<table>
<thead>
<tr>
<th>Year</th>
<th>Edition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988</td>
<td>First printing</td>
<td>New</td>
</tr>
<tr>
<td>November 1997</td>
<td>Second printing</td>
<td>Revised</td>
</tr>
<tr>
<td>January 1998</td>
<td>Third printing</td>
<td>Revised</td>
</tr>
<tr>
<td>September 2000</td>
<td>Fourth printing</td>
<td>Revised for Version 5.0 (Release 12)</td>
</tr>
<tr>
<td>July 2002</td>
<td>Fifth printing</td>
<td>Revised for Version 6.0 (Release 13)</td>
</tr>
<tr>
<td>December 2002</td>
<td>Online only</td>
<td>Revised for Version 6.1 (Release 13+)</td>
</tr>
<tr>
<td>June 2004</td>
<td>Online only</td>
<td>Revised for Version 6.2 (Release 14)</td>
</tr>
<tr>
<td>October 2004</td>
<td>Online only</td>
<td>Revised for Version 6.2.1 (Release 14SP1)</td>
</tr>
<tr>
<td>March 2005</td>
<td>Online only</td>
<td>Revised for Version 6.2.1 (Release 14SP2)</td>
</tr>
<tr>
<td>September 2005</td>
<td>Online only</td>
<td>Revised for Version 6.4 (Release 14SP3)</td>
</tr>
<tr>
<td>March 2006</td>
<td>Sixth printing</td>
<td>Revised for Version 6.5 (Release 2006a)</td>
</tr>
<tr>
<td>September 2006</td>
<td>Online only</td>
<td>Revised for Version 6.6 (Release 2006b)</td>
</tr>
<tr>
<td>March 2007</td>
<td>Online only</td>
<td>Revised for Version 6.7 (Release 2007a)</td>
</tr>
<tr>
<td>September 2007</td>
<td>Online only</td>
<td>Revised for Version 6.8 (Release 2007b)</td>
</tr>
<tr>
<td>March 2008</td>
<td>Online only</td>
<td>Revised for Version 6.9 (Release 2008a)</td>
</tr>
<tr>
<td>October 2008</td>
<td>Online only</td>
<td>Revised for Version 6.10 (Release 2008b)</td>
</tr>
<tr>
<td>March 2009</td>
<td>Online only</td>
<td>Revised for Version 6.11 (Release 2009a)</td>
</tr>
<tr>
<td>September 2009</td>
<td>Online only</td>
<td>Revised for Version 6.12 (Release 2009b)</td>
</tr>
<tr>
<td>March 2010</td>
<td>Online only</td>
<td>Revised for Version 6.13 (Release 2010a)</td>
</tr>
<tr>
<td>September 2010</td>
<td>Online only</td>
<td>Revised for Version 6.14 (Release 2010b)</td>
</tr>
<tr>
<td>April 2011</td>
<td>Online only</td>
<td>Revised for Version 6.15 (Release 2011a)</td>
</tr>
<tr>
<td>September 2011</td>
<td>Online only</td>
<td>Revised for Version 6.16 (Release 2011b)</td>
</tr>
<tr>
<td>March 2012</td>
<td>Online only</td>
<td>Revised for Version 6.17 (Release 2012a)</td>
</tr>
<tr>
<td>September 2012</td>
<td>Online only</td>
<td>Revised for Version 6.18 (Release 2012b)</td>
</tr>
<tr>
<td>March 2013</td>
<td>Online only</td>
<td>Revised for Version 6.19 (Release 2013a)</td>
</tr>
<tr>
<td>September 2013</td>
<td>Online only</td>
<td>Revised for Version 6.20 (Release 2013b)</td>
</tr>
<tr>
<td>March 2014</td>
<td>Online only</td>
<td>Revised for Version 6.21 (Release 2014a)</td>
</tr>
<tr>
<td>October 2014</td>
<td>Online only</td>
<td>Revised for Version 6.22 (Release 2014b)</td>
</tr>
<tr>
<td>March 2015</td>
<td>Online only</td>
<td>Revised for Version 7.0 (Release 2015a)</td>
</tr>
<tr>
<td>September 2015</td>
<td>Online only</td>
<td>Revised for Version 7.1 (Release 2015b)</td>
</tr>
<tr>
<td>March 2016</td>
<td>Online only</td>
<td>Revised for Version 7.2 (Release 2016a)</td>
</tr>
<tr>
<td>September 2016</td>
<td>Online only</td>
<td>Revised for Version 7.3 (Release 2016b)</td>
</tr>
<tr>
<td>March 2017</td>
<td>Online only</td>
<td>Revised for Version 7.4 (Release 2017a)</td>
</tr>
<tr>
<td>September 2017</td>
<td>Online only</td>
<td>Revised for Version 7.5 (Release 2017b)</td>
</tr>
<tr>
<td>March 2018</td>
<td>Online only</td>
<td>Revised for Version 8.0 (Release 2018a)</td>
</tr>
<tr>
<td>September 2018</td>
<td>Online only</td>
<td>Revised for Version 8.1 (Release 2018b)</td>
</tr>
<tr>
<td>March 2019</td>
<td>Online only</td>
<td>Revised for Version 8.2 (Release 2019a)</td>
</tr>
<tr>
<td>September 2019</td>
<td>Online only</td>
<td>Revised for Version 8.3 (Release 2019b)</td>
</tr>
</tbody>
</table>
Functions — Alphabetical List
ac2poly

Convert autocorrelation sequence to prediction polynomial

Syntax

\[ a = \text{ac2poly}(r) \]
\[ [a, \text{efinal}] = \text{ac2poly}(r) \]

Description

\[ a = \text{ac2poly}(r) \] 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 \( r \).

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

Examples

Prediction Polynomial from Autocorrelation Sequence

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

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

\[ [a, \text{efinal}] = \text{ac2poly}(r) \]

\( a \) is a 1x6 array:

\[
\begin{array}{cccccc}
1.0000 & 0.6147 & 0.9898 & 0.0004 & 0.0034 & -0.0077 \\
\end{array}
\]

\( \text{efinal} = 0.1791 \)
Tips
You can apply this function to real or complex data.

References

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


See Also

ac2poly | poly2rc | rc2ac

Introduced before R2006a
**addLabelDefinitions**

Add label definitions to labeled signal set

**Syntax**

```matlab
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.

```matlab
load whales
lss
```

```matlab
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.

```matlab
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.

```matlab
addLabelDefinitions(lss,calf)
getLabelNames(lss)
ans = 4x1 string array
    "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'.

```matlab
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","lo
gical") 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.

```matlab
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.

```matlab
addMembers(lss,song2)
```
```
ls
```
```
ls =
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 = 4x1
```
```
    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` | `timetable` | `audio datastore`

Input data source, specified as a matrix, a timetable, or an audio datastore. The particular form of `src` depends on the “Source” property of `lss`.

- If “Source” 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 an audio datastore, then add members by setting src as another audio 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.

\textbf{tinfo — Time information for new members}

\begin{verbatim}
scalar | vector | matrix | duration scalar | duration vector
\end{verbatim}

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” property to 'sampleRate’, 'sampleTime’, or 'timeValues’ to make lss interpret \( tinfo \) correctly.

Example: \( \text{addMembers}(ks, \{\text{randn}(10,5), \text{randn}(10,3)\}, \text{seconds}([1 2])) \) adds two new members with different time information to \( ks = \) labeledSignalSet(\text{randn}(10,3), 'SampleTime', \text{seconds}(1)).

Example: \( \text{addMembers}(ks, \{\text{randn}(10,5), \text{randn}(10,3)\}, [1:10; 2:2:20]) \) adds two new members with different time information to \( ks = \) labeledSignalSet(\text{randn}(10,3), 'TimeValues', 1:10).

\textbf{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: \( \text{labeledSignalSet} \{\text{randn}(100,1) \text{ randn}(10,1)\}, 'MemberNames', \{'llama' 'alpaca'\}) \) specifies a set of random signals with two members, 'llama' and 'alpaca'.

\textbf{See Also}

Signal Labeler | labeledSignalSet | signalLabelDefinition

\textbf{Introduced in R2018b}
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 $X_a$ consists of $L_X$ zeros. All samples of $X$ are lost.
• If $-D \geq L_Y$, then $Y_a$ 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 `[]`.

```matlab
[Xa,Ya,D] = alignsignals( ___ )
```

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.

```matlab
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.

\[
X = [1 \ 2 \ 3]; \\
Y = [0 \ 0 \ 1 \ 2 \ 3]; \\
\text{maxlag} = 2;
\]

\[
[Xa,Ya,D] = \text{alignsignals}(X,Y,\text{maxlag})
\]

\[
Xa = 1\times5 \\
\begin{array}{c}
0 \\
0 \\
1 \\
2 \\
3 \\
\end{array}
\]

\[
Ya = 1\times5 \\
\begin{array}{c}
0 \\
0 \\
1 \\
2 \\
3 \\
\end{array}
\]

\[
D = 2
\]

**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.

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

\[
[Xa,Ya,D] = \text{alignsignals}(X,Y)
\]

\[
Xa = 1\times6 \\
\begin{array}{c}
0 \\
0 \\
1 \\
2 \\
3 \\
0 \\
\end{array}
\]

\[
Ya = 1\times6 \\
\begin{array}{c}
0.0200 \\
0.1200 \\
1.0800 \\
2.2100 \\
2.9500 \\
-0.0900
\end{array}
\]
\[ D = 0 \]

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.

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

\[
[Xa,Ya,D] = \text{alignsignals}(X,Y,[],'truncate')
\]

\[
Xa = 1\times3
\]

\[
\begin{array}{ccc}
0 & 0 & 1 \\
\end{array}
\]

\[
Ya = 1\times5
\]

\[
\begin{array}{cccc}
0 & 0 & 1 & 2 \\
\end{array}
\]

\[
D = 2
\]

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.

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

\[
[Xa,Ya,D] = \text{alignsignals}(X,Y,[],'truncate')
\]

Warning: All original data in the first input \( X \) has been truncated because the length of 

\[
Xa = 1\times3
\]
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.

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

\([Xa,Ya,D] = \text{alignsignals}(X,Y)\)

\(Xa = 1\times4\)

\[
0 \quad 1 \quad 2 \quad 3
\]

\(Ya = 1\times13\)

\[
0 \quad 1 \quad 2 \quad 3 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 2 \quad 3 \quad 0 \quad 0
\]

\(D = -1\)

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

\( \text{maxlag} \) — Maximum window size or lag
scalar integer | []

Maximum window size, or lag, specified as an integer-valued scalar. By default, \( \text{maxlag} \) is equal to \( \text{max}(\text{length}(X), \text{length}(Y)) - 1 \). If \( \text{maxlag} \) is input as [], it is replaced by the default value. If \( \text{maxlag} \) is negative, it is replaced by its absolute value. If \( \text{maxlag} \) is not integer-valued, or is complex, Inf, or NaN, then \text{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, \( X_a \). If input argument \( Y \) is a row vector, then \( Y_a \) is also a row vector. If input argument \( Y \) is a column vector, then \( Y_a \) is also a column vector. If you specify the ‘truncate’ option and the estimated delay \( D \) is negative, then \( Y_a \) 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 \( \text{maxlag} \), then \( D \) must be less than or equal to \( \text{maxlag} \).

**Algorithms**

- You can find the theory on delay estimation in the specification of the \texttt{finddelay} function (see “Algorithms” on page 1-868).
- The \texttt{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 \texttt{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].

**References**

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

\[ a = \text{arburg}(x,p) \]
\[ [a,e] = \text{arburg}(x,p) \]
\[ [a,e,rc] = \text{arburg}(x,p) \]

Description

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

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

\[ [a,e,rc] = \text{arburg}(x,p) \] returns the reflection coefficients in \( rc \).

Examples

Parameter Estimation Using Burg’s Method

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

\[ \text{rng default} \]

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

\[ y = \text{filter}(1,A,0.2*\text{randn}(1024,1)); \]
arcoeffs = arburg(y,4)
arcoeffs = 1x5
    1.0000   -2.7743    3.8408   -2.6843    0.9360

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

nrealiz = 50;

noisestdz = rand(1,nrealiz)+0.5;

randnoise = randn(1024,nrealiz);

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

plot(noisestdz.^2,noisevar,'*')
title('Noise Variance')
xlabel('Input')
ylabel('Estimated')
Repeat the procedure using `arburg`'s multichannel syntax.

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

AR(p) Model

In an AR model of order $p$, the current output is a linear combination of the past $p$ outputs plus a white noise input. The weights on the $p$ past outputs minimize the mean-square prediction error of the autoregression. If $y(n)$ is the current value of the output and $x(n)$ is a zero mean white noise input, the AR($p$) model is:
\[ y(n) + \sum_{k=1}^{p} a(k)y(n-k) = x(n). \]

**Reflection Coefficients**

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

**Algorithms**

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

**References**


**See Also**

arccov | armcov | aryule | levinson | lpc

**Topics**

“Parametric Modeling”

**Introduced before R2006a**
ar cov

Autoregressive all-pole model parameters — covariance method

Syntax

\[ a = \text{arcov}(x, p) \]
\[ [a, e] = \text{arcov}(x, p) \]

Description

\( a = \text{arcov}(x, p) \) uses the covariance method to fit a \( p \)th-order autoregressive (AR) model to the input signal, \( x \), which is assumed to be the output of an AR system driven by white noise. This method minimizes the forward prediction error in the least-squares sense. The output array, \( a \), contains normalized estimates of the AR system parameters, \( A(z) \), in descending powers of \( z \). \( a \) has \( p + 1 \) columns. If \( x \) is a vector, then \( a \) is a row vector. If \( a \) is a matrix, then the coefficients along the \( n \)th row of \( a \) model the \( n \)th column of \( x \).

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

Examples

Parameter Estimation Using the Covariance Method

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

\[ \text{rng default} \]

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

\[ y = \text{filter}(1, A, 0.2*\text{randn}(1024, 1)); \]
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);

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 `arcov`'s multichannel syntax.

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

AR(p) Model

Let $y(n)$ be a wide-sense stationary random process obtained by filtering white noise of variance $e$ with the system function $A(z)$. If $P_y(e^{i\omega})$ is the power spectral density of $y(n)$, then
\[ P_y(e^{j\omega}) = \frac{e}{|A(e^{j\omega})|^2} = \frac{e}{1 + \sum_{k=1}^{p} a(k)e^{-j\omega k}}. \]

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

**See Also**

arburg | armcov | aryule | lpc | pcov | prony

*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) uses the modified covariance method to fit a pth-order autoregressive (AR) model to the input signal, x, which is assumed to be the output of an AR system driven by white noise. This method minimizes the forward and backward prediction errors in the least-squares sense. The output array, a, contains the normalized estimates of the AR system parameters, A(z), in descending powers of z. a has p + 1 columns. If x is a vector, then a is a row vector. If a is a matrix, then the coefficients along the nth row of a model the nth column of x.

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

Examples

Parameter Estimation Using the Modified Covariance Method

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

rng default

A = [1 -2.7607 3.8106 -2.6535 0.9238];
y = filter(1,A,0.2*randn(1024,1));
arcoeffs = armcov(y,4)
arcoeffs = 1×5
    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);

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

plot(noisestdz.^2,noisevar,'*')
title('Noise Variance')
xlabel('Input')
ylabel('Estimated')
Repeat the procedure using armcov's multichannel syntax.

```matlab
realiz = bsxfun(@times,noisestdz,randnoise);
Y = filter(1,A,realiz);

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

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

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

AR(p) Model

Let $y(n)$ be a wide-sense stationary random process obtained by filtering a white noise input with variance $e$ with the system function $A(z)$. If $P_y(e^{j\omega})$ is the power spectral density of $y(n)$, then
\[ P_y(e^{j\omega}) = \frac{e}{|A(e^{j\omega})|^2} = \frac{e}{1 + \sum_{k=1}^{p} a(k)e^{-j\omega k}}. \]

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

**See Also**

arburg | arcov | aryule | lpc | pmcov | prony

*Introduced before R2006a*
aryule

Autoregressive all-pole model parameters — Yule-Walker method

Syntax

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

Description

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

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

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

Examples

Parameter Estimation Using the Yule-Walker Method

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

\[ \text{rng default} \]
\[ A = [1 -2.7607 3.8106 -2.6535 0.9238]; \]
\[ y = \text{filter}(1,A,0.2*\text{randn}(1024,1)); \]
arcoeffs = aryule(y,4)
arcoeffs = 1x5
    1.0000  -2.7262   3.7296  -2.5753   0.8927

Generate 50 realizations of the process, changing each time the variance of the input noise. Compare the Yule-Walker-estimated variances to the actual values.
nrealiz = 50;
noisestdz = rand(1,nrealiz)+0.5;
randnoise = randn(1024,nrealiz);

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

plot(noisestdz.^2,noisevar,'*')
title('Noise Variance')
xlabel('Input')
ylabel('Estimated')
Repeat the procedure using *aryule*’s multichannel syntax.

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

```matlab
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.
[ar_coeffs,NoiseVariance,reflect_coeffs] = aryule(y,10);

stem(reflect_coeffs)
axis([-0.05 10.5 -1 1])
title('Reflection Coefficients by Lag')

The reflection coefficients decay to zero after lag 2, which indicates that an AR(10) model significantly overestimates the time dependence in the data.
More About

AR(p) Model

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

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

Reflection Coefficients

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

Algorithms

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

References


See Also

arburg|arcov|armcov|levinson|lpc

Topics

“Parametric Modeling”
Introduced before R2006a
**bandpass**

Bandpass-filter signals

**Syntax**

\[
y = \text{bandpass}(x,\text{wpass})
\]
\[
y = \text{bandpass}(x,f\text{pass},fs)
\]
\[
y = \text{bandpass}(xt,f\text{pass})
\]
\[
y = \text{bandpass}(\_,\text{Name},\text{Value})
\]
\[
[y,d] = \text{bandpass}(\_)
\]
\[
\text{bandpass}(\_)
\]

**Description**

\[
y = \text{bandpass}(x,\text{wpass})
\] filters the input signal \(x\) using a bandpass filter with a passband frequency range specified by the two-element vector \(\text{wpass}\) and expressed in normalized units of \(\pi\) rad/sample. \text{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.

\[
y = \text{bandpass}(x,\text{fpass},fs)
\] specifies that \(x\) is sampled at a rate of \(fs\) hertz. The two-element vector \(\text{fpass}\) specifies the passband frequency range of the filter in hertz.

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

\[
y = \text{bandpass}(\_,\text{Name},\text{Value})
\] specifies additional options for any of the previous syntaxes using name-value pair arguments. You can change the stopband attenuation, the transition band steepness, and the type of impulse response of the filter.

\[
[y,d] = \text{bandpass}(\_)
\] also returns the \text{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.

```matlab
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.

```matlab
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.

```matlab
fs = 2e3;
t = 0:1/fs:0.3-1/fs;
[l; m; h] = [0 130.81 146.83 164.81 174.61 196.00 220 246.94; 0 261.63 293.66 329.63 349.23 392.00 440 493.88; 0 523.25 587.33 659.25 698.46 783.99 880 987.77];
```

![Bandpass Filtering (Fpass = [100 200] Hz)](image)

![Power Spectrum (dB)](image)
note = @(f,g) [1 1 1]*sin(2*pi*[l(g) m(g) h(f)]'.*t);
mel = [3 2 1 2 3 3 0 2 2 2 0 3 5 5 0 3 2 1 2 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.

\[
\text{pong} = \text{bandpass}(\text{song}, [230, 450], \text{fs})
\]

% To hear, type sound(pong, fs)

\[
\text{bandpass}(\text{song}, [230, 450], \text{fs})
\]
Plot the spectrogram of the middle register.

```matlab
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.

```matlab
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);
```
Compute and plot the frequency responses of the filters.

```matlab
[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.

```matlab
[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', ...
```
Make the filters asymmetric by specifying different values of steepness at the lower and higher passband frequencies.

```matlab
[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]);
ppectrum([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: \( \sin(2\pi(0:127)/16) + \text{randn}(1,128)/100 \) 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 $(0, \frac{f_s}{2})$

Passband frequency range, specified as a two-element vector with elements in the interval $(0, \frac{f_s}{2})$.

**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” (MATLAB).

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 comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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 \texttt{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 1-59 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(d)` 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 \( 1 \times \pi \text{ rad/sample} \) 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 \( f_{pass}^{lower} - f_{stop}^{lower} \), where \( f_{pass}^{lower} \) is the first element of \( f_{pass} \).

• The upper transition width of the filter, \( W^{upper} \), is \( f_{stop}^{upper} - f_{pass}^{upper} \), where \( f_{pass}^{upper} \) is the second element of \( f_{pass} \).

• 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\text{lower}, s\text{upper}], or a scalar. When you specify 'Steepness' as a vector, the function:

- Computes the lower transition width as
  \[ W_{\text{lower}} = (1 - s_{\text{lower}}) \times f_{\text{pass}}_{\text{lower}}. \]

- When the first element of 'Steepness' is equal to 0.5, the transition width is 50% of \( f_{\text{pass}}_{\text{lower}} \).

- As the first element of 'Steepness' approaches 1, the transition width becomes progressively narrower until it reaches a minimum value of 1% of \( f_{\text{pass}}_{\text{lower}} \).

- Computes the upper transition width as
  \[ W_{\text{upper}} = (1 - s_{\text{upper}}) \times (f_{\text{Nyquist}} - f_{\text{pass}}_{\text{upper}}). \]
• When the second element of 'Steepness' is equal to 0.5, the transition width is 50% of \((f_{\text{Nyquist}} - f_{\text{pass upper}})\).

• As the second element of 'Steepness' approaches 1, the transition width becomes progressively narrower until it reaches a minimum value of 1% of \((f_{\text{Nyquist}} - f_{\text{pass upper}})\).

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 | firl | highpass | lowpass

Introduced in R2018a
**bandpower**

Band power

**Syntax**

\[
p = \text{bandpower}(x)
\]

\[
p = \text{bandpower}(x,fs,freqrange)
\]

\[
p = \text{bandpower}(pxx,f,'psd')
\]

\[
p = \text{bandpower}(pxx,f,freqrange,'psd')
\]

**Description**

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

\( p = \text{bandpower}(x,fs,freqrange) \) 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. \text{bandpower} uses a modified periodogram to determine the average power in \( freqrange \).

\( p = \text{bandpower}(pxx,f,'psd') \) 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.

\( p = \text{bandpower}(pxx,f,freqrange,'psd') \) 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 $\ell_2$ norm.

```matlab
>> 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.

```matlab
>> 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.

```matlab
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.

```matlab
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)
```
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.

$Fs = 1000;$
$t = 0:1/Fs:1-1/Fs;$
$f = [100;200;300];$
$x = \cos(2\times\pi\times f\times t)'+\text{randn(length}(t),3);$

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

$p = \text{bandpower}(x)$
$p = 1\times 3$
\[
\begin{array}{ccc}
1.5264 & 1.5382 & 1.4717 \\
\end{array}
\]

$l2norm = \text{dot}(x,x)/\text{length}(x)$
$l2norm = 1\times 3$
\[
\begin{array}{ccc}
1.5264 & 1.5382 & 1.4717 \\
\end{array}
\]

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

Example: \( \cos(\pi/\begin{bmatrix} 4; 2 \end{bmatrix}*(0:159))' + \text{randn}(160,2) \) 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, \( \text{freqrange} \) must be within the following intervals:

- \([0, \text{fs}/2]\) if \( x \) is real-valued and \( N \) is even
- \([0, (N - 1)\text{fs}/(2N)]\) if \( x \) is real-valued and \( N \) is odd
- \([- (N - 2)\text{fs}/(2N), \text{fs}/2]\) if \( x \) is complex-valued and \( N \) is even
- \([- (N - 1)\text{fs}/(2N), (N - 1)\text{fs}/(2N)]\) 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 \text{db2pow} to convert decibel values to power values.

Example: \([\text{pxx},f] = \text{periodogram}(\cos(\pi/\begin{bmatrix} 4; 2 \end{bmatrix}*(0:159))' + \text{randn}(160,2))\) specifies the periodogram PSD estimate of a noisy two-channel sinusoid sampled at \( 2\pi \) 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

See Also

periodogram | sfdr

Introduced in R2013a
bandstop

Bandstop-filter signals

Syntax

\[
y = \text{bandstop}(x,\text{wpass}) \\
y = \text{bandstop}(x,\text{fpass},fs) \\
y = \text{bandstop}(\text{xt},\text{fpass}) \\
y = \text{bandstop}(\_,\text{Name},\text{Value}) \\
[y,d] = \text{bandstop}(\_) \\
\text{bandstop}(\_)
\]

Description

\([y,\text{wpass})\) filters the input signal \(x\) using a bandstop filter with a stopband frequency range specified by the two-element vector \(\text{wpass}\) and expressed in normalized units of \(\pi\) 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.

\([y,\text{fpass},fs)\) specifies that \(x\) is sampled at a rate of \(fs\) hertz. The two-element vector \(\text{fpass}\) specifies the stopband frequency range of the filter in hertz.

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

\([y,\_\_,\text{Name},\text{Value})\) specifies additional options for any of the previous syntaxes using name-value pair arguments. You can change the stopband attenuation, the transition band steepness, and the type of impulse response of the filter.

\([y,d] = \text{bandstop}(\_\_,\_\_,\_\_,\_\_)\) also returns the \(\text{digitalFilter}\) 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.

```matlab
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];
```

![Bandstop Filtering](image)
note = @(f,g) [1 1 1]*sin(2*pi*[l(g) m(g) h(f)]'.*t);

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

```matlab
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.

```matlab
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);
```
Compute and plot the frequency responses of the filters.

```matlab
[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.

```matlab
[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', ...
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.

\[
\begin{align*}
[h1,f] &= \text{freqz}(d1,1024,fs); \\
[h2,~] &= \text{freqz}(d2,1024,fs); \\
[h3,~] &= \text{freqz}(d3,1024,fs);
\end{align*}
\]

\[
\begin{align*}
\text{plot}(f,\text{mag2db(abs([h1 \ h2 \ h3]))}) \\
\text{ylim([-120 \ 20])}
\end{align*}
\]
**Input Arguments**

*x* — Input signal
type: 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, \(f_s/2\))

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

**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” (MATLAB).

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 comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: `'ImpulseResponse','iir','StopbandAttenuation',30` filters the input using a minimum-order IIR filter that attenuates by 30 dB the frequencies from \(f_{pass}(1)\) to \(f_{pass}(2)\).
**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 1-82 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(d)` 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 $f_{stop\, lower} - f_{pass\, lower}$.

• The upper transition width of the filter, $W_{upper}$, is $f_{pass\, upper} - f_{stop\, upper}$.

• 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
  \[ W_{lower} = (1 - s_{lower}) \times (f_{center} - f_{pass}^{lower}). \]
- When the first element of 'Steepness' is equal to 0.5, the transition width is 50% of \((f_{center} - f_{pass}^{lower})\).
- As the first element of 'Steepness' approaches 1, the transition width becomes progressively narrower until it reaches a minimum value of 1% of \((f_{center} - f_{pass}^{lower})\).
- Computes the upper transition width as
\[ W_{\text{upper}} = (1 - s_{\text{upper}}) \times (f_{\text{pass}}^{\text{upper}} - f_{\text{center}}). \]

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

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**

\[ w = \text{barthannwin}(L) \]

**Description**

\[ w = \text{barthannwin}(L) \] returns an \( L \)-point modified Bartlett-Hann window in the column vector \( w \). Like Bartlett, Hann, and Hamming windows, this window has a mainlobe at the origin and asymptotically decaying sidelobes on both sides. It is a linear combination of weighted Bartlett and Hann windows with near sidelobes lower than both Bartlett and Hann and with far sidelobes lower than both Bartlett and Hamming windows. The mainlobe width of the modified Bartlett-Hann window is not increased relative to either Bartlett or Hann window mainlobes.

**Note** The Hann window is also called the Hanning window.

**Examples**

**Bartlett-Hann Window**

Create a 64-point Bartlett-Hann window. Display the result using \texttt{wvtool}.

\[
\begin{align*}
L &= 64; \\
\text{wvtool(barthannwin(L))}
\end{align*}
\]
**Algorithms**

The equation for computing the coefficients of a Modified Bartlett-Hanning window is

$$w(n) = 0.62 - 0.48\left|\left(\frac{n}{N} - 0.5\right)\right| + 0.38\cos\left(2\pi\left(\frac{n}{N} - 0.5\right)\right)$$

where $0 \leq n \leq N$ and the window length is $L = N + 1$. 
References


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

\[
w = \text{bartlett}(L)\]

Description

\[w = \text{bartlett}(L)\] returns an \(L\)-point symmetric Bartlett window.

Examples

Bartlett Window

Create a 64-point Bartlett window. Display the result using \texttt{wvtool}.

\[
\begin{align*}
L &= 64; \\
bw &= \text{bartlett}(L); \\
wvtool(bw)
\end{align*}
\]
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:

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

The window length \( L = N + 1 \).

The Bartlett window is very similar to a triangular window as returned by the \texttt{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 \texttt{bartlett(L)} are equivalent to \texttt{triang(L-2)}.

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

## References


## 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. \( n \) must be less than or equal to 25. The function returns the poles in the length \( n \) column vector \( p \) and the gain in scalar \( k \). \( z \) is an empty matrix because there are no zeros. The transfer function is

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

*besselap* normalizes the poles and gain so that at low frequency and high frequency the Bessel prototype is asymptotically equivalent to the Butterworth prototype of the same order [1]. The magnitude of the filter is less than \( 1/\sqrt{2} \) at the unity cutoff frequency \( \Omega_c = 1 \).

Analog Bessel filters are characterized by a group delay that is maximally flat at zero frequency and almost constant throughout the passband. The group delay at zero frequency is

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

**Examples**
Frequency Response of an Analog Bessel Filter

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

```plaintext
[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
```
Algorithms

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

References


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\textsuperscript{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 2n.

[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`.

```matlab
[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.

```matlab
[h,w] = freqs(b,a,1000);
grpdelay = diff(unwrap(angle(h)))./diff(w);
```
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],\text{'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.

Data Types: double

\( \text{Wo} \) — 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 \texttt{besself}\ designs a lowpass or highpass filter with cutoff frequency \(Wo\).
- If \(Wo\) is a two-element vector \([w1 \ w2]\), where \(w1 < w2\), then \texttt{besself}\ designs a bandpass or bandstop filter with lower cutoff frequency \(w1\) and higher cutoff frequency \(w2\).

Data Types: \texttt{double}

**ftype — Filter type**  
\texttt{'low'} | \texttt{'bandpass'} | \texttt{'high'} | \texttt{'stop'}

Filter type, specified as:

- \texttt{'low'} — a lowpass filter with cutoff frequency \(Wo\). \texttt{'low'} is the default for scalar \(Wo\).
- \texttt{'high'} — a highpass filter with cutoff frequency \(Wo\).
- \texttt{'bandpass'} — a bandpass filter of order 2\(n\) if \(Wo\) is a two-element vector. \texttt{'bandpass'} is the default when \(Wo\) has two elements.
- \texttt{'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 \(2n + 1\) for bandpass and bandstop filters. The transfer function is expressed in terms of \(b\) and \(a\) as

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

Data Types: \texttt{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

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

Data Types: double

**A, B, C, D — State-space matrices**

matrices

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

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: 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:
Find lowpass analog prototype poles, zeros, and gain using the `besselap` function.

Convert the poles, zeros, and gain into state-space form.

If required, use a state-space transformation to convert the lowpass filter into a bandpass, highpass, or bandstop filter with the desired frequency constraints.

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

**References**


**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.

```matlab
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.
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

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

Match frequency, specified as a positive scalar.

Output Arguments

\( zd \) — Zeros

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

\( pd \) — Poles

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

\( kd \) — Gain

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

\( numd \) — Numerator coefficients

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**

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**

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**

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**

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

\[
H(z) = H(s)|_s = 2f_s \frac{z - 1}{z + 1}.
\]

This transformation maps the \( j\Omega \) axis (from \( \Omega = -\infty \) to \( +\infty \)) repeatedly around the unit circle (\( e^{j\omega} \), from \( \omega = -\pi \) to \( \pi \)) by

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

`bilinear` can accept an optional parameter \( F_p \) that specifies prewarping. \( F_p \), 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

\[
H(z) = H(s)|_s = \frac{2\pi F_p}{\tan\left(\frac{\pi F_p}{f_s}\right)} \frac{z - 1}{z + 1}.
\]

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 \( \pi \)) by

\[
\omega = 2\tan^{-1}\left(\frac{\Omega \tan\left(\frac{\pi F_p}{f_s}\right)}{2\pi F_p}\right).
\]
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 \(f_p\) is present, it prewarps:
   \[
   f_p = 2\pi f_p; \\
   f_s = f_p / \tan(f_p/f_s/2)
   \]
   otherwise, \( f_s = 2f_s \).

2. It strips any zeros at ±∞ using
   \[
   z = z(\text{finite}(z)); 
   \]

3. It transforms the zeros, poles, and gain using
   \[
   p_d = (1+p/f_s)/(1-p/f_s); \quad \% \text{Do bilinear transformation} \\
   z_d = (1+z/f_s)/(1-z/f_s); \\
   k_d = \text{real}(k*\text{prod}(f_s-z)/\text{prod}(f_s-p));
   \]

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:
\[
\begin{align*}
x[n+1] &= A_d x[n] + B_d u[n], \\
y[n] &= C_d x[n] + D_d u[n].
\end{align*}
\]

To convert an analog system in state-space form, `bilinear` performs two steps:

1. If \( f_p \) is present, let
   \[
   \lambda = \frac{\pi f_p}{\tan(\pi f_p / f_s)}.
   \]
   If \( f_p \) is not present, let \( \lambda = f_s \).

2. Compute \( A_d \), \( B_d \), \( C_d \), and \( D_d \) in terms of \( A \), \( B \), \( C \), and \( D \) using
   \[
   \begin{align*}
   A_d &= (I + A \frac{1}{2\lambda}), \\
   B_d &= \frac{1}{\sqrt{\lambda}} (I B), \\
   C_d &= \frac{1}{\sqrt{\lambda}} C(I), \\
   D_d &= \frac{1}{2\lambda} C(I B + D).
   \end{align*}
   \]

**Transfer Function**

For a system in transfer function form, `bilinear` converts an \( s \)-domain transfer function given by \( \text{num} \) and \( \text{den} \) to a discrete equivalent. Row vectors \( \text{num} \) and \( \text{den} \) 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:

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

\( f_s \) is the sample rate in hertz. `bilinear` returns the discrete equivalent in row vectors \( \text{numd} \) and \( \text{dend} \) in descending powers of \( z \) (ascending powers of \( z^{-1} \)). \( f_p \) is the optional match frequency, in hertz, for prewarping.
References


See Also

impinvar | lp2bp | lp2bs | lp2hp | lp2lp

Introduced before R2006a
**bitrevorder**

Permute data into bit-reversed order

**Syntax**

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

**Description**

*bitrevorder* is useful for pre-arranging filter coefficients so that bit-reversed ordering does not have to be performed as part of an *fft* or inverse FFT (*ifft*) computation. This can improve run-time efficiency for external applications or for Simulink® blockset models. Both MATLAB® *fft* and *ifft* functions process linear input and output.

**Note** Using *bitrevorder* is equivalent to using *digitrevorder* with radix base 2.

\[ y = \text{bitrevorder}(x) \] returns the input data in bit-reversed order in vector or matrix \( y \). The length of \( x \) must be an integer power of 2. If \( x \) is a matrix, the bit-reversal occurs on the first dimension of \( x \) with size greater than 1. \( y \) is the same size as \( x \).

\[ [y,i] = \text{bitrevorder}(x) \] returns the bit-reversed vector or matrix \( y \) and the bit-reversed indices \( i \), such that \( y = x(i) \). Recall that MATLAB matrices use 1-based indexing, so the first index of \( y \) will be 1, not 0.

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

<table>
<thead>
<tr>
<th>Linear Index</th>
<th>Bits</th>
<th>Bit-Reversed</th>
<th>Bit-Reversed Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>010</td>
<td>2</td>
</tr>
</tbody>
</table>
Linear Index | Bits | Bit-Reversed | Bit-Reversed Index
--- | --- | --- | ---
3 | 011 | 110 | 6
4 | 100 | 001 | 1
5 | 101 | 101 | 5
6 | 110 | 011 | 3
7 | 111 | 111 | 7

**Examples**

**Vector in Bit-Reversed Order**

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

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

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

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

<table>
<thead>
<tr>
<th>x</th>
<th>x_bin</th>
<th>v</th>
<th>v_bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>3</td>
<td>0011</td>
</tr>
</tbody>
</table>
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™.

See Also
digitrevorder | fft | ifft

Introduced before R2006a
blackman

Blackman window

Syntax

\[ w = \text{blackman}(L) \]
\[ w = \text{blackman}(L,sflag) \]

Description

\[ w = \text{blackman}(L) \] returns an \( L \)-point symmetric Blackman window.

\[ w = \text{blackman}(L,sflag) \] 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 \texttt{wvtool}.

\[ L = 64; \]
\[ \text{wvtool}(\text{blackman}(L)) \]
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 $N$:

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

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, $M \leq n \leq N - 1$, 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

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

\[ w = \text{blackmanharris}(N) \]
\[ w = \text{blackmanharris}(N, \text{sflag}) \]

Description

\[ w = \text{blackmanharris}(N) \] returns an \( N \)-point symmetric four-term Blackman-Harris window.

\[ w = \text{blackmanharris}(N, \text{sflag}) \] returns a Blackman-Harris window using the window sampling method specified by \( \text{sflag} \).

Examples

**Blackman-Harris Window**

Create a 32-point symmetric Blackman-Harris window. Display the result using \texttt{wvtool}.

\[ N = 32; \]
\[ \text{wvtool(\text{blackmanharris}(N))} \]
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

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

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

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

The periodic window is $N$-periodic.

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>0.35875</td>
</tr>
<tr>
<td>$a_1$</td>
<td>0.48829</td>
</tr>
<tr>
<td>$a_2$</td>
<td>0.14128</td>
</tr>
<tr>
<td>$a_3$</td>
<td>0.01168</td>
</tr>
</tbody>
</table>

**References**

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

\[ w = \text{bohmanwin}(L) \]

Description

\[ w = \text{bohmanwin}(L) \] returns an \( L \)-point Bohman window in \( w \).

Examples

Bohman Window

Compute a 64-point Bohman window. Display the result using \texttt{wvtool}.

\[
\begin{align*}
L &= 64; \\
bw &= \text{bohmanwin}(L); \\
wvtool(bw)
\end{align*}
\]
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

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

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

**References**


**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

\[
\begin{align*}
  y &= \text{buffer}(x,n) \\
  y &= \text{buffer}(x,n,p) \\
  y &= \text{buffer}(x,n,p,\text{opt}) \\
  [y,z] &= \text{buffer}(\ldots) \\
  [y,z,\text{opt}] &= \text{buffer}(\ldots)
\end{align*}
\]

Description

\( y = \text{buffer}(x,n) \) partitions a length-\( L \) signal vector \( x \) into nonoverlapping data segments (frames) of length \( n \). Each data frame occupies one column of matrix output \( y \), which has \( n \) rows and \( \text{ceil}(L/n) \) columns. If \( L \) is not evenly divisible by \( n \), the last column is zero-padded to length \( n \).

\( y = \text{buffer}(x,n,p) \) overlaps or underlaps successive frames in the output matrix by \( p \) samples:

- For \( 0 < p < n \) (overlap), \text{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.

\[
\begin{array}{cccccccccc}
  y &= 0 & 2 & 5 & 10 & 14 & 18 & 22 & 25 \\
  0 &= 0 & 3 & 7 & 12 & 15 & 19 & 21 & 23 \\
  1 &= 4 & 8 & 13 & 17 & 20 & 24 & 25 & 29 \\
  2 &= 6 & 10 & 14 & 18 & 22 & 23 & 28 & 30 \\
  3 &= 7 & 11 & 15 & 19 & 23 & 24 & 28 & 30 \\
  4 &= 8 & 12 & 16 & 20 & 23 & 24 & 28 & 30 \\
\end{array}
\]

The first frame starts with \( p \) zeros (the default initial condition), and the number of columns in \( y \) is \( \text{ceil}(L/(n-p)) \).
• For \( p < 0 \) (underlap), *buffer* skips \( p \) samples between consecutive frames. For example, if \( x = 1:30 \) and \( n = 7 \), a buffer with underlap of \( p = -3 \) looks like this.

\[
\begin{array}{cccc}
1 & 11 & 21 & \text{skipped}\{8\ 18\ 28\} \\
2 & 12 & 22 & \\
3 & 13 & 23 & \\
4 & 14 & 24 & \\
5 & 15 & 25 & \\
6 & 16 & 26 & \\
7 & 17 & 27 & \\
\end{array}
\]

The number of columns in \( y \) is \( \text{ceil}(L/(n-p)) \).

\[
y = \text{buffer}(x,n,p,\text{opt})
\]
specifies a vector of samples to precede \( x(1) \) in an overlapping buffer, or the number of initial samples to skip in an underlapping buffer:

• For \( 0 < p < n \) (overlap), \( \text{opt} \) 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, \( \text{opt} \) should contain the final \( p \) samples of the previous buffer in the sequence. See “Continuous Buffering” on page 1-139 below.

By default, \( \text{opt} \) is \( \text{zeros}(p,1) \) for an overlapping buffer. Set \( \text{opt} \) to 'nodelay' to skip the initial condition and begin filling the buffer immediately with \( x(1) \). In this case, \( L \) must be \( \text{length}(p) \) or longer. For example, if \( x = 1:30 \) and \( n = 7 \), a buffer with overlap of \( p = 3 \) looks like this.

\[
\begin{array}{cccccccccccccccccccccccc}
1 & 5 & 9 & 13 & 17 & 21 & 25 & 29 & 30 \\
2 & 6 & 10 & 14 & 18 & 22 & 26 & 28 & 0 \\
3 & 7 & 11 & 15 & 19 & 23 & 27 & 0 & 0 \\
4 & 8 & 12 & 16 & 20 & 24 & 28 & 0 & 0 \\
5 & 9 & 13 & 17 & 21 & 25 & 29 & 0 & 0 \\
6 & 10 & 14 & 18 & 22 & 26 & 28 & 0 & 0 \\
7 & 11 & 15 & 19 & 23 & 27 & 0 & 0 & 0 \\
\end{array}
\]

• For \( p < 0 \) (underlap), \( \text{opt} \) is an integer value in the range \([0,-p]\) specifying the number of initial input samples, \( x(1:\text{opt}) \), to skip before adding samples to the buffer. The first value in the buffer is therefore \( x(\text{opt}+1) \). By default, \( \text{opt} \) 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 1-139 for an example of how this works in practice.

\[ y, z = \text{buffer}(\ldots) \] partitions the length-L signal vector \( x \) into frames of length \( n \), and outputs only the full frames in \( y \). If \( y \) is an overlapping buffer, it has \( n \) rows and \( m \) columns, where \( m = \text{floor}(L/(n-p)) \) when \( \text{length}(\text{opt}) = p \) or \( m = \text{floor}((L-n)/(n-p))+1 \) when \( \text{opt} = \text{'nodelay'} \).

If \( y \) is an underlapping buffer, it has \( n \) rows and \( m \) columns, where \( m = \text{floor}((L-\text{opt})/(n-p)) + (\text{rem((L-\text{opt}),(n-p))} >= n) \).

If the number of samples in the input vector (after the appropriate overlapping or underlapping operations) exceeds the number of places available in the \( n \)-by-\( m \) buffer, the remaining samples in \( x \) are output in vector \( z \), which for an overlapping buffer has length \( L - m*(n-p) \) when \( \text{length}(\text{opt}) = p \) or \( L - ((m-1)*(n-p)+n) \) when \( \text{opt} = \text{'nodelay'} \), and for an underlapping buffer has length \( (L-\text{opt}) - m*(n-p) \).

Output \( z \) shares the same orientation (row or column) as \( x \). If there are no remaining samples in the input after the buffer with the specified overlap or underlap is filled, \( z \) is an empty vector.

\[ y, z, \text{opt} = \text{buffer}(\ldots) \] returns the last \( p \) samples of an overlapping buffer in output \( \text{opt} \). In an underlapping buffer, \( \text{opt} \) is the difference between the total number of points to skip between frames \(-p\) and the number of points in \( x \) that were available to be skipped after filling the last frame:

- For \( 0 < p < n \) (overlap), \( \text{opt} \) (as an output) contains the final \( p \) samples in the last frame of the buffer. This vector can be used as the initial condition for a subsequent buffering operation in a sequence of consecutive buffering operations. This allows the desired frame overlap to be maintained from one buffer to the next. See “Continuous Buffering” on page 1-139 below.
- For \( p = 0 \) (underlap), \( \text{opt} \) (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: \( \text{opt} = m*(n-p) + \text{opt} - L \) where \( \text{opt} \) on the right is the input argument to buffer, and \( \text{opt} \) on the left is the output argument. \( z \) is the empty vector. Here \( m \) is the number of columns in the buffer, which is \( m = \text{floor}((L-\text{opt})/(n-p)) + (\text{rem((L-\text{opt}),(n-p))}>=n) \).
Note that for an underlapping buffer output \( \text{opt} \) is always zero when output \( z \) contains data.

The \( \text{opt} \) output for an underlapping buffer is especially useful when the current buffering operation is one in a sequence of consecutive buffering operations. The \( \text{opt} \) 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 \( \text{opt} \) points need to be removed from the first frame of the next buffer.

In a sequence of buffering operations, the \( \text{opt} \) output from each operation should be used as the \( \text{opt} \) 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 1-139 below for an example of how this works in practice.

**Examples**

**Continuous Overlapping Buffers**

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

\[
data = \text{buffer}(1:1100,11);
\]

Take the frames (columns) in the matrix \( \text{data} \) to be the sequential outputs of a data acquisition board sampling a physical signal: \( \text{data}(:,1) \) is the first D/A output, containing the first 11 signal samples; \( \text{data}(:,2) \) 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 \text{buffer} \) to operate on each successive input frame, using the \( \text{opt} \) 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.

```matlab
for i = 1:size(data,2)
    x = data(:,i);
    [y,z,opt] = buffer([z;x],n,p,opt);
end
```

Here's what happens during the first four iterations.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Input frame [z;x]'</th>
<th>opt (input)</th>
<th>opt (output)</th>
<th>Output buffer (y)</th>
<th>Overflow (z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=1</td>
<td>[1:11]</td>
<td>-5</td>
<td>9</td>
<td>[-5 3 6]</td>
<td>[10 11]</td>
</tr>
<tr>
<td>i=2</td>
<td>[10 11 12:22]</td>
<td>9</td>
<td>21</td>
<td>[12 15 18]</td>
<td>[22]</td>
</tr>
<tr>
<td>i=4</td>
<td>[34:44]</td>
<td>33</td>
<td>42</td>
<td>[33 36 39]</td>
<td>[43 44]</td>
</tr>
</tbody>
</table>

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.
**Continuous Underlapping Buffers**

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

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

Take `data(:,1)` as the first D/A 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.

```matlab
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`.

```matlab
for i = 1:size(data,2)
    x = data(:,i);
    [y,z,opt] = buffer([z';x],n,p,opt);
end
```

Here's what happens during the first three iterations.
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.

### Diagnostics

Error messages are displayed when $p \geq n$ or $\text{length(opt)} \neq \text{length(p)}$ in an overlapping buffer case:
Frame overlap $P$ must be less than the buffer size $N$. Initial conditions must be specified as a length-$P$ vector.

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.

As an example, you might acquire data from an A/D card in frames of 64 samples. In the simplest case, you could rebuffer the data into frames of 16 samples; `buffer` with $n = 16$ creates a buffer of four frames from each 64-element input frame. The result is that the signal of frame size 64 has been converted to a signal of frame size 16; no samples were added or removed.

In the general case where the original signal frame size, $L$, is not equally divisible by the new frame size, $n$, the overflow from the last frame needs to be captured and recycled into the following buffer. You can do this by iteratively calling `buffer` on input $x$ with the two-output-argument syntax:

```matlab
[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.
See Also
reshape

Introduced before R2006a
**Buttap**

Butterworth filter prototype

**Syntax**

\[ [z, p, k] = \text{buttap}(n) \]

**Description**

\[ [z, p, k] = \text{buttap}(n) \] returns the poles and gain of an order \( n \) Butterworth analog lowpass filter prototype. The function returns the poles in the length \( n \) column vector \( p \) and the gain in scalar \( k \). \( z \) is an empty matrix because there are no zeros. The transfer function is

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

Butterworth filters are characterized by a magnitude response that is maximally flat in the passband and monotonic overall. In the lowpass case, the first \( 2n-1 \) derivatives of the squared magnitude response are zero at \( \omega = 0 \). The squared magnitude response function is

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

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

**Note** The function *buttap* returns zeros, poles, and gain \((z, p, \text{ and } k)\) 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.
Examples

Frequency Response of a Butterworth Analog Filter

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

```matlab
[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
```

![Graph showing magnitude and phase responses of a Butterworth analog filter](image-url)

- **Magnitude Response**: The response decreases as the frequency increases, typical of a lowpass filter.
- **Phase Response**: The phase response shows a linear phase characteristic, which is one of the key properties of a Butterworth filter.
Algorithms

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

References


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**

\[
[b,a] = \text{butter}(n,Wn)
\]

\[
[b,a] = \text{butter}(n,Wn,ftype)
\]

\[
[z,p,k] = \text{butter}(___)
\]

\[
[A,B,C,D] = \text{butter}(___)
\]

\[
[____] = \text{butter}(___,'s')
\]

**Description**

\[
[b,a] = \text{butter}(n,Wn)
\]

returns the transfer function coefficients of an \(n\)th-order lowpass digital Butterworth filter with normalized cutoff frequency \(Wn\).

\[
[b,a] = \text{butter}(n,Wn,ftype)
\]

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 1-154 for information about numerical issues that affect forming the transfer function.

\[
[z,p,k] = \text{butter}(___)
\]

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.

\[
[A,B,C,D] = \text{butter}(___)
\]

designs a lowpass, highpass, bandpass, or bandstop digital Butterworth filter and returns the matrices that specify its state-space representation.

\[
[____] = \text{butter}(___,'s')
\]

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.

```matlab
fc = 300;
fs = 1000;

[b,a] = butter(6,fc/(fs/2));
freqz(b,a)
```
dataIn = randn(1000,1);
dataOut = filter(b,a,dataIn);

**Bandstop Butterworth Filter**

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

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

**Highpass Butterworth Filter**

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

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

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

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

```matlab
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 $2\pi$ to convert the frequency to radians per second. Compute the frequency response of the filter at 4096 points.

```matlab
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.

```matlab
[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.

```matlab
[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.

```matlab
[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.

```matlab
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)')
```

1-150
The Butterworth and Chebyshev Type II filters have flat passbands and wide transition bands. The Chebyshev Type I and elliptic filters roll off faster but have passband ripple. The frequency input to the Chebyshev Type II design function sets the beginning of the stopband rather than the end of the passband.
Input Arguments

\( n \) — Filter order
integer scalar

Filter order, specified as an integer scalar.
Data Types: double

\( Wn \) — Cutoff frequency
scalar | two-element vector

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

- If \( Wn \) is scalar, then \texttt{butter} designs a lowpass or highpass filter with cutoff frequency \( Wn \).

  If \( Wn \) is the two-element vector \([w1 \ w2]\), where \( w1 < w2 \), then \texttt{butter} designs a bandpass or bandstop filter with lower cutoff frequency \( w1 \) and higher cutoff frequency \( w2 \).

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

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

Data Types: double

\( \texttt{ftype} \) — 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**

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

- For digital filters, the transfer function is expressed in terms of \( b \) and \( a \) as
  \[
  H(z) = \frac{B(z)}{A(z)} = \frac{b(1)+b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{a(1)+a(2)z^{-1} + \cdots + a(n+1)z^{-n}}.
  \]
- For analog filters, the transfer function is expressed in terms of \( b \) and \( a \) as
  \[
  H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \cdots + b(n+1)}{a(1)s^n + a(2)s^{n-1} + \cdots + a(n+1)}.
  \]

Data Types: double

**z, p, k — Zeros, poles, and gain**

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

- For digital filters, the transfer function is expressed in terms of \( z \), \( p \), and \( k \) as
  \[
  H(z) = k \frac{(1 - z(1)z^{-1})(1 - z(2)z^{-1})\cdots(1 - z(n)z^{-1})}{(1 - p(1)z^{-1})(1 - p(2)z^{-1})\cdots(1 - p(n)z^{-1})}.
  \]
- For analog filters, the transfer function is expressed in terms of \( z \), \( p \), and \( k \) as
  \[
  H(s) = k \frac{(s - z(1))(s - z(2))\cdots(s - z(n))}{(s - p(1))(s - p(2))\cdots(s - p(n))}.
  \]

Data Types: double

**A, B, C, D — State-space matrices**

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

- For digital filters, the state-space matrices relate the state vector \( x \), the input \( u \), and the output \( y \) through
  \[
  x(k + 1) = A x(k) + B u(k) \\
  y(k) = C x(k) + D u(k).
  \]

- For analog filters, the state-space matrices relate the state vector \( x \), the input \( u \), and the output \( y \) through
  \[
  \dot{x} = A x + B u \\
  y = C x + D u.
  \]

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.

```matlab
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.

The `butter` function 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 $W_n$ or at $w_1$ and $w_2$.
5 It converts the state-space filter back to its transfer function or zero-pole-gain form, as required.

ExtendedCapabilities

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

Usage notes and limitations:
Filter coefficients must be constants. Expressions or variables are allowed if their values do not change.

See Also
besself | buttap | buttord | cheby1 | cheby2 | designfilt | ellip | filter | maxflat | sosfilt

Introduced before R2006a
**buttord**

Butterworth filter order and cutoff frequency

**Syntax**

\[
[n,Wn] = \text{buttord}(Wp,Ws,Rp,Rs)
\]

\[
[n,Wn] = \text{buttord}(Wp,Ws,Rp,Rs,'s')
\]

**Description**

\([n,Wn] = \text{buttord}(Wp,Ws,Rp,Rs)\) 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 \(\pi\) 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 \text{butter}.

\([n,Wn] = \text{buttord}(Wp,Ws,Rp,Rs,'s')\) 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.

\[
Wp = 40/500;
\]

\[
Ws = 150/500;
\]

\([n,Wn] = \text{buttord}(Wp,Ws,3,60)\)
n = 5
Wn = 0.0810

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

\[ [z,p,k] = \text{butter}(n,Wn); \]
\[ \text{sos} = \text{zp2sos}(z,p,k); \]
\[ \text{freqz}(\text{sos},512,1000) \]
\[ \text{title}(\text{sprintf}(\text{'}n = %d \text{ Butterworth Lowpass Filter}',n)) \]
**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.

\[
\begin{align*}
W_p &= \left[ \frac{100}{500}, \frac{200}{500} \right]; \\
W_s &= \left[ \frac{50}{500}, \frac{250}{500} \right]; \\
R_p &= 3; \\
R_s &= 40; \\
[n, W_n] &= \text{buttord}(W_p, W_s, R_p, R_s)
\end{align*}
\]

\[
\begin{align*}
n &= 8 \\
W_n &= 1 \times 2 \\
&\quad 0.1951 \quad 0.4080
\end{align*}
\]

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

\[
\begin{align*}
[z, p, k] &= \text{butter}(n, W_n); \\
sos &= \text{zp2sos}(z, p, k); \\
\text{freqz}(sos, 128, 1000) \\
\text{title(sprintf('n = %d Butterworth Bandpass Filter', n))}
\end{align*}
\]
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, \( \pi \) rad/sample.
• If \( W_p \) and \( W_s \) are both scalars and \( W_p < W_s \), then \texttt{buttord} returns the order and cutoff frequency of a lowpass filter. The stopband of the filter ranges from \( W_s \) to 1 and the passband ranges from 0 to \( W_p \).

• If \( W_p \) and \( W_s \) are both scalars and \( W_p > W_s \), then \texttt{buttord} returns the order and cutoff frequency of a highpass filter. The stopband of the filter ranges from 0 to \( W_s \) and the passband ranges from \( W_p \) to 1.

• If \( W_p \) and \( W_s \) are both vectors and the interval specified by \( W_s \) contains the one specified by \( W_p \) (\( W_s(1) < W_p(1) < W_p(2) < W_s(2) \)), then \texttt{buttord} returns the order and cutoff frequencies of a bandpass filter. The stopband of the filter ranges from 0 to \( W_s(1) \) and from \( W_s(2) \) to 1. The passband ranges from \( W_p(1) \) to \( W_p(2) \).

• If \( W_p \) and \( W_s \) are both vectors and the interval specified by \( W_p \) contains the one specified by \( W_s \) (\( W_p(1) < W_s(1) < W_s(2) < W_p(2) \)), then \texttt{buttord} returns the order and cutoff frequencies of a bandstop filter. The stopband of the filter ranges from \( W_s(1) \) to \( W_s(2) \). The passband ranges from 0 to \( W_p(1) \) and from \( W_p(2) \) to 1.

**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, \( \pi \) 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.

\( \text{Wn} \) — Cutoff frequencies
scalar | vector

Cutoff frequencies, returned as a scalar or vector.

Algorithms

\texttt{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.

\texttt{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

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**

\[
\begin{align*}
xhat &= \text{cceps}(x) \\
[xhat, nd] &= \text{cceps}(x) \\
[xhat, nd, xhat1] &= \text{cceps}(x) \\
[\_\_\_] &= \text{cceps}(x, n)
\end{align*}
\]

**Description**

\( xhat = \text{cceps}(x) \) returns the complex cepstrum \( xhat \) of the real data sequence \( x \) using the Fourier transform.

**Note** cceps only works on real data.

\( [xhat, nd] = \text{cceps}(x) \) returns the number of samples \( nd \) of (circular) delay added to \( x \) prior to finding the complex cepstrum.

\( [xhat, nd, xhat1] = \text{cceps}(x) \) returns a second complex cepstrum, \( xhat1 \).

\( [\_\_] = \text{cceps}(x, n) \) zero pads \( x \) to length \( n \) and returns the length \( n \).

**Examples**

**Using cceps to show an echo**

This example uses \texttt{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 \( \pi \) radians.

\( n \) — Length of zero-padded signal
positive real integer

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

Output Arguments

\( \text{xhat} \) — 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.

\( \text{xhat1} \) — Second complex cepstrum
vector

Second complex cepstrum, returned as a vector. \( \text{xhat1} \) 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]. \texttt{ccps} 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 \texttt{cceps}:

\begin{verbatim}
h = fft(x);
logh = log(abs(h)) + sqrt(-1)*rcunwrap(angle(h));
y = real(ifft(logh));
\end{verbatim}

\textbf{Note} \texttt{rcunwrap} in the above code segment is a special version of \texttt{unwrap} that subtracts a straight line from the phase. \texttt{rcunwrap} is a local function within \texttt{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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fourier</td>
<td>Can be used for any signal.</td>
<td>Requires phase unwrapping.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Output is aliased.</td>
</tr>
<tr>
<td>Factorization</td>
<td>Does not require phase unwrapping. No aliasing</td>
<td>Can be used only for short duration signals. Input signal must have an all-zero Z-transform with no zeros on the unit circle.</td>
</tr>
</tbody>
</table>

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).

\textbf{References}


See Also
hilbert | icceps | 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.

\[
a = [2 \ 1 \ 2 \ 1]; \\
b = [1 \ 2 \ 3 \ 4]; \\
c = \text{cconv}(a,b,4)
\]

\[
c = 1\times4 \\
14 \ 16 \ 14 \ 16
\]

Circular Cross-Correlation

Generate two complex sequences. Use \text{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.

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

Compare the result to the cross-correlation computed using \text{xcorr}.

\[
cref = \text{xcorr}(a,b); \\
dif = \text{norm}(c-cref)
\]

\[
dif = 3.3565e-15
\]

Circular Convolution with Varying Output Length

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

\[
x1 = \text{conv}([1 \ 1 \ 1],[1 \ 1 \ 1])
\]
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
Compute the modulo-3 circular convolution and compare it to the aliased linear convolution.

\[ cc_{n3} = cconv(x_1, x_2, 3) \]

\[ cc_{n3} = 1 \times 3 \]

\[ \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \]

\[ mod_{3} = \text{sum} \left( \text{reshape} \left( lcnv, 3, \frac{nl}{3} \right)' \right) \]

\[ mod_{3} = 1 \times 3 \]

\[ \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \]

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

\[ c = 5; \]
\[ z = \text{zeros}(c \times \text{ceil}(\frac{nl}{c}), 1); \]
\[ z(1:nl) = lcnv; \]

\[ cc_{nc} = cconv(x_1, x_2, c) \]

\[ cc_{nc} = 1 \times 5 \]

\[ \begin{bmatrix} 0.0000 & -1.0000 & -1.0000 & 1.0000 & 1.0000 \end{bmatrix} \]

\[ modc = \text{sum} \left( \text{reshape} \left( z, c, \text{numel}(z)/c \right)' \right) \]

\[ modc = 1 \times 5 \]

\[ \begin{bmatrix} 0 & -1 & -1 & 1 & 1 \end{bmatrix} \]

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
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 =
1.8496e-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


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

cconv | xcorr

Introduced in R2007a
cell2sos

Convert second-order sections cell array to matrix

Syntax

\[ m = \text{cell2sos}(c) \]

Description

\[ m = \text{cell2sos}(c) \] 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 \( \text{tf2sos} \). You can use \( m = \text{cell2sos}(c) \) to invert the results of \( c = \text{sos2cell}(m) \).

c must be a cell array of the form

\[ c = \{ \{ b_1 \ a_1 \} \{ b_2 \ a_2 \} \ldots \{ b_L \ a_L \} \} \]

where both \( b_i \) and \( a_i \) are row vectors of at most length 3, and \( i = 1, 2, \ldots, L \). The resulting matrix \( m \) is given by

\[ m = [b_1 \ a_1; b_2 \ a_2; \ldots; b_L \ a_L] \]

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.

\[ \text{cll} = \{\{[3 \ 6 \ 7] \ [1 \ 1 \ 2]\} \]
\{[1 \ 4 \ 5] \ [1 \ 9 \ 3]\} \]
\{[2 \ 7 \ 1] \ [1 \ 7 \ 8]\}\};
\]
sos = \text{cell2sos}(\text{cll})
sos = 3x6

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(...,'skip_stage2')
b = cfirpm(...,'debug')
b = cfirpm(...,{lgrid})
[b,delta] = cfirpm(...)
[b,delta,opt] = cfirpm(...)
```

**Description**

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

`b = cfirpm(n,f,@fresp)` returns a length `n+1` FIR filter with the best approximation to the desired frequency response as returned by function `fresp`, which is called by its function handle (`@fresp`). `f` is a vector of frequency band edge pairs, specified in the range `-1` and `1`, where `1` corresponds to the normalized Nyquist frequency. The frequencies must be in increasing order, and `f` must have even length. The frequency bands span `f(k)` to `f(k+1)` for `k` odd; the intervals `f(k+1)` to `f(k+2)` for `k` odd are “transition bands” or “don't care” regions during optimization.

