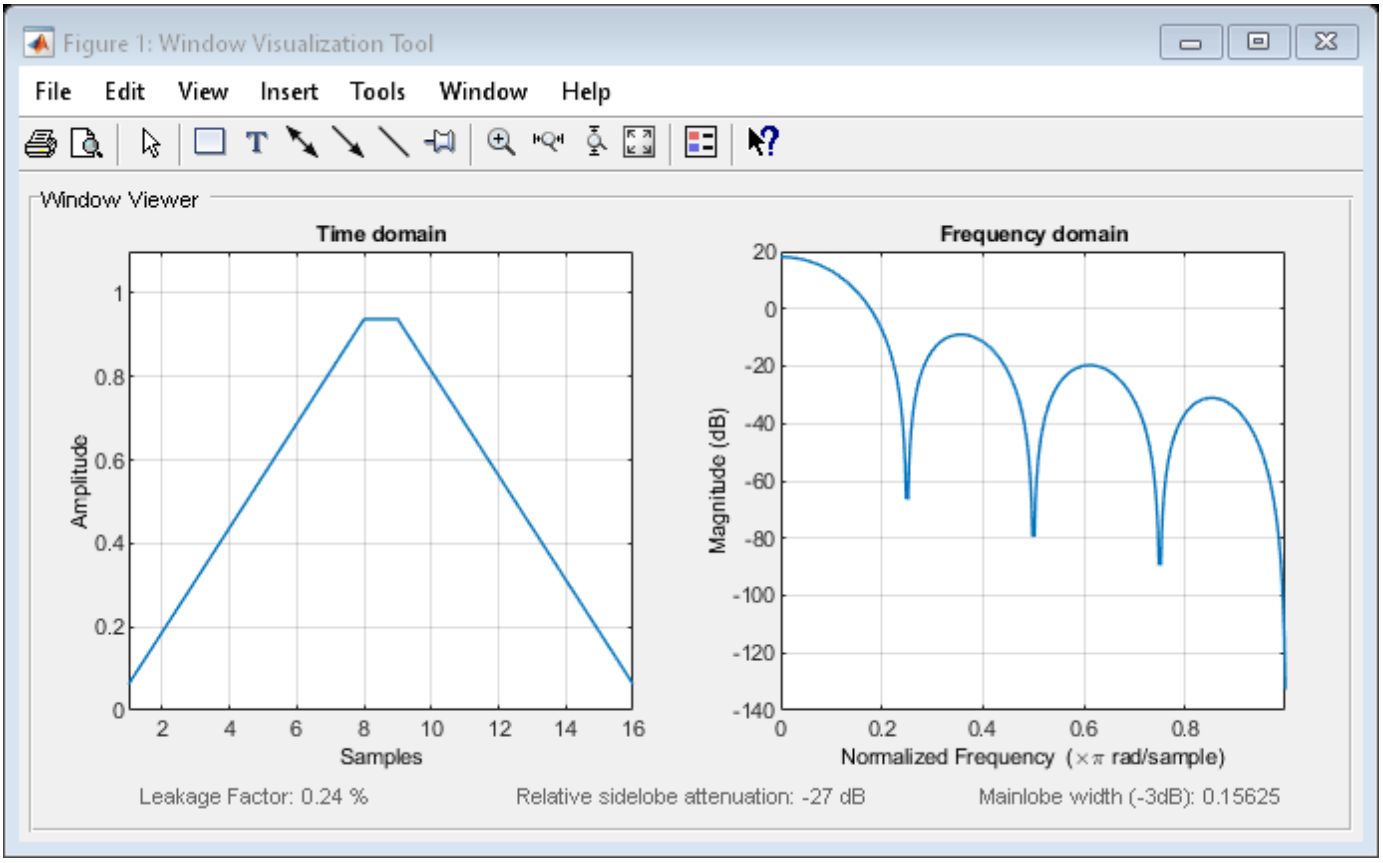
```
H = sigwin.triang(16);

win = generate(H)
```

```
win = 16×1

    0.0625
    0.1875
    0.3125
    0.4375
    0.5625
    0.6875
    0.8125
    0.9375
    0.9375
    0.8125
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x17 char array
    'Triangular Window'
    '-----------------'
    'Length  : 16     '
```

```
wvtool(H)
```



See Also

`triang` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Triangular Window

Generate a triangular window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

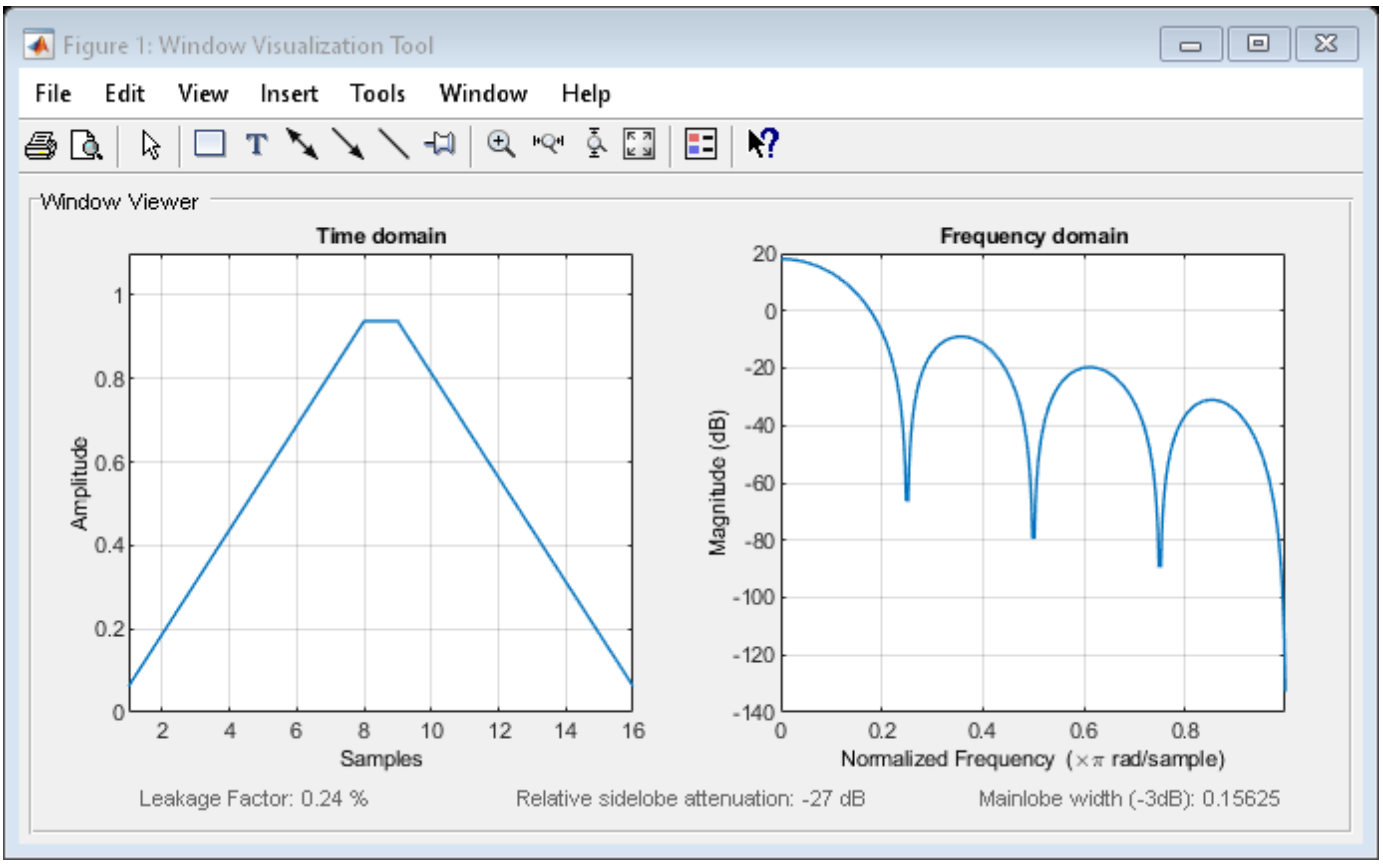
```
H = sigwin.triang(16);

win = generate(H)
```

```
win = 16×1

    0.0625
    0.1875
    0.3125
    0.4375
    0.5625
    0.6875
    0.8125
    0.9375
    0.9375
    0.8125
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x17 char array
    'Triangular Window'
    '-----------------'
    'Length  : 16     '
```

```
wvtool(H)
```



See Also

`triang` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

Triangular Window

Generate a triangular window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

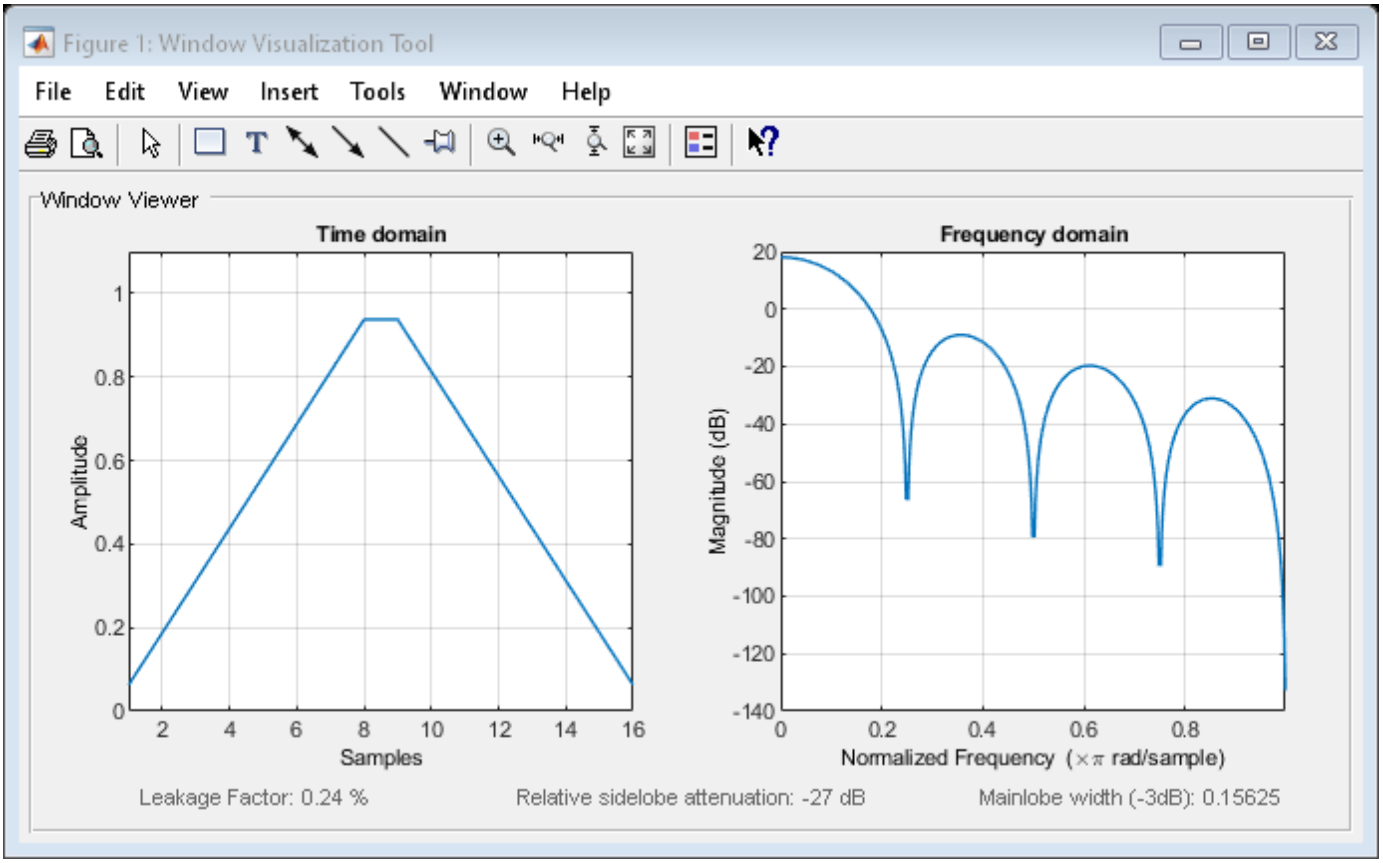
```
H = sigwin.triang(16);

win = generate(H)
```

```
win = 16×1

    0.0625
    0.1875
    0.3125
    0.4375
    0.5625
    0.6875
    0.8125
    0.9375
    0.9375
    0.8125
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 3x17 char array
    'Triangular Window'
    '------------------'
```

```
    'Length  : 16     '

wvtool(H)
```



See Also

`triang` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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:

$$w(x) = \begin{cases} \frac{1}{2}\left\{1 + \cos(\frac{2\pi}{\alpha}[x - \alpha/2])\right\} & 0 \le x < \frac{\alpha}{2} \\ 1 & \frac{\alpha}{2} \le x < 1 - \frac{\alpha}{2} \\ \frac{1}{2}\left\{1 + \cos(\frac{2\pi}{\alpha}[x - 1 + \alpha/2])\right\} & 1 - \frac{\alpha}{2} \le x \le 1 \end{cases}$$

where x is a N–point linearly spaced vector generated using `linspace`. The parameter α is the ratio of cosine-tapered section length to the entire window length with $0 \le \alpha \le 1$. 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 $\alpha \le 0$, an N-point rectangular window is returned. If you specify $\alpha \ge 1$, 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 $0 \le \alpha \le 1$. 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

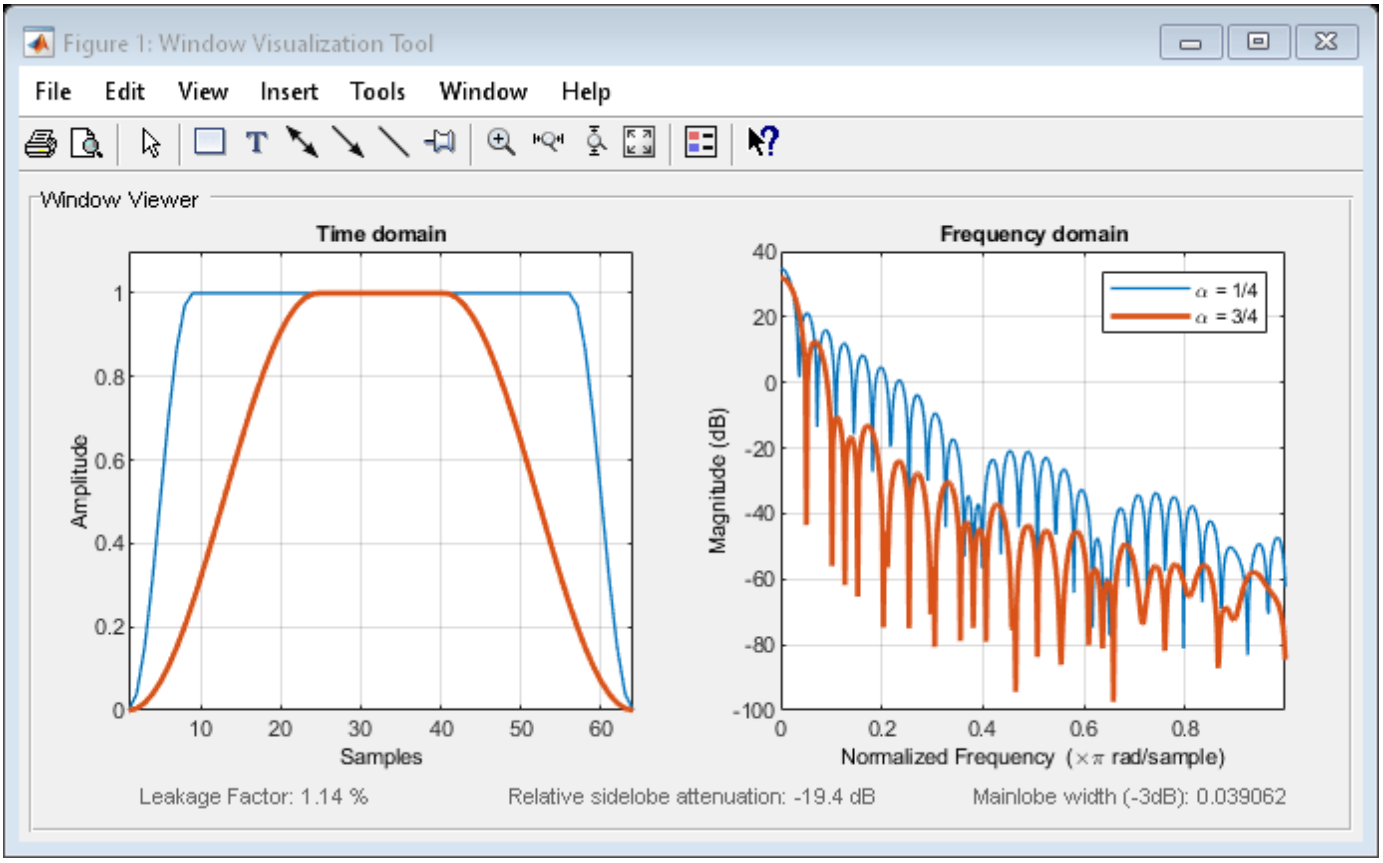
Tukey Windows

Generate two Tukey windows of length $N = 64$:

- The first window has $\alpha = 1/4$. α is the ratio of tapered window section length to constant section length.
- The second window has $\alpha = 3/4$.

Display the two windows.

```
H14 = sigwin.tukeywin(64,1/4);
H34 = sigwin.tukeywin(64,3/4);

wvt = wvtool(H14,H34);
legend(wvt.CurrentAxes,'\alpha = 1/4','\alpha = 3/4')
```



Generate a Tukey window with length $N = 16$ and the default $\alpha = 1/2$. Return its values as a column vector. Show information about the window object. Display the window.

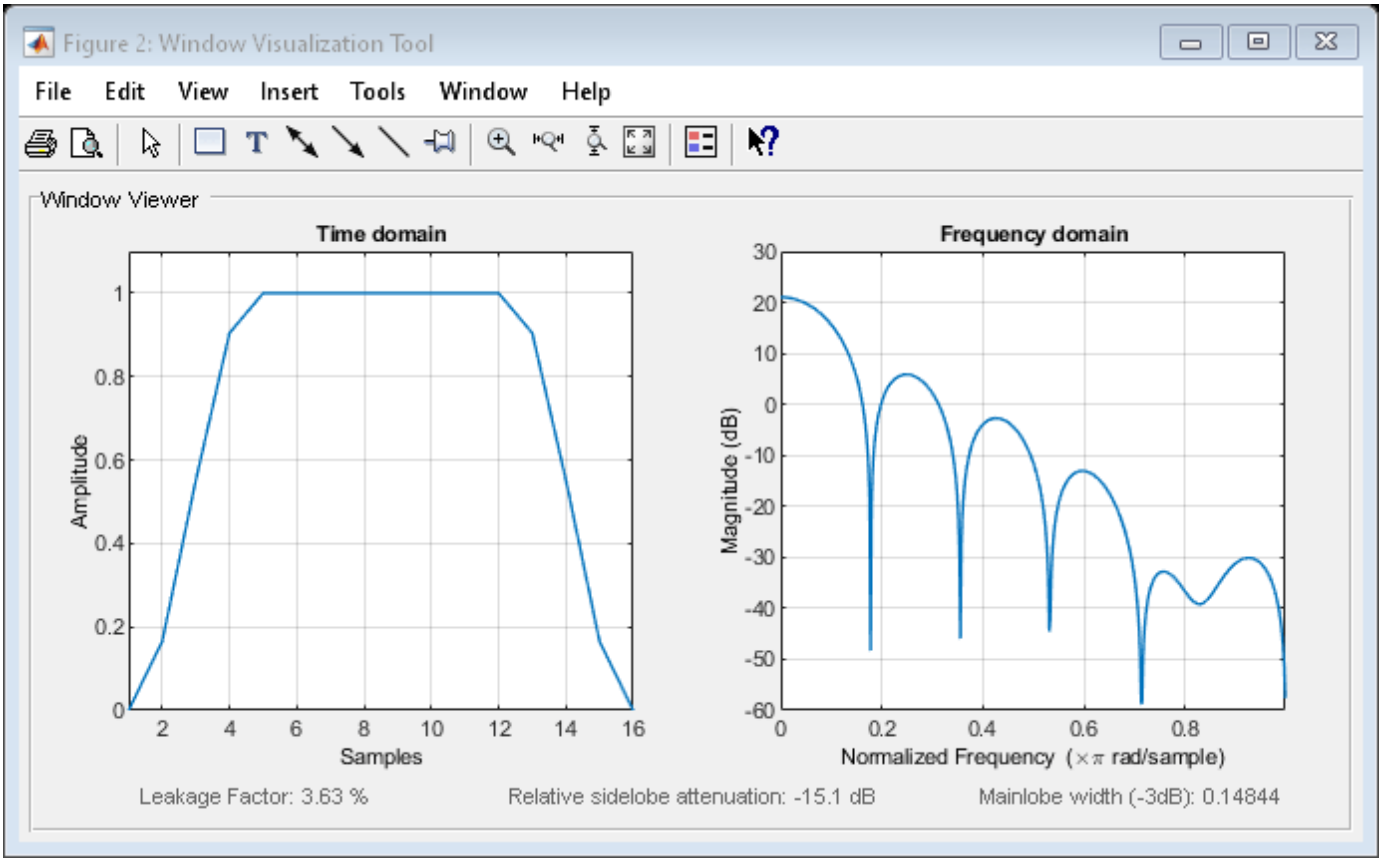
```
H = sigwin.tukeywin(16);

win = generate(H)
```

```
win = 16×1

         0
    0.1654
    0.5523
    0.9045
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x13 char array
    'Tukey Window '
    '------------ '
```

```
    'Length  : 16 '
    'Alpha   : 0.5'

wvtool(H)
```



References

[1] Bloomfield, P. *Fourier Analysis of Time Series: An Introduction*. New York: Wiley-Interscience, 2000.

See Also

`tukeywin` | `window` | **WVTool**

Topics
“Windows”
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

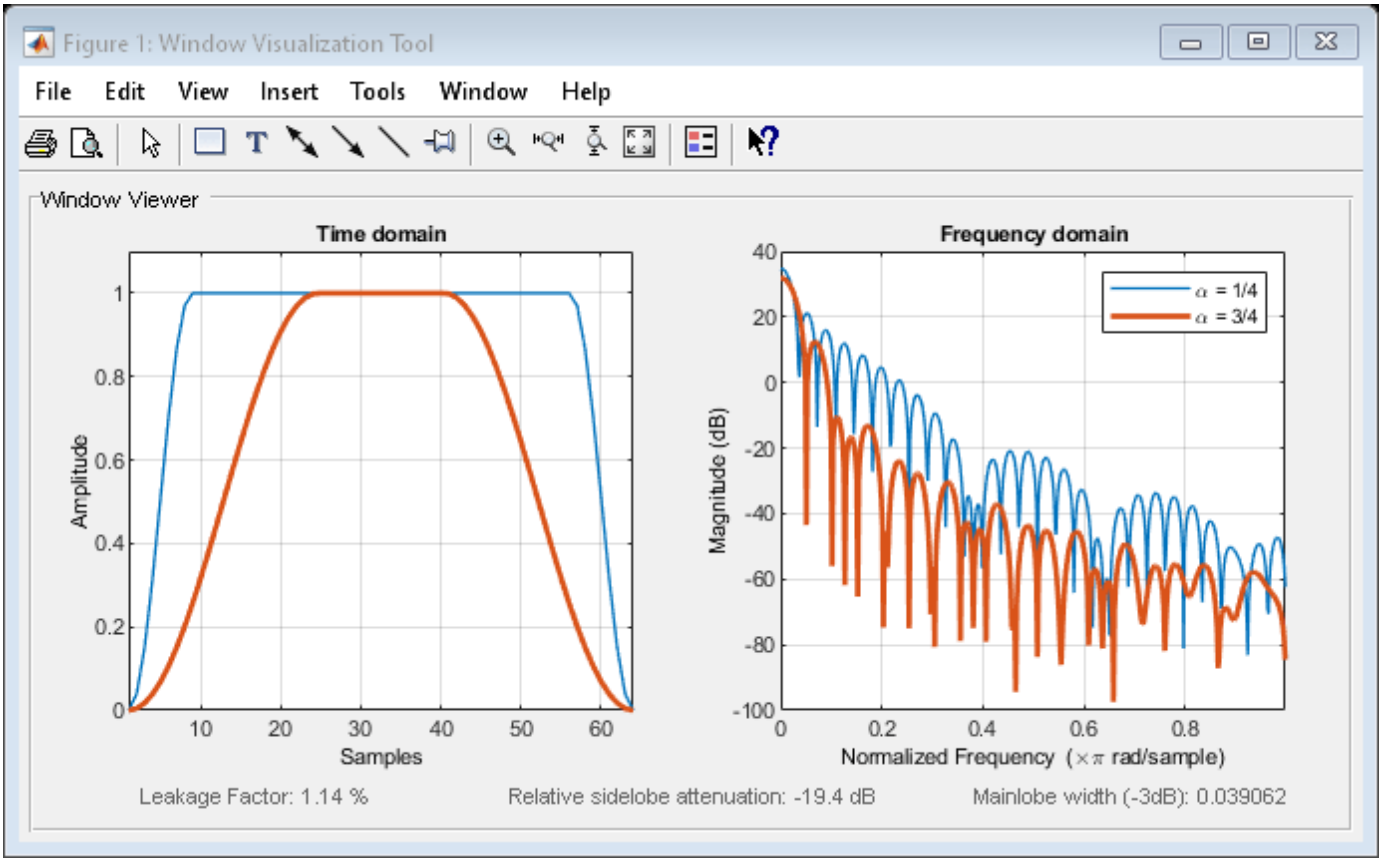
Tukey Windows

Generate two Tukey windows of length $N = 64$:

- The first window has $\alpha = 1/4$. α is the ratio of tapered window section length to constant section length.
- The second window has $\alpha = 3/4$.

Display the two windows.

```
H14 = sigwin.tukeywin(64,1/4);
H34 = sigwin.tukeywin(64,3/4);

wvt = wvtool(H14,H34);
legend(wvt.CurrentAxes,'\alpha = 1/4','\alpha = 3/4')
```



Generate a Tukey window with length $N = 16$ and the default $\alpha = 1/2$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.tukeywin(16);

win = generate(H)
```

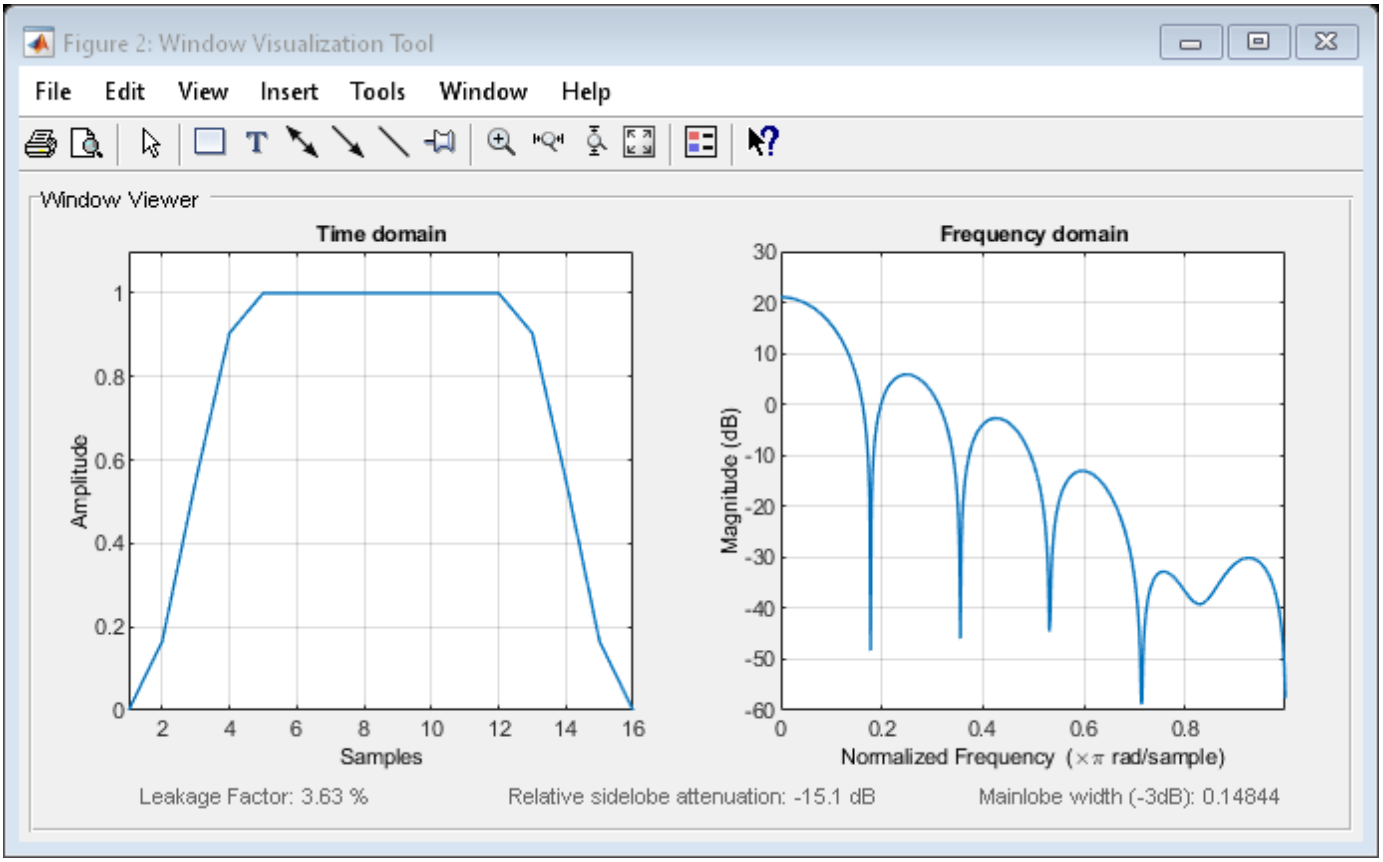
```
win = 16×1

         0
    0.1654
    0.5523
    0.9045
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x13 char array
    'Tukey Window '
    '------------ '
```

```
    'Length  : 16 '
    'Alpha   : 0.5'
```

```
wvtool(H)
```



See Also

`tukeywin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

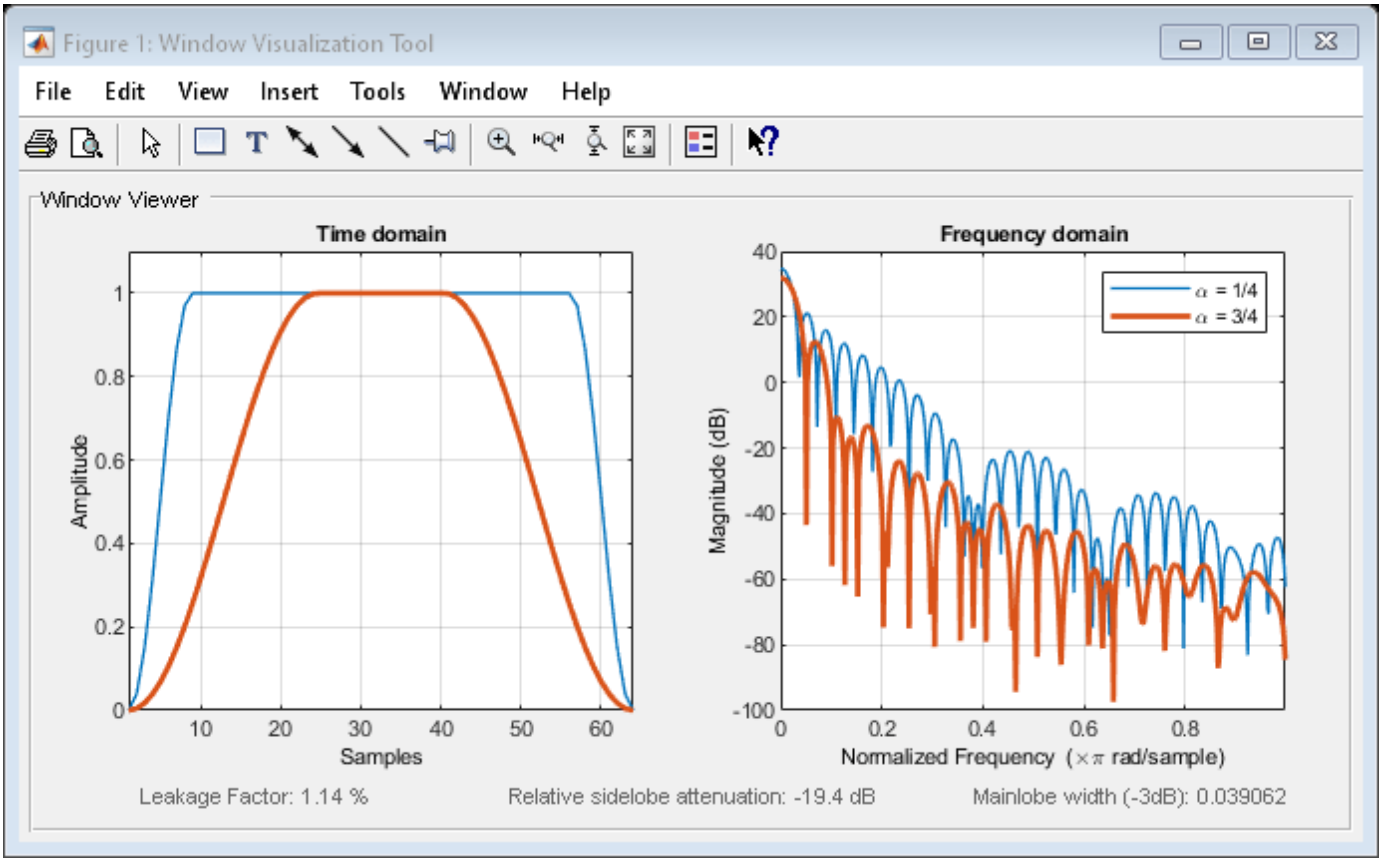
Tukey Windows

Generate two Tukey windows of length $N = 64$:

- The first window has $\alpha = 1/4$. α is the ratio of tapered window section length to constant section length.
- The second window has $\alpha = 3/4$.

Display the two windows.

```
H14 = sigwin.tukeywin(64,1/4);
H34 = sigwin.tukeywin(64,3/4);

wvt = wvtool(H14,H34);
legend(wvt.CurrentAxes,'\alpha = 1/4','\alpha = 3/4')
```



Generate a Tukey window with length $N = 16$ and the default $\alpha = 1/2$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.tukeywin(16);

win = generate(H)
```

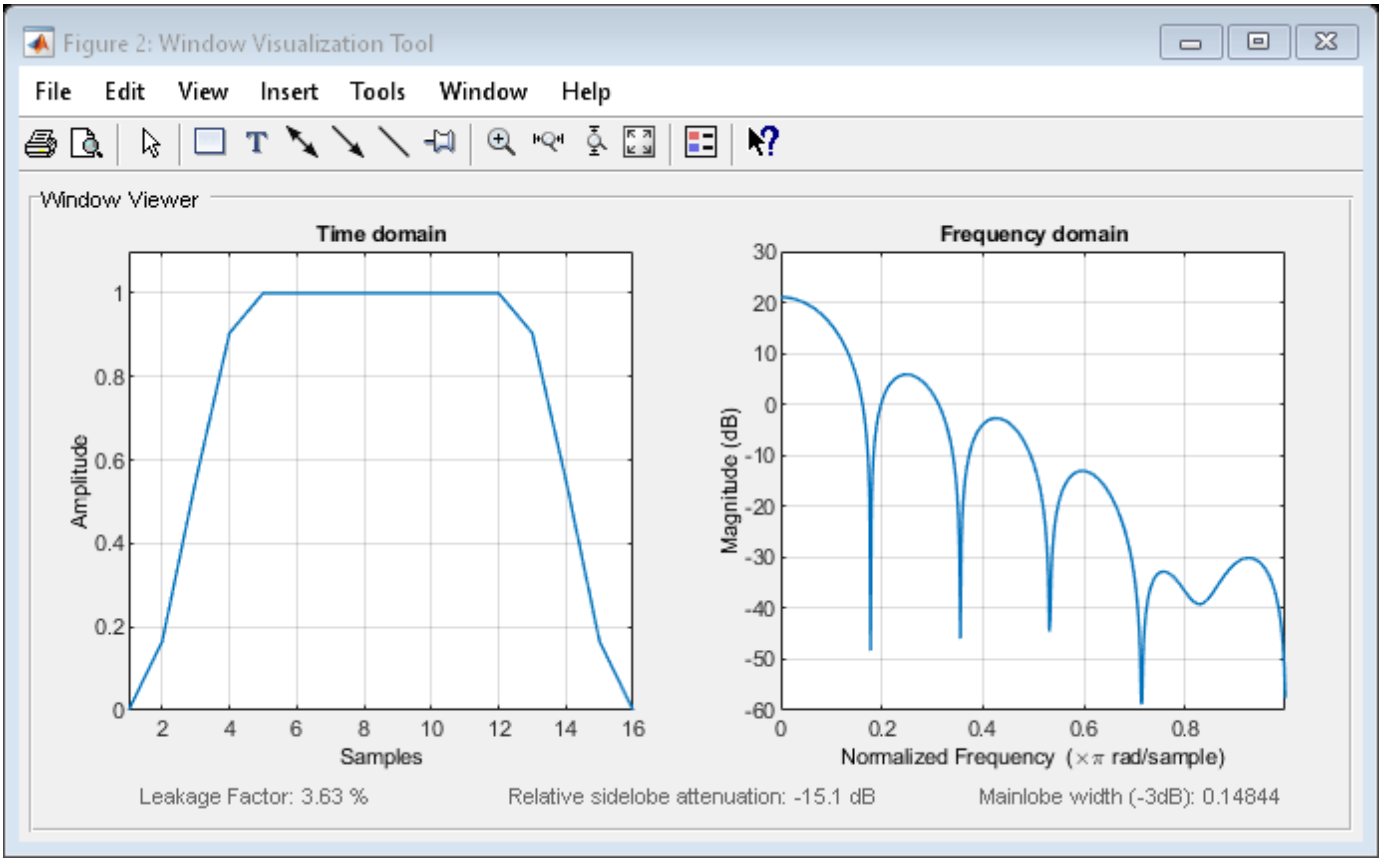
```
win = 16×1

         0
    0.1654
    0.5523
    0.9045
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x13 char array
    'Tukey Window '
    '------------ '
```

```
    'Length  : 16 '
    'Alpha   : 0.5'
```

```
wvtool(H)
```



See Also

`tukeywin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

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

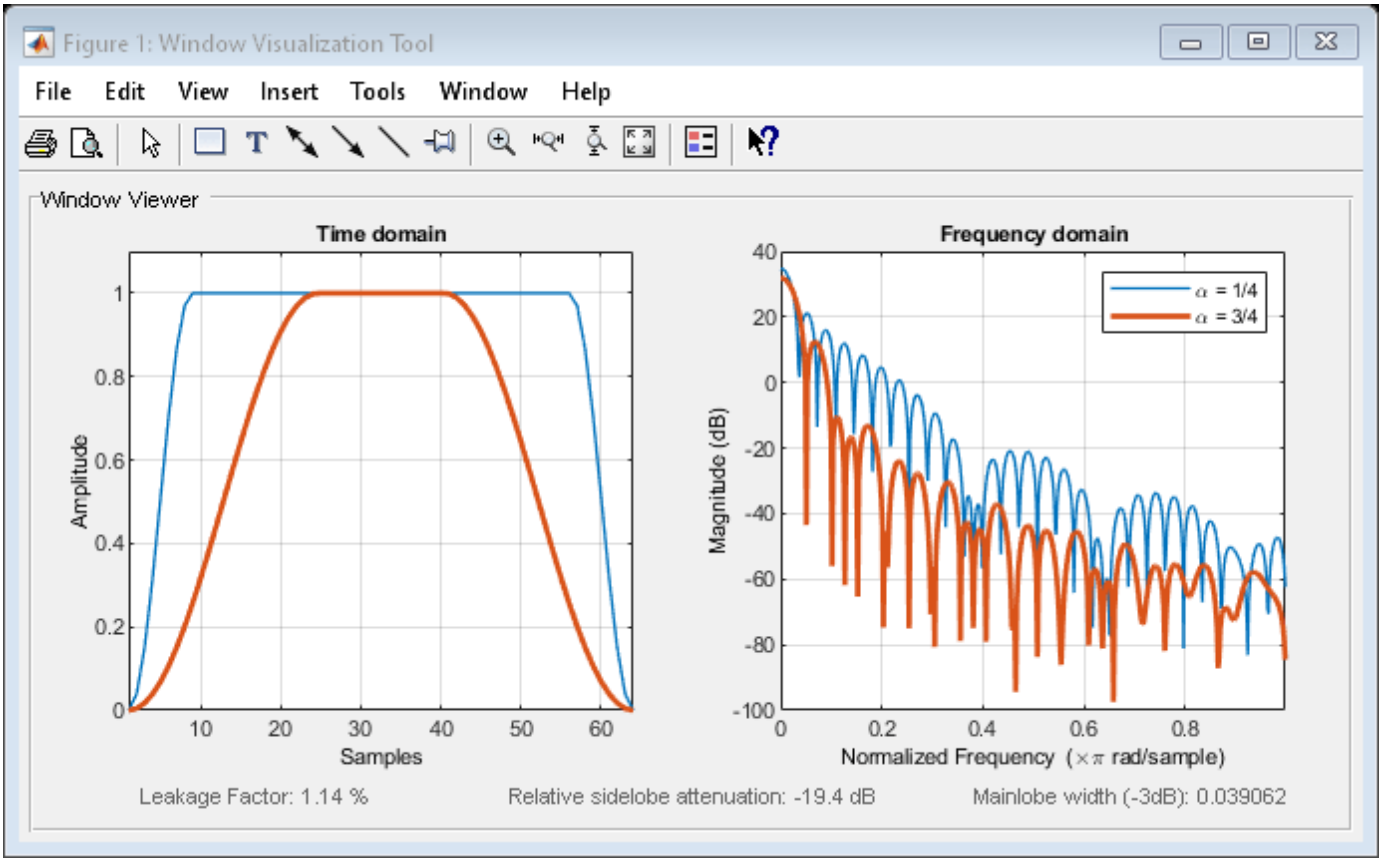
Tukey Windows

Generate two Tukey windows of length $N = 64$:

- The first window has $\alpha = 1/4$. α is the ratio of tapered window section length to constant section length.
- The second window has $\alpha = 3/4$.

Display the two windows.

```
H14 = sigwin.tukeywin(64,1/4);
H34 = sigwin.tukeywin(64,3/4);

wvt = wvtool(H14,H34);
legend(wvt.CurrentAxes,'\alpha = 1/4','\alpha = 3/4')
```



Generate a Tukey window with length $N = 16$ and the default $\alpha = 1/2$. Return its values as a column vector. Show information about the window object. Display the window.

```
H = sigwin.tukeywin(16);

win = generate(H)
```

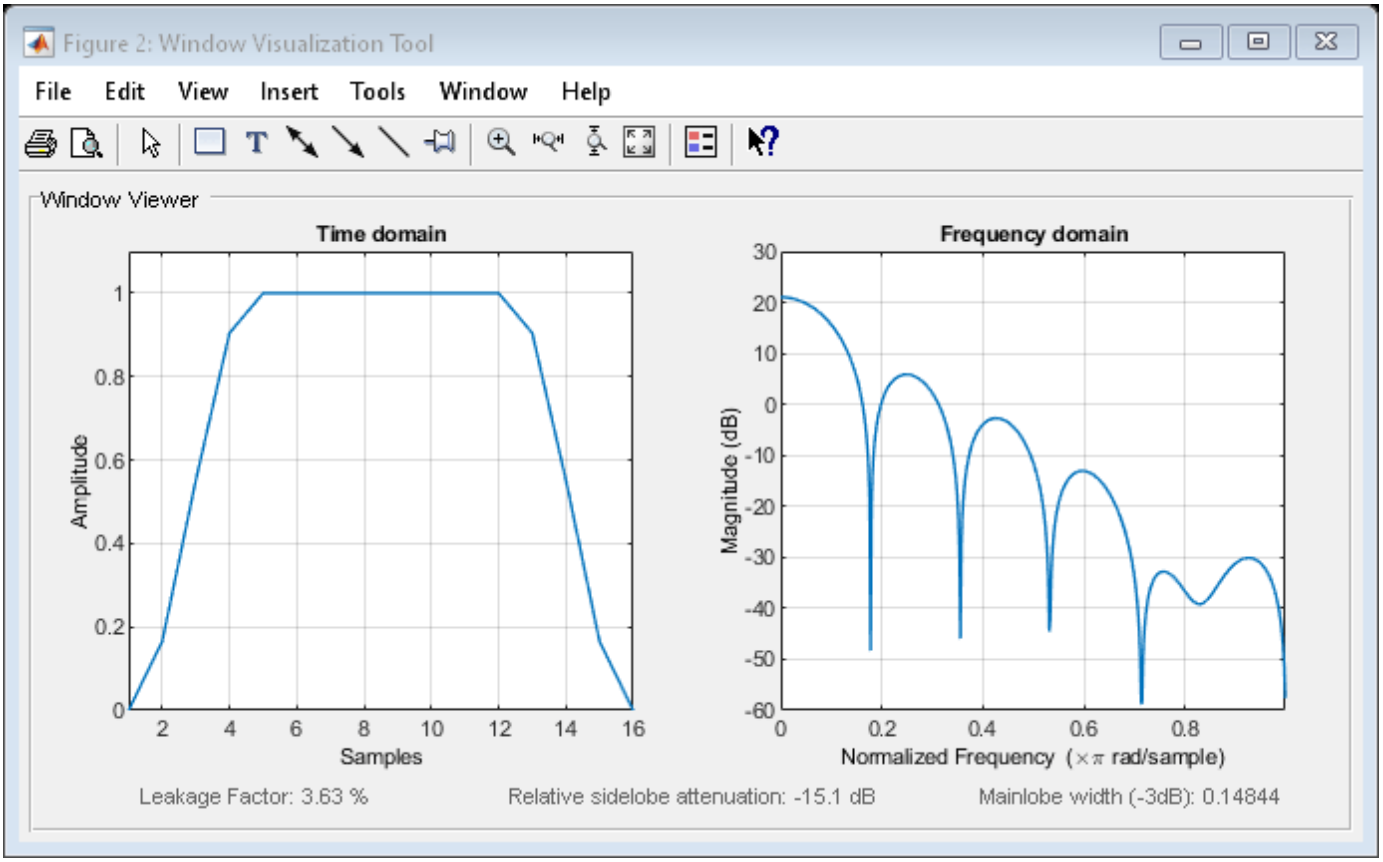
```
win = 16×1

         0
    0.1654
    0.5523
    0.9045
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
      ⋮
```

```
wininfo = info(H)
```

```
wininfo = 4x13 char array
    'Tukey Window '
    '------------ '
```

```
    'Length  : 16 '
    'Alpha   : 0.5'
```

```
wvtool(H)
```



See Also

`tukeywin` | `window` | **WVTool**

Topics

“Windows”
Class Attributes
Property Attributes

Simulation Data Inspector

Inspect and compare data and simulation results to validate and iterate model designs

Description

The Simulation Data Inspector visualizes and compares multiple kinds of data.

Using the Simulation Data Inspector, you can inspect and compare time series data at multiple stages of your workflow. This example workflow shows how the Simulation Data Inspector supports all stages of the design cycle:

1. “View Data in the Simulation Data Inspector” (Simulink).

   Run a simulation in a model configured to log data to the Simulation Data Inspector, or import data from the workspace or a MAT-file. You can view and verify model input data or inspect logged simulation data while iteratively modifying your model diagram, parameter values, or model configuration.

2. “Inspect Simulation Data” (Simulink).

   Plot signals on multiple subplots, zoom in and out on specified plot axes, and use data cursors to understand and evaluate the data. “Create Plots Using the Simulation Data Inspector” (Simulink) to tell your story.

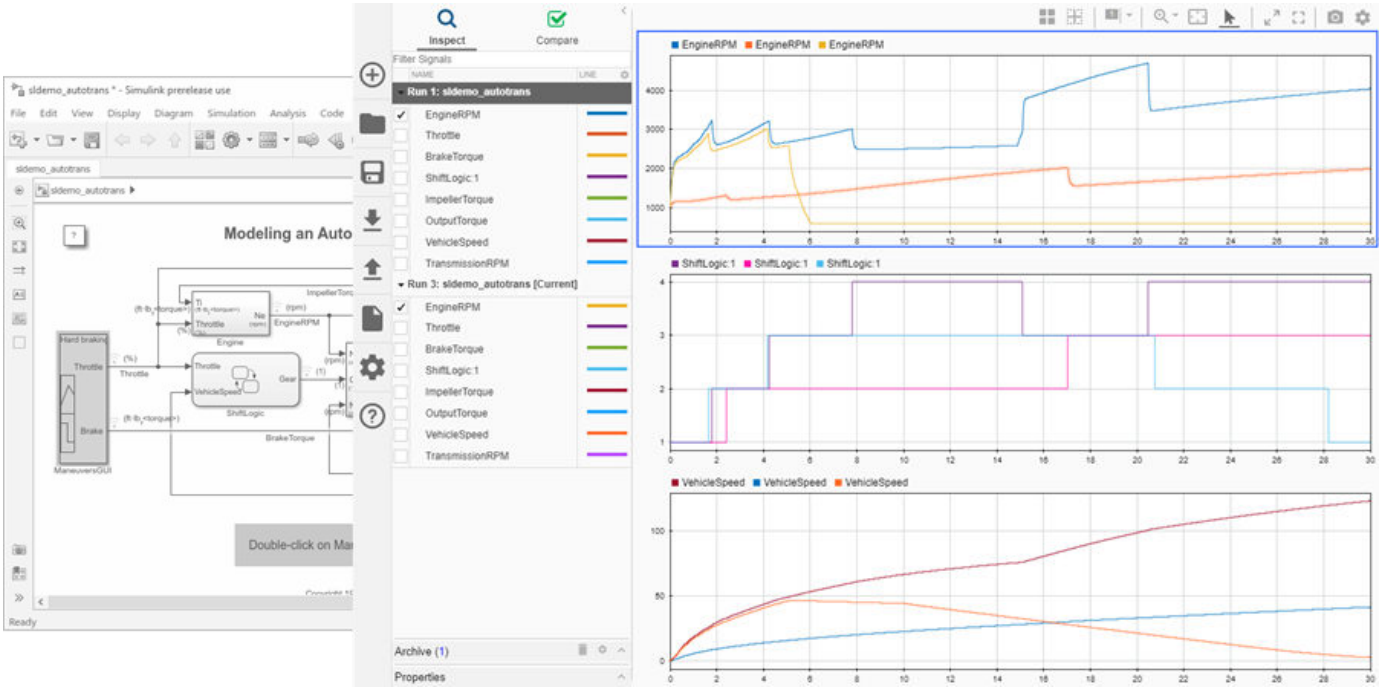
3. “Compare Simulation Data” (Simulink)

   Compare individual signals or simulation runs and analyze your comparison results with relative, absolute, and time tolerances. The compare tools in the Simulation Data Inspector facilitate iterative design and allow you to highlight signals that do not meet your tolerance requirements. For more information about the comparison operation, see “How the Simulation Data Inspector Compares Data” (Simulink).

4. “Save and Share Simulation Data Inspector Data and Views” (Simulink).

   Share your findings with others by saving Simulation Data Inspector data and views.

You can also harness the capabilities of the Simulation Data Inspector from the command line. For more information, see “Inspect and Compare Data Programmatically” (Simulink).



Open the Simulation Data Inspector

- Simulink Toolstrip: On the **Simulation** tab, under **Review Results**, click **Data Inspector**.
- Click the streaming badge on a signal to open the Simulation Data Inspector and plot the signal.
- MATLAB command prompt: Enter `Simulink.sdi.view`.

Examples

Apply a Tolerance to a Signal in Multiple Runs

You can use the Simulation Data Inspector programmatic interface to modify a parameter for the same signal in multiple runs. This example adds an absolute tolerance of `0.1` to a signal in all four runs of data.

First, clear the workspace and load the Simulation Data Inspector session with the data. The session includes logged data from four simulations of a Simulink® model of a longitudinal controller for an aircraft.

```
Simulink.sdi.clear
Simulink.sdi.load('AircraftExample.mldatx');
```

Use the `Simulink.sdi.getRunCount` function to get the number of runs in the Simulation Data Inspector. You can use this number as the index for a for loop that operates on each run.

```
count = Simulink.sdi.getRunCount;
```

Then, use a for loop to assign the absolute tolerance of `0.1` to the first signal in each run.

```
for a = 1:count
    runID = Simulink.sdi.getRunIDByIndex(a);
    aircraftRun = Simulink.sdi.getRun(runID);
    sig = getSignalByIndex(aircraftRun,1);
    sig.AbsTol = 0.1;
end
```

- “View Data in the Simulation Data Inspector” (Simulink)
- “Inspect Simulation Data” (Simulink)
- “Compare Simulation Data” (Simulink)
- “Iterate Model Design Using the Simulation Data Inspector” (Simulink)

Programmatic Use

`Simulink.sdi.view` opens the Simulation Data Inspector from the MATLAB command line.

See Also

Functions

`Simulink.sdi.clear` | `Simulink.sdi.clearPreferences` | `Simulink.sdi.snapshot`

Topics

“View Data in the Simulation Data Inspector” (Simulink)
“Inspect Simulation Data” (Simulink)
“Compare Simulation Data” (Simulink)
“Iterate Model Design Using the Simulation Data Inspector” (Simulink)

Introduced in R2010b

Simulink.sdi.compareRuns

Package: `Simulink.sdi`

Compare data in two simulation runs

Syntax

```
diffResult = Simulink.sdi.compareRuns(runID1,runID2)
diffResult = Simulink.sdi.compareRuns(runID1,runID2,Name=Value)
```

Description

`diffResult = Simulink.sdi.compareRuns(runID1,runID2)` compares the data in the runs that correspond to `runID1` and `runID2` and returns the result in the `Simulink.sdi.DiffRunResult` object `diffResult`. For more information about the comparison algorithm, see “How the Simulation Data Inspector Compares Data” (Simulink).

`diffResult = Simulink.sdi.compareRuns(runID1,runID2,Name=Value)` compares the simulation runs that correspond to `runID1` and `runID2` using the options specified by one or more name-value arguments. For more information about comparison options, see “How the Simulation Data Inspector Compares Data” (Simulink).

Examples

Compare Runs with Global Tolerance

You can specify global tolerance values to use when comparing two simulation runs. Global tolerance values are applied to all signals within the run. This example shows how to specify global tolerance values for a run comparison and how to analyze and save the comparison results.

First, load the session file that contains the data to compare. The session file contains data for four simulations of an aircraft longitudinal controller. This example compares data from two runs that use different input filter time constants.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

To access the run data to compare, use the `Simulink.sdi.getAllRunIDs` (Simulink) function to get the run IDs that correspond to the last two simulation runs.

```
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);
```

Use the `Simulink.sdi.compareRuns` (Simulink) function to compare the runs. Specify a global relative tolerance value of `0.2` and a global time tolerance value of `0.5`.

```
runResult = Simulink.sdi.compareRuns(runID1,runID2,'reltol',0.2,'timetol',0.5);
```

Check the `Summary` property of the returned `Simulink.sdi.DiffRunResult` object to see whether signals were within the tolerance values or out of tolerance.

```
runResult.Summary
```

```
ans = struct with fields:
       OutOfTolerance: 0
      WithinTolerance: 3
            Unaligned: 0
        UnitsMismatch: 0
                Empty: 0
             Canceled: 0
          EmptySynced: 0
     DataTypeMismatch: 0
         TimeMismatch: 0
    StartStopMismatch: 0
          Unsupported: 0
```

All three signal comparison results fell within the specified global tolerance.

You can save the comparison results to an MLDATX file using the `saveResult` (Simulink) function.

```
saveResult(runResult,'InputFilterComparison');
```

Analyze Simulation Data Using Signal Tolerances

You can programmatically specify signal tolerance values to use in comparisons performed using the Simulation Data Inspector. In this example, you compare data collected by simulating a model of an aircraft longitudinal flight control system. Each simulation uses a different value for the input filter time constant and logs the input and output signals. You analyze the effect of the time constant change by comparing results using the Simulation Data Inspector and signal tolerances.

First, load the session file that contains the simulation data.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

The session file contains four runs. In this example, you compare data from the first two runs in the file. Access the `Simulink.sdi.Run` objects for the first two runs loaded from the file.

```
runIDs = Simulink.sdi.getAllRunIDs;
runIDTs1 = runIDs(end-3);
runIDTs2 = runIDs(end-2);
```

Now, compare the two runs without specifying any tolerances.

```
noTolDiffResult = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
```

Use the `getResultByIndex` function to access the comparison results for the `q` and `alpha` signals.

```
qResult = getResultByIndex(noTolDiffResult,1);
alphaResult = getResultByIndex(noTolDiffResult,2);
```

Check the `Status` of each signal result to see whether the comparison result fell within our out of tolerance.

```
qResult.Status
```

```
ans =
  ComparisonSignalStatus enumeration
```

```
    OutOfTolerance
```

```
alphaResult.Status
```

```
ans =
  ComparisonSignalStatus enumeration

    OutOfTolerance
```

The comparison used a value of `0` for all tolerances, so the `OutOfTolerance` result means the signals are not identical.

You can further analyze the effect of the time constant by specifying tolerance values for the signals. Specify the tolerances by setting the properties for the `Simulink.sdi.Signal` objects that correspond to the signals being compared. Comparisons use tolerances specified for the baseline signals. This example specifies a time tolerance and an absolute tolerance.

To specify a tolerance, first access the `Signal` objects from the baseline run.

```
runTs1 = Simulink.sdi.getRun(runIDTs1);
qSig = getSignalsByName(runTs1,'q, rad/sec');
alphaSig = getSignalsByName(runTs1,'alpha, rad');
```

Specify an absolute tolerance of `0.1` and a time tolerance of `0.6` for the `q` signal using the `AbsTol` and `TimeTol` properties.

```
qSig.AbsTol = 0.1;
qSig.TimeTol = 0.6;
```

Specify an absolute tolerance of `0.2` and a time tolerance of `0.8` for the `alpha` signal.

```
alphaSig.AbsTol = 0.2;
alphaSig.TimeTol = 0.8;
```

Compare the results again. Access the results from the comparison and check the `Status` property for each signal.

```
tolDiffResult = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
qResult2 = getResultByIndex(tolDiffResult,1);
alphaResult2 = getResultByIndex(tolDiffResult,2);

qResult2.Status
```

```
ans =
  ComparisonSignalStatus enumeration

    WithinTolerance
```

```
alphaResult2.Status
```

```
ans =
  ComparisonSignalStatus enumeration

    WithinTolerance
```

Configure Comparisons to Check Metadata

You can use the `Simulink.sdi.compareRuns` function to compare signal data and metadata, including data type and start and stop times. A single comparison may check for mismatches in one or more pieces of metadata. When you check for mismatches in signal metadata, the `Summary` property of the `Simulink.sdi.DiffRunResult` object may differ from a basic comparison because the `Status` property for a `Simulink.sdi.DiffSignalResult` object can indicate the metadata mismatch. You can configure comparisons using the `Simulink.sdi.compareRuns` function for imported data and for data logged from a simulation.

This example configures a comparison of runs created from workspace data three ways to show how the `Summary` of the `DiffSignalResult` object can provide specific information about signal mismatches.

Create Workspace Data

The `Simulink.sdi.compareRuns` function compares time series data. Create data for a sine wave to use as the baseline signal, using the `timeseries` format. Give the `timeseries` the name `Wave Data`.

```
time = 0:0.1:20;
sig1vals = sin(2*pi/5*time);
sig1_ts = timeseries(sig1vals,time);
sig1_ts.Name = 'Wave Data';
```

Create a second sine wave to compare against the baseline signal. Use a slightly different time vector and attenuate the signal so the two signals are not identical. Cast the signal data to the `single` data type. Also name this `timeseries` object `Wave Data`. The Simulation Data Inspector comparison algorithm will align these signals for comparison using the name.

```
time2 = 0:0.1:22;
sig2vals = single(0.98*sin(2*pi/5*time2));
sig2_ts = timeseries(sig2vals,time2);
sig2_ts.Name = 'Wave Data';
```

Create and Compare Runs in the Simulation Data Inspector

The `Simulink.sdi.compareRuns` function compares data contained in `Simulink.sdi.Run` objects. Use the `Simulink.sdi.createRun` function to create runs in the Simulation Data Inspector for the data. The `Simulink.sdi.createRun` function returns the run ID for each created run.

```
runID1 = Simulink.sdi.createRun('Baseline Run','vars',sig1_ts);
runID2 = Simulink.sdi.createRun('Compare to Run','vars',sig2_ts);
```

You can use the `Simulink.sdi.compareRuns` function to compare the runs. The comparison algorithm converts the signal data to the `double` data type and synchronizes the signal data before computing the difference signal.

```
basic_DRR = Simulink.sdi.compareRuns(runID1,runID2);
```

Check the `Summary` property of the returned `Simulink.sdi.DiffRunResult` object to see the result of the comparison.

```
basic_DRR.Summary
```

```
ans = struct with fields:
       OutOfTolerance: 1
      WithinTolerance: 0
            Unaligned: 0
        UnitsMismatch: 0
                Empty: 0
             Canceled: 0
          EmptySynced: 0
     DataTypeMismatch: 0
         TimeMismatch: 0
    StartStopMismatch: 0
          Unsupported: 0
```

The difference between the signals is out of tolerance.

Compare Runs and Check for Data Type Match

Depending on your system requirements, you may want the data types for signals you compare to match. You can use the `Simulink.sdi.compareRuns` function to configure the comparison algorithm to check for and report data type mismatches.

```
dataType_DRR = Simulink.sdi.compareRuns(runID1,runID2,'DataType','MustMatch');
dataType_DRR.Summary
```

```
ans = struct with fields:
       OutOfTolerance: 0
      WithinTolerance: 0
            Unaligned: 0
        UnitsMismatch: 0
                Empty: 0
             Canceled: 0
          EmptySynced: 0
     DataTypeMismatch: 1
         TimeMismatch: 0
    StartStopMismatch: 0
          Unsupported: 0
```

The result of the signal comparison is now `DataTypeMismatch` because the data for the baseline signal is `double` data type, while the data for the signal compared to the baseline is `single` data type.

Compare Runs and Check for Start and Stop Time Match

You can use the `Simulink.sdi.compareRuns` function to configure the comparison algorithm to check whether the aligned signals have the same start and stop times.

```
startStop_DRR = Simulink.sdi.compareRuns(runID1,runID2,'StartStop','MustMatch');
startStop_DRR.Summary
```

```
ans = struct with fields:
       OutOfTolerance: 0
      WithinTolerance: 0
            Unaligned: 0
        UnitsMismatch: 0
```

```
                Empty: 0
             Canceled: 0
          EmptySynced: 0
       DataTypeMismatch: 0
           TimeMismatch: 0
      StartStopMismatch: 1
            Unsupported: 0
```

The signal comparison result is now `StartStopMismatch` because the signals created in the workspace have different stop times.

Compare Runs with Alignment Criteria

When you compare runs using the Simulation Data Inspector, you can specify alignment criteria that determine how signals are paired with each other for comparison. This example compares data from simulations of a model of an aircraft longitudinal control system. The simulations used a square wave input. The first simulation used an input filter time constant of `0.1s` and the second simulation used an input filter time constant of `0.5s`.

First, load the simulation data from the session file that contains the data for this example.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

The session file contains data for four simulations. This example compares data from the first two runs. Access the run IDs for the first two runs loaded from the session file.

```
runIDs = Simulink.sdi.getAllRunIDs;
runIDTs1 = runIDs(end-3);
runIDTs2 = runIDs(end-2);
```

Before running the comparison, define how you want the Simulation Data Inspector to align the signals between the runs. This example aligns signals by their name, then by their block path, and then by their Simulink identifier.

```
alignMethods = [Simulink.sdi.AlignType.SignalName
                Simulink.sdi.AlignType.BlockPath
                Simulink.sdi.AlignType.SID];
```

Compare the simulation data in your two runs, using the alignment criteria you specified. The comparison uses a small time tolerance to account for the effect of differences in the step size used by the solver on the transition of the square wave input.

```
diffResults = Simulink.sdi.compareRuns(runIDTs1,runIDTs2,'align',alignMethods,...
    'timetol',0.005);
```

You can use the `getResultByIndex` function to access the comparison results for the aligned signals in the runs you compared. You can use the `Count` property of the `Simulink.sdi.DiffRunResult` object to set up a `for` loop to check the `Status` property for each `Simulink.sdi.DiffSignalResult` object.

```
numComparisons = diffResults.count;

for k = 1:numComparisons
    resultAtIdx = getResultByIndex(diffResults,k);
```

```
    sigID1 = resultAtIdx.signalID1;
    sigID2 = resultAtIdx.signalID2;

    sig1 = Simulink.sdi.getSignal(sigID1);
    sig2 = Simulink.sdi.getSignal(sigID2);

    displayStr = 'Signals %s and %s: %s \n';
    fprintf(displayStr,sig1.Name,sig2.Name,resultAtIdx.Status);
end
```

```
Signals q, rad/sec and q, rad/sec: OutOfTolerance
Signals alpha, rad and alpha, rad: OutOfTolerance
Signals Stick and Stick: WithinTolerance
```

Input Arguments

`runID1` — Baseline run identifier

integer

Numeric identifier for the baseline run in the comparison, specified as a run ID that corresponds to a run in the Simulation Data Inspector. The Simulation Data Inspector assigns run IDs when runs are created. You can get the run ID for a run by using the `ID` property of the `Simulink.sdi.Run` object, the `Simulink.sdi.getAllRunIDs` function, or the `Simulink.sdi.getRunIDByIndex` function.

`runID2` — Identifier for run to compare

integer

Numeric identifier for the run to compare, specified as a run ID that corresponds to a run in the Simulation Data Inspector. The Simulation Data Inspector assigns run IDs when runs are created. You can get the run ID for a run by using the `ID` property of the `Simulink.sdi.Run` object, the `Simulink.sdi.getAllRunIDs` function, or the `Simulink.sdi.getRunIDByIndex` function.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `AbsTol=x,Align=alignOpts`

`Align` — Signal alignment options

`Simulink.sdi.AlignType` scalar | `Simulink.sdi.AlignType` vector

Signal alignment options, specified as a `Simulink.sdi.AlignType` scalar or vector. The `Simulink.sdi.AlignType` enumeration includes a value for each option available for pairing each signal in the baseline run with a signal in the comparison run. You can specify one or more alignment options for the comparison. To use more than one alignment option, specify an array. When you specify multiple alignment options, the Simulation Data Inspector aligns signals first by the option in the first element of the array, then by the option in the second element array, and so on. For more information, see “Signal Alignment” (Simulink).

Value	Aligns By
`Simulink.sdi.AlignType.BlockPath`	Path to the source block for the signal
`Simulink.sdi.AlignType.SID`	Simulink identifier For more information, see “Simulink Identifiers” (Simulink).
`Simulink.sdi.AlignType.SignalName`	Signal name
`Simulink.sdi.AlignType.DataSource`	Path of the variable in the MATLAB workspace

Example: `[Simulink.sdi.AlignType.SignalName,Simulink.sdi.AlignType.BlockPath]` specifies signal alignment by signal name and then by block path.

`AbsTol` — Global absolute tolerance for comparison
`0` (default) | positive-valued scalar

Global absolute tolerance for comparison, specified as a positive-valued scalar.

Global tolerances apply to all signals in the run comparison. To use a different tolerance value for a signal in the comparison, specify the tolerance you want to use on the `Simulink.sdi.Signal` object in the baseline run and set the `OverrideGlobalTol` property for that signal to `true`.

For more information about how tolerances are used in comparisons, see “Tolerance Specification” (Simulink).

Example: `0.5`

Data Types: `double`

`RelTol` — Global relative tolerance for comparison
`0` (default) | positive-valued scalar

Global relative tolerance for comparison, specified as a positive-valued scalar. The relative tolerance is expressed as a fractional multiplier. For example, `0.1` specifies a 10 percent tolerance.

Global tolerances apply to all signals in the run comparison. To use a different tolerance value for a signal in the comparison, specify the tolerance you want to use on the `Simulink.sdi.Signal` object in the baseline run and set the `OverrideGlobalTol` property for that signal to `true`.

For more information about how tolerances are used in comparisons, see “Tolerance Specification” (Simulink).

Example: `0.1`

Data Types: `double`

`TimeTol` — Global time tolerance for comparison
`0` (default) | positive-valued scalar

Global time tolerance for comparison, specified as a positive-valued scalar, using units of seconds.

Global tolerances apply to all signals in the run comparison. To use a different tolerance value for a signal in the comparison, specify the tolerance you want to use on the `Simulink.sdi.Signal` object in the baseline run and set the `OverrideGlobalTol` property for that signal to `true`.

For more information about tolerances in the Simulation Data Inspector, see “Tolerance Specification” (Simulink).

Example: `0.2`

Data Types: `double`

`DataType` — Comparison sensitivity to signal data types
`"MustMatch"`

Comparison sensitivity to signal data types, specified as `"MustMatch"`. Specify `DataType="MustMatch"` when you want the comparison to be sensitive to data type mismatches in compared signals. When you specify this name-value argument, the algorithm compares the data types for aligned signals before synchronizing and comparing the signal data.

When signal data types do not match, the `Status` property of the `Simulink.sdi.DiffSignalResult` object for the result is set to `DataTypeMismatch`.

The `Simulink.sdi.compareRuns` function does not compare the data types of aligned signals unless you specify this name-value argument. When you do not specify this name-value argument, the comparison does compute results for signals with different data types.

When you specify that data types must match and configure the comparison to stop on the first mismatch, a data type mismatch stops the comparison. A stopped comparison may not compute results for all signals.

`Time` — Comparison sensitivity to signal time vectors
`"MustMatch"`

Comparison sensitivity to signal time vectors, specified as `"MustMatch"`. Specify `Time="MustMatch"` when you want the comparison to be sensitive to mismatches in the time vectors of compared signals. When you specify this name-value argument, the algorithm compares the time vectors of aligned signals before synchronizing and comparing the signal data.

When the time vectors for signals do not match, the `Status` property of the `Simulink.sdi.DiffSignalResult` object for the result is set to `TimeMismatch`.

Comparisons are not sensitive to differences in signal time vectors unless you specify this name-value argument. For comparisons that are not sensitive to differences in the time vectors, the comparison algorithm synchronizes the signals prior to the comparison. For more information about how synchronization works, see “How the Simulation Data Inspector Compares Data” (Simulink).

When you specify that time vectors must match and configure the comparison to stop on the first mismatch, a time vector mismatch stops the comparison. A stopped comparison may not compute results for all signals.

`StartStop` — Comparison sensitivity to signal start and stop times
`"MustMatch"`

Comparison sensitivity to signal start and stop times, specified as `"MustMatch"`. Specify `StartStop="MustMatch"` when you want the comparison to be sensitive to mismatches in signal start and stop times. When you specify this name-value argument, the algorithm compares the start and stop times for aligned signals before synchronizing and comparing the signal data.

When the start times and stop times do not match, the `Status` property of the `Simulink.sdi.DiffSignalResult` object for the result is set to `StartStopMismatch`.

When you specify that start and stop times must match and configure the comparison to stop on the first mismatch, a start or stop time mismatch stops the comparison. A stopped comparison may not compute results for all signals.

`StopOnFirstMismatch` — Whether comparison stops on first detected mismatch
`"Metadata"` | `"Any"`

Whether comparison stops on first detected mismatch without comparing remaining signals, specified as `"Metadata"` or `"Any"`. A stopped comparison may not compute results for all signals, and can return a mismatched result more quickly.

- `Metadata` — A mismatch in metadata for aligned signals causes the comparison to stop. Metadata comparisons happen before comparing signal data.

  The Simulation Data Inspector always aligns signals and compares signal units. When you configure the comparison to stop on the first mismatch, an unaligned signal or mismatched units always causes the comparison to stop. You can specify additional name-value arguments to configure the comparison to check and stop on the first mismatch for additional metadata, such as signal data type, start and stop times, and time vectors.
- `Any` — A mismatch in metadata or signal data for aligned signals causes the comparison to stop.

`ExpandChannels` — Whether to compute comparison results for each channel in multidimensional signals
`true` or `1` (default) | `false` or `0`

Whether to compute comparison results for each channel in multidimensional signals, specified as logical `true` (`1`) or `false` (`0`).

- `true` or `1` — Comparison expands multidimensional signals represented as a single signal with nonscalar sample values to a set of signals with scalar sample values and computes a comparison result for each of these signals.

  The representation of the multidimensional signal in the Simulation Data Inspector as a single signal with nonscalar sample values does not change.
- `false` or `0` — Comparison does not compute results for multidimensional signals represented as a single signal with nonscalar sample values.

Output Arguments

`diffResult` — Comparison results
`Simulink.sdi.DiffRunResult` object

Comparison results, returned as a `Simulink.sdi.DiffRunResult` object.

Limitations

The Simulation Data Inspector does not support comparing:

- Signals of data types `int64` or `uint64`.
- Variable-size signals.

See Also

Functions
`Simulink.sdi.compareSignals` | `Simulink.sdi.getRunIDByIndex` | `Simulink.sdi.getRunCount` | `getResultByIndex`

Objects
`Simulink.sdi.DiffRunResult` | `Simulink.sdi.DiffSignalResult`

Topics
“Inspect and Compare Data Programmatically” (Simulink)
“Compare Simulation Data” (Simulink)
“How the Simulation Data Inspector Compares Data” (Simulink)

Introduced in R2011b

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 sample rate `fs` of the input signal `x`. If you do not specify `fs`, then the sample rate 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 (in dB) 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)
```

```
r = 32.0412
```

```
powfund = 1;
powharm = 0.025^2;
thSINAD = 10*log10(powfund/powharm)
```

```
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)
```

```
r = 22.8085
```

```
varnoise = 0.05^2;
thSINAD = 10*log10(powfund/(powharm+varnoise))
```

```
thSINAD = 25.0515
```

SINAD for Signal with Sample Rate

Create a signal with a fundamental frequency of 1 kHz and unit amplitude, 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)
```

```
r = 32.2058
```

```
powfund = 1;
powharm = 0.02^2;
varnoise = 0.01^2;
thSINAD = 10*log10(powfund/(powharm+varnoise*(1/fs)))
```

```
thSINAD = 33.9794
```

SINAD from Periodogram

Create a signal with a fundamental frequency of 1 kHz and unit amplitude, 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`.

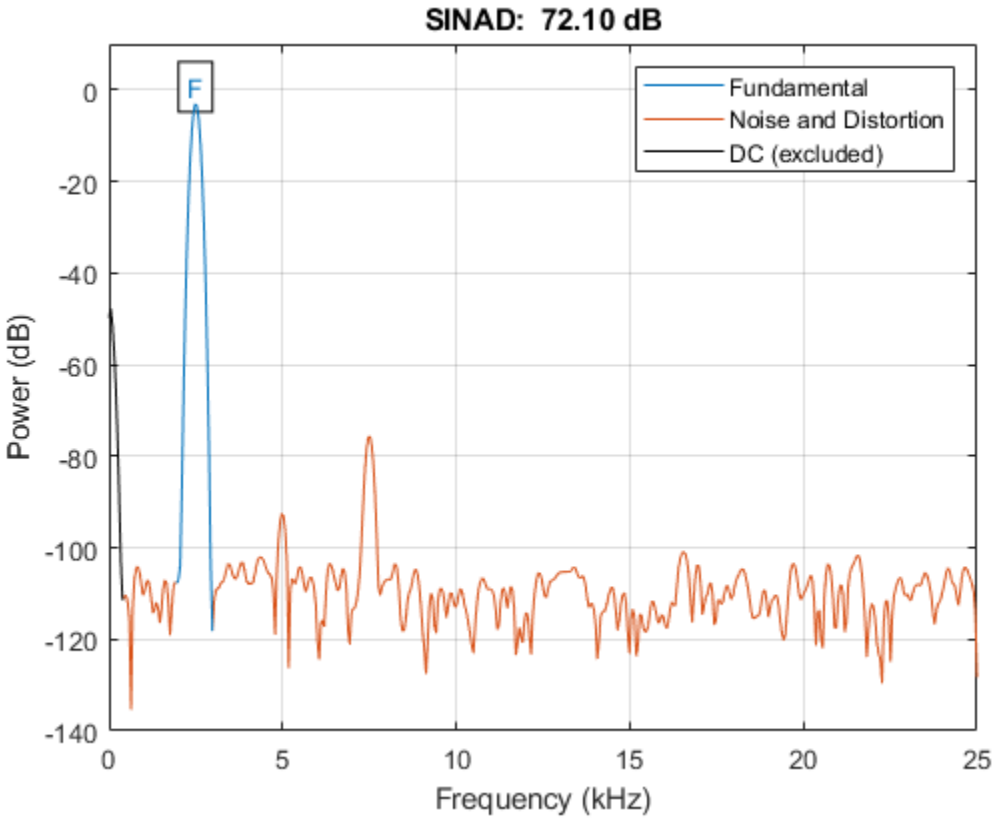
```
fs = 48e4;
t = 0:1/fs:1-1/fs;

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. Add Gaussian white noise with standard deviation 0.00005 to the signal. Pass the result through a weakly nonlinear amplifier. Plot the SINAD.

```
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.

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 — 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 has units of Hz.

pxx — One-sided PSD estimate

vector

One-sided PSD estimate, specified as a real-valued, nonnegative column vector.

The power spectral density must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[pxx,f] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` specifies the periodogram PSD estimate of a noisy two-channel sinusoid sampled at 2π Hz and the frequencies at which it is computed.

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.

The power spectrum must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[sxx,w] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2),'power')` specifies the periodogram power spectrum estimate of a two-channel sinusoid embedded in white Gaussian noise and the normalized frequencies at which it is computed.

`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, returned as a real-valued scalar.

`totdistpow` — Total noise and harmonic distortion power of the signal
real-valued scalar

Total noise and harmonic distortion power of the signal, returned as a real-valued scalar expressed in dB.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If supplied, input arguments `'power'` and `'psd'` must be compile-time constants.

See Also

`sfdr` | `snr` | `thd` | `toi`

Topics

“Analyzing Harmonic Distortion”

Introduced in R2013b

sinc

Sinc function

Syntax

```
y = sinc(x)
```

Description

`y = sinc(x)` returns an array, `y`, whose elements are the “sinc” on page 1-2395 of the elements of the input, `x`. The output `y` is the same size as `x`.

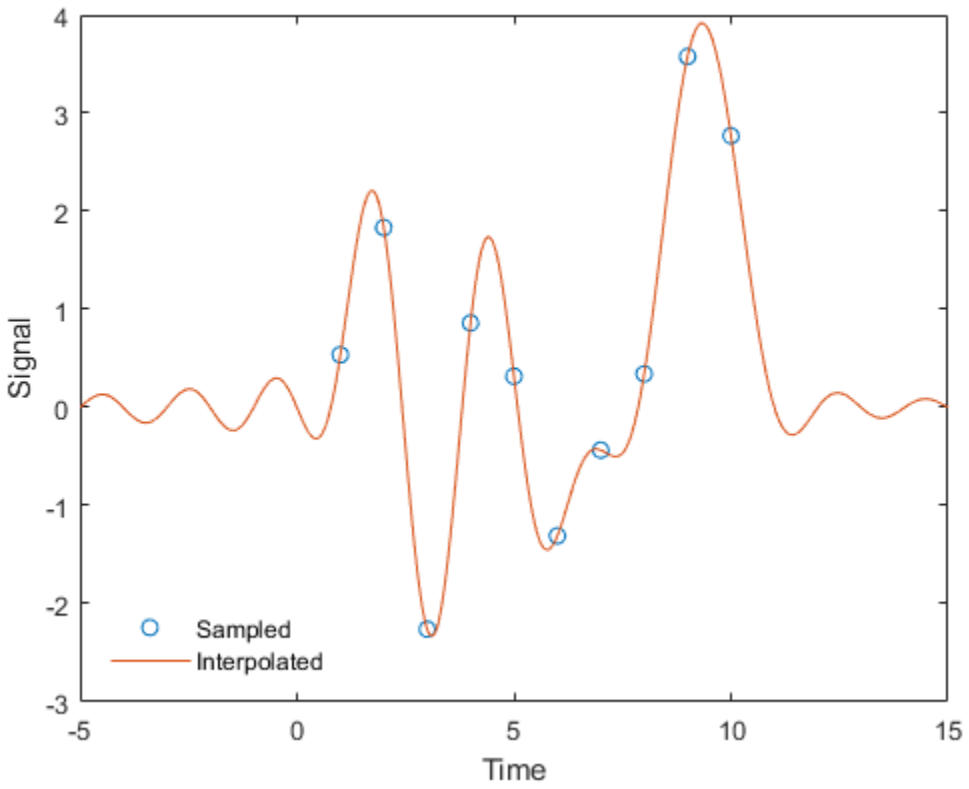
Examples

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
```



Input Arguments

x — Input array

scalar value | vector | matrix | *N*-D array | `gpuArray` object

Input array, specified as a real-valued or complex-valued scalar, vector, matrix, *N*-D array, or `gpuArray` object. When `x` is nonscalar, `sinc` is an element-wise operation.

See “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox) and “GPU Support by Release” (Parallel Computing Toolbox) for details on `gpuArray` objects.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

y — Sinc of input

scalar value | vector | matrix | *N*-D array | `gpuArray` object

Sinc of the input array, `x`, returned as a real-valued or complex-valued scalar, vector, matrix, *N*-D array, or `gpuArray` object of the same size as `x`.

More About

sinc

The sinc function is defined by

$$\mathrm{sinc}t = \begin{cases} \dfrac{\sin \pi t}{\pi t} & t \neq 0, \\ 1 & t = 0. \end{cases}$$

This analytic expression corresponds to the continuous inverse Fourier transform of a rectangular pulse of width 2π and height 1:

$$\mathrm{sinc}t = \frac{1}{2\pi} \int_{-\pi}^{\pi} e^{j\omega t}\, d\omega \,.$$

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:

$$g(t) = \sum_{n=-\infty}^{\infty} g(n)\mathrm{sinc}(t-n) \,.$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Thread-Based Environment

Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

This function fully supports thread-based environments. For more information, see “Run MATLAB Functions in Thread-Based Environment”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`chirp` | `cos` | `diric` | `gauspuls` | `pulstran` | `rectpuls` | `sawtooth` | `sin` | `square` | `tripuls`

Introduced before R2006a

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 0.2π rad/sample and a normalized stopband frequency of 0.55π 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)

classd =
'double'

classe = class(e.Coefficients)

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 0.5π 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`

Introduced in R2014a

slewrate

Slew rate of bilevel waveform

Syntax

```
s = slewrate(x)
s = slewrate(x,Fs)
s = slewrate(x,t)
[s,lt,ut] = slewrate( ___ )
[s,lt,ut,ll,ul] = slewrate( ___ )
s = slewrate( ___ ,Name,Value)
slewrate( ___ )
```

Description

`s = slewrate(x)` returns the slew rate for all transitions found in the bilevel waveform `x`. To determine the transitions, the `slewrate` function estimates the state levels of the input waveform by a histogram method and identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state.

`s = slewrate(x,Fs)` specifies the sample rate `Fs`.

`s = slewrate(x,t)` specifies the sample instants `t`.

`[s,lt,ut] = slewrate( ___ )` 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%, respectively. You can specify an input combination from any of the previous syntaxes.

`[s,lt,ut,ll,ul] = slewrate( ___ )` returns the waveform values that correspond to the lower-reference levels `ll` and upper-reference levels `ul`.

`s = slewrate( ___ ,Name,Value)` returns the slew rate for all transitions with additional options specified by one or more `Name,Value` arguments.

`slewrate( ___ )` 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 also indicates the state levels and associated lower and upper tolerances.

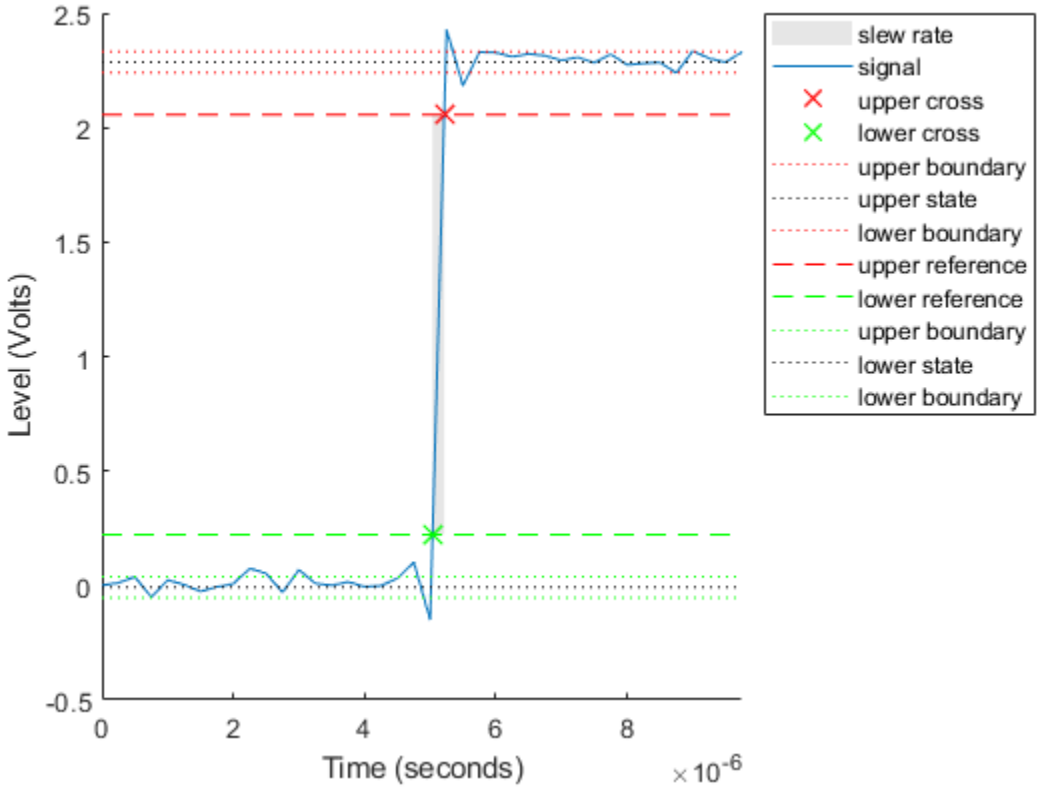
Examples

Slew Rate For One-Transition Waveform

Use `slewrate` 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. Annotate the slew rate in a plot of the waveform.

```
load('transitionex.mat','x','t')

slewrate(x,t)
```




```
ans = 1.0310e+07
```

Slew Rates for Three-Transition Waveform

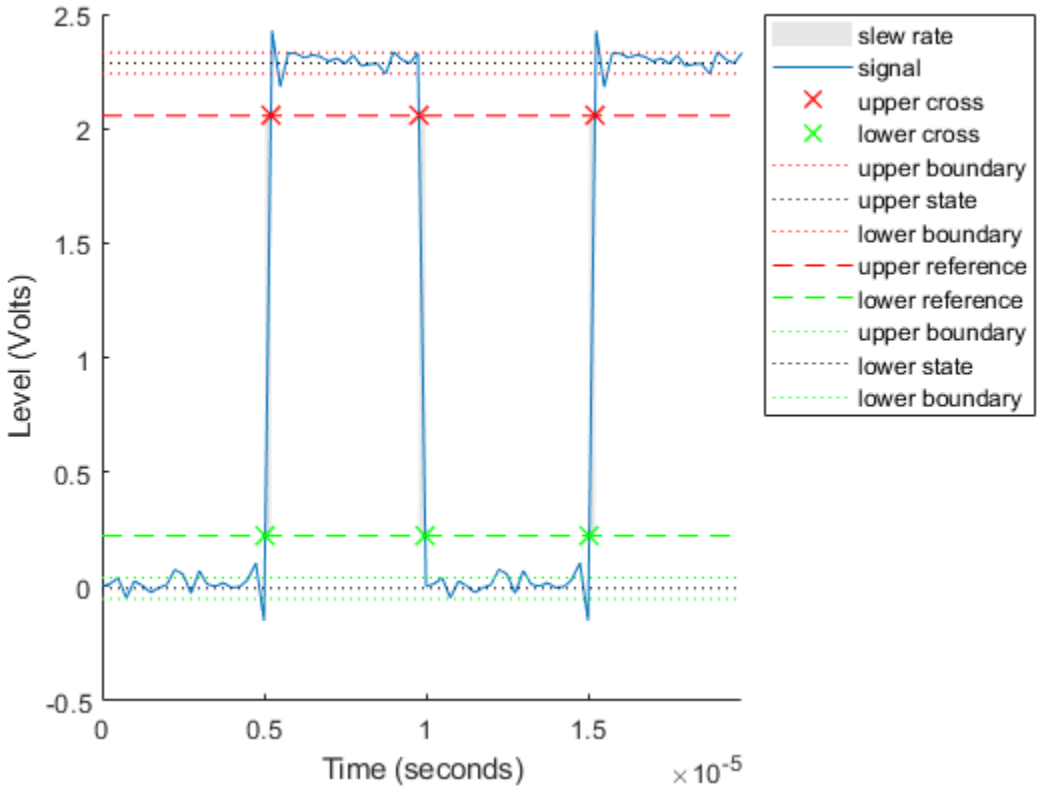
Create a bilevel waveform with three transitions, two positive and one negative. The sample rate is 4 MHz. Obtain the slew rates for the three transitions.

```
load('transitionex.mat','x')
fs = 4e6;

y = [x;fliplr(x)];
t = (0:length(y)-1)/4e6;

S = slewrate(y,t)
```

```
S = 3×1
10^7 ×

    1.0310
   -0.9320
    1.0310
```

Annotate the result on a plot of the waveform.

```
slewrate(y,t);
```



Lower and Upper Transition Times

Return the lower- and upper-transition times for a three-transition waveform sampled at 4 MHz.

```
load('transitionex.mat','x')
fs = 4e6;

y = [x;fliplr(x)];
t = (0:length(y)-1)/fs;

[~,LT,UT] = slewrate(y,t)
```

```
LT = 3×1
10-4 ×

    0.0504
    0.0998
    0.1504


UT = 3×1
10-4 ×
```

```
    0.0521
    0.0978
    0.1521
```

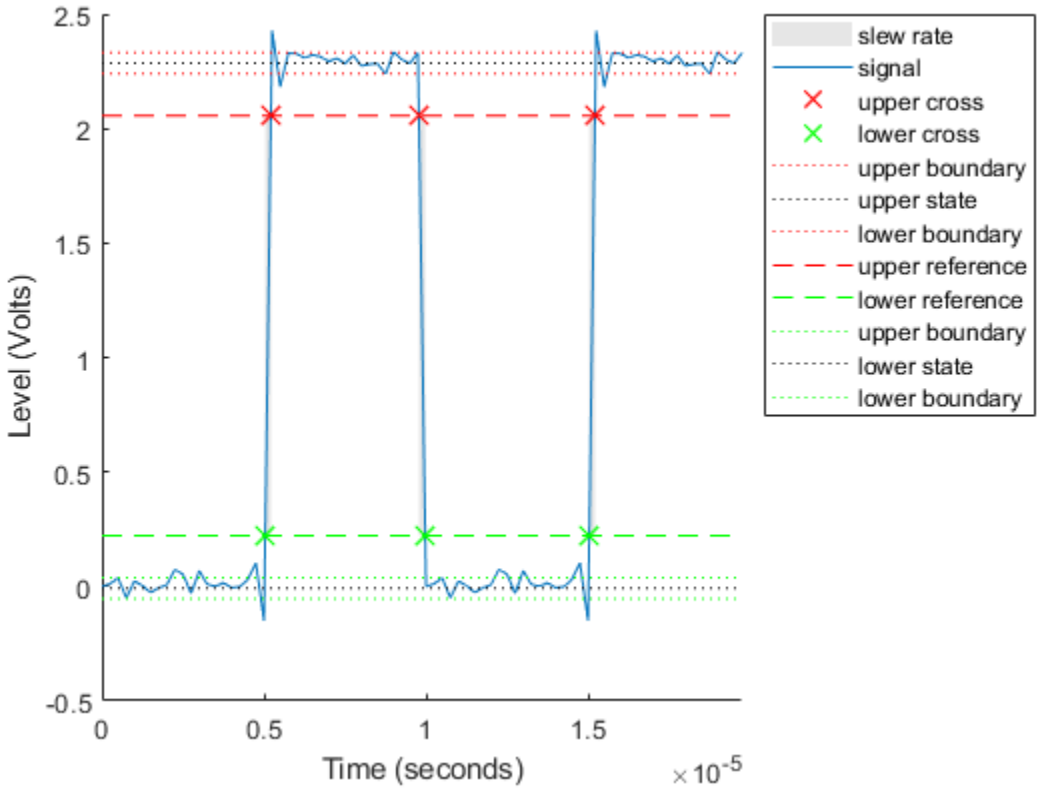
Repeat using the sample rate instead of the time vector.

```
[~,LT,UT] = slewrate(y,fs)
```

```
LT = 3×1
10^-4 ×

    0.0504
    0.0998
    0.1504


UT = 3×1
10^-4 ×

    0.0521
    0.0978
    0.1521
```

Annotate the result on a plot of the waveform.

```
slewrate(y,fs);
```



Lower and Upper Reference Levels

Return the waveform values corresponding to the lower- and upper-reference levels for a three-transition waveform sampled at 4 MHz. Compute these values for 10% and 90%, the default levels.

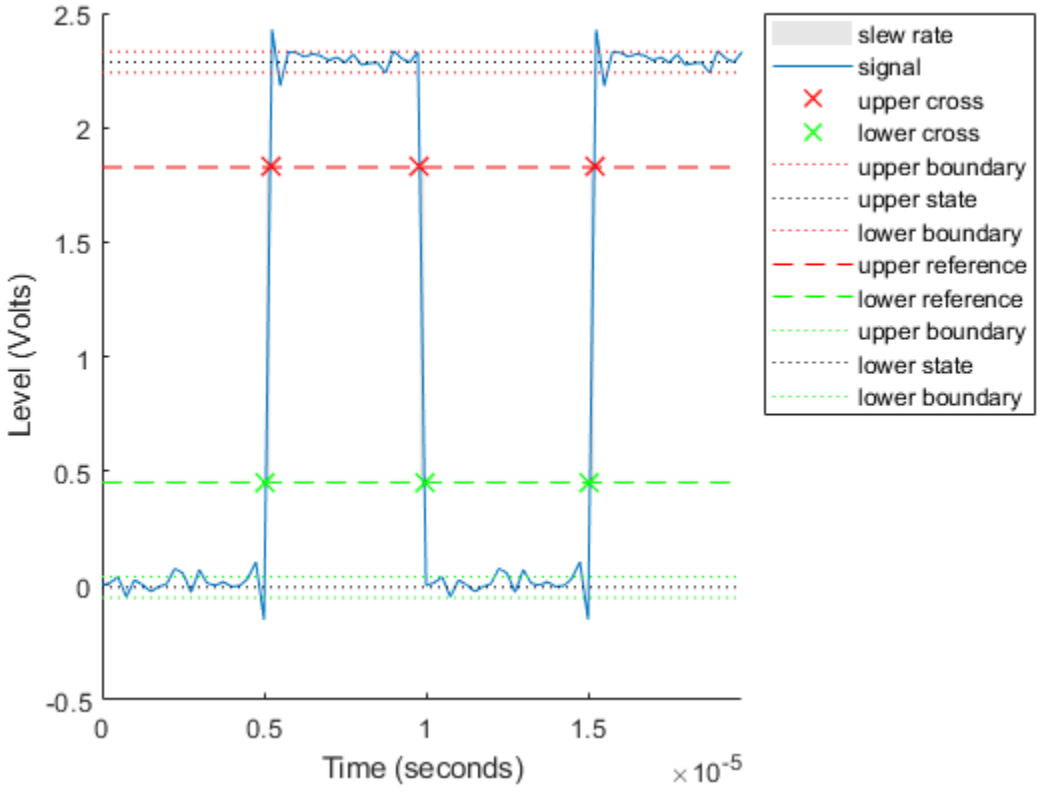
```
load("transitionex.mat","x")
fs = 4e6;

y = [x;fliplr(x)];
t = (0:length(y)-1)/fs;

[~,~,~,LL,UL] = slewrate(y,t)
```

```
LL = 0.2212
```

```
UL = 2.0564
```

Repeat the calculation for 20% and 80%. Annotate the result on a plot of the waveform

```
slewrate(y,t,PercentReferenceLevels=[20 80]);
```



Input Arguments

x — Bilevel waveform

real-valued vector

Bilevel waveform, specified as a real-valued vector. If the input waveform does not have at least one transition, the function returns an empty matrix. The first time instant in `x` corresponds to t=0.

Fs — Sample rate

positive real scalar

Sample rate, specified as a positive real scalar in hertz.

t — Sample instants

vector

Sample instants, specified as a vector. The length of `t` must equal the length of the input bilevel waveform `x`. The sample instants correspond to the indices of the input vector.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `s = slewrate(x,t,Tolerance=5)`

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `s = slewrate(x,t,'Tolerance',5)`

PercentReferenceLevels — Percent reference levels

`[10,90]` (default) | 1-by-2 real-valued vector

Percent reference levels, specified as a 1-by-2 real-valued vector. For more information, see “Percent Reference Levels” on page 1-2404.

StateLevels — Low- and high-state levels

1-by-2 real-valued vector

Low- and high-state levels, specified as a 1-by-2 real-valued vector. The first element is the low-state level and the second element is the high-state level. If you do not specify `StateLevels`, the function estimates the state levels from the input waveform using the histogram method.

Tolerance — Tolerance levels

2 (default) | scalar

Tolerance levels (lower and upper state boundaries), specified as a scalar and expressed as a percentage. 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. For more information, see “State-Level Tolerances” on page 1-2405.

Output Arguments

`s` — Slew rate

vector

Slew rate, returned as a vector. 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. The slew rate is the slope of the line connecting the 10% and 90% reference levels.

`lt` — Time instants when input signal crosses lower percent reference level

vector

Time instants when the input signal crosses the lower percent reference level, returned as a vector.

`ut` — Time instants when input signal crosses upper percent reference level

vector

Time instants when the input signal crosses the upper-percent reference level, returned as a vector.

`ll` — Waveform values at lower reference level

vector

Waveform values at the lower-percent reference level, returned as a vector.

`ul` — Waveform values at upper reference level

vector

Waveform values at the upper-percent reference level, returned as a vector.

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, then 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 1-2404, the slew rate is

$$\frac{S_1 + \frac{U}{100}(S_2 - S_1) - \left\{S_1 + \frac{L}{100}(S_2 - S_1)\right\}}{t_U - t_L} = \frac{U - L}{100}\frac{S_2 - S_1}{t_U - t_L}.$$

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.

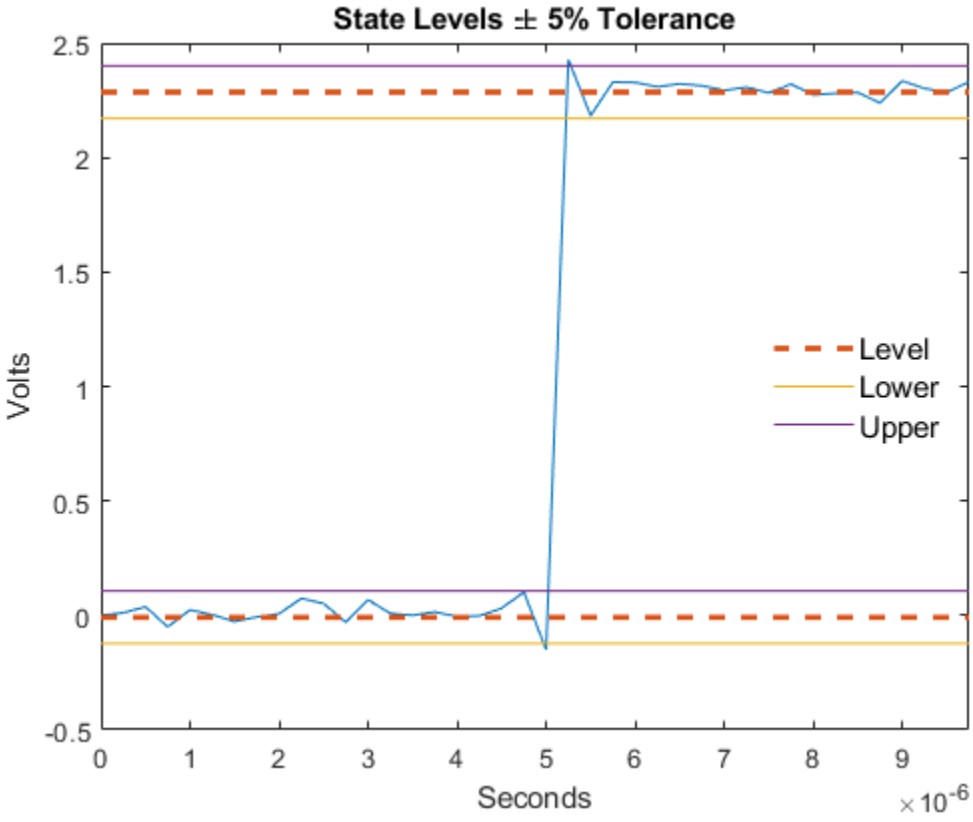
State-Level Tolerances

You can specify lower- and upper-state boundaries for each state level. Define the boundaries as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, specify the scalar as a small number such as 2/100 or 3/100. In general, the $\alpha\%$ 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.

This figure shows lower and upper 5% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The thick dashed lines indicate the estimated state levels.



References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`falltime` | `midcross` | `pulsewidth` | `risetime` | `settlingtime` | `statelevels`

Introduced in R2012a

snr

Signal-to-noise ratio

Syntax

```
r = snr(xi,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 = snr(___,'aliased')

[r,noisepow] = snr(___)

snr(___)
```

Description

`r = snr(xi,y)` returns the signal-to-noise ratio (SNR) in decibels of a signal, `xi`, by computing the ratio of its summed squared magnitude to that of the noise `y`:
`r = mag2db(rssq(xi(:))/rssq(y(:)))`.
`y` must have the same dimensions as `xi`. 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 = snr( ___ ,'aliased')` removes harmonics of the fundamental that are aliased into the Nyquist range. Use this option when the sinusoidal input signal is undersampled. If you do not specify this option, or if you set it to `'omitaliases'`, then the function treats as noise any harmonics of the fundamental frequency that lie beyond the Nyquist range.

`[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

Generate a 20-millisecond rectangular pulse sampled for 2 seconds at 10 kHz.

```
Tpulse = 20e-3;
Fs = 10e3;
t = -1:1/Fs:1;
x = rectpuls(t,Tpulse);
```

Embed the pulse in white Gaussian noise such that the signal-to-noise ratio (SNR) is 53 dB. Reset the random number generator for reproducible results.

```
rng default

SNR = 53;
y = randn(size(x))*std(x)/db2mag(SNR);

s = x + y;
```

Use the `snr` function to compute the SNR of the noisy signal.

```
pulseSNR = snr(x,y)
```

```
pulseSNR = 53.1255
```

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.

```
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]
```

```
SN = 1×2

   17.0178   16.9897
```

```
THD = thd(x);
defTHD = 10*log10(powharm/powfund);
TH = [THD defTHD]
```

```
TH = 1×2

   -7.9546   -7.9588
```

```
SINAD = sinad(x);
defSINAD = 10*log10(powfund/(powharm+varnoise));
SI = [SINAD defSINAD]
```

```
SI = 1×2

    7.4571    7.4473
```

Signal-to-Noise Ratio of Sinusoid

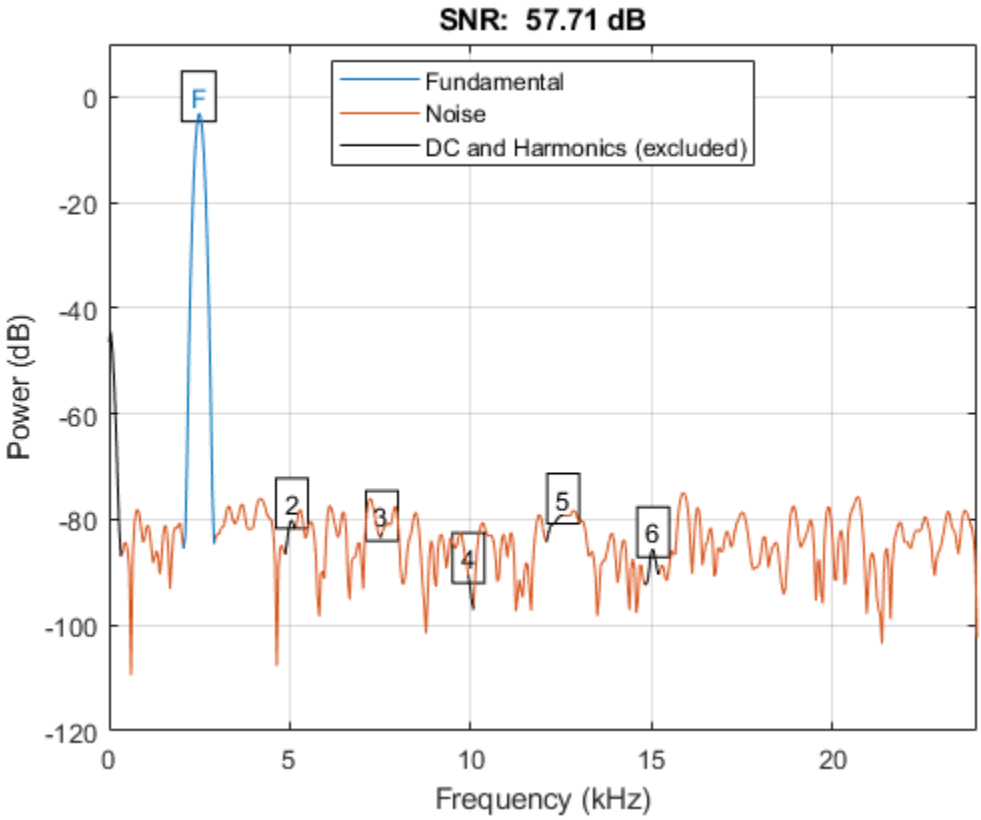
Compute the SNR of a 2.5 kHz sinusoid sampled at 48 kHz. Add white noise with variance 0.001^2.

```
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
```

Plot the spectrum and annotate the SNR.

```
snr(x,Fs);
```



SNR of Sinusoid Using 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 Sinusoid Using Power Spectrum

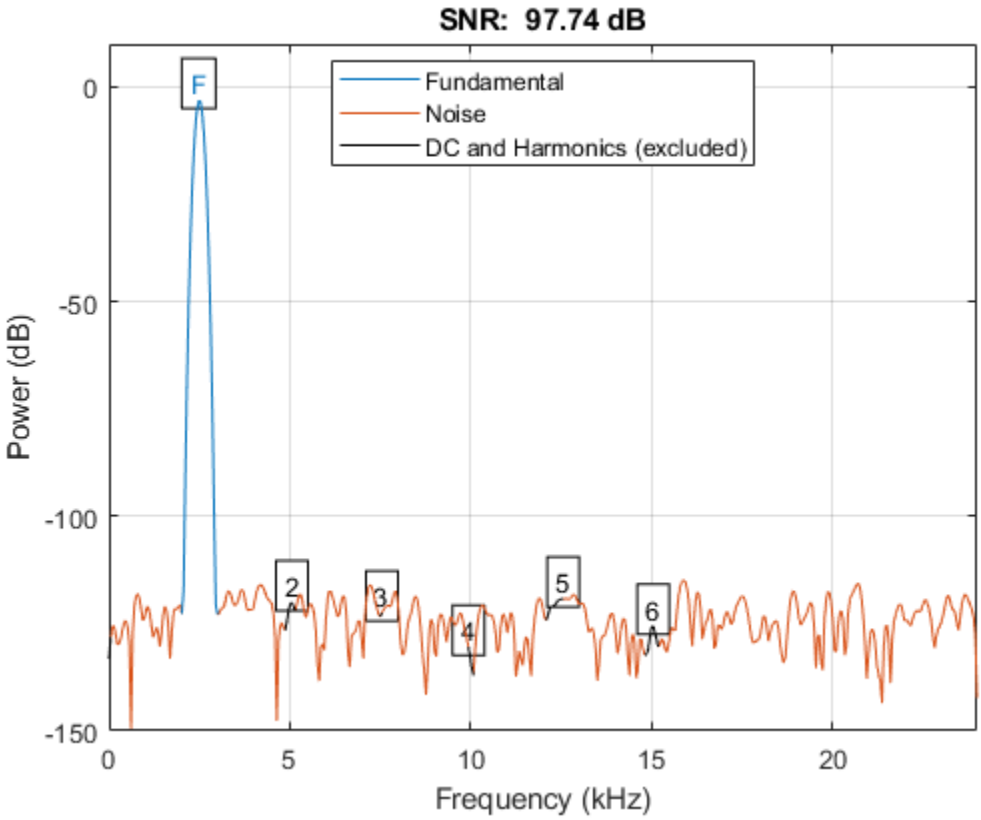
Using the power spectrum, compute the SNR of a 2.5 kHz sinusoid sampled at 48 kHz and embedded in white noise with a standard deviation of 0.00001. Reset the random number generator for reproducible results.

```
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
```

Plot the spectrum of the signal and annotate the SNR.

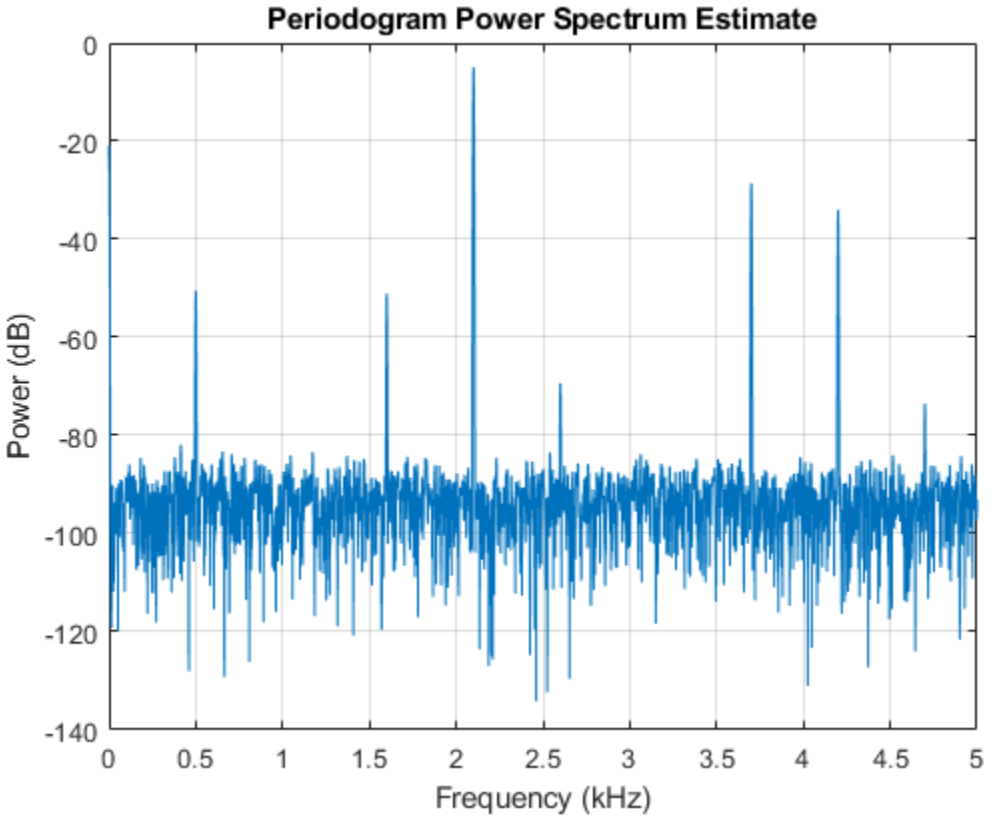
```
snr(Sxx,F,rbw,'power');
```



SNR with and Without Aliased Harmonics

Generate a signal that resembles the output of a weakly nonlinear amplifier with a 2.1 kHz tone as input. The signal is sampled for 1 second at 10 kHz. Compute and plot the power spectrum of the signal. Use a Kaiser window with $\beta = 38$ for the computation.

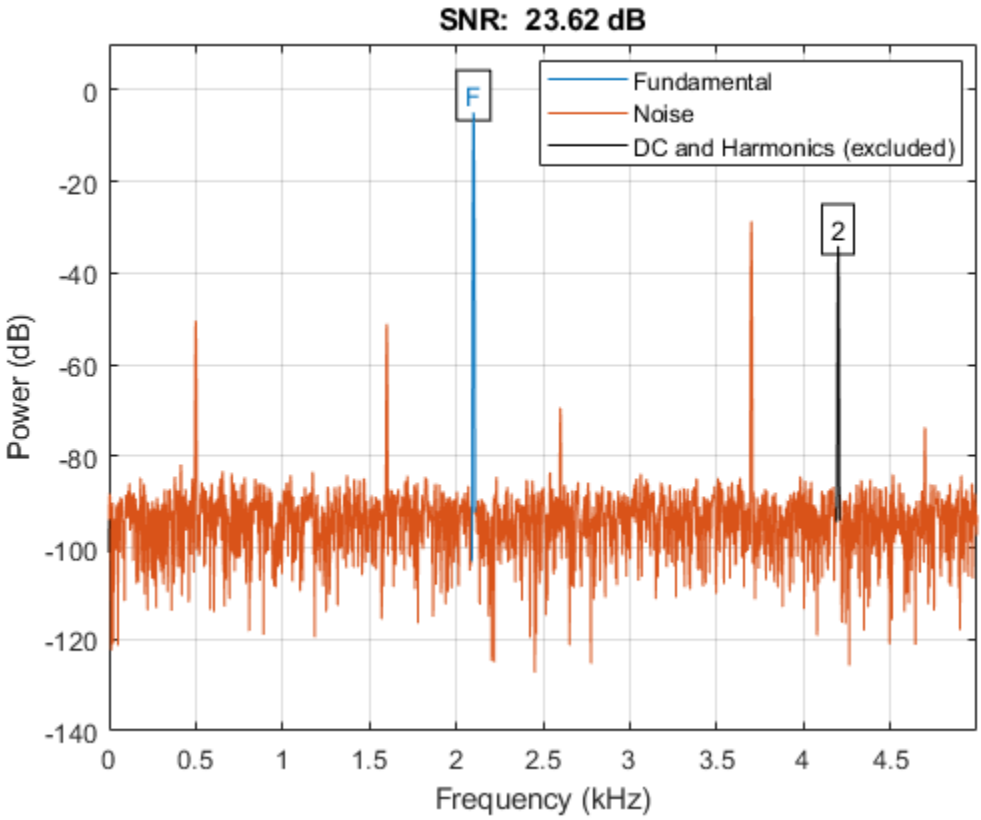
```
Fs = 10000;
f = 2100;

t = 0:1/Fs:1;
x = tanh(sin(2*pi*f*t)+0.1) + 0.001*randn(1,length(t));

periodogram(x,kaiser(length(x),38),[],Fs,'power')
```



Harmonics stick out from the noise at frequencies of 4.2 kHz, 6.3 kHz, 8.4 kHz, 10.5 kHz, 12.6 kHz, and 14.7 kHz. All frequencies except for the first one are greater than the Nyquist frequency. The harmonics are aliased respectively into 3.7 kHz, 1.6 kHz, 0.5 kHz, 2.6 kHz, and 4.7 kHz.

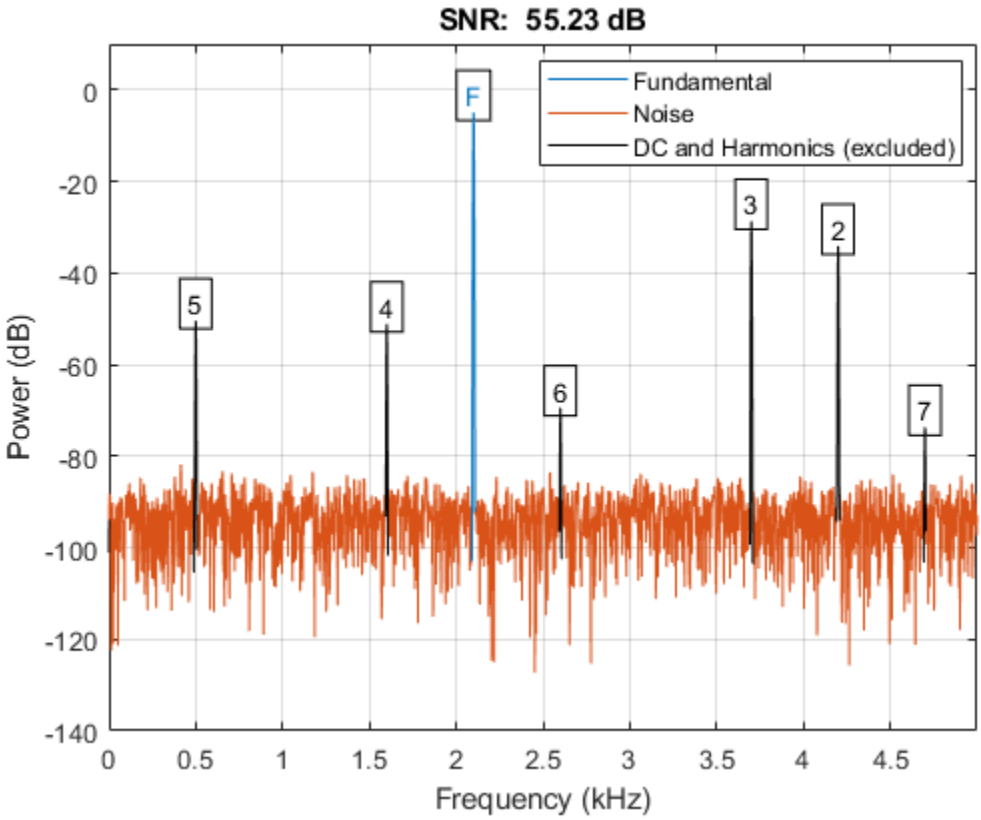
Compute the signal-to-noise ratio of the signal. By default, `snr` treats the aliased harmonics as part of the noise.

```
snr(x,Fs,7);
```



Repeat the computation, but now treat the aliased harmonics as part of the signal.

```
snr(x,Fs,7,'aliased');
```



Noise Power

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.

```
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));
```

Compute the noise power in the signal. Verify that it agrees with the definition.

```
[SNR,npow] = snr(x,fs);
compare = [10*log10(powfund)-npow SNR]
```

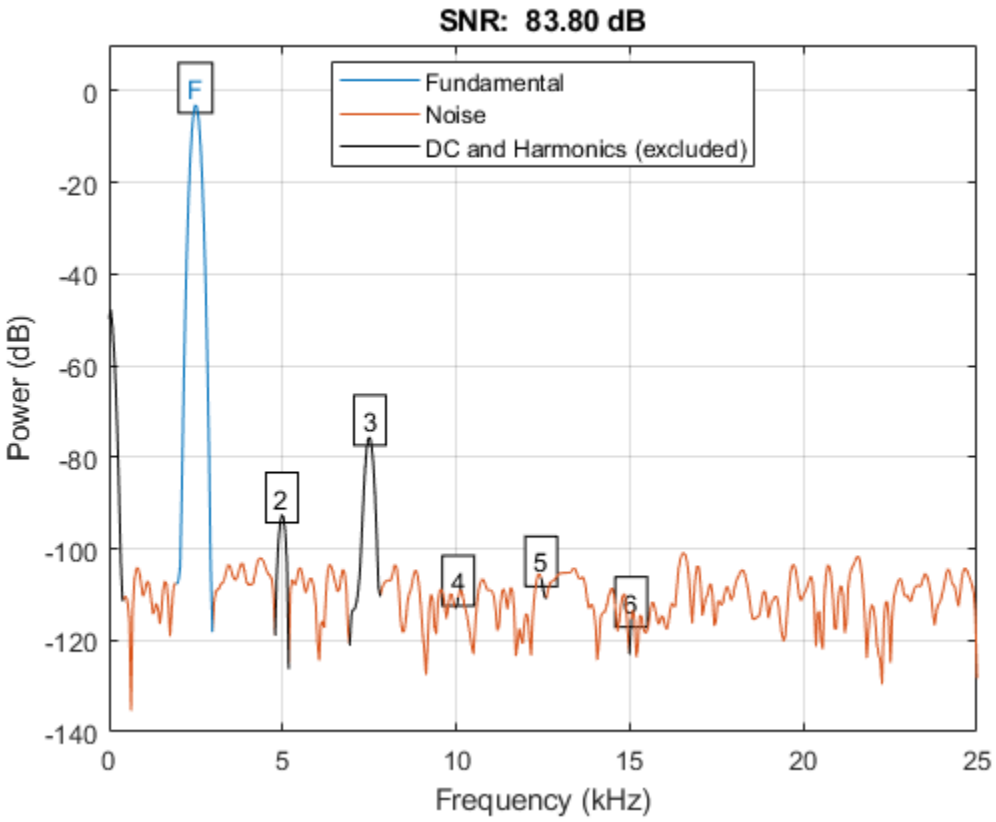
```
compare = 1×2
```

```
   17.0281   17.0178
```

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.

Input Arguments

`xi` — Input signal

vector | matrix | *N*-D array

Input signal, specified as a vector, matrix, or *N*-D array.

Data Types: `double` | `single`
Complex Number Support: Yes

`y` — Noise estimate

vector | matrix | *N*-D array

Estimate of the noise in the input signal, specified as a vector, matrix, or *N*-D array with the same dimensions as `xi`.

Data Types: `double` | `single`
Complex Number Support: Yes

`x` — Real-valued sinusoidal input signal

vector

Real-valued sinusoidal input signal, specified as a row or column vector.

Data Types: `double` | `single`

`fs` — Sample rate

1 (default) | positive real 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 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.

The power spectral density must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[pxx,f] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` specifies the periodogram PSD estimate of a noisy two-channel sinusoid sampled at 2π Hz and the frequencies at which it is computed.

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.

The power spectrum must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[sxx,w] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2),'power')` specifies the periodogram power spectrum estimate of a two-channel sinusoid embedded in white Gaussian noise and the normalized frequencies at which it is computed.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If `x` and `y` are complex at compile time, then their size must be constant at compile time.

See Also

`sfdr` | `sinad` | `thd` | `toi`

Topics

“Analyzing Harmonic Distortion”

Introduced in R2013b

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

where both `bi` and `ai`, with *i* = 1, ..., *L*, are 1-by-3 row vectors. The resulting `c` is a 1-by-*L* cell array of cells of the form

```
c = { {b1 a1} {b2 a2} ... {bL aL} }
```

`c = sos2cell(m,g)` with the optional gain term `g`, prepends the constant value `g` to `c`. When you use the added gain term in the command, `c` is a 1-by-*L* cell array of cells of the form

```
c = {{g,1} {b1,a1} {b2,a2}...{bL,aL} }
```

Examples

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 = 2×3

    1.0000    0.1677    0.2575
    1.0000    0.1677    0.2575
```

See Also

`tf2sos` | `cell2sos`

Introduced before R2006a

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

`[A,B,C,D] = sos2ss(sos)` converts a second-order section representation of a digital filter `sos` to its equivalent state-space form.

`[A,B,C,D] = sos2ss(sos,g)` converts a second-order section representation of a digital filter `sos` to its equivalent state-space form with gain `g`.

Examples

State-Space Representation of 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 = 4×4

   -10     0    10     1
     1     0     0     0
     0     1     0     0
     0     0     1     0


B = 4×1

     1
     0
     0
     0


C = 1×4

    42     4   -32    -2


D = -4
```

Input Arguments

sos — Second-order section representation

matrix

Second-order section representation, specified as a matrix. `sos` is an L-by- 6 matrix of the form

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

The entries of `sos` must be real for proper conversion to state space.

g — Overall system gain

real-valued scalar

Overall system gain, specified as a real-valued scalar. The function applies the gain to the system as

$$H(z) = g \prod_{k=1}^{L} H_k(z).$$

Output Arguments

A — State matrix

$2L$-by-$2L$ matrix

State matrix, returned as a $2L$-by-$2L$ matrix.

B — Input-to-state vector

$2L$-by-1 vector

Input-to-state vector, returned as a $2L$-by-1 vector.

C — Output-to-state vector

1-by-$2L$ vector

Output-to-state vector, returned as a 1-by-$2L$ vector.

D — Feedthrough matrix

scalar

Feedthrough matrix, returned as a scalar.

More About

Transfer Function

The discrete transfer function in second-order section form is given by

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

Algorithms

The `sos2ss` function first converts from second-order sections to transfer function using the `sos2tf` function, and then from transfer function to state-space form using the `tf2ss` function.

The single-input, single-output state-space representation is given by

$$x(n+1) = Ax(n) + Bu(n),$$
$$y(n) = Cx(n) + Du(n).$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`sos2tf` | `sos2zp` | `ss2sos` | `tf2ss` | `zp2ss`

Introduced before R2006a

sos2tf

Convert digital filter second-order section data to transfer function form

Syntax

```
[b,a] = sos2tf(sos)
[b,a] = sos2tf(sos,g)
```

Description

`[b,a] = sos2tf(sos)` returns the transfer function coefficients of a discrete-time system described in second-order section form by `sos`.

`[b,a] = sos2tf(sos,g)` returns the transfer function coefficients of a discrete-time system described in second-order section form by `sos` with gain `g`.

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 = 1×5

    -2     1     2     4     1


a = 1×5

     1    10     0   -10    -1
```

Input Arguments

sos — Second-order section representation

matrix

Second-order section representation, specified as a matrix. `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 $H(z)$:

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

Example: `[2 4 2 6 0 2;3 3 0 6 0 0]` specifies a third-order Butterworth filter with normalized 3 dB frequency 0.5π rad/sample.

Data Types: `double`
Complex Number Support: Yes

g — Overall system gain

real scalar

Overall system gain, specified as a real scalar.

Data Types: `double`

Output Arguments

b, a — Transfer function coefficients

row vectors

Transfer function coefficients, returned as row vectors. `b` and `a` contain the numerator and denominator coefficients of $H(z)$ stored in descending powers of z:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2z^{-1} + \cdots + b_{n+1}z^{-n}}{a_1 + a_2z^{-1} + \cdots + a_{m+1}z^{-m}}.$$

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 round-off errors may occur when forming the transfer function.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`latc2tf` | `sos2ss` | `sos2zp` | `ss2tf` | `tf2sos` | `zp2tf`

Introduced before R2006a

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

`[z,p,k] = sos2zp(sos)` returns the zeros, poles, and gain of a system whose second-order section representation is given by `sos`.

`[z,p,k] = sos2zp(sos,g)` returns the zeros, poles, and gain of a system whose second-order section representation is given by `sos` with gain `g`.

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 = 4×1 complex

  -0.5000 + 0.8660i
  -0.5000 - 0.8660i
   1.7808 + 0.0000i
  -0.2808 + 0.0000i


p = 4×1

   -1.0000
    1.0000
   -9.8990
   -0.1010


k = -2
```

Input Arguments

sos — Second-order section representation

matrix

Second-order section representation, specified as a matrix. `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 $H(z)$:

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

Example: `[2 4 2 6 0 2;3 3 0 6 0 0]` specifies a third-order Butterworth filter with normalized 3 dB frequency 0.5π rad/sample.

Data Types: `single` | `double`
Complex Number Support: Yes

g — Overall system gain
real scalar

Overall system gain, specified as a real scalar.

Data Types: `single` | `double`

Output Arguments

z — Zeros
vector

Zeros of the system, returned as a vector.

p — Poles
vector

Poles of the system, returned as a vector.

k — Scalar gain
scalar

Scalar gain of the system, returned as a scalar.

Algorithms

`sos2zp` finds the poles and zeros of each second-order section by repeatedly calling `tf2zp`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The complexity of outputs, `z`, `p`, and `k`, might be different in MATLAB and the generated code.
- The order of outputs, `z` and `p`, might be different in MATLAB and the generated code.

See Also

`sos2ss` | `sos2tf` | `ss2zp` | `tf2zp` | `tf2zpk` | `zp2sos`

Introduced before R2006a

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 input signal `x`.

- If `x` is a matrix, then the function operates along the first dimension and returns the filtered data for each column.
- If `x` is a multidimensional array, then the function operates along the first array dimension with size greater than 1.

`y = sosfilt(sos,x,dim)` operates along the dimension `dim`.

Examples

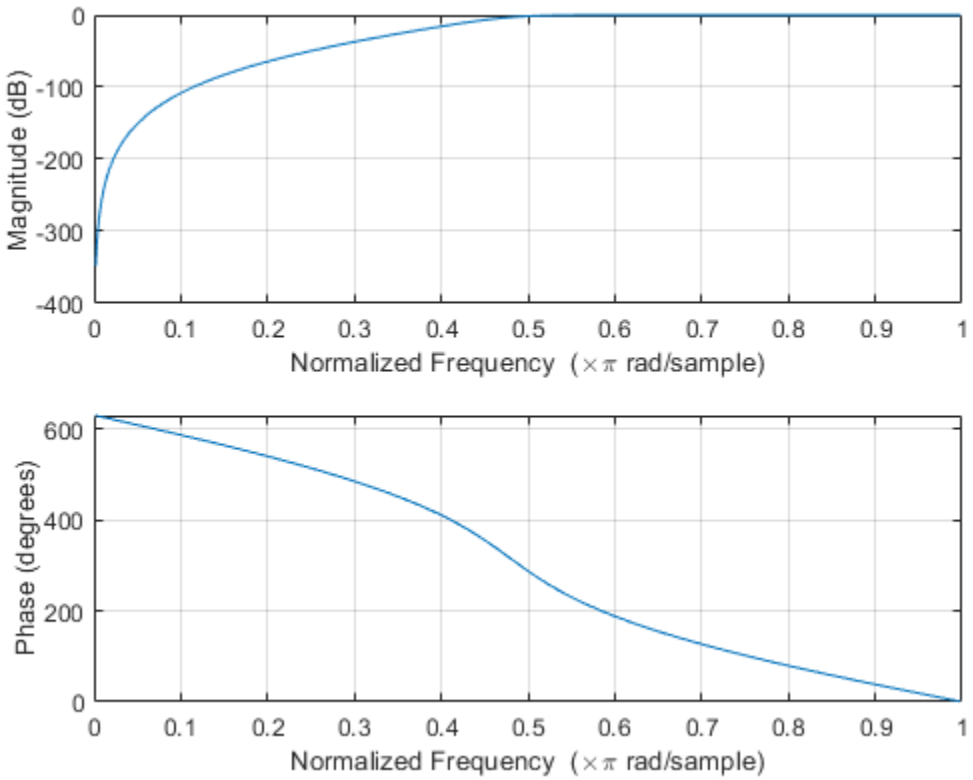
Second-Order Section Filtering

Load `chirp.mat`. The file contains a signal, `y`, that has most of its power above `Fs/4`, or half the Nyquist frequency. The sample rate is 8192 Hz.

```
load chirp

t = (0:length(y)-1)/Fs;
```

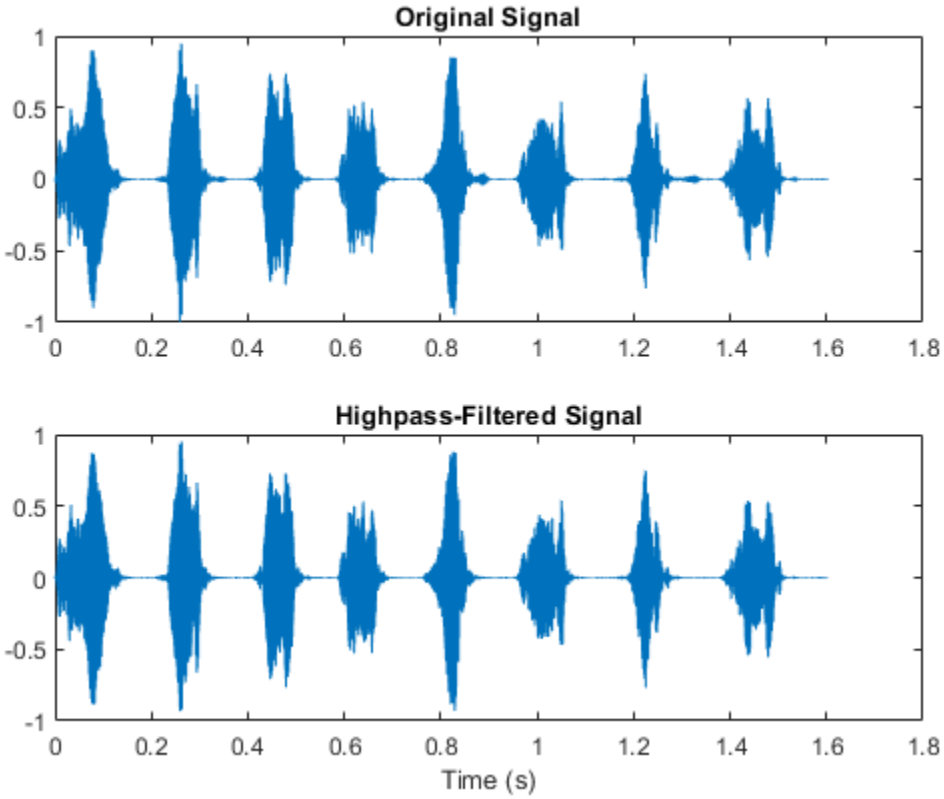
Design a seventh-order Butterworth highpass filter to attenuate the components of the signal below `Fs/4`. Use a normalized cutoff frequency of 0.48π rad/sample. Express the filter coefficients in terms of second-order sections.

```
[zhi,phi,khi] = butter(7,0.48,'high');
soshi = zp2sos(zhi,phi,khi);

freqz(soshi)
```



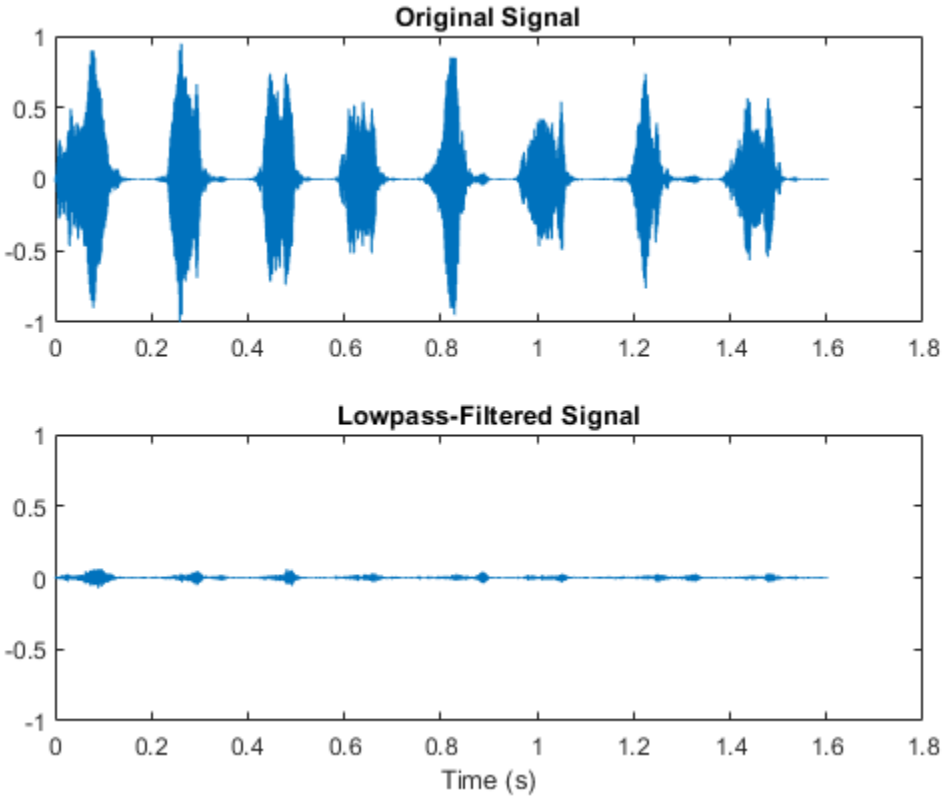
Filter the signal. Display the original and highpass-filtered signals. Use the same *y*-axis scale for both plots.

```
outhi = sosfilt(soshi,y);

figure
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. The result is mostly noise.

```
[zlo,plo,klo] = butter(7,0.48);
soslo = zp2sos(zlo,plo,klo);

outlo = sosfilt(soslo,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)
```



Input Arguments

sos — Second-order section digital filter

L-by-6 matrix

Second-order section digital filter, specified as an *L*-by-6 matrix where *L* is the number of second-order sections. The matrix

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

represents the second-order section digital filter

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

Example: `[b,a] = butter(3,1/32); sos = tf2sos(b,a)` specifies a third-order Butterworth filter with a normalized 3 dB frequency of $\pi/32$ rad/sample.

Data Types: `single` | `double`

x — Input signal
vector | matrix | *N*-D array

Input signal, specified as a vector, matrix, or *N*-D array.

Example: `x = [2 1].*sin(2*pi*(0:127)'./[16 64])` specifies a two-channel sinusoid.

Data Types: `single` | `double`
Complex Number Support: Yes

dim — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. By default, the function operates along the first array dimension of `x` with size greater than 1.

Data Types: `single` | `double`

Output Arguments

y — Filtered signal
vector | matrix | *N*-D array

Filtered signal, returned as a vector, matrix, or *N*-D array. `y` has the same size as `x`.

References

[1] Bank, Balázs. "Converting Infinite Impulse Response Filters to Parallel Form". *IEEE Signal Processing Magazine*. Vol. 35, Number 3, May 2018, pp. 124-130.

[2] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Input filter `sos` must be stable. Use `isstable` to check for filter stability.
- All second-order subsections of the input filter must be IIR.
- The `gpuArray` version of sosfilt uses a parallel algorithm [1] which is different from the MATLAB version. The algorithms give different results for complex-valued input with `NaN` or `Inf` values:
  - In the MATLAB version, the `NaN`s and `Inf`s propagate only in the real part.
  - In the `gpuArray` version, the `NaN`s and `Inf`s propagate in both the real part and the imaginary part.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`filter` | `medfilt1` | `sgolayfilt`

Introduced before R2006a

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)

[ ___ ,ps] = spectrogram( ___ )

[ ___ ] = spectrogram( ___ ,'reassigned')
[ ___ ,ps,fc,tc] = spectrogram( ___ )

[ ___ ] = spectrogram( ___ ,freqrange)
[ ___ ] = spectrogram( ___ ,Name,Value)

[ ___ ] = 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 segments and perform windowing.

`s = spectrogram(x,window,noverlap)` uses `noverlap` samples of overlap between adjoining segments.

`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`. `fs` must be the fifth input to `spectrogram`. To input a sample rate and still use the default values of the preceding optional arguments, specify these arguments as empty, `[]`.

`[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`.

`[___,ps] = spectrogram(___)` also returns a matrix, `ps`, containing an estimate of the power spectral density (PSD) or the power spectrum of each segment.

`[___] = spectrogram(___,'reassigned')` reassigns each PSD or power spectrum estimate to the location of its center of energy. If your signal contains well-localized temporal or spectral components, then this option generates a sharper spectrogram.

`[___,ps,fc,tc] = spectrogram(___)` also returns two matrices, `fc` and `tc`, containing the frequency and time of the center of energy of each PSD or power spectrum estimate.

`[___] = spectrogram(___,freqrange)` returns the PSD or power spectrum estimate over the frequency range specified by `freqrange`. Valid options for `freqrange` are `'onesided'`, `'twosided'`, and `'centered'`.

`[___] = spectrogram(___,Name,Value)` specifies additional options using name-value pair arguments. Options include the minimum threshold and output time dimension.

`[___] = spectrogram(___,spectrumtype)` returns PSD estimates if `spectrumtype` is specified as `'psd'` and returns power spectrum estimates 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

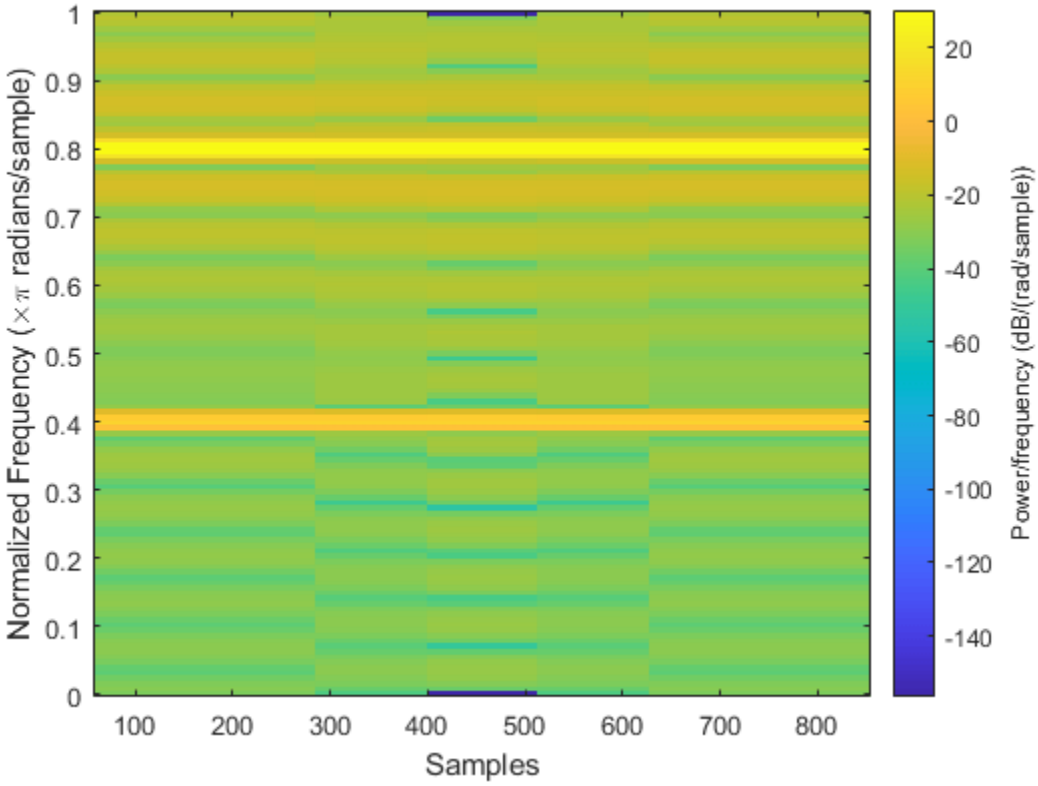
Default Values of Spectrogram

Generate $N_x = 1024$ samples of a signal that consists of a sum of sinusoids. The normalized frequencies of the sinusoids are $2\pi/5$ rad/sample and $4\pi/5$ rad/sample. The higher frequency sinusoid has 10 times the amplitude of the other sinusoid.

```
N = 1024;
n = 0:N-1;

w0 = 2*pi/5;
x = sin(w0*n)+10*sin(2*w0*n);
```

Compute the short-time Fourier transform using the function defaults. Plot the spectrogram.

```
s = spectrogram(x);

spectrogram(x,'yaxis')
```



Repeat the computation.

- Divide the signal into sections of length nsc = $\lfloor N_x/4.5 \rfloor$.
- Window the sections using a Hamming window.
- Specify 50% overlap between contiguous sections.
- To compute the FFT, use $\max(256, 2^p)$ points, where $p = \lceil \log_2 \text{nsc} \rceil$.

Verify that the two approaches give identical results.

```
Nx = length(x);
nsc = floor(Nx/4.5);
nov = floor(nsc/2);
nff = max(256,2^nextpow2(nsc));

t = spectrogram(x,hamming(nsc),nov,nff);

maxerr = max(abs(abs(t(:))-abs(s(:))))
```

```
maxerr = 0
```

Divide the signal into 8 sections of equal length, with 50% overlap between sections. Specify the same FFT length as in the preceding step. Compute the short-time Fourier transform and verify that it gives the same result as the previous two procedures.

```
ns = 8;
ov = 0.5;
```

```
lsc = floor(Nx/(ns-(ns-1)*ov));

t = spectrogram(x,lsc,floor(ov*lsc),nff);

maxerr = max(abs(abs(t(:))-abs(s(:))))

maxerr = 0
```

Frequency Along *x*-Axis

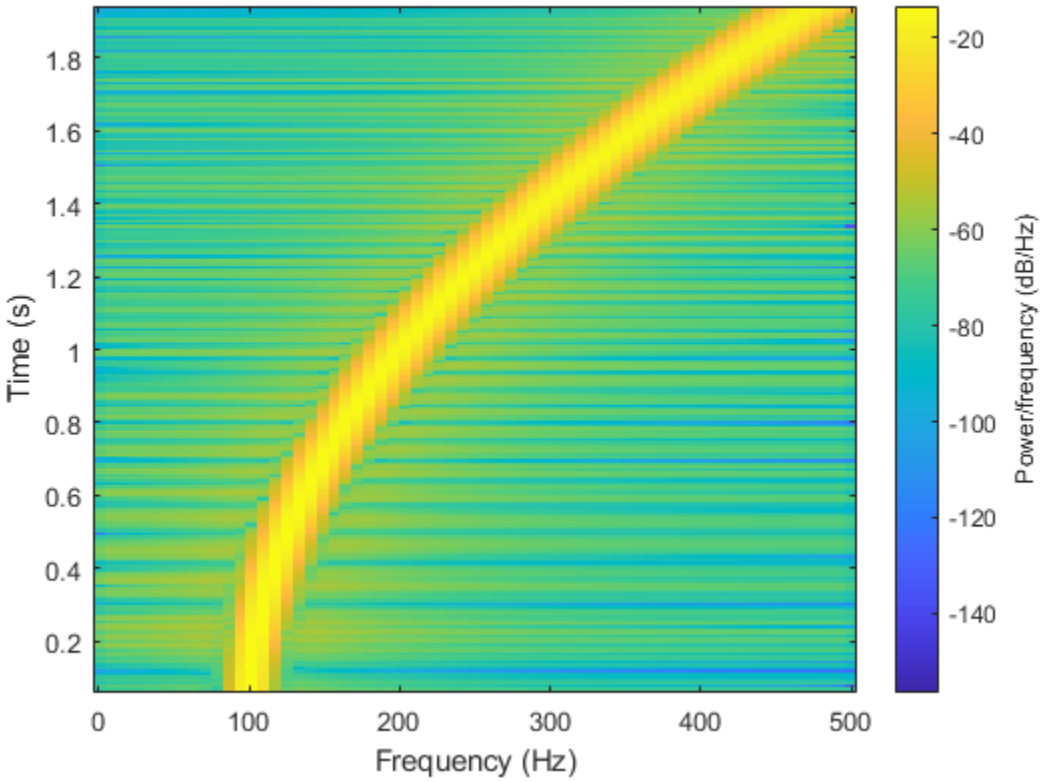
Generate a quadratic chirp, `x`, sampled at 1 kHz for 2 seconds. The frequency of the chirp is 100 Hz initially and crosses 200 Hz at `t` = 1 s.

```
t = 0:0.001:2;
x = chirp(t,100,1,200,'quadratic');
```

Compute and display the spectrogram of `x`.

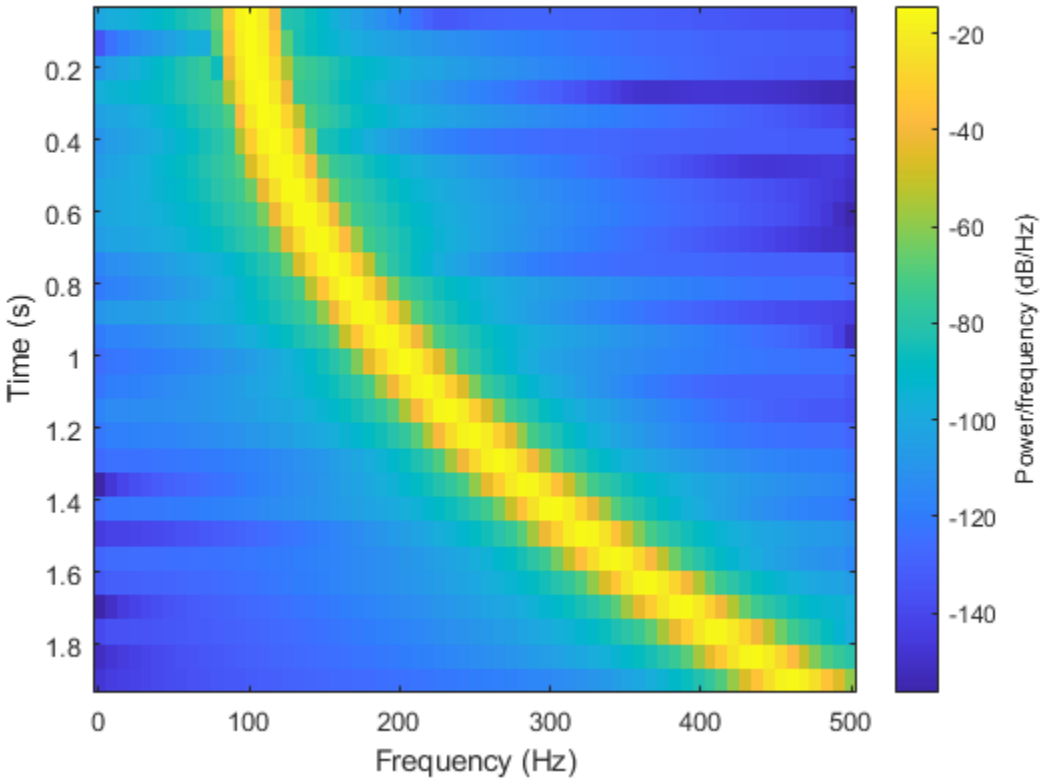
- Divide the signal into sections of length 128, windowed with a Hamming window.
- Specify 120 samples of overlap between adjoining sections.
- Evaluate the spectrum at ⌊128/2 + 1⌋ = 65 frequencies and ⌊(length(x) − 120)/(128 − 120)⌋ = 235 time bins.

```
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.

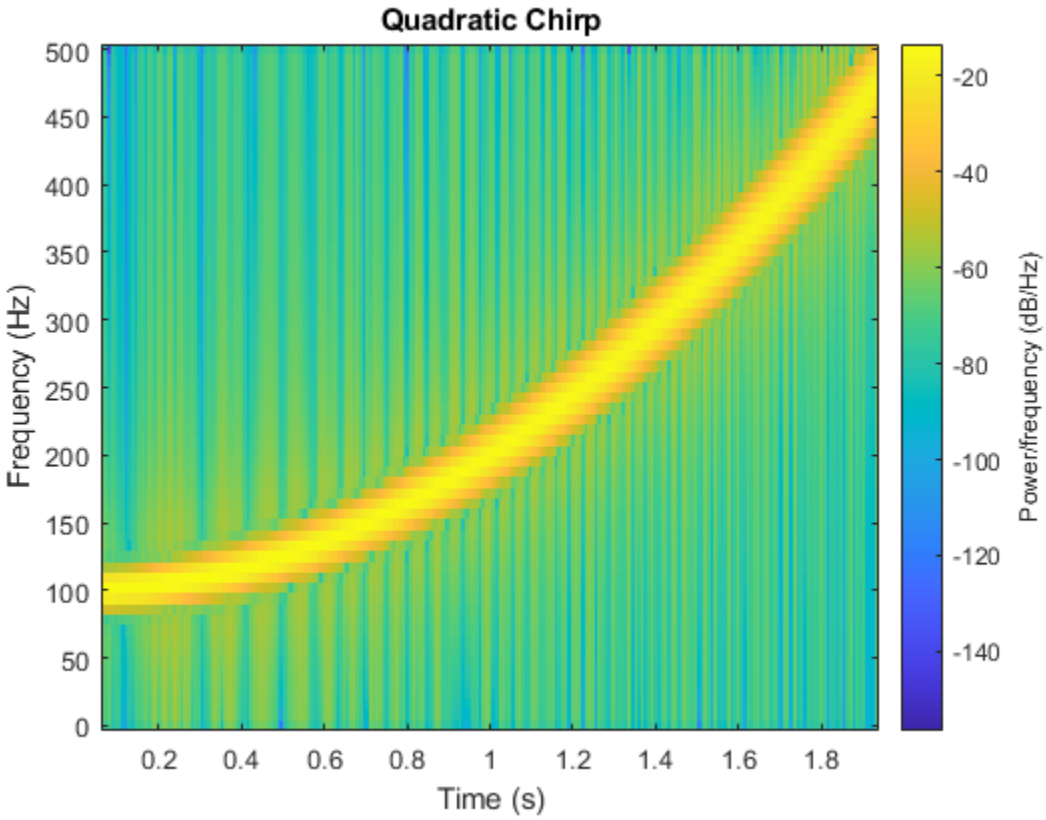
```
spectrogram(x,blackman(128),60,128,1e3)
ax = gca;
ax.YDir = 'reverse';
```



Power Spectral Densities of Chirps

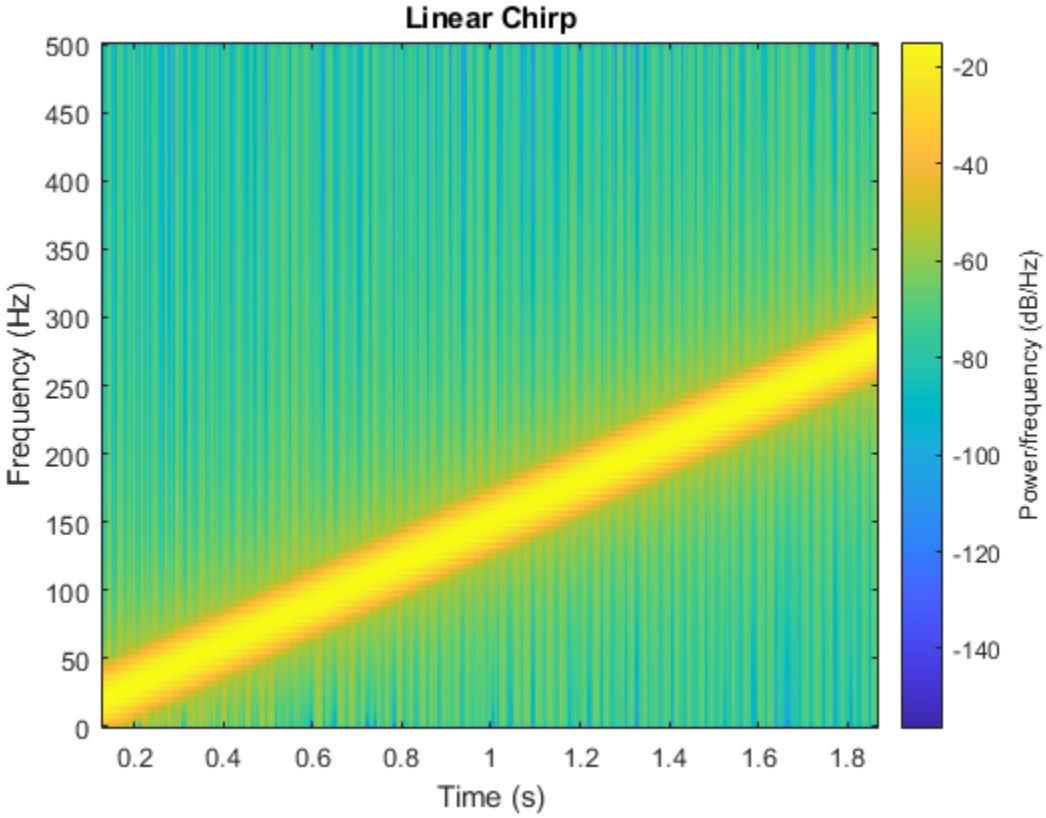
Compute and display the PSD of each segment of a quadratic chirp that starts at 100 Hz and crosses 200 Hz at $t = 1$ second. Specify a sample rate of 1 kHz, a segment length of 128 samples, and an overlap of 120 samples. Use 128 DFT points and the default Hamming window.

```
fs = 1000;
t = 0:1/fs:2;
x = chirp(t,100,1,200,'quadratic');

spectrogram(x,128,120,128,fs,'yaxis')
title('Quadratic Chirp')
```



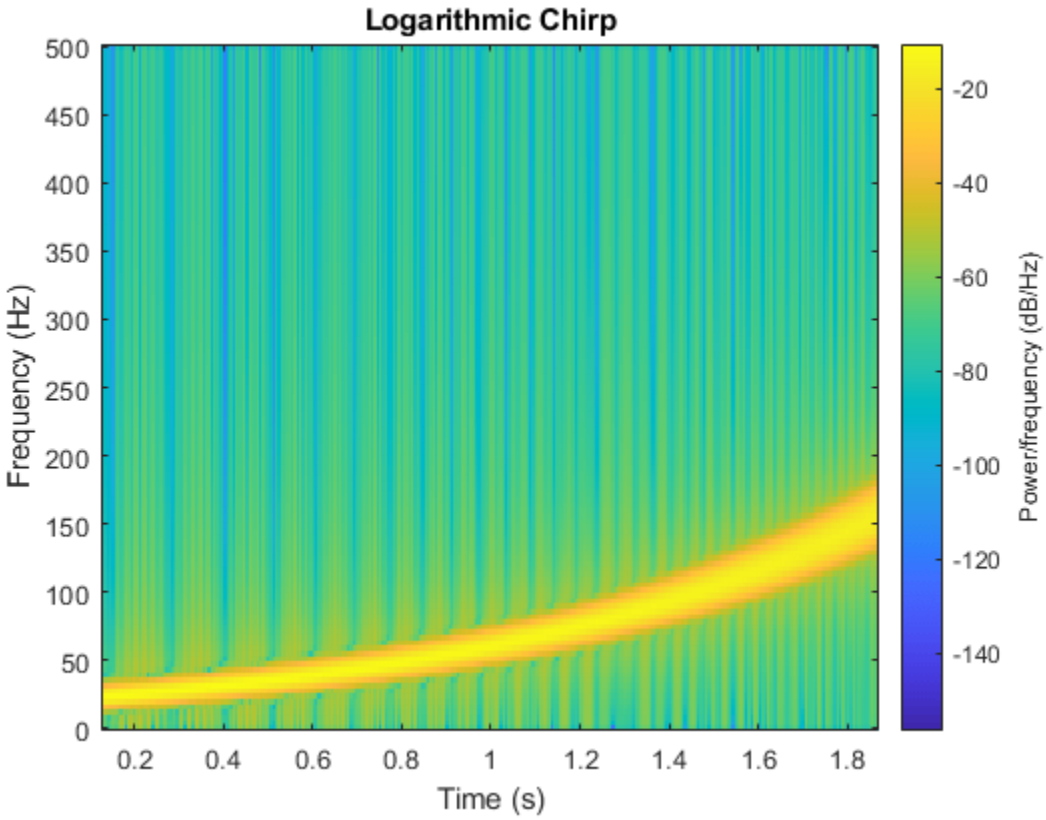
Compute and display the PSD of each segment of a linear chirp sampled at 1 kHz that starts at DC and crosses 150 Hz at $t = 1$ second. Specify a segment length of 256 samples and an overlap of 250 samples. Use the default Hamming window and 256 DFT points.

```
x = chirp(t,0,1,150);

spectrogram(x,256,250,256,fs,'yaxis')
title('Linear Chirp')
```



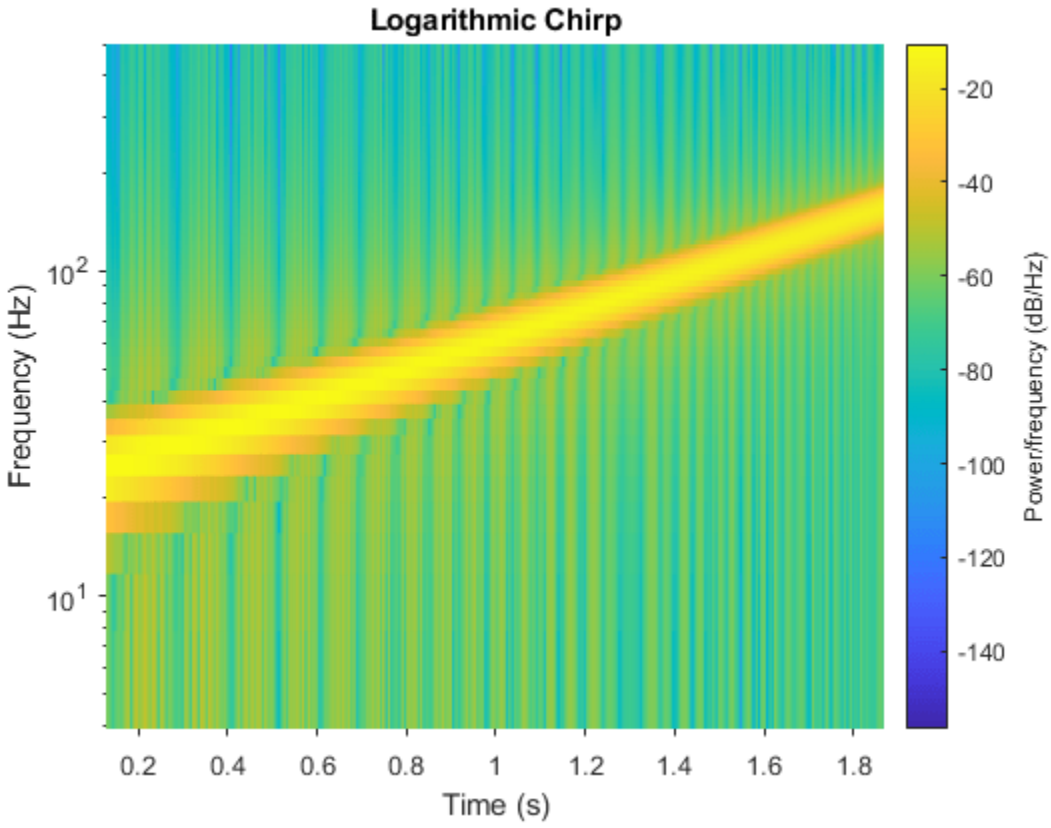
Compute and display the PSD of each segment of a logarithmic chirp sampled at 1 kHz that starts at 20 Hz and crosses 60 Hz at $t = 1$ second. Specify a segment length of 256 samples and an overlap of 250 samples. Use the default Hamming window and 256 DFT points.

```
x = chirp(t,20,1,60,'logarithmic');

spectrogram(x,256,250,[],fs,'yaxis')
title('Logarithmic Chirp')
```



Use a logarithmic scale for the frequency axis. The spectrogram becomes a line.

```
ax = gca;
ax.YScale = 'log';
```



Spectrogram and Instantaneous Frequency

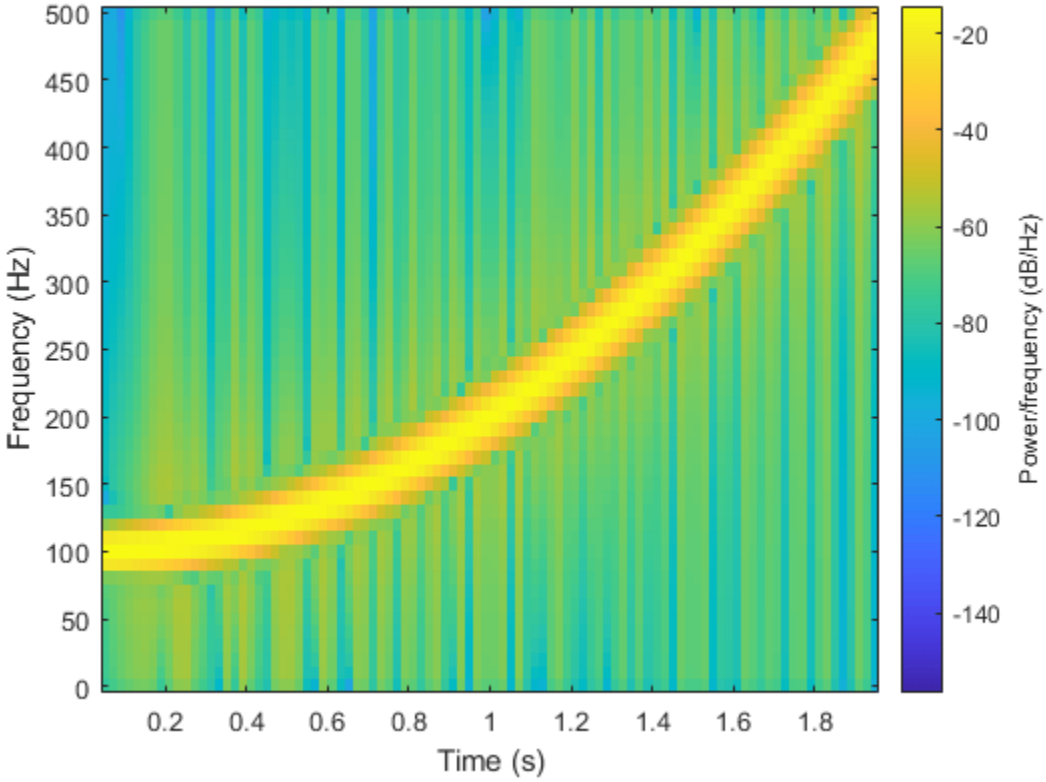
Use the `spectrogram` function to measure and track the instantaneous frequency of a signal.

Generate a quadratic chirp sampled at 1 kHz for two seconds. Specify the chirp so that 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');
```

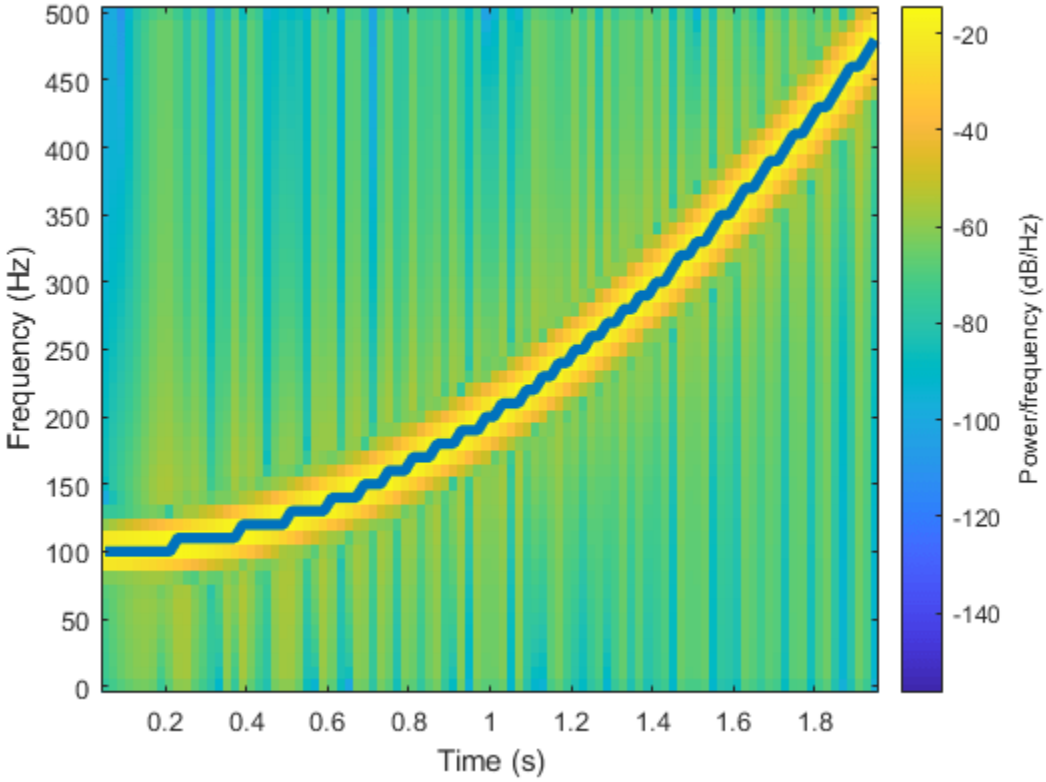
Estimate the spectrum of the chirp using the short-time Fourier transform implemented in the `spectrogram` function. Divide the signal into sections of length 100, windowed with a Hamming window. Specify 80 samples of overlap between adjoining sections and evaluate the spectrum at $\lfloor 100/2 + 1 \rfloor = 51$ frequencies.

```
spectrogram(y,100,80,100,fs,'yaxis')
```



Track the chirp frequency by finding the time-frequency ridge with highest energy across the $\lfloor (2000-80)/(100-80) \rfloor = 96$ time points. Overlay the instantaneous frequency on the spectrogram plot.

```
[~,f,t,p] = spectrogram(y,100,80,100,fs);

[fridge,~,lr] = tfridge(p,f);

hold on
plot3(t,fridge,abs(p(lr)),'LineWidth',4)
hold off
```



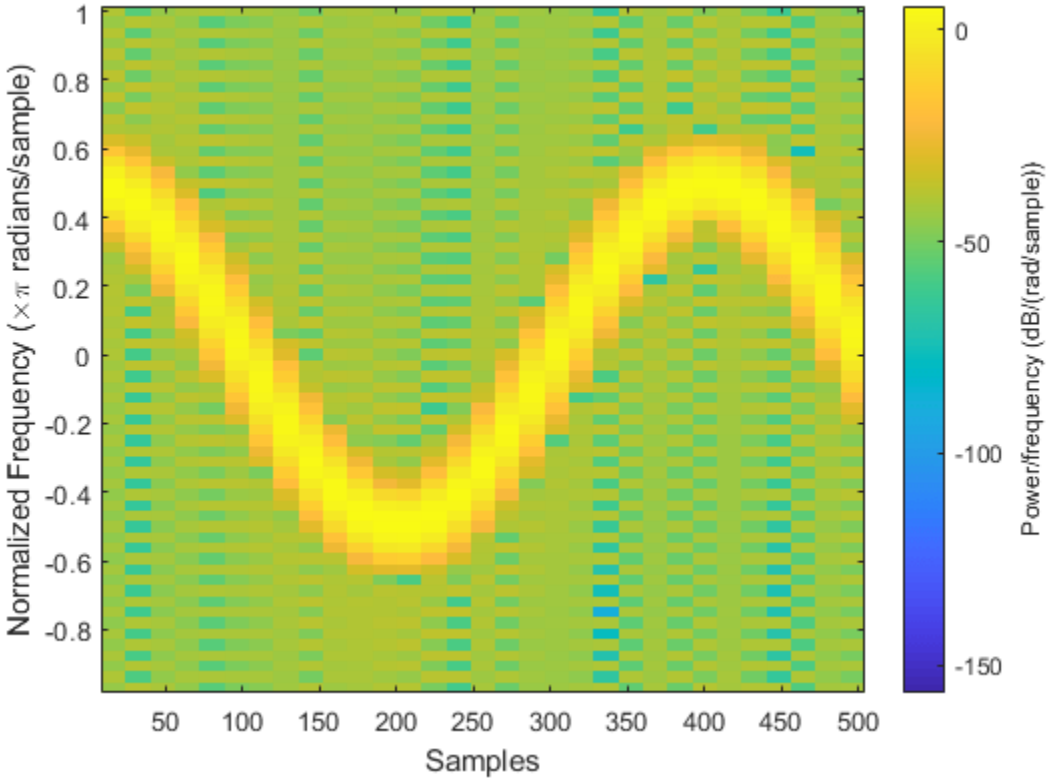
Spectrogram of Complex Signal

Generate 512 samples of a chirp with sinusoidally varying frequency content.

```
N = 512;
n = 0:N-1;

x = exp(1j*pi*sin(8*n/N)*32);
```

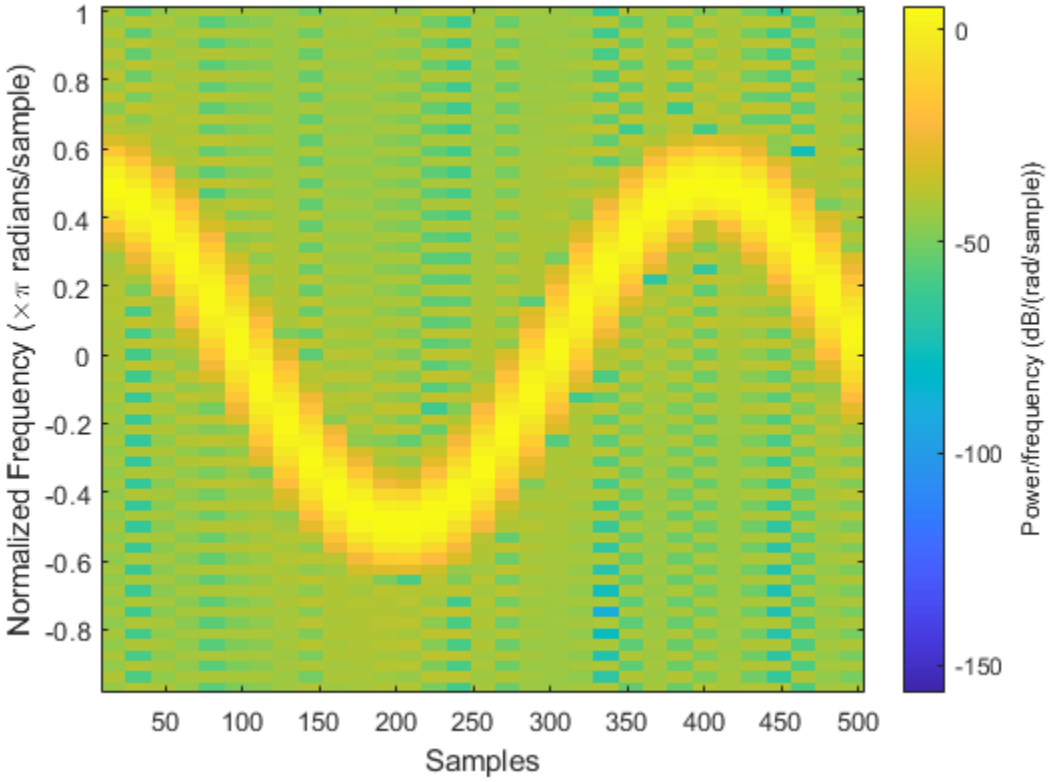
Compute the centered two-sided short-time Fourier transform of the chirp. Divide the signal into 32-sample segments with 16-sample overlap. Specify 64 DFT points. Plot the spectrogram.

```
[scalar,fs,ts] = spectrogram(x,32,16,64,'centered');

spectrogram(x,32,16,64,'centered','yaxis')
```



Obtain the same result by computing the spectrogram on 64 equispaced frequencies over the interval $(-\pi,\pi]$. The `'centered'` option is not necessary.

```
fintv = -pi+pi/32:pi/32:pi;

[vector,fv,tv] = spectrogram(x,32,16,fintv);

spectrogram(x,32,16,fintv,'yaxis')
```



Reassigned Spectrogram of Quadratic Chirp

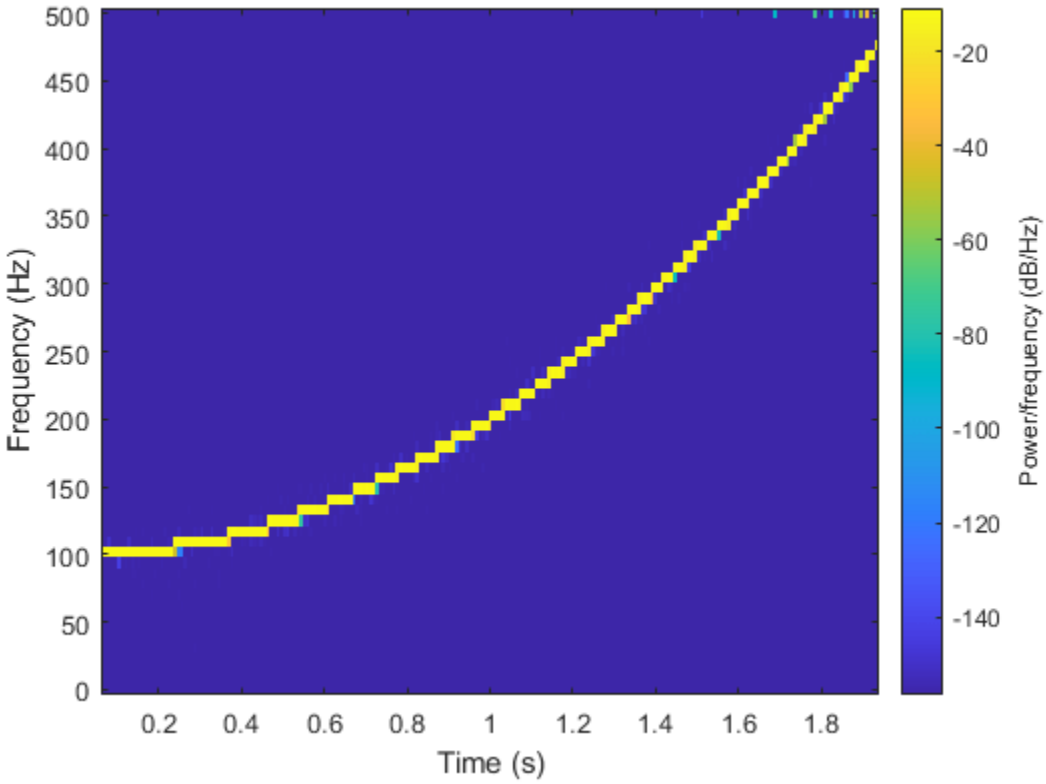
Generate a chirp signal sampled for 2 seconds at 1 kHz. Specify the chirp so that its frequency is initially 100 Hz and increases to 200 Hz after 1 second.

```
Fs = 1000;
t = 0:1/Fs:2;
y = chirp(t,100,1,200,'quadratic');
```

Estimate the reassigned spectrogram of the signal.

- Divide the signal into sections of length 128, windowed with a Kaiser window with shape parameter $\beta = 18$.
- Specify 120 samples of overlap between adjoining sections.
- Evaluate the spectrum at $\lfloor 128/2 \rfloor = 65$ frequencies and $\lfloor (\text{length}(x) - 120)/(128 - 120) \rfloor = 235$ time bins.

```
spectrogram(y,kaiser(128,18),120,128,Fs,'reassigned','yaxis')
```



Spectrogram with Threshold

Generate a chirp signal sampled for 2 seconds at 1 kHz. Specify the chirp so that its frequency is initially 100 Hz and increases to 200 Hz after 1 second.

```
Fs = 1000;
t = 0:1/Fs:2;
y = chirp(t,100,1,200,'quadratic');
```

Estimate the time-dependent power spectral density (PSD) of the signal.

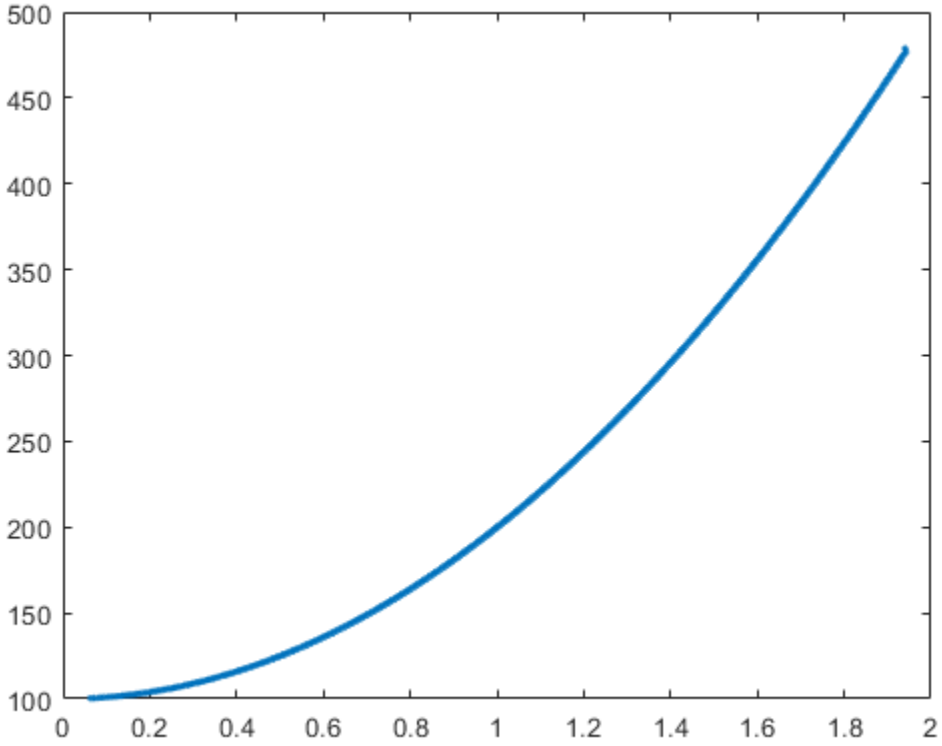
- Divide the signal into sections of length 128, windowed with a Kaiser window with shape parameter $\beta = 18$.
- Specify 120 samples of overlap between adjoining sections.
- Evaluate the spectrum at $\lfloor 128/2 \rfloor = 65$ frequencies and $\lfloor (\text{length}(x) - 120)/(128 - 120) \rfloor = 235$ time bins.

Output the frequency and time of the center of gravity of each PSD estimate. Set to zero those elements of the PSD smaller than −30 dB.

```
[~,~,~,pxx,fc,tc] = spectrogram(y,kaiser(128,18),120,128,Fs, ...
    'MinThreshold',-30);
```

Plot the nonzero elements as functions of the center-of-gravity frequencies and times.

```
plot(tc(pxx>0),fc(pxx>0),'.')
```



Spectrogram Reassignment and Thresholding

Generate a signal sampled at 1024 Hz for 2 seconds.

```
nSamp = 2048;
Fs = 1024;
t = (0:nSamp-1)'/Fs;
```

During the first second, the signal consists of a 400 Hz sinusoid and a concave quadratic chirp. Specify the chirp so that it is symmetric about the interval midpoint, starting and ending at a frequency of 250 Hz and attaining a minimum of 150 Hz.

```
t1 = t(1:nSamp/2);

x11 = sin(2*pi*400*t1);
x12 = chirp(t1-t1(nSamp/4),150,nSamp/Fs,1750,'quadratic');
x1 = x11+x12;
```

The rest of the signal consists of two linear chirps of decreasing frequency. One chirp has an initial frequency of 250 Hz that decreases to 100 Hz. The other chirp has an initial frequency of 400 Hz that decreases to 250 Hz.

```
t2 = t(nSamp/2+1:nSamp);
```

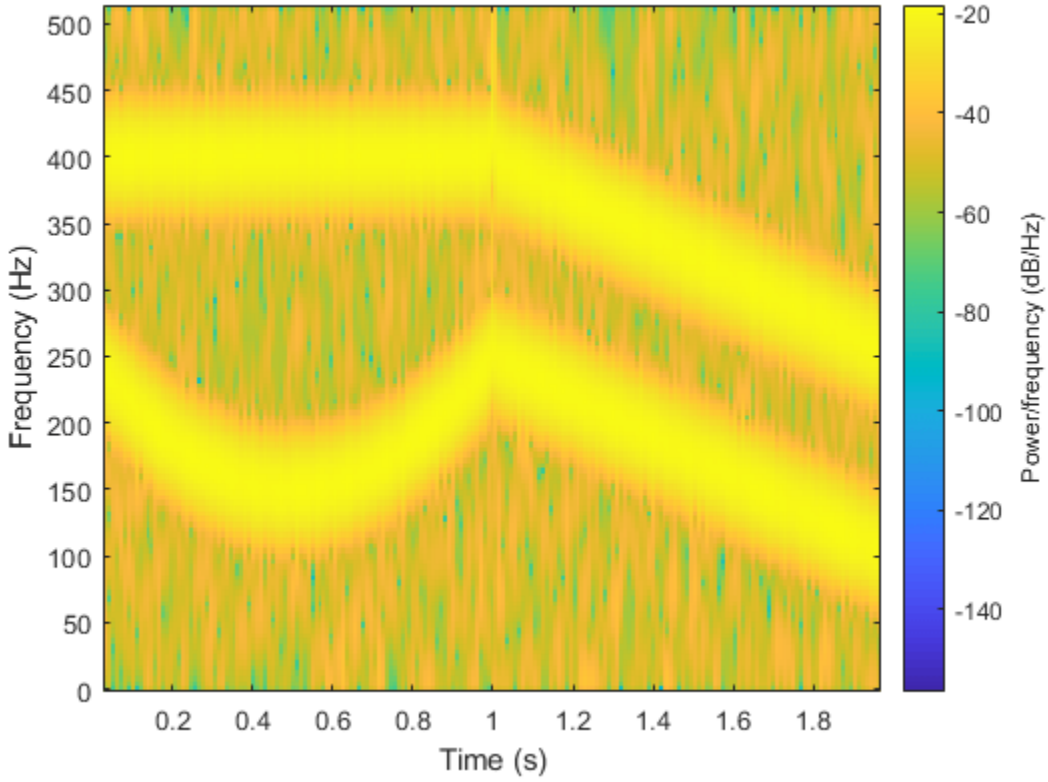
```
x21 = chirp(t2,400,nSamp/Fs,100);
x22 = chirp(t2,550,nSamp/Fs,250);
x2 = x21+x22;
```

Add white Gaussian noise to the signal. Specify a signal-to-noise ratio of 20 dB. Reset the random number generator for reproducible results.

```
SNR = 20;
rng('default')

sig = [x1;x2];
sig = sig + randn(size(sig))*std(sig)/db2mag(SNR);
```

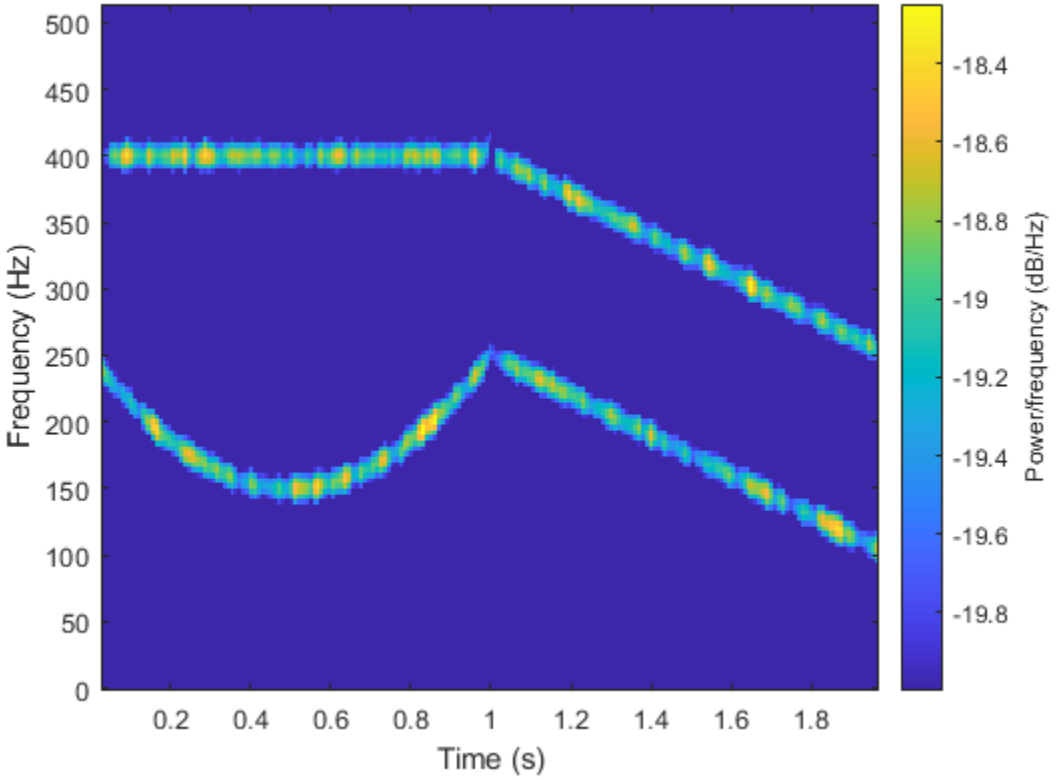
Compute and plot the spectrogram of the signal. Specify a Kaiser window of length 63 with a shape parameter $\beta = 17$, 10 fewer samples of overlap between adjoining sections, and an FFT length of 256.

```
nwin = 63;
wind = kaiser(nwin,17);
nlap = nwin-10;
nfft = 256;

spectrogram(sig,wind,nlap,nfft,Fs,'yaxis')
```



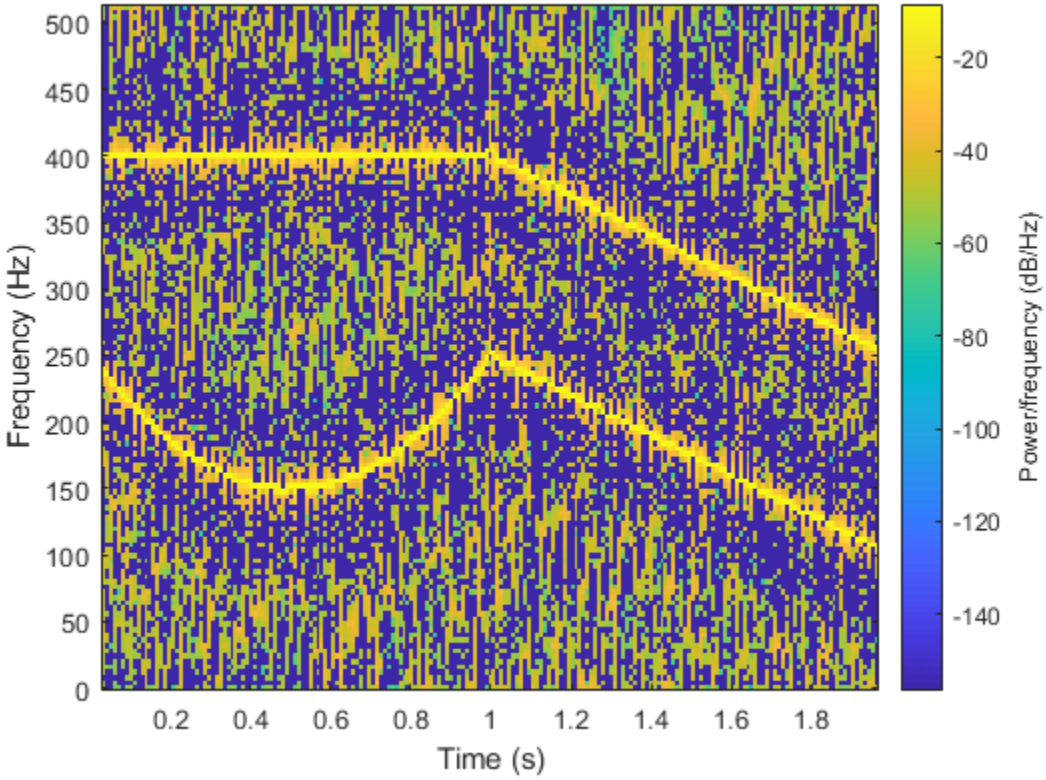
Threshold the spectrogram so that any elements with values smaller than the SNR are set to zero.

```
spectrogram(sig,wind,nlap,nfft,Fs,'MinThreshold',-SNR,'yaxis')
```



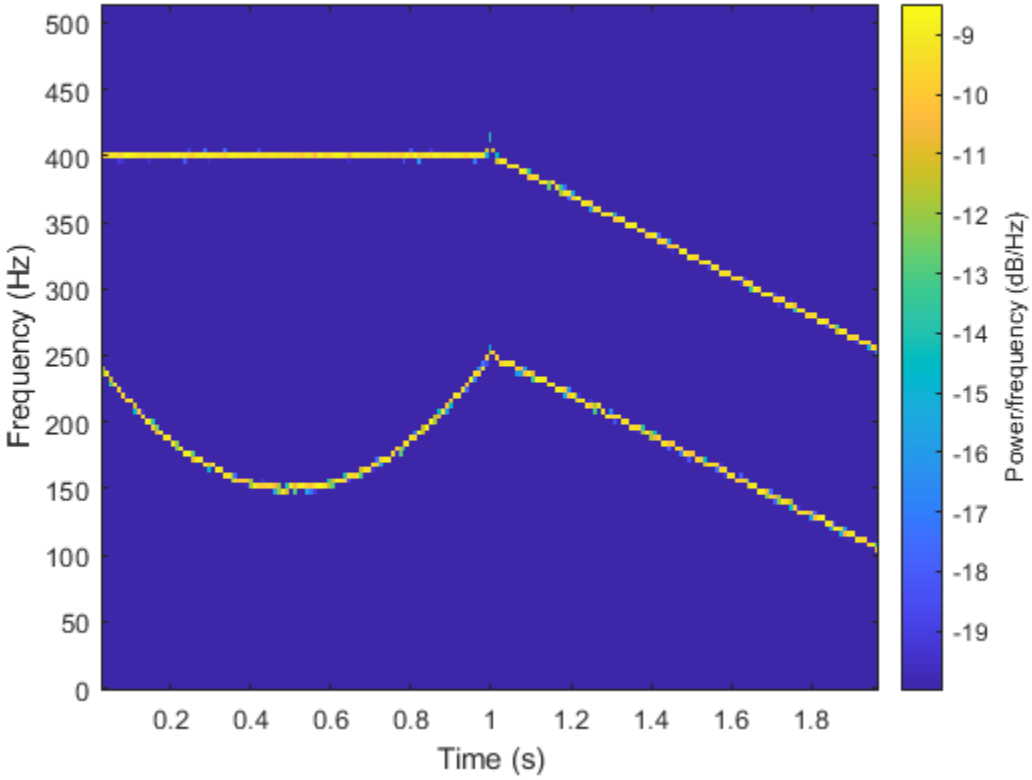
Reassign each PSD estimate to the location of its center of energy.

```
spectrogram(sig,wind,nlap,nfft,Fs,'reassign','yaxis')
```



Threshold the reassigned spectrogram so that any elements with values smaller than the SNR are set to zero.

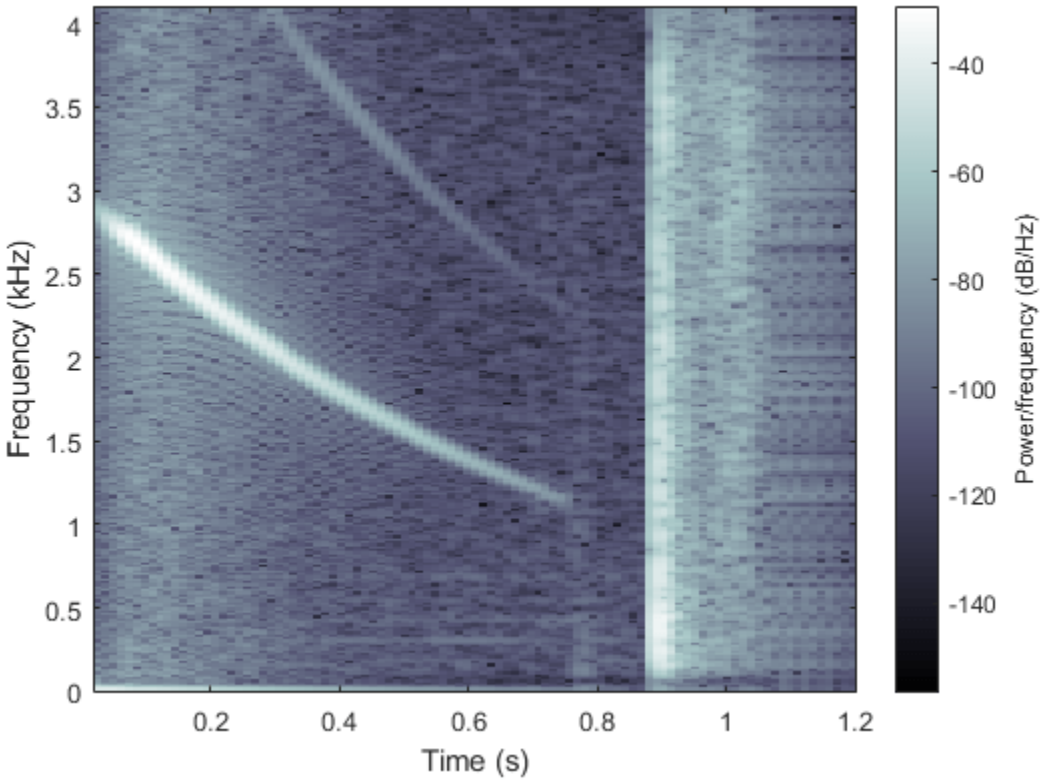
```
spectrogram(sig,wind,nlap,nfft,Fs,'reassign','MinThreshold',-SNR,'yaxis')
```



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.

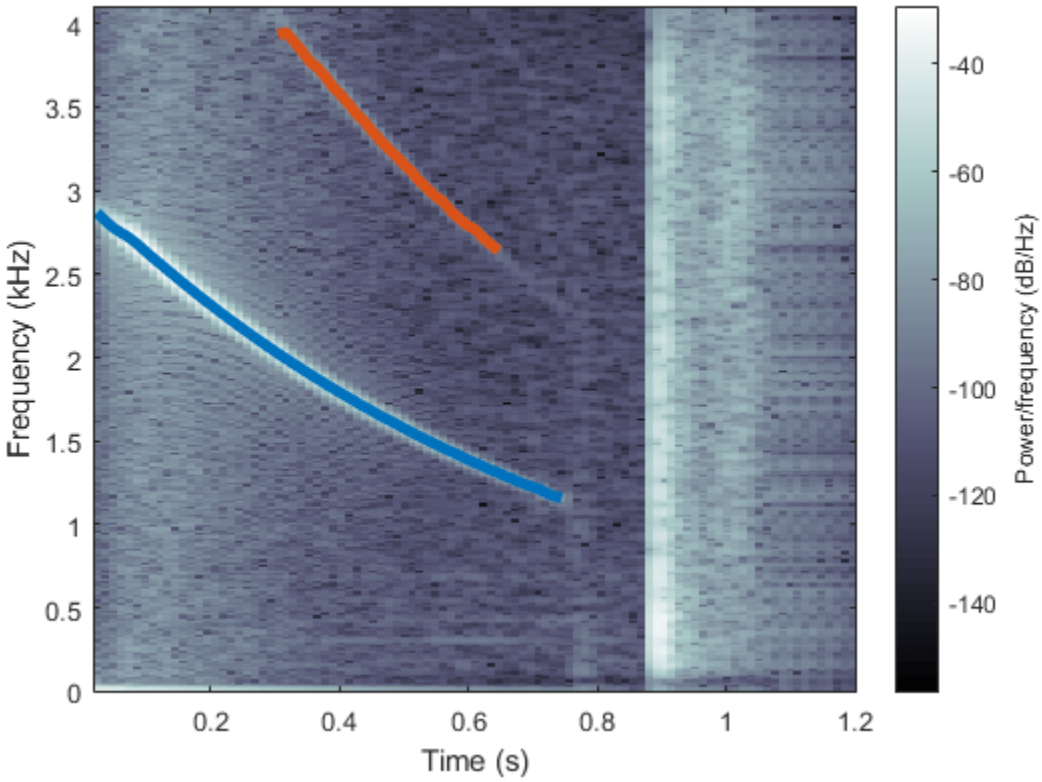
```
f1 = f > 100;
t1 = t < 0.75;

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
```



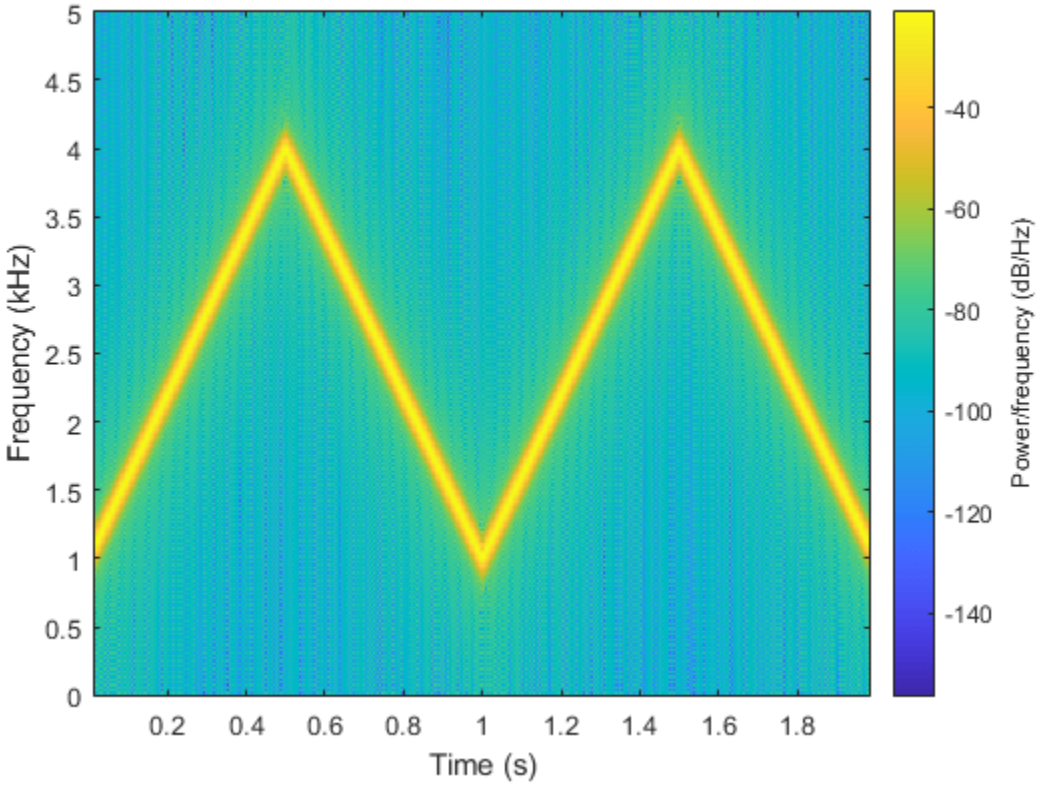
3D Spectrogram Visualization

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);
```

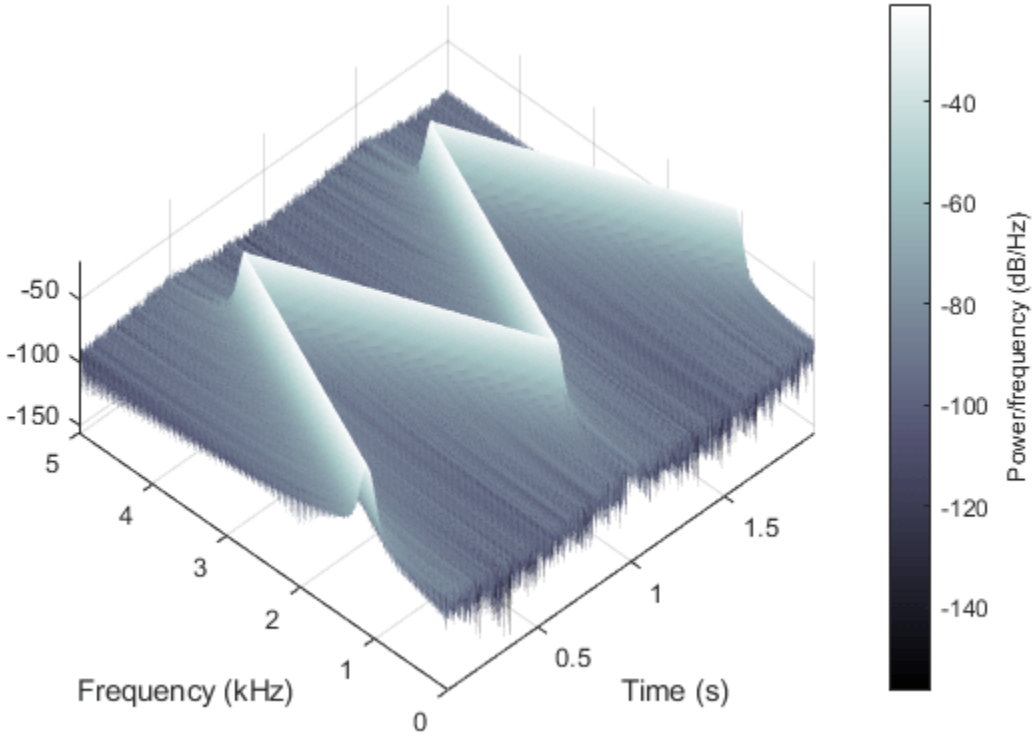
Compute and plot the spectrogram of the signal. Use a Kaiser window of length 256 and shape parameter $\beta = 5$. Specify 220 samples of section-to-section overlap and 512 DFT points. Plot the frequency on the *y*-axis. Use the default colormap and view.

```
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.

```
view(-45,65)
colormap bone
```



Input Arguments

x — Input signal

vector | `gpuArray` objects

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 segments:

- If `window` is an integer, then `spectrogram` divides `x` into segments of length `window` and windows each segment with a Hamming window of that length.
- If `window` is a vector, then `spectrogram` divides `x` into segments of the same length as the vector and windows each segment using `window`.

If the length of `x` cannot be divided exactly into an integer number of segments 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 segments 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`.

`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 segments. If the segment length is unspecified, the function sets `noverlap` to $\lfloor N_x/4.5 \rfloor$, where N_x is the length of the input signal and the $\lfloor\rfloor$ symbols denote the floor function.

`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 $p = \lceil \log_2 N_w \rceil$, the $\lceil\rceil$ symbols denote the ceiling function, and

- N_w = `window` if `window` is a scalar.
- N_w = `length(window)` if `window` is a vector.

`w` — Normalized frequencies
vector

Normalized frequencies, specified as a vector. `w` must have at least two elements, because otherwise the function interprets it as `nfft`. Normalized frequencies are in rad/sample.

Example: `pi./[2 4]`

`f` — Cyclical frequencies
vector

Cyclical frequencies, specified as a vector. `f` must have at least two elements, because otherwise the function interprets it as `nfft`. The units of `f` are specified by the sample rate, `fs`.

`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, then the sample rate is in Hz.

`freqrange` — 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'`, and specifying `'onesided'` results in an error.

- `'onesided'` — returns the one-sided spectrogram of a real input signal. If `nfft` is even, then `ps` has `nfft`/2 + 1 rows and is computed over the interval [0, π] rad/sample. If `nfft` is odd, then `ps` has (`nfft` + 1)/2 rows and the interval is [0, π) 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. `ps` has `nfft` rows and is computed over the interval [0, 2π) rad/sample. If you specify `fs`, then the interval is [0, `fs`) cycles/unit time.
- `'centered'` — returns the centered two-sided spectrogram of a real or complex signal. `ps` has `nfft` rows. If `nfft` is even, then `ps` is computed over the interval (–π, π] rad/sample. If `nfft` is odd, then `ps` is computed over (–π, π) 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.

`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. If the `'reassigned'` option is on, the function integrates the PSD over the width of each frequency bin before reassigning.

`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.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `spectrogram(x,100,'OutputTimeDimension','downrows')` divides `x` into segments of length 100 and windows each segment with a Hamming window of that length The output of the spectrogram has time dimension down the rows.

`MinThreshold` — Threshold
`-Inf` (default) | real scalar

Threshold, specified as the comma-separated pair consisting of `MinThreshold` and a real scalar expressed in decibels. `spectrogram` sets to zero those elements of `s` such that $10\log_{10}(s) \le$ `thresh`.

`OutputTimeDimension` — Output time dimension
`acrosscolumns` (default) | `downrows`

Output time dimension, specified as the comma-separated pair consisting of `OutputTimeDimension` and `acrosscolumns` or `downrows`. Set this value to `downrows`, if you want the time dimension of `s`,

`ps`, `fc`, and `tc` down the rows and the frequency dimension along the columns. Set this value to `acrosscolumns`, if you want the time dimension of `s`, `ps`, `fc`, and `tc` across the columns and frequency dimension along the rows. This input is ignored if the function is called without output arguments.

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
  - $k = \lfloor (N_x - \texttt{noverlap})/(\texttt{window} - \texttt{noverlap}) \rfloor$ if `window` is a scalar.
  - $k = \lfloor (N_x - \texttt{noverlap})/(\texttt{length(window)} - \texttt{noverlap}) \rfloor$ if `window` is a vector.
- If `x` is real and `nfft` is even, then `s` has (`nfft`/2 + 1) rows.
- If `x` is real and `nfft` is odd, then `s` has (`nfft` + 1)/2 rows.
- If `x` is complex, then `s` has `nfft` rows.

Note When `freqrange` is set to `'onesided'`, `spectrogram` outputs the `s` values in the positive Nyquist range and does not conserve the total power.

`s` is not affected by the `'reassigned'` option.

`w` — Normalized frequencies

vector

Normalized frequencies, returned as a vector. `w` has a length equal to the number of rows of `s`.

`t` — Time instants

vector

Time instants, returned as a vector. The time values in `t` correspond to the midpoint of each segment.

`f` — Cyclical frequencies

vector

Cyclical frequencies, returned as a vector. `f` has a length equal to the number of rows of `s`.

`ps` — Power spectral density or power spectrum

matrix

Power spectral density (PSD) or power spectrum, returned as a matrix.

- If `x` is real and `freqrange` is set to `'onesided'`, then `ps` contains the one-sided modified periodogram estimate of the PSD or power spectrum of each segment. The function multiplies the power by 2 at all frequencies except 0 and the Nyquist frequency to conserve the total power.

- If `x` is complex, if `freqrange` is set to `'twosided'` or `'centered'`, or if you specify a vector of frequencies, then `ps` contains the two-sided modified periodogram estimate of the PSD or power spectrum of each segment.

`fc`, `tc` — Center-of-energy frequencies and times
matrices

Center-of-energy frequencies and times, returned as matrices of the same size as the short-time Fourier transform. If you do not specify a sample rate, then the elements of `fc` are returned as normalized frequencies.

Tips

If a short-time Fourier transform has zeros, its conversion to decibels results in negative infinities that cannot be plotted. To avoid this potential difficulty, `spectrogram` adds `eps` to the short-time Fourier transform when you call it with no output arguments.

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.

[3] Chassande-Motin, Éric, François Auger, and Patrick Flandrin. “Reassignment.” In *Time-Frequency Analysis: Concepts and Methods*. Edited by Franz Hlawatsch and François Auger. London: ISTE/John Wiley and Sons, 2008.

[4] Fulop, Sean A., and Kelly Fitz. “Algorithms for computing the time-corrected instantaneous frequency (reassigned) spectrogram, with applications.” *Journal of the Acoustical Society of America*. Vol. 119, January 2006, pp. 360–371.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Input must be a tall column vector.
- The `window` argument must always be specified.
- `OutputTimeDimension` must be always specified and set to `'downrows'`.
- The `reassigned` option is not supported.
- Syntaxes with no output arguments are not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Arguments specified using name-value pairs must be compile-time constants.
- Variable sized `window` must be double precision.

Thread-Based Environment
Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

Usage notes and limitations:

- The syntax with no output arguments is not supported.
- The frequency vector must be uniformly spaced.

For more information, see “Run MATLAB Functions in Thread-Based Environment”.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Apps
Signal Analyzer

Functions
`goertzel` | `periodogram` | `pspectrum` | `pwelch`

Topics
“Formant Estimation with LPC Coefficients”
“Time-Frequency Gallery”

Introduced before R2006a

spectrum

Spectral estimation

Syntax

```
Hs = spectrum.estmethod(input1,...)
```

Description

Note The use of `spectrum.`*`estmethod`* is not recommended. Use the corresponding function instead. See Spectrum Estimation Methods for the functional forms.

`Hs = spectrum.`*`estmethod`*`(input1,...)` returns a spectral estimation object `Hs` of type *`estmethod`*. This object contains all the parameter information needed for the specified estimation method. Each estimation method takes one or more inputs, which are described on the individual reference pages.

Estimation Methods

Estimation methods for `spectrum` specify the type of spectral estimation method to use. Available estimation methods for `spectrum` are listed below.

Note You must use a spectral *`estmethod`* with `spectrum`.

Spectrum Estimation Methods

spectrum.estmethod	Description	Corresponding Function
`spectrum.burg`	Burg	`pburg`
`spectrum.cov`	Covariance	`pcov`
`spectrum.eigenvector`	Eigenvector	`peig`
`spectrum.mcov`	Modified covariance	`pmcov`
`spectrum.mtm`	Thomson 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 1-2471).

Methods

Methods provide ways of performing functions directly on your `spectrum` object without having to specify the spectral estimation parameters again. You can apply these methods directly on the variable you assigned to your `spectrum` object. For more information on any of these methods, use the syntax `help spectrum/`*`method`* at the MATLAB prompt or refer to the table below.

Spectrum Methods

Method	Description

`msspectrum`	Note that the `msspectrum` method is only available for the `periodogram` and `welch` spectrum estimation objects.
	
	The mean-squared spectrum is intended for discrete spectra (from periodic, discrete-time signals). The distribution of the mean square value across frequency is the msspectrum. Unlike the power spectral density (see `psd` below), the peaks in the mean-square spectrum reflect the power in the signal at a given frequency. For the PSD, the power is reflected as the area in a frequency band. The units of the mean-squared spectrum are units of power.
	
	`Hmss = msspectrum(Hs,X)` returns a mean-square spectrum object containing the mean-square (power) estimate of the discrete-time signal `X` using the spectrum object `Hs`. Default for real `X` is the `'onesided'` Nyquist frequency range and for complex `X` the default is the `'twosided'` Nyquist frequency range.
	
	`Hmss` contains a vector of normalized frequencies `W`, at which the mean-square spectrum is estimated. For real signals, the range of `W` is [0,π] if the number of FFT points (`NFFT`) is even, and [0,π) if `NFFT` is odd. For complex signals, the range of `W` is [0,2π). To estimate the spectrum on a vector of specific frequencies, see `FreqPoints` property below.
	
	The `msspectrum` method includes these properties, which you can set using this `msspectrum` method or via the `msspectrumopts` method. These properties are listed here and described in the `msspectrumopts` section below:
	`SpectrumType` — `'onesided'` or `'twosided'`
	`NormalizedFrequency` – normalizes frequency between 0 and 1
	`Fs` — sampling frequency in Hz
	`NFFT` — number of FFT points
	`CenterDC` — shifts data and frequencies to center DC component
	`FreqPoints` — `'All'` or `'User Defined'`
	`FrequencyVector` — frequencies at which to compute spectrum
	`ConfLevel` — confidence level to calculate the confidence interval. Value must be from 0 to 1.
	
	For example, `Hmss = msspectrum(Hs,X,'FreqPoints','User Defined', FreqVector,fvect)` returns a mean-square spectrum object where the spectrum is calculated only on the frequency points defined in the frequency vector, `fvect`.
	
	`msspectrum(...)` with no output arguments plots the mean-square spectrum in dB.

Method	Description
msspectrumopts	Hopts = msspectrumopts(Hs) returns an object that contains options for the spectrum object Hs. Hopts = msspectrumopts(Hs,X) returns an object with data-specific options and defaults. You can pass an Hopts options object as an argument to the msspectrum method. Any individual option you specify after the Hopts object overrides the value in Hopts. For example, Hmss = msspectrum(Hs,X,Hopts, 'SpectrumType', 'twosided') overrides the default SpectrumType value in Hopts. The following properties apply to both msspectrumopts and msspectrum methods. Hmss = msspectrum (..., 'SpectrumType', 'twosided') returns the two-sided mean-square spectrum. The spectrum length (NFFT) is computed over [0,2π), or if Fs is specified, [0,Fs) . Entering 'onesided' returns the one-sided mean-square spectrum, which contains the total signal power in half the Nyquist range. Default is 'onesided'. Hmss = msspectrum(Hs,X,'NormalizedFrequency',true) returns a mean-square spectrum object with frequency values normalized between 0 and 1. Default is true. Hmss = msspectrum(Hs,X,'Fs',Fs) returns a mean-square spectrum object computed as a function of frequency, where Fs is the sampling frequency in Hz. Note that you can set Fs only if NormalizedFrequency is set to false. Hmss = msspectrum(...,'NFFT',nfft) specifies the number of FFT points to use. Valid values are a positive integer, 'Nextpow2' or 'Auto'. 'Nextpow2' uses the next power of 2 greater than the input length or 256, whichever is greater. 'Auto' uses the input length or 256, whichever is greater. Default is 'Nextpow2'. Note that for spectrum.welch, 'Nextpow2' and 'Auto' are compared to the SegmentLength instead of the input length. Hmss = msspectrum (..., 'Centerdc', true) shifts the data and frequency values so that the DC component is at the center of the spectrum. Default is false. To estimate the spectrum on a vector of specific frequencies, first set the number of frequency points to 'User Defined', which replaces the NFFT property of msspectrum with a FrequencyVector property. Hopts.FreqPoints = 'User Defined' (Note that the default for FreqPoints is 'All', which causes msspectrum to use the NFFT property as described above.) Then, specify the frequency vector F to use. Hopts.FrequencyVector = F (Note that the default value for FrequencyVector is 'Auto'. In this case, the number of frequency points used follows the same rule as described for NFFT 'Auto' above.) Hmms = msspectrum(...,'ConfLevel',p) specifies the confidence level p for computing the confidence interval, which is an estimate of the error in the calculated

Method	Description
	mean-squared spectrum. The confidence level (`p`) is between 0 and 1. For example, `Hmss = msspectrum(Hs,X,'ConfLevel',0.95)` returns the 95% confidence interval.

`psd`	Note that `music` and `eigenvector` spectrum objects do not support the `psd` method. See the `pseudospectrum` method below.
	
	The power spectral density (PSD) is intended for continuous spectra. The integral of the PSD over a given frequency band computes the average power in the signal in that frequency band. In contrast to the `msspectrum`, the peaks in this spectra do not reflect the power at a given frequency. The units of the PSD are power per unit of frequency. See the `avgpower` method of `dspdata` for more information.
	
	`Hpsd = psd (Hs,X)` returns a power spectral density object containing the power spectral density estimate of the discrete-time signal `X` using the spectrum object `Hs`. The PSD is the distribution of power per unit frequency. Default for real `X` is `'onesided'` and for complex `X` is `'twosided'`.
	
	`Hpsd` contains a vector of normalized frequencies `W`, at which the PSD is estimated. For real signals, the range of `W` is [0,π] if the number of FFT points (`NFFT`) is even, and [0,π) if `NFFT` is odd. For complex signals, the range of `W` is [0,2π).
	
	The `psd` method includes these properties, which you can set using this `psd` method or via the `psdopts` method. These properties are listed here and described in the `psdopts` section below:
	`SpectrumType` — `'onesided'` or `'twosided'`
	`NormalizedFrequency` — normalizes frequency between 0 and 1
	`Fs` — sampling frequency in Hz
	`NFFT` — number of FFT points
	`CenterDC` — shifts data and frequencies to center DC component
	`FreqPoints` — `'All'` or `'User Defined'`
	`FrequencyVector` – frequencies at which to compute spectrum
	`ConfLevel` — confidence level to calculate the confidence interval. Value must be from 0 to 1.
	
	For example, `Hmss = psd(Hs,X,'FreqPoints','User Defined', FreqVector,fvect)` returns a PSD object where the spectrum is calculated only on the frequency points defined in the frequency vector, `fvect`.
	
	`psd(...)` with no output arguments plots PSD in dB per unit frequency.

Method	Description
psdopts	`Hopts = psdopts(Hs)` returns an object that contains options for the spectrum object `Hs`. `Hopts = psdopts(Hs,X)` returns an object with data-specific options and defaults. You can pass an `Hopts` options object as an argument to the `psd` method. Any individual option you specify after the `Hopts` object overrides the value in `Hopts`. For example, `Hpsd = psd(Hs,X,Hopts,'SpectrumType', 'twosided')` overrides the `SpectrumType` value in `Hopts`. The following properties apply to both `psdmopts` and `psd` methods. `Hpsd = psd (Hs,X,'SpectrumType','twosided')` returns the two-sided power spectral density of `X`. The spectrum length is NFFT and is computed over [0,2π) if `Fs` is not specified or [0,`Fs`) if `Fs` is specified. Entering `'onesided'` returns the one-sided PSD, which contains the total signal power. `Hmss = psd(Hs,X,'NormalizedFrequency',true)` returns a power spectral density object with frequency values normalized between 0 and 1. Default is `true`. `Hpsd = psd (...,'Fs',Fs)` returns a power spectral density object computed as a function of frequency, where `Fs` is the sampling frequency in Hz. `Hmss = psd(...,'NFFT',nfft)` specifies the number of FFT points to use. Valid values are a positive integer, `'Nextpow2'` or `'Auto'`. `'Nextpow2'` uses the next power of 2 greater than the input length or 256, whichever is greater. `'Auto'` uses the input length or 256, whichever is greater. Default is `'Nextpow2'`. Note that for `spectrum.welch`, `'Nextpow2'` and `'Auto'` are compared to the `SegmentLength` instead of the input length. `Hmss = psd (..., 'Centerdc', true)` shifts the data and frequency values so that the DC component is at the center of the spectrum. Default is `false`. To estimate the spectrum on a vector of specific frequencies, first set the number of frequency points to `'User Defined'`, which replaces the `NFFT` property of `psd` with a `FrequencyVector` property. `Hopts.FreqPoints = 'User Defined'` (Note that the default for `FreqPoints` is `'All'` which causes `psd` to use the `NFFT` property as described above.) `Hmms = psd(...,'ConfLevel',p)` specifies the confidence level `p` for computing the confidence interval, which is an estimate of the error in the calculated PSD. The confidence level (`p`) is between 0 and 1. For example, `Hmss = psd(Hs,X,'ConfLevel',0.95)` returns the 95% confidence interval.

Method	Description

`pseudospectrum`	Note that this method is used for only `music` or `eigenvector` spectrum objects.
	
	`Hps = pseudospectrum(Hs,X)` returns an object containing the pseudospectrum estimate of the discrete-time signal `X` using the spectrum object `Hs`. `Hs` must be a `music` or `eigenvector` object. Default for real `X` is `'half'` and for complex `X` is the `'whole'` Nyquist frequency range.
	
	`Hps` contains a vector of normalized frequencies `W`, at which the pseudospectrum is estimated. For real signals, the range of `W` is [0,π] if the number of FFT points (`NFFT`) is even, and [0,π) if `NFFT` is odd. For complex signals, the range of `W` is [0,2π).
	
	The `pseudospectrum` method includes these properties, which you can set using this `pseudospectrum` method or via the `pseudospectrumopts` method. These properties are described below:
	`SpectrumRange` — `'half'` or `'whole'`
	`NormalizedFrequency` — normalizes frequency between 0 and 1
	`Fs` — sampling frequency in Hz
	`NFFT` — number of FFT points
	`CenterDC` — shifts data and frequencies to center DC component
	`FreqPoints` — `'All'` or `'User Defined'`
	`FrequencyVector` — frequencies at which to compute spectrum
	
	For example, `Hmss = psd(Hs,X,'FreqPoints','User Defined', FreqVector,fvect)` returns a PSD object where the spectrum is calculated only on the frequency points defined in the frequency vector, `fvect`.
	
	`pseudospectrum(...)` with no output arguments plots the pseudospectrum in dB.

Method	Description
pseudospectrumopts	`Hopts = pseudospectrumopts(Hs)` returns an object that contains options for the spectrum object `Hs`. `Hopts = pseudospectrumopts(Hs,X)` returns an object with data-specific options and defaults. You can pass an `Hopts` options object as an argument to the `pseudospectrum` method. Any individual option you specify after the `Hopts` object overrides the value in `Hopts`. For example, `Hpseudospectrum= pseudospectrum(Hs,X, Hopts,'SpectrumRange', 'whole')` overrides the `SpectrumRange` value in `Hopts`. `Hmps = pseudospectrum (..., 'SpectrumRange', 'whole')` returns the pseudospectrum over the whole Nyquist range. The spectrum length is NFFT and is computed over [0,2π) if `Fs` is not specified or [0,`Fs`) if `Fs` is specified. Entering `'half'` returns the pseudospectrum calculated over half the Nyquist range. `Hmss = pseudospectrum(Hs,X,'NormalizedFrequency',true)` returns a pseudospectrum object with frequency values normalized between 0 and 1. Default is `true`. `Hps = pseudospectrum(Hs,X,'Fs',Fs)` returns a pseudospectrum object computed as a function of frequency, where `Fs` is the sampling frequency in Hz. `Hps = pseudospectrum(...,'NFFT',nfft)` specifies the number of FFT points to use. Valid values are a positive integer, `'Nextpow2'` or `'Auto'`. `'Nextpow2'` uses the next power of 2 greater than the input length or 256, whichever is greater. `'Auto'` uses the input length or 256, whichever is greater. Default is `'Nextpow2'`. `Hps = pseudospectrum(...,'Centerdc',true)` shifts the data and frequency values so that the DC component is at the center of the spectrum. The default value is `false`. To estimate the spectrum on a vector of specific frequencies, first set the number of frequency points to `'User Defined'`, which replaces the `NFFT` property of `pseudospectrum` with a `FrequencyVector` property. `Hopts.FreqPoints = 'User Defined'` (Note that the default for `FreqPoints` is `'All'`, which causes `pseudospectrum` to use the `NFFT` property as described above.)
powerest	Note that `powerest` is available only for `music` and `eigenvector` spectrum objects. `POW = powerest(Hs,X)` returns a vector `POW` containing estimates of the powers of the complex sinusoids in `X`. The input `X` can be a vector or a matrix. If it is a matrix it can be a data matrix, where $X'*X = R$ or a correlation matrix R. The value the `InputType` property of `Hs` determines how `X` is interpreted. `Hs` must be a `music` or `eigenvector` spectrum object. `[POW,W]=powerest(Hs,X)` returns `POW` and a vector `W` of the frequencies in rad/sample of the sinusoids in `X`. `[POW,F]=powerest(Hs,X,Fs)` returns `POW` and a vector `F` of the frequencies in Hz of the sinusoids in `X`. `Fs` is the sampling frequency.

Viewing Object Properties

As with any object, you can use `get` to view a `spectrum` object's properties. To see a specific property, use

```
get(Hs,'property')
```

where `'property'` is the specific property name.

To see all properties for an object, use

```
get(Hs)
```

Changing Object Properties

To set specific properties, use

```
set(Hs,'property1',value, 'property2',value,...)
```

where `'property1'`, `'property2'`, etc. are the specific property names.

To view the options for a property use `set` without specifying a value

```
set(Hs,'property')
```

Note that you must use single quotation marks around the property name. For example, to change the order of a Burg `spectrum` object `Hs` to 6, use

```
set(Hs,'order',6)
```

Another example of using `set` to change an object's properties is this example of changing the dynamically created window property of a periodogram `spectrum` object.

```
Hs=spectrum.periodogram       % Create periodogram object

Hs =

    EstimationMethod: 'Periodogram'
          WindowName: 'Rectangular'

set(Hs,'WindowName','Chebyshev')   % Change window type
Hs                                 % View changed object

Hs =

    EstimationMethod: 'Periodogram'
          WindowName: 'Chebyshev'  % Note changed property
       SidelobeAtten: 100

set(Hs,'SidelobeAtten',150)   % Change dynamic property
Hs                            % View changed object

Hs =

    EstimationMethod: 'Periodogram'
          WindowName: 'Chebyshev'
       SidelobeAtten: 150
```

All `spectrum` object properties can be changed using the `set` command, except for the `EstimationMethod` property.

Another way to change an object's properties is by using the `inspect` command which opens the Property Inspector window where you can edit any property, except dynamic properties, such as those used with windows.

```
inspect(Hs)
```

Copying an Object

To create a copy of an object, use the `copy` method.

```
H2 = copy(Hs)
```

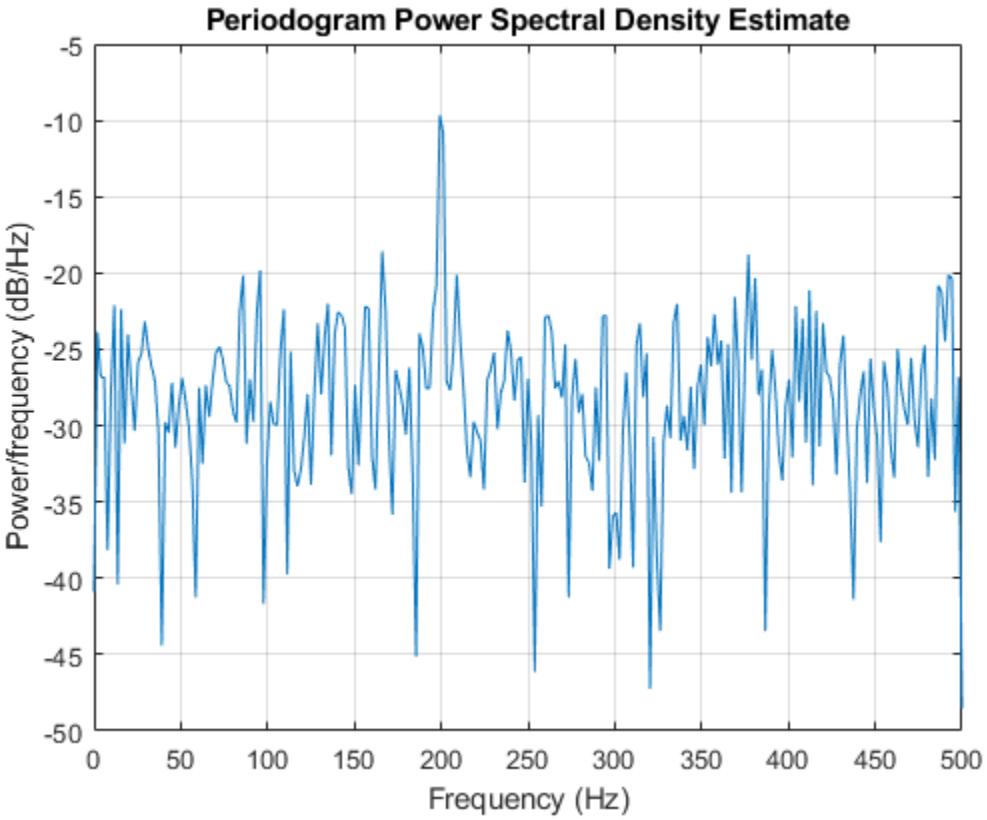
Note Using the syntax `H2 = Hs` copies only the object handle and does not create a new object.

Examples

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.

Introduced before R2006a

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`

Introduced before R2006a

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`

Introduced before R2006a

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,SubspaceThreshold)
Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold,InputType)
```

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 or `'User Defined'` (see `window` for valid window names). For more information on each window, refer to its reference page. This argument can also be a cell array containing the window name or `'User Defined'` and, if used for the particular window, an optional parameter value. The syntax is `{wname,wparam}`. You can use `set` to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window (see `spectrum` for information on using `set`).
`SubspaceThreshold`	`0`	Threshold is the cutoff for signal and noise separation. The threshold is multiplied by λ_{min} , the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold (λ_{min}`*threshold`) are assigned to the noise subspace.
`InputType`	`'Vector'`	Type of input that will be used with this spectrum object. Valid values are `'Vector'`, `'DataMatrix'` and `'CorrelationMatrix'`.

`Hs = spectrum.eigenvector(NSinusoids)` returns a spectrum object, `Hs`, with the specified number of sinusoids and default values for all other properties. Refer to the table above for default values.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength)` returns a spectrum object, `Hs`, with the specified segment length.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...`
`OverlapPercent)` returns a spectrum object, `Hs`, with the specified overlap between segments.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...`
`OverlapPercent,WindowName)` returns a spectrum object, `Hs`, with the specified window.

Note Window names must be enclosed in single quotes, such as `spectrum.eigenvector(3,32,50,'chebyshev')` or `spectrum.eigenvector(3,32,50,{'chebyshev',60})`.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold)` returns a spectrum object, `Hs`, with the specified subspace threshold.

`Hs = spectrum.eigenvector(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold,InputType)` returns a spectrum object, `Hs`, with the specified input type.

Note See `peig` for more information on the eigenanalysis algorithm.

Examples

Define a complex signal with three sinusoids, add noise, and view its pseudospectrum using eigenanalysis. Set the FFT length to 128.

```
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`

Introduced before R2006a

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`

Introduced before R2006a

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.

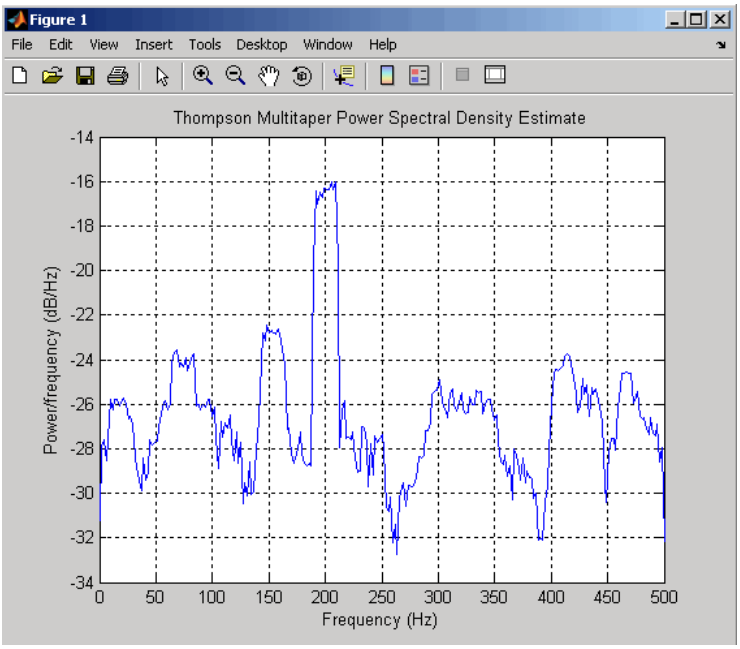
Note You can either specify the time-bandwidth product (`TimeBW`) or the `DPSS` data tapers and their `Concentrations`. See `dpss` and `pmtm` for more information.

`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`

Introduced before R2006a

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,SubspaceThreshold)
Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold,InputType)
```

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 or `'User Defined'` (see `window` for valid window names). For more information on each window, refer to its reference page). This argument can also be a cell array containing the window name or `'User Defined'` and, if used for the particular window, an optional parameter value. The syntax is `{wname,wparam}`. You can use `set` to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window (see `spectrum` for information on using `set`).
`SubspaceThreshold`	`0`	Threshold is the cutoff for signal and noise separation. The threshold is multiplied by λ_{min} , the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold (λ_{min}*`threshold`) are assigned to the noise subspace.
`InputType`	`'Vector'`	Type of input that will be used with this spectrum object. Valid values are `'Vector'`, `'DataMatrix'` and `'CorrelationMatrix'`.

`Hs = spectrum.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,SubspaceThreshold)` returns a spectrum object, `Hs`, with the specified subspace threshold.

`Hs = spectrum.music(NSinusoids,SegmentLength,...
OverlapPercent,WindowName,SubspaceThreshold,InputType)` returns a spectrum object, `Hs`, with the specified input type.

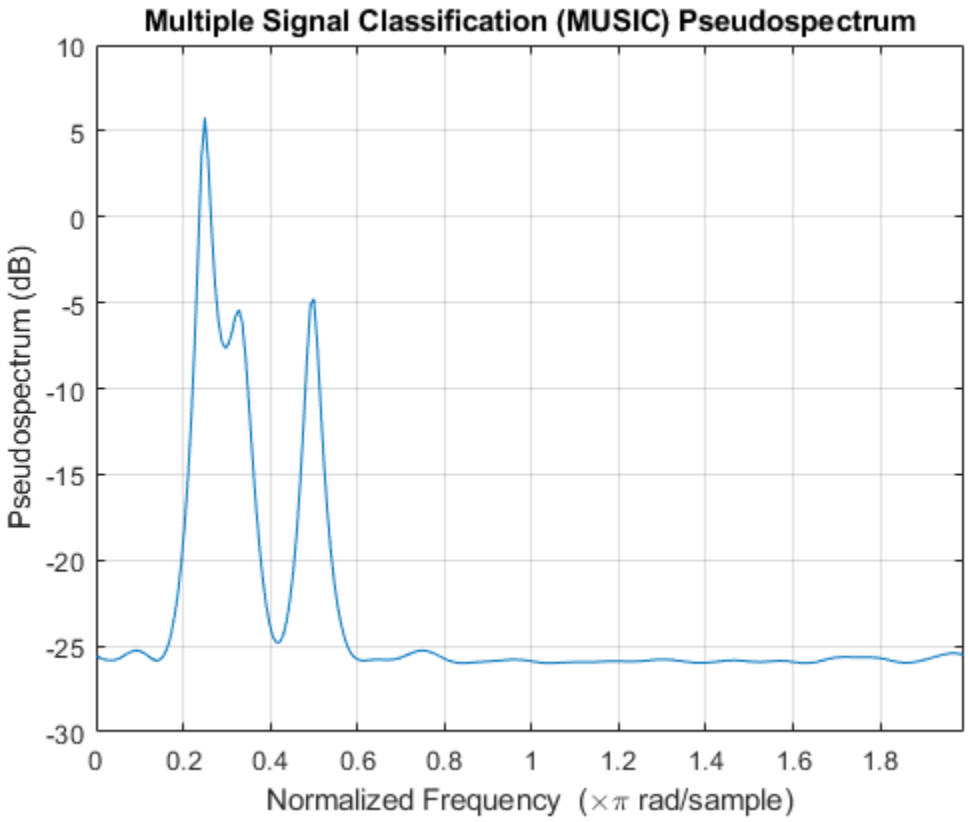
Note See `pmusic` for more information on the MUSIC algorithm.

Examples

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`

Introduced before R2006a

spectrum.periodogram

Periodogram spectrum

Syntax

```
Hs = spectrum.periodogram
Hs = spectrum.periodogram(winname)
Hs = spectrum.periodogram({windowname,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({windowname,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 the window name and the parameter value. This object uses the default FFT length.

Valid window names are:

```
'Bartlett'
'Bartlett-Hann'
'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})`.

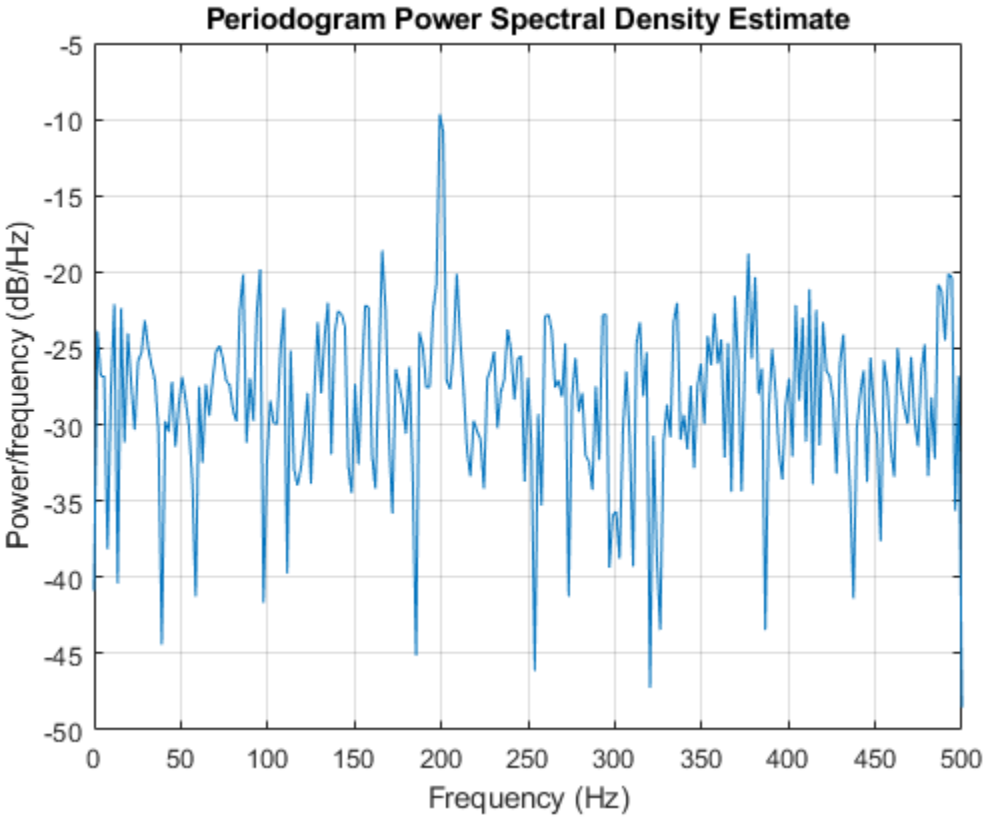
Note See `periodogram` for more information on the periodogram algorithm.

Examples

Periodogram Spectral Estimate of Sinusoid

Define a cosine of 200 Hz sampled at 1 kHz. Add noise and view the spectral content of the signal using the periodogram spectral estimation technique with default values.

```
Fs = 1000;
t = 0:1/Fs:.3;
x = cos(2*pi*t*200)+randn(size(t));
Hs = spectrum.periodogram;
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`

Introduced before R2006a

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}` Cell array containing `WindowName` and optional window parameter	`'Hamming'`, `SamplingFlag: symmetric`	Cell array containing the window name or `'User Defined'` and, if used for the particular window, an optional parameter value. (See `window` for valid window names and for more information on each window, refer to its reference page.) You can use `set` to change the value of the additional parameter or to define the MATLAB expression and parameters for a user-defined window. (See `spectrum` for information on using `set`.)

Property Name	Default Value	Description

`WindowName`	`'Hamming'`,
	
	`SamplingFlag: symmetric`
		Valid `windowname` options are:
		
		`'Bartlett'`
		`'Bartlett-Hann'`
		`'Blackman'`
		`'Blackman-Harris'`
		`'Bohman'`
		`'Chebyshev'`
		`'Flat Top'`
		`'Gaussian'`
		`'Hamming'`
		`'Hann'`
		`'Kaiser'`
		`'Nuttall'`
		`'Parzen'`
		`'Rectangular'`
		`'Triangular'`
		`'Tukey'`
		`'User Defined'`
		
		Window names must be enclosed in single quotes, such as `spectrum.welch('tukey') or spectrum.welch({'tukey',0.7})`.
		
		See `window` and the corresponding window function page for window parameter information. You can use `set` to change the value of the additional window parameter or to define the MATLAB expression and parameters for a user-defined window (see `spectrum` for information on using `set`).

`SegmentLength`	`64`	Length of each of the time-based segments into which the input signal is divided. A modified periodogram is computed on each segment and the average of the periodograms forms the spectral estimate. Choosing the segment length is a compromise between estimate reliability (shorter segments) and frequency resolution (longer segments). A long segment length 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

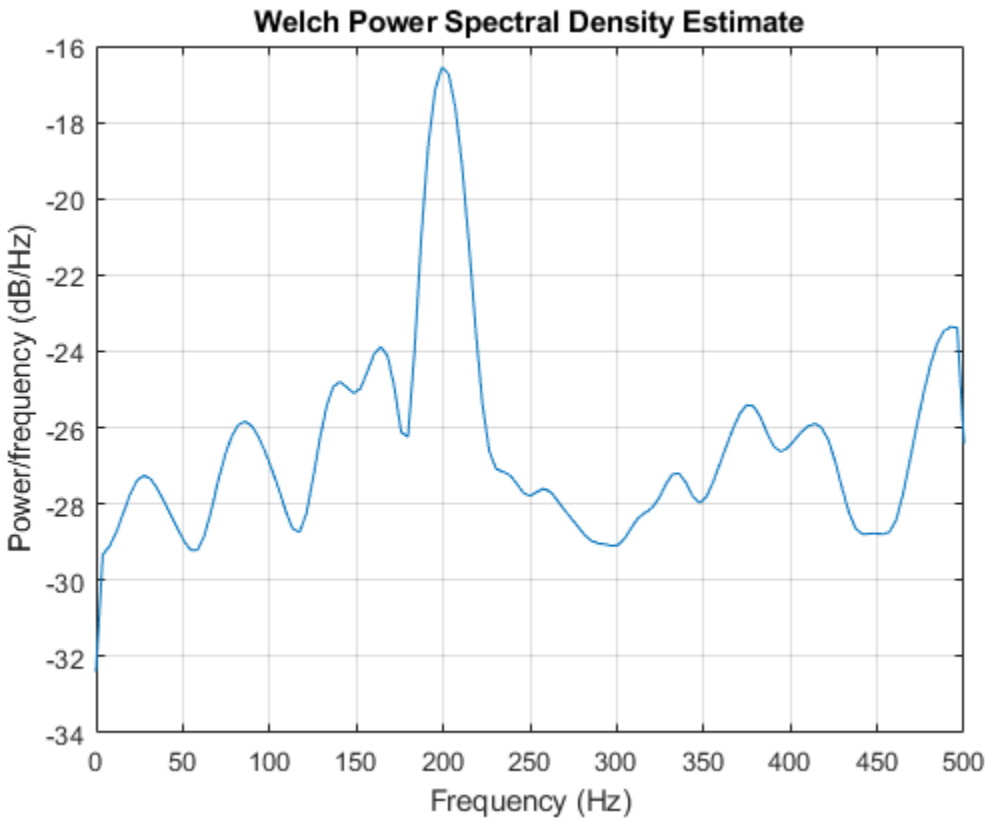
Spectral Content of Sinusoid

Define a cosine of 200 Hz embedded in white noise.

```
Fs = 1000;
t = 0:1/Fs:.3;
x = cos(2*pi*t*200)+randn(size(t));
```

View the spectral content of the signal using the Welch algorithm.

```
Hs = spectrum.welch;
psd(Hs,x,'Fs',Fs)
```



PSD Estimate Using Hann Window

Define a cosine of 200 Hz embedded in white noise.

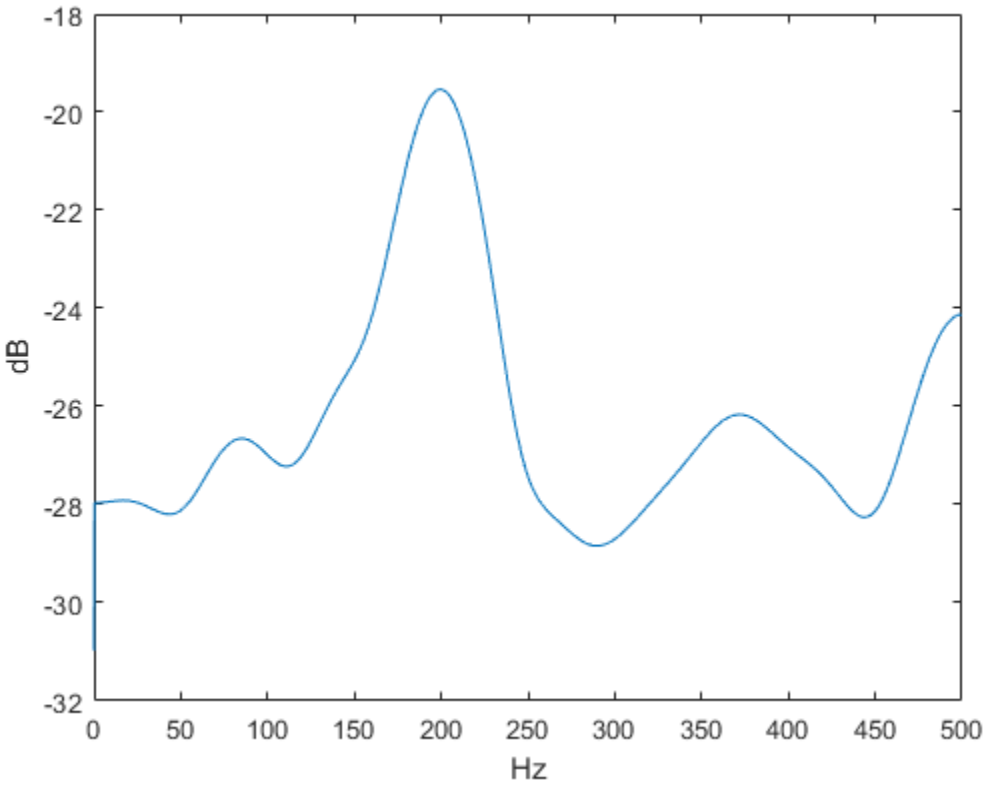
```
Fs = 1000;
t = 0:1/Fs:0.3;
x = cos(2*pi*t*200)+randn(size(t));
```

Compute Welch's power spectral density estimate of the signal using a Hann window.

```
window = 33;
noverlap = 32;
nfft = 4097;

h = spectrum.welch('Hann',window,100*noverlap/window);
hpsd = psd(h,x,'NFFT',nfft,'Fs',Fs);
```

Visualize the power spectral density expressed in decibels.

```
Pw = hpsd.Data;
Fw = hpsd.Frequencies;
plot(Fw,pow2db(Pw))
xlabel('Hz')
ylabel('dB')
```



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).