Predefined `fresp` frequency response functions are included for a number of common filter designs, as described below. (See “Create Function Handle” (MATLAB) for more information on how to create a custom `fresp` function.) For all of the predefined frequency response functions, the symmetry option `'sym'` defaults to `'even'` if no negative frequencies are contained in `f` and `d = 0`; otherwise `'sym'` defaults to `'none'`. 
(See the 'sym' option below for details.) For all of the predefined frequency response functions, \( d \) specifies a group-delay offset such that the filter response has a group delay of \( \frac{n}{2}+d \) in units of the sample interval. Negative values create less delay; positive values create more delay. By default \( d = 0 \):

- \texttt{@lowpass}, \texttt{@highpass}, \texttt{@allpass}, \texttt{@bandpass}, \texttt{@bandstop}

These functions share a common syntax, exemplified below by \texttt{@lowpass}.

\[
b = \text{cfirpm}(n,f,\texttt{@lowpass},...)\text{ and }\]
\[
b = \text{cfirpm}(n,f,\{\texttt{@lowpass},d\},...) \text{ design a linear-phase } (n/2+d \text{ delay}) \text{ filter.}
\]

**Note** For \texttt{@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.

- \texttt{@multiband} designs a linear-phase frequency response filter with arbitrary band amplitudes.

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

- \texttt{@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.

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

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

\[
b = \text{cfirpm}(n,f,\texttt{@hilbfilt},...) \text{ and }\]
\[
b = \text{cfirpm}(N,F,\{\texttt{@hilbfilt},d\},...) \text{ design a linear-phase } (n/2+d \text{ delay}) \text{ Hilbert transform filter.}
• @invsinc designs a linear-phase inverse-sinc filter response.

\[ b = \text{cfirpm}(n,f,\{@\text{invsinc},a\},...) \text{ and } \]

\[ b = \text{cfirpm}(n,f,\{@\text{invsinc},a,d\},...) \]

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

\[ b = \text{cfirpm}(n,f,\text{fresp},w) \]

uses the real, nonnegative weights in vector \( w \) to weight the fit in each frequency band. The length of \( w \) is half the length of \( f \), so there is exactly one weight per band.

\[ b = \text{cfirpm}(n,f,a) \]

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

\[ b = \text{cfirpm}(n,f,a,w) \]

applies an optional set of positive weights, one per band, for use during optimization. If \( w \) is not specified, the weights are set to unity.

\[ b = \text{cfirpm}(\ldots,'\text{sym}') \]

imposes a symmetry constraint on the impulse response of the design, where 'sym' may be one of the following:

• 'none' indicates no symmetry constraint. This is the default if any negative band edge frequencies are passed, or if \( \text{fresp} \) does not supply a default.
• 'even' indicates a real and even impulse response. This is the default for highpass, lowpass, allpass, bandpass, bandstop, inverse-sinc, and multiband designs.
• 'odd' indicates a real and odd impulse response. This is the default for Hilbert and differentiator designs.
• 'real' indicates conjugate symmetry for the frequency response

If any 'sym' option other than 'none' is specified, the band edges should be specified only over positive frequencies; the negative frequency region is filled in from symmetry. If a 'sym' option is not specified, the \( \text{fresp} \) function is queried for a default setting. Any user-supplied \( \text{fresp} \) function should return a valid 'sym' option when it is passed 'defaults' as the filter order \( N \).

\[ b = \text{cfirpm}(\ldots,'\text{skip}\_\text{stage2}') \]

disables the second-stage optimization algorithm, which executes only when \( \text{cfirpm} \) determines that an optimal solution has not been reached by the standard \( \text{firpm} \) error-exchange. Disabling this algorithm may increase
the speed of computation, but may incur a reduction in accuracy. By default, the second-stage optimization is enabled.

\[ b = \text{cfirpm}(\ldots, 'debug') \] enables the display of intermediate results during the filter design, where 'debug' may be one of 'trace', 'plots', 'both', or 'off'. By default it is set to 'off'.

\[ b = \text{cfirpm}(\ldots, \{lgrid\}) \] uses the integer \( lgrid \) to control the density of the frequency grid, which has roughly \( 2^{\text{nextpow2}}(lgrid*n) \) frequency points. The default value for \( lgrid \) is 25. Note that the \{lgrid\} argument must be a 1-by-1 cell array.

Any combination of the 'sym', 'skip_stage2', 'debug', and \{lgrid\} options may be specified.

\[ [b,delta] = \text{cfirpm}(\ldots) \] returns the maximum ripple height delta.

\[ [b,delta,opt] = \text{cfirpm}(\ldots) \] returns a structure \( \text{opt} \) of optional results computed by \text{cfirpm} and contains the following fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opt.fgrid</td>
<td>Frequency grid vector used for the filter design optimization</td>
</tr>
<tr>
<td>opt.des</td>
<td>Desired frequency response for each point in opt.fgrid</td>
</tr>
<tr>
<td>opt.wt</td>
<td>Weighting for each point in opt.fgrid</td>
</tr>
<tr>
<td>opt.H</td>
<td>Actual frequency response for each point in opt.fgrid</td>
</tr>
<tr>
<td>opt.error</td>
<td>Error at each point in opt.fgrid</td>
</tr>
<tr>
<td>opt.iextr</td>
<td>Vector of indices into opt.fgrid for extremal frequencies</td>
</tr>
<tr>
<td>opt.fextr</td>
<td>Vector of extremal frequencies</td>
</tr>
</tbody>
</table>

User-definable functions may be used, instead of the predefined frequency response functions for @\text{fresp}. The function is called from within \text{cfirpm} using the following syntax

\[ [dh,dw] = \text{fresp}(n,f,gf,w,p1,p2,\ldots) \]

where:

- \( n \) is the filter order.
- \( f \) is the vector of frequency band edges that appear monotonically between -1 and 1, where 1 corresponds to the Nyquist frequency.
• \( \text{gf} \) is a vector of grid points that have been linearly interpolated over each specified frequency band by \text{cfirpm}. \( \text{gf} \) determines the frequency grid at which the response function must be evaluated. This is the same data returned by \text{cfirpm} in the \text{fgrid} field of the \text{opt} structure.

• \( \text{w} \) is a vector of real, positive weights, one per band, used during optimization. \( \text{w} \) is optional in the call to \text{cfirpm}; if not specified, it is set to unity weighting before being passed to \text{fresp}.

• \( \text{dh} \) and \( \text{dw} \) are the desired complex frequency response and band weight vectors, respectively, evaluated at each frequency in grid \( \text{gf} \).

• \( \text{p1}, \text{p2}, \ldots \), are optional parameters that may be passed to \text{fresp}.

Additionally, a preliminary call is made to \text{fresp} to determine the default symmetry property \( '\text{sym}' \). This call is made using the syntax:

\[
\text{sym} = \text{fresp}(\text{'defaults'},\{\text{n},\text{f},[],\text{w},\text{p1},\text{p2},\ldots\})
\]

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

**Examples**

**Equiripple Lowpass Filter**

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

\[
\text{b} = \text{cfirpm}(30,[-1 -0.5 -0.4 0.7 0.8 1],\text{@lowpass});
\]

\[
\text{fvtool(b,1,'OverlayedAnalysis','phase')}
\]
**FIR Approximation to Allpass Response**

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

```matlab
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.

```matlab
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')
```
Algorithms

An extended version of the Remez exchange method is implemented for the complex case. This exchange method obtains the optimal filter when the equiripple nature of the filter is restricted to have $n+2$ extremals. When it does not converge, the algorithm switches to an ascent-descent algorithm that takes over to finish the convergence to the optimal solution. See the references for further details.

References


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

\[ [z,p,k] = \text{cheb1ap}(n,Rp) \]

Description

\[ [z,p,k] = \text{cheb1ap}(n,Rp) \] returns the poles and gain of an order \( n \) Chebyshev Type I analog lowpass filter prototype with \( Rp \) dB of ripple in the passband. The function returns the poles in the length \( n \) column vector \( p \) and the gain in scalar \( k \). \( z \) is an empty matrix, because there are no zeros. The transfer function is

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

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

Examples

Frequency Response of an Analog Chebyshev Type I Filter

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

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

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 | cheb2ap | cheby1 | 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
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.

\[
W_p = \frac{[60\ 200]}{500};
\]
\[
W_s = \frac{[50\ 250]}{500};
\]
Rp = 3;
Rs = 40;
[n,Wp] = cheb1ord(Wp,Ws,Rp,Rs)

n = 7
Wp = 1x2
    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, \( \pi \) 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 \( W_p \) and \( W_s \) determine the type of filter `cheb1ord` returns:

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

Use the following guide to specify filters of different types.
Filter Type Stopband and Passband Specifications

<table>
<thead>
<tr>
<th>Filter Type</th>
<th>Stopband and Passband Conditions</th>
<th>Stopband</th>
<th>Passband</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowpass</td>
<td>$W_p &lt; W_s$, both scalars</td>
<td>($W_s,1$)</td>
<td>($0,W_p$)</td>
</tr>
<tr>
<td>Highpass</td>
<td>$W_p &gt; W_s$, both scalars</td>
<td>($0,W_s$)</td>
<td>($W_p,1$)</td>
</tr>
<tr>
<td>Bandpass</td>
<td>The interval specified by $W_s$ contains the one specified by $W_p$ ($W_s(1) &lt; W_p(1) &lt; W_p(2) &lt; W_s(2)$).</td>
<td>($0,W_s(1)$) and ($W_s(2),1$)</td>
<td>($W_p(1),W_p(2)$)</td>
</tr>
<tr>
<td>Bandstop</td>
<td>The interval specified by $W_p$ contains the one specified by $W_s$ ($W_p(1) &lt; W_s(1) &lt; W_s(2) &lt; W_p(2)$).</td>
<td>($0,W_p(1)$) and ($W_p(2),1$)</td>
<td>($W_s(1),W_s(2)$)</td>
</tr>
</tbody>
</table>

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 $W_p$ and $W_s$ 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 cheby2 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


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` | `cheb2ord` | `cheby2` | `ellipord` | `kaiserord`

Introduced before R2006a
cheb2ap

Chebyshev Type II analog lowpass filter prototype

Syntax

\[ [z,p,k] = \text{cheb2ap}(n,Rs) \]

Description

\[ [z,p,k] = \text{cheb2ap}(n,Rs) \]
finds the zeros, poles, and gain of an order \( n \) Chebyshev Type II analog lowpass filter prototype with stopband ripple \( Rs \) dB down from the passband peak value. \text{cheb2ap} returns the zeros and poles in length \( n \) column vectors \( z \) and \( p \) and the gain in scalar \( k \). If \( n \) is odd, \( z \) is length \( n-1 \). The transfer function is

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

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

Examples

Frequency Response of an Analog Chebyshev Type II Filter

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

\[
[z,p,k] = \text{cheb2ap}(6,70); \quad \% \text{Lowpass filter prototype}
\]

\[
[num,den] = \text{zp2tf}(z,p,k); \quad \% \text{Convert to transfer function form}
\]

\[
freqs(num,den) \quad \% \text{Frequency response of analog filter}
\]
Algorithms

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

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


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 | cheblap | 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.

\[
Wp = 40/500;
Ws = 150/500;
Rp = 3;
Rs = 60;
\]

\([n,Ws] = \text{cheb2ord}(Wp,Ws,Rp,Rs)\)

\( n = 4 \)
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 = 1x2
    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, \( \pi \) 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 \( W_p \) and \( W_s \) are both scalars and \( W_p < W_s \), then \texttt{cheb2ord} returns the order and cutoff frequency of a lowpass filter. The stopband of the filter ranges from \( W_s \) to 1, and the passband ranges from 0 to \( W_p \).

• If \( W_p \) and \( W_s \) are both scalars and \( W_p > W_s \), then \texttt{cheb2ord} returns the order and cutoff frequency of a highpass filter. The stopband of the filter ranges from 0 to \( W_s \), and the passband ranges from \( W_p \) to 1.

• If \( W_p \) and \( W_s \) are both vectors and the interval specified by \( W_s \) contains the interval specified by \( W_p \) (\( W_s(1) < W_p(1) < W_p(2) < W_s(2) \)), then \texttt{cheb2ord} returns the order and cutoff frequencies of a bandpass filter. The stopband of the filter ranges from 0 to \( W_s(1) \) and from \( W_s(2) \) to 1. The passband ranges from \( W_p(1) \) to \( W_p(2) \).

• If \( W_p \) and \( W_s \) are both vectors and the interval specified by \( W_p \) contains the interval specified by \( W_s \) (\( W_p(1) < W_s(1) < W_s(2) < W_p(2) \)), then \texttt{cheb2ord} returns the order and cutoff frequencies of a bandstop filter. The stopband of the filter ranges from \( W_s(1) \) to \( W_s(2) \). The passband ranges from 0 to \( W_p(1) \) and from \( W_p(2) \) to 1.

Use the following guide to specify filters of different types.

### Filter Type Stopband and Passband Specifications

<table>
<thead>
<tr>
<th>Filter Type</th>
<th>Stopband and Passband Conditions</th>
<th>Stopband</th>
<th>Passband</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowpass</td>
<td>( W_p &lt; W_s ), both scalars</td>
<td>( (W_s,1) )</td>
<td>( (0,W_p) )</td>
</tr>
<tr>
<td>Highpass</td>
<td>( W_p &gt; W_s ), both scalars</td>
<td>( (0,W_s) )</td>
<td>( (W_p,1) )</td>
</tr>
<tr>
<td>Bandpass</td>
<td>The interval specified by ( W_s ) contains the one specified by ( W_p ) (( W_s(1) &lt; W_p(1) &lt; W_p(2) &lt; W_s(2) )).</td>
<td>( (0,W_s(1)) ) and ( (W_s(2),1) )</td>
<td>( (W_p(1),W_p(2)) )</td>
</tr>
<tr>
<td>Bandstop</td>
<td>The interval specified by ( W_p ) contains the one specified by ( W_s ) (( W_p(1) &lt; W_s(1) &lt; W_s(2) &lt; W_p(2) )).</td>
<td>( (0,W_p(1)) ) and ( (W_p(2),1) )</td>
<td>( (W_s(1),W_s(2)) )</td>
</tr>
</tbody>
</table>

Data Types: \texttt{single} | \texttt{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


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
chepwin

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


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**

\[
[b,a] = \text{cheby1}(n,Rp,Wp)
\]

\[
[b,a] = \text{cheby1}(n,Rp,Wp,\text{ftype})
\]

\[
[z,p,k] = \text{cheby1}(__)
\]

\[
[A,B,C,D] = \text{cheby1}(__)
\]

\[
[\text{___}] = \text{cheby1}(\text{___},'s')
\]

**Description**

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

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

**Note:** See “Limitations” on page 1-221 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.

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

Lowpass Chebyshev Type I Transfer Function

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

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

Bandstop Chebyshev Type I Filter

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

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

**Highpass Chebyshev Type I Filter**

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

```matlab
[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`.

```matlab
[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`.

```matlab
sos = ss2sos(A,B,C,D);
fvt = fvtool(sos,d,'Fs',1500);
legend(fvt,'cheby1','designfilt')
```
Comparison of Analog IIR Lowpass Filters

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

\[ n = 5; \]
\[ f = 2e9; \]

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

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

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

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

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

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.

\[ [\text{z3}, \text{p3}, \text{k3}] = \text{ellip}(n, 3, 30, 2\pi f, 's'); \]
\[ [\text{b3}, \text{a3}] = \text{zp2tf}(\text{z3}, \text{p3}, \text{k3}); \]
\[ [\text{h3}, \text{w3}] = \text{freqs}(\text{b3}, \text{a3}, 4096); \]

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

\[ \text{plot}(\text{wb}/(2e9\pi), \text{mag2db}(\text{abs}(\text{hb}))) \]
\[ \text{hold on} \]
\[ \text{plot}(\text{w1}/(2e9\pi), \text{mag2db}(\text{abs}(\text{h1}))) \]
\[ \text{plot}(\text{w2}/(2e9\pi), \text{mag2db}(\text{abs}(\text{h2}))) \]
\[ \text{plot}(\text{we}/(2e9\pi), \text{mag2db}(\text{abs}(\text{h3}))) \]
\[ \text{axis([0 4 -40 5])} \]
\[ \text{grid} \]
\[ \text{xlabel('Frequency (GHz)')} \]
ylabel('Attenuation (dB)')
legend('butter','cheby1','cheby2','ellip')

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

n — Filter order
integer scalar

Filter order, specified as an integer scalar.

Data Types: double

Rp — Peak-to-peak passband ripple
positive scalar

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

If your specification, 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 \pi rad/sample.

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

Data Types: double

ftype — Filter type
'low' | 'bandpass' | 'high' | 'stop'

cheby1
Filter type, specified as one of the following:

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

Output Arguments

\( b, a \) — Transfer function coefficients
row vectors

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

- For digital filters, the transfer function is expressed in terms of \( b \) and \( a \) as
  \[
  H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2) z^{-1} + \ldots + b(n+1) z^{-n}}{a(1) + a(2) z^{-1} + \ldots + a(n+1) z^{-n}}.
  \]
- For analog filters, the transfer function is expressed in terms of \( b \) and \( a \) as
  \[
  H(s) = \frac{B(s)}{A(s)} = \frac{b(1) s^n + b(2) s^{n-1} + \ldots + b(n+1)}{a(1) s^n + a(2) s^{n-1} + \ldots + a(n+1)}.
  \]

Data Types: double

\( z, p, k \) — Zeros, poles, and gain
column vectors, scalar

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

- For digital filters, the transfer function is expressed in terms of \( z \), \( p \), and \( k \) as
  \[
  H(z) = k \frac{(1 - z(1) z^{-1})(1 - z(2) z^{-1}) \ldots (1 - z(n) z^{-1})}{(1 - p(1) z^{-1})(1 - p(2) z^{-1}) \ldots (1 - p(n) z^{-1})}.
  \]
• For analog filters, the transfer function is expressed in terms of \( z, p, \) and \( k \) as

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

Data Types: double

**A, B, C, D — State-space matrices**

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

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

\[
\begin{align*}
x(k + 1) &= A x(k) + B u(k) \\
y(k) &= C x(k) + D u(k).
\end{align*}
\]

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

\[
\begin{align*}
\dot{x} &= A x + B u \\
y &= C x + D u.
\end{align*}
\]

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.

\[
\begin{align*}
n &= 6; \\
Rp &= 0.1;
\end{align*}
\]
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 \( W_p \) or \( w_1 \) and \( w_2 \).
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

\[
[b,a] = \text{cheby2}(n,Rs,Ws) \\
[b,a] = \text{cheby2}(n,Rs,Ws,ftype) \\
[z,p,k] = \text{cheby2}(\_\_) \\
[A,B,C,D] = \text{cheby2}(\_\_) \\
[\_\_] = \text{cheby2}(\_\_, 's')
\]

Description

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

\[b,a\] = \text{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 1-235 for information about numerical issues that affect forming the transfer function.

\[z,p,k\] = \text{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.

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

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

Lowpass Chebyshev Type II Transfer Function

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

\[ [b,a] = \text{cheby2}(6,40,0.6); \]
\[ \text{freqz}(b,a) \]
Bandstop Chebyshev Type II Filter

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

```
[b,a] = cheby2(3,50,[0.2 0.6], 'stop');
freqz(b,a)
```

![Magnitude and Phase Responses](image)
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.

\[
\begin{align*}
[A,B,C,D] &= \text{cheby2}(10,40,[500\quad 560]/750); \\
d &= \text{designfilt}('\text{bandpassiir}', 'FilterOrder', 20, ...) \\
& \quad '\text{StopbandFrequency1}', 500, '\text{StopbandFrequency2}', 560, ... \\
& \quad '\text{StopbandAttenuation}', 40, '\text{SampleRate}', 1500); \\
\end{align*}
\]
Convert the state-space representation to second-order sections. Visualize the frequency responses using `fvtool`.

```matlab
sos = ss2sos(A,B,C,D);
fvt = fvtool(sos,d,'Fs',1500);
legend(fvt,'cheby2','designfilt')
```
**Comparison of Analog IIR Lowpass Filters**

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

```matlab
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.

```matlab
[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.

```matlab
[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.

```matlab
[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.

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

\( n \) — Filter order
integer scalar

Filter order, specified as an integer scalar.

Data Types: double

\( Rs \) — Stopband attenuation
positive scalar

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

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

Data Types: double

\( Ws \) — Stopband edge frequency
scalar | two-element vector

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

- If \( Ws \) is a scalar, then \texttt{cheby2} designs a lowpass or highpass filter with edge frequency \( Ws \).
  
  If \( Ws \) is the two-element vector \([w1 \ w2]\), where \( w1 < w2 \), then \texttt{cheby2} designs a bandpass or bandstop filter with lower edge frequency \( w1 \) and higher edge frequency \( w2 \).

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

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

Data Types: double

\( ftype \) — Filter type
'low' | 'bandpass' | 'high' | 'stop'

\texttt{cheby2}
Filter type, specified as one of the following:

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

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

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

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

Data Types: double

\( z, p, k \) — Zeros, poles, and gain

column vectors, scalar

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

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

\[
H(z) = k \frac{(1 - z(1)z^{-1})(1 - z(2)z^{-1})\cdots(1 - z(n)z^{-1})}{(1 - p(1)z^{-1})(1 - p(2)z^{-1})\cdots(1 - p(n)z^{-1})}.
\]
For analog filters, the transfer function is expressed in terms of $z$, $p$, and $k$ as

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

Data Types: double

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

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

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

$$x(k + 1) = A x(k) + B u(k)$$
$$y(k) = C x(k) + D u(k).$$

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

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

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.

```matlab
n = 6;
Rs = 80;
```
Wn = [2.5e6 29e6]/500e6;
ftype = 'bandpass';

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

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

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

Chebyshev Type II filters are monotonic in the passband and equiripple in the stopband. Type II filters do not roll off as fast as Type I filters, but are free of passband ripple.

cheby2 uses a five-step algorithm:

1. It finds the lowpass analog prototype poles, zeros, and gain using the function cheb2ap.
2. It converts poles, zeros, and gain into state-space form.
3. If required, it uses a state-space transformation to convert the lowpass filter into a bandpass, highpass, or bandstop filter with the desired frequency constraints.
4. For digital filter design, it uses bilinear to convert the analog filter into a digital filter through a bilinear transformation with frequency prewarping. Careful frequency adjustment the analog filters and the digital filters to have the same frequency response magnitude at \( W_s \) or \( w_1 \) and \( w_2 \).
5. It converts the state-space filter back to transfer function or zero-pole-gain form, as required.

 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 | cheb1ord | cheb2ap | cheby1 | designfilt | ellip | filter | sosfilt
Introduced before R2006a
chirp

Swept-frequency cosine

Syntax

\[ y = \text{chirp}(t, f_0, t_1, f_1) \]
\[ y = \text{chirp}(t, f_0, t_1, f_1, \text{method}) \]
\[ y = \text{chirp}(t, f_0, t_1, f_1, \text{method}, \phi) \]
\[ y = \text{chirp}(t, f_0, t_1, f_1, '\text{quadratic}', \phi, \text{shape}) \]

Description

\( y = \text{chirp}(t, f_0, t_1, f_1) \) generates samples of a linear swept-frequency cosine signal at the time instances defined in array \( t \). The instantaneous frequency at time 0 is \( f_0 \), and the instantaneous frequency at time \( t_1 \) is \( f_1 \).

\( y = \text{chirp}(t, f_0, t_1, f_1, \text{method}) \) specifies an alternative sweep method option.

\( y = \text{chirp}(t, f_0, t_1, f_1, \text{method}, \phi) \) specifies the initial phase.

\( y = \text{chirp}(t, f_0, t_1, f_1, '\text{quadratic}', \phi, \text{shape}) \) specifies the shape of the spectrogram of a quadratic swept-frequency signal.

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.

\[ t = 0:1/1e3:2; \]
\[ y = \text{chirp}(t, 0, 1, 250); \]
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.

```matlab
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.

```matlab
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.

\[ \text{pspectrum}(y,t,'\text{spectrogram'},'\text{TimeResolution}',0.1, \ldots '\text{OverlapPercent}',99,'\text{Leakage}',0.85) \]

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)
**Input Arguments**

**t — Time array**
vector

Time array, specified as a vector.
Data Types: single | double

**f0 — Instantaneous frequency at time 0**
positive scalar in Hz

Initial instantaneous frequency at time 0, specified as a positive 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) | positive scalar in Hz

Instantaneous frequency at time t1, specified as a positive 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 \( f(t) \) given by
  \[
  f(t) = f_0 + \beta t,
  \]
  where
  \[
  \beta = (f_1 - f_0)/t_1
  \]
  and the default value for \( f_0 \) is 0. The coefficient \( \beta \) 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 \( f_0 < f_1 \).
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™.

See Also

\( \text{cos} | \text{diric} | \text{gauspuls} | \text{pulstran} | \text{rectpuls} | \text{sawtooth} | \text{sin} | \text{sinc} | \text{square} | \text{tripuls} \)

Introduced before R2006a
**concatenate**

Concatenate two or more labeled signal sets

**Syntax**

\[ lssnew = concatenate(lss1, \ldots, lssN) \]

**Description**

\[ lssnew = concatenate(lss1, \ldots, lssN) \] concatenates \( N \) labeled signal set objects, \( lss1, \ldots, lssN \), 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.

```matlab
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 \texttt{lss}.

\begin{verbatim}
lsslss = lss

lsslss =
    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.
\end{verbatim}

Concatenate the two signal sets.

\begin{verbatim}
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.
\end{verbatim}

**Input Arguments**

\begin{verbatim}
\texttt{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.
\end{verbatim}
Output Arguments

\texttt{lssnew} — Concatenated labeled signal set  
\texttt{labeledSignalSet} object

Concatenated labeled signal set, returned as a \texttt{labeledSignalSet} object. The set \texttt{lssnew} contains a signal source, label definitions, and label values that are independent of the input labeled signal sets.

See Also

Signal Labeler | labeledSignalSet | signalLabelDefinition

Introduced in R2018b
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 = 1x2

0.0008    0.0444

t1rat = tcnv\tmtx

t1rat = 52.7041

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

```
1×2
0.3773  0.0994
```

tmrat = tcnv/tmtx

```
tmrat = 3.7946
```

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 | conv2 | convn | corrmtx | dftmtx`

Introduced before R2006a
corrmtx

Data matrix for autocorrelation matrix estimation

Syntax

\[
X = \text{corrmtx}(x,m)
\]

\[
X = \text{corrmtx}(x,m,'method')
\]

\[
[X,R] = \text{corrmtx}(...)
\]

Description

\(X = \text{corrmtx}(x,m)\) returns an \((n + m)\)-by-\((m + 1)\) rectangular Toeplitz matrix \(X\), such that \(X'X\) is a (biased) estimate of the autocorrelation matrix for the length-\(n\) data vector \(x\). \(m\) must be a positive integer strictly smaller than the length of the input \(x\).

\(X = \text{corrmtx}(x,m,'method')\) computes the matrix \(X\) according to the method specified by 'method':

- 'autocorrelation': (default) \(X\) is the \((n + m)\)-by-\((m + 1)\) rectangular Toeplitz matrix that generates an autocorrelation estimate for the length-\(n\) data vector \(x\), derived using prewindowed and postwindowed data, based on an \(m\)th-order prediction error model.
- 'prewindowed': \(X\) is the \(n\)-by-\((m + 1)\) rectangular Toeplitz matrix that generates an autocorrelation estimate for the length-\(n\) data vector \(x\), derived using prewindowed data, based on an \(m\)th-order prediction error model.
- 'postwindowed': \(X\) is the \(n\)-by-\((m + 1)\) rectangular Toeplitz matrix that generates an autocorrelation estimate for the length-\(n\) data vector \(x\), derived using postwindowed data, based on an \(m\)th-order prediction error model.
- 'covariance': \(X\) is the \((n - m)\)-by-\((m + 1)\) rectangular Toeplitz matrix that generates an autocorrelation estimate for the length-\(n\) data vector \(x\), derived using nonwindowed data, based on an \(m\)th-order prediction error model.
- 'modified': \(X\) is the \(2(n - m)\)-by-\((m + 1)\) modified rectangular Toeplitz matrix that generates an autocorrelation estimate for the length-\(n\) data vector \(x\), derived using forward and backward prediction error estimates, based on an \(m\)th-order prediction error model.
[X,R] = corrmtx(...) also returns the \((m + 1)\)-by-\((m + 1)\) autocorrelation matrix estimate \(R\), calculated as \(X'X\).

**Examples**

**Modified Data and Autocorrelation Matrices**

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

```matlab
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.

```matlab
[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)')
```
Algorithms

The Toeplitz data matrix computed by `corrmtx` depends on the method you select. The matrix determined by the autocorrelation (default) method is given by the following matrix.
In this matrix, \( m \) is the same as the input argument \( m \) to \texttt{corrmtx}, and \( n \) is \texttt{length(x)}. Variations of this matrix are used to return the output \( X \) of \texttt{corrmtx} for each method:

- 'autocorrelation' — (default) \( X = X \), above.
- 'prewindowed' — \( X \) is the \( n \)-by-(\( m + 1 \)) submatrix of \( X \) whose first row is \([x(1) \ldots 0]\) and whose last row is \([x(n) \ldots x(n - m)]\).
- 'postwindowed' — \( X \) is the \( n \)-by-(\( m + 1 \)) submatrix of \( X \) whose first row is \([x(m + 1) \ldots x(1)]\) and whose last row is \([0 \ldots x(n)]\).
- 'covariance' — \( X \) is the \((n - m)\)-by-(\( m + 1 \)) submatrix of \( X \) whose first row is \([x(m + 1) \ldots x(1)]\) and whose last row is \([x(n) \ldots x(n - m)]\).
- 'modified' — \( X \) is the \( 2(n - m) \)-by-(\( m + 1 \)) matrix \( X_{\text{mod}} \) shown below.
References


Extended Capabilities

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

See Also
peig|pmusic|rooteig|rootmusic|xcorr

Introduced before R2006a
cpsd
Cross power spectral density

Syntax

\[
\begin{align*}
\text{pxy} &= \text{cpsd}(x, y) \\
\text{pxy} &= \text{cpsd}(x, y, \text{window}) \\
\text{pxy} &= \text{cpsd}(x, y, \text{window, noverlap}) \\
\text{pxy} &= \text{cpsd}(x, y, \text{window, noverlap, nfft}) \\
\text{pxy} &= \text{cpsd}(\_\_\_, 'mimo') \\
[\text{pxy}, w] &= \text{cpsd}(\_\_) \\
[\text{pxy}, f] &= \text{cpsd}(\_\_, \text{fs}) \\
[\text{pxy}, w] &= \text{cpsd}(x, y, \text{window, noverlap, w}) \\
[\text{pxy}, f] &= \text{cpsd}(x, y, \text{window, nooverlap, f, fs}) \\
[\_] &= \text{cpsd}(x, y, \_\_, \text{freqrange}) \\
\text{cpsd}(\_\_) 
\end{align*}
\]

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,__,frequrange) returns the cross power spectral density
estimate over the frequency range specified by frequrange. Valid options for frequrange
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.

```matlab
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.

```matlab
fs = 1e3;
t = (0:1/fs:1-1/fs)';
q = 2*sin(2*pi*[200 300].*t);
```
\begin{verbatim}
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 \texttt{cpsd} to plot the result.

cpsd(q,r,bartlett(256),128,2048,fs)
\end{verbatim}

By default, \texttt{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 $1/4^2$.

$$\begin{align*} 
    Fs &= 1000; \\
    t &= 0:1/Fs:0.296; \\

    x &= \cos(2\pi t \cdot 100) + 0.25 \cdot \text{randn(size(t))}; \\
    tau &= 1/400; \\
    y &= \cos(2\pi \cdot 100 \cdot (t - tau)) + 0.25 \cdot \text{randn(size(t))}; \\
\end{align*}$$

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.

$$\text{cpsd}(x,y,[],[],[],Fs)$$
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.

\[
[Cxy,F] = \text{mscohere}(x,y,[],[],[],Fs);
\]

\[
[Pxy,F] = \text{cpsd}(x,y,[],[],[],Fs);
\]

```matlab
subplot(2,1,1)
plot(F,Cxy)
title('Magnitude-Squared Coherence')
```

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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
```
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).

Generate signals corresponding to all the symbols. Sample each tone at 4 kHz for half a second. Prepare a reference table.

```matlab
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.

\[ [\sim, \text{loc}] = \max(\text{rms}(\text{abs}(\text{pxy}))); \]

digit = \text{nms(loc)}

digit = '8'
Input Arguments

x, y — Input signals
vectors | matrices

Input signals, specified as vectors or matrices.

Example: \( \cos(\pi/4*(0:159))+\text{randn}(1,160) \) 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 nooverlap 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 nooverlap 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

nooverlap — Number of overlapped samples
positive integer | []

Number of overlapped samples, specified as a positive integer.

- If window is scalar, then nooverlap 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 $[N/4.5]$, 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 $\text{max}(256, 2^p)$, 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 \([0,\pi]\) rad/sample. If \( nfft \) is odd, \( pxy \) has \((nfft + 1)/2\) rows and the interval is \([0,\pi]\) 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 \([0,2\pi]\) 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 \((-\pi,\pi]\) rad/sample for even \( nfft \) and \((-\pi,\pi)\) 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 \([0,\pi]\) when \( nfft \) is even and \([0,\pi)\) when \( nfft \) is odd.
- If \( pxy \) is two-sided, \( w \) spans the interval \([0,2\pi]\).
- If \( pxy \) is DC-centered, \( w \) spans the interval \((-\pi,\pi]\) when \( nfft \) is even and \((-\pi,\pi)\) 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**


Extended Capabilities

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

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**

\[ [i_{\text{upper}}, i_{\text{lower}}] = \text{cusum}(x) \]

\[ [i_{\text{upper}}, i_{\text{lower}}] = \text{cusum}(x, \text{climit}, \text{mshift}, \text{tmean}, \text{tdev}) \]

\[ [i_{\text{upper}}, i_{\text{lower}}] = \text{cusum}([], 'all') \]

\[ [i_{\text{upper}}, i_{\text{lower}}, u_{\text{upper}}, l_{\text{lower}}] = \text{cusum}([], []) \]

\[ \text{cusum}([]) \]

**Description**

\[ [i_{\text{upper}}, i_{\text{lower}}] = \text{cusum}(x) \]

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 \( x \).

\[ [i_{\text{upper}}, i_{\text{lower}}] = \text{cusum}(x, \text{climit}, \text{mshift}, \text{tmean}, \text{tdev}) \]

specifies \( \text{climit} \), 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.

\[ [i_{\text{upper}}, i_{\text{lower}}] = \text{cusum}([], 'all') \]

returns all the indices at which the upper and lower cumulative sums exceed the control limit.

\[ [i_{\text{upper}}, i_{\text{lower}}, u_{\text{upper}}, l_{\text{lower}}] = \text{cusum}([], []) \]

also returns the upper and lower cumulative sums.

\[ \text{cusum}([]) \]

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 \texttt{cusum} to the function using the default values of the input arguments.

\texttt{cusum(fnc)}
Compute the mean and standard deviation of the first 25 samples. Apply \texttt{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.

\begin{verbatim}
mfnc = mean(fnc(1:25));
sfnc = std(fnc(1:25));

cusum(fnc,5,1,mfnc,sfnc)
\end{verbatim}
Repeat the calculation using a negative linear trend.

\[ \text{nnc} = rnds - \text{trnd}; \]

\[ \text{cusum(nnc)} \]
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.

```matlab
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 = -3.6463e-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 4 4 5 3 5 4 4 3 4];

nms = {'Ben'; 'Jen'; 'Ken'};

Ben = [4 3 4 2 3 5 2 3 3 4 3 2 3 3 3 2 3];
Jen = [4 3 4 3 4 4 3 4 4 5 3 4 5 3 3 3 3];
Ken = [4 3 4 5 5 4 4 4 5 3 5 4 5 4 3 5];

T = table(hole', par', Ben', Jen', Ken', ...
    'VariableNames', ['hole'; 'par'; nms])

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)

Data Types: single | double
**climit — Control limit**
5 (default) | real scalar

Control limit, specified as a real scalar expressed in standard deviations.

Data Types: single | double

**mshift — Minimum mean shift to detect**
1 (default) | real scalar

Minimum mean shift to detect, specified as a real scalar expressed in standard deviations.

Data Types: single | double

**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`.

Data Types: single | double

**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`.

Data Types: single | double

**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 \(\sigma_x\), 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 \textit{cusum} by the \texttt{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 \textit{cusum} by the \texttt{climit} argument.

By default, the function returns the first violation it detects. If you specify the 'all' flag, the function returns every violation.

See Also
findchangepts | mean

Introduced in R2016a
czt

Chirp Z-transform

Syntax

\[ y = \text{czt}(x, m, w, a) \]
\[ y = \text{czt}(x) \]

Description

\[ y = \text{czt}(x, m, w, a) \] returns the chirp Z-transform of signal \( x \). The chirp Z-transform is the Z-transform of \( x \) along a spiral contour defined by \( w \) and \( a \). \( m \) is a scalar that specifies the length of the transform, \( w \) is the ratio between points along the \( z \)-plane spiral contour of interest, and scalar \( a \) is the complex starting point on that contour. The contour, a spiral or “chirp” in the \( z \)-plane, is given by

\[ z = a*(w.^(0:m-1)) \]

\( y = \text{czt}(x) \) uses the following default values:

- \( m = \text{length}(x) \)
- \( w = \exp(-j*2*pi/m) \)
- \( a = 1 \)

With these defaults, \( \text{czt} \) returns the Z-transform of \( x \) at \( m \) equally spaced points around the unit circle. This is equivalent to the discrete Fourier transform of \( x \), or \( \text{fft}(x) \). The empty matrix \([\ ]\) specifies the default value for a parameter.

If \( x \) is a matrix, \( \text{czt}(x, m, w, a) \) transforms the columns of \( x \).

Examples
CZT of a Random Vector

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

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

Narrow-Band Section of a Frequency Response

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

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

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

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

```matlab
m = 1024;
y = fft(h,m);
f1 = 100;
f2 = 150;
w = exp(-j*2*pi*(f2-f1)/(m*fs));
a = exp(j*2*pi*f1/fs);
z = czt(h,m,w,a);
```

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

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

```matlab
subplot(2,1,1)
plot(fy,abs(y))
axis([f1 f2 0 1.2])
```
Diagnostics

If \( m, w, \) or \( a \) is not a scalar, \texttt{czt} gives the following error message:
Inputs M, W, and A must be scalars.

**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**


**Extended Capabilities**

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

Usage notes and limitations:
See “Automatic dimension restriction” (MATLAB Coder).

**See Also**

fft | freqz

Introduced before R2006a
**db**

Convert energy or power measurements to decibels

**Syntax**

\[
\text{dboutput} = \text{db}(x) \\
\text{dboutput} = \text{db}(x,\text{SignalType}) \\
\text{dboutput} = \text{db}(x,R) \\
\text{dboutput} = \text{db}(x,\text{'voltage'},R)
\]

**Description**

\( \text{dboutput} = \text{db}(x) \) converts the elements of \( x \) to decibels (dB). This syntax assumes that \( x \) contains voltage measurements across a resistance of 1 Ω.

\( \text{dboutput} = \text{db}(x,\text{SignalType}) \) specifies the signal type represented by the elements of \( x \) as either \text{'voltage'} or \text{'power'}.

\( \text{dboutput} = \text{db}(x,R) \) specifies the resistance, \( R \), for voltage measurements.

\( \text{dboutput} = \text{db}(x,\text{'voltage'},R) \) is equivalent to \( \text{db}(x,R) \).

**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, \( 10\log_{10}\frac{1}{2} \).

\[
V = 1; \\
R = 2; \\
\text{dboutput} = \text{db}(V,2); \\
\text{compvoltage} = [\text{dboutput} 10*\log10(1/2)]
\]
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 $10 \log_{10}(|x|^2/R)$.
- If the input x contains power measurements, then dboutput is $10 \log_{10}x$.

See Also
db2mag | db2pow | mag2db | pow2db

Introduced in R2011b
db2mag

Convert decibels to magnitude

Syntax

\[ y = \text{db2mag}(ydb) \]

Description

\[ y = \text{db2mag}(ydb) \]

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.

```matlab
r = randn(2,4,2);
mags = db2mag(r)
```

```matlab
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
```

1-304
Use the definition to check the calculation.

\[
\text{chck} = 10^{\frac{r}{20}}
\]

\[
\text{chck} =
\begin{bmatrix}
1.0639 & 0.7710 & 1.0374 & 0.9513 \\
1.2351 & 1.1044 & 0.8602 & 1.0402
\end{bmatrix}
\]

\[
\text{chck}(:,:,2) =
\begin{bmatrix}
1.5098 & 0.8561 & 1.0871 & 1.0858 \\
1.3755 & 1.4182 & 0.9928 & 0.9767
\end{bmatrix}
\]

**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, \text{db2mag} is an element-wise operation.

Data Types: \text{single} | \text{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

\[ y = \text{db2pow}(ydb) \]

Description

\[ y = \text{db2pow}(ydb) \]
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.

\[ r = \text{randn}(2,4,2); \]
\[ \text{pows} = \text{db2pow}(r) \]

\[ \text{pows} = \]
\[ \text{pows}(:,:,1) = \]
\[ 1.1318 \quad 0.5944 \quad 1.0762 \quad 0.9050 \]
\[ 1.5254 \quad 1.2196 \quad 0.7400 \quad 1.0821 \]

\[ \text{pows}(:,:,2) = \]
\[ 2.2795 \quad 0.7328 \quad 1.1818 \quad 1.1789 \]
Use the definition to check the calculation.

\[ \text{chck} = 10.^(r/10) \]

\[ \text{chck}(:,:,1) = \]
\[ \begin{array}{cccc}
1.1318 & 0.5944 & 1.0762 & 0.9050 \\
1.5254 & 1.2196 & 0.7400 & 1.0821 \\
\end{array} \]

\[ \text{chck}(:,:,2) = \]
\[ \begin{array}{cccc}
2.2795 & 0.7328 & 1.1818 & 1.1789 \\
1.8921 & 2.0114 & 0.9856 & 0.9539 \\
\end{array} \]

**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, \text{db2pow} is an element-wise operation.

Data Types: \text{single} | \text{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™.

See Also
db | db2mag | mag2db | pow2db

Introduced in R2007b
**dct**

Discrete cosine transform

**Syntax**

\[ y = \text{dct}(x) \]

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

\[ y = \text{dct}(x,n,dim) \]

\[ y = \text{dct}(\_\_,'Type',\text{dcttype}) \]

**Description**

\( y = \text{dct}(x) \) 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 \( \text{dct} \) operates along the first array dimension with size greater than 1.

\( y = \text{dct}(x,n) \) zero-pads or truncates the relevant dimension of \( x \) to length \( n \) before transforming.

\( y = \text{dct}(x,n,dim) \) computes the transform along dimension \( dim \). To input a dimension and use the default value of \( n \), specify the second argument as empty, \( [] \).

\( y = \text{dct}(\_\_,'Type',\text{dcttype}) \) specifies the type of discrete cosine transform to compute. See “Discrete Cosine Transform” on page 1-320 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.

```matlab
load penny
surf(P)
view(2)
colormap copper
```
Compute the discrete cosine transform of the image data. Operate first along the rows and then along the columns.

```matlab
Q = dct(P, [], 1);
R = dct(Q, [], 2);
```

Find what percentage of DCT coefficients contain 99.9% of the energy in the image.

```matlab
X = R(:);
[~, ind] = sort(abs(R(:)), 'descend');
```
coeffs = 1;
while norm(X(ind(1:coeffs)))/norm(X) < 0.999
    coeffs = coeffs + 1;
end
fprintf('%3.1f%% of the coefficients are sufficient\n',coeffs/numel(R)*100)

6.9% of the coefficients are sufficient

Reconstruct the image using only the necessary coefficients.

R(abs(R) < X(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.

```matlab
load penny
surf(P)
view(2)
colormap copper
```
Compute the discrete cosine transform of the image data using the DCT-1 variant. Operate first along the rows and then along the columns.

\[ Q = \text{dct}(P,[],1,'Type',1); \]
\[ R = \text{dct}(Q,[],2,'Type',1); \]

Invert the transform. Truncate the inverse so that each dimension of the reconstructed image is one-half the length of the original.

\[ S = \text{idct}(R,\text{size}(P,2)/2,2,'Type',1); \]
\[ T = \text{idct}(S,\text{size}(P,1)/2,1,'Type',1); \]
Invert the transform again. Zero-pad the inverse so that each dimension of the reconstructed image is twice the length of the original.

\[
U = \text{idct}(R,\text{size}(P,2)*2,2,\text{'Type',1});
V = \text{idct}(U,\text{size}(P,1)*2,1,\text{'Type',1});
\]

Display the original and reconstructed images.

\[
\text{surf}(V)\hspace{1cm}\text{view}(2)\hspace{1cm}\text{shading}\hspace{0.2cm}\text{interp}\hspace{1cm}\text{hold}\hspace{0.2cm}\text{on}
\]

\[
\text{surf}(P)\hspace{1cm}\text{view}(2)\hspace{1cm}\text{shading}\hspace{0.2cm}\text{interp}
\]

\[
\text{surf}(T)\hspace{1cm}\text{view}(2)\hspace{1cm}\text{shading}\hspace{0.2cm}\text{interp}\hspace{1cm}\text{hold}\hspace{0.2cm}\text{off}\hspace{1cm}\text{axis}\hspace{0.2cm}ij\hspace{0.2cm}\text{equal}\hspace{0.2cm}\text{off}
\]
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: \( \text{sin}(2\pi(0:255)/4) \) 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 1-320 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. In particular, DCT-1 and DCT-4 are their own inverses, and DCT-2 and DCT-3 are inverses of each other.
References


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.

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

dct2 | fft | idct | idct2
Topics
“DCT for Speech Signal Compression”

Introduced before R2006a
**decimate**

Decimation — decrease sample rate by integer factor

**Syntax**

```matlab
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.

```matlab
t = 0:.00025:1;
x = sin(2*pi*30*t) + sin(2*pi*60*t);
y = decimate(x,4);
```
Plot the original and decimated signals.

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

```matlab
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.

```matlab
r = 13;
n = 16:365;
lx = length(n);
x = sin(2*pi*n/153) + cos(2*pi*n/127);
```
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 ($y$), $y(\text{end})$ matches $x(\text{end})$ when the IIR filter is used, and $y(1)$ matches $x(1)$ when the FIR filter is used.

**References**


**See Also**

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

**Introduced before R2006a**
**demod**

Demodulation for communications simulation

**Syntax**

```matlab
x = demod(y,fc,fs,'method')
x = demod(y,fc,fs,'method',opt)
x = demod(y,fc,fs,'pwm','centered')
```

**Description**

`demod` performs demodulation, that is, it obtains the original signal from a modulated version of the signal. `demod` undoes the operation performed by `modulate`.

```matlab
x = demod(y,fc,fs,'method')
```

```matlab
x = demod(y,fc,fs,'method',opt)
```

`x = demod(y,fc,fs,'method',opt)` demodulate the real carrier signal `y` with a carrier frequency `fc` and sampling frequency `fs`, using one of the options listed below for `method`. (Note that some methods accept an option, `opt`.)

**Note** Use `demod` and `modulate` in the Signal Processing Toolbox with real-valued signals to obtain real-valued outputs. `demod` and `modulate` are not intended to accept complex-valued inputs or produce complex-valued outputs.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amdsb-sc or am</td>
<td>Amplitude demodulation, double sideband, suppressed carrier. Multiplies y by a sinusoid of frequency fc and applies a fifth-order Butterworth lowpass filter using filtfilt.</td>
</tr>
<tr>
<td></td>
<td>x = y.<em>cos(2</em>pi<em>fc</em>t);</td>
</tr>
<tr>
<td></td>
<td>[b,a] = butter(5,fc*2/fs);</td>
</tr>
<tr>
<td></td>
<td>x = filtfilt(b,a,x);</td>
</tr>
<tr>
<td>amdsb-tc</td>
<td>Amplitude demodulation, double sideband, transmitted carrier. Multiplies y by a sinusoid of frequency fc and applies a fifth-order Butterworth lowpass filter using filtfilt.</td>
</tr>
<tr>
<td></td>
<td>x = y.<em>cos(2</em>pi<em>fc</em>t);</td>
</tr>
<tr>
<td></td>
<td>[b,a] = butter(5,fc*2/fs);</td>
</tr>
<tr>
<td></td>
<td>x = filtfilt(b,a,x);</td>
</tr>
<tr>
<td></td>
<td>If you specify opt, demod subtracts scalar opt from x. The default value for opt is 0.</td>
</tr>
<tr>
<td>amssb</td>
<td>Amplitude demodulation, single sideband. Multiplies y by a sinusoid of frequency fc and applies a fifth-order Butterworth lowpass filter using filtfilt.</td>
</tr>
<tr>
<td></td>
<td>x = y.<em>cos(2</em>pi<em>fc</em>t);</td>
</tr>
<tr>
<td></td>
<td>[b,a] = butter(5,fc*2/fs);</td>
</tr>
<tr>
<td></td>
<td>x = filtfilt(b,a,x);</td>
</tr>
<tr>
<td>fm</td>
<td>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.</td>
</tr>
<tr>
<td>pm</td>
<td>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.</td>
</tr>
<tr>
<td><strong>Method</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>ppm</td>
<td>Pulse-position demodulation. Finds the pulse positions of a pulse-position modulated signal $y$. For correct demodulation, the pulses cannot overlap. $x$ is length $\text{length}(t) * fc/fs$.</td>
</tr>
<tr>
<td>pwm</td>
<td>Pulse-width demodulation. Finds the pulse widths of a pulse-width modulated signal $y$. <code>demod</code> 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.</td>
</tr>
</tbody>
</table>
| qam | Quadrature amplitude demodulation.  
\[
[x1,x2] = \text{demod}(y,fc,fs,'qam')
\]  
Multiplies $y$ by a cosine and a sine of frequency $fc$ and applies a fifth-order Butterworth lowpass filter using `filtfilt`.  
\[
x1 = y.*\cos(2\pi fc*t);
\]
\[
x2 = y.*\sin(2\pi fc*t);
\]
\[
[b,a] = \text{butter}(5,fc*2/fs);
\]
\[
x1 = \text{filtfilt}(b,a,x1);
\]
\[
x2 = \text{filtfilt}(b,a,x2);
\]

The default method is 'am'. In all cases except 'ppm' and 'pwm', $x$ is the same size as $y$.

If $y$ is a matrix, `demod` demodulates its columns.

\[
x = \text{demod}(y,fc,fs,'pwm','centered')
\]  
Finds the pulse widths assuming they are centered at the beginning of each period. $x$ is length $\text{length}(y) * fc/fs$.

### See Also
`fskdemod` | `genqamdemod` | `modulate` | `mskdemod` | `pamdemod` | `pmdemod` | `qamdemod` | `vco`

**Introduced before R2006a**
designfilt

Design digital filters

Syntax

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

Description

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

- **Frequency constraints** correspond to the frequencies at which a filter exhibits a desired behavior. Examples include 'PassbandFrequency' and 'CutoffFrequency'. (See the complete list under “Name-Value Pair Arguments” on page 1-364.) You must always specify the frequency constraints.

- **Magnitude constraints** describe the filter behavior at particular frequency ranges. Examples include 'PassbandRipple' and 'StopbandAttenuation'. (See the complete list under “Name-Value Pair Arguments” on page 1-364.) designfilt provides default values for magnitude constraints left unspecified. In arbitrary-magnitude designs you must always specify the vectors of desired amplitudes.

- 'FilterOrder'. Some design methods let you specify the order. Others produce minimum-order designs. That is, they generate the smallest filters that satisfy the specified constraints.

- 'DesignMethod' is the algorithm used to design the filter. Examples include constrained least squares ('cls') and 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 options** are parameters specific to a given design method. Examples include 'Window' for the 'window' method and optimization 'Weights' for arbitrary-
magnitude equiripple designs. (See the complete list under “Name-Value Pair Arguments” on page 1-364.) designfilt provides default values for design options left unspecified.

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

**Note** If you specify an incomplete or inconsistent set of name-value pairs at the command line, designfilt offers to open a “Filter Design Assistant” on page 1-373. 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 1-373. The assistant helps you design the filter, comments out the faulty code in the function or script, and pastes the corrected MATLAB code on the next line.

- Use `filter` in the form `dataOut = filter(d,dataIn)` to filter a signal with a `digitalFilter`, `d`.
- Use `FVTool` to visualize a `digitalFilter`, `d`.
- 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 1-373 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 $0.25\pi$ rad/s, stopband frequency $0.35\pi$ rad/s, passband ripple 0.5 dB, and stopband attenuation
65 dB. Use a Kaiser window to design the filter. Visualize its magnitude response. Use it to filter a vector of random data.

```matlab
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.

```matlab
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.

```matlab
dataIn = randn(1000,1); dataOut = filter(lpFilt,dataIn);
```
Output the filter coefficients, expressed as second-order sections.

\[ \text{sos} = \text{lpFilt.Coefficients} \]

\[ \text{sos} = 4 \times 6 \]

\[
\begin{array}{cccccc}
0.2666 & 0.5333 & 0.2666 & 1.0000 & -0.8346 & 0.9073 \\
0.1943 & 0.3886 & 0.1943 & 1.0000 & -0.9586 & 0.7403 \\
0.1012 & 0.2023 & 0.1012 & 1.0000 & -1.1912 & 0.5983 \\
0.0318 & 0.0636 & 0.0318 & 1.0000 & -1.3810 & 0.5090 \\
\end{array}
\]

**Highpass FIR Filter**

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

\[
\text{hpFilt} = \text{designfilt('highpassfir','StopbandFrequency',0.25, ...}
\quad \text{'PassbandFrequency',0.35,'PassbandRipple',0.5, ...}
\quad \text{'StopbandAttenuation',65,'DesignMethod','kaiserwin');}
\]

\[
\text{fvtool(hpFilt)}
\]
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 = 1x21
**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.