See Also

`periodogram` | `pmtm` | `pwelch`

Introduced before R2006a

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`

Introduced before R2006a

splitlabels

Find indices to split labels according to specified proportions

Syntax

```
idxs = splitlabels(lblsrc,p)
idxs = splitlabels(lblsrc,p,'randomized')
idxs = splitlabels( ___ ,Name,Value)
```

Description

Use this function when you are working on a machine or deep learning classification problem and you want to split a dataset into training, testing, and validation sets that hold the same proportion of label values.

`idxs = splitlabels(lblsrc,p)` finds logical indices that split the labels in `lblsrc` based on the proportions or number of labels specified in `p`.

`idxs = splitlabels(lblsrc,p,'randomized')` randomly assigns the specified proportion of label values to each index set in `idxs`.

`idxs = splitlabels( ___ ,Name,Value)` specifies additional input arguments using name-value pairs. For example, `'UnderlyingDatastoreIndex',3` splits the labels only in the third underlying datastore of a combined datastore.

Examples

Split Vowels

Read William Shakespeare's sonnets with the `fileread` function. Extract all the vowels from the text and convert them to lowercase.

```
sonnets = fileread("sonnets.txt");
vowels = lower(sonnets(regexp(sonnets,"[AEIOUaeiou]")))';
```

Count the number of instances of each vowel.

```
cnts = countlabels(vowels)
```

```
cnts=5×3 table
    Label    Count    Percent
    _____    _____    _______

      a      4940     18.368
      e      9028     33.569
      i      4895     18.201
      o      5710     21.232
      u      2321     8.6302
```

Split the vowels into a training set containing 500 instances of each vowel, a validation set containing 300, and a testing set with the rest. All vowels are represented with equal weights in the first two sets but not in the third.

```
spltn = splitlabels(vowels,[500 300]);

for kj = 1:length(spltn)
    cntsn{kj} = countlabels(vowels(spltn{kj}));
end
cntsn{:}
```

```
ans=5×3 table
    Label    Count    Percent
    _____    _____    _______

      a        500       20
      e        500       20
      i        500       20
      o        500       20
      u        500       20


ans=5×3 table
    Label    Count    Percent
    _____    _____    _______

      a        300       20
      e        300       20
      i        300       20
      o        300       20
      u        300       20


ans=5×3 table
    Label    Count    Percent
    _____    _____    _______

      a       4140     18.083
      e       8228      35.94
      i       4095     17.887
      o       4910     21.447
      u       1521     6.6437
```

Split the vowels into a training set containing 50% of the instances, a validation set containing another 30%, and a testing set with the rest. All vowels are represented with the same weight across all three sets.

```
spltp = splitlabels(vowels,[0.5 0.3]);

for kj = 1:length(spltp)
    cntsp{kj} = countlabels(vowels(spltp{kj}));
end
cntsp{:}
```

```
ans=5×3 table
    Label    Count    Percent
    _____    _____    _______
```

```
      a      2470     18.367
      e      4514     33.566
      i      2448     18.203
      o      2855      21.23
      u      1161     8.6333


ans=5×3 table
    Label    Count    Percent
    _____    _____    _______

      a      1482     18.371
      e      2708     33.569
      i      1468     18.198
      o      1713     21.235
      u       696     8.6277


ans=5×3 table
    Label    Count    Percent
    _____    _____    _______

      a       988     18.368
      e      1806     33.575
      i       979       18.2
      o      1142     21.231
      u       464     8.6261
```

Split Vowels and Consonants

Read William Shakespeare's sonnets with the `fileread` function. Remove all nonalphabetic characters from the text and convert to lowercase.

```
sonnets = fileread("sonnets.txt");
letters = lower(sonnets(regexp(sonnets,"[A-z]")))';
```

Classify the letters as consonants or vowels and create a table with the results. Show the first few rows of the table.

```
type = repmat("consonant",size(letters));
type(regexp(letters','[aeiou]")) = "vowel";

T = table(letters,type,'VariableNames',["Letter" "Type"]);
head(T)
```

```
ans=8×2 table
    Letter       Type
    ______    __________

      t       "consonant"
      h       "consonant"
      e       "vowel"
      s       "consonant"
      o       "vowel"
```

```
      n       "consonant"
      n       "consonant"
      e       "vowel"
```

Display the number of instances of each category.

```
cnt = countlabels(T,'TableVariable',"Type")
```

```
cnt=2×3 table
      Type       Count    Percent
    _________    _____    _______

    consonant    46516    63.365
    vowel        26894    36.635
```

Split the table into two sets, one containing 60% of the consonants and vowels and the other containing 40%. Display the number of instances of each category.

```
splt = splitlabels(T,0.6,'TableVariable',"Type");

sixty = countlabels(T(splt{1},:),'TableVariable',"Type")
```

```
sixty=2×3 table
      Type       Count    Percent
    _________    _____    _______

    consonant    27910    63.366
    vowel        16136    36.634
```

```
forty = countlabels(T(splt{2},:),'TableVariable',"Type")
```

```
forty=2×3 table
      Type       Count    Percent
    _________    _____    _______

    consonant    18606    63.363
    vowel        10758    36.637
```

Split the table into two sets, one containing 60% of each particular letter and the other containing 40%. Exclude the letter *y*, which sometimes acts as a consonant and sometimes as a vowel. Display the number of instances of each category.

```
splt = splitlabels(T,0.6,'Exclude',"y");

sixti = countlabels(T(splt{1},:),'TableVariable',"Type")
```

```
sixti=2×3 table
      Type       Count    Percent
    _________    _____    _______

    consonant    26719    62.346
    vowel        16137    37.654
```

```
forti = countlabels(T(splt{2},:),'TableVariable',"Type")
```

```
forti=2×3 table
      Type       Count    Percent
    _________    _____    _______

    consonant    17813    62.349
    vowel        10757    37.651
```

Split the table into two sets of the same size. Include only the letters *e* and *s*. Randomize the sets.

```
halves = splitlabels(T,0.5,'randomized','Include',["e" "s"]);

cnt = countlabels(T(halves{1},:))
```

```
cnt=2×3 table
    Letter    Count    Percent
    ______    _____    _______

      e       4514     64.385
      s       2497     35.615
```

Split Data in Datastore

Create a dataset that consists of 100 Gaussian random numbers. Label 40 of the numbers as `A`, 30 as `B`, and 30 as `C`. Store the data in a combined datastore containing two datastores. The first datastore has the data and the second datastore contains the labels.

```
dsData = arrayDatastore(randn(100,1));
dsLabels = arrayDatastore([repmat("A",40,1); repmat("B",30,1); repmat("C",30,1)]);
dsDataset = combine(dsData,dsLabels);
cnt = countlabels(dsDataset,'UnderlyingDatastoreIndex',2)
```

```
cnt=3×3 table
    Label    Count    Percent
    _____    _____    _______

      A       40        40
      B       30        30
      C       30        30
```

Split the data set into two sets, one containing 60% of the numbers and the other with the rest.

```
splitIndices = splitlabels(dsDataset,0.6,'UnderlyingDatastoreIndex',2);

dsDataset1 = subset(dsDataset,splitIndices{1});
cnt1 = countlabels(dsDataset1,'UnderlyingDatastoreIndex',2)
```

```
cnt1=3×3 table
    Label    Count    Percent
    _____    _____    _______

      A       24        40
      B       18        30
```

```
      C       18        30

dsDataset2 = subset(dsDataset,splitIndices{2});
cnt2 = countlabels(dsDataset2,'UnderlyingDatastoreIndex',2)

cnt2=3×3 table
    Label    Count    Percent
    _____    _____    _______

      A       16        40
      B       12        30
      C       12        30
```

Input Arguments

`lblsrc` — Input label source

categorical vector | string vector | logical vector | numeric vector | cell array | table | datastore | `CombinedDatastore` object

Input label source, specified as one of these:

- A categorical vector.
- A string vector or a cell array of character vectors.
- A numeric vector or a cell array of numeric scalars.
- A logical vector or a cell array of logical scalars.
- A table with variables containing any of the previous data types.
- A datastore whose `readall` function returns any of the previous data types.
- A `CombinedDatastore` object containing an underlying datastore whose `readall` function returns any of the previous data types. In this case, you must specify the index of the underlying datastore that has the label values.

`lblsrc` must contain labels that can be converted to a vector with a discrete set of categories.

Example: `lblsrc = categorical(["B" "C" "A" "E" "B" "A" "A" "B" "C" "A"],["A" "B" "C" "D"])` creates the label source as a ten-sample categorical vector with four categories: `A`, `B`, `C`, and `D`.

Example: `lblsrc = [0 7 2 5 11 17 15 7 7 11]` creates the label source as a ten-sample numeric vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `table` | `cell` | `categorical`

`p` — Proportions or numbers of labels

integer scalar | scalar in (0, 1) | vector of integers | vector of fractions

Proportions or numbers of labels, specified as an integer scalar, a scalar in the range (0, 1), a vector of integers, or a vector of fractions.

- If `p` is a scalar, `splitlabels` finds two splitting index sets and returns a two-element cell array in `idxs`.

  - If `p` is an integer, the first element of `idxs` contains a vector of indices pointing to the first `p` values of each label category. The second element of `idxs` contains indices pointing to the remaining values of each label category.
  - If `p` is a value in the range (0, 1) and `lblsrc` has K_i elements in the ith category, the first element of `idxs` contains a vector of indices pointing to the first `p` × K_i values of each label category. The second element of `idxs` contains the indices of the remaining values of each label category.
- If `p` is a vector with N elements of the form p_1, p_2, ..., p_N, `splitlabels` finds N + 1 splitting index sets and returns an (N + 1)-element cell array in `idxs`.
  - If `p` is a vector of integers, the first element of `idxs` is a vector of indices pointing to the first p_1 values of each label category, the next element of `idxs` contains the next p_2 values of each label category, and so on. The last element in `idxs` contains the remaining indices of each label category.
  - If `p` is a vector of fractions and `lblsrc` has K_i elements of the ith category, the first element of `idxs` is a vector of indices concatenating the first p_1 × K_i values of each category, the next element of `idxs` contains the next p_2 × K_i values of each label category, and so on. The last element in `idxs` contains the remaining indices of each label category.

Note

- If `p` contains fractions, then the sum of its elements must not be greater than one.
- If `p` contains numbers of label values, then the sum of its elements must not be greater than the smallest number of labels available for any of the label categories.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'TableVariable',"AreaCode",'Exclude',["617" "508"]` specifies that the function split labels based on telephone area code and exclude numbers from Boston and Natick.

`Include` — Labels to include in index sets

vector of label categories | cell array of label categories

Labels to include in the index sets, specified as a vector or cell array of label categories. The categories specified with this argument must be of the same type as the labels in `lblsrc`. Each category in the vector or cell array must match one of the label categories in `lblsrc`.

`Exclude` — Labels to exclude from index sets

vector of label categories | cell array of label categories

Labels to exclude from the index sets, specified as a vector or cell array of label categories. The categories specified with this argument must be of the same type as the labels in `lblsrc`. Each category in the vector or cell array must match one of the label categories in `lblsrc`.

`TableVariable` — Table variable to read
first table variable (default) | character vector | string scalar

Table variable to read, specified as a character vector or string scalar. If this argument is not specified, then `splitlabels` uses the first table variable.

`UnderlyingDatastoreIndex` — Underlying datastore index
integer scalar

Underlying datastore index, specified as an integer scalar. This argument applies when `lblsrc` is a `CombinedDatastore` object. `splitlabels` counts the labels in the datastore obtained using the `UnderlyingDatastores` property of `lblsrc`.

Output Arguments

`idxs` — Splitting indices
cell array

Splitting indices, returned as a cell array.

See Also

Signal Labeler | `labeledSignalSet` | `signalLabelDefinition` | `countlabels` | `folders2labels`

Introduced in R2021a

sptool

(Removed) Open interactive digital signal processing tool

Note SPTool has been removed.

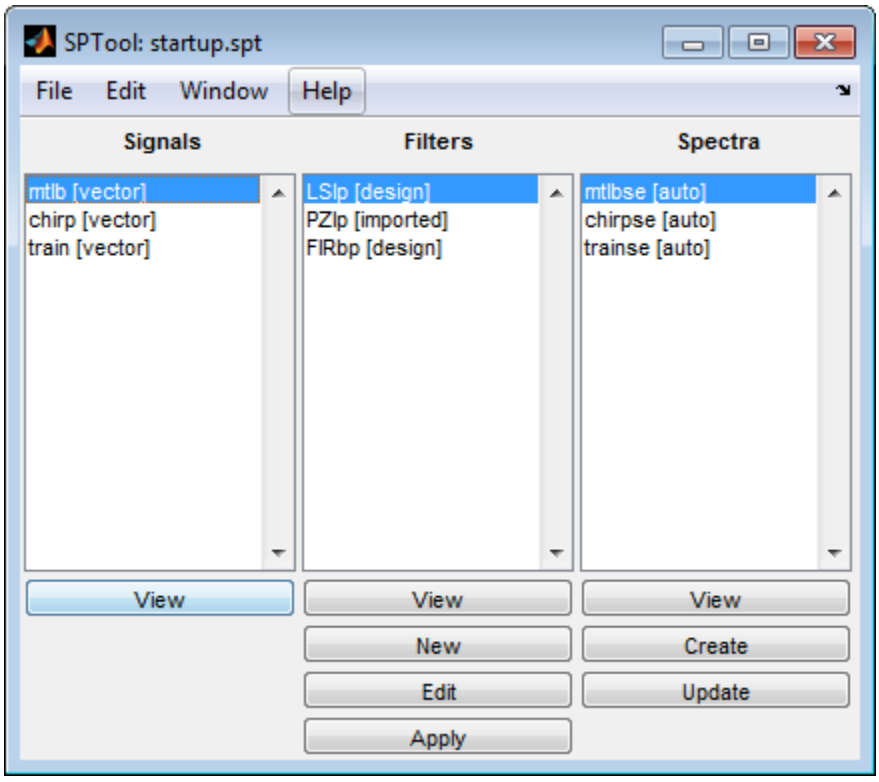
- For signal and spectral analysis, use the **Signal Analyzer** app instead.
- For filter design, use the **Filter Designer** app instead.
- For filter visualization, use **FVTool** instead.

Syntax

```
sptool
s = sptool('Signals')
f = sptool('Filters')
s = sptool('Spectra')
[s,ind] = sptool( ___ )
s = sptool( ___ ,0)
struc = sptool('create',paramlist)
sptool('load',struc)
struc = sptool('load',paramlist)
```

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:

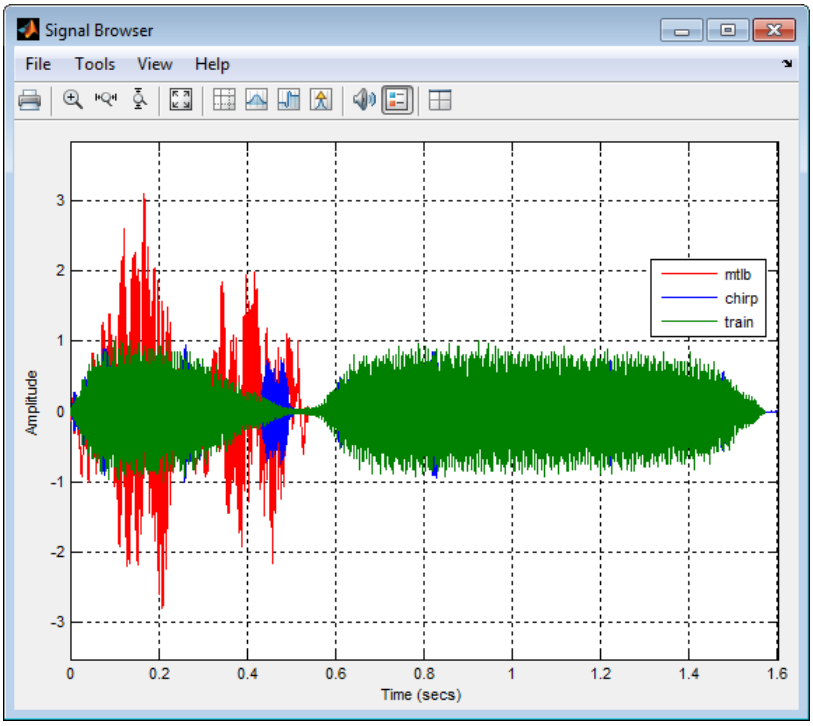
- Analyze signals listed in the **Signals** list box with the Signal Browser.
- Design or edit filters with the Filter Design and Analysis Tool (includes a Pole/Zero Editor).
- Analyze filter responses for filters listed in the **Filters** list box with FVTool.
- Apply filters in the **Filters** list 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 1-2504
- “Filter Designer App” on page 1-2513
- “Filter Visualization Tool” on page 1-2513
- “Spectrum Viewer” on page 1-2513

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 1-2505
- “Signal Display” on page 1-2506
- “Measurements Panels” on page 1-2508
- “Visuals — Time Domain Options” on page 1-2508
- “Style Dialog Box” on page 1-2511

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.

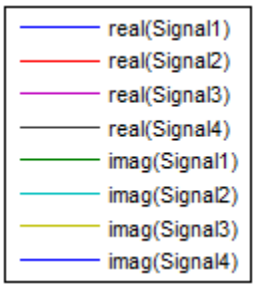
Multiple Signal Colors

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:

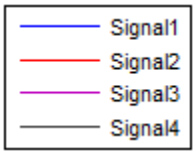
1. Blue
2. Dark Green
3. Red

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.

Use Multiple Displays

You can display multiple channels of data on different displays in the window. In the toolbar, select **View > Layout**, or select the Layout button ( ).

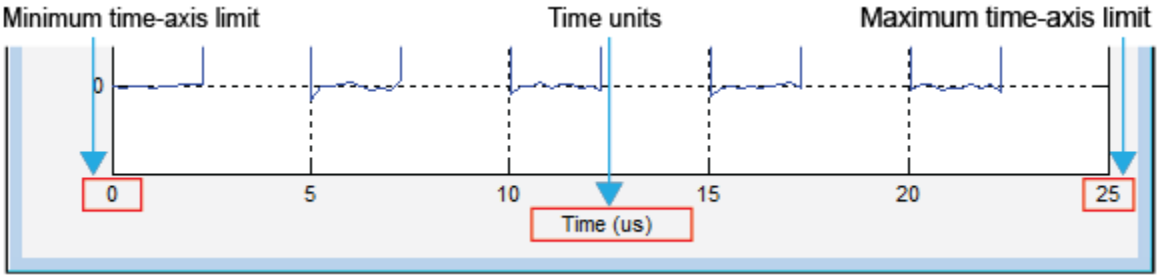
You can tile the window into multiple displays. For example, if there are three inputs to the tool, you can display the signals in three separate displays. The layout grid shows a 4 by 4 grid, but you can select up to 16 by 16 by clicking and dragging within the layout grid.

When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the active display. The 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.

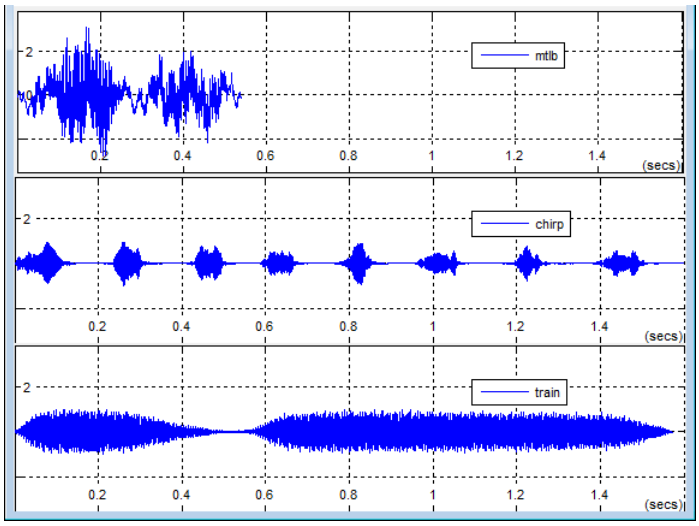
  To hide both the word `Time` and the values on the *time*-axis, set the **Show time-axis labels** parameter to `None`. To hide both the word `Time` and the values on the *time*-axis in all displays except the bottom ones in each column of displays, set this parameter to `Bottom Displays Only`. This behavior 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 Text

Signal Browser uses the names of the signals in the SPTool as the text displayed in the legends. If you change the name of any selected signal in the **Signals** list box, its corresponding legend entry 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 to reflect the new signal name you entered. Similarly, if you modify any entry in a legend in Signal Browser, then 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 1-2508 section for more information.

Measurements Panels

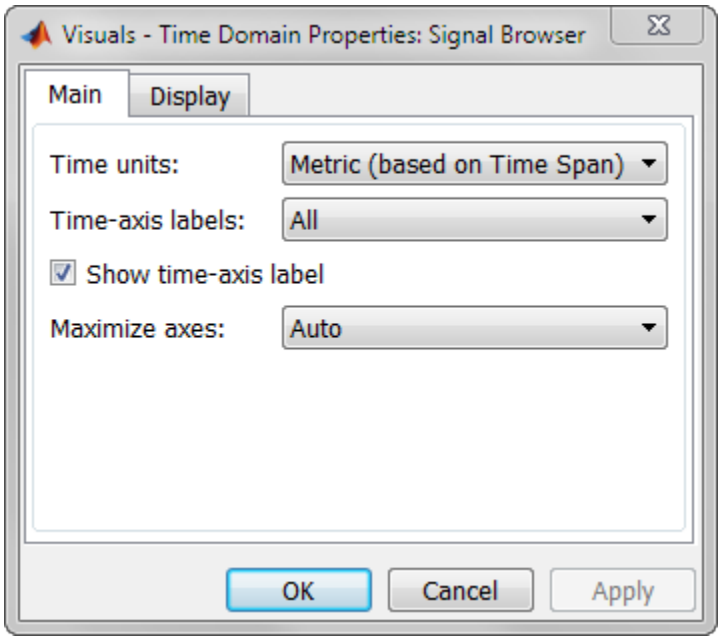
The Measurements panels are the five panels that appear at the right side of the Signal Browser. These panels are labeled **Trace selection**, **Cursor measurements**, **Signal statistics**, **Bilevel measurements**, and **Peak finder**.

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

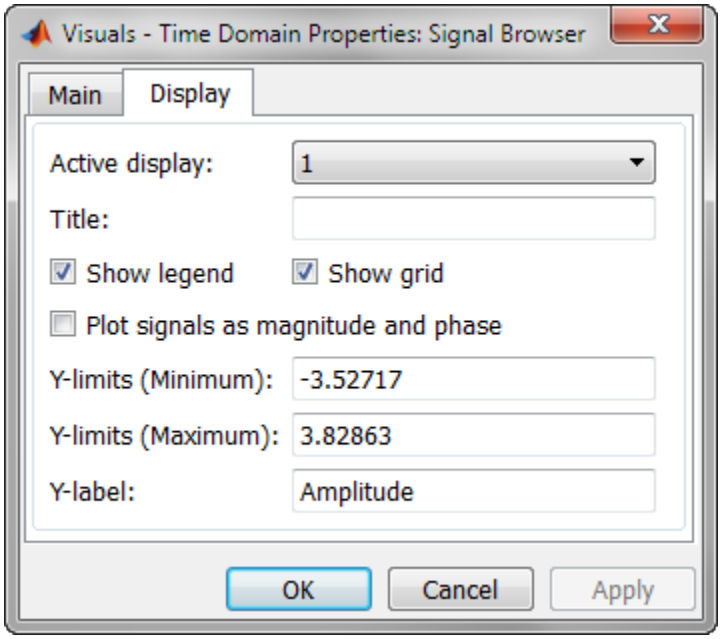
Specify whether to display the Signal Browser in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. In each display, there is no space to show labels. Tick

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 text. 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.

You can edit the name of any channel in the legend by double-clicking the current name and entering 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**. The legend lets you modify what signals are shown. To show only one signal, click the signal name. To toggle a signal on/off, right-click the signal name.

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. 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.

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 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**.

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.

Preserve colors for copy to clipboard

Specify whether or not to use the displayed color of the scope when copying.

When you select **File > Copy to Clipboard**, the software changes the color of the scope to be printer friendly (white background, visible lines). If you want to copy and paste the scope with the colors displayed, select this check box.

Default: Off

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.

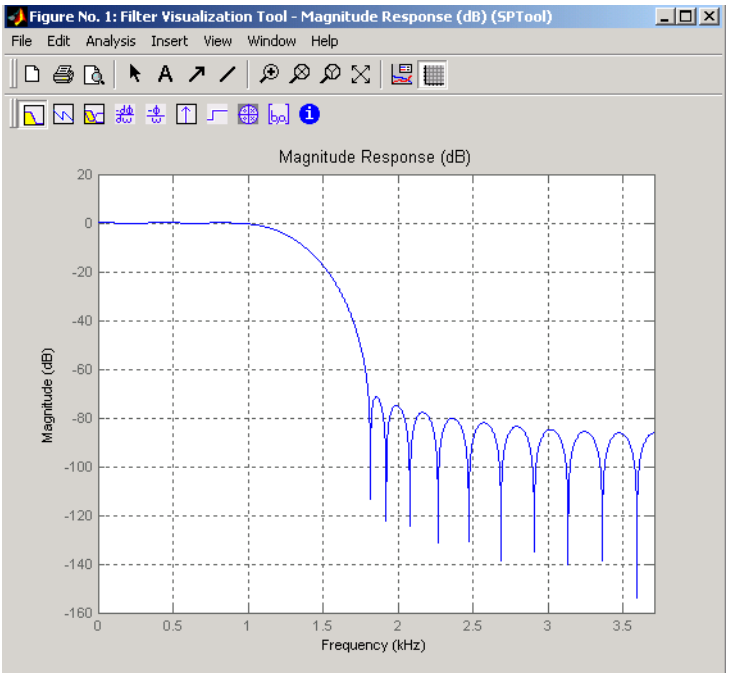
Filter Designer App

The **Filter Designer** app allows you to design and edit FIR and IIR filters. To launch the app, press either the **New** button or the **Edit** button under the **Filters** list box in SPTool.

The Filter Designer app has a Pole/Zero Editor you can access by selecting the icon in the left column.

Filter Visualization Tool

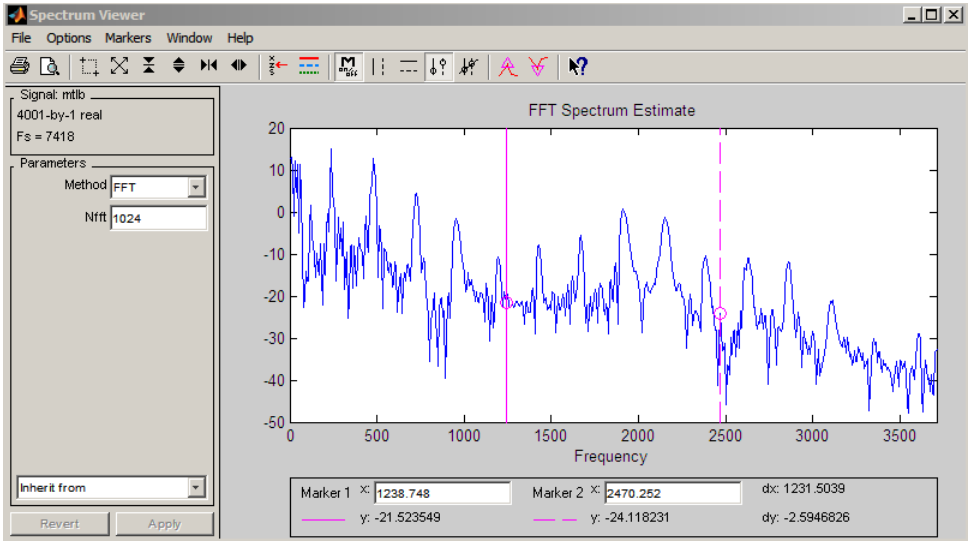
The Filter Visualization Tool (**FVTool**) allows you to view the characteristics of a designed or imported filter, including its magnitude response, phase response, group delay, phase delay, pole-zero plot, impulse response, and step response. To activate FVTool, click the **View** button under the **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.

Controlling SPTool from the MATLAB Command Line

You can import or export data from SPTool using the command line.

Exporting Component Structures from SPTool

The following commands export component structures from the currently open SPTool:

- `s = sptool('Signals')` returns a structure array of all the signals.
- `f = sptool('Filters')` returns a structure array of all the filters.
- `s = sptool('Spectra')` returns a structure array of all the spectra.
- `[s,ind] = sptool( ___ )` returns an index vector indicating which of the elements of `s` are currently selected in SPTool.
- `s = sptool( ___ ,0)` returns only the currently selected objects.

Creating and Loading Component Structures

The following commands create component structures and load them into SPTool, opening SPTool if necessary:

- `struc = sptool('create',paramlist)` creates in the workspace a component structure, `struc`, defined by `paramlist`.

  `sptool('load',struc)` loads `struc` into SPTool.
- `struc = sptool('load',paramlist)` loads the component structure defined by `paramlist` into SPTool. If you specify an output argument, then the command also creates a component structure in the workspace.

Example: Create and load a 5th-order Butterworth filter with a cutoff frequency of 0.5π rad/sample. Specify the filter in state-space representation, label it `Butterworth` within SPTool, and set it to filter digital signals sampled at 1 kHz.

```
[z,p,k] = butter(5,0.5);
struc = sptool('create','Filter','zpk',z,p,k,1e3,'Butterworth');
sptool('load',struc)
```

Example: Load into SPTool the periodogram PSD estimate of a 512-sample sinusoidal signal embedded in white noise. Work in normalized units and specify a sinusoid frequency of $\pi/4$ rad/sample. Label the spectrum `PSD` within SPTool.

```
n = 0:511;
x = sin(pi/4*n)+randn(size(n))/10;
[pxx,w] = periodogram(x);
sptool('load','Spectrum',pxx,w,'PSD')
```

Example: Create and load a quadratic chirp modulated by a Gaussian. Specify a sample rate of 2 kHz and a signal duration of 2 seconds. Generate a copy of the structure in the workspace.

```
t = 0:1/2000:2-1/2000;
q = chirp(t-2,4,1/2,6,'quadratic',100,'convex').*exp(-4*(t-1).^2);
Chirp = sptool('load',q,2000)
```

```
Chirp =

  struct with fields:

            data: [4000×1 double]
              Fs: 2000
            type: 'vector'
        lineinfo: []
   SPTIdentifier: [1×1 struct]
           label: 'sig'
```

The parameters in `paramlist` must be input in the following order:

Component	`paramlist` Parameters
Signals	`component_name`, `data`, `fs`, `label`
Filters	`component_name`, `form`, `filter_params`, `fs`, `label`
Spectra	`component_name`, `data`, `f`, `label`

The parameters are defined as follows:

Parameter	Definition
`component_name`	Specify as one of `'Signal'`, `'Filter'`, or `'Spectrum'`. If omitted, `component_name` defaults to `'Signal'`.
`form`	Form or structure of a filter. Specify as one of `'tf'`, `'ss'`, `'sos'`, or `'zpk'`.
`data`	Vector of doubles representing a signal or spectrum.

Parameter	Definition
`filter_params`	Filter representation. • Specify `num` and `den` when `form` is `'tf'`. • Specify an SOS matrix when `form` is `'sos'`. • Specify `z`, `p`, and `k` when `form` is `'zpk'`. • Specify `A`, `B`, `C`, and `D` when `form` is `'ss'`.
`fs`	Optional parameter that specifies the sample rate. If omitted, `fs` defaults to 1.
`f`	Frequency vector. This parameter applies only if `component_name` is `'Spectrum'`.
`label`	Optional parameter that specifies the variable name of the component within SPTool. If omitted, `label` defaults to: • `'sig'` if `component_name` is `'Signal'` • `'filt'` if `component_name` is `'Filter'` • `'spec'` if `component_name` is `'Spectrum'`

Compatibility Considerations

sptool has been removed
Errors starting in R2022a

SPTool has been removed.

- For signal and spectral analysis, use the **Signal Analyzer** app instead.
- For filter design, use the **Filter Designer** app instead.
- For filter visualization, use **FVTool** instead.

See Also

Apps
Filter Designer | **Signal Analyzer**

Functions
`findpeaks` | **FVTool**

Introduced before R2006a

square

Square wave

Syntax

```
x = square(t)
x = square(t,duty)
```

Description

`x = square(t)` generates a square wave with period 2π for the elements of the time array `t`. `square` is similar to the sine function but creates a square wave with values of –1 and 1.

`x = square(t,duty)` generates a square wave with specified duty cycle `duty`. The *duty cycle* is the percent of the signal period in which the square wave is positive.

Examples

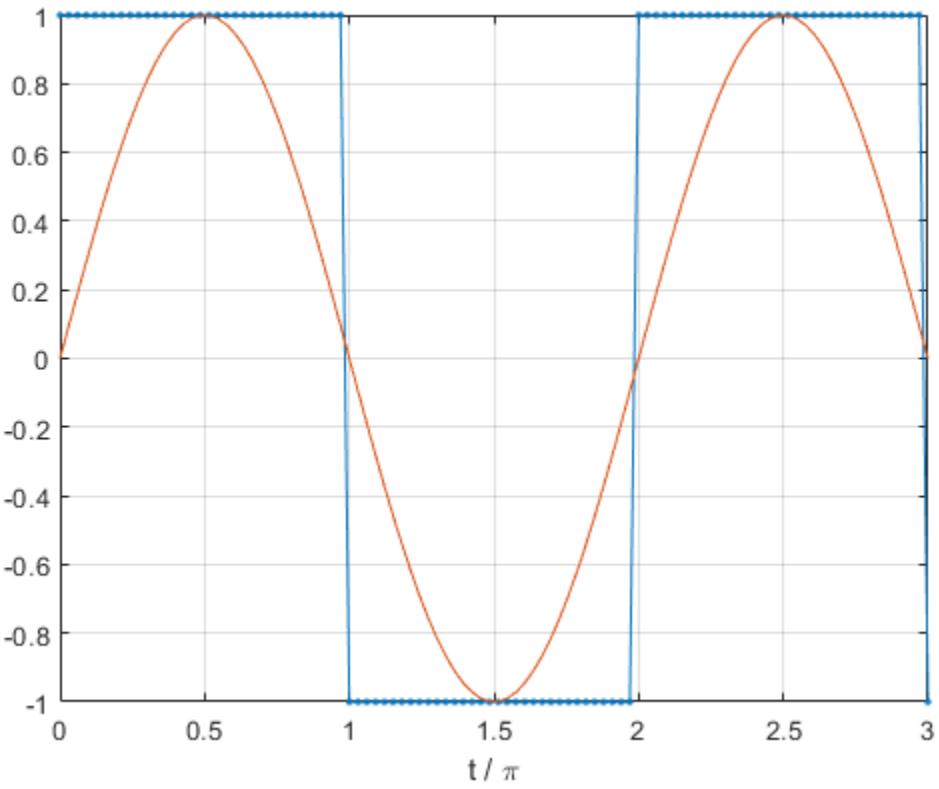
Generate Square Waves

Create a vector of 100 equally spaced numbers from 0 to 3π. Generate a square wave with a period of 2π.

```
t = linspace(0,3*pi)';
x = square(t);
```

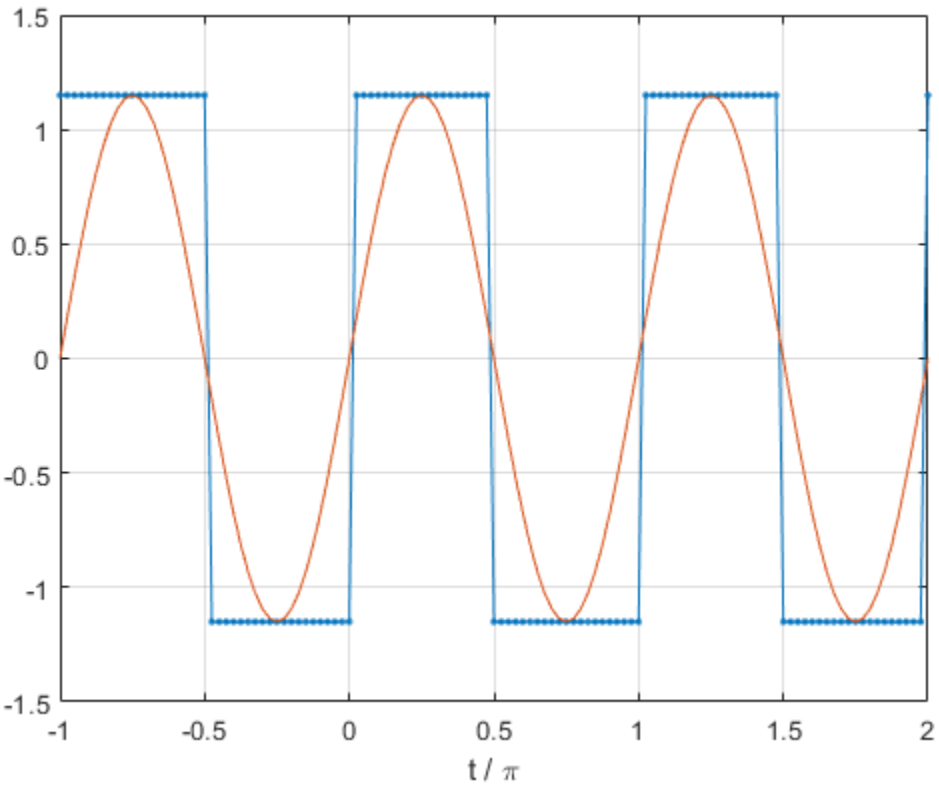
Plot the square wave and overlay a sine. Normalize the x-axis by π. The generated square wave has a value of 1 for intervals $[n\pi, (n+1)\pi)$ with even n and a value of -1 for intervals $[n\pi, (n+1)\pi)$ with odd n. The wave never has a value of 0.

```
plot(t/pi,x,'.-',t/pi,sin(t))
xlabel('t / \pi')
grid on
```



Repeat the calculation, but now evaluate `square(2*t)` at 121 equally spaced numbers between $-\pi$ and 2π. Change the amplitude to 1.15. Plot the wave and overlay a sine with the same parameters. This new wave is negative at $t = 0$ and positive at the endpoints, $-\pi$ and 2π.

```
t = linspace(-pi,2*pi,121);
x = 1.15*square(2*t);

plot(t/pi,x,'.-',t/pi,1.15*sin(2*t))
xlabel('t / \pi')
grid on
```



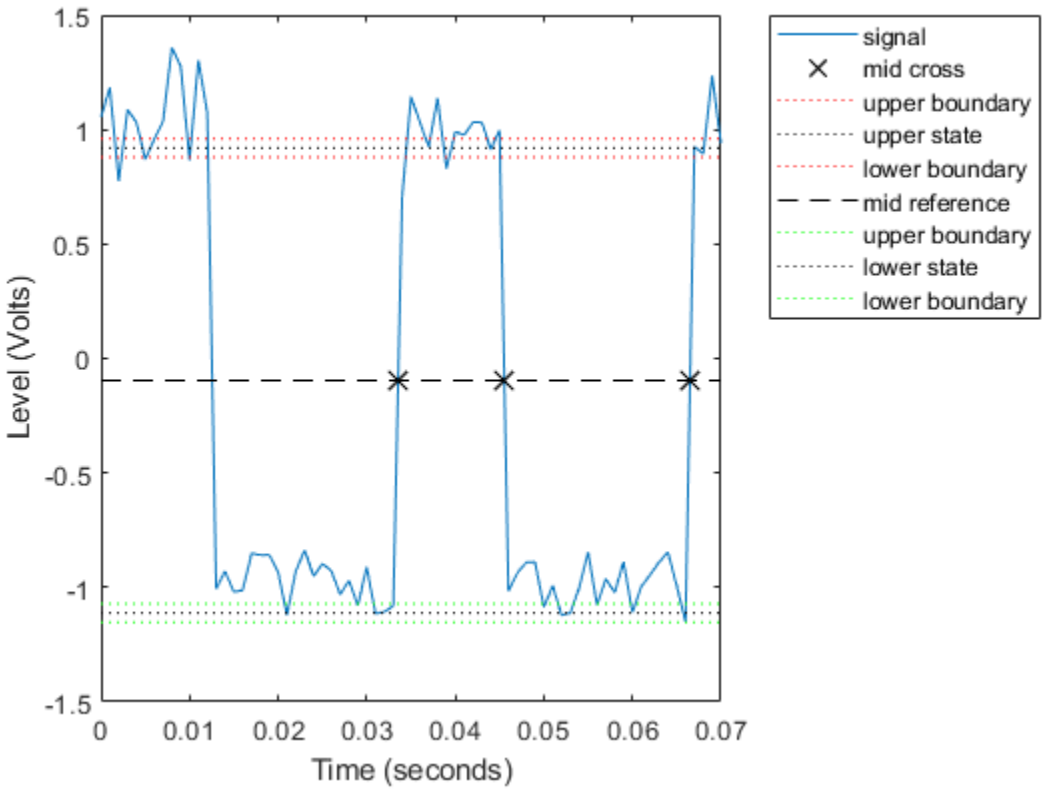
Duty Cycle of Square Wave

Generate a 30 Hz square wave sampled at 1 kHz for 70 ms. Specify a duty cycle of 37%. Add white Gaussian noise with a variance of 1/100.

```
t = 0:1/1e3:0.07;
y = square(2*pi*30*t,37)+randn(size(t))/10;
```

Compute the duty cycle of the wave. Plot the waveform and annotate the duty cycle.

```
dutycycle(y,t)
```




```
ans = 0.3639
```

Input Arguments

`t` — Time array

vector | matrix | *N*-D array

Time array, specified as a vector, matrix, or *N*-D array. `square` operates along the first array dimension of `t` with size greater than 1.

Data Types: `single` | `double`

`duty` — Duty cycle

`50` (default) | real scalar from 0 to 100

Duty cycle, specified as a real scalar from 0 to 100.

Data Types: `single` | `double`

Output Arguments

`x` — Square wave

vector | matrix | *N*-D array

Square wave, returned as a vector, matrix, or *N*-D array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`chirp` | `cos` | `diric` | `gauspuls` | `pulstran` | `rectpuls` | `sawtooth` | `sin` | `tripuls`

Introduced before R2006a

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

$$x(k+1) = Ax(k) + Bu(k),$$
$$y(k) = Cx(k) + Du(k),$$

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 0.2π 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 = 6×6

    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 = 6×1

    0.0913
         0
    0.0046
         0
    0.0001
         0


C = 1×6

    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 0.5π 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`

Introduced in R2014a

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

`[sos,g] = ss2sos(A,B,C,D)` returns second-order section form `sos` with gain `g` that is equivalent to the state-space system represented by input arguments `A`, `B`, `C`, and `D`. The input state-space system must be single-output and real.

`[sos,g] = ss2sos(A,B,C,D,iu)` specifies index `iu` that indicates which input of the state-space system `A`, `B`, `C`, `D` the function uses in the conversion.

`[sos,g] = ss2sos(A,B,C,D,order)` specifies the order of the rows in `sos` with `order`.

`[sos,g] = ss2sos(A,B,C,D,iu,order)` specifies both the index `ui` and the order of the rows `order`.

`[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.

`sos = ss2sos( ___ )` embeds the overall system gain `g` in the first section. You can specify an input combination from any of the previous syntaxes.

Examples

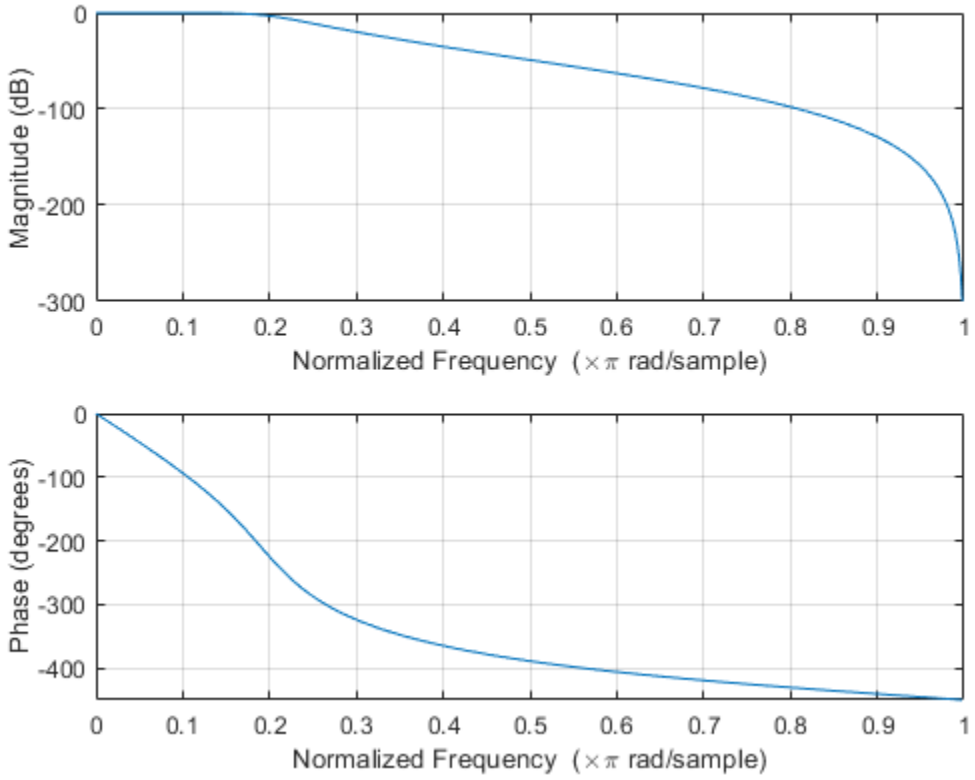
Second-Order Section Form of Filter

Design a fifth-order Butterworth lowpass filter, specifying a cutoff frequency of 0.2π rad/sample and expressing 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)
```

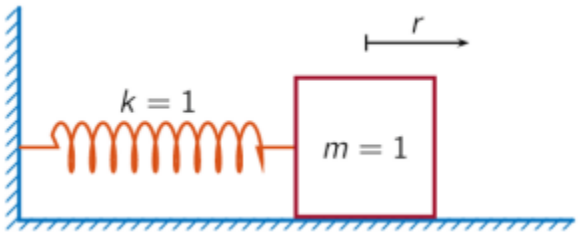
```
sos = 3×6

    0.0013    0.0013         0    1.0000   -0.5095         0
    1.0000    1.9996    0.9996    1.0000   -1.0966    0.3554
    1.0000    2.0000    1.0000    1.0000   -1.3693    0.6926
```

```
freqz(sos)
```



Mass-Spring System

A one-dimensional discrete-time oscillating system consists of a unit mass, m, attached to a wall by a spring of unit elastic constant. A sensor measures the acceleration, a, of the mass.



The system is sampled at F_s = 5 Hz. Generate 50 time samples. Define the sampling interval $\Delta t = 1/F_s$.

```
Fs = 5;
dt = 1/Fs;
N = 50;
t = dt*(0:N-1);
```

The oscillator can be described by the state-space equations

$$x(k+1) = Ax(k) + Bu(k),$$
$$y(k) = Cx(k) + Du(k),$$

where $x = (r\ v)^T$ is the state vector, r and v are respectively the position and velocity of the mass, and the matrices

$$A = \begin{pmatrix} \cos\Delta t & \sin\Delta t \\ -\sin\Delta t & \cos\Delta t \end{pmatrix}, \quad B = \begin{pmatrix} 1-\cos\Delta t \\ \sin\Delta t \end{pmatrix}, \quad C = (-1\ 0), \quad D = (1).$$

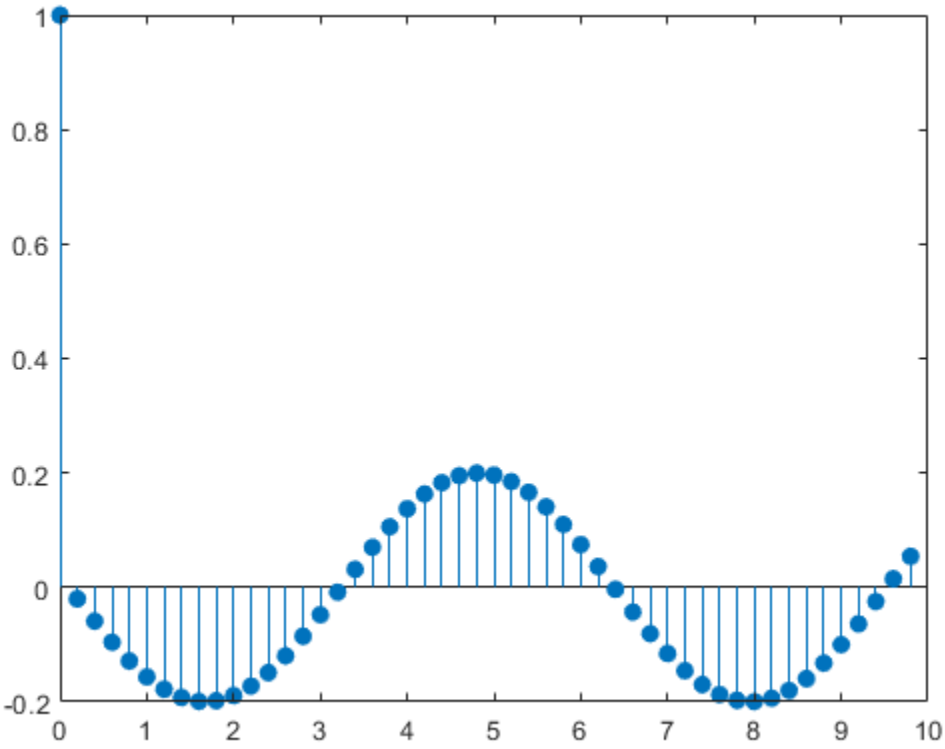
```
A = [cos(dt) sin(dt);-sin(dt) cos(dt)];
B = [1-cos(dt);sin(dt)];
C = [-1 0];
D = 1;
```

The system is excited with a unit impulse in the positive direction. Use the state-space model to compute the time evolution of the system starting from an all-zero initial state.

```
u = [1 zeros(1,N-1)];

x = [0;0];
for k = 1:N
    y(k) = C*x + D*u(k);
    x = A*x + B*u(k);
end
```

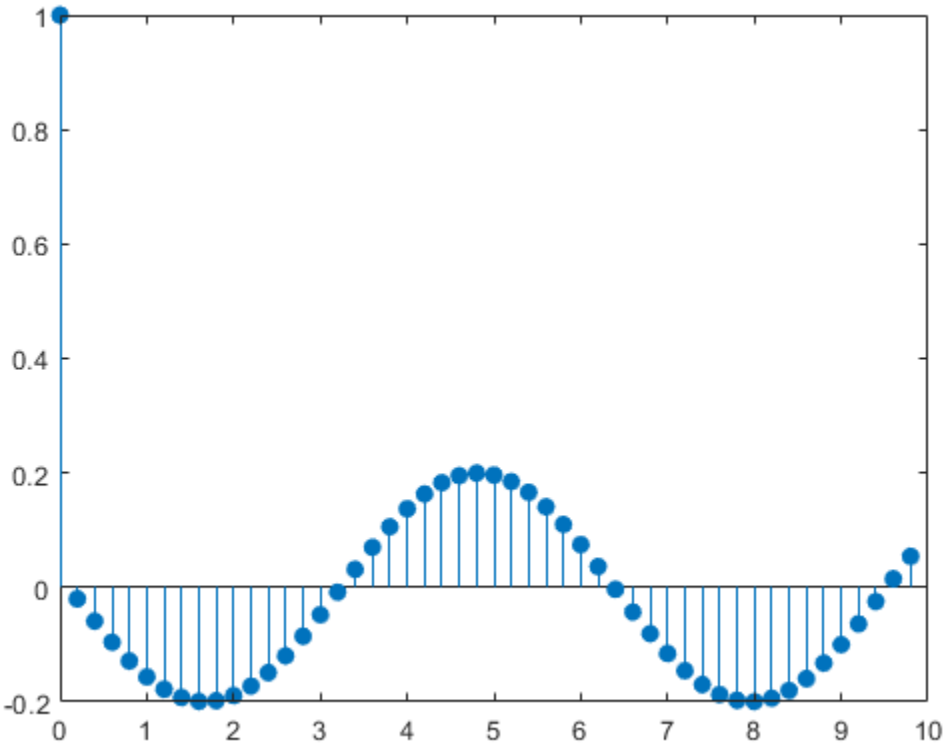
Plot the acceleration of the mass as a function of time.

```
stem(t,y,'filled')
```



Compute the time-dependent acceleration using the transfer function to filter the input. Express the transfer function as second-order sections. Plot the result.

```
sos = ss2sos(A,B,C,D);
yt = sosfilt(sos,u);
stem(t,yt,'filled')
```



The result is the same in both cases.

Input Arguments

`A` — State matrix

matrix

State matrix, specified as a matrix. If the system has p inputs and q outputs and is described by n state variables, then `A` is of size n-by-n.

`B` — Input-to-state matrix

matrix

Input-to-state matrix, specified as a matrix. If the system has p inputs and q outputs and is described by n state variables, then `B` is of size n-by-p.

`C` — Output-to-state matrix

matrix

Output-to-state matrix, specified as a matrix. If the system has p inputs and q outputs and is described by n state variables, then `C` is of size q-by-n.

`D` — Feedthrough matrix

matrix

Feedthrough matrix, specified as a matrix. If the system has *p* inputs and *q* outputs and is described by *n* state variables, then `D` is of size *q*-by-*p*.

`iu` — Index
1 (default) | integer

Index, specified as an integer.

`order` — Row order
`'up'` (default) | `'down'`

Row order in `sos`, specified as one of these values:

- `'down'` — Order the sections so that the first row of `sos` contains the poles that are closest to the unit circle.
- `'up'` — Order the sections so that the first row of `sos` contains the poles that are farthest from the unit circle.

The zeros are paired with the poles that are closest to them.

`scale` — Scaling of gain and numerator coefficients
`'none'` (default) | `'inf'`

Scaling of the gain and numerator coefficients, specified as one of these values:

- `'none'` — Apply no scaling.
- `'inf'` — Apply infinity-norm scaling.
- `'two'` — 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 for only direct-form II implementations.

Output Arguments

`sos` — Second-order section representation
matrix

Second-order section representation, returned as a matrix. `sos` is an *L*-by-6 matrix of the form

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

whose rows contain the numerator and denominator coefficients b_{ik} and a_{ik} of the second-order sections of $H(z)$, which is given by

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

g — Overall system gain
real-valued scalar

Overall system gain, returned as a real-valued scalar.

If you call the function with one output argument, the function embeds the gain 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 can result in erratic scaling. To avoid embedding the gain, use the function with two outputs: `sos` and `g`.

Algorithms

The `ss2sos` function uses this four-step algorithm to determine the second-order section representation for an input state-space system.

1. Find the poles and zeros of the system given by `A`, `B`, `C`, and `D`.
2. Use 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 these rules:
   a. Match the poles that are closest to the unit circle with the zeros that are closest to those poles.
   b. Match the poles that are next closest to the unit circle with the zeros that are closest to those poles.
   c. Continue this process until all of the poles and zeros are matched.

   The `ss2sos` function groups real poles into sections with the real poles that are closest to them in absolute value. The same rule holds for real zeros.
3. Order the sections according to the proximity of the pole pairs to the unit circle. The `ss2sos` function normally orders the sections with poles that are closest to the unit circle last in the cascade. You can specify for `ss2sos` to order the sections in the reverse order by setting the order input to `'down'`.
4. Scale the sections by the norm specified by the `scale` input. 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 ∞ or 2. For details, see the references. This scaling is an attempt to minimize overflow or peak round-off noise in fixed-point filter implementations.

References

[1] Jackson, Leland B. *Digital Filters and Signal Processing*. Boston: Kluwer Academic Publishers, 1996.

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

[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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Any character or string input must be a constant at compile time.

See Also

`cplxpair` | `sos2ss` | `ss2tf` | `ss2zp` | `tf2sos` | `zp2sos`

Introduced before R2006a

ss2zp

Convert state-space filter parameters to zero-pole-gain form

Syntax

```
[z,p,k] = ss2zp(A,B,C,D)
[z,p,k] = ss2zp(A,B,C,D,ni)
```

Description

`[z,p,k] = ss2zp(A,B,C,D)` converts a state-space representation

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

of a given continuous-time or discrete-time system to an equivalent zero-pole-gain representation

$$H(s) = \frac{Z(s)}{P(s)} = k\frac{(s - z_1)(s - z_2)\cdots(s - z_n)}{(s - p_1)(s - p_2)\cdots(s - p_n)}$$

whose zeros, poles, and gains represent the transfer function in factored form.

`[z,p,k] = ss2zp(A,B,C,D,ni)` indicates that the system has multiple inputs and that the `ni`th input has been excited by a unit impulse.

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 = 2×1

         0
   -1.5000


p = 2×1 complex

  -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 = 2×1

   -1.5000
    0.0000


p = 2×1 complex

  -0.2000 + 0.9798i
  -0.2000 - 0.9798i


k = 2
```

Input Arguments

`A` — State matrix

matrix

State matrix. If the system has r inputs and q outputs and is described by n state variables, then `A` is n-by-n.

Data Types: `single` | `double`

`B` — Input-to-state matrix

matrix

Input-to-state matrix. If the system has r inputs and q outputs and is described by n state variables, then `B` is n-by-r.

Data Types: `single` | `double`

`C` — State-to-output matrix

matrix

Input-to-state matrix. If the system has r inputs and q outputs and is described by n state variables, then `C` is q-by-n.

Data Types: `single` | `double`

`D` — Feedthrough matrix

matrix

Feedthrough matrix. If the system has r inputs and q outputs and is described by n state variables, then `D` is q-by-r.

Data Types: `single` | `double`

ni — Input index

1 (default) | integer scalar

Input index, specified as an integer scalar. If the system has *r* inputs, use `ss2zp` with a trailing argument `ni` = 1, ..., *r* to compute the response to a unit impulse applied to the `ni`th input. Specifying this argument causes `ss2zp` to use the `ni`th columns of `B` and `D`.

Data Types: `single` | `double`

Output Arguments

z — Zeros

matrix

Zeros of the system, returned as a matrix. `z` contains the numerator zeros in its columns. `z` has as many columns as there are outputs (rows in `C`).

p — Poles

column vector

Poles of the system, returned as a column vector. `p` contains the pole locations of the denominator coefficients of the transfer function.

k — Gains

column vector

Gains of the system, returned as a column vector. `k` contains the gains for each numerator transfer function.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Outputs `z` and `p` are always complex.
- The order of outputs, `z` and `p`, might be different in MATLAB and the generated code.

See Also

`sos2zp` | `ss2sos` | `ss2tf` | `tf2zp` | `tf2zpk` | `zp2ss`

Introduced before R2006a

statelevels

State-level estimation for bilevel waveform with histogram method

Syntax

```
levels = statelevels(x)
levels = statelevels(x,nbins)
levels = statelevels(x,nbins,method)
levels = statelevels(x,nbins,method,bounds)

[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. For more information, see “Algorithms” on page 1-2542.

`levels = statelevels(x,nbins)` specifies the number of bins to use in the histogram as a positive scalar.

`levels = statelevels(x,nbins,method)` estimates state levels using the mean or mode of the subhistograms.

`levels = statelevels(x,nbins,method,bounds)` specifies the lower and upper bounds of the histogram in the two-element real row vector `bounds`. `statelevels` ignores any values of `x` that lie outside these bounds when it computes the histogram.

`[levels,histogram] = statelevels( ___ )` returns the histogram of the values in `x`.

`[levels,histogram,binlevels] = statelevels( ___ )` returns the centers of the histogram bins.

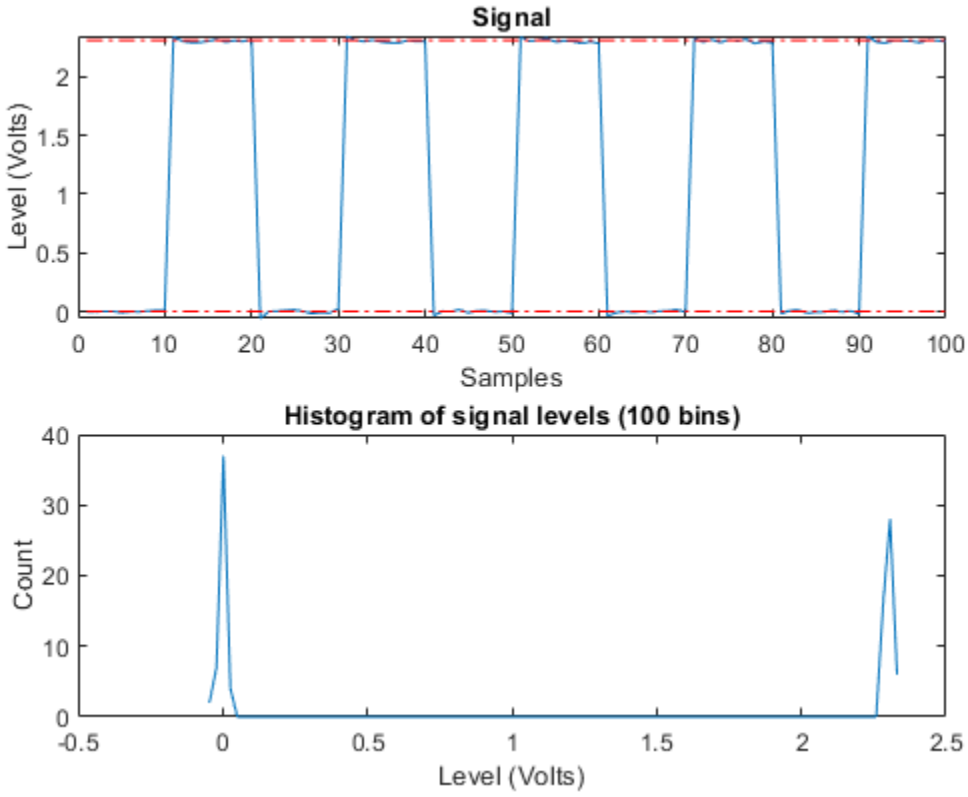
`statelevels( ___ )` displays a plot of the signal and the corresponding histogram.

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)
```




```
ans = 1×2

    0.0027    2.3068
```

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.

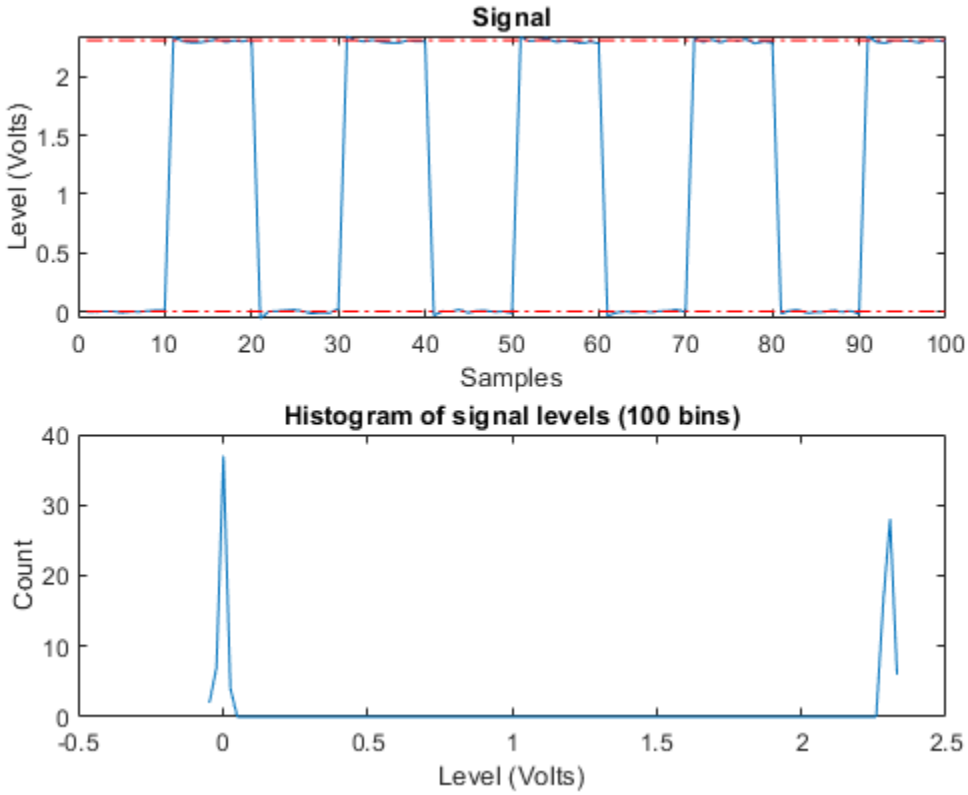
Use the default number of bins and modes of the subhistograms to estimate the state levels.

```
load('clockex.mat','x','t')
levs = statelevels(x)
```

```
levs = 1×2

    0.0027    2.3068
```

Plot the clock data with the lines indicating the estimated low- and high-state levels.

```
statelevels(x)
```




```
ans = 1×2

    0.0027    2.3068
```

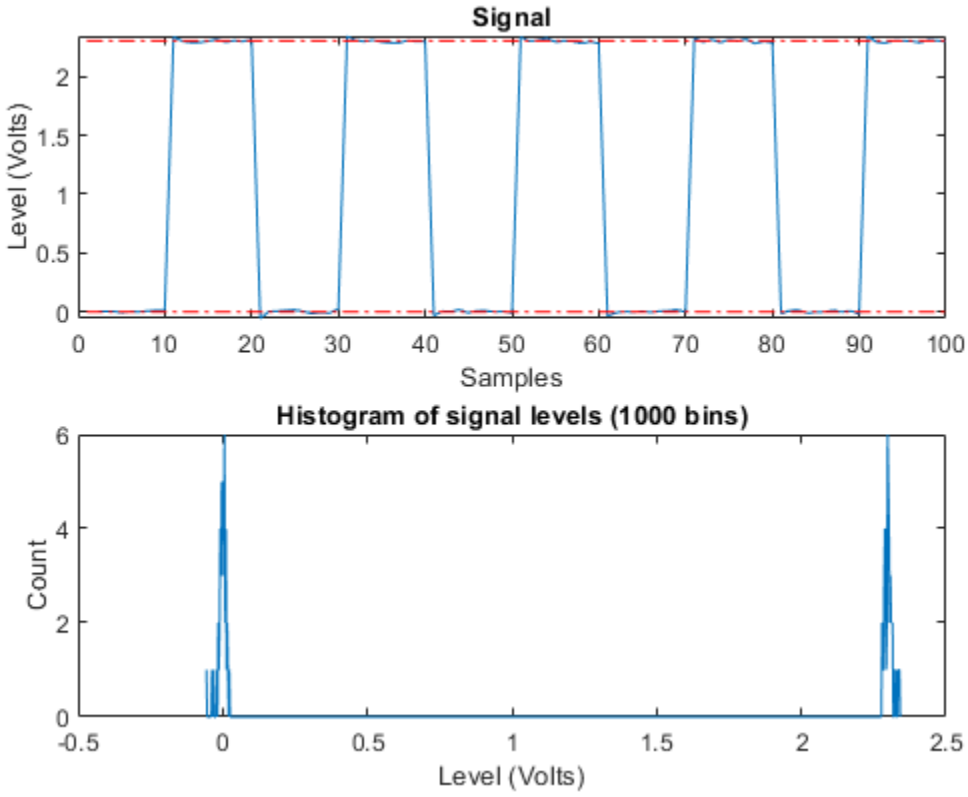
State Levels Using Means of Subhistograms

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')
```

```
statelevels(x,1e3,'mean')
```




```
ans = 1×2

   -0.0014    2.3014
```

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. Use four bins.

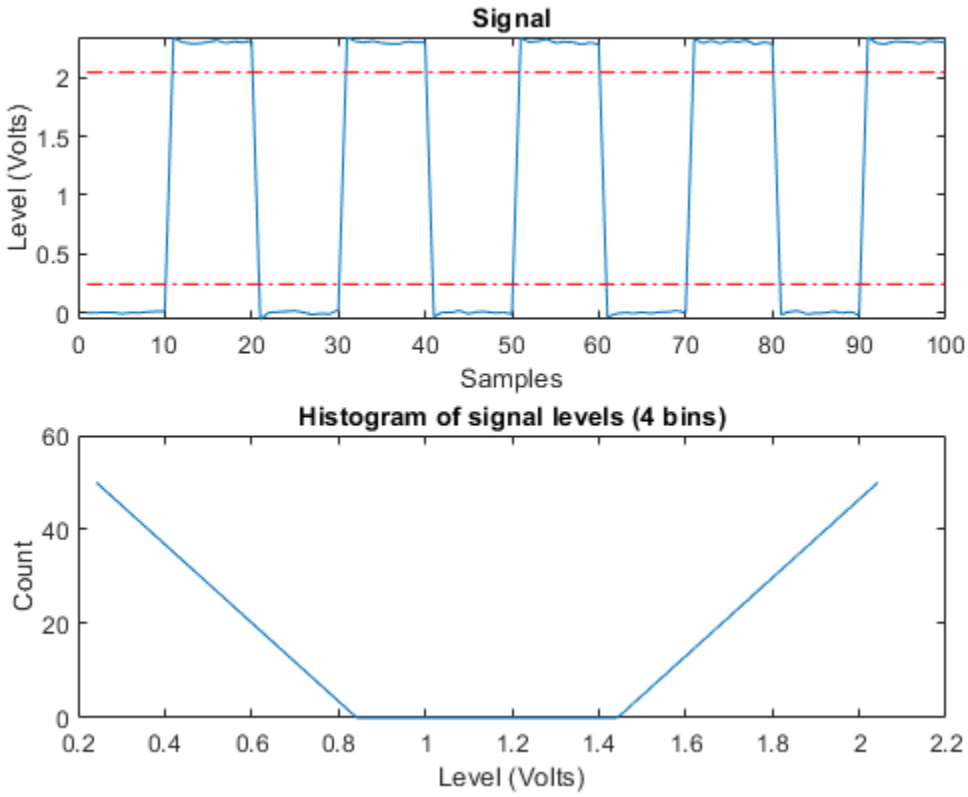
```
load('clockex.mat','x','t')
[levs,histog,bilevs] = statelevels(x,4)
```

```
levs = 1×2

    0.2427    2.0428


histog = 4×1

    50
     0
     0
    50
```

```
bilevs = 4×1

    0.2427
    0.8427
    1.4428
    2.0428
```

Plot the waveform and annotate the levels.

```
statelevels(x,4)
```




```
ans = 1×2

    0.2427    2.0428
```

Input Arguments

x — Bilevel waveform

real vector

Bilevel waveform, specified as a real-valued vector.

nbins — Number of histogram bins

100 (default) | positive integer

Number of histogram bins, specified as a real positive scalar.

`method` — State-level estimation method
`'mode'` (default) | `'mean'`

State-level estimation method in the subhistograms, specified as `'mode'` or `'mean'`. `method` specifies the statistic to use for the estimation of the low- and high-state levels. See “Algorithms” on page 1-2542.

`bounds` — Histogram lower and upper bounds
two-element real row vector

Histogram lower and upper bounds, specified as a two-element real row vector. `statelevels` ignores any values of `x` that lie outside these bounds when it computes the histogram.

Output Arguments

`levels` — Levels of low and high states
two-element positive row vector

Levels of low and high states, returned as a two-element positive row vector. The vector of state levels is estimated by the histogram method. The first element of `levels` is the low-state level and the second element is the high-state level.

`histogram` — Histogram counts
column vector

Histogram counts, returned as a column vector with `nbins` elements containing the number of data values in each histogram bin.

`binlevels` — Histogram bin centers
column vector

Histogram bin centers, returned as a column vector. The column vectors contain the bin centers for the histogram counts in `histogram`

More About

State

A state is 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.

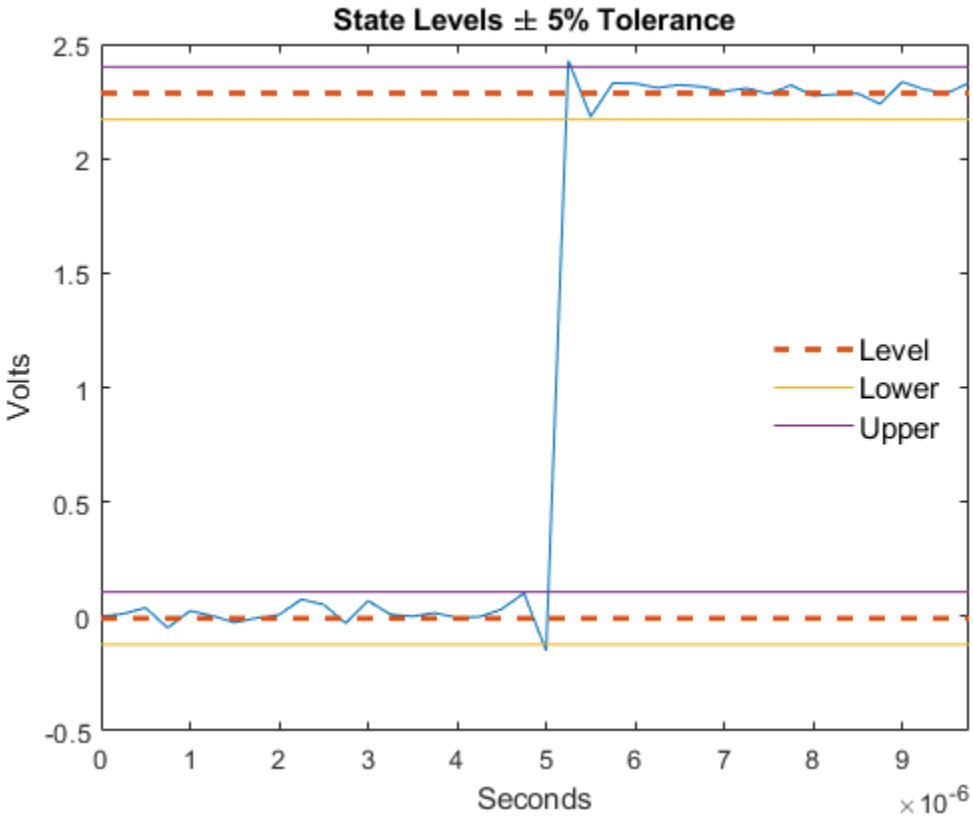
State-Level Tolerances

You can specify lower- and upper-state boundaries for each state level. Define the boundaries as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, specify the scalar as a small number such as 2/100 or 3/100. In general, the $\alpha\%$ 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.

This figure shows lower and upper 5% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The thick 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] on page 1-2543. The steps of this method are:

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, which is the ratio of the amplitude range to the number of bins.
3. Sort the data values into the histogram bins.
4. Identify the lowest-indexed histogram bin, 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 $i_{\text{low}} \le i \le \frac{1}{2}(i_{\text{high}} - i_{\text{low}})$.

   The indices of the upper histogram bins are $i_{\text{low}} + \frac{1}{2}(i_{\text{high}} - i_{\text{low}}) \le i \le i_{\text{high}}$.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`midcross` | `overshoot` | `risetime` | `undershoot`

Introduced in R2012a

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 vector `h` and the corresponding sample times `t` for the digital filter with transfer function coefficients stored in `b` and `a`.

`[h,t] = stepz(sos)` returns the step response corresponding to the second-order sections matrix `sos`.

`[h,t] = stepz(d)` returns the step response for the digital filter `d`.

`[h,t] = stepz(___,n)` computes the first `n` samples of the step response. This syntax can include any combination of input arguments from the previous syntaxes.

`[h,t] = stepz(___,n,fs)` computes `n` samples and produces a vector `t` so that the samples are spaced `1/fs` units apart.

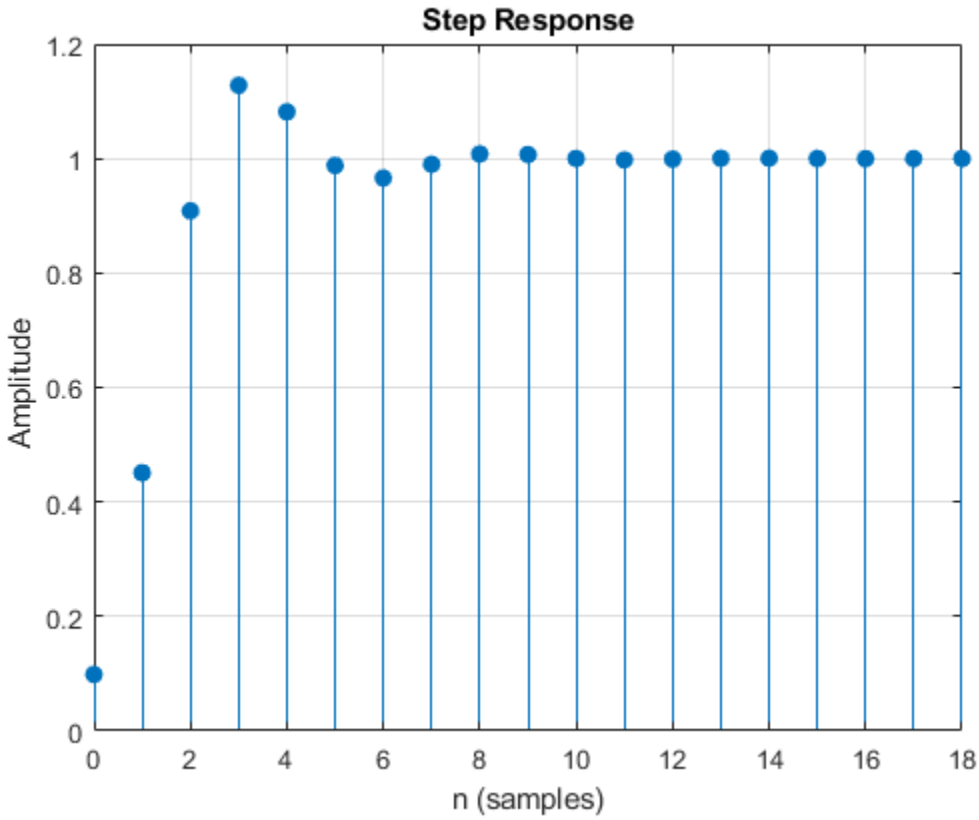
`stepz(___)` with no output arguments plots the step response of the filter. If you input a `digitalFilter`, the step response is displayed in **FVTool**.

Examples

Step Response of a Butterworth Filter

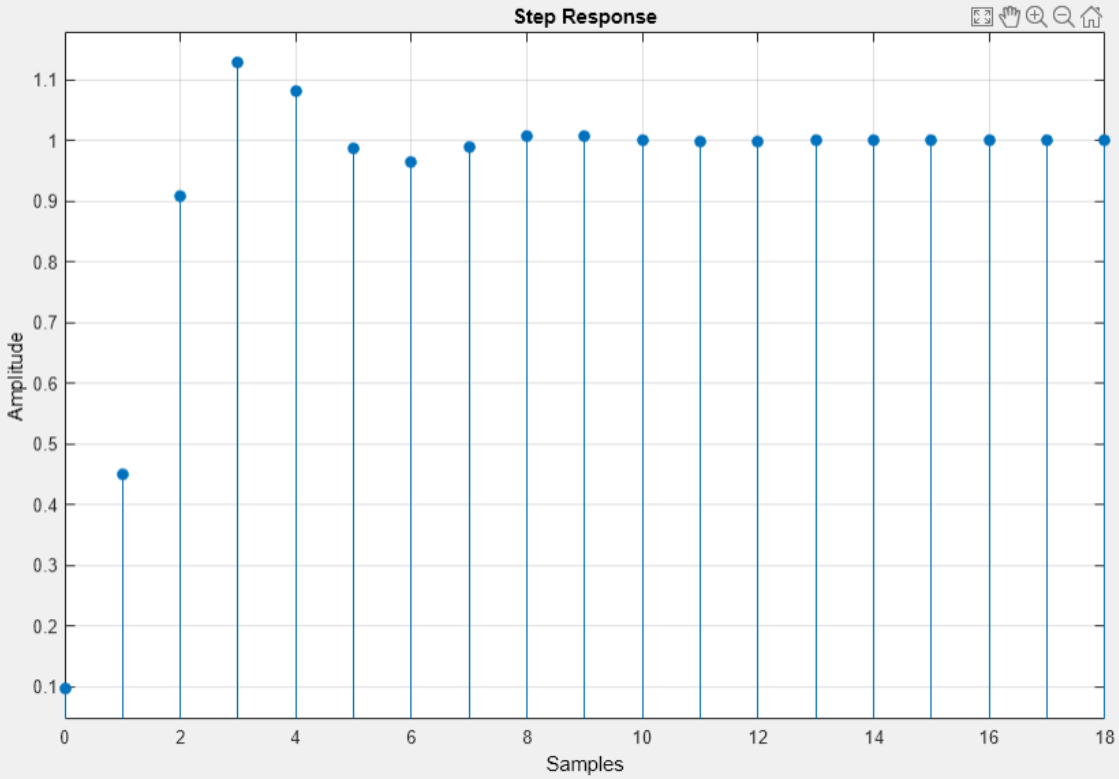
Create a third-order Butterworth filter with normalized half-power frequency 0.4π rad/sample. Display its step response.

```
[b,a] = butter(3,0.4);
stepz(b,a)
grid
```



Create an identical filter using `designfilt` and display its step response using `fvtool`.

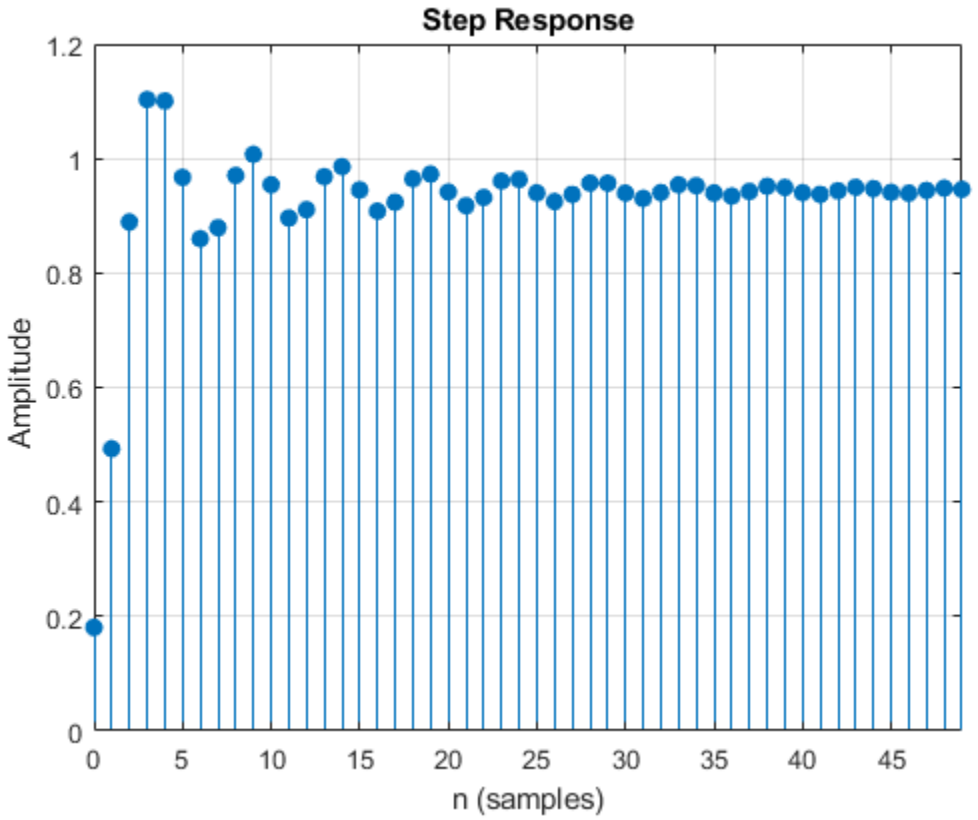
```
d = designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.4);
stepz(d)
```



Step Response of an Elliptic Filter

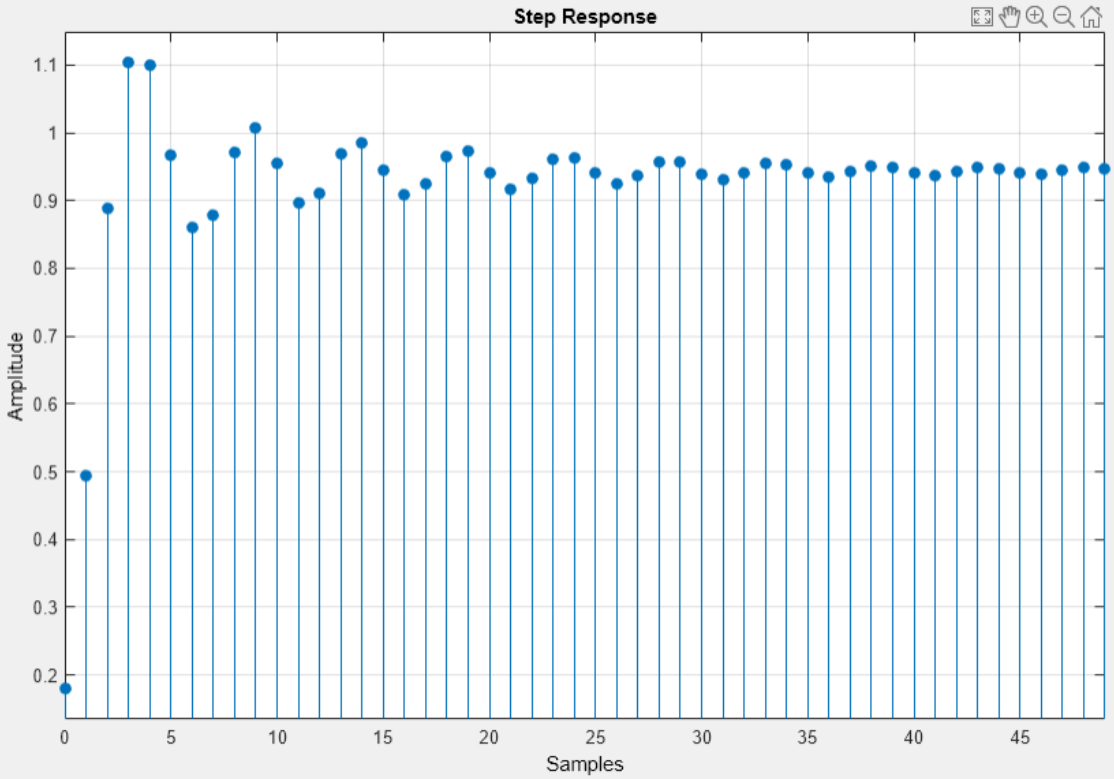
Design a fourth-order lowpass elliptic filter with normalized passband frequency 0.4π rad/sample. Specify a passband ripple of 0.5 dB and a stopband attenuation of 20 dB. Plot the first 50 samples of the filter's step response.

```
[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)
```



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

$$H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})} = \frac{\mathrm{b(1)+b(2)}\,e^{-j\omega} + \mathrm{b(3)}\,e^{-j2\omega} + \cdots + \mathrm{b(M)}\,e^{-j(M-1)\omega}}{\mathrm{a(1)+a(2)}\,e^{-j\omega} + \mathrm{a(3)}\,e^{-j2\omega} + \cdots + \mathrm{a(N)}\,e^{-j(N-1)\omega}}.$$

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 0.5π rad/sample.

Data Types: `double` | `single`
Complex Number Support: Yes

n — Number of evaluation points

positive integer scalar | positive integer vector

Number of evaluation points, specified as a positive integer scalar or positive integer vector. If `n` is a positive integer scalar (`t = [0:n-1]'`), the function computes the first `n` samples of the step response. If `n` is a vector of integers, the step response is computed only at those integer values, with 0 denoting the time origin.

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, the function treats the input as 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 0.5π 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 0.5π rad/sample.

`fs` — Sample rate
positive scalar

Sample rate, specified as a positive scalar. When the unit of time is seconds, `fs` is expressed in hertz.

Data Types: `double`

Output Arguments

`h` — Step response
column vector

Step response, returned as a column vector. If the input to `stepz` is single precision, the function computes the step response using single-precision arithmetic. The output `h` is single precision.

`t` — Sample times
vector

Sample times, returned as a vector.

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 5×10^{-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 5×10^{-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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If the first input to `stepz` is a variable-size matrix at compile time, then it must not become a vector at runtime.

See Also

`designfilt` | `digitalFilter` | `freqz` | `grpdelay` | `impz` | `phasez` | `zplane`

Introduced before R2006a

stft

Short-time Fourier transform

Syntax

```
s = stft(x)
s = stft(x,fs)
s = stft(x,ts)

s = stft(___,Name,Value)

[s,f] = stft(___)
[s,f,t] = stft(___)

stft(___)
```

Description

`s = stft(x)` returns the "Short-Time Fourier Transform" on page 1-2563 (STFT) of `x`.

`s = stft(x,fs)` returns the STFT of `x` using sample rate `fs`.

`s = stft(x,ts)` returns the STFT of `x` using sample time `ts`.

`s = stft(___,Name,Value)` specifies additional options using name-value pair arguments. Options include the FFT window and length. These arguments can be added to any of the previous input syntaxes.

`[s,f] = stft(___)` returns the frequencies `f` at which the STFT is evaluated.

`[s,f,t] = stft(___)` returns the times at which the STFT is evaluated.

`stft(___)` with no output arguments plots the magnitude of the STFT in the current figure window.

Examples

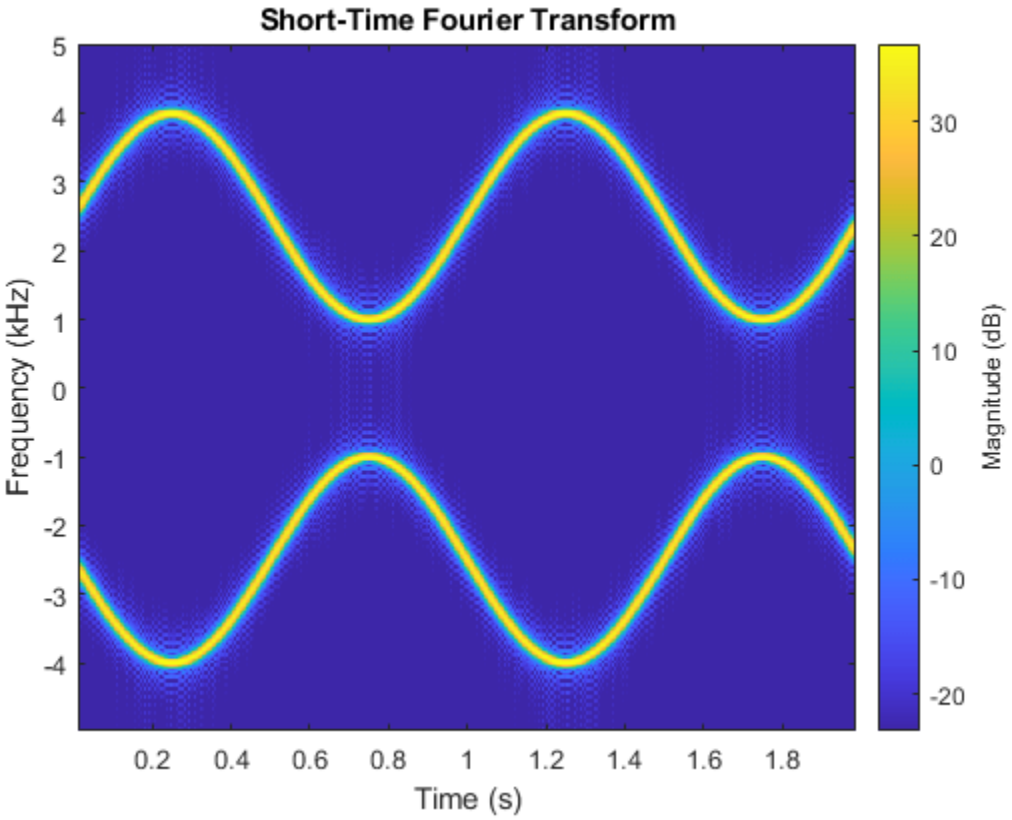
3D STFT Visualization

Generate two seconds of a voltage controlled oscillator output, controlled by a sinusoid sampled at 10 kHz.

```
fs = 10e3;
t = 0:1/fs:2;
x = vco(sin(2*pi*t),[0.1 0.4]*fs,fs);
```

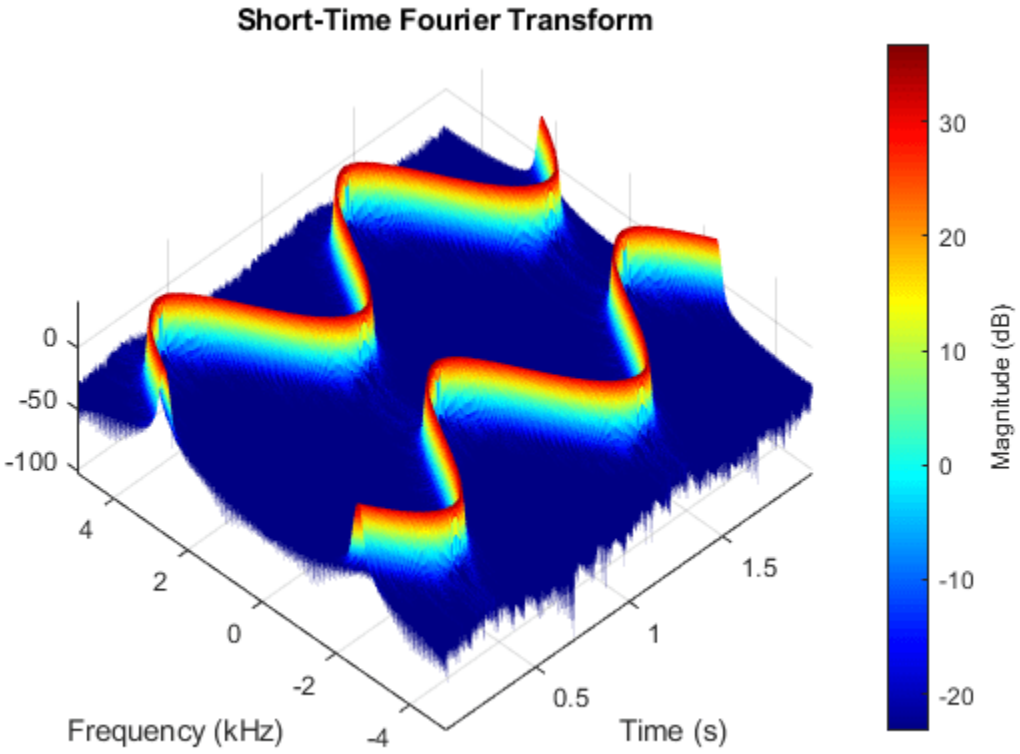
Compute and plot the STFT of the signal. Use a Kaiser window of length 256 and shape parameter $\beta = 5$. Specify the length of overlap as 220 samples and DFT length as 512 points. Plot the STFT with default colormap and view.

```
stft(x,fs,'Window',kaiser(256,5),'OverlapLength',220,'FFTLength',512)
```



Change the view to display the STFT as a waterfall plot. Set the colormap to `jet`.

```
view(-45,65)
colormap jet
```



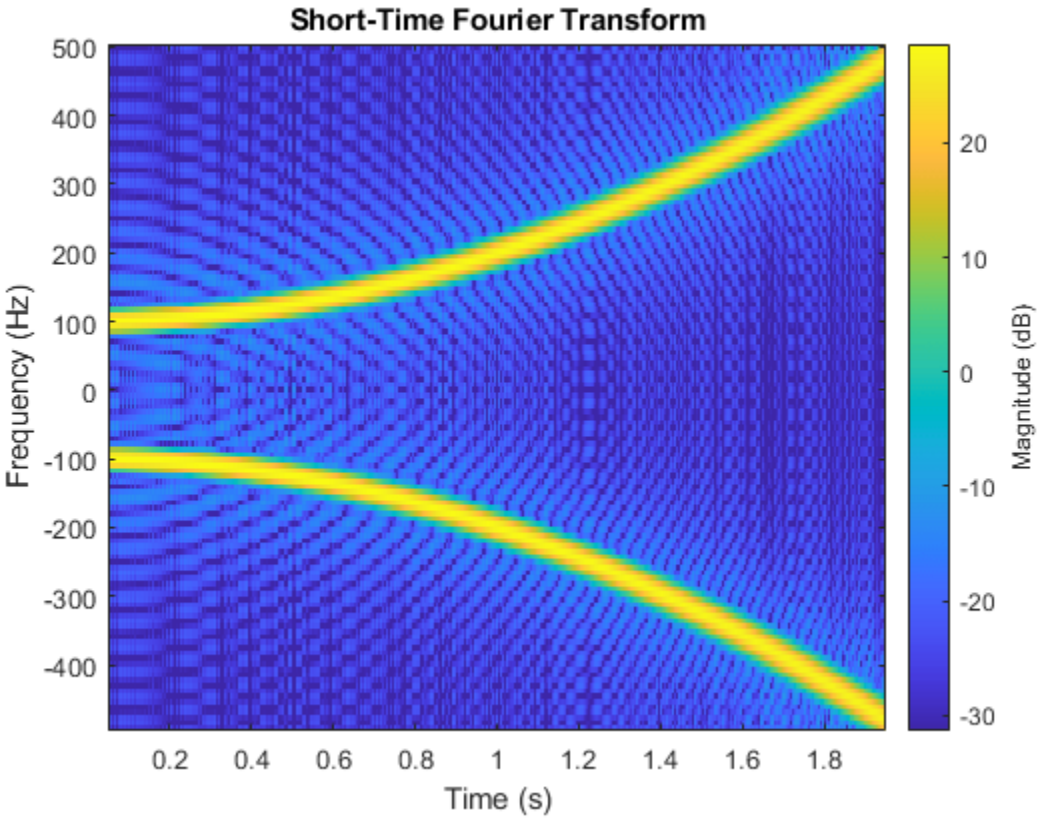
STFT of Quadratic Chirp

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

```
ts = 0:1/1e3:2;

f0 = 100;
f1 = 200;

x = chirp(ts,f0,1,f1,'quadratic',[],'concave');
```

Compute and display the STFT of the quadratic chirp with a duration of 1 ms.

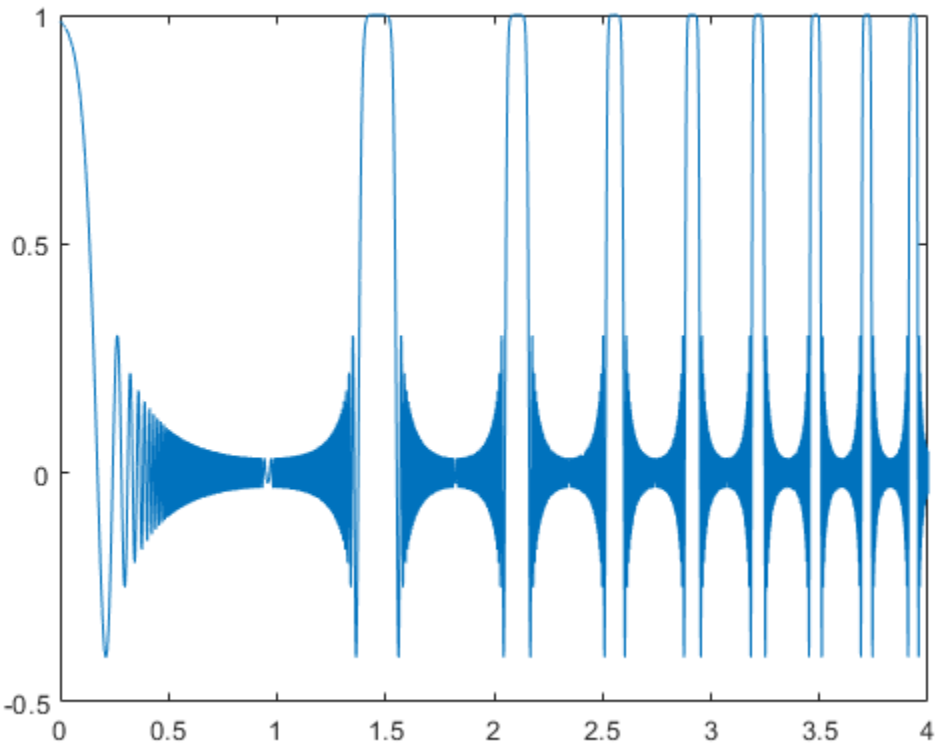
```
d = seconds(1e-3);
win = hamming(100,'periodic');

stft(x,d,'Window',win,'OverlapLength',98,'FFTLength',128);
```



STFT Frequency Ranges

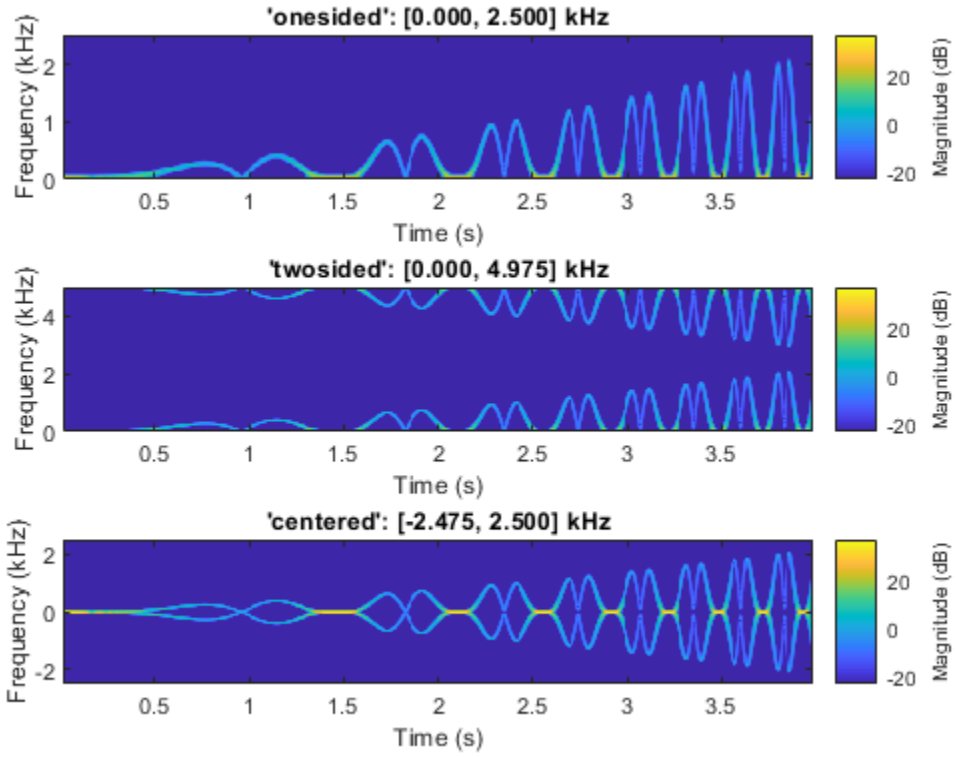
Generate a signal sampled at 5 kHz for 4 seconds. The signal consists of a set of pulses of decreasing duration separated by regions of oscillating amplitude and fluctuating frequency with an increasing trend. Plot the signal.

```
fs = 5000;
t = 0:1/fs:4-1/fs;

x = besselj(0,600*(sin(2*pi*(t+1).^3/30).^5));

plot(t,x)
```



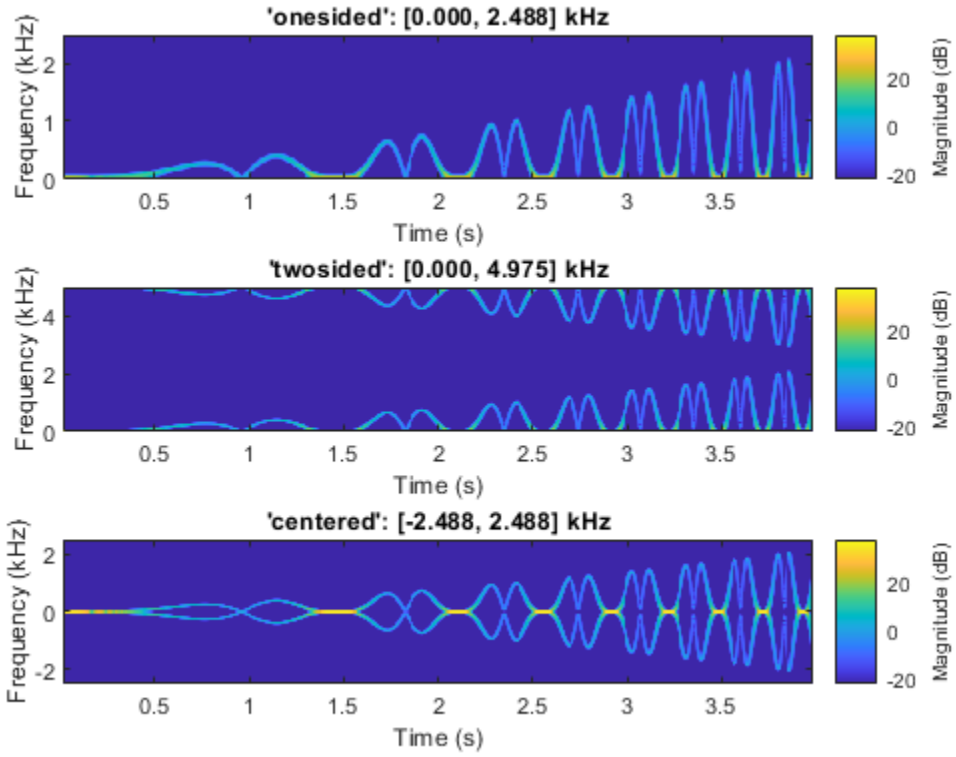
Compute the one-sided, two-sided, and centered short-time Fourier transforms of the signal. In all cases, use a 202-sample Kaiser window with shape factor $\beta = 10$ to window the signal segments. Display the frequency range used to compute each transform.

```
rngs = ["onesided" "twosided" "centered"];

for kj = 1:length(rngs)

    opts = {'Window',kaiser(202,10),'FrequencyRange',rngs(kj)};

    [~,f] = stft(x,fs,opts{:});
    subplot(length(rngs),1,kj)
    stft(x,fs,opts{:})
    title(sprintf('''%s'': [%5.3f, %5.3f] kHz',rngs(kj),[f(1) f(end)]/1000))

end
```



Repeat the computation, but now change the length of the Kaiser window to 203, an odd number. The `'twosided'` frequency interval does not change. The other two frequency intervals become open at the higher end.

```
for kj = 1:length(rngs)

    opts = {'Window',kaiser(203,10),'FrequencyRange',rngs(kj)};

    [~,f] = stft(x,fs,opts{:});
    subplot(length(rngs),1,kj)
    stft(x,fs,opts{:})
    title(sprintf('''%s'': [%5.3f, %5.3f] kHz',rngs(kj),[f(1) f(end)]/1000))

end
```



STFT of Multichannel Signals

Generate a three-channel signal consisting of three different chirps sampled at 1 kHz for one second.

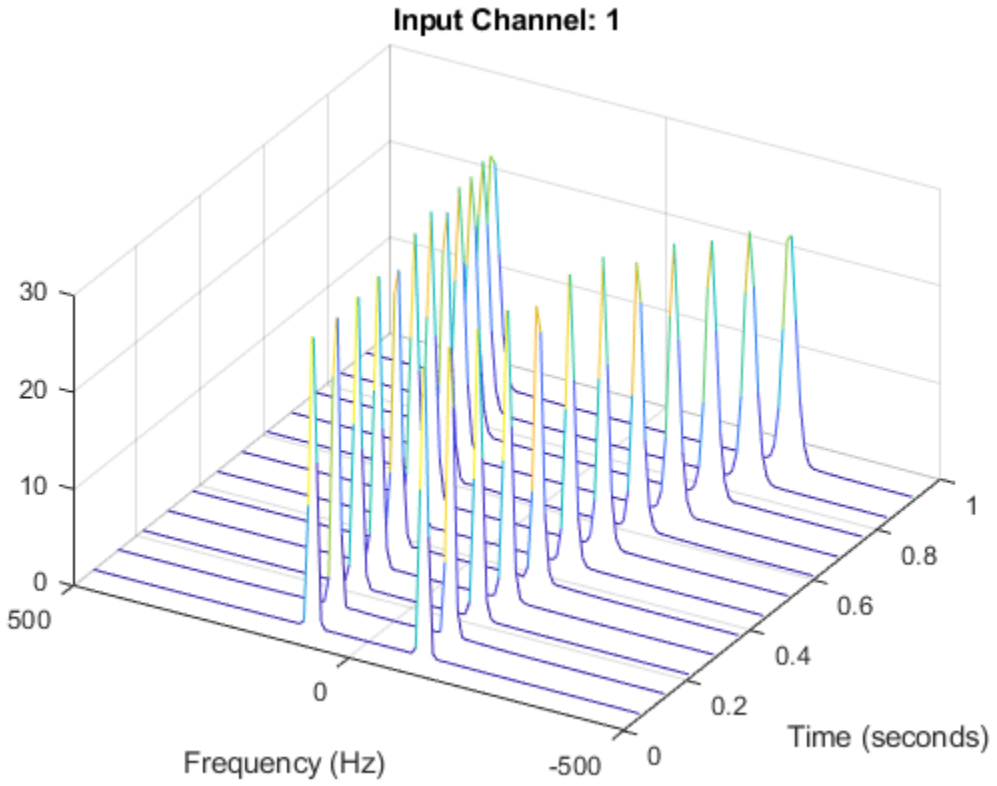
1. The first channel consists of a concave quadratic chirp with instantaneous frequency 100 Hz at $t = 0$ and crosses 300 Hz at $t = 1$ second. It has an initial phase equal to 45 degrees.
2. The second channel consists of a convex quadratic chirp with instantaneous frequency 100 Hz at $t = 0$ and crosses 500 Hz at $t = 1$ second.
3. The third channel consists of a logarithmic chirp with instantaneous frequency 300 Hz at $t = 0$ and crosses 500 Hz at $t = 1$ second.

Compute the STFT of the multichannel signal using a periodic Hamming window of length 128 and an overlap length of 50 samples.

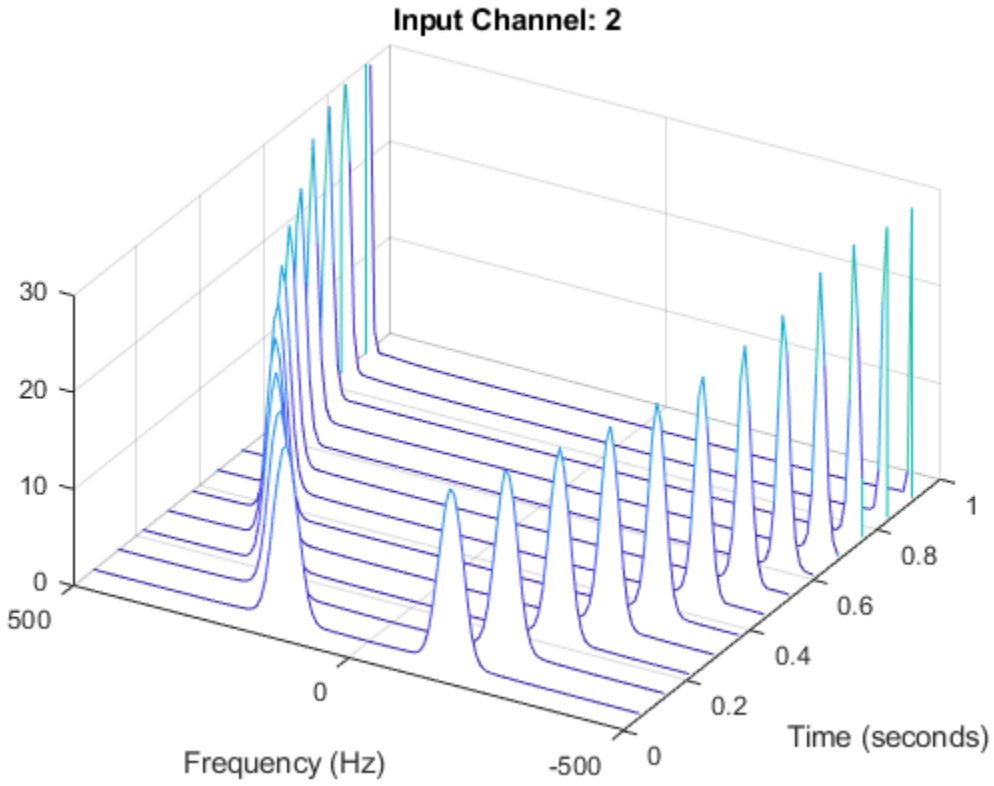
```
fs = 1e3;
t = 0:1/fs:1-1/fs;

x = [chirp(t,100,1,300,'quadratic',45,'concave');
      chirp(t,100,1,500,'quadratic',[],'convex');
      chirp(t,300,1,500,'logarithmic')]';

[S,F,T] = stft(x,fs,'Window',hamming(128,'periodic'),'OverlapLength',50);
```

Visualize the STFT of each channel as a waterfall plot. Control the behavior of the axes using the helper function `helperGraphicsOpt`.

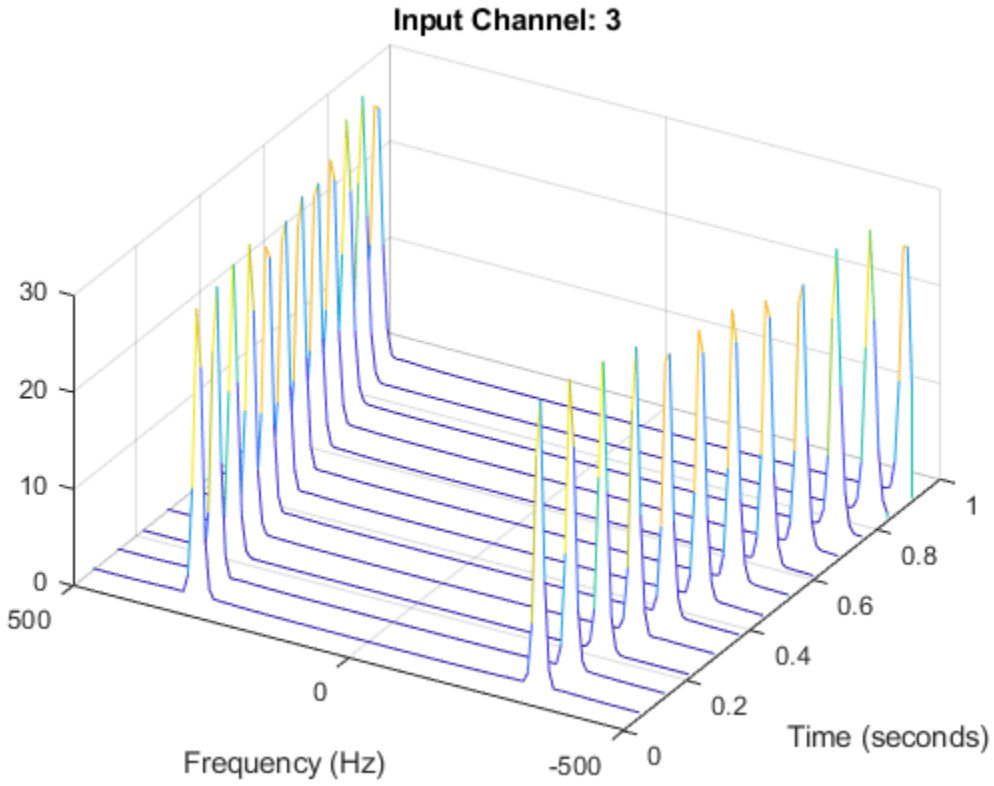
```
waterfall(F,T,abs(S(:,:,1))')
helperGraphicsOpt(1)
```




```
waterfall(F,T,abs(S(:,:,2))')
helperGraphicsOpt(2)
```




```
waterfall(F,T,abs(S(:,:,3))')
helperGraphicsOpt(3)
```



This helper function sets the appearance and behavior of the current axes.

```
function helperGraphicsOpt(ChannelId)
ax = gca;
ax.XDir = 'reverse';
ax.ZLim = [0 30];
ax.Title.String = ['Input Channel: ' num2str(ChannelId)];
ax.XLabel.String = 'Frequency (Hz)';
ax.YLabel.String = 'Time (seconds)';
ax.View = [30 45];
end
```

Input Arguments

x — Input signal

vector | matrix | timetable

Input signal, specified as a vector, a matrix, or a MATLAB `timetable`.

Note If you invert `s` using `istft` and want the result to be the same length as `x`, the value of `(length(x)-noverlap)/(length(window)-noverlap)` must be an integer.

- If the input has multiple channels, specify `x` as a matrix where each column corresponds to a channel.

- For timetable input, `x` must contain uniformly increasing finite row times. If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.
- For multichannel timetable input, specify `x` as a timetable with a single variable containing a matrix or a timetable with multiple variables each containing a column vector. All variables must have the same precision.

Each channel of `x` must have a length greater than or equal to the window length.

Example: `chirp(0:1/4e3:2,250,1,500,'quadratic')` specifies a single-channel chirp.

Example: `timetable(rand(5,2),'SampleRate',1)` specifies a two-channel random variable sampled at 1 Hz for 4 seconds.

Example: `timetable(rand(5,1),rand(5,1),'SampleRate',1)` specifies a two-channel random variable sampled at 1 Hz for 4 seconds.

Data Types: `double` | `single`
Complex Number Support: Yes

`fs` — Sample rate

2π (default) | positive scalar

Sample rate, specified as a positive scalar. This argument applies only when `x` is a vector or a matrix.

Data Types: `double` | `single`

`ts` — Sample time

duration scalar

Sample time, specified as a `duration` scalar. This argument applies only when `x` is a vector or a matrix

Example: `seconds(1)` is a `duration` scalar representing a 1-second time difference between consecutive signal samples.

Data Types: `duration`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Window',hamming(100),'OverlapLength',50,'FFTLength',128` windows the data using a 100-sample Hamming window, with 50 samples of overlap between adjoining segments and a 128-point FFT.

`Window` — Spectral window

`hann(128,'periodic')` (default) | vector

Spectral window, specified as a vector. If you do not specify the window or specify it as empty, the function uses a Hann window of length 128. The length of `'Window'` must be greater than or equal to 2.

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: `double` | `single`

`OverlapLength` — Number of overlapped samples
75% of window length (default) | nonnegative integer

Number of overlapped samples, specified as a positive integer smaller than the length of `'Window'`. If you omit `'OverlapLength'` or specify it as empty, it is set to the largest integer less than 75% of the window length, which is 96 samples for the default Hann window.

Data Types: `double` | `single`

`FFTLength` — Number of DFT points
128 (default) | positive integer

Number of DFT points, specified as a positive integer. The value must be greater than or equal to the window length. If the length of the input signal is less than the DFT length, the data is padded with zeros.

Data Types: `double` | `single`

`FrequencyRange` — STFT frequency range
`'centered'` (default) | `'twosided'` | `'onesided'`

STFT frequency range, specified as `'centered'`, `'twosided'`, or `'onesided'`.

- `'centered'` — Compute a two-sided, centered STFT. If `'FFTLength'` is even, then `s` is computed over the interval $(-\pi, \pi]$ rad/sample. If `'FFTLength'` is odd, then `s` is computed over the interval $(-\pi, \pi)$ rad/sample. If you specify time information, then the intervals are $(-f_s, f_s/2]$ cycles/unit time and $(-f_s, f_s/2)$ cycles/unit time, respectively, where f_s is the effective sample rate.
- `'twosided'` — Compute a two-sided STFT over the interval $[0, 2\pi)$ rad/sample. If you specify time information, then the interval is $[0, f_s)$ cycles/unit time.
- `'onesided'` — Compute a one-sided STFT. If `'FFTLength'` is even, then `s` is computed over the interval $[0, \pi]$ rad/sample. If `'FFTLength'` is odd, then `s` is computed over the interval $[0, \pi)$ rad/sample. If you specify time information, then the intervals are $[0, f_s/2]$ cycles/unit time and $[0, f_s/2)$ cycles/unit time, respectively, where f_s is the effective sample rate. This option is valid only for real signals.

  Note When this argument is set to `'onesided'`, `stft` outputs the values in the positive Nyquist range and does not conserve the total power.

For an example, see “STFT Frequency Ranges” on page 1-2554.

Data Types: `char` | `string`

`OutputTimeDimension` — Output time dimension
`'acrosscolumns'` (default) | `'downrows'`

Output time dimension, specified as `'acrosscolumns'` or `'downrows'`. Set this value to `'downrows'` if you want the time dimension of `s` down the rows and the frequency dimension across the columns. Set this value to `'acrosscolumns'` if you want the time dimension of `s` across the columns and the frequency dimension down the rows. This input is ignored if the function is called without output arguments.

Output Arguments

`s` — Short-time Fourier transform

matrix | 3-D array

Short-time Fourier transform, returned as a matrix or a 3-D array. Time increases across the columns of `s` and frequency increases down the rows. The third dimension, if present, corresponds to the input channels.

- If the signal `x` has N_x time samples, then `s` has k columns, where $k = \lfloor (N_x – L)/(M – L) \rfloor$, M is the length of `'Window'`, L is the `'OverlapLength'`, and the $\lfloor\ \rfloor$ symbols denote the floor function.
- The number of rows in `s` is equal to the value specified in `'FFTLength'`.

Data Types: `double` | `single`

`f` — Frequencies

vector

Frequencies at which the STFT is evaluated, returned as a vector.

Data Types: `double` | `single`

`t` — Time instants

vector

Time instants, returned as a vector. `t` contains the time values corresponding to the centers of the data segments used to compute short-time power spectrum estimates.

- If a sample rate `fs` is provided, then the vector contains time values in seconds.
- If a sample time `ts` is provided, then the vector is a duration array with the same time format as the input.
- If no time information is provided, then the vector contains sample numbers.

Data Types: `double` | `single`

More About

Short-Time Fourier Transform

The short-time Fourier transform (STFT) is used to analyze how the frequency content of a nonstationary signal changes over time.

The STFT of a signal is calculated by sliding an analysis window of length M over the signal and calculating the discrete Fourier transform of the windowed data. The window hops over the original signal at intervals of R samples. Most window functions taper off at the edges to avoid spectral ringing. If a nonzero overlap length L is specified, overlap-adding the windowed segments compensates for the signal attenuation at the window edges. The DFT of each windowed segment is added to a matrix that contains the magnitude and phase for each point in time and frequency. The number of columns in the STFT matrix is given by

$$k = \left\lfloor \frac{N_x - L}{M - L} \right\rfloor,$$

where N_x is the length of the original signal $x(n)$ and the $\lfloor\rfloor$ symbols denote the floor function. The number of rows in the matrix equals N_{DFT}, the number of DFT points, for centered and two-sided transforms and $\lfloor N_{DFT}/2 \rfloor + 1$ for one-sided transforms.

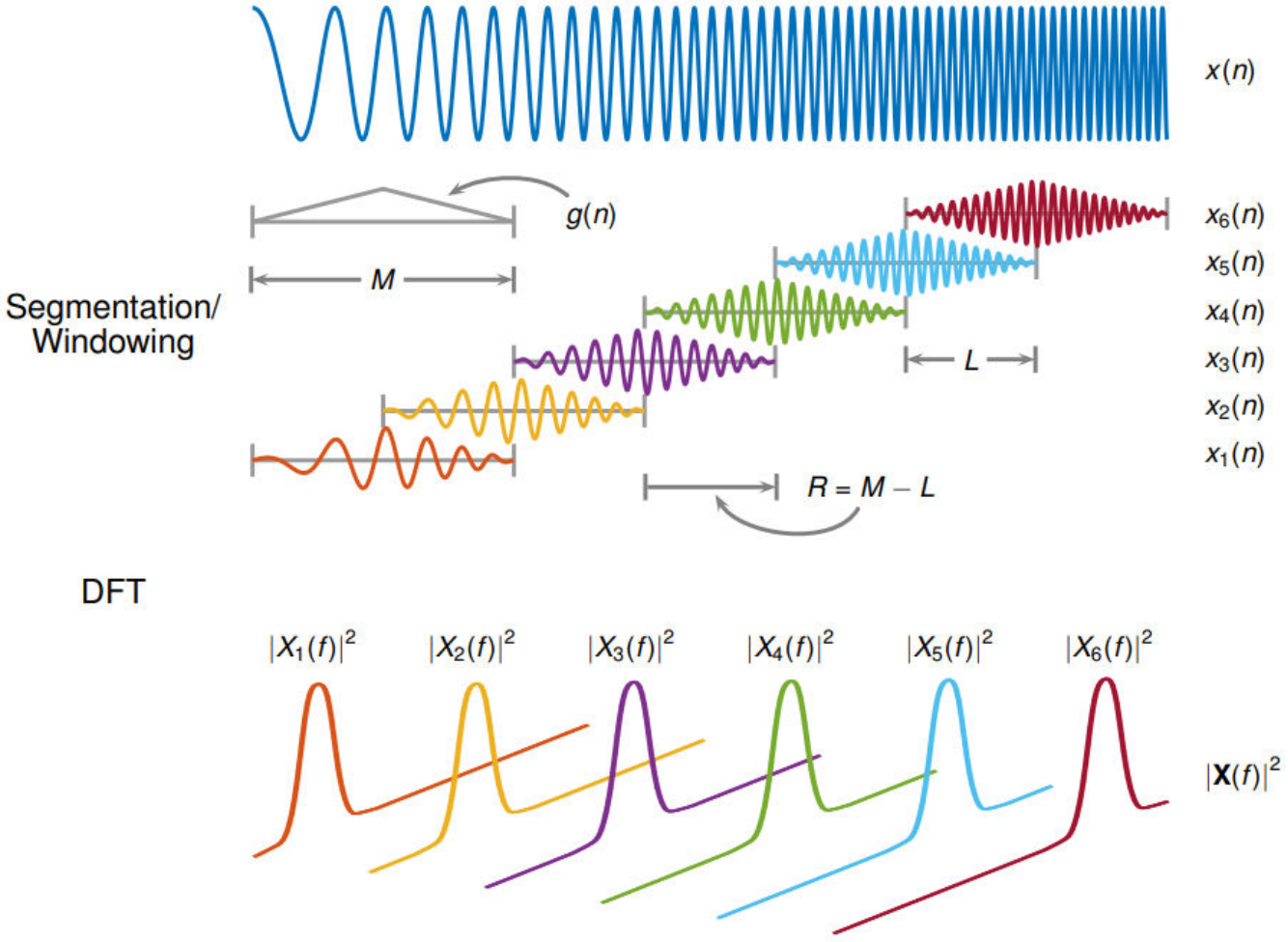
The STFT matrix is given by $\mathbf{X}(f) = \left[X_1(f)\ X_2(f)\ X_3(f)\ \cdots\ X_k(f)\right]$ such that the mth element of this matrix is

$$X_m(f) = \sum_{n=-\infty}^{\infty} x(n)g(n - mR)e^{-j2\pi fn},$$

where

- $g(n)$ — Window function of length M.
- $X_m(f)$ — DFT of windowed data centered about time mR.
- R — Hop size between successive DFTs. The hop size is the difference between the window length $M$and the overlap length L.

The magnitude squared of the STFT yields the `spectrogram` representation of the power spectral density of the function.



Perfect Reconstruction

In general, computing the STFT of an input signal and inverting it does not result in perfect reconstruction. If you want the output of ISTFT to match the original input signal as closely as possible, the signal and the window must satisfy the following conditions:

- Input size — If you invert the output of `stft` using `istft` and want the result to be the same length as the input signal `x`, the value of $k = \frac{(length(x) - noverlap)}{(length(window) - noverlap)}$ must be an integer.
- COLA compliance — Use COLA-compliant windows, assuming that you have not modified the short-time Fourier transform of the signal.
- Padding — If the length of the input signal is such that the value of k is not an integer, zero-pad the signal before computing the short-time Fourier transform. Remove the extra zeros after inverting the signal.

References

[1] Mitra, Sanjit K. *Digital Signal Processing: A Computer-Based Approach*. 2nd Ed. New York: McGraw-Hill, 2001.

[2] Sharpe, Bruce. *Invertibility of Overlap-Add Processing*. https://gauss256.github.io/blog/cola.html, accessed July 2019.

[3] Smith, Julius Orion. *Spectral Audio Signal Processing*. https://ccrma.stanford.edu/~jos/sasp/, online book, 2011 edition, accessed Nov 2018.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

`'OutputTimeDimension'` must always be specified and set to `'downrows'`.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Timetables are not supported for code generation.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

Timetables are not supported for code generation.

Thread-Based Environment

Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

Usage notes and limitations:

- The syntax with no output arguments is not supported.

For more information, see “Run MATLAB Functions in Thread-Based Environment”.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Functions
`dlstft` | `istft` | `iscola` | `pspectrum` | `stftmag2sig`

Objects
`stftLayer`

Topics
“Time-Frequency Gallery”

Introduced in R2019a

stftLayer

Short-time Fourier transform layer

Description

An STFT layer computes the short-time Fourier transform of the input. Use of this layer requires Deep Learning Toolbox.

Creation

Syntax

```
layer = stftLayer
layer = stftLayer(Name=Value)
```

Description

`layer = stftLayer` creates a “Short-Time Fourier Transform” on page 1-2572 (STFT) layer. The input to `stftLayer` must be a `dlarray` object in `"CBT"` format with a size along the time dimension greater than the length of `Window`.

`layer = stftLayer(Name=Value)` specifies optional parameters using name-value arguments. You can specify the analysis window and the format of the output, among others.

Properties

STFT

`Window` — Analysis window
`hann(128,'periodic')` (default) | vector

This property is read-only.

Analysis window used to compute the STFT, specified as a vector with two or more elements.

Example: `(1-cos(2*pi*(0:127)'/127))/2` and `hann(128)` both specify a Hann window of length 128.

Data Types: `double` | `single`

`OverlapLength` — Number of overlapped samples
`96` (default) | positive integer

This property is read-only.

Number of overlapped samples, specified as a positive integer strictly smaller than the length of `Window`.

The stride between consecutive windows is the difference between the window length and the number of overlapped samples.

Data Types: `double` | `single`

`FFTLength` — Number of DFT points
`128` (default) | positive integer

This property is read-only.

Number of frequency points used to compute the discrete Fourier transform, specified as a positive integer greater than or equal to the window length. If not specified, this argument defaults to the length of the window.

If the length of the input data along the time dimension is less than the number of DFT points, `stftLayer` right-pads the data and the window with zeros so they have a length equal to `FFTLength`.

Data Types: `double` | `single`

`TransformMode` — Layer transform mode
`"mag"` (default) | `"squaremag"` | `"logmag"` | `"logsquaremag"` | `"realimag"`

Layer transform mode, specified as one of these:

- `"mag"` — STFT magnitude
- `"squaremag"` — STFT squared magnitude
- `"logmag"` — Natural logarithm of the STFT magnitude
- `"logsquaremag"` — Natural logarithm of the STFT squared magnitude
- `"realimag"` — Real and imaginary parts of the STFT, concatenated along the channel dimension

Data Types: `char` | `string`

`OutputMode` — Layer output mode
`"spatiotemporal"` (default) | `"spatial"` | `"temporal"`

Layer output mode, specified as one of these:

- `"spatiotemporal"` — Format the output as a sequence of 1-D images where the image height corresponds to frequency, the second dimension corresponds to channel, the third dimension corresponds to batch, and the fourth dimension corresponds to time.

  You can use this output mode to feed the output of `stftLayer` to a 1-D convolutional layer when you want to convolve along frequency. For more information, see `convolution1dLayer`.
- `"spatial"` — Format the output as a sequence of 2-D images where the image height corresponds to frequency and the image width corresponds to time. The third and fourth dimensions correspond to channel and batch, respectively.

  You can use this output mode to feed the output of `stftLayer` to a 2-D convolutional layer when you want to convolve along the two spatial dimensions. For more information, see `convolution2dLayer`.
- `"temporal"` — Format the output as a 1-D sequence. This format takes the `"spatiotemporal"` output format and flattens the image height into the channel dimension. The second dimension of the STFT output corresponds to batch and the third dimension corresponds to time.

You can use this output mode to feed the output of `stftLayer` to a 1-D convolutional layer when you want to convolve along time. For more information, see `convolution1dLayer`. You can also use this output mode to use `stftLayer` as part of a recurrent neural network. For more information, see `lstmLayer` and `gruLayer`.

Data Types: `char` | `string`

Layer

`WeightLearnRateFactor` — Multiplier for weight learning rate
`0` (default) | nonnegative scalar

Multiplier for weight learning rate, specified as a nonnegative scalar. If not specified, this property defaults to zero, resulting in weights that do not update with training. You can also set this property using the `setLearnRateFactor` function.

Data Types: `double` | `single`

`Name` — Layer name
`''` (default) | character vector | string scalar

Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with name `''`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs
`1` (default)

This property is read-only.

Number of inputs of the layer. This layer accepts a single input only.

Data Types: `double`

`InputNames` — Input names
`{'in'}` (default)

This property is read-only.

Input names of the layer. This layer accepts a single input only.

Data Types: `cell`

`NumOutputs` — Number of outputs
`1` (default)

This property is read-only.

Number of outputs of the layer. This layer has a single output only.

Data Types: `double`

`OutputNames` — Output names
`{'out'}` (default)

This property is read-only.

Output names of the layer. This layer has a single output only.

Data Types: `cell`

Examples

Short-Time Fourier Transform of Chirp

Generate a signal sampled at 600 Hz for 2 seconds. The signal consists of a chirp with sinusoidally varying frequency content. Store the signal in a deep learning array with `"CTB"` format.

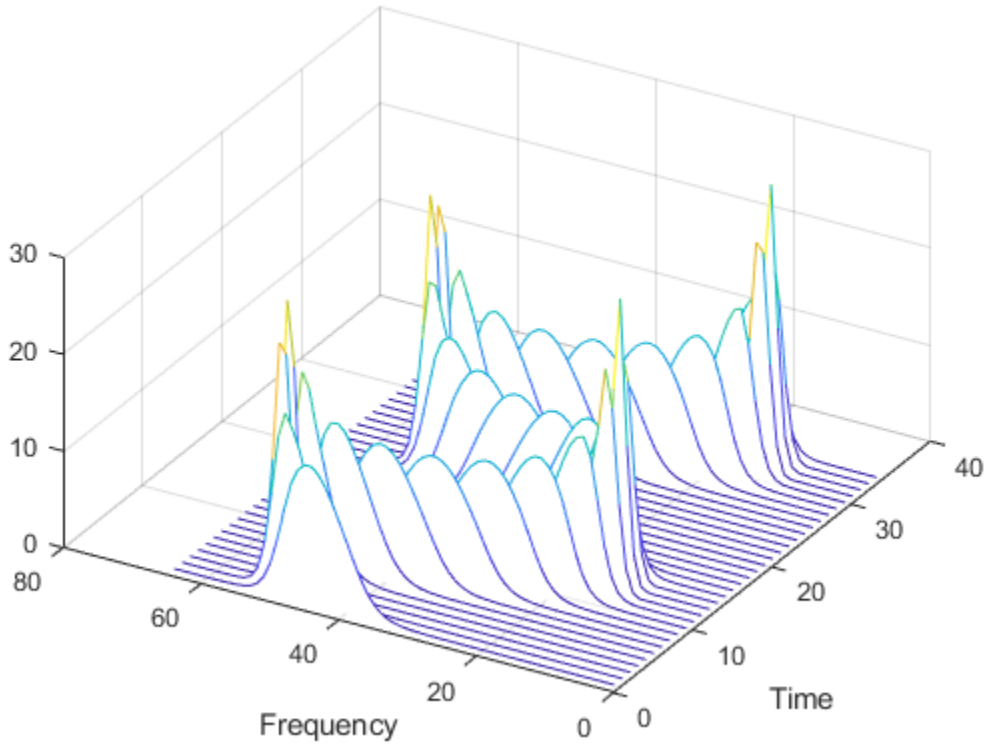
```
fs = 6e2;
x = vco(sin(2*pi*(0:1/fs:2)),[0.1 0.4]*fs,fs);

dlx = dlarray(x,"CTB");
```

Create a short-time Fourier transform layer with default properties. Create a `dlnetwork` object consisting of a sequence input layer and the short-time Fourier transform layer. Specify a minimum sequence length of 128 samples. Run the signal through the `predict` method of the network.

```
ftl = stftLayer;

dlnet = dlnetwork([sequenceInputLayer(1,MinLength=128) ftl]);
netout = predict(dlnet,dlx);
```

Convert the network output to a numeric array. Use the `squeeze` function to remove the length-1 channel and batch dimensions. Plot the magnitude of the STFT. The first dimension of the array corresponds to frequency and the second to time.

```
q = extractdata(netout);

waterfall(squeeze(q)')
set(gca,XDir="reverse",View=[30 45])
xlabel("Frequency")
ylabel("Time")
```



Short-Time Fourier Transform of Sinusoid

Generate a 3 × 160 (× 1) array containing one batch of a three-channel, 160-sample sinusoidal signal. The normalized sinusoid frequencies are $\pi/4$ rad/sample, $\pi/2$ rad/sample, and $3\pi/4$ rad/sample. Save the signal as a `dlarray`, specifying the dimensions in order. `dlarray` permutes the array dimensions to the `"CBT"` shape expected by a deep learning network.

```
nch = 3;
N = 160;
x = dlarray(cos(pi.*(1:nch)'/4*(0:N-1)),"CTB");
```

Create a short-time Fourier transform layer that can be used with the sinusoid. Specify a 64-sample rectangular window, 48 samples of overlap between adjoining windows, and 1024 DFT points. Specify the layer output mode as `"spatial"`. By default, the layer outputs the magnitude of the STFT.

```
stfl = stftLayer(Window=rectwin(64), ...
    OverlapLength=48, ...
    FFTLength=1024, ...
    OutputMode="spatial");
```

Create a two-layer `dlnetwork` object containing a sequence input layer and the STFT layer you just created. Treat each channel of the sinusoid as a feature. Specify the signal length as the minimum sequence length for the input layer.

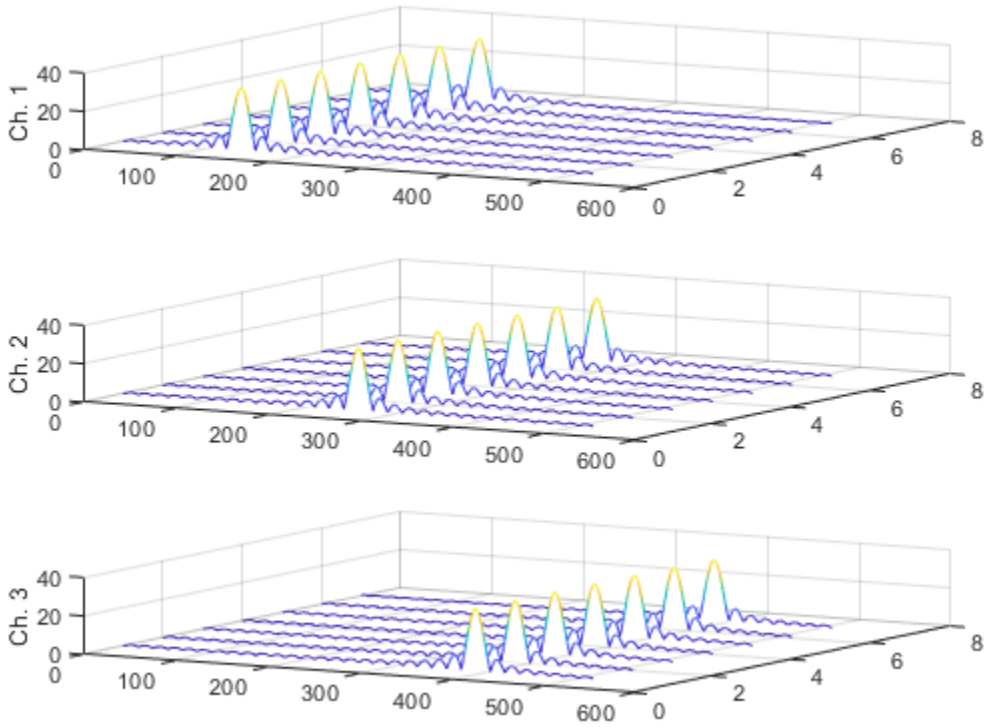
```
layers = [sequenceInputLayer(nch,MinLength=N) stfl];
dlnet = dlnetwork(layers);
```

Run the sinusoid through the `forward` method of the network.

```
dataout = forward(dlnet,x);
```

Convert the network output to a numeric array. Use the `squeeze` function to collapse the size-1 batch dimension. Plot the STFT magnitude separately for each channel in a waterfall plot.

```
q = squeeze(extractdata(dataout));

for kj = 1:nch
    subplot(nch,1,kj)
    waterfall(q(:,:,kj)')
    view(30,45)
    zlabel("Ch. "+string(kj))
end
```



More About

Short-Time Fourier Transform

The short-time Fourier transform (STFT) is used to analyze how the frequency content of a nonstationary signal changes over time.

The STFT of a signal is calculated by sliding an analysis window of length M over the signal and calculating the discrete Fourier transform of the windowed data. The window hops over the original signal at intervals of R samples. Most window functions taper off at the edges to avoid spectral ringing. If a nonzero overlap length L is specified, overlap-adding the windowed segments compensates for the signal attenuation at the window edges. The DFT of each windowed segment is added to a matrix that contains the magnitude and phase for each point in time and frequency. The number of columns in the STFT matrix is given by

$$k = \left\lfloor \frac{N_x - L}{M - L} \right\rfloor,$$

where N_x is the length of the original signal $x(n)$ and the $\lfloor \rfloor$ symbols denote the floor function. The number of rows in the matrix equals N_{DFT}, the number of DFT points, for centered and two-sided transforms and $\lfloor N_{\mathrm{DFT}}/2 \rfloor + 1$ for one-sided transforms.

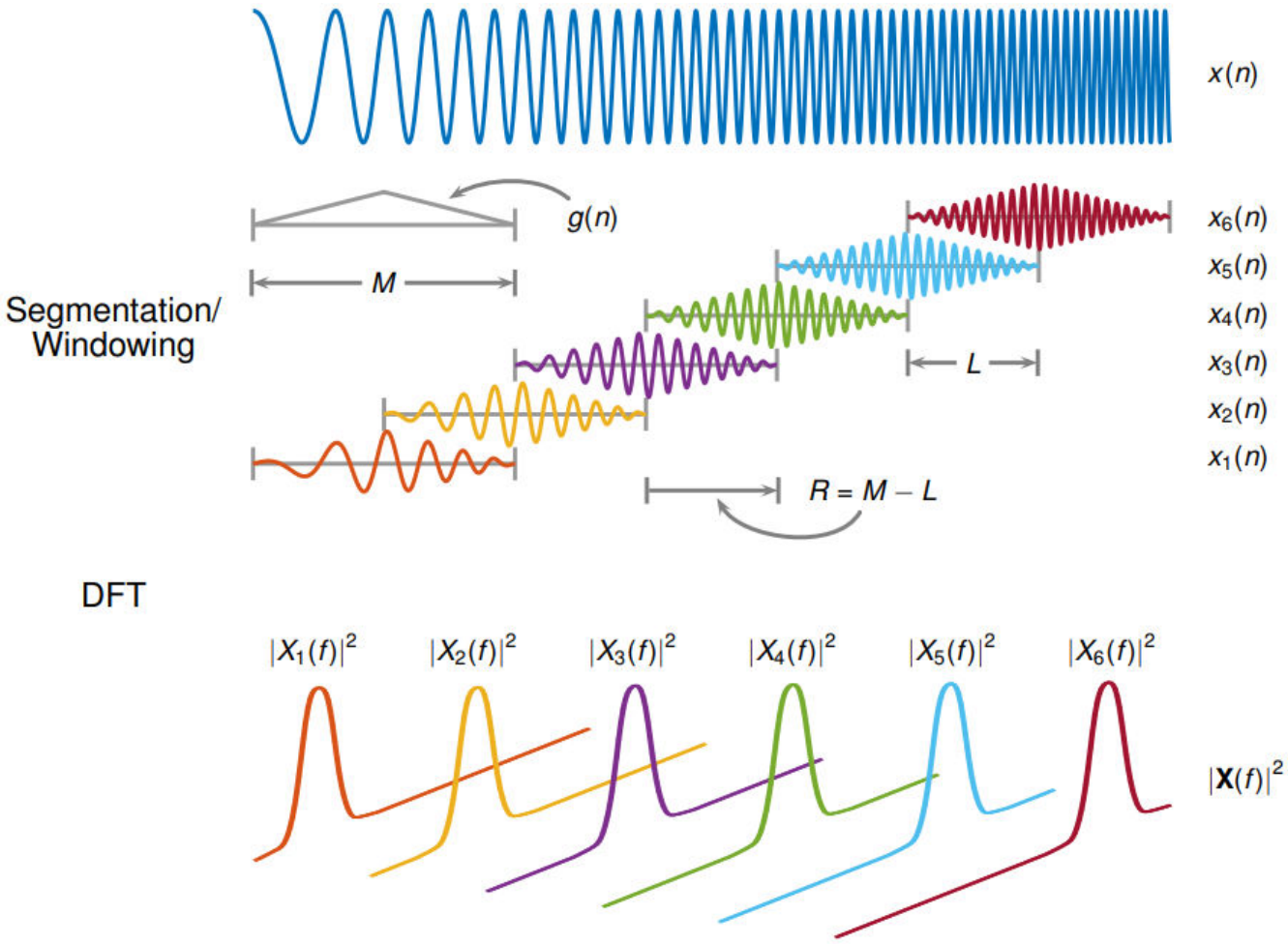
The STFT matrix is given by $\mathbf{X}(f) = \left[X_1(f)\ X_2(f)\ X_3(f)\ \cdots\ X_k(f)\right]$ such that the mth element of this matrix is

$$X_m(f) = \sum_{n=-\infty}^{\infty} x(n) g(n - mR) e^{-j2\pi f n},$$

where

- $g(n)$ — Window function of length M.
- $X_m(f)$ — DFT of windowed data centered about time mR.
- R — Hop size between successive DFTs. The hop size is the difference between the window length $M$and the overlap length L.

The magnitude squared of the STFT yields the `spectrogram` representation of the power spectral density of the function.



See Also

Apps
Deep Network Designer

Objects
`dlarray` | `dlnetwork`

Functions
`dlstft` | `stft` | `istft` | `stftmag2sig`

Topics
“List of Deep Learning Layers” (Deep Learning Toolbox)

Introduced in R2021b

stftmag2sig

Signal reconstruction from STFT magnitude

Syntax

```
x = stftmag2sig(s,nfft)
x = stftmag2sig(s,nfft,fs)
x = stftmag2sig(s,nfft,ts)

x = stftmag2sig( ___ ,Name,Value)

[x,t,info] = stftmag2sig( ___ )
```

Description

`x = stftmag2sig(s,nfft)` returns a reconstructed time-domain real signal, `x`, estimated from the "Short-Time Fourier Transform" on page 1-2585 (STFT) magnitude, `s`, based on the Griffin-Lim algorithm. The function assumes `s` was computed using discrete Fourier transform (DFT) length `nfft`.

`x = stftmag2sig(s,nfft,fs)` returns the reconstructed signal assuming that `s` was sampled at rate `fs`.

`x = stftmag2sig(s,nfft,ts)` returns the reconstructed signal assuming that `s` was sampled with sample time `ts`.

`x = stftmag2sig( ___ ,Name,Value)` specifies additional options using name-value pair arguments. Options include, among others, the FFT window and the method to specify initial phases. These arguments can be added to any of the previous input syntaxes. For example, `'FrequencyRange','onesided','InitializePhaseMethod','random'` specifies that the signal is reconstructed from a one-sided STFT with random initial phases.

`[x,t,info] = stftmag2sig( ___ )` also returns the time instants at which the signal is reconstructed and a structure that contains information about the reconstruction process.

Examples

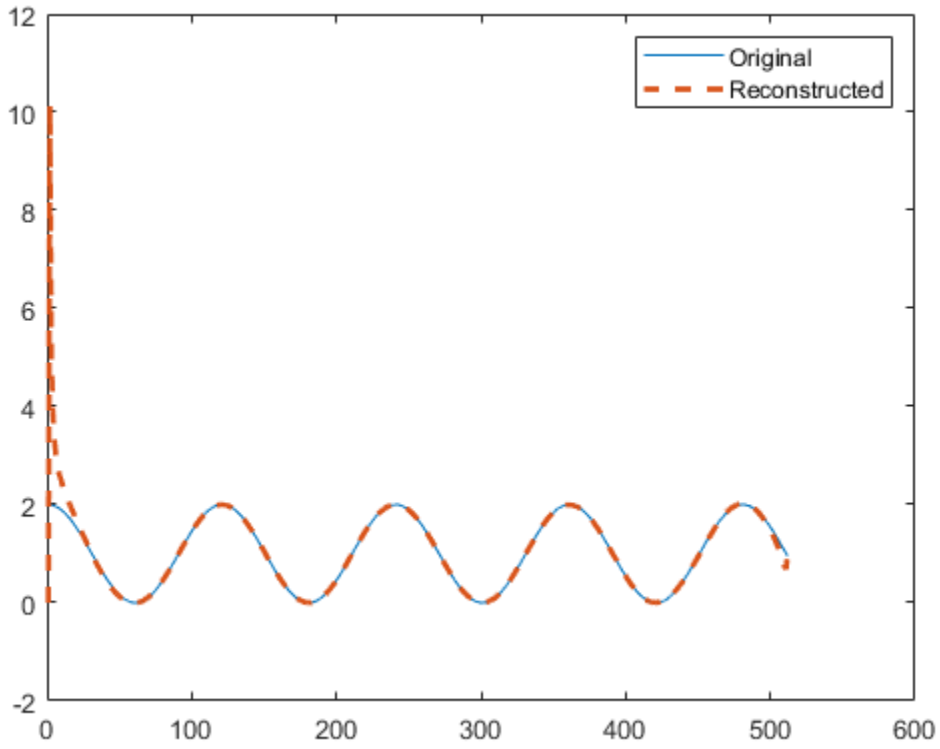
Reconstruct Sinusoid from STFT Magnitude

Consider 512 samples of a sinusoid with a normalized frequency of $\pi/60$ rad/sample and a DC value of 1. Compute the STFT of the signal.

```
n = 512;

x = cos(pi/60*(0:n-1)')+1;

S = stft(x);
```

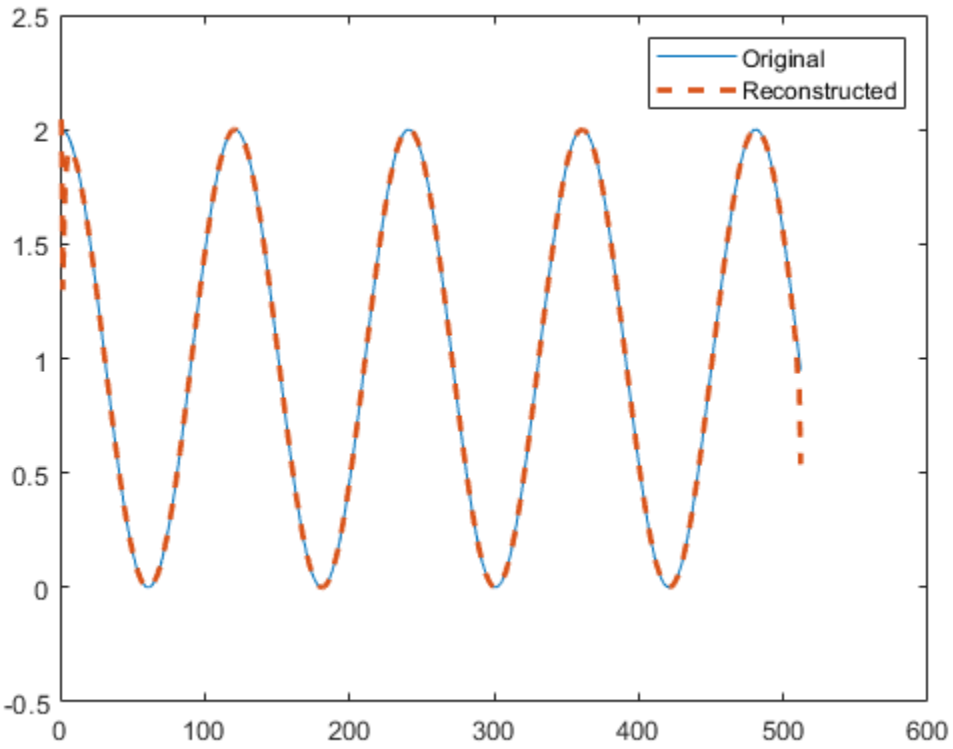
Reconstruct the sinusoid from the magnitude of the STFT. Plot the original and reconstructed signals.

```
xr = stftmag2sig(abs(S),size(S,1));

plot(x)
hold on
plot(xr,'--','LineWidth',2)
hold off
legend('Original','Reconstructed')
```



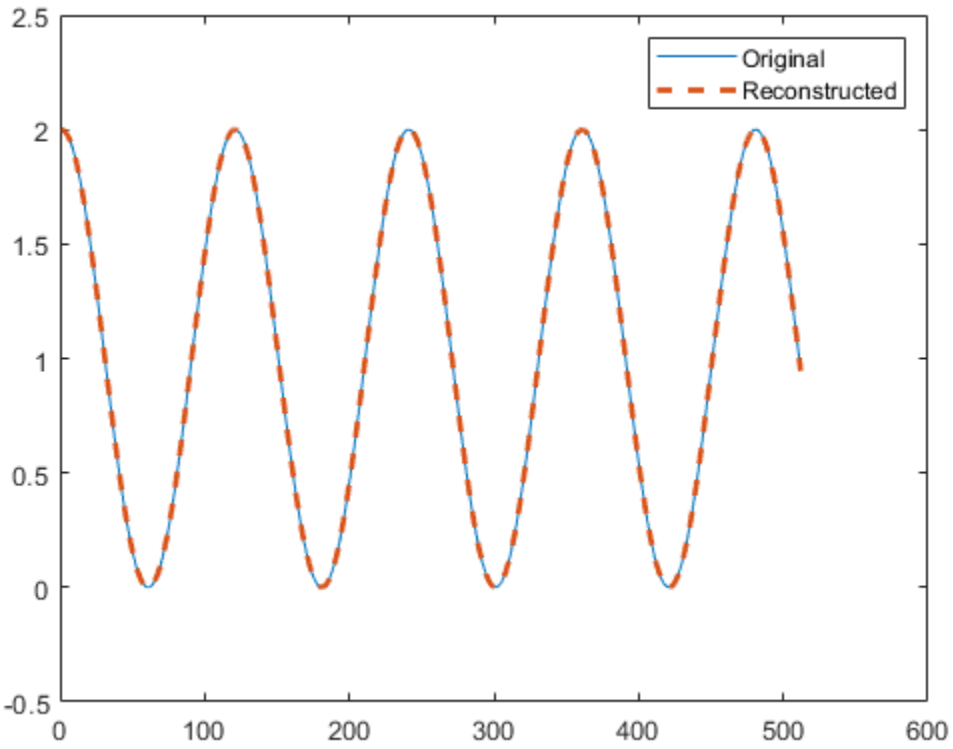
Repeat the computation, but now pad the signal with zeros to decrease edge effects.

```
xz = circshift([x; zeros(n,1)],n/2);

Sz = stft(xz);
xr = stftmag2sig(abs(Sz),size(Sz,1));

xz = xz(n/2+(1:n));
xr = xr(n/2+(1:n));

plot(xz)
hold on
plot(xr,'--','LineWidth',2)
hold off
legend('Original','Reconstructed')
```



Repeat the computation, but now decrease edge effects by assuming that `x` is a segment of a signal twice as long.

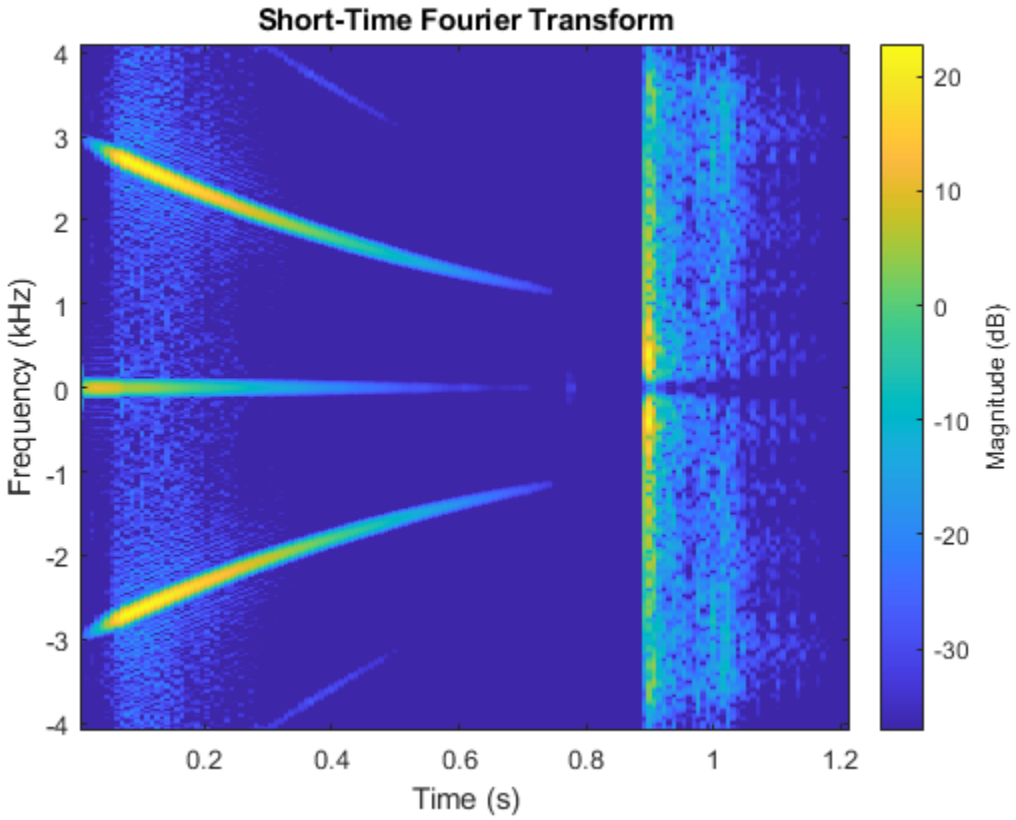
```
xx = cos(pi/60*(-n/2:n/2+n-1)')+1;

Sx = stft(xx);
xr = stftmag2sig(abs(Sx),size(Sx,1));

xx = xx(n/2+(1:n));
xr = xr(n/2+(1:n));

plot(xx)
hold on
plot(xr,'--','LineWidth',2)
hold off
legend('Original','Reconstructed')
```



Reconstruct Audio Signal from STFT Magnitude

Load an audio signal that contains two decreasing chirps and a wideband splatter sound. The signal is sampled at 8192 Hz. Plot the STFT of the signal. Divide the waveform into 128-sample segments and window the segments using a Hamming window. Specify 64 samples of overlap between adjoining segments and 1024 FFT points.

```
load splat
ty = (0:length(y)-1)/Fs;

% To hear, type sound(y,Fs)

wind = hamming(128);
olen = 64;
nfft = 1024;

stft(y,Fs,'Window',wind,'OverlapLength',olen,'FFTLength',nfft)
```



Compute the magnitude and phase of the STFT.

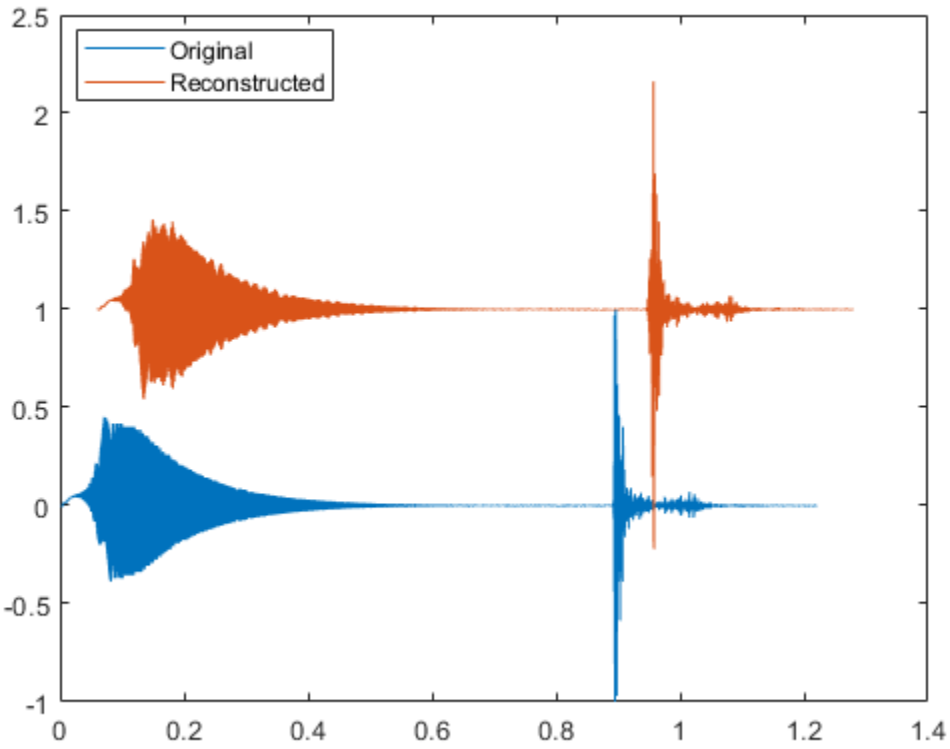
```
s = stft(y,Fs,'Window',wind,'OverlapLength',olen,'FFTLength',nfft);

smag = abs(s);
sphs = angle(s);
```

Reconstruct the signal based on the magnitude of the STFT. Use the same parameters that you used to compute the STFT. By default, `stftmag2sig` initializes the phases to zero and uses 100 optimization iterations.

```
[x,tx,info] = stftmag2sig(smag,nfft,Fs,'Window',wind,'OverlapLength',olen);

% To hear, type sound(x,Fs)
```

Plot the original and reconstructed signals. For better comparison, offset the reconstructed signal up and to the right.

```
plot(ty,y,tx+500/Fs,x+1)
legend('Original','Reconstructed','Location','best')
```



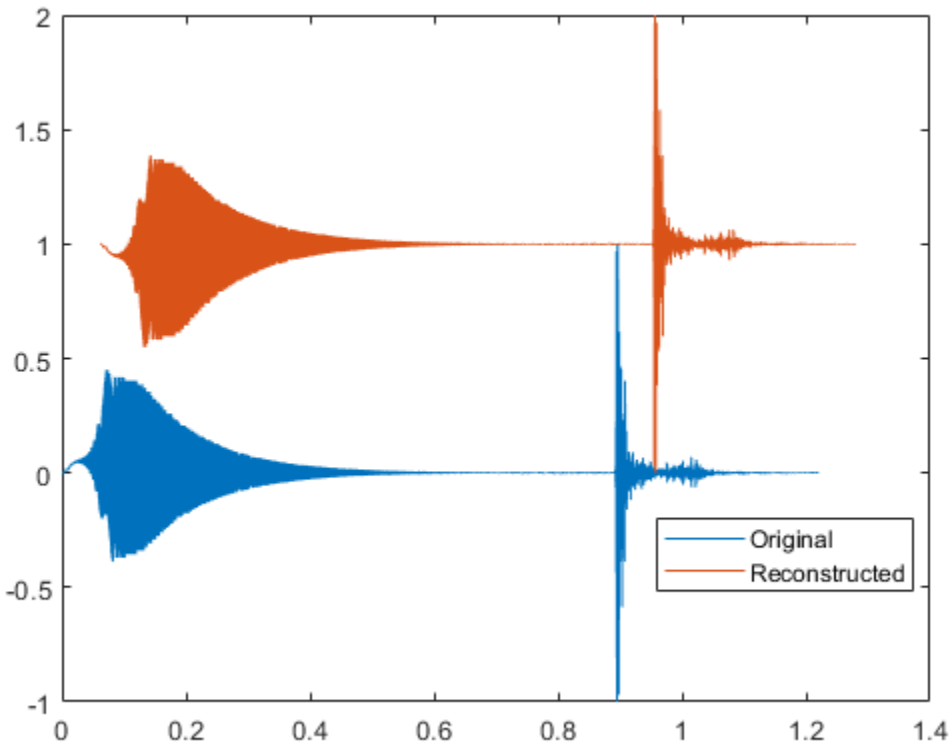
Output the relative improvement toward convergence between the last two iterations.

```
impr = info.Inconsistency
```

```
impr = 0.0424
```

Improve the reconstruction by doubling the number of optimization iterations and setting the initial phases to the actual phases from the STFT. Plot the original and reconstructed signals. For better comparison, plot the negative of the reconstructed signal and offset it up and to the right.

```
[x,tx,info] = stftmag2sig(smag,nfft,Fs,'Window',wind,'OverlapLength',olen, ...
    'MaxIterations',200,'InitialPhase',sphs);

% To hear, type sound(x,Fs)

plot(ty,y,tx+500/Fs,-x+1)
legend('Original','Reconstructed','Location','best')
```



Output the relative improvement toward convergence between the last two iterations.

```
impr = info.Inconsistency
```

```
impr = 1.3919e-16
```

Input Arguments

`s` — STFT magnitude

matrix

STFT magnitude, specified as a matrix. `s` must correspond to a single-channel, real-valued signal.

Example: `abs(stft(sin(pi/2*(0:255)),'FFTLength',128))` specifies the STFT magnitude of a sinusoid.

Example: `abs(stft(chirp(0:1/1e3:1,25,1,50)))` specifies the STFT magnitude of a chirp sampled at 1 kHz.

Data Types: `single` | `double`

`nfft` — Number of DFT points

positive integer scalar

Number of DFT points, specified as a positive integer scalar. This argument is always required.

Data Types: `single` | `double`

`fs` — Sample rate
2π (default) | positive numeric scalar

Sample rate, specified as a positive numeric scalar.

`ts` — Sample time
`duration` scalar

Sample time, specified as a `duration` scalar. Specifying `ts` is equivalent to setting a sample rate f_s = 1/`ts`.

Example: `seconds(1)` is a `duration` scalar representing a 1-second time difference between consecutive signal samples.

Data Types: `duration`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'FrequencyRange','onesided','InitializePhaseMethod','random'` specifies that the signal is reconstructed from a one-sided STFT with random initial phases.

`Display` — Inconsistency display option
`false` (default) | `true`

Inconsistency display option, specified as the comma-separated pair consisting of `'Display'` and a logical value. If this option is set to `true`, then `stftmag2sig` displays the normalized inconsistency after every 20 optimization iterations, and it also displays stopping information at the end of the run.

Data Types: `logical`

`FrequencyRange` — Frequency range of STFT magnitude
`'centered'` (default) | `'twosided'` | `'onesided'`

Frequency range of STFT magnitude, specified as the comma-separated pair consisting of `'FrequencyRange'` and `'centered'`, `'twosided'`, or `'onesided'`.

- `'centered'` — Treat `s` as the magnitude of a two-sided, centered STFT. If `nfft` is even, then `s` is taken to have been computed over the interval (–π, π] rad/sample. If `nfft` is odd, then `s` is taken to have been computed over the interval (–π, π) rad/sample. If you specify time information, then the intervals are (–f_s, f_s/2] cycles/unit time and (–f_s, f_s/2) cycles/unit time, respectively, where f_s is the sample rate.
- `'twosided'` — Treat `s` as the magnitude of a two-sided STFT computed over the interval [0, 2π) rad/sample. If you specify time information, then the interval is [0, f_s) cycles/unit time.
- `'onesided'` — Treat `s` as the magnitude of a one-sided STFT. If `nfft` is even, then `s` is taken to have been computed over the interval [0, π] rad/sample. If `nfft` is odd, then `s` is taken to have been computed over the interval [0, π) rad/sample. If you specify time information, then the intervals are [0, f_s/2] cycles/unit time and [0, f_s/2) cycles/unit time, respectively, where f_s is the sample rate.

Data Types: `char` | `string`

`InconsistencyTolerance` — Inconsistency tolerance of reconstruction process
`1e-4` (default) | positive scalar

Inconsistency tolerance of reconstruction process, specified as the comma-separated pair consisting of `'InconsistencyTolerance'` and a positive scalar. The reconstruction process stops when the "Normalized Inconsistency" on page 1-2587 is lower than the tolerance.

Data Types: `single` | `double`

`InitializePhaseMethod` — Phase initialization
`'zeros'` (default) | `'random'`

Phase initialization, specified as the comma-separated pair consisting of `'InitializePhaseMethod'` and `'zeros'` or `'random'`. Specify only one of `'InitializePhaseMethod'` or `'InitialPhase'`.

- `'zeros'` — The function initializes the phases as zeros.
- `'random'` — The function initializes the phases as random numbers distributed uniformly in the interval [–*π*, *π*].

Data Types: `char` | `string`

`InitialPhase` — Initial phases
real numeric matrix in the range [–*π*, *π*]

Initial phases, specified as the comma-separated pair consisting of `'InitialPhase'` and a real numeric matrix in the range [–*π*, *π*]. The matrix must have the same size as `s`. Specify only one of `'InitializePhaseMethod'` or `'InitialPhase'`.

Example: `angle(stft(randn(1000,1)))` specifies the phases of the short-time Fourier transform of a random signal.

Example: `2*pi*(rand(size(stft(randn(1000,1))))-1/2)` specifies a matrix of random phases distributed uniformly in the interval [–*π*, *π*]. The matrix has the same size as the short-time Fourier transform of a random signal.

.

Data Types: `single` | `double`

`InputTimeDimension` — Input time dimension
`'acrosscolumns'` (default) | `'downrows'`

Input time dimension, specified as the comma-separated pair consisting of `'InputTimeDimension'` and `'acrosscolumns'` or `'downrows'`.

- `'acrosscolumns'` — The function assumes that the time dimension of `s` is across the columns and the frequency dimension is down the rows.
- `'downrows'` — The function assumes that the time dimension of `s` is down the rows and the frequency dimension is across the columns.

Data Types: `char` | `string`

`MaxIterations` — Maximum number of optimization iterations
`100` (default) | positive integer scalar

Maximum number of optimization iterations, specified as the comma-separated pair consisting of `'MaxIterations'` and a positive integer scalar. The reconstruction process stops when the number of iterations is greater than `'MaxIterations'`.

Data Types: `single` | `double`

`Method` — Signal reconstruction algorithm
`'gla'` (default) | `'fgla'` | `'legla'`

Signal reconstruction algorithm, specified as the comma-separated pair consisting of `'Method'` and one of these:

- `'gla'` — The original reconstruction algorithm, proposed by Griffin and Lim and described in [1].
- `'fgla'` — A fast Griffin-Lim algorithm proposed by Perraudin, Balazs, and Søndergaard and described in [2].
- `'legla'` — A fast algorithm proposed by Le Roux, Kameoka, Ono, and Sagayama and described in [3].

Data Types: `char` | `string`

`OverlapLength` — Number of overlapped samples
75% of window length (default) | nonnegative integer

Number of overlapped samples between adjoining segments, specified as the comma-separated pair consisting of `'OverlapLength'` and a positive integer smaller than the length of `'Window'`. Successful signal reconstruction requires `'OverlapLength'` to match the number of overlapped segments used to generate the STFT magnitude. If you omit `'OverlapLength'` or specify it as empty, it is set to the largest integer less than or equal to 75% of the window length, which is 96 samples for the default Hann window.

Data Types: `double` | `single`

`TruncationOrder` — Truncation order for `'legla'` update rule
positive integer

Truncation order for `'legla'` update rule, specified as the comma-separated pair consisting of `'TruncationOrder'` and a positive integer. This argument applies only when `'Method'` is set to `'legla'` and controls the number of phase values updated in each iteration of that method. If not specified, `'TruncationOrder'` is determined using an adaptive algorithm.

Data Types: `single` | `double`

`UpdateParameter` — Update parameter for fast Griffin-Lim algorithm
`0.99` (default) | positive scalar

Update parameter for the fast Griffin-Lim algorithm, specified as the comma-separated pair consisting of `'UpdateParameter'` and a positive scalar. This argument applies only when `'Method'` is set to `'fgla'` and specifies the parameter for that method's update rule.

Data Types: `single` | `double`
Complex Number Support: Yes

`Window` — Spectral window
`hann(128,'periodic')` (default) | vector

Spectral window, specified as the comma-separated pair consisting of `'Window'` and a vector. Successful signal reconstruction requires `'Window'` to match the window used to generate the STFT

magnitude. If you do not specify the window or specify it as empty, the function uses a periodic Hann window of length 128. The length of `'Window'` must be greater than or equal to 2.

For a list of available windows, see “Windows”.

Example: `hann(128,'periodic')` and `(1-cos(2*pi*(128:-1:1)'/128))/2` both specify the default window used by `stftmag2sig`.

Data Types: `double` | `single`

Output Arguments

`x` — Reconstructed time-domain signal

vector

Reconstructed time-domain signal, returned as a vector.

`t` — Time instants

vector

Time instants at which the signal is reconstructed, returned as a vector.

`info` — Reconstruction process information

structure

Reconstruction process information, returned as a structure containing these fields:

- `ExitFlag` — Termination flag.
  - A value of `0` indicates the algorithm stopped when it reached the maximum number of iterations.
  - A value of `1` indicates the algorithm stopped when it met the relative tolerance.
- `NumIterations` — Total number of iterations.
- `Inconsistency` — Average relative improvement toward convergence between the last two iterations.
- `ReconstructedPhase` — Reconstructed phase at the last iteration.
- `ReconstructedSTFT` — Reconstructed short-time Fourier transform at the last iteration.

More About

Short-Time Fourier Transform

The short-time Fourier transform (STFT) is used to analyze how the frequency content of a nonstationary signal changes over time.

The STFT of a signal is calculated by sliding an analysis window of length M over the signal and calculating the discrete Fourier transform of the windowed data. The window hops over the original signal at intervals of R samples. Most window functions taper off at the edges to avoid spectral ringing. If a nonzero overlap length L is specified, overlap-adding the windowed segments compensates for the signal attenuation at the window edges. The DFT of each windowed segment is added to a matrix that contains the magnitude and phase for each point in time and frequency. The number of columns in the STFT matrix is given by

$$k = \left\lfloor \frac{N_x - L}{M - L} \right\rfloor,$$

where N_x is the length of the original signal $x(n)$ and the $\lfloor\rfloor$ symbols denote the floor function. The number of rows in the matrix equals N_{DFT}, the number of DFT points, for centered and two-sided transforms and $\lfloor N_{\mathrm{DFT}}/2 \rfloor + 1$ for one-sided transforms.

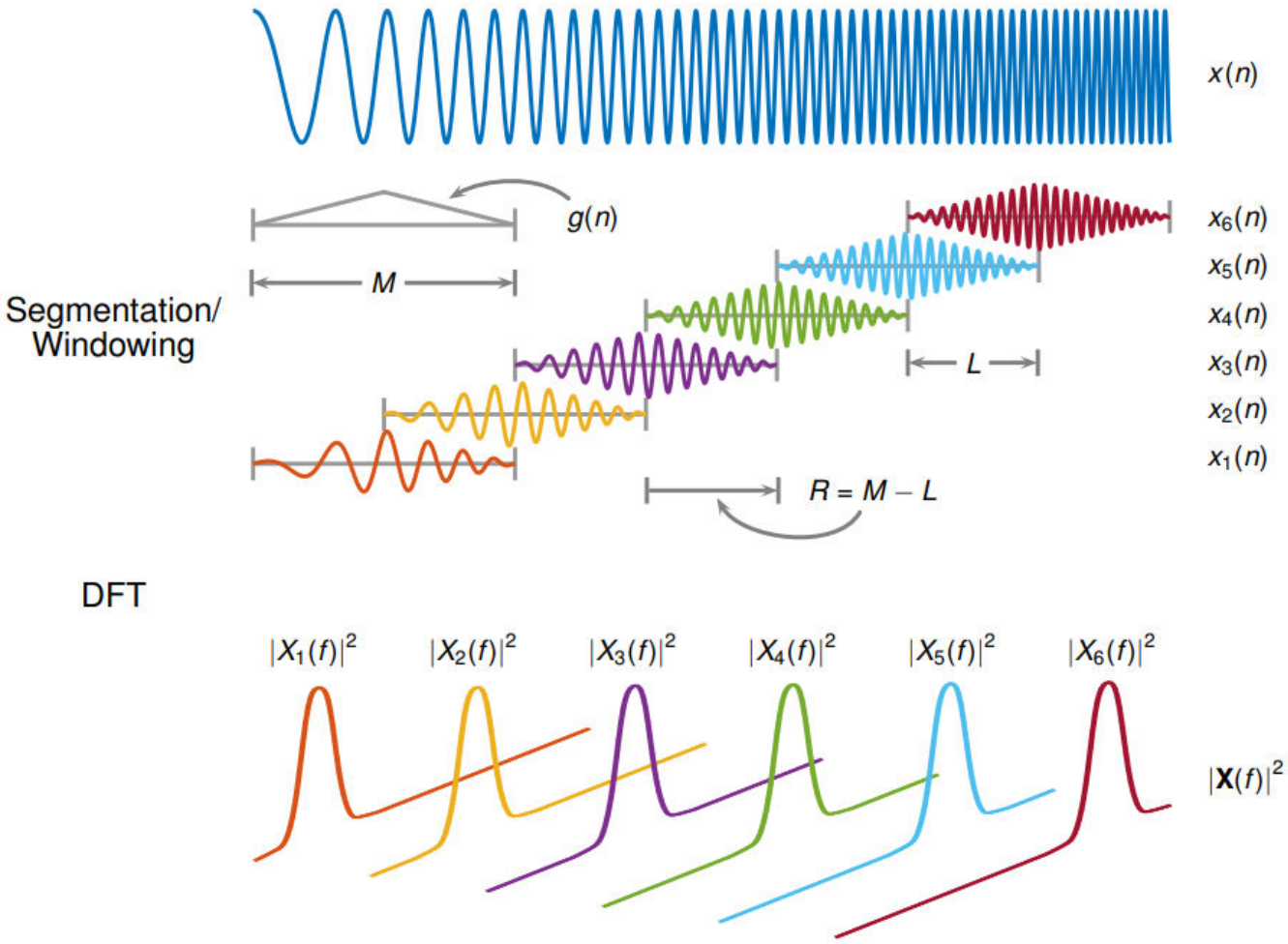
The STFT matrix is given by $\mathbf{X}(f) = \left[X_1(f)\ X_2(f)\ X_3(f)\ \cdots\ X_k(f)\right]$ such that the mth element of this matrix is

$$X_m(f) = \sum_{n=-\infty}^{\infty} x(n)g(n - mR)e^{-j2\pi fn},$$

where

- $g(n)$ — Window function of length M.
- $X_m(f)$ — DFT of windowed data centered about time mR.
- R — Hop size between successive DFTs. The hop size is the difference between the window length $M$and the overlap length L.

The magnitude squared of the STFT yields the `spectrogram` representation of the power spectral density of the function.



Normalized Inconsistency

The normalized inconsistency measures the improvement toward convergence of the reconstruction process in successive optimization iterations.

The normalized inconsistency is defined as

$$\text{Inconsistency} = \frac{\|\text{STFT}(\text{ISTFT}(s_{\text{est}})) - s_{\text{est}}\|}{\|s_{\text{est}}\|},$$

where s_{est} is the complex short-time Fourier transform estimated at each iteration, the brackets denote the matrix norm, STFT denotes the short-time Fourier transform, and ISTFT denotes its inverse. `stftmag2sig` uses the MATLAB function `norm` to compute matrix norms. For more information about the STFT and its inverse, see “Short-Time Fourier Transform” on page 1-2585 and “Inverse Short-Time Fourier Transform” on page 1-1165.

References

[1] Griffin, Daniel W., and Jae S. Lim. "Signal Estimation from Modified Short-Time Fourier Transform." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 32, Number 2, April 1984, pp. 236–243. https://doi.org/10.1109/TASSP.1984.1164317.

[2] Perraudin, Nathanaël, Peter Balazs, and Peter L. Søndergaard. "A Fast Griffin-Lim Algorithm." In *2013 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, New Paltz, NY, October 20–23, 2013. https://doi.org/10.1109/WASPAA.2013.6701851.

[3] Le Roux, Jonathan, Hirokazu Kameoka, Nobutaka Ono, and Shigeki Sagayama. "Fast Signal Reconstruction from Magnitude STFT Spectrogram Based on Spectrogram Consistency." In *Proceedings of the 13th International Conference on Digital Audio Effects (DAFx-10)*, Graz, Austria, September 6–10, 2010.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `'legla'` method is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Functions

`iscola` | `istft` | `pspectrum` | `spectrogram` | `stft`

Introduced in R2020b

stmcb

Compute linear model using Steiglitz-McBride iteration

Syntax

```
[b,a] = stmcb(h,nb,na)
[b,a] = stmcb(h,nb,na,niter)
[b,a] = stmcb(h,nb,na,niter,ai)
[b,a] = stmcb(y,x,___)
```

Description

`[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(h,nb,na,niter)` uses `niter` iterations. The default number of iterations is 5.

`[b,a] = stmcb(h,nb,na,niter,ai)` uses the vector `ai` as the initial estimate of the denominator coefficients.

`[b,a] = stmcb(y,x,___)` finds the coefficients with system output `y` and input `x` replacing `h`. `y` and `x` must be the same length.

Examples

Steiglitz-McBride Approximation of Filter

Approximate the impulse response of an IIR filter with a system of a lower order.

Specify a 6th-order Butterworth filter with normalized 3-dB frequency of 0.2π rad/sample.

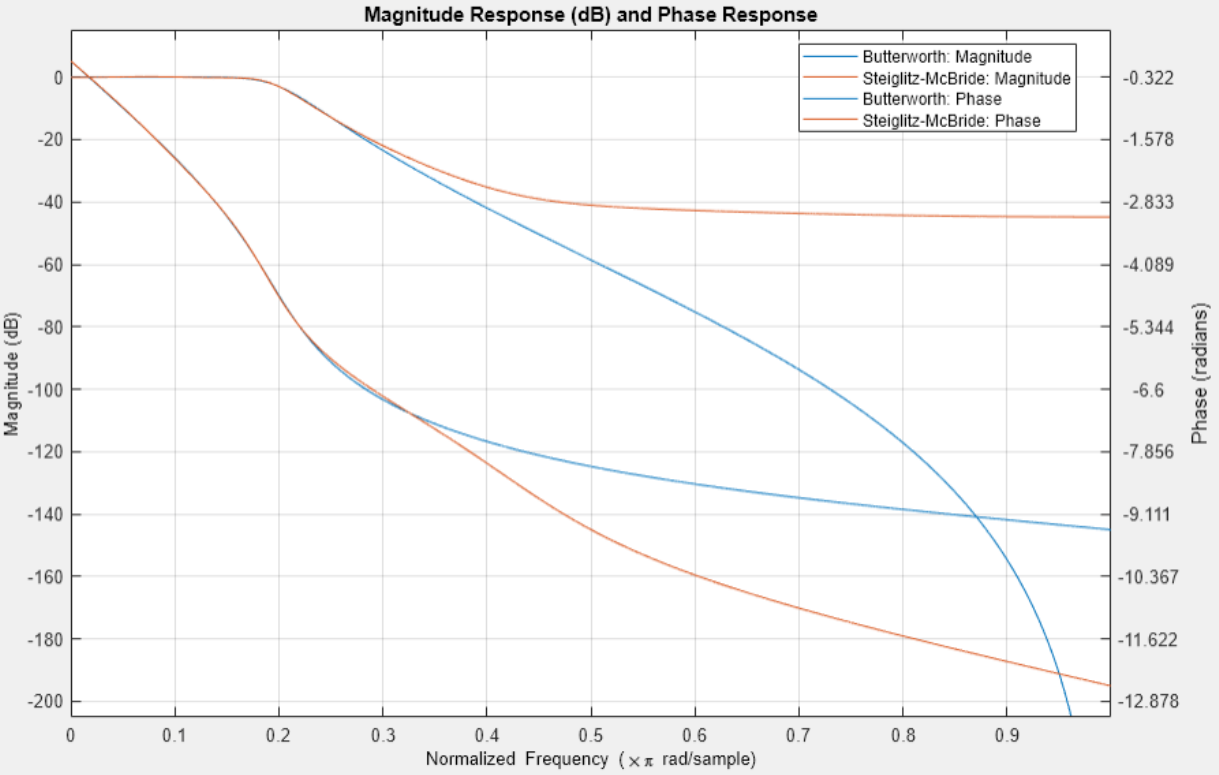
```
d = designfilt('lowpassiir','FilterOrder',6, ...
    'HalfPowerFrequency',0.2,'DesignMethod','butter');
```

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')
```



Input Arguments

`h` — Impulse response

vector

Impulse response, specified as a vector.

Data Types: `single` | `double`
Complex Number Support: Yes

`nb, na` — Numerator and denominator orders

positive integer scalars

Numerator and denominator orders, specified as positive integer scalars.

- If you want an all-pole transfer function, specify `nb` as `0`.
- If you want an all-zero transfer function, specify `na` as `0`.

Data Types: `single` | `double`

`niter` — Number of iterations

5 (default) | positive scalar

Number of iterations, specified as a positive scalar.

`ai` — Estimate of denominator coefficients
vector

Initial estimate of denominator coefficients, specified as a vector. If not specified, the `stmcb` function uses the output of `prony` with the order of the numerator set to `0`.

Data Types: `single` | `double`
Complex Number Support: Yes

`y` — Output signal
vector

Output signal of the system, specified as a vector.

Data Types: `single` | `double`
Complex Number Support: Yes

`x` — Input signal
vector

Input signal of the system, specified as a vector.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`b`, `a` — IIR filter coefficients
row vectors

IIR filter coefficients, returned as row vectors. `b` has length `nb` + 1 and `a` has length `na` + 1. The filter coefficients are ordered in descending powers of z.

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

Algorithms

The `stmcb` function attempts to minimize the squared error between the impulse response h of $b(z)/a(z)$ and the input signal x.

$$\min_{a,b} \sum_{i=0}^{\infty} |x(i) - h(i)|^2$$

The function 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 \.

The function repeats this process `niter` times. The function does not check to see if the `b` and `a` coefficients have converged in fewer than `niter` iterations.

References

[1] Steiglitz, K., and L. McBride. “A Technique for the Identification of Linear Systems.” IEEE *Transactions on Automatic Control* 10, no. 4 (October 1965): 461–64. https://doi.org/10.1109/TAC.1965.1098181.

[2] Ljung, Lennart. *System Identification: Theory for the User*. 2nd ed. Prentice Hall Information and System Sciences Series. Upper Saddle River, NJ: Prentice Hall PTR, 1999.

See Also

`levinson` | `lpc` | `aryule` | `prony`

Introduced before R2006a

strips

Strip plot

Syntax

```
strips(x)
strips(x,n)
strips(x,sd,fs)
strips(x,sd,fs,scale)
```

Description

`strips(x)` plots `x` in horizontal strips of length 250.

`strips(x,n)` plots `x` in strips that are each `n` samples long.

`strips(x,sd,fs)` plots `x` in strips of duration `sd` given the sample rate of `fs` samples per second.

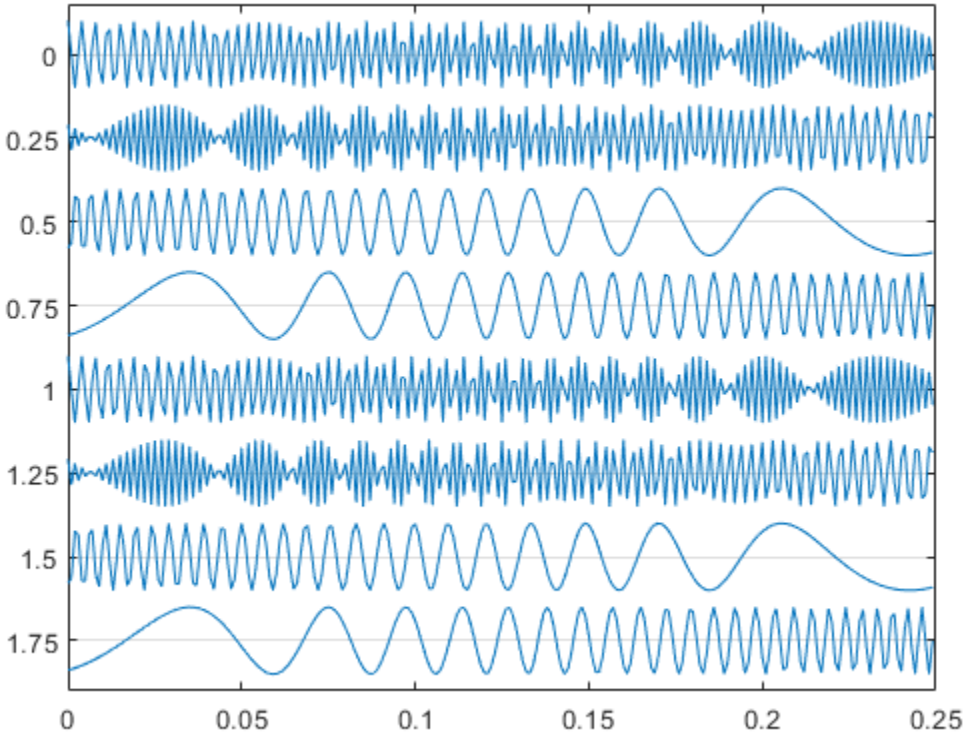
`strips(x,sd,fs,scale)` also scales the vertical axes.

Examples

Strip Plot of Frequency-Modulated Sinusoid

Plot two seconds of a frequency-modulated sinusoid in 0.25-second strips. Specify a sample rate of 1 kHz.

```
fs = 1000;
t = 0:1/fs:2;
x = vco(sin(2*pi*t),[10 490],fs);

strips(x,0.25,fs)
```



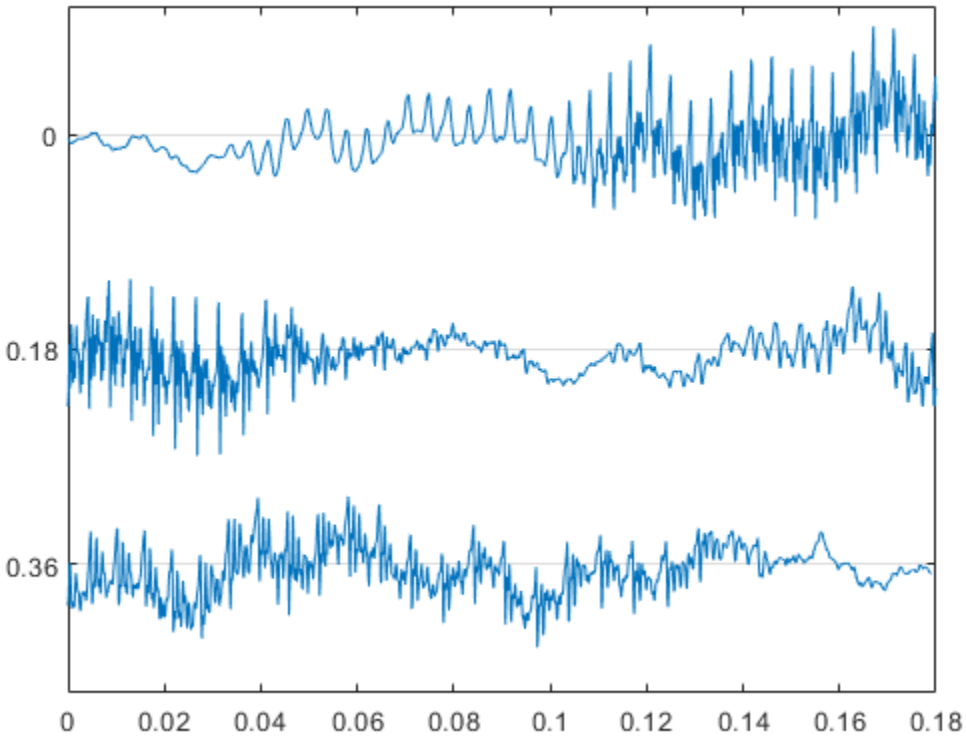
Strip Plot of Speech Signal

Load a speech signal sampled at F_s = 7418 Hz. The file contains the recording of a female voice saying the word "MATLAB®."

```
load mtlb
```

Plot the signal in 0.18-second long strips. Scale the vertical axes to 125%.

```
strips(mtlb,0.18,Fs,1.25)
```



Input Arguments

x — Input signal
vector | matrix

Input signal, specified as a vector or matrix. If `x` is a matrix, the `strips` function plots each column of `x` as a horizontal strip on the same plot. The function ignores the imaginary part of complex-valued `x`.

Data Types: `single` | `double`
Complex Number Support: Yes

n — Length
`250` (default) | real positive scalar

Length of strips, specified as a real positive scalar.

sd — Duration
real positive scalar

Duration in seconds, specified as a real positive scalar. If `sd` is specified, then you must also specify `fs`.

fs — Sample rate
real positive scalar

Sample rate, specified as a real positive scalar. `fs` has units of hertz.

`scale` — Scale factor
scalar

Scale factor, specified as a scalar. The `strips` function ignores the imaginary part of complex-valued `scale`.

Data Types: `single` | `double`
Complex Number Support: Yes

See Also

`plot` | `stem`

Introduced before R2006a

taylorwin

Taylor window

Syntax

```
w = taylorwin(L)
w = taylorwin(L,nbar)
w = taylorwin(L,nbar,sll)
```

Description

`w = taylorwin(L)` returns an `L`-point Taylor window.

`w = taylorwin(L,nbar)` returns an `L`-point Taylor window with a number (`nbar`) of nearly constant-level sidelobes adjacent to the mainlobe.

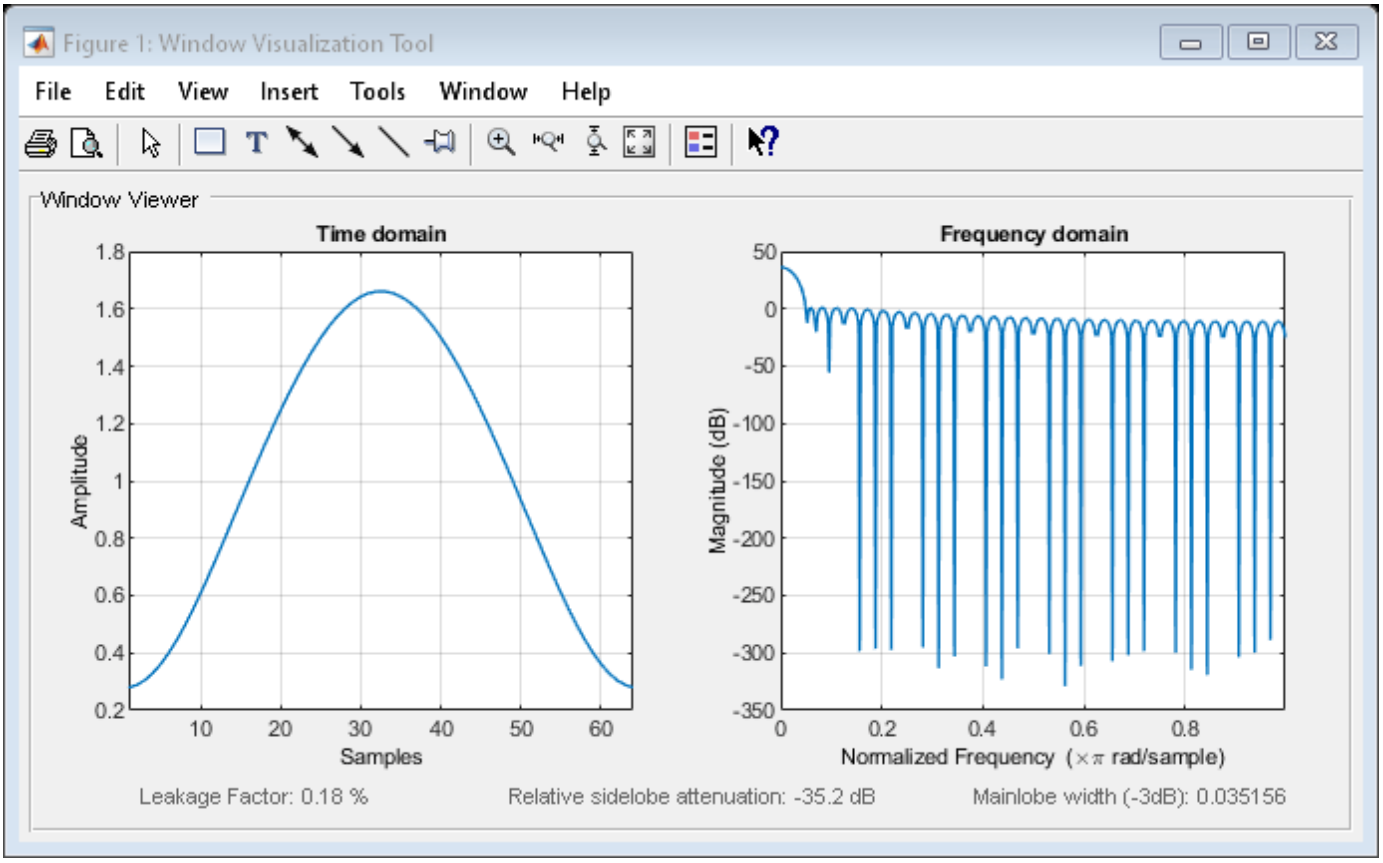
`w = taylorwin(L,nbar,sll)` returns an `L`-point Taylor window with a maximum sidelobe level of `sll` dB relative to the mainlobe peak.

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)
```



Input Arguments

L — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

nbar — Number of constant level sidelobes

4 (default) | positive integer

Number of nearly constant-level sidelobes adjacent to the mainlobe, specified as a positive integer. These sidelobes are “nearly constant-level” because some decay occurs in the transition region.

sll — Maximum sidelobe level relative to mainlobe peak

`-30` (default) | real negative scalar

Maximum sidelobe level relative to mainlobe peak, specified as a real negative scalar in dB. It produces sidelobes with peaks `sll` dB down below the mainlobe peak.

Output Arguments

`w` — Taylor window
column vector

Taylor window, returned as a column vector.

Algorithms

Taylor windows are similar to Chebyshev windows. A Chebyshev window has the narrowest possible mainlobe for a specified sidelobe level, but 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.

References

[1] Brookner, Eli. *Practical Phased Array Antenna Systems*. Boston: Artech House, 1991.

[2] Carrara, Walter G., Ronald M. Majewski, and Ron S. Goodman. *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms*. Boston: Artech House, 1995, Appendix D.2.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Apps
Window Designer

Functions
`blackman` | **WVTool** | `hamming` | `hann`

Introduced in R2006a

tachorpm

Extract RPM signal from tachometer pulses

Syntax

```
rpm = tachorpm(x,fs)
[rpm,t,tp] = tachorpm(x,fs)

[ ___ ] = tachorpm(x,fs,Name,Value)

tachorpm( ___ )
```

Description

`rpm = tachorpm(x,fs)` extracts a rotational speed signal, `rpm`, from a tachometer pulse signal vector, `x`, that has been sampled at a rate of `fs` Hz.

If you do not have a tachometer pulse signal, use `rpmtrack` to extract `rpm` from a vibration signal.

`[rpm,t,tp] = tachorpm(x,fs)` also returns the time vector, `t`, and the detected pulse locations, `tp`.

`[ ___ ] = tachorpm(x,fs,Name,Value)` specifies options using `Name,Value` pairs and any of the previous syntaxes.

`tachorpm( ___ )` with no output arguments plots the generated RPM signal and the tachometer signal with the detected pulses.

Examples

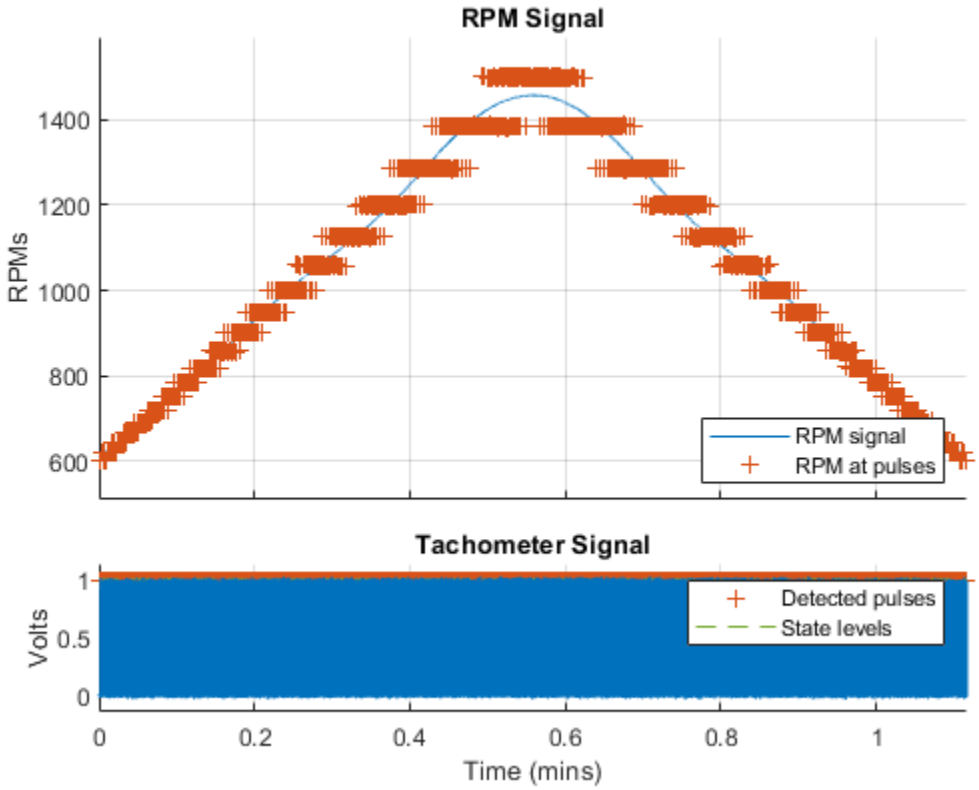
RPM Peak

Load a simulated tachometer signal sampled at 300 Hz.

```
load tacho
```

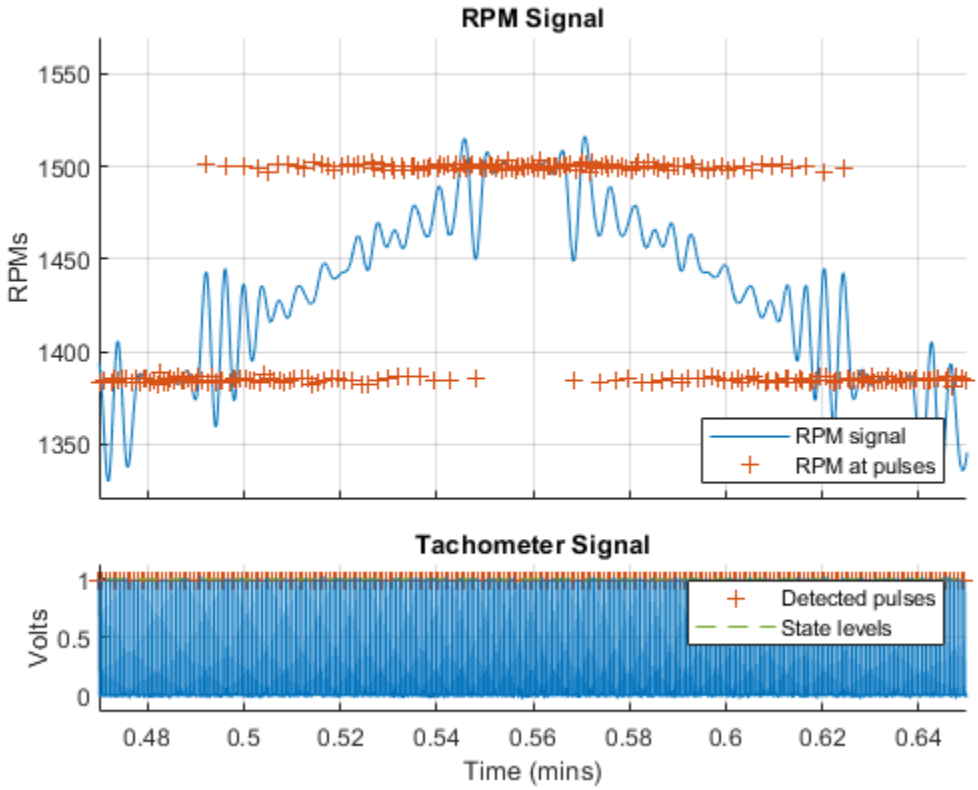
Compute and visualize the RPM signal using `tachorpm` with the default values.

```
tachorpm(Yn,fs)
```



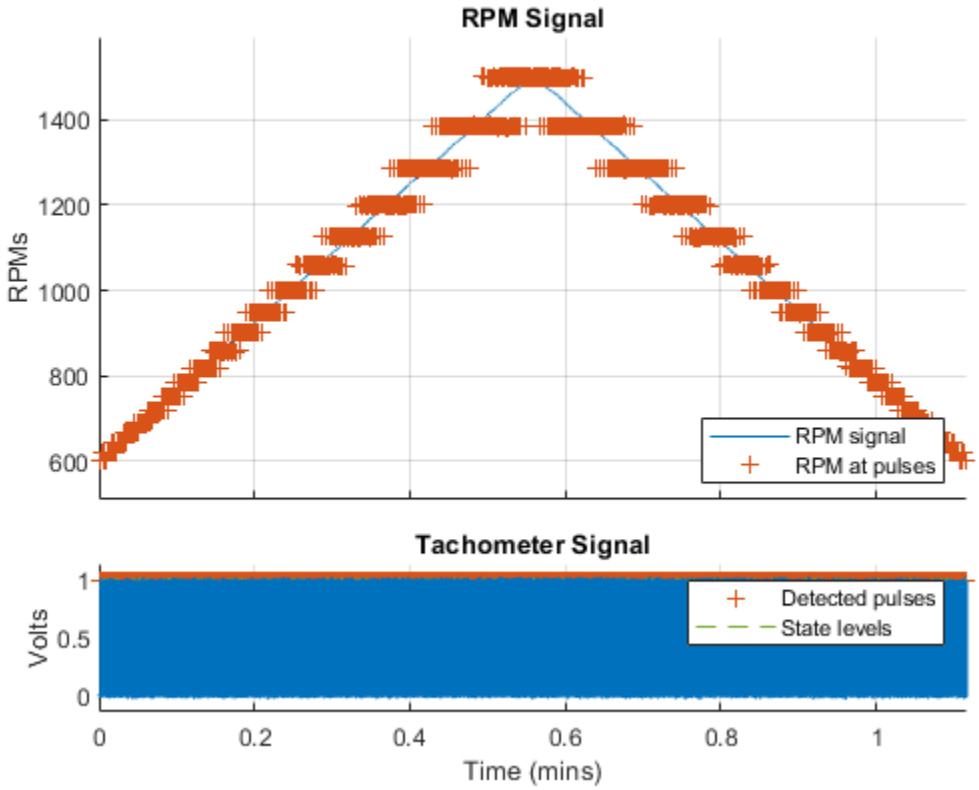
Increase the number of fit points to capture the RPM peak. Too many points result in overfitting. Verify this result by zooming in on the area around the peak.

```
tachorpm(Yn,fs,'FitPoints',600)

axis([0.47 0.65 1320 1570])
```



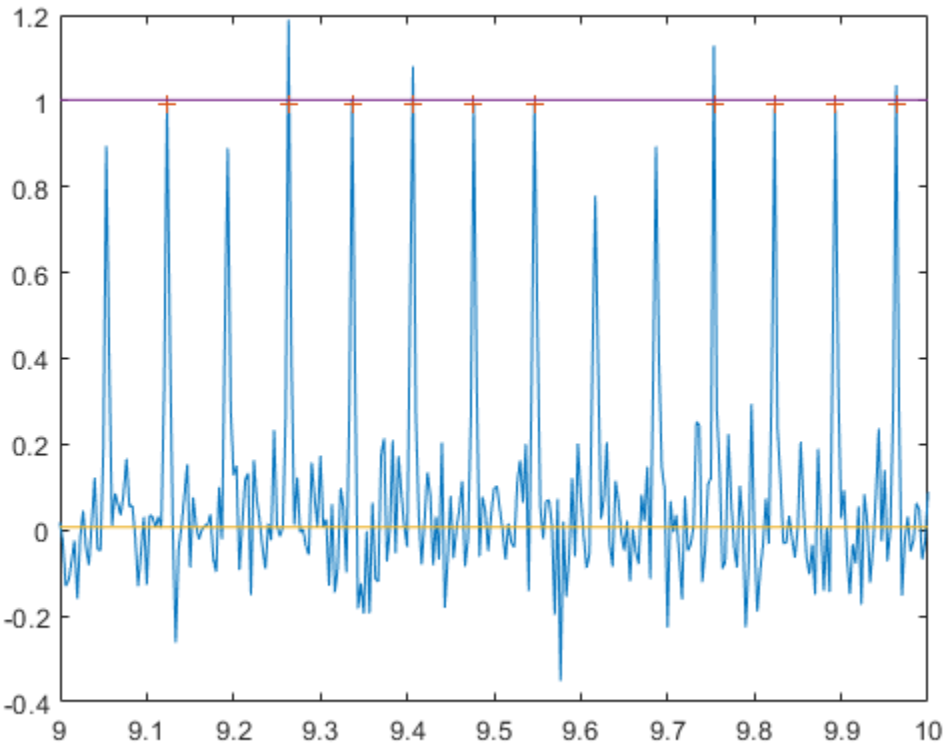
Choose a moderate number of points to obtain a better result.

```
tachorpm(Yn,fs,'FitPoints',100)
```



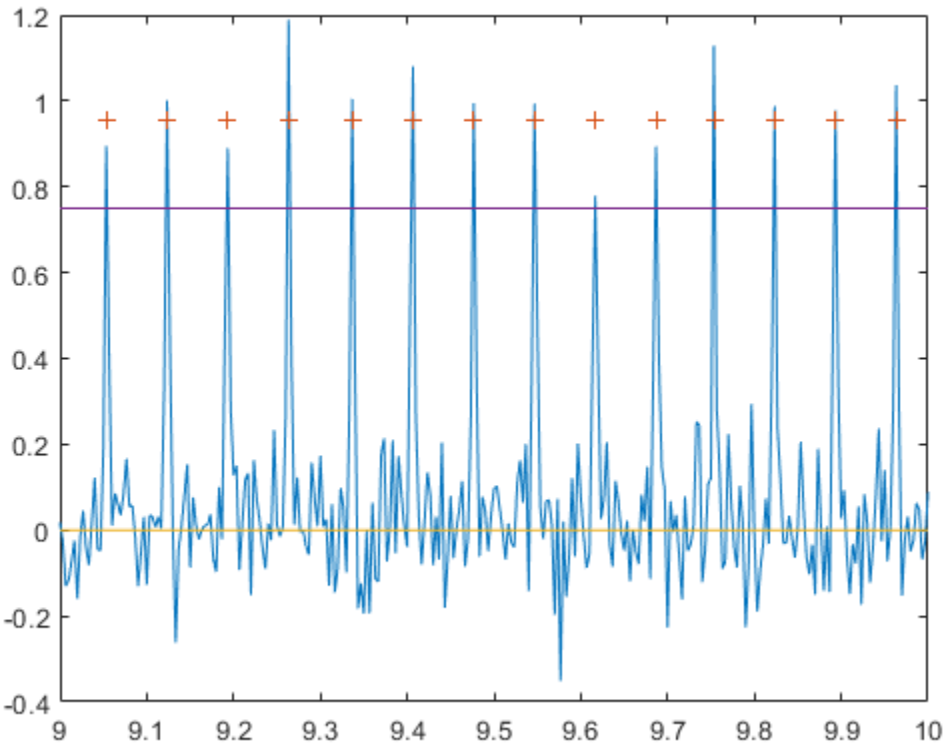
Add white Gaussian noise to the tachometer signal. The default pulse-finding mechanism misses pulses and returns a jagged signal profile. Verify this result by zooming in on a two-second time interval.

```
rng default
wgn = randn(size(Yn))/10;
Yn = Yn+wgn;

[rpm,t,tp] = tachorpm(Yn,fs,'FitPoints',100);

figure
plot(t,Yn,tp,mean(interp1(t,Yn,tp))*ones(size(tp)),'+')
hold on
sl = statelevels(Yn);
plot(t,sl(1)*ones(size(t)),t,sl(2)*ones(size(t)))
hold off
xlim([9 10])
```



Adjust the state levels to improve the pulse finding.

```
sl = [0 0.75];

[rpm,t,tp] = tachorpm(Yn,fs,'FitPoints',100,'StateLevels',sl);

plot(t,Yn,tp,mean(interp1(t,Yn,tp))*ones(size(tp)),'+')
hold on
plot(t,sl(1)*ones(size(t)),t,sl(2)*ones(size(t)))
hold off
xlim([9 10])
```



Input Arguments

x — Tachometer pulse signal

vector

Tachometer pulse signal, specified as a row or column vector.

Example: `double(chirp((-1.5:1/2e2:1.5),14,1.1,8,'quadratic')>0.98)` resembles a tachometer signal, sampled for three seconds at 200 Hz, and obtained during a quadratic run-up/coast-down test.

fs — Sample rate

positive scalar

Sample rate, specified as a positive scalar expressed in Hz.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'PulsesPerRev',3,'OutputFs',1e3` specifies that there are three tachometer pulses per revolution and that the returned RPM signal is to be sampled at 1 kHz.

`PulsesPerRev` — Number of tachometer pulses per revolution
`1` (default) | real scalar

Number of tachometer pulses per revolution, specified as the comma-separated pair consisting of `'PulsesPerRev'` and a real scalar.

`StateLevels` — State levels used to identify pulses
two-element real vector

State levels used to identify pulses, specified as the comma-separated pair consisting of `'StateLevels'` and a two-element real vector. The first element of the vector corresponds to the low-state level and the second element corresponds to the high-state level. Choose the state levels so that all pulse edges cross within 10% of both of them. If this option is not specified, then `tachorpm` computes the levels automatically using the histogram method, as in the `statelevels` function.

`OutputFs` — Output sample rate
`fs` (default) | real scalar

Output sample rate, specified as the comma-separated pair consisting of `'OutputFs'` and a real scalar.

`FitType` — Fitting method
`'smooth'` (default) | `'linear'`

Fitting method, specified as the comma-separated pair consisting of `'FitType'` and one of either `'smooth'` or `'linear'`.

- `'smooth'` — Fit a least-squares B-spline to the pulse RPM values.
- `'linear'` — Interpolate linearly between pulse RPM values.

`FitPoints` — B-spline breakpoints
`10` (default) | real scalar

B-spline breakpoints, specified as the comma-separated pair consisting of `'FitPoints'` and a real scalar. The number of breakpoints is a trade-off between curve smoothness and closeness to the underlying data. Choosing too many breakpoints can result in overfitting. This argument is ignored if `'FitType'` is set to `'linear'`.

Output Arguments

`rpm` — Rotational speeds
vector

Rotational speeds, returned as a vector expressed in revolutions per minute. `rpm` has the same length as `x`.

`t` — Time vector
vector of positive values

Time vector, returned as a vector of positive values expressed in seconds.

`tp` — Pulse locations
vector of positive values

Pulse locations, returned as a vector of positive values expressed in seconds.

Algorithms

The `tachorpm` function performs these steps:

1. Uses `statelevels` to determine the low and high states of the tachometer signal.
2. Uses `risetime` and `falltime` to find the times at which each pulse starts and ends. It then averages these readings to locate the time of each pulse.
3. Uses `diff` to determine the time intervals between pulse centers and computes the RPM values at the interval midpoints using RPM = 60 / Δt.
4. If `'FitType'` is specified as `'smooth'`, then the function performs least-squares fitting using splines. If `'FitType'` is specified as `'linear'`, then the function performs linear interpolation using `interp1`.

References

[1] Brandt, Anders. *Noise and Vibration Analysis: Signal Analysis and Experimental Procedures*. Chichester, UK: John Wiley & Sons, 2011.

[2] Vold, Håvard, and Jan Leuridan. “High Resolution Order Tracking at Extreme Slew Rates Using Kalman Tracking Filters.” *Shock and Vibration*. Vol. 2, 1995, pp. 507–515.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`orderspectrum` | `ordertrack` | `orderwaveform` | `rpmfreqmap` | `rpmordermap` | `rpmtrack` | `statelevels`

Introduced in R2016b

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 6th-order highpass FIR filter with a passband frequency of 75 kHz and a passband ripple of 0.2 dB. Specify a sample rate of 200 kHz. Compute the coefficients of the equivalent transfer function.

```
hpFilt = designfilt('highpassiir','FilterOrder',6, ...
         'PassbandFrequency',75e3,'PassbandRipple',0.2, ...
         'SampleRate',200e3);
[b,a] = tf(hpFilt)
```

```
b = 1×7

    0.0003   -0.0019    0.0048   -0.0064    0.0048   -0.0019    0.0003


a = 1×7

    1.0000    4.0580    7.5656    8.1243    5.2561    1.9348    0.3164
```

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 0.5π 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`

Introduced in R2014a

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` | `latcfilt` | `tf2sos` | `tf2ss` | `tf2zp` | `tf2zpk`

Introduced before R2006a

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

`[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 `b` and `a`.

`[sos,g] = tf2sos(b,a,order)` specifies the order of the rows in `sos`.

`[sos,g] = tf2sos(b,a,order,scale)` specifies the scaling of the gain and numerator coefficients of all second-order sections.

`sos = tf2sos( ___ )` embeds the overall system gain in the first section.

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]
```

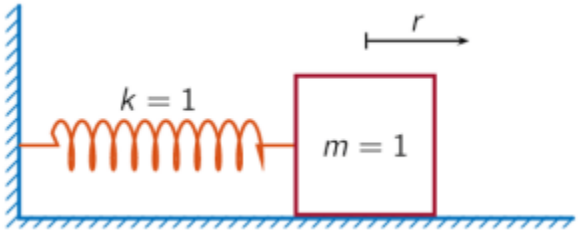
```
numers = 2×5

    0.0940    0.3759    0.5639    0.3759    0.0940
    0.0940    0.3759    0.5639    0.3759    0.0940
```

```
denoms = [conv(ss(1,4:6),ss(2,4:6));dn]
```

```
denoms = 2×5

    1.0000    0.0000    0.4860    0.0000    0.0177
    1.0000    0.0000    0.4860    0.0000    0.0177
```

Mass-Spring System

A one-dimensional discrete-time oscillating system consists of a unit mass, m, attached to a wall by a spring of unit elastic constant. A sensor samples the acceleration, a, of the mass at F_s = 5 Hz.



Generate 50 time samples. Define the sampling interval $\Delta t = 1/F_s$.

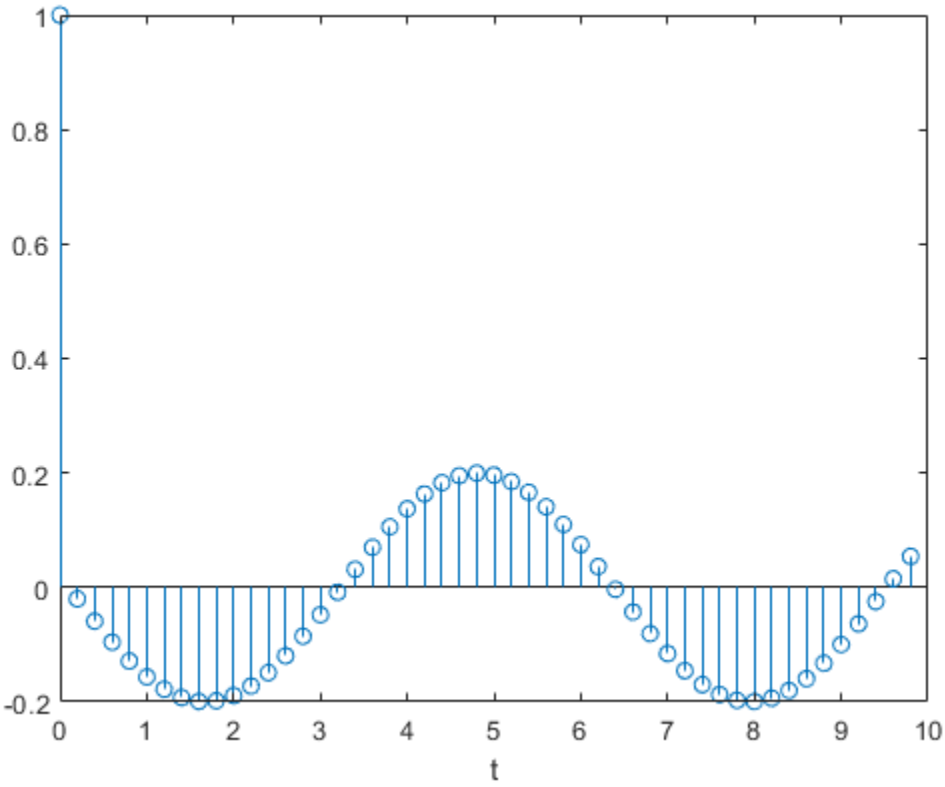
```
Fs = 5;
dt = 1/Fs;
N = 50;
t = dt*(0:N-1);
u = [1 zeros(1,N-1)];
```

The transfer function of the system has an analytic expression:

$$H(z) = \frac{1 - z^{-1}(1 + \cos\Delta t) + z^{-2}\cos\Delta t}{1 - 2z^{-1}\cos\Delta t + z^{-2}}.$$

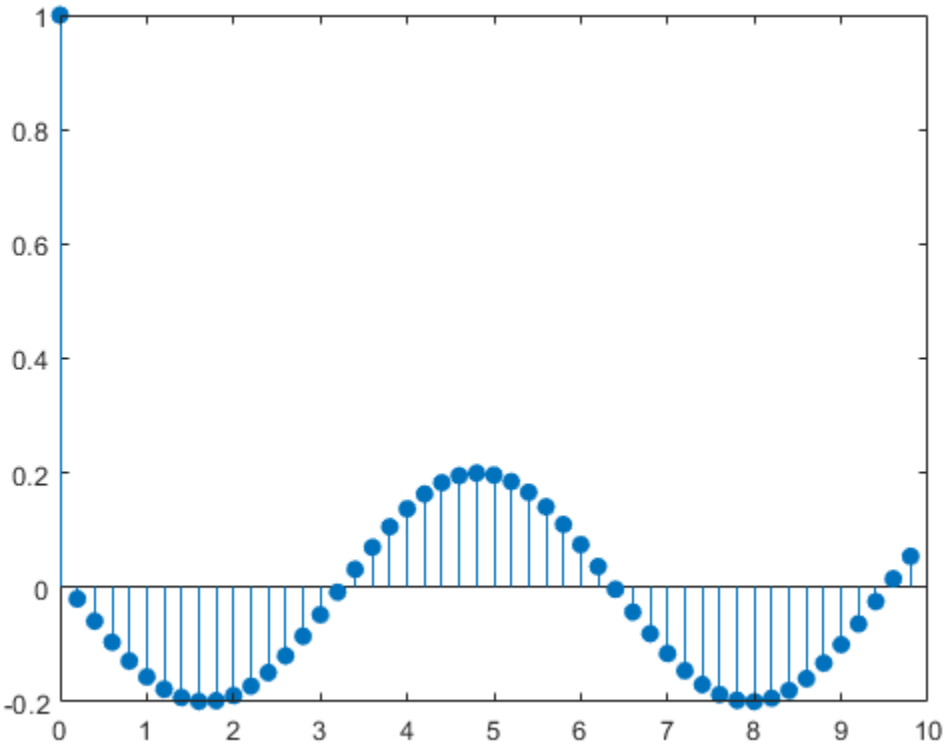
The system is excited with a unit impulse in the positive direction. Compute the time evolution of the system using the transfer function. Plot the response.

```
bf = [1 -(1+cos(dt)) cos(dt)];
af = [1 -2*cos(dt) 1];
yf = filter(bf,af,u);
stem(t,yf,'o')
xlabel('t')
```



Compute the time-dependent acceleration using the second-order sections representation of the transfer function to filter the input. Plot the result. The result is the same in both cases.

```
sos = tf2sos(bf,af);
yt = sosfilt(sos,u);
stem(t,yt,'filled')
```



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

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \cdots + b_{n+1} z^{-n}}{a_1 + a_2 z^{-1} + \cdots + a_{m+1} z^{-m}}.$$

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 0.5π rad/sample.

Data Types: `double`

order — Row order

`'up'` (default) | `'down'`

Row order, specified as one of the following:

- `'up'` — Order the sections so the first row of `sos` contains the poles farthest from the unit circle.
- `'down'` — Order the sections so the first row of `sos` contains the poles closest to the unit circle.

Data Types: `char`

`scale` — Scaling of gain and numerator coefficients

`'none'` (default) | `'inf'` | `'two'`

Scaling of gain and numerator coefficients, specified as one of the following:

- `'none'` — Apply no scaling.
- `'inf'` — Apply infinity-norm scaling.
- `'two'` — Apply 2-norm scaling.

Using infinity-norm scaling with `'up'`-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling with `'down'`-ordering minimizes the peak round-off noise.

Note Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

Data Types: `char`

Output Arguments

`sos` — Second-order section representation

matrix

Second-order section representation, returned as a matrix. `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 $H(z)$:

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

`g` — Overall system gain

real scalar

Overall system gain, returned as a real scalar.

If you call `tf2sos` with one output argument, the function embeds the overall system gain 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 can result in erratic scaling. To avoid embedding the gain, use `tf2sos` with two outputs.

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 `order` as `'down'`.
4. `tf2sos` scales the sections by the norm specified in `scale`. 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 ∞ or 2. See the references for details on the scaling. The algorithm follows this scaling in 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.

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

[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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Any character or string input must be a constant at compile time.

See Also

cplxpair | sos2tf | ss2sos | tf2ss | tf2zp | tf2zpk | zp2sos

Introduced before R2006a

tf2ss

Convert transfer function filter parameters to state-space form

Syntax

```
[A,B,C,D] = tf2ss(b,a)
```

Description

`[A,B,C,D] = tf2ss(b,a)` converts a continuous-time or discrete-time single-input transfer function into an equivalent state-space representation.

Examples

Convert Transfer Function to State-Space Form

Consider the system described by the transfer function

$$H(s) = \frac{\begin{bmatrix} 2s+3 \\ s^2+2s+1 \end{bmatrix}}{s^2+0.4s+1}.$$

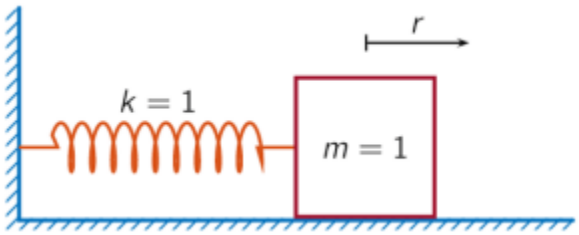
Convert it to state-space form using `tf2ss`.

```
b = [0 2 3; 1 2 1];
a = [1 0.4 1];
[A,B,C,D] = tf2ss(b,a)
```

```
A = 2×2

   -0.4000   -1.0000
    1.0000         0


B = 2×1

     1
     0


C = 2×2

    2.0000    3.0000
    1.6000         0


D = 2×1

     0
     1
```

Mass-Spring System

A one-dimensional discrete-time oscillating system consists of a unit mass, m, attached to a wall by a spring of unit elastic constant. A sensor samples the acceleration, a, of the mass at F_s = 5 Hz.



Generate 50 time samples. Define the sampling interval $\Delta t = 1/F_s$.

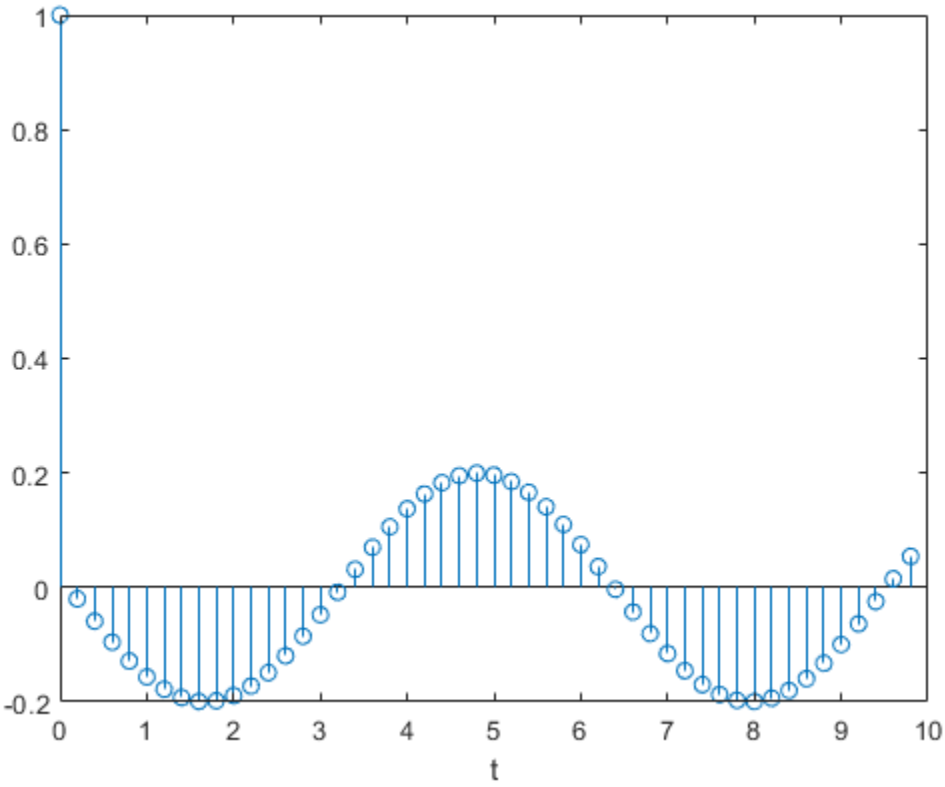
```
Fs = 5;
dt = 1/Fs;
N = 50;
t = dt*(0:N-1);
u = [1 zeros(1,N-1)];
```

The transfer function of the system has an analytic expression:

$$H(z) = \frac{1 - z^{-1}(1 + \cos\Delta t) + z^{-2}\cos\Delta t}{1 - 2z^{-1}\cos\Delta t + z^{-2}}.$$

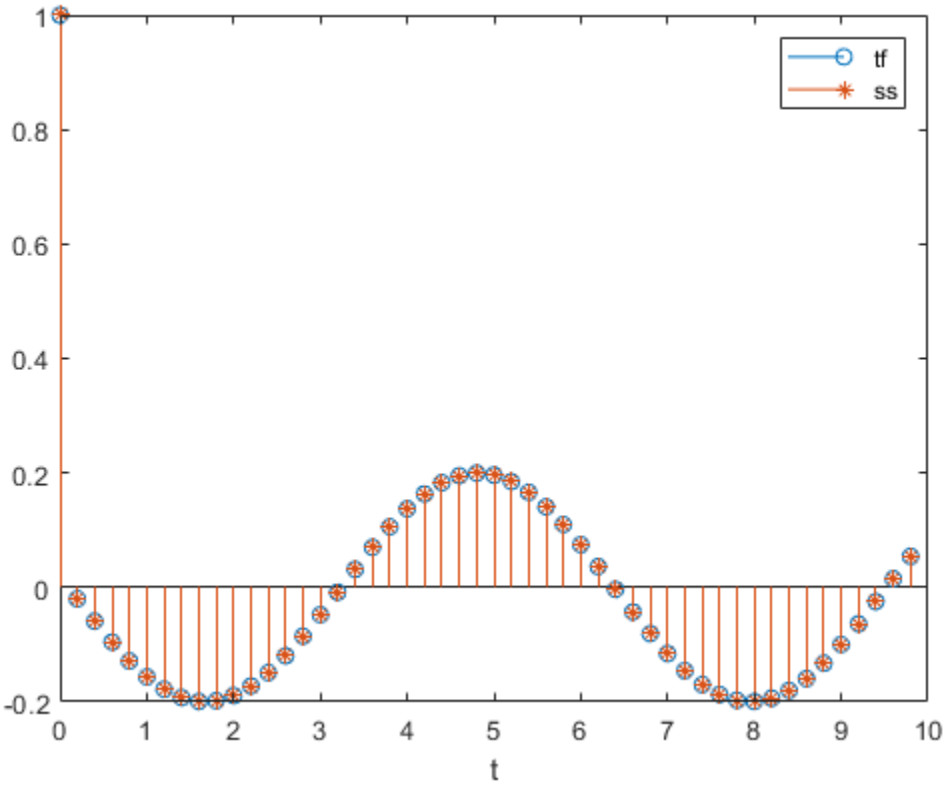
The system is excited with a unit impulse in the positive direction. Compute the time evolution of the system using the transfer function. Plot the response.

```
bf = [1 -(1+cos(dt)) cos(dt)];
af = [1 -2*cos(dt) 1];
yf = filter(bf,af,u);

stem(t,yf,'o')
xlabel('t')
```



Find the state-space representation of the system. Compute the time evolution starting from an all-zero initial state. Compare it to the transfer function prediction.

```
[A,B,C,D] = tf2ss(bf,af);

x = [0;0];
for k = 1:N
    y(k) = C*x + D*u(k);
    x = A*x + B*u(k);
end

hold on
stem(t,y,'*')
hold off
legend('tf','ss')
```



Input Arguments

b — Transfer function numerator coefficients

vector | matrix

Transfer function numerator coefficients, specified as a vector or matrix. If `b` is a matrix, then each row of `b` corresponds to an output of the system.

- For discrete-time systems, `b` contains the coefficients in descending powers of z.
- For continuous-time systems, `b` contains the coefficients in descending powers of s.

For discrete-time systems, `b` must have a number of columns equal to the length of `a`. If the numbers differ, make them equal by padding zeros. You can use the function `eqtflength` to accomplish this.

a — Transfer function denominator coefficients

vector

Transfer function denominator coefficients, specified as a vector.

- For discrete-time systems, `a` contains the coefficients in descending powers of z.
- For continuous-time systems, `a` contains the coefficients in descending powers of s.

Output Arguments

A — State matrix

matrix

State matrix, returned as a matrix. If the system is described by n state variables, then A is n-by-n.

Data Types: single | double

B — Input-to-state matrix

matrix

Input-to-state matrix, returned as a matrix. If the system is described by n state variables, then B is n-by-1.

Data Types: single | double

C — State-to-output matrix

matrix

State-to-output matrix, returned as a matrix. If the system has q outputs and is described by n state variables, then C is q-by-n.

Data Types: single | double

D — Feedthrough matrix

matrix

Feedthrough matrix, returned as a matrix. If the system has q outputs, then D is q-by-1.

Data Types: single | double

More About

Transfer Function

tf2ss converts the parameters of a transfer function representation of a given system to those of an equivalent state-space representation.

- For discrete-time systems, the state-space matrices relate the state vector x, the input u, and the output y:

  $$x(k+1) = Ax(k) + Bu(k),$$
  $$y(k) = Cx(k) + Du(k).$$

  The transfer function is the Z-transform of the system's impulse response. It can be expressed in terms of the state-space matrices as

  $$H(z) = C(zI - A)^{-1}B + D.$$

- For continuous-time systems, the state-space matrices relate the state vector x, the input u, and the output y:

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

The transfer function is the Laplace transform of the system's impulse response. It can be expressed in terms of the state-space matrices as

$$H(s) = \frac{B(s)}{A(s)} = \frac{b_1 s^{n-1} + \cdots + b_{n-1}s + b_n}{a_1 s^{m-1} + \cdots + a_{m-1}s + a_m} = C(sI - A)^{-1}B + D\,.$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`sos2ss` | `ss2tf` | `tf2sos` | `tf2zp` | `tf2zpk` | `zp2ss`

Introduced before R2006a

tf2zp

Convert transfer function filter parameters to zero-pole-gain form

Syntax

```
[z,p,k] = tf2zp(b,a)
```

Description

`[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 function converts a polynomial transfer-function representation

$$H(s) = \frac{B(s)}{A(s)} = \frac{b_1 s^{n-1} + \cdots + b_{n-1}s + b_n}{a_1 s^{m-1} + \cdots + a_{m-1}s + a_m}$$

of a single-input/multi-output (SIMO) continuous-time system to a factored transfer function form

$$H(s) = \frac{Z(s)}{P(s)} = k\frac{(s-z_1)(s-z_2)\cdots(s-z_m)}{(s-p_1)(s-p_2)\cdots(s-p_n)}.$$

Note 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 ($1 + z^{-1} + z^{-2}$).

Examples

Zeros, Poles, and Gain of 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.

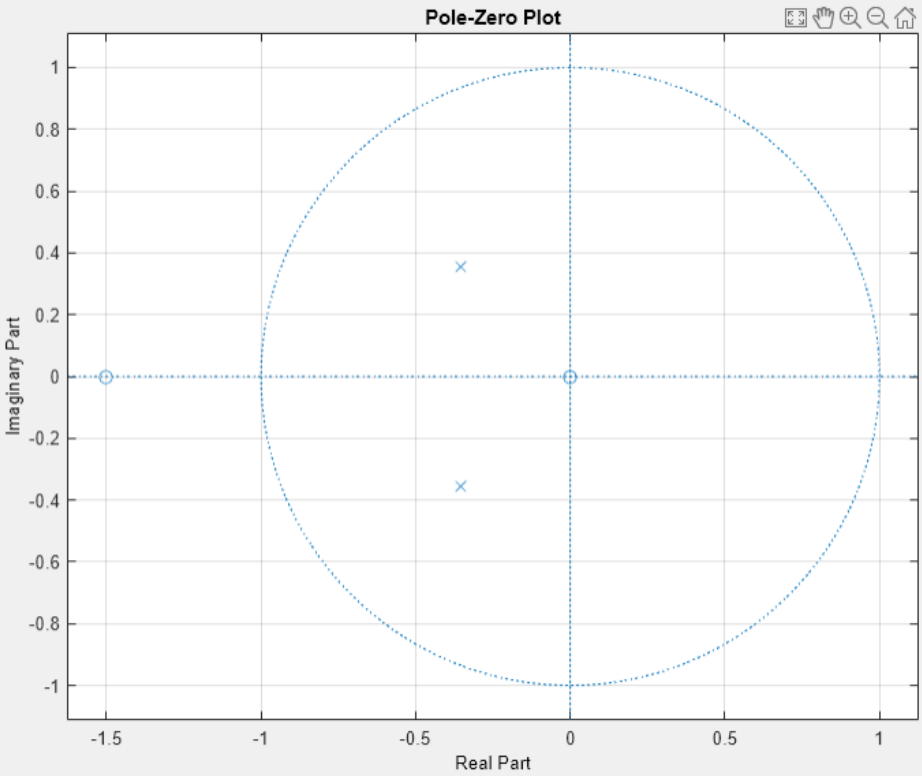
```
b = [2 3];
a = [1 1/sqrt(2) 1/4];

[b,a] = eqtflength(b,a);
[z,p,k] = tf2zp(b,a)
```

```
z = 2×1

          0
```

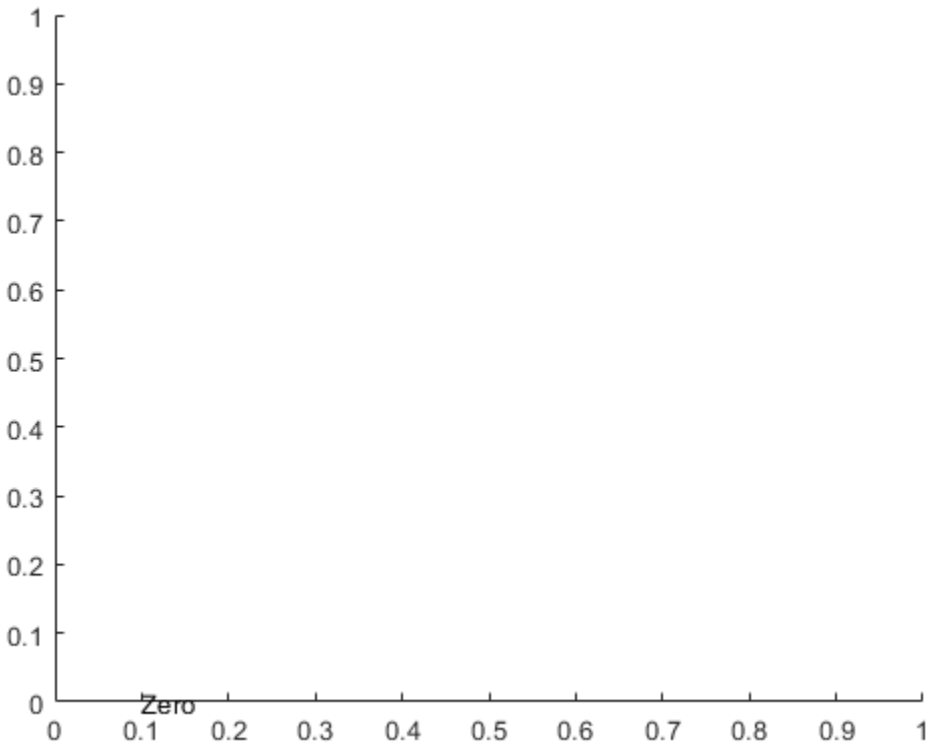
```
   -1.5000

p = 2×1 complex

  -0.3536 + 0.3536i
  -0.3536 - 0.3536i

k = 2
```

Plot the poles and zeros to verify that they are in the expected locations.

```
fvtool(b,a,'polezero')
```




```
text(real(z)+.1,imag(z),'Zero')
text(real(p)+.1,imag(p),'Pole')
```



Input Arguments

b — Transfer function numerator coefficients

vector | matrix

Transfer function numerator coefficients, specified as a vector or matrix. If `b` is a matrix, then each row of `b` corresponds to an output of the system. `b` contains the coefficients in descending powers of *s*. The number of columns of `b` must be less than or equal to the length of `a`.

Data Types: `single` | `double`

a — Transfer function denominator coefficients

vector

Transfer function denominator coefficients, specified as a vector. `a` contains the coefficients in descending powers of *s*.

Data Types: `single` | `double`

Output Arguments

z — Zeros

matrix

Zeros of the system, returned as a matrix. `z` contains the numerator zeros in its columns. `z` has as many columns as there are outputs.

p — Poles
column vector

Poles of the system, returned as a column vector. `p` contains the pole locations of the denominator coefficients of the transfer function.

k — Gains
column vector

Gains of the system, returned as a column vector. `k` contains the gains for each numerator transfer function.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The complexity of outputs, `z` and `k`, might be different in MATLAB and the generated code.
- The order of outputs, `z` and `p`, might be different in MATLAB and the generated code.

See Also
`sos2zp` | `ss2zp` | `tf2sos` | `tf2ss` | `tf2zpk` | `zp2tf`

Introduced before R2006a

tf2zpk

Convert transfer function filter parameters to zero-pole-gain form

Syntax

```
[z,p,k] = tf2zpk(b,a)
```

Description

`[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 function converts a polynomial transfer-function representation

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} \cdots + b_{n-1} z^{-n} + b_n z^{-n-1}}{a_1 + a_2 z^{-1} \cdots + a_{m-1} z^{-m} + a_m z^{-m-1}}$$

of a single-input/multi-output (SIMO) discrete-time system to a factored transfer function form

$$H(z) = \frac{Z(z)}{P(z)} = k \frac{(z - z_1)(z - z_2) \cdots (z - z_m)}{(z - p_1)(z - p_2) \cdots (z - p_n)}.$$

Note Use `tf2zpk` when working with transfer functions expressed in inverse powers ($1 + z^{-1} + z^{-2}$). A similar function, `tf2zp`, is more useful for working with positive powers ($s^2 + s + 1$), such as in continuous-time transfer functions.

Examples

Poles, Zeros, and Gain of IIR Filter

Design a 3rd-order Butterworth filter with normalized cutoff frequency 0.4π rad/sample. Find the poles, zeros, and gain of the filter.