```matlab
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, ...
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.

dataIn = randn(1000,1);
dataOut = filter(bsFilt, dataIn);
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/s. Display in linear units the magnitude response of the filter. Use it to filter a 1000-sample vector of random data.
hFilt = designfilt('hilbertfir','FilterOrder',18,'TransitionWidth',0.25);
fvtool(hFilt,'MagnitudeDisplay','magnitude')

dataIn = randn(1000,1);
dataOut = filter(hFilt,dataIn);

**Arbitrary-Magnitude FIR Filter**

You are given a signal sampled at 1 kHz. Design 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 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. Click one of the possible values of **resp** to expand a table of allowed specification sets.

'lowpassfir'

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

```matlab
d = designfilt('lowpassfir', ... % Response type
    'FilterOrder',25, ... % Filter order
    'PassbandFrequency',400, ... % Frequency constraints
    'StopbandFrequency',550, ... % Frequency constraints
    'DesignMethod','ls', ... % Design method
    'PassbandWeight',1, ... % Design method options
    'StopbandWeight',2, ... % 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.

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

'lowpassiir'

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

```matlab
d = designfilt('lowpassiir', ...    % Response type
    'PassbandFrequency',400, ...    % Frequency constraints
    'StopbandFrequency',550, ...    % Magnitude constraints
    'PassbandRipple',4, ...         % Design method
    'StopbandAttenuation',55, ...   % Design method
    'DesignMethod','ellip', ...    % Design method
```
• 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.

<table>
<thead>
<tr>
<th>Filter Order Argument Names</th>
<th>Frequency Constraint Argument Names</th>
<th>Magnitude Constraint Argument Names</th>
<th>'DesignMethod' Argument Values</th>
<th>Design Option Argument Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A (Minimum-order design)</td>
<td>'PassbandFrequency'</td>
<td>'PassbandRipple'</td>
<td>'butter' (default)</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td></td>
<td>'StopbandFrequency'</td>
<td>'StopbandAttenuation'</td>
<td>'cheby1'</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'cheby2'</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'ellip'</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'HalfPowerFrequency'</td>
<td>N/A</td>
<td>'butter'</td>
<td>N/A</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'PassbandFrequency'</td>
<td>'PassbandRipple'</td>
<td>'cheby1'</td>
<td>N/A</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'PassbandFrequency'</td>
<td>'PassbandRipple'</td>
<td>'ellip'</td>
<td>N/A</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'StopbandFrequency'</td>
<td>'StopbandAttenuation'</td>
<td>'cheby2'</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Neglecting the order of the filter, the table provides the following options:

<table>
<thead>
<tr>
<th>Filter Order Argument Names</th>
<th>Frequency Constraint Argument Names</th>
<th>Magnitude Constraint Argument Names</th>
<th>'DesignMethod' Argument Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>'NumeratorOrder'</td>
<td>'HalfPowerFrequency'</td>
<td>N/A</td>
<td>'butter'</td>
</tr>
<tr>
<td>'DenominatorOrder'</td>
<td></td>
<td></td>
<td>N/A</td>
</tr>
</tbody>
</table>

For the 'highpassfir' option:

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

```matlab
d = designfilt('highpassfir', ... % Response type
    'StopbandFrequency',400, ... % Frequency constraints
    'PassbandFrequency',550, ...
    'StopbandAttenuation',55, ...
    'PassbandRipple',4, ... % Magnitude constraints
    '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.

<table>
<thead>
<tr>
<th>Filter Order Argument Names</th>
<th>Frequency Constraint Argument Names</th>
<th>Magnitude Constraint Argument Names</th>
<th>'DesignMethod' Argument Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A (Minimum-order design)</td>
<td>'StopbandFrequency'</td>
<td>'StopbandAttenuation'</td>
<td>'equiripple' (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
</tr>
</tbody>
</table>
### Filter Order Argument Names

<table>
<thead>
<tr>
<th>Filter Order Argument Names</th>
<th>Frequency Constraint Argument Names</th>
<th>Magnitude Constraint Argument Names</th>
<th>'DesignMethod' Argument Values</th>
<th>Design Option Argument Names</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'PassbandFrequency'</td>
<td>'PassbandRipple'</td>
<td>'kaiserwin'</td>
<td>'ScalePassband'</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'CutoffFrequency'</td>
<td>N/A</td>
<td>'window'</td>
<td>'Window'</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'CutoffFrequency'</td>
<td>'StopbandAttenuation'</td>
<td>'cls'</td>
<td>'PassbandOffset'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'PassbandRipple'</td>
<td></td>
<td>'ZeroPhase'</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'StopbandFrequency'</td>
<td>N/A</td>
<td>'equiripple' (default)</td>
<td>'PassbandWeight'</td>
</tr>
<tr>
<td></td>
<td>'PassbandFrequency'</td>
<td></td>
<td></td>
<td>'StopbandWeight'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'ls'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'PassbandWeight'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'StopbandWeight'</td>
</tr>
</tbody>
</table>

### 'highpassiir'

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

```matlab
d = designfilt('highpassiir', ...               % Response type
    'StopbandFrequency',400, ...                 % Frequency constraints
    'PassbandFrequency',550, ...                 % Frequency constraints
    'StopbandAttenuation',55, ...                 % Magnitude constraints
    'PassbandRipple',4, ...                       % Magnitude constraints
    '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.

<table>
<thead>
<tr>
<th>Filter Order Argument Names</th>
<th>Frequency Constraint Argument Names</th>
<th>Magnitude Constraint Argument Names</th>
<th>'DesignMethod' Argument Values</th>
<th>Design Option Argument Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A (Minimum-order design)</td>
<td>'StopbandFrequency'</td>
<td>'StopbandAttenuation'</td>
<td>'butter' (default)</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td></td>
<td>'PassbandFrequency'</td>
<td>'PassbandRipple'</td>
<td>'cheby1'</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'cheby2'</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'ellip'</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'HalfPowerFrequency'</td>
<td>N/A</td>
<td>'butter'</td>
<td>N/A</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'PassbandFrequency'</td>
<td>'PassbandRipple'</td>
<td>'cheby1'</td>
<td>N/A</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'PassbandFrequency'</td>
<td>'StopbandAttenuation'</td>
<td>'ellip'</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'PassbandRipple'</td>
<td>'cheby2'</td>
<td>N/A</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'StopbandFrequency'</td>
<td>'StopbandAttenuation'</td>
<td>'cheby2'</td>
<td>N/A</td>
</tr>
<tr>
<td>'NumeratorOrder'</td>
<td>'HalfPowerFrequency'</td>
<td>N/A</td>
<td>'butter'</td>
<td>N/A</td>
</tr>
<tr>
<td>'DenominatorOrder'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Choose this option to design a finite impulse response (FIR) bandpass filter. This example uses the fourth specification set from the following table.

```matlab
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.

<table>
<thead>
<tr>
<th>Filter Order Argument Names</th>
<th>Frequency Constraint Argument Names</th>
<th>Magnitude Constraint Argument Names</th>
<th>'DesignMethod' Argument Values</th>
<th>Design Option Argument Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A (Minimum-order design)</td>
<td>'StopbandFrequency1'</td>
<td>'StopbandAttenuation1'</td>
<td>'equiripple' (default)</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>'PassbandFrequency1'</td>
<td>'PassbandRipple'</td>
<td>'kaiserwin'</td>
<td>'ScalePassband'</td>
</tr>
<tr>
<td></td>
<td>'PassbandFrequency2'</td>
<td>'StopbandAttenuation2'</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'StopbandFrequency2'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filter Order Argument Names</td>
<td>Frequency Constraint Argument Names</td>
<td>Magnitude Constraint Argument Names</td>
<td>'DesignMethod' Argument Values</td>
<td>Design Option Argument Names</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------</td>
<td>------------------------------------</td>
<td>-------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'CutoffFrequency1'</td>
<td>N/A</td>
<td>'window'</td>
<td>'Window'</td>
</tr>
<tr>
<td></td>
<td>'CutoffFrequency2'</td>
<td></td>
<td></td>
<td>'ScalePassband'</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'CutoffFrequency1'</td>
<td>'StopbandAttenuation1'</td>
<td>'cls'</td>
<td>'PassbandOffset'</td>
</tr>
<tr>
<td></td>
<td>'CutoffFrequency2'</td>
<td>'PassbandRipple'</td>
<td></td>
<td>'ZeroPhase'</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'StopbandFrequency1'</td>
<td>N/A</td>
<td>'equiripple' (default)</td>
<td>'StopbandWeight1'</td>
</tr>
<tr>
<td></td>
<td>'PassbandFrequency1'</td>
<td></td>
<td></td>
<td>'PassbandWeight'</td>
</tr>
<tr>
<td></td>
<td>'PassbandFrequency2'</td>
<td></td>
<td></td>
<td>'StopbandWeight2'</td>
</tr>
<tr>
<td></td>
<td>'StopbandFrequency2'</td>
<td></td>
<td></td>
<td>'ls'</td>
</tr>
</tbody>
</table>

'bandpassiir'

Choose this option to design an infinite impulse response (IIR) bandpass filter. This example uses the first specification set from the following table.
d = designfilt('bandpassiir', ... % Response type
    'StopbandFrequency1',400, ... % Frequency constraints
    'PassbandFrequency1',450, ...
    'PassbandFrequency2',600, ...
    'StopbandFrequency2',650, ...
    'StopbandAttenuation1',40, ... % Magnitude constraints
    'PassbandRipple',1, ...
    'StopbandAttenuation2',50, ...
    'DesignMethod','ellip', ... % Design method
    'MatchExactly','passband', ... % Design method options
    'SampleRate',2000)               % Sample rate

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

<table>
<thead>
<tr>
<th>Filter Order Argument Names</th>
<th>Frequency Constraint Argument Names</th>
<th>Magnitude Constraint Argument Names</th>
<th>'DesignMethod' Argument Values</th>
<th>Design Option Argument Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A (Minimum-order design)</td>
<td>'StopbandFrequency1'</td>
<td>'StopbandAttenuation1'</td>
<td>'butter' (default)</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td></td>
<td>'PassbandFrequency1'</td>
<td>'PassbandRipple'</td>
<td>'cheby1'</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td></td>
<td>'PassbandFrequency2'</td>
<td>'StopbandAttenuation2'</td>
<td>'cheby2'</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td></td>
<td>'StopbandFrequency2'</td>
<td></td>
<td>'ellip'</td>
<td>'MatchExactly'</td>
</tr>
</tbody>
</table>
Choose this option to design a finite impulse response (FIR) bandstop filter. This example uses the fourth specification set from the following table.

```matlab
d = designfilt('bandstopfir', ...       % Response type
    'FilterOrder',32, ...            % Filter order
    'PassbandFrequency1',400, ...    % Frequency constraints
    'StopbandFrequency1',500, ...    % Frequency constraints
    'PassbandFrequency2',700, ...    % Frequency constraints
    'StopbandFrequency2',850, ...    % Frequency constraints
    'DesignMethod','ls', ...         % Design method
);```
• 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.

<table>
<thead>
<tr>
<th>Filter Order Argument Names</th>
<th>Frequency Constraint Argument Names</th>
<th>Magnitude Constraint Argument Names</th>
<th>'DesignMethod' Argument Values</th>
<th>Design Option Argument Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A (Minimum-order design)</td>
<td>'PassbandFrequency1'</td>
<td>'PassbandRipple1'</td>
<td>'equiripple' (default)</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>'StopbandFrequency1'</td>
<td>'StopbandAttenuation'</td>
<td>'kaiserwin'</td>
<td>'ScalePassband'</td>
</tr>
<tr>
<td></td>
<td>'StopbandFrequency2'</td>
<td>'PassbandRipple2'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'CutoffFrequency1'</td>
<td>N/A</td>
<td>'window'</td>
<td>'ScalePassband'</td>
</tr>
<tr>
<td></td>
<td>'CutoffFrequency2'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filter Order Argument Names</td>
<td>Frequency Constraint Argument Names</td>
<td>Magnitude Constraint Argument Names</td>
<td>'DesignMethod' Argument Values</td>
<td>Design Option Argument Names</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------</td>
<td>------------------------------------</td>
<td>--------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'CutoffFrequency1'</td>
<td>'PassbandRipple1'</td>
<td>'cls'</td>
<td>'PassbandOffset'</td>
</tr>
<tr>
<td></td>
<td>'CutoffFrequency2'</td>
<td>'StopbandAttenuation'</td>
<td></td>
<td>'ZeroPhase'</td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'PassbandFrequency1'</td>
<td>N/A</td>
<td>'equiripple'</td>
<td>'PassbandWeight1'</td>
</tr>
<tr>
<td></td>
<td>'StopbandFrequency1'</td>
<td></td>
<td>(default)</td>
<td>'StopbandWeight'</td>
</tr>
<tr>
<td></td>
<td>'StopbandFrequency2'</td>
<td></td>
<td></td>
<td>'PassbandWeight2'</td>
</tr>
<tr>
<td></td>
<td>'PassbandFrequency2'</td>
<td></td>
<td></td>
<td>'ls'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'PassbandWeight1'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'StopbandWeight'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'PassbandWeight2'</td>
</tr>
</tbody>
</table>

**'bandstopiir'**

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

```matlab
d = designfilt('bandstopiir', ...            % Response type
     'PassbandFrequency1',400, ...              % Frequency constraints
     'StopbandFrequency1',500, ...              % Frequency constraints
     'StopbandFrequency2',700, ...              % Frequency constraints
     'PassbandFrequency2',850, ...              % Frequency constraints
     'PassbandRipple1',1, ...                   % Magnitude constraints
     'StopbandAttenuation',55, ...              % Magnitude constraints
```

1-359
• 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.

<table>
<thead>
<tr>
<th>Filter Order Argument Names</th>
<th>Frequency Constraint Argument Names</th>
<th>Magnitude Constraint Argument Names</th>
<th>'DesignMethod' Argument Values</th>
<th>Design Option Argument Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A (Minimum-order design)</td>
<td>'PassbandFrequency1'</td>
<td>'PassbandRipple1'</td>
<td>'butter' (default)</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td></td>
<td>'StopbandFrequency1'</td>
<td></td>
<td>'cheby1'</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td></td>
<td>'StopbandFrequency2'</td>
<td></td>
<td>'cheby2'</td>
<td>'MatchExactly'</td>
</tr>
<tr>
<td></td>
<td>'PassbandFrequency2'</td>
<td></td>
<td>'ellip'</td>
<td>'MatchExactly'</td>
</tr>
</tbody>
</table>

<p>| 'FilterOrder'               | 'HalfPowerFrequency1'              | N/A                                | 'butter'                      | N/A                         |
|                             | 'HalfPowerFrequency2'              |                                    |                               |                             |
| 'FilterOrder'               | 'PassbandFrequency1'               | 'PassbandRipple'                   | 'cheby1'                      | N/A                         |
|                             | 'PassbandFrequency2'               |                                    |                               |                             |</p>
<table>
<thead>
<tr>
<th>Filter Order Argument Names</th>
<th>Frequency Constraint Argument Names</th>
<th>Magnitude Constraint Argument Names</th>
<th>'DesignMethod' Argument Values</th>
<th>Design Option Argument Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>'FilterOrder'</td>
<td>'PassbandFrequency1'</td>
<td>'PassbandRipple'</td>
<td>'ellip'</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>'PassbandFrequency2'</td>
<td>'StopbandAttenuation'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'FilterOrder'</td>
<td>'StopbandFrequency1'</td>
<td>'StopbandAttenuation'</td>
<td>'cheby2'</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>'StopbandFrequency2'</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

'differentiatorfir'

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

d = designfilt('differentiatorfir', ... % Response type
    'FilterOrder',42, ... % Filter order
    'PassbandFrequency',400, ... % Frequency constraints
    'StopbandFrequency',500, ... % Frequency constraints
    'DesignMethod','equiripple', ... % Design method
    'PassbandWeight',1, ... % Design method options
    'StopbandWeight',4, ... % Design method options
    '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.
### 'hilbertfir'

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

```matlab
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.
'arbmagfir'

Choose this option to design a finite impulse response (FIR) filter of arbitrary magnitude response. This example uses the second specification set from the following table.

```matlab
d = designfilt('arbmagfir', ...         % Response type
    'FilterOrder',88, ...            % Filter order
    'NumBands',4, ...                % Frequency constraints
    'BandFrequencies1',[0 20], ...   % Frequency constraints
    'BandFrequencies2',[25 40], ...  % Frequency constraints
    'BandFrequencies3',[45 65], ...  % Frequency constraints
    'BandFrequencies4',[70 100], ... % Frequency constraints
    'BandAmplitudes1',[2 2], ...     % Magnitude constraints
    'BandAmplitudes2',[0 0], ...     % Magnitude constraints
    'BandAmplitudes3',[1 1], ...     % Magnitude constraints
    'BandAmplitudes4',[0 0], ...     % Magnitude constraints
    'DesignMethod','ls', ...         % Design method
    'BandWeights1',[1 1]/10, ...     % Design method options
    'BandWeights2',[3 1], ...        % Design method options
    'BandWeights3',[2 4], ...        % Design method options
    'BandWeights4',[5 1], ...        % Design method options
    '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.

<table>
<thead>
<tr>
<th>Filter Order Argument Names</th>
<th>Frequency Constraint Argument Names</th>
<th>Magnitude Constraint Argument Names</th>
<th>'DesignMethod' Argument Values</th>
<th>Design Option Argument Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>'FilterOrder'</td>
<td>'Frequencies'</td>
<td>'Amplitudes'</td>
<td>'freqsamp' (default)</td>
<td>'Window'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'equiripple'</td>
<td>'Weights'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'ls'</td>
<td>'Weights'</td>
</tr>
<tr>
<td>Filter Order Argument Names</td>
<td>Frequency Constraint Argument Names</td>
<td>Magnitude Constraint Argument Names</td>
<td>'DesignMethod' Argument Values</td>
<td>Design Option Argument Names</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------</td>
<td>------------------------------------</td>
<td>-------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>'FilterOrder', 'NumBands'</td>
<td>'BandFrequencies1'</td>
<td>'BandAmplitudes1'</td>
<td>'equiripple' (default)</td>
<td>'BandWeights1'</td>
</tr>
<tr>
<td></td>
<td>'BandFrequencies2'</td>
<td>'BandAmplitudes2'</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>'BandFrequenciesN'</td>
<td>'BandAmplitudesN'</td>
<td></td>
<td>'BandWeightsN'</td>
</tr>
<tr>
<td></td>
<td>'ls'</td>
<td></td>
<td></td>
<td>'BandWeights1'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'BandWeightsN'</td>
</tr>
</tbody>
</table>

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 comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Not all combinations of Name, Value pairs are valid. The valid combinations depend on the filter response that you need and on the frequency and magnitude constraints of your design.

Example: 'FilterOrder', 20, 'CutoffFrequency', 0.4 suffices to specify a lowpass FIR filter.

**Filter Order**

FilterOrder — Filter order

positive integer scalar
Filter order, specified as the comma-separated pair consisting of 'FilterOrder' and a positive integer scalar.

Data Types: double

NumeratorOrder — Numerator order
positive integer scalar

Numerator order of an IIR design, specified as the comma-separated pair consisting of 'NumeratorOrder' and a positive integer scalar.

Data Types: double

DenominatorOrder — Denominator order
positive integer scalar

Denominator order of an IIR design, specified as the comma-separated pair consisting of 'DenominatorOrder' and a positive integer scalar.

Data Types: double

Frequency Constraints

PassbandFrequency, PassbandFrequency1, PassbandFrequency2 — Passband frequency
positive scalar

Passband frequency, specified as the comma-separated pair consisting of 'PassbandFrequency' and a positive scalar. The frequency value must be within the Nyquist range.

'PassbandFrequency1' is the lower passband frequency for a bandpass or bandstop design.

'PassbandFrequency2' is the higher passband frequency for a bandpass or bandstop design.

Data Types: double

StopbandFrequency, StopbandFrequency1, StopbandFrequency2 — Stopband frequency
positive scalar
Stopband frequency, specified as the comma-separated pair consisting of 'StopbandFrequency' and a positive scalar. The frequency value must be within the Nyquist range.

'StopbandFrequency1' is the lower stopband frequency for a bandpass or bandstop design.

'StopbandFrequency2' is the higher stopband frequency for a bandpass or bandstop design.

Data Types: double

CutoffFrequency, CutoffFrequency1, CutoffFrequency2 — 6-dB frequency
positive scalar

6-dB frequency, specified as the comma-separated pair consisting of 'CutoffFrequency' and a positive scalar. The frequency value must be within the Nyquist range.

'CutoffFrequency1' is the lower 6-dB frequency for a bandpass or bandstop design.

'CutoffFrequency2' is the higher 6-dB frequency for a bandpass or bandstop design.

Data Types: double

HalfPowerFrequency, HalfPowerFrequency1, HalfPowerFrequency2 — 3-dB frequency
positive scalar

3-dB frequency, specified as the comma-separated pair consisting of 'HalfPowerFrequency' and a positive scalar. The frequency value must be within the Nyquist range.

'HalfPowerFrequency1' is the lower 3-dB frequency for a bandpass or bandstop design.

'HalfPowerFrequency2' is the higher 3-dB frequency for a bandpass or bandstop design.

Data Types: double

TransitionWidth — Width of transition region
positive scalar
Width of the transition region between passband and stopband for a Hilbert transformer, specified as the comma-separated pair consisting of 'TransitionWidth' and a positive scalar.

Data Types: double

**Frequencies — Response frequencies**

vector

Response frequencies, specified as the comma-separated pair consisting of 'Frequencies' and a vector. Use this variable to list the frequencies at which a filter of arbitrary magnitude response has desired amplitudes. The frequencies must be monotonically increasing and lie within the Nyquist range. The first element of the vector must be either 0 or \(-f_s/2\), where \(f_s\) is the sample rate, and its last element must be \(f_s/2\). If you do not specify a sample rate, `designfilt` uses the default value of 2 Hz.

Data Types: double

**NumBands — Number of bands**

positive integer scalar

Number of bands in a multiband design, specified as the comma-separated pair consisting of 'NumBands' and a positive integer scalar not greater than 10.

Data Types: double

**BandFrequencies1, ..., BandFrequenciesN — Multiband response frequencies**

vectors

Multiband response frequencies, specified as comma-separated pairs consisting of 'BandFrequencies\(_i\)' and a numeric vector. 'BandFrequencies\(_i\)', where \(i\) runs from 1 through `NumBands`, is a vector containing the frequencies at which the \(i\)th band of a multiband design has the desired values, 'BandAmplitudes\(_i\)'. `NumBands` can be at most 10. The frequencies must lie within the Nyquist range and must be specified in monotonically increasing order. Adjacent frequency bands must have the same amplitude at their junction.

Data Types: double

**Magnitude Constraints**

**PassbandRipple, PassbandRipple1, PassbandRipple2 — Passband ripple**

1 (default) | positive scalar
Passband ripple, specified as the comma-separated pair consisting of 'PassbandRipple' and a positive scalar expressed in decibels.

'PassbandRipple1' is the lower-band passband ripple for a bandstop design.

'PassbandRipple2' is the higher-band passband ripple for a bandstop design.

Data Types: double

StopbandAttenuation, StopbandAttenuation1, StopbandAttenuation2 — Stopband attenuation
60 (default) | positive scalar

Stopband attenuation, specified as the comma-separated pair consisting of 'StopbandAttenuation' and a positive scalar expressed in decibels.

'StopbandAttenuation1' is the lower-band stopband attenuation for a bandpass design.

'StopbandAttenuation2' is the higher-band stopband attenuation for a bandpass design.

Data Types: double

Amplitudes — Desired response amplitudes
vector

Desired response amplitudes of an arbitrary magnitude response filter, specified as the comma-separated pair consisting of 'Amplitudes' and a vector. Express the amplitudes in linear units. The vector must have the same length as 'Frequencies'.

Data Types: double

BandAmplitudes1, ..., BandAmplitudesN — Multiband response amplitudes
vectors

Multiband response amplitudes, specified as comma-separated pairs consisting of 'BandAmplitudesi' and a numeric vector. 'BandAmplitudesi', where i runs from 1 through NumBands, is a vector containing the desired amplitudes in the i\textsuperscript{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 the comma-separated pair consisting of 'DesignMethod' and 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

Window — Window
numeric vector | window name | function handle | cell array

Window, specified as the comma-separated pair consisting of 'Window' and a vector of length \( N + 1 \), where \( N \) is the filter order. 'Window' can also be paired with a 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 the comma-separated pair consisting of 'MatchExactly' and either 'stopband', 'passband', or 'both'. 'both' is available only for the elliptic design method, where it is the default. 'stopband' is the default for the 'butter' and 'cheby2' methods. 'passband' is the default for 'cheby1'.

Data Types: char | string

PassbandOffset — Passband offset
0 (default) | positive scalar
Passband offset, specified as the comma-separated pair consisting of 'PassbandOffset' and a positive scalar expressed in decibels. 'PassbandOffset' specifies the filter gain in the passband.

Example: 'PassbandOffset',0 results in a filter with unit gain in the passband.
Example: 'PassbandOffset',2 results in a filter with a passband gain of 2 dB or 1.259.

Data Types: double

ScalePassband — Scale passband

true (default) | false

Scale passband, specified as the comma-separated pair consisting of 'ScalePassband' and a logical scalar. When you set 'ScalePassband' to true, the passband is scaled, after windowing, so that the filter has unit gain at zero frequency.

Example: 'Window',{@kaiser,0.1},'ScalePassband',true help specify a filter whose magnitude response at zero frequency is exactly 0 dB. This is not the case when you specify 'ScalePassband',false. To verify, visualize the filter with fvtool and zoom in.

Data Types: logical

ZeroPhase — Zero phase

false (default) | true

Zero phase, specified as the comma-separated pair consisting of 'ZeroPhase' and a logical scalar. When you set 'ZeroPhase' to true, the zero-phase response of the resulting filter is always positive. This lets you perform spectral factorization on the result and obtain a minimum-phase filter from it.

Data Types: logical

PassbandWeight, PassbandWeight1, PassbandWeight2 — Passband optimization weight

1 (default) | positive scalar

Passband optimization weight, specified as the comma-separated pair consisting of 'PassbandWeight' and a positive scalar.

'PassbandWeight1' is the lower-band passband optimization weight for a bandstop FIR design.
'PassbandWeight2' is the higher-band passband optimization weight for a bandstop FIR design.

Data Types: double

StopbandWeight, StopbandWeight1, StopbandWeight2 — Stopband optimization weight
1 (default) | positive scalar

Stopband optimization weight, specified as the comma-separated pair consisting of 'StopbandWeight' and a positive scalar.

'StopbandWeight1' is the lower-band stopband optimization weight for a bandpass FIR design.

'StopbandWeight2' is the higher-band stopband optimization weight for a bandpass FIR design.

Data Types: double

Weights — Optimization weights
1 (default) | positive scalar | vector

Optimization weights, specified as the comma-separated pair consisting of 'Weights' and a positive scalar or a vector of the same length as 'Amplitudes'.

Data Types: double

BandWeights1, ..., BandWeightsN — Multiband weights
1 (default) | positive scalar | vectors

Multiband weights, specified as comma-separated pairs consisting of 'BandWeightsi' and a set 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 the comma-separated pair consisting of 'SampleRate' and a positive scalar expressed in hertz. To work with normalized frequencies, set 'SampleRate' to 2, or simply omit it.

Data Types: double

Output Arguments

d — Digital filter
digitalFilter object

Digital filter, returned as a digitalFilter object.

More About

Filter Design Assistant

If you specify an incomplete or inconsistent set of design parameters, designfilt offers to open a Filter Design Assistant.

(In the argument description for resp there is a complete list of valid specification sets for all available response types.)

The assistant behaves differently if you call designfilt at the command line or within a script or function.

Filter Design Assistant at the Command Line

You are given a signal sampled at 2 kHz. You are asked to design a lowpass FIR filter that suppresses frequency components higher than 650 Hz. The “cutoff frequency” sounds like a good candidate for a specification parameter. At the MATLAB command line, you type the following.

```matlab
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 the following code on the command line.

```matlab
dee = designfilt('lowpassfir', 'FilterOrder', 10, ...
               'CutoffFrequency', Fctff, 'SampleRate', Fsamp);
```

Typing the name of the filter reiterates the information from the dialog box.

```matlab
dee
```

```matlab
dee =
digitalFilter with properties:
```
Coefficients: [1x11 double]
Specifications:
  FrequencyResponse: 'lowpass'
  ImpulseResponse: 'fir'
  SampleRate: 2000
  FilterOrder: 10
  CutoffFrequency: 650
  DesignMethod: 'window'
Use fvtool to visualize filter
Use filter function to filter data

You invoke **FVTool** and get a plot of **dee**’s frequency response.

fvtool(dee)

![Filter Visualization Tool](image)

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.

```matlab
dee = designfilt('lowpassfir', 'FilterOrder', 10, ...  
    'CutoffFrequency', Fctff, ...  
    'PassbandRipple', 1, 'StopbandAttenuation', 60, ...  
    'SampleRate', Fsamp);
fvtool(dee)
```

The cutoff still does not look sharp. The attenuation is indeed 60 dB, but for frequencies above 900 Hz.

Again invoke `designfilt` with your filter as input.

```matlab
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.

deep = designfilt('lowpassfir', 'FilterOrder', 50, 'CutoffFrequency', 650, 'PassbandRipple', 1, 'StopbandAttenuation', 60, 'SampleRate', 2000);

dee = designfilt('lowpassfir', 'FilterOrder', 50, ...
    'CutoffFrequency', 650, ...
    'PassbandRipple', 1, 'StopbandAttenuation', 60, ...
    'SampleRate', 2000);

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.

```matlab
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.

```matlab
function dataOut = hipassfilt(N,dataIn)
    hpFilter = designfilt('highpassiir','FilterOrder',N);
    dataOut = filter(hpFilter,dataIn);
end
```

To test your function, create a signal composed of two sinusoids with frequencies 500 and 800 Hz and generate samples for 0.1 s. A 5th-order filter seems reasonable as an initial guess. Create a script called `driveHPfilt.m`.

```matlab
% script driveHPfilt.m
Fsamp = 2e3;
Fsm = 500;
Fbg = 800;
t = 0:1/Fsamp:0.1;
sgin = sin(2*pi*Fsm*t)+sin(2*pi*Fbg*t);
Order = 5;
sgout = hipassfilt(Order,sgin);
```
When you run the script at the command line, you get an error message.

```
>> driveHPfilt
Error using designfilt (line 457)

Click here to launch an assistant that can correct your code.

You have specified too few parameters for 'highpassiir'.
The following are valid parameter sets that are close to your inputs:
  - FilterOrder, HalfPowerFrequency
  - FilterOrder, PassbandFrequency, PassbandRipple
  - FilterOrder, PassbandFrequency, StopbandAttenuation, PassbandRipple
  - FilterOrder, StopbandFrequency, StopbandAttenuation

Error in hipassfilt (line 2)
hpfilter = designfilt('highpassiir','FilterOrder',N);
Error in driveHPfilt (line 7)
sout = hipassfilt(Order,sgin);
```

The error message gives you the choice of opening an assistant to correct the MATLAB code. Click to get the Filter Design Assistant on your screen.

You see the problem: You did not specify the frequency constraint. You also forgot to set a sample rate. After experimenting, you find that you can specify Frequency units as Hz,
Passband frequency equal to 700 Hz, and Input Fs equal to 2000 Hz. The Design method changes from Butterworth to Chebyshev type I. You click OK and get the following.

The assistant has correctly identified the file where you call designfilt. Click Yes to accept the change. The function has the corrected MATLAB code.

```matlab
function dataOut = hipassfilt(N,dataIn)
    % hpFilter = designfilt('highpassiir','FilterOrder',N);
    hpFilter = designfilt('highpassiir', 'FilterOrder', N, ...
                           'PassbandFrequency', 700, 'PassbandRipple', 1, ...
                           'SampleRate', 2000);
    dataOut = filter(hpFilter,dataIn);
end
```

You can now run the script with different values of the filter order. Depending on your design constraints, you can change your specification set.

**Filter Design Assistant Preferences**

You can set designfilt to never offer the Filter Design Assistant. This action sets a MATLAB preference that can be unset with setpref:

- Use `setpref('dontshowmeagain','filterDesignAssistant',false)` to be offered the assistant every time. With this command, you can get the assistant again after having disabled it.
- Use `setpref('dontshowmeagain','filterDesignAssistant',true)` to disable the assistant permanently. You can also click Do not show this message again in the initial dialog box.

You can set designfilt to always correct faulty specifications without asking. This action sets a MATLAB preference that can be unset by using setpref:

- Use `setpref('dontshowmeagain','filterDesignAssistantCodeCorrection',false)`
lse) to have designfilt correct your MATLAB code without asking for confirmation. You can also click **Always accept** in the confirmation dialog box.

- Use
  setpref('dontshowmeagain','filterDesignAssistantCodeCorrection',true) to ensure that designfilt corrects your MATLAB code only when you confirm you want the changes. With this command, you can undo the effect of having clicked **Always accept** in the confirmation dialog box.

**Troubleshooting**

There are some instances in which, given an invalid set of specifications, designfilt does not offer a Filter Design Assistant, either through a dialog box or through a link in an error message.

- You are not offered an assistant if you use code-section evaluation, either from the MATLAB Toolstrip or by pressing **Ctrl+Enter**. (See “Divide Your File into Code Sections” (MATLAB) 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” (MATLAB) 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.

See Also
FVTool | digitalFilter | double | fftfilt | filt2block | filter | filtfilt | filtord | firtype | freqz | grpdelay | impz | impzlength | info | isallpass | isdouble | isfird | 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
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.

<table>
<thead>
<tr>
<th>dfilt.structure</th>
<th>Description</th>
<th>Coefficient Mapping Support in realizemdl</th>
<th>Target Filter Block for block Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfilt.delay</td>
<td>Delay</td>
<td>Not supported</td>
<td>Delay</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Requires DSP System Toolbox</td>
</tr>
<tr>
<td>dfilt.df1</td>
<td>Direct-form I</td>
<td>Supported</td>
<td>Discrete Filter</td>
</tr>
<tr>
<td>dfilt.structure</td>
<td>Description</td>
<td>Coefficient Mapping Support in realizemdl</td>
<td>Target Filter Block for block Method</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>------------------------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>dfilt.df1sos</td>
<td>Direct-form I, second-order sections</td>
<td>Supported</td>
<td>Discrete Filter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Requires DSP System Toolbox</td>
</tr>
<tr>
<td>dfilt.df1t</td>
<td>Direct-form I transposed</td>
<td>Supported</td>
<td>Discrete Filter</td>
</tr>
<tr>
<td>dfilt.df1tsos</td>
<td>Direct-form I transposed, second-order sections</td>
<td>Supported</td>
<td>Biquad Filter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Requires DSP System Toolbox</td>
</tr>
<tr>
<td>dfilt.df2</td>
<td>Direct-form II</td>
<td>Supported</td>
<td>Discrete Filter</td>
</tr>
<tr>
<td>dfilt.df2sos</td>
<td>Direct-form II, second-order sections</td>
<td>Supported</td>
<td>Discrete Filter</td>
</tr>
<tr>
<td>dfilt.df2t</td>
<td>Direct-form II transposed</td>
<td>Supported</td>
<td>Discrete Filter</td>
</tr>
<tr>
<td>dfilt.df2tsos</td>
<td>Direct-form II transposed, second-order sections</td>
<td>Supported</td>
<td>Biquad Filter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Requires DSP System Toolbox</td>
</tr>
<tr>
<td>dfilt.dffir</td>
<td>Direct-form FIR</td>
<td>Supported</td>
<td>Discrete FIR Filter</td>
</tr>
<tr>
<td>dfilt.dffirt</td>
<td>Direct-form FIR transposed</td>
<td>Supported</td>
<td>Discrete FIR Filter</td>
</tr>
<tr>
<td>dfilt.dfsymfir</td>
<td>Direct-form symmetric FIR</td>
<td>Supported</td>
<td>Discrete FIR Filter</td>
</tr>
<tr>
<td>dfilt.dfasymfir</td>
<td>Direct-form antisymmetric FIR</td>
<td>Supported</td>
<td>Discrete FIR Filter</td>
</tr>
<tr>
<td>dfilt.fft fir</td>
<td>Overlap-add FIR</td>
<td>Not supported</td>
<td>Overlap-Add FFT Filter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Requires DSP System Toolbox</td>
</tr>
<tr>
<td>dfilt.latticeallpass</td>
<td>Lattice allpass</td>
<td>Supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>dfilt.structure</td>
<td>Description</td>
<td>Coefficient Mapping Support in realizemdl</td>
<td>Target Filter Block for block Method</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------</td>
<td>------------------------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>dfilt.latticear</td>
<td>Lattice autoregressive (AR)</td>
<td>Supported</td>
<td>Allpole Filter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Requires DSP System Toolbox</td>
</tr>
<tr>
<td>dfilt.latticearma</td>
<td>Lattice autoregressive moving-average (ARMA)</td>
<td>Supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>dfilt.latticearma</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfilt.latticemamax</td>
<td>Lattice moving-average (MA) for maximum phase</td>
<td>Supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>dfilt.latticemamin</td>
<td>Lattice moving-average (MA) for minimum phase</td>
<td>Supported</td>
<td>Discrete FIR Filter</td>
</tr>
<tr>
<td>dfilt.statespace</td>
<td>State-space</td>
<td>Supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>dfilt.scalar</td>
<td>Scalar gain object</td>
<td>Supported</td>
<td>Gain</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Requires DSP System Toolbox</td>
</tr>
<tr>
<td>dfilt.cascade</td>
<td>Filters arranged in series</td>
<td>Supported</td>
<td>Target blocks depend on filter structures in the series</td>
</tr>
<tr>
<td>dfilt.parallel</td>
<td>Filters arranged in parallel</td>
<td>Supported</td>
<td>Target blocks depend on filter structures in the parallel system</td>
</tr>
</tbody>
</table>

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 \( h = \text{freqz}(Hd) \). You can use all of the methods below in this way.

**Note** If your variable is a 1-D array of \texttt{dfilt} filters, the method is applied to each object in the array. Only \texttt{freqz}, \texttt{grpdelay}, \texttt{impz}, \texttt{is*}, \texttt{order}, and \texttt{stepz} methods can be applied to arrays. The \texttt{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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addstage</td>
<td>Adds a stage to a cascade or parallel object, where a stage is a separate, modular filter. See \texttt{dfilt.cascade} and \texttt{dfilt.parallel}.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>block</td>
<td>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 1-384 for a mapping between the target blocks and filter structures. The block method can specify these properties/values:</td>
</tr>
<tr>
<td></td>
<td>'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.</td>
</tr>
<tr>
<td></td>
<td>'InputProcessing' specifies sample-based, 'elementsaschannels', frame-based, 'columnaschannels', 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.</td>
</tr>
<tr>
<td>cascade</td>
<td>Returns the series combination of two dfilt objects. See dfilt.cascade.</td>
</tr>
<tr>
<td>coeffs</td>
<td>Returns the filter coefficients in a structure containing fields that use the same property names as those in the original dfilt.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>convert</td>
<td>Converts a dfilt object from one filter structure to another filter structure.</td>
</tr>
<tr>
<td>fcfwrite</td>
<td>Writes a filter coefficient ASCII file. The file can contain a single filter or a vector of objects. Default file name is untitled.fcf.</td>
</tr>
</tbody>
</table>

```matlab
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.                                                     |

```matlab
y = filter(Hd,x) filters x using the Hd filter and returns the filtered data in y. See “Using Filter States” on page 1-395 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.

```matlab
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. |
<p>| 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.                                                              |</p>
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>impz</td>
<td>Plots the impulse response in FVTool.</td>
</tr>
<tr>
<td>impzlength</td>
<td>Returns the length of the impulse response.</td>
</tr>
<tr>
<td>info</td>
<td>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.</td>
</tr>
<tr>
<td>isallpass</td>
<td>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.</td>
</tr>
<tr>
<td>iscascade</td>
<td>Returns a logical 1 if the dfilt object is cascaded or a logical 0 if it is not.</td>
</tr>
<tr>
<td>isfir</td>
<td>Returns a logical 1 if the dfilt object has finite impulse response (FIR) or a logical 0 if it does not.</td>
</tr>
<tr>
<td>islinphase</td>
<td>Returns a logical 1 if the dfilt object is linear phase or a logical 0 if it is not.</td>
</tr>
<tr>
<td>ismaxphase</td>
<td>Returns a logical 1 if the dfilt object is maximum-phase or a logical 0 if it is not.</td>
</tr>
<tr>
<td>isminphase</td>
<td>Returns a logical 1 if the dfilt object is minimum-phase or a logical 0 if it is not.</td>
</tr>
<tr>
<td>isparallel</td>
<td>Returns a logical 1 if the dfilt object has parallel stages or a logical 0 if it does not.</td>
</tr>
<tr>
<td>isreal</td>
<td>Returns a logical 1 if the dfilt object has real-valued coefficients or a logical 0 if it does not.</td>
</tr>
<tr>
<td>isscalar</td>
<td>Returns a logical 1 if the dfilt object is a scalar or a logical 0 if it is not scalar.</td>
</tr>
<tr>
<td>issos</td>
<td>Returns a logical 1 if the dfilt object has second-order sections or a logical 0 if it does not.</td>
</tr>
<tr>
<td>isstable</td>
<td>Returns a logical 1 if the dfilt object is stable or a logical 0 if it are not.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>nsections</td>
<td>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).</td>
</tr>
<tr>
<td>nstages</td>
<td>Returns the number of stages of the filter, where a stage is a separate, modular filter.</td>
</tr>
<tr>
<td>nstates</td>
<td>Returns the number of states for an object.</td>
</tr>
<tr>
<td>order</td>
<td>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.</td>
</tr>
<tr>
<td>parallel</td>
<td>Returns the parallel combination of two dfilt filters. See dfilt.parallel.</td>
</tr>
<tr>
<td>phasez</td>
<td>Plots the phase response in FVTool.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>realizemdl</td>
<td>(Available only with Simulink software.)</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'OptimizeDelayChains'</td>
<td>'OptimizeDelayChains' replaces cascaded chains of delay block with a single integer delay block set to the appropriate delay.</td>
</tr>
<tr>
<td>removestage</td>
<td>Removes a stage from a cascade or parallel dfilt. See dfilt.cascade and dfilt.parallel.</td>
</tr>
<tr>
<td>setstage</td>
<td>Overwrites a stage of a cascade or parallel dfilt. See dfilt.cascade and dfilt.parallel.</td>
</tr>
<tr>
<td>sos</td>
<td>Converts the dfilt to a second-order sections dfilt. If Hd has a single section, the returned filter has the same class.</td>
</tr>
<tr>
<td>sos(Hd,flag)</td>
<td>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.</td>
</tr>
<tr>
<td>sos(Hd,flag,scale)</td>
<td>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.</td>
</tr>
<tr>
<td>ss</td>
<td>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).</td>
</tr>
<tr>
<td>stepz</td>
<td>Plots the step response in FVTool. stepz(Hd,n) computes the first n samples of the step response.</td>
</tr>
<tr>
<td>stepz(Hd,n,Fs)</td>
<td>stepz(Hd,n,Fs) separates the time samples by T = 1/Fs, where Fs is assumed to be in Hz.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>sysobj</td>
<td>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.</td>
</tr>
<tr>
<td>tf</td>
<td>Converts the dfilt to a transfer function.</td>
</tr>
<tr>
<td>zerophase</td>
<td>Plots the zero-phase response in FVTool.</td>
</tr>
<tr>
<td>zpk</td>
<td>Converts the dfilt to zeros-pole-gain form.</td>
</tr>
<tr>
<td>zplane</td>
<td>Plots a pole-zero plot in FVTool.</td>
</tr>
</tbody>
</table>

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

```matlab
get(Hd,'property')
```

To see all properties for an object, use

```matlab
get(Hd)
```

**Changing Properties**

To set specific properties, use

```matlab
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`:

```matlab
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.

\[ H2 = \text{copy}(Hd) \]

**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

\[ Hd2 = \text{convert}(Hd,'structure_name'); \]

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.

\[ [b,a] = \text{butter}(8,0.25); \]
\[ \text{Hd} = \text{dfilt.df1}(b,a) \]

If a \text{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

\[ \text{get(Hd,'numerator')} \]

or alternatively

\[ \text{Hd.numerator} \]

Refer to the reference pages for each structure for more examples.

See Also

Apps
Filter Designer | Signal Analyzer

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
  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.

```matlab
Hs = Hd.states       % Where Hd is the dfilt.df1 object and
double (Hs)          % Hs is the filtstates object
```

The vector is

```
[zb(1)
zb(2)
...zbn]
zan(1)
zan(2)
...zann]
```
Examples

Create a direct-form I discrete-time filter with coefficients from a fourth-order lowpass Butterworth design

\[
[b,a] = \text{butter}(4,.5);
\]
\[
Hd = \text{dfilt.df1}(b,a)
\]

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.

\[
Hs = \text{Hd.states} \quad \% \text{Where Hd is the dfilt.df1 object and double (Hs) \quad \% Hs is the filtstates object}
\]

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.

\[
\text{Hs} = \text{Hd.states} \quad \% \text{Where Hd is the dfilt.df1 object and double (Hs) } \% \text{Hs is the filtstates object}
\]

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:

\[
\begin{align*}
b &= [0.05 \ 0.9 \ 0.05]; \\
\text{Hd} &= \text{dfilt.df1t}(b,1); \\
\text{Hd.States} & \% \text{Returns} \\
& \% \text{Numerator: [2x1 double]}
\end{align*}
\]
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:

```matlab
[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:

\[
[z,p,k] = \text{ellip}(6,1,60,.4); \quad \% \text{Obtain filter coefficients}
\]
\[
[s,g] = \text{zp2sos}(z,p,k); \quad \% \text{Convert to SOS}
\]
\[
\text{Hd} = \text{dfilt.df1tsos}(s,g)
\]
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:

```matlab
[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:

```
dfilt.df2sos
```
[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:

```matlab
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:

```matlab
[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**

```matlab
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
\[
\begin{pmatrix}
z_1(1) & z_2(1) \\
z_1(2) & z_2(2)
\end{pmatrix}
\]

**Examples**

**Elliptic Filter as Second-Order Sections**

Design a second-order sections, direct-form II, transposed discrete-time filter starting from a 6th-order lowpass elliptic filter. Specify a passband edge frequency of 0.4\(\pi\) rad/sample, a passband ripple of 1 dB, and a stopband attenuation of 60 dB. Visualize the filter response.

\[
[z,p,k] = \text{ellip}(6,1,60,0.4); \quad %\text{Obtain filter coefficients}
\]
\[
[s,g] = \text{zp2sos}(z,p,k); \quad %\text{Convert to SOS}
\]
\[
\text{Hd} = \text{dfilt.df2tsos}(s,g);
\]
\[
\text{fvtool(Hd)}
\]
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
\[ b[(end+1)/2] = 0 \]
**dfasymfir**

(Antisymmetric FIR)

Even number of coefficients, length(b) = 6

The resulting filter states column vector for the odd number of coefficients example above is

\[
\begin{bmatrix}
z(1) \\
z(2) \\
z(3) \\
z(4) \\
z(5) \\
z(6)
\end{bmatrix}
\]

\[b(i) = -b(\text{end} - i + 1)\]
Examples

Odd-Order Antisymmetric FIR Filter Structure

Create a Type-4 25th-order highpass direct-form antisymmetric FIR filter structure for a dfilt object.

```matlab
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:

```matlab
impz(Hd)
```
**Even-Order Antisymmetric FIR Filter Structure**

Create a 44th-order lowpass direct-form antisymmetric FIR differentiator filter structure for a dfilt object.

```matlab
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.

```matlab
impz(Hd)
```
See Also

Signal Analyzer | designfilt

Introduced before R2006a
**dfilt.dffir**

Discrete-time, direct-form, FIR filter

**Syntax**

\[ \text{Hd} = \text{dfilt.dffir}(b) \]
\[ \text{Hd} = \text{dfilt.dffir} \]

**Description**

\[ \text{Hd} = \text{dfilt.dffir}(b) \] returns a discrete-time, direct-form finite impulse response (FIR) filter, \( \text{Hd} \), with numerator coefficients, \( b \).

\[ \text{Hd} = \text{dfilt.dffir} \] returns a default, discrete-time, direct-form FIR filter, \( \text{Hd} \), with \( b=1 \). This filter passes the input through to the output unchanged.

**dfir**

*(Direct-form FIR = Tapped delay line)*

![Diagram of dfir filter](image-url)
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:

```matlab
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 30\textsuperscript{th} order lowpass equiripple design:

```matlab
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).
dfsymfir
(Symmetric FIR)
Even order
Odd number of coefficients, length(b) = 5
b(i) == b(end - i + 1)
The resulting filter states column vector for the odd number of coefficients example above is

\[
\begin{bmatrix}
z(1) \\
z(2) \\
z(3) \\
z(4)
\end{bmatrix}
\]
Examples

Odd-Order Symmetric FIR Filter Structure

Create a Type-2 15th-order direct-form symmetric FIR filter structure for a dfilt object.

```matlab
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.

```matlab
Num_coeffs = fir1(16,0.5);
Hd = dfilt.dfsymfir(Num_coeffs);
```

Display the impulse response of the filter.

```matlab
impz(Hd)
```
See Also
Signal Analyzer | designfilt

Introduced before R2006a
**dfilt.fftfir**

Discrete-time, overlap-add, FIR filter

**Syntax**

```plaintext
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.

\[
Hd = dfilt.fftfir(b,len) \text{ returns a discrete-time, FFT, FIR filter, } Hd, \text{ with numerator coefficients, } b \text{ and block length, } len. \text{ The block length is the number of input points to use for each overlap-add computation.}
\]

\[
Hd = dfilt.fftfir(b) \text{ returns a discrete-time, FFT, FIR filter, } Hd, \text{ with numerator coefficients, } b \text{ and block length, } len=100.
\]

\[
Hd = dfilt.fftfir \text{ returns a default, discrete-time, FFT, FIR filter, } Hd, \text{ with the numerator } b=1 \text{ and block length, } len=100. \text{ This filter passes the input through to the output unchanged.}
\]

**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,

![Diagram](image)

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:

```matlab
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.

```matlab
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

\[
\begin{bmatrix}
    z(1) \\
    z(2)
\end{bmatrix}
\]
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:

\[ k = [.66 \ 0.7 \ 0.44] \]
\[ Hd = dfilt.latticear(k) \]

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

![Diagram of latticearma filter](image-url)
Examples

Form a third-order lattice autoregressive, moving-average filter structure for a dfilt object, Hd, using the following lattice coefficients:

\[
\begin{bmatrix}
  z(1) \\
  z(2) \\
  z(3)
\end{bmatrix}
\]

\[k = [.66 .7 .44];\]
\[Hd = dfilt.latticearma(k)\]

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:

\[
k = [0.66 
  0.7 
  0.44 
  0.33];
\]

\[
Hd = dfilt.latticemamax(k)
\]

*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, \(H_d\), 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.

\[ k1 = [-0.0154 \quad 0.9846 \quad -0.3048 \quad 0.5601]; \]
\[ Hd1 = dfilt.latticeallpass(k1); \]
\[ k2 = [-0.1294 \quad 0.8341 \quad -0.4165]; \]
\[ Hd2 = dfilt.latticeallpass(k2); \]
\[ Hpar = parallel(Hd1,Hd2); \]
\[ gain = dfilt.scalar(0.5); \quad \% \text{Normalize passband gain} \]
\[ Hcas = cascade(gain,Hpar); \]

For details on the stages of this filter, use

\[ \text{info(Hcas.Stage(1))} \]

and

\[ \text{info(Hcas.Stage(2))} \]

To view this filter, use

\[ \text{fvtool(Hcas)} \]
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

```matlab
Hd.stage(1)
```

and

```matlab
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 \( A=\begin{bmatrix} \end{bmatrix} \), \( B=\begin{bmatrix} \end{bmatrix} \), \( C=\begin{bmatrix} \end{bmatrix} \), 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.

```matlab
[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.

```matlab
x = 1:256;
y1 = fft(x);
n = length(x);
y2 = x*dftmtx(n);
norm(y1-y2)
ans = 1.8305e-09
```
Input Arguments

\( n \) — Discrete Fourier transform length  
positive integer

Discrete Fourier transform length, specified as an integer.
Data Types: \textit{single} | \textit{double}

Output Arguments

\( a \) — Discrete Fourier transform matrix  
matrix

Discrete Fourier transform matrix, returned as a matrix.

More About

Discrete Fourier Transform Matrix

A \textit{discrete Fourier transform matrix} is a complex matrix whose matrix product with a vector computes the discrete Fourier transform of the vector. \texttt{dftmtx} takes the FFT of the identity matrix to generate the transform matrix.

For a column vector \( x \),
\[
y = \text{dftmtx}(n) * x
\]
is the same as \( y = \text{fft}(x,n) \). The inverse discrete Fourier transform matrix is
\[
a_{\text{inv}} = \text{conj(} \text{dftmtx}(n) \text{)}/n
\]
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fftfilt</code></td>
<td>Filters a signal with a <code>digitalFilter</code> using an FFT-based overlap-add method</td>
</tr>
<tr>
<td><code>filter</code></td>
<td>Filters a signal using a <code>digitalFilter</code></td>
</tr>
<tr>
<td><code>filtfilt</code></td>
<td>Performs zero-phase filtering of a signal with a <code>digitalFilter</code></td>
</tr>
</tbody>
</table>
## Filter Analysis

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>Casts the coefficients of a <code>digitalFilter</code> to double precision</td>
</tr>
<tr>
<td>filt2block</td>
<td>Generates a Simulink filter block corresponding to a <code>digitalFilter</code></td>
</tr>
<tr>
<td>filtord</td>
<td>Returns the filter order of a <code>digitalFilter</code></td>
</tr>
<tr>
<td>firtype</td>
<td>Returns the type (1, 2, 3, or 4) of an FIR <code>digitalFilter</code></td>
</tr>
<tr>
<td>freqz</td>
<td>Returns or plots the frequency response of a <code>digitalFilter</code></td>
</tr>
<tr>
<td>fvtool</td>
<td>Opens the Filter Visualization Tool and displays the magnitude response of a <code>digitalFilter</code></td>
</tr>
<tr>
<td>grpdelay</td>
<td>Returns or plots the group delay response of a <code>digitalFilter</code></td>
</tr>
<tr>
<td>impz</td>
<td>Returns or plots the impulse response of a <code>digitalFilter</code></td>
</tr>
<tr>
<td>impzlength</td>
<td>Returns the length of the impulse response of a <code>digitalFilter</code>, whether actual (for FIR filters) or effective (for IIR filters)</td>
</tr>
<tr>
<td>info</td>
<td>Returns a character array with information about a <code>digitalFilter</code></td>
</tr>
<tr>
<td>isallpass</td>
<td>Returns true if a <code>digitalFilter</code> is allpass</td>
</tr>
<tr>
<td>isdouble</td>
<td>Returns true if the coefficients of a <code>digitalFilter</code> are double precision</td>
</tr>
<tr>
<td>isfir</td>
<td>Returns true if a <code>digitalFilter</code> has a finite impulse response</td>
</tr>
<tr>
<td>islinphase</td>
<td>Returns true if a <code>digitalFilter</code> has linear phase</td>
</tr>
<tr>
<td>ismaxphase</td>
<td>Returns true if a <code>digitalFilter</code> is maximum phase</td>
</tr>
<tr>
<td>isminphase</td>
<td>Returns true if a <code>digitalFilter</code> is minimum phase</td>
</tr>
<tr>
<td>issingle</td>
<td>Returns true if the coefficients of a <code>digitalFilter</code> are single precision</td>
</tr>
<tr>
<td>isstable</td>
<td>Returns true if a <code>digitalFilter</code> is stable</td>
</tr>
<tr>
<td>phasedelay</td>
<td>Returns or plots the phase delay response of a <code>digitalFilter</code></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>phasez</td>
<td>Returns or plots the (unwrapped) phase response of a digitalFilter</td>
</tr>
<tr>
<td>single</td>
<td>Casts the coefficients of a digitalFilter to single precision</td>
</tr>
<tr>
<td>ss</td>
<td>Returns the state-space representation of a digitalFilter</td>
</tr>
<tr>
<td>stepz</td>
<td>Returns or plots the step response of a digitalFilter</td>
</tr>
<tr>
<td>tf</td>
<td>Returns the transfer function representation of a digitalFilter</td>
</tr>
<tr>
<td>zerophase</td>
<td>Returns or plots the zero-phase response of a digitalFilter</td>
</tr>
<tr>
<td>zpk</td>
<td>Returns the zero-pole-gain representation of a digitalFilter</td>
</tr>
<tr>
<td>zplane</td>
<td>Displays the poles and zeros of the transfer function represented by a digitalFilter</td>
</tr>
</tbody>
</table>

## 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.

```matlab
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.

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

Output the filter coefficients, expressed as second-order sections.