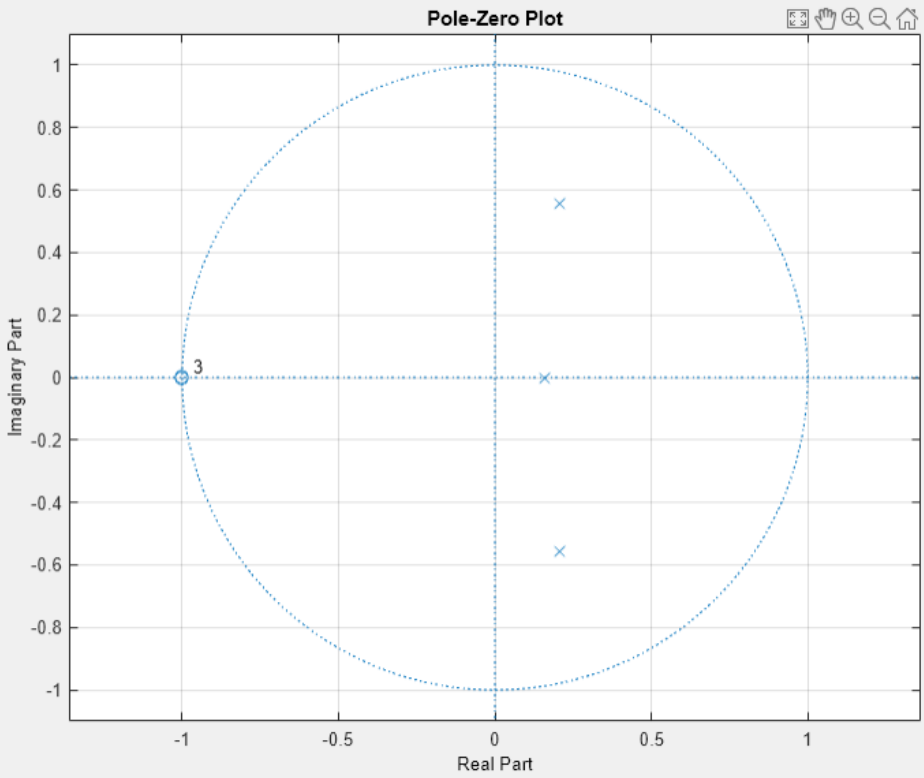
```
[b,a] = butter(3,.4);
[z,p,k] = tf2zpk(b,a)
```

```
z = 3×1 complex

  -1.0000 + 0.0000i
  -1.0000 - 0.0000i
  -1.0000 + 0.0000i


p = 3×1 complex

   0.2094 + 0.5582i
   0.2094 - 0.5582i
   0.1584 + 0.0000i
```

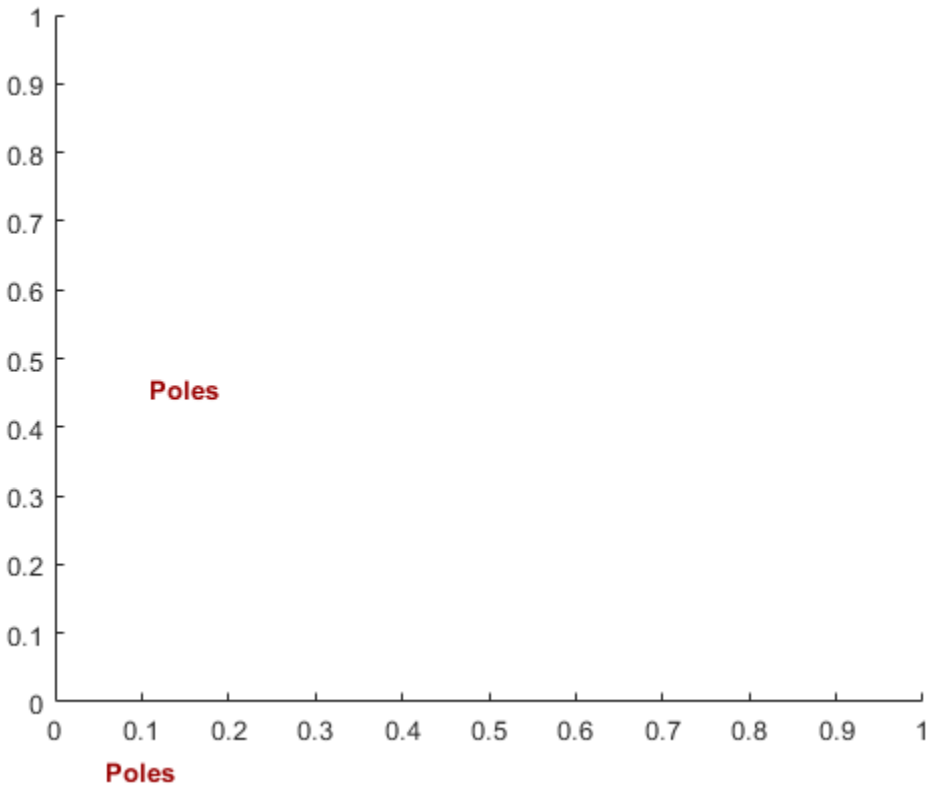
```
k = 0.0985
```

Plot the poles and zeros to verify that they are where expected.

```
fvtool(b,a,'polezero')
```




```
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])
```



Input Arguments

b — Transfer function numerator coefficients

vector | matrix

Transfer function numerator coefficients, specified as a vector or matrix. If `b` is a matrix, then each row of `b` corresponds to an output of the system. `b` contains the coefficients in descending powers of z. The number of columns of `b` must be equal to the length of `a`. If the numbers differ, make them equal by padding zeros. You can use the function `eqtflength` to accomplish this.

Data Types: `single` | `double`

a — Transfer function denominator coefficients

vector

Transfer function denominator coefficients, specified as a vector. `a` contains the coefficients in descending powers of z.

Data Types: `single` | `double`

Output Arguments

z — Zeros

matrix

Zeros of the system, returned as a matrix. `z` contains the numerator zeros in its columns. `z` has as many columns as there are outputs.

p — Poles
column vector

Poles of the system, returned as a column vector. `p` contains the pole locations of the denominator coefficients of the transfer function

k — Gains
column vector

Gains of the system, returned as a column vector. `k` contains the gains for each numerator transfer function.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The complexity of outputs, `z` and `k`, might be different in MATLAB and the generated code.
- The order of outputs, `z` and `p`, might be different in MATLAB and the generated code.

See Also
`sos2zp` | `ss2zp` | `tf2sos` | `tf2ss` | `tf2zp` | `zp2tf`

Introduced before R2006a

tfestimate

Transfer function estimate

Syntax

```
txy = tfestimate(x,y)
txy = tfestimate(x,y,window)
txy = tfestimate(x,y,window,noverlap)
txy = tfestimate(x,y,window,noverlap,nfft)

txy = tfestimate(___,'mimo')

[txy,w] = tfestimate(___)
[txy,f] = tfestimate(___,fs)

[txy,w] = tfestimate(x,y,window,noverlap,w)
[txy,f] = tfestimate(x,y,window,noverlap,f,fs)

[___] = tfestimate(x,y,___,freqrange)

[___] = tfestimate(___,'Estimator',est)

tfestimate(___)
```

Description

`txy = tfestimate(x,y)` finds a transfer function estimate, `txy`, given an input signal, `x`, and an output signal, `y`.

- If `x` and `y` are both vectors, they must have the same length.
- If one of the signals is a matrix and the other is a vector, then the length of the vector must equal the number of rows in the matrix. The function expands the vector and returns a matrix of column-by-column transfer function estimates.
- If `x` and `y` are matrices with the same number of rows but different numbers of columns, then `txy` is a multi-input/multi-output (MIMO) transfer function that combines all input and output signals. `txy` is a three-dimensional array. If `x` has *m* columns and `y` has *n* columns, then `txy` has *n* columns and *m* pages. See “Transfer Function” on page 1-2648 for more information.
- If `x` and `y` are matrices of equal size, then `tfestimate` operates column-wise: `txy(:,n) = tfestimate(x(:,n),y(:,n))`. To obtain a MIMO estimate, append `'mimo'` to the argument list.

`txy = tfestimate(x,y,window)` uses `window` to divide `x` and `y` into segments and perform windowing.

`txy = tfestimate(x,y,window,noverlap)` uses `noverlap` samples of overlap between adjoining segments.

`txy = tfestimate(x,y,window,noverlap,nfft)` uses `nfft` sampling points to calculate the discrete Fourier transform.

`txy = tfestimate( ___ ,'mimo')` computes a MIMO transfer function for matrix inputs. This syntax can include any combination of input arguments from previous syntaxes.

`[txy,w] = tfestimate( ___ )` returns a vector of normalized frequencies, `w`, at which the transfer function is estimated.

`[txy,f] = tfestimate( ___ ,fs)` returns a vector of frequencies, `f`, expressed in terms of the sample rate, `fs`, at which the transfer function is estimated. `fs` must be the sixth numeric input to `tfestimate`. To input a sample rate and still use the default values of the preceding optional arguments, specify these arguments as empty `[]`.

`[txy,w] = tfestimate(x,y,window,noverlap,w)` returns the transfer function estimate at the normalized frequencies specified in `w`.

`[txy,f] = tfestimate(x,y,window,noverlap,f,fs)` returns the transfer function estimate at the frequencies specified in `f`.

`[ ___ ] = tfestimate(x,y, ___ ,freqrange)` returns the transfer function estimate over the frequency range specified by `freqrange`. Valid options for `freqrange` are `'onesided'`, `'twosided'`, and `'centered'`.

`[ ___ ] = tfestimate( ___ ,'Estimator',est)` estimates transfer functions using the estimator `est`. Valid options for `est` are `'H1'` and `'H2'`.

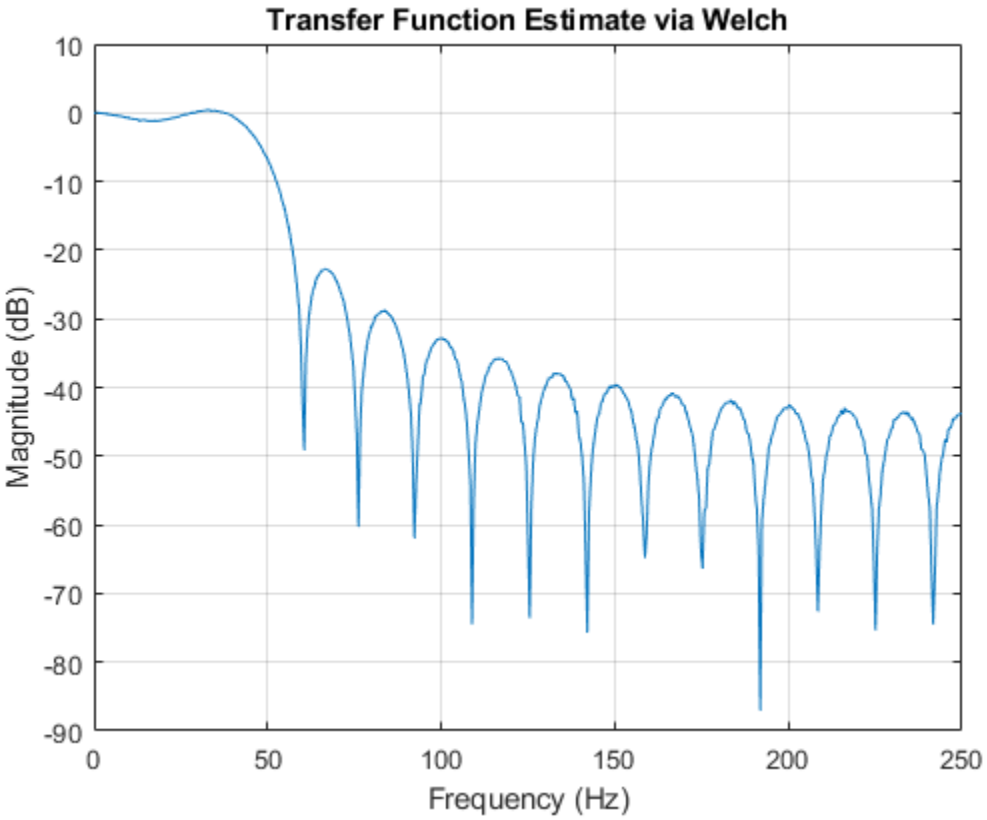
`tfestimate( ___ )` with no output arguments plots the transfer function estimate in the current figure window.

Examples

Transfer Function Between Two Sequences

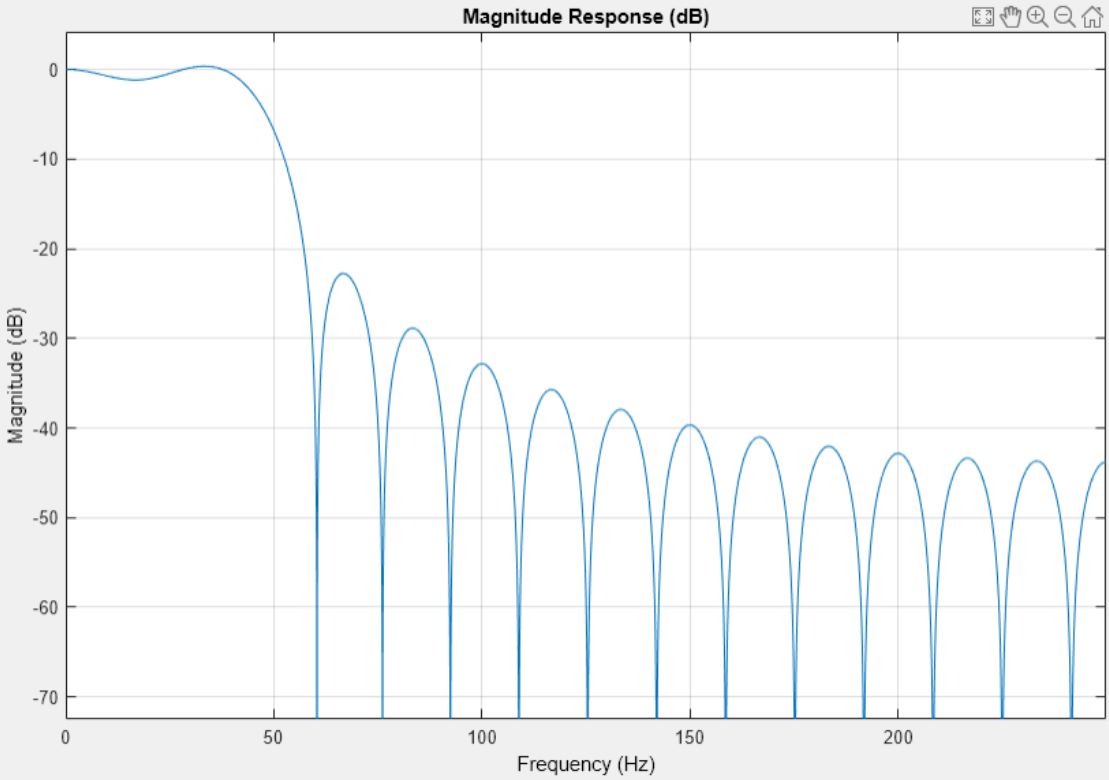
Compute and plot the transfer function estimate between two sequences, `x` and `y`. The sequence `x` consists of white Gaussian noise. `y` results from filtering `x` with a 30th-order lowpass filter with normalized cutoff frequency 0.2π rad/sample. Use a rectangular window to design the filter. Specify a sample rate of 500 Hz and a Hamming window of length 1024 for the transfer function estimate.

```
h = fir1(30,0.2,rectwin(31));
x = randn(16384,1);
y = filter(h,1,x);

fs = 500;
tfestimate(x,y,1024,[],[],fs)
```



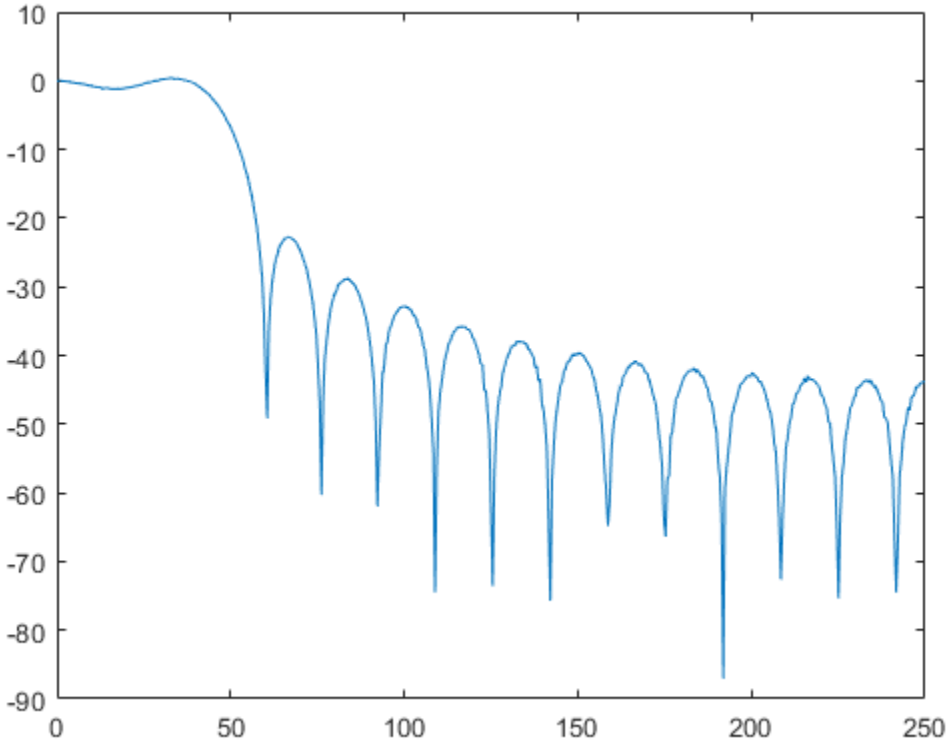
Use `fvtool` to verify that the transfer function approximates the frequency response of the filter.

```
fvtool(h,1,'Fs',fs)
```



Obtain the same result by returning the transfer function estimate in a variable and plotting its absolute value in decibels.

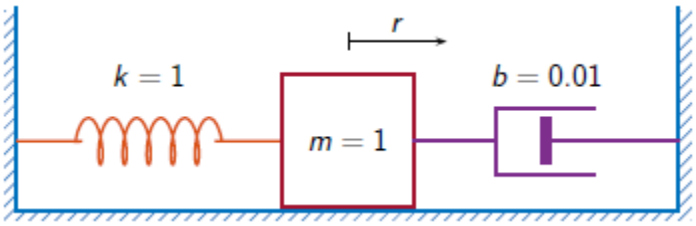
```
[Txy,f] = tfestimate(x,y,1024,[],[],fs);

plot(f,mag2db(abs(Txy)))
```



SISO Transfer Function

Estimate the transfer function for a simple single-input/single-output system and compare it to the definition.

A one-dimensional discrete-time oscillating system consists of a unit mass, m, attached to a wall by a spring of unit elastic constant. A sensor samples the acceleration, a, of the mass at $F_s = 1$ Hz. A damper impedes the motion of the mass by exerting on it a force proportional to speed, with damping constant $b = 0.01$.



Generate 2000 time samples. Define the sampling interval $\Delta t = 1/F_s$.

```
Fs = 1;
dt = 1/Fs;
N = 2000;
```

```
t = dt*(0:N-1);
b = 0.01;
```

The system can be described by the state-space model

$$x(k+1) = Ax(k) + Bu(k),$$
$$y(k) = Cx(k) + Du(k),$$

where $x = [r\ v]^T$ is the state vector, r and v are respectively the position and velocity of the mass, u is the driving force, and $y = a$ is the measured output. The state-space matrices are

$$A = \exp(A_c \Delta t), \quad B = A_c^{-1}(A - I)B_c, \quad C = [-1\ -b], \quad D = 1,$$

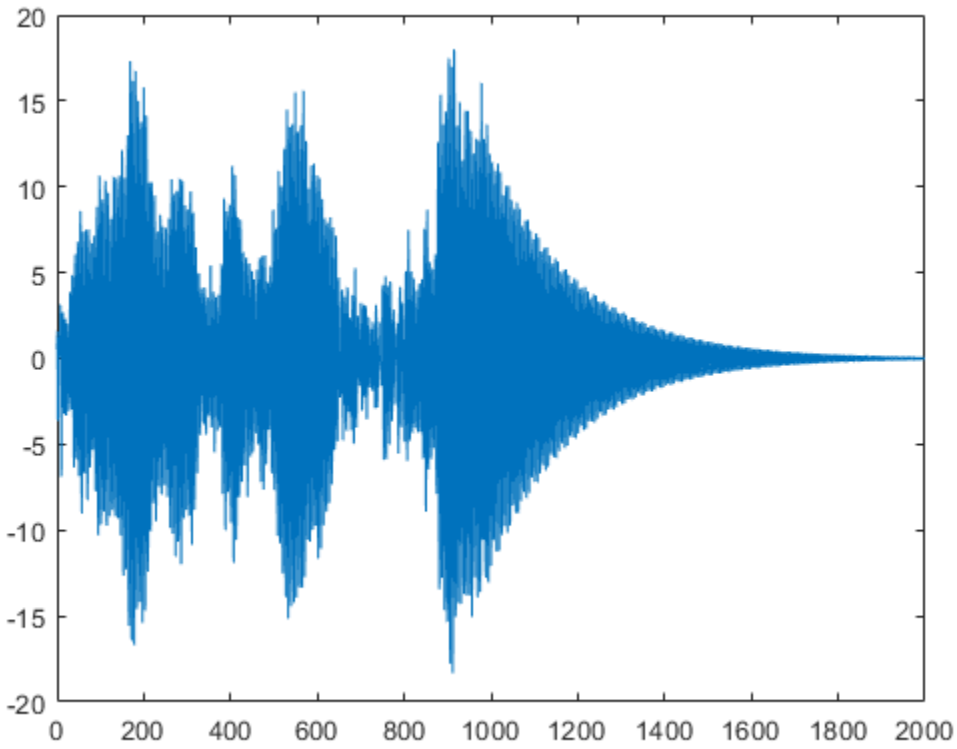
I is the 2×2 identity, and the continuous-time state-space matrices are

$$A_c = \begin{bmatrix} 0 & 1 \\ -1 & -b \end{bmatrix}, \quad B_c = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

```
Ac = [0 1;-1 -b];
A = expm(Ac*dt);

Bc = [0;1];
B = Ac\(A-eye(size(A)))*Bc;

C = [-1 -b];
D = 1;
```

The mass is driven by random input for half of the measurement interval. Use the state-space model to compute the time evolution of the system starting from an all-zero initial state. Plot the acceleration of the mass as a function of time.

```
rng default

u = zeros(1,N);
u(1:N/2) = randn(1,N/2);

y = 0;
x = [0;0];
for k = 1:N
    y(k) = C*x + D*u(k);
    x = A*x + B*u(k);
end

plot(t,y)
```

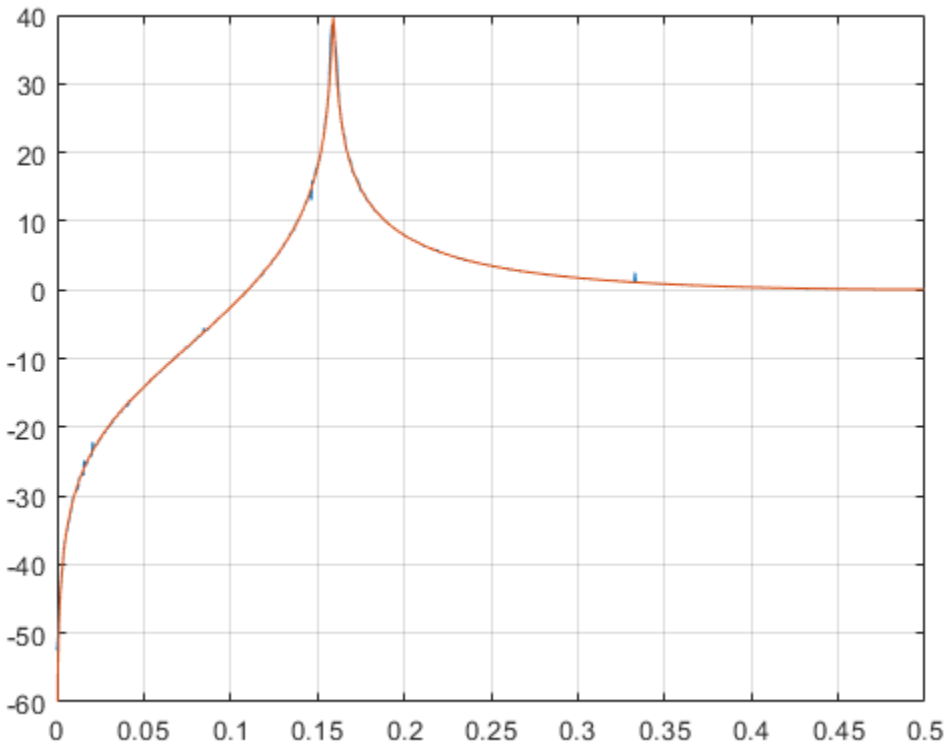


Estimate the transfer function of the system as a function of frequency. Use 2048 DFT points and specify a Kaiser window with a shape factor of 15. Use the default value of overlap between adjoining segments.

```
nfs = 2048;
wind = kaiser(N,15);

[txy,ft] = tfestimate(u,y,wind,[],nfs,Fs);
```

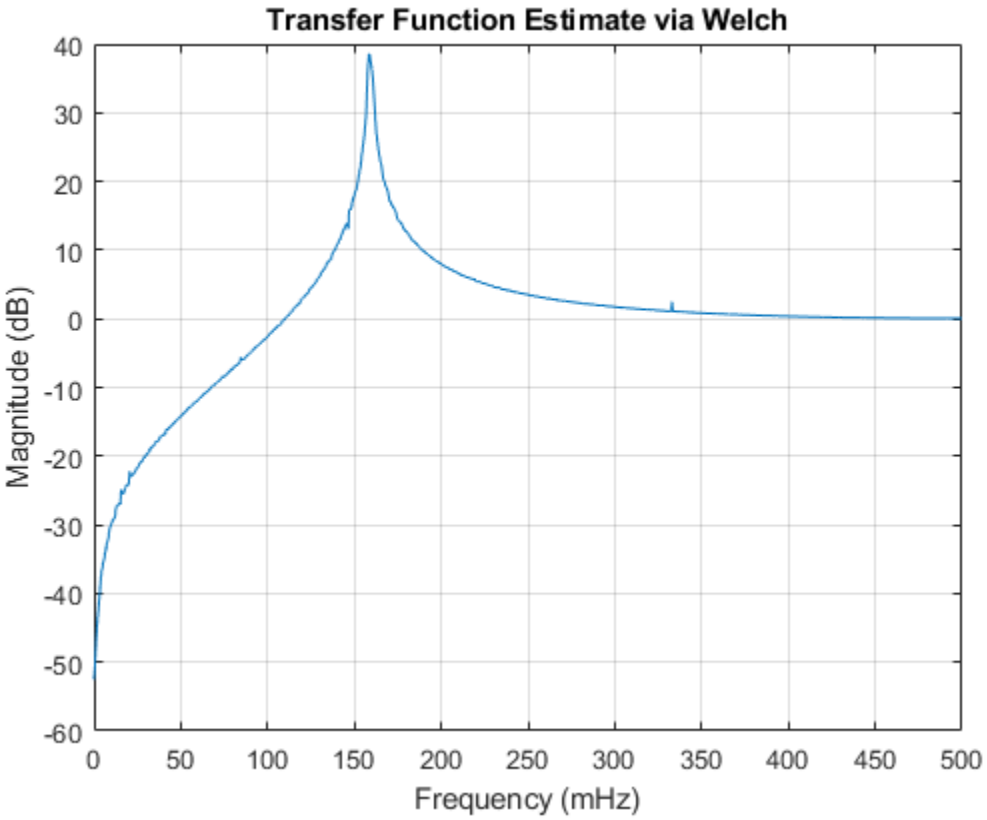
The frequency-response function of a discrete-time system can be expressed as the Z-transform of the time-domain transfer function of the system, evaluated at the unit circle. Verify that the estimate computed by `tfestimate` coincides with this definition.

```
[b,a] = ss2tf(A,B,C,D);

fz = 0:1/nfs:1/2-1/nfs;
z = exp(2j*pi*fz);
frf = polyval(b,z)./polyval(a,z);

plot(ft,20*log10(abs(txy)))
hold on
plot(fz,20*log10(abs(frf)))
hold off
grid
ylim([-60 40])
```



Plot the estimate using the built-in functionality of `tfestimate`.

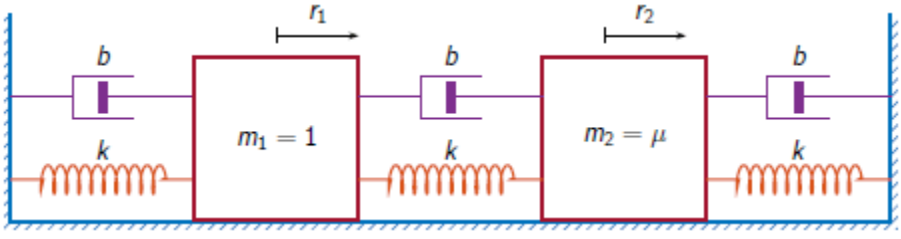
```
tfestimate(u,y,wind,[],nfs,Fs)
```



Transfer Function of MIMO System

Estimate the transfer function for a simple multi-input/multi-output system.

An ideal one-dimensional oscillating system consists of two masses, m_1 and m_2, confined between two walls. The units are such that $m_1 = 1$ and $m_2 = \mu$. Each mass is attached to the nearest wall by a spring with an elastic constant k. An identical spring connects the two masses. Three dampers impede the motion of the masses by exerting on them forces proportional to speed, with damping constant b. Sensors sample a_1 and a_2, the accelerations of the masses, at $F_s = 50$ Hz.



Generate 30000 time samples, equivalent to 600 seconds. Define the sampling interval $\Delta t = 1/F_s$.

```
Fs = 50;
dt = 1/Fs;
```

```
N = 30000;
t = dt*(0:N-1);
```

The system can be described by the state-space model

$$x(k+1) = Ax(k) + Bu(k),$$
$$y(k) = Cx(k) + Du(k),$$

where $x = \begin{bmatrix} r_1 & v_1 & r_2 & v_2 \end{bmatrix}^T$ is the state vector, r_i and v_i are respectively the location and the velocity of the ith mass, $u = \begin{bmatrix} u_1 & u_2 \end{bmatrix}^T$ is the vector of input driving forces, and $y = \begin{bmatrix} a_1 & a_2 \end{bmatrix}^T$ is the output vector. The state-space matrices are

$$A = \exp(A_c \Delta t), \quad B = A_c^{-1}(A - I)B_c, \quad C = \begin{bmatrix} -2k & -2b & k & b \\ k/\mu & b/\mu & -2k/\mu & -2b/\mu \end{bmatrix}, \quad D = \begin{bmatrix} 1 & 0 \\ 0 & 1/\mu \end{bmatrix},$$

I is the 4×4 identity, and the continuous-time state-space matrices are

$$A_c = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -2k & -2b & k & b \\ 0 & 0 & 0 & 1 \\ k/\mu & b/\mu & -2k/\mu & -2b/\mu \end{bmatrix}, \quad B_c = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1/\mu \end{bmatrix}.$$

Set $k = 400$, $b = 0$, and $\mu = 1/10$.

```
k = 400;
b = 0;
m = 1/10;

Ac = [0 1 0 0;-2*k -2*b k b;0 0 0 1;k/m b/m -2*k/m -2*b/m];
A = expm(Ac*dt);
Bc = [0 0;1 0;0 0;0 1/m];
B = Ac\(A-eye(4))*Bc;
C = [-2*k -2*b k b;k/m b/m -2*k/m -2*b/m];
D = [1 0;0 1/m];
```

The masses are driven by random input throughout the measurement. Use the state-space model to compute the time evolution of the system starting from an all-zero initial state.

```
rng default
u = randn(2,N);

x = [0;0;0;0];
for kk = 1:N
    y(:,kk) = C*x + D*u(:,kk);
    x = A*x + B*u(:,kk);
end
```

Use the input and output data to estimate the transfer function of the system as a function of frequency. Specify the `'mimo'` option to produce all four transfer functions. Use a 5000-sample Hann window to divide the signals into segments. Specify 2500 samples of overlap between adjoining segments and 2^{14} DFT points. Plot the estimates.

```
wind = hann(5000);
nov = 2500;
```

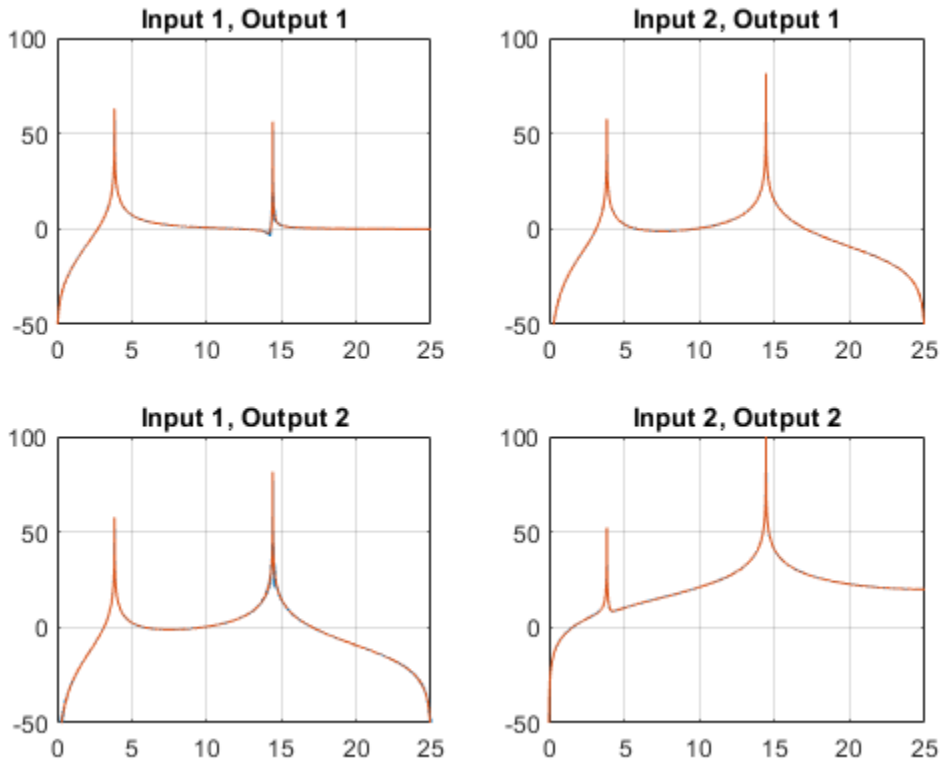
```
[q,fq] = tfestimate(u',y',wind,nov,2^14,Fs,'mimo');
```

Compute the theoretical transfer function as the Z-transform of the time-domain transfer function, evaluated at the unit circle.

```
nfs = 2^14;

fz = 0:1/nfs:1/2-1/nfs;
z = exp(2j*pi*fz);

[b1,a1] = ss2tf(A,B,C,D,1);
[b2,a2] = ss2tf(A,B,C,D,2);

frf(1,:,1) = polyval(b1(1,:),z)./polyval(a1,z);
frf(1,:,2) = polyval(b1(2,:),z)./polyval(a1,z);

frf(2,:,1) = polyval(b2(1,:),z)./polyval(a2,z);
frf(2,:,2) = polyval(b2(2,:),z)./polyval(a2,z);
```

Plot the theoretical transfer functions and their corresponding estimates.

```
for jk = 1:2
    for kj = 1:2
        subplot(2,2,2*(jk-1)+kj)
        plot(fq,20*log10(abs(q(:,jk,kj))))
        hold on
        plot(fz*Fs,20*log10(abs(frf(jk,:,kj))))
        hold off
        grid
        title(['Input ' int2str(kj) ', Output ' int2str(jk)])
        axis([0 Fs/2 -50 100])
    end
end
```



The transfer functions have maxima at the expected values, $\omega_{1,2}/2\pi$, where the ω are the eigenvalues of the modal matrix.

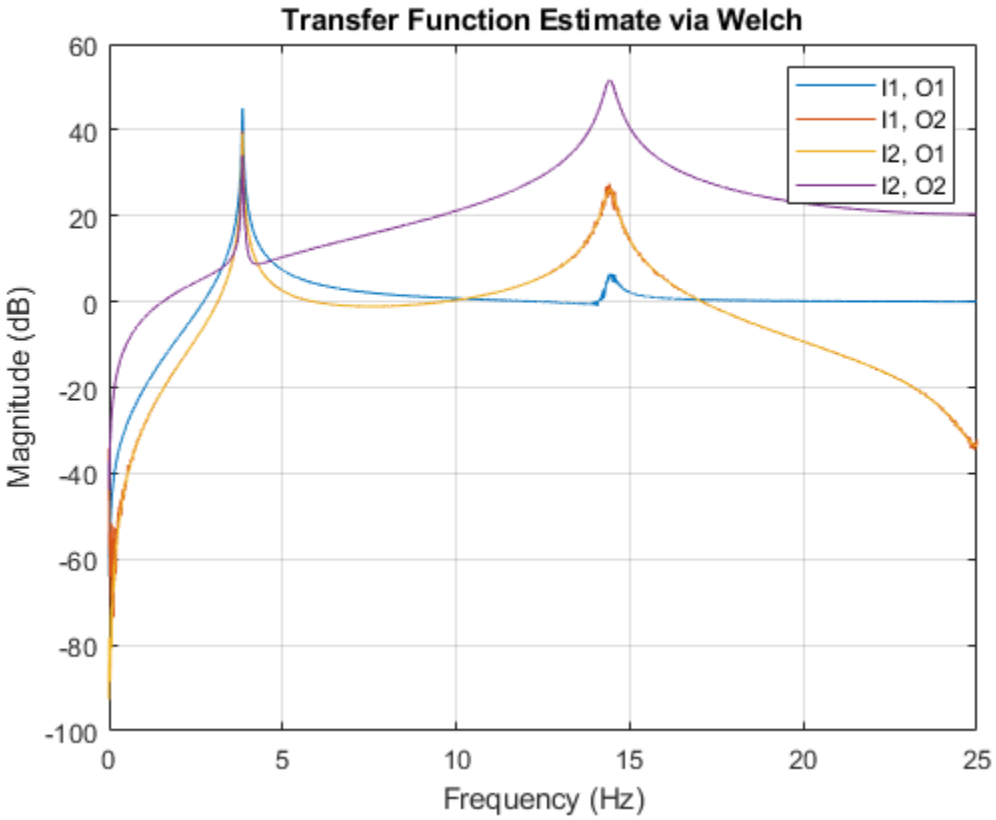
```
sqrt(eig(k*[2 -1;-1/m 2/m]))/(2*pi)
```

```
ans = 2×1

    3.8470
   14.4259
```

Add damping to the system by setting $b = 0.1$. Compute the time evolution of the damped system with the same driving forces. Compute the H_2 estimate of the MIMO transfer function using the same window and overlap. Plot the estimates using the `tfestimate` functionality.

```
b = 0.1;

Ac = [0 1 0 0;-2*k -2*b k b;0 0 0 1;k/m b/m -2*k/m -2*b/m];
A = expm(Ac*dt);
B = Ac\(A-eye(4))*Bc;
C = [-2*k -2*b k b;k/m b/m -2*k/m -2*b/m];

x = [0;0;0;0];
for kk = 1:N
    y(:,kk) = C*x + D*u(:,kk);
    x = A*x + B*u(:,kk);
end
```

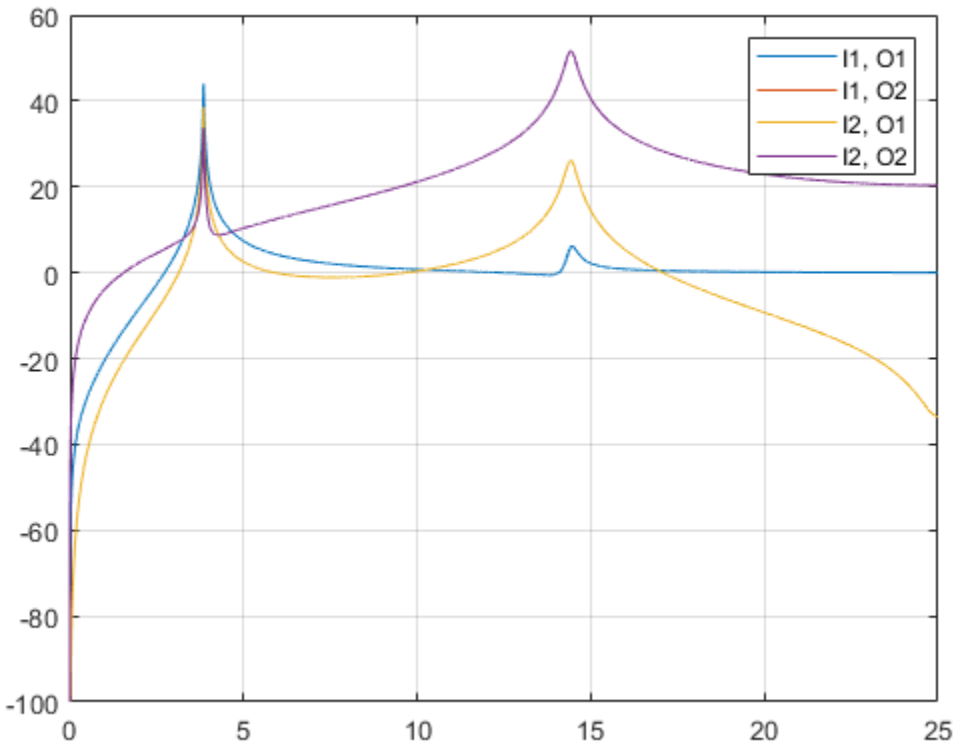
```
clf
tfestimate(u',y',wind,nov,[],Fs,'mimo','Estimator','H2')
legend('I1, O1','I1, O2','I2, O1','I2, O2')
```




```
yl = ylim;
```

Compare the estimates to the theoretical predictions.

```
[b1,a1] = ss2tf(A,B,C,D,1);
[b2,a2] = ss2tf(A,B,C,D,2);

frf(1,:,1) = polyval(b1(1,:),z)./polyval(a1,z);
frf(1,:,2) = polyval(b1(2,:),z)./polyval(a1,z);

frf(2,:,1) = polyval(b2(1,:),z)./polyval(a2,z);
frf(2,:,2) = polyval(b2(2,:),z)./polyval(a2,z);

plot(fz*Fs,20*log10(abs(reshape(permute(frf,[2 1 3]),[nfs/2 4]))))
legend('I1, O1','I1, O2','I2, O1','I2, O2')
ylim(yl)
grid
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix.

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

y — Output signal

vector | matrix

Output signal, specified as a vector or matrix.

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 segments.

- If `window` is an integer, then `tfestimate` divides `x` and `y` into segments of length `window` and windows each segment with a Hamming window of that length.
- If `window` is a vector, then `tfestimate` divides `x` and `y` into segments of the same length as the vector and windows each segment using `window`.

If the length of `x` and `y` cannot be divided exactly into an integer number of segments with `noverlap` overlapping samples, then the signals are truncated accordingly.

If you specify `window` as empty, then `tfestimate` uses a Hamming window such that `x` and `y` are divided into eight segments 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 `tfestimate` uses a number that produces 50% overlap between segments.

Data Types: `double` | `single`

`nfft` — Number of DFT points
positive integer | `[]`

Number of DFT points, specified as a positive integer. If you specify `nfft` as empty, then `tfestimate` sets this argument to $\max(256,2^p)$, where $p = \lceil \log_2 N \rceil$ for input signals of length N and the $\lceil\ \rceil$ symbols denote the ceiling function.

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 has units of Hz.

`w` — Normalized frequencies
vector

Normalized frequencies, specified as a row or column vector with at least two elements. Normalized frequencies are in rad/sample.

Example: `w = [pi/4 pi/2]`

Data Types: `double`

`f` — Frequencies
vector

Frequencies, specified as a row or column vector with at least two elements. The frequencies are in cycles per unit time. The unit time is specified by the sample rate, `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 transfer function estimate
`'onesided'` | `'twosided'` | `'centered'`

Frequency range for the transfer function estimate, specified as a one of `'onesided'`, `'twosided'`, or `'centered'`. The default is `'onesided'` for real-valued signals and `'twosided'` for complex-valued signals.

- `'onesided'` — Returns the one-sided estimate of the transfer function between two real-valued input signals, `x` and `y`. If `nfft` is even, `txy` has `nfft`/2 + 1 rows and is computed over the interval [0,π] rad/sample. If `nfft` is odd, `txy` has (`nfft` + 1)/2 rows and the interval is [0,π) rad/sample. If you specify `fs`, the corresponding intervals are [0,`fs`/2] cycles/unit time for even `nfft` and [0,`fs`/2) cycles/unit time for odd `nfft`.
- `'twosided'` — Returns the two-sided estimate of the transfer function between two real-valued or complex-valued input signals, `x` and `y`. In this case, `txy` has `nfft` rows and is computed over the interval [0,2π) rad/sample. If you specify `fs`, the interval is [0,`fs`) cycles/unit time.
- `'centered'` — Returns the centered two-sided estimate of the transfer function between two real-valued or complex-valued input signals, `x` and `y`. In this case, `txy` has `nfft` rows and is computed over the interval (–π,π] rad/sample for even `nfft` and (–π,π) rad/sample for odd `nfft`. If you specify `fs`, the corresponding intervals are (–`fs`/2, `fs`/2] cycles/unit time for even `nfft` and (–`fs`/2, `fs`/2) cycles/unit time for odd `nfft`.

`est` — Transfer function estimator
`'H1'` (default) | `'H2'`

Transfer function estimator, specified as `'H1'` or `'H2'`.

- Use `'H1'` when the noise is uncorrelated with the input signals.
- Use `'H2'` when the noise is uncorrelated with the output signals. In this case, the number of input signals must equal the number of output signals.

See “Transfer Function” on page 1-2648 for more information.

Output Arguments

`txy` — Transfer function estimate
vector | matrix | three-dimensional array

Transfer function estimate, returned as a vector, matrix, or three-dimensional array.

`w` — Normalized frequencies
vector

Normalized frequencies, returned as a real-valued column vector.

`f` — Cyclical frequencies
vector

Cyclical frequencies, returned as a real-valued column vector.

More About

Transfer Function

The relationship between the input x and output y is modeled by the linear, time-invariant *transfer function* `txy`. In the frequency domain, $Y(f) = H(f)X(f)$.

- For a single-input/single-output system, the H_1 estimate of the transfer function is given by

  $$H_1(f) = \frac{P_{yx}(f)}{P_{xx}(f)},$$

  where P_{yx} is the cross power spectral density of x and y, and P_{xx} is the power spectral density of x. This estimate assumes that the noise is not correlated with the system input.

  For multi-input/multi-output (MIMO) systems, the H_1 estimator becomes

  $$H_1(f) = P_{YX}(f)P_{XX}^{-1}(f) = \begin{bmatrix} P_{y_1x_1}(f) & P_{y_1x_2}(f) & \cdots & P_{y_1x_m}(f) \\ P_{y_2x_1}(f) & P_{y_2x_2}(f) & \cdots & P_{y_2x_m}(f) \\ \vdots & \vdots & \ddots & \vdots \\ P_{y_nx_1}(f) & P_{y_nx_2}(f) & \cdots & P_{y_nx_m}(f) \end{bmatrix} \begin{bmatrix} P_{x_1x_1}(f) & P_{x_1x_2}(f) & \cdots & P_{x_1x_m}(f) \\ P_{x_2x_1}(f) & P_{x_2x_2}(f) & \cdots & P_{x_2x_m}(f) \\ \vdots & \vdots & \ddots & \vdots \\ P_{x_mx_1}(f) & P_{x_mx_2}(f) & \cdots & P_{x_mx_m}(f) \end{bmatrix}^{-1}$$

  for m inputs and n outputs, where:

  - $P_{y_ix_k}$ is the cross power spectral density of the kth input and the ith output.
  - $P_{x_ix_k}$ is the cross power spectral density of the kth and ith inputs.

  For two inputs and two outputs, the estimator is the matrix

  $$H_1(f) = \frac{\begin{bmatrix} P_{y_1x_1}(f)P_{x_2x_2}(f) - P_{y_1x_2}(f)P_{x_2x_1}(f) & P_{y_1x_2}(f)P_{x_1x_1}(f) - P_{y_1x_1}(f)P_{x_1x_2}(f) \\ P_{y_2x_1}(f)P_{x_2x_2}(f) - P_{y_2x_2}(f)P_{x_2x_1}(f) & P_{y_2x_2}(f)P_{x_1x_1}(f) - P_{y_2x_1}(f)P_{x_1x_2}(f) \end{bmatrix}}{P_{x_1x_1}(f)P_{x_2x_2}(f) - P_{x_1x_2}(f)P_{x_2x_1}(f)}.$$

- For a single-input/single-output system, the H_2 estimate of the transfer function is given by

  $$H_2(f) = \frac{P_{yy}(f)}{P_{xy}(f)},$$

  where P_{yy} is the power spectral density of y and $P_{xy} = P^*_{yx}$ is the complex conjugate of the cross power spectral density of x and y. This estimate assumes that the noise is not correlated with the system output.

  For MIMO systems, the H_2 estimator is well-defined only for equal numbers of inputs and outputs: $n = m$. The estimator becomes

  $$H_2(f) = P_{YY}(f)P_{XY}^{-1}(f) = \begin{bmatrix} P_{y_1y_1}(f) & P_{y_1y_2}(f) & \cdots & P_{y_1y_n}(f) \\ P_{y_2y_1}(f) & P_{y_2y_2}(f) & \cdots & P_{y_2y_n}(f) \\ \vdots & \vdots & \ddots & \vdots \\ P_{y_ny_1}(f) & P_{y_ny_2}(f) & \cdots & P_{y_ny_n}(f) \end{bmatrix} \begin{bmatrix} P_{x_1y_1}(f) & P_{x_1y_2}(f) & \cdots & P_{x_1y_n}(f) \\ P_{x_2y_1}(f) & P_{x_2y_2}(f) & \cdots & P_{x_2y_n}(f) \\ \vdots & \vdots & \ddots & \vdots \\ P_{x_ny_1}(f) & P_{x_ny_2}(f) & \cdots & P_{x_ny_n}(f) \end{bmatrix}^{-1},$$

where:

- $P_{y_iy_k}$ is the cross power spectral density of the kth and ith outputs.
- $P_{x_iy_k}$ is the complex conjugate of the cross power spectral density of the ith input and the kth output.

Algorithms

`tfestimate` uses Welch's averaged periodogram method. See `pwelch` for details.

References

[1] Vold, Håvard, John Crowley, and G. Thomas Rocklin. “New Ways of Estimating Frequency Response Functions.” *Sound and Vibration*. Vol. 18, November 1984, pp. 34–38.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Arguments specified using name value pairs must be compile time constants.
- Variable sized `window` must be double precision.

See Also

`cpsd` | `mscohere` | `periodogram` | `pwelch`

Introduced before R2006a

tfridge

Time-frequency ridges

Syntax

```
fridge = tfridge(tfm,f)
[fridge,iridge] = tfridge(tfm,f)
[fridge,iridge,lridge] = tfridge(tfm,f)

[___] = tfridge(tfm,f,penalty)

[___] = tfridge(___,'NumRidges',nr)
[___] = tfridge(___,'NumRidges',nr,'NumFrequencyBins',nbins)
```

Description

`fridge = tfridge(tfm,f)` extracts the maximum-energy time-frequency ridge from the time-frequency matrix, `tfm`, and the frequency vector, `f`, and outputs the time-dependent frequency, `fridge`.

`[fridge,iridge] = tfridge(tfm,f)` also returns the row-index vector corresponding to the maximum-energy ridge.

`[fridge,iridge,lridge] = tfridge(tfm,f)` also returns the linear indices, `lridge`, such that `tfm(lridge)` are the values of `tfm` along the maximum-energy ridge.

`[___] = tfridge(tfm,f,penalty)` penalizes changes in frequency by scaling the squared distance between frequency bins by `penalty`.

`[___] = tfridge(___,'NumRidges',nr)` extracts the `nr` time-frequency ridges with the highest energy. This syntax accepts any combination of input arguments from previous syntaxes.

`[___] = tfridge(___,'NumRidges',nr,'NumFrequencyBins',nbins)` specifies the number of frequency bins around a ridge that are removed from `tfm` when extracting multiple ridges.

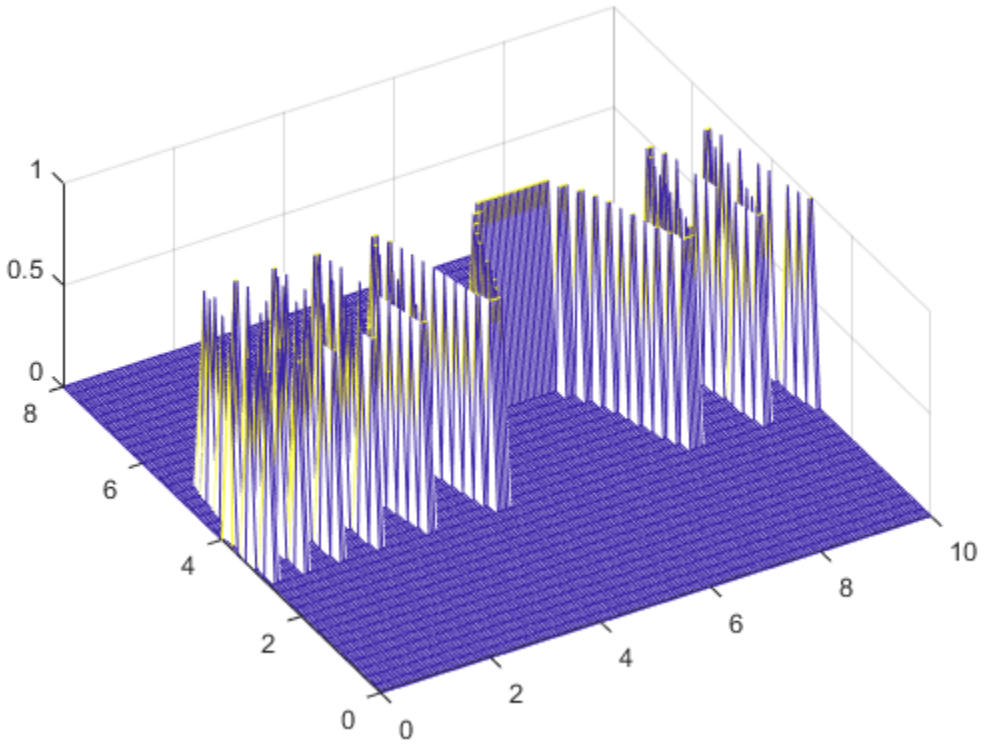
Examples

Find Ridge of Noisy Signal

Create a matrix that resembles a time-frequency matrix with a sharp ridge. Visualize the matrix in three dimensions.

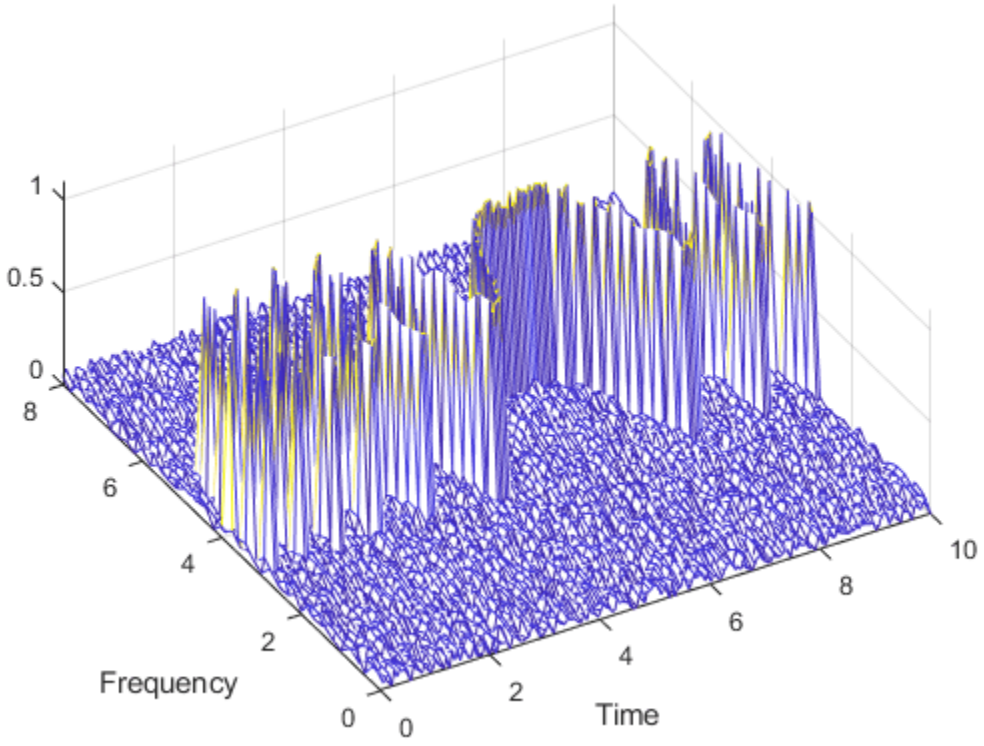
```
t = 0:0.05:10;
f = 0:0.2:8;
rv = 1;

[F,T] = ndgrid(f,t);

S = zeros(size(T));
S(abs((F-4)-cos((T-6).^2))<0.1) = rv;
```

```
mesh(T,F,S)
view(-30,60)
```



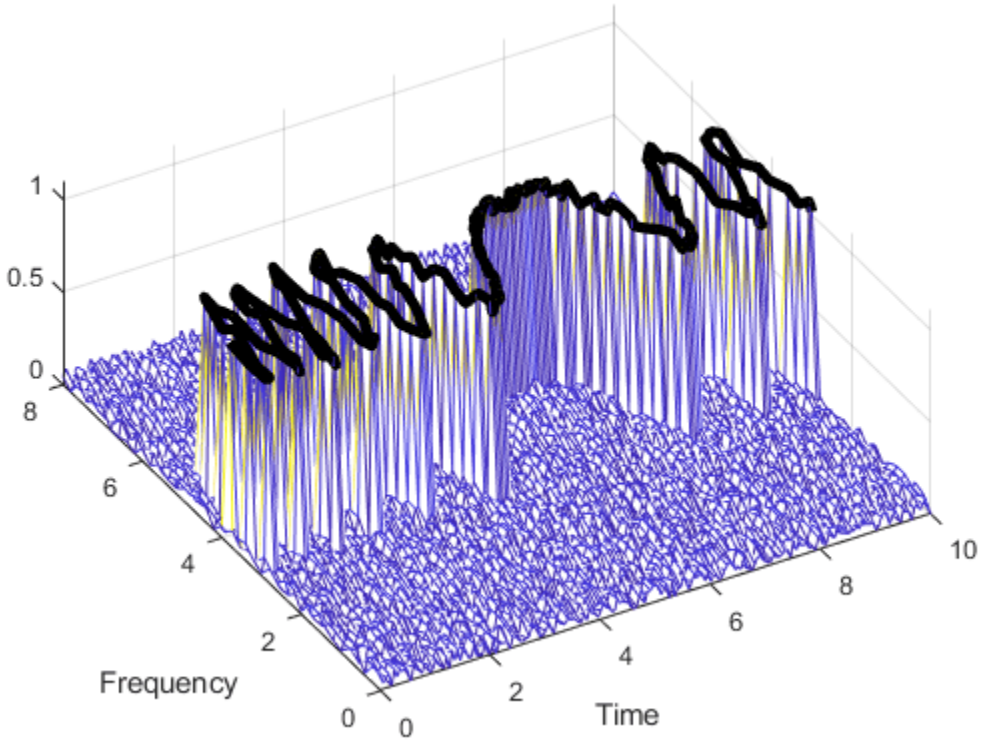
Add noise to the matrix and redisplay the plot.

```
S = S+rand(size(S))/10;

mesh(T,F,S)
view(-30,60)
xlabel('Time')
ylabel('Frequency')
```



Extract the ridge and plot the result.

```
[fridge,~,lridge] = tfridge(S,f);

rvals = S(lridge);

hold on
plot3(t,fridge,rvals,'k','linewidth',4)
hold off
```



Extract High-Energy Ridges from Multicomponent Signal

Generate a signal that is sampled at 3 kHz for one second. The signal consists of two tones and a quadratic chirp.

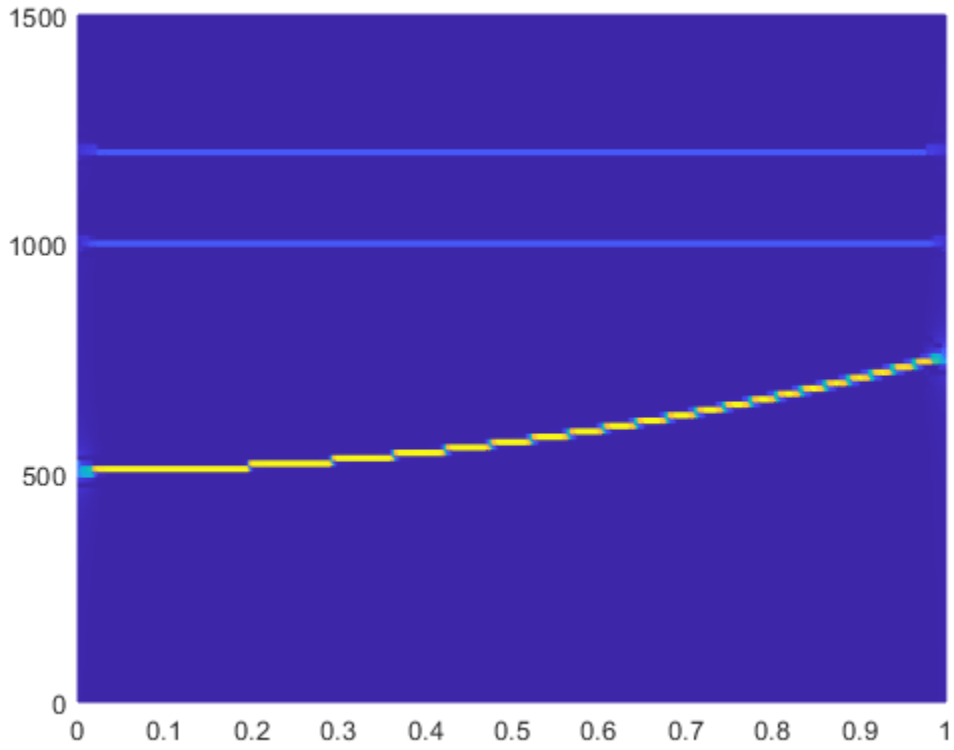
- The first tone has a frequency of 1000 Hz and unit amplitude.
- The second tone has a frequency of 1200 Hz and unit amplitude.
- The chirp has an initial frequency of 500 Hz and reaches 750 Hz at the end of the sampling. It has an amplitude of six.

```
fs = 3000;
t = 0:1/fs:1-1/fs;

x1 = 6*chirp(t,fs/6,t(end),fs/4,'quadratic');
x2 = sin(2*pi*fs/3*t);
x3 = sin(2*pi*fs/2.5*t);

x = x1+x2+x3;
```

Compute and display the Fourier synchrosqueezed transform of the signal.

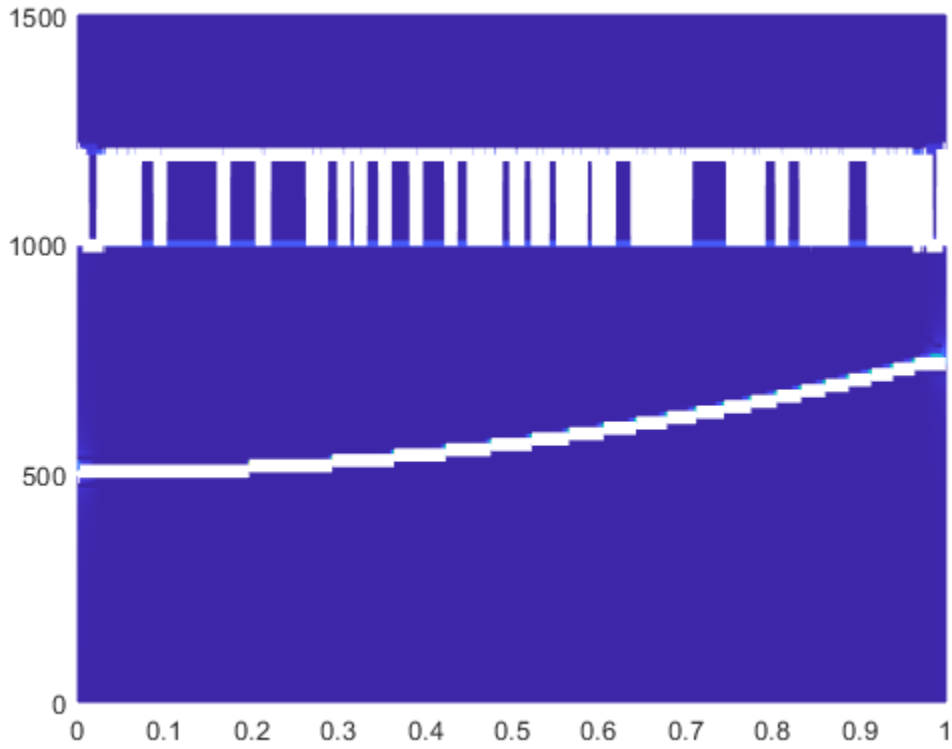
```
[sst,f] = fsst(x,fs);
mx = max(abs(sst(:)))*ones(size(t));
```

```
mesh(t,f,abs(sst))
view(2)
```



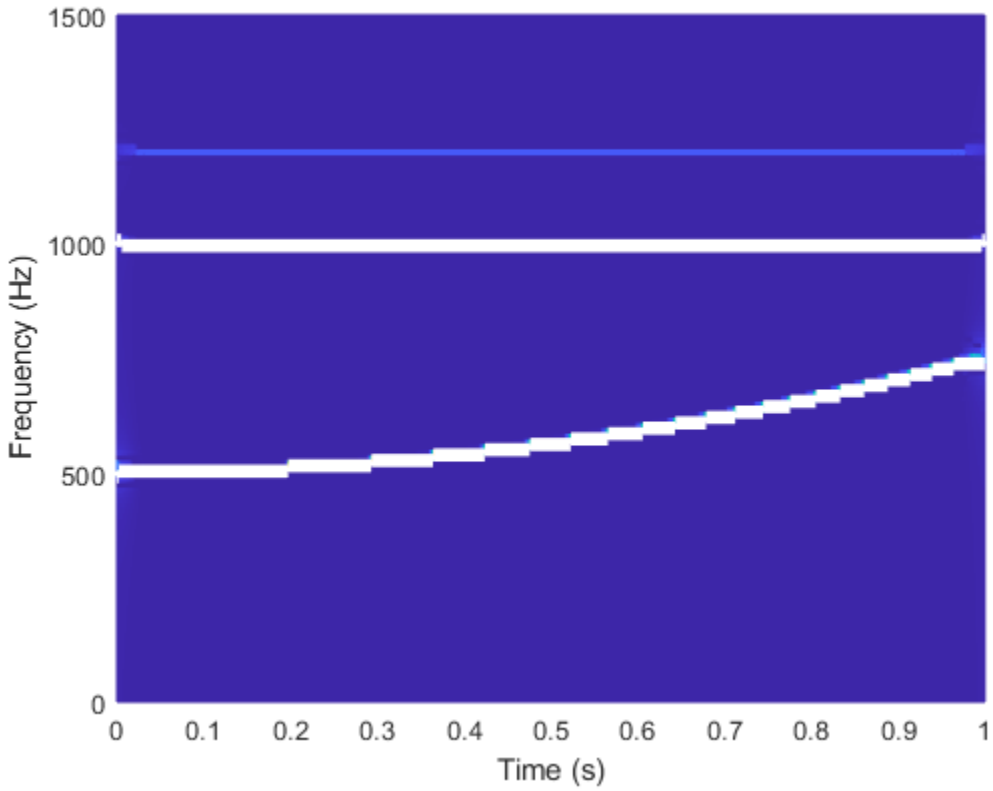
Extract and plot the two highest-energy signal components. Set no penalty for changing frequency.

```
penval = 0;

fridge = tfridge(sst,f,penval,'NumRidges',2);

hold on
plot3(t,fridge,mx,'w','linewidth',5)
hold off
```



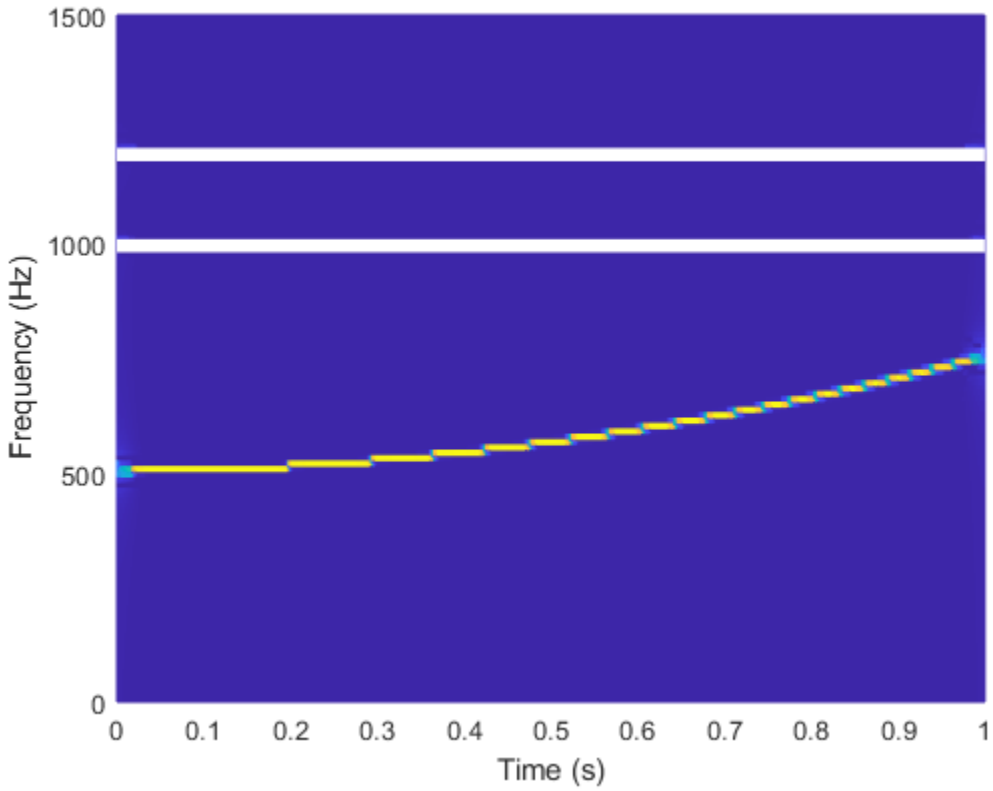
The two tones have the same amplitude, and the algorithm jumps between them. Set the penalty for changing frequency to 1.

```
penval = 1;

fridge = tfridge(sst,f,penval,'NumRidges',2);

mesh(t,f,abs(sst))
view(2)
xlabel('Time (s)')
ylabel('Frequency (Hz)')

hold on
plot3(t,fridge,mx,'w','linewidth',5)
hold off
```



Set the penalty to a high value for comparison. The chirp is penalized because its frequency is not constant.

```
penval = 1000;

fridge = tfridge(sst,f,penval,'NumRidges',2);

mesh(t,f,abs(sst))
view(2)
xlabel('Time (s)')
ylabel('Frequency (Hz)')

hold on
plot3(t,fridge,mx,'w','linewidth',5)
hold off
```



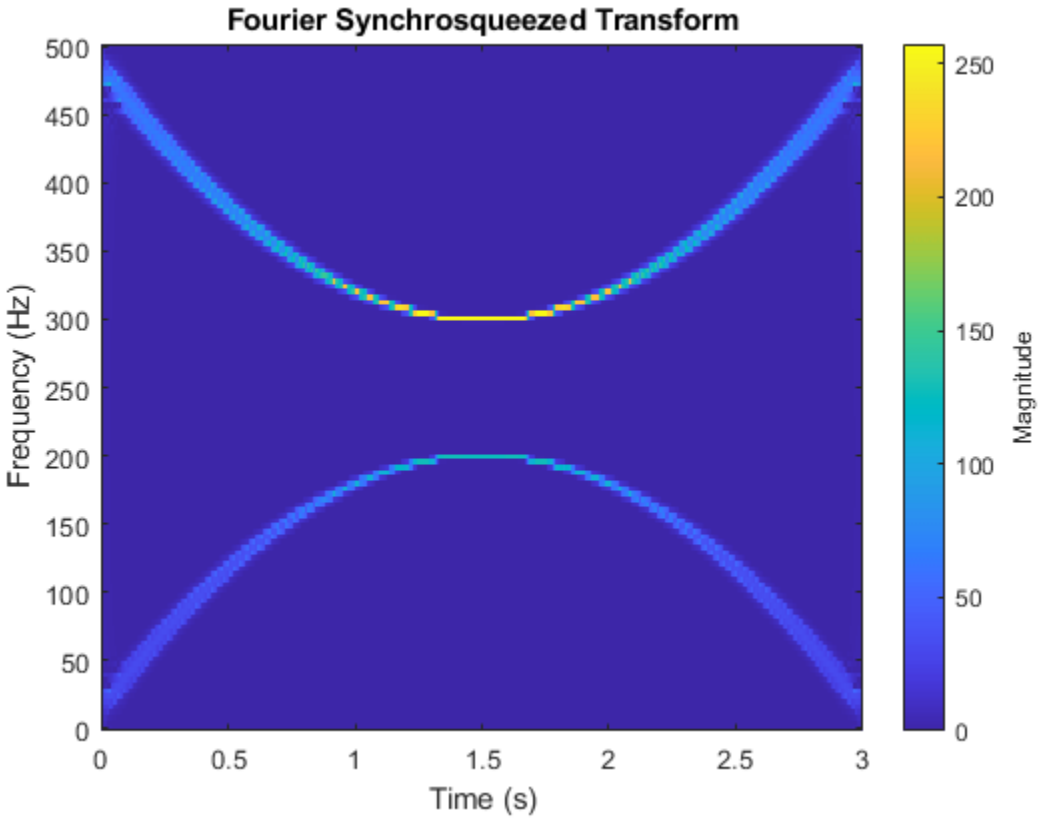
Effect of Neighbor Bin Removal

Generate a signal composed of two quadratic chirps. The signal is sampled at 1 kHz for 3 seconds. The chirps are such that the instantaneous frequency is symmetric about the halfway point of the sampling interval. One chirp is concave and the other chirp is convex. The concave chirp has twice the amplitude of the convex chirp.

```
fs = 1e3;
t = 0:1/fs:3;

x =   chirp(t-1.5,100,1.1,200,'quadratic',[],'convex');
y = 2*chirp(t-1.5,300,1.1,400,'quadratic',[],'concave');

% To hear, type soundsc(x+y,fs)
```

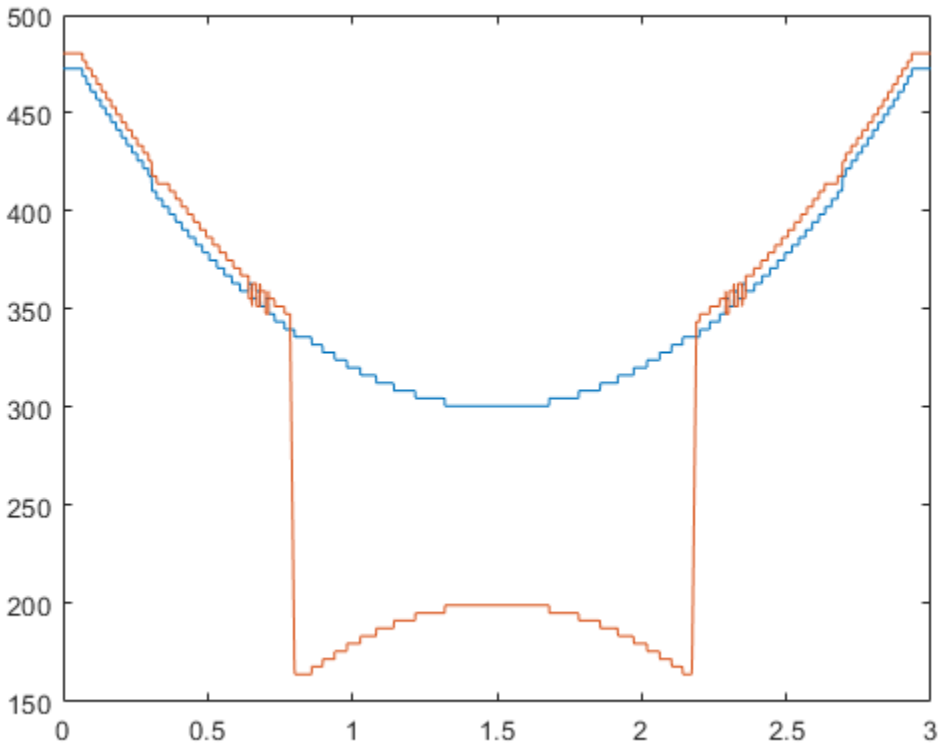
Compute and display the Fourier synchrosqueezed transform of the signal.

```
sig = x+y;

[sst,f,t] = fsst(sig,fs);

fsst(sig,fs,'yaxis')
```



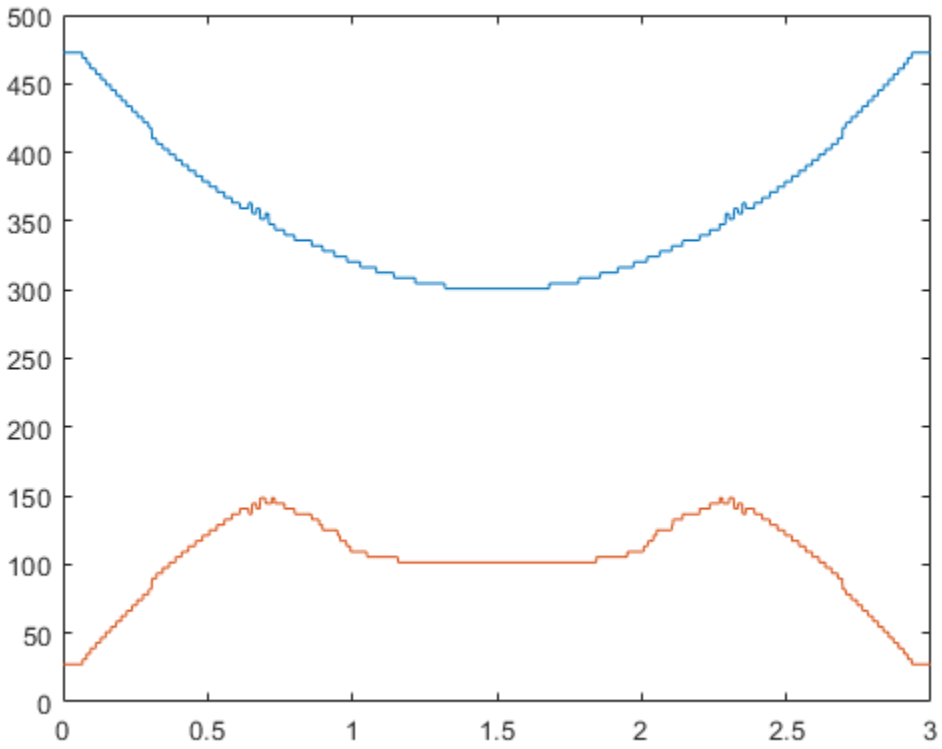
Extract the two time-frequency ridges that have the highest energy. Specify a penalty of 1 for changes in frequency. Remove 1 frequency bin around the ridge with the highest energy before extracting the second ridge. Plot the ridges.

```
nfb = 1;
[fr,ir] = tfridge(sst,f,1,'NumRidges',2,'NumFrequencyBins',nfb);

plot(t,fr)
```



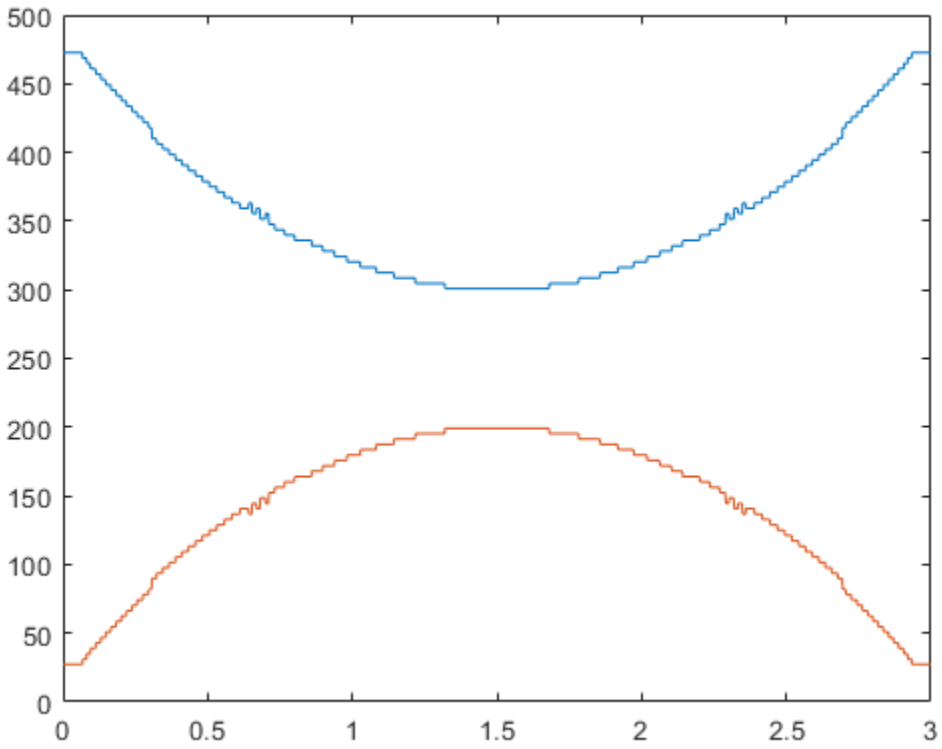
One bin is insufficient: The function finds a second ridge that is partly on the slope of the first. Increase to 50 the number of bins to remove and repeat the calculation.

```
nfb = 50;
[fr,ir] = tfridge(sst,f,1,'NumRidges',2,'NumFrequencyBins',nfb);

plot(t,fr)
```



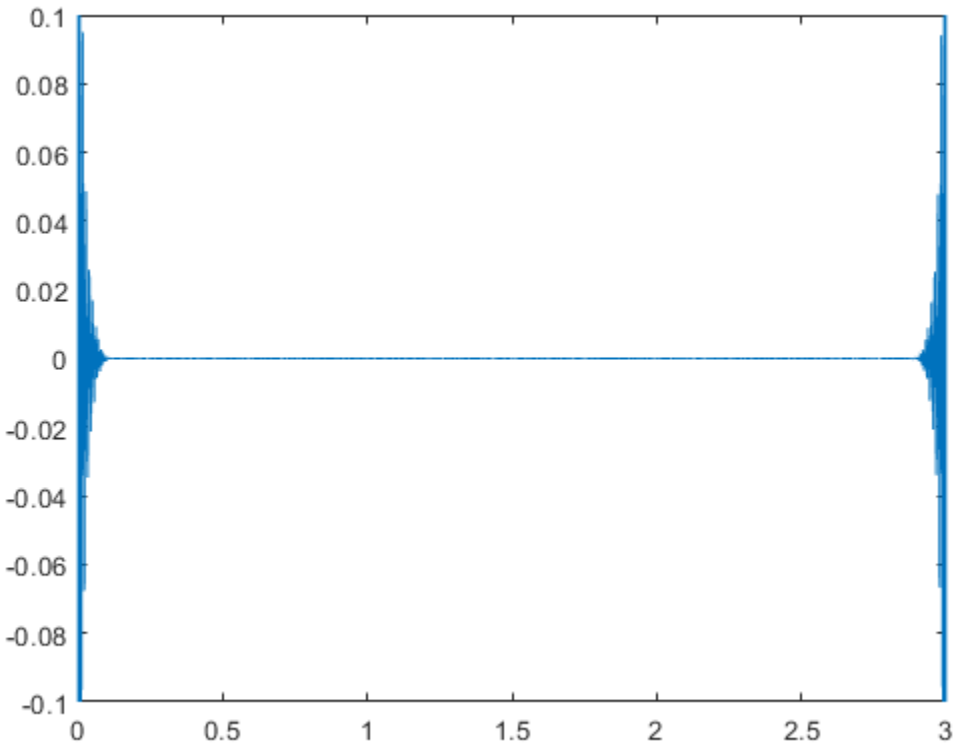
Removing too many bins distorts lower-energy ridges. Decrease the number to 15 and repeat the calculation.

```
nfb = 15;
[fr,ir] = tfridge(sst,f,1,'NumRidges',2,'NumFrequencyBins',nfb);

plot(t,fr)
```



Invert the transform corresponding to the two ridges. Add the ridges to reconstruct the signal. Plot the difference between the reconstructed signal and the chirps.

```
itr = ifsst(sst,[],ir,'NumFrequencyBins',nfb);
xrec = sum(itr');

plot(t,xrec-(x+y))
ylim([-.1 .1])
```




```
% To hear, type soundsc(xrec,fs)
```

The agreement is good most of the time but deteriorates at the ends, where frequencies change most rapidly.

Input Arguments

`tfm` — Time-frequency matrix

matrix

Time-frequency matrix, specified as a matrix.

Example: `fsst(cos(pi/4*(0:159)))` specifies the synchrosqueezed transform of a sinusoid.

Data Types: `single` | `double`
Complex Number Support: Yes

`f` — Sampling frequencies

vector

Sampling frequencies, specified as a vector. The length of `f` must equal the number of rows in `tfm`.

Data Types: `single` | `double`

`penalty` — Penalty for changing frequency

0 (default) | nonnegative real scalar

Penalty for changing frequency, specified as a nonnegative real scalar.

Data Types: `single` | `double`

`nr` — Number of time-frequency ridges to extract

1 (default) | positive integer scalar

Number of time-frequency ridges to extract, specified as the comma-separated pair consisting of `'NumRidges'` and a positive integer scalar. You can specify this name-value pair anywhere after `tfm` in the input argument list.

When `nr` is greater than 1, `tfridge`:

1. Extracts the time-frequency ridge with the highest energy
2. Removes from `tfm` the energy contained in the ridge it extracted and in the `nbins` adjacent frequency bins on either side of the ridge
3. Extracts the highest-energy ridge in the modified `tfm`
4. Iterates until it has extracted `nr` ridges

Data Types: `single` | `double`

`nbins` — Number of bins to remove

4 (default) | positive integer scalar

Number of bins to remove when extracting multiple ridges, specified as the comma-separated pair consisting of `'NumFrequencyBins'` and a positive integer scalar. `nbins` must be smaller than 1/4 of the sampling frequencies. Indices close to the frequency edges that have fewer than `nbins` bins on one side are reconstructed using a smaller number of bins.

Data Types: `single` | `double`

Output Arguments

`fridge` — Time-frequency ridges

matrix

Time-frequency ridges, returned as a matrix with `nr` columns. The number of rows in `fridge` equals the number of columns in `tfm`. The first column contains the frequencies that correspond to the highest-energy ridge. Subsequent columns contain the frequencies of the other ridges in decreasing order of energy.

`iridge` — Ridge row indices

matrix

Ridge row indices, returned as a matrix with `nr` columns. The number of rows in `iridge` equals the number of columns in `tfm`. The first column contains the indices that correspond to the highest-energy ridge. Subsequent columns contain the indices of the other ridges in decreasing order of energy.

`lridge` — Ridge linear indices

matrix

Ridge linear indices, returned as a matrix with `nr` columns. `lridge` is defined so that `tfm(lridge)` is the amplitude of `tfm` along the ridges. The number of rows in `lridge` equals the number of

columns in `tfm`. The first column contains the indices that correspond to the highest-energy ridge. Subsequent columns contain the indices of the other ridges in decreasing order of energy.

Example: `lridge` is equivalent to `sub2ind(size(tfm),iridge,repmat((1:size(tfm,2))',1,nr))`.

Algorithms

The function uses a penalized forward-backward greedy algorithm to extract the maximum-energy ridges from a time-frequency matrix. The algorithm finds the maximum time-frequency ridge by minimizing –ln *A* at each time point, where *A* is the absolute value of the matrix. Minimizing –ln *A* is equivalent to maximizing the value of *A*. The algorithm optionally constrains jumps in frequency with a penalty that is proportional to the distance between frequency bins.

The following example illustrates the time-frequency ridge algorithm using a penalty that is two times the distance between frequency bins. Specifically, the distance between the elements `(j,k)` and `(m,n)` is defined as $(j-m)^2$. The time-frequency matrix has three frequency bins and three time steps. The matrix columns correspond to time steps, and the matrix rows correspond to frequency bins. The values in the second row represent a sine wave.

1 Suppose you have the matrix:

```
1   4   4
2   2   2
5   5   4
```

2 Update the value for the (1,2) element as follows.

a Leave the values at the first time point unaltered. Begin the algorithm with the (1,2) element of the matrix, which presents the first frequency bin at the second time point. The bin value is 4. Penalize the values in the first column based on their distance from the (1,2) element. Applying the penalty to the first column produces

```
original value + penalty × distance

1 + 2 × 0 =  1
2 + 2 × 1 =  4
5 + 2 × 4 = 13

 1   4
 4   2
13   5
```

The minimum value of the first column is 1, which is in bin 1.

b Add the minimum value in column 1 to the current bin value, 4. The updated value for (1,2) becomes 5, which came from bin 1.

3 Update the values for the remaining elements in column 2 as follows.

Recompute the original column 1 values with the penalty factor using the same process as in Step 2a. Obtain the remaining second column values using the same process as in Step 2b. For example, when updating the (2,2) element, which has bin value 2, applying the penalty to the column yields

```
original value + penalty × distance

1 + 2 × 1 =  3
```

```
2 + 2 × 0 =  2
5 + 2 × 1 =  7
```

Add the minimum value, 2, to the current bin value. The updated value for (2,2) becomes 4. After updating the (3,2) element, the matrix is

```
1   5(1)  4
2   4(2)  2
5   9(2)  4
```

Only the second column has been updated. The subscripts indicate the index of the bin in the previous column from which a value came.

4. Repeat Step 2 for the third column. But now the penalty is applied to the updated second column. For example, when updating the (1,3) element, the penalty is

```
5 + 2 × 0 =  5
4 + 2 × 1 =  6
9 + 2 × 4 = 17
```

The minimum value, 5, which is in the first bin, is added to the (1,3) bin value. After updating all the values in the third column, the final matrix is

```
1   5(1)   9(1)
2   4(2)   6(2)
5   9(2)  10(2)
```

5. Starting at the last column of the matrix, find the minimum value. Walk back in time through the matrix by going from the current bin to the origin of that bin at the previous time point. Keep track of the bin indices, which form the path composing the ridge. The algorithm smooths the transition by using the origin bin instead of the bin with the minimum value. For this example, the ridge indices are 2, 2, 2, which matches the energy path of the sine wave in row 2 of the matrix shown in Step 1.

If you are extracting multiple ridges, the algorithm removes the first ridge from the time-frequency matrix and repeats the process.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Arguments specified using name value pairs must be compile time constants.

See Also

Apps

Signal Analyzer

Functions

`fsst` | `ifsst` | `pspectrum` | `spectrogram` | `sub2ind`

Topics

“Practical Introduction to Time-Frequency Analysis”

Introduced in R2016b

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 = thd(___,'aliased')

[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 sample 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 = thd(___,'aliased')` reports harmonics of the fundamental that are aliased into the Nyquist range. Use this option when the input signal is undersampled. If you do not specify this option, or if you set it to `'omitaliases'`, then the function ignores any harmonics of the fundamental frequency that lie beyond the Nyquist range.

`[r,harmpow,harmfreq] = thd(___)` returns the powers (in dB) 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)
```

```
tharmdist = -45.0515
```

```
r = thd(x)
```

```
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.

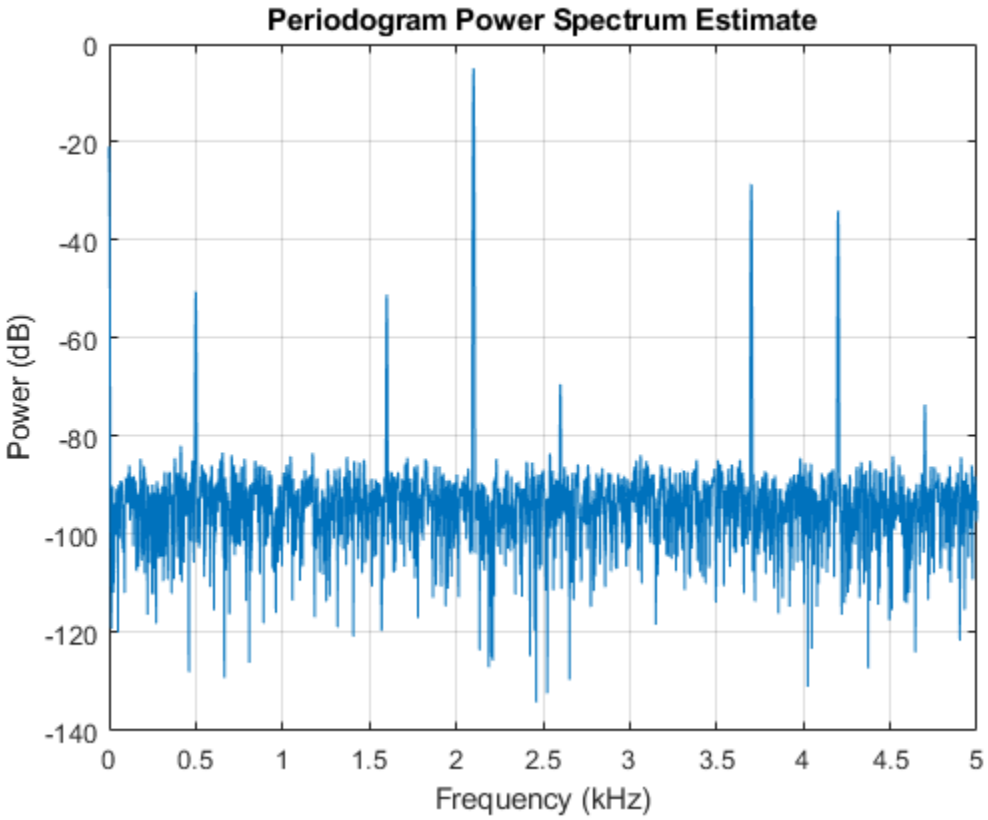
```
fs = 10000;
t = 0:1/fs:1-1/fs;
x = 2*cos(2*pi*100*t)+0.01*cos(2*pi*300*t)+ ...
    0.005*cos(2*pi*500*t)+0.0025*sin(2*pi*700*t);
[sxx,f] = periodogram(x,hamming(length(x)),length(x),fs,'power');
rbw = enbw(hamming(length(x)),fs);
r = thd(sxx,f,rbw,7,'power')
```

```
r = -44.8396
```

THD with and Without Aliased Harmonics

Generate a signal that resembles the output of a weakly nonlinear amplifier with a 2.1 kHz tone as input. The signal is sampled for 1 second at 10 kHz. Compute and plot the power spectrum of the signal. Use a Kaiser window with $\beta = 38$ for the computation.

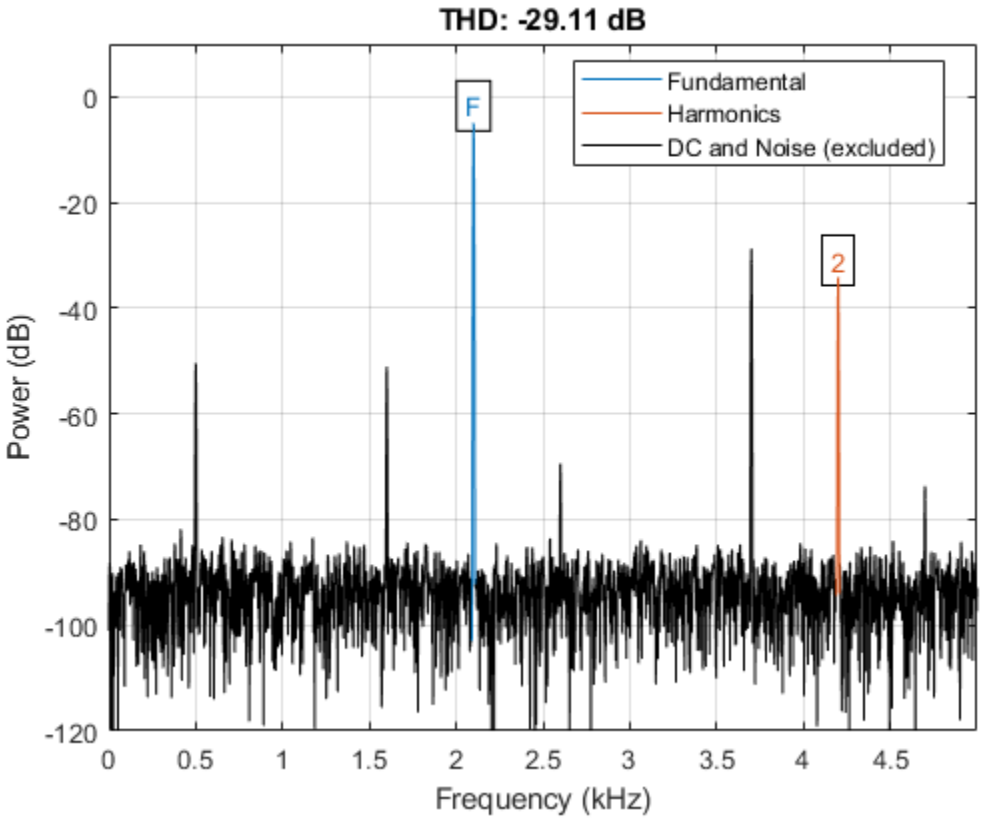
```
Fs = 10000;
f = 2100;

t = 0:1/Fs:1;
x = tanh(sin(2*pi*f*t)+0.1) + 0.001*randn(1,length(t));

periodogram(x,kaiser(length(x),38),[],Fs,'power')
```



Harmonics stick out from the noise at frequencies of 4.2 kHz, 6.3 kHz, 8.4 kHz, 10.5 kHz, 12.6 kHz, and 14.7 kHz. All frequencies except for the first one are greater than the Nyquist frequency. The harmonics are aliased respectively into 3.7 kHz, 1.6 kHz, 0.5 kHz, 2.6 kHz, and 4.7 kHz.

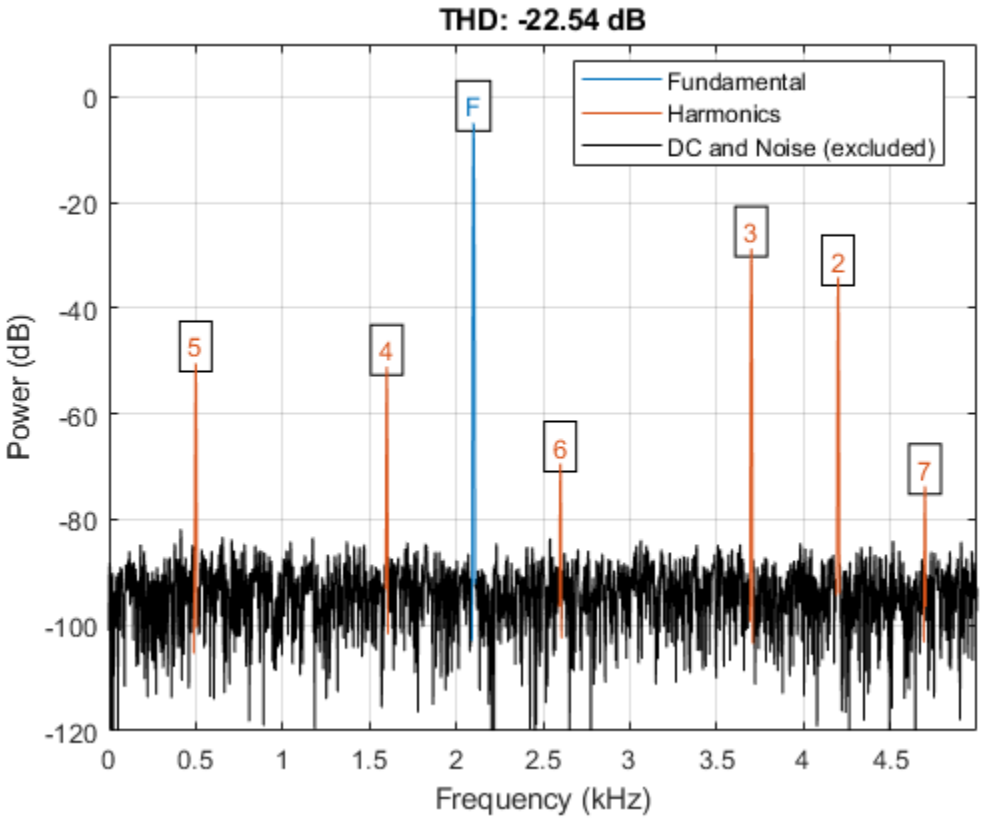
Compute the total harmonic distortion of the signal. By default, `thd` treats the aliased harmonics as part of the noise.

```
thd(x,Fs,7);
```



Repeat the computation, but now treat the aliased harmonics as part of the signal.

```
thd(x,Fs,7,'aliased');
```



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,harmpow,harmfreq] = thd(x,10000,7);
[harmfreq harmpow]
```

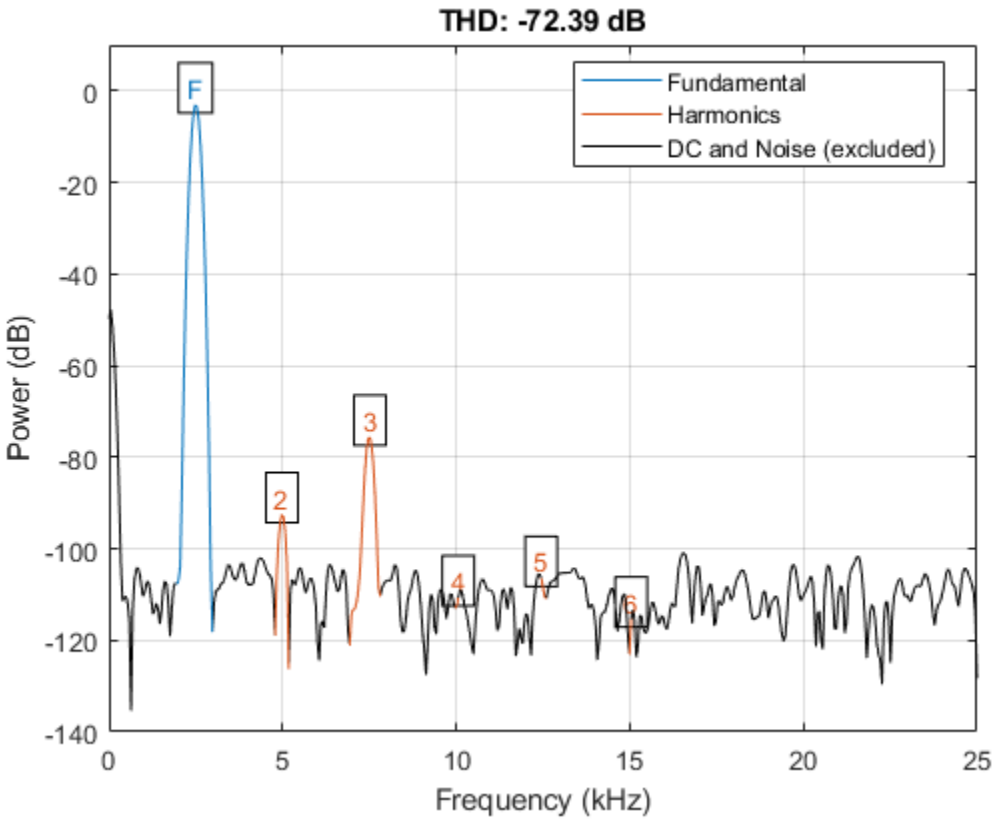
```
ans = 7×2

  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. Add Gaussian white noise with standard deviation 0.00005 to the signal. Pass the result through a weakly nonlinear amplifier. Plot the THD.

```
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.

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 — 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 has units of Hz.

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.

The power spectral density must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[pxx,f] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` specifies the periodogram PSD estimate of a noisy two-channel sinusoid sampled at 2π Hz and the frequencies at which it is computed.

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.

The power spectrum must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[sxx,w] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2),'power')` specifies the periodogram power spectrum estimate of a two-channel sinusoid embedded in white Gaussian noise and the normalized frequencies at which it is computed.

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, returned as a real-valued scalar.

`harmpow` — Power of the harmonics

real-valued scalar or vector

Power of the harmonics, returned as a real-valued scalar or vector expressed in dB. 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, returned 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If supplied, input arguments `'power'`, `'psd'`, `'aliased'`, and `'omitaliases'` must be compile-time constants.

See Also

`sfdr` | `sinad` | `snr` | `toi`

Topics

"Analyzing Harmonic Distortion"

Introduced in R2013b

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 sample 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,Fim3
```

```
myTOI = 1.3844

Fim3 = 1×2
10^3 ×

    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);
```

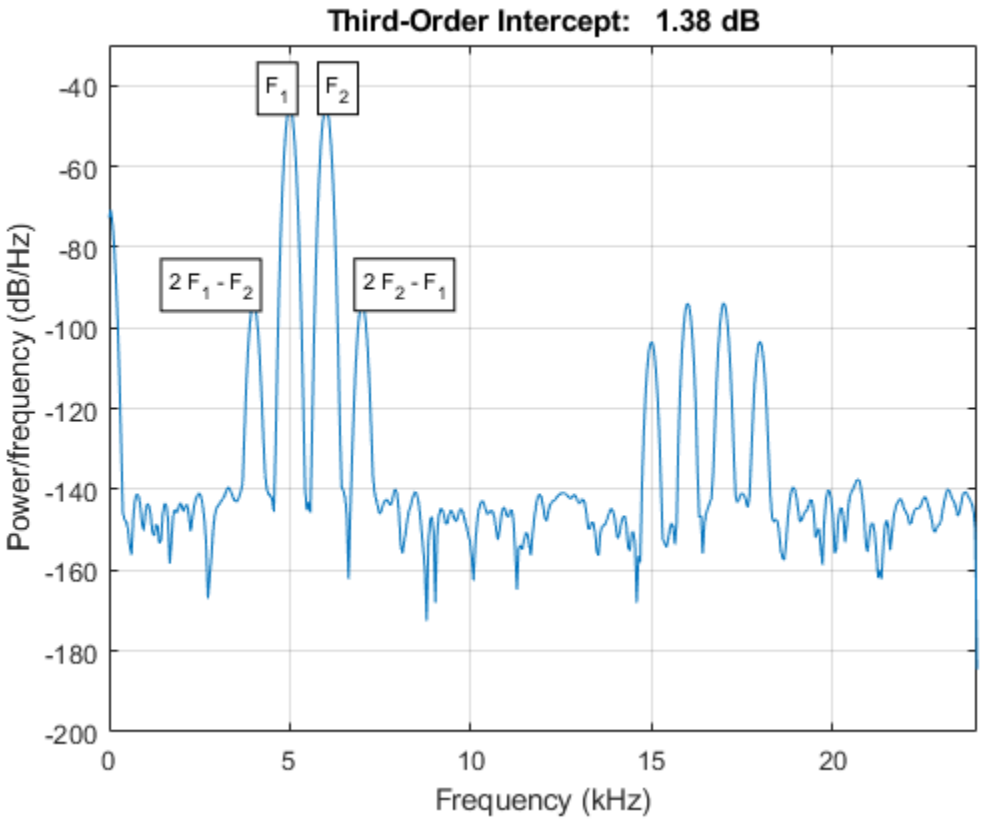
Evaluate the periodogram of the signal using a Kaiser window. Compute the TOI using the power spectral density. Plot the result.

```
w = kaiser(numel(y),38);

[Sxx, F] = periodogram(y,w,N,Fs,'psd');
[myTOI,Pfund,Ffund,Pim3,Fim3] = toi(Sxx,F,'psd')
```

```
myTOI = 1.3843

Pfund = 1×2

  -22.9133  -22.9132


Ffund = 1×2
10^3 ×

    5.0000    6.0000


Pim3 = 1×2

  -71.4868  -71.5299
```

```
Fim3 = 1×2
10^3 ×

    4.0002    6.9998
```

```
toi(Sxx,F,'psd');
```



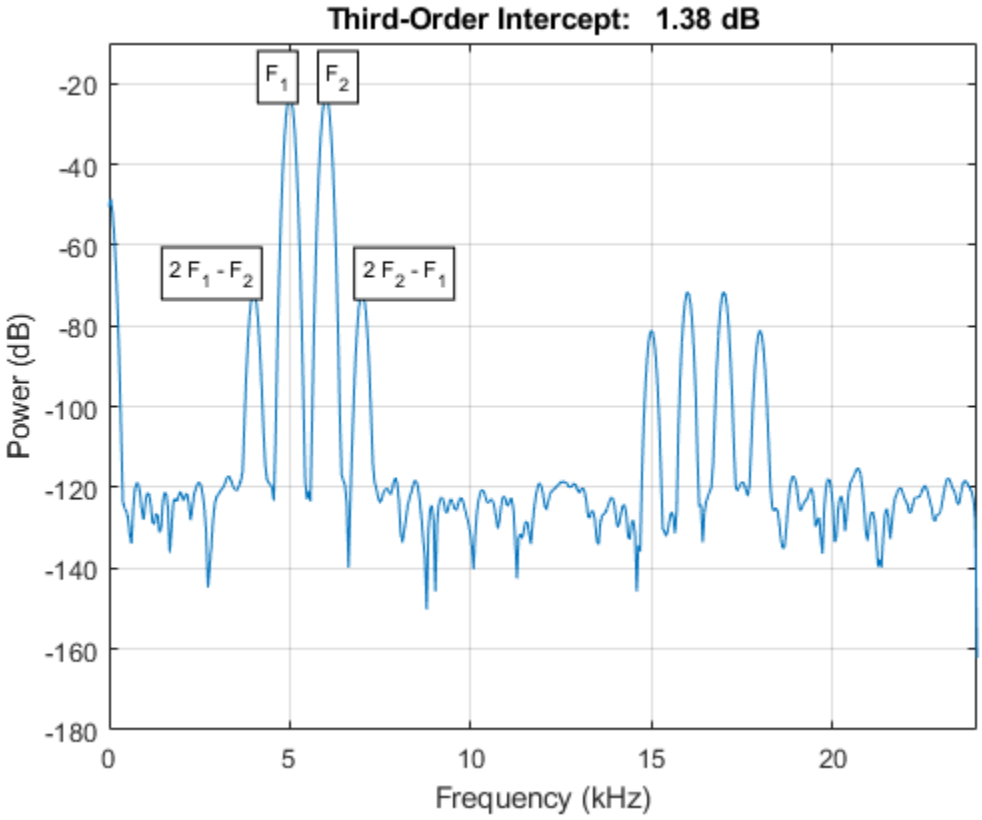
Third-Order Intercept Point from Power Spectrum

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)
```

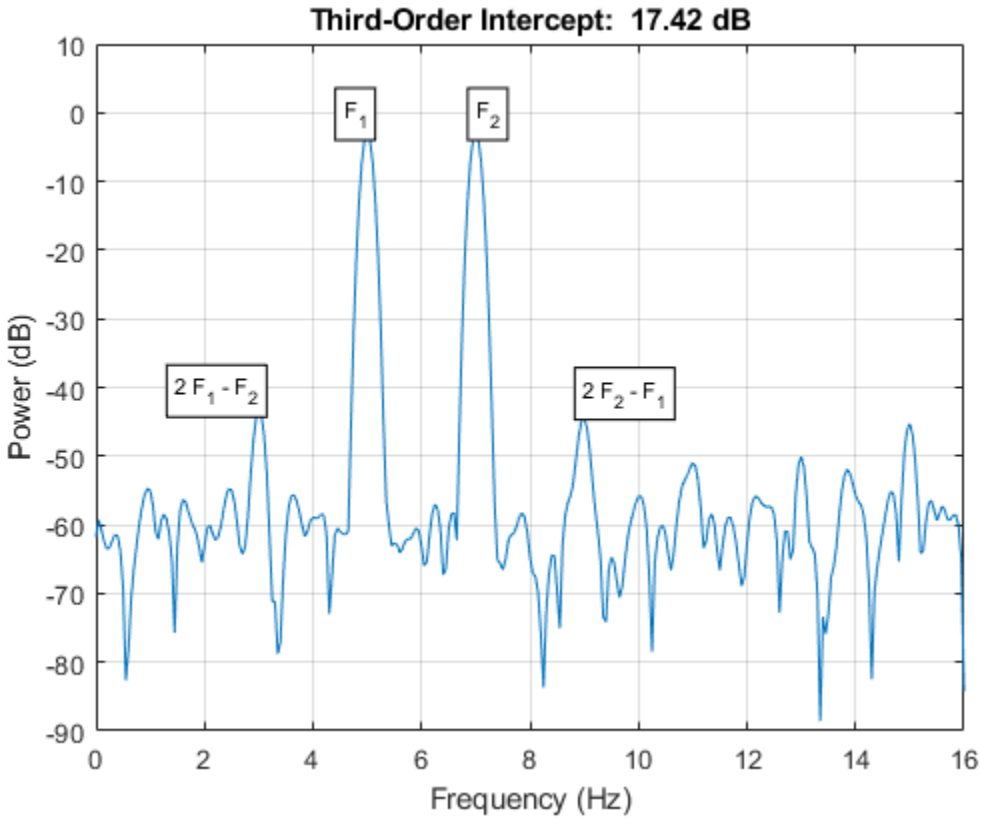
```
myTOI = 17.4230
```

```
Pfund = 1×2

   -2.8350   -2.8201

Ffund = 1×2

    5.0000    7.0001

Pim3 = 1×2

  -43.1362  -43.5211

Fim3 = 1×2

    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 — Sample rate

1 (default) | positive real scalar

Sample rate, specified as a positive real scalar. The sample rate is the number of samples per unit time. If the unit of time is seconds, then the sample rate 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.

The power spectral density must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[pxx,f] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2))` specifies the periodogram PSD estimate of a noisy two-channel sinusoid sampled at 2π Hz and the frequencies at which it is computed.

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.

The power spectrum must be expressed in linear units, not decibels. Use `db2pow` to convert decibel values to power values.

Example: `[sxx,w] = periodogram(cos(pi./[4;2]*(0:159))'+randn(160,2),'power')` specifies the periodogram power spectrum estimate of a two-channel sinusoid embedded in white Gaussian noise and the normalized frequencies at which it is computed.

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." *Designer's Guide Community*. May, 2002. https://designers-guide.org/analysis/intercept-point.pdf.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If supplied, input arguments `'power'` and `'psd'` must be compile-time constants.

See Also

`sfdr` | `sinad` | `snr` | `thd`

Introduced in R2013b

triang

Triangular window

Syntax

```
w = triang(L)
```

Description

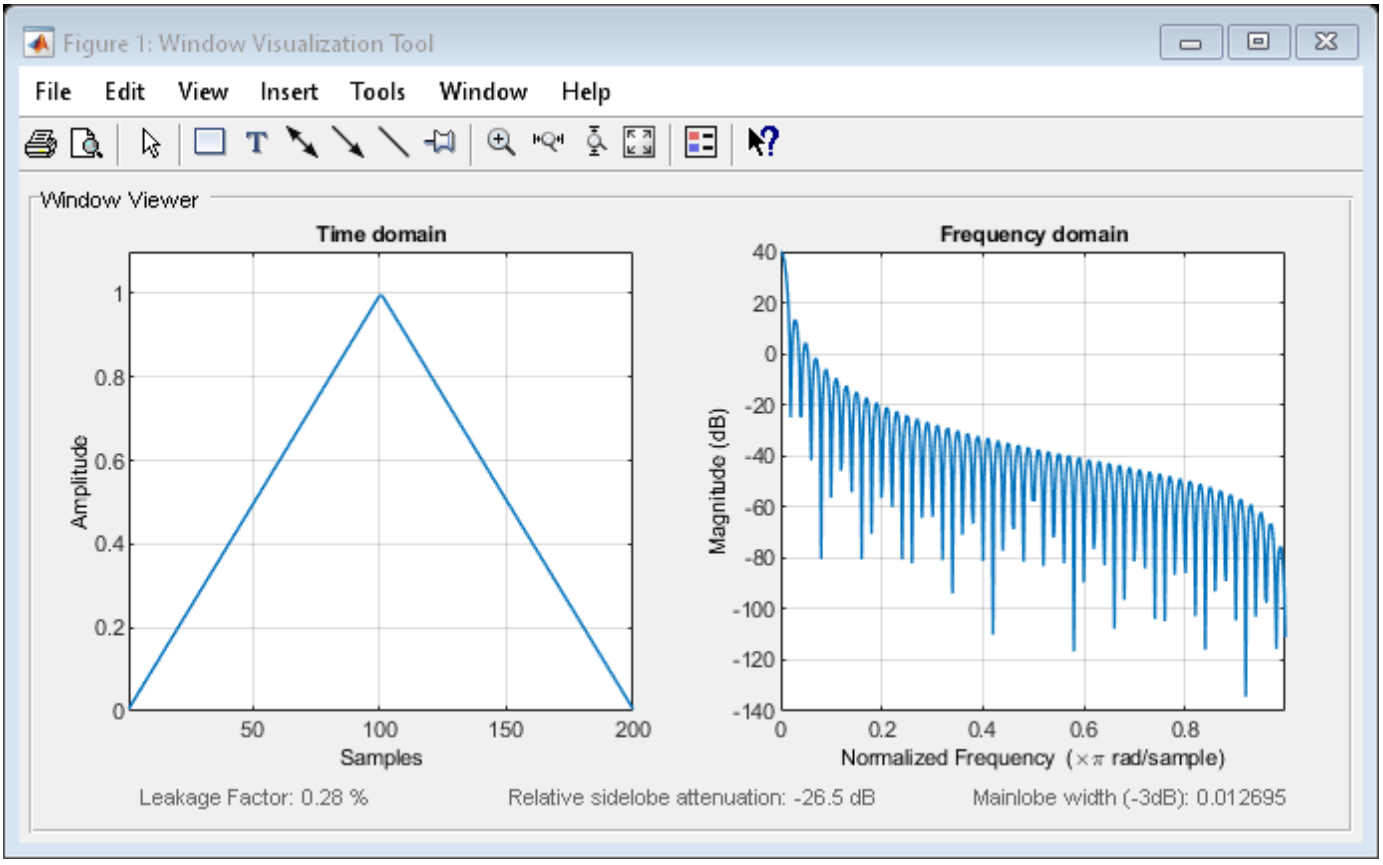
`w = triang(L)` returns an `L`-point triangular window.

Examples

Triangular Window

Create a 200-point triangular window. Display the result using `wvtool`.

```
L = 200;
w = triang(L);
wvtool(w)
```



Input Arguments

L — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

Output Arguments

w — Triangular window

column vector

Triangular window, returned as a column vector.

Algorithms

The coefficients of a triangular window are:

For L odd:

$$w(n) = \begin{cases} \dfrac{2n}{L+1} & 1 \le n \le \dfrac{L+1}{2} \\ 2 - \dfrac{2n}{L+1} & \dfrac{L+1}{2} + 1 \le n \le L \end{cases}$$

For L even:

$$w(n) = \begin{cases} \dfrac{(2n-1)}{L} & 1 \le n \le \dfrac{L}{2} \\ 2 - \dfrac{(2n-1)}{L} & \dfrac{L}{2} + 1 \le n \le L \end{cases}$$

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps

Window Designer

Functions

`flattopwin` | **WVTool** | `hamming` | `hann`

Introduced before R2006a

tripuls

Sampled aperiodic triangle

Syntax

```
y = tripuls(t)
y = tripuls(t,w,s)
```

Description

`y = tripuls(t)` returns a continuous, aperiodic, symmetric, unit-height triangular pulse at the sample times indicated in array `t`, centered about `t = 0`.

`y = tripuls(t,w,s)` generates a triangle of width `w` and skew `s`.

Examples

Generate and Displace Triangular Pulse

Generate 200 ms of a symmetric triangular pulse with a sample rate of 10 kHz and a width of 40 ms.

```
fs = 10e3;
t = -0.1:1/fs:0.1;

w = 40e-3;

x = tripuls(t,w);
```

Generate two variations of the same pulse:

- One displaced 45 ms into the past and skewed 45% to the left.

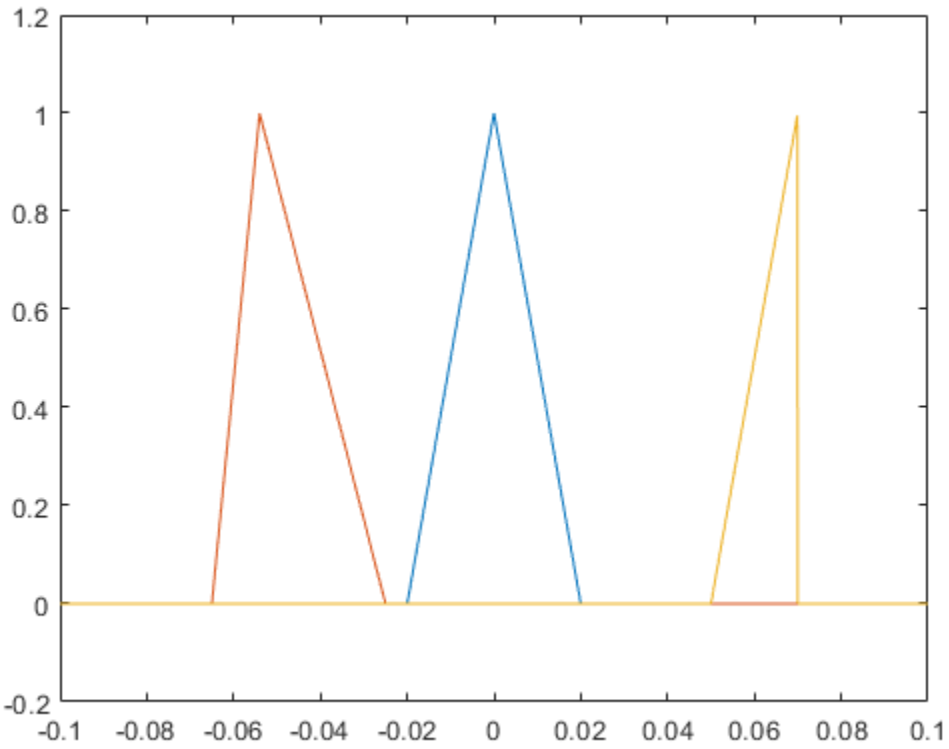
```
tpast = -45e-3;
spast = -0.45;
xpast = tripuls(t-tpast,w,spast);
```

- One displaced 60 ms into the future, half as wide, and skewed completely to the right.

```
tfutr = 60e-3;
sfutr = 1;
xfutr = tripuls(t-tfutr,w/2,sfutr);
```

Plot the original pulse and the two copies on the same axes.

```
plot(t,x,t,xpast,t,xfutr)
ylim([-0.2 1.2])
```



Input Arguments

`t` — Sample times

vector

Sample times of unit triangular pulse, specified as a vector.

Data Types: `single` | `double`

`w` — Triangle width

`1` (default) | positive number

Triangle width, specified as a positive number.

Data Types: `single` | `double`

`s` — Triangle skew

`0` (default) | real number

Triangle skew, specified as a real number such that –1 ≤ `s` ≤ 1. When `s` is 0, the function generates a symmetric triangular pulse.

Data Types: `single` | `double`

Output Arguments

y — Triangular pulse

vector

Triangular pulse of unit amplitude, returned as a vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`chirp` | `cos` | `diric` | `gauspuls` | `pulstran` | `rectpuls` | `sawtooth` | `sin` | `square`

Introduced before R2006a

tsa

Time-synchronous signal average

Syntax

```
ta = tsa(x,fs,tp)
ta = tsa(x,t,tp)
ta = tsa(xt,tp)

ta = tsa( ___ ,Name,Value)

[ta,t,p,rpm] = tsa( ___ )

tsa( ___ )
```

Description

`ta = tsa(x,fs,tp)` returns a time-synchronous average of a signal `x`, sampled at a rate `fs`, on the pulse times specified in `tp`.

`ta = tsa(x,t,tp)` returns a time-synchronous average of `x` sampled at the time values stored in `t`.

`ta = tsa(xt,tp)` returns a time-synchronous average of a signal stored in the MATLAB timetable `xt`.

`ta = tsa( ___ ,Name,Value)` specifies additional options for any of the previous syntaxes using name-value pair arguments. For example, you can specify the number of tachometer pulses per shaft rotation or choose to average the signal in the time domain or the frequency domain.

`[ta,t,p,rpm] = tsa( ___ )` also returns `t`, a vector of sample times corresponding to `ta`; a vector `p` of phase values; and `rpm`, the constant rotational speed (in revolutions per minute) corresponding to `ta`.

`tsa( ___ )` with no output arguments plots the time-synchronous average signal and the time-domain signals corresponding to each signal segment in the current figure.

Examples

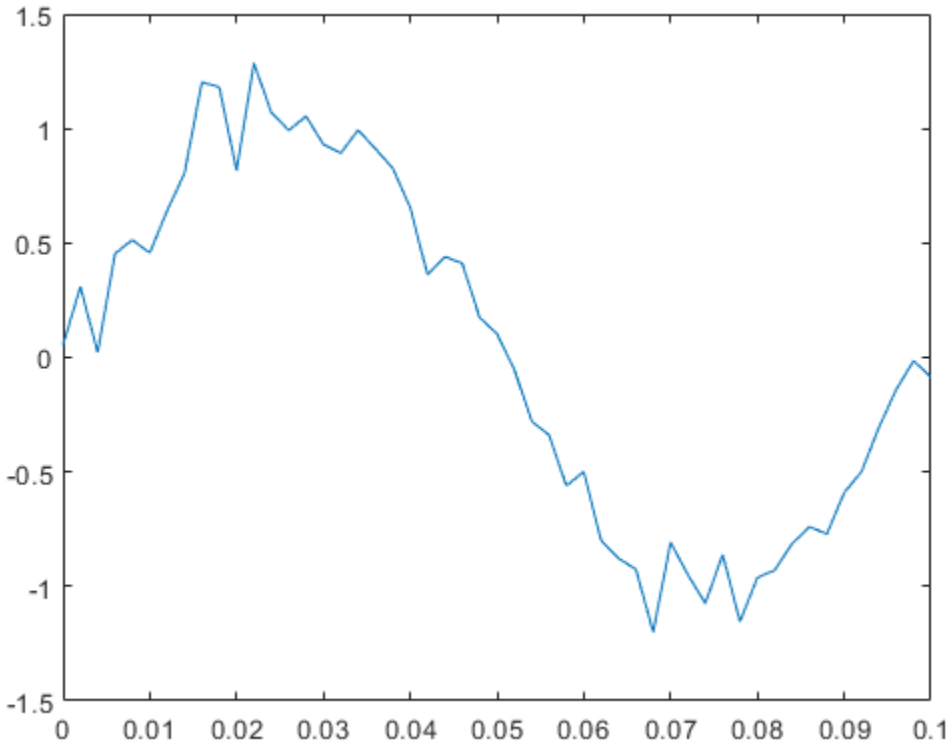
Time-Synchronous Average of Sinusoid

Compute the time-synchronous average of a noisy sinusoid.

Generate a signal consisting of a sinusoid embedded in white Gaussian noise. The signal is sampled at 500 Hz for 20 seconds. Specify a sinusoid frequency of 10 Hz and a noise variance of 0.01. Plot one period of the signal.

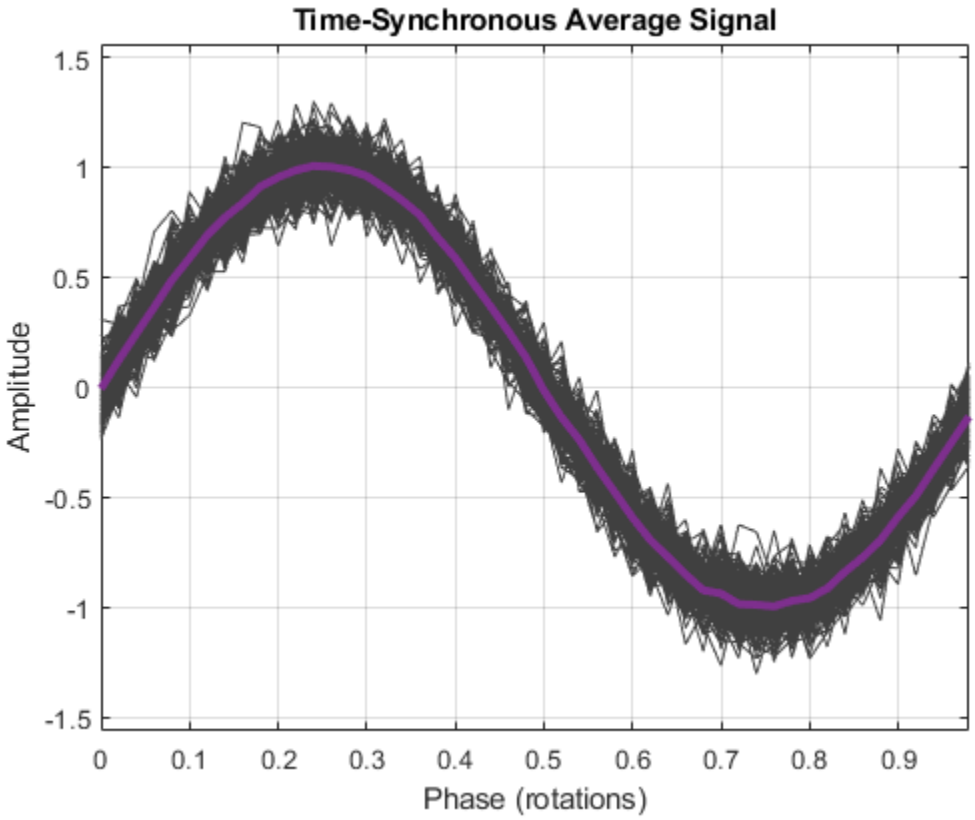
```
fs = 500;
t = 0:1/fs:20-1/fs;

f0 = 10;
```

```
y = sin(2*pi*f0*t) + randn(size(t))/10;

plot(t,y)
xlim([0 1/f0])
```



Compute the time-synchronous average of the signal. For the synchronizing signal, use a set of pulses with the same period as the sinusoid. Use `tsa` without output arguments to display the result.

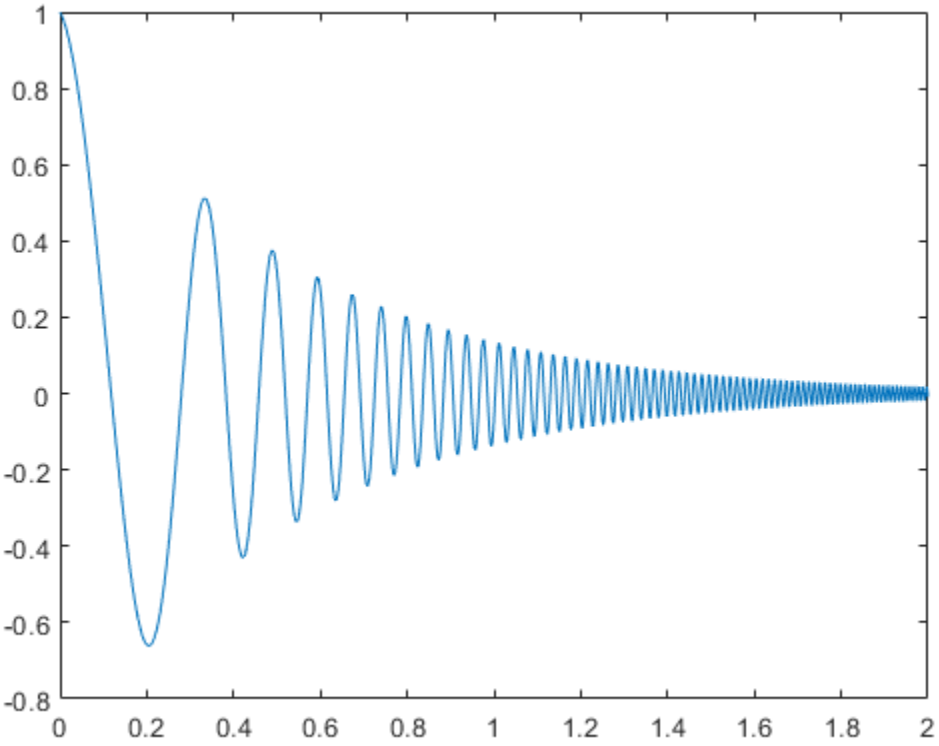
```
tPulse = 0:1/f0:max(t);

tsa(y,fs,tPulse)
```



Time-Synchronous Average of Timetable

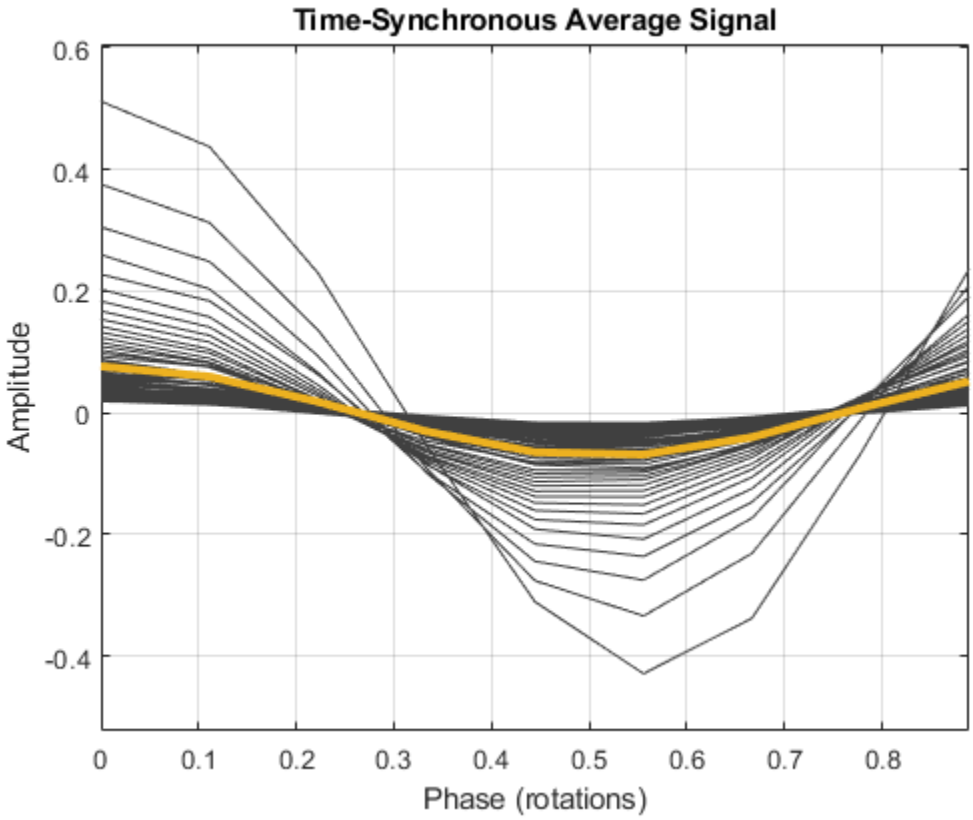
Generate a signal that consists of an exponentially damped quadratic chirp. The signal is sampled at 1 kHz for 2 seconds. The chirp has an initial frequency of 2 Hz that increases to 28 Hz after the first second. The damping has a characteristic time of 1/2 second. Plot the signal.

```
fs = 1e3;
t = 0:1/fs:2;

x = exp(-2*t').*chirp(t',2,1,28,'quadratic');

plot(t,x)
```



Create a duration array using the time vector. Construct a timetable with the duration array and the signal. Determine the pulse times using the locations of the signal peaks. Display the time-synchronous average.

```
ts = seconds(t)';
tx = timetable(ts,x);

[~,lc] = findpeaks(x,t);
tsa(tx,lc)
```



Compute the time-synchronous average. View the types of the output arguments. The sample times are stored in a duration array.

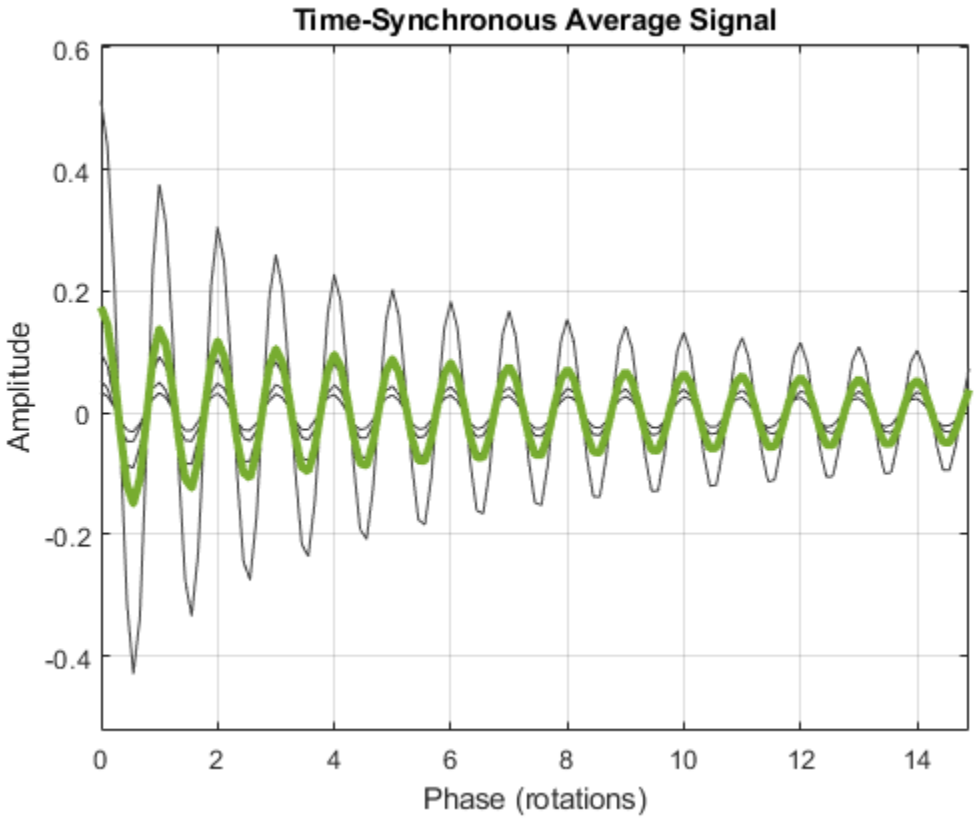
```
[xta,xt,xp,xrpm] = tsa(tx,lc);
whos x*
```

```
  Name          Size              Bytes  Class        Attributes

  x          2001x1              16008  double
  xp            9x1                1131  timetable
  xrpm          1x1                   8  double
  xt            9x1                  74  duration
  xta           9x1                1129  timetable
```

Convert the duration array to a `datetime` vector. Construct a timetable using the `datetime` vector and the signal. Compute the time-synchronous average, but now average over sets of 15 rotations.

View the types of the output arguments. The sample times are again stored in a duration array, even though the input timetable used a `datetime` vector.

```
dtb = datetime(datevec(ts));
dtt = timetable(dtb,x);

nr = 15;
tsa(dtt,lc,'NumRotations',nr)
```




```
[dta,dt,dp,drpm] = tsa(dtt,lc,'NumRotations',nr);
whos d*
```

```
  Name          Size              Bytes  Class        Attributes

  dp          135x1                3147  timetable
  drpm          1x1                   8  double
  dt          135x1                1082  duration
  dta         135x1                3145  timetable
  dtb        2001x1               32016  datetime
  dtt        2001x1               49001  timetable
```

Fan Switchoff

Compute the time-synchronous average of the position of a fan blade as it slows down after switchoff.

A desk fan spinning at 2400 rpm is turned off. Air resistance (with a negligible contribution from bearing friction) causes the fan rotor to stop in approximately 5 seconds. A high-speed camera measures the *x*-coordinate of one of the fan blades at a rate of 1 kHz.

```
fs = 1000;
t = 0:1/fs:5-1/fs;

rpm0 = 2400;
```

Idealize the fan blade as a point mass circling the rotor center at a radius of 10 cm. The blade experiences a drag force proportional to speed, resulting in the following expression for the phase angle:

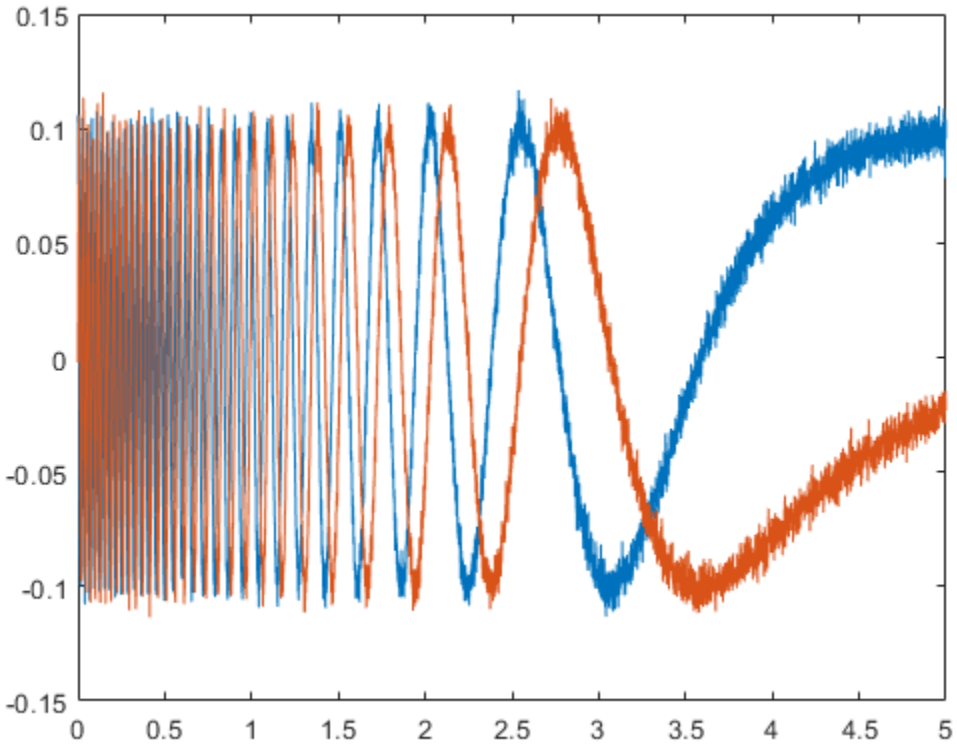
$$\phi = 2\pi f_0 T\left(1 - e^{-t/T}\right),$$

where f_0 is the initial frequency and $T = 0.75$ second is the decay time.

```
a = 0.1;
f0 = rpm0/60;
T = 0.75;

phi = 2*pi*f0*T*(1-exp(-t/T));
```

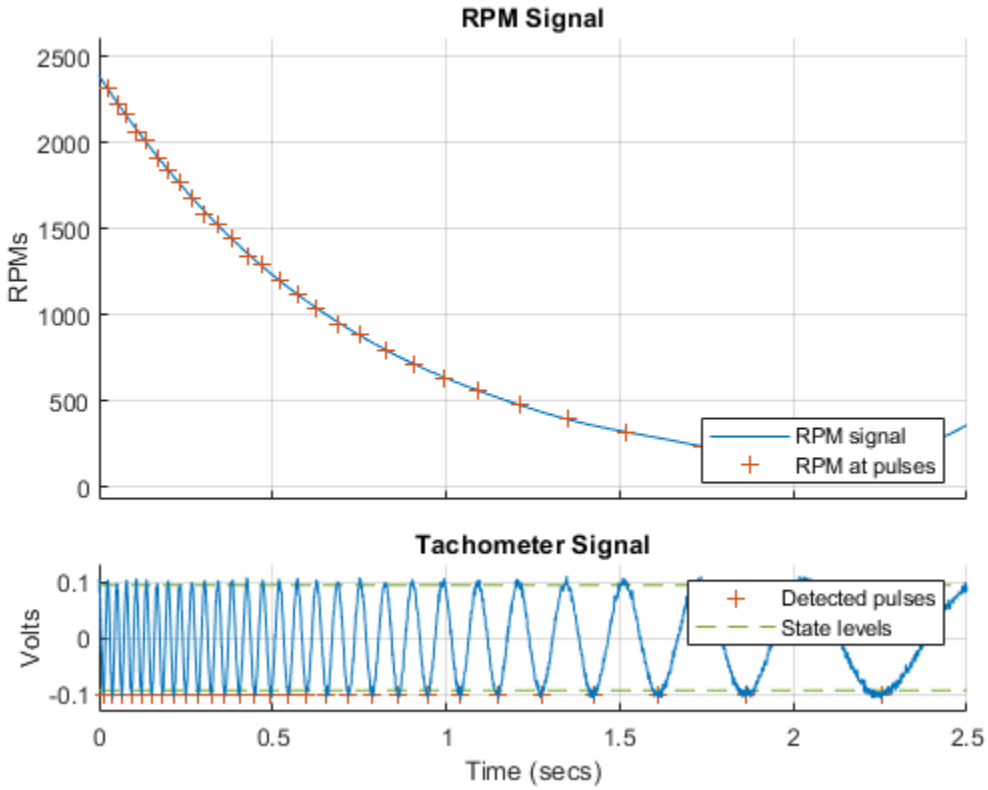
Compute and plot the *x*- and *y*-coordinates. Add white Gaussian noise.

```
x = a*cos(phi) + randn(size(phi))/200;
y = a*sin(phi) + randn(size(phi))/200;

plot(t,x,t,y)
```



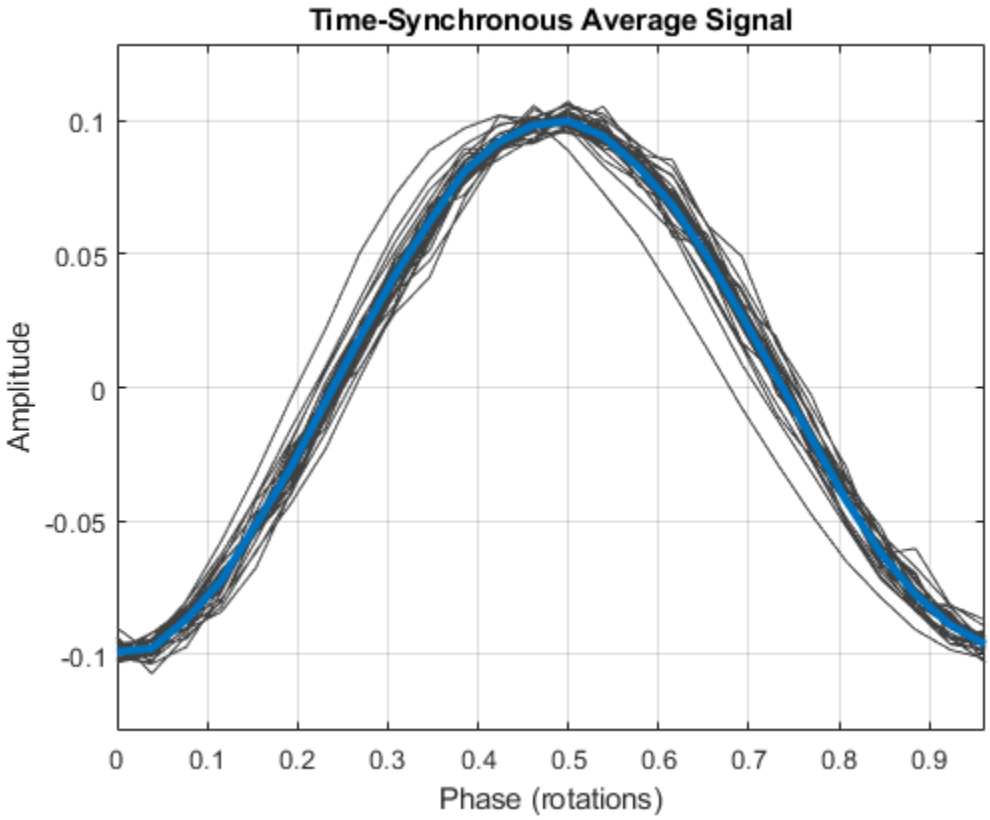
Determine the synchronizing signal. Use the `tachorpm` function to find the pulse times. Limit the search to times before 2.5 seconds. Plot the rotational speed to see its exponential decay.

```
[rpm,~,tp] = tachorpm(x(t<2.5),fs);
tachorpm(x(t<2.5),fs)
```



Compute and plot the time-synchronous average signal, which corresponds to a period of a sinusoid. Perform the averaging in the frequency domain.

```
clf
tsa(x,fs,tp,'Method','fft')
```



Input Arguments

x — Input signal

vector

Input signal, specified as a vector.

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

Data Types: `single` | `double`

fs — Sample rate

positive scalar

Sample rate, specified as a positive scalar.

Data Types: `single` | `double`

tp — Pulse times

scalar | vector

Pulse times, specified as a scalar or a vector.

- Scalar — a constant time interval over which rotations occur.
- Vector — nonnegative, strictly increasing instants that define constant rotational phase.

Use `tachorpm` to extract tachometer pulse times from a tachometer signal.

Data Types: `single` | `double`

`t` — Sample times
vector | `duration` scalar | `duration` array

Sample times, specified as a vector, a `duration` scalar, or a `duration` array.

- Scalar — the time interval between consecutive samples of `x`.
- Vector or `duration` array — the time instant corresponding to each element of `x`.

Data Types: `single` | `double` | `duration`

`xt` — Input timetable
timetable

Input timetable. `xt` must contain increasing finite row times and only one variable consisting of a vector.

If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

Example: `timetable(seconds(0:4)',randn(5,2))` specifies a two-channel, random variable sampled at 1 Hz for 4 seconds.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Method','pchip','ResampleFactor',10` specifies that the signal is to be upsampled by a factor of 10 and averaged in the time domain using piecewise-cubic Hermite interpolation.

`Method` — Averaging algorithm
`'linear'` (default) | `'spline'` | `'pchip'` | `'fft'`

Interpolation scheme, specified as the comma-separated pair consisting of `'InterpMethod'` and one of these values:

- `'linear'` — Perform linear interpolation and average in the time domain.
- `'spline'` — Perform cubic spline interpolation and average in the time domain.
- `'pchip'` — Perform piecewise-cubic Hermite interpolation and average in the time domain.
- `'fft'` — Perform frequency-domain averaging.

`NumRotations` — Number of shaft rotations
1 (default) | positive integer scalar

Number of shaft rotations in `ta`, specified as the comma-separated pair consisting of `'NumRotations'` and a positive integer scalar.

Data Types: `single` | `double`

`PulsesPerRotation` — Number of time instants per shaft rotation
1 (default) | positive scalar

Number of time instants per shaft rotation, specified as the comma-separated pair consisting of `'PulsesPerRotation'` and a positive scalar.

Data Types: `single` | `double`

`ResampleFactor` — Factor by which to increase the sample rate
1 (default) | positive integer scalar

Factor by which to increase the sample rate, specified as the comma-separated pair consisting of `'ResampleFactor'` and a positive integer scalar.

Data Types: `single` | `double`

Output Arguments

`ta` — Time-synchronous signal average
vector | timetable

Time-synchronous signal average, returned as a vector or timetable. If the input to `tsa` is a timetable, then `ta` is also a timetable.

- If the input timetable stores the time values as a `duration` array, then the time values of `ta` are also a `duration` array.
- If the input timetable stores the time values as a `datetime` array, then the time values of `ta` are a `duration` array expressed in seconds.

`t` — Sample times
vector | `duration` array

Sample times, returned as a vector or `duration` array.

- If the input to `tsa` is a timetable that stores time values as a `duration` array, then `t` has the same format as the input time values.
- If the input to `tsa` is a timetable that stores time values as a `datetime` array, then `t` is a `duration` vector expressed in seconds.
- If the input to `tsa` is a numeric vector and the input sample times `t` are stored in a `duration` scalar or a `duration` array, then `t` is a `duration` array with the same units as the input `t`.

`p` — Phase values
vector | timetable

Phase values, returned as a vector or timetable expressed in revolutions.

If the input to `tsa` is a timetable, then `p` is also a timetable. `p` has the same values as the time values of `ta`.

`rpm` — Constant rotational speed
scalar

Constant rotational speed, returned as a scalar expressed in revolutions per minute.

Algorithms

Given an input signal, a sample rate, and a set of tachometer pulses, `tsa` performs these steps:

1. Determines cycle start and end times based on the tachometer pulses and the value specified for `'PulsesPerRotation'`.
2. Resamples the input signal based on the value specified for `'ResampleFactor'`.
3. Averages the resampled signal based on the option specified for `'Method'`.
   - If `'Method'` is set to `'fft'`, the function:
     a. Breaks the signal into segments corresponding to the different cycles.
     b. Computes the discrete Fourier transform of each segment.
     c. Truncates the longer transforms so all transforms have the same length.
     d. Averages the spectra.
     e. Computes the inverse discrete Fourier transform of the average to convert it to the time domain.
   - If `'Method'` is set to one of the time-domain methods, the function:
     a. Using the specified method, interpolates the signal onto grids of equally spaced samples corresponding to the different cycles.
     b. Concatenates the resampled signal segments based on the value specified for `'NumRotations'`.
     c. Computes the average of all the segments.

References

[1] Bechhoefer, Eric, and Michael Kingsley. "A Review of Time-Synchronous Average Algorithms." *Proceedings of the Annual Conference of the Prognostics and Health Management Society*, San Diego, CA, September-October, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Timetables are not supported for code generation.

See Also

`tachorpm`

Topics

“Vibration Analysis of Rotating Machinery”

Introduced in R2017b

tukeywin

Tukey (tapered cosine) window

Syntax

```
w = tukeywin(L,r)
```

Description

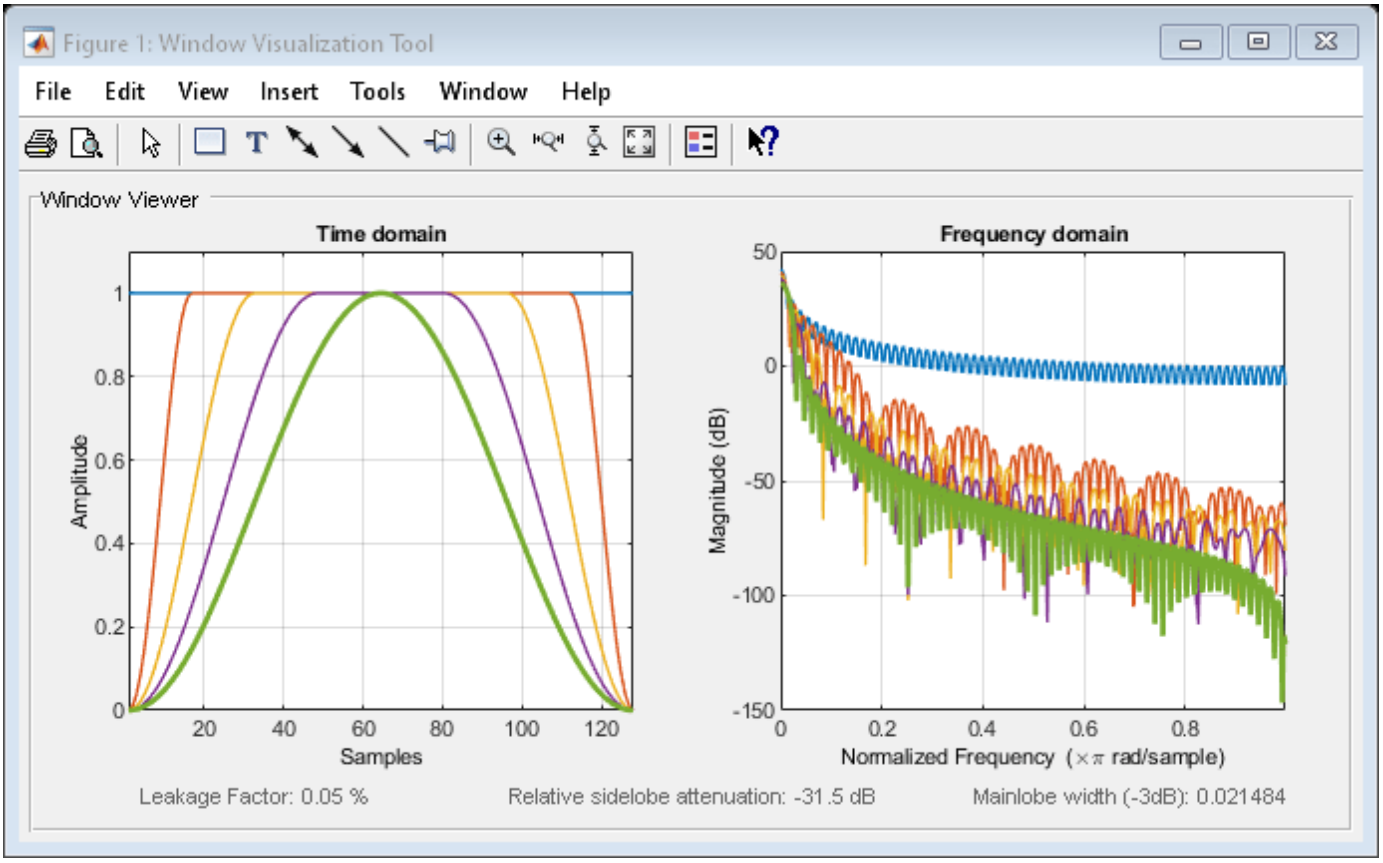
`w = tukeywin(L,r)` returns an `L`-point Tukey window with cosine fraction `r`.

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)
```



Input Arguments

`L` — Window length

positive integer

Window length, specified as a positive integer.

Data Types: `single` | `double`

`r` — Cosine fraction

`0.5` (default) | real scalar

Cosine fraction, specified as a real scalar. The Tukey window is a rectangular window with the first and last `r/2` percent of the samples equal to parts of a cosine. 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 ≤ 0`, an `L`-point rectangular window is returned. If you specify `r ≥ 1`, an `L`-point von Hann window is returned.

Data Types: `single` | `double`

Output Arguments

`w` — Tukey window

column vector

Tukey window, returned as a column vector.

Algorithms

The following equation defines the L-point Tukey window:

$$w(x) = \begin{cases} \frac{1}{2}\left\{1 + \cos(\frac{2\pi}{r}[x - r/2])\right\}, & 0 \le x < \frac{r}{2} \\ 1, & \frac{r}{2} \le x < 1 - \frac{r}{2} \\ \frac{1}{2}\left\{1 + \cos(\frac{2\pi}{r}[x - 1 + r/2])\right\}, & 1 - \frac{r}{2} \le x \le 1 \end{cases}$$

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 $0 \le r \le 1$. 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 $r \le 0$, an L-point rectangular window is returned. If you specify $r \ge 1$, an L-point von Hann window is returned.

References

[1] Bloomfield, P. *Fourier Analysis of Time Series: An Introduction*. New York: Wiley-Interscience, 2000.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Apps

Window Designer

Functions

`chebwin` | `gausswin` | `kaiser` | **WVTool**

Introduced before R2006a

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 `'saturatemode'`, which can be set to one of the following:

- `'saturate'` — Saturate overflows. This is the default method for treating overflows.
  - Entries in signed inputs `u` whose values are outside of the range $[-2^n/2, (2^n/2) - 1]$ are assigned the value determined by the closest endpoint of this interval.
  - Entries in unsigned inputs `u` whose values are outside of the range $[0, 2^n-1]$ are assigned the value determined by the closest endpoint of this interval.
- `'wrap'` — Wrap all overflows according to the following:
  - Entries in signed inputs `u` whose values are outside of the range $[-2^n/2, (2^n/2) - 1]$ are wrapped back into that range using modulo 2^n arithmetic (calculated using `u = mod(u+2^n/2,2^n)-(2^n/2)`).
  - Entries in unsigned inputs `u` whose values are outside of the range $[0, 2^n - 1]$ are wrapped back into the required range before decoding using modulo 2^n arithmetic (calculated using `u = mod(u,2^n)`).

Examples

Use udecode to Decode Integers

Create a vector of 8-bit signed integers. Decode with three bits.

```
u = int8([-1 1 2 -5]);
ysat = udecode(u,3)
```

```
ysat = 1×4

   -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 = 1×4

   -1.5000    1.5000    3.0000   -6.0000
```

The last input entry still saturates. Wrap the overflows.

```
ywrap = udecode(u,3,6,'wrap')
```

```
ywrap = 1×4

   -1.5000    1.5000    3.0000    4.5000
```

Add more quantization levels.

```
yprec = udecode(u,5)
```

```
yprec = 1×4

   -0.0625    0.0625    0.1250   -0.3125
```

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

Introduced before R2006a

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. 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 $[0, 2^n-1]$.

`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).

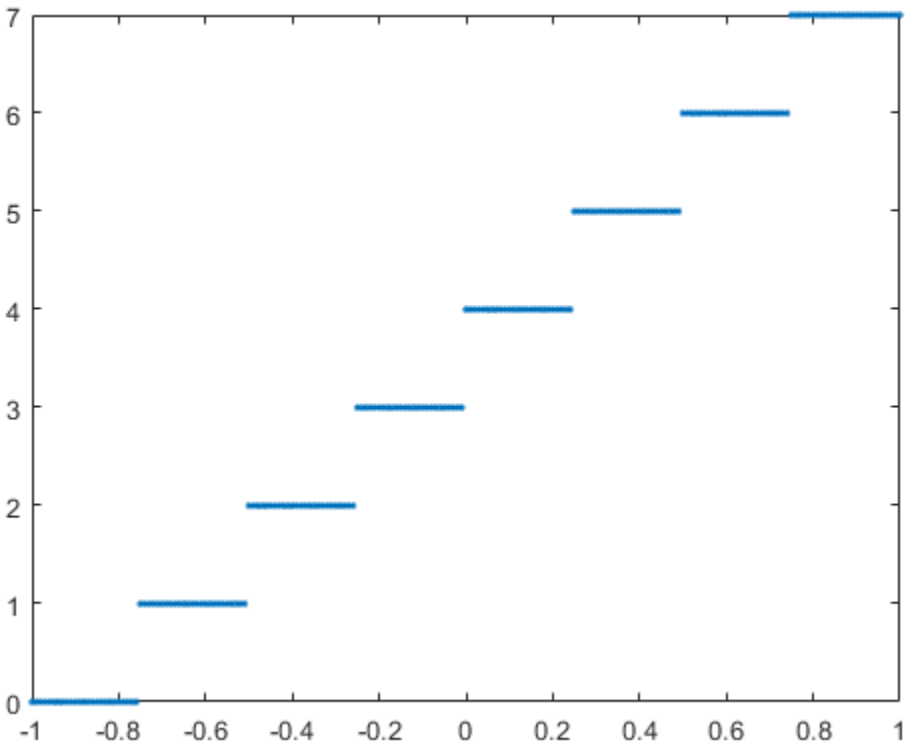
`y = uencode(u,n,v,'SignFlag')` maps entries in a multidimensional array of floating-point numbers `u` whose entries have values in the range `[-v,v]` to an integer output `y`. Input entries outside this range are saturated.

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 = 1x9 uint8 row vector

    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 = 1x9 int8 row vector

   -16   -12    -8    -4     0     4     8    12    15
```

Input Arguments

u — Floating point input

matrix | vector

Floating point input, specified as a matrix or a vector. The input may be real or complex. Elements of the input `u` outside of the range `[-1,1]` are treated as overflows and are saturated as:

- 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.

Data Types: `single` | `double`

`n` — Measure of number of quantization levels
positive integer scalar

Measure of number of quantization levels, specified as a positive integer scalar. `n` must be an integer between 2 and 32 (inclusive).

`v` — Peak value
1 (default) | positive real scalar

Peak value, specified as a positive real scalar. Elements of `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.

`'SignFlag'` — Sign of output
`'signed'` | `'unsigned'`

Sign of output, specified as `'signed'` or `'unsigned'`. The integer type of the output depends on the number of quantization levels 2^n and the value of `'SignFlag'`, which 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].

Output Arguments

`y` — Encoded integer outputs
vector | matrix

Encoded integer outputs, returned as a vector or a matrix.

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.

The output data types are optimized for the number of bits as shown in the table below.

n	Unsigned Integer	Signed Integer
2 to 8	`uint8`	`int8`
9 to 16	`uint16`	`int16`
17 to 32	`uint32`	`int32`

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

Introduced before R2006a

unshiftdata

Inverse of shiftdata

Syntax

```
y = unshiftdata(x,perm,nshifts)
```

Description

`y = unshiftdata(x,perm,nshifts)` restores the orientation of data `x` that was shifted using the `shiftdata` function with permutation `perm`.

Note Use the `unshiftdata` function in tandem with the `shiftdata` function. 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 = 3×3

     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 = 3×3

     8     3     4
     1     5     9
     6     7     2


perm = 1×2

     2     1
```

```
nshifts =

     []
```

Shift the matrix back to its original shape.

```
y = unshiftdata(x,perm,nshifts)
```

```
y = 3×3

     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 = 1×5

     1     2     3     4     5
```

Define `dim` as empty to shift the first nonsingleton 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 = 5×1

     1
     2
     3
     4
     5


perm =

     []

nshifts = 1
```

Using `unshiftdata`, restore `x` to its original shape.

```
y = unshiftdata(x,perm,nshifts)
```

```
y = 1×5

     1     2     3     4     5
```

Input Arguments

x — Data

vector | matrix

Data, specified as a vector or matrix.

Data Types: `single` | `double`

perm — Permutation

vector

Permutation, specified as a vector.

Data Types: `single` | `double`

nshifts — Number of shifts

scalar

Number of shifts, specified as a scalar. `nshift` is returned by the `shiftdata` function.

Data Types: `single` | `double`

Output Arguments

y — Restored data

vector | matrix

Restored data, returned as a vector or matrix.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`ipermute` | `shiftdata` | `shiftdim`

Introduced in R2012b

upfirdn

Upsample, apply FIR filter, and downsample

Syntax

```
yout = upfirdn(xin,h)
yout = upfirdn(xin,h,p)
yout = upfirdn(xin,h,p,q)
```

Description

`yout = upfirdn(xin,h)` filters the input signal `xin` using an FIR filter with impulse response `h`. No upsampling or downsampling is implemented with this syntax.

`yout = upfirdn(xin,h,p)` specifies the integer upsampling factor `p`.

`yout = upfirdn(xin,h,p,q)` specifies the integer downsampling factor `q`.

Examples

Convert from DAT Rate to CD Sample Rate

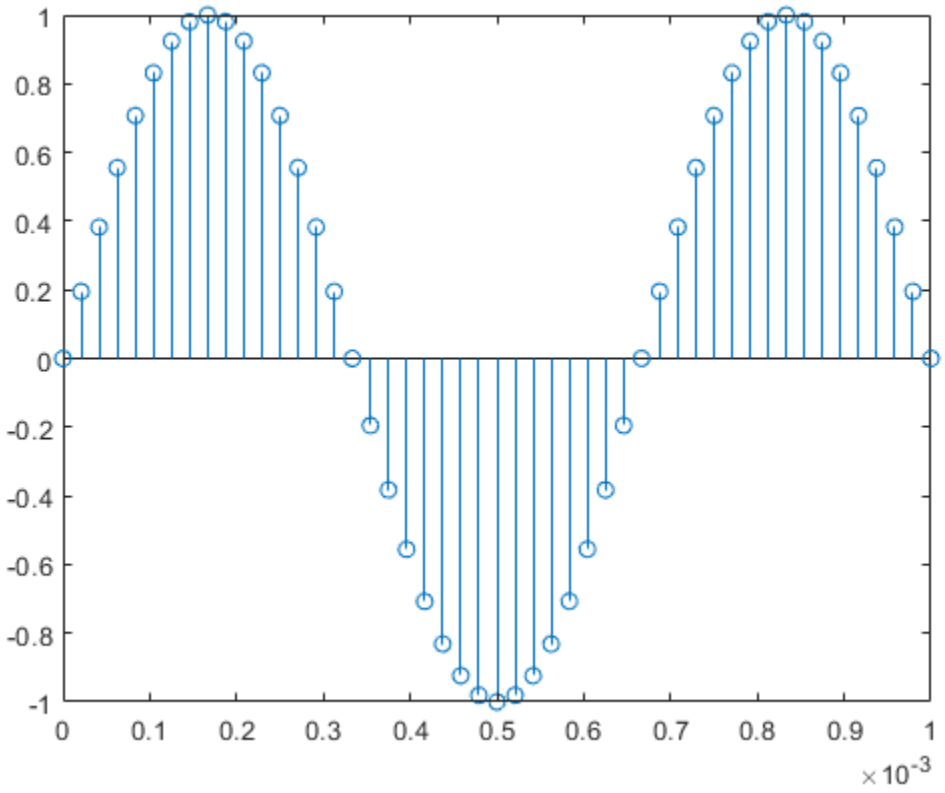
Change the sample rate of a signal by a rational conversion factor from the DAT rate of 48 kHz to the CD sample rate of 44.1 kHz. Use the `rat` function to find the numerator `L` and the denominator `M` of the rational factor.

```
Fdat = 48e3;
Fcd = 44.1e3;
[L,M] = rat(Fcd/Fdat)
```

```
L = 147

M = 160
```

Generate a 1.5 kHz sinusoid sampled at f_{DAT} for 0.25 seconds. Plot the first millisecond of the signal.

```
t = 0:1/Fdat:0.25-1/Fdat;
x = sin(2*pi*1.5e3*t);
stem(t,x)
xlim([0 0.001])
hold on
```



Design an antialiasing lowpass filter using a Kaiser window. Set the filter band edges as 90% and 110% of the cutoff frequency, $(f_{DAT}/2) \times \min(1/L, 1/M)$. Specify a passband ripple of 5 dB and a stopband attenuation of 40 dB. Set the passband gain to `L`.

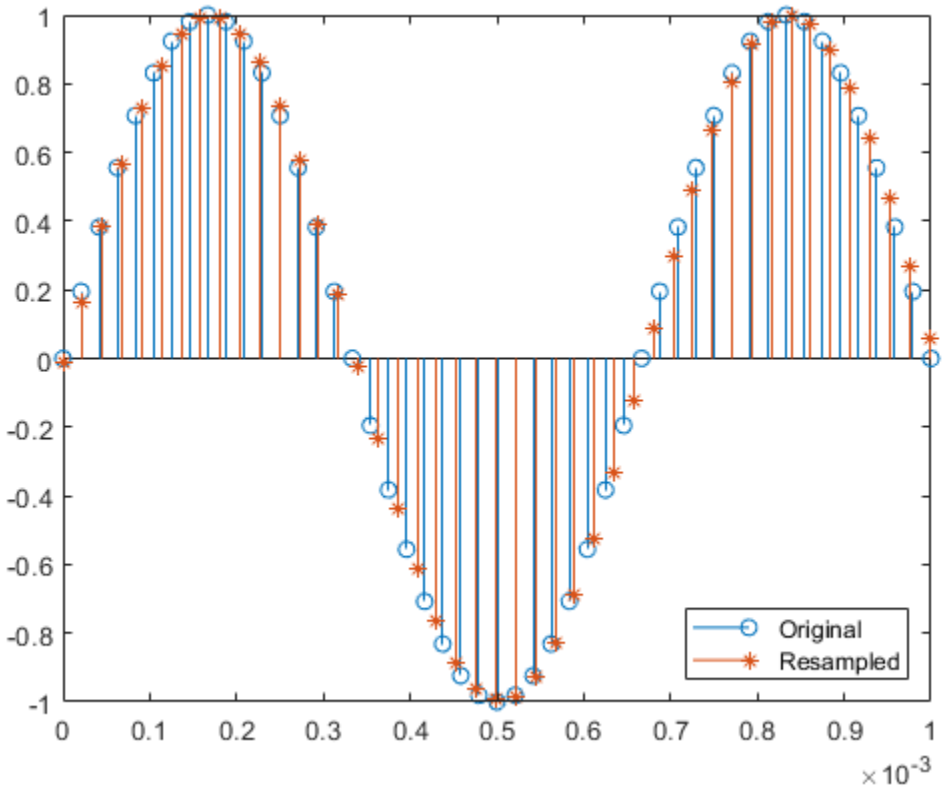
```
f = (Fdat/2)*min(1/L,1/M);
d = designfilt('lowpassfir', ...
    'PassbandFrequency',0.9*f,'StopbandFrequency',1.1*f, ...
    'PassbandRipple',5,'StopbandAttenuation',40, ...
    'DesignMethod','kaiserwin','SampleRate',48e3);
h = L*tf(d);
```

Use `upfirdn` with the filter `h` to resample the sinusoid. Compute and compensate for the delay introduced by the filter. Generate the corresponding resampled time vector.

```
y = upfirdn(x,h,L,M);

delay = floor(((filtord(d)-1)/2-(L-1))/L);
y = y(delay+1:end);
t_res = (0:(length(y)-1))/Fcd;
```

Overlay the resampled signal on the plot.

```
stem(t_res,y,'*')
legend('Original','Resampled','Location','southeast')
hold off
```



Input Arguments

`xin` — Input signal

vector | matrix

Input signal, specified as a vector or matrix. If `xin` is a vector, then it represents a single signal. If `xin` is a matrix,then each column is filtered independently. See “Tips” on page 1-2719 for more details.

`h` — Filter impulse response

vector | matrix

Filter impulse response, specified as a vector or matrix. If `h`is a vector, then it represents one FIR filter. If `h` is a matrix, then each column is a separate FIR impulse response sequence. See “Tips” on page 1-2719 for more details.

`p` — Upsampling factor

1 (default) | positive integer

Upsampling factor, specified as a positive integer.

`q` — Downsampling factor

1 (default) | positive integer

Downsampling factor, specified as a positive integer.

Output Arguments

yout — Output signal
vector | matrix

Output signal, returned as a vector or matrix. Each column of `yout` has length `ceil(((length(xin)-1)*p+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`.

Tips

The valid combinations of the sizes of `xin` and `h` are:

1. `xin` is a vector and `h` is a vector.

   The inputs are 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 vector; otherwise, `yout` is a column vector.

2. `xin` is a matrix and `h` is a vector.

   The inputs are one filter and many signals, so the function convolves `h` with each column of `xin`. The resulting `yout` is a matrix with the same number of columns as `xin`.

3. `xin` is a vector and `h` is a matrix.

   The inputs are multiple filters and one signal, so the function convolves each column of `h` with `xin`. The resulting `yout` is 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.

   The inputs are multiple filters and multiple 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 $(L_h L_x - pL_x)/q$ where L_h and L_x are the lengths of $h(n)$ and $x(n)$, respectively. For long signals, this formula is often exact.

`upfirdn` performs a cascade of three operations:

1. Upsample the input data in the matrix `xin` by a factor of the integer `p` (inserting zeros)
2. FIR filter the upsampled signal data with the impulse response sequence given in the vector or matrix `h`
3. Downsample the result by a factor of the integer `q` (throwing away samples)

The FIR filter is usually a lowpass filter, which you must design using another function such as `firpm` or `firl`.

Note The function `resample` performs an FIR design using `firls`, followed by rate changing implemented with `upfirdn`.

References

[1] 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.

[2] Crochiere, R. E., and Lawrence R. Rabiner. *Multirate Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1983.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`conv` | `decimate` | `downsample` | `filter` | `interp` | `intfilt` | `resample` | `upsample`

Introduced before R2006a

upsample

Increase sample rate by integer factor

Syntax

```
y = upsample(x,n)
y = upsample(x,n,phase)
```

Description

`y = upsample(x,n)` increases the sample rate of `x` by inserting `n` – 1 zeros between samples. If `x` is a matrix, the function treats each column as a separate sequence.

`y = upsample(x,n,phase)` specifies the number of samples by which to offset the upsampled sequence.

Examples

Increase Sample Rates

Increase the sample rate of a sequence by a factor of 3.

```
x = [1 2 3 4];
y = upsample(x,3)
```

```
y = 1×12

     1     0     0     2     0     0     3     0     0     4     0     0
```

Increase the sample rate of the sequence by a factor of 3 and add a phase offset of 2.

```
x = [1 2 3 4];
y = upsample(x,3,2)
```

```
y = 1×12

     0     0     1     0     0     2     0     0     3     0     0     4
```

Increase the sample rate of a matrix by a factor of 3.

```
x = [1 2;
     3 4;
     5 6];
y = upsample(x,3)
```

```
y = 9×2

     1     2
     0     0
```

```
     0     0
     3     4
     0     0
     0     0
     5     6
     0     0
     0     0
```

Input Arguments

x — Input array

vector | matrix

Input array, specified as a vector or matrix. If `x` is a matrix, the function treats the columns as independent channels.

Example: `cos(pi/4*(0:159)) + randn(1,160)` specifies a sinusoid embedded in white Gaussian noise.

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

n — Upsampling factor

positive integer

Upsampling factor, specified as a positive integer.

Data Types: `single` | `double`

phase — Offset

`0` (default) | positive integer

Offset, specified as a positive integer from 0 to `n` – 1.

Data Types: `single` | `double`

Output Arguments

y — Upsampled array

vector | matrix

Upsampled array, returned as a vector or matrix. `y` has `x` × `n` samples.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

decimate | downsample | interp | interp1 | resample | spline | upfirdn

Introduced before R2006a

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 undershoot expressed as a percentage of the difference between the state levels in the input bilevel waveform. The values in `us` correspond to the greatest deviations below the final state levels of each transition.

`us = undershoot(x,fs)` specifies the sample rate `fs` in hertz.

`us = undershoot(x,t)` specifies the sample instants `t`.

`[us,uslev,usinst] = undershoot( ___ )` returns the levels `uslev` and sample instants `usinst` of the undershoots for each transition. Specify an input combination from any of the previous syntaxes.

`[ ___ ] = undershoot( ___ ,Name,Value)` specifies additional options using one or more name-value arguments.

`undershoot( ___ )` plots the bilevel waveform and marks the location of the undershoot of each transition. The function also plots the lower and upper reference-level instants and associated reference levels and the state levels and associated lower- and upper-state boundaries.

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. Determine the maximum percent undershoot of the transition. Determine also the level and sample instant of the undershoot. In this example, the maximum undershoot in the posttransition region occurs near index 23.

```
load('transitionex.mat','x')

[uu,lv,nst] = undershoot(x)
```

```
uu = 4.5012
```

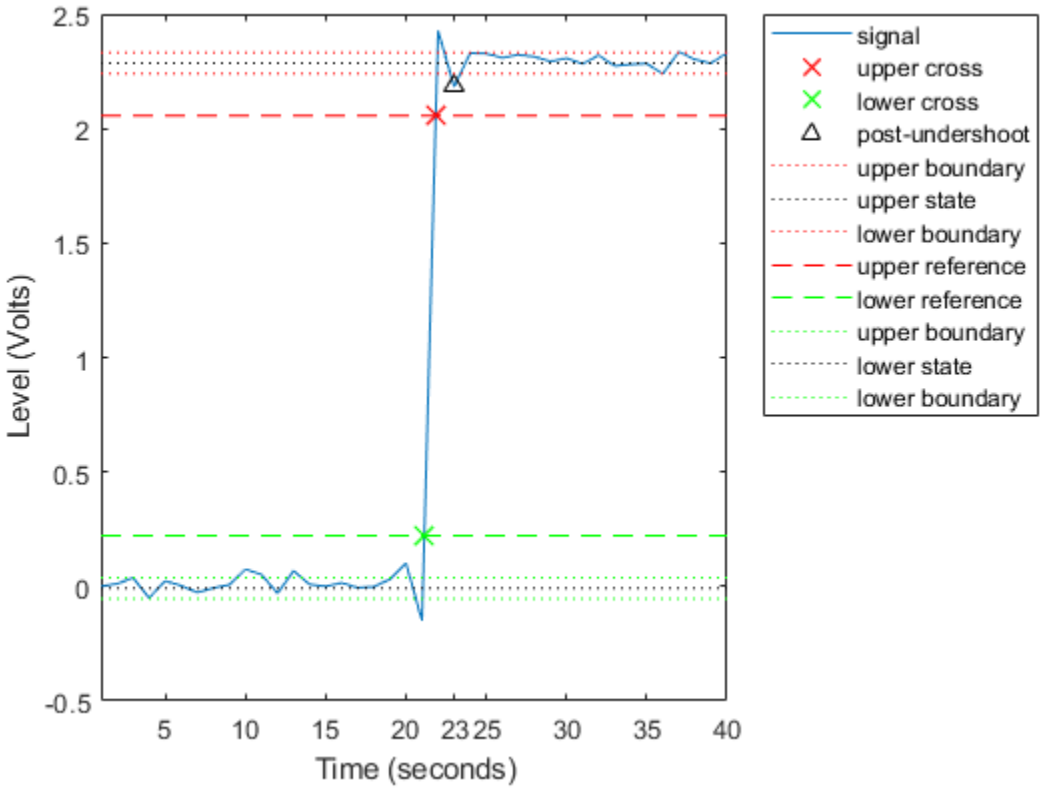
```
lv = 2.1826
```

```
nst = 23
```

Plot the waveform. Annotate the overshoot and the corresponding sample instant.

```
undershoot(x);
```

```
ax = gca;
ax.XTick = sort([ax.XTick nst]);
```



Undershoot Percentage, Levels, and Time 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. The clock data are sampled at 4 MHz.

```
load('transitionex.mat','x','t')
```

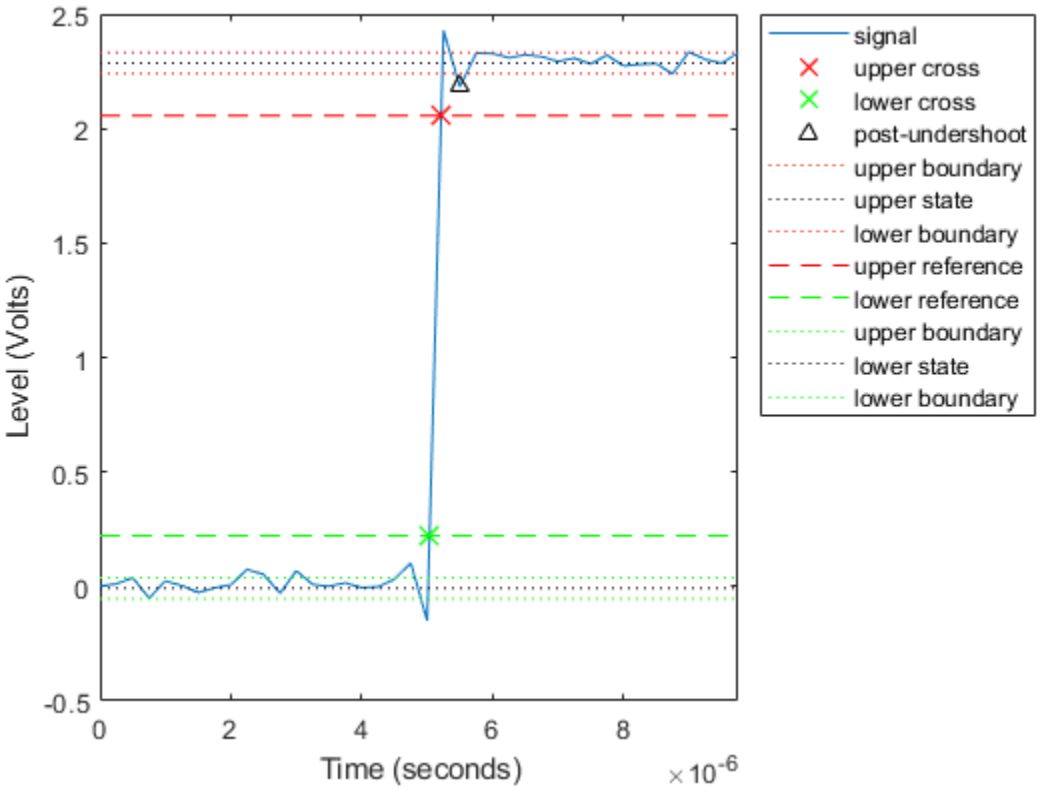
Determine the maximum percent undershoot, the level of the undershoot in volts, and the time instant where the maximum undershoot occurs. Plot the result.

```
[us,uslev,usinst] = undershoot(x,t)
```

```
us = 4.5012
```

```
uslev = 2.1826

usinst = 5.5000e-06
```

```
undershoot(x,t);
```



Undershoot Percentage, Levels, and Time 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 pretransition metrics.

Load the 2.3 V clock data with sampling instants. The clock data are sampled at 4 MHz.

```
load('transitionex.mat','x','t')
```

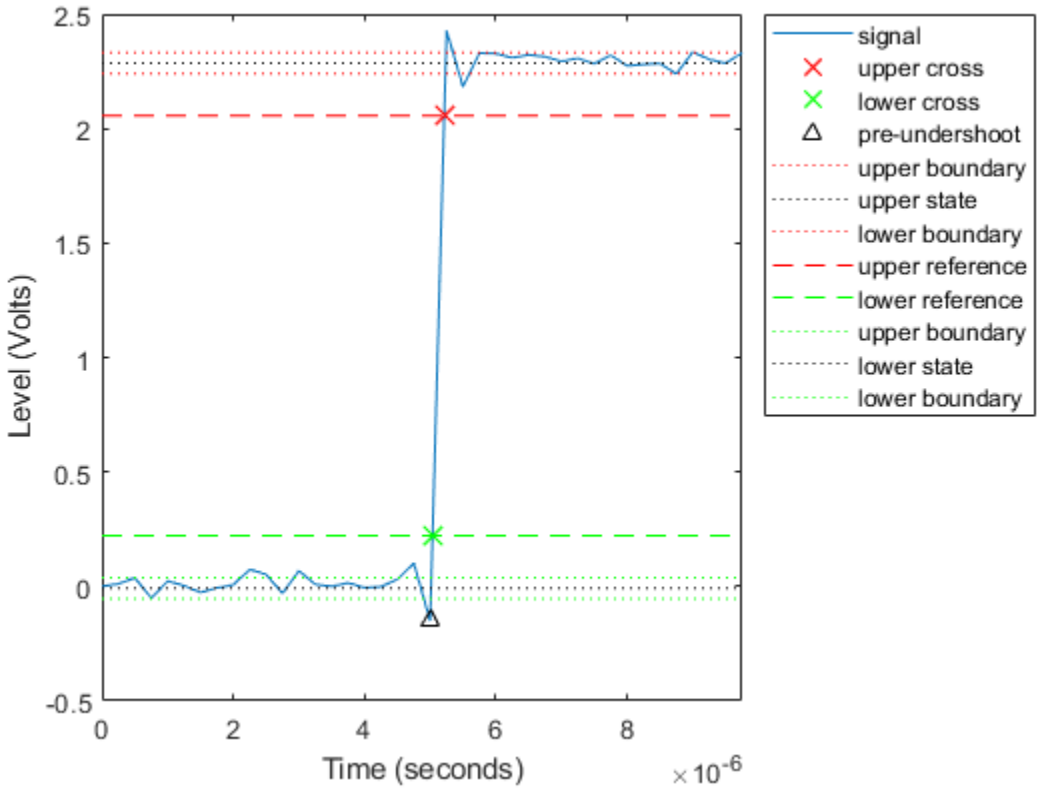
Determine the maximum percent undershoot, the level of the undershoot in volts, and the sampling instant where the maximum undershoot occurs. Plot the result.

```
[us,uslev,usinst] = undershoot(x,t,'Region','Preshoot')
```

```
us = 6.1798

uslev = -0.1500
```

```
usinst = 5.0000e-06
```

```
undershoot(x,t,'Region','Preshoot');
```



Input Arguments

x — Bilevel waveform

real-valued vector

Bilevel waveform, specified as a real-valued row or column vector. The sample instants in X correspond to the vector indices. The first sample instant in x corresponds to t = 0.

fs — Sample rate

real positive scalar

Sample rate in hertz, specified as a real positive scalar. The sample rate determines the sample instants corresponding to the elements in x.

t — Sample instants

vector

Sample instants, specified as a vector. The length of t must equal the length of the input bilevel waveform x.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Tolerance',5` computes the undershoot with a 5% tolerance region.

`PercentReferenceLevels` — Reference levels

`[10 90]` (default) | 1-by-2 real-valued vector

Reference levels as a percentage of the waveform amplitude, specified as a 1-by-2 real-valued vector. The function defines the lower-state level to be 0 percent and the upper-state level to be 100 percent. The first element corresponds to the lower percent reference level and the second element corresponds to the upper percent reference level.

`Region` — Aberration region

`'Postshoot'` (default) | `'Preshoot'`

Aberration region over which to compute the undershoot, specified as `'Preshoot'` or `'Postshoot'`. If you specify `'Preshoot'`, the function defines the end of the pretransition aberration region as the last instant when the signal exits the first state. If you specify `'Postshoot'`, the function defines the start of the posttransition aberration region as the instant when the signal enters the second state. By default, the function computes undershoots for posttransition aberration regions.

`SeekFactor` — Aberration region duration

3 (default) | real-valued scalar

Aberration region duration, specified as a real-valued scalar. The function computes the undershoot over the specified duration 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 aberration region duration elapses, the duration is truncated to the edge of the waveform or the start of the intervening transition.

`StateLevels` — Low- and high-state levels

1-by-2 real-valued vector

Low- and high-state levels, specified as a 1-by-2 real-valued vector. The first element corresponds to the low-state level and the second element corresponds to the high-state level of the input waveform.

`Tolerance` — Tolerance level

2 (default) | real-valued scalar

Tolerance level, specified as a real-valued scalar. The function expresses tolerance as a percentage of the difference between the upper- and lower-state levels. The initial and final levels of each transition must be within the respective state levels.

Output Arguments

`us` — Undershoots

vector

Undershoots expressed as a percentage of the state levels, returned as a vector. The length of `us` corresponds to the number of transitions detected in the input signal. For more information, see “Undershoot” on page 1-2729.

`uslev` — Undershoot level
column vector

Undershoot level, returned as a column vector.

`usinst` — Sample instants
column vector

Sample instants of pretransition or posttransition undershoots, returned as a column vector. If you specify `fs` or `t`, the undershoot instants are in seconds. If you do not specify`fs` or `t`, the undershoot instants are the indices of the input vector.

More About

State-Level Estimation

To determine the transitions, the `undershoot` function estimates the state levels of the input bilevel waveform `x` by a histogram method with these steps.

1. Determine the minimum and maximum amplitudes of the data.
2. For the specified number of histogram bins, determine the bin width, which is the ratio of the amplitude range to the number of bins.
3. Sort the data values into the histogram bins.
4. Identify the lowest and highest indexed histogram bins with nonzero counts.
5. Divide the histogram into two subhistograms.
6. Compute the state levels by determining the mode or mean of the upper and lower histograms.

The function 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.

Undershoot

The function computes the undershoot percentages based on the greatest deviation from the final state level in each transition.

For a positive-going (positive-polarity) pulse, the undershoot is given by

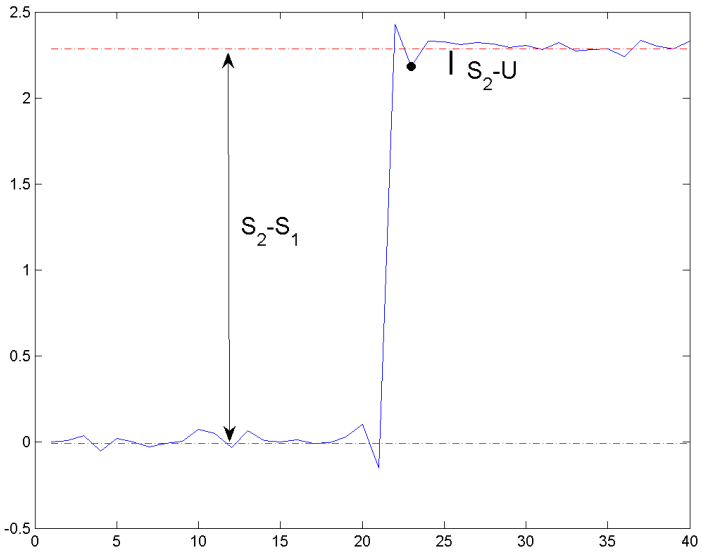
$$100\frac{(S_2 - U)}{(S_2 - S_1)}$$

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, the undershoot is given by

$$100\frac{(S_1 - U)}{(S_2 - S_1)}$$

This figure shows 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.

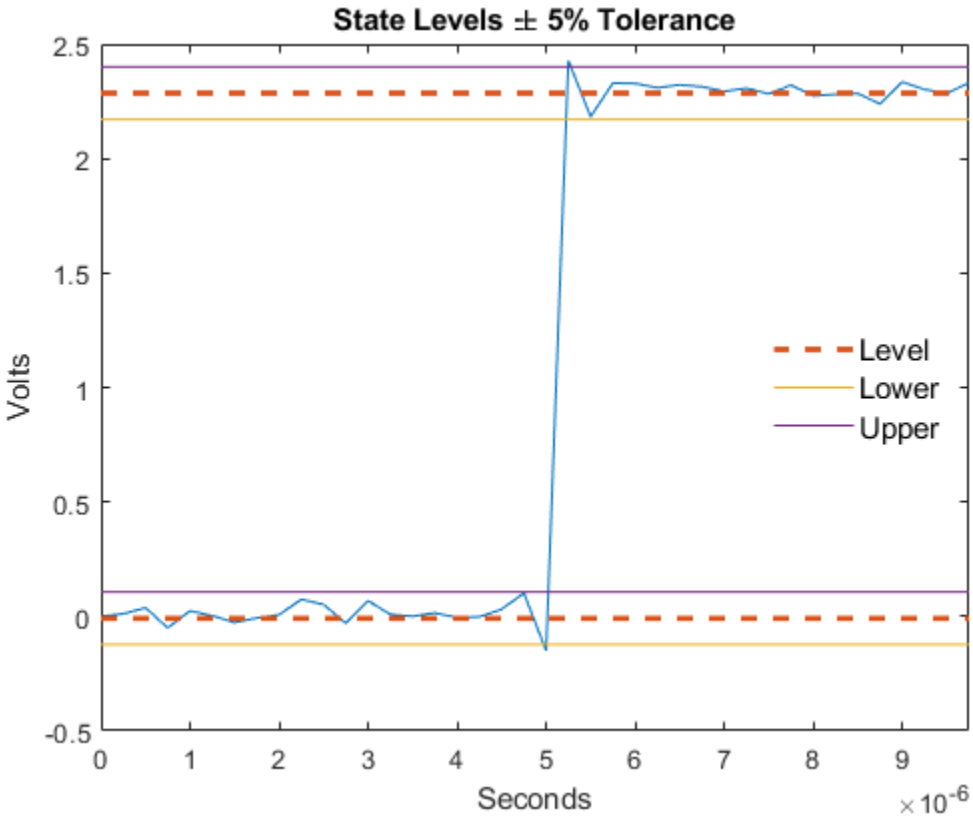
State-Level Tolerances

You can specify lower- and upper-state boundaries for each state level. Define the boundaries as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, specify the scalar as a small number such as 2/100 or 3/100. In general, the $\alpha\%$ 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.

This figure shows lower and upper 5% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The thick dashed lines indicate the estimated state levels.



References

[1] IEEE Standard 181. *IEEE Standard on Transitions, Pulses, and Related Waveforms* (2003): 15–17.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`overshoot` | `settlingtime` | `statelevels`

Introduced in R2012a

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 matrix `x` with sampling frequency `fs`. If `x` is a matrix, `vco` produces a matrix whose columns oscillate according to the columns of `x`.

`y = vco(x,[Fmin Fmax],fs)` scales the frequency modulation range so that ±1 values of `x` yield oscillations of `Fmin` Hz and `Fmax` Hz, respectively.

Examples

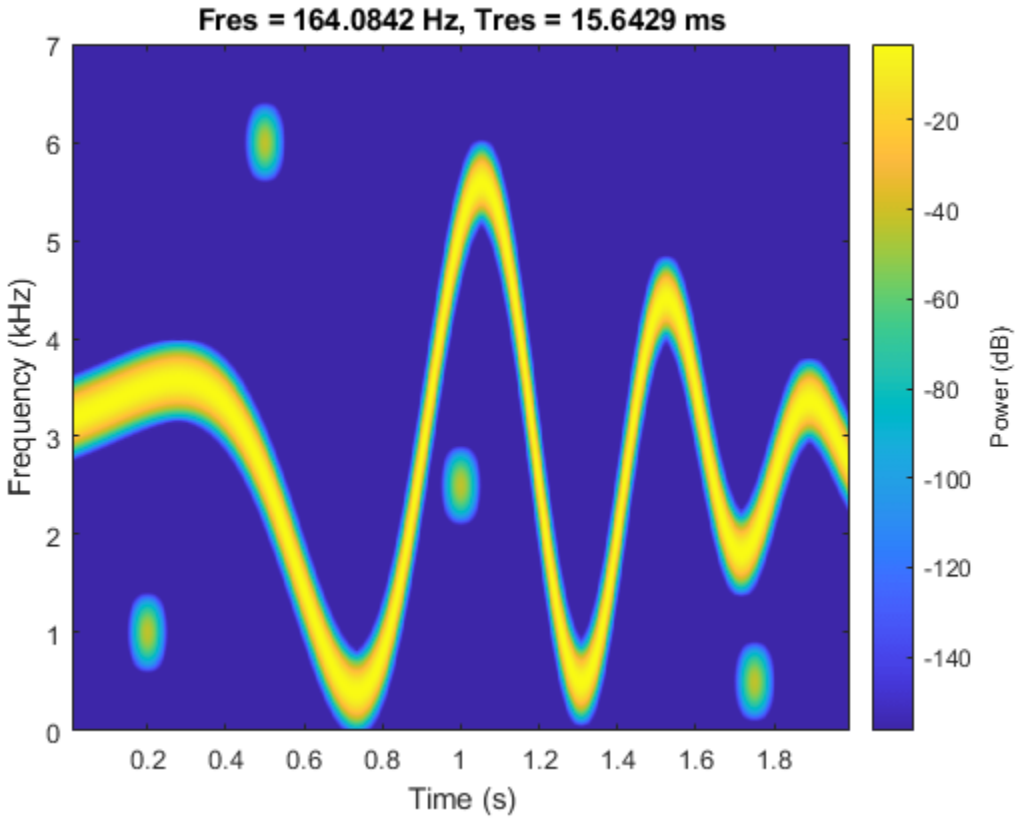
Spectrogram of Chirp Signal

Generate two seconds of a signal composed of a voltage-controlled oscillator (vco) and four Gaussian atoms. The instantaneous frequency is modulated by a chirp function. The sample rate is 14 kHz.

```
fs = 14000;
t = (0:1/fs:2)';

gaussFun = @(A,x,mu,f) exp(-(x-mu).^2/(2*0.01^2)).*sin(2*pi*f.*x)*A';
s = gaussFun([1 1 1 1],t,[0.2 0.5 1 1.75],[10 60 25 5]*100)/10;
x = vco(chirp(t+.1,0,t(end),3).*exp(-2*(t-1).^2),0.2*fs,fs);

s = s/10+x;
```

Plot the spectrogram of the generated signal. Specify 90% overlap and moderate spectral leakage.

```
pspectrum(s,fs,'spectrogram','OverlapPercent',90,'Leakage',0.5)
```



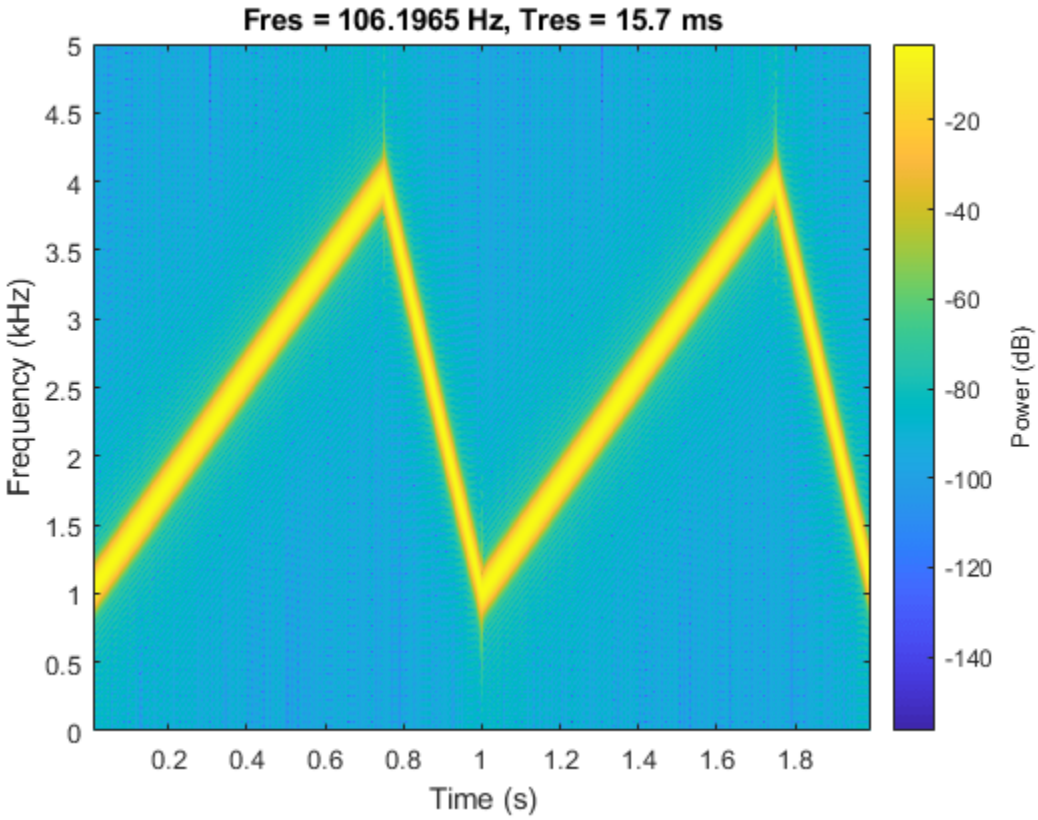
Spectrogram of Sawtooth Signal

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. Specify the leakage as `0.80` and the overlap between adjoining segments as 95%.

```
pspectrum(x,fs,'spectrogram','Leakage',0.80,'OverlapPercent',95)
```



Input Arguments

x — Input data

real vector | real matrix

Input data, specified as a real vector or real matrix. `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`.

fc — Carrier frequency

`fs`/4 (default) | real positive scalar

Carrier or reference frequency used to modulate the input signal, specified as a real positive scalar.

Fmin, Fmax — Frequency modulation range limits

real vector

Frequency modulation range limits, specified as a real vector. For best results, `Fmin` and `Fmax` should be in the range 0 to `fs/2`.

Note vco performs FM modulation using the `modulate` function.

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 has units of Hz.

Output Arguments

y — Output signal

real vector | real matrix

Oscillating output signal, returned as a real vector or real matrix. `y` is the same size as `x` and has amplitude equal to `1`.

See Also

`demod` | `modulate`

Introduced before R2006a

vmd

Variational mode decomposition

Syntax

```
imf = vmd(x)
[imf,residual] = vmd(x)
[imf,residual,info] = vmd(x)

[___] = vmd(x,Name,Value)

vmd(___)
```

Description

`imf = vmd(x)` returns the variational mode decomposition of `x`. Use `vmd` to decompose and simplify complicated signals into a finite number of intrinsic mode functions (IMFs) required to perform Hilbert spectral analysis.

`[imf,residual] = vmd(x)` also returns the residual signal `residual` corresponding to the variational mode decomposition of `x`.

`[imf,residual,info] = vmd(x)` returns additional information `info` on IMFs and the residual signal for diagnostic purposes.

`[___] = vmd(x,Name,Value)` performs the variational mode decomposition with additional options specified by one or more `Name,Value` pair arguments.

`vmd(___)` plots the original signal, IMFs, and the residual signal as subplots in the same figure.

Examples

Variational Mode Decomposition of Dial Tone Signal

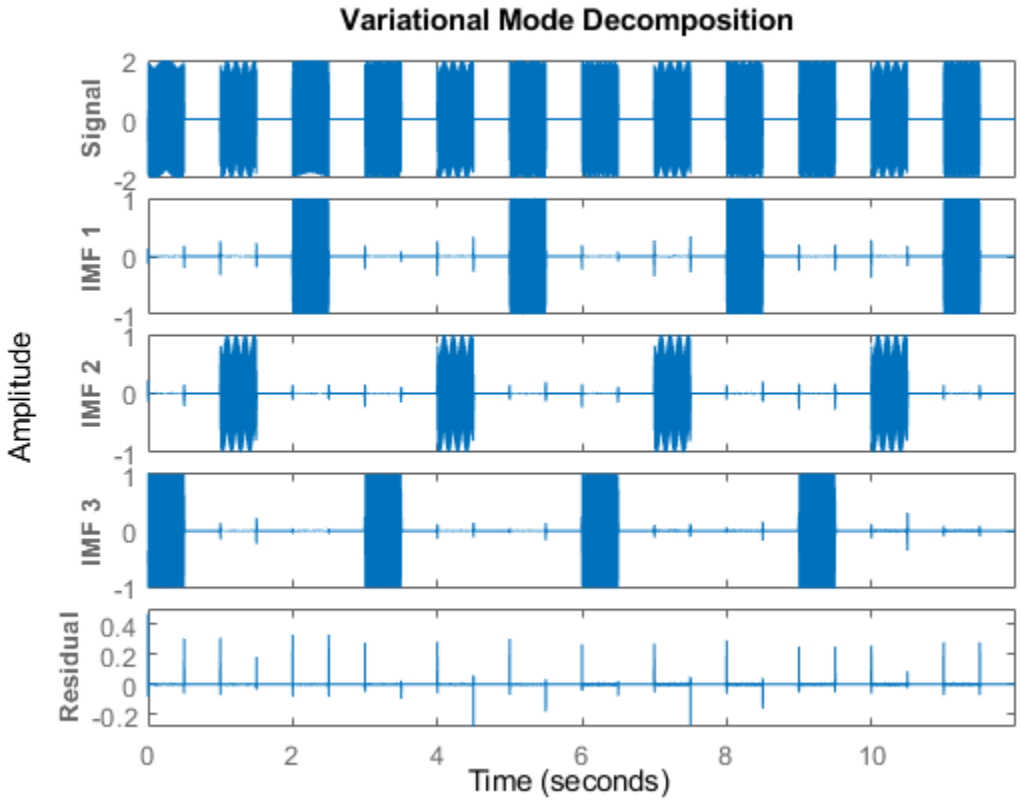
Create a signal, sampled at 4 kHz, that resembles dialing all the keys of a digital telephone. Save the signal as a MATLAB® timetable.

```
fs = 4e3;
t = 0:1/fs:0.5-1/fs;

ver = [697 770 852 941];
hor = [1209 1336 1477];

tones = [];

for k = 1:length(ver)
    for l = 1:length(hor)
        tone = sum(sin(2*pi*[ver(k);hor(l)].*t))';
        tones = [tones;tone;zeros(size(tone))];
    end
```

```
end

% To hear, type soundsc(tones,fs)

S = timetable(tones,'SampleRate',fs);
```

Plot the variational mode decomposition of the timetable.

```
vmd(S)
```



VMD of Multicomponent Signal

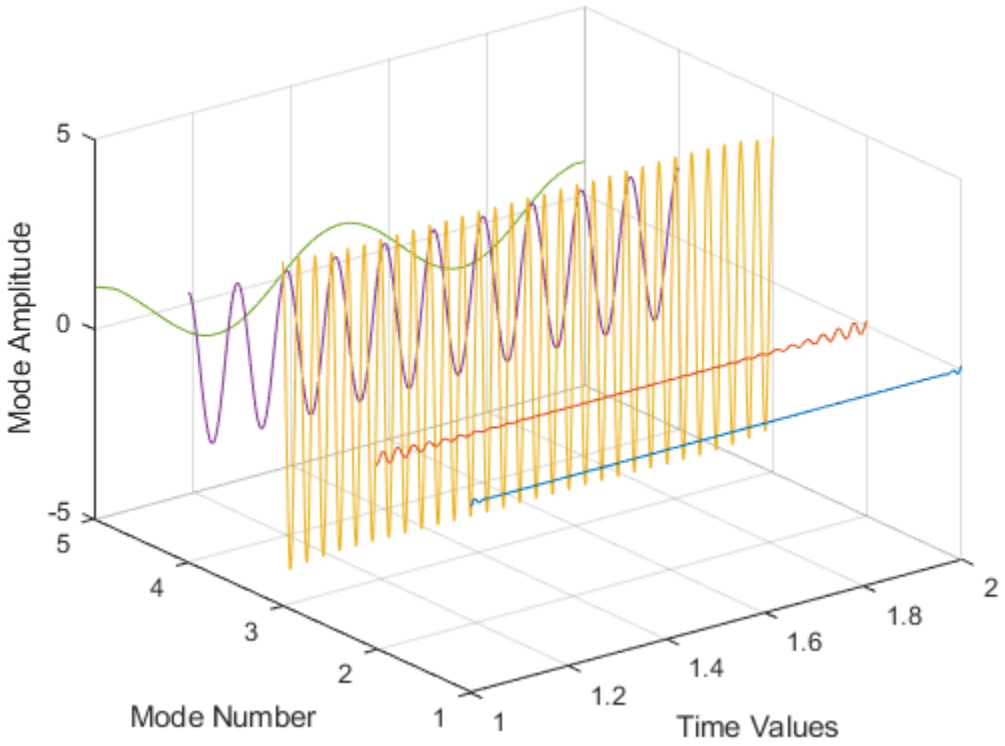
Generate a multicomponent signal consisting of three sinusoids of frequencies 2 Hz, 10 Hz, and 30 Hz. The sinusoids are sampled at 1 kHz for 2 seconds. Embed the signal in white Gaussian noise of variance 0.01^2.

```
fs = 1e3;
t = 1:1/fs:2-1/fs;
x = cos(2*pi*2*t) + 2*cos(2*pi*10*t) + 4*cos(2*pi*30*t) + 0.01*randn(1,length(t));
```

Compute the IMFs of the noisy signal and visualize them in a 3-D plot.

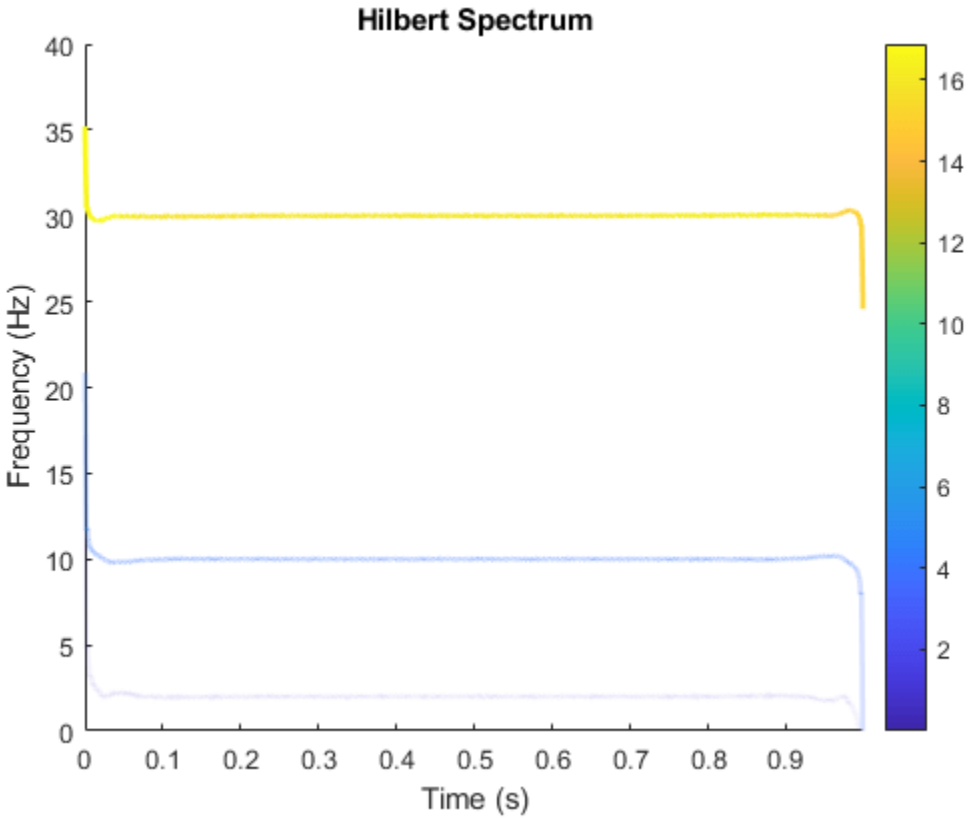
```
imf = vmd(x);
[p,q] = ndgrid(t,1:size(imf,2));
plot3(p,q,imf)
```

```
grid on
xlabel('Time Values')
ylabel('Mode Number')
zlabel('Mode Amplitude')
```



Use the computed IMFs to plot the Hilbert spectrum of the multicomponent signal. Restrict the frequency range to [0, 40] Hz.

```
hht(imf,fs,'FrequencyLimits',[0,40])
```



VMD of Piecewise Signal

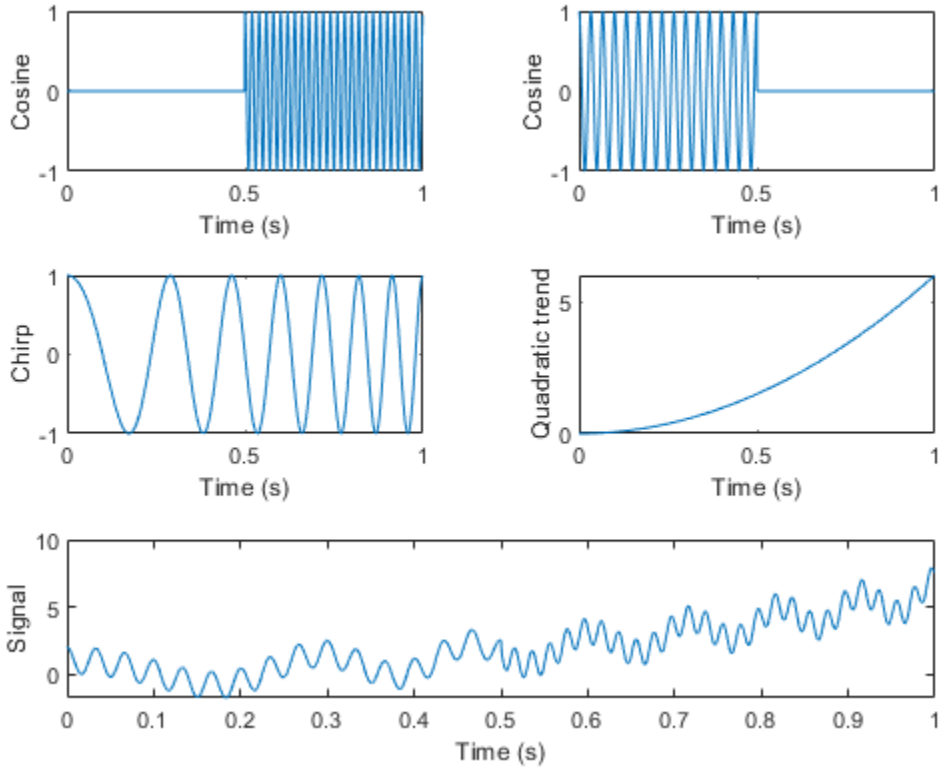
Generate a piecewise composite signal consisting of a quadratic trend, a chirp, and a cosine with a sharp transition between two constant frequencies at $t = 0.5$.

$$x(t) = 6t^2 + \cos\left(4\pi t + 10\pi t^2\right) + \begin{cases} \cos(60\pi t), & t \le 0.5, \\ \cos(100\pi t - 10\pi), & t > 0.5. \end{cases}$$

The signal is sampled at 1 kHz for 1 second. Plot each individual component and the composite signal.

```
fs = 1e3;
t = 0:1/fs:1-1/fs;

x = 6*t.^2 + cos(4*pi*t+10*pi*t.^2) + ...
    [cos(60*pi*(t(t<=0.5))) cos(100*pi*(t(t>0.5)-10*pi))];

tiledlayout('flow')
nexttile
plot(t,[zeros(1,length(t)/2) cos(100*pi*(t(length(t)/2+1:end))-10*pi)])
xlabel('Time (s)')
ylabel('Cosine')

nexttile
```

```
plot(t,[cos(60*pi*(t(1:length(t)/2))) zeros(1,length(t)/2)])
xlabel('Time (s)')
ylabel('Cosine')

nexttile
plot(t,cos(4*pi*t+10*pi*t.^2))
xlabel('Time (s)')
ylabel('Chirp')


nexttile
plot(t,6*t.^2)
xlabel('Time (s)')
ylabel('Quadratic trend')

nexttile(5,[1 2])
plot(t,x)
xlabel('Time (s)')
ylabel('Signal')
```



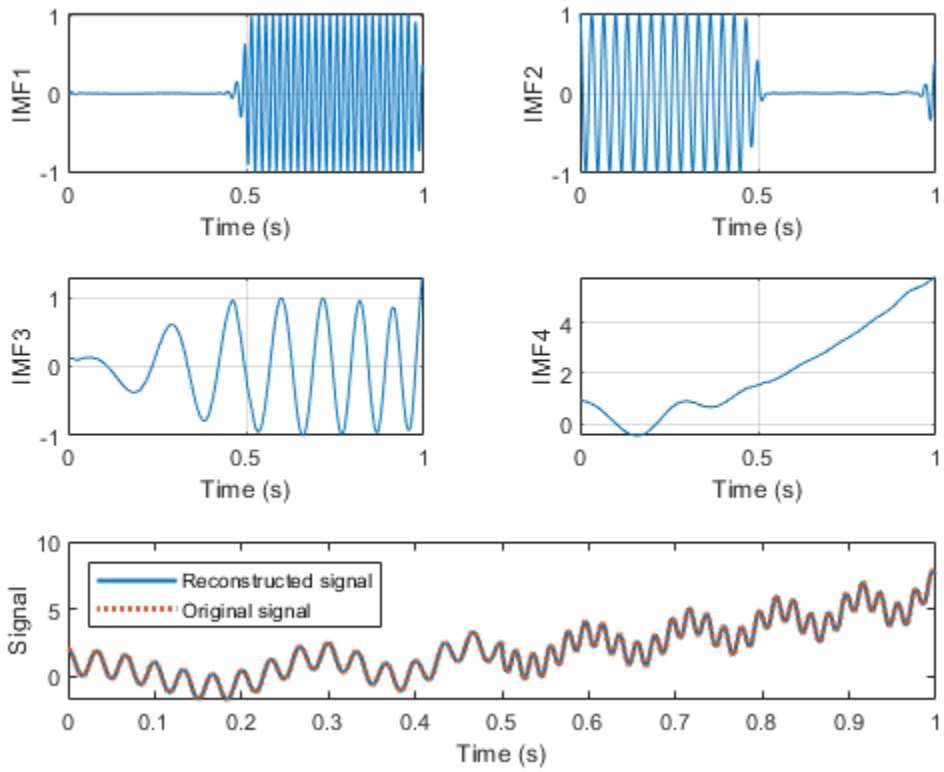
Perform variational mode decomposition to compute four intrinsic mode functions. The four distinct components of the signal are recovered.

```
[imf,res] = vmd(x,'NumIMFs',4);

tiledlayout('flow')

for i = 1:4
```

```
    nexttile
    plot(t,imf(:,i))
    txt = ['IMF',num2str(i)];
    ylabel(txt)
    xlabel('Time (s)')
    grid on
end
```

Reconstruct the signal by adding the mode functions and the residual. Plot and compare the original and reconstructed signals.

```
sig = sum(imf,2) + res;

nexttile(5,[1 2])
plot(t,sig,'LineWidth',1.5)

hold on

plot(t,x,':','LineWidth',2)
xlabel('Time (s)')
ylabel('Signal')
hold off
legend('Reconstructed signal','Original signal', ...
       'Location','northwest')
```



Calculate the norm of the difference between the original and the reconstructed signals.

```
norm(x-sig',Inf)
```

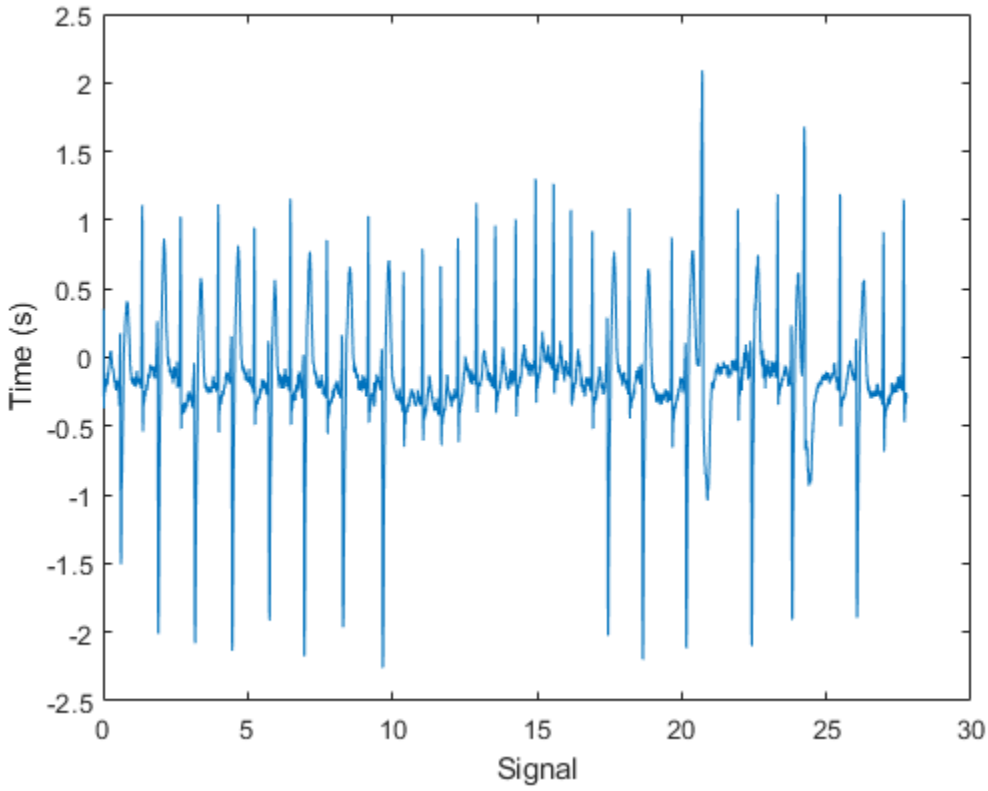
```
ans = 0
```

Noise Removal from ECG Signal Using VMD

The signals labeled in this example are from the MIT-BIH Arrhythmia Database [3]. The signal in the database was sampled at 360 Hz.

Load the MIT database signals corresponding to record 200 and plot the signal.

```
load mit200
Fs = 360;
plot(tm,ecgsig)
ylabel('Time (s)')
xlabel('Signal')
```

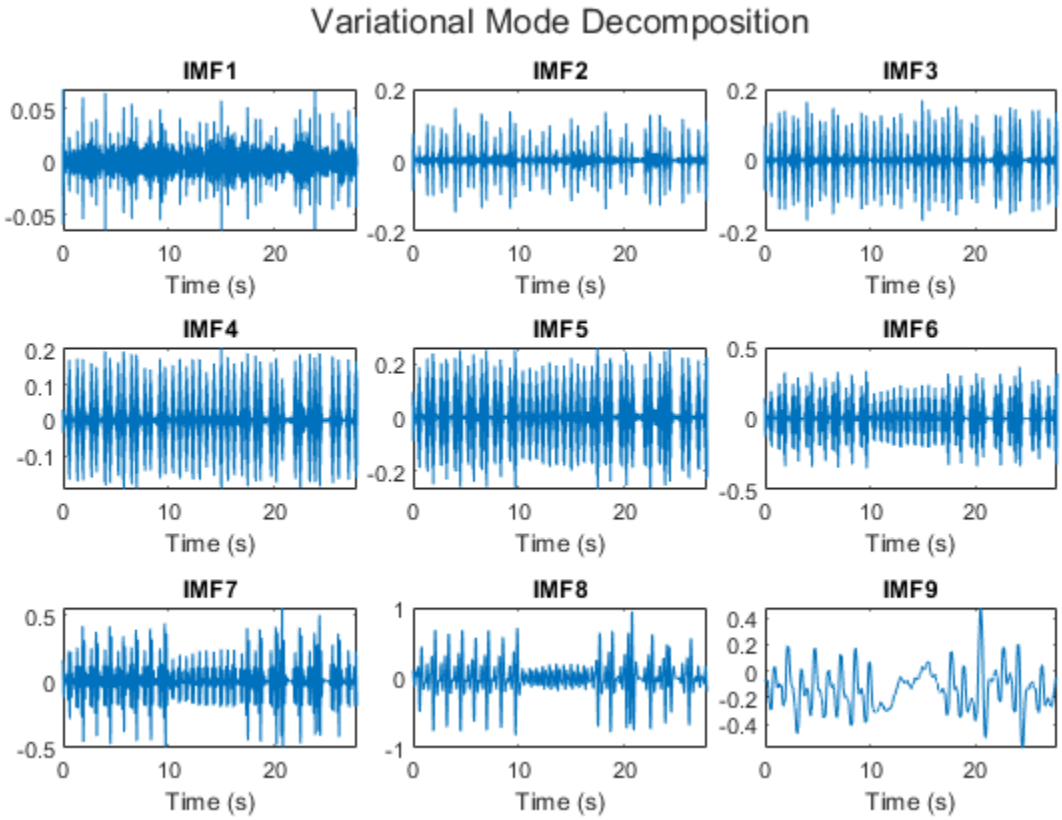


The ECG signal contains spikes driven by the rhythm of the heartbeat and an oscillating low frequency pattern. The distinct spokes of the ECG create important higher order harmonics.

Calculate nine intrinsic mode functions of the windowed signal. Visualize the IMFs.

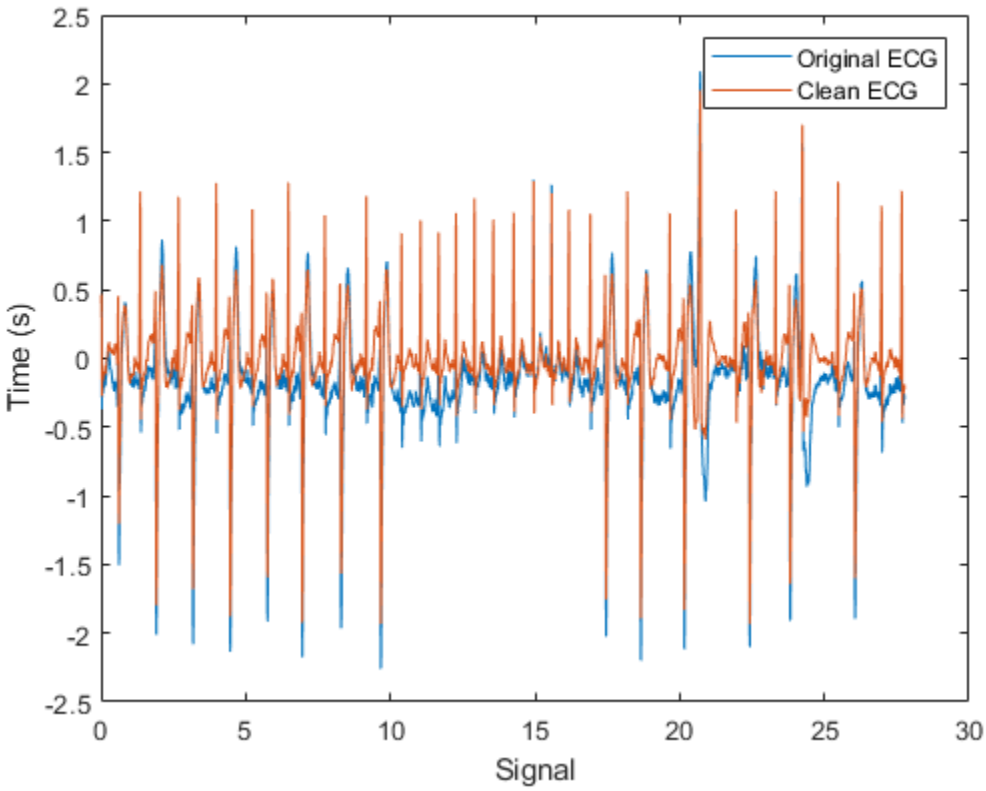
```
[imf,residual] = vmd(ecgsig,'NumIMF',9);
t = tiledlayout(3,3,'TileSpacing','compact','Padding','compact');
for n = 1:9
    ax(n) = nexttile(t);
    plot(tm,imf(:,n)')
```

```
    xlim([tm(1) tm(end)])
    txt = ['IMF',num2str(n)];
    title(txt)
    xlabel('Time (s)')
end
title(t,'Variational Mode Decomposition')
```



The first mode contains the most noise, and the second mode oscillates at the frequency of the heartbeat. Construct a clean ECG signal by summing all but the first and last VMD modes, thus discarding the low frequency baseline oscillation and most of the high frequency noise.

```
cleanECG = sum(imf(:,2:8),2);
figure
plot(tm,ecgsig,tm,cleanECG)
legend('Original ECG','Clean ECG')
ylabel('Time (s)')
xlabel('Signal')
```



Input Arguments

x — Uniformly sampled time-domain signal

vector | timetable

Uniformly sampled time-domain signal, specified as either a vector or a timetable. If `x` is a timetable, then it must contain increasing finite row times.. The timetable must contain only one numeric data vector with finite load values.

Note If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'NumIMF',10`

AbsoluteTolerance — Mode convergence absolute tolerance

`5e-6` (default) | positive real scalar

Mode convergence absolute tolerance, specified as the comma-separated pair consisting of 'AbsoluteTolerance' and a positive real scalar. AbsoluteTolerance is one of the stopping criteria for optimization, that is, optimization stops when the average squared absolute improvement toward convergence of IMFs, in two consecutive iterations, is less than AbsoluteTolerance.

RelativeTolerance — Mode convergence relative tolerance
AbsoluteTolerance*1e3 (default) | positive real scalar

Mode convergence relative tolerance, specified as the comma-separated pair consisting of 'RelativeTolerance' and a positive real scalar. RelativeTolerance is one of the stopping criteria for optimization, that is, optimization stops when the average relative improvement toward convergence of IMFs, in two consecutive iterations, is less than RelativeTolerance.

Note The optimization process stops when 'AbsoluteTolerance' and 'RelativeTolerance' are jointly satisfied.

MaxIterations — Maximum number of optimization iterations
500 (default) | positive scalar integer

Maximum number of optimization iterations, specified as the comma-separated pair consisting of 'MaxIterations' and a positive scalar integer. MaxIterations is one of the stopping criteria for optimization, that is, optimization stops when the number of iterations is greater than MaxIterations.

MaxIterations can be specified using only positive whole numbers.

NumIMF — Number of IMFs extracted
5 (default) | positive scalar integer

Number of IMFs extracted, specified as the comma-separated pair consisting of 'NumIMF' and a positive scalar integer.

CentralFrequencies — Initial central IMF frequencies
vector

Initial central IMF frequencies, specified as the comma-separated pair consisting of 'CentralFrequencies' and a vector of length NumIMFs. Vector values must be within [0, 0.5] cycles/sample, which indicates that the true frequencies are within [0, f_s/2], where f_s is the sample rate.

InitialIMFs — Initial IMFs
zero matrix (default) | real matrix

Initial IMFs, specified as the comma-separated pair consisting of 'InitialIMFs' and a real matrix. The rows correspond to time samples and columns correspond to modes.

PenaltyFactor — Penalty factor
1000 (default) | positive real scalar

Penalty factor, specified as the comma-separated pair consisting of 'PenaltyFactor' and a positive real scalar. This argument determines the reconstruction fidelity. Use smaller values of penalty factor to obtain stricter data fidelity.

`InitialLM` — Initial Lagrange multiplier
complex vector of zeros (default) | complex vector

Initial Lagrange multiplier, specified as the comma-separated pair consisting of '`InitialLM`' and a complex vector. The range of the initial Lagrange multiplier in the frequency domain is [0, 0.5] cycles/sample. The multiplier enforces the reconstruction constraint. The length of the multiplier depends on the input size.

`LMUpdateRate` — Update rate for Lagrange multiplier
`0.01` (default) | real scalar

Update rate for the Lagrange multiplier in each iteration, specified as the comma-separated pair consisting of '`LMUpdateRate`' and a positive real scalar. A higher rate results in faster convergence, but increases the chance of the optimization process getting stuck in a local optimum.

`InitializeMethod` — Method to initialize central frequencies
`'peaks'` (default) | `'random'` | `'grid'`

Method to initialize the central frequencies, specified as the comma-separated pair consisting of '`InitializeMethod`' and either `'peaks'`, `'random'`, or `'grid'`.

Specify `InitializeMethod` as:

- `'peaks'` to initialize the central frequencies as the peak locations of the signal in the frequency domain (default).
- `'random'` to initialize the central frequencies as random numbers distributed uniformly in the interval [`0,0.5`] cycles/sample.
- `'grid'` to initialize the central frequencies as a uniformly sampled grid in the interval [`0,0.5`] cycles/sample.

`Display` — Toggle information display in command window
`false` or 0 (default) | `true` or 1

Toggle progress display in the command window, specified as the comma-separated pair consisting of '`Display`' and either `'true'` (or 1) or `'false'` (or 0). If you specify `'true'`, the function displays the average absolute and relative improvement of modes and central frequencies every 20 iterations, and show the final stopping information.

Specify `Display` as 1 to show the table or 0 to hide the table.

Output Arguments

`imf` — Intrinsic mode function
matrix | timetable

Intrinsic mode functions, returned as a matrix or timetable. Each `imf` is an amplitude and frequency modulated signal with positive and slowly varying envelopes. Each mode has an instantaneous frequency that is nondecreasing, varies slowly, and is concentrated around a central value. Use `imf` to apply Hilbert-Huang transform to perform spectral analysis on the signal.

`imf` is returned as:

- A matrix where each column is an `imf`, when `x` is a vector

- A timetable, when `x` is a single data column timetable

`residual` — Residual signal
column vector | single data column timetable

Residual signal, returned as a column vector or a single data column timetable. `residual` represents the portion of the original signal `x` not decomposed by `vmd`.

`residual` is returned as:

- A column vector, when `x` is a vector.
- A single data column timetable, when `x` is a single data column timetable.

`info` — Additional information for diagnostics
structure

Additional information for diagnostics, returned as a structure with these fields:

- `ExitFlag` – Termination flag. A value of `0` indicates the algorithm stopped when it reached the maximum number of iterations. A value of `1` indicates the algorithm stopped when it met the absolute and relative tolerances.
- `CentralFrequencies` – Central frequencies of the IMFs.
- `NumIterations` – Total number of iterations.
- `AbsoluteImprovement` – Average squared absolute improvement toward convergence of the IMFs between the final two iterations.
- `RelativeImprovement` – Average relative improvement toward convergence of the IMFs between the final two iterations.
- `LagrangeMultiplier` – Frequency-domain Lagrange multiplier at the last iteration.

More About

Intrinsic Mode Functions

The `vmd` function decomposes a signal $x(t)$ into a small number K of narrowband intrinsic mode functions (IMFs):

$$x(t) = \sum_{k=1}^{K} u_k(t)\,.$$

The IMFs have these characteristics:

1. Each mode u_k is an amplitude and frequency modulated signal of the form

   $$u_k(t) = A_k(t)\cos(\phi_k(t)),$$

   where $\phi_k(t)$ is the phase of the mode and $A_k(t)$ is its envelope.
2. The modes have positive and slowly varying envelopes.
3. Each mode has an instantaneous frequency $\phi'_k(t)$ that is nondecreasing, varies slowly, and is concentrated around a central value f_k.

The variational mode decomposition method simultaneously calculates all the mode waveforms and their central frequencies. The process consists of finding a set of $u_k(t)$ and $f_k(t)$ that minimize the constrained variational problem.

Optimization

To calculate u_k and f_k, the procedure finds an optimum of the augmented Lagrangian

$$L(u_k(t), f_k, \lambda(t)) = \alpha \underbrace{\sum_{k=1}^{K} \left\| \frac{d}{dt}\left[\left(\delta(t) + \frac{j}{\pi t}\right) * u_k(t)\right] e^{-j2\pi f_k t} \right\|_2^2}_{(i)} + \underbrace{\left\| x(t) - \sum_{k=1}^{K} u_k(t) \right\|_2^2}_{(ii)} + \underbrace{\left\langle \lambda(t), x(t) - \sum_{k=1}^{K} u_k(t) \right\rangle}_{(iii)},$$

where the inner product $\langle p(t), q(t) \rangle = \int_{-\infty}^{\infty} p^*(t)\, q(t)\, dt$ and the 2-norm $\|p(t)\|_2^2 = \langle p(t), p(t) \rangle$. The regularization term (i) includes these steps:

1. Use the Hilbert transform to calculate the analytic signal associated with each mode, where * denotes convolution. This results in each mode having a purely positive spectrum.
2. Demodulate the analytic signal to baseband by multiplying it with a complex exponential.
3. Estimate the bandwidth by calculating the squared 2-norm of the gradient of the demodulated analytic signal.

Terms (ii) and (iii) enforce the constraint $x(t) = \sum_{k=1}^{K} u_k(t)$ by imposing a quadratic penalty and incorporating a Lagrange multiplier. The `PenaltyFactor` α measures the relative importance of (i) compared to (ii) and (iii).

The algorithm solves the optimization problem using the alternating direction method of multipliers described in [1].

Algorithms

The `vmd` function calculates the IMFs in the frequency domain, reconstructing $X(f) = \text{DFT}\{x(t)\}$ in terms of $U_k(f) = \text{DFT}\{u_k(t)\}$. To remove edge effects, the algorithm extends the signal by mirroring half its length on either side.

The Lagrange multiplier introduced in “Optimization” on page 1-2748 has the Fourier transform $\Lambda(f)$. The length of the Lagrange multiplier vector is the length of the extended signal.

Unless otherwise specified in `'InitialIMFs'`, the IMFs are initialized at zero. Initialize `'CentralFrequencies'` using one of the methods specified in `'InitializeMethod'`. `vmd` iteratively updates the modes until one of these conditions is met:

- $\sum_k \|u_k^{n+1}(t) - u_k^n(t)\|_2^2 / \|u_k^n(t)\|_2^2 < \varepsilon_r$ and $\sum_k \|u_k^{n+1}(t) - u_k^n(t)\|_2^2 < \varepsilon_a$ are jointly satisfied, where ε_r and ε_a are specified using `'RelativeTolerance'` and `'AbsoluteTolerance'`, respectively.
- The algorithm exceeds the maximum number of iterations specified in `'MaxIterations'`.

For the $(n + 1)$ -th iteration, the algorithm performs these steps:

1. Iterate over the K modes of the signal (specified using `'NumIMF'`) to compute:

a The frequency-domain waveforms for each mode using

$$U_k^{n+1}(f) = \frac{X(f) - \sum_{i<k} U_k^{n+1}(f) - \sum_{i>k} U_k^n(f) + \frac{\Lambda^n}{2}(f)}{1 + 2\alpha\left\{2\pi(f - f_k^n)\right\}^2},$$

where $U_k^{n+1}(f)$ is the Fourier transform of the kth mode calculated in the $(n + 1)$-th iteration.

b The kth central frequency f_k^{n+1} using

$$f_k^{n+1} = \frac{\int_0^\infty \left|U_k^{n+1}(f)\right|^2 f\, df}{\int_0^\infty \left|U_k^{n+1}(f)\right|^2 df} \approx \frac{\sum f \left|U_k^{n+1}(f)\right|^2}{\sum \left|U_k^{n+1}(f)\right|^2}.$$

2 Update the Lagrange multiplier using $\Lambda^{n+1}(f) = \Lambda^n(f) + \tau(X(f) - \sum_k U_k^{n+1}(f))$, where τ is the update rate of the Lagrange multiplier, specified using `'LMUpdateRate'`.

References

[1] Boyd, Stephen, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers." *Foundations and Trends® in Machine Learning*. Vol 3, Number 1, 2011, pp. 1–122.

[2] Dragomiretskiy, Konstantin, and Dominique Zosso. "Variational Mode Decomposition." *IEEE Transactions on Signal Processing*. Vol. 62, Number 3, 2014, pp. 531–534.

[3] Moody, George B., and Roger G. Mark. "The impact of the MIT-BIH Arrhythmia Database." *IEEE Engineering in Medicine and Biology Magazine*. Vol. 20, No. 3, May–June 2001, pp. 45–50.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Timetables are not supported for code generation.

See Also

Apps

Signal Multiresolution Analyzer

Functions

`hht` | `emd`

Topics

“Time-Frequency Gallery”

Introduced in R2020a

window

Window function gateway

Syntax

```
window
w = window(fhandle,n)
w = window(fhandle,n,winopt)
```

Description

`window` opens the **Window Designer** app.

`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 include a window parameter using the next syntax.

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 descriptor. For `chebwin`, `kaiser`, and `tukeywin`, you must enter a *`winopt`* value. For the other windows listed in the following table, *`winopt`* values are optional.

Window	winopt Description	winopt Value
`blackman`	window sampling	`'periodic'` or `'symmetric'`

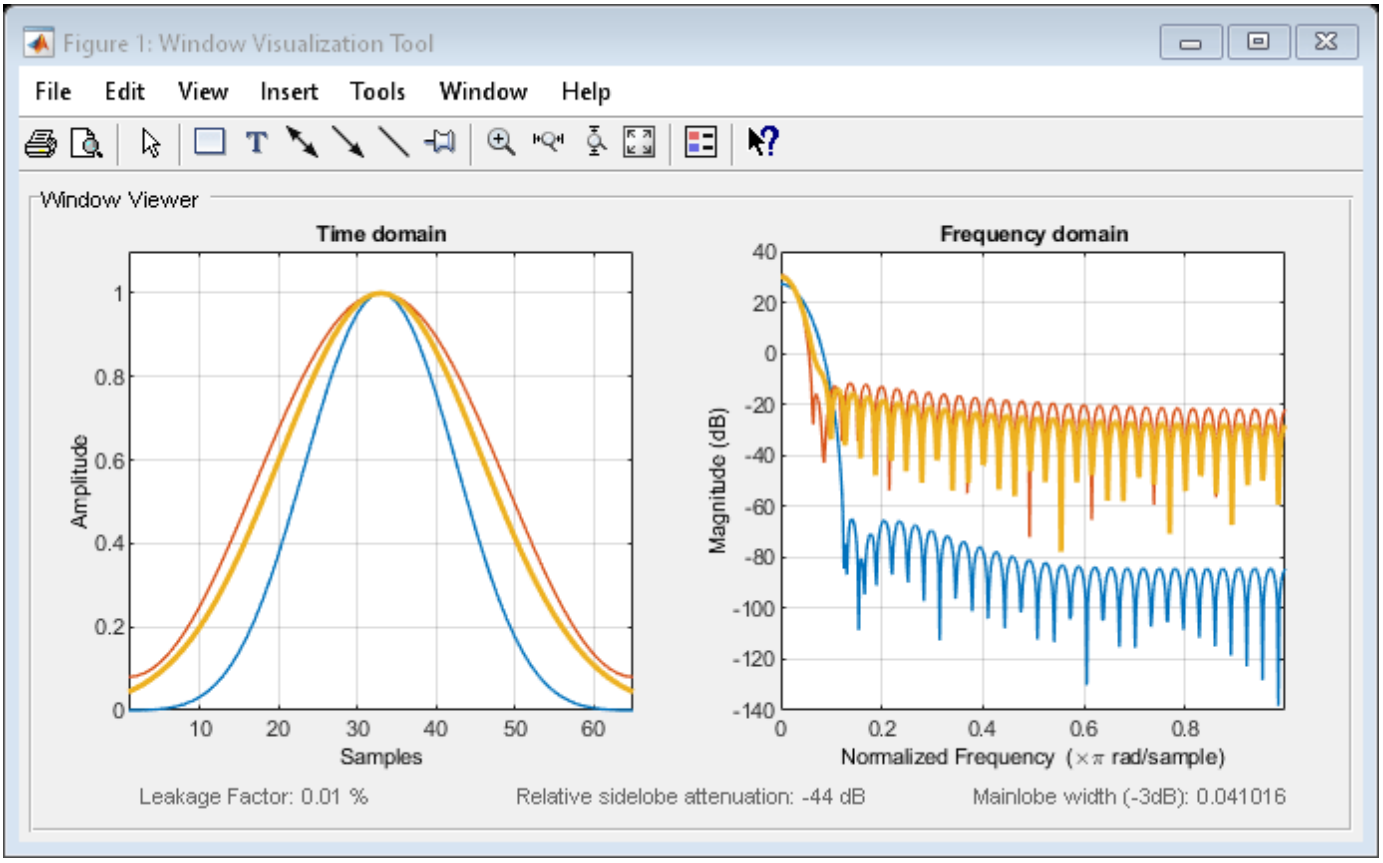
Window	winopt Description	winopt Value
`chebwin`	sidelobe attenuation relative to mainlobe	numeric
`flattopwin`	window sampling	`'periodic'` or `'symmetric'`
`gausswin`	alpha value (reciprocal of standard deviation)	numeric
`hamming`	window sampling	`'periodic'` or `'symmetric'`
`hann`	window sampling	`'periodic'` or `'symmetric'`
`kaiser`	beta value	numeric
`taylorwin`	1. number of sidelobes 2. maximum sidelobe level in dB relative to mainlobe peak	1. integer greater than or equal to 1 2. negative value
`tukeywin`	ratio of taper to constant sections	numeric

Examples

Blackman-Harris, Hamming, and Gaussian Windows

Create Blackman Harris, Hamming, and Gaussian windows and plot them in the same WVTool.

```
N = 65;
w = window(@blackmanharris,N);
w1 = window(@hamming,N);
w2 = window(@gausswin,N,2.5);
wvtool(w,w1,w2)
```



See Also

barthannwin | bartlett | blackman | blackmanharris | bohmanwin | chebwin | flattopwin | gausswin | hamming | hann | kaiser | nuttallwin | parzenwin | rectwin | triang | taylorwin | tukeywin

Introduced before R2006a

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`.

`fcnhndl` is a handle to a filter design function that returns a window vector, such as the `hamming` or `blackman` functions. `fcnarg` is an optional argument that returns a window. You pass the function to `window`. Refer to example 1 in the following section to see the function argument used to design the filter.

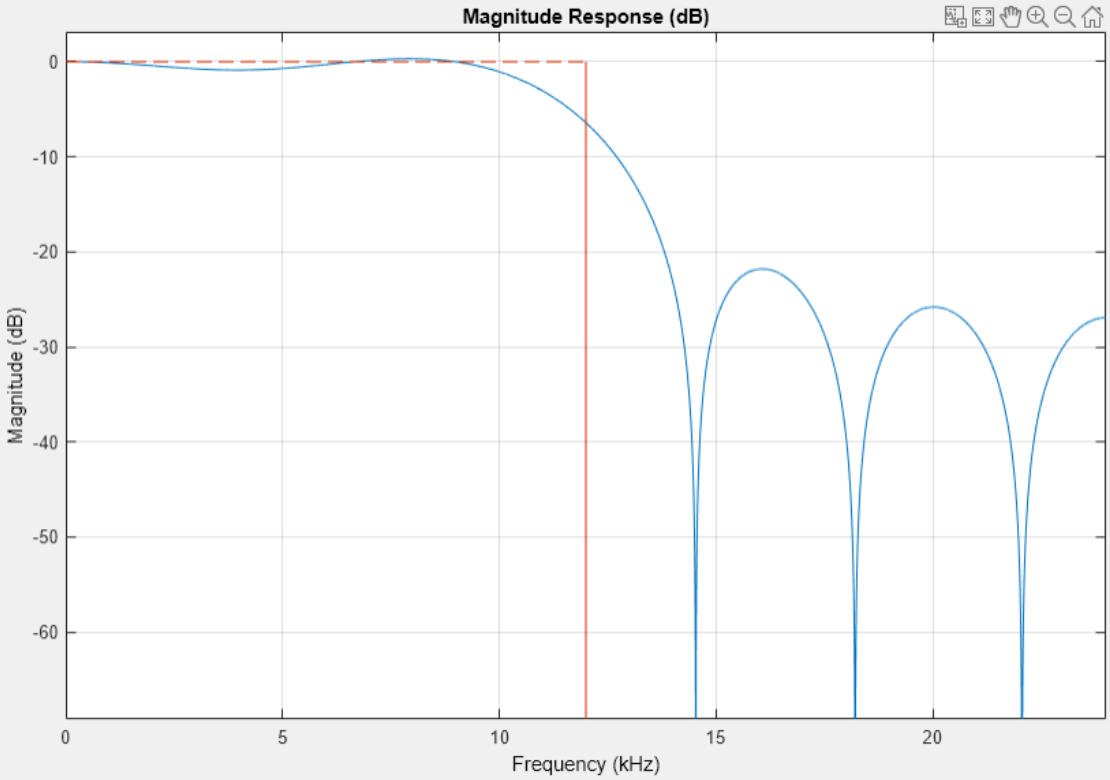
`h = window(d,win)` designs a filter using the vector you supply in `win`. The length of vector `win` must be the same as the impulse response of the filter, which is equal to the filter order plus one.

Examples

Lowpass Filter Window Design

Construct a lowpass filter specification object of order 10 with a cutoff frequency of 12 kHz. Use a sample rate of 48 kHz. Use a function handle to the `kaiser` function to provide the window.