```matlab
sos = lpFilt.Coefficients

sos = 4x6

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
```
See Also

FVTool | designfilt | double | fftfilt | filt2block | filter | filtfilt | filtord | firtype | freqz | 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

\[ y = \text{digitrevorder}(x,r) \]
\[ [y,i] = \text{digitrevorder}(x,r) \]

Description

digitrevorder is useful for pre-ordering a vector of filter coefficients for use in frequency-domain filtering algorithms, in which the \texttt{fft} and \texttt{ifft} transforms are computed without digit-reversed ordering for improved run-time efficiency.

\[ y = \text{digitrevorder}(x,r) \] returns the input data in digit-reversed order in vector or matrix \( y \). The digit-reversal is computed using the number system base (radix base) \( r \), which can be any integer from 2 to 36. The length of \( x \) must be an integer power of \( r \). If \( x \) is a matrix, the digit reversal occurs on the first dimension of \( x \) with size greater than 1. \( y \) is the same size as \( x \).

\[ [y,i] = \text{digitrevorder}(x,r) \] returns the digit-reversed vector or matrix \( y \) and the digit-reversed indices \( i \), such that \( y = x(i) \). Recall that MATLAB matrices use 1-based indexing, so the first index of \( y \) will be 1, not 0.

The following table shows the numbers 0 through 15, the corresponding digits and the digit-reversed numbers using radix base-4. The corresponding radix base-2 bits and bit-reversed indices are also shown.

<table>
<thead>
<tr>
<th>Linear Index</th>
<th>Base-4 Digits</th>
<th>Digit-Reversed</th>
<th>Digit-Reversed Index</th>
<th>Base-2 Bits</th>
<th>Base-2 Reversed (bitrevorder)</th>
<th>Bit-Reversed Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>0</td>
<td>0000</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>10</td>
<td>4</td>
<td>0001</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>20</td>
<td>8</td>
<td>0010</td>
<td>0100</td>
<td>4</td>
</tr>
</tbody>
</table>
### 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.

```matlab
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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>01</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>02</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>10</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>11</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>12</td>
<td>21</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>20</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>21</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>22</td>
<td>22</td>
<td>8</td>
</tr>
</tbody>
</table>

**See Also**

bitrevorder | fft | ifft

**Introduced before R2006a**
diric

Dirichlet or periodic sinc function

Syntax

\[ y = \text{diric}(x, n) \]

Description

\[ y = \text{diric}(x, n) \] returns the “Dirichlet Function“ on page 1-476 of degree \( n \) evaluated at the elements of the input array \( x \).

Examples

**Dirichlet Function**

Compute and plot the Dirichlet function between \(-2\pi\) and \(2\pi\) for \( N = 7 \) and \( N = 8 \). The function has a period of \(2\pi\) for odd \( N \) and \(4\pi\) for even \( N \).

\[
x = \text{linspace}(-2\pi, 2\pi, 301);
\]

\[
d7 = \text{diric}(x, 7);
d8 = \text{diric}(x, 8);
\]

\[
\text{subplot}(2,1,1)
\text{plot}(x/pi,d7)
\text{ylabel('N = 7')}
\text{title('Dirichlet Function')}
\]

\[
\text{subplot}(2,1,2)
\text{plot}(x/pi,d8)
\text{ylabel('N = 8')}
xlabel('x / \pi')
\]
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.

```matlab
xmax = 4;
x = linspace(-xmax,xmax,1001)';
N = 6;
```

```matlab
diric
```
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(6x/2) / sinc(x/2)')

Repeat the calculation for N = 13.
\[ N = 13; \]
\[ yd = \text{diric}(x\pi,N); \]
\[ ys = \text{sinc}(Nx/2)/\text{sinc}(x/2); \]
\[ ys(-\text{mod}(x,2)) = (-1)^{x(-\text{mod}(x,2))/2(N-1)}; \]

```matlab
subplot(2,1,1)
plot(x,yd)
title('D_{13}(x\pi)')
subplot(2,1,2)
plot(x,ys)
title('sinc(13x/2) / sinc(x/2)')
```
Input Arguments

\( x \) — Input array
real scalar | real vector | real matrix | real multidimensional array

Input array, specified as a real scalar, vector, matrix, or multidimensional array. When \( x \) is nonscalar, \texttt{diric} is an element-wise operation.

Data Types: \texttt{double} | \texttt{single}

\( n \) — Function degree
positive integer scalar

Function degree, specified as a positive integer scalar.

Data Types: \texttt{double} | \texttt{single}

Output Arguments

\( y \) — Output array
real scalar | real vector | real matrix | real multidimensional array

Output array, returned as a real-valued scalar, vector, matrix, or multidimensional array of the same size as \( x \).

More About

Dirichlet Function

The Dirichlet function, or periodic sinc function, is

\[
D_N(x) = \begin{cases} 
\frac{\sin(Nx/2)}{N\sin(x/2)} & x \neq 2nk, \quad k = 0, \pm 1, \pm 2, \pm 3, \ldots \\
(-1)^{k(N-1)} & x = 2nk, \quad k = 0, \pm 1, \pm 2, \pm 3, \ldots 
\end{cases}
\]

for any nonzero integer \( N \).

This function has period \( 2\pi \) for odd \( N \) and period \( 4\pi \) 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*
**double**

Cast coefficients of digital filter to double precision

**Syntax**

\[ f2 = \text{double}(f1) \]

**Description**

\[ f2 = \text{double}(f1) \] casts coefficients in a digital filter, \( f1 \), to double precision and returns a new digital filter, \( f2 \), that contains these coefficients.

**Examples**

**Lowpass FIR Filter in Single and Double Precision**

Use `designfilt` to design a 5th-order FIR lowpass filter. Specify a normalized passband frequency of \( 0.2\pi \) rad/sample and a normalized stopband frequency of \( 0.55\pi \) rad/sample.

Cast the filter to single precision and cast it back to double precision. Display the first coefficient of each filter.

```plaintext
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

coe = e.Coefficients(1)
```
coee = single
    0.0039479

coeff = f.Coefficients(1)
coeff =
    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

\[ y = \text{downsample}(x,n) \]
\[ y = \text{downsample}(x,n,\text{phase}) \]

Description

\( y = \text{downsample}(x,n) \) 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.

\( y = \text{downsample}(x,n,\text{phase}) \) 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.

\[ x = [1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10]; \]
\[ y = \text{downsample}(x,3) \]
\[ y = 1\times4 \]
\[ 1 \ 4 \ 7 \ 10 \]

Decrease the sample rate of the sequence by a factor of 3 and add a phase offset of 2.

\[ y = \text{downsample}(x,3,2) \]
Decrease the sample rate of a matrix by a factor of 3.

\[
x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix};
\]

\[
y = \text{downsample}(x,3)
\]

\[
y = \begin{bmatrix} 1 & 2 & 3 \\ 10 & 11 & 12 \end{bmatrix}
\]

**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: \( \cos(\pi/4 \times (0:159)) + \text{randn}(1,160) \) specifies a sinusoid embedded in white Gaussian noise.

  Example: \( \cos(\pi ./ [4;2] \times (0:159))' + \text{randn}(160,2) \) specifies a two-channel noisy sinusoid.

- **n** — Downsampling factor
  
  positive integer

  Downsampling factor, specified as a positive integer.

  Data Types: single | double

- **phase** — Offset
  
  \( \theta \) (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™.

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`.

\[
\text{...} = \text{dpss}(\ldots, \text{Ni}) \text{ interpolates from DPSSs of length Ni in the database dpss.mat.}
\]

\[
\text{...} = \text{dpss}(\ldots, '\text{trace}') \text{ prints the method used to compute the DPSSs in the command window. Possible methods include: direct, spline interpolation, and linear interpolation.}
\]

**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.

```matlab
seq_length = 512;
time_halfbandwidth = 2.5;
um_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 $2W$. Both the sequence length and bandwidth affect how many Slepian sequences have concentration ratios near one. As a rule, there are $2NW - 1$ Slepian sequences with energy concentration ratios approximately equal to one. Beyond $2NW - 1$ Slepian sequences, the concentration ratios begin to approach zero. Common choices for the time half bandwidth product are: 2.5, 3, 3.5, and 4.

You can specify the bandwidth of the Slepian sequences in Hz by defining the time half bandwidth product as $NW/F_s$, where $F_s$ is the sample rate.
References


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**

```matlab
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 an `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 $Fs$ 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 $2W$. Both the sequence length and bandwidth affect how many Slepian sequences have concentration ratios near one. As a rule, there
are $2NW - 1$ Slepian sequences with energy concentration ratios approximately equal to one. Beyond $2NW - 1$ Slepian sequences, the concentration ratios begin to approach zero. Common choices for the time half bandwidth product are: 2.5, 3, 3.5, and 4.

You can specify the bandwidth of the Slepian sequences in Hz by defining the time half bandwidth product as $NW/F_s$, where $F_s$ is the sample rate.

**References**


**See Also**

dpss | dpssclear | dpssdir | dpssload

**Introduced before R2006a**
**dspdata**

DSP data parameter information

**Syntax**

\[ Hs = \text{dspdata.} dataobj(input1,\ldots) \]

**Description**

**Note** The use of `dspdata.dataobj` is not recommended. Use the appropriate function interface instead.

\[ Hs = \text{dspdata.} dataobj(input1,\ldots) \]

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.

<table>
<thead>
<tr>
<th><code>dspdata.dataobj</code></th>
<th>Description</th>
<th>Corresponding Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dspdata.msspectrum</code></td>
<td>Mean-square spectrum data (power)</td>
<td><code>periodogram</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>pwelch</code></td>
</tr>
<tr>
<td>dspdata.dataobj</td>
<td>Description</td>
<td>Corresponding Functions</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>dspdata.psd</td>
<td>Power spectral density data</td>
<td>pburg</td>
</tr>
<tr>
<td></td>
<td>(power/frequency)</td>
<td>pcov</td>
</tr>
<tr>
<td></td>
<td></td>
<td>periodogram</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pmcov</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pmtm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pwelch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pyulear</td>
</tr>
<tr>
<td>dspdata.pseudospectrum</td>
<td>Pseudospectrum data (power)</td>
<td>peig</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pmusic</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| avgpower   | This method applies only to dspdata.psd objects.  

\[
\text{avgpower}(Hs) \text{ computes the average power of a signal, } Hs, \text{ 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] \text{ if the number of frequency points is even and } [0,\pi) \text{ if it is odd. For a two-sided spectrum, the range is } [0,2\pi). \\

\text{avgpower}(Hs,freqrange) \text{ 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, \text{ the next closest value is used. The first frequency value in } freqrange \text{ 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.  

\text{centerdc}(Hs,'false') \text{ shifts the data and frequency values so that the DC component is at the left edge of the spectrum.} |
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>findpeaks</td>
<td>findpeaks(Hs) finds local maxima or peaks. If no peaks are found, findpeaks returns an empty vector.</td>
</tr>
<tr>
<td></td>
<td>findpeaks(x) returns the peaks’ values, pks, and the frequencies, frqs, at which they occur.</td>
</tr>
<tr>
<td></td>
<td>findpeaks(x,'minpeakheight',mph) returns only peaks greater than the minimum peak height mph, where mph is a real scalar. The default is -Inf.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>halfrange</td>
<td><code>halfrange(Hs)</code> converts the spectrum of <code>Hs</code> to a spectrum calculated over half the Nyquist interval. All associated properties affected by the new frequency range are adjusted automatically. This method is used for <code>dspdata.pseudospectrum</code> objects. The spectrum is assumed to be from a real signal. That is, <code>halfrange</code> uses half the data points regardless of whether the data is symmetric.</td>
</tr>
<tr>
<td>normalizefreq</td>
<td><code>normalizefreq(Hs)</code> or <code>normalizefreq(Hs,true)</code> normalizes the frequency specifications in the <code>Hs</code> object to <code>Fs</code> so the frequencies are between 0 and 1. It also sets the <code>NormalizedFrequency</code> property to <code>true</code>. <code>normalizefreq(Hs,false)</code> converts the frequencies to linear frequencies. <code>normalizefreq(Hs,false,Fs)</code> sets a new sampling frequency, <code>Fs</code>. This can be used only with <code>false</code>.</td>
</tr>
<tr>
<td>onesided</td>
<td><code>onesided(Hs)</code> converts the spectrum of <code>Hs</code> to a spectrum calculated over half the Nyquist interval and containing the total signal power. All associated properties affected by the new frequency range are adjusted automatically. This method is used for <code>dspdata.psd</code> and <code>dspdata.msspectrum</code> objects. The spectrum is assumed to be from a real signal. That is, <code>onesided</code> uses half the data points regardless of whether the data is symmetric.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 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. |
### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wholerange</td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>'Mean-square Spectrum'</td>
<td>Read-only character vector</td>
</tr>
<tr>
<td>Property</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Freqencies</td>
<td>[]</td>
<td>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, ( n )) or [0, ( F_s/2 )) for odd length, and [0, ( n )) or [0, ( F_s/2 )) for even length, if ( F_s ) is specified. For a two-sided spectrum, it is [0, 2( \pi )) or [0, ( F_s )). The length of the Freqencies vector must match the length of the columns of ( \text{Data} ). If you do not specify Freqencies, 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 Freqencies, the last frequency point is compared to the next-to-last point and to ( \pi ) (or ( F_s/2 ), if ( F_s ) is specified). If the last point is closer to ( \pi ) (or ( F_s/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.</td>
</tr>
<tr>
<td>Fs</td>
<td>'Normalized'</td>
<td>Sampling frequency, which is 'Normalized' if NormalizedFrequency is true. If NormalizedFrequency is false, ( F_s ) defaults to 1 Hz.</td>
</tr>
<tr>
<td>Property</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SpectrumType</td>
<td>'Onesided'</td>
<td>Nyquist interval over which the spectral density is calculated. Valid values are 'Onesided' and 'Twosided'. See the onesided and twosided methods in dspdata for information on changing this property. The interval for Onesided is $[0 \pi)$ or $[0 \pi]$ depending on the number of FFT points, and for Twosided the interval is $[0 2\pi)$.</td>
</tr>
<tr>
<td>NormalizedFrequency</td>
<td>true</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 $[0 \pi)$ or $[0 \pi]$, set the SpectrumType to onesided; for data that ranges from $[0 2\pi)$, 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

\[ H_{mss} = \text{normalizefreq}(H_s) \]

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.

\[
Fs = 32e3; \\
t = 0:1/Fs:1-1/Fs; \\
x = \cos(2*\pi*t*1.24e3)+\cos(2*\pi*t*10e3)+\text{randn(size}(t));
\]

Compute the one-sided PSD estimate of the signal. Use the result to construct a dspdata object. Plot the mean-square spectrum.

\[
P = \text{periodogram}(x,[],[],Fs);
\]

\[H_{mss} = \text{dspdata.msspectrum}(P,'Fs',Fs,'spectrumtype','onesided');\]
plot(Hmss)

Mean-Square Spectrum

<table>
<thead>
<tr>
<th>Power (dB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-100</td>
</tr>
<tr>
<td>-90</td>
</tr>
<tr>
<td>-80</td>
</tr>
<tr>
<td>-70</td>
</tr>
<tr>
<td>-60</td>
</tr>
<tr>
<td>-50</td>
</tr>
<tr>
<td>-40</td>
</tr>
<tr>
<td>-30</td>
</tr>
<tr>
<td>-20</td>
</tr>
<tr>
<td>-10</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frequency (kHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>16</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>'Power Spectral Density'</td>
<td>Read-only character vector</td>
</tr>
<tr>
<td>Property</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Frequencies</td>
<td>[]</td>
<td>Vector of frequencies at which the power spectral density is evaluated. The range of this vector depends on the SpectrumType value. For one-sided, the default range is ([0, \pi]) or ([0, Fs/2]) for odd length, and ([0, \pi]) or ([0, Fs/2]) for even length, if Fs is specified. For two-sided, it is ([0, 2\pi]) or ([0, Fs]). If you do not specify Frequencies, a default vector is created. If one-sided is selected, then the whole number of FFT points (nFFT) for this vector is assumed to be even. If onesided is selected and you specify Frequencies, the last frequency point is compared to the next-to-last point and to (\pi) (or Fs/2, if Fs is specified). If the last point is closer to (\pi) (or Fs/2) than it is to the previous point, nFFT is assumed to be even. If it is closer to the previous point, nFFT is assumed to be odd. The length of the Frequencies vector must match the length of the columns of Data.</td>
</tr>
<tr>
<td>Fs</td>
<td>'Normalized'</td>
<td>Sampling frequency, which is 'Normalized' if NormalizedFrequency is true. If NormalizedFrequency is false Fs defaults to 1.</td>
</tr>
<tr>
<td>Property</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SpectrumType</td>
<td>'Onesided'</td>
<td>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 π) or [0 π] depending on the number of FFT points. For the whole Nyquist interval, the range is [0 2π).</td>
</tr>
<tr>
<td>NormalizedFrequency</td>
<td>true</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

```matlab
Hpsd = dspdata.psd(Pxx, 'Fs', Fs, 'SpectrumType', 'twosided');
plot(Hpsd)
```
See Also

pburng | pccov | periodogram | pmcow | 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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>'Pseudospectrum'</td>
<td>Read-only character vector</td>
</tr>
<tr>
<td>Property</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Frequencies</td>
<td>[]</td>
<td>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.</td>
</tr>
<tr>
<td>Fs</td>
<td>'Normalized'</td>
<td>Sampling frequency, which is 'Normalized' if NormalizedFrequency is true. If NormalizedFrequency is false Fs defaults to 1.</td>
</tr>
<tr>
<td>SpectrumRange</td>
<td>'Half'</td>
<td>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).</td>
</tr>
<tr>
<td>Property</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NormalizedFrequency</td>
<td>true</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 \([0 \pi]\) or \([0 \pi]\), set the SpectrumRange to half; for data that ranges from \([0 2\pi]\), 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

\[ \text{Hps} = \text{normalizefreq(Hps)} \]

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.

```matlab
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);

hps = dspdata.pseudospectrum(P,'Fs',Fs);
plot(hps)
```

Create a pseudospectrum data object to store the results. Plot the pseudospectrum.
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 Sum, Gain, and Delay blocks, in a Simulink model.

If you also have the DSP System Toolbox software installed, you can create a Biquad Filter block or a Discrete FIR Filter block instead of a subsystem block, by deselecting the **Build model using basic elements** check box.

**See Also**

**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(iy)` 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.

```matlab
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.

```matlab
dtw(x,y);
```
Change the sinusoid frequency to twice its initial value. Repeat the computation.

\[ y = \cos(2\pi \times 18 \times (1:399)/400); \]

\[ \text{dtw}(x,y); \]
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.

\[ x = \exp(2i\pi(3\cdot(1:1000)/1000)^2); \]
\[ y = \exp(2i\pi9\cdot(1:399)/400); \]

\[ \text{dtw}(x,y, 'squared'); \]
**Align Writing Samples**

Devise a typeface that resembles the output of early computers. Use it to write the word MATLAB®.

```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 \texttt{dtw}.

\texttt{dtw(one,two);}
Constrained Warping Path

Generate two signals consisting of two distinct peaks separated by valleys of different lengths. Plot the signals.

\[
x_1 = [0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0] \cdot 0.95;
\]

\[
x_2 = [0 \ 1 \ 0 \ 1 \ 0] \cdot 0.95;
\]

```matlab
subplot(2,1,1)
plot(x1)
xl = xlim;
```
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.

```matlab
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.

```matlab
[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®."

```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.

```matlab
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(fullfile(matlabroot,'examples','signal','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')
\[ [d, i_{\text{her}}, i_{\text{him}}] = \text{dtw}(\text{her}, \text{him}, '\text{absolute}'); \]
\[
\text{duration} = \frac{\text{numel}(i_{\text{her}})}{\text{fs}}
\]
\[
\text{duration} = 0.8394
\]

% To hear, type soundsc(\text{her}(i_{\text{her}}), \text{fs}), \text{pause}(2), \text{soundsc(\text{him}(i_{\text{him}}), \text{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.

```matlab
samp1 = fullfile(matlabroot,'examples','signal','MATLAB1.gif');
samp2 = fullfile(matlabroot,'examples','signal','MATLAB2.gif');

x = double(imread(samp1));
y = double(imread(samp2));

dtw(x,y);
```

![Euclidean Distance: 105656.932801](image)

Original Signal (X)  |  Aligned Signal (X)
---|---
MATLAB | MATLAB

Original Signal (Y)  |  Aligned Signal (Y)
---|---
MATLAB | MATLAB

Overlaid Original Signals  |  Overlaid Aligned Signals
---|---
MATLAB | MATLAB
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 $d_{mn}(X,Y)$, the distance between the $m$th sample of X and the $n$th sample of Y. See “Dynamic Time Warping” on page 1-540 for more information about $d_{mn}(X,Y)$.

- 'euclidean' — Root sum of squared differences, also known as the Euclidean or $\ell_2$ metric:

  $$d_{mn}(X,Y) = \sqrt{\sum_{k=1}^{K} (x_{k,m} - y_{k,n})^*(x_{k,m} - y_{k,n})}$$

- 'absolute' — Sum of absolute differences, also known as the Manhattan, city block, taxicab, or $\ell_1$ metric:

  $$d_{mn}(X,Y) = \sum_{k=1}^{K} |x_{k,m} - y_{k,n}| = \sum_{k=1}^{K} \sqrt{(x_{k,m} - y_{k,n})^*(x_{k,m} - y_{k,n})}$$
• 'squared' — Square of the Euclidean metric, consisting of the sum of squared differences:

\[ d_{mn}(X, Y) = \sum_{k=1}^{K} (x_k, m - y_k, n)^*(x_k, m - y_k, n) \]

• 'symmkl' — Symmetric Kullback-Leibler metric. This metric is valid only for real and positive \(X\) and \(Y\):

\[ d_{mn}(X, Y) = \sum_{k=1}^{K} (x_k, m - y_k, n)(\log x_k, m - \log y_k, n) \]

**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 $m$th sample of $\mathbf{X}$ and the $n$th 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
\[ d = \sum_{m \in ix, n \in iy} d_{mn}(X, Y) \]

is minimum. Acceptable distance paths start at \( d_{11}(X, Y) \), end at \( d_{MN}(X, Y) \), 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 \( d_{11}(X, Y) \) and \( d_{MN}(X, Y) \). 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:

![Lattice Diagram](image)

**References**


**See Also**
alignsignals | edr | finddelay | findsignal | xcorr

**Introduced in R2016a**
dutycycle

Duty cycle of pulse waveform

Syntax

\[
D = \text{dutycycle}(X) \\
D = \text{dutycycle}(X,FS) \\
D = \text{dutycycle}(X,T) \\
D = \text{dutycycle}(\text{TAU},\text{PRF}) \\
[D,\text{INITCROSS}] = \text{dutycycle}(X,...) \\
[D,\text{INITCROSS},\text{FINALCROSS}] = \text{dutycycle}(X,...) \\
[D,\text{INITCROSS},\text{FINALCROSS},\text{NEXTCROSS}] = \text{dutycycle}(X,...) \\
[D,\text{INITCROSS},\text{FINALCROSS},\text{NEXTCROSS},\text{MIDLEV}] = \text{dutycycle}(X,...) \\
[D,\text{INITCROSS},\text{FINALCROSS},\text{NEXTCROSS}] = \text{dutycycle}(X,...,\text{Name,Value}) \\
\text{dutycycle}(X,...)
\]

Description

\(D = \text{dutycycle}(X)\) returns the ratio of pulse width to pulse period for each positive-polarity pulse. \(D\) has length equal to the number of pulse periods in \(X\). The sample instants of \(X\) correspond to the indices of \(X\). To determine the transitions that define each pulse, \text{dutycycle} estimates the state levels of the input waveform by a histogram method. \text{dutycycle} identifies all regions, which cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-553.

\(D = \text{dutycycle}(X,FS)\) specifies the sample rate, \(FS\), in hertz as a positive scalar. The first sample instant of \(X\) corresponds to \(t=0\).

\(D = \text{dutycycle}(X,T)\) specifies the sample instants, \(T\), as a vector with the same number of elements as \(X\).

\(D = \text{dutycycle}(\text{TAU},\text{PRF})\) returns the ratio of pulse width to pulse period for a pulse width of \(\text{TAU}\) seconds and a pulse repetition frequency of \(\text{PRF}\). The product of \(\text{TAU}\) and \(\text{PRF}\) must be less than or equal to 1.
[D,INITCROSS] = dutycycle(X,...) returns a vector, INITCROSS, whose elements correspond to the mid-crossings (mid-reference level instants) of the initial transition of each pulse with a corresponding NEXTCROSS.

[D,INITCROSS,FINALCROSS] = dutycycle(X,...) returns a vector, FINALCROSS, whose elements correspond to the mid-crossings (mid-reference level instants) of the final transition of each pulse with a corresponding NEXTCROSS.

[D,INITCROSS,FINALCROSS,NEXTCROSS] = dutycycle(X,...) returns a vector, NEXTCROSS, whose elements correspond to the mid-crossings (mid-reference level instants) of the next detected transition for each pulse.

[D,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = dutycycle(X,...) returns the mid-reference level, MIDLEV. Because in a bilevel pulse waveform the state levels are constant, MIDLEV is a scalar.

[D,INITCROSS,FINALCROSS,NEXTCROSS] = dutycycle(X,...,Name,Value) returns the ratio of pulse width to pulse period with additional options specified by one or more Name,Value pair arguments.

dutycycle(X,...) plots the waveform, X, and marks the location of the mid-reference level instants and the associated reference levels. The state levels and associated lower and upper state boundaries are also plotted.

**Input Arguments**

**X**
Bilevel waveform. X is a real-valued row or column vector.

**FS**
Sample rate in hertz.

**T**
Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.
**TAU**

Pulse width in seconds. The product of TAU and PRF must be less than or equal to 1.

**PRF**

Pulse repetition frequency in pulses/second. The product of TAU and PRF must be less than or equal to 1.

**Name-Value Pair Arguments**

**MidPercentReferenceLevel**

Mid-reference level as a percentage of the waveform amplitude.

**Default:** 50

**Polarity**

Pulse polarity. Specify the polarity as 'positive' or 'negative'. If you specify 'positive', `dutycycle` looks for pulses with positive-going (positive polarity) initial transitions. If you specify 'negative', `dutycycle` looks for pulses with negative-going (negative polarity) initial transitions. See “Pulse Polarity” on page 1-552 for examples of positive and negative-polarity pulses.

**Default:** 'positive'

**StateLevels**

Low- and high-state levels. `StateLevels` is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low- and high-state levels, `dutycycle` estimates the state levels from the input waveform using the histogram method.

**Tolerance**

Tolerance levels (lower- and upper-state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-553.

**Default:** 2
Output Arguments

D

Duty cycle. Duty cycle is the ratio of the pulse width to the pulse period. Because the pulse width cannot exceed the pulse period, \(0 \leq D \leq 1\).

INITCROSS

Mid-reference level instant of initial transition. Because the duty cycle is defined as the ratio of pulse width to pulse period, initial transitions are only reported when dutycycle finds a corresponding NEXTCROSS.

FINALCROSS

Mid-reference level instant of final transition. The duty cycle is defined as the ratio of pulse width to pulse period. Thus, final transitions are only reported when dutycycle finds a corresponding NEXTCROSS.

NEXTCROSS

Mid-reference level instant of the first initial transition after the final transition of the preceding pulse.

MIDLEV

Mid-reference level. The waveform value that corresponds to the mid-reference level.

Examples

Duty Cycle of Bilevel Waveform

Determine the duty cycle of a bilevel waveform. Use the vector indices as the sample instants.

load('pulseex.mat','x')

d = dutycycle(x)

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.

```matlab
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.

\texttt{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')
More About

Duty Cycle

The energy in a bilevel, or rectangular, pulse is equal to the product of the peak power, $P_t$, and the pulse width, $\tau$. Devices to measure energy in a waveform operate on time scales longer than the duration of a single pulse. Therefore, it is common to measure the average power.
\[ P_{av} = \frac{P_t \tau}{T}, \]

where \( T \) is the pulse period.

The ratio of average power to peak power is the duty cycle:

\[ D = \frac{P_t \tau / T}{P_t} \]

**Pulse Polarity**

If the pulse has a positive-going initial transition, the pulse has positive polarity. The following figure shows a positive polarity pulse.

Equivalently, a positive-polarity (positive-going) pulse has a terminating state more positive than the originating state.

If the pulse has a negative-going initial transition, the pulse has negative polarity. The following figure shows a negative-polarity pulse.

Equivalently, a negative-polarity (negative-going) pulse has a originating state more positive than the terminating state.
State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the $\alpha$% region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1),$$

where $S_1$ is the low-state level and $S_2$ is the high-state level. Replace the first term in the equation with $S_2$ to obtain the $\alpha$% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.
References


See Also
midcross | pulseperiod | pulsesep | pulsewidth

1-554
Introduced in R2012a
**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 `getLabelNames(lss)` to see a list of labels and sublabels.
Use `setLabelValue` to add data to the set.

getLabelNames(lss)

ans = 3x1 string array
    "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
Modify the label to incorporate sperm whales and killer whales. Verify that the labeled signal set includes the two new whale types.

```matlab
editLabelDefinition(lss,'WhaleType', ...
   'Categories',{'sperm','killer'})
```

```matlab
lbldefs = getLabelDefinitions(lss);
types = lbldefs(1).Categories
```

```
5x1 string array
   "blue"
   "humpback"
   "white"
   "sperm"
   "killer"
```

The definition for trill regions has a sublabel that identifies peaks.

```matlab
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.

```matlab
editLabelDefinition(lss,["TrillRegions" "TrillPeaks"],'Description','Peaks of trill regions')
```
```matlab
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**
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 1-573 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.

```matlab
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.

```matlab
tol = 0.1;
edr(x,y,tol)
```
ans = 617

Change the sinusoid frequency to twice its initial value. Repeat the computation.

\[ y = \cos(2\pi \times 18 \times (1:399)/400); \]
\[ y(100:115) = 7; \]

`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.

\[ x = \exp(2i\pi(3\times(1:1000)/1000)^2); \]
\[ y = \exp(2i\pi9\times(1:399)/400); \]
\[ x(400:405) = 5+3j; \]
\[ x(405:410) = 7; \]
\[ y(100:107) = 3j; \]
\[ y(108:115) = 7-3j; \]
\[ edr(x,y,tol,'squared'); \]
**Edit Distance and Warping Path**

Generate two signals consisting of two distinct peaks separated by valleys of different lengths. Plot the signals.

\[
x_1 = [0 \ 1 \ 0 \ 1 \ 0]*.95;
\]

\[
x_2 = [0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0]*.95;
\]

```
subplot(2,1,1)
plot(x1)
xlim([0 12])
```
Compute the edit distance between the signals. Set a small tolerance so that the only matches are between equal samples.

```matlab
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 \(x_2\) or add seven zeros to \(x_1\).

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 1-573.

```matlab
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 ...
```

```matlab
```
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    0    0
0    0    0    0    0    0    0    0    0    0    0    NaN
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)

1-568
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);
```
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 = fullfile(matlabroot,'examples','signal','MATLAB1.gif');
samp2 = fullfile(matlabroot,'examples','signal','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 \( d_{mn}(X,Y) \), 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 \( \ell_2 \) metric:

\[
    d_{mn}(X,Y) = \sqrt{\sum_{k=1}^{K} (x_{k,m} - y_{k,n})^2}
\]
• 'absolute' — Sum of absolute differences, also known as the Manhattan, city block, taxicab, or $\ell_1$ metric:

$$d_{mn}(X, Y) = \sum_{k=1}^{K} |x_k, m - y_k, n| = \sum_{k=1}^{K} \sqrt{(x_k, m - y_k, n)^*(x_k, m - y_k, n)}$$

• 'squared' — Square of the Euclidean metric, consisting of the sum of squared differences:

$$d_{mn}(X, Y) = \sum_{k=1}^{K} (x_k, m - y_k, n)^*(x_k, m - y_k, n)$$

• 'symmkl' — Symmetric Kullback-Leibler metric. This metric is valid only for real and positive $X$ and $Y$:

$$d_{mn}(X, Y) = \sum_{k=1}^{K} (x_k, m - y_k, n)(\log x_k, m - \log y_k, n)$$

**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

$$
X = \begin{bmatrix}
x_{1,1} & x_{1,2} & \cdots & x_{1,M} \\
x_{2,1} & x_{2,2} & \cdots & x_{2,M} \\
\vdots & \vdots & \ddots & \vdots \\
x_{K,1} & x_{K,2} & \cdots & x_{K,M}
\end{bmatrix}
$$

and

$$
Y = \begin{bmatrix}
y_{1,1} & y_{1,2} & \cdots & y_{1,N} \\
y_{2,1} & y_{2,2} & \cdots & y_{2,N} \\
\vdots & \vdots & \ddots & \vdots \\
y_{K,1} & y_{K,2} & \cdots & y_{K,N}
\end{bmatrix}
$$

which have $M$ and $N$ samples, respectively. Given $d_{mn}(X,Y)$, the distance between the $m$th sample of $X$ and the $n$th sample of $Y$ specified in metric, the edr function stretches $X$ and $Y$ onto a common set of instants such that the edit distance between the signals is smallest.

Given $\varepsilon$, a real number that is the tolerance specified in tol, declare that the $m$th sample of $X$ and the $n$th sample of $Y$ match if $d_{mn}(X,Y) < \varepsilon$. 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 $X$ and $Y$, start from these facts:
Two empty signals have zero distance between them.

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, \(D\), such that:

1. \(D_{1,1} = 0\).
2. \(D_{m,1} = m - 1\) for \(m = 2, ..., M + 1\).
3. \(D_{1,n} = n - 1\) for \(n = 2, ..., N + 1\).
4. For \(m, n > 1\),

\[
D_{m,n} = \min \begin{cases} 
D_{m-1,n} + 1 \\
D_{m,n-1} + 1 \\
D_{m-1,n-1} + \begin{cases} 
0 & \text{if } d_{m,n}(X,Y) \leq \varepsilon \\
1 & \text{if } d_{m,n}(X,Y) > \varepsilon 
\end{cases}
\end{cases}
\]

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, \(i_x\) and \(i_y\), 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 \(d_{m,n}(X,Y) \leq \varepsilon\) or corresponds to removing one sample from each signal if \(d_{m,n}(X,Y) > \varepsilon\). 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 \(d_{1,1}(X,Y)\) and \(d_{M,N}(X,Y)\). This extra constraint, adjusted by the \texttt{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


See Also
alignsignals | dtw | finddelay | findsignal | xcorr

Introduced in R2016b
**ellip**

Elliptic filter design

**Syntax**

```matlab
[b,a] = ellip(n,Rp,Rs,Wp)
[b,a] = ellip(n,Rp,Rs,Wp,ftype)
```

```matlab
[z,p,k] = ellip(___)
[A,B,C,D] = ellip(___,ftype)
```

```matlab
[___] = ellip(___, 's')
```

**Description**

`[b,a] = ellip(n,Rp,Rs,Wp)` returns the transfer function coefficients of an nth-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 1-588 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(___,ftype)` 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 5 dB of passband ripple, 40 dB of stopband attenuation, and a passband edge frequency of 300 Hz, which, for data sampled at 1000 Hz, corresponds to $0.6\pi$ rad/sample. Plot its magnitude and phase responses. Use it to filter a 1000-sample random signal.

```matlab
[b,a] = ellip(6,5,40,0.6);
freqz(b,a)
```
dataIn = randn(1000,1);
dataOut = filter(b,a,dataIn);

**Bandstop Elliptic Filter**

Design a 6th-order elliptic bandstop filter with normalized edge frequencies of $0.2\pi$ and $0.6\pi$ rad/sample, 5 dB of passband ripple, and 50 dB of stopband attenuation. Plot its magnitude and phase responses. Use it to filter random data.

$[b,a] = \text{ellip}(3,5,50,[0.2\ 0.6], 'stop');$
freqz(b,a)
dataIn = randn(1000,1);
dataOut = filter(b,a,dataIn);

**Highpass Elliptic Filter**

Design a 6th-order highpass elliptic filter with a passband edge frequency of 300 Hz, which, for data sampled at 1000 Hz, corresponds to $0.6\pi$ rad/sample.Specify 3 dB of passband ripple and 50 dB of stopband attenuation. Plot the magnitude and phase responses. Convert the zeros, poles, and gain to second-order sections for use by `fvtool`.

```
[z,p,k] = ellip(6,3,50,300/500,'high');
sos = zp2sos(z,p,k);
fvtool(sos,'Analysis','freq')
```
Bandpass Elliptic Filter

Design a 20th-order elliptic bandpass filter with a lower passband frequency of 500 Hz and a higher passband frequency of 560 Hz. Specify a passband ripple of 3 dB, a stopband attenuation of 40 dB, and a sample rate of 1500 Hz. Use the state-space representation. Design an identical filter using designfilt.

\[
[A, B, C, D] = \text{ellip}(10, 3, 40, [500 560]/750);
\]
\[
d = \text{designfilt}('bandpassiir', 'FilterOrder', 20, ...
                           'PassbandFrequency1', 500, 'PassbandFrequency2', 560, ...
                           'PassbandRipple', 3, ...
\]
Convert the state-space representation to second-order sections. Visualize the frequency responses using `fvtool`.

```matlab
sos = ss2sos(A,B,C,D);
fvt = fvtool(sos,d,'Fs',1500);
legend(fvt,'ellip','designfilt')
```
Comparison of Analog IIR Lowpass Filters

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

\[
n = 5; \\
f = 2e9;
\]

\[
[zb, pb, kb] = \text{butter}(n, 2\times\pi\times f, 's'); \\
[bb, ab] = \text{zp2tf}(zb, pb, kb); \\
[hb, wb] = \text{freqs}(bb, ab, 4096);
\]

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

\[
[z1, p1, k1] = \text{cheby1}(n, 3, 2\times\pi\times f, 's'); \\
[b1, a1] = \text{zp2tf}(z1, p1, k1); \\
[h1, w1] = \text{freqs}(b1, a1, 4096);
\]

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

\[
[z2, p2, k2] = \text{cheby2}(n, 30, 2\times\pi\times f, 's'); \\
[b2, a2] = \text{zp2tf}(z2, p2, k2); \\
[h2, w2] = \text{freqs}(b2, a2, 4096);
\]

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.

\[
[ze, pe, ke] = \text{ellip}(n, 3, 30, 2\times\pi\times f, 's'); \\
[be, ae] = \text{zp2tf}(ze, pe, ke); \\
[he, we] = \text{freqs}(be, ae, 4096);
\]

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

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

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

**n — Filter order**
integer scalar

Filter order, specified as an integer scalar.
Data Types: double

**Rp — Peak-to-peak passband ripple**
positive scalar

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

If your specification, \( \ell \), is in linear units, you can convert it to decibels using 
\[ Rp = 40 \log_{10}\left(\frac{1+\ell}{1-\ell}\right). \]
Data Types: double

**Rs — Stopband attenuation**
positive scalar

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

If your specification, \( \ell \), is in linear units, you can convert it to decibels using 
\[ Rs = -20 \log_{10}\ell. \]
Data Types: double

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

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

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

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

Data Types: double
**z, p, k — Zeros, poles, and gain**
column vectors, scalar

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

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

\[
H(z) = k \frac{(1 - z(1) z^{-1}) (1 - z(2) z^{-1}) \cdots (1 - z(n) z^{-1})}{(1 - p(1) z^{-1}) (1 - p(2) z^{-1}) \cdots (1 - p(n) z^{-1})}.
\]

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

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

Data Types: double

**A, B, C, D — State-space matrices**
matri\ntes

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 × m, B is m × 1, C is 1 × 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

\[
x(k + 1) = A x(k) + B u(k)\\ y(k) = C x(k) + D u(k).
\]

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

\[
\dot{x} = A x + B u\\ y = C x + D u.
\]

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 \texttt{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.

\begin{verbatim}
\begin{verbatim}
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')
\end{verbatim}
\end{verbatim}
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.
It converts the poles, zeros, and gain into state-space form.

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.

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 \( W_p \) or \( w_1 \) and \( w_2 \).

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

\[ [z,p,k] = \text{ellipap}(n,Rp,Rs) \]

Description

\[ [z,p,k] = \text{ellipap}(n,Rp,Rs) \] 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.

dellipap sets the passband edge angular frequency \( \omega_0 \) of the elliptic filter to 1 for a normalized result. The *passband edge angular frequency* is the frequency at which the passband ends and the filter has a magnitude response of \( 10^{Rp/20} \).

Algorithms

dellipap 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


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

\[
[n,Wp] = \text{ellipord}(Wp,Ws,Rp,Rs)
\]
\[
[n,Wp] = \text{ellipord}(Wp,Ws,Rp,Rs,'s')
\]

Description

ellipord calculates the minimum order of a digital or analog elliptic filter required to meet a set of filter design specifications.

Digital Domain

\[
[n,Wp] = \text{ellipord}(Wp,Ws,Rp,Rs)
\]
returns the lowest order, \(n\), of the elliptic filter that loses no more than \(Rp\) dB in the passband and has at least \(Rs\) dB of attenuation in the stopband. The scalar (or vector) of corresponding cutoff frequencies \(Wp\), is also returned. Use the output arguments \(n\) and \(Wp\) in ellip.

Choose the input arguments to specify the stopband and passband according to the following table.
Description of Stopband and Passband Filter Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( W_p )</td>
<td>Passband corner frequency ( W_p ), the cutoff frequency, is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency, ( \pi ) radians per sample.</td>
</tr>
<tr>
<td>( W_s )</td>
<td>Stopband corner frequency ( W_s ), is a scalar or a two-element vector with values between 0 and 1, with 1 corresponding to the normalized Nyquist frequency.</td>
</tr>
<tr>
<td>( R_p )</td>
<td>Passband ripple, in decibels. This value is the maximum permissible passband loss in decibels.</td>
</tr>
<tr>
<td>( R_s )</td>
<td>Stopband attenuation, in decibels. This value is the number of decibels the stopband is attenuated with respect to the passband response.</td>
</tr>
</tbody>
</table>

Use the following guide to specify filters of different types.

Filter Type Stopband and Passband Specifications

<table>
<thead>
<tr>
<th>Filter Type</th>
<th>Stopband and Passband Conditions</th>
<th>Stopband</th>
<th>Passband</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowpass</td>
<td>( W_p &lt; W_s ), both scalars</td>
<td>((W_s,1))</td>
<td>((0,W_p))</td>
</tr>
<tr>
<td>Highpass</td>
<td>( W_p &gt; W_s ), both scalars</td>
<td>((0,W_s))</td>
<td>((W_p,1))</td>
</tr>
<tr>
<td>Bandpass</td>
<td>The interval specified by ( W_s ) contains the one specified by ( W_p ) (( W_s(1) &lt; W_p(1) &lt; W_p(2) &lt; W_s(2) )).</td>
<td>((0,W_s(1))) and ((W_p(2),1))</td>
<td>((W_p(1),W_p(2)))</td>
</tr>
<tr>
<td>Bandstop</td>
<td>The interval specified by ( W_p ) contains the one specified by ( W_s ) (( W_p(1) &lt; W_s(1) &lt; W_s(2) &lt; W_p(2) )).</td>
<td>((0,W_p(1))) and ((W_p(2),1))</td>
<td>((W_s(1),W_s(2)))</td>
</tr>
</tbody>
</table>

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

Analog Domain

\([n,W_p] = \text{ellipord}(W_p,W_s,R_p,R_s,'s')\) finds the minimum order \( n \) and cutoff frequencies \( W_p \) for an analog filter. You specify the frequencies \( W_p \) and \( W_s \) similar to those described in the Description of Stopband and Passband Filter Parameters table above,
only in this case you specify the frequency in radians per second, and the passband or the stopband can be infinite.

Use *ellipord* for lowpass, highpass, bandpass, and bandstop filters as described in the Filter Type Stopband and Passband Specifications table above.

**Examples**

**Lowpass Elliptic Filter Order**

For 1000 Hz data, design a lowpass filter with less than 3 dB of ripple in the passband, defined from 0 to 40 Hz, and at least 60 dB of ripple in the stopband, defined from 150 Hz to the Nyquist frequency, 500 Hz. Find the filter order and cutoff frequency.

\[
\begin{align*}
W_p &= 40/500; \\
W_s &= 150/500; \\
R_p &= 3; \\
R_s &= 60; \\
\text{[n,Wp]} &= \text{ellipord}(W_p,W_s,R_p,R_s)
\end{align*}
\]

\[
\begin{align*}
n &= 4 \\
W_p &= 0.0800
\end{align*}
\]

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

\[
\begin{align*}
[z,p,k] &= \text{ellip}(n,R_p,R_s,W_p); \\
sos &= \text{zp2sos}(z,p,k); \\
\text{freqz}(sos,512,1000) \\
\text{title}\left(\text{sprintf('n = \%d Elliptic Lowpass Filter',n)}\right)
\end{align*}
\]
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.

\[
W_p = \frac{[60 \ 200]}{500}; \\
W_s = \frac{[50 \ 250]}{500}; \\
R_p = 3; \\
R_s = 40;
\]
\begin{verbatim}
[n,Wp] = ellipord(Wp, Ws, Rp, Rs)
n = 5
Wp = 1\times2
 \begin{array}{ll}
0.1200 & 0.4000 \\
\end{array}

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))
\end{verbatim}
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


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) estimates emd with additional options specified by one or more Name,Value pair arguments.

demd____ 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.
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.

```matlab
load('sinusoidalSignalExampleData.mat','X','fs')
t = (0:length(X)-1)/fs;
plot(t,X)
xlabel('Time(s)')
```

```
[imf,residual,info] = emd(X,'Interpolation','pchip');
```
The decomposition stopped because the number of extrema of the residual signal is less than 'MaxNumExtrema'.

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.

```matlab
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.

```matlab
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.

```matlab
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 \( z \).

\[
\text{[imf,~,info]} = \text{emd}(s);
\]

Current IMF | #Sift Iter | Relative Tol | Stop Criterion Hit
---|---|---|---
1 | 1 | 1.8025e-06 | SiftMaxRelativeTolerance

The decomposition stopped because the current energy ratio is greater than 'MaxEnergyRatio'.

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.

\[
\begin{align*}
fs &= 10000; \\
\theta &= 25; \\
N &= 8; \\
d &= 0.02; \\
p &= 0.12;
\end{align*}
\]

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 \( \theta \) 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;

\[
\text{tx} = 0:1/\text{bpfo}::\text{t(end)};
\]
\[
\text{tx} = [\text{tx}; 1.3.^{\text{tx}-2}];
\]

nWear = 49000;
nSamples = 100000;
yImpact = pulstran(t,tImpact,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.
Add white Gaussian noise to the signals. Specify a noise variance of $1/150^2$.

```matlab
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). 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.

```matlab
imfGood = emd(yGood,'MaxNumIMF',5);
```

<table>
<thead>
<tr>
<th>Current IMF</th>
<th>#Sift Iter</th>
<th>Relative Tol</th>
<th>Stop Criterion Hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>0.017132</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.12694</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>0.14582</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0.011082</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.03463</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
</tbody>
</table>

The decomposition stopped because 'MaxNumIMF' was reached.
Use `emd` without output arguments to visualize the first three modes and the residual. Set 'Display' to 0 to hide the table.

```matlab
emd(yGood, 'MaxNumIMF', 5, 'Display', 0)
```

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.

```matlab
imfBad = emd(yBad, 'MaxNumIMF', 5);
```

```plaintext
<table>
<thead>
<tr>
<th>Current IMF</th>
<th>#Sift Iter</th>
<th>Relative Tol</th>
<th>Stop Criterion Hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.041274</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.16695</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
</tbody>
</table>
```
The decomposition stopped because 'MaxNumIMF' was reached.

\[
\text{emd}(yBad, 'MaxNumIMF', 5, 'Display', 0)
\]

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 1-1166 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 $f_s$.

```matlab
gd('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.

```matlab
emd(X,'Interpolation','pchip')
```

<table>
<thead>
<tr>
<th>Current IMF</th>
<th>#Sift Iter</th>
<th>Relative Tol</th>
<th>Stop Criterion Hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.026352</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.0039573</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.024838</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0.05929</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.11317</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0.12599</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>0.13802</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>0.15937</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>0.15923</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
</tbody>
</table>

The decomposition stopped because the number of extrema of the residual signal is less
An interactive plot with the original signal, the first 3 IMFs, and the residual is generated. 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 specifying Display 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** — Uniformly sampled time-domain signal  
vector | single data column timetable

Uniformly sampled time-domain signal, specified as either a vector or single data column timetable.
Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

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 current relative tolerance is less than SiftRelativeTolerance.

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 \(\text{MaxNumExtrema}\).

\(\text{MaxNumExtrema}\) can be specified using only positive whole numbers.

\textbf{\textit{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. \(\text{MaxEnergyRatio}\) is the ratio of the energy of the signal at the beginning of sifting and the average envelope energy. \(\text{MaxEnergyRatio}\) is one of the decomposition stop criteria, that is, decomposition stops when current energy ratio is larger than \(\text{MaxEnergyRatio}\).

\textbf{\textit{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 \textit{Interpolation} as:

- 'spline', if \(X\) is a smooth signal
- 'pchip', if \(X\) is a nonsmooth signal

'spline' interpolation method uses cubic spline, while 'pchip' uses piecewise cubic Hermite interpolating polynomial method.

\textbf{\textit{Display — Toggle information display in the command window}}

1 (default) | 0

Toggle information display in the command window, specified as the comma-separated pair consisting of 'Display' and either 1 or 0. 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 \textit{Display} as 1 to show the table or 0 to hide the table.

\section*{Output Arguments}

\textbf{\textit{imf — Intrinsic mode function}}

matrix | timetable
Intrinsic mode function, returned as a matrix or timetable. \texttt{imf} is any function whose envelope is symmetric with respect to zero and whose numbers of extrema and zero crossings differ by at most one. Use \texttt{imf} to apply Hilbert-Huang transform to perform spectral analysis on the signal.

\texttt{imf} is returned as:

- A matrix whose each column is an \texttt{imf}, when \texttt{X} is a vector
- A timetable, when \texttt{X} is a single data column timetable

\textbf{residual} — Residual of the signal
\texttt{column vector} | \texttt{single data column timetable}

Residual of the signal, returned as a column vector or a single data column timetable. \texttt{residual} represents the portion of the original signal \texttt{X} not decomposed by \texttt{emd}.

\texttt{residual} is returned as:

- A column vector, when \texttt{X} is a vector.
- A single data column timetable, when \texttt{X} is a single data column timetable.

\textbf{info} — Additional information for diagnostics
\texttt{structure}

Additional information for diagnostics, returned as a structure with the following fields:

- \texttt{NumIMF} - Number of IMFs extracted from the signal
- \texttt{NumExtrema} - Number of extrema in each IMF
- \texttt{NumZeroCrossing} - Number of zero crossings for each IMF
- \texttt{NumSifting} - Number of siftings performed for each IMF
- \texttt{MeanEnvelopeEnergy} - Energy of the mean of upper and lower envelope obtained in each IMF computation
- \texttt{RelativeTolerance} - Relative tolerance attained for each IMF

\textbf{Algorithms}

\textbf{Empirical Mode Decomposition}
emd decomposes a signal $X(t)$ into $k$ number of intrinsic mode functions (IMF), and residual $r_k(t)$ using the sifting process. A brief overview of the sifting process, listed in [1] and [2], is as follows:

1. Find local maxima and minima for signal $X(t)$ to construct an upper envelope $s_+(t)$, and a lower envelope $s_-(t)$.
2. Compute mean envelope for $i$th iteration, $m_{k,i}(t)$,
   \[
   m_{k,i}(t) = \frac{1}{2}[s_+(t) + s_-(t)]
   \]
3. With $c_k(t) = X(t)$ for the first iteration, subtract mean envelope from residual signal, $c_k(t) = c_k(t) - m_{k,i}(t)$
   If $c_k(t)$ does not match the criteria of an IMF, steps 4 and 5 are skipped. The procedure is iterated again at step 1 with the new value of $c_k(t)$.
4. If $c_k(t)$ matches the criteria of an IMF, a new residual is computed. To update the residual signal, subtract the $k$th IMF from the previous residual signal,
   \[
   r_k(t) = r_{k-1}(t) - c_k(t)
   \]
5. Then begin from step 1, using the residual obtained as a new signal $r_k(t)$, and store $c_k(t)$ as an intrinsic mode function.

For $N$ intrinsic mode functions, the original signal is represented as,

\[
X(t) = \sum_{i=1}^{N} c_i(t) + r_N(t)
\]

For more information about the sifting process, see [1] and [2].

**SiftRelativeTolerance**

SiftRelativeTolerance 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

\[
\text{RelativeTolerance} \triangleq \frac{\|c(t)_{\text{previous}} - c(t)_{\text{current}}\|^2}{\|c(t)_{\text{current}}\|^2}.
\]

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.

**MaxEnergyRatio**

Energy ratio is the ratio of the energy of the signal at the beginning of sifting and the average envelope energy [3]. Decomposition stops when current energy ratio is larger than `MaxEnergyRatio`. For \( k \) IMFs, `EnergyRatio` is defined as

\[
\text{EnergyRatio} = 10 \log_{10} \left( \frac{\|X(t)\|^2}{\|r_k(t)\|^2} \right).
\]

**References**


Extended Capabilities

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

See Also
hht

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)
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.

```matlab
Fs = 10000;
win = hann(1000);
windft = fftshift(fft(win));
```

Obtain the equivalent (rectangular) noise bandwidth of the Von Hann window.

```matlab
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.

```matlab
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 1-628 for an example that plots the equivalent rectangular bandwidth over the magnitude spectrum of a von Hann window.

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')

evolve(____ )

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.

evolve(____ ) 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.

```matlab
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.
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.
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.

```matlab
[up,lo] = envelope(q,100,"analytic");
hold on
```
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.

```matlab
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')

evelope(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.

\[
t = 0.5:1/100:-2.49;
\]
\[
z = \text{airy}(t*10).*\exp(-t.^2);
\]

plot(z)
Determine the envelopes of the sequence using the magnitude of its analytic signal. Plot the envelopes.

\texttt{envelope(z)}
Compute the analytic envelope of the signal using a 50-tap Hilbert filter.

evelope(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.

\texttt{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

\[
es = \text{envspectrum}(x,fs) \\
es = \text{envspectrum}(xt) \\
es = \text{envspectrum}(___,\text{Name},\text{Value}) \\
[es,f,env,t] = \text{envspectrum}(___) \\
\text{envspectrum}(___)
\]

Description

\[es = \text{envspectrum}(x,fs)\] 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 \(es\).

\[es = \text{envspectrum}(xt)\] returns the envelope spectrum of a signal stored in the MATLAB timetable \(xt\).

\[es = \text{envspectrum}(___,\text{Name},\text{Value})\] 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.

\[[es,f,env,t] = \text{envspectrum}(___)\] returns \(f\), a vector of frequencies at which \(es\) is computed; \(env\), the envelope signal; and \(t\), the times at which \(env\) is computed.

\[\text{envspectrum}(___)\] 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.

```matlab
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.

\[ t = 0:1/fs:1-1/fs; \]
\[ z = [1 0.5 0.2 0.1 0.05]*\sin(2*\pi*f0*[1 2 3 4 5]'\cdot t)/5; \]

```
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 \( \theta \) is the bearing contact angle. Assume a contact angle of zero and compute the BPFO.

\[
\begin{align*}
\text{ca} & = 0; \\
\text{bpfo} & = n f_0 / 2 \cdot (1 - d / p \cdot \cos(\text{ca})) \\
\text{bpfo} & = 83.3333
\end{align*}
\]

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.

\[
\begin{align*}
\text{fImpact} & = 3000; \\
\text{tImpact} & = 0:1/\text{fs}:5e-3-1/\text{fs}; \\
\text{xImpact} & = \sin(2\pi f_{\text{Impact}} \cdot \text{tImpact}) \cdot \text{flattopwin(length(tImpact))}' / 10; \\
\text{xComb} & = \text{zeros(size(t))}; \\
\text{xComb}(1:\text{fs}/\text{bpfo}:end) & = 1; \\
\text{x} & = \text{conv(xComb, xImpact, 'same')}/3; \\
\text{plot(t, x+z)} \\
\text{xlim([0.4 0.5])}
\end{align*}
\]
Add white Gaussian noise to the signals. Specify a noise variance of \(1/30^2\). Plot 0.1 second of data.

\[
y_{\text{Good}} = z + \frac{\text{randn(size(z))}}{30};
\]

\[
y_{\text{Bad}} = x+z + \frac{\text{randn(size(z))}}{30};
\]

plot(t,yGood,t,yBad)

xlim([0.4 0.5])

legend('Healthy','Damaged')
Compute and plot the envelope signals and spectra.

```matlab
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.

```matlab
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.

```matlab
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.

\texttt{xlim([0 10*bpfo]/1000)}
The spectrum of the faulty bearing shows BPFO harmonics modulated by the impact frequency.

```matlab
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.

```plaintext
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.

```matlab
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.

```matlab
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.

```matlab
envspectrum(ttb,'Band',[100 4900])
```
The envelope spectra of the signals have peaks at integer multiples of the repetition rate of \(1/0.01 = 0.1 \text{ kHz}\). 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.

\[
[~,~,ttbenv,ttbt] = \text{envspectrum}(ttb,'\text{Band}',[100 \ 4900]);
\]

whos `ttb`*

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ttb</td>
<td>2000x1</td>
<td>49034</td>
<td>timetable</td>
<td></td>
</tr>
</tbody>
</table>

* `whos ttb`
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.

```matlab
btb = timetable(dt,Y(:,1),Y(:,2));
[~,~,btbenv,btbt] = envspectrum(btb,'Band',[100 4900]);
whos btb*
```

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.

```matlab
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.

```matlab
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')
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 \texttt{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 \texttt{envspectrum}.

\begin{verbatim}
envspectrum(z,fs,'Method','hilbert','FilterOrder',10)
\end{verbatim}
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 \texttt{envspectrum} computes the envelope spectrum independently for each column and returns the result in the corresponding column of \texttt{es}.

Example: \( \cos\left(\pi/4 \times (0:159)\right) + \text{randn}(1,160) \) is a single-channel row-vector signal.

Example: \( \cos\left(\pi \times [4;2] \times (0:159)\right)' + \text{randn}(160,2) \) 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” (MATLAB).

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 comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

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 1-670 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 \texttt{env} has a single variable containing a matrix.
• If the input is a timetable with multiple variables consisting of vectors, then \texttt{env} has multiple variables consisting of vectors.

\textbf{t — Time values}

\texttt{vector}

Time values at which the envelope signal is computed, returned as a vector.

If the input to \texttt{envspectrum} is a timetable, then \texttt{t} has the same format as the time values of the input timetable.

\section*{Algorithms}

\texttt{envspectrum} initially removes the DC bias from the input signal, \texttt{x}, and then computes the envelope signal.