```
d = fdesign.lowpass('n,fc',10,12000,48000);
Hd = window(d,'window',@kaiser);
fvtool(Hd)
```



See Also

Apps
Filter Designer

Functions
`designfilt` | `fdesign` | `filt2block`

Introduced in R2009a

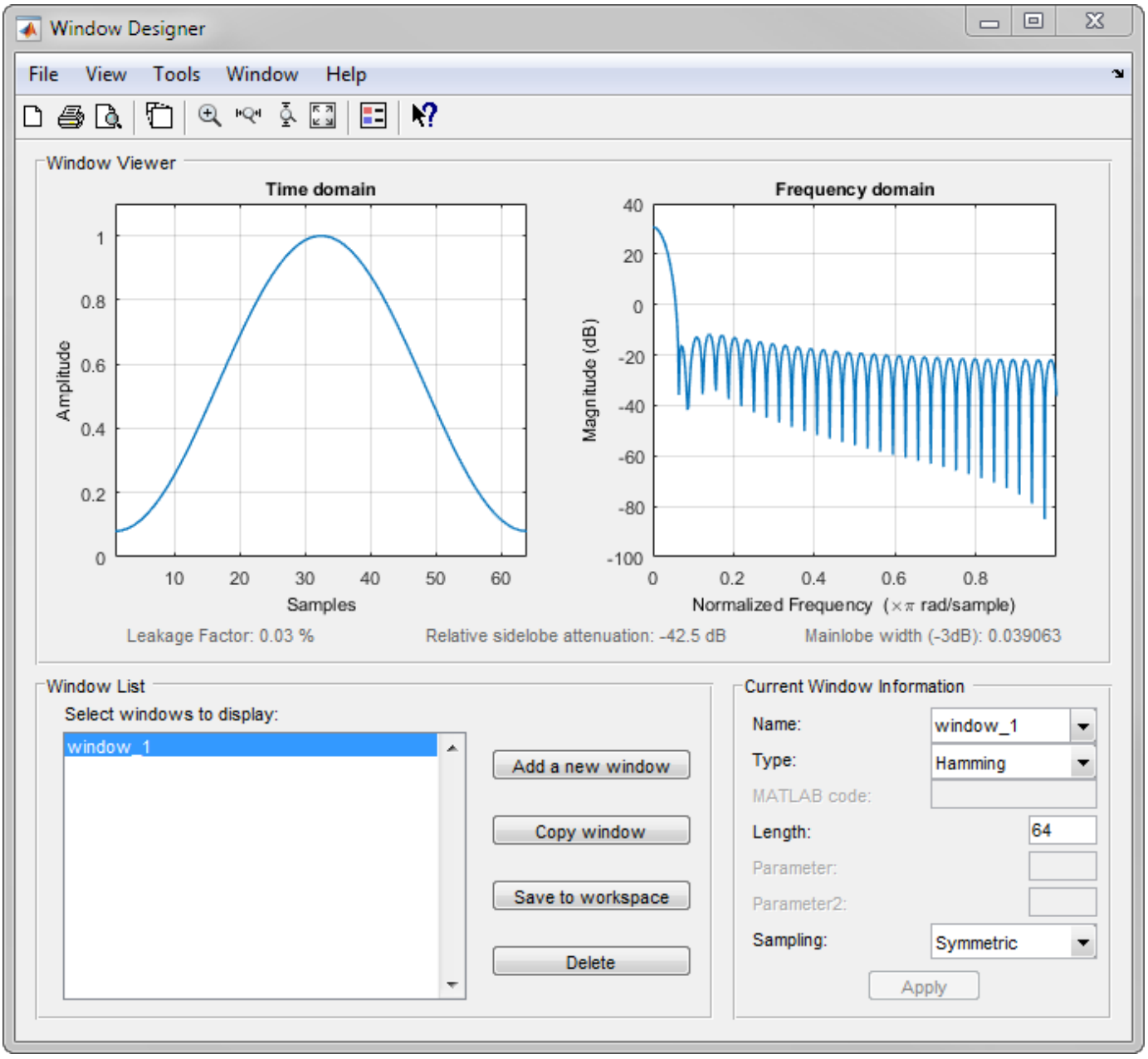
Window Designer

Design and analyze spectral windows

Description

The **Window Designer** app enables you to design and analyze spectral windows. Using this app, you can:

- Display the time-domain and frequency-domain representations of one or more windows.
- Study how the behavior of a window changes as a function of its length and other parameters.
- Design windows graphically and export them to the MATLAB workspace.



Open the Window Designer App

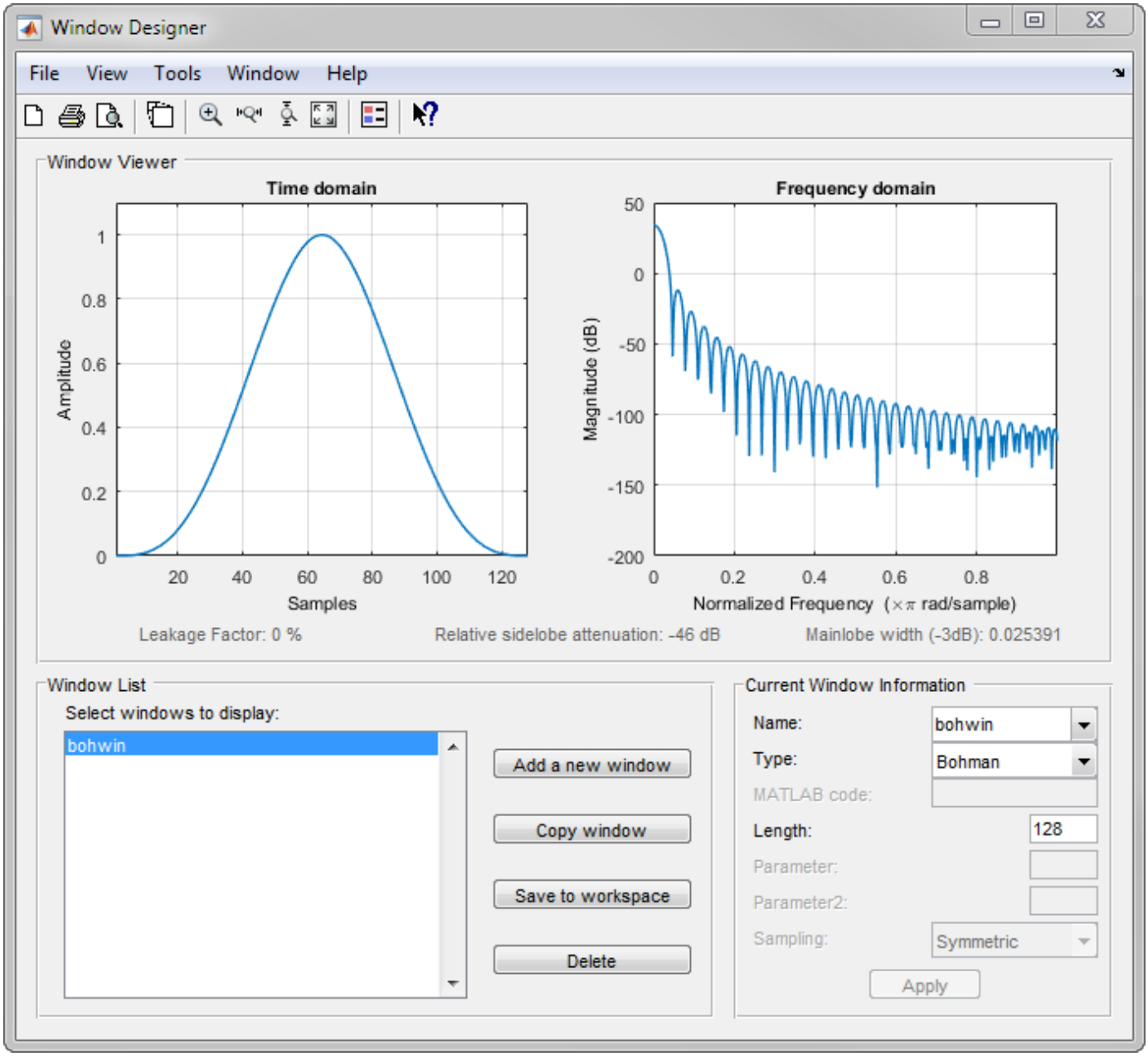
- MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- MATLAB command prompt: Enter `windowDesigner`.

Examples

Bohman Window

Use **Window Designer** to specify a Bohman window of length 128 and export it to the workspace.

1. Create a Bohman window.
   - From the **Type** list, select `Bohman`.
   - Under **Length**, enter `128`.
   - Under **Name**, enter `bohwin`.
2. Click **Apply**. The **Window Viewer** box shows the window in the time and frequency domains.



3. Click **Save to workspace**. In the command line, you see this message:

```
bohwin has been exported to the workspace.
```

4 Verify that the new window is present in the workspace.

```
whos bohwin
```

```
  Name          Size            Bytes  Class     Attributes

  bohwin      128x1              1024  double
```

Kaiser Windows

Use **Window Designer** to see how the behavior of a Kaiser window depends on the window length and the shape parameter, β.

1 Create a Kaiser window.

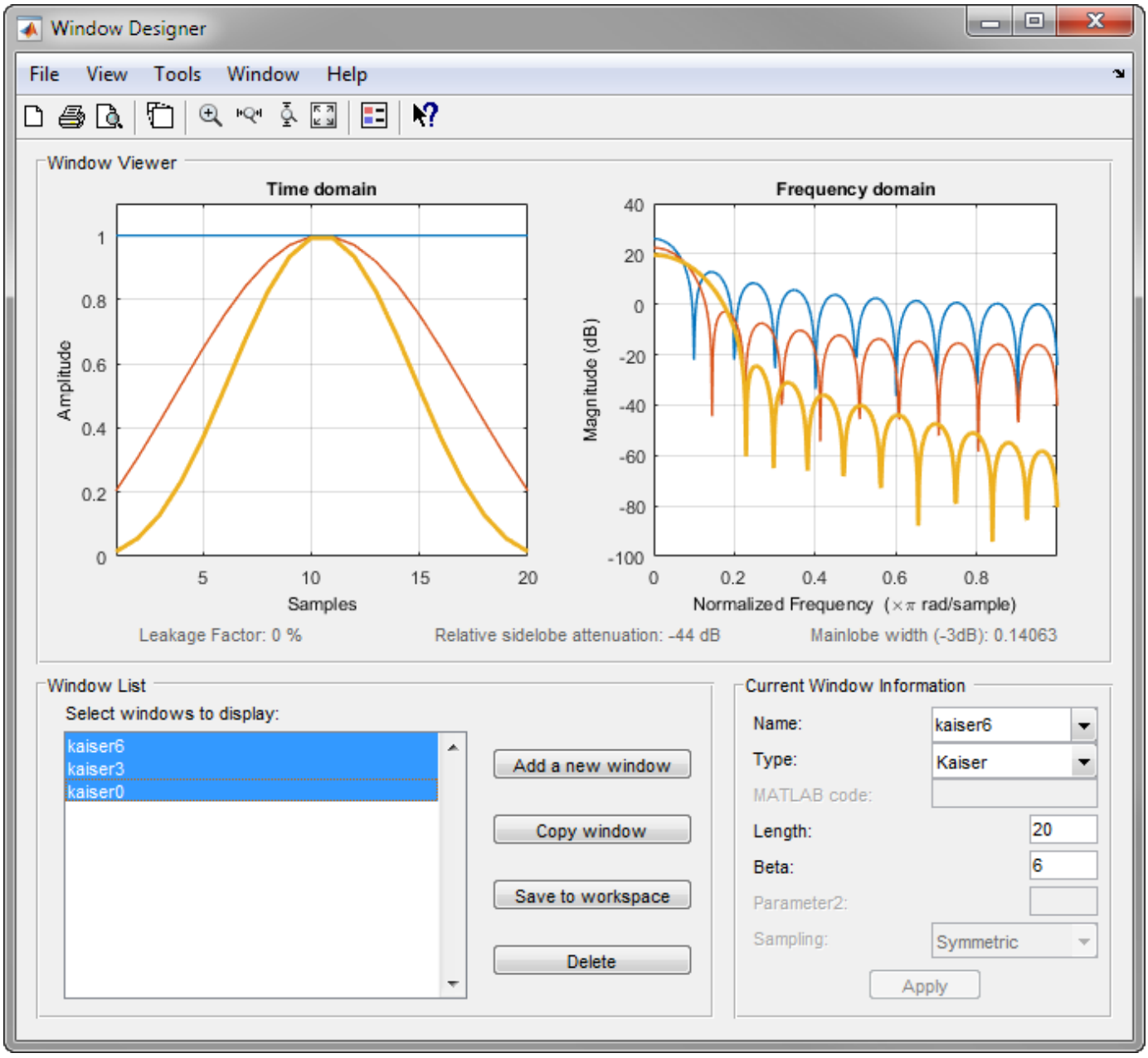
- From the **Type** list, select `Kaiser`.
- Under **Length**, enter `20`.
- Under **Beta**, enter `0`.
- Under **Name**, enter `kaiser0`.

2 Click **Apply**. The **Window Viewer** box shows the window in the time and frequency domains.

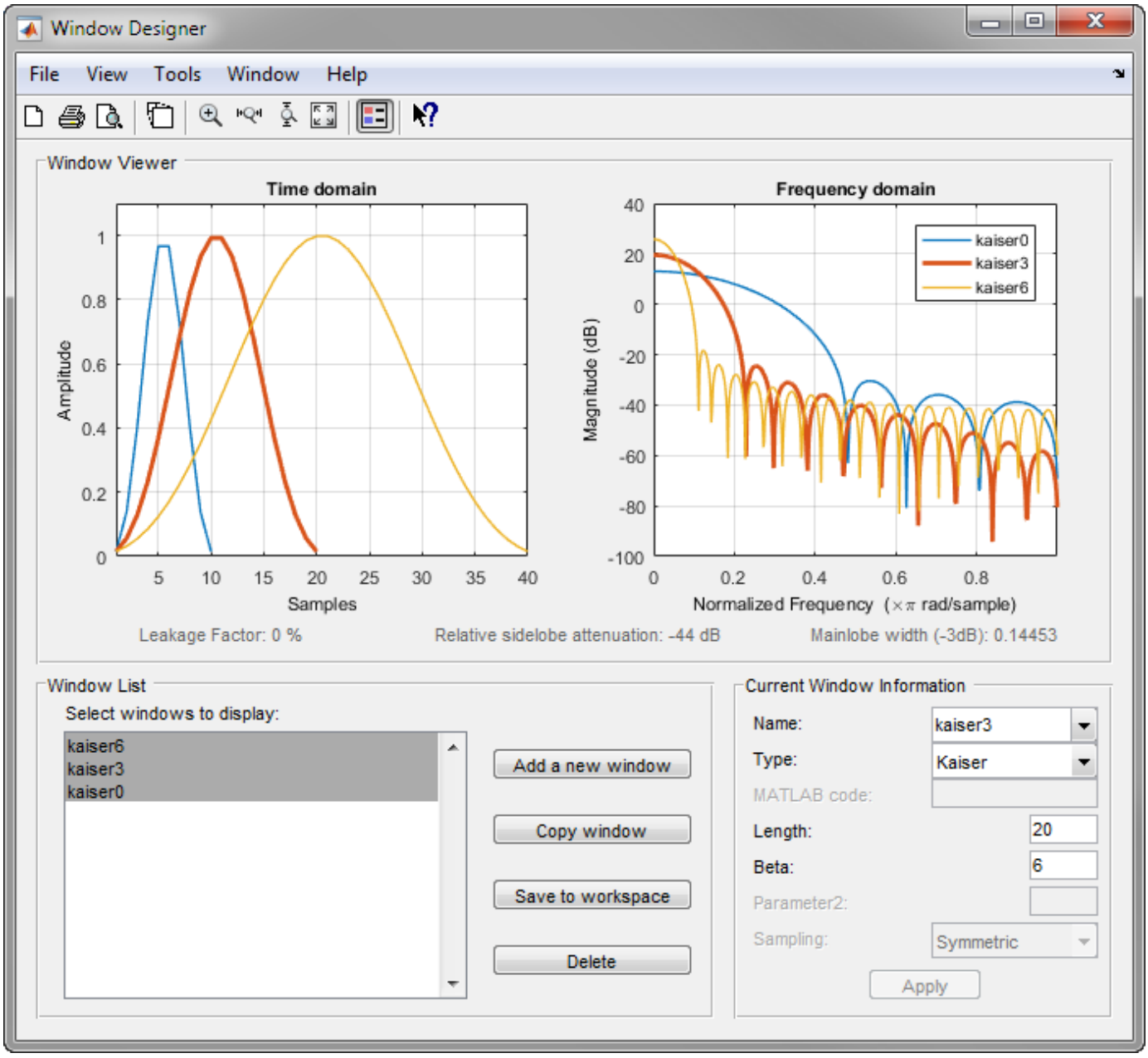
3 Click **Add a new window**. Create a Kaiser window of length `20` with **Beta** equal to `3`. Name the window `kaiser3` and click **Apply**.

4 Click **Copy window** to create a third Kaiser window, `kaiser6`, with **Beta** equal to `6`. Click **Apply**.

5 Under **Window List** select the three windows.



6. Select `kaiser0` from the **Name** list to emphasize it in the **Window Viewer** plots. Set **Length** to `10` and **Beta** to `6`. Click **Apply**.
7. Select `kaiser3` from the **Name** list. Leave **Length** set to `20` and set **Beta** to `6`. Click **Apply**.
8. Select `kaiser6` from the **Name** list. Leave **Beta** set to `6` and set **Length** to `40`. Click **Apply**.
9. Select `kaiser3` from the **Name** list. Click the **Turn Legend On** button .



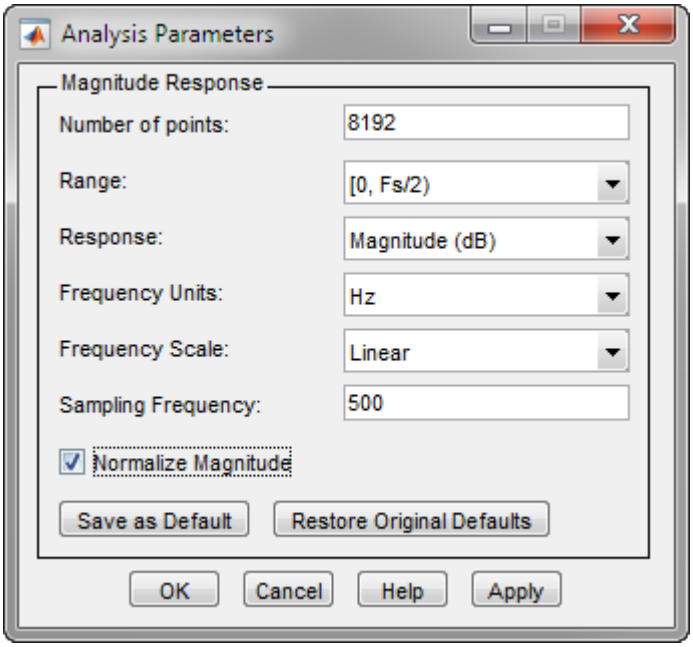
Hamming Window Sidelobes

Use **Window Designer** to visualize the sidelobes of the default 64-sample Hamming window.

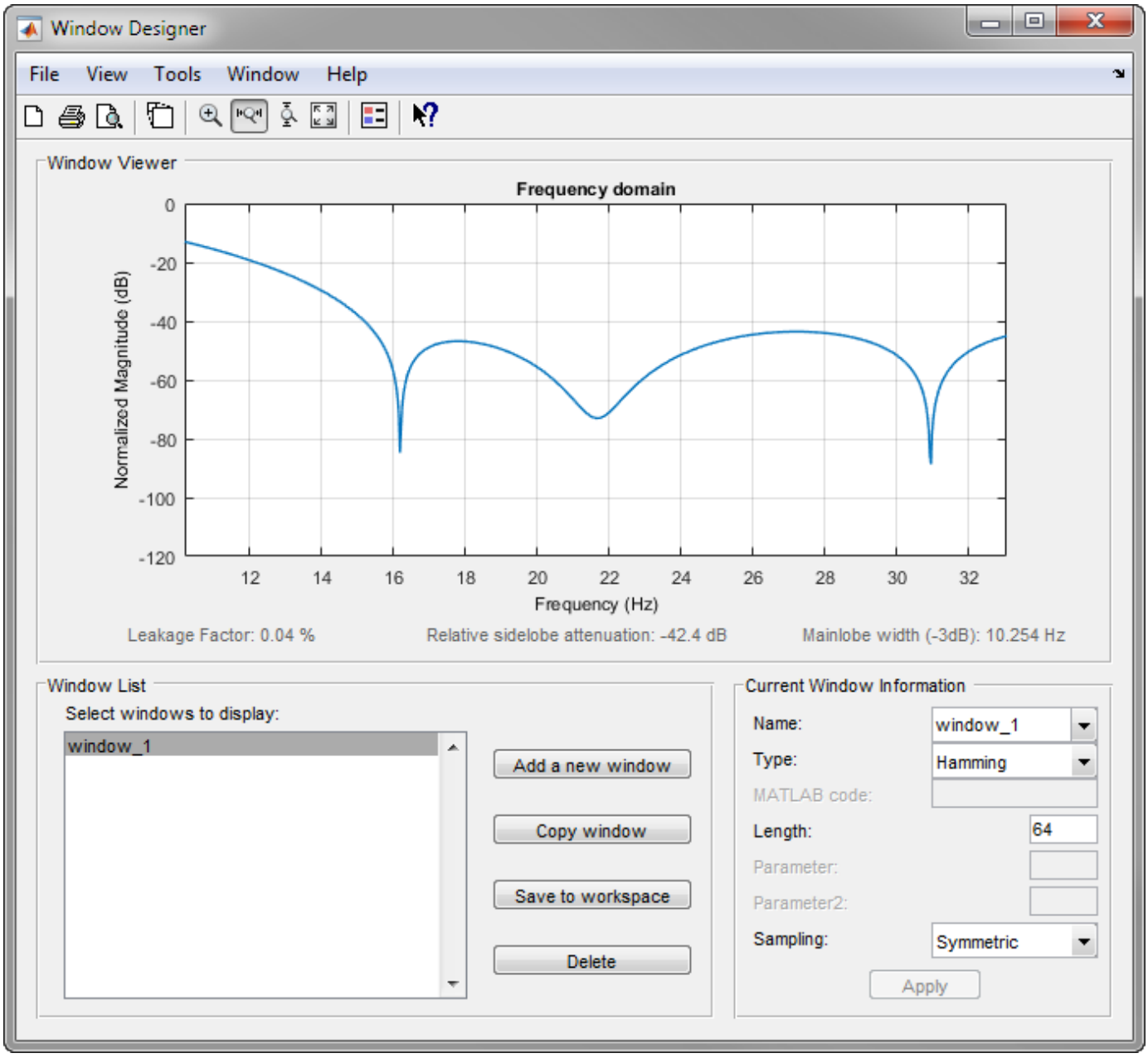
1. In the **View** menu, clear **Time domain** and click **Analysis Parameters**.
2. In the dialog box, specify these parameters:

   - Under **Number of points**, enter 8192.
   - Set **Frequency Units** to Hz.

- Set **Sampling Frequency** to `500` Hz
- Select **Normalize Magnitude**.



3 Click **OK** to close the dialog box. Click the **Zoom X-Axis** button. Zoom into the region between 10 Hz and 35 Hz to view the window's first two sidelobes in detail.



- “Get Started with Window Designer”
- “Generalized Cosine Windows”
- “Kaiser Window”
- “Chebyshev Window”

See Also

Apps
Filter Designer | Signal Analyzer

Tools
WVTool

Topics
“Get Started with Window Designer”
“Generalized Cosine Windows”
“Kaiser Window”
“Chebyshev Window”

Introduced before R2006a

wvd

Wigner-Ville distribution and smoothed pseudo Wigner-Ville distribution

Syntax

```
d = wvd(x)
d = wvd(x,fs)
d = wvd(x,ts)

d = wvd(___,'smoothedPseudo')
d = wvd(___,'smoothedPseudo',twin,fwin)
d = wvd(___,'smoothedPseudo',Name,Value)

d = wvd(___,'MinThreshold',thresh)

[d,f,t] = wvd(___)

wvd(___)
```

Description

`d = wvd(x)` returns the Wigner-Ville distribution of `x`.

`d = wvd(x,fs)` returns the Wigner-Ville distribution when `x` is sampled at a rate `fs`.

`d = wvd(x,ts)` returns the Wigner-Ville distribution when `x` is sampled with a time interval `ts` between samples.

`d = wvd(___,'smoothedPseudo')` returns the smoothed pseudo Wigner-Ville distribution of `x`. The function uses the length of the input signal to choose the lengths of the windows used for time and frequency smoothing. This syntax can include any combination of input arguments from previous syntaxes.

`d = wvd(___,'smoothedPseudo',twin,fwin)` specifies the time window, `twin`, and the frequency window, `fwin`, used for smoothing. To use the default window for either time or frequency smoothing, specify the corresponding argument as empty, `[]`.

`d = wvd(___,'smoothedPseudo',Name,Value)` specifies additional options for the smoothed pseudo Wigner-Ville distribution using name-value pair arguments. You can specify `twin` and `fwin` in this syntax, or you can omit them.

`d = wvd(___,'MinThreshold',thresh)` sets to zero those elements of `d` whose amplitude is less than `thresh`. This syntax applies to both the Wigner-Ville distribution and the smoothed pseudo Wigner-Ville distribution.

`[d,f,t] = wvd(___)` also returns a vector of frequencies, `f`, and a vector of times, `t`, at which `d` is computed.

`wvd(___)` with no output arguments plots the Wigner-Ville or smoothed pseudo Wigner-Ville distribution in the current figure.

Examples

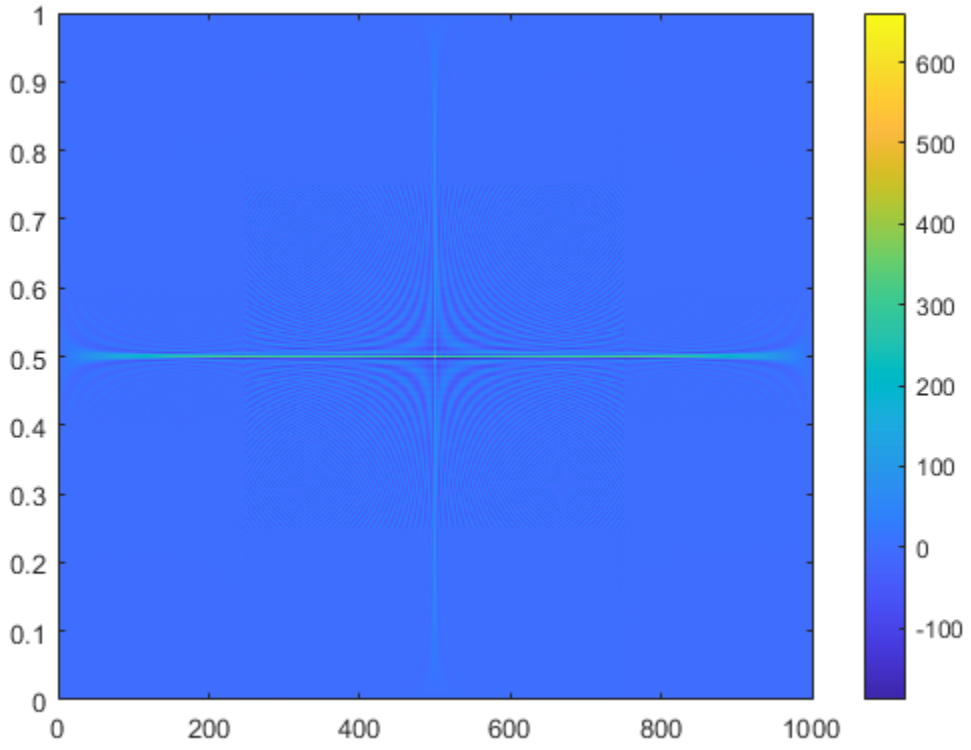
Wigner-Ville Distribution of Impulse and Tone

Generate a 1000-sample impulse and a 1000-sample tone with normalized frequency $\pi/2$. Compute the Wigner-Ville distribution of the sum of the two signals.

```
x = zeros(1001,1);
x(500) = 10;

y = sin(pi*(0:1000)/2)';

[d,f,t] = wvd(x+y);
```

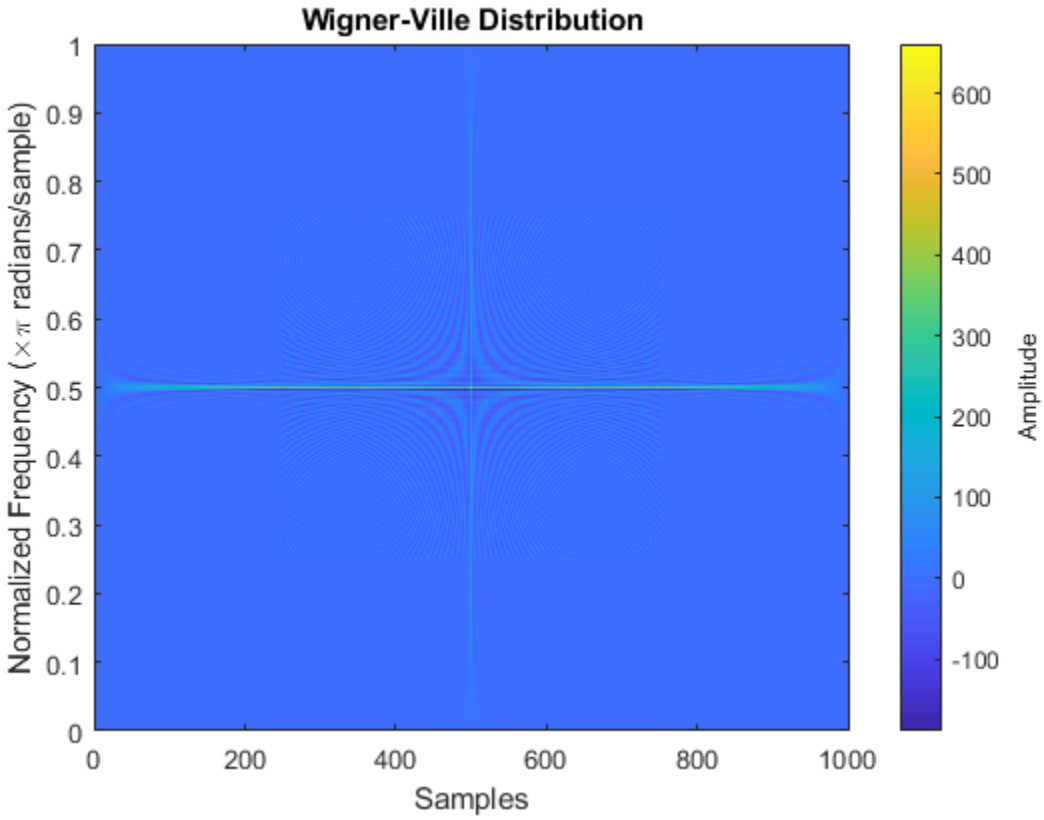
Plot the Wigner-Ville distribution.

```
imagesc(t,f,d)
axis xy
colorbar
```



Reproduce the result by calling `wvd` with no output arguments.

```
wvd(x+y)
```



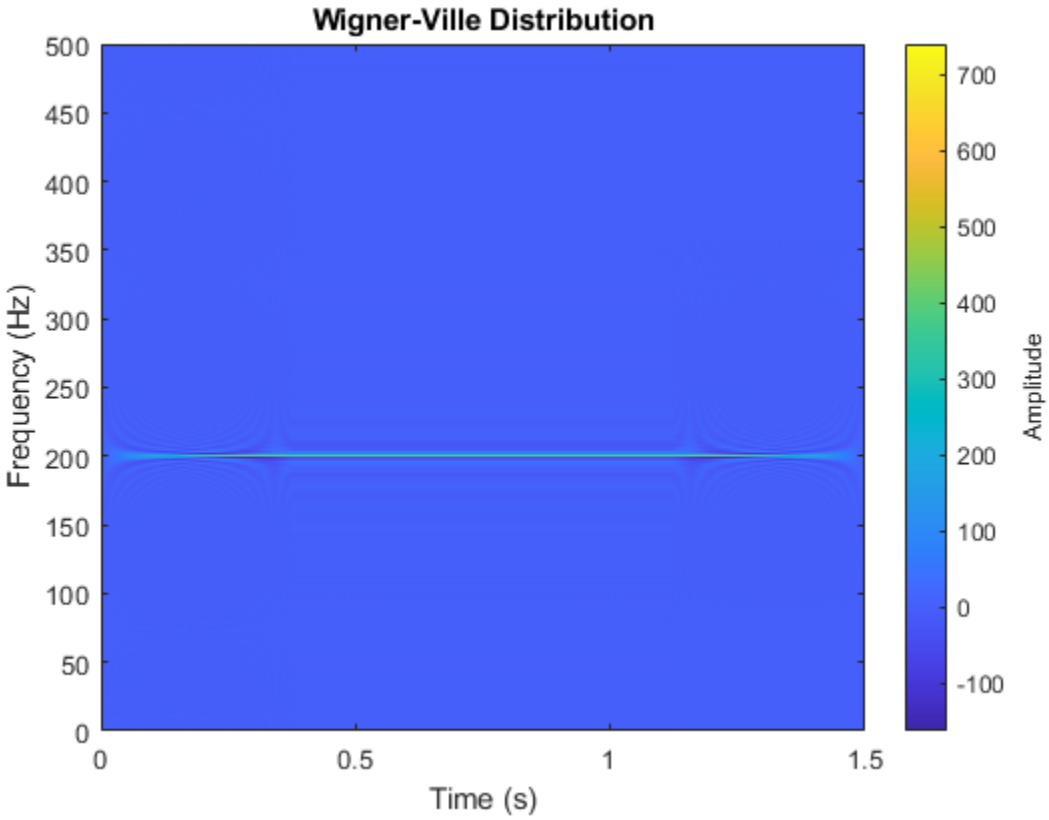
Wigner-Ville Distribution of Sinusoids

Generate a signal consisting of a 200 Hz sinusoid sampled at 1 kHz for 1.5 seconds.

```
fs = 1000;
t = (0:1/fs:1.5)';
x = cos(2*pi*t*200);
```

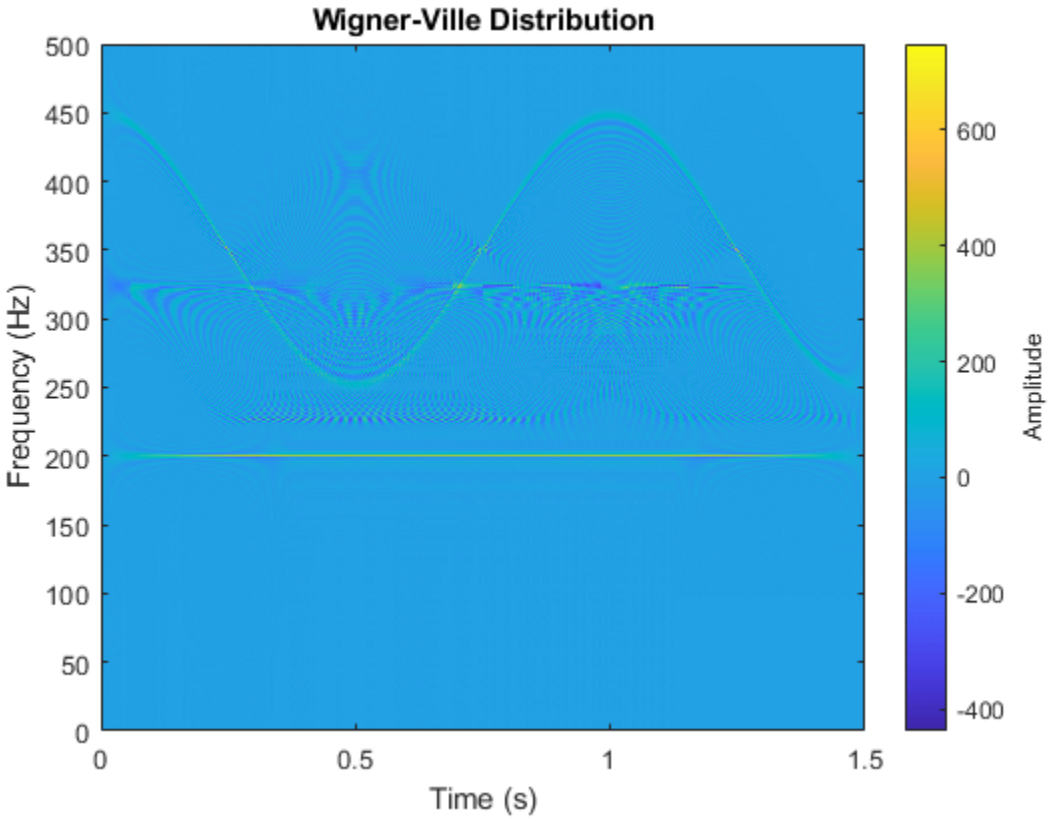
Compute the Wigner-Ville distribution of the signal.

```
wvd(x,fs)
```



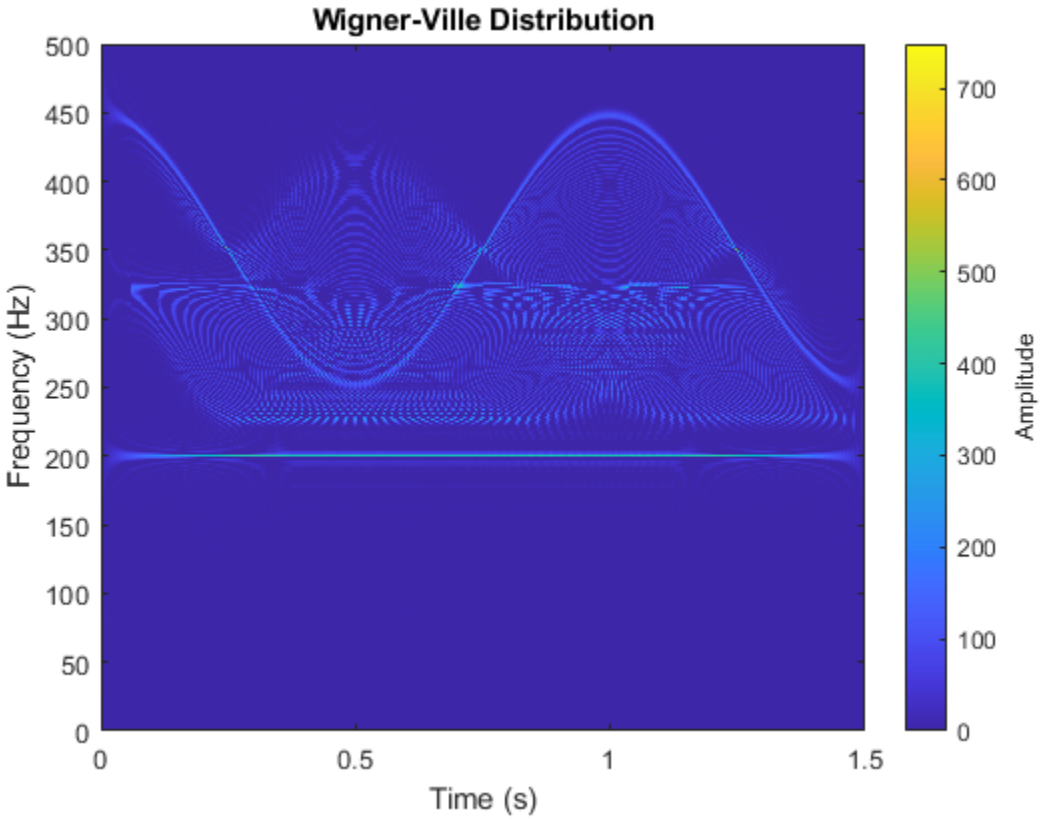
Add to the signal a chirp whose frequency varies sinusoidally between 250 Hz and 450 Hz. Convert the signal to a MATLAB® timetable. Compute the Wigner-Ville distribution.

```
x = x + vco(cos(2*pi*t),[250 450],fs);
xt = timetable(seconds(t),x);

wvd(xt)
```



Set to zero the distribution elements with amplitude less than 0.

```
wvd(xt,'MinThreshold',0)
```



Wigner-Ville Distribution of Chirps

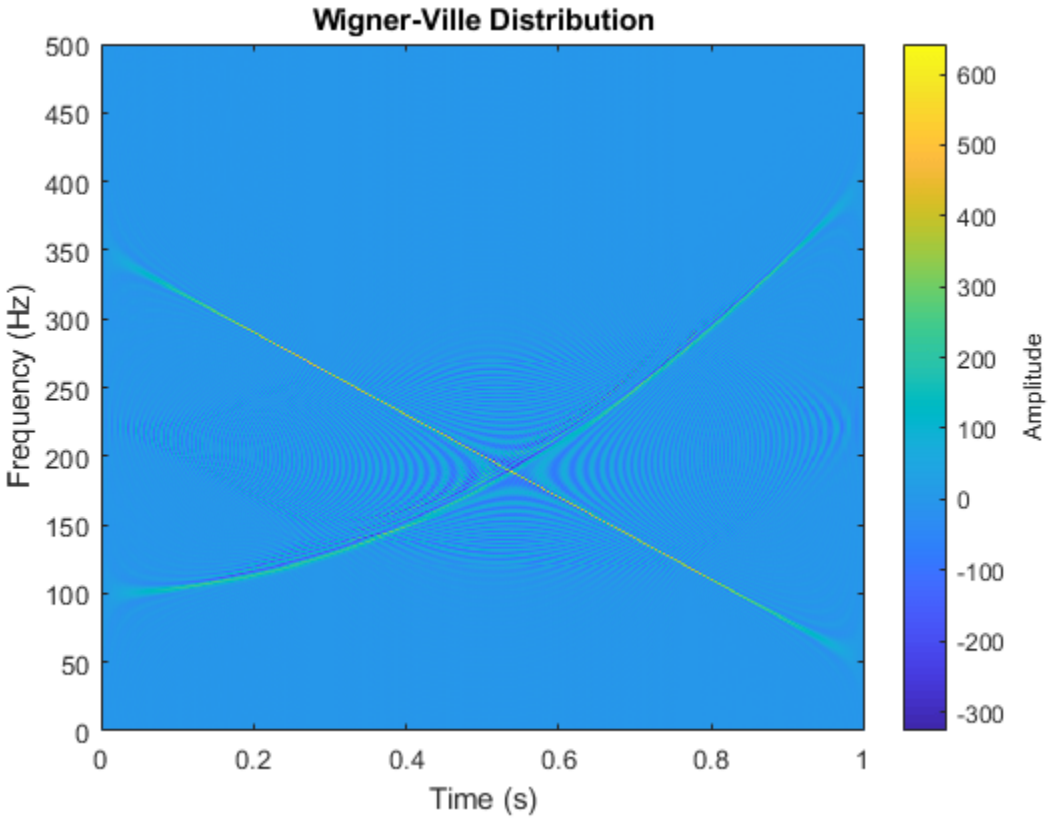
Generate a signal sampled at 1 kHz for 1 second. One component of the signal is a chirp that increases in frequency quadratically from 100 Hz to 400 Hz during the measurement. The other component of the signal is a chirp that decreases in frequency linearly from 350 Hz to 50 Hz in the same lapse.

Store the signal in a timetable.

```
fs = 1000;
t = 0:1/fs:1;

x = chirp(t,100,1,400,'quadratic') + chirp(t,350,1,50);
```

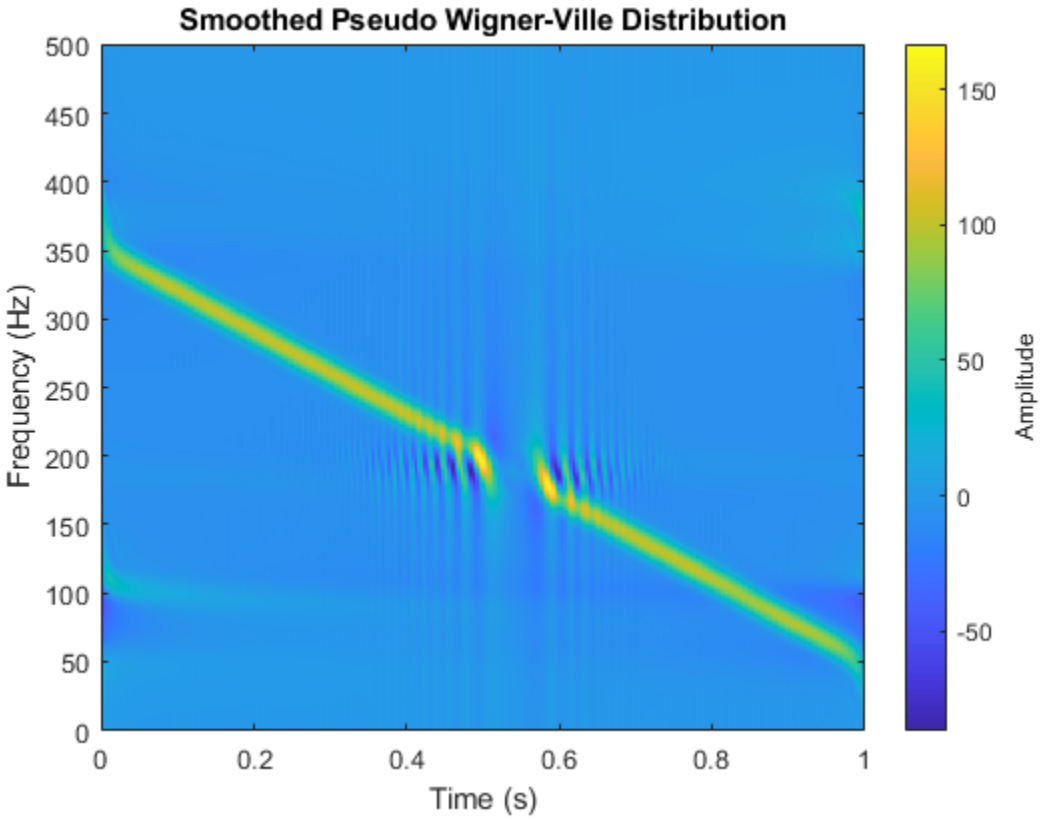
Compute the Wigner-Ville distribution of the signal.

```
wvd(x,fs)
```



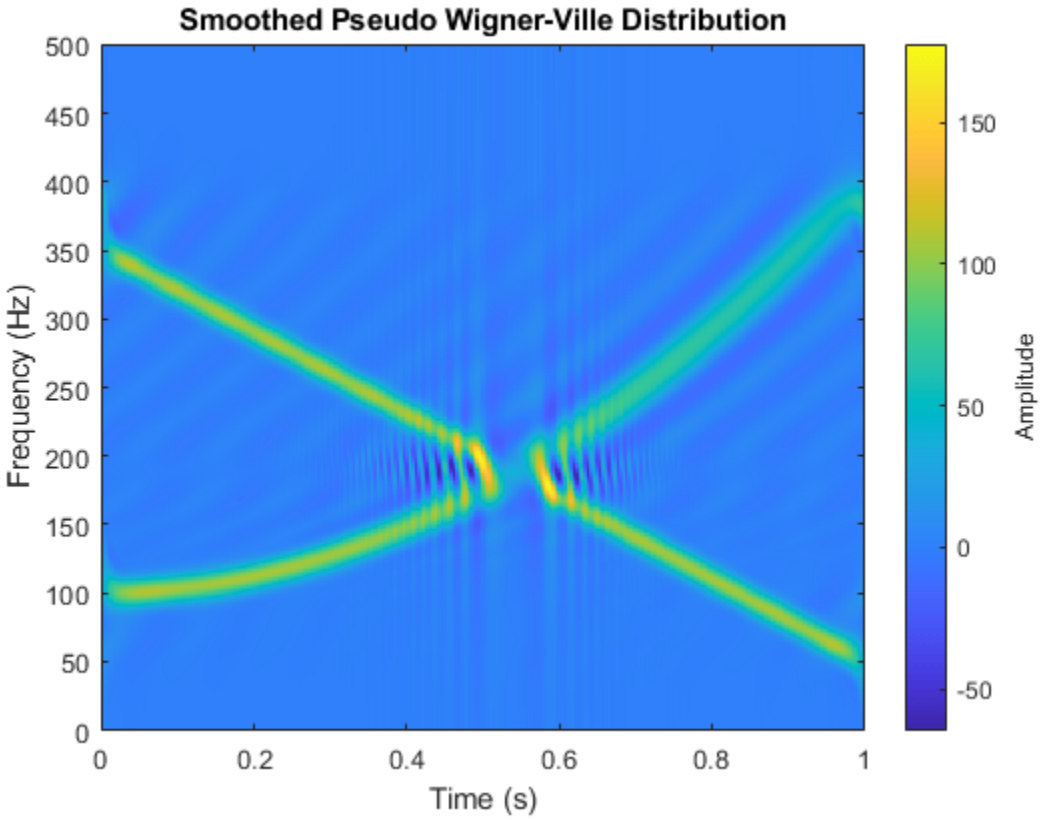
Compute the smoothed pseudo Wigner-Ville distribution of the signal. Specify 501 frequency points and 502 time points.

```
wvd(x,fs,'smoothedPseudo','NumFrequencyPoints',501,'NumTimePoints',502)
```



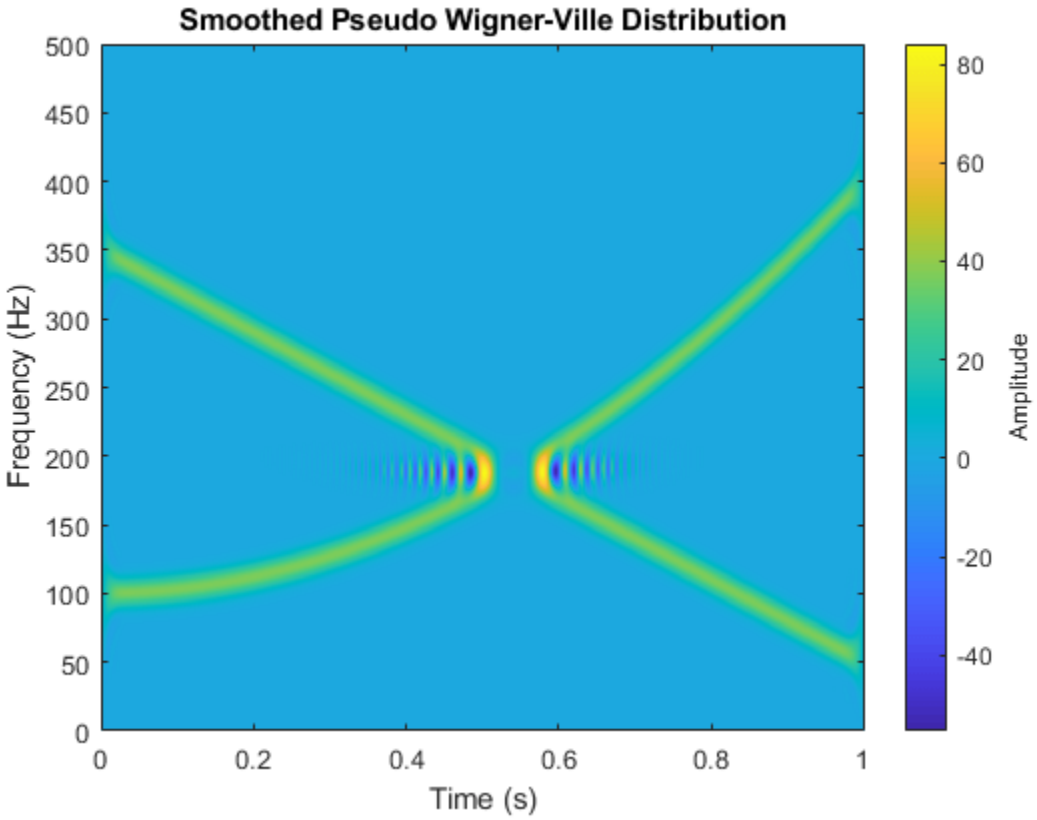
Increase the number of time points so the quadratic chirp becomes visible.

```
wvd(x,fs,'smoothedPseudo','NumFrequencyPoints',501,'NumTimePoints',522)
```



Increase the frequency points and time points to get a sharper image.

```
wvd(x,fs,'smoothedPseudo','NumFrequencyPoints',1000,'NumTimePoints',1502)
```



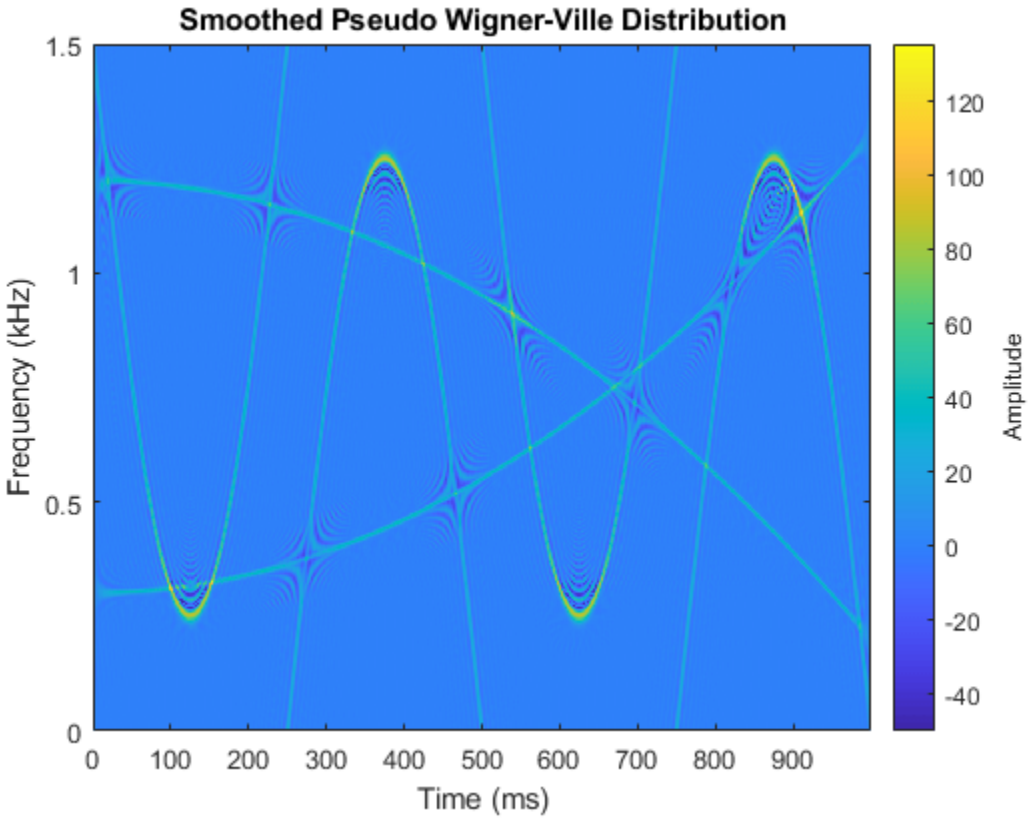
Smoothed Pseudo Wigner-Ville Distribution of Complex Signal

Generate a two-component signal sampled at 3 kHz for 1 second. The first component is a quadratic chirp whose frequency increases from 300 Hz to 1300 Hz during the measurement. The second component is a chirp with sinusoidally varying frequency content. The signal is embedded in white Gaussian noise. Express the time between consecutive samples as a `duration` scalar.

```
fs = 3000;
t = 0:1/fs:1-1/fs;
dt = seconds(t(2)-t(1));

x1 = chirp(t,300,t(end),1300,'quadratic');
x2 = exp(2j*pi*100*cos(2*pi*2*t));

x = x1 + x2 + randn(size(t))/10;
```

Compute and plot the smoothed pseudo Wigner Ville of the signal. Window the distribution in time using a 601-sample Hamming window and in frequency using a 305-sample rectangular window. Use 600 frequency points for the display. Set to zero those components of the distribution with amplitude less than −50.

```
wvd(x,dt,'smoothedPseudo',hamming(601),rectwin(305), ...
    'NumFrequencyPoints',600,'MinThreshold',-50)
```



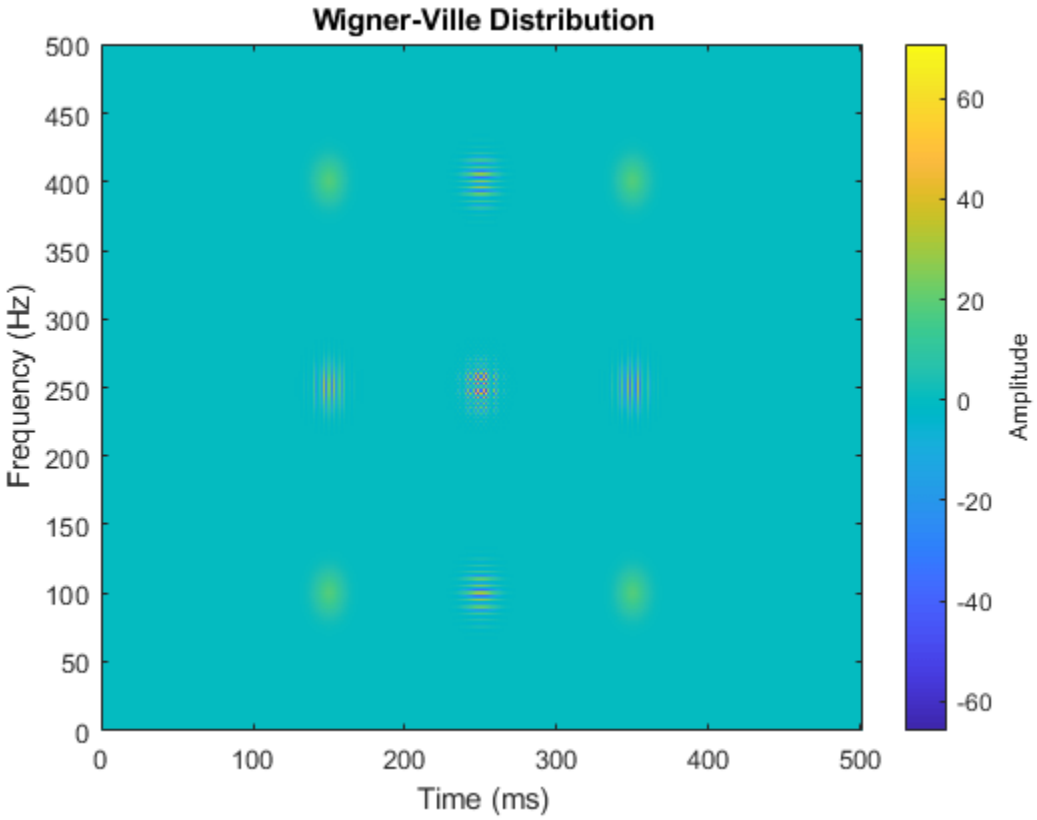
Interference Terms

Generate a signal composed of four Gaussian atoms. Each atom consists of a sinusoid modulated by a Gaussian. The sinusoids have frequencies of 100 Hz and 400 Hz. The Gaussians are centered at 150 milliseconds and 350 milliseconds and have a variance of 0.01^2. All atoms have unit amplitude. The signal is sampled at 1 kHz for half a second.

```
fs = 1000;
t = (0:1/fs:0.5)';

f1 = 100;
f2 = 400;

mu1 = 0.15;
mu2 = 0.35;

gaussFun = @(A,x,mu,f) exp(-(x-mu).^2/(2*0.01^2)).*sin(2*pi*f.*x)*A';

s = gaussFun([1 1 1 1],t,[mu1 mu1 mu2 mu2],[f1 f2 f1 f2]);
```

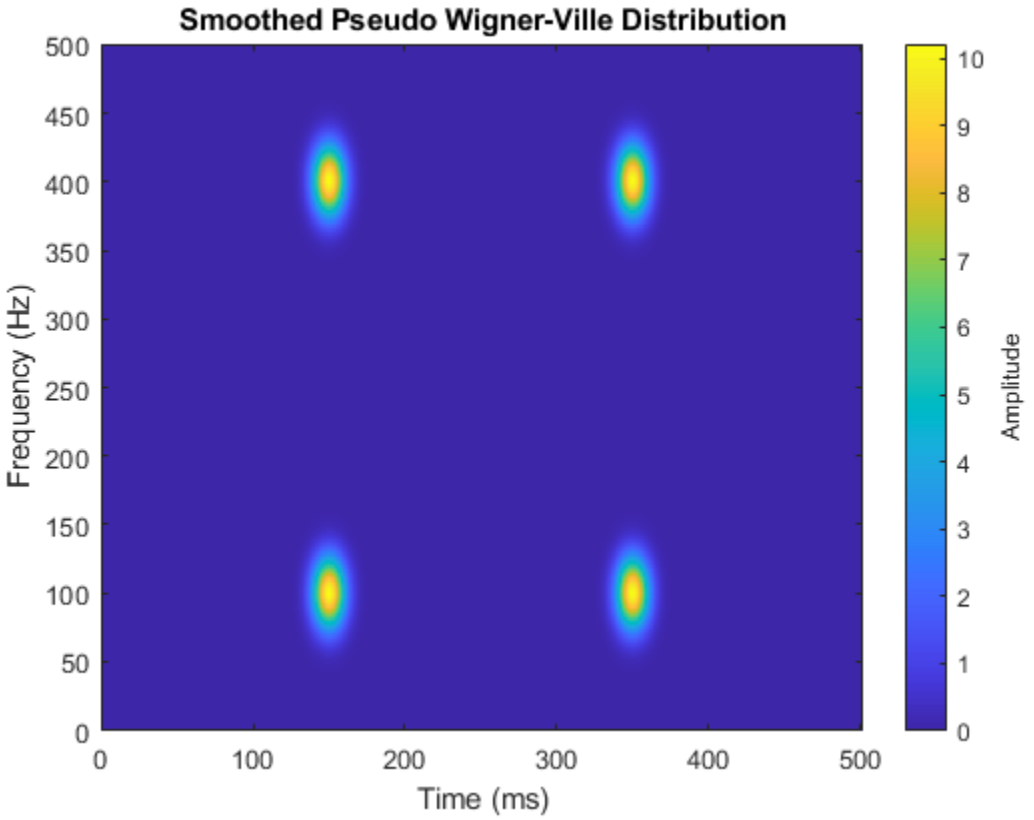
Compute and display the Wigner-Ville distribution of the signal. Interference terms, which can have negative values, appear halfway between each pair of auto-terms.

```
wvd(s,fs)
```



Compute and display the smoothed pseudo Wigner-Ville distribution of the signal. Smoothing in time and frequency attenuates the interference terms.

```
wvd(s,fs,'SmoothedPseudo')
```



Input Arguments

x — Input signal

vector | timetable

Input signal, specified as a vector or a MATLAB timetable containing a single vector variable.

- If `x` is a timetable, then it must contain increasing finite row times.
- If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

If the input signal has odd length, the function appends a zero to make the length even.

Example: `cos(pi/8*(0:159))'+randn(160,1)/10` specifies a sinusoid embedded in white noise.

Example: `timetable(seconds(0:5)',rand(6,1))` specifies a random variable sampled at 1 Hz for 5 seconds.

Data Types: `single` | `double`
Complex Number Support: Yes

fs — Sample rate

`2*pi` (default) | positive numeric scalar

Sample rate, specified as a positive numeric scalar.

`ts` — Sample time
`duration` scalar

Sample time, specified as a `duration` scalar.

`twin`, `fwin` — Time and frequency windows
vectors of odd length

Time and frequency windows used for smoothing, specified as vectors of odd length. By default, `wvd` uses Kaiser windows with shape factor $\beta = 20$.

- The default length of `twin` is the smallest odd integer greater than or equal to `round(length(x)/10)`.
- The default length of `fwin` is the smallest odd integer greater than or equal to `nf/4`, where `nf` is specified using `NumFrequencyPoints`.

Each window must have a length smaller than or equal to `2*ceil(length(x)/2)`.

Example: `kaiser(65,0.5)` specifies a 65-sample Kaiser window with a shape factor of 0.5.

`thresh` — Minimum nonzero value
`-Inf` (default) | real scalar

Minimum nonzero value, specified as a real scalar. The function sets to zero those elements of `d` whose amplitudes are less than `thresh`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'NumFrequencyPoints',201,'NumTimePoints',300` computes the Wigner-Ville distribution at 201 frequency points and 300 time points.

`NumFrequencyPoints` — Number of frequency points
`2*ceil(length(x)/2)` (default) | integer

Number of frequency points, specified as the comma-separated pair consisting of `'NumFrequencyPoints'` and an integer. This argument controls the degree of oversampling in frequency. The number of frequency points must be at least `(length(fwin)+1)/2` and cannot be greater than the default.

`NumTimePoints` — Number of time points
`4*ceil(length(x)/2)` (default) | even integer

Number of time points, specified as the comma-separated pair consisting of `'NumTimePoints'` and an even integer. This argument controls the degree of oversampling in time [3]. The number of time points must be at least `2*length(twin)` and cannot be greater than the default.

Tip If the input signal is large, reduce the number of time points to lower the memory requirements and speed up the computation.

Output Arguments

`d` — Wigner-Ville distribution

matrix

Wigner-Ville distribution, returned as a matrix. Time increases across the columns of `d`, and frequency increases down the rows. The matrix is of size $N_f \times N_t$, where N_f is the length of `f` and N_t is the length of `t`.

`f` — Frequencies

vector

Frequencies, returned as a vector.

- If the input has time information, then `f` contains frequencies expressed in Hz.
- If the input does not have time information, then `f` contains normalized frequencies expressed in rad/sample.

`t` — Time instants

vector

Time instants, returned as a vector.

- If the input has time information, then `t` contains time values expressed in seconds.
- If the input does not have time information, then `t` contains sample numbers.

More About

Wigner-Ville Distribution

The Wigner-Ville distribution provides a high-resolution time-frequency representation of a signal. The distribution has applications in signal visualization, detection, and estimation.

For a continuous signal $x(t)$, the Wigner-Ville distribution is defined as

$$\mathrm{WVD}_x(t,f) = \int_{-\infty}^{\infty} x\left(t+\frac{\tau}{2}\right) x^*\left(t-\frac{\tau}{2}\right) e^{-j2\pi f\tau}\, d\tau\,.$$

For a discrete signal with N samples, the distribution becomes

$$\mathrm{WVD}_x(n,k) = \sum_{m=-N}^{N} x(n+m/2)\, x^*(n-m/2)\, e^{-j2\pi km/N}\,.$$

For odd values of m, the definition requires evaluation of the signal at half-integer sample values. It therefore requires interpolation, which makes it necessary to zero-pad the discrete Fourier transform to avoid aliasing.

The Wigner-Ville distribution contains interference terms that often complicate its interpretation. To sharpen the distribution, one can filter the definition with lowpass windows. The smoothed pseudo Wigner-Ville distribution uses independent windows to smooth in time and frequency:

$$\mathrm{SPWVD}_x^{g,H}(t,f) = \int_{-\infty}^{\infty} g(t)\, H(f)\, x\left(t+\frac{\tau}{2}\right) x^*\left(t-\frac{\tau}{2}\right) e^{-j2\pi f\tau}\, d\tau\,.$$

References

[1] Cohen, Leon. *Time-Frequency Analysis: Theory and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[2] Mallat, Stéphane. *A Wavelet Tour of Signal Processing*. Second Edition. San Diego, CA: Academic Press, 1999.

[3] O'Toole, John M., and Boualem Boashash. "Fast and Memory-Efficient algorithms for Computing Quadratic Time-Frequency Distributions." *Applied and Computational Harmonic Analysis*. Vol. 35, Number 2, 2013, pp. 350–358.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Arguments specified using name-value pairs must be compile-time constants.
- Timetables are not supported for code generation.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Functions

`fsst` | `pspectrum` | `spectrogram` | `xwvd`

Topics

“Time-Frequency Gallery”

Introduced in R2018b

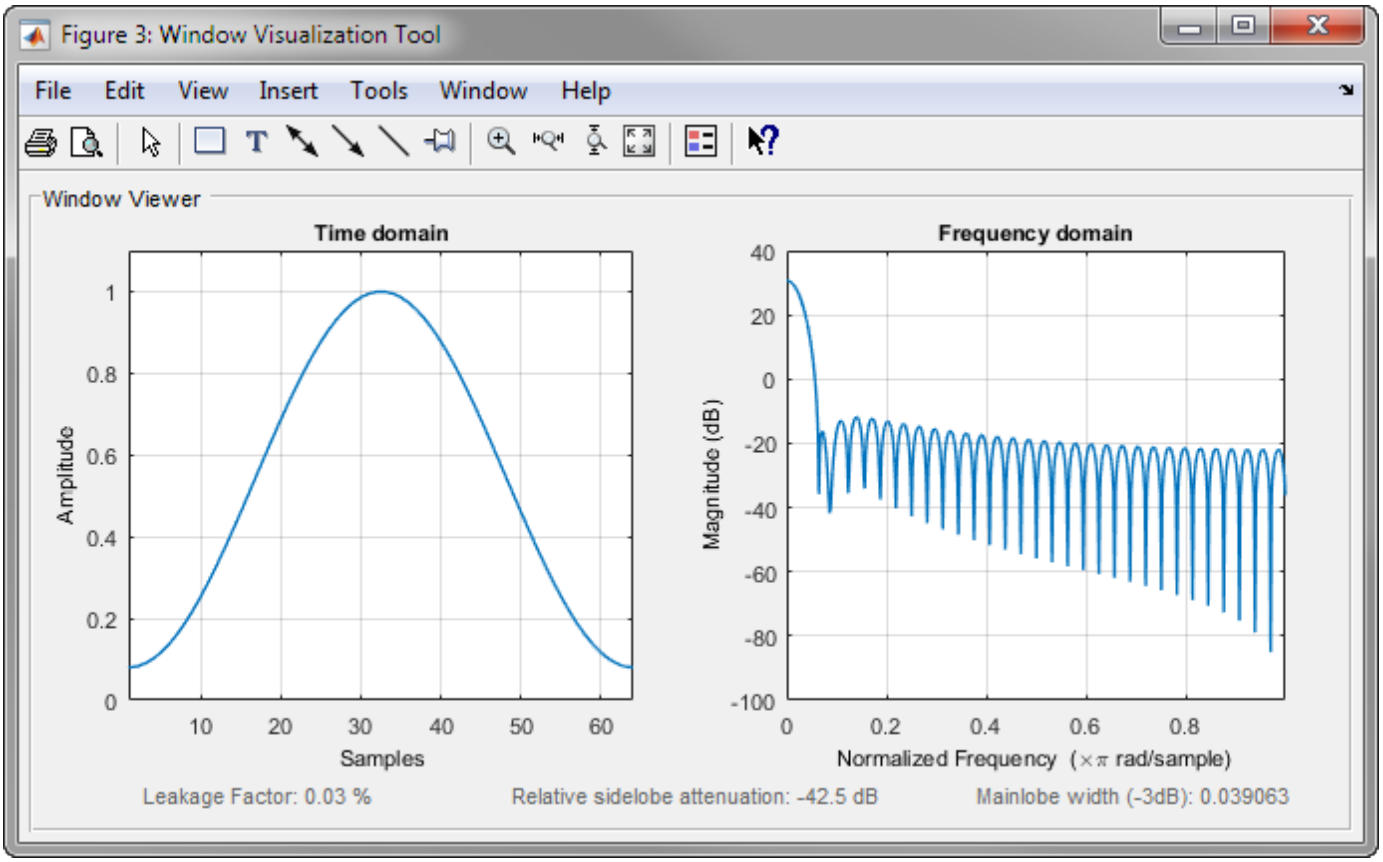
WVTool

Open Window Visualization Tool

Description

Window Visualization Tool is an interactive tool that enables you to visualize time and frequency domain plots of the window vector. 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.

Note A related tool, **Window Designer**, is available for designing and analyzing windows.



Open the WVTool

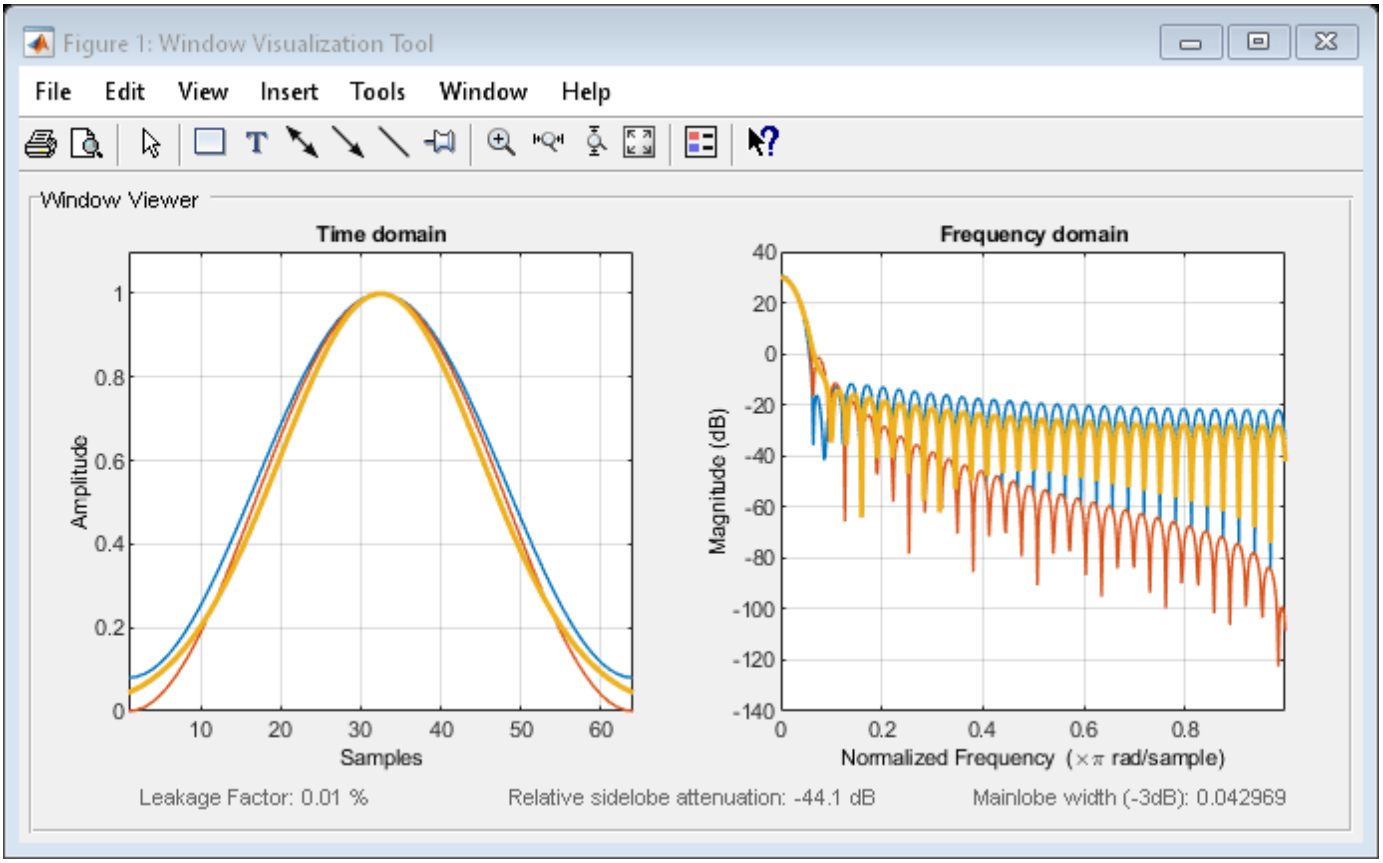
WVTool can be opened programmatically using one of the methods described in “Programmatic Use” on page 1-2783.

Examples

Display and Compare Windows

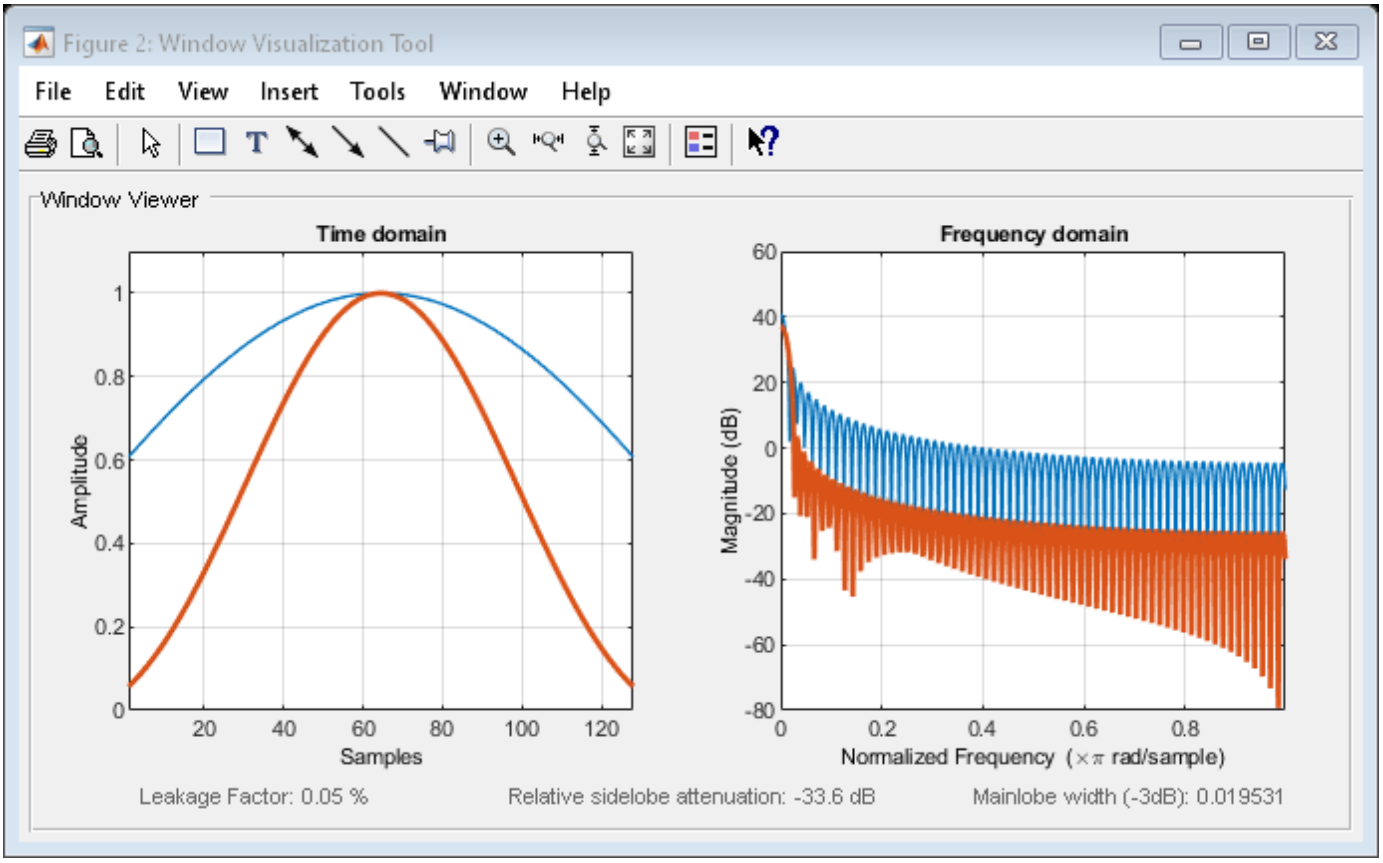
Use `wvtool` to display and compare 64-point Hamming, Hann, and Gaussian windows.

```
wvtool(hamming(64),hann(64),gausswin(64))
```



Compare 128-point Kaiser windows with different values of β.

```
wvtool(kaiser(128,1.5),kaiser(128,4.5))
```



Programmatic Use

`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`.

More About

Note If you launch WVTool from **Filter Designer**, an **Add/Replace** icon, which controls how new windows are added from **Filter Designer**, 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.

See Also

Apps
Filter Designer | Window Designer

Introduced before R2006a

xcorr2

2-D cross-correlation

Syntax

```
c = xcorr2(a,b)
c = xcorr2(a)
```

Description

`c = xcorr2(a,b)` returns the cross-correlation of matrices `a` and `b` with no scaling. `xcorr2` is the two-dimensional version of `xcorr`.

`c = xcorr2(a)` is the autocorrelation matrix of input matrix `a`. This syntax is equivalent to `xcorr2(a,a)`.

Examples

Output Matrix Size and Element Computation

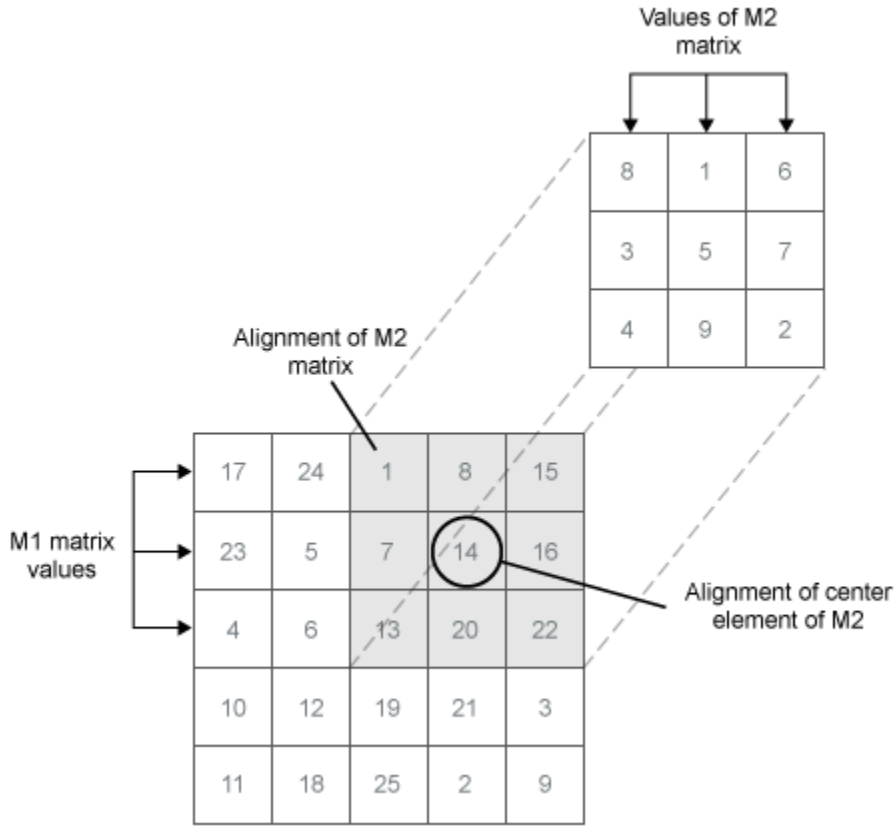
Create two matrices, `M1` and `M2`.

```
M1 = [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];

M2 = [8 1 6;
      3 5 7;
      4 9 2];
```

`M1` is 5-by-5 and `M2` is 3-by-3, so their cross-correlation has size (5+3-1)-by-(5+3-1), or 7-by-7. In terms of lags, the resulting matrix is

$$C = \begin{pmatrix} c_{-2,-2} & c_{-2,-1} & c_{-2,0} & c_{-2,1} & c_{-2,2} & c_{-2,3} & c_{-2,4} \\ c_{-1,-2} & c_{-1,-1} & c_{-1,0} & c_{-1,1} & c_{-1,2} & c_{-1,3} & c_{-1,4} \\ c_{0,-2} & c_{0,-1} & c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} & c_{0,4} \\ c_{1,-2} & c_{1,-1} & c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,-2} & c_{2,-1} & c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,-2} & c_{3,-1} & c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,-2} & c_{4,-1} & c_{4,0} & c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{pmatrix}.$$

As an example, compute the element $c_{0,2}$ (or `C(3,5)` in MATLAB®, since `M2` is 3-by-3). Line up the two matrices so their `(1,1)` elements coincide. This placement corresponds to $c_{0,0}$. To find $c_{0,2}$, slide `M2` two rows to the right.



Now `M2` is on top of the matrix `M1(1:3,3:5)`. Compute the element-by-element products and sum them. The answer should be

$$1 \times 8 + 7 \times 3 + 13 \times 4 + 8 \times 1 + 14 \times 5 + 20 \times 9 + 15 \times 6 + 16 \times 7 + 22 \times 2 = 585 .$$

```
[r2,c2] = size(M2);

CC = sum(sum(M1(0+(1:r2),2+(1:c2)).*M2))

CC = 585
```

Verify the result using `xcorr2`.

```
D = xcorr2(M1,M2);

DD = D(0+r2,2+c2)

DD = 585
```

Two-Dimensional Cross-Correlation of Arbitrary Complex Matrices

Given a matrix X of size $M \times N$ and a matrix $\mathscr{H}$ of size $P \times Q$, their two-dimensional cross-correlation, $C = X \star \mathscr{H}$, is a matrix of size $(M + P - 1) \times (N + Q - 1)$ with elements

$$C(k,l) = \mathrm{Tr}\left\{\widetilde{X}\widetilde{\mathscr{H}}_{kl}^{\dagger}\right\} \qquad \begin{array}{l} 1 \le k \le M+P-1, \\ 1 \le l \le N+Q-1\,. \end{array}$$

Tr is the trace and the dagger denotes Hermitian conjugation. The matrices $\widetilde{X}$ and $\widetilde{\mathscr{H}}_{kl}$ have size $(M + 2(P-1)) \times (N + 2(Q-1))$ and nonzero elements given by

$$\widetilde{X}(m,n) = X(m-P+1, n-Q+1), \qquad \begin{array}{l} P \le m \le M+P-1, \\ Q \le n \le N+Q-1 \end{array}$$

and

$$\widetilde{\mathscr{H}}_{kl}(p,q) = \mathscr{H}(p-k+1, q-l+1), \qquad \begin{array}{l} k \le p \le P+k-1, \\ l \le q \le Q+l-1\,. \end{array}$$

Calling `xcorr2` is equivalent to this procedure for general complex matrices of arbitrary size.

Create two complex matrices, X of size 7×22 and $\mathscr{H}$ of size 6×17.

```
X = randn([7 22])+1j*randn([7 22]);
H = randn([6 17])+1j*randn([6 17]);

[M,N] = size(X);
m = 1:M;
n = 1:N;

[P,Q] = size(H);
p = 1:P;
q = 1:Q;
```

Initialize $\widetilde{X}$ and C.

```
Xt = zeros([M+2*(P-1) N+2*(Q-1)]);
Xt(m+P-1,n+Q-1) = X;
C = zeros([M+P-1 N+Q-1]);
```

Compute the elements of C by looping over k and l. Reset $\widetilde{\mathscr{H}}_{kl}$ to zero at each step. Save time and memory by summing element products instead of multiplying and taking the trace.

```
for k = 1:M+P-1
    for l = 1:N+Q-1
        Hkl = zeros([M+2*(P-1) N+2*(Q-1)]);
        Hkl(p+k-1,q+l-1) = H;
        C(k,l) = sum(sum(Xt.*conj(Hkl)));
    end
end

max(max(abs(C-xcorr2(X,H))))

ans = 1.5139e-14
```

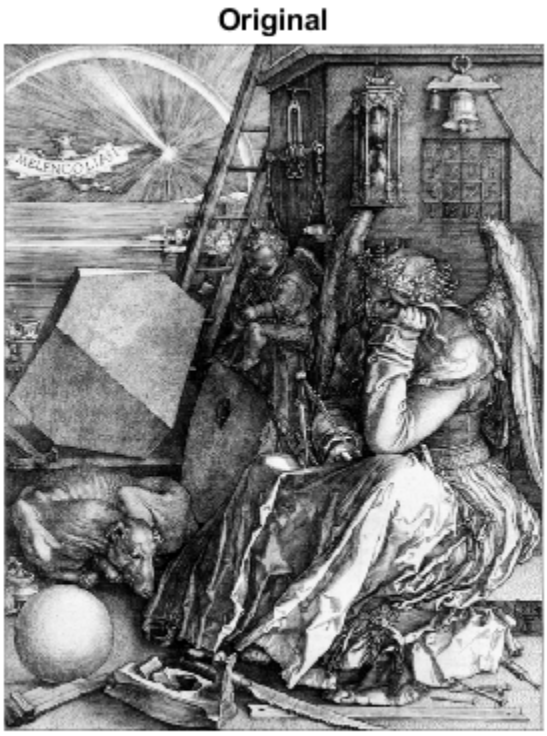
The answer coincides to machine precision with the output of `xcorr2`.

Align Two Images Using Cross-Correlation

Use cross-correlation to find where a section of an image fits in the whole. Cross-correlation enables you to find the regions in which two signals most resemble each other. For two-dimensional signals, like images, use `xcorr2`.

Load a black-and-white test image into the workspace. Display it with `imagesc`.

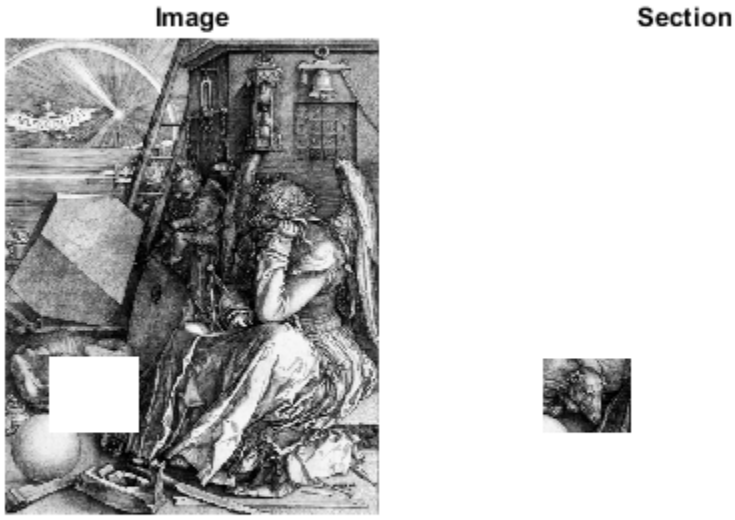
```
load durer
img = X;
White = max(max(img));

imagesc(img)
axis image off
colormap gray
title('Original')
```



Select a rectangular section of the image. Display the larger image with the section missing.

```
x = 435;
X = 535;
szx = x:X;

y = 62;
Y = 182;
szy = y:Y;
```

```
Sect = img(szx,szy);

kimg = img;
kimg(szx,szy) = White;

kumg = White*ones(size(img));
kumg(szx,szy) = Sect;

subplot(1,2,1)
imagesc(kimg)
axis image off
colormap gray
title('Image')

subplot(1,2,2)
imagesc(kumg)
axis image off
colormap gray
title('Section')
```

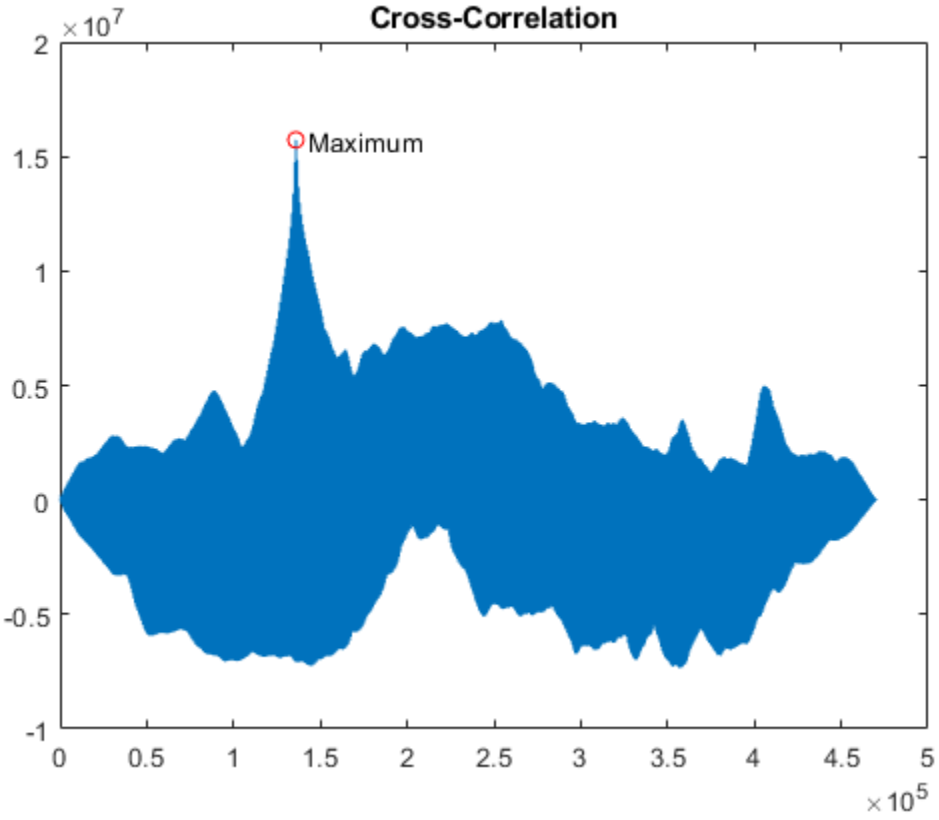


Use `xcorr2` to find where the small image fits in the larger image. Subtract the mean value so that there are roughly equal numbers of negative and positive values.

```
nimg = img-mean(mean(img));
nSec = nimg(szx,szy);

crr = xcorr2(nimg,nSec);
```

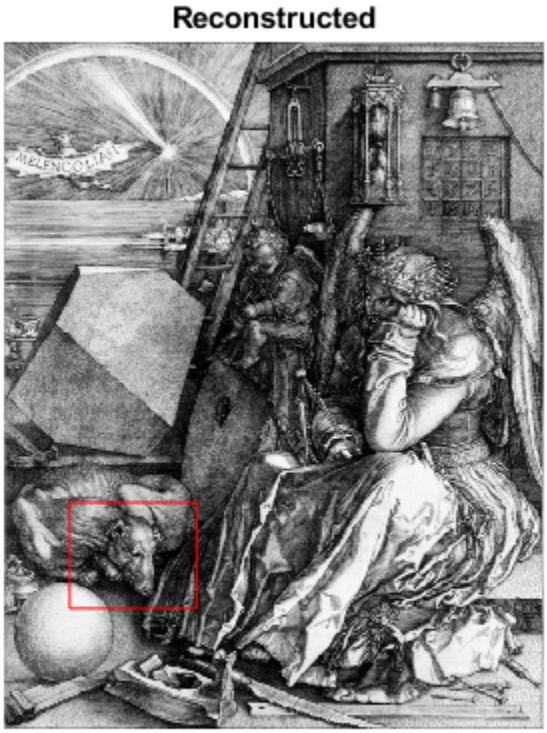
The maximum of the cross-correlation corresponds to the estimated location of the lower-right corner of the section. Use `ind2sub` to convert the one-dimensional location of the maximum to two-dimensional coordinates.

```
[ssr,snd] = max(crr(:));
[ij,ji] = ind2sub(size(crr),snd);

figure
plot(crr(:))
title('Cross-Correlation')
hold on
plot(snd,ssr,'or')
hold off
text(snd*1.05,ssr,'Maximum')
```



Place the smaller image inside the larger image. Rotate the smaller image to comply with the convention that MATLAB® uses to display images. Draw a rectangle around it.

```
img(ij:-1:ij-size(Sect,1)+1,ji:-1:ji-size(Sect,2)+1) = rot90(Sect,2);

imagesc(img)
axis image off
colormap gray
title('Reconstructed')
hold on
plot([y y Y Y y],[x X X x x],'r')
hold off
```



Recovery of Template Shift with Cross-Correlation

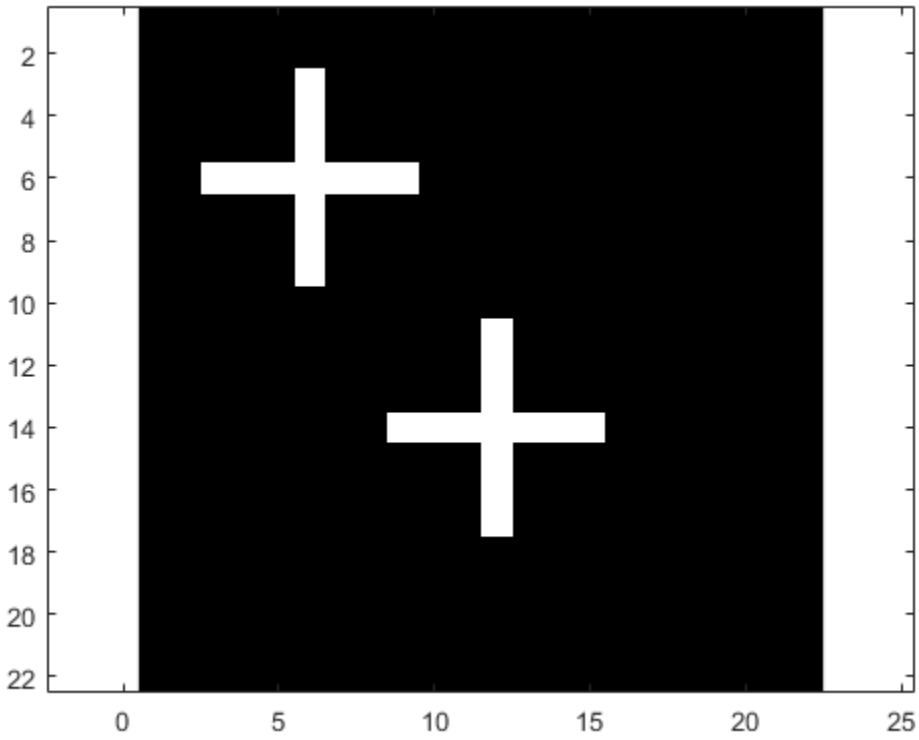
Shift a template by a known amount and recover the shift using cross-correlation.

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 = 0.2*ones(11);
template(6,3:9) = 0.6;
template(3:9,6) = 0.6;
offsetTemplate = 0.2*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)
axis equal
```



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)
```

```
ans = logical
   1
```

The shift obtained from the cross-correlation equals the known template shift in the row and column dimensions.

GPU Acceleration for Cross-Correlation Matrix Computation

This example requires Parallel Computing Toolbox™ software. Refer to “GPU Support by Release” (Parallel Computing Toolbox) to see what GPUs are supported.

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 = 0.2*ones(11);
template(6,3:9) = 0.6;
template(3:9,6) = 0.6;
offsetTemplate = 0.2*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)
```

```
ans = logical
   1
```

Input Arguments

a, b — Input arrays

matrices | `gpuArray` objects

Input arrays, specified as matrices or `gpuArray` objects.

See “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox) and “GPU Support by Release” (Parallel Computing Toolbox) for details on using `xcorr2` with `gpuArray` objects.

Example: `sin(2*pi*(0:9)'/10)*sin(2*pi*(0:13)/20)` specifies a two-dimensional sinusoidal surface.

Example: `gpuArray(sin(2*pi*(0:9)'/10)*sin(2*pi*(0:13)/20))` specifies a two-dimensional sinusoidal surface as a `gpuArray` object.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

c — 2-D cross-correlation or autocorrelation matrix

matrix | `gpuArray` object

2-D cross-correlation or autocorrelation matrix, returned as a matrix or a `gpuArray` object.

More About

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 $N+Q-1$. Its elements are given by

$$C(k,l) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X(m,n)\,\overline{H}(m-k,n-l), \qquad \begin{matrix} -(P-1) \le k \le M-1, \\ -(Q-1) \le l \le N-1, \end{matrix}$$

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 of H.

To cast the indices in MATLAB form, add the size of H: the element $C(k,l)$ corresponds to `C(k+P,l+Q)` in the workspace.

For example, consider this 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. To compute the `C(1,1)` element, shift `H` two rows up 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

$$C(-2,-1) = \sum_{m=0}^{1} \sum_{n=0}^{2} X(m,n)\,\overline{H}(m+2,n+1) = X(0,0)\,\overline{H}(2,1) = 1 \times 6 = 6,$$

with all other terms in the double sum equal to zero.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`conv2` | `filter2` | `xcorr`

Introduced before R2006a

xspectrogram

Cross-spectrogram using short-time Fourier transforms

Syntax

```
s = xspectrogram(x,y)
s = xspectrogram(x,y,window)
s = xspectrogram(x,y,window,noverlap)
s = xspectrogram(x,y,window,noverlap,nfft)

[s,w,t] = xspectrogram( ___ )
[s,f,t] = xspectrogram( ___ ,fs)

[s,w,t] = xspectrogram(x,y,window,noverlap,w)
[s,f,t] = xspectrogram(x,y,window,noverlap,f,fs)

[ ___ ,c] = xspectrogram( ___ )

[ ___ ] = xspectrogram( ___ ,freqrange)
[ ___ ] = xspectrogram( ___ ,Name,Value)

[ ___ ] = xspectrogram( ___ ,spectrumtype)

xspectrogram( ___ )
xspectrogram( ___ ,freqloc)
```

Description

`s = xspectrogram(x,y)` returns the cross-spectrogram of the signals specified by `x` and `y`. The input signals must be vectors with the same number of elements. Each column of `s` contains an estimate of the short-term, time localized frequency content common to `x` and `y`.

`s = xspectrogram(x,y,window)` uses `window` to divide `x` and `y` into segments and perform windowing.

`s = xspectrogram(x,y,window,noverlap)` uses `noverlap` samples of overlap between adjoining segments.

`s = xspectrogram(x,y,window,noverlap,nfft)` uses `nfft` sampling points to calculate the discrete Fourier transform.

`[s,w,t] = xspectrogram( ___ )` returns a vector of normalized frequencies, `w`, and a vector of time instants, `t`, at which the cross-spectrogram is computed. This syntax can include any combination of input arguments from previous syntaxes.

`[s,f,t] = xspectrogram( ___ ,fs)` returns a vector of frequencies, `f`, expressed in terms of `fs`, the sample rate. `fs` must be the sixth input to `xspectrogram`. To input a sample rate and still use the default values of the preceding optional arguments, specify these arguments as empty, `[]`.

`[s,w,t] = xspectrogram(x,y,window,noverlap,w)` returns the cross-spectrogram at the normalized frequencies specified in `w`.

`[s,f,t] = xspectrogram(x,y,window,noverlap,f,fs)` returns the cross-spectrogram at the frequencies specified in `f`.

`[ ___ ,c] = xspectrogram( ___ )` also returns a matrix, `c`, containing an estimate of the time-varying complex cross-spectrum of the input signals. The cross-spectrogram, `s`, is the magnitude of `c`.

`[ ___ ] = xspectrogram( ___ ,freqrange)` returns the cross-spectrogram over the frequency range specified by `freqrange`. Valid options for `freqrange` are `'onesided'`, `'twosided'`, and `'centered'`.

`[ ___ ] = xspectrogram( ___ ,Name,Value)` specifies additional options using name-value arguments. Options include the minimum threshold and output time dimension.

`[ ___ ] = xspectrogram( ___ ,spectrumtype)` returns short-term cross power spectral density estimates if `spectrumtype` is specified as `'psd'` and returns short-term cross power spectrum estimates if `spectrumtype` is specified as `'power'`.

`xspectrogram( ___ )` with no output arguments plots the cross-spectrogram in the current figure window.

`xspectrogram( ___ ,freqloc)` specifies the axis on which to plot the frequency. Specify `freqloc` as either `'xaxis'` or `'yaxis'`.

Examples

Cross-Spectrogram of Linear Chirps

Generate two linear chirps sampled at 1 MHz for 10 milliseconds.

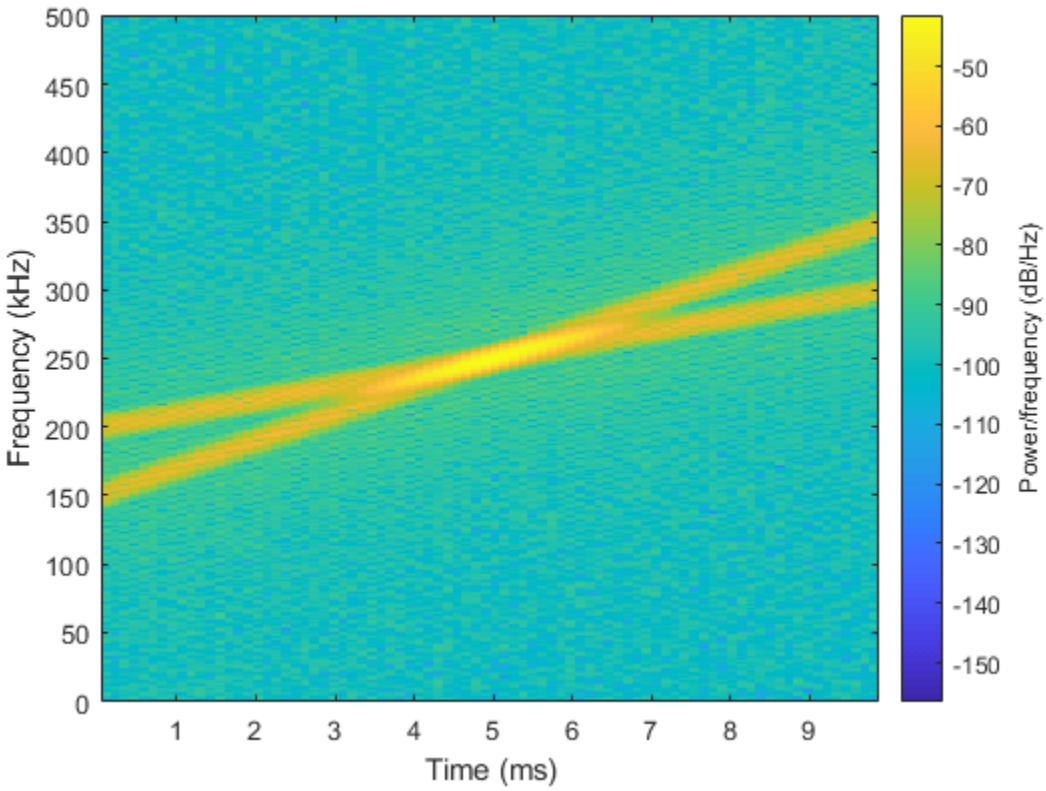
- The first chirp has an initial frequency of 150 kHz that increases to 350 kHz by the end of the measurement.
- The second chirp has an initial frequency of 200 kHz that increases to 300 kHz by the end of the measurement.

Add white Gaussian noise such that the signal-to-noise ratio is 40 dB.

```
nSamp = 10000;
Fs = 1000e3;
SNR = 40;
t = (0:nSamp-1)'/Fs;

x1 = chirp(t,150e3,t(end),350e3);
x1 = x1+randn(size(x1))*std(x1)/db2mag(SNR);

x2 = chirp(t,200e3,t(end),300e3);
x2 = x2+randn(size(x2))*std(x2)/db2mag(SNR);
```

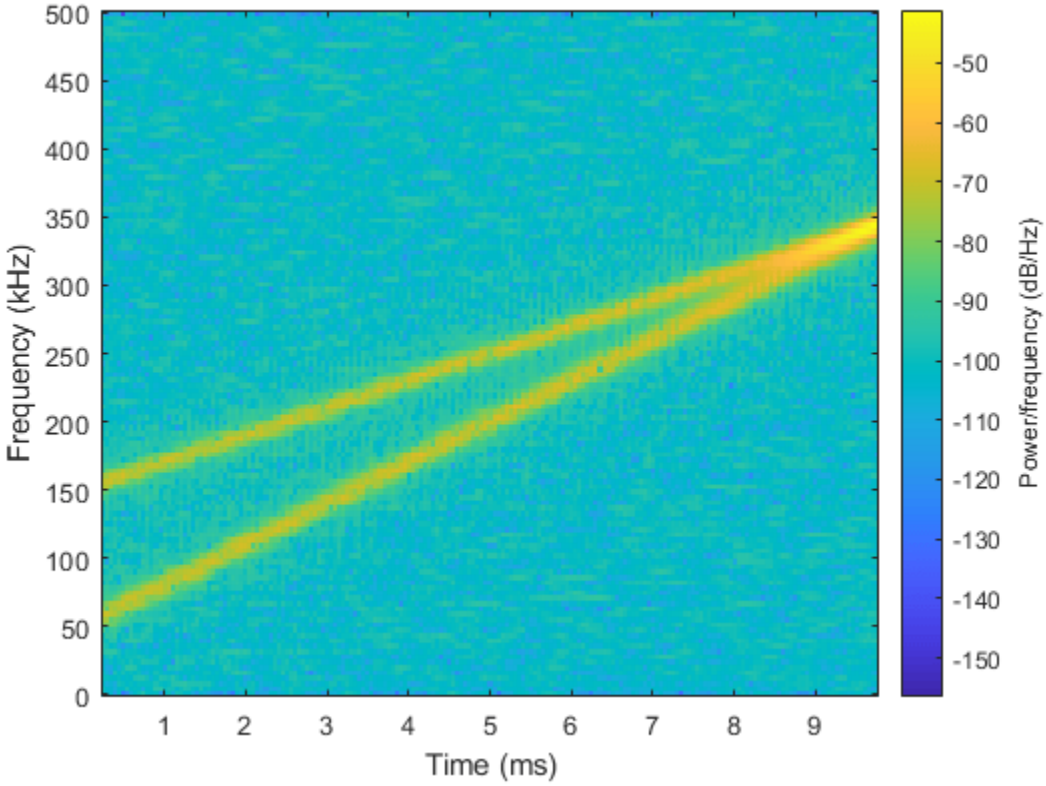
Compute and plot the cross-spectrogram of the two chirps. Divide the signals into 200-sample segments and window each segment with a Hamming window. Specify 80 samples of overlap between adjoining segments and a DFT length of 1024 samples.

```
xspectrogram(x1,x2,hamming(200),80,1024,Fs,'yaxis')
```



Modify the second chirp so that the frequency rises from 50 kHz to 350 kHz during the measurement. Use a 500-sample Kaiser window with shape factor $\beta = 5$ to window the segments. Specify 450 samples of overlap and a DFT length of 256. Compute and plot the cross-spectrogram.

```
x2 = chirp(t,50e3,t(end),350e3);
x2 = x2+randn(size(x2))*std(x2)/db2mag(SNR);

xspectrogram(x1,x2,kaiser(500,5),450,256,Fs,'yaxis')
```



In both cases, the function highlights the frequency content that the two signals have in common.

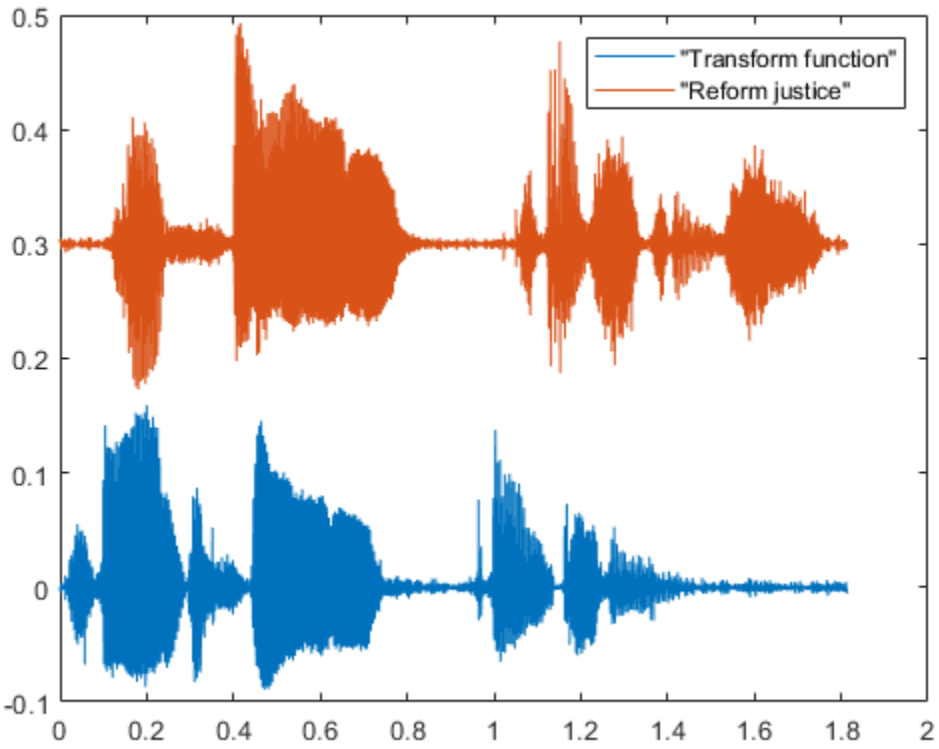
Cross-Spectrogram of Speech Signals

Load a file containing two speech signals sampled at 44,100 Hz.

- The first signal is a recording of a female voice saying "transform function."
- The second signal is a recording of the same female voice saying "reform justice."

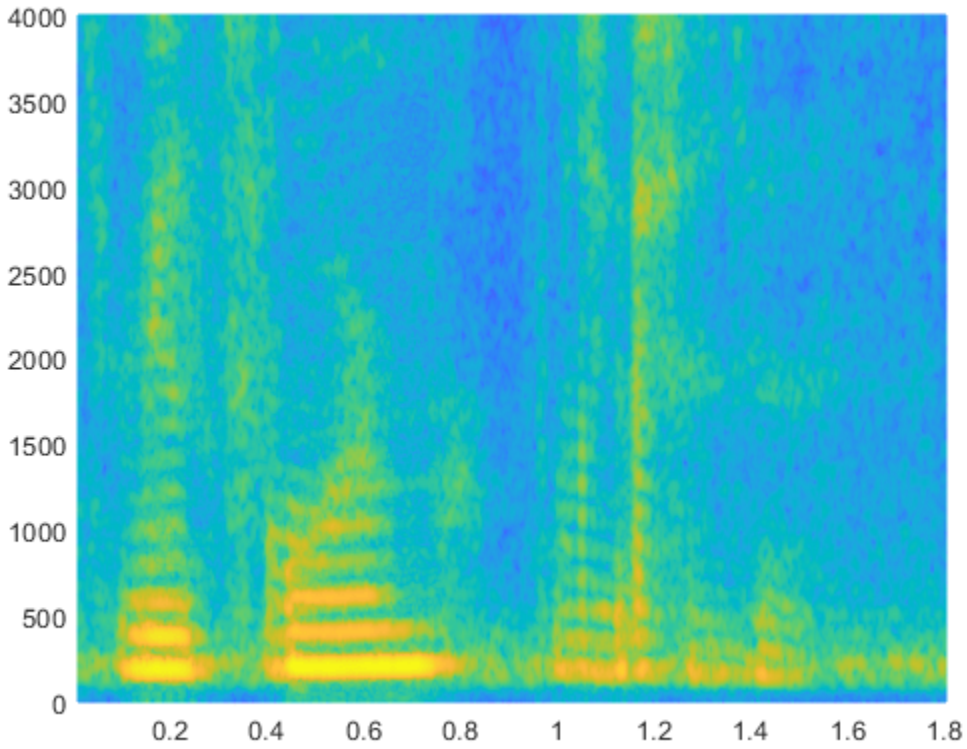
Plot the two signals. Offset the second signal vertically so both are visible.

```
load('voice.mat')

% To hear, type soundsc(transform,fs),pause(2),soundsc(reform,fs)

t = (0:length(reform)-1)/fs;

plot(t,transform,t,reform+0.3)
legend('"Transform function"','"Reform justice"')
```



Compute the cross-spectrogram of the two signals. Divide the signals into 1000-sample segments and window them with a Hamming window. Specify 800 samples of overlap between adjoining segments. Include only frequencies up to 4 kHz.

```
nwin = 1000;
nvlp = 800;
fint = 0:4000;

[s,f,t] = xspectrogram(transform,reform,hamming(nwin),nvlp,fint,fs);

mesh(t,f,20*log10(s))
view(2)
axis tight
```



The cross-spectrogram highlights the time intervals where the signals have more frequency content in common. The syllable "form" is particularly noticeable.

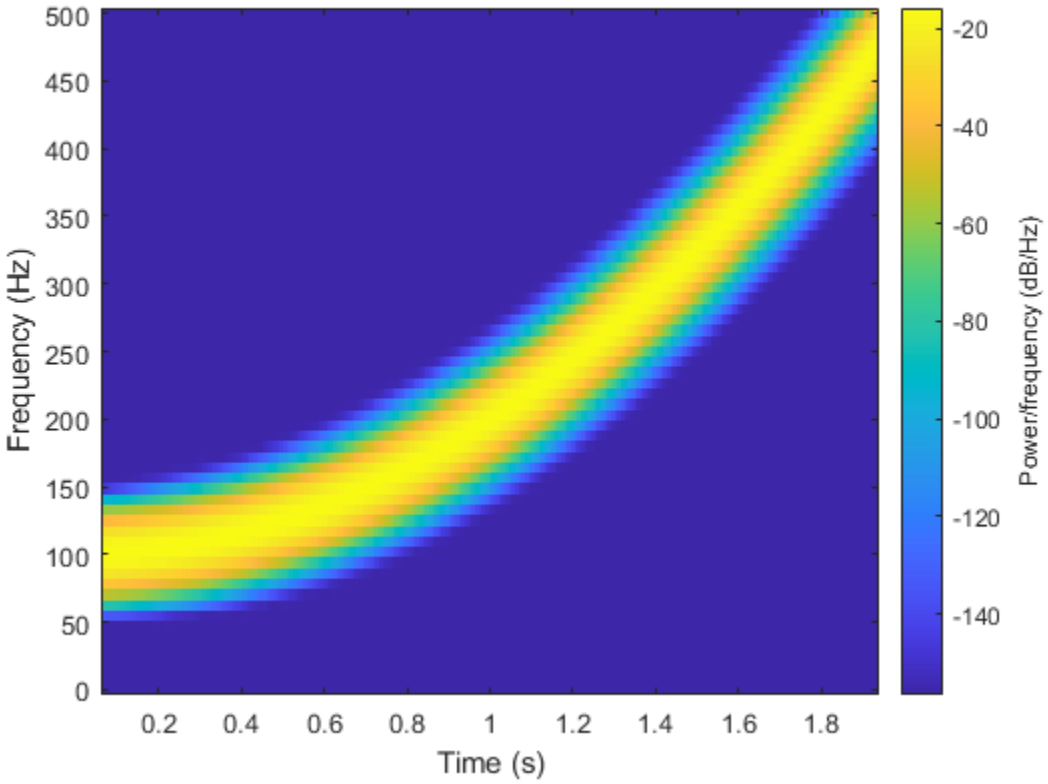
Phase Shift Between Two Quadratic Chirps

Generate two quadratic chirps, each sampled at 1 kHz for 2 seconds. Both chirps have an initial frequency of 100 Hz that increases to 200 Hz midway through the measurement. The second chirp has a phase difference of 23° compared to the first.

```
fs = 1e3;
t = 0:1/fs:2;

y1 = chirp(t,100,1,200,'quadratic',0);
y2 = chirp(t,100,1,200,'quadratic',23);
```

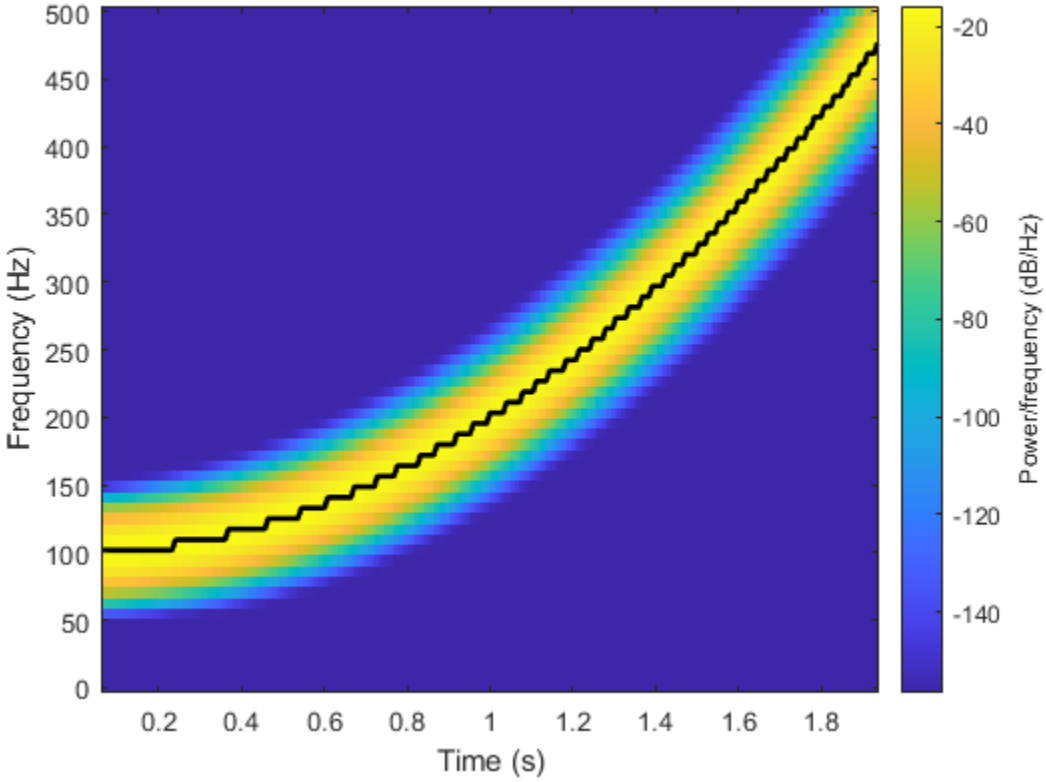
Compute the complex cross-spectrogram of the chirps to extract the phase shift between them. Divide the signals into 128-sample segments. Specify 120 samples of overlap between adjoining segments. Window each segment using a Kaiser window with shape factor $\beta = 18$ and specify a DFT length of 128 samples. Use the plotting functionality of `xspectrogram` to display the cross-spectrogram.

```
[~,f,xt,c] = xspectrogram(y1,y2,kaiser(128,18),120,128,fs);

xspectrogram(y1,y2,kaiser(128,18),120,128,fs,'yaxis')
```



Extract and display the maximum-energy time-frequency ridge of the cross-spectrogram.

```
[tfr,~,lridge] = tfridge(c,f);

hold on
plot(xt,tfr,'k','linewidth',2)
hold off
```



The phase shift is the ratio of imaginary part to real part of the time-varying cross-spectrum along the ridge. Compute the phase shift and express it in degrees. Display its mean value.

```
pshft = angle(c(lridge))*180/pi;
```

```
mean(pshft)
```

```
ans = -23.0000
```

Cross-Spectrogram of Complex Signals

Generate two signals, each sampled at 3 kHz for 1 second. The first signal is a quadratic chirp whose frequency increases from 300 Hz to 1300 Hz during the measurement. The chirp is embedded in white Gaussian noise. The second signal, also embedded in white noise, is a chirp with sinusoidally varying frequency content.

```
fs = 3000;
t = 0:1/fs:1-1/fs;
```

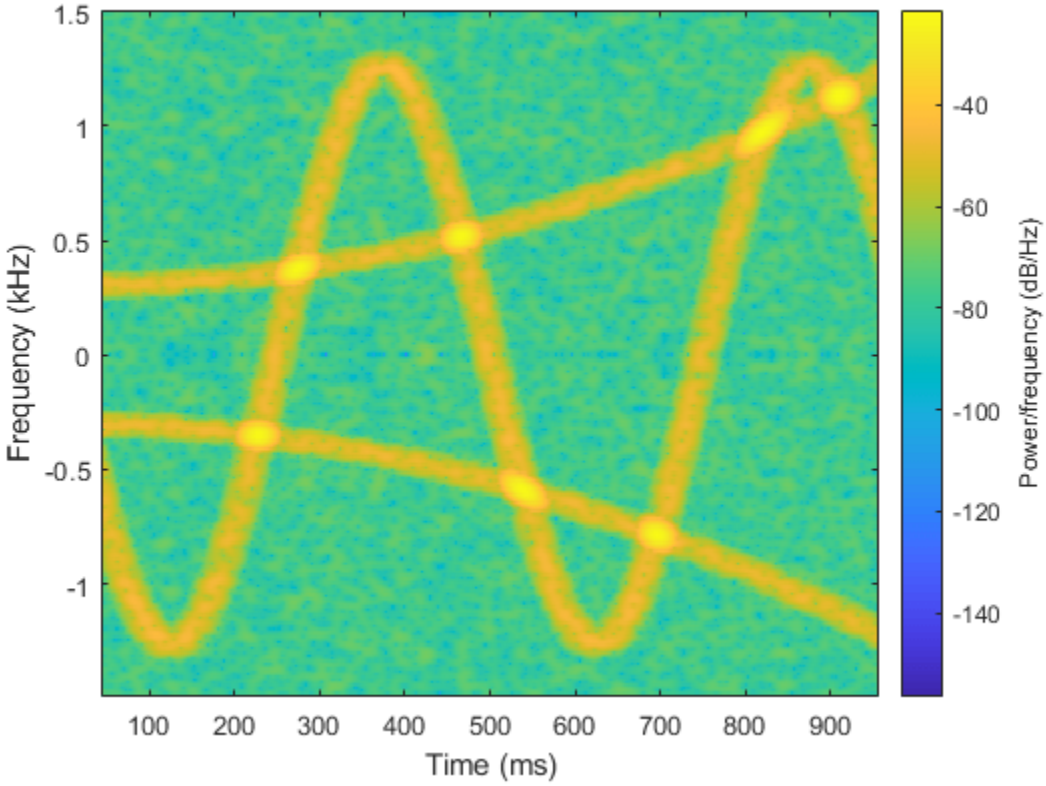
```
x1 = chirp(t,300,t(end),1300,'quadratic')+randn(size(t))/100;
```

```
x2 = exp(2j*pi*100*cos(2*pi*2*t))+randn(size(t))/100;
```

Compute and plot the cross-spectrogram of the two signals. Divide the signals into 256-sample segments with 255 samples of overlap between adjoining segments. Use a Kaiser window with shape

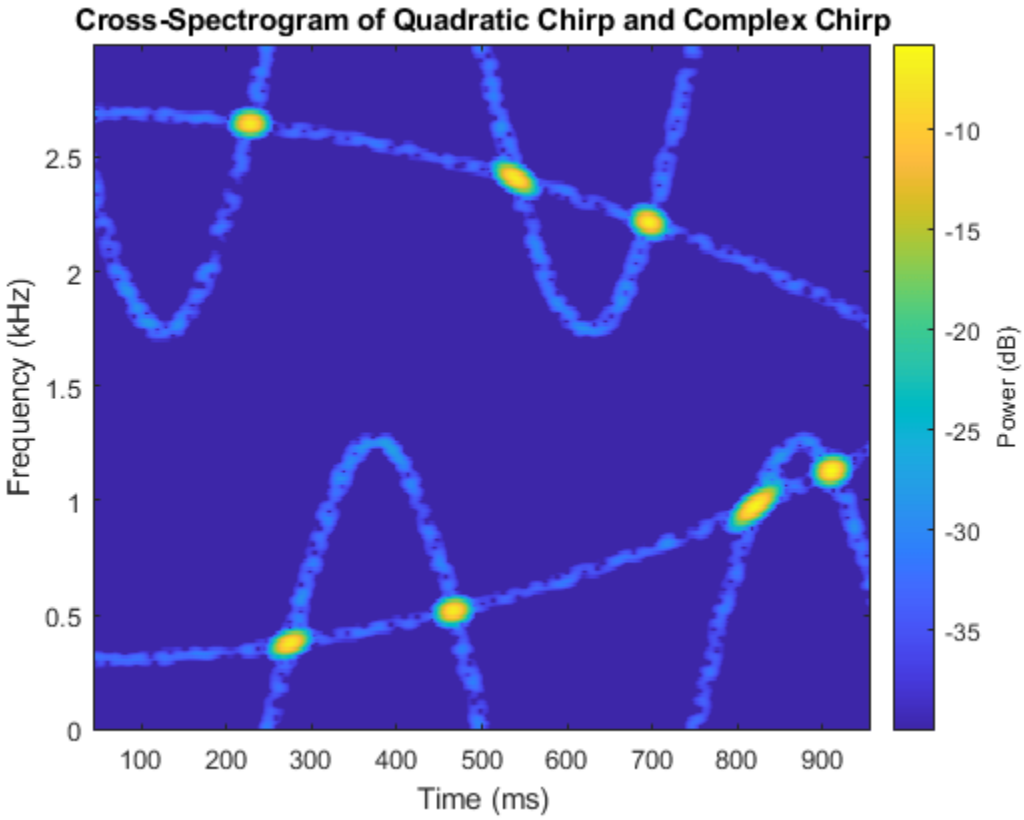
factor $\beta = 30$ to window the segments. Use the default number of DFT points. Center the cross-spectrogram at zero frequency.

```
nwin = 256;

xspectrogram(x1,x2,kaiser(nwin,30),nwin-1,[],fs,'centered','yaxis')
```



Compute the power spectrum instead of the power spectral density. Set to zero the values smaller than –40 dB. Center the plot at the Nyquist frequency.

```
xspectrogram(x1,x2,kaiser(nwin,30),nwin-1,[],fs, ...
    'power','MinThreshold',-40,'yaxis')
title('Cross-Spectrogram of Quadratic Chirp and Complex Chirp')
```



The thresholding further highlights the regions of common frequency.