• If 'Method' is set to 'hilbert', the function:

\begin{enumerate}
\item Bandpass-filters the signal. The FIR filter has an order specified by 'FilterOrder' and cutoff frequencies at \texttt{ba(1)} and \texttt{ba(2)}, where \texttt{ba} is a frequency band specified using 'Band'.
\item Computes the analytic signal using the \texttt{hilbert} function.
\item Computes the envelope signal as the absolute value of the analytic signal.
\end{enumerate}

• If 'Method' is set to 'demod', the function:

\begin{enumerate}
\item Performs complex demodulation of the signal. The signal is multiplied by \(\exp(j2\pi f_0 t)\), where \(f_0 = \frac{\texttt{ba(1)} + \texttt{ba(2)}}{2}\).
\item Lowpass-filters the demodulated signal to compute the analytic signal. The FIR filter has an order specified by 'FilterOrder' and a cutoff frequency of \((\texttt{ba(2)} - \texttt{ba(1)})/2\).
\item Computes the envelope signal as twice the absolute value of the analytic signal.
\end{enumerate}

After computing the envelope signal, the function removes the DC bias from the envelope and computes the envelope spectrum using the FFT.
References


Extended Capabilities

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

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's numerator and denominator

Syntax

\[[b,a] = \text{eqtflength}(\text{num},\text{den})\]
\[[b,a,n,m] = \text{eqtflength}(\text{num},\text{den})\]

Description

\[[b,a] = \text{eqtflength}(\text{num},\text{den})\] modifies the vector \(\text{num}\) and/or the vector \(\text{den}\), so that the resulting output vectors \(b\) and \(a\) have the same length. The input vectors \(\text{num}\) and \(\text{den}\) may have different lengths. The vector \(\text{num}\) represents the numerator polynomial of a given discrete-time transfer function, and the vector \(\text{den}\) represents its denominator. The resulting numerator \(b\) and denominator \(a\) represent the same discrete-time transfer function, but these vectors have the same length.

\[[b,a,n,m] = \text{eqtflength}(\text{num},\text{den})\] modifies the vectors as above and also returns the numerator order \(n\) and the denominator \(m\), not including any trailing zeros.

Use eqtflength to obtain a numerator and denominator of equal length before applying transfer function conversion functions such as tf2ss and tf2zp to discrete-time models.

Examples

\[\text{num} = [1 \ 0.5];\]
\[\text{den} = [1 \ 0.75 \ 0.6 \ 0];\]
\[[b,a,n,m] = \text{eqtflength}(\text{num},\text{den});\]

Algorithms

eqtflength\((\text{num},\text{den})\) appends zeros to either \(\text{num}\) or \(\text{den}\) as necessary. If both \(\text{num}\) and \(\text{den}\) have trailing zeros in common, these are removed.
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 10% reference level to the 90% reference level (See “Percent Reference Levels” on page 1-687). 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 1-687). 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)
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])

**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)

title('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

\textbf{X} — Bilevel waveform

\texttt{real vector}

Bilevel waveform, specified as a real-valued vector.
**Fs — Sample rate**
positive real scalar

Sample rate, specified in as a positive real integer in Hz.

**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 comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

**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 1-687 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 1-687 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 lower 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 lower 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**

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the $\alpha\%$ region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1),$$

where $S_1$ is the low-state level and $S_2$ is the high-state level. Replace the first term in the equation with $S_2$ to obtain the $\alpha\%$ tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.
References


See Also

risetime | slewrate | statelevels

Introduced in R2012a
fftfilt

FFT-based FIR filtering using overlap-add method

Syntax

\[
y = \text{fftfilt}(b,x) \\
y = \text{fftfilt}(b,x,n) \\
y = \text{fftfilt}(d,x) \\
y = \text{fftfilt}(d,x,n) \\
y = \text{fftfilt}(\text{gpuArray}b,\text{gpuArray}X,n)
\]

Description

\[
y = \text{fftfilt}(b,x) \text{ filters the data in vector } x \text{ with the filter described by coefficient vector } b.
\]
\[
y = \text{fftfilt}(b,x,n) \text{ uses } n \text{ to determine the length of the FFT.}
\]
\[
y = \text{fftfilt}(d,x) \text{ filters the data in vector } x \text{ with a digitalFilter object } d.
\]
\[
y = \text{fftfilt}(d,x,n) \text{ uses } n \text{ to determine the length of the FFT.}
\]
\[
y = \text{fftfilt}(\text{gpuArray}b,\text{gpuArray}X,n) \text{ filters the data in the gpuArray object } \text{gpuArray}X \text{ with the FIR filter coefficients in the gpuArray stored in } \text{gpuArray}b.
\]

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 the average times.

fprintf('%4d-tap filter averages: fftfilt: %f s; filter: %f s
',shrt,tfs,tls)

20-tap filter averages: fftfilt: 0.286835 s; filter: 0.010940 s

fprintf('%4d-tap filter averages: fftfilt: %f s; filter: %f s
',long,tfl,tll)
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.

```matlab
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`.

```matlab
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.

```matlab
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, \texttt{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 \( x \). \texttt{fftfilt} works for both real and complex inputs.

\textbf{\texttt{n} — FFT length}

\texttt{positive integer}

FFT length, specified as a positive integer. By default, \texttt{fftfilt} chooses an FFT length and a data block length that guarantee efficient execution time.

\textbf{\texttt{d} — Digital filter}

\texttt{digitalFilter object}

Digital filter, specified as a \texttt{digitalFilter} object. Use \texttt{designfilt} to generate \( d \) based on frequency-response specifications.

\textbf{\texttt{gpuArrayb, gpuArrayX} — GPU arrays}

\texttt{gpuArray}

GPU arrays, specified as a \texttt{gpuArray} object. \texttt{gpuArrayb} contains the filter coefficients, and \texttt{gpuArrayX} is the input data. See “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox) for details on \texttt{gpuArray} objects. Using \texttt{fftfilt} with \texttt{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 \texttt{gpuArray} object. See “Overlap-Add Filtering on the GPU” on page 1-691 for an example of overlap-add filtering on the GPU.

\textbf{Output Arguments}

\textbf{\texttt{y} — Output data}

\texttt{vector | matrix | gpuArray}

Output data, returned as a vector, matrix or \texttt{gpuArray} object.

\textbf{More About}

\textbf{Comparison to filter function}

When the input signal is relatively large, \texttt{fftfilt} is faster than \texttt{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
\[
\frac{L(1 + \log_2 L)}{(NL)} = \frac{1 + \log_2 L}{N}
\]
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:

\[
Y(z) = (b(1) + b(2)z^{-1} + \cdots + b(nb + 1)z^{-nb})X(z)
\]

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 \( N \).

\[
x \quad 0 \quad 1 \quad 2 \quad \cdots \quad \text{ceil}(nx/L) \cdot L
\]

and convolves each block with the filter \( b \) by

\[
y = \text{ifft}(\text{fft}(x(i:i+L-1),\text{nfft}).*\text{fft}(b,\text{nfft}));
\]

where \( \text{nfft} \) is the FFT length. fftfilt overlaps successive output sections by \( n-1 \) points, where \( n \) is the length of the filter, and sums them.

\[
x \quad 0 \quad L \quad nb-1 \quad 2L \quad nb-1 \quad 3L \quad nb-1 \quad \cdots
\]
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

\[ 2^{\text{nextpow2}(\text{length}(b) + \text{length}(x) - 1)} \]

This computes

\[ y = \text{ifft} (\text{fft}(B,nfft).*\text{fft}(X,nfft)) \]

If you supply a value for n, fftfilt chooses an FFT length, nfft, of \( 2^{\text{nextpow2}(n)} \) and a data block length of \( \text{nfft} - \text{length}(b) + 1 \). If n is less than \( \text{length}(b) \), fftfilt sets n to \( \text{length}(b) \).

References


Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

Digital filter 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

\[
\begin{align*}
y &= \text{fillgaps}(x) \\
y &= \text{fillgaps}(x, \text{maxlen}) \\
y &= \text{fillgaps}(x, \text{maxlen}, \text{order}) \\
\end{align*}
\]

Description

\( y = \text{fillgaps}(x) \) replaces any NaNs 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.

\( y = \text{fillgaps}(x, \text{maxlen}) \) 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.

\( y = \text{fillgaps}(x, \text{maxlen}, \text{order}) \) specifies the order of the autoregressive model used to reconstruct the gaps.

\text{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.

\[
P(50:60, 80:90) = \text{NaN}; \\
P(100:110, 20:30) = \text{NaN}; \\
P(100:110, 100:110) = \text{NaN}; \\
P(20:30, 110:120) = \text{NaN};
\]

contour(P,nc)
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.

```matlab
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.

```matlab
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).

\[ h = f; \]

\[ h(x > 0.8 \& x < 0.6) = \text{NaN}; \]
\[ h(x > 0.2 \& x < 0.1) = \text{NaN}; \]
\[ h(x > 0.4 \& x < 0.7) = \text{NaN}; \]

Fill the gaps using the default settings of fillgaps. Plot the original and reconstructed functions.

\[ y = \text{fillgaps}(h); \]
\[ \text{plot}(x,f,'.',x,y) \]
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.

```matlab
y = fillgaps(h,3,1);
plot(x,f,'.',x,y)
```
Specify a maximum prediction-sequence length of 80 samples and a model order of 40. Plot the original and reconstructed functions.

```matlab
y = fillgaps(h,80,40);
plot(x,f,'.',x,y)
```
Change the model order to 70. Plot the original and reconstructed functions.

```matlab
y = fillgaps(h,80,70);
plot(x,f,'.',x,y)
```
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.

```matlab
Fs = 1000;
t = 0:1/Fs:1-1/Fs;
```
\begin{verbatim}
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)
\end{verbatim}
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 NaNs 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: \(\cos\left(\pi./\left[4;2\right] \ast (0:159)\right) \ast \text{reshape}\left(\text{ones}(64,1) \ast [0 \text{ NaN} 0 \text{ NaN} 0], 160, 2\right)\) is a two-channel signal with large gaps.

Data Types: single | double

\textbf{maxlen — Maximum length of prediction sequences}

positive integer

Maximum length of prediction sequences, specified as a positive integer. If you leave \text{maxlen} unspecified, then \text{fillgaps} iteratively fits autoregressive models using all previous points for forward estimation and all future points for backward estimation.

Data Types: single | double

\textbf{order — Autoregressive model order}

'aic' (default) | positive integer

Autoregressive model order, specified as 'aic' or a positive integer. The order is truncated when \text{order} is infinite or when there are not enough available samples. If you specify \text{order} as 'aic', or leave it unspecified, then \text{fillgaps} selects the order that minimizes the Akaike information criterion.

Data Types: single | double | char | string

**Output Arguments**

\textbf{y — Reconstructed signal}

vector | matrix

Reconstructed signal, returned as a vector or matrix.

**References**


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.

<table>
<thead>
<tr>
<th>Response Method</th>
<th>Description of Resulting Filter Design</th>
<th>Filter Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arbgrpdelay</code> on page 1-731</td>
<td>Arbitrary group delay filter design</td>
<td><code>fdesign.arbgrpdelay</code></td>
</tr>
<tr>
<td><code>arbmag</code> on page 1-731</td>
<td>Arbitrary magnitude filter design</td>
<td><code>fdesign.arbmag</code></td>
</tr>
<tr>
<td><code>arbmagnphase</code> on page 1-731</td>
<td>Arbitrary response filter (magnitude and phase)</td>
<td><code>fdesign.arbmagnphase</code></td>
</tr>
<tr>
<td><code>audioweighting</code> on page 1-735</td>
<td>Audio weighting filter</td>
<td><code>fdesign.audioweighting</code></td>
</tr>
<tr>
<td><code>bandpass</code> on page 1-736 or bp</td>
<td>Bandpass filter</td>
<td><code>fdesign.bandpass</code></td>
</tr>
<tr>
<td><code>bandstop</code> on page 1-742 or bs</td>
<td>Bandstop filter</td>
<td><code>fdesign.bandstop</code></td>
</tr>
<tr>
<td><code>cic</code> on page 1-748</td>
<td>CIC filter</td>
<td><code>fdesign.decimator(M,'cic',...)</code> or <code>fdesign.interpolator(L,'cic',...)</code> See <code>fdesign.decimator</code> and <code>fdesign.interpolator</code></td>
</tr>
<tr>
<td><code>ciccomp</code> on page 1-750</td>
<td>CIC compensator</td>
<td><code>fdesign.ciccomp</code></td>
</tr>
<tr>
<td><code>comb</code> on page 1-754</td>
<td>Comb filter</td>
<td><code>fdesign.comb</code></td>
</tr>
</tbody>
</table>
### Response Method | Description of Resulting Filter Design | Filter Object
---|---|---
diff on page 1-757 | Differentiator filter | fdesign.differentiator
fracdelay on page 1-761 | Fractional delay filter | fdesign.fracdelay
halfband on page 1-762 or hb | Halfband filter | fdesign.halfband
highpass on page 1-766 or hp | Highpass filter | fdesign.highpass
hilb on page 1-771 | Hilbert filter | fdesign.hilbert
isinc on page 1-775, isinclp on page 1-775, or isinchp on page 1-775 | Inverse sinc lowpass or highpass filter | fdesign.isinclp and fdesign.isinchp
lowpass on page 1-781 or lp | Lowpass filter (default) | fdesign.lowpass
notch on page 1-786 | Notch filter | fdesign.notch
nyquist on page 1-786 | Nyquist filter | fdesign.nyquist
octave on page 1-792 | Octave filter | fdesign.octave
parameq on page 1-793 | Parametric equalizer filter | fdesign.parameq
peak on page 1-797 | 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).

**Note** The filterBuilder dialog box includes an Apply option. Each time you click Apply, filterBuilder writes the modified filter to your MATLAB workspace. This modified filter has the variable name you assign in **Save variable as**. To apply changes...
without overwriting the variable in your workspace, change the variable name in **Save variable as** before you click **Apply**.

There are three tabs in the 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.
<table>
<thead>
<tr>
<th>Arithmetic List Entry</th>
<th>Effect on the Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double precision</td>
<td>All filtering operations and coefficients use double-precision, floating-point representations and math. When you use <code>filterBuilder</code> to create a filter, double precision is the default value for the Arithmetic property.</td>
</tr>
<tr>
<td>Single precision</td>
<td>All filtering operations and coefficients use single-precision floating-point representations and math.</td>
</tr>
<tr>
<td>Fixed point</td>
<td>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 <code>filterBuilder</code> is available only when you install Fixed-Point Designer™ software along with DSP System Toolbox software.</td>
</tr>
</tbody>
</table>

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 associated 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.
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**

\( F_s \), specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

**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 \( H(\omega) = \text{sinc}(C\omega)^{(-P)} \) and for a highpass design through \( H(\omega) = \text{sinc}(C(1-\omega))^{(-P)} \).

Sinc power
Negative power of passband magnitude response. The sinc power, \( P \), parameterizes the passband magnitude response for a lowpass design through \( H(\omega) = \text{sinc}(C\omega)^{(-P)} \) and for a highpass design through \( H(\omega) = \text{sinc}(C(1-\omega))^{(-P)} \).

Filter implementation
Structure
For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure.

Use a System object to implement filter
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 ($F_{\text{pass}}$) and Stopband frequency ($F_{\text{stop}}$) represent transition regions where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- Passband and stopband 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 1-797.

**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:

$$bw = \frac{Fs}{2 \times \text{Band}}.$$ 

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 \( \text{db}(\text{sqrt}(.5)) \), or -3 dB relative to the center frequency. The Bandwidth property only applies when the Frequency
**constraints** are: Center frequency, bandwidth, passband width, Center frequency, bandwidth, stopband width, or Center frequency, bandwidth.

### Passband width
The passband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Passband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is **Center frequency, bandwidth, passband width**.

### Stopband width
The stopband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Stopband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is **Center frequency, bandwidth, stopband width**.

### Low frequency
Enter the low frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is **Low frequency, high frequency**. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

### High frequency
Enter the high frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is **Low frequency, high frequency**. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

### Gain specifications
Depending on the filter order and frequency constraints, the possible gain constraints change. Also, once you choose the gain constraints the input boxes in this area change to reflect the selection.

### Gain constraints
Select the specification array to represent gain constraints, and remember that not all of these options are available for all configurations. The following is a list of all available options:

- Reference, center frequency, bandwidth, passband
- Reference, center frequency, bandwidth, stopband
- Reference, center frequency, bandwidth, passband, stopband
- Reference, center frequency, bandwidth

**Gain units**

Specify the gain units either dB or squared. These units are used for all gain specifications in the dialog box.

**Reference gain**

The reference gain determines the level to which the filter magnitude attenuates in **Gain units**. The reference gain is a floor gain for the filter magnitude response. For example, you may use the reference gain together with the Center frequency gain to leave certain frequencies unattenuated (reference gain of 0 dB) while boosting other frequencies.

**Bandwidth gain**

Specifies the gain in **Gain units** at which the bandwidth is defined. This property applies only when the **Frequency constraints** specification contains a bandwidth parameter, or is Low frequency, high frequency.

**Center frequency gain**

Specify the center frequency in **Gain units**

**Passband gain**

The passband gain determines the level in **Gain units** at which the passband is defined. The passband is determined either by the Passband width value, or the Low frequency and High frequency values in the **Frequency specifications** section.

**Stopband gain**

The stopband gain is the level in **Gain units** at which the stopband is defined. This property applies only when the Order mode is minimum and the **Frequency constraints** are Center frequency, bandwidth, stopband width.

**Boost/cut gain**

The boost/cut gain applies only when the designing a shelving filter. Shelving filters include the Shelf type parameter in the **Frequency constraints** specification. The gain in the passband of the shelving filter is increased by **Boost/cut gain** dB from a floor gain of 0 dB.
**Algorithm**

**Design method**
Select the design method from the drop-down list. Different IIR design methods are available depending on the filter constraints you specify.

**Scale SOS filter coefficients to reduce chance of overflow**
Select the check box to scale the filter coefficients.

**Filter implementation**

**Structure**
Select filter structure. The possible choices are:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

**Use a System object to implement filter**
Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is 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.

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.

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

• “Getting Started with Filter Designer”
• “FIR Bandpass Filter with Asymmetric Attenuation”
• “Arbitrary Magnitude Filter”

See Also

Apps
Signal Analyzer | Window Designer
**Functions**

*FVTool* | *designfilt* | *wvtool*

**Topics**

“Getting Started with Filter Designer”
“FIR Bandpass Filter with Asymmetric Attenuation”
“Arbitrary Magnitude Filter”

**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 $0.5\pi$ rad/s and a filter order of 5.
[b,a] = butter(5,0.5);
L2 = filternorm(b,a,2,1e-10)
L2 = 0.7071

Compute the infinity-norm of an FIR Hilbert transformer of order 30 and normalized transition width 0.2\pi rad/s.

b = firpm(30,[.1 .9],[1 1],'Hilbert');
Linf = filternorm(b,1,inf)
Linf = 1.0028

**Algorithms**

Given a filter with frequency response $H(e^{j\omega})$, the $L_p$-norm for $1 \leq p < \infty$ is given by

\[
\|H(e^{j\omega})\|_p = \left( \frac{1}{2\pi} \int_{-\pi}^{\pi} |H(e^{j\omega})|^p d\omega \right)^{1/p}.
\]

For the case $p \to \infty$, the $L_\infty$-norm is

\[
\|H(e^{j\omega})\|_\infty = \max_{-\pi \leq \omega \leq \pi} |H(e^{j\omega})|.
\]

For the case $p = 2$, Parseval's theorem states that

\[
\|H(e^{j\omega})\|_2 = \left( \frac{1}{2\pi} \int_{-\pi}^{\pi} |H(e^{j\omega})|^2 d\omega \right)^{1/2} = \left( \sum_n |h(n)|^2 \right)^{1/2},
\]

where $h(n)$ is the impulse response of the filter. The energy of the impulse response is the squared $L_2$-norm.

**References**

See Also
norm | zp2sos

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, `filtfilt` reverses the filtered sequence and runs it back through the filter. The result has the following characteristics:

- Zero phase distortion.
- A filter transfer function 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**

1-808
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.

Use `filtfilt` to 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.

```matlab
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')
```

```matlab
function wform = ecg(N)
    % Generate a synthetic ECG waveform
    % N is the number of samples
    % wform is the output waveform
    fs = 1000; % Sampling frequency
    t = 0:1/fs:(N-1)/fs;
    QRS = [0 0.1 0.1 0 0 0.1 0.1 0];
    PR = [0 0.05 0.05 0];
    ST = [0 0.1 0.1 0.1 0 0];
    wform = zeros(1,N);
    for i = 1:length(QRS)
        x = sin(2*pi*250*QRS(i)*t);
        wform = wform + x;
    end
    % Add PR and ST segments
    for i = 1:length(PR)
        x = sin(2*pi*200*PR(i)*t);
        wform = wform + x;
    end
    for i = 1:length(ST)
        x = sin(2*pi*100*ST(i)*t);
        wform = wform + x;
    end
end
```
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.

```matlab
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);
```
Zero-phase filtering reduces noise in the signal and preserves the QRS complex at the same time it occurs in the original. Conventional filtering reduces noise in the signal, but delays the QRS complex.

Repeat the above using a Butterworth second-order section filter.
This is the function that generates the ECG waveform.

```matlab
function x = ecg(L)
% ECG Electrocardiogram (ECG) signal generator.
```

```matlab
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')
```
% 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
\[ d0 = [0,27,59,91,131,141,163,185,195,275,307,339,357,390,440]\];
\[ a = a0 / \text{max}(a0); \]
\[ d = \text{round}(d0 * L / d0(15)); \] % Scale them to fit in length L
\[ d(15)=L; \]

\begin{verbatim}
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{verbatim}

\begin{verbatim}
end
\end{verbatim}

**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: \( b = \left[ \frac{1 3 3 1}{6} \right] \) and \( a = \left[ \frac{3 0 1 0}{3} \right] \) 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. **filtfilt** operates along the first array dimension of \( x \) with size greater than 1.
Example: \( \cos\left(\frac{\pi}{4} \times (0:159)\right) + \text{randn}(1,160) \) is a single-channel row-vector signal.

Example: \( \cos\left(\frac{\pi}{4} \times [4;2] \times (0:159)\right) + \text{randn}(160,2) \) 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 filtfilt 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 \( [b_i(1) \ b_i(2) \ b_i(3) \ a_i(1) \ a_i(2) \ a_i(3)] \).

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 \( 0.5\pi \) 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: \( d = \text{designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5) } \) specifies a third-order Butterworth filter with a normalized 3-dB frequency of \( 0.5\pi \) 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


Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

Digital filter objects are not supported for code generation.

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

\[ z = \text{filtic}(b,a,y,x) \]
\[ z = \text{filtic}(b,a,y) \]

Description

\[ z = \text{filtic}(b,a,y,x) \] finds the initial conditions, \( z \), for the delays in the transposed direct-form II filter implementation given past outputs \( y \) and inputs \( x \). The vectors \( b \) and \( a \) represent the numerator and denominator coefficients, respectively, of the filter's transfer function.

The vectors \( x \) and \( y \) contain the most recent input or output first, and oldest input or output last.

\[
\begin{align*}
x &= [x(-1), x(-2), x(-3), \ldots, x(-n)] \\
y &= [y(-1), y(-2), y(-3), \ldots, y(-m)]
\end{align*}
\]

where \( n \) is \( \text{length}(b)-1 \) (the numerator order) and \( m \) is \( \text{length}(a)-1 \) (the denominator order). If \( \text{length}(x) \) is less than \( n \), \text{filtic} pads it with zeros to length \( n \); if \( \text{length}(y) \) is less than \( m \), \text{filtic} pads it with zeros to length \( m \). Elements of \( x \) beyond \( x(n-1) \) and elements of \( y \) beyond \( y(m-1) \) are unnecessary so \text{filtic} ignores them.

Output \( z \) is a column vector of length equal to the larger of \( n \) and \( m \). \( z \) describes the state of the delays given past inputs \( x \) and past outputs \( y \).

\[ z = \text{filtic}(b,a,y) \] assumes that the input \( x \) is 0 in the past.

The transposed direct-form II structure is shown in the following illustration.
\( n - 1 \) is the filter order.

`filtic` works for both real and complex inputs.

**Diagnostics**

If any of the input arguments `y`, `x`, `b`, or `a` is not a vector (that is, if any argument is a scalar or array), `filtic` gives the following error message:

 Requires vector inputs.

**Algorithms**

`filtic` performs a reverse difference equation to obtain the delay states `z`.

**References**


**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 ith 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.

```matlab
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.

```matlab
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: \([b,a] = \text{butter}(20,0.25)\)

Data Types: single | double
Complex Number Support: Yes

\(\textbf{sos} \rightarrow \textbf{Matrix of second-order sections}\)

Matrix of second order-sections, specified as a \(K\)-by-6 matrix. The system function of the \(K\)th biquad filter has the rational \(Z\)-transform

\[
H_k(z) = \frac{B_k(1) + B_k(2)z^{-1} + B_k(3)z^{-2}}{A_k(1) + A_k(2)z^{-1} + A_k(3)z^{-2}}.
\]

The coefficients in the \(K\)th row of the matrix, \(sos\), are ordered as follows.

\[
[B_k(1) B_k(2) B_k(3) A_k(1) A_k(2) A_k(3)].
\]

The frequency response of the filter is the system function evaluated on the unit circle with

\[
z = e^{j2\pi f}.
\]

Data Types: single | double
Complex Number Support: Yes

\(\textbf{d} \rightarrow \textbf{Digital filter}\)

digitalFilter object

Digital filter, specified as a \texttt{digitalFilter} object. Use \texttt{designfilt} to generate a digital filter based on frequency-response specifications.

Example: \(d = \text{designfilt}('\text{lowpassiir}', '\text{FilterOrder}', 3, '\text{HalfPowerFrequency}', 0.5)\)

specifies a third-order Butterworth filter with normalized 3 dB frequency \(0.5\pi\) rad/sample.
Output Arguments

n — Filter order
integer

Filter order, specified as an integer.

Extended Capabilities

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

See Also

designfilt | digitalFilter | isallpass | ismaxphase | isminphase | 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.

<table>
<thead>
<tr>
<th>filtstates.structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filtstates.dfiir</td>
<td>filtstates for IIR direct-form I filters (dfilt.df1, dfilt.df1t, dfilt.df1sos, and dfilt.df1tsos)</td>
</tr>
<tr>
<td>filtstates.cic</td>
<td>filtstates for cascaded integrator comb filters. (Available only with DSP System Toolbox and Fixed-Point Designer products.)</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>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.</td>
</tr>
<tr>
<td>single</td>
<td>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.)</td>
</tr>
</tbody>
</table>
Examples

This example demonstrates the interaction of filtstates with a dfilt.df1 object.

```matlab
[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.

\[
b = \text{fir1}(30,0.25);
\]

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.

\[ [b, a] = \text{butter}(30, 0.25); \]
\[ \text{filt2block}(b, a) \]

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.

\[ b = \text{firl}(30, 0.25); \]
\[ \text{filt2block}(b, '\text{FilterStructure}', '\text{directFormTransposed}') \]
Generate IIR Filter with Direct Form I Structure

Design a 30th-order IIR Butterworth filter. Specify a cutoff frequency of π/4 rad/sample. Create a Simulink® block with a direct form I structure.

\[
[b,a] = \text{butter}(30,0.25);
\]

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 π/5 rad/sample. Obtain the filter in biquad form and generate a Simulink® subsystem block from the second order sections.

\[
[z,p,k] = \text{butter}(5,0.2);
\]

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.

\[
B = \text{fir1}(30,.25);
\]

filt2block(B,'subsystem','BlockName','Lowpass FIR',... 'FrameBasedProcessing',false)
Design a highpass elliptic filter with normalized stopband frequency 0.45 and normalized passband frequency 0.55. Specify a stopband attenuation of 40 dB and a passband ripple of 0.5 dB. Implement the filter as a Direct Form II structure, call it "HP", and place it in a new Simulink® model.

```matlab
b = fir1(30,0.25);
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 Kth 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 Kth row of the matrix, sos, are ordered as follows:

\[
[B_k(1) \ B_k(2) \ B_k(3) \ A_k(1) \ A_k(2) \ A_k(3)]
\]

The frequency response of the filter is its transfer function evaluated on the unit circle with \( z = e^{i2\pi f} \).

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: 
\[d = \text{designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)}\]

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.

<table>
<thead>
<tr>
<th>Input</th>
<th>Filter Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b</code></td>
<td>'directForm' (default), 'directFormTransposed', 'directFormSymmetric', 'directFormAntiSymmetric', 'overlapAdd'. The 'overlapAdd' structure is only available when you omit 'subsystem' and requires a DSP System Toolbox software license.</td>
</tr>
<tr>
<td><code>a</code></td>
<td>'directForm2' (default), 'directForm1', 'directForm1Transposed', 'directForm2', 'directForm2Transposed'</td>
</tr>
<tr>
<td><code>sos</code></td>
<td>'directForm2Transposed' (default), 'directForm1', 'directForm1Transposed', 'directForm2'</td>
</tr>
</tbody>
</table>
| `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 comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: `filt2block(...,'subsystem','BlockName','Lowpass FIR','FrameBasedProcessing',false)`

### Destination — Destination for Simulink filter block

'current' (default) | 'new' | 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 \texttt{true} or \texttt{false}. The default is \texttt{true}.

Data Types: \texttt{logical}

**See Also**

designfilt | digitalFilter

**Introduced in R2013a**
findchangepts

Find abrupt changes in signal

Syntax

\[
\begin{align*}
\text{ipt} &= \text{findchangepts}(x) \\
\text{ipt} &= \text{findchangepts}(x,\text{Name,Value}) \\
[\text{ipt, residual}] &= \text{findchangepts}(\_\_\_)
\end{align*}
\]

Description

\text{ipt} = \text{findchangepts}(x) \text{ returns the index at which the mean of } x \text{ changes most significantly.}

- If \(x\) is a vector with \(N\) elements, then \text{findchangepts} partitions \(x\) into two regions, \(x(1:\text{ipt}-1)\) and \(x(\text{ipt}:N)\), 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 \text{findchangepts} partitions \(x\) into two regions, \(x(1:M,1:\text{ipt}-1)\) and \(x(1:M,\text{ipt}:N)\), returning the column index that minimizes the sum of the residual error of each region from its local \(M\)-dimensional mean.

\text{ipt} = \text{findchangepts}(x,\text{Name,Value}) \text{ specifies additional options using name-value pair arguments. Options include the number of changepoints to report and the statistic to measure instead of the mean. See “Changepoint Detection” on page 1-859 for more information.}

\[\text{ipt, residual}] = \text{findchangepts}(\_\_\_) \text{ also returns the residual error of the signal against the modeled changes, incorporating any of the previous specifications.}\]

\text{findchangepts}(\_\_\_) \text{ without output arguments plots the signal and any detected changepoints. See ““Statistic“” on page 1-0 for more information.}
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.

```matlab
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.

```matlab
[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.

```matlab
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.

```matlab
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.

```matlab
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)

q = 5×1
    132
    778
   1646
   2500
   3454

r = -4.4055e+03

Plot the signal and display the changepoints.

findchangepts(mtlb,'Statistic','rms','MaxNumChanges',numc)
To play the sound with a pause after each of the segments, uncomment the following lines.

% soundsc(1:q(1),Fs)
% for k = 1:length(q)-1
%     soundsc(mtlb(q(k):q(k+1)),Fs)
%     pause(1)
% end
% soundsc(q(end):length(mtlb),Fs)
Change of Mean, RMS Level, Standard Deviation, and Slope

Create a signal that consists of two sinusoids with varying amplitude and a linear trend.

\[
vc = \sin(2\pi \cdot (0:201)/17) \cdot \sin(2\pi \cdot (0:201)/19) \cdot \ldots \\
\left[ \sqrt{0:0.01:1} \ (1:-0.01:0)^2 \right] + (0:201)/401;
\]

Find the points where the signal mean changes most significantly. The 'Statistic' name-value pair is optional in this case. Specify a minimum residual error improvement of 1.

\[
\text{findchangepts}(vc, '\text{Statistic}', 'mean', '\text{MinThreshold}', 1)
\]

Find the points where the root-mean-square level of the signal changes the most. Specify a minimum residual error improvement of 6.
Find the points where the standard deviation of the signal changes most significantly. Specify a minimum residual error improvement of 10.

```matlab
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 \( k \)th 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.

```matlab
findchangepts(crv,'Statistic','linear','MaxNumChanges',19)
```
Generate and plot a three-dimensional Bézier curve with 20 random control points.

```matlab
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')
```

Number of changepoints = 19
Total residual error = 0.090782
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.

```matlab
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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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.

**MaxNumChanges — Maximum number of significant changes to return**

1 (default) | integer scalar

Maximum number of significant changes to return, specified as the comma-separated pair consisting of 'MaxNumChanges' and 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.

**Statistic — Type of change to detect**

Type of change to detect, specified as the comma-separated pair consisting of 'Statistic' and 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 the comma-separated pair consisting of 'MinDistance' and 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 the comma-separated pair consisting of 'MinThreshold' and 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 \(x_1, x_2, \ldots, x_N\), the function finds \(k\) such that

\[
J = \sum_{i=1}^{k-1} (x_i - \langle x \rangle_{1}^{k-1})^2 + \sum_{i=k}^{N} (x_i - \langle x \rangle_{k}^{N})^2
\]

\[
= \sum_{i=1}^{k-1} \left(x_i - \frac{1}{k-1} \sum_{r=1}^{k-1} x_r\right)^2 + \sum_{i=k}^{N} \left(x_i - \frac{1}{N-k+1} \sum_{r=k}^{N} x_r\right)^2
\]

\[
= (k - 1) \text{var}([x_1 \ldots x_{k-1}]) + (N - k + 1) \text{var}([x_k \ldots x_N]),
\]

is smallest. The mean and variance have the usual definitions. (The variance is normalized by the number of samples, without subtracting 1.)

This result can be generalized to incorporate other statistics. `findchangepts` finds \(k\) such that

\[
J(k) = \sum_{i=1}^{k-1} \Delta(x_i; \chi([x_1 \ldots x_{k-1}])) + \sum_{i=k}^{N} \Delta(x_i; \chi([x_k \ldots x_N]))
\]

is smallest, given the section empirical estimate \(\chi\) and the deviation measurement \(\Delta\). Minimizing the residual error is equivalent to maximizing the log likelihood. Given a normal distribution with mean \(\mu\) and variance \(\sigma^2\), the log-likelihood for \(N\) independent observations is

\[
\log \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i - \mu)^2}{2\sigma^2}} = -\frac{N}{2} (\log 2\pi + \log \sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^{N} (x_i - \mu)^2.
\]

- If 'Statistic' is specified as 'mean', then the variance is fixed and the function uses

\[
\sum_{i=m}^{n} \Delta(x_i; \chi([x_m \ldots x_n])) = \sum_{i=m}^{n} (x_i - \mu([x_m \ldots x_n]))^2
\]

\[
= (n - m + 1) \text{var}([x_m \ldots x_n]),
\]

as obtained previously.
- If 'Statistic' is specified as 'std', then the mean is fixed and the function uses
\[ \sum_{i=m}^{n} \Delta(x_i; \bar{x}([x_m \cdots x_n])) = (n - m + 1) \log \left( \frac{1}{n - m + 1} \sum_{i=m}^{n} \left( x_i - \frac{1}{n - m + 1} \sum_{r=m}^{n} x_r \right)^2 \right) \]

\[ = (n - m + 1) \log \left( \frac{1}{n - m + 1} \sum_{i=m}^{n} x_i^2 \right). \]

- If 'Statistic' is specified as 'rms', the total deviation is the same as for 'std' but with the mean set to zero:

\[ \sum_{i=m}^{n} \Delta(x_i; \bar{x}([x_m \cdots x_n])) = (n - m + 1) \log \left( \frac{1}{n - m + 1} \sum_{r=m}^{n} x_r^2 \right). \]

- If 'Statistic' is specified as 'linear', the function uses

\[ \sum_{i=m}^{n} \Delta(x_i; \bar{x}([x_m \cdots x_n])) = (n - m + 1) \text{var}([x_m \cdots x_n]) \]

\[ - \left( \frac{\sum_{i=m}^{n} (x_i - \mu([x_m \cdots x_n]))(i - \mu([m \cdots n]))}{(n - m + 1) \text{var}([m \cdots n])} \right)^2 \]

\[ \equiv S_{xx} - \frac{S_{xt}^2}{S_{tt}}. \]

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

\[ J(K) = \sum_{r=0}^{K-1} \sum_{i=k_r}^{k_r+1} \Delta(x_i; \bar{x}([x_{k_r} \cdots x_{k_r+1-1}])) + \beta K, \]

where \( k_0 \) and \( k_K \) are respectively the first and the last sample of the signal.

- The proportionality constant, denoted by \( \beta \) 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 less changes than the specified value.

To perform the minimization itself, findchangepts uses an exhaustive algorithm based 
on dynamic programming with early abandonment.

References


changepoints with a linear computational cost.” *Journal of the American 

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.

\[
X = [0 \ 0 \ 1 \ 2 \ 3 \ 0]; \\
Y = [0.02 \ 0.12 \ 1.08 \ 2.21 \ 2.95 \ -0.09]; \\
D = \text{finddelay}(X,Y)
\]

\( D = 0 \)

If \( Y \) is a periodic version of \( X \), the smallest possible delay is returned.

\[
X = [0 \ 1 \ 2 \ 3]; \\
Y = [1 \ 2 \ 3 \ 0 \ 0 \ 0 \ 0 \ 1 \ 2 \ 3 \ 0 \ 0]; \\
D = \text{finddelay}(X,Y)
\]

\( D = -1 \)

**X Is a Vector, Y Is a Matrix, and maxlag Is a Scalar**

\( \text{maxlag} \) is specified as a scalar (same maximum window sizes).

\[
X = [0 \ 1 \ 2]; \\
Y = [0 \ 1 \ 0 \ 0; \\
    1 \ 2 \ 0 \ 0; \\
    2 \ 0 \ 1 \ 0; \\
    0 \ 0 \ 2 \ 1]; \\
\text{maxlag} = 3; \\
D = \text{finddelay}(X,Y,\text{maxlag})
\]

\( D = 1\times4 \)

\[
\begin{bmatrix}
0 \\
-1 \\
1 \\
1
\end{bmatrix}
\]

**X and Y Are Matrices, and maxlag Is Not Specified**

Specify \( X \) and \( Y \) of the same size. \( \text{finddelay} \) works column-by-column.

\[
X = [0 \ 1 \ 0 \ 0; \\
    1 \ 2 \ 0 \ 0; \\
    2 \ 0 \ 1 \ 0; \\
    1 \ 2 \ 3 \ 0 \ 0 \ 0 \ 0 \ 1 \ 2 \ 3 \ 0 \ 0]
\]
\begin{verbatim}
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 = 1x4
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 = 1x4
1 2 -1 0
\end{verbatim}

**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.

\begin{verbatim}
X = [1 3 2 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]';
\end{verbatim}

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 = 1x2
   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 = 1x2
   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 = 1x2
   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

\[\max(\text{length}(x), \text{length}(y)) - 1|\max(\text{size}(x,1), \text{size}(y,1)) - 1|\]
\[\max(\text{length}(x), \text{size}(y,1)) - 1|\max(\text{size}(x,1), \text{length}(y)) - 1|\]
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 \( x_m \):

Each column of \( x_m \) corresponds to a single channel, each one containing the samples 1 and 0.

\[
x_m = [1 1 1 1; \ 0 0 0 0]
\]

**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. Non-Inf signal endpoints are excluded. If a peak is flat, the function returns only the point with the lowest index.

[pks,locs] = findpeaks(data) additionally returns the indices at which the peaks occur.

[pks,locs,w,p] = findpeaks(data) additionally returns the widths of the peaks as the vector w and the prominences of the peaks as the vector p.

[___] = findpeaks(data,x) specifies x as the location vector and returns any of the output arguments from previous syntaxes. locs and w are expressed in terms of x.

[___] = findpeaks(data,Fs) specifies the sample rate, Fs, of the data. The first sample of data is assumed to have been taken at time zero. locs and w are converted to time units.
[___] = findpeaks(___,Name,Value) specifies options using name-value pair arguments in addition to any of the input arguments in previous syntaxes.

findpeaks(___) without output arguments plots the signal and overlays the peak values.

Examples

Find Peaks in a Vector

Define a vector with three peaks and plot it.

data = [25 8 15 5 6 10 10 3 1 20 7];
plot(data)
Find the local maxima. The peaks are output in order of occurrence. The first sample is not included despite being the maximum. For the flat peak, the function returns only the point with lowest index.

\[ \text{pks} = \text{findpeaks}(\text{data}) \]

\[ \text{pks} = 1\times3 \]

\[ 15 \quad 10 \quad 20 \]

Use `findpeaks` without output arguments to display the peaks.

\[ \text{findpeaks}(\text{data}) \]
**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.

```matlab
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.

```matlab
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.

```matlab
[pks,locs,widths,proms] = findpeaks(PeakSig,x);
widths
```

```matlab
disp(widths)
disp(
    0.0154    0.0431    0.0377    0.0625    0.0274    0.0409
)
```

```matlab
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.

```matlab
findpeaks(avSpots,year,'MinPeakDistance',6)
```
Use the peak locations returned by `findpeaks` to compute the mean interval between maxima.

```matlab
[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`.

```matlab
ty = datetime(year,3,20);
```
Compute the mean sunspot cycle using `datetime` functionality.

```matlab
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.

```matlab
TT = timetable(years(year),avSpots);
plot(TT.Time,TT.Variables)
```
entries = TT(end-4:end,:)

entries=5x2 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.

```matlab
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.

```matlab
findpeaks(select,Fs,'MinPeakDistance',0.005)
```

![Graph showing peaks selected from a signal](image)
Find the peaks that have an amplitude of at least 1 V.

```matlab
findpeaks(select,Fs,'MinPeakHeight',1)
```

Find the peaks that are at least 1 V higher than their neighboring samples.

```matlab
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.

```matlab
[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.

\[
x = \text{linspace}(0,1,1000);
\]

\[
\text{Pos} = [1\ 2\ 3\ 5\ 7\ 8]/10;
\]

\[
\text{Hgt} = [4\ 4\ 2\ 2\ 2\ 3];
\]

\[
\text{Wdt} = [3\ 8\ 4\ 3\ 4\ 6]/100;
\]
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.

```matlab
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 comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'SortStr','descend','NPeaks',3 finds the three tallest peaks of the signal.

**NPeaks — Maximum number of peaks**
positive integer scalar

Maximum number of peaks to return, specified as the comma-separated pair consisting of 'NPeaks' and a positive integer scalar. findpeaks operates from the first element of the input data and terminates when the number of peaks reaches the value of 'NPeaks'.

Data Types: double | single

**SortStr — Peak sorting**

'none' (default) | 'ascend' | 'descend'

Peak sorting, specified as the comma-separated pair consisting of 'SortStr' and one of these values:

- 'none' returns the peaks in the order in which they occur in the input data.
- 'ascend' returns the peaks in ascending or increasing order, from the smallest to the largest value.
• 'descend' returns the peaks in descending order, from the largest to the smallest value.

**MinPeakHeight — Minimum peak height**

-Inf (default) | real scalar

Minimum peak height, specified as the comma-separated pair consisting of 'MinPeakHeight' and a real scalar. Use this argument to have findpeaks return only those peaks higher than 'MinPeakHeight'. Specifying a minimum peak height can reduce processing time.

Data Types: double | single

**MinPeakProminence — Minimum peak prominence**

0 (default) | real scalar

Minimum peak prominence, specified as the comma-separated pair consisting of 'MinPeakProminence' and a real scalar. Use this argument to have findpeaks return only those peaks that have a relative importance of at least 'MinPeakProminence'. See “Prominence” on page 1-900 for more information.

Data Types: double | single

**Threshold — Minimum height difference**

0 (default) | nonnegative real scalar

Minimum height difference between a peak and its neighbors, specified as the comma-separated pair consisting of 'Threshold' and a nonnegative real scalar. Use this argument to have findpeaks return only those peaks that exceed their immediate neighboring values by at least the value of 'Threshold'.

Data Types: double | single

**MinPeakDistance — Minimum peak separation**

0 (default) | positive real scalar

Minimum peak separation, specified as the comma-separated pair consisting of 'MinPeakDistance' and a positive real scalar. When you specify a value for 'MinPeakDistance', the algorithm chooses the tallest peak in the signal and ignores all peaks within 'MinPeakDistance' of it. The function then repeats the procedure for the tallest remaining peak and iterates until it runs out of peaks to consider.
• If you specify a location vector, \( x \), then 'MinPeakDistance' must be expressed in terms of \( x \). If \( x \) 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 \texttt{findpeaks} ignore small peaks that occur in the neighborhood of a larger peak.

Data Types: double | single | duration

\textbf{WidthReference — Reference height for width measurements}

'halfprom' (default) | 'halfheight'

Reference height for width measurements, specified as the comma-separated pair consisting of 'WidthReference' and either 'halfprom' or 'halfheight'. \texttt{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 1-900 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.

\textbf{MinPeakWidth — Minimum peak width}

0 (default) | positive real scalar

Minimum peak width, specified as the comma-separated pair consisting of 'MinPeakWidth' and a positive real scalar. Use this argument to select only those peaks that have widths of at least 'MinPeakWidth'.

• If you specify a location vector, \( x \), then 'MinPeakWidth' must be expressed in terms of \( x \). If \( x \) 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 the comma-separated pair consisting of 'MaxPeakWidth' and a positive real scalar. Use this argument to select only those peaks that have widths of at most 'MaxPeakWidth'.

• If you specify a location vector, x, then 'MaxPeakWidth' must be expressed in terms of x. If 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 the comma-separated pair consisting of 'Annotate' and one of these values:

• 'peaks' plots the signal and annotates the location and value of every peak.
• 'extents' plots the signal and annotates the location, value, width, and prominence of every peak.

This argument is ignored if you call findpeaks with output arguments.

Output Arguments

pks — Local maxima
vector
Local maxima, returned as a vector of signal values. If there are no local maxima, then \texttt{pks} is empty.

\textbf{\texttt{locs} — Peak locations}
vector

Peak locations, returned as a vector.

- If you specify a location vector, \texttt{x}, then \texttt{locs} contains the values of \texttt{x} at the peak indices.
- If you specify a sample rate, \texttt{Fs}, then \texttt{locs} is a vector of time instants.
- If you specify neither \texttt{x} nor \texttt{Fs}, then \texttt{locs} is a vector of integer indices.

\textbf{\texttt{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 \texttt{WidthReference}. The points themselves are found by linear interpolation.

- If you specify a location vector, \texttt{x}, then the widths are expressed in terms of \texttt{x}.
- If you specify a sample rate, \texttt{Fs}, then the widths are expressed in units of time.
- If you specify neither \texttt{x} nor \texttt{Fs}, then the widths are expressed in units of samples.

\textbf{\texttt{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 1-900 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:
<table>
<thead>
<tr>
<th>Peak Number</th>
<th>Left Interval Lies Between Peak and</th>
<th>Right Interval Lies Between Peak and</th>
<th>Lowest Point on the Left Interval</th>
<th>Lowest Point on the Right Interval</th>
<th>Reference Level (Highest Minimum)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Left end</td>
<td>Crossing due to peak 2</td>
<td>Left endpoint</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>Left end</td>
<td>Right end</td>
<td>Left endpoint</td>
<td>h</td>
<td>Left endpoint</td>
</tr>
<tr>
<td>3</td>
<td>Crossing due to peak 2</td>
<td>Crossing due to peak 4</td>
<td>b</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>Crossing due to peak 2</td>
<td>Crossing due to peak 6</td>
<td>b</td>
<td>d</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>Crossing due to peak 4</td>
<td>Crossing due to peak 6</td>
<td>d</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>Crossing due to peak 2</td>
<td>Right end</td>
<td>d</td>
<td>h</td>
<td>d</td>
</tr>
<tr>
<td>7</td>
<td>Crossing due to peak 6</td>
<td>Crossing due to peak 8</td>
<td>f</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>Peak Number</td>
<td>Left Interval Lies Between Peak and</td>
<td>Right Interval Lies Between Peak and</td>
<td>Lowest Point on the Left Interval</td>
<td>Lowest Point on the Right Interval</td>
<td>Reference Level (Highest Minimum)</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------</td>
<td>-------------------------------------</td>
<td>----------------------------------</td>
<td>-----------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>Crossing due to peak 6</td>
<td>Right end</td>
<td>f</td>
<td>h</td>
<td>f</td>
</tr>
<tr>
<td>9</td>
<td>Crossing due to peak 8</td>
<td>Crossing due to right endpoint</td>
<td>h</td>
<td>i</td>
<td>i</td>
</tr>
</tbody>
</table>

**Extended Capabilities**

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

**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

\[ [\text{istart}, \text{istop}, \text{dist}] = \text{findsignal}(\text{data}, \text{signal}) \]
\[ [\text{istart}, \text{istop}, \text{dist}] = \text{findsignal}(\text{data}, \text{signal}, \text{Name}, \text{Value}) \]

\text{findsignal}(\_\_\_)

Description

\[ [\text{istart}, \text{istop}, \text{dist}] = \text{findsignal}(\text{data}, \text{signal}) \] returns the start and stop indices of a segment of the data array, \text{data}, that best matches the search array, \text{signal}. The best-matching segment is such that \text{dist}, the squared Euclidean distance between the segment and the search array, is smallest. If \text{data} and \text{signal} are matrices, then \text{findsignal} finds the start and end columns of the region of \text{data} that best matches \text{signal}. In that case, \text{data} and \text{signal} must have the same number of rows.

\[ [\text{istart}, \text{istop}, \text{dist}] = \text{findsignal}(\text{data}, \text{signal}, \text{Name}, \text{Value}) \] 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.

\text{findsignal}(\_\_\_\_\_\_\_\_\_\_\_\_) without output arguments plots \text{data} and highlights any identified instances of \text{signal}.

- If the arrays are real vectors, the function displays \text{data} as a function of sample number.
- If the arrays are complex vectors, the function displays \text{data} on an Argand diagram.
- If the arrays are real matrices, the function uses \text{imagesc} to display \text{signal} on a subplot and \text{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.

```matlab
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.

```matlab
findsignal(dt,signal,'TimeAlignment','dtw','Metric','absolute')
```
Add two more outlying sections to the data set.

\[
\begin{align*}
\text{dt}(t>0.1\&t<0.11) &= 2.1; \\
\text{dt}(t>0.11\&t<0.12) &= -2.1;
\end{align*}
\]

findsignal(dt,signal,'TimeAlignment','dtw','Metric','absolute')
Find the two data segments closest to the signal.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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'))

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'))

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.

\[ \text{sg} = \frac{\text{randn}(1,2\times\text{lr})}{2}; \]

\[ \text{findsignal}(t,\text{sg}) \]
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.

```matlab
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.

```matlab
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.

\[ sg = \text{randn}(1,2*lr)*3-4; \]

\[ \text{findsignal}(t,sg) \]
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.

\[
\text{corr} = [c(M) \ c(A) \ c(T) \ c(L) \ c(A) \ c(B)];
\]

\[
\text{sgn} = c(A);
\]

\[
[\text{ist}, \text{ind}, \text{dst}] = \text{findsignal(corr, sgn)};
\]

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
Repeat the computation using the built-in functionality of \texttt{findsignal}. Divide by the local mean to normalize the data and the signal. Use the symmetric Kullback-Leibler metric.

\begin{verbatim}
findsignal(corr,sgn,'TimeAlignment','dtw', ...
    'Normalization','power','Metric','symmkl','Annotate','all')
\end{verbatim}
**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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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 X and Y are both K-dimensional signals, then Metric prescribes \( d_{mn}(X,Y) \), the distance between the \( m \)th sample of X and the \( n \)th sample of Y. See “Dynamic Time Warping” on page 1-540 for more information about \( d_{mn}(X,Y) \).

- 'squared' — Square of the Euclidean metric, consisting of the sum of squared differences:

\[
d_{mn}(X,Y) = \sum_{k=1}^{K} (x_{k,m} - y_{k,n})^2
\]

- 'euclidean' — Root sum of squared differences, also known as the Euclidean or \( \ell_2 \) metric:

\[
d_{mn}(X,Y) = \sqrt{\sum_{k=1}^{K} (x_{k,m} - y_{k,n})^2}
\]

- 'absolute' — Sum of absolute differences, also known as the Manhattan, city block, taxicab, or \( \ell_1 \) metric:

\[
d_{mn}(X,Y) = \sum_{k=1}^{K} |x_{k,m} - y_{k,n}|
\]

- 'symmkl' — Symmetric Kullback-Leibler metric. This metric is valid only for real and positive X and Y:

\[
d_{mn}(X,Y) = \sum_{k=1}^{K} (x_{k,m} - y_{k,n}) \log(x_{k,m} - \log(y_{k,n})
\]

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.

See Also

alignsignals, dtw, edr, finddelay, findpeaks, strfind, xcorr

Topics
“Finding a Signal in Data”
“Find a Signal in a Measurement”

Introduced in R2016b
**fir1**

Window-based FIR filter design

**Syntax**

\[
b = \text{fir1}(n,Wn) \\
b = \text{fir1}(n,Wn,ftype) \\
b = \text{fir1}(\_,\_,\text{window}) \\
b = \text{fir1}(\_,\_,\text{scaleopt})
\]

**Description**

\[
b = \text{fir1}(n,Wn) \text{ uses a Hamming window to design an } n\text{-order lowpass, bandpass, or multiband FIR filter with linear phase. The filter type depends on the number of elements of } Wn.
\]

\[
b = \text{fir1}(n,Wn,ftype) \text{ designs a lowpass, highpass, bandpass, bandstop, or multiband filter, depending on the value of } ftype \text{ and the number of elements of } Wn.
\]

\[
b = \text{fir1}(\_,\_,\text{window}) \text{ designs the filter using the vector specified in } \text{window} \text{ and any of the arguments from previous syntaxes.}
\]

\[
b = \text{fir1}(\_,\_,\text{scaleopt}) \text{ additionally specifies whether or not the magnitude response of the filter is normalized.}
\]

**Note:** Use fir2 for windowed filters with arbitrary frequency response.

**Examples**

**FIR Bandpass Filter**

Design a 48th-order FIR bandpass filter with passband \(0.35\pi \leq \omega \leq 0.65\pi\) rad/sample. Visualize its magnitude and phase responses.
b = fir1(48, [0.35 0.65]);
freqz(b, 1, 512)

**FIR Highpass Filter**

Load chirp.mat. The file contains a signal, y, that has most of its power above 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)

subplot(2,1,2)
plot(t,y)
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.

```matlab
blo = fir1(34, 0.48, chebwin(35, 30));
```
```matlab
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 $0.4\pi$ rad/sample and between $0.6\pi$ and $0.9\pi$ rad/sample. Call it $bM$.

\[
\text{ord} = 46;
\]
\[
\text{low} = 0.4;
\]
\[
\text{bnd} = [0.6 \ 0.9];
\]
\[
bM = \text{fir1}(	ext{ord},[\text{low} \ \text{bnd}]);
\]

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.

\[
bW = \text{fir1}(	ext{ord},[\text{low} \ \text{bnd}],'DC-1');
\]
\[
hfvt = \text{fvtool}(bM,1,bW,1);
\]
\[
\text{legend(hfvt,'bM','bW')}
\]
Redesign bM using a Hann window. (The 'DC-0' is optional.) Compare the magnitude responses of the Hamming and Hann designs.

```matlab
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.

```matlab
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 \( \pi \) 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 \([w1 \ w2 \ldots \ wn]\), where \( w1 < w2 < \ldots < wn \), then `fir1` returns an nth-order multiband filter with bands \( 0 < \omega < w1, w1 < \omega < w2, \ldots, wn < \omega < 1 \).

**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 \texttt{fir1} uses a Hamming window. For a list of available windows, see “Windows”.

\texttt{fir1} does not automatically increase the length of window if you attempt to design a highpass or bandstop filter of odd order.

Example: \texttt{kaiser(n+1,0.5)} specifies a Kaiser window with shape parameter 0.5 to use with a filter of order \( n \).

Example: \texttt{hamming(n+1)} is equivalent to leaving the window unspecified.

Data Types: \texttt{double}

\texttt{scaleopt — Normalization option}

'\texttt{scale}' (default) | '\texttt{noscale}'

Normalization option, specified as either '\texttt{scale}' or '\texttt{noscale}'.

- '\texttt{scale}' normalizes the coefficients so that the magnitude response of the filter at the center of the passband is 1 (0 dB).
- '\texttt{noscale}' does not normalize the coefficients.

\textbf{Output Arguments}

\textbf{b — Filter coefficients}

\texttt{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 \( z \):

\[ B(z) = b(1) + b(2)z + \cdots + b(n+1)z^n. \]

\textbf{Algorithms}

\texttt{fir1} uses a least-squares approximation to compute the filter coefficients and then smooths the impulse response with \texttt{window}. 
References


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**

```matlab
b = fir2(n,f,m)
b = fir2(n,f,m,npt,lap)
b = fir2(____,window)
```

**Description**

`b = fir2(n,f,m)` returns an nth-order FIR filter with frequency-magnitude characteristics specified in the vectors `f` and `m`. The function linearly interpolates the desired frequency response onto a dense grid and then uses the inverse Fourier transform and a Hamming window to obtain the filter coefficients.

`b = fir2(n,f,m,npt,lap)` specifies `npt`, the number of points in the interpolation grid, and `lap`, the length of the region that `fir2` inserts around duplicate frequency points which specify steps in the frequency response.

`b = fir2(____,window)` specifies a window vector to use in the design in addition to any input arguments from previous syntaxes.

**Note:** Use `fir1` for window-based standard lowpass, bandpass, highpass, bandstop, and multiband configurations.

**Examples**

**Attenuation of Low Frequencies**

Load the MAT-file `chirp`. The file contains a signal, `y`, sampled at a frequency $F_s = 8192$ Hz. `y` has most of its power above $F_s/4$, or half the Nyquist frequency. Add random noise to the signal.
load chirp
y = y + 0.25*(rand(size(y))-0.5);

Design a 34th-order FIR highpass filter to attenuate the components of the signal below $F_s/4$. Specify a cutoff frequency of 0.48. Visualize the frequency response of the filter.

```matlab
f = [0 0.48 0.48 1];
mhi = [0 0 1 1];
bhi = fir2(34,f,mhi);

defz(bhi,1)
```

Filter the chirp signal. Plot the signal before and after filtering.
outhi = filter(bhi,1,y);
t = (0:length(y)-1)/Fs;

subplot(2,1,1)
plot(t,y)
title('Original Signal')
ylim([[-1.2 1.2]])

subplot(2,1,2)
plot(t,outhi)
title('Higpass Filtered Signal')
xlabel('Time (s)')
ylabel('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.

```matlab
mlo = [1 1 0 0];
blo = fir2(34,f,mlo);
outlo = filter(blo,1,y);

subplot(2,1,1)
plot(t,y)
title('Original Signal')
ylim([-1.2 1.2])

subplot(2,1,2)
plot(t,outlo)
title('Lowpass Filtered Signal')
xlabel('Time (s)')
ylim([-1.2 1.2])
```
FIR Lowpass Filter

Design a 30th-order lowpass filter with a normalized cutoff frequency of $0.6\pi$ rad/sample. Plot the ideal frequency response overlaid with the actual frequency response.

```matlab
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))
```
Redesign the filter using a 64-point interpolation grid.

\begin{verbatim}
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)
\end{verbatim}
Redesign the filter using the 64-point interpolation grid and a 13-point interval around the cutoff frequency.

```matlab
b3 = fir2(30,f,m,64,13);
h3 = freqz(b3,1);

plot(w/pi,abs(h3))
lg{4} = 'lap = 13';
legend(lgs)
```
**Arbitrary Magnitude Filter**

Design an FIR filter with the following frequency response:

- A sinusoid between 0 and $0.18\pi$ rad/sample.

\[
F_1 = 0:0.01:0.18;
A_1 = 0.5 + \sin(2\pi \times 7.5 \times F_1)/4;
\]

- A piecewise linear section between $0.2\pi$ rad/sample and $0.78\pi$ rad/sample.
F2 = [0.2 0.38 0.4 0.55 0.562 0.585 0.6 0.78];
A2 = [0.5 2.3 1 1 -0.2 -0.2 1 1];

• A quadratic section between 0.79π 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

\[ b \rightarrow \text{Filter coefficients} \]
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

\texttt{fir2} uses frequency sampling to design filters. The function interpolates the desired frequency response linearly onto a dense, evenly spaced grid of length \( npt \). \texttt{fir2} also sets up regions of \texttt{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 \texttt{window}.

References


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 | firpm | hamming | maxflat | yulewalk

**Introduced before R2006a**
**fircls**

Constrained-least-squares FIR multiband filter design

**Syntax**

\[ b = \text{fircls}(n,f,\text{amp},\text{up},\text{lo}) \]

\[ \text{fircls}(n,f,\text{amp},\text{up},\text{lo},'\text{design\_flag}') \]

**Description**

\[ b = \text{fircls}(n,f,\text{amp},\text{up},\text{lo}) \] generates a length \( n+1 \) linear phase FIR filter \( b \). The frequency-magnitude characteristics of this filter match those given by vectors \( f \) and \( \text{amp} \):

- \( f \) is a vector of transition frequencies in the range from 0 to 1, where 1 corresponds to the Nyquist frequency. The first point of \( f \) must be 0 and the last point 1. The frequency points must be in increasing order.
- \( \text{amp} \) is a vector describing the piecewise-constant desired amplitude of the frequency response. The length of \( \text{amp} \) is equal to the number of bands in the response and should be equal to \( \text{length}(f)-1 \).
- \( \text{up} \) and \( \text{lo} \) are vectors with the same length as \( \text{amp} \). They define the upper and lower bounds for the frequency response in each band.

**fircls** always uses an even filter order for configurations with a passband at the Nyquist frequency (that is, highpass and bandstop filters). This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued \( n \), **fircls** increments it by 1.

\[ \text{fircls}(n,f,\text{amp},\text{up},\text{lo},'\text{design\_flag}') \] enables you to monitor the filter design, where \( '\text{design\_flag}' \) can be

- \( '\text{trace}' \), for a textual display of the design error at each iteration step.
- \( '\text{plots}' \), for a collection of plots showing the filter's full-band magnitude response and a zoomed view of the magnitude response in each sub-band. All plots are updated at each iteration step. The O's on the plot are the estimated extremals of the new iteration and the X's are the estimated extremals of the previous iteration, where the
extremals are the peaks (maximum and minimum) of the filter ripples. Only ripples that have a corresponding O and X are made equal.

- 'both', for both the textual display and plots.

**Note** Normally, the lower value in the stopband will be specified as negative. By setting \( lo \) equal to 0 in the stopbands, a nonnegative frequency response amplitude can be obtained. Such filters can be spectrally factored to obtain minimum phase filters.

## Examples

### Constrained Least-Squares Lowpass Filter

Design a 150th-order lowpass filter with a normalized cutoff frequency of \( 0.4\pi \) rad/sample. Specify a maximum absolute error of 0.02 in the passband and 0.01 in the stopband. Display plots of the bands.

```plaintext
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)
```
Algorithms

`fircls` uses an iterative least-squares algorithm to obtain an equiripple response. The algorithm is a multiple exchange algorithm that uses Lagrange multipliers and Kuhn-Tucker conditions on each iteration.

References


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

\[
b = \text{fircls1}(n, wo, dp, ds) \\
b = \text{fircls1}(n, wo, dp, ds, 'high') \\
b = \text{fircls1}(n, wo, dp, ds, wt) \\
b = \text{fircls1}(n, wo, dp, ds, wt, 'high') \\
b = \text{fircls1}(n, wo, dp, ds, wp, ws, k) \\
b = \text{fircls1}(n, wo, dp, ds, wp, ws, k, 'high') \\
b = \text{fircls1}(n, wo, dp, ds, ... , 'design_flag')
\]

Description

\[
b = \text{fircls1}(n, wo, dp, ds) \text{ generates a lowpass FIR filter } b, \text{ where } n+1 \text{ is the filter length, } wo \text{ is the normalized cutoff frequency in the range between 0 and 1 (where 1 corresponds to the Nyquist frequency), } dp \text{ is the maximum passband deviation from 1 (passband ripple), and } ds \text{ is the maximum stopband deviation from 0 (stopband ripple).}
\]

\[
b = \text{fircls1}(n, wo, dp, ds, 'high') \text{ generates a highpass FIR filter } b. \text{ fircls1 always uses an even filter order for the highpass configuration. This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued } n, \text{ fircls1 increments it by 1.}
\]

\[
b = \text{fircls1}(n, wo, dp, ds, wt) \text{ and }
\]

\[
b = \text{fircls1}(n, wo, dp, ds, wt, 'high') \text{ specifies a frequency } wt \text{ above which (for } wt > wo) \text{ or below which (for } wt < wo) \text{ the filter is guaranteed to meet the given band criterion. This will help you design a filter that meets a passband or stopband edge requirement. There are four cases:}
\]

- **Lowpass:**
  - \( 0 < wt < wo < 1 \): the amplitude of the filter is within \( dp \) of 1 over the frequency range \( 0 < \omega < wt \).
• \( 0 < \omega_0 < w_t < 1 \): the amplitude of the filter is within \( ds \) of 0 over the frequency range \( w_t < \omega < 1 \).

• Highpass:

  • \( 0 < w_t < \omega_0 < 1 \): the amplitude of the filter is within \( ds \) of 0 over the frequency range \( 0 < \omega < w_t \).
  
  • \( 0 < \omega_0 < w_t < 1 \): the amplitude of the filter is within \( dp \) of 1 over the frequency range \( w_t < \omega < 1 \).

\[
b = \text{fircls1}(n,\omega_0,dp,ds,wp,ws,k)
\]
generates a lowpass FIR filter \( b \) with a weighted function, where \( n+1 \) is the filter length, \( \omega_0 \) 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 \( wp < \omega_0 < ws \). \( k \) is the ratio (passband L2 error)/(stopband L2 error)

\[
k = \frac{\int_{0}^{wp}|A(\omega) - D(\omega)|^2d\omega}{\int_{\omega_2}^{\pi}|A(\omega) - D(\omega)|^2d\omega}
\]

\[
b = \text{fircls1}(n,\omega_0,dp,ds,wp,ws,k,'\text{high}')</n\)
generates a highpass FIR filter \( b \) with a weighted function, where \( ws < \omega_0 < wp \).

\[
b = \text{fircls1}(n,\omega_0,dp,ds,\ldots,'\text{design_flag}')
\]
enables you to monitor the filter design, where 'design_flag' can be

• 'trace', for a textual display of the design table used in the design
• 'plots', for plots of the filter's magnitude, group delay, and zeros and poles. All plots are updated at each iteration step. The O's on the plot are the estimated extremals of the new iteration and the X's are the estimated extremals of the previous iteration, where the extremals are the peaks (maximum and minimum) of the filter ripples. Only ripples that have a corresponding O and X are made equal.
• 'both', for both the textual display and plots

**Note** In the design of very narrow band filters with small \( dp \) 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.

\[
\begin{align*}
n &= 55; \\
wo &= 0.3; \\
dp &= 0.02; \\
ds &= 0.008; \\
b &= \text{fircls1}(n, wo, dp, ds, 'both'); \\
\end{align*}
\]

\[
\begin{align*}
\text{Bound Violation} &= 0.0870385343920 \\
\text{Bound Violation} &= 0.0149343456540 \\
\text{Bound Violation} &= 0.0056513587932 \\
\text{Bound Violation} &= 0.0001056264205 \\
\text{Bound Violation} &= 0.0000967624352 \\
\text{Bound Violation} &= 0.000000226538
\end{align*}
\]
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

drclsl 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


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(n,f,a,'ftype')
b = firls(n,f,a,w,'ftype')

Description

firls designs a linear-phase FIR filter that minimizes the weighted, integrated squared error between an ideal piecewise linear function and the magnitude response of the filter over a set of desired frequency bands.

b = firls(n,f,a) returns row vector b containing the n+1 coefficients of the order n FIR filter whose frequency-amplitude characteristics approximately match those given by vectors f and a. The output filter coefficients, or “taps,” in b obey the symmetry relation.

\[ b(k) = b(n + 2 - k), \quad k = 1, \ldots, n + 1 \]

These are type I (n odd) and type II (n even) linear-phase filters. Vectors f and a specify the frequency-amplitude characteristics of the filter:

- f is a vector of pairs of frequency points, specified in the range between 0 and 1, where 1 corresponds to the Nyquist frequency. The frequencies must be in increasing order. Duplicate frequency points are allowed and, in fact, can be used to design a filter exactly the same as those returned by the fir1 and fir2 functions with a rectangular (rectwin) window.
- a is a vector containing the desired amplitude at the points specified in f.

The desired amplitude function at frequencies between pairs of points \((f(k), f(k+1))\) for \(k\) odd is the line segment connecting the points \((f(k), a(k))\) and \((f(k+1), a(k+1))\).
The desired amplitude function at frequencies between pairs of points \( f(k), f(k+1) \) for \( k \) even is unspecified. These are transition or “don’t care” regions.

- \( f \) and \( a \) are the same length. This length must be an even number.

\texttt{firls} always uses an even filter order for configurations with a passband at the Nyquist frequency. This is because for odd orders, the frequency response at the Nyquist frequency is necessarily 0. If you specify an odd-valued \( n \), \texttt{firls} increments it by 1.

The figure below illustrates the relationship between the \( f \) and \( a \) vectors in defining a desired amplitude response.
b = firls(n,f,a,w) uses the weights in vector w to weight the fit in each frequency band. The length of w is half the length of f and a, so there is exactly one weight per band.

b = firls(n,f,a,'ftype') and

b = firls(n,f,a,w,'ftype') specify a filter type, where 'ftype' is:

• 'hilbert' for linear-phase filters with odd symmetry (type III and type IV). The output coefficients in b obey the relation

\[ b(k) = -b(n + 2 - k), \quad k = 1, \ldots, n + 1. \]

This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.

• 'differentiator' for type III and type IV filters, using a special weighting technique. For nonzero amplitude bands, the integrated squared error has a weight of \((1/f)^2\) so that the error at low frequencies is much smaller than at high frequencies. For FIR differentiators, which have an amplitude characteristic proportional to frequency, the filters minimize the relative integrated squared error (the integral of the square of the ratio of the error to the desired amplitude).

**Examples**

**Filter with Transition Band**

Design an FIR lowpass filter of order 255 with transition band between 0.25\(\pi\) and 0.3\(\pi\). Use fvtool to display the magnitude and phase responses of the filter.

```matlab
b = firls(255,[0 0.25 0.3 1],[1 1 0 0]);
fvtool(b,1,'OverlaidAnalysis','phase')
```
Design of a Differentiator

An ideal differentiator has a frequency response given by $D(\omega) = j\omega$. Design a differentiator of order 30 that attenuates frequencies above 0.9$\pi$. Include a factor of $\pi$ in the amplitude because the frequencies are normalized by $\pi$. Display the zero-phase response of the filter.

```matlab
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.

\[ F = [0 \ 0.3 \ 0.4 \ 0.6 \ 0.7 \ 0.9] \]
\[ A = [0 \ 1.0 \ 0.0 \ 0.0 \ 0.5 \ 0.5] \]
\[ b = 	ext{fir}ls(24,F,A,'hilbert'); \]

Plot the desired and actual frequency responses.

\[ [H,f] = 	ext{freqz}(b,1,512,2); \]
\[ 	ext{plot}(f,\text{abs}(H)) \]
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 $0.45\pi$ rad/sample. The stopband ranges from $0.55\pi$ 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.

```matlab
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.

```matlab
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.

```matlab
b = firls(18,[0 0.45 0.55 1],[1 1 0 0],[1 1]);
```

Visualize the magnitude responses of the three filters.

```matlab
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]');
```


**Diagnostics**

One of the following diagnostic messages is displayed when an incorrect argument is used:

- F must be even length.
- F and A must be equal lengths.
- Requires symmetry to be 'hilbert' or 'differentiator'.
- Requires one weight per band.
- Frequencies in F must be nondecreasing.
- Frequencies in F must be in range [0,1].
A more serious warning message is

Warning: Matrix is close to singular or badly scaled.

This tends to happen when the product of the filter length and transition width grows large. In this case, the filter coefficients b might not represent the desired filter. You can check the filter by looking at its frequency response.

**Algorithms**

Reference [1] describes the theoretical approach behind firls. The function solves a system of linear equations involving an inner product matrix of size roughly n/2 using the MATLAB \ operator.

This function designs type I, II, III, and IV linear-phase filters. Type I and II are the defaults for n even and odd respectively, while the 'hilbert' and 'differentiator' flags produce type III (n even) and IV (n odd) filters. The various filter types have different symmetries and constraints on their frequency responses (see [2] for details).

<table>
<thead>
<tr>
<th>Linear Phase Filter Type</th>
<th>Filter Order</th>
<th>Symmetry of Coefficients</th>
<th>Response H(f), f = 0</th>
<th>Response H(f), f = 1 (Nyquist)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type I</td>
<td>Even</td>
<td>$b(k) = b(n + 2 - k)$, $k = 1, ..., n + 1$</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>Type II</td>
<td>Odd</td>
<td>$b(k) = b(n + 2 - k)$, $k = 1, ..., n + 1$</td>
<td>No restriction</td>
<td>$H(1) = 0$</td>
</tr>
<tr>
<td>Type III</td>
<td>Even</td>
<td>$b(k) = -b(n + 2 - k)$, $k = 1, ..., n$</td>
<td>$H(0) = 0$</td>
<td>$H(1) = 0$</td>
</tr>
<tr>
<td>Type IV</td>
<td>Odd</td>
<td>$b(k) = -b(n + 2 - k)$, $k = 1, ..., n$</td>
<td>$H(0) = 0$</td>
<td>No restriction</td>
</tr>
</tbody>
</table>

**References**


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**

\[ b = \text{firpm}(n,f,a) \]
\[ b = \text{firpm}(n,f,a,w) \]
\[ b = \text{firpm}(n,f,a,ftype) \]
\[ b = \text{firpm}(n,f,a,lgrid) \]
\[ [b,err] = \text{firpm}(\_\_\_\_) \]
\[ [b,err,res] = \text{firpm}(\_\_\_\_) \]
\[ b = \text{firpm}(n,f,fresp,w) \]
\[ b = \text{firpm}(n,f,fresp,w,ftype) \]

**Description**

\[ b = \text{firpm}(n,f,a) \] 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 \( a \).

\[ b = \text{firpm}(n,f,a,w) \] uses \( w \) to weigh the frequency bins.

\[ b = \text{firpm}(n,f,a,ftype) \] uses a filter type specified by 'ftype'.

\[ b = \text{firpm}(n,f,a,lgrid) \] uses the integer \( lgrid \) to control the density of the frequency grid.

\[ [b,err] = \text{firpm}(\_\_\_\_) \] returns the maximum ripple height in \( err \). You can use this with any of the previous input syntaxes.

\[ [b,err,res] = \text{firpm}(\_\_\_\_) \] returns the frequency response characteristics as a structure \( res \).

\[ b = \text{firpm}(n,f,fresp,w) \] 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 \( 0.3\pi \) and \( 0.7\pi \) rad/sample and normalized passband frequencies of \( 0.4\pi \) and \( 0.6\pi \) rad/sample. Plot the ideal and actual magnitude responses.

```matlab
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.

\[ N = 50; \]
\[ F_s = 1e3; \]

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)
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.

```matlab
[n,fo,ao,w] = firpmord([1500 2000],[1 0],[0.001 0.01],8000);
b = firpm(n,fo,ao,w);
fvtool(b,1)
```
Input Arguments

\textbf{n — Filter order}
real positive scalar

Filter order, specified as a real positive scalar.

\textbf{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, specified as either 'hilbert' or 'differentiator':

- 'hilbert' — For linear-phase filters with odd symmetry (type III and type IV)

  The output coefficients in b obey the relation \(b(k) = -b(n + 2 - k)\), \(k = 1, ..., n + 1\). This class of filters includes the Hilbert transformer, which has a desired amplitude of 1 across the entire band.

  For example,

  \[
  h = \text{firpm}(30, [0.1 0.9], [1 1], '\text{hilbert}')
  \]

  designs an approximate FIR Hilbert transformer of length 31.

- 'differentiator' — For type III and type IV filters, using a special weighting technique

  For nonzero amplitude bands, the filter weights the error by a factor of \(1/f\) so that the error at low frequencies is much smaller than at high frequencies. For FIR
differentiators, which have an amplitude characteristic proportional to frequency, these filters minimize the maximum relative error (the maximum of the ratio of the error to the desired amplitude).

**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 \texttt{firpm} with this syntax:

\[
[dh,dw] = \text{fresp}(n,f,gf,w)
\]

The arguments are similar to those for \texttt{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 \texttt{firpm}. \(gf\) determines the frequency grid at which the response function must be evaluated, and contains the same data returned by \texttt{cfirpm} in the field of the \texttt{opt} structure.
- \(w\) is a vector of real, positive weights, one per band, used during optimization. \(w\) is optional in the call to \texttt{firpm}; if not specified, it is set to unity weighting before being passed to \texttt{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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>res.fgrid</code></td>
<td>Frequency grid vector used for the filter design optimization</td>
</tr>
<tr>
<td><code>res.des</code></td>
<td>Desired frequency response for each point in <code>res.fgrid</code></td>
</tr>
<tr>
<td><code>res.wt</code></td>
<td>Weighting for each point in <code>opt.fgrid</code></td>
</tr>
<tr>
<td><code>res.H</code></td>
<td>Actual frequency response for each point in <code>res.fgrid</code></td>
</tr>
<tr>
<td><code>res.error</code></td>
<td>Error at each point in <code>res.fgrid</code> ((\text{res.des} - \text{res.H}))</td>
</tr>
<tr>
<td><code>res.iextr</code></td>
<td>Vector of indices into <code>res.fgrid</code> for extremal frequencies</td>
</tr>
<tr>
<td><code>res.fextr</code></td>
<td>Vector of extremal frequencies</td>
</tr>
</tbody>
</table>

**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 [1]. The Parks-McClellan algorithm uses the Remez exchange algorithm and Chebyshev approximation theory to design filters with an optimal fit between the desired and actual frequency responses. The filters are optimal in the sense that the maximum error between the desired frequency response and the actual frequency response is minimized. Filters designed this way exhibit an equiripple behavior in their frequency responses and are sometimes called equiripple filters. `firpm` exhibits discontinuities at the head and tail of its impulse response due to this equiripple nature.

The relationship between the `f` and `a` vectors in defining a desired frequency response is shown in the illustration below.
**firpm** always uses an even filter order for configurations with 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 [5] for more details.)
<table>
<thead>
<tr>
<th>Linear Phase Filter Type</th>
<th>Filter Order</th>
<th>Symmetry of Coefficients</th>
<th>Response $H(f)$, $f = 0$</th>
<th>Response $H(f)$, $f = 1$ (Nyquist)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type I</td>
<td>Even</td>
<td>even: $b(k) = b(n + 2 - k)$, $k = 1, ..., n + 1$</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>Type II</td>
<td>Odd</td>
<td>even: $b(k) = b(n + 2 - k)$, $k = 1, ..., n + 1$</td>
<td>No restriction</td>
<td>$H(1) = 0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>firpm increments the filter order by 1 if you attempt to construct a type II filter with a nonzero passband at the Nyquist frequency.</td>
</tr>
<tr>
<td>Type III</td>
<td>Even</td>
<td>odd: $b(k) = -b(n + 2 - k)$, $k = 1, ..., n + 1$</td>
<td>$H(0) = 0$</td>
<td>$H(1) = 0$</td>
</tr>
<tr>
<td>Type IV</td>
<td>Odd</td>
<td>odd: $b(k) = -b(n + 2 - k)$, $k = 1, ..., n + 1$</td>
<td>$H(0) = 0$</td>
<td>No restriction</td>
</tr>
</tbody>
</table>

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** $b = \text{firpm}(n,f,a,w)$ is equivalent to $b = \text{firpm}(n,f,\{\text{@firpmfrf},a\},w)$, 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” (MATLAB) for more information.
References


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 | cffirpm | cheby1 | cheby2 | 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)
[n,fo,ao,w] = firpmord(f,a,dev,fs)
c = firpmord(f,a,dev,fs,'cell')
```

**Description**

```
[n,fo,ao,w] = firpmord(f,a,dev)\ finds the approximate order, normalized frequency band edges, frequency band amplitudes, and weights that meet input specifications f, a, and dev.

- f is a vector of frequency band edges (between 0 and \( F_s/2 \), where \( F_s \) is the sampling frequency), and a is a vector specifying the desired amplitude on the bands defined by f. The length of f is two less than twice the length of a. The desired function is piecewise constant.
- dev is a vector the same size as a that specifies the maximum allowable deviation or ripples between the frequency response and the desired amplitude of the output filter for each band.

Use firpm with the resulting order n, frequency vector fo, amplitude response vector ao, and weights w to design the filter b which approximately meets the specifications given by firpmord input parameters f, a, and dev.
```

```
b = firpm(n,fo,ao,w)
```

```
[n,fo,ao,w] = firpmord(f,a,dev,fs)\ specifies a sampling frequency fs. fs defaults to 2 Hz, implying a Nyquist frequency of 1 Hz. You can therefore specify band edges scaled to a particular application’s sampling frequency.
```

```
c = firpmord(f,a,dev,fs,'cell')\ generates a cell-array whose elements are the parameters to firpm.
```
**Note** In some cases, `firpmord` underestimates or overestimates the order \( n \). If the filter does not meet the specifications, try a higher order such as \( n+1 \) or \( n+2 \).

**Examples**

**Minimum-Order Lowpass Filter**

Design a minimum-order lowpass filter with a 500 Hz passband cutoff frequency and 600 Hz stopband cutoff frequency. Specify a sampling frequency of 2000 Hz. Require at least 40 dB of attenuation in the stopband and less than 3 dB of ripple in the passband.

```matlab
rp = 3;           % Passband ripple
rs = 40;          % Stopband ripple
fs = 2000;        % Sampling frequency
f = [500 600];    % Cutoff frequencies
a = [1 0];        % Desired amplitudes

% Convert the deviations to linear units. Design the filter and visualize its magnitude and
% phase responses.

dev = [(10^(rp/20)-1)/(10^(rp/20)+1)  10^(-rs/20)];
[n,fo,ao,w] = firpmord(f,a,dev,fs);
b = firpm(n,fo,ao,w);
freqz(b,1,1024,fs)
title('Lowpass Filter Designed to Specifications')
```


Note that the filter falls slightly short of meeting the stopband attenuation and passband ripple specifications. Using \( n+1 \) in the call to \texttt{firpm} instead of \( n \) achieves the desired amplitude characteristics.

**Parks-McClellan Order of Lowpass Filter**

Design a lowpass filter with a 1500 Hz passband cutoff frequency and 2000 Hz stopband cutoff frequency. Specify a sampling frequency of 8000 Hz. Require a maximum stopband amplitude of 0.1 and a maximum passband error (ripple) of 0.01.
\[ n, fo, ao, w \] = \text{firpmord}([1500 2000], [1 0], [0.01 0.1], 8000);
\[ b = \text{firpm}(n, fo, ao, w); \]

Obtain an equivalent result by having \text{firpmord} generate a cell array. Visualize the frequency response of the filter.

\[ c = \text{firpmord}([1500 2000], [1 0], [0.01 0.1], 8000, '\text{cell}') ; \]
\[ B = \text{firpm}(c{:}); \]
\[ \text{freqz}(B, 1, 1024, 8000) \]
Algorithms

firpmord uses the algorithm suggested in [1]. This method is inaccurate for band edges close to either 0 or the Nyquist frequency, fs/2.

References


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 | firpm | kaiserord

Introduced before R2006a
**firtype**

Type of linear phase FIR filter

**Syntax**

\[ t = \text{firtype}(b) \]
\[ t = \text{firtype}(d) \]

**Description**

\[ t = \text{firtype}(b) \] 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.

\[ t = \text{firtype}(d) \] 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.

```matlab
subplot(2,1,1)
b = fir1(8,0.5);
impz(b), title([\'Type ' int2str(firtype(b))])
```

```matlab
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.

```matlab
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.

```matlab
d1 = designfilt('lowpassfir','DesignMethod','window', ... 
    'FilterOrder',8,'CutoffFrequency',0.5);
disp(['d1 is of type ' int2str(firtype(d1))])
d1 is of type 1
```

1-997
\[ d2 = \text{designfilt}('\text{lowpassfir}', 'DesignMethod', '\text{window}', ... \\
\quad 'FilterOrder', 9, 'CutoffFrequency', 0.5); \]
\[ \text{disp}(['d2 is of type ' int2str(firtype(d2))]) \]
d2 is of type 2

\[ d3 = \text{designfilt}('\text{hilbertfir}', 'DesignMethod', '\text{equiripple}', ... \\
\quad 'FilterOrder', 8, '\text{TransitionWidth}', 0.4); \]
\[ \text{disp}(['d3 is of type ' int2str(firtype(d3))]) \]
d3 is of type 3

\[ d4 = \text{designfilt}('\text{hilbertfir}', 'DesignMethod', '\text{equiripple}', ... \\
\quad 'FilterOrder', 9, '\text{TransitionWidth}', 0.4); \]
\[ \text{disp}(['d4 is of type ' int2str(firtype(d4))]) \]
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 | dfilt object  
FIR filter, specified as any of the following:  
- A digitalFilter object. Use designfilt to generate a digital filter based on frequency-response specifications.  
- A dfilt filter object.

### 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

\[ w = \text{flattopwin}(L) \]
\[ w = \text{flattopwin}(L,\text{sflag}) \]

Description

\[ w = \text{flattopwin}(L) \] returns an \( L \)-point symmetric flat top window

\[ w = \text{flattopwin}(L,\text{sflag}) \] returns an \( L \)-point symmetric flat top window using the window sampling method specified by \text{sflag}.

Examples

Flat Top Window

Create a 64-point symmetric flat top window. View the result using \text{wvtool}.

\[ N = 64; \]
\[ w = \text{flattopwin}(N); \]
\[ \text{wvtool}(w) \]
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 \( 0 \leq n \leq L - 1 \). The coefficient values are:

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_0 )</td>
<td>0.21557895</td>
</tr>
<tr>
<td>( a_1 )</td>
<td>0.41663158</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>0.277263158</td>
</tr>
<tr>
<td>( a_3 )</td>
<td>0.083578947</td>
</tr>
<tr>
<td>( a_4 )</td>
<td>0.006947368</td>
</tr>
</tbody>
</table>

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


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
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.

```matlab
figure
dfreqs(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`.

```matlab
[b,a] = besself(5,10000);  % Bessel analog filter design
freqs(b,a)                 % Plot frequency response
```

![Frequency Response of a Lowpass Analog Bessel Filter](image-url)
**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 \((2\pi \times 10^6 \text{ rad/second})\) and 1 GHz \((2\pi \times 10^9 \text{ rad/second})\).

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 \(s = j\omega\):

\[
s = 1j*w;
h = \text{polyval}(b,s)./\text{polyval}(a,s);
\]
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.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description of Resulting Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>dffir</td>
<td>Direct-form FIR filter</td>
</tr>
<tr>
<td>dffirt</td>
<td>Transposed direct-form FIR filter</td>
</tr>
<tr>
<td>dfsymfir</td>
<td>Symmetrical direct-form FIR filter</td>
</tr>
<tr>
<td>dfasymfir</td>
<td>Asymmetrical direct-form FIR filter</td>
</tr>
<tr>
<td>fftfir</td>
<td>Fast Fourier transform FIR filter</td>
</tr>
</tbody>
</table>

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.

```matlab
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);
hd = design(d,'freqsamp','Window','rectwin');
fvtool(hd)

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;
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.

\[ b_0 = 0.05634; \]
\[ b1 = [1 1]; \]
\[ b2 = [1 -1.0166 1]; \]
\[ a1 = [1 -0.683]; \]
\[ a2 = [1 -1.4461 0.7957]; \]

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\text{ rad/sample})')
ylabel('Magnitude (dB)')
Frequency Response of an FIR filter

Design an FIR lowpass filter of order 80 using a Kaiser window with $\beta = 8$. Specify a normalized cutoff frequency of $0.5\pi$ rad/sample. Display the magnitude and phase responses of the filter.

$$b = \text{fir1}(80, 0.5, \text{kaiser}(81, 8));$$

$$\text{freqz}(b, 1)$$
Design the same filter using `designfilt`. Display its magnitude and phase responses using `fvtool`.

```matlab
d = designfilt('lowpassfir','FilterOrder',80, ... 
    'CutoffFrequency',0.5,'Window',{"kaiser',8});
freqz(d)
```
Frequency Response of an FIR Bandpass Filter

Design an FIR bandpass filter with passband between \(0.35\pi\) and \(0.8\pi\) rad/sample and 3 dB of ripple. The first stopband goes from 0 to \(0.1\pi\) rad/sample and has an attenuation of 40 dB. The second stopband goes from \(0.9\pi\) rad/sample to the Nyquist frequency and has an attenuation of 30 dB. Compute the frequency response. Plot its magnitude in both linear units and decibels. Highlight the passband.

\[
sf1 = 0.1;
pf1 = 0.35;
pf2 = 0.8;
\]
sf2 = 0.9;
pb = linspace(pf1,pf2,1e3)*pi;

bp = designfilt('bandpassfir', ...
    'StopbandAttenuation1',40, 'StopbandFrequency1',sf1,...
    'PassbandFrequency1',pf1,'PassbandRipple',3,'PassbandFrequency2',pf2, ...
    'StopbandFrequency2',sf2,'StopbandAttenuation2',30);

[h,w] = freqz(bp,1024);
hpb = freqz(bp,pb);

subplot(2,1,1)
plot(w/pi,abs(h),pb/pi,abs(hpb),'.-')
axis([0 1 -1 2])
legend('Response','Passband','Location','South')
ylabel('Magnitude')

subplot(2,1,2)
plot(w/pi,db(h),pb/pi,db(hpb),'.-')
axis([0 1 -60 10])
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Magnitude (dB)')
Input Arguments

\textbf{b, a} — Transfer function coefficients vectors

Transfer function coefficients, specified as vectors. Express the transfer function in terms of \( b \) and \( a \) as

\[
H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})} = \frac{b(1)+b(2)\ e^{-j\omega} + b(3)\ e^{-j2\omega} + \cdots + b(M)\ e^{-j(M-1)\omega}}{a(1)+a(2)\ e^{-j\omega} + a(3)\ e^{-j2\omega} + \cdots + a(N)\ e^{-j(N-1)\omega}}.
\]
Example: \( b = \left[ \frac{1}{6} \ 3 \ 3 \ 1 \right] \) and \( a = \left[ \frac{3}{3} \ 0 \ 1 \ 0 \right] \) specify a third-order Butterworth filter with normalized 3 dB frequency 0.5\( \pi \) rad/sample.

Data Types: double | single
Complex Number Support: Yes

\( n \) — Number of evaluation points
512 (default) | positive integer scalar

Number of evaluation points, specified as a positive integer scalar no less than 2. When \( n \) is absent, it defaults to 512. For best results, set \( n \) to a value greater than the filter order.

Data Types: double

\( sos \) — Second-order section coefficients
matrix

Second-order section coefficients, specified as a matrix. \( sos \) is a \( K \)-by-6 matrix, where the number of sections, \( K \), must be greater than or equal to 2. If the number of sections is less than 2, 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: \( s = \left[ \begin{array}{cc} 2 & 4 & 2 & 6 & 0 & 2 \\ 3 & 3 & 0 & 6 & 0 & 0 \end{array} \right] \) specifies a third-order Butterworth filter with normalized 3 dB frequency 0.5\( \pi \) rad/sample.

Data Types: double | single
Complex Number Support: Yes

\( d \) — Digital filter
digitalFilter object

Digital filter, specified as a digitalFilter object. Use designfilt to generate a digital filter based on frequency-response specifications.

Example: \( d = \) designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5) specifies a third-order Butterworth filter with normalized 3 dB frequency 0.5\( \pi \) rad/sample.

\( fs \) — 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 \( n \). \( w = \pi \) 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 \( \text{freqz} \) 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 \( \pi \). If you specify 'whole' in your input, the values in \( w \) range from 0 to \( 2\pi \). If you specify \( n \), \( w \) has length \( n \). If you do not specify \( n \), or specify \( n \) as the empty vector, \( w \) has length 512.

\( f \) — Frequencies vector

Frequencies, returned as a vector expressed in hertz. \( f \) has values ranging from 0 to \( f_s/2 \) Hz. If you specify 'whole' in your input, the values in \( f \) range from 0 to \( f_s \) Hz. If you specify \( n \), \( f \) has length \( n \). If you do not specify \( n \), or specify \( n \) as the empty vector, \( f \) has length 512.
Algorithms

The frequency response of a digital filter can be interpreted as the transfer function evaluated at \( z = e^{j\omega} \) [1].

\texttt{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.

\texttt{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, \texttt{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


Extended Capabilities

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

Usage notes and limitations:

Does not support variable-size inputs.

See Also
abs | angle | designfilt | digitalFilter | fft | filter | freqs | impz | invfreqs | logspace
Introduced before R2006a
**fsst**

Fourier synchrosqueezed transform

**Syntax**

```plaintext
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 $\frac{2\pi}{5}$ rad/sample and $\frac{4\pi}{5}$ rad/sample. The higher frequency sinusoid has 3 times the amplitude of the other sinusoid.

```matlab
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.
```

```matlab
[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.

```matlab
[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.

```matlab
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 \texttt{spectrogram} function. Specify the default parameters used by \texttt{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

\[
\text{[stft,f,t]} = \text{spectrogram}([x1+x2,\text{kaiser}(256,10),255,256,fs]);
\]

Plot the absolute value of the STFT.

\[
\text{mesh}(t,f,\text{abs(stft)})
\]

\[
\text{xlabel('Time (s)')}
\]

\[
\text{ylabel('Frequency (Hz)')}
\]

\[
\text{title('Short-Time Fourier Transform')}
\]

\[
\text{axis tight}
\]

\[
\text{view(2)}
\]
**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.

```matlab
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.

```matlab
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 \( \beta = 20 \).

```matlab
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)')
```

1-1037
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
Load a speech signal sampled at \( F_s = 7418 \) Hz. The file contains a recording of a female voice saying the word "MATLAB®."

```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.

**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®."
Use `ifsst` to invert the transform. Compare the original and reconstructed signals.

```matlab
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-mltb,Fs)

**Input Arguments**

**x — Input signal**  
vector

Input signal, specified as a vector.

Example: \( \cos(\pi/4*(0:159)) + \text{randn}(1,160) \) 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 $\left(1 - \cos\left(2\pi\left(\theta:N\right)/N\right)\right)/2$ 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. 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”
\[ \Omega_g f(t, \eta) = \frac{1}{j2\pi} \frac{\partial}{\partial t} V_g f(t, \eta) = \eta - \frac{1}{j2\pi} \frac{V_g \partial \Omega_g f(t, \eta)}{V_g f(t, \eta)}. \]

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. \( \text{fsst} \) 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 \( \Omega_g f(t, \eta) \):

\[ f_k(t) \approx \frac{1}{g(0)} \int \omega - \Omega_k(t) < \epsilon T_g f(t, \omega) \, d\omega, \]

where \( \epsilon \) 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 \([-\Delta, \Delta]\), 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”.

1-1045
References


Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

Window must be double precision.

See Also

Apps
Signal Analyzer

Functions
ifsst | pspectrum | spectrogram | tfridge

Topics
“Hilbert Transform and Instantaneous Frequency”
“Practical Introduction to Time-Frequency Analysis”
Introduced in R2016b
**FVTool**

Filter Visualization Tool

**Description**

*Filter Visualization Tool* is an interactive tool that enables you to display the magnitude, phase response, group delay, impulse response, step response, pole-zero plot, and coefficients of a filter. You can export the displayed response to a file with **File > Export**.

If the DSP System Toolbox product is installed, **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`.

**Open the FVTool**

The FVTool can be opened programmatically using one of the methods described in “Programmatic Use” on page 1-1059.

**Examples**

**Magnitude Response of Elliptic Filter**

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. Start FVTool from the command line.

```matlab
[b,a] = ellip(6,3,50,300/500);
fvtool(b,a)
```
**Display Analysis Parameters**

Display and analyze multiple FIR filters, starting FVTool from the command line.

\[
b1 = \text{firpm}(20, [0 \ 0.4 \ 0.5 \ 1], [1 \ 1 \ 0 \ 0]);
\]

\[
b2 = \text{firpm}(40, [0 \ 0.4 \ 0.5 \ 1], [1 \ 1 \ 0 \ 0]);
\]

fvtool(b1,1,b2,1)
Display the associated analysis parameters by selecting **Analysis > Analysis Parameters**.
FVTool Figure Handle Commands

Start FVTool from the command line. 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.

\[[b,a] = \text{ellip}(6,3,50,300/500);\]
\[h = \text{fvtool}(b,a)\]
h = Figure (filtervisualizationtool) with properties:

    Number: 1
    Name: 'Filter Visualization Tool - Magnitude Response (dB)'
    Color: [0.9400 0.9400 0.9400]
    Position: [361 292 560 420]
    Units: 'pixels'

Use get to show all properties

Display the phase response of the filter.
h.Analysis = 'phase'

h = Figure (filtervisualizationtool) with properties:
Number: 1
   Name: 'Filter Visualization Tool - Phase Response'
   Color: [0.9400 0.9400 0.9400]
   Position: [361 292 560 420]
   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.

```matlab
h.Fs = 1000;
h.FrequencyRange = '[-Fs/2, Fs/2]'
```
h =
    Figure (filtervisualizationtool) with properties:

    Number: 1
    Name: 'Filter Visualization Tool - Phase Response'
    Color: [0.9400 0.9400 0.9400]
    Position: [361 292 560 420]
    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'
AnalysisToolbar: 'on'
FigureToolbar: 'on'
DesignMask: 'off'
SOSViewSettings: [1x1 dspopts.sosview]
Fs: 1000
Alphamap: [1x64 double]
CloseRequestFcn: 'closereq'
Color: [0.9400 0.9400 0.9400]
Colormap: [256x3 double]
CurrentAxes: [1x1 Axes]
CurrentCharacter: ''
CurrentObject: [0x0 GraphicsPlaceholder]
CurrentPoint: [0 0]
DockControls: 'on'
FileName: ''
IntegerHandle: 'on'
InvertHardcopy: 'on'
KeyPressFcn: ''
KeyReleaseFcn: ''
MenuBar: 'none'
Name: 'Filter Visualization Tool - Phase Response'
NextPlot: 'new'
NumberTitle: 'on'
PaperUnits: 'inches'
PaperOrientation: 'portrait'
PaperPosition: [1.3333 3.3125 5.8333 4.3750]
PaperPositionMode: 'auto'
PageSize: [8.5000 11]
PaperType: 'usletter'
Pointer: 'arrow'
PointerShapeCData: [16x16 double]
PointerShapeHotSpot: [1 1]
Position: [361 292 560 420]
Renderer: 'opengl'
RendererMode: 'auto'
Resize: 'on'
ResizeFcn: ''
SelectionType: 'normal'
ToolBar: 'auto'
Type: 'figure'
Units: 'pixels'
Magnitude and Phase Response of Bandpass FIR Filter

Design a 50th-order bandpass FIR filter with stopband frequencies 150 Hz and 350 Hz and passband frequencies 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')
• “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`.

`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 \text{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, \text{fvtool} considers the input to be a numerator vector.

\text{fvtool}(d) \text{ opens FVTool and displays the magnitude response of a digital filter, } d. \text{ Use \text{designfilt} \text{ to generate } d \text{ based on frequency-response specifications.}

\text{fvtool}(b1,a1,b2,a2,...,bN,aN) \text{ opens FVTool and displays the magnitude responses of multiple filters defined with numerators } b1, ..., bN \text{ and denominators } a1, ..., aN.

\text{fvtool}(\text{sos1},\text{sos2},...,\text{sosN}) \text{ opens FVTool and displays the magnitude responses of multiple filters defined with second order section matrices } \text{sos1}, \text{sos2}, ..., \text{sosN}.

\text{fvtool}(\text{Hd}) \text{ opens FVTool and displays the magnitude responses for the dfilt filter object } \text{Hd} \text{ or the array of dfilt filter objects.}

\text{fvtool}(\text{Hd1},\text{Hd2},...,\text{HdN}) \text{ opens FVTool and displays the magnitude responses of the filters in the dfilt objects } \text{Hd1}, \text{Hd2}, ..., \text{HdN}.

\text{h} = \text{fvtool}(\_\_\_) \text{ returns a figure handle } h. \text{ You can use this handle to interact with FVTool from the command line. See "Controlling FVTool from the MATLAB Command Line".}

**More About**

**Controlling FVTool from the GUI**

FVTool has two toolbars:
• An extended version of the MATLAB plot editing toolbar. The following table shows the toolbar icons specific to FVTool:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Icon]</td>
<td>Restore default view. This view displays buffer regions around the data and shows only significant data. To see the response using standard MATLAB plotting, which shows all data values, use View &gt; Full View.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Toggle legend</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Toggle grid</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Link to Filter Designer (appears only if FVTool was started from Filter Designer)</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Toggle Add mode/Replace mode (appears only if FVTool was launched from Filter Designer)</td>
</tr>
</tbody>
</table>

• Analysis toolbar with the following icons:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Icon]</td>
<td>Magnitude response of the current filter. See freqz and zerophase for more information.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>To see the zero-phase response, right-click the y-axis label of the Magnitude plot and select Zero-phase from the context menu.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Phase response of the current filter. See phasez for more information.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Magnitude response and the phase response of the current filter superimposed on one another. See freqz for more information.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Group delay of the current filter. Group delay is the average delay of the filter as a function of frequency. See grpdelay for more information.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Phase delay of the current filter. Phase delay is the time delay the filter imposes on each component of the input signal. See phasedelay for more information.</td>
</tr>
<tr>
<td>Icon</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td><img src="image" alt="impulse_response" /></td>
<td>Impulse response of the current filter. The impulse response is the response of the filter to an impulse input. See <code>impz</code> for more information.</td>
</tr>
<tr>
<td><img src="image" alt="step_response" /></td>
<td>Step response of the current filter. The step response is the response of the filter to a step input. See <code>stepz</code> for more information.</td>
</tr>
<tr>
<td><img src="image" alt="pole_zero_plot" /></td>
<td>Pole-zero plot, which shows the pole and zero locations of the current filter on the z-plane. See <code>zplane</code> for more information.</td>
</tr>
<tr>
<td><img src="image" alt="filter_coefficients" /></td>
<td>Filter coefficients of the current filter, which depend on the filter structure (direct-form or lattice) in a text box. For SOS filters, each section is displayed as a separate filter.</td>
</tr>
<tr>
<td><img src="image" alt="filter_info" /></td>
<td>Detailed filter information.</td>
</tr>
</tbody>
</table>
Linking to Filter Designer

In the **Filter Designer** app, selecting **View > Filter Visualization Tool** or the **Full View Analysis** toolbar button when an analysis is displayed starts FVTool for the current filter. You can synchronize **Filter Designer** and FVTool with the toolbar button. Any changes made to the filter in **Filter Designer** are immediately reflected in FVTool.

Two link modes are provided via the toggle toolbar button:
• Replace — removes the filter currently displayed in FVTool and inserts the new filter.

• Add — retains the filter currently displayed in FVTool and adds the new filter to the display.

**Overlaying a Response**

You can overlay a second response on the plot by selecting **Analysis > Overlay Analysis** and selecting an available response. A second y-axis is added to the right side of the response plot. The Analysis Parameters dialog box shows parameters for the x-axis and both y-axes. See “Display Analysis Parameters” on page 1-1049 for a sample Analysis Parameters dialog box.

**See Also**

**Apps**
Filter Designer | Signal Analyzer

**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

\[ y = \text{fwht}(x) \]
\[ y = \text{fwht}(x,n) \]
\[ y = \text{fwht}(x,n,\text{ordering}) \]

Description

\[ y = \text{fwht}(x) \] returns the coefficients of the discrete Walsh-Hadamard transform of the input \( x \). If \( x \) is a matrix, the FWHT is calculated on each column of \( x \). The FWHT operates only on signals with length equal to a power of 2. If the length of \( x \) is less than a power of 2, its length is padded with zeros to the next greater power of two before processing.

\[ y = \text{fwht}(x,n) \] returns the \( n \)-point discrete Walsh-Hadamard transform, where \( n \) must be a power of 2. \( x \) and \( n \) must be the same length. If \( x \) is longer than \( n \), \( x \) is truncated; if \( x \) is shorter than \( n \), \( x \) is padded with zeros.

\[ y = \text{fwht}(x,n,\text{ordering}) \] specifies the ordering to use for the returned Walsh-Hadamard transform coefficients. To specify the ordering, you must enter a value for the length \( n \) or, to use the default behavior, specify an empty vector ([ ]) for \( n \). Valid values for the ordering are the following:

<table>
<thead>
<tr>
<th>Ordering</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'sequency'</td>
<td>Coefficients in order of increasing sequency value, where each row has an additional zero crossing. This is the default ordering.</td>
</tr>
<tr>
<td>'hadamard'</td>
<td>Coefficients in normal Hadamard order.</td>
</tr>
<tr>
<td>'dyadic'</td>
<td>Coefficients in Gray code order, where a single bit change occurs from one coefficient to the next.</td>
</tr>
</tbody>
</table>

For more information on the Walsh functions and ordering, see “Walsh-Hadamard Transform”.

1-1065
Examples

Walsh-Hadamard Transform of a Signal

This example shows a simple input signal and its Walsh-Hadamard transform.

\[ x = [19 \ -1 \ 11 \ -9 \ 13 \ -15 \ 5]; \]
\[ y = fwh(x) \]
\[ y = 1 \times 8 \]
\[
\begin{array}{cccccccc}
2 & 3 & 0 & 4 & 0 & 0 & 10 & 0 \\
\end{array}
\]

\[ y \] 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 \( x \).

\[ w_0 = [1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1]; \]
\[ w_1 = [1 \ 1 \ 1 \ 1 \ -1 \ -1 \ -1 \ -1]; \]
\[ w_3 = [1 \ 1 \ -1 \ -1 \ 1 \ 1 \ -1 \ -1]; \]
\[ w_6 = [1 \ -1 \ 1 \ -1 \ -1 \ 1 \ -1 \ 1]; \]
\[ w = y(\theta+1)*w_0 + y(1+1)*w_1 + y(3+1)*w_3 + y(6+1)*w_6 \]
\[ w = 1 \times 8 \]
\[
\begin{array}{cccccccc}
19 & -1 & 11 & -9 & -7 & 13 & -15 & 5 \\
\end{array}
\]

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


See Also
dct | fft | idct | ifft | ifwht

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')
```

![Plot of Gaussian RF Pulse](image)
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 \( t \).

\[ y_q \] — Quadrature Gaussian pulse
vector

Quadrature Gaussian-modulated sinusoidal pulse, returned as a vector of unit amplitude at the times indicated by the time vector \( t \).

\[ y_e \] — RF signal envelope
vector

RF signal envelope of unit amplitude at the times indicated by the time vector \( t \).

\[ t_c \] — Cutoff time
positive real scalar

The cutoff time in seconds at which the trailing pulse envelope falls below \( t_{pe} \) 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

References


See Also
rcosdesign

Introduced in R2013b
gausswin

Gaussian window

Syntax

\[ w = \text{gausswin}(L) \]
\[ w = \text{gausswin}(L,\alpha) \]

Description

\[ w = \text{gausswin}(L) \] returns an \( L \)-point Gaussian window.

\[ w = \text{gausswin}(L,\alpha) \] 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 \texttt{wvtool}.

\[ L = 64; \]
\[ \text{wvtool(gausswin(L))} \]
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 \texttt{fftshift} to center the Fourier transform at zero frequency (DC).

\begin{verbatim}
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 \text{ rad/sample}')
title('Fourier Transform of Gaussian Window')
\end{verbatim}
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:

\[
w(n) = e^{-\frac{1}{2} \left( \alpha \frac{n}{(L-1)/2} \right)^2} = e^{-\frac{n^2}{2 \sigma^2}},
\]

where \(-(L - 1)/2 \leq n \leq (L - 1)/2\), and \( \alpha \) is inversely proportional to the standard deviation, \( \sigma \), 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


## Extended Capabilities

### C/C++ Code Generation

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

Apps
Window Designer

Functions
chebwin | kaiser | tukeywin | wvtool

Introduced before R2006a
getLabelDefinitions
Get label definitions in labeled signal set

Syntax
lbldefs = getLabelDefinitions(lss)

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.

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"
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'.

**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 idx.

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
    TimeInterval: "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.

```
[lbs,info] = getLabeledSignal(lss,2)
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
```

Extract the signal. Determine its sample rate and use the sample rate to compute the time vector:

```
fs = info.SampleRate;
sg = getSignal(lss,2);
t = (0:length(sg)-1)/fs;
```
Identify the moan and trill regions of interest and the peaks of the trill region.

```matlab
mvals = getLabelValues(lss,2,'MoanRegions');
tvals = getLabelValues(lss,2,'TrillRegions');

peaks = getLabelValues(lss,2,{'TrillRegions','TrillPeaks'});

Plot the signal. Highlight the regions of interest and the peaks.

plot(t,sg)
hold on
[X,Y] = meshgrid([mvals.ROILimits;tvals.ROILimits],ylim);
plot(X,Y,':k')
topts = {'HorizontalAlignment','center','FontWeight','bold', ...
         'FontSize',12,'Color',[139 69 19]/255);
text((X(1,1:4)+X(1,5:end))/2,Y(2,5:end)-0.1, ...
     ["moan" "moan" "moan" "trill"],topts{:})
hold off
```
Input Arguments

\texttt{lss} — Labeled signal set
\begin{verbatim}
labeledSignalSet object
\end{verbatim}

Labeled signal set, specified as a \texttt{labeledSignalSet} object.

Example: \texttt{labeledSignalSet([\texttt{randn(100,1)} \newline \texttt{randn(10,1)}], signalLabelDefinition('female'))} specifies a two-member set of random signals containing the attribute 'female'.

1-1088
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
getLabelNames

Get label names in labeled signal set

**Syntax**

```matlab
lblnames = getLabelNames(lss)
sublblnames = getLabelNames(lss,lblname)
```

**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`.

**Examples**

**Get Label Names**

Load a labeled signal set containing recordings of whale songs.

```matlab
load whales
lss
```

`lss =` labeledSignalSet with properties:

```matlab
    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.

```matlab
str = getLabelNames(lss)
```

`str = 3x1 string array`

```
"WhaleType"
"MoanRegions"
"TrillRegions"
```

Verify that only the 'TrillRegions' label has sublabels.

```matlab
for kj = 1:length(str)
    sbstr = str{kj};
    sbl = [sbstr getLabelNames(lss,sbstr)]
end
```

```matlab
sbl =
'WhaleType'

sbl =
'MoanRegions'

sbl = 1x2 string array
    "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'.

1-1091
**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.

**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 idx.

[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.

```matlab
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.

```matlab
lbls = getLabelValues(lss)
```

```plaintext
lbls=2x3 table

| WhaleType | MoanRegions     | TrillRegions
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Member{1}</td>
<td>blue {3x2 table}</td>
<td>{1x3 table}</td>
</tr>
<tr>
<td>Member{2}</td>
<td>blue {3x2 table}</td>
<td>{1x3 table}</td>
</tr>
</tbody>
</table>
```

Display the moan ROI limits for the second signal of the set.

```matlab
lbb = getLabelValues(lss,2,'MoanRegions')
```

```plaintext
lbb=3x2 table

<table>
<thead>
<tr>
<th>ROILimits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>3.5</td>
</tr>
<tr>
<td>5.8</td>
<td>8</td>
</tr>
<tr>
<td>15.4</td>
<td>16.7</td>
</tr>
</tbody>
</table>
```

1-1094
Plot the trill region of the signal between the ROI limits. Display the labeled trill peaks.

```matlab
tvals = getLabelValues(lss,2,'TrillRegions');
peaks = getLabelValues(lss,2,{'TrillRegions','TrillPeaks'});

sg = getSignal(lss,2);
plot((0:length(sg)-1)/lss.SampleRate,s)
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)

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.

**srldx — 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 array
  "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 array
  "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

\[[s,info] = \text{getSignal}(\text{lss,midx})\]

Description

\[[s,info] = \text{getSignal}(\text{lss,midx})\] returns the values for the signals contained in member \text{midx} of the labeled signal set \text{lss}.

Examples

Get Signal

Load a labeled signal set containing recordings of whale songs.

\begin{verbatim}
load whales
lss
\end{verbatim}

\text{lss = }
   labeledSignalSet with properties:
   
   Source: \{2x1 cell\}
   NumMembers: 2
   TimeInformation: "sampleRate"
   SampleRate: 4000
   Labels: [2x3 table]
   Description: "Characterize wave song regions"

Use \text{labelDefinitionsHierarchy} to see a list of labels and sublabels. Use \text{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'.

\textbf{midx} — Member row number

positive integer

Member row number, specified as a positive integer. \textit{midx} specifies the member row number as it appears in the “Labels” on page 1-0 table of a labeled signal set.

\section*{Output Arguments}

\textbf{s} — Signal values

vector | matrix | timetable | cell array

Signal values, returned as vector, matrix, timetable, or cell array.

\textbf{info} — Time information

structure

Time information, returned as a structure.

\section*{See Also}

Signal Labeler \mid labeledSignalSet \mid signalLabelDefinition

\textbf{Introduced in R2018b}
gmonopuls

Gaussian monopulse

Syntax

\[ y = \text{gmonopuls}(t,fc) \]
\[ tc = \text{gmonopuls}('\text{cutoff}',fc) \]

Description

\( y = \text{gmonopuls}(t,fc) \) returns samples of the unit-amplitude Gaussian monopulse with center frequency \( fc \) (in hertz) at the times indicated in array \( t \). By default, \( fc = 1000 \) Hz.

\( tc = \text{gmonopuls}('\text{cutoff}',fc) \) 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 \( 2t_c \).

\[
fc = 2e9; \\
fs = 100e9; \\
tc = \text{gmonopuls}('\text{cutoff}',fc); \\
t = -2*tc:1/fs:2*tc; \\
y = \text{gmonopuls}(t,fc);
\]

The monopulse is defined by the equation
\[ y(t) = e^{1/2}(t/\sigma)\exp\left(-\left(\frac{t}{\sigma}\right)^2/2\right), \]

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.

\[
\begin{align*}
fc &= 2e9; \\
fs &= 100e9; \\
tc &= \text{gmonopuls('cutoff',fc)}; \\
D &= ((0:2)*7.5+2.5)*1e-9;
\end{align*}
\]

Generate the pulse train such that the total duration is \( 150t_c \). Plot the result.

\[
\begin{align*}
t &= 0:1/fs:150*tc; \\
yp &= \text{pulstran}(t,D,'gmonopuls',fc); \\
\text{plot}(t,yp)
\end{align*}
\]
**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_data = goertzel(data)
dft_data = goertzel(data,freq_indices)
dft_data = goertzel(data,freq_indices,dim)

Description

dft_data = goertzel(data) returns the discrete Fourier transform (DFT) of the input data, data, using a second-order Goertzel algorithm. If data is a matrix, goertzel computes the DFT of each column separately. You can specify data types as double or single precision.

dft_data = goertzel(data,freq_indices) returns the DFT for the frequency indices freq_indices. The values of freq_indices can be any whole number or fraction.

dft_data = goertzel(data,freq_indices,dim) computes the DFT of the matrix data along the dimension dim.

Examples

Estimate Telephone Keypad Frequencies

Estimate the frequencies of the tone generated by pressing the 1 button on a telephone keypad.

The number 1 produces a tone with frequencies 697 and 1209 Hz. Generate 205 samples of the tone with a sample rate of 8 kHz.

Fs = 8000;
N = 205;
lo = sin(2*pi*697*(0:N-1)/Fs);
hi = sin(2*pi*1209*(0:N-1)/Fs);
data = lo + hi;

Use the Goertzel algorithm to compute the DFT of the tone. Choose the indices corresponding to the frequencies used to generate the numbers 0 through 9.

f = [697 770 852 941 1209 1336 1477];
freq_indices = round(f/Fs*N) + 1;
dft_data = goertzel(data, freq_indices);

Plot the DFT magnitude.

stem(f, abs(dft_data))

ax = gca;
ax.XTick = f;
xlabel('Frequency (Hz)')
title('DFT Magnitude')
Resolve Frequency Components of a Noisy Tone

Generate a noisy cosine with frequency components at 1.24 kHz, 1.26 kHz, and 10 kHz. Specify a sample rate of 32 kHz. Reset the random number generator for reproducible results.