Cross-Spectrogram of Two Sequences

Compute and plot the cross-spectrogram of two sequences.

Specify each sequence to be 4096 samples long.

```
N = 4096;
```

To create the first sequence, generate a convex quadratic chirp embedded in white Gaussian noise and bandpass filter it.

- The chirp has an initial normalized frequency of 0.1π that increases to 0.8π by the end of the measurement.
- The 16th-order filter passes normalized frequencies between 0.2π and 0.4π rad/sample and has a stopband attenuation of 60 dB.

```
rx = chirp(0:N-1,0.1/2,N,0.8/2,'quadratic',[],'convex')'+randn(N,1)/100;
dx = designfilt('bandpassiir','FilterOrder',16, ...
    'StopbandFrequency1',0.2,'StopbandFrequency2',0.4, ...
    'StopbandAttenuation',60);
x = filter(dx,rx);
```

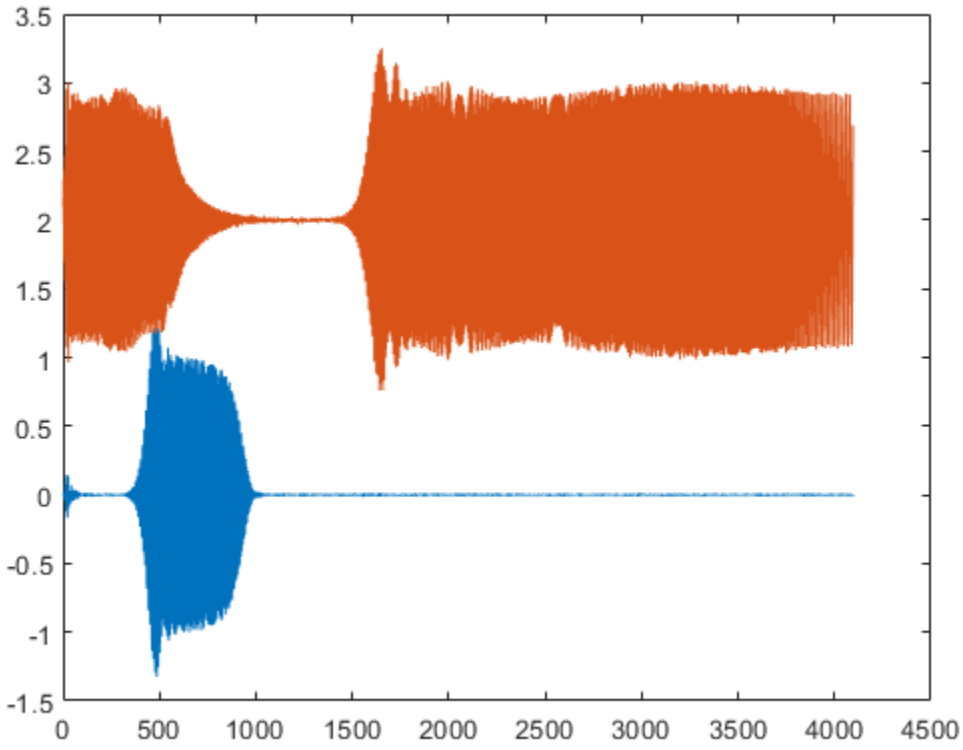
To create the second sequence, generate a linear chirp embedded in white Gaussian noise and bandstop filter it.

- The chirp has an initial normalized frequency of 0.9π that decreases to 0.1π by the end of the measurement.
- The 16th-order filter stops normalized frequencies between 0.6π and 0.8π rad/sample and has a passband ripple of 1 dB.

```
ry = chirp(0:N-1,0.9/2,N,0.1/2)'+randn(N,1)/100;
dy = designfilt('bandstopiir','FilterOrder',16, ...
    'PassbandFrequency1',0.6,'PassbandFrequency2',0.8, ...
    'PassbandRipple',1);
y = filter(dy,ry);
```

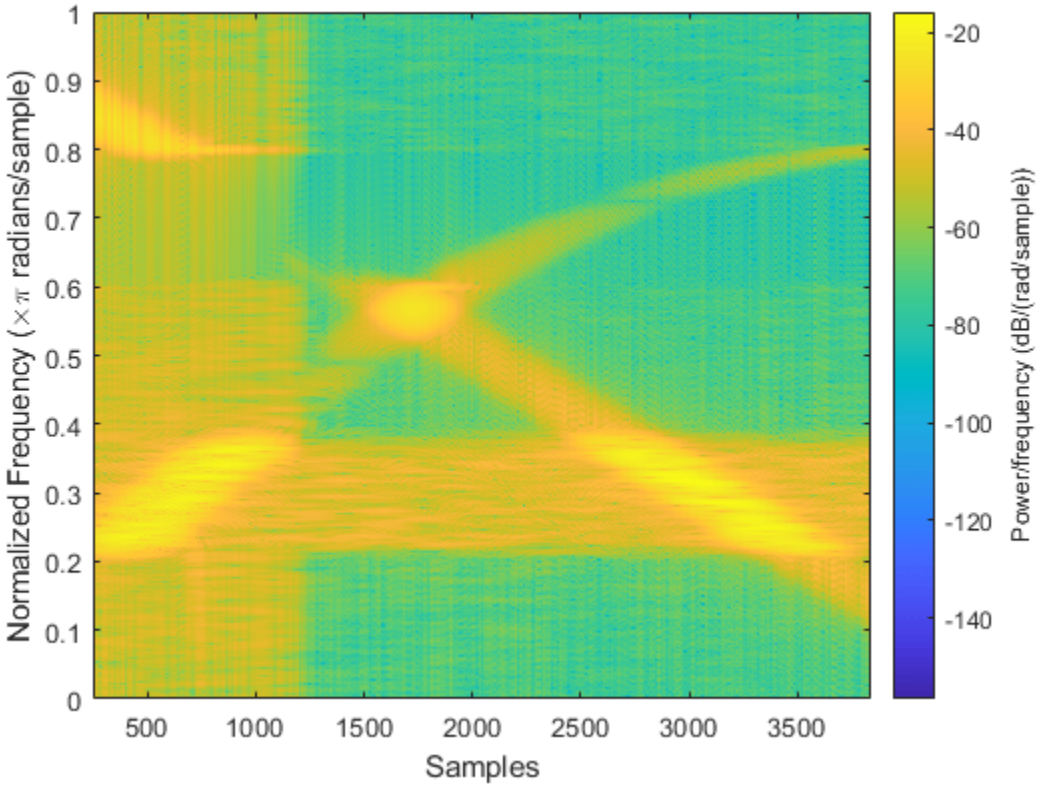
Plot the two sequences. Offset the second sequence vertically so both are visible.

```
plot([x y+2])
```



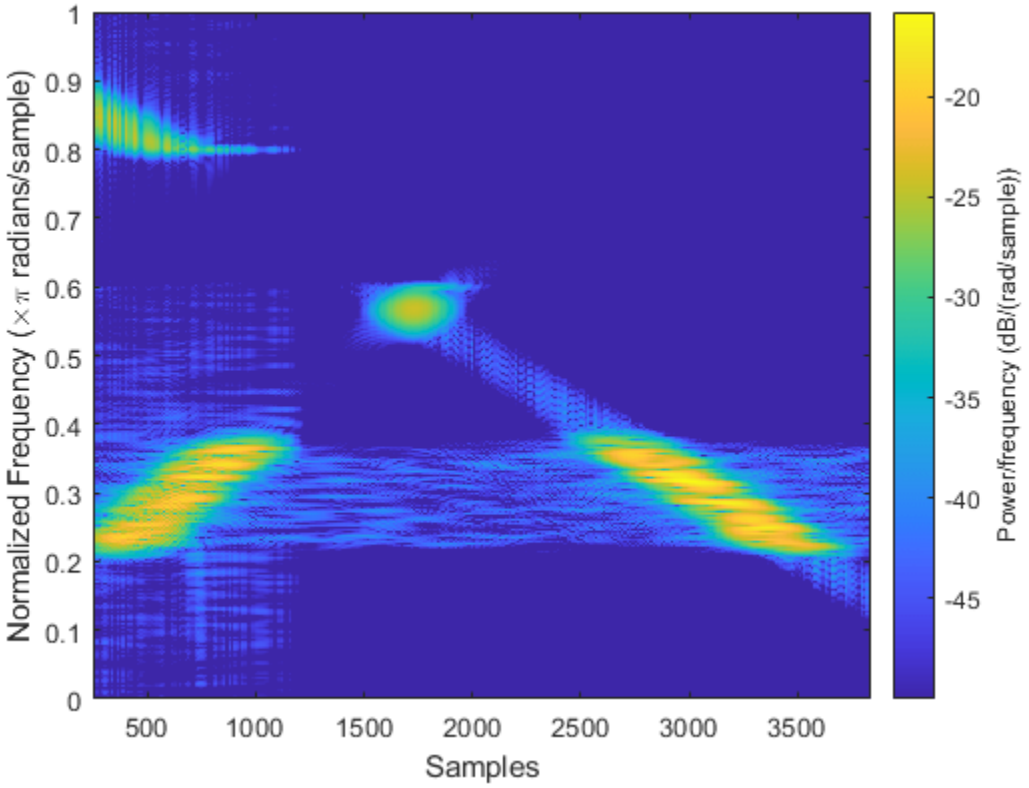
Compute and plot the cross-spectrogram of `x` and `y`. Use a 512-sample Hamming window. Specify 500 samples of overlap between adjoining segments and 2048 DFT points.

```
xspectrogram(x,y,hamming(512),500,2048,'yaxis')
```



Set to zero the cross-spectrogram values smaller than –50 dB.

```
xspectrogram(x,y,hamming(512),500,2048,'MinThreshold',-50,'yaxis')
```



The spectrogram shows the frequency regions that are enhanced or suppressed by the filters.

Input Arguments

x, y — Input signals

vectors

Input signals, specified as vectors.

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 signals into segments.

- If `window` is an integer, then `xspectrogram` divides `x` and `y` into segments of length `window` and windows each segment with a Hamming window of that length.
- If `window` is a vector, then `xspectrogram` divides `x` and `y` into segments of the same length as the vector and windows each segment using `window`.

If the input signals cannot be divided exactly into an integer number of segments with `noverlap` overlapping samples, then they are truncated accordingly.

If you specify `window` as empty, then `xspectrogram` uses a Hamming window such that `x` and `y` are divided into eight segments 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 `xspectrogram` uses a number that produces 50% overlap between segments. If the segment length is unspecified, the function sets `noverlap` to $\lfloor N/4.5 \rfloor$, where N is the length of the input signals.

Data Types: `double` | `single`

`nfft` — Number of DFT points
positive integer | `[]`

Number of DFT points, specified as a positive integer scalar. If you specify `nfft` as empty, then `xspectrogram` sets the DFT length to $\max(256, 2^p)$, where $p = \lceil \log_2 N_w \rceil$ and

- N_w = `window` if `window` is a scalar.
- N_w = `length(window)` if `window` is a vector.

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` — Frequencies
vector

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, then the sample rate is in Hz.

Data Types: `double` | `single`

`freqrange` — Frequency range for cross-spectrum estimate
`'onesided'` | `'twosided'` | `'centered'`

Frequency range for the cross-spectrum estimate, specified as `'onesided'`, `'twosided'`, or `'centered'`. For real-valued signals, the default is `'onesided'`. For complex-valued signals, the default is `'twosided'`, and specifying `'onesided'` results in an error.

- `'onesided'` — Returns the one-sided cross-spectrogram of a real input signal. If `nfft` is even, then `s` has `nfft`/2 + 1 rows and is computed over the interval [0, π] rad/sample. If `nfft` is odd, then `s` has (`nfft` + 1)/2 rows and the interval is [0, π) 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 cross-spectrogram of a real or complex signal. `s` has `nfft` rows and is computed over the interval [0, 2π) rad/sample. If you specify `fs`, then the interval is [0, `fs`) cycles/unit time.
- `'centered'` — Returns the centered two-sided cross-spectrogram for a real or complex signal. `s` has `nfft` rows. If `nfft` is even, then `s` is computed over the interval (–π, π] rad/sample. If `nfft` is odd, then `s` is computed over (–π, π) 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.

`spectrumtype` — Cross power spectrum scaling
`'psd'` (default) | `'power'`

Cross power spectrum scaling, specified as `'psd'` or `'power'`.

- Omitting `spectrumtype`, or specifying `'psd'`, returns the cross power spectral density.
- Specifying `'power'` scales each estimate of the cross power spectral density by the resolution bandwidth, which depends on the equivalent noise bandwidth of the window and the segment duration. The result is an estimate of the power at each frequency.

`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 `xspectrogram` with output arguments.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `xspectrogram(x,100,'OutputTimeDimension','downrows')` divides `x` and `y` into segments of length 100 and windows each segment with a Hamming window of that length. The output of the spectrogram has time dimension down the rows.

`MinThreshold` — Threshold
`-Inf` (default) | real scalar

Threshold, specified as a real scalar expressed in decibels. `xspectrogram` sets to zero those elements of `s` such that $10 \log_{10}(s) \le$ `thresh`.

`OutputTimeDimension` — Output time dimension
`acrosscolumns` (default) | `downrows`

Output time dimension, specified as `acrosscolumns` or `downrows`. Set this value to `downrows`, if you want the time dimension of `s`, `ps`, `fc`, and `tc` down the rows and the frequency dimension along the columns. Set this value to `acrosscolumns`, if you want the time dimension of `s`, `ps`, `fc`, and `tc` across the columns and frequency dimension along the rows. This input is ignored if the function is called without output arguments.

Output Arguments

`s` — Cross-spectrogram
matrix

Cross-spectrogram, returned as a matrix. Time increases across the columns of `s` and frequency increases down the rows, starting from zero.

- If the input signals `x` and `y` are of length N, then `s` has k columns, where:
  - $k = \lfloor (N - \text{noverlap})/(\text{window} - \text{noverlap}) \rfloor$ if `window` is a scalar.
  - $k = \lfloor (N - \text{noverlap})/(\text{length(window)} - \text{noverlap}) \rfloor$ if `window` is a vector.
- If the input signals are real and `nfft` is even, then `s` has (`nfft`/2 + 1) rows.
- If the input signals are real and `nfft` is odd, then `s` has (`nfft` + 1)/2 rows.
- If the input signals are 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 segment specified using `window`.

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`

c — Time-varying complex cross-spectrum
matrix

Time-varying complex cross-spectrum, returned as a matrix. The cross-spectrogram, `s`, is the magnitude of `c`.

Data Types: `double` | `single`

References

[1] Mitra, Sanjit K. *Digital Signal Processing: A Computer-Based Approach*. 2nd Ed. New York: McGraw-Hill, 2001.

[2] Oppenheim, Alan V., and Ronald W. Schafer, with John R. Buck. *Discrete-Time Signal Processing*. 2nd Ed. Upper Saddle River, NJ: Prentice Hall, 1999.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Arguments specified using name value pairs must be compile time constants.
- Window must be double precision.

Thread-Based Environment

Run code in the background using MATLAB® `backgroundPool` or accelerate code with Parallel Computing Toolbox™ `ThreadPool`.

Usage notes and limitations:

- The syntax with no output arguments is not supported.
- The frequency vector must be uniformly spaced.

For more information, see “Run MATLAB Functions in Thread-Based Environment”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The window length must not be greater than `nfft` or the length of `f`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`cpsd` | `mscohere` | `spectrogram`

Topics

“Time-Frequency Gallery”

Introduced in R2017a

xwvd

Cross Wigner-Ville distribution and cross smoothed pseudo Wigner-Ville distribution

Syntax

```
d = xwvd(x,y)
d = xwvd(x,y,fs)
d = xwvd(x,y,ts)

d = xwvd(___,'smoothedPseudo')
d = xwvd(___,'smoothedPseudo',twin,fwin)
d = xwvd(___,'smoothedPseudo','NumFrequencyPoints',nf)

d = xwvd(___,'MinThreshold',thresh)

[d,f,t] = xwvd(___)

xwvd(___)
```

Description

`d = xwvd(x,y)` returns the cross Wigner-Ville distribution of `x` and `y`.

`d = xwvd(x,y,fs)` returns the cross Wigner-Ville distribution when `x` and `y` are sampled at a rate `fs`.

`d = xwvd(x,y,ts)` returns the cross Wigner-Ville distribution when `x` and `y` are sampled with a time interval `ts` between samples.

`d = xwvd(___,'smoothedPseudo')` returns the cross smoothed pseudo Wigner-Ville distribution of `x` and `y`. The function uses the length of the input signals to choose the lengths of the windows used for time and frequency smoothing. This syntax can include any combination of input arguments from previous syntaxes.

`d = xwvd(___,'smoothedPseudo',twin,fwin)` specifies the time window, `twin`, and the frequency window, `fwin`, used for smoothing. To use the default window for either time or frequency smoothing, specify the corresponding argument as empty, `[]`.

`d = xwvd(___,'smoothedPseudo','NumFrequencyPoints',nf)` computes the cross smoothed pseudo Wigner-Ville distribution using `nf` frequency points. You can specify `twin` and `fwin` in this syntax, or you can omit them.

`d = xwvd(___,'MinThreshold',thresh)` sets to zero those elements of `d` whose amplitude is less than `thresh`. This syntax applies to both the cross Wigner-Ville distribution and the cross smoothed pseudo Wigner-Ville distribution.

`[d,f,t] = xwvd(___)` also returns a vector of frequencies, `f`, and a vector of times, `t`, at which `d` is computed.

`xwvd(___)` with no output arguments plots the real part of the cross Wigner-Ville or cross smoothed pseudo Wigner-Ville distribution in the current figure.

Examples

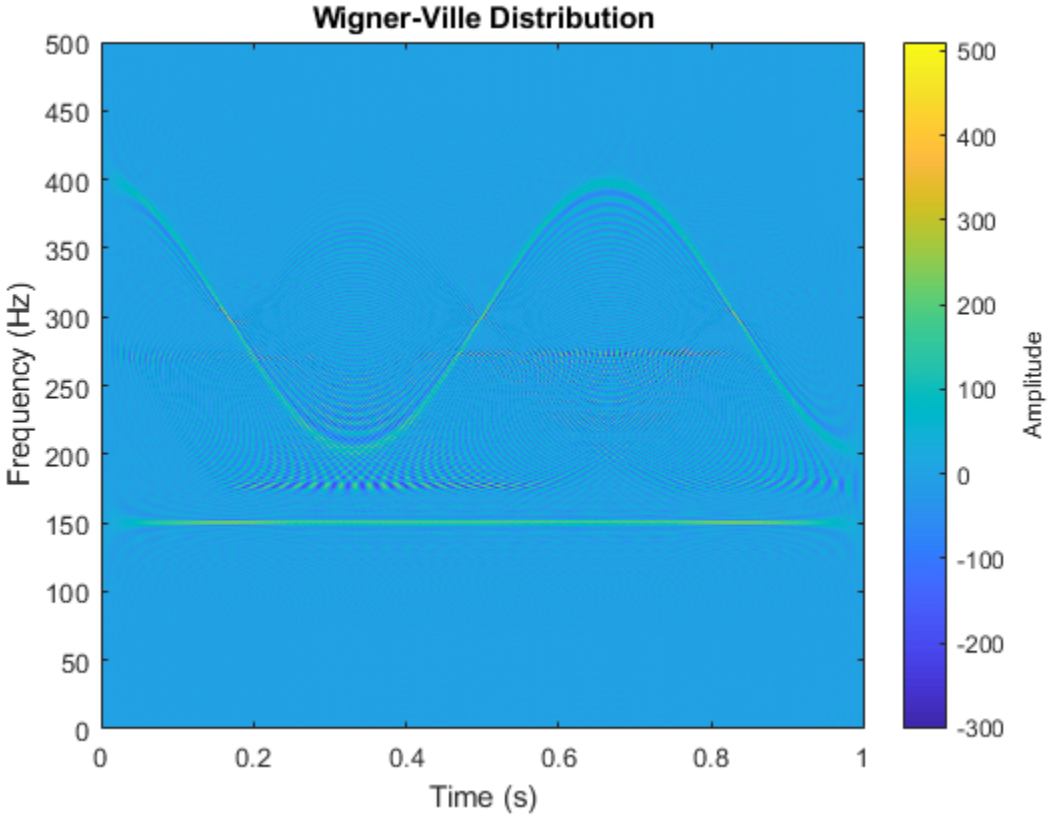
Cross Wigner-Ville Distribution of Signals

Generate two signals sampled at 1 kHz for 1 second and embedded in white noise. One signal is a sinusoid of frequency 150 Hz. The other signal is a chirp whose frequency varies sinusoidally between 200 Hz and 400 Hz. The noise has a variance of 0.1^2.

```
fs = 1000;
t = (0:1/fs:1)';

x = cos(2*pi*t*150) + 0.1*randn(size(t));
y = vco(cos(3*pi*t),[200 400],fs) + 0.1*randn(size(t));
```

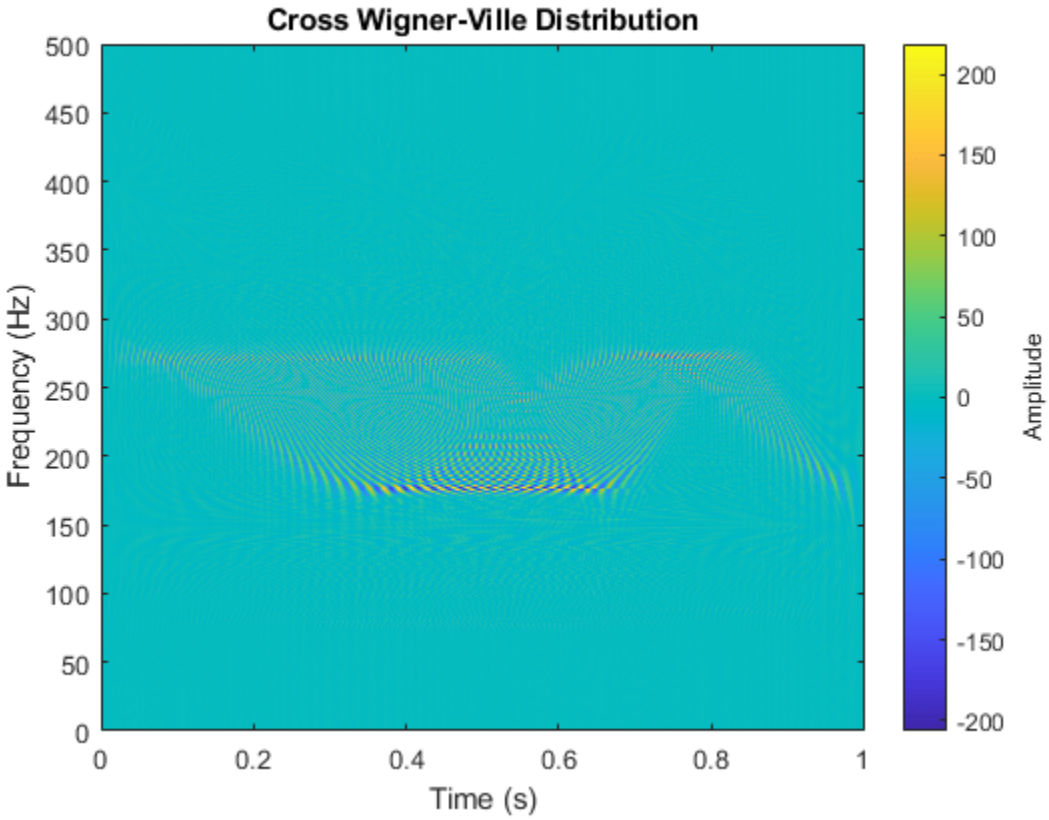
Compute the Wigner-Ville distribution of the sum of the signals.

```
wvd(x+y,fs)
```



Compute and plot the cross Wigner-Ville distribution of the signals. The cross-distribution corresponds to the cross-terms of the Wigner-Ville distribution.

```
xwvd(x,y,fs)
```



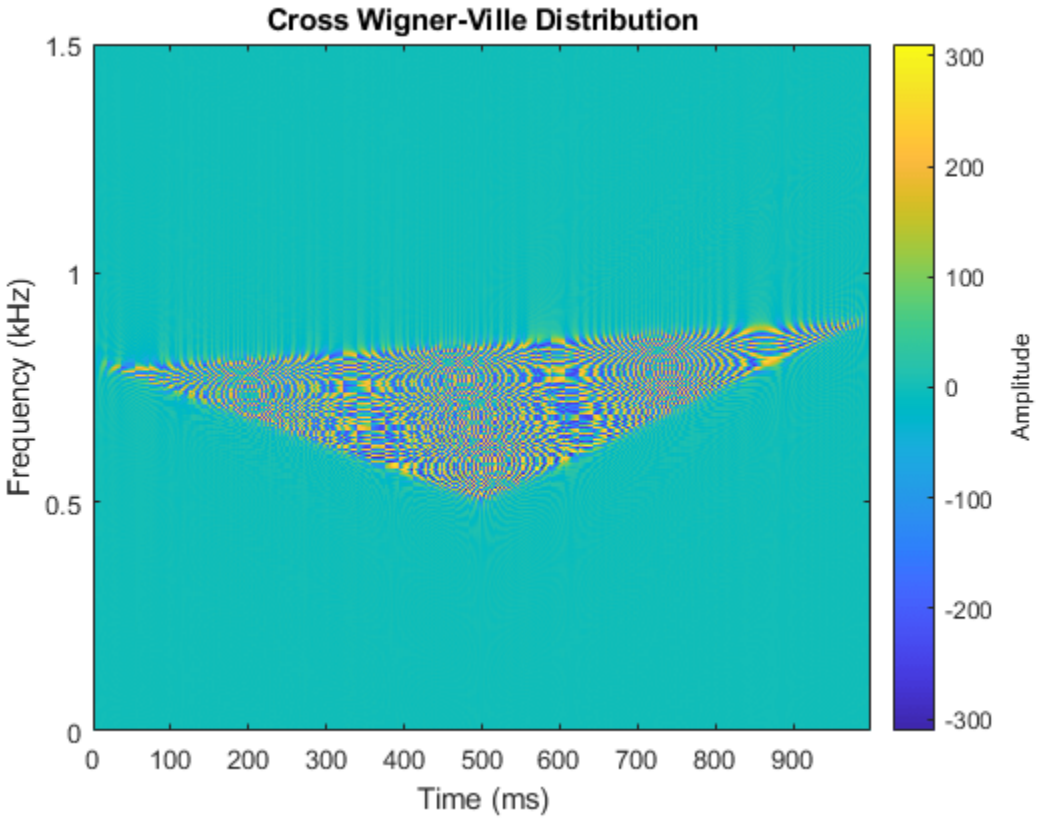
Cross Wigner-Ville Distribution of Chirps

Generate a two-channel signal that consists of two chirps. The signal is sampled at 3 kHz for one second. The first chirp has an initial frequency of 400 Hz and reaches 800 Hz at the end of the sampling. The second chirp starts at 500 Hz and reaches 1000 Hz at the end. The second chirp has twice the amplitude of the first chirp.

```
fs = 3000;
t = (0:1/fs:1-1/fs)';

x1 = chirp(t,1400,t(end),800);
x2 = 2*chirp(t,200,t(end),1000);
```

Store the signal as a timetable. Compute and plot the cross Wigner-Ville distribution of the two channels.

```
xt = timetable(seconds(t),x1,x2);

xwvd(xt(:,1),xt(:,2))
```



Use Cross Wigner-Ville Distribution to Estimate Instantaneous Frequency

Compute the instantaneous frequency of a signal by using a known reference signal and the cross Wigner-Ville distribution.

Create a reference signal consisting of a Gaussian atom sampled at 1 kHz for 1 second. A Gaussian atom is a sinusoid modulated by a Gaussian. Specify a sinusoid frequency of 50 Hz. The Gaussian is centered at 64 milliseconds and has a variance of 0.01^2.

```
fs = 1e3;
t = (0:1/fs:1-1/fs)';

mu = 0.064;
sigma = 0.01;
fsin = 50;

xr = exp(-(t-mu).^2/(2*sigma^2)).*sin(2*pi*fsin*t);
```

Create the "unknown" signal to analyze, consisting of a chirp. The signal starts suddenly at 0.4 second and ends suddenly half a second later. In that lapse, the frequency of the chirp decreases linearly from 400 Hz to 100 Hz.

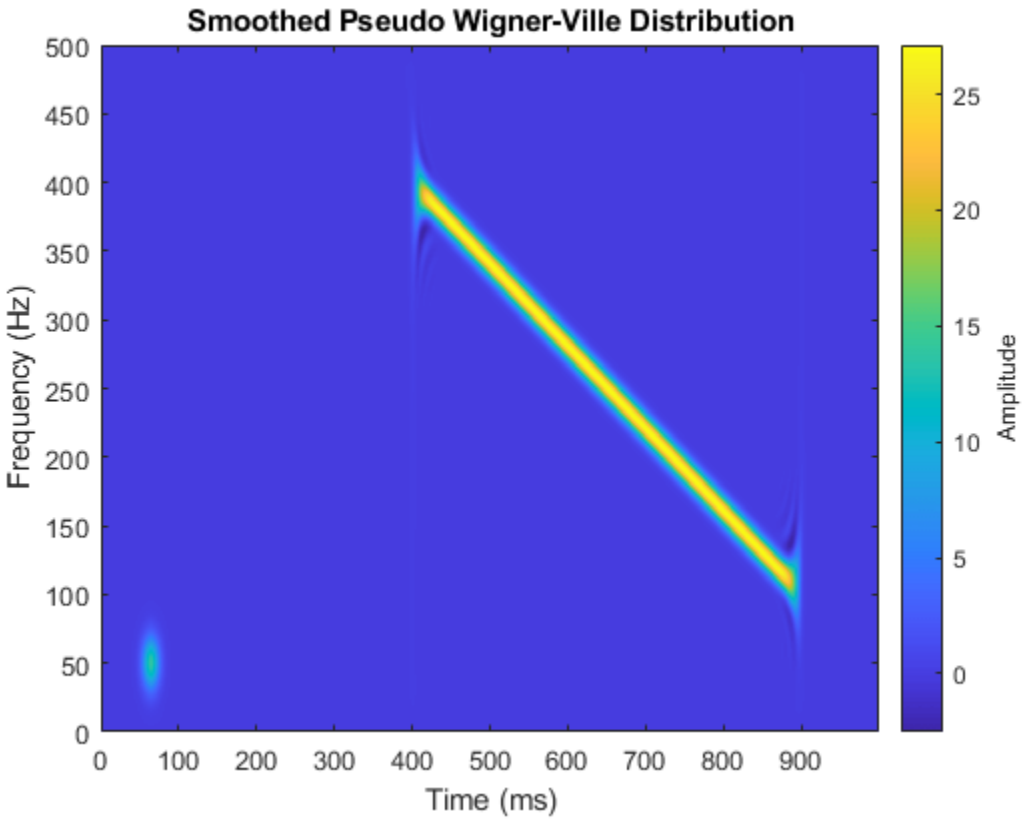
```
f0 = 400;
f1 = 100;
```

```
xa = zeros(size(t));
xa(t>0.4 & t<=0.9) = chirp((0:1/fs:0.5-1/fs)',f0,0.5,f1);
```

Create a two-component signal consisting of the sum of the unknown and reference signals. The smoothed pseudo Wigner-Ville distribution of the result provides an "ideal" time-frequency representation.

Compute and display the smoothed pseudo Wigner-Ville distribution.

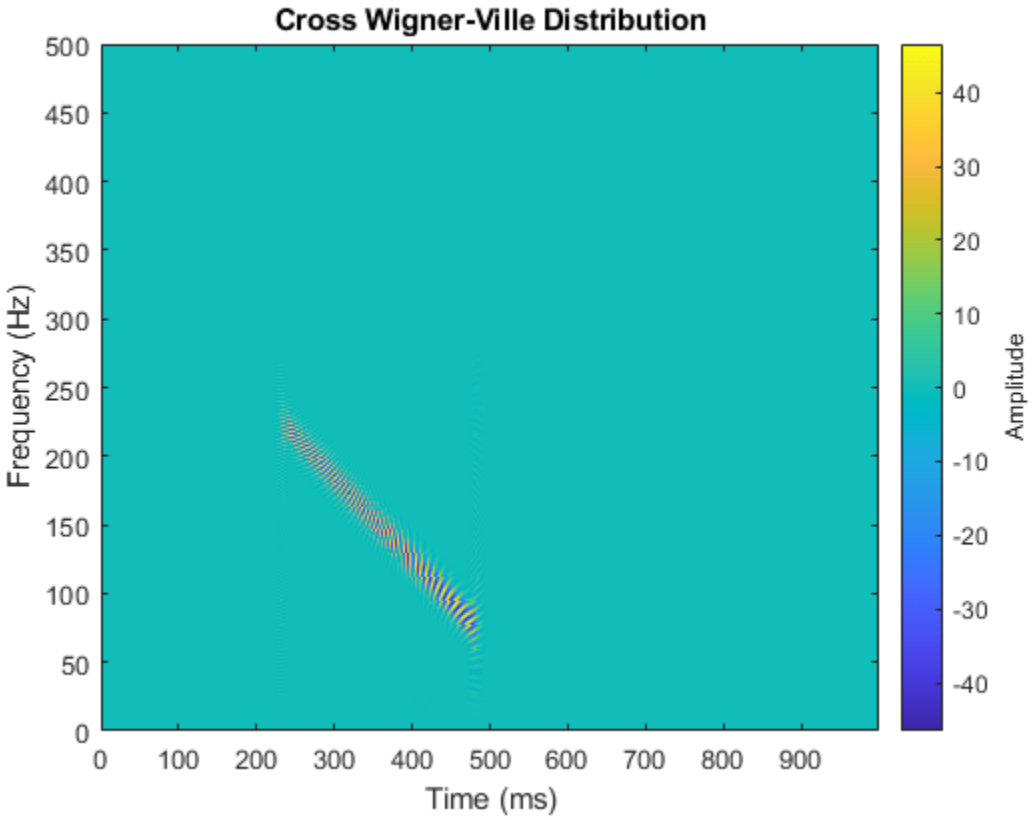
```
w = wvd(xa+xr,fs,'smoothedPseudo');

wvd(xa+xr,fs,'smoothedPseudo')
```



Compute the cross Wigner-Ville distribution of the unknown and reference signals. Take the absolute value of the distribution and set to zero the elements with amplitude less than 10. The cross Wigner-Ville distribution is equal to the cross-terms of the two-component signal.

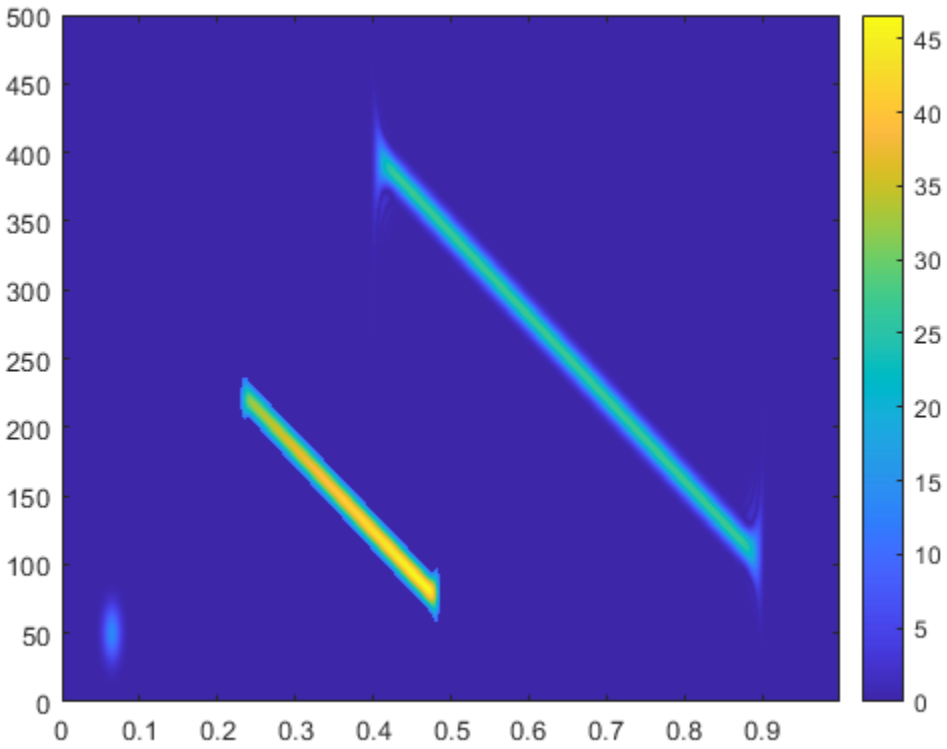
Plot the real part of the cross Wigner-Ville distribution.

```
[c,fc,tc] = xwvd(xa,xr,fs);
c = abs(c);
c(c<10) = 0;

xwvd(xa,xr,fs)
```



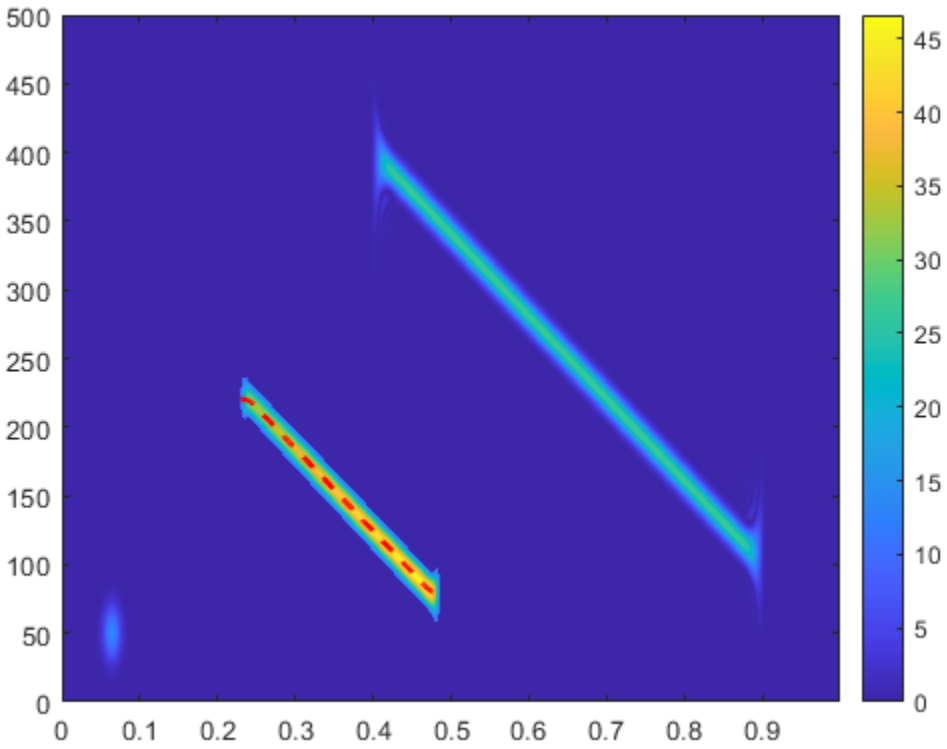
Enhance the Wigner-Ville cross-terms by adding the ideal time-frequency representation to the cross Wigner-Ville distribution. The cross-terms of the Wigner-Ville distribution occur halfway between the reference signal and the unknown signal.

```
d = w + c;

d = abs(real(d));

imagesc(tc,fc,d)
axis xy
colorbar
```



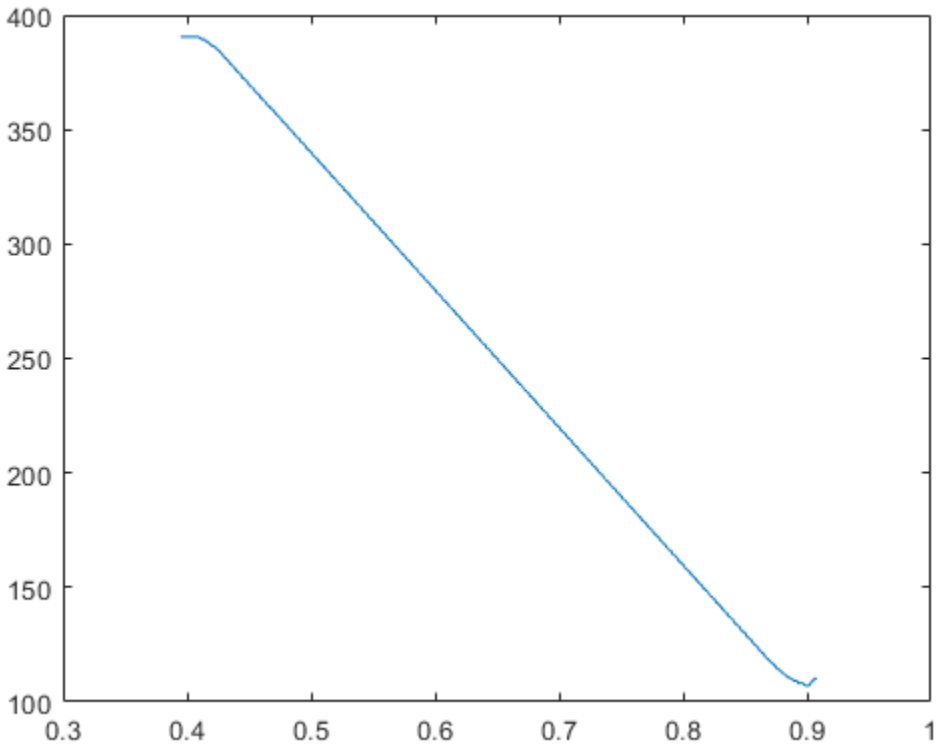
Identify and plot the high-energy ridge corresponding to the cross-terms. To isolate the ridge, find the time values where the cross-distribution has nonzero energy.

```
ff = tfridge(c,fc);

tv = sum(c)>0;

ff = ff(tv);
tc = tc(tv);

hold on
plot(tc,ff,'r--','linewidth',2)
hold off
```



Reconstruct the instantaneous frequency of the unknown signal by using the ridge and the reference function. Plot the instantaneous frequency as a function of time.

```
tEst = 2*tc - mu;
fEst = 2*ff - fsin;

plot(tEst,fEst)
```



Input Arguments

x, y — Input signals

vectors | timetables

Input signals, specified as vectors or MATLAB timetables each containing a single vector variable. `x` and `y` must both be vectors or both be timetables and must have the same length.

- If `x` and `y` are timetables, then they must contain increasing finite row times.
- If a timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”.

If the input signals have odd length, the function appends a zero to make the length even.

Example: `cos(pi/8*(0:159))'+randn(160,1)/10` specifies a sinusoid embedded in white noise.

Example: `timetable(seconds(0:5)',rand(6,1))` specifies a random variable sampled at 1 Hz for 4 seconds.

Data Types: `single` | `double`
Complex Number Support: Yes

fs — Sample rate

`2*pi` (default) | positive numeric scalar

Sample rate, specified as a positive numeric scalar.

`ts` — Sample time
`duration` scalar

Sample time, specified as a `duration` scalar.

`twin`, `fwin` — Time and frequency windows
vectors of odd length

Time and frequency windows used for smoothing, specified as vectors of odd length. By default, `xwvd` uses Kaiser windows with shape factor $\beta = 20$.

- The default length of `twin` is the smallest odd integer greater than or equal to `round(length(x)/10)`.
- The default length of `fwin` is the smallest odd integer greater than or equal to `nf/4`.

Each window must have a length smaller than or equal to `2*ceil(length(x)/2)`.

Example: `kaiser(65,0.5)` specifies a 65-sample Kaiser window with a shape factor of 0.5.

`nf` — Number of frequency points
`2*ceil(length(x)/2)` (default) | integer

Number of frequency points, specified as an integer. This argument controls the degree of oversampling in frequency. The number of frequency points must be at least `(length(fwin)+1)/2` and cannot be greater than the default.

`thresh` — Minimum nonzero value
`-Inf` (default) | real scalar

Minimum nonzero value, specified as a real scalar. The function sets to zero those elements of `d` whose amplitudes are less than `thresh`.

Output Arguments

`d` — Cross Wigner-Ville distribution
matrix

Cross Wigner-Ville distribution, returned as a matrix. Time increases across the columns of `d`, and frequency increases down the rows. The matrix is of size $N_f \times N_t$, where N_f is the length of `f` and N_t is the length of `t`.

`f` — Frequencies
vector

Frequencies, returned as a vector.

- If the input has time information, then `f` contains frequencies expressed in Hz.
- If the input does not have time information, then `f` contains normalized frequencies expressed in rad/sample.

`t` — Time instants
vector

Time instants, returned as a vector.

- If the input has time information, then `t` contains time values expressed in seconds.
- If the input does not have time information, then `t` contains sample numbers.

The number of time points is fixed as `4*ceil(length(x)/2)`.

More About

Cross Wigner-Ville Distribution

For continuous signals $x(t)$ and $y(t)$, the *cross Wigner-Ville distribution* is defined as

$$\mathrm{XWVD}_{x,y}(t,f) = \int_{-\infty}^{\infty} x\left(t+\frac{\tau}{2}\right) y^*\left(t-\frac{\tau}{2}\right) e^{-j2\pi f\tau}\, d\tau\,.$$

For a discrete signal with N samples, the distribution becomes

$$\mathrm{XWVD}_{x,y}(n,k) = \sum_{m=-N}^{N} x(n+m/2)\, y^*(n-m/2)\, e^{-j2\pi km/N}\,.$$

For odd values of m, the definition requires evaluation of the signal at half-integer sample values. It therefore requires interpolation, which makes it necessary to zero-pad the discrete Fourier transform to avoid aliasing.

The cross Wigner-Ville distribution contains interference terms that often complicate its interpretation. To sharpen the distribution, one can filter the definition with lowpass windows. The cross smoothed pseudo Wigner-Ville distribution uses independent windows to smooth in time and frequency:

$$\mathrm{XSPWVD}_{x,y}^{g,H}(t,f) = \int_{-\infty}^{\infty} g(t)\, H(f)\, x\left(t+\frac{\tau}{2}\right) y^*\left(t-\frac{\tau}{2}\right) e^{-j2\pi f\tau}\, d\tau\,.$$

References

[1] Cohen, Leon. *Time-Frequency Analysis: Theory and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[2] Mallat, Stéphane. *A Wavelet Tour of Signal Processing*. Second Edition. San Diego, CA: Academic Press, 1999.

[3] Malnar, Damir, Victor Sucic, and Boualem Boashash. "A cross-terms geometry based method for components instantaneous frequency estimation using the cross Wigner-Ville distribution." In *11th International Conference on Information Sciences, Signal Processing and their Applications (ISSPA)*, pp. 1217–1222. Montréal: IEEE, 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Arguments specified using name-value pairs must be compile-time constants.

- Timetables are not supported for code generation.

See Also

Functions
`cpsd` | `fsst` | `pspectrum` | `xspectrogram` | `wvd`

Topics
“Time-Frequency Gallery”

Introduced in R2018b

yulewalk

Recursive digital filter design

Syntax

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

Description

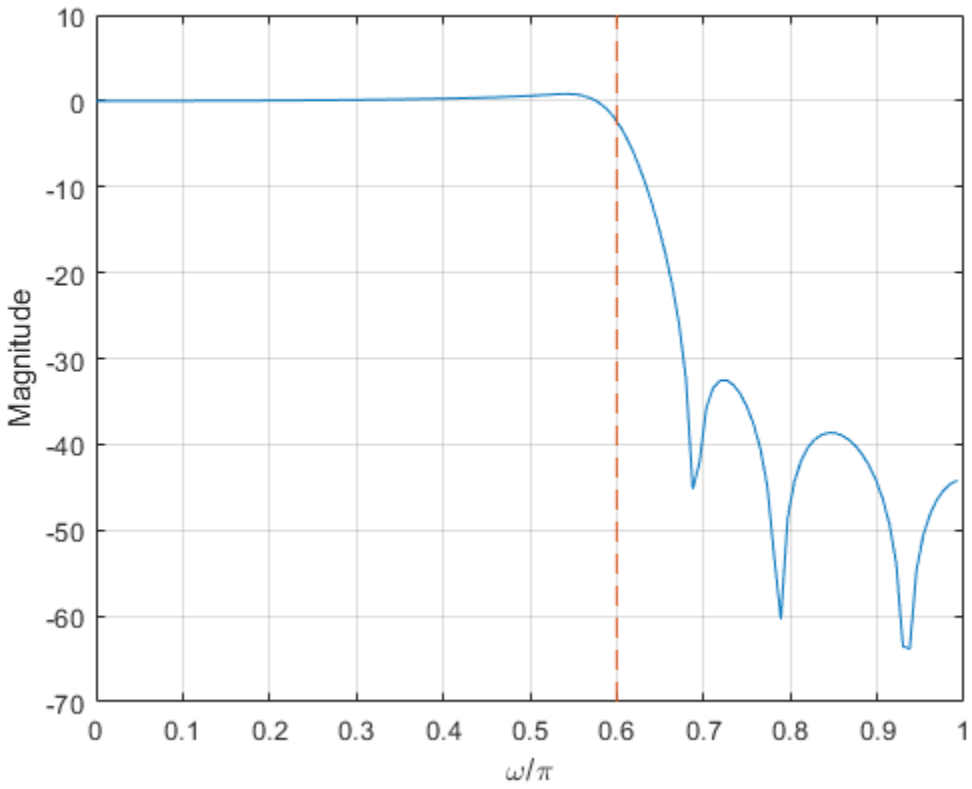
`[b,a] = yulewalk(n,f,m)` returns the transfer function coefficients of an `n`th-order IIR filter whose frequency magnitude response approximately matches the values given in `f` and `m`.

Examples

Yule-Walker Design of Lowpass Filter

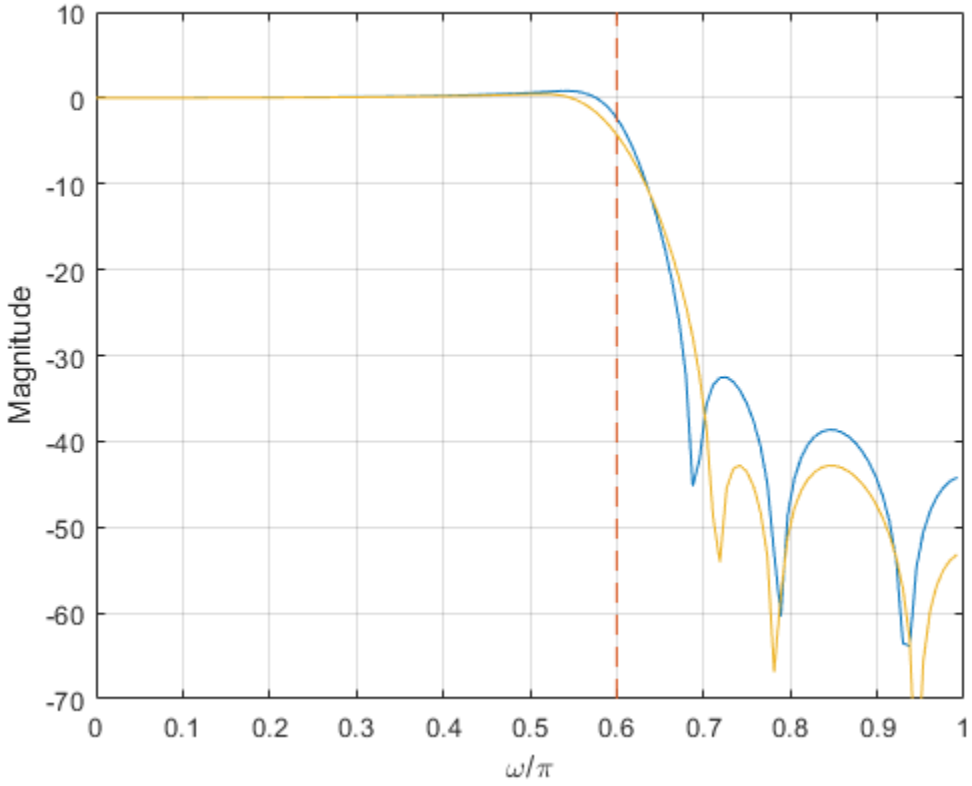
Design an 8th-order lowpass filter with normalized cutoff frequency 0.6. Plot its frequency response and overlay the response of the corresponding ideal filter.

```
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,mag2db(abs(h)))
yl = ylim;
hold on
plot(f(2:3),yl,'--')
xlabel('\omega/\pi')
ylabel('Magnitude')
grid
```



Increase the stopband attenuation by specifying a wider transition band.

```
f = [0 0.55 0.6 0.65 1];
m = [1 1 0.5 0 0];

[b,a] = yulewalk(8,f,m);
h = freqz(b,a,128);

hold on
plot(w/pi,mag2db(abs(h)))
hold off
ylim(yl)
```



Input Arguments

`n` — Filter order

positive integer scalar

Filter order, specified as a positive integer scalar.

Data Types: `single` | `double`

`f` — Frequency points

vector

Frequency points, specified as a vector of points in the range between 0 and 1, where 1 corresponds to the Nyquist frequency, or half the sample rate. The first point of `f` must be 0 and the last point 1. All intermediate points must be in increasing order. `f` can have duplicate frequency points corresponding to steps in the frequency response.

Example: `[0 0.25 0.4 0.5 0.5 0.7 1]` specifies an irregularly sampled Nyquist range.

Data Types: `single` | `double`

`m` — Magnitude response

vector

Magnitude response, specified as a vector containing the desired responses at the points specified in `f`. `m` must be the same length as `f`.

Example: `[0 1 1 1 0 0 0]` specifies a bandpass magnitude response.

Data Types: `single` | `double`

Output Arguments

b, a — Filter coefficients

row vectors

Filter coefficients, returned as row vectors. The output filter coefficients are ordered in descending powers of z:

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

Tips

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.

Algorithms

`yulewalk` designs recursive IIR digital filters using a least-squares fit to a specified frequency response. The function performs the fit in the time domain.

- To compute the denominator coefficients, `yulewalk` uses modified Yule-Walker equations, with correlation coefficients computed by inverse Fourier transformation of the specified frequency response.
- To compute the numerator, `yulewalk` follows these steps:

  1. Compute a numerator polynomial corresponding to an additive decomposition of the power frequency response.
  2. Evaluate the complete frequency response corresponding to the numerator and denominator polynomials.
  3. Use a spectral factorization technique to obtain the impulse response of the filter.
  4. Obtain 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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

If specified, the order of recursion must be a constant. Expressions or variables are allowed if their values do not change.

See Also

`butter` | `cheby1` | `cheby2` | `ellip` | `fir2` | `firls` | `maxflat` | `firpm`

Introduced before R2006a

zerocrossrate

Zero-crossing rate

Syntax

```
rate = zerocrossrate(x)
rate = zerocrossrate(TT)
rate = zerocrossrate( ___ ,Name,Value)
[rate,count] = zerocrossrate( ___ )
[rate,count,indices] = zerocrossrate( ___ )
zerocrossrate( ___ )
```

Description

`rate = zerocrossrate(x)` returns the zero-crossing rate of `x`. If `x` is a matrix, then the function analyzes each column as a separate channel and returns the zero-crossing rate as a row vector where each value corresponds to a channel.

`rate = zerocrossrate(TT)` returns the zero-crossing rate of the data stored in the MATLAB timetable `TT`. If `TT` contains multiple channels, then the function analyzes each channel independently.

`rate = zerocrossrate( ___ ,Name,Value)` specifies additional name-value arguments. Use this syntax with any of the input arguments in previous syntaxes.

`[rate,count] = zerocrossrate( ___ )` also returns the total number of crossings in `count`.

`[rate,count,indices] = zerocrossrate( ___ )` also returns logical indices at the signal locations where a crossing occurs.

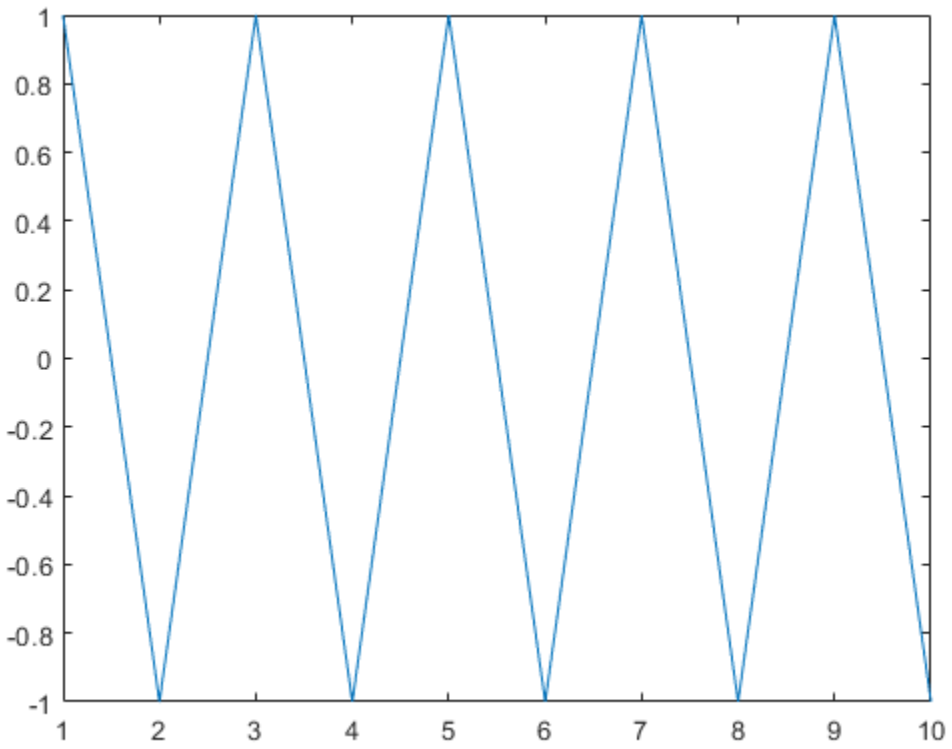
`zerocrossrate( ___ )` with no output arguments plots `rate` along the *y*-axis and the corresponding window number along the *x*-axis. If the window length is equal to the full signal length, then the function plots the length of the window along the *x*-axis and the crossing rate in the middle of the window.

Examples

Count Zero Crossings in Signal

Consider a vector of ones with alternating signs. Plot the data.

```
x = [1 -1 1 -1 1 -1 1 -1 1 -1];
plot(x)
```



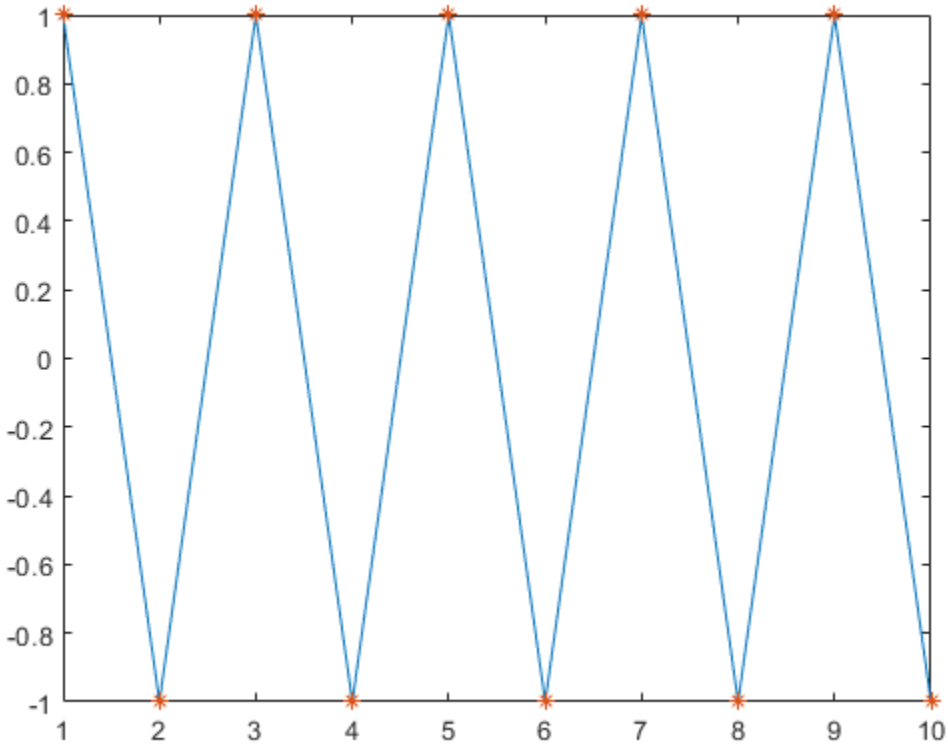
Compute the zero-crossing rate of x.

```
r = zerocrossrate(x)
```

```
r = 0.9500
```

Use the third output argument to find the locations where the crossings occur. Plot x and the zero-crossing locations. The function returns an index at the next sample after a crossing, not necessarily the exact crossing location. The first sample is marked as a crossing point because the function considers the initial state of x to be zero by default.

```
[~,~,indices] = zerocrossrate(x);
plot(x)
hold on
plot(x(indices),'*')
hold off
```



Compute the zero-crossing rate of `x` using the comparison method. The rate differs from the value computed using the difference method.

```
rC = zerocrossrate(x,Method="comparison")
```

```
rC = 0.9000
```

Compute the zero-crossing rate of `x` again using the difference method and specify zero as positive. The rate is equal to the value computed using the comparison method.

```
rZ = zerocrossrate(x,ZeroPositive=1)
```

```
rZ = 0.9000
```

Now specify the initial state of `x` as `1`. The rate is equal to the previous result.

```
rI = zerocrossrate(x,InitialState=1)
```

```
rI = 0.9000
```

Count Level Crossings in Temperature Data

Load a set of temperature readings in Celsius taken every hour at Logan Airport in Boston for the entire month of January, 2011. Create a `timetable` and use `retime` to aggregate the data into daily means.

```
load bostemp

t = hours(1:24*31)';
TT = timetable(t,tempC);
rTT = retime(TT,'daily','mean');
```

Count the number of days the temperature crosses the monthly average. Plot the data and include a horizontal line at the monthly average temperature to visualize where the crossings occur.

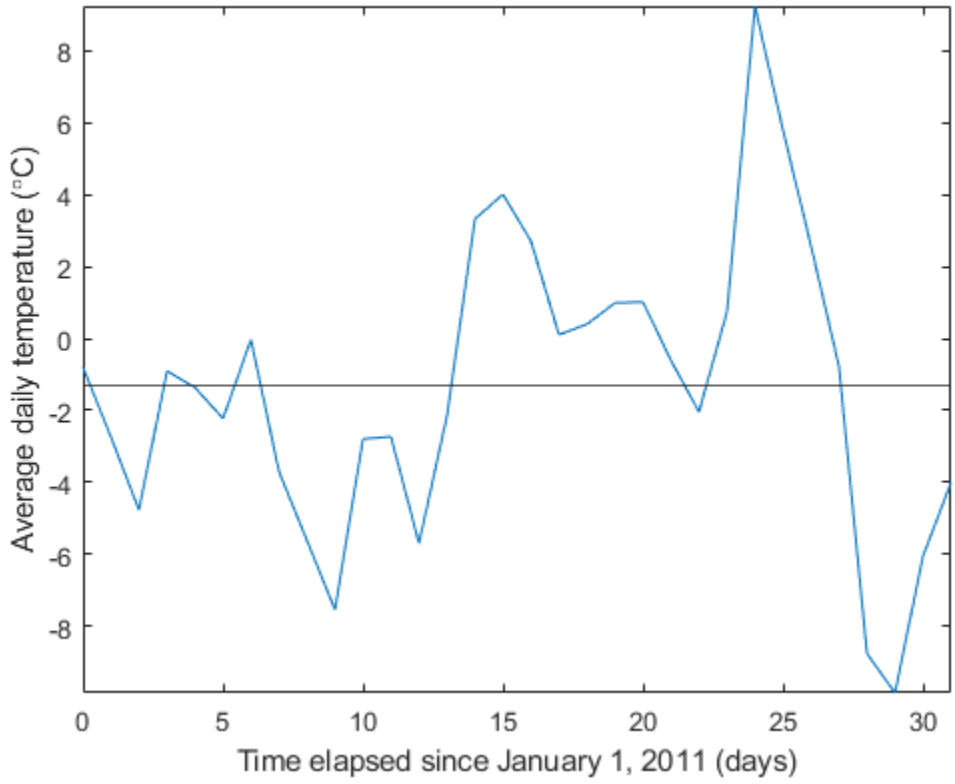
```
avg = mean(TT.tempC)
```

```
avg = -1.3007
```

```
[~,count] = zerocrossrate(rTT,Level=avg)
```

```
count = 9
```

```
plot(hours(rTT.t/24),rTT.tempC)
yline(avg)
xlabel('Time elapsed since January 1, 2011 (days)')
ylabel('Average daily temperature (\circC)')
axis tight
```



Identify Voiced and Unvoiced Speech Using Zero Crossings

Speech can be characterized as being voiced or unvoiced. *Voiced* speech, such as vowel sounds, occurs when the vocal cords vibrate. In *unvoiced* speech, such as most consonant sounds, the vocal chords do not vibrate. You can use zero crossings to classify the voiced and unvoiced regions in an audio signal.

Load an audio signal into the MATLAB® workspace. The voice says, "Oak is strong, and also gives shade".

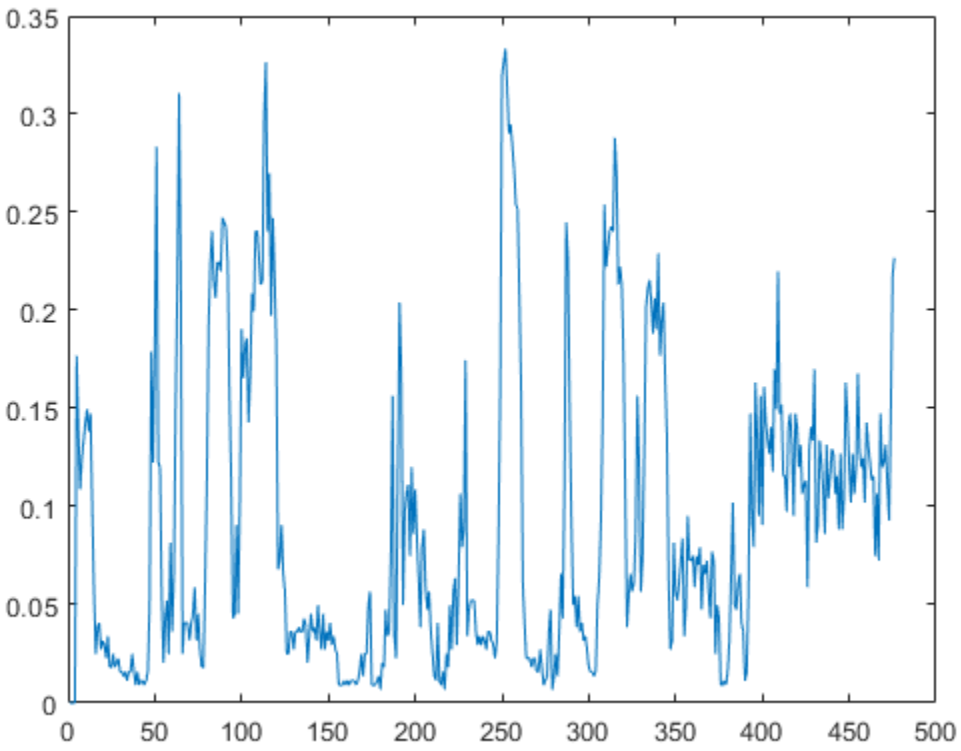
```
[y,fs] = audioread("oak.m4a");

% To hear, type soundsc(y,fs)
```

The signal is sampled at 44.1 kHz. Calculate the zero-crossing rate for 10 ms windows using the comparison method.

```
win = fs*0.01;
rate = zerocrossrate(y,WindowLength=win,Method="comparison");
```

Plot `rate` to visualize the crossing rate for each segment. Voiced speech is expected to have a low crossing rate, while unvoiced speech is expected to have a high crossing rate.

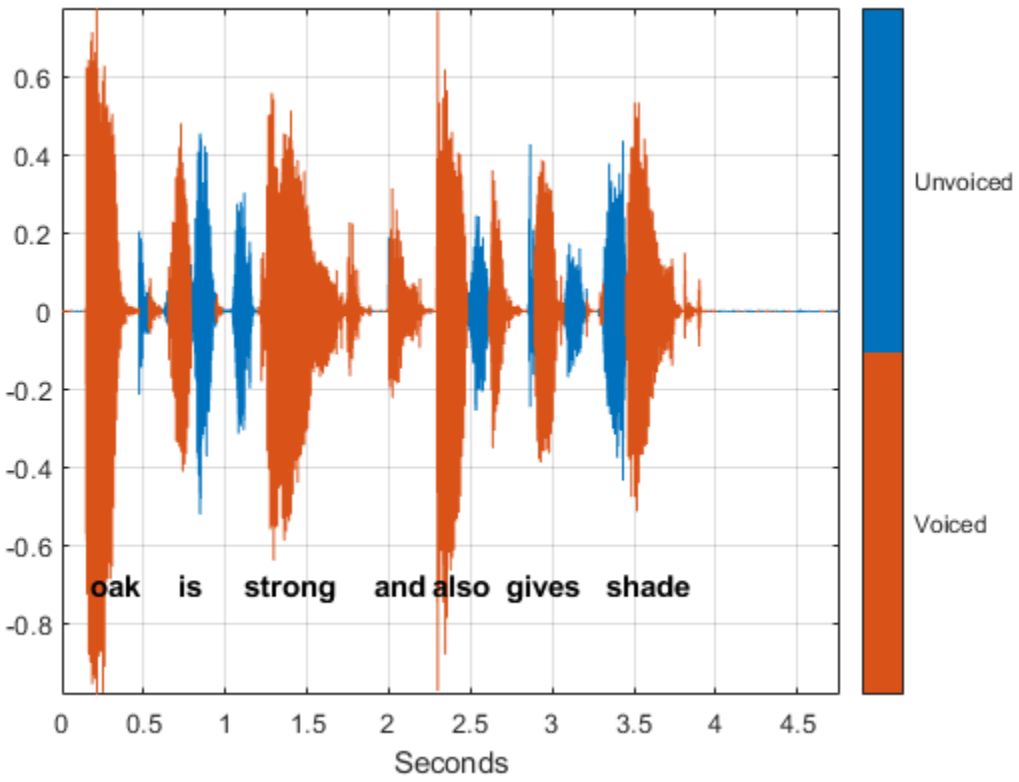
```
plot(rate)
```



Use a threshold of `0.1` to differentiate between voiced and unvoiced segments. Create a `signalMask` object that has two categories ("Unvoiced" and "Voiced") and plot the regions of interest (ROIs). Compare the regions of voiced and unvoiced speech to the location of each spoken word.

IBM® Watson Speech to Text API and Audio Toolbox™ software can be used to extract words from an audio file. Load `Transcription.mat` into the workspace. The labeled signal set contains the audio signal, ROI limits, and labels for each spoken word. For details, see “Label Spoken Words in Audio Signals”. Display the spoken words on the plot.

```
h = 0.1;
idu = find(rate > h);
idu(1:2) = [];
vi = [(idu-1) idu]*win;

m = sigroi2binmask(vi,length(y));
mask = signalMask([m ~m],Categories=["Unvoiced" "Voiced"],SampleRate=fs);
plotsigroi(mask,y)

load Transcription

ln = getLabelNames(transcribedAudio);
v = getLabelValues(transcribedAudio,1,ln);
v.Value = categorical(v.Value,v.Value);

RL = v.ROILimits;
VL = v.Value;

hold on
text(mean(RL,2),-0.7*ones(size(VL)),VL,HorizontalAlignment="center", ...
    FontSize=11,FontWeight="bold")
hold off
```



Zero-Crossing Rate of Streaming Data

Load an audio file containing 15 seconds of acoustic guitar music. The sample rate is 44.1 kHz. To play the music, uncomment the last line of code.

```
Fs = 44100;
y = audioread("guitartune.wav");

% sound(y,Fs)
```

Buffer the audio signal into overlapping frames, each with 4096 samples. Use an overlap of 512 samples.

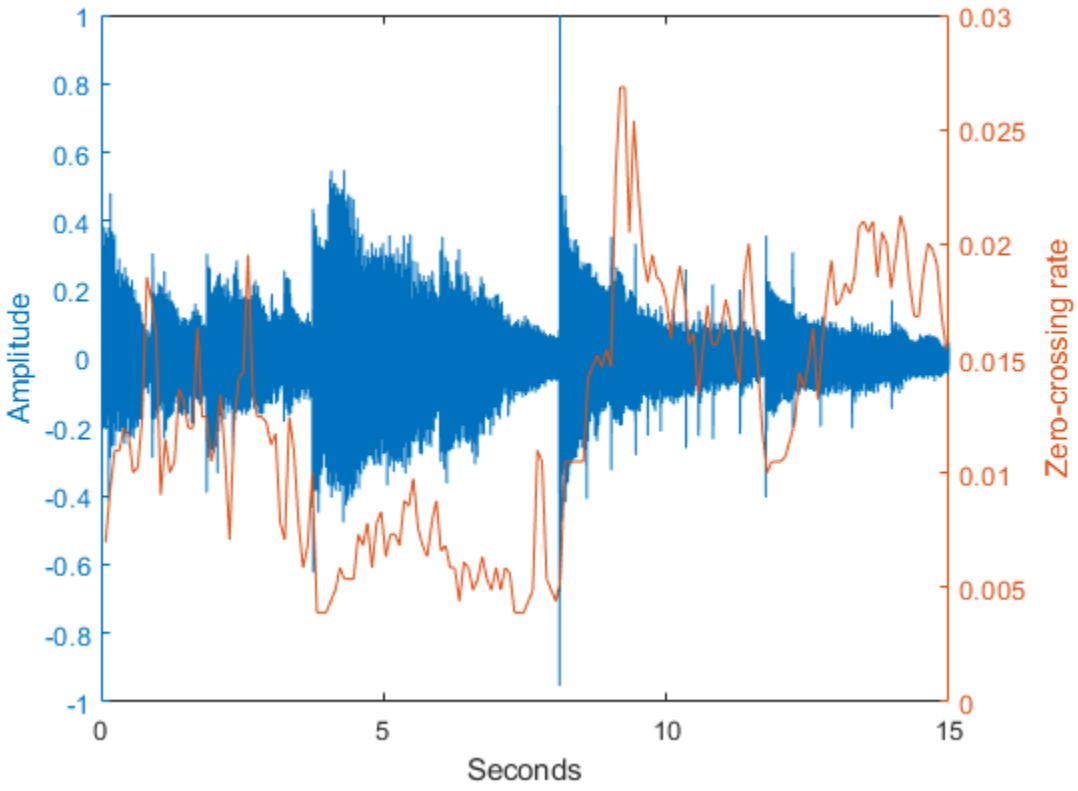
```
winLength = 4096;
overlap = 512;

[yB,~] = buffer(y,winLength,overlap,"nodelay");
```

Obtain the zero-crossing rate for each frame. To account for frame overlap, specify the initial state as the previous value of the first sample in each frame.

```
init = winLength - overlap;
state = 0;
zcr = [];
for i = 1:size(yB,2)
    zcr = [zcr;zerocrossrate(yB(:,i),InitialState=state)];
    state = yB(init,i);
end
```

Plot the audio signal and overlay the zero-crossing rate for each frame.

```
figure
yyaxis left
x = 0:1/Fs:(numel(y)-1)/Fs;
plot(x',y)
xlabel("Seconds")
ylabel("Amplitude")
yyaxis right
xx = (1:size(yB,2))*((winLength-overlap)/Fs);
plot(xx',zcr)
ylabel("Zero-crossing rate")
```



Compute the zero-crossing rate of the unbuffered signal. To obtain a result equivalent to the zero-crossing rate of the buffered signal, set `WindowLength` to `4096` and `OverlapLength` to `512`. Determine if the two results are equal.

```
zcr_batch = zerocrossrate(y,WindowLength=winLength,OverlapLength=overlap);

isequal(zcr_batch,zcr)
```

```
ans = logical
   1
```

Input Arguments

x — Data

real-valued vector | real-valued matrix

Data, specified as a real-valued vector or matrix. If `x` is a matrix, the function returns the zero-crossing rate as a row vector where each value corresponds to a column of data.

Data Types: `single` | `double`

TT — Input timetable

timetable

Input timetable, specified as a `timetable`. `TT` must contain uniformly sampled single- or double-precision data. The “RowTimes” property must contain a `duration` or `datetime` vector with

increasing and finite values. If TT is a timetable with a single variable containing a matrix, or a timetable with multiple variables each containing a vector, then the function analyzes each channel independently.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `zerocrossrate(x,Method="comparison",Level=7,transitionEdge="rising")` uses the comparison method to compute the rate at which `x` positively transitions across `7`.

`InitialState` — Previous states

`0` (default) | vector

Previous states of `x`, specified as a vector whose number of elements is equal to the number of input channels.

Example: `zerocrossrate(x,InitialState=[1 0 –1 3])` returns the crossing rates of a four-channel input signal `x`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Method` — Method for computing zero-crossing rate

`"difference"` (default) | `"comparison"`

Method for computing the zero-crossing rate, specified as `"difference"` or `"comparison"`. If you do not specify `Method`, the function uses the difference method to compute the crossing rate.

- `comparison` — The function marks the `indices` as true where a crossing is fully completed.
- `difference` — The function marks the `indices` as true where abs(sign(x_i)–sign(x_{i-1})) > 0.

Example: `zerocrossrate(x,Method="comparison")` computes the crossing rate of `x` using the comparison method.

Data Types: `char` | `string`

`WindowLength` — Window length

positive integer

Window length over which to compute the crossing rate, specified as a positive integer. The default window length is the signal length.

Example: `zerocrossrate(x,WindowLength=20)` returns the crossing rates for 20-sample windows in `x`.

Example: `zerocrossrate(x,WindowLength=fs*0.05)` returns the crossing rates for 50 ms windows in `x` given a sample rate `fs`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`OverlapLength` — Number of overlapping samples

`0` (default) | positive integer

Number of overlapping samples between adjoining segments, specified as a positive integer. The overlap must be smaller than the window length.

Example: `zerocrossrate(x,OverlapLength=0)` returns the crossing rates of segments with no overlap.

Example: `zerocrossrate(x,WindowLength=20,OverlapLength=5)` returns the crossing rates of overlapping segments with five samples of overlap.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Level` — Signal level
`0` (default) | real scalar

Signal level for which the crossing rate is computed, specified as a real scalar. The function subtracts the `Level` value from the signal and then finds the zero crossings. If you do not specify `Level`, the function uses the default value of `0` and returns the zero-crossing rate.

Example: `zerocrossrate(x,Level=1)` returns the rate at which the input signal `x` crosses `1`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Threshold` — Threshold
`0` (default) | real scalar

Threshold value above and below the `Level` value over which the crossing rate is computed, specified as a real scalar. The function sets all the values of the input in the range [`–threshold`, `threshold`] to `0` and then finds the zero crossings.

Example: `zerocrossrate(x,Threshold=0.1)` returns the crossing rate with a tolerance of –0.1 to 0.1.

Note When you specify both `Level` and `Threshold`, the function first subtracts the level value from the input and then sets to `0` the resulting input values that are in the range [`–threshold`, `threshold`].

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TransitionEdge` — Transitions
`"both"` (default) | `"falling"` | `"rising"`

Transitions to include when counting the zero crossings, specified as `"falling"`, `"rising"`, or `"both"`. If you specify `"falling"`, the function counts only negative-going transitions. If you specify `"rising"`, the function counts only positive-going transitions.

Example: `zerocrossrate(x,TransitionEdge="rising")` returns the crossing rate of `x` for only positive-going transitions.

Data Types: `char` | `string`

`ZeroPositive` — Sign convention
`0` or `false` (default) | `1` or `true`

Sign convention, specified as a logical scalar. If you specify `ZeroPositive` as true, the function considers `0` to be positive. If you specify `ZeroPositive` as false, the function considers `0`, `–1`, and `+1` to have distinct signs following the convention of the `sign` function.

Example: `zerocrossrate(x,ZeroPositive=1)` returns the crossing rate of the input signal `x` and considers zero as positive.

Data Types: `logical`

Output Arguments

`rate` — Zero-crossing rate

row vector | matrix

Zero-crossing rate, returned as a row vector or a matrix. When `WindowLength` is equal to the signal length, `rate` is a row vector whose number of elements is equal to the number of channels in `x` or `TT`. When `WindowLength` is smaller than the signal length, the function returns `rate` as a matrix where the *i*-th row contains the crossing rate for the *i*-th window and the *j*-th column corresponds to the *j*-th input channel.

`count` — Number of crossings

N-by-*M* matrix

Number of crossings, returned as an *N*-by-*M* matrix where *N* is the number of windows and *M* is the number of input channels. The *i*-th row corresponds to the crossing count for the *i*-th window and the *j*-th column corresponds to the crossing count for the *j*-th channel.

`indices` — Logical indices

N-by-`WindowLength`-by-*M* array

Logical indices at the signal locations where crossings occur, returned as an *N*-by-`WindowLength`-by-*M* array where *N* is the number of windows and *M* is the number of input channels.

Note Indices might not represent exact signal crossing locations. The `zerocrossrate` function returns an index at the next sample following a crossing.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support disabling dynamic memory allocation when the window length is specified and the input is more than one channel.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Functions

`midcross`

Introduced in R2021b

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,f] = zerophase(___,fs)
[Hr,f] = zerophase(___,"whole",fs)

Hr = zerophase(___,w)
Hr = zerophase(___,f,fs)

[Hr,w,phi] = zerophase(___)

zerophase(___)
```

Description

`[Hr,w] = zerophase(b,a)` returns the zero-phase response `Hr` and the angular frequencies `w` at which the zero-phase response is computed, given a filter defined by numerator coefficients `b` and denominator coefficients `a`. The function evaluates the zero-phase response at `512` equally spaced points on the upper half of the unit circle.

`[Hr,w] = zerophase(sos)` returns the zero-phase response for the second order sections matrix `sos`.

`[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)` uses `nfft` frequency points on the upper half of the unit circle to compute the zero-phase response. You can use an input combination from any of the previous syntaxes.

`[Hr,w] = zerophase(___,nfft,"whole")` uses `nfft` frequency points around the whole unit circle to compute the zero-phase response.

`[Hr,f] = zerophase(___,fs)` uses the sample rate `fs` to determine the frequencies `f` at which the zero-phase response is computed.

`[Hr,f] = zerophase(___,"whole",fs)` uses the sample rate `fs` to determine the frequencies `f` around the whole unit circle at which the zero-phase response is computed.

`Hr = zerophase(___,w)` returns the zero-phase response `Hr` evaluated at the angular frequencies specified in `w`.

`Hr = zerophase(___,f,fs)` returns the zero-phase response `Hr` evaluated at the frequencies specified in `f`.

`[Hr,w,phi] = zerophase(___)` returns the zero-phase response `Hr`, angular frequencies `w`, and continuous phase component `phi`.

`zerophase(___)` plots the zero-phase response versus frequency. If you input the filter coefficients or second order sections matrix, the function plots in the current figure window. If you input a `digitalFilter`, the function displays the step response in **FVTool**.

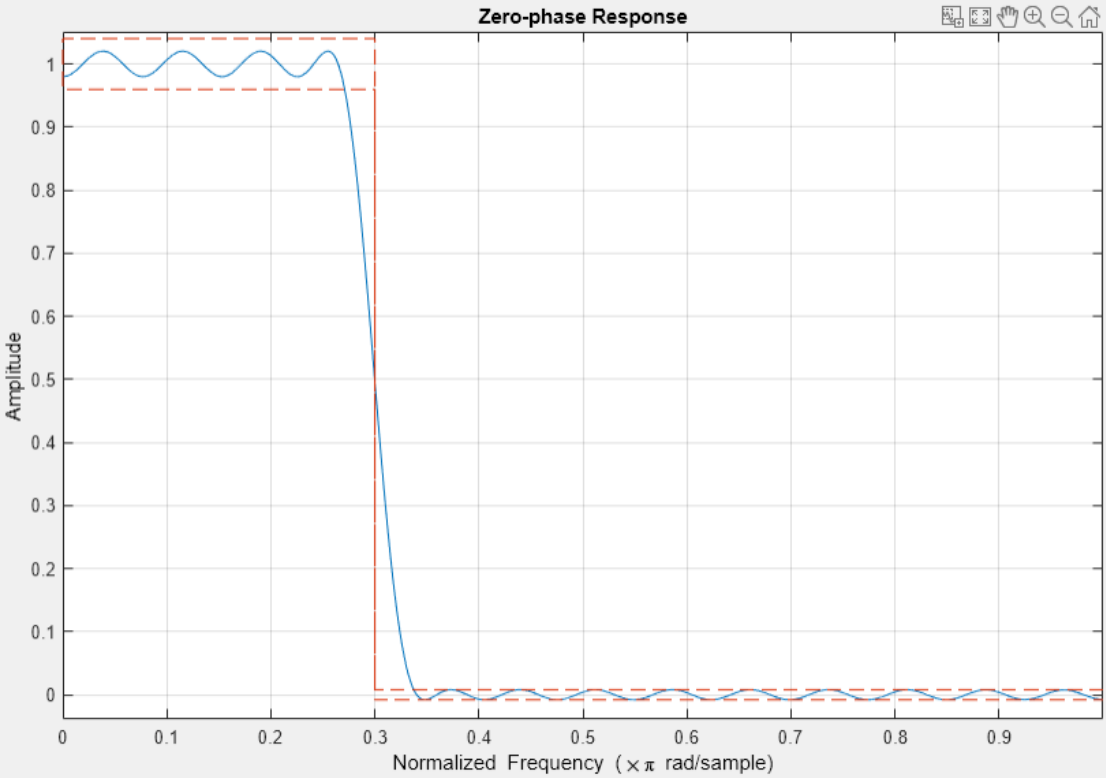
Note If the input to `zerophase` is single precision, the function calculates the zero-phase response using single-precision arithmetic. The output `Hr` is single precision.

Examples

Zero-Phase Response of FIR Filter

Use `designfilt` to design a 54th-order FIR filter with a normalized cutoff frequency of 0.3π rad/sample, a passband ripple of 0.7 dB, and a stopband attenuation of 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)
```



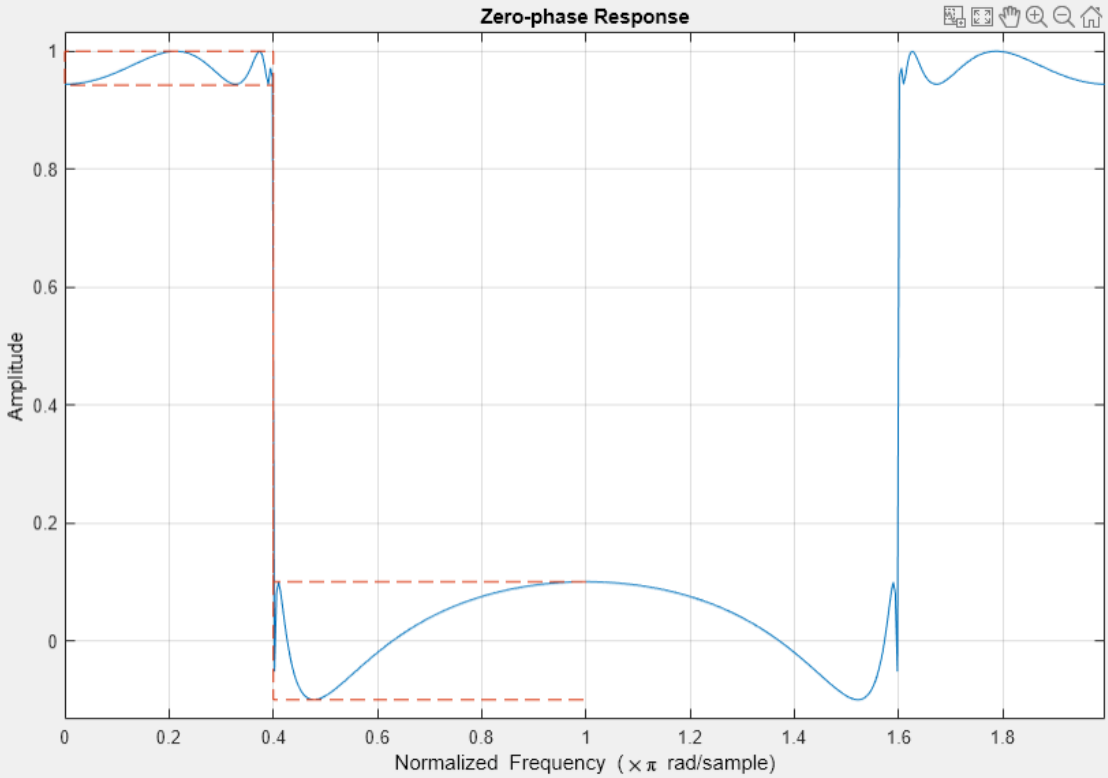
Design the same filter using `fircls1`, which uses linear units to measure the ripple and attenuation. Display the zero-phase response.

```
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 Elliptic Filter

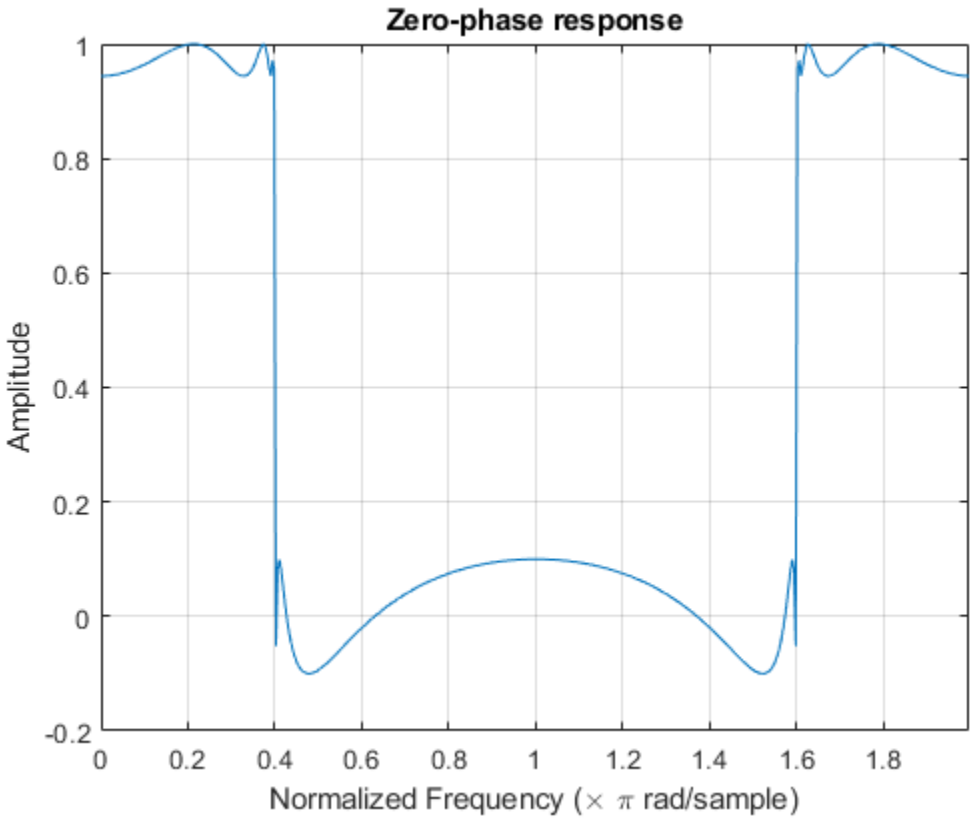
Design a 10th-order elliptic lowpass IIR filter with a normalized passband frequency of 0.4π rad/sample, a passband ripple of 0.5 dB, and a stopband attenuation of 20 dB. Display the zero-phase response of the filter on 512 frequency points around the whole unit circle.

```
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")
```



Input Arguments

b, a — Filter coefficients

vectors

Filter coefficients, specified as vectors. For FIR filters where `a=1`, you can omit the value `a` from the command.

Data Types: `single` | `double`

sos — Second order sections

k-by-6 matrix

Second order sections, specified as 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, the function 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)]`.

Data Types: `single` | `double`

d — Digital filter

`digitalFilter` object

Digital filter, specified as a `digitalFilter` object.

`nfft` — Number of frequency points

512 (default) | positive integer scalar

Number of frequency points used to compute the zero-phase response, specified as a positive integer scalar. For best results, set `nfft` to a value greater than the filter order.

Data Types: `double`

`w` — Angular frequencies

vector

Angular frequencies at which the zero-phase response is computed, specified as a vector and expressed in radians/sample. `w` must have at least two elements.

Data Types: `double`

`f` — Frequencies

vector

Frequencies at which the zero-phase response is computed, specified as a vector and expressed in hertz. `f` must have at least two elements.

Data Types: `double`

`fs` — Sample rate

positive scalar

Sample rate, specified as a positive scalar.

Data Types: `double`

Output Arguments

`Hr` — Zero-phase response

vector

Zero-phase response, returned as a vector.

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.

`w` — Angular frequencies

vector

Angular frequencies at which the zero-phase response is computed, returned as a vector.

`f` — Frequencies

vector

Frequencies at which the zero-phase response is computed, returned as a vector.

`phi` — Continuous phase component
vector

Continuous phase component, returned as a vector. This quantity is not equivalent to the phase response of the filter when the zero-phase response is negative.

References

[1] Antoniou, Andreas. *Digital Filters*. New York: McGraw-Hill, Inc., 1993.

See Also

`designfilt` | `digitalFilter` | `freqs` | `freqz` | **FVTool** | `grpdelay` | `invfreqz` | `phasedelay` | `phasez`

Introduced before R2006a

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

`[sos,g] = zp2sos(z,p,k)` finds a second-order section matrix `sos` with gain `g` that is equivalent to the transfer function $H(z)$ whose n zeros, m poles, and scalar gain are specified in `z`, `p`, and `k`:

$$H(z) = k\frac{(z-z_1)(z-z_2)\cdots(z-z_n)}{(z-p_1)(z-p_2)\cdots(z-p_m)}.$$

`[sos,g] = zp2sos(z,p,k,order)` specifies the order of the rows in `sos`.

`[sos,g] = zp2sos(z,p,k,order,scale)` specifies the scaling of the gain and numerator coefficients of all second-order sections.

`[sos,g] = zp2sos(z,p,k,order,scale,zeroflag)` specifies the handling of real zeros that are negatives of each other.

`sos = zp2sos( ___ )` embeds the overall system gain in the first section.

Examples

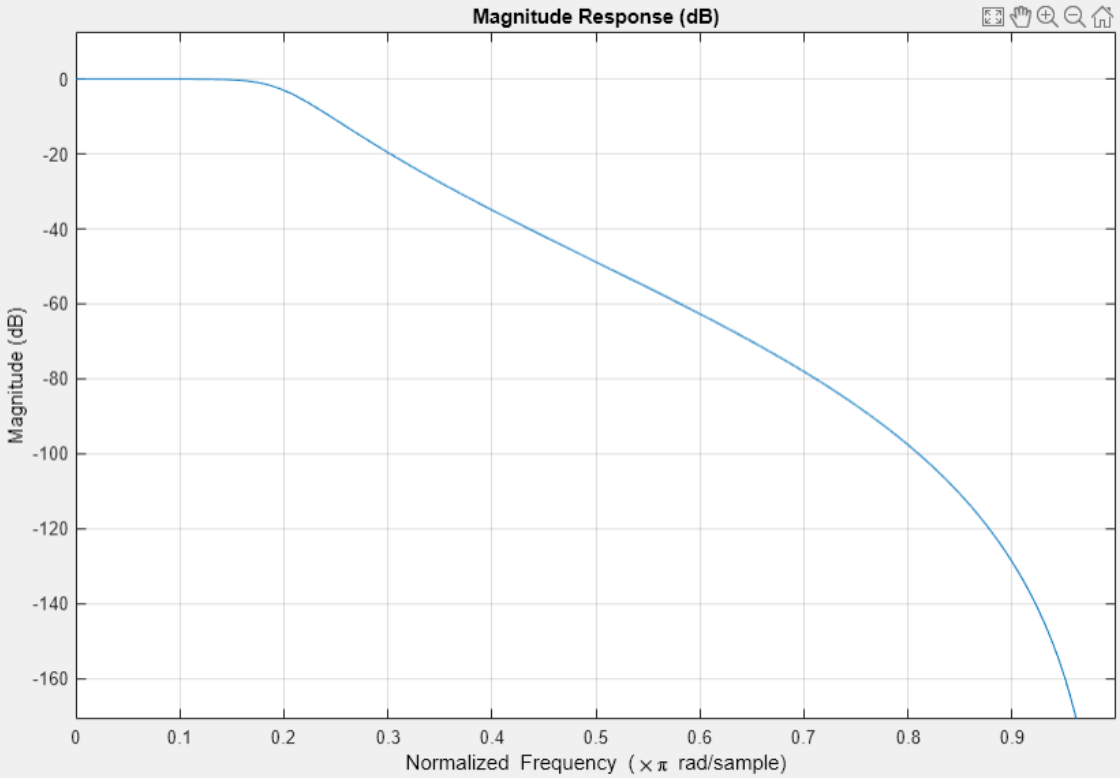
Second-Order Sections from Zero-Pole-Gain Parameters

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)
```

```
sos = 3×6

    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
```

```
fvtool(sos)
```



Input Arguments

z — Zeros

vector

Zeros of the system, specified as a vector. The zeros must be real or come in complex conjugate pairs.

Example: `[1 (1+1j)/2 (1-1j)/2]'`

Data Types: `double`

p — Poles

vector

Poles of the system, specified as a vector. The poles must be real or come in complex conjugate pairs.

Example: `[1 (1+1j)/2 (1-1j)/2]'`

Data Types: `double`

k — Scalar gain

scalar

Scalar gain of the system, specified as a scalar.

Data Types: `double`

`order` — Row order

`'up'` (default) | `'down'`

Row order, specified as one of the following:

- `'up'` — Order the sections so the first row of `sos` contains the poles farthest from the unit circle.
- `'down'` — Order the sections so the first row of `sos` contains the poles closest to the unit circle.

Data Types: `char`

`scale` — Scaling of gain and numerator coefficients

`'none'` (default) | `'inf'` | `'two'`

Scaling of gain and numerator coefficients, specified as one of the following:

- `'none'` — Apply no scaling.
- `'inf'` — Apply infinity-norm scaling.
- `'two'` — Apply 2-norm scaling.

Using infinity-norm scaling with `'up'`-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling with `'down'`-ordering minimizes the peak round-off noise.

Note Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

Data Types: `char`

`zeroflag` — Ordering of real zeros

`false` (default) | `true`

Ordering of real zeros that are negatives of each other, specified as a logical scalar.

- If you specify `zeroflag` as `false`, the function orders those zeros according to proximity to poles.
- If you specify `zeroflag` as `true`, the function keeps those zeros together. This option results in a numerator with a middle coefficient equal to zero.

Data Types: `logical`

Output Arguments

`sos` — Second-order section representation

matrix

Second-order section representation, returned as a matrix. `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 $H(z)$:

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

If the transfer function has n zeros and m poles, then L is the closest integer greater than or equal to $\max(n/2, m/2)$.

g — Overall system gain
real scalar

Overall system gain, returned as a real scalar.

If you call `zp2sos` with one output argument, the function embeds the overall system gain 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 can result in erratic scaling. To avoid embedding the gain, use `zp2sos` with two outputs.

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 using the `order` argument.
4. `zp2sos` scales the sections by the norm specified in `scale`. 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 infinity or 2. 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.

[2] Mitra, Sanjit Kumar. *Digital Signal Processing: A Computer-Based Approach*. 3rd ed. New York: McGraw-Hill Higher Education, 2006.

[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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Any character or string input must be a constant at compile time.

See Also

`cplxpair` | `filternorm` | `sos2zp` | `ss2sos` | `tf2sos` | `zp2ss` | `zp2tf`

Introduced before R2006a

zp2ss

Convert zero-pole-gain filter parameters to state-space form

Syntax

```
[A,B,C,D] = zp2ss(z,p,k)
```

Description

`[A,B,C,D] = zp2ss(z,p,k)` finds a state-space representation

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

such that it is equivalent to 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.

Examples

State-Space Representation of Mass-Spring System

Generate the state-space representation of a damped mass-spring system that obeys the differential equation

$$\ddot{w} + 0.01\dot{w} + w = u(t).$$

The measurable quantity is the acceleration, $y = \ddot{w}$, and $u(t)$ is the driving force. In Laplace space, the system is represented by

$$Y(s) = \frac{s^2\, U(s)}{s^2 + 0.01s + 1}.$$

The system has unit gain, a double zero at $s = 0$, and two complex-conjugate poles.

```
z = [0 0];
p = roots([1 0.01 1])
```

```
p = 2×1 complex

  -0.0050 + 1.0000i
  -0.0050 - 1.0000i
```

```
k = 1;
```

Use `zp2ss` to find the state-space matrices.

```
[A,B,C,D] = zp2ss(z,p,k)
```

```
A = 2×2

   -0.0100   -1.0000
    1.0000         0


B = 2×1

     1
     0


C = 1×2

   -0.0100   -1.0000


D = 1
```

Input Arguments

z — Zeros

vector

Zeros of the system, specified as a vector. The zeros must be real or come in complex conjugate pairs.

`Inf` values may be used as place holders in `z` if some columns have fewer zeros than others.

Example: `[1 (1+1j)/2 (1-1j)/2]'`

Data Types: `double`

p — Poles

vector

Poles of the system, specified as a vector. The poles must be real or come in complex conjugate pairs.

Example: `[1 (1+1j)/2 (1-1j)/2]'`

Data Types: `double`

k — Scalar gain

scalar

Scalar gain of the system, specified as a scalar.

Data Types: `double`

Output Arguments

A — State matrix

matrix

State matrix, returned as a matrix. If the system is described by *n* state variables, then `A` is *n*-by-*n*.

Data Types: `single` | `double`

`B` — Input-to-state matrix
matrix

Input-to-state matrix, returned as a matrix. If the system is described by *n* state variables, then `B` is *n*-by-1.

Data Types: `single` | `double`

`C` — State-to-output matrix
matrix

State-to-output matrix, returned as a matrix. If the system has *q* outputs and is described by *n* state variables, then `C` is *q*-by-*n*.

Data Types: `single` | `double`

`D` — Feedthrough matrix
matrix

Feedthrough matrix, returned as a matrix. If the system has *q* outputs, then `D` is *q*-by-1.

Data Types: `single` | `double`

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`sos2ss` | `ss2zp` | `tf2ss` | `zp2sos` | `zp2tf`

Introduced before R2006a

zp2tf

Convert zero-pole-gain filter parameters to transfer function form

Syntax

```
[b,a] = zp2tf(z,p,k)
```

Description

`[b,a] = zp2tf(z,p,k)` converts a factored transfer function representation

$$H(s) = \frac{Z(s)}{P(s)} = k\frac{(s-z_1)(s-z_2)\cdots(s-z_m)}{(s-p_1)(s-p_2)\cdots(s-p_n)}$$

of a single-input/multi-output (SIMO) system to a polynomial transfer function representation

$$\frac{B(s)}{A(s)} = \frac{b_1 s^{(n-1)} + \cdots + b_{(n-1)}s + b_n}{a_1 s^{(m-1)} + \cdots + a_{(m-1)}s + a_m}.$$

Examples

Transfer Function of Mass-Spring System

Compute the transfer function of a damped mass-spring system that obeys the differential equation

$$\ddot{w} + 0.01\dot{w} + w = u(t).$$

The measurable quantity is the acceleration, $y = \ddot{w}$, and $u(t)$ is the driving force. In Laplace space, the system is represented by

$$Y(s) = \frac{s^2\, U(s)}{s^2 + 0.01s + 1}.$$

The system has unit gain, a double zero at $s = 0$, and two complex-conjugate poles.

```
k = 1;
z = [0 0]';
p = roots([1 0.01 1])
```

```
p = 2×1 complex

  -0.0050 + 1.0000i
  -0.0050 - 1.0000i
```

Use `zp2tf` to find the transfer function.

```
[b,a] = zp2tf(z,p,k)
```

```
b = 1×3
```

```
     1     0     0

a = 1×3

    1.0000    0.0100    1.0000
```

Input Arguments

z — Zeros

column vector | matrix

Zeros of the system, specified as a column vector or a matrix. `z` has as many columns as there are outputs. The zeros must be real or come in complex conjugate pairs. Use `Inf` values as placeholders in `z` if some columns have fewer zeros than others.

Example: `[1 (1+1j)/2 (1-1j)/2]'`

Data Types: `single` | `double`

p — Poles

column vector

Poles of the system, specified as a column vector. The poles must be real or come in complex conjugate pairs.

Example: `[1 (1+1j)/2 (1-1j)/2]'`

Data Types: `single` | `double`

k — Gains

column vector

Gains of the system, specified as a column vector.

Example: `[1 2 3]'`

Data Types: `single` | `double`

Output Arguments

b — Transfer function numerator coefficients

row vector | matrix

Transfer function numerator coefficients, returned as a row vector or a matrix. If `b` is a matrix, then it has a number of rows equal to the number of columns of `z`.

a — Transfer function denominator coefficients

row vector

Transfer function denominator coefficients, returned as a row vector.

Algorithms

The system is converted to transfer function form using `poly` with `p` and the columns of `z`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`sos2tf` | `ss2tf` | `tf2zp` | `tf2zpk` | `zp2sos` | `zp2ss`

Introduced before R2006a

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('highpassiir','FilterOrder',8, ...
         'PassbandFrequency',75e3,'PassbandRipple',0.2, ...
         'SampleRate',200e3);
[z,p,k] = zpk(hpFilt)
```

```
z = 8×1

     1
     1
     1
     1
     1
     1
     1
     1


p = 8×1 complex

  -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 0.5π 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`

Introduced in R2014a

zplane

Zero-pole plot for discrete-time systems

Syntax

```
zplane(z,p)
zplane(b,a)
[hz,hp,ht] = zplane( ___ )

zplane(d)
[vz,vp,vk] = zplane(d)
```

Description

`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 matrices, then `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 the numerator coefficients `b` and the denominator coefficients `a`.

`[hz,hp,ht] = zplane( ___ )` 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.

`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**.

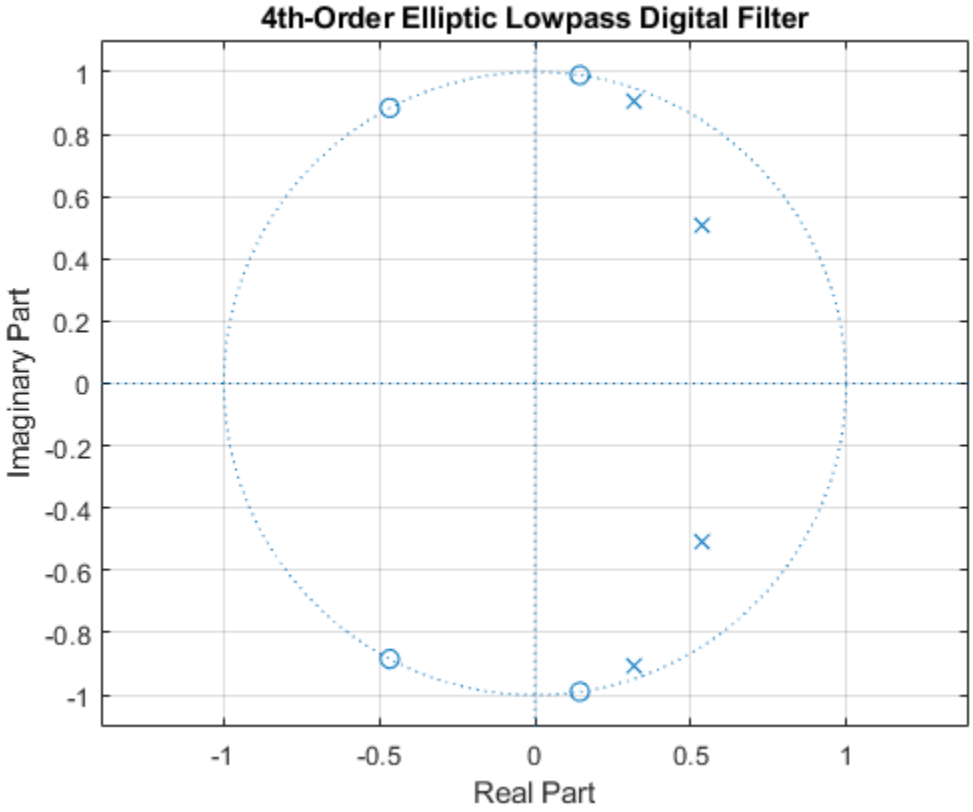
`[vz,vp,vk] = zplane(d)` returns the zeros (vector `vz`), poles (vector `vp`), and gain (scalar `vk`) corresponding to the digital filter `d`.

Examples

Poles and Zeros of Elliptic Lowpass Filter

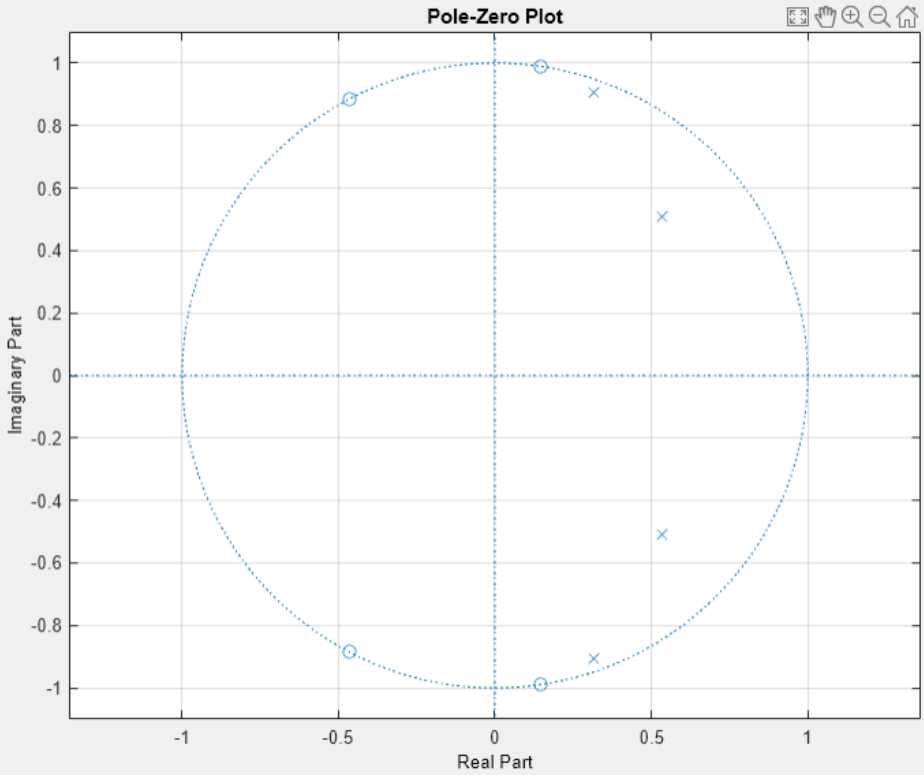
For data sampled at 1000 Hz, plot the poles and zeros of a 4th-order elliptic lowpass digital filter with cutoff frequency 200 Hz, 3 dB of ripple in the passband, and 30 dB of attenuation in the stopband.

```
[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 `zplane` to plot the poles and zeros. Note that this syntax of `zplane` calls `fvtool`.

```
d = designfilt('lowpassiir','FilterOrder',4,'PassbandFrequency',200, ...
               'PassbandRipple',3,'StopbandAttenuation',30, ...
               'DesignMethod','ellip','SampleRate',1000);
zplane(d)
```



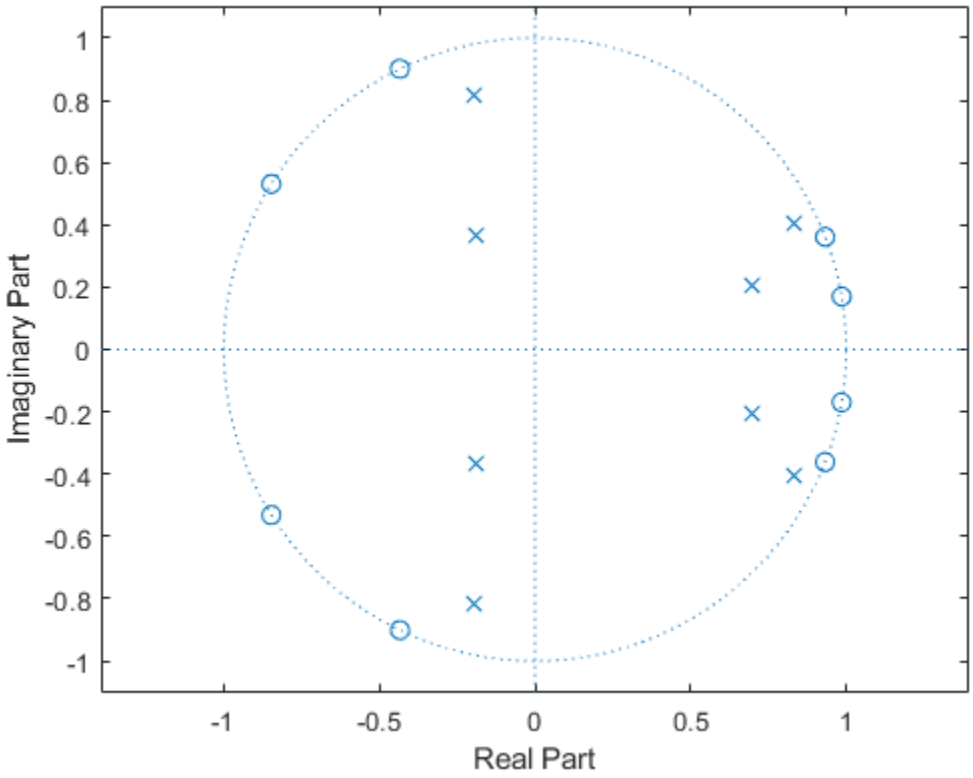
Zeros and Poles of Transfer Function

Design an 8th-order Chebyshev Type II bandpass filter with a stopband attenuation of 20 dB. Specify the stopband edge frequencies as $\pi/8$ rad/sample and $5\pi/8$ rad/sample.

```
[b,a] = cheby2(8/2,20,[1 5]/8);
```

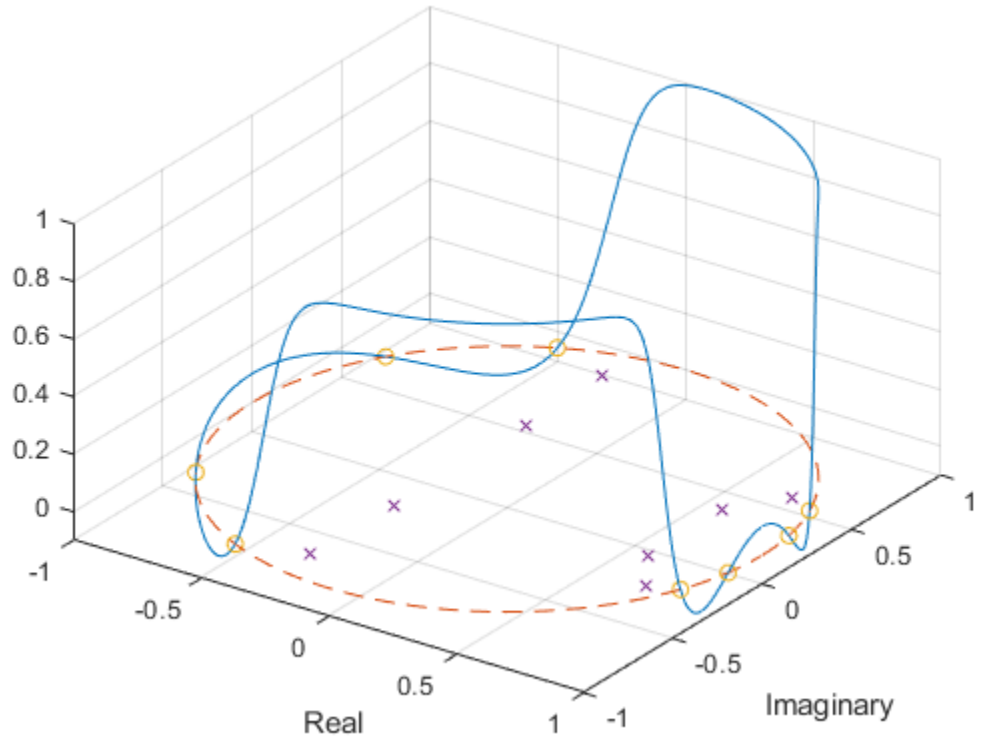
Use `zplane` to plot the poles and zeros of the transfer function.

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



Visualize the zero-phase response of the filter. Overlay the unit circle and the pole and zero locations.

```
[hw,fw] = zerophase(b,a,1024,"whole");

z = roots(b);
p = roots(a);

plot3(cos(fw),sin(fw),hw)
hold on
plot3(cos(fw),sin(fw),zeros(size(fw)),'--')
plot3(real(z),imag(z),zeros(size(z)),'o')
plot3(real(p),imag(p),zeros(size(p)),'x')
hold off
xlabel("Real")
ylabel("Imaginary")
view(35,40)
grid
```



Input Arguments

z, p — Zeros and poles

column vectors | matrices

Zeros and poles, specified as column vectors or matrices. If `z` and `p` are matrices, then `zplane` plots the poles and zeros in the columns of `z` and `p` in different colors.

Data Types: `single` | `double`
Complex Number Support: Yes

b, a — Transfer function coefficients

row vectors

Transfer function coefficients, specified as row vectors. The transfer function is defined in terms of z^{-1}:

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

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 0.5π rad/sample.

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 0.5π rad/sample.

Output Arguments

hz, hp, ht — Vectors of handles
vectors

Vectors of handles to the zero lines, `hz`, and the pole lines, `hp`, of the pole-zero plot. `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, `[]`.

vz, vp, vk — Zeros, poles, and gain
column vectors and scalar

Zeros, poles, and gain of a digital filter, `d`, returned as column vectors and a scalar.

Tips

- You can override the automatic scaling of `zplane` using

  ```
  axis([xmin xmax ymin ymax])
  ```

  after calling `zplane`. This scaling is useful when one or more zeros or poles have such a large magnitude that the others are grouped tightly around the origin and are hard to distinguish.

See Also

Apps
Filter Designer

Functions
`designfilt` | `digitalFilter` | `freqz`

Topics
“Speaker Crossover Filters”

Introduced before R2006a