```matlab
rng default
Fs = 32e3;
t = 0:1/Fs:2.96;
```
x = cos(2*pi*t*10e3) + cos(2*pi*t*1.24e3) + cos(2*pi*t*1.26e3) ... + randn(size(t));

Generate the frequency vector. Use the Goertzel algorithm to compute the DFT. Restrict the range of frequencies to between 1.2 and 1.3 kHz.

N = (length(x)+1)/2;
f = (Fs/2)/N*(0:N-1);
indx = find(f>1.2e3 & f<1.3e3);
X = goertzel(x,indx);

Plot the mean squared spectrum expressed in decibels.

plot(f(indx)/1e3,mag2db(abs(X)/length(X)))

title('Mean Squared Spectrum')
xlabel('Frequency (kHz)')
ylabel('Power (dB)')
grid
Algorithms

The Goertzel algorithm implements the DFT as a recursive difference equation. To establish this difference equation, express the DFT as the convolution of an $N$-point input, $x(n)$, with the impulse response $h(n) = W_N^{-kn}u(n)$, where $W_N^{-kn} = e^{-j2nk/N}$ and $u(n)$ is the unit step sequence.

The Z-transform of the impulse response is
\[ H(z) = \frac{1 - W_N^k z^{-1}}{1 - 2\cos(2\pi k/N) z^{-1} + z^{-2}}. \]

The direct form II implementation is:

**Alternatives**

You can also compute the DFT with:

- `fft`: less efficient than the Goertzel algorithm when you only need the DFT at a few frequencies.
- `czt`, the chirp Z-transform. `czt` calculates the Z-transform of an input on a circular or spiral contour and includes the DFT as a special case.

**References**


Extended Capabilities

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

Usage notes and limitations:
See “Automatic dimension restriction” (MATLAB Coder).

Introduced before R2006a
grpdelay

Average filter delay (group delay)

Syntax

\[
\begin{align*}
[gd,w] &= \text{grpdelay}(b,a,n) \\
[gd,w] &= \text{grpdelay}(\text{sos},n) \\
[gd,w] &= \text{grpdelay}(d,n) \\
[gd,w] &= \text{grpdelay}(\_,\_\_,'\text{whole}') \\
[gd,f] &= \text{grpdelay}(\_,\_\_,n,fs) \\
[gd,f] &= \text{grpdelay}(\_,\_\_,n,'\text{whole}',fs) \\
gd &= \text{grpdelay}(\_,\_\_,\text{win}) \\
gd &= \text{grpdelay}(\_,\_\_,\text{fin},fs) \\
\text{grpdelay}(\_) 
\end{align*}
\]

Description

\([gd,w] = \text{grpdelay}(b,a,n)\) 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 \(a\).

\([gd,w] = \text{grpdelay}(\text{sos},n)\) returns the \(n\)-point group delay response corresponding to the second-order sections matrix \(\text{sos}\).

\([gd,w] = \text{grpdelay}(d,n)\) returns the \(n\)-point group delay response for the digital filter \(d\).

\([gd,w] = \text{grpdelay}(\_,\_\_,\_\_,'\text{whole}')\) returns the group delay at \(n\) sample points around the entire unit circle.

\([gd,f] = \text{grpdelay}(\_,\_\_,n,fs)\) 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.

```matlab
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 \text{ rad/sample})'
ylabel 'Group and phase delays'
legend('Group delay','Phase delay')
```
**Group Delay Response of a Butterworth digitalFilter**

Use `designfilt` to design a sixth-order Butterworth Filter with normalized 3-dB frequency $0.2\pi$ rad/sample. Display its group delay response.

```matlab
d = designfilt('lowpassiir','FilterOrder',6,'HalfPowerFrequency',0.2,'DesignMethod','butter');
grpdelay(d)
```

![Graph showing the group delay response of a Butterworth filter](image)
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.

```matlab
fs = 200;
d = designfilt('arbmagfir', ...
    'FilterOrder',88, ... 
    'NumBands',4, ... 
);```
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

\[
H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})} = \frac{b(1) + b(2) e^{-j\omega} + b(3) e^{-j2\omega} + \cdots + b(M) e^{-j(M-1)\omega}}{a(1) + a(2) e^{-j\omega} + a(3) e^{-j2\omega} + \cdots + a(N) e^{-j(N-1)\omega}}.
\]

Example: \( b = [1 \ 3 \ 3 \ 1]/6 \) and \( a = [3 \ 0 \ 1 \ 0]/3 \) specify a third-order Butterworth filter with normalized 3 dB frequency \( 0.5\pi \) rad/sample.

Data Types: double | single
Complex Number Support: Yes

\( n \) — Number of evaluation points

512 (default) | positive integer scalar

Number of evaluation points, specified as a positive integer scalar no less than 2. When \( n \) is absent, it defaults to 512. For best results, set \( n \) to a value greater than the filter order.

Data Types: double

\( sos \) — Second-order section coefficients

matrix

Second-order section coefficients, specified as a matrix. \( sos \) is a \( K \)-by-6 matrix, where the number of sections, \( K \), must be greater than or equal to 2. If the number of sections is less than 2, 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 \([b_i(1) \ b_i(2) \ b_i(3) \ a_i(1) \ a_i(2) \ a_i(3)]\).

Example: \( s = [2 \ 4 \ 2 \ 6 \ 0 \ 2;3 \ 3 \ 0 \ 6 \ 0 \ 0] \) specifies a third-order Butterworth filter with normalized 3 dB frequency \( 0.5\pi \) rad/sample.

Data Types: double | single
Complex Number Support: Yes

\( d \) — Digital filter

digitalFilter object

Digital filter, specified as a digitalFilter object. Use designfilt to generate a digital filter based on frequency-response specifications.

Example: \( d = \) designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5) specifies a third-order Butterworth filter with normalized 3 dB frequency \( 0.5\pi \) rad/sample.
**fs — 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 $H(e^{j\omega})$.

See Also

FVTool | cceps | designfilt | digitalFilter | fft | freqz | hilbert | icceps | phasedelay | rceps

Introduced before R2006a
hamming

Hamming window

Syntax

\[ w = \text{hamming}(L) \]
\[ w = \text{hamming}(L,\text{sflag}) \]

Description

\[ w = \text{hamming}(L) \] returns an \( L \)-point symmetric Hamming window.

\[ w = \text{hamming}(L,\text{sflag}) \] returns a Hamming window using the window sampling specified by \( \text{sflag} \).

Examples

Hamming Window

Create a 64-point Hamming window. Display the result using \text{wvtool}.

\[
L = 64; \\
\text{wvtool(hamming(L))}
\]
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.

```matlab
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:

$$w(n) = 0.54 - 0.46 \cos\left(2\pi \frac{n}{N}\right), \quad 0 \leq n \leq N.$$ 

The window length $L = N + 1$.

References


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**

\[
y = \text{hampel}(x) \\
y = \text{hampel}(x,k) \\
y = \text{hampel}(x,k,\text{nsigma})
\]

\[
[y,j] = \text{hampel}(\text{___}) \\
[y,j,\text{xmedian,\text{xsigma}}] = \text{hampel}(\text{___})
\]

\[
\text{hampel}(\text{___})
\]

**Description**

\( y = \text{hampel}(x) \) 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 \( \text{hampel} \) treats each column of \( x \) as an independent channel.

\( y = \text{hampel}(x,k) \) specifies the number of neighbors, \( k \), on either side of each sample of \( x \) in the measurement window. \( k \) defaults to 3.

\( y = \text{hampel}(x,k,\text{nsigma}) \) specifies a number of standard deviations, \( \text{nsigma} \), by which a sample of \( x \) must differ from the local median for it to be replaced with the median. \( \text{nsigma} \) defaults to 3.

\( [y,j] = \text{hampel}(\text{___}) \) 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.

```matlab
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.

```matlab
[y, i, xmedian, xsigma] = hampel(x);
```

Plot the filtered signal and annotate the outliers.

```matlab
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.

```matlab
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.

```matlab
[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.

```matlab
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.

```matlab
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.

```matlab
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.

\[
\begin{align*}
n &= 1:100; \\
x &= \sin(2\pi n/100); \\
x(6) &= 2; \\
x(20) &= -2;
\end{align*}
\]

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 $2 \times 3 + 1 = 7$.
• The points that differ from their window median by more than three standard deviations are considered outliers.

Plot the result.

```matlab
[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 $2 \times 10 + 1 = 21$ and two standard deviations as the criteria for identifying outliers.

```matlab
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')
```

![Graph showing signal, lower and upper limits, and outliers.](image)
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: \( \cos\left(\pi/4*(0:159)\right)+\operatorname{randn}(1,160) \) is a single-channel row-vector signal.

Example: \( \cos\left(\pi./[4;2]*(0:159)\right)'+\operatorname{randn}(160,2) \) 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.

\[ \text{Window length} \]

\[ \begin{array}{ccccccccc}
\bullet & \bullet & \bullet & \bullet & \circ & \bullet & \bullet & \bullet & \bullet \\
\hline
\end{array} \]

\( k \)

\( x_s \)

Data Types: `single` | `double`

\( \text{nsigma} \) — 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 \( \text{nsigma} \) as a real scalar. The function estimates the standard deviation by scaling the local median absolute deviation (MAD) by a factor of

\[ \kappa = \frac{1}{\sqrt{2\operatorname{erfc}^{-1}1/2}} \approx 1.4826. \]

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, ..., x_n` and a sliding window of length `k`, define point-to-point median and standard-deviation estimates using:

- Local median — \( m_i = \text{median}(x_i - k, x_i - k + 1, x_i - k + 2, ..., x_i, ..., x_i + k - 2, x_i + k - 1, x_i + k) \)
• Standard deviation — $\sigma_i = \kappa \text{median}(|x_i - m_i|, ..., |x_i + k - m_i|)$, where
\[
\kappa = \frac{1}{\sqrt{2\text{erfc}^{-1/2}}} \approx 1.4826
\]

The quantity $\sigma_i/\kappa$ 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 $n_\sigma$, 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 $\sigma_i$:

• $i < k + 1$

\[
m_i = \text{median}(x_1, x_2, x_3, ..., x_i, ..., x_i + k - 2, x_i + k - 1, x_i + k)
\]

\[
\sigma_i = \kappa \text{median}(|x_1 - m_1|, ..., |x_i + k - m_i|)
\]

• $i > n - k$

\[
m_i = \text{median}(x_i - k, x_i - k + 1, x_i - k + 2, ..., x_i, ..., x_n - 2, x_n - 1, x_n)
\]

\[
\sigma_i = \kappa \text{median}(|x_i - k - m_i|, ..., |x_n - m_n|)
\]

References


See Also

filloutliers | filter | isoutlier | mad | medfilt1 | median | movmad | movmedian | sgolayfilt
Introduced in R2015b
**hann**

Hann (Hanning) window

**Syntax**

\[ w = \text{hann}(L) \]
\[ w = \text{hann}(L, \text{sflag}) \]

**Description**

\[ w = \text{hann}(L) \] returns an \( L \)-point symmetric Hann window.

\[ w = \text{hann}(L, \text{sflag}) \] returns a Hann window using the window sampling specified by \text{sflag}.

**Examples**

**Hann Window**

Create a 64-point Hann window. Display the result using \text{wvtool}.

\[ L = 64; \]
\[ \text{wvtool(hann(L))} \]
Input Arguments

\( L \) — Window length
positive integer

Window length, specified as a positive integer.
Data Types: single | double

\( s\text{flag} \) — 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:

$$w(n) = 0.5 \left(1 - \cos\left(2\pi \frac{n}{N}\right)\right), \quad 0 \leq n \leq N.$$  

The window length $L = N + 1$.

References


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
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

<table>
<thead>
<tr>
<th>WhaleType</th>
<th>MoanRegions</th>
<th>TrillRegions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member{1}</td>
<td>blue</td>
<td>{3x2 table}</td>
</tr>
<tr>
<td>Member{2}</td>
<td>blue</td>
<td>{3x2 table}</td>
</tr>
</tbody>
</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
**hht**

Hilbert-Huang transform

**Syntax**

```plaintext
hs = hht(imf)
hs = hht(imf,fs)
[hs,f,t] = hht(____)
[hs,f,t,imfinsf,imfinse] = hht(____)
[____] = hht(____,Name,Value)

hht(____)
```

**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.

```matlab
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. 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.

```
emd(q)
```

<table>
<thead>
<tr>
<th>Current IMF</th>
<th>#Sift Iter</th>
<th>Relative Tol</th>
<th>Stop Criterion Hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.0063952</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.1007</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0.01189</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0.0075124</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
</tbody>
</table>

The decomposition stopped because the number of extrema of the residual signal is less...
Compute the IMFs of the signal. Set 'Display' to 0 to hide the table.

```matlab
imf = emd(q,'Display',0);
```

Use the computed IMFs to plot the Hilbert spectrum of the quadratic chirp. Restrict the frequency range to [0, 20] Hz.

```matlab
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 $f_s$.

```matlab
load('sinusoidalSignalExampleData.mat','X','fs')
t = (0:length(X)-1)/fs;
plot(t,X)
xlabel('Time(s)')
```

Hilbert Spectrum
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.

\[
\text{[imf, residual, info] = emd(X,'Interpolation','pchip');}
\]

<table>
<thead>
<tr>
<th>Current IMF</th>
<th>#Sift Iter</th>
<th>Relative Tol</th>
<th>Stop Criterion Hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.026352</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.0039573</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.024838</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0.05929</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
</tbody>
</table>
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*.

```matlab
whaleFile = fullfile(matlabroot,'examples','matlab','bluewhale.au');
[w,fs] = audioread(whaleFile);
whale = timetable(w,'SampleRate',fs);
stackedplot(whale);
```

Use `emd` to visualize the first three intrinsic mode functions (IMFs) and the residual. 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.

```matlab
emd(whale,'MaxNumIMF',3)
```
Current IMF | #Sift Iter | Relative Tol | Stop Criterion Hit
--- | --- | --- | ---
1 | 1 | 0.13523 | SiftMaxRelativeTolerance
2 | 2 | 0.030198 | SiftMaxRelativeTolerance
3 | 2 | 0.01908 | SiftMaxRelativeTolerance

The decomposition stopped because 'MaxNumIMF' was reached.

Compute the first three IMFs of the signal. Set 'Display' to 0 to hide the table.

```matlab
imf = emd(whale,'MaxNumIMF',3,'Display',0);
```

Use the computed IMFs to plot the Hilbert spectrum of the signal. Restrict the frequency range to [0, 1400] Hz.

```matlab
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.

```matlab
[hs,f,t] = hht(imf,'FrequencyLimits',[0 1400]);
mesh(seconds(t),f,hs,'EdgeColor','none','FaceColor','interp');
xlabel('Time');
ylabel('Frequency');
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 $f_s$.

```matlab
load('sinusoidalSignalExampleData.mat','X','fs')
t = (0:length(X)-1)/fs;
plot(t,X)
xlabel('Time(s)')
```

1-1164
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.

\[
[\text{imf, residual, info}] = \text{emd}(X, '\text{Interpolation}', 'pchip');
\]

<table>
<thead>
<tr>
<th>Current IMF</th>
<th>#Sift Iter</th>
<th>Relative Tol</th>
<th>Stop Criterion Hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.026352</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.0039573</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.024838</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0.05929</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
</tbody>
</table>
The decomposition stopped because the number of extrema of the residual signal is less than 'MaxNumExtrema'.

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.

**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.

```matlab
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.

\[
t = 0:1/fs:10-1/fs; \\
y_{\text{Healthy}} = [1 \ 0.5 \ 0.2 \ 0.1 \ 0.05] \times \sin(2\pi f_0 [1 \ 2 \ 3 \ 4 \ 5]' \times t)/5;
\]

A resonance is excited in the bearing vibration halfway through the measurement process.

\[
y_{\text{Healthy}} = (1+1./(1+linspace(-10,10,length(y_{\text{Healthy}})).^4)) \times y_{\text{Healthy}};
\]

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 \( \theta \) is the bearing contact angle. Assume a contact angle of 15° and compute the BPFO.

\[
\begin{align*}
ca &= 15; \\
bpfo &= n * f_0 / 2 * (1 - d/p * \cos(d(ca))) \\
\end{align*}
\]

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.

\[
\begin{align*}
f_{\text{Impact}} &= 3000; \\
t_{\text{Impact}} &= 0:1/fs:5e-3-1/fs; \\
w_{\text{Impact}} &= \text{flattopwin}(\text{length}(t_{\text{Impact} })')/10; \\
x_{\text{Impact}} &= \sin(2*\pi*f_{\text{Impact}}*t_{\text{Impact}}).*w_{\text{Impact}}; \\
\end{align*}
\]

\[
\begin{align*}
tx &= 0:1/bpfo:t\text{(end)}; \\
tx &= [tx; 1.3.^{tx-2}]; \\
\end{align*}
\]

\[
\begin{align*}
n_{\text{Wear}} &= 49000; \\
n_{\text{Samples}} &= 100000; \\
y_{\text{Impact}} &= \text{pulstran}(t,tx',x_{\text{Impact}},fs)/5; \\
y_{\text{Impact}} &= [\text{zeros}(1,n_{\text{Wear}}) ~ y_{\text{Impact}}(1,(n_{\text{Wear}}+1):n_{\text{Samples}})]; \\
\end{align*}
\]

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.

\[
\begin{align*}
y_{\text{BPFO}} &= y_{\text{Impact}} + y_{\text{Healthy}}; \\
x_{\text{LimLeft}} &= 5.0; \\
x_{\text{LimRight}} &= 5.3; \\
y_{\text{Min}} &= -0.6; \\
y_{\text{Max}} &= 0.6; \\
\text{plot}(t,y_{\text{BPFO}}) \\
\text{hold on} \\
[\text{limLeft},\text{limRight}] &= \text{meshgrid}([x_{\text{LimLeft}} x_{\text{LimRight}}],[y_{\text{Min}} y_{\text{Max}}]); \\
\text{plot(limLeft,limRight, '--;')} \\
\text{hold off}
\end{align*}
\]
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 $1/150^2$.

\[
\text{rn} = 150;
\]
\[
\text{yGood} = \text{yHealthy} + \frac{\text{randn(size(yHealthy))}}{\text{rn}};
\]
\[
\text{yBad} = \text{yBPF0} + \frac{\text{randn(size(yHealthy))}}{\text{rn}};
\]

\[
\text{plot(t,yGood,t,yBad)}
\]
\[
\text{xlim([xLimLeft xLimRight])}
\]
\[
\text{legend('Healthy','Damaged')}
\]
Use `emd` to perform an empirical mode decomposition of the healthy bearing signal. Compute the first five intrinsic mode functions (IMFs). 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.

```matlab
imfGood = emd(yGood,'MaxNumIMF',5);
```

<table>
<thead>
<tr>
<th>Current IMF</th>
<th>#Sift Iter</th>
<th>Relative Tol</th>
<th>Stop Criterion Hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>0.017132</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.12694</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>0.14582</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0.011082</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.03463</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
</tbody>
</table>

The decomposition stopped because 'MaxNumIMF' was reached.
Use `emd` without output arguments to visualize the first three modes and the residual. Set 'Display' to 0 to hide the table.

```matlab
emd(yGood,'MaxNumIMF',5,'Display',0)
```

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.

```matlab
imfBad = emd(yBad,'MaxNumIMF',5);
```

<table>
<thead>
<tr>
<th>Current IMF</th>
<th>#Sift Iter</th>
<th>Relative Tol</th>
<th>Stop Criterion Hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.041274</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.16695</td>
<td>SiftMaxRelativeTolerance</td>
</tr>
</tbody>
</table>
emd(yBad,'MaxNumIMF',5,'Display',0)

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 to [0, 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.

```matlab
subplot(2,1,1)
 hht(imfGood(:,1),fs)
 subplot(2,1,2)
 hht(imfGood(:,3),fs,'FrequencyLimits',[0 100])
```
Input Arguments

\texttt{imf} — Intrinsic mode function  
\begin{verbatim}
matrix | timetable
\end{verbatim}

Intrinsic mode function, specified as a matrix or timetable. \texttt{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. \texttt{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 comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

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 10log_{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 \( \text{imf} \) is specified as an array.
- A duration array, if \( \text{imf} \) 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.
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.

**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** 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 **imfinsef**.
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


See Also

emd | hilbert

Topics

“Analytic Signal and Hilbert Transform”
“Hilbert Transform and Instantaneous Frequency”

Introduced in R2018a
highpass

Highpass-filter signals

Syntax

\[ y = \text{highpass}(x, w_{\text{pass}}) \]
\[ y = \text{highpass}(x, f_{\text{pass}}, fs) \]
\[ y = \text{highpass}(xt, f_{\text{pass}}) \]
\[ y = \text{highpass}(\text{___}, \text{Name}, \text{Value}) \]
\[ [y, d] = \text{highpass}(\text{___}) \]
\[ \text{highpass}(\text{___}) \]

Description

\[ y = \text{highpass}(x, w_{\text{pass}}) \] filters the input signal \( x \) using a highpass filter with normalized passband frequency \( w_{\text{pass}} \) in units of \( \pi \) rad/sample. \text{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.

\[ y = \text{highpass}(x, f_{\text{pass}}, fs) \] specifies that \( x \) is sampled at a rate of \( fs \) hertz. \( f_{\text{pass}} \) is the passband frequency of the filter in hertz.

\[ y = \text{highpass}(xt, f_{\text{pass}}) \] highpass-filters the data in timetable \( xt \) using a filter with a passband frequency of \( f_{\text{pass}} \) hertz. The function independently filters all variables in the timetable and all columns inside each variable.

\[ y = \text{highpass}(\text{___}, \text{Name}, \text{Value}) \] specifies additional options for any of the previous syntaxes using name-value pair arguments. You can change the stopband attenuation, the transition band steepness, and the type of impulse response of the filter.

\[ [y, d] = \text{highpass}(\text{___}) \] also returns the \text{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.

```matlab
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 0 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 2 0 2 2 3 0 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.

```matlab
hong = highpass(song,450,fs);

% To hear, type sound(hong/fs)
highpass(song,450,fs)
```
Plot the spectrogram of the melody.

```matlab
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.

```matlab
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);```

Fres = 8.2798 Hz, Tres = 310 ms

- Frequency (kHz)
- Power (dB)
- Time (s)
Compute and plot the frequency responses of the filters.

```matlab
[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')
```

![Frequency Response Graph](image)

```matlab
[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

\textbf{wpass} — Normalized passband frequency
scalar in (0, 1)

Normalized passband frequency, specified as a scalar in the interval (0, 1).

\textbf{fpass} — Passband frequency
scalar in (0, \(f_s/2\))

Passband frequency, specified as a scalar in the interval (0, \(f_s/2\)).

\textbf{fs} — Sample rate
positive real scalar

Sample rate, specified as a positive real scalar.

\textbf{xt} — Input timetable
timetable

Input timetable. \(xt\) must contain increasing, finite, and equally spaced row times of type \texttt{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” (MATLAB).

Example: \texttt{timetable(seconds(0:4)',randn(5,1))} specifies a random variable sampled at 1 Hz for 4 seconds.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of \texttt{Name,Value} arguments. \texttt{Name} is the argument name and \texttt{Value} is the corresponding value. \texttt{Name} must appear inside quotes. You can specify several name and value pair arguments in any order as \texttt{Name1,Value1,...,NameN,ValueN}.

Example: \texttt{ImpulseResponse,'iir','StopbandAttenuation',30} filters the input using a minimum-order IIR filter that attenuates frequencies lower than \(f_{pass}\) by 30 dB.

\textbf{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 \texttt{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:

  1. 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.
  2. 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.
  3. 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.
  4. 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 1-1192 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(d) 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, \( f_{\text{Nyquist}} \), is the highest frequency component of a signal that can be sampled at a given rate without aliasing. \( f_{\text{Nyquist}} \) is 1 (\( \times \pi \) rad/sample) when the input signal has no time information, and \( f_{\text{Nyquist}} \) = \( fs/2 \) hertz when the input signal is a timetable or when you specify a sample rate.
• The *stopband frequency* of the filter, \( f_{\text{stop}} \), 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 \( f_{\text{pass}} - f_{\text{stop}} \).

• 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

\[
W = (1 - s) \times f_{\text{pass}}.
\]

• When 'Steepness' is equal to 0.5, the transition width is 50% of \( f_{\text{pass}} \).
• As 'Steepness' approaches 1, the transition width becomes progressively narrower until it reaches a minimum value of 1% of \( f_{\text{pass}} \).
• The default value of 'Steepness' is 0.85, which corresponds to a transition width that is 15% of \( f_{\text{pass}} \).

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

\[
x = \text{hilbert}(xr)
\]
\[
x = \text{hilbert}(xr, n)
\]

Description

\(x = \text{hilbert}(xr)\) returns the analytic signal, \(x\), from a real data sequence, \(xr\). If \(xr\) is a matrix, then \text{hilbert} finds the analytic signal corresponding to each column.

\(x = \text{hilbert}(xr, n)\) 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 \text{hilbert}.

\[
xr = [1 2 3 4];
\]
\[
x = \text{hilbert}(xr)
\]
\[
x = 1x4 \text{ complex}
\]

\[
1.0000 + 1.0000i \quad 2.0000 - 1.0000i \quad 3.0000 - 1.0000i \quad 4.0000 + 1.0000i
\]

The imaginary part of \(x\) is the Hilbert transform of \(xr\), and the real part is \(xr\) itself.

\[
imx = \text{imag}(x)
\]
\[
imx = 1x4
\]
Analyze the matrix:

\[
\begin{bmatrix}
1 & -1 & -1 & 1
\end{bmatrix}
\]

\[\text{rex} = \text{real}(x)\]
\[\text{rex} = 1 \times 4\]
\[1 \quad 2 \quad 3 \quad 4\]

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 \(\text{fft}(x)\) are purely real.

\[\text{dft} = \text{fft}(x)\]
\[\text{dft} = 1 \times 4 \text{ complex} \]
\[10.0000 + 0.0000i \quad -4.0000 + 4.0000i \quad -2.0000 + 0.0000i \quad 0.0000 + 0.0000i\]

**Analytic Signal and Hilbert Transform**

The \text{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 \text{hilbert} function to compute the analytic signal. Plot it between 0.01 seconds and 0.03 seconds.

\[
\begin{align*}
\text{fs} &= 1e4; \\
\text{t} &= 0:1/\text{fs}:1; \\
\text{x} &= 2.5 + \cos(2\pi*203*t) + \sin(2\pi*721*t) + \cos(2\pi*1001*t); \\
\text{y} &= \text{hilbert}(x); \\
\end{align*}
\]

\[\text{plot(t,real(y),t,imag(y))} \]
\[\text{xlim([0.01 0.03])} \]
\[\text{legend('real','imaginary')} \]
\[\text{title('hilbert Function')}\]
Compute Welch estimates of the power spectral densities of the original sequence and the analytic signal. Divide the sequences into Hamming-windowed, nonoverlapping sections of length 256. Verify that the analytic signal has no power at negative frequencies.

```matlab
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.

```matlab
fo = 60;

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

d = designfilt('hilbertfir','FilterOrder',fo, ... 'TransitionWidth',400,'SampleRate',fs);

% Visualize the frequency response of the filter
freqz(d,1024,fs)
Filter the sinusoidal sequence to approximate the imaginary part of the analytic signal.

\[ hb = \text{filter}(d,x); \]

The group delay of the filter, \( \text{grd} \), is equal to one-half the filter order. Compensate for this delay. Remove the first \( \text{grd} \) samples of the imaginary part and the last \( \text{grd} \) samples of the real part and the time vector. Plot the result between 0.01 seconds and 0.03 seconds.

\[ \text{grd} = \text{fo}/2; \]

\[ y2 = x(1:\text{end}-\text{grd}) + 1j*hb(\text{grd}+1:\text{end}); \]

\[ t2 = t(1:\text{end}-\text{grd}); \]

\[ \text{plot}(t2,\text{real}(y2),t2,\text{imag}(y2)) \]
Estimate the power spectral density (PSD) of the approximate analytic signal and compare it to the hilbert result.

```matlab
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 \( x \) is complex, then \texttt{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 \( x_r \) 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 \( i = 2, 3, \ldots, (n/2) \)
   - 0 for \( i = (n/2)+2, \ldots, n \)
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


**Extended Capabilities**

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

**See Also**

fft | ifft | rceps

*Introduced before R2006a*
icceps

Inverse complex cepstrum

Syntax

\[ x = \text{icceps}(x\hat{a},nd) \]

Description

\[ x = \text{icceps}(x\hat{a},nd) \] returns the inverse complex cepstrum of the real data sequence \( x\hat{a} \), 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.

\[
\begin{align*}
Fs &= 100; \\
t &= 0:1/Fs:1.27; \\
s1 &= \sin(2*\pi*45*t); \\
s2 &= s1 + 0.5*[\text{zeros}(1,20) \ s1(1:108)]; \\
c &= \text{cceps}(s2);
\end{align*}
\]

Compute the inverse complex cepstrum. Plot the echo data and its inverse complex cepstrum.

\[
\begin{align*}
x &= \text{icceps}(c); \\
plot(t,s2,t,x,'r--') \\
xlabel('Time (s)') \\
legend('Echo signal','Inverse complex cepstrum')
\end{align*}
\]
**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 \( \text{round}(\text{unwrap}(\text{angle}(\text{fft}(x)))/\pi) \) corresponding to \( \pi \) 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**


**See Also**

cceps | hilbert | rceps | unwrap

**Introduced before R2006a**
idct
Inverse discrete cosine transform

Syntax

\begin{align*}
x &= \text{idct}(y) \\
x &= \text{idct}(y,n) \\
x &= \text{idct}(y,n,\text{dim}) \\
y &= \text{idct}(\_\_, 'Type', \text{dcttype})
\end{align*}

Description

\(x = \text{idct}(y)\) 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 \text{idct} operates along the first array dimension with size greater than 1.

\(x = \text{idct}(y,n)\) zero-pads or truncates the relevant dimension of \(y\) to length \(n\) before transforming.

\(x = \text{idct}(y,n,\text{dim})\) computes the transform along dimension \(\text{dim}\). To input a dimension and use the default value of \(n\), specify the second argument as empty, \([]\).

\(y = \text{idct}(\_\_, 'Type', \text{dcttype})\) specifies the type of inverse discrete cosine transform to compute. See “Inverse Discrete Cosine Transform” on page 1-1213 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.

\[ s = \text{randn}(1000,1); \]

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: \( \text{dct}(\sin(2\pi(0:255)/4)) \) specifies the discrete cosine transform of a sinusoid.

Example: \( \text{dct}(\sin(2\pi*[0.1;0.3]*(0:39))') \) 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

\( \text{dim} \) — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar.

Data Types: single | double

\( \text{dcttype} \) — 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. In particular, DCT-1 and DCT-4 are their own inverses, and DCT-2 and DCT-3 are inverses of each other.

References


**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™.

This function fully supports GPU arrays. 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

\[ x = \text{ifsst}(s) \]
\[ x = \text{ifsst}(s, \text{window}) \]
\[ x = \text{ifsst}(s, \text{window}, f, \text{freqrange}) \]
\[ x = \text{ifsst}(s, \text{window}, \text{iridge}) \]
\[ x = \text{ifsst}(s, \text{window}, \text{iridge}, '\text{NumFrequencyBins}', \text{nbins}) \]

Description

\( x = \text{ifsst}(s) \) returns the inverse Fourier synchrosqueezed transform of \( s \). \( x \) is reconstructed using the entire time-frequency plane in \( s \).

\( x = \text{ifsst}(s, \text{window}) \) reconstructs the signal whose Fourier synchrosqueezed transform was computed using \( \text{window} \).

\( x = \text{ifsst}(s, \text{window}, f, \text{freqrange}) \) inverts the synchrosqueezed transform assuming it was sampled at the frequencies \( f \), which lie within \( \text{freqrange} \). The synchrosqueezed transform is inverted for the bins in \( s \) whose frequencies are within \( \text{freqrange} \).

\( x = \text{ifsst}(s, \text{window}, \text{iridge}) \) inverts the synchrosqueezed transform along the time-frequency ridges specified by the index vector or matrix \( \text{iridge} \). If \( \text{iridge} \) is a matrix, then \( \text{ifsst} \) initially performs the inversion along the first column of \( \text{iridge} \) and then proceeds iteratively along the subsequent columns. The output is a vector or matrix with the same size as \( \text{iridge} \).

\( x = \text{ifsst}(s, \text{window}, \text{iridge}, '\text{NumFrequencyBins}', \text{nbins}) \) specifies the number of frequency bins around the indices in \( \text{iridge} \) 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.

```matlab
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.

```matlab
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 $\ell_\infty$ norm of the difference between the original signal and the inverse transform.

$$L_{\infty} = \text{norm}(\text{abs}(\text{mtlb} - \text{xrec}), \text{Inf})$$

$L_{\infty} = 1.9762 \times 10^{-14}$

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.

\[ t1 = t(1:nSamp/2); \]
\[ x11 = \sin(2\pi*400*t1); \]
\[ x12 = \text{chirp}(t1-t1(nSamp/4),150,nSamp/Fs,1750,'\text{quadratic}'); \]
\[ x1 = x11+x12; \]

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.

\[ t2 = t(nSamp/2+1:nSamp); \]
\[ x21 = \text{chirp}(t2,400,nSamp/Fs,100); \]
\[ x22 = \text{chirp}(t2,550,nSamp/Fs,250); \]
\[ x2 = x21+x22; \]

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 \texttt{fsst} to display the result.

\[ \text{sig} = [x1;x2]; \]
\[ \text{wind} = \text{kaiser}(256,120); \]
\[ [\text{sigtr,ftr,ttr}] = \text{fsst(sig,Fs,wind);} \]
\[ \text{fsst(sig,Fs,wind,'yaxis')} \]
Invert the transform to reconstruct the function. Plot the original and inverted signals and the difference between them.

```matlab
x = ifsst(sigtr,wind);
plot(t,sig,t,x,t,x-sig)
legend('Original','Reconstructed','Difference')
```
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™.

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 1-1044

Introduced in R2016b
**ifwht**

Inverse Fast Walsh-Hadamard transform

**Syntax**

\[
\begin{align*}
y &= \text{ifwht}(x) \\
y &= \text{ifwht}(x, n) \\
y &= \text{ifwht}(x, n, \text{ordering})
\end{align*}
\]

**Description**

\( y = \text{ifwht}(x) \) returns the coefficients of the inverse discrete fast Walsh-Hadamard transform of the input \( x \). If \( x \) is a matrix, the inverse fast Walsh-Hadamard transform is calculated on each column of \( x \). The inverse fast Walsh-Hadamard transform operates only on signals with length equal to a power of 2. If the length of \( x \) is less than a power of 2, its length is padded with zeros to the next greater power of two before processing.

\( y = \text{ifwht}(x, n) \) returns the \( n \)-point inverse discrete Walsh-Hadamard transform, where \( n \) must be a power of 2.

\( y = \text{ifwht}(x, n, \text{ordering}) \) specifies the ordering to use for the returned inverse Walsh-Hadamard transform coefficients. To specify the ordering, you must enter a value for the length \( n \) or, to use the default behavior, specify an empty vector ([ ]) for \( n \). Valid values for the ordering are the following:

<table>
<thead>
<tr>
<th>Ordering</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'sequency'</td>
<td>Coefficients in order of ascending sequency value, where each row has an additional zero crossing. This is the default ordering.</td>
</tr>
<tr>
<td>'hadamard'</td>
<td>Coefficients in normal Hadamard order.</td>
</tr>
<tr>
<td>'dyadic'</td>
<td>Coefficients in Gray code order, where a single bit change occurs from one coefficient to the next.</td>
</tr>
</tbody>
</table>
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.

```matlab
ex = 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.

```matlab
y = fwht(x);
```

```matlab
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.

```matlab
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.

**Algorithms**

The inverse fast Walsh-Hadamard transform algorithm is similar to the Cooley-Tukey algorithm used for the inverse FFT. Both use a butterfly structure to determine the transform coefficients. See the references for details.
References


See Also
dct | fft | fwhit | idct | 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. If you leave out the argument fs, or specify fs as the empty vector [], it takes the default value of 1 Hz.

[bz,az] = impinvar(b,a,fs,tol) uses the tolerance specified by tol to determine whether poles are repeated. A larger tolerance increases the likelihood that impinvar interprets closely located poles as multiplicities (repeated ones). The default is 0.001, or 0.1% of a pole's magnitude. The accuracy of the pole values is still limited to the accuracy obtainable by the roots function.

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);
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.

```matlab
freqz(bz,az,1024,fs)
```
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

$$H(s) = \sum_k \frac{r_k}{s - p_k} \iff h(t) = \sum_k r_k e^{p_k t}.$$
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.

```matlab
[r,p] = residue(b,a);
t = linspace(0,4,1000);
h = real(r.'*exp(p.*t)/fs);
hold on
plot(t,h)
hold off
```

![Impulse Response](image-url)
Algorithms

`impinvar` performs the impulse-invariant method of analog-to-digital transfer function conversion discussed in reference [1]:

1. It finds the partial fraction expansion of the system represented by `b` and `a`.
2. It replaces the poles `p` by the poles `\exp(p/\text{fs})`.
3. It finds the transfer function coefficients of the system from the residues from step 1 and the poles from step 2.

References


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 \texttt{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.

\texttt{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.

\[ [b,a] = \text{ellip}(4,0.5,20,0.4); \]
\[ \text{impz}(b,a,50) \]
Design the same filter using `designfilt`. Plot the first 50 samples of its impulse response.

```matlab
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.

```matlab
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.

```matlab
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

\[
H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})} = \frac{b(1) + b(2) e^{-j\omega} + b(3) e^{-j2\omega} + \ldots + b(M) e^{-j(M-1)\omega}}{a(1) + a(2) e^{-j\omega} + a(3) e^{-j2\omega} + \ldots + a(N) e^{-j(N-1)\omega}}.
\]
Example: \( b = [1 3 3 1]/6 \) and \( a = [3 0 1 0]/3 \) specify a third-order Butterworth filter with normalized 3 dB frequency 0.5\( \pi \) rad/sample.

Data Types: double | single
Complex Number Support: Yes

**sos — Second-order section coefficients**

Second-order section coefficients, specified as a matrix. \( \text{sos} \) 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 \( \text{sos} \) corresponds to the coefficients of a second-order (biquad) filter. The \( i \)th row of \( \text{sos} \) corresponds to \([b_i(1) \ b_i(2) \ b_i(3) \ a_i(1) \ a_i(2) \ a_i(3)]\).

Example: \( s = [2 \ 4 \ 2 \ 6 \ 0 \ 2;3 \ 3 \ 0 \ 6 \ 0 \ 0] \) specifies a third-order Butterworth filter with normalized 3 dB frequency 0.5\( \pi \) rad/sample.

Data Types: double | single
Complex Number Support: Yes

**d — Digital filter**

digitalFilter object

Digital filter, specified as a digitalFilter object. Use designfilt to generate a digital filter based on frequency-response specifications.

Example: \( d = \text{designfilt}('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5) \) specifies a third-order Butterworth filter with normalized 3 dB frequency 0.5\( \pi \) 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, \( \text{impz} \) computes the first \( n \) samples of the impulse response and returns \( t \) as \((0:n-1)\).
- If \( n \) is a vector of nonnegative integers, \( \text{impz} \) computes the impulse response at the locations specified in the vector.
- If \( n \) is an empty vector, \( \text{impz} \) computes the number of samples automatically. See “Algorithms” on page 1-1244 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 \( 5 \times 10^{-5} \) 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 \( 5 \times 10^{-5} \) 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.

**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(hd)
len = impzlength(____,tol)

Description

len = impzlength(b,a) returns the impulse response length for the causal discrete-time filter with the rational system function specified by the numerator, b, and denominator, a, polynomials in $z^{-1}$. For stable IIR filters, len is the effective impulse response sequence length. Terms in the IIR filter’s impulse response after the len-th term are essentially zero.

len = impzlength(sos) returns the effective impulse response length for the IIR filter specified by the second order sections matrix, sos. sos is a K-by-6 matrix, where the number of sections, K, must be greater than or equal to 2. If the number of sections is less than 2, impzlength considers the input to be the numerator vector, b. Each row of sos corresponds to the coefficients of a second order (biquad) filter. The $i$th row of the sos matrix corresponds to $[bi(1)\ bi(2)\ bi(3)\ ai(1)\ ai(2)\ ai(3)]$.

len = impzlength(d) returns the impulse response length for the digital filter, d. Use designfilt to generate d based on frequency-response specifications.

len = impzlength(hd) returns the impulse response length for the dfilt filter object, hd. You can also input an array of filter objects. If hd is an array of filter objects, each column of len is the impulse response length of the corresponding filter object.

len = impzlength(____,tol) specifies a tolerance for estimating the effective length of an IIR filter’s impulse response. By default, tol is $5e-5$. Increasing the value of tol
estimates a shorter effective length for an IIR filter’s impulse response. Decreasing the value of tol produces a longer effective length for an IIR filter’s impulse response.

**Examples**

**IIR Filter Effective Impulse Response Length — Coefficients**

Create a lowpass allpole IIR filter with a pole at 0.9. Calculate the effective impulse response length. Obtain the impulse response. Plot the result.

```matlab
b = 1;
a = [1 -0.9];
len = impzlength(b,a)

len = 93
```

```matlab
[h,t] = impz(b,a);
stem(t,h)
```
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 $0.4\pi$ rad/sample. Specify 1 dB of passband ripple and 60 dB of stopband attenuation. Determine the effective impulse response sequence length and visualize it.

```matlab
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: \( b = \text{fir1}(20,0.25) \)

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 Kth row of the matrix, sos, are ordered as follows.

\[ [B_k(1) \ B_k(2) \ B_k(3) \ A_k(1) \ A_k(2) \ A_k(3)] \]

The frequency response of the filter is the system function evaluated on the unit circle with

\[ z = e^{j2\pi f}. \]

**d — Digital filter**
digitalFilter object

Digital filter, specified as a digitalFilter object. Use designfilt to generate a digital filter based on frequency-response specifications.

Example: d =
```
designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)
```
specifies a third-order Butterworth filter with normalized 3-dB frequency 0.5π rad/sample.

**hd — Filter object**
dfilt object

Filter object, specified as a dfilt object.

**tol — Tolerance for IIR filter effective impulse response length**
5e-5 (default) | positive scalar
Tolerance for IIR filter effective impulse response length, specified as a positive number. The tolerance determines the term in the absolutely summable sequence after which subsequent terms are considered to be 0. The default tolerance is $5e^{-5}$. Increasing the tolerance returns a shorter effective impulse response sequence length. Decreasing the tolerance returns a longer effective impulse response sequence length.

**Output Arguments**

len — Length of impulse response
positive integer

Length of the impulse response, specified as a positive integer. For stable IIR filters with absolutely summable impulse responses, `impzlength` returns an effective length for the impulse response beyond which the coefficients are essentially zero. You can control this cutoff point by specifying the optional `tol` input argument.

**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.
See Also
designfilt | digitalFilter | impz | zp2sos

Introduced in R2013a
**info**

Information about digital filter

**Syntax**

\[ s = \text{info}(d) \]

**Description**

\[ s = \text{info}(d) \] 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 \( 0.4\pi \text{ rad/sample} \) and normalized stopband frequency \( 0.45\pi \text{ rad/sample} \). Obtain information about the filter just designed.

\[ d = \text{designfilt}('\text{lowpassfir}', '\text{PassbandFrequency}', 0.4, '\text{StopbandFrequency}', 0.45); \]
\[ s = \text{info}(d) \]

\[ s = 17x44 \text{ char array} \]

- '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
'Stopband Atten.     : 60 dB                    
'Stopband Edge       : 0.45                     
'Passband Edge       : 0.4                      
'Passband Ripple     : 1 dB

Input Arguments

d — Digital filter
digitalFilter object

Digital filter, specified as a digitalFilter object. Use designfilt to generate a
digital filter based on frequency-response specifications.

Example: d =
designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)
specifies a third-order Butterworth filter with normalized 3 dB frequency 0.5π rad/sample.

Output Arguments

s — Information table
character array

Information table, returned as a character array.

See Also

designfilt | digitalFilter

Introduced in R2014a
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));
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.

```matlab
instfreq(s, fs)
```
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.

```matlab
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)
```

![Instantaneous Frequency Estimate](image)

1-1262
Use the `pspectrum` function with default settings to estimate the power spectrum of the signal. Use the estimate to compute the instantaneous frequency.

```matlab
[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.

```matlab
[s,sf,st] = fsst(q,fs,hann(500));
instfreq(abs(s).^2,sf,st)
```
Compare the instantaneous frequencies found using the two different methods.

\[
[psf, pst] = \text{instfreq}(p, f, t);
[fsf, fst] = \text{instfreq}(\text{abs}(s).^2, sf, st);
\]

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.

\[
fs = 1000; \\
t = (0:1/fs:0.3-1/fs)'; \\
x = \sin(2\pi*200*t) + \text{randn(size}(t))/4;
\]
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.

```matlab
[p,fd,td] = pspectrum(x,t,'spectrogram','FrequencyResolution',25);
instfreq(p,fd,td)
```
**Input Arguments**

\( x \) — Input signal  
vector | matrix

Input signal, specified as a vector or matrix. If \( x \) is a vector, then \texttt{instfreq} treats it as a single channel. If \( x \) is a matrix, then \texttt{instfreq} computes the instantaneous frequency independently for each column and returns the result in the corresponding column of \texttt{ifq}.

Example: \( \sin(2\pi(0:127)/16) + \text{randn}(1,128)/100 \) 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” (MATLAB).

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 comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: 'Method','hilbert','FrequencyLimits',[25 50] computes the instantaneous frequency in the range from 25 Hz to 50 Hz by differentiating the analytic signal of the input.

**FrequencyLimits — Frequency range**

\([0 \: fs/2]\) (default) | two-element vector in hertz

Frequency range, specified as the comma-separated pair consisting of 'FrequencyLimits' and a two-element vector in hertz. 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 \( \text{instfreq} \) 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 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 \( \text{instfreq} \) 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 $\phi$ is the phase of the analytic signal of the input.

References


See Also

`hilbert` | `pspectrum`

Topics

“Hilbert Transform and Instantaneous Frequency”
“Instantaneous Frequency of Complex Chirp”
Introduced in R2018a
 interp

Interpolation — increase sample rate by integer factor

Syntax

\[
y = \text{interp}(x, r) \\
y = \text{interp}(x, r, n, \text{cutoff}) \\
[y, b] = \text{interp}(x, r, n, \text{cutoff})
\]

Description

\( y = \text{interp}(x, r) \) increases the sample rate of \( x \), the input signal, by a factor of \( r \).

\( y = \text{interp}(x, r, n, \text{cutoff}) \) specifies two additional values:

- \( n \) is half the number of original sample values used to interpolate the expanded signal.
- \( \text{cutoff} \) is the normalized cutoff frequency of the input signal, specified as a fraction of the Nyquist frequency.

\([y, b] = \text{interp}(x, r, n, \text{cutoff})\) 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.

\[
t = 0:0.001:1; \\
x = \sin(2\pi * 30 * t) + \sin(2\pi * 60 * t); \\
y = \text{interp}(x, 4);
\]

Plot the original and interpolated signals.
subplot 211
stem(0:30,x(1:31),'filled','markersize',3)
grid on
xlabel 'Sample number', ylabel Original
subplot 212
stem(0:120,y(1:121),'filled','markersize',3)
grid on
xlabel 'Sample number', ylabel Interpolated
Input Arguments

x — Input signal
type

Input signal, specified as a vector.
Data Types: double | single

r — Interpolation factor
type

Interpolation factor, specified as a positive integer.
Data Types: double | single

n — Half the number of input samples used for interpolation

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 × n × r + 1.
Data Types: double | single

cutoff — Normalized cutoff frequency

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
type

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


See Also
decimate | downsample | interp1 | intfilt | resample | spline | upfirdn | upsample

Introduced before R2006a
**intfilt**

Interpolation FIR filter design

**Syntax**

\[ b = \text{intfilt}(l,p,\alpha) \]
\[ b = \text{intfilt}(l,n,'\text{Lagrange}') \]

**Description**

\[ b = \text{intfilt}(l,p,\alpha) \] designs a linear phase FIR filter that performs ideal bandlimited interpolation using the nearest \(2*p\) nonzero samples, when used on a sequence interleaved with \(l-1\) consecutive zeros every \(l\) samples. It assumes an original bandlimitedness of \(\alpha\) times the Nyquist frequency. The returned filter is identical to that used by \text{interp}. \(b\) is length \(2*l*p-1\).

\(\alpha\) is inversely proportional to the transition bandwidth of the filter and it also affects the bandwidth of the don't-care regions in the stopband. Specifying \(\alpha\) allows you to specify how much of the Nyquist interval your input signal occupies. This is beneficial, particularly for signals to be interpolated, because it allows you to increase the transition bandwidth without affecting the interpolation and results in better stopband attenuation for a given \(l\) and \(p\). If you set \(\alpha\) to 1, your signal is assumed to occupy the entire Nyquist interval. Setting \(\alpha\) to less than one allows for don't-care regions in the stopband. For example, if your input occupies half the Nyquist interval, you could set \(\alpha\) to 0.5.

\[ b = \text{intfilt}(l,n,'\text{Lagrange}') \] designs an FIR filter that performs \(n\)th-order Lagrange polynomial interpolation on a sequence interleaved with \(l-1\) consecutive zeros every \(l\) samples. \(b\) has length \((n+1)*l\) for \(n\) even, and length \((n+1)*l-1\) for \(n\) odd. If both \(n\) and \(l\) are even, the filter designed is not linear phase.

Both types of filters are basically lowpass and have a gain of \(l\) in the passband.

**Examples**
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 \(2 \times 2\) samples in the interpolation.

\[
\text{upfac} = 7; \\
\alpha = 0.5; \\
h1 = \text{intfilt}(\text{upfac},2,\alpha);
\]

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.

\[
\text{lowp} = \text{fir1}(40,\alpha);
\]

\[
\text{rng('default')} \\
x = \text{filter}(	ext{lowp},1,\text{randn}(200,1));
\]

Increase the sample rate of the signal by inserting zeros between each pair of samples of \(x\).

\[
xr = \text{upsample}(x,\text{upfac});
\]

Use the \text{filter} function to produce an interpolated signal.

\[
y = \text{filter}(h1,1,xr);
\]

Compensate for the delay introduced by the filter. Plot the original and interpolated signals.

\[
delay = \text{mean}(	ext{grpdelay}(h1)); \\
y(1:delay) = [];
\]

\[
\text{stem}(1:\text{upfac}:\text{upfac}*\text{length}(x),x) \\
\text{hold on} \\
\text{plot}(y) \\
\text{xlim}([400 \ 700])
\]
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.

```matlab
h2 = intfilt(upfac,1,'Lagrange');

y2 = filter(h2,1,xr);
y2(1:floor(mean(grpdelay(h2)))) = [];
```
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.
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**

\[
[b,a] = \text{invfreqs}(h,w,n,m)
\]
\[
[b,a] = \text{invfreqs}(h,w,n,m,wt)
\]
\[
[b,a] = \text{invfreqs}(h,w,n,m,wt,iter)
\]
\[
[b,a] = \text{invfreqs}(h,w,n,m,wt,iter,tol)
\]
\[
[b,a] = \text{invfreqs}(h,w,n,m,wt,iter,tol,'trace')
\]
\[
[b,a] = \text{invfreqs}(h,w,'\text{complex}',n,m,...)
\]

**Description**

invfreqs is the inverse operation of freqs. It finds a continuous-time transfer function that corresponds to a given complex frequency response. From a laboratory analysis standpoint, invfreqs is useful in converting magnitude and phase data into transfer functions.

\[
[b,a] = \text{invfreqs}(h,w,n,m)
\]

returns the real numerator and denominator coefficient vectors \(b\) and \(a\) of the transfer function

\[
H(s) = \frac{B(s)}{A(s)} = \frac{b(1)s^n + b(2)s^{n-1} + \ldots + b(n+1)}{a(1)s^m + a(2)s^{m-1} + \ldots + a(m+1)}
\]

whose complex frequency response is given in vector \(h\) at the frequency points specified in vector \(w\). Scalars \(n\) and \(m\) specify the desired orders of the numerator and denominator polynomials.

The length of \(h\) must be the same as the length of \(w\). invfreqs uses \(\text{conj}(h)\) at \(-w\) to ensure the proper frequency domain symmetry for a real filter.

\[
[b,a] = \text{invfreqs}(h,w,n,m,wt)
\]

weights the fit-errors versus frequency, where \(wt\) is a vector of weighting factors the same length as \(w\).

\[
[b,a] = \text{invfreqs}(h,w,n,m,wt,iter)
\]

and
invfreqs(h,w,n,m,wt,iter,tol) provide a superior algorithm that guarantees stability of the resulting linear system and searches for the best fit using a numerical, iterative scheme. The iter parameter tells invfreqs to end the iteration when the solution has converged, or after iter iterations, whichever comes first. invfreqs defines convergence as occurring when the norm of the (modified) gradient vector is less than tol, where tol is an optional parameter that defaults to 0.01. To obtain a weight vector of all ones, use

invfreqs(h,w,n,m,[],iter,tol)

[b,a] = invfreqs(h,w,n,m,wt,iter,tol,'trace') displays a textual progress report of the iteration.

[b,a] = invfreqs(h,w,'complex',n,m,...) creates a complex filter. In this case no symmetry is enforced, and the frequency is specified in radians between $-\pi$ and $\pi$.

### Examples

#### Transfer Function to Frequency Response Conversion

Convert a simple transfer function to frequency-response data and then back to the original filter coefficients.

```plaintext
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 = 1x5
    1.0000    2.0000    3.0000    2.0000    3.0000

aa = 1x6
    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.
Use the iterative algorithm of `invfreqs` to find a stable approximation to the system.

```
[bbb,aaa] = invfreqs(h,w,4,5,[],30)
```

```plaintext
bbb = 1x5
   0.6816  2.1015  2.6694  0.9113 -0.1218

aaa = 1x6
   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.

```matlab
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);

dfreqs(b,a)
**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, \texttt{invfreqs} uses an equation error method to identify the best model from the data. This finds \( b \) and \( a \) in

\[
\min_{b,a} \sum_{k=1}^{n} \left| wt(k)h(k)A(w(k)) - B(w(k)) \right|^2
\]

by creating a system of linear equations and solving them with the MATLAB \( \setminus \) 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.

\[
\min_{b,a} \sum_{k=1}^{n} \left| wt(k) \left( h(k) - \frac{B(w(k))}{A(w(k))} \right) \right|^2
\]

**References**


See Also
freqs | freqz | invfreqz | prony

Introduced before R2006a
invfreqz

Identify discrete-time filter parameters from frequency response data

Syntax

\[ [b,a] = \text{invfreqz}(h,w,n,m) \]
\[ [b,a] = \text{invfreqz}(h,w,n,m,wt) \]
\[ [b,a] = \text{invfreqz}(h,w,n,m,wt,\text{iter}) \]
\[ [b,a] = \text{invfreqz}(h,w,n,m,wt,\text{iter},\text{tol}) \]
\[ [b,a] = \text{invfreqz}(h,w,n,m,wt,\text{iter},\text{tol},'\text{trace}') \]
\[ [b,a] = \text{invfreqz}(h,w,'\text{complex'},n,m,...) \]

Description

invfreqz is the inverse operation of freqz; it finds a discrete-time transfer function that corresponds to a given complex frequency response. From a laboratory analysis standpoint, invfreqz can be used to convert magnitude and phase data into transfer functions.

\[ [b,a] = \text{invfreqz}(h,w,n,m) \]
returns the real numerator and denominator coefficients in vectors \( b \) and \( a \) of the transfer function

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

whose complex frequency response is given in vector \( h \) at the frequency points specified in vector \( w \). Scalars \( n \) and \( m \) specify the desired orders of the numerator and denominator polynomials.

Frequency is specified in radians between 0 and \( \pi \), and the length of \( h \) must be the same as the length of \( w \). invfreqz uses \( \text{conj}(h) \) at \( -w \) to ensure the proper frequency domain symmetry for a real filter.

\[ [b,a] = \text{invfreqz}(h,w,n,m,wt) \]
weights the fit-errors versus frequency, where \( wt \) is a vector of weighting factors the same length as \( w \).
\[ [b,a] = \text{invfreqz}(h,w,n,m,wt,\text{iter}) \] and
\[ [b,a] = \text{invfreqz}(h,w,n,m,wt,\text{iter},\text{tol}) \]

provide a superior algorithm that guarantees stability of the resulting linear system and searches for the best fit using a numerical, iterative scheme. The \text{iter} parameter tells \text{invfreqz} to end the iteration when the solution has converged, or after \text{iter} iterations, whichever comes first. \text{invfreqz} defines convergence as occurring when the norm of the (modified) gradient vector is less than \text{tol}, where \text{tol} is an optional parameter that defaults to 0.01. To obtain a weight vector of all ones, use
\[ \text{invfreqz}(h,w,n,m,[],\text{iter},\text{tol}) \]
\[ [b,a] = \text{invfreqz}(h,w,n,m,wt,\text{iter},\text{tol},'\text{trace}') \]

displays a textual progress report of the iteration.

\[ [b,a] = \text{invfreqz}(h,w,'\text{complex}',n,m,...) \]
creates a complex filter. In this case no symmetry is enforced, and the frequency is specified in radians between -\pi and \pi.

**Examples**

**Stable Approximate Transfer Function**

Convert a simple transfer function to frequency response data and then back to the original filter coefficients. Sketch the zeros and poles of the function.

\[
a = [1 2 3 2 1 4];
b = [1 2 3 2 3];
\]
\[
[h,w] = \text{freqz}(b,a,64);
[bb,aa] = \text{invfreqz}(h,w,4,5)
\]

\[
bb = 1\times5
\]

\[
\begin{array}{ccccc}
1.0000 & 2.0000 & 3.0000 & 2.0000 & 3.0000 \\
\end{array}
\]

\[
aa = 1\times6
\]

\[
\begin{array}{ccccc}
1.0000 & 2.0000 & 3.0000 & 2.0000 & 1.0000 & 4.0000 \\
\end{array}
\]
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.

\[
[\text{bbb}, \text{aaa}] = \text{invfreqz}(\text{h}, \text{w}, 4, 5, [], 30)
\]

\[
\text{bbb} = 1 \times 5
\]

\[
\begin{array}{ccccc}
0.2427 & 0.2788 & 0.0069 & 0.0971 & 0.1980 \\
\end{array}
\]
Algorithms

By default, `invfreqz` uses an equation error method to identify the best model from the data. This finds b and a in
\[
\min_{b, a} \sum_{k=1}^{n} wt(k) |h(k)A(w(k)) - B(w(k))|^2
\]

by creating a system of linear equations and solving them with the MATLAB \ operator. Here \(A(\omega(k))\) and \(B(\omega(k))\) are the Fourier transforms of the polynomials \(a\) 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


See Also
frevs | 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)
flag = isallpass(hd)

Description

flag = isallpass(b,a) returns a logical output, flag, equal to true if the filter specified by numerator coefficients, b, and denominator coefficients, a, is an allpass filter. If the filter is not an allpass filter, flag is equal to false.

flag = isallpass(sos) returns true if the filter specified by second order sections matrix, sos, is an allpass filter. sos is a K-by-6 matrix, where the number of sections, K, must be greater than or equal to 2. Each row of sos corresponds to the coefficients of a second order (biquad) filter. The ith 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.
### Details for Fixed-Point Arithmetic

<table>
<thead>
<tr>
<th>System Object State</th>
<th>Coefficient Data Type</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlocked</td>
<td>'Same as input'</td>
<td>The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.</td>
</tr>
<tr>
<td>Unlocked</td>
<td>'Custom'</td>
<td>The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.</td>
</tr>
<tr>
<td>Locked</td>
<td>'Same as input'</td>
<td>When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.</td>
</tr>
<tr>
<td>Locked</td>
<td>'Custom'</td>
<td>The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.</td>
</tr>
</tbody>
</table>

*flag = isallpass(hd)* returns true if the filter object *hd* is an allpass filter.

### Examples

#### Allpass Filters

Create an allpass filter and verify that the frequency response is allpass.
b = [1/3 1/4 1/5 1];
a = fliplr(b);
flag = isallpass(b,a)

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

\[
\begin{align*}
tf &= \text{iscola}(\text{window},noverlap) \\
tf &= \text{iscola}(\text{window},noverlap,\text{method}) \\
[tf,m] &= \text{iscola}(\_\_\_\_) \\
[tf,m,\text{maxDeviation}] &= \text{iscola}(\_\_\_\_)
\end{align*}
\]

Description

\(\text{tf} = \text{iscola}(\text{window},noverlap)\) checks that the specified window and overlap satisfy the Constant Overlap-Add (COLA) constraint to ensure that the inverse short-time Fourier transform results in perfect reconstruction for nonmodified spectra.

\(\text{tf} = \text{iscola}(\text{window},noverlap,\text{method})\) specifies the inversion method to use.

\([\text{tf},m] = \text{iscola}(\_\_\_)\) also returns the median of the COLA summation. You can use these output arguments with any of the previous input syntaxes.

\([\text{tf},m,\text{maxDeviation}] = \text{iscola}(\_\_\_\_)\) 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.

\[
\begin{align*}
\text{win} &= \sqrt{\text{hann}(120,'\text{periodic}')}; \\
\text{noverlap} &= 60;
\end{align*}
\]
Check whether the window is COLA compliant with a 50% overlap.

```matlab
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'.

```matlab
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.

```matlab
[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:

\[ x(n) = \sum_{m = -\infty}^{\infty} \int_{-1/2}^{1/2} X_m(f)e^{j2\pi fn} df \]

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 \( \sum_{m = -\infty}^{\infty} g^{d+1}(n - mR) = c \forall n \in \mathbb{Z} \) 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.
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

\[
\sum_{m=-\infty}^{\infty} g^{a+1}(n - mR) = c \quad \forall n \in \mathbb{Z},
\]

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

\[
G(f_k) = 0, \quad k = 1, 2, \ldots, R - 1, \quad f_k \equiv \frac{k}{R}.
\]

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

\[
G(f) = 0, \quad f \geq \frac{1}{2R}.
\]

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 \texttt{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

\[
\sum_{m=-\infty}^{\infty} g^{a+1}(n - mR) = c \quad \forall n \in \mathbb{Z}
\]

for weighted overlap-add (WOLA), and \(a = 0\) for overlap-add (OLA). By default, \texttt{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 \( k = \frac{(\text{length}(x) - \text{noverlap})}{(\text{length}(\text{window}) - \text{noverlap})} \) 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


Extended Capabilities

**C/C++ Code Generation**

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

Functions
bartlett | istft | pspectrum | stft

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.

```matlab
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)
flag = islinphase(h)

Description

flag = islinphase(b,a) returns a logical output, flag, equal to true if the filter coefficients in b and a define a linear phase filter. flag is equal to false if the filter does not have linear phase.

flag = islinphase(sos) returns true if the filter specified by second order sections matrix, sos, has linear phase. sos is a K-by-6 matrix, where the number of sections, K, must be greater than or equal to 2. Each row of sos corresponds to the coefficients of a second order (biquad) filter. The ith 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).
**Details for Fixed-Point Arithmetic**

<table>
<thead>
<tr>
<th>System Object State</th>
<th>Coefficient Data Type</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlocked</td>
<td>'Same as input'</td>
<td>The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.</td>
</tr>
<tr>
<td>Unlocked</td>
<td>'Custom'</td>
<td>The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsData</code> property.</td>
</tr>
<tr>
<td>Locked</td>
<td>'Same as input'</td>
<td>When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.</td>
</tr>
<tr>
<td>Locked</td>
<td>'Custom'</td>
<td>The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsData</code> property.</td>
</tr>
</tbody>
</table>

flag = islinphase(h) determines if the dfilt filter object h has linear phase.

**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.

```matlab
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.

```matlab
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)
flag = isminphase(h)

Description

A filter is minimum phase when all the zeros of its transfer function are on or inside the unit circle, or the numerator is a scalar. An equivalent definition for a minimum phase filter is a causal and stable system with a causal and stable inverse.

flag = isminphase(b,a) returns a logical output, flag, equal to true if the filter specified by numerator coefficients, b, and denominator coefficients, a, is a minimum phase filter.

flag = isminphase(sos) returns true if the filter specified by second order sections matrix, sos, is minimum phase. sos is a K-by-6 matrix, where the number of sections, K, must be greater than or equal to 2. Each row of sos corresponds to the coefficients of a second order (biquad) filter. The ith row of the sos matrix corresponds to [bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)].

flag = isminphase(d) returns true if the digital filter, d, has minimum phase. Use designfilt to generate d based on frequency-response specifications.

flag = isminphase(...,tol) uses the tolerance, tol, to determine when two numbers are close enough to be considered equal. If not specified, tol, defaults to eps^(2/3).
## Details for Fixed-Point Arithmetic

<table>
<thead>
<tr>
<th>System Object State</th>
<th>Coefficient Data Type</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlocked</td>
<td>'Same as input'</td>
<td>The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.</td>
</tr>
<tr>
<td>Unlocked</td>
<td>'Custom'</td>
<td>The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.</td>
</tr>
<tr>
<td>Locked</td>
<td>'Same as input'</td>
<td>When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.</td>
</tr>
<tr>
<td>Locked</td>
<td>'Custom'</td>
<td>The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.</td>
</tr>
</tbody>
</table>

flag = isminphase(h) determines if the dfilt filter object h is minimum phase.

## Examples
**Minimum Phase Filters**

Design a sixth-order lowpass Butterworth IIR filter using second order sections. Specify a normalized 3-dB frequency of 0.15. Check if the filter has minimum phase.

\[
[z,p,k] = \text{butter}(6,0.15);
\]

\[
\text{SOS} = \text{zp2sos}(z,p,k);
\]

\[
\text{min\_flag} = \text{isminphase}(\text{SOS})
\]

\[
\text{min\_flag} = \text{logical}
\]

Redesign the filter using `designfilt`. Check that the zeros and poles of the transfer function are on or within the unit circle.

\[
d = \text{designfilt}('\text{lowpassiir}','\text{DesignMethod}','\text{butter}','\text{FilterOrder}',6,'\text{HalfPowerFrequency}',0.25);
\]

\[
d\_\text{flag} = \text{isminphase}(d)
\]

\[
d\_\text{flag} = \text{logical}
\]

\[
z\text{plane}(d)
\]
Given a filter defined with a set of single-precision numerator and denominator coefficients, check if it has minimum phase for different tolerance values.

```matlab
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)
```

1-1321
min_flag2 = logical
1

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)
flag = ismaxphase(h)

Description

flag = ismaxphase(b,a) returns a logical output, flag, equal to true if the filter specified by numerator coefficients, b, and denominator coefficients, a, is a maximum phase filter.

flag = ismaxphase(sos) returns true if the filter specified by second order sections matrix, sos, is a maximum phase filter. sos is a K-by-6 matrix, where the number of sections, K, must be greater than or equal to 2. Each row of sos corresponds to the coefficients of a second order (biquad) filter. The ith 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).
Details for Fixed-Point Arithmetic

<table>
<thead>
<tr>
<th>System Object State</th>
<th>Coefficient Data Type</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlocked</td>
<td>'Same as input'</td>
<td>The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.</td>
</tr>
<tr>
<td>Unlocked</td>
<td>'Custom'</td>
<td>The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.</td>
</tr>
<tr>
<td>Locked</td>
<td>'Same as input'</td>
<td>When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.</td>
</tr>
<tr>
<td>Locked</td>
<td>'Custom'</td>
<td>The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.</td>
</tr>
</tbody>
</table>

flag = ismaxphase(h) returns true if the dfilt filter object h is a maximum phase filter.

Examples
Maximum- and Minimum-Phase Filters

Design maximum-phase and minimum-phase lattice filters and verify their phase type.

```matlab
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.

```matlab
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 | isdouble | single

Introduced in R2014a
isstable

Determine whether filter is stable

Syntax

flag = isstable(b,a)
flag = isstable(sos)
flag = isstable(d)
flag = isstable(h)

Description

flag = isstable(b,a) returns a logical output, flag, equal to true if the filter specified by numerator coefficients, b, and denominator coefficients, a, is a stable filter. If the poles lie on or outside the circle, isstable returns false. If the poles are inside the circle, isstable returns true.

flag = isstable(sos) returns true if the filter specified by second order sections matrix, sos, is stable. sos is a K-by-6 matrix, where the number of sections, K, must be greater than or equal to 2. Each row of sos corresponds to the coefficients of a second order (biquad) filter. The ith row of the sos matrix corresponds to [bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)].

flag = isstable(d) returns true if the digital filter, d, is stable. Use designfilt to generate d based on frequency-response specifications.
### Details for Fixed-Point Arithmetic

<table>
<thead>
<tr>
<th>System Object State</th>
<th>Coefficient Data Type</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlocked</td>
<td>'Same as input'</td>
<td>The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.</td>
</tr>
<tr>
<td>Unlocked</td>
<td>'Custom'</td>
<td>The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.</td>
</tr>
<tr>
<td>Locked</td>
<td>'Same as input'</td>
<td>When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.</td>
</tr>
<tr>
<td>Locked</td>
<td>'Custom'</td>
<td>The function performs fixed-point analysis based on the setting of the <code>CustomCoefficientsDataType</code> property.</td>
</tr>
</tbody>
</table>

flag = isstable(h) returns true if the dfilt filter object h is stable.

### 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.

\[
[z,p,k] = \text{butter}(6,0.7,'\text{high}');
\]
\[
\text{SOS} = \text{zp2sos}(z,p,k);
\]
\[
\text{flag} = \text{isstable}(\text{SOS})
\]

\[
\text{flag} = \text{logical}
\]
\[
1
\]

\[
\text{zplane}(z,p)
\]
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

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 = 1x5

0.3090  0.9801  0.0031  0.0082  -0.0082

References

See Also
ac2rc | lar2rc | poly2rc | rc2is

Introduced before R2006a
istft

Inverse short-time Fourier transform

Syntax

\[ x = \text{istft}(s) \]
\[ x = \text{istft}(s,fs) \]
\[ x = \text{istft}(s,ts) \]
\[ x = \text{istft}(\_,\text{Name,Value}) \]
\[ [x,t] = \text{istft}(\_) \]

Description

\( x = \text{istft}(s) \) returns the inverse short-time Fourier transform (ISTFT) of \( s \).

\( x = \text{istft}(s,fs) \) returns the ISTFT of \( s \) using sample rate \( fs \).

\( x = \text{istft}(s,ts) \) returns the ISTFT using sample time \( ts \).

\( x = \text{istft}(\_,\text{Name,Value}) \) 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.

\[ [x,t] = \text{istft}(\_) \] 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.
The first channel consists of a concave quadratic chirp with instantaneous frequency 100Hz at \( t = 0 \) and crosses 300 Hz at \( t = 1 \) second. It has an initial phase equal to 45 degrees.

The second channel consists of a convex quadratic chirp with instantaneous frequency 200Hz at \( t = 0 \) and crosses 600 Hz at \( t = 1 \) second.

The third channel consists of a logarithmic chirp with instantaneous frequency 300Hz 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.

```matlab
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 ($N_{S,Out} > N_{S,In}$), 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.

```matlab
load handel
```

Design a root-Hann window of length 512. Set analysis overlap length as 192 and synthesis overlap length as 166.

```matlab
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.

\[ S = \text{stft}(y, Fs, 'Window', \text{win}, 'OverlapLength', \text{noverlapA}); \]
\[ iy = \text{istft}(S, Fs, 'Window', \text{win}, 'OverlapLength', \text{noverlapS}); \]

%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.

\[ \text{hopRatio} = (\text{wlen}-\text{noverlapS})/(\text{wlen}-\text{noverlapA}); \]
\[ iyg = iy*\text{hopRatio}; \]
\[ Fp = Fs*\text{hopRatio}; \]

%To hear, type soundsc(iwg,Fs), pause(15), soundsc(iwg,Fp);

Plot the original signal and the time stretched signal with fixed gain.

\[
\text{plot}((0:\text{length}(iyg)-1)/Fs,iyg,(0:\text{length}(y)-1)/Fs,y) \\
xlabel('Time (s)') \\
xlim([0 (\text{length}(iyg)-1)/Fs]) \\
legend('Time Stretched Signal with Fixed Gain', 'Original Signal', 'Location', 'best');
\]
Compare the time-stretched signal and the pitch shifted signal on the same plot.

```matlab
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 \( F_s \) over 2 seconds.

\[
t = 0:1/F_s:2;
\]

\[
x = \sin(2\pi*10*t);
\]

Calculate the short-time Fourier transform and the inverse short-time Fourier transform with overlap lengths 192 and 166 respectively.

\[
S_x = \text{stft}(x,F_s,'Window',\text{win},'OverlapLength',\text{noverlapA});
\]

\[
\text{ix} = \text{istft}(S_x,F_s,'Window',\text{win},'OverlapLength',\text{noverlapS});
\]

Plot the original signal on one plot and the time-stretched and pitch shifted signal on another.
```matlab
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.

```matlab
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(100,'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 \( \frac{\text{length}(xZero) - \text{overlap}}{\text{nwin} - \text{overlap}} \) is an integer. Set the FFT length to 128. Compute the short-time Fourier transform of the complex signal.

\[
xZero = [\text{zeros}(1,\text{nwin}) \times \text{zeros}(1,\text{nwin})];
\]

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.

\[
[\text{is},\text{ti}] = \text{istft}(s,fs,'Window',win,'OverlapLength',noverlap,'FFTLength',fftlen);
\]

is(1:nwin) = [];
is(end-nwin+1:end) = [];
ti = ti(1:end-2*nwin);

Plot the real parts of the original and reconstructed signals. The imaginary part of the signal is also reconstructed perfectly.

\[
\text{plot}(ts,\text{real}(x))
\]

hold on
\[
\text{plot}(ti,\text{real}(is),'- - ')
\]

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.

```matlab
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','ConjugateSymmetric',true);

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 \((\text{length}(x) - \text{noverlap})/(\text{length}(\text{window}) - \text{noverlap})\) must be an integer.

Data Types: double | single
Complex Number Support: Yes

fs — Sample rate
2\pi (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 comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

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 point FFT.

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”.

1-1349
Example: \( \text{hann}(N+1) \) 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 `FFTLength` 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 the following:

- '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.
**Centered — Frequency range**

true (default) | false

Frequency range, specified as the comma-separated pair consisting of `Centered` and `true` or `false`. If this option is set to `true`, then the spectrum is centered and is computed over the interval $-\pi$ to $\pi$. Otherwise, the spectrum is computed over the interval $0$ to $2\pi$.

**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 $f_s$ is provided, then $t$ contains time values in seconds.
- If a duration $t_s$ 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:

\[
x(n) = \int_{-1/2}^{1/2} \sum_{m = -\infty}^{\infty} X_m(f)e^{j2\pi fn}df
\]

\[
= \sum_{m = -\infty}^{\infty} \int_{-1/2}^{1/2} X_m(f)e^{j2\pi fn}df
\]

\[
= \sum_{m = -\infty}^{\infty} x_m(n)
\]

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.
Segmentation/Windowing

\[ g(n) \]

\[ x(n) \]

\[ x_6(n) \]
\[ x_5(n) \]
\[ x_4(n) \]
\[ x_3(n) \]
\[ x_2(n) \]
\[ x_1(n) \]

\[ R = M - L \]

DFT

\[ |X_1(f)|^2 \]
\[ |X_2(f)|^2 \]
\[ |X_3(f)|^2 \]
\[ |X_4(f)|^2 \]
\[ |X_5(f)|^2 \]
\[ |X_6(f)|^2 \]

\[ |X(f)|^2 \]

Inverse DFT

\[ \hat{x}_1(n) \]
\[ \hat{x}_2(n) \]
\[ \hat{x}_3(n) \]
\[ \hat{x}_4(n) \]
\[ \hat{x}_5(n) \]
\[ \hat{x}_6(n) \]

Reconstruction

\[ \hat{x} = \sum_{k} \hat{x}_k \]
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
  \[ G(f_k) = 0, \quad k = 1, 2, ..., R - 1, \quad f_k = \frac{k}{R}. \]

  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.

- **Strong COLA compliance** requires the Fourier transform of the window to be bandlimited consistently with downsampling by the frame rate such that
  \[ G(f) = 0, \quad f \geq \frac{1}{2R}. \]

  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 \(k = \frac{\text{length}(x) - \text{noverlap}}{\text{length}(\text{window}) - \text{noverlap}}\) 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**


Extended Capabilities

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.

See Also

Functions
iscola | psspectrum | stft

Introduced in R2019a
kaiser

Kaiser window

Syntax

\[ w = \text{kaiser}(L,\beta) \]

Description

\( w = \text{kaiser}(L,\beta) \) 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 \text{wvtool}.

\[
\begin{align*}
 w &= \text{kaiser}(200,2.5) \\
 \text{wvtool}(w)
\end{align*}
\]
**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:

\[
    w(n) = \frac{I_0\left(\beta \sqrt{1 - \left(\frac{n - N/2}{N/2}\right)^2}\right)}{I_0(\beta)}, \quad 0 \leq n \leq N,
\]

where \( I_0 \) is the zeroth-order modified Bessel function of the first kind. The length \( L = N + 1 \). \( \text{kaiser}(L,\text{beta}) \) is equivalent to

\[
    \text{besseli}(0,\text{beta}^*\sqrt{1-(((0:L-1)-(L-1)/2)/((L-1)/2)).^2})/\text{besseli}(0,\text{beta})
\]

To obtain a Kaiser window that represents an FIR filter with sidelobe attenuation of \( \alpha \) dB, use the following \( \beta \).

\[
    \beta = \begin{cases} 
    0.1102(\alpha - 8.7), & \alpha > 50 \\
    0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 50 \geq \alpha \geq 21 \\
    0, & \alpha < 21
    \end{cases}
\]

Increasing \( \beta \) widens the mainlobe and decreases the amplitude of the sidelobes (i.e., increases the attenuation).

References

Extended Capabilities

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

See Also

Apps
Signal Analyzer | Window Designer

Functions
chebwin | gausswin | kaiserord | pspectrum | tukeywin | wvtool

Introduced before R2006a
kaiserord

Kaiser window FIR filter design estimation parameters

Syntax

\[
\begin{align*}
[n,\textit{Wn},\beta,\textit{ftype}] &= \text{kaiserord}(f, a, \text{dev}) \\
[n,\textit{Wn},\beta,\textit{ftype}] &= \text{kaiserord}(f, a, \text{dev}, \text{fs}) \\
c &= \text{kaiserord}(f, a, \text{dev}, \text{fs}, '\text{cell}')
\end{align*}
\]

Description

\([n,\textit{Wn},\beta,\textit{ftype}] = \text{kaiserord}(f, a, \text{dev})\) returns a filter order \(n\), normalized frequency band edges \(\textit{Wn}\), and a shape factor \(\beta\) that specify a Kaiser window for use with the \texttt{fir1} function. To design an FIR filter \(b\) that approximately meets the specifications given by \(f\), \(a\), and \(\text{dev}\), use \(b = \text{fir1}(n,\textit{Wn},\text{kaiser}(n + 1,\beta),\textit{ftype},'\text{noglob}')\).

\([n,\textit{Wn},\beta,\textit{ftype}] = \text{kaiserord}(f, a, \text{dev}, \text{fs})\) uses a sample rate \(\text{fs}\) in Hz.

\(c = \text{kaiserord}(f, a, \text{dev}, \text{fs}, '\text{cell}')\) returns a cell array whose elements are the parameters to \texttt{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.

\[
\begin{align*}
\text{fsamp} &= 8000; \\
\text{fcuts} &= [1000\ 1500]; \\
\text{mags} &= [1\ 0]; \\
\text{devs} &= [0.05\ 0.01];
\end{align*}
\]
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:

```matlab
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.

```matlab
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 twice the length of \(2*\text{length}(a) - 2\).

**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 (unit-less).

**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

\[
\beta = \begin{cases} 
0.1102(\alpha - 8.7), & \alpha > 50 \\
0.5842(\alpha - 21)^{0.4} + 0.07886(\alpha - 21), & 21 \leq \alpha \leq 50 \\
0, & \alpha < 21
\end{cases}
\]

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


Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.
See Also
fir1 | firpmord | kaiser

Introduced before R2006a
**kaiserwin**

Kaiser window filter from specification object

**Syntax**

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
d = fdesign.lowpass;
Hd = design(d,'kaiserwin');
fvtool(Hd)
```

![Magnitude Response (dB)](image-url)
See Also

Apps
Filter Designer

Functions
designfilt | fdesign

Introduced in R2009a
kurtogram

Visualize spectral kurtosis

Syntax

\[
kgram = \text{kurtogram}(x) \\
kgram = \text{kurtogram}(x,\text{sampx}) \\
kgram = \text{kurtogram}(xt) \\
kgram = \text{kurtogram}(\_,\text{level}) \\
[kgram,f,w,fc,wc,bw] = \text{kurtogram}(\_) \\
\text{kurtogram}(\_) \\
\]

Description

\(kgram = \text{kurtogram}(x)\) returns the fast kurtogram on page 1-1384 of signal vector \(x\) as a matrix. \text{kurtogram} uses normalized frequency (evenly spaced frequency vector spanning \([0 \pi]\)) to compute the time values.

\(kgram = \text{kurtogram}(x,\text{sampx})\) returns the fast kurtogram on page 1-1384 of signal vector \(x\) sampled at rate or time interval \(\text{sampx}\), as a matrix.

\(kgram = \text{kurtogram}(xt)\) returns the fast kurtogram on page 1-1384 of signal vector \(xt\) as a matrix.

\(kgram = \text{kurtogram}(\_,\text{level})\) returns the fast kurtogram on page 1-1384 using a specified \(\text{level}\). \(\text{level}\) determines the level of window resolution to use, and therefore how many spectral kurtosis cases to calculate.

\([kgram,f,w,fc,wc,bw] = \text{kurtogram}(\_)\) returns the fast kurtogram on page 1-1384 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.

```matlab
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 $fs$.

```matlab
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.

```matlab
code = 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.

```matlab
[kgram,f,w,fc,wc,bw] = kurtogram(x);
wc
wc = 256
fc
fc = 2.4421
```
These values match the optimal window size, center frequency, and bandwidth of the first plot. \texttt{kgram} is the actual kurtogram matrix, and \texttt{f} and \texttt{w} are the frequency and window-size vectors that accompany it.

### Plot Spectral Kurtosis Using a Customized Window Size

The \texttt{pkurtosis} function uses the default \texttt{pspectrum} window size (time resolution). You can specify the window size to use instead. In this example, use the function \texttt{kurtogram} to return an optimal window size and use that result for \texttt{pkurtosis}.

Create a chirp signal with white Gaussian noise.

```matlab
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.

```matlab
pkurtosis(x,fs)
title('Spectral Kurtosis with Default Window Size')
```
Now compute the optimal window size using `kurtogram`.

```markdown
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 $w_0$ into \texttt{pkurtosis}.

\begin{verbatim}
w0 = 256;
pkurtosis(x,fs,w0)
title('Spectral Kurtosis with Optimum Window Size of 256')
\end{verbatim}
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**

```plaintext
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**

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 1-1374.

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 \([0 \pi]\).

**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” (MATLAB). \( xt \) 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^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 1-1378.

**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 1-1773 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


See Also
pentropy|pkurtosis|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'.

1-1387
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    DefaultValue    Sublabels             Tag    Description
______________    ___________    _____________    __________    __________________    ____________    ___________________________    ___    ____________________________
"WhaleType"       "attribute"    "categorical"    {3x1 string}       {"N/A"     }       {0x0 double}    {0x0 double               }    ""     "Whale type"
"MoanRegions"     "roi"          "logical"        {"N/A"     }       {0x0 double}       {0x0 double}    {0x0 double               }    ""     "Regions where moans occur"
"TrillRegions"    "roi"          "logical"        {"N/A"     }       {0x0 double}       {0x0 double}    {1x1 signalLabelDefinition}    ""     "Regions where trills occur"

Visualize the properties of the TrillRegions label.

labelDefinitionsSummary(lss, "TrillRegions")

ans=1×9 table
LabelName     LabelType    LabelDataType    Categories    ValidationFunction     DefaultValue    Tag    Description
____________    _________    _____________    __________    __________________    ____________    ___    _____________
"TrillRegions"      "roi"        "logical"      {"N/A"}        {0x0 double}           {0x0 double}    ""     "Regions where trills occur"

Visualize the properties of the TrillRegions sublabels.

labelDefinitionsSummary(lss, "TrillRegions", 'sublbls')

ans=1×8 table
LabelName     LabelType    LabelDataType    Categories    ValidationFunction    DefaultValue    D"
**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:
```matlab
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.
The `lss = labeledSignalSet(src,lbldefs,'MemberNames',mnames)` function 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 1-1394 using name-value pairs. You can specify multiple name-value pairs. Enclose each property name in quotes. `lbldefs` is optional.

### Input Arguments

- **src** — Input data source
  
  - `matrix` | `cell array` | `timetable` | `audio datastore`

  Input data source, specified as a matrix, a timetable, a cell array, or an audio datastore. `src` specifies the number of members of the set, the number of signals in each member, and the data in each signal.

- **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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addLabelDefinitions</td>
<td>Add label definitions to labeled signal set</td>
</tr>
<tr>
<td>addMembers</td>
<td>Add members to labeled signal set</td>
</tr>
<tr>
<td>concatenate</td>
<td>Concatenate two or more labeled signal sets</td>
</tr>
<tr>
<td>editLabelDefinition</td>
<td>Edit label definition properties</td>
</tr>
<tr>
<td>getLabelDefinitions</td>
<td>Get label definitions in labeled signal set</td>
</tr>
<tr>
<td>getLabeledSignal</td>
<td>Get labeled signals from labeled signal set</td>
</tr>
<tr>
<td>getLabelNames</td>
<td>Get label names in labeled signal set</td>
</tr>
<tr>
<td>getLabelValues</td>
<td>Get label values from labeled signal set</td>
</tr>
<tr>
<td>getMemberNames</td>
<td>Get member names in labeled signal set</td>
</tr>
<tr>
<td>getSignal</td>
<td>Get signals from labeled signal set</td>
</tr>
<tr>
<td>head</td>
<td>Get top rows of labels table</td>
</tr>
<tr>
<td>labelDefinitionsHierarchy</td>
<td>Get hierarchical list of label and sublabel names</td>
</tr>
<tr>
<td>labelDefinitionsSummary</td>
<td>Get summary table of signal label definitions</td>
</tr>
<tr>
<td>removeLabelDefinition</td>
<td>Remove label definition from labeled signal set</td>
</tr>
<tr>
<td>removeMembers</td>
<td>Remove members from labeled signal set</td>
</tr>
<tr>
<td>removePointValue</td>
<td>Remove row from point label</td>
</tr>
<tr>
<td>removeRegionValue</td>
<td>Remove row from ROI label</td>
</tr>
<tr>
<td>resetLabelValues</td>
<td>Reset labels to default values</td>
</tr>
<tr>
<td>setLabelValue</td>
<td>Set label value in labeled signal set</td>
</tr>
<tr>
<td>setMemberNames</td>
<td>Set member names in labeled signal set</td>
</tr>
<tr>
<td>subset</td>
<td>Get new labeled signal set with subset of members</td>
</tr>
</tbody>
</table>
Examples

Label Definitions for Whale Songs

Consider a set of whale sound recordings. The recorded whale sounds consist of trills and moans. 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.

```matlab
load labelwhalesignals
lbldefs = [dWhaleType dMoans dTrills];

lss = labeledSignalSet({whale1 whale2},lbldefs,...
    'SampleRate',Fs,'Description','Characterize wave 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=3x9 table
LabelName    LabelType       LabelDataType      Categories               ValidationFunction
------------- ----------- ---------- ------------------- -----------------------------
    "WhaleType"    "attribute"    "categorical"    {3x1 string}                {["N/A" ]}
      "MoanRegions"    "roi"      "logical"           {["N/A" ]}                  {0x0 double}
      "TrillRegions"    "roi"      "logical"           {["N/A" ]}                  {0x0 double}

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

<table>
<thead>
<tr>
<th>WhaleType</th>
<th>MoanRegions</th>
<th>TrillRegions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member{1}</td>
<td>blue</td>
<td>{0x2 table}</td>
</tr>
<tr>
<td>Member{2}</td>
<td>blue</td>
<td>{0x2 table}</td>
</tr>
</tbody>
</table>

**Visualize Regions and Points**

Visualize the whale songs to identify the trill and moan regions.

```matlab
subplot(2,1,1)
plot((0:length(whale1)-1)/Fs,whale1)
ylabel('Whale 1')
hold on

subplot(2,1,2)
plot((0:length(whale2)-1)/Fs,whale2)
ylabel('Whale 2')
hold on
```
Add the moan and trill regions to the labeled set. For ROI labels, specify the ROI limits in seconds and the label values. Label the different regions in the plots using an auxiliary function.

```matlab
moanRegionsWhale1 = [6.1 7.7; 11.4 13.1; 16.5 18.1];
setLabelValue(lss,1,'MoanRegions',moanRegionsWhale1,[true true true]);

trillRegionWhale1 = [1.4 3.1];
setLabelValue(lss,1,'TrillRegions',trillRegionWhale1,true);

subplot(2,1,1)
labelIntervals([moanRegionsWhale1;trillRegionWhale1])

moanRegionsWhale2 = [2.5 3.5; 5.8 8; 15.4 16.7];
```
setLabelValue(lss,2,'MoanRegions',moanRegionsWhale2,[true true true]);
trillRegionWhale2 = [11.1 13];
setLabelValue(lss,2,'TrillRegions',trillRegionWhale2,true);

subplot(2,1,2)
labelIntervals([moanRegionsWhale2;trillRegionWhale2])

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`.

```matlab
getLabelValues(lss)
```

ans=2×3 table

<table>
<thead>
<tr>
<th>WhaleType</th>
<th>MoanRegions</th>
<th>TrillRegions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member{1}</td>
<td>blue</td>
<td>{3x2 table}</td>
</tr>
<tr>
<td>Member{2}</td>
<td>blue</td>
<td>{3x2 table}</td>
</tr>
</tbody>
</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
    __________    _____    ___________
     1.4    3.1    {[1]}    {3×2 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=1x2 table
     Location    Value
        ______    __________
          11.437    {[0.1500]}

function labelIntervals(moansTrills)
% Auxiliary function to label moan and trill regions in plots
     [X,Y] = meshgrid(moansTrills,ylim);
     plot(X,Y,'k:');
     topts = {'HorizontalAlignment','center','FontWeight','bold', ...
          'FontSize',12,'Color',[139 69 19]/255};
     text((X(1,1:4)+X(1,5:end))/2,Y(2,5:end)-0.1, ...
          ['moan' 'moan' 'moan' 'trill'],topts{:})
end

See Also
Signal Labeler | signalLabelDefinition

Introduced in R2018b
**lar2rc**

Convert log area ratio parameters to reflection coefficients

**Syntax**

\[ \text{k} = \text{lar2rc}(\text{g}) \]

**Description**

\[ \text{k} = \text{lar2rc}(\text{g}) \] 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.

\[ \text{g} = [0.6389 \ 4.5989 \ 0.0063 \ 0.0163 \ -0.0163] \; ; \]
\[ \text{k} = \text{lar2rc}(\text{g}) \]
\[ \text{k} = 1 \times 5 \]
\[ 0.3090 \quad 0.9801 \quad 0.0031 \quad 0.0081 \quad -0.0081 \]

**References**

See Also
ac2rc | is2rc | poly2rc | rc2lar

Introduced before R2006a
**latc2tf**

Convert lattice filter parameters to transfer function form

**Syntax**

\[
[num,den] = \text{latc2tf}(k,v)
\]

\[
[num,den] = \text{latc2tf}(k,'iiroption')
\]

\[
um = \text{latc2tf}(k,'firoption')
\]

**Description**

\[
[num,den] = \text{latc2tf}(k,v)
\] finds the transfer function numerator \(\text{num}\) and denominator \(\text{den}\) from the IIR lattice coefficients \(k\) and ladder coefficients \(v\).

\[
[num,den] = \text{latc2tf}(k,'iiroption')
\] 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 \(k\).

\[
um = \text{latc2tf}(k,'firoption')
\] 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**

```matlab
[f,g] = latcfilt(k,x)
[f,g] = latcfilt(k,v,x)
[f,g] = latcfilt(k,1,x)
[f,g,zf] = latcfilt(...,'ic',zi)
[f,g,zf] = latcfilt(...,dim)
```

**Description**

When filtering data, lattice coefficients can be used to represent

- FIR filters
- All-pole IIR filters
- Allpass IIR filters
- General IIR filters

\[ [f,g] = \text{latcfilt}(k,x) \]

filters \( x \) with the FIR lattice coefficients in the vector \( k \). The forward lattice filter result is \( f \) and \( g \) is the backward filter result. If \(|k| \leq 1\), \( f \) corresponds to the minimum-phase output, and \( g \) corresponds to the maximum-phase output.

If \( k \) and \( x \) are vectors, the result is a (signal) vector. Matrix arguments are permitted under the following rules:

- If \( x \) is a matrix and \( k \) is a vector, each column of \( x \) is processed through the lattice filter specified by \( k \).
- If \( x \) is a vector and \( k \) is a matrix, each column of \( k \) is used to filter \( x \), and a signal matrix is returned.
- If \( x \) and \( k \) are both matrices with the same number of columns, then the \( i \)th column of \( k \) is used to filter the \( i \)th column of \( x \). A signal matrix is returned.
[f,g] = latcfilt(k,v,x) filters x with the IIR lattice coefficients k and ladder coefficients v. Both k and v must be vectors, while x can be a signal matrix.

[f,g] = latcfilt(k,1,x) filters x with the IIR lattice specified by k, where k and x can be vectors or matrices. f is the all-pole lattice filter result and g is the allpass filter result.

[f,g,zf] = latcfilt(...,'ic',zi) accepts a length-k vector zi specifying the initial condition of the lattice states. Output zf is a length-k vector specifying the final condition of the lattice states.

[f,g,zf] = latcfilt(...,dim) filters x along the dimension dim. To specify a dim value, the FIR lattice coefficients k must be a vector and you must specify all previous input parameters in order. Use the empty vector [ ] for any parameters you do not want to specify. zf returns the final conditions in columns, regardless of the shape of x.

Examples

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')
See Also
filter | latc2tf | tf2latc

Introduced before R2006a
levinson

Levinson-Durbin recursion

Syntax

\[
a = \text{levinson}(r)
\]
\[
a = \text{levinson}(r,n)
\]
\[
[a,e] = \text{levinson}(r,n)
\]
\[
[a,e,k] = \text{levinson}(r,n)
\]

Description

The Levinson-Durbin recursion is an algorithm for finding an all-pole IIR filter with a prescribed deterministic autocorrelation sequence. It has applications in filter design, coding, and spectral estimation. The filter that \textit{levinson} produces is minimum phase.

\[a = \text{levinson}(r)\] finds the coefficients of a \text{length}(r)-1 order autoregressive linear process which has \(r\) as its autocorrelation sequence. \(r\) is a real or complex deterministic autocorrelation sequence. If \(r\) is a matrix, \textit{levinson} finds the coefficients for each column of \(r\) and returns them in the rows of \(a\). \(n=\text{length}(r)-1\) is the default order of the denominator polynomial \(A(z)\); that is, \(a = [1 \ a(2) \ldots \ a(n+1)]\). The filter coefficients are ordered in descending powers of \(z^{-1}\).

\[
H(z) = \frac{1}{A(z)} = \frac{1}{1 + a(2)z^{-1} + \ldots + a(n+1)z^{-n}}
\]

\(a = \text{levinson}(r,n)\) returns the coefficients for an autoregressive model of order \(n\).

\([a,e] = \text{levinson}(r,n)\) returns the prediction error, \(e\), of order \(n\).

\([a,e,k] = \text{levinson}(r,n)\) returns the reflection coefficients \(k\) as a column vector of length \(n\).

\textbf{Note} \(k\) is computed internally while computing the \(a\) coefficients, so returning \(k\) simultaneously is more efficient than converting \(a\) to \(k\) with \textit{tf2latc}.  

1-1416
Examples

**Autoregressive Process Coefficients**

Estimate the coefficients of an autoregressive process given by

\[ x(n) = 0.1 x(n-1) - 0.8 x(n-2) + w(n). \]

\[ a = [1 0.1 -0.8]; \]

Generate a realization of the process by filtering white noise of variance 0.4.

\[ \text{rng('default')} \]
\[ v = 0.4; \]
\[ w = \text{sqrt}(v) * \text{randn}(15000,1); \]
\[ x = \text{filter}(1,a,w); \]

Estimate the correlation function. Discard the correlation values at negative lags. Use the Levinson-Durbin recursion to estimate the model coefficients.

\[ [r,lg] = \text{xcorr}(x, 'biased'); \]
\[ r(lg<0) = []; \]
\[ \text{ar} = \text{levinson}(r, \text{numel}(a)-1) \]
\[ \text{ar} = 1 \times 3 \]
\[ \begin{bmatrix} 1.0000 & 0.0957 & -0.8026 \end{bmatrix} \]

**Algorithms**

`levinson` solves the symmetric Toeplitz system of linear equations

\[
\begin{bmatrix}
  r(1) & r(2)* & \ldots & r(n)* \\
  r(2) & r(1) & \ldots & r(n-1)* \\
  \vdots & \vdots & \ddots & \vdots \\
  r(n) & \ldots & r(2) & r(1)
\end{bmatrix}
\begin{bmatrix}
  a(2) \\
  a(3) \\
  \vdots \\
  a(n+1)
\end{bmatrix}
= \begin{bmatrix}
  -r(2) \\
  -r(3) \\
  \vdots \\
  -r(n+1)
\end{bmatrix}
\]
where \( r = [r(1) \ldots r(n+1)] \) 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 NaNs 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 \( \backslash \) for low orders to provide the fastest possible execution.

### References


### 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.

### See Also

`lpc` | `prony` | `rlevinson` | `schurrc` | `stmcb`

**Introduced before R2006a**
**lowpass**

Lowpass-filter signals

**Syntax**

\[y = \text{lowpass}(x,\text{wpass})\]

\[y = \text{lowpass}(x,\text{fpass},\text{fs})\]

\[y = \text{lowpass}(\text{xt},\text{fpass})\]

\[y = \text{lowpass}(\_,\text{Name},\text{Value})\]

\[[y,d] = \text{lowpass}(\_)\]

\[
\text{lowpass}(\_)
\]

**Description**

\[y = \text{lowpass}(x,\text{wpass})\] filters the input signal \(x\) using a lowpass filter with normalized passband frequency \(\text{wpass}\) in units of \(\pi\) rad/sample. \text{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.

\[y = \text{lowpass}(x,\text{fpass},\text{fs})\] specifies that \(x\) is sampled at a rate of \(\text{fs}\) hertz. \(\text{fpass}\) is the passband frequency of the filter in hertz.

\[y = \text{lowpass}(\text{xt},\text{fpass})\] lowpass-filters the data in timetable \(\text{xt}\) using a filter with a passband frequency of \(\text{fpass}\) hertz. The function independently filters all variables in the timetable and all columns inside each variable.

\[y = \text{lowpass}(\_,\text{Name},\text{Value})\] specifies additional options for any of the previous syntaxes using name-value pair arguments. You can change the stopband attenuation, the transition band steepness, and the type of impulse response of the filter.

\[[y,d] = \text{lowpass}(\_)\] 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.

```matlab
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.

```matlab
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.

```matlab
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 0 2 2 2 0 3 5 5 0 3 2 1 2 3 3 3 2 2 3 2 1]+1;
acc = [3 0 5 0 3 0 3 2 0 2 2 3 0 5 5 3 0 5 0 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.

```matlab
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.

```matlab
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: \( \sin(2\pi(0:127)/16) + \text{randn}(1,128)/100 \) 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, $f_s$/2)

Passband frequency, specified as a scalar in the interval (0, $f_s$/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” (MATLAB).

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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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 \texttt{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 1-1430 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(d)` 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, \( f_{\text{stop}} \), 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 \( f_{\text{stop}} - f_{\text{pass}} \).
• 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
\[
W = (1 - s) \times (f_{\text{Nyquist}} - f_{\text{pass}}).
\]
• When 'Steepness' is equal to 0.5, the transition width is 50% of \( f_{\text{Nyquist}} - f_{\text{pass}} \).
• As 'Steepness' approaches 1, the transition width becomes progressively narrower until it reaches a minimum value of 1% of \( (f_{Nyquist} - f_{pass}) \).

• The default value of 'Steepness' is 0.85, which corresponds to a transition width that is 15% of \( (f_{Nyquist} - f_{pass}) \).

See Also

Apps
Signal Analyzer

Functions
bandpass | bandstop | designfilt | filter | filtfilt | fir1 | highpass

Introduced in R2018a
**lp2bp**

Transform lowpass analog filters to bandpass

**Syntax**

\[
[bt,at] = \text{lp2bp}(b,a,Wo,Bw)
\]

\[
[At,Bt,Ct,Dt] = \text{lp2bp}(A,B,C,D,Wo,Bw)
\]

**Description**

`lp2bp` transforms analog lowpass filter prototypes with a cutoff angular frequency of 1 rad/s into bandpass filters with desired bandwidth and center frequency. The transformation is one step in the digital filter design process for the `butter`, `cheby1`, `cheby2`, and `ellip` functions.

`lp2bp` can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

**Transfer Function Form (Polynomial)**

\[
[bt,at] = \text{lp2bp}(b,a,Wo,Bw)
\]

transforms an analog lowpass filter prototype given by polynomial coefficients into a bandpass filter with center frequency \( Wo \) and bandwidth \( Bw \). Row vectors \( b \) and \( a \) specify the coefficients of the numerator and denominator of the prototype in descending powers of \( s \).

\[
\frac{B(s)}{A(s)} = \frac{b(1)s^n + \ldots + b(ns + b(n + 1)}{a(1)s^m + \ldots + a(ms + a(m + 1)}
\]

Scalars \( Wo \) and \( Bw \) specify the center frequency and bandwidth in units of rad/s. For a filter with lower band edge \( w1 \) and upper band edge \( w2 \), use \( Wo = \sqrt{w1*w2} \) and \( Bw = w2-w1 \).

`lp2bp` returns the frequency transformed filter in row vectors \( bt \) and \( at \).
State-Space Form

\[ [A_t, B_t, C_t, D_t] = \text{lp2bp}(A, B, C, D, \omega_0, B_w) \]

converts the continuous-time state-space lowpass filter prototype in matrices \( A, B, C, D \) shown below

\[
\begin{align*}
\dot{x} &= Ax + Bu \\
y &=Cx + Du
\end{align*}
\]

into a bandpass filter with center frequency \( \omega_0 \) and bandwidth \( B_w \). For a filter with lower band edge \( w_1 \) and upper band edge \( w_2 \), use \( \omega_0 = \sqrt{w_1 w_2} \) and \( B_w = w_2 - w_1 \).

The bandpass filter is returned in matrices \( A_t, B_t, C_t, D_t \).

Algorithms

\text{lp2bp} \text{ is a highly accurate state-space formulation of the classic analog filter frequency transformation. Consider the state-space system}

\[
\begin{align*}
\dot{x} &= Ax + Bu \\
y &=Cx + Du
\end{align*}
\]

where \( u \) is the input, \( x \) is the state vector, and \( y \) is the output. The Laplace transform of the first equation (assuming zero initial conditions) is

\[ sX(s) = AX(s) + BU(s) \]

Now if a bandpass filter is to have center frequency \( \omega_0 \) and bandwidth \( B_w \), the standard \( s \)-domain transformation is

\[ s = Q(p^2 + 1)/p \]

where \( Q = \omega_0/B_w \) and \( p = s/\omega_0 \). Substituting this for \( s \) in the Laplace transformed state-space equation, and considering the operator \( p \) as \( d/dt \) results in

\[ Q\ddot{x} + Q\dot{x} = \dot{A}x + \dot{B}u \]

or

\[ Q\ddot{x} - \dot{A}x - \dot{B}u = -Qx \]

Now define
\[ Q\dot{\omega} = -Qx \]

which, when substituted, leads to

\[ \dot{Q}x = Ax + Q\omega + Bu \]

The last two equations give equations of state. Write them in standard form and multiply the differential equations by \( \omega_0 \) to recover the time/frequency scaling represented by \( p \) and find state matrices for the bandpass filter:

\[
Q = Wo/Bw; \quad [ma,m] = size(A);
At = Wo*[A/Q eye(ma,m);-eye(ma,m) zeros(ma,m)];
Bt = Wo*[B/Q; zeros(ma,n)];
Ct = [C zeros(mc,ma)];
Dt = d;
\]

If the input to \( \text{lp2bp} \) is in transfer function form, the function transforms it into state-space form before applying this algorithm.

**See Also**

\textit{bilinear} | \textit{impinvar} | \textit{lp2bs} | \textit{lp2hp} | \textit{lp2lp}

**Introduced before R2006a**
lp2bs
Transform lowpass analog filters to bandstop

Syntax

\[ [bt, at] = \text{lp2bs}(b, a, Wo, Bw) \]
\[ [At, Bt, Ct, Dt] = \text{lp2bs}(A, B, C, D, Wo, Bw) \]

Description

lp2bs transforms analog lowpass filter prototypes with a cutoff angular frequency of 1 rad/s into bandstop filters with desired bandwidth and center frequency. The transformation is one step in the digital filter design process for the butter, cheby1, cheby2, and ellip functions.

lp2bs can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

Transfer Function Form (Polynomial)

\[ [bt, at] = \text{lp2bs}(b, a, Wo, Bw) \] transforms an analog lowpass filter prototype given by polynomial coefficients into a bandstop filter with center frequency Wo and bandwidth Bw. Row vectors b and a specify the coefficients of the numerator and denominator of the prototype in descending powers of s.

\[
\frac{B(s)}{A(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n+1)}{a(1)s^m + \cdots + a(m)s + a(m+1)}
\]

Scalars Wo and Bw specify the center frequency and bandwidth in units of rad/s. For a filter with lower band edge w1 and upper band edge w2, use Wo = \( \sqrt{w_1 \cdot w_2} \) and Bw = w2 - w1.

lp2bs returns the frequency transformed filter in row vectors bt and at.
State-Space Form

\[ [A_t, B_t, C_t, D_t] = \text{lp2bs}(A, B, C, D, W_0, B_w) \]

converts the continuous-time state-space lowpass filter prototype in matrices \( A, B, C, D \) shown below

\[
\begin{align*}
\dot{x} &= Ax + Bu \\
y &=Cx + Du
\end{align*}
\]

into a bandstop filter with center frequency \( W_0 \) and bandwidth \( B_w \). For a filter with lower band edge \( w_1 \) and upper band edge \( w_2 \), use

\[
W_0 = \sqrt{w_1 \cdot w_2} \quad \text{and} \quad B_w = w_2 - w_1.
\]

The bandstop filter is returned in matrices \( A_t, B_t, C_t, D_t \).

Algorithms

\text{lp2bs} is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a bandstop filter is to have center frequency \( \omega_0 \) and bandwidth \( B_w \), the standard \( s \)-domain transformation is

\[
s = \frac{p}{Q(p^2 + 1)}
\]

where \( Q = \omega_0/B_w \) and \( p = s/\omega_0 \). The state-space version of this transformation is

\[
\begin{align*}
Q &= \frac{W_0}{B_w} \\
A_t &= [W_0/Q*\text{inv}(A) \; W_0*\text{eye}(ma); -W_0*\text{eye}(ma) \; \text{zeros}(ma)] \\
B_t &= [-[W_0/Q*(A\backslash B); \; \text{zeros}(ma,n)] \\
C_t &= [C/A \; \text{zeros}(mc,ma)] \\
D_t &= D - C/A*B
\end{align*}
\]

See \text{lp2bp} for a derivation of the bandpass version of this transformation.

See Also

bilinear | impinvar | lp2bp | lp2hp | lp2lp

Introduced before R2006a
**lp2hp**

Transform lowpass analog filters to highpass

**Syntax**

\[
[bt,at] = \text{lp2hp}(b,a,Wo)
\]

\[
[At,Bt,Ct,Dt] = \text{lp2hp}(A,B,C,D,Wo)
\]

**Description**

\[
[bt,at] = \text{lp2hp}(b,a,Wo)
\] 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.

\[
[At,Bt,Ct,Dt] = \text{lp2hp}(A,B,C,D,Wo)
\] 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.

\[
f = 100;
\]

\[
[ze,pe,ke] = \text{ellipap}(5,3,30);
\]

\[
[be,ae] = \text{zp2tf}(ze,pe,ke);
\]

Transform the prototype to a highpass filter. Specify the cutoff frequency in rad/s.

\[
[bh,ah] = \text{lp2hp}(be,ae,2*pi*f);
\]
Compute and plot the frequency response of the filter. Divide the normalized frequency by $2\pi$ so the x-axis of the plot is in Hz.

$$[hh,wh] = \text{freqs}(bh,ah,4096);$$

```matlab
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

\[
\begin{align*}
\dot{x} &= Ax + Bu \\
y &= Cx + Du
\end{align*}
\]

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 $\omega_0$, the standard s-domain transformation is

$$s = \frac{\omega_0}{p}.$$  

The state-space version of this transformation is:

At = Wo*inv(A);  
Bt = -Wo*(A\B);  
Ct = C/A;  
Dt = D - C/A*B;

See lp2bp for a derivation of the bandpass version of this transformation.

**See Also**

bilinear | impinvar | lp2bp | lp2bs | lp2lp

**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**

`lp2lp` transforms an analog lowpass filter prototype with a cutoff angular frequency of 1 rad/s into a lowpass filter with any specified cutoff angular frequency. The transformation is one step in the digital filter design process for the butter, cheby1, cheby2, and ellip functions.

The `lp2lp` function can perform the transformation on two different linear system representations: transfer function form and state-space form. In both cases, the input system must be an analog filter prototype.

**Transfer Function Form (Polynomial)**

```
[bt,at] = lp2lp(b,a,Wo)
```

transforms an analog lowpass filter prototype given by polynomial coefficients into a lowpass filter with cutoff angular frequency `Wo`. Row vectors `b` and `a` specify the coefficients of the numerator and denominator of the prototype in descending powers of `s`.

\[
\frac{B(s)}{A(s)} = \frac{b(1)s^n + \cdots + b(n)s + b(n + 1)}{a(1)s^m + \cdots + a(m)s + a(m + 1)}
\]

Scalar `Wo` specifies the cutoff angular frequency in units of rad/s. `lp2lp` returns the frequency transformed filter in row vectors `bt` and `at`. 
**State-Space Form**

\[ [A_t, B_t, C_t, D_t] = \text{lp2lp}(A, B, C, D, W_0) \]

converts the continuous-time state-space lowpass filter prototype in matrices \( A, B, C, D \) below

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

into a lowpass filter with cutoff angular frequency \( W_0 \). \( \text{lp2lp} \) returns the lowpass filter in matrices \( A_t, B_t, C_t, D_t \).

**Algorithms**

\( \text{lp2lp} \) is a highly accurate state-space formulation of the classic analog filter frequency transformation. If a lowpass filter is to have cutoff angular frequency \( \omega_0 \), the standard \( s \)-domain transformation is

\[
s = \frac{p}{\omega_0}
\]

The state-space version of this transformation is

\[
A_t = W_0 \cdot A; \\
B_t = W_0 \cdot B; \\
C_t = C; \\
D_t = D;
\]

See \( \text{lp2bp} \) for a derivation of the bandpass version of this transformation.

**See Also**

bilinear | impinvar | \( \text{lp2bp} \) | \( \text{lp2bs} \) | \( \text{lp2hp} \)

**Introduced before R2006a**
**Ipc**

Linear prediction filter coefficients

**Syntax**

\[ [a,g] = \text{lpc}(x,p) \]

**Description**

\[ [a,g] = \text{lpc}(x,p) \] finds the coefficients of a pth-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.

\[
\text{noise} = \text{randn}(50000,1); \\
x = \text{filter}(1,[1 1/2 1/3 1/4],\text{noise}); \\
x = x(\text{end}-4096+1:\text{end});
\]

Compute the predictor coefficients and the estimated signal.

\[
\text{a} = \text{lpc}(x,3); \\
\text{est}_x = \text{filter}([0 -\text{a}(2:\text{end})],1,x);
\]
Compare the predicted signal to the original signal by plotting the last 100 samples of each.

```matlab
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.
\begin{verbatim}
e = x - est_x;
[acs, lags] = xcorr(e, 'coeff');

plot(lags, acs)
grid
xlabel('Lags')
ylabel('Normalized Autocorrelation')
ylim([-0.1 1.1])
\end{verbatim}
**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:

$$\hat{x}(n) = -a(2)x(n-1) - a(3)x(n-2) - \ldots - a(p+1)x(n-p).$$

$g$ — 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

$$
X = \begin{bmatrix}
  x(1) & 0 & \cdots & 0 \\
  x(2) & x(1) & \cdots & : \\
  \vdots & x(2) & \cdots & 0 \\
  x(m) & : & : & x(1) \\
  0 & x(m) & \cdots & x(2) \\
  \vdots & \vdots & \vdots & \vdots \\
  0 & \cdots & 0 & x(m)
\end{bmatrix}, \quad a = \begin{bmatrix}
  1 \\
  a(2) \\
  \vdots \\
  a(p + 1)
\end{bmatrix}, \quad b = \begin{bmatrix}
  1 \\
  0 \\
  \vdots \\
  0
\end{bmatrix},
$$

and $m$ is the length of $x$. Solving the least-squares problem using the normal equations $X^HXa = X^Hb$ leads to the Yule-Walker equations.
\[
\begin{bmatrix}
  r(1) & r(2)^* & \cdots & r(p)^* \\
  r(2) & r(1) & \cdots & \vdots \\
  \vdots & \vdots & \ddots & \vdots \\
  r(p) & \cdots & r(2) & r(1)
\end{bmatrix}
\begin{bmatrix}
  a(2) \\
  a(3) \\
  \vdots \\
  a(p+1)
\end{bmatrix}
= 
\begin{bmatrix}
  -r(2) \\
  -r(3) \\
  \vdots \\
  -r(p+1)
\end{bmatrix},
\]

where \( r = [r(1) \cdots r(p+1)] \) is an autocorrelation estimate for \( x \) computed using \texttt{xcorr}. The Levinson-Durbin algorithm (see \texttt{levinson}) solves the Yule-Walker equations in \( O(p^2) \) flops.

**References**


**See Also**

\texttt{aryule} | \texttt{levinson} | \texttt{prony} | \texttt{pyulear} | \texttt{stmcb}

**Introduced before R2006a**
**lsf2poly**

Convert line spectral frequencies to prediction filter coefficients

**Syntax**

```matlab
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 × 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 × (M + 1)`.

**Examples**

**Prediction Coefficients from Line Spectral Frequencies**

Given a vector, `lsf`, of line spectral frequencies, determine the equivalent prediction filter coefficients.

```matlab
lsf = [0.7842 1.5605 1.8776 1.8984 2.3593];
a = lsf2poly(lsf)
a = 1x6
     1.0000    0.6148    0.9899    0.0001    0.0031   -0.0081
```
References


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**

\[ ydb = \text{mag2db}(y) \]

**Description**

\[ ydb = \text{mag2db}(y) \] expresses in decibels (dB) the magnitude measurements specified in \( y \). The relationship between magnitude and decibels is \( ydb = 20 \log_{10}(y) \).

**Examples**

**Magnitude Response of a Highpass Filter**

Design a 3rd-order highpass Butterworth filter having a normalized 3-dB frequency of \( 0.5\pi \) rad/sample. Compute its frequency response. Express the magnitude response in decibels and plot it.

\[
[b,a] = \text{butter}(3,0.5,'\text{high}')
\]
\[
[h,w] = \text{freqz}(b,a);
\]
\[
dB = \text{mag2db}(\text{abs}(h)));
\]
\[
\text{plot}(w/pi,dB)
\]
\[
\text{xlabel('\omega / \pi')}
\]
\[
\text{ylabel('Magnitude (dB)')}
\]
\[
\text{ylim([-82 5])}
\]
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

\[ Q = \text{marcumq}(a,b) \]
\[ Q = \text{marcumq}(a,b,m) \]

Description

\[ Q = \text{marcumq}(a,b) \] computes the Marcum Q function of \( a \) and \( b \), defined by

\[
Q(a, b) = \int_{b}^{\infty} x \exp \left( -\frac{x^2 + a^2}{2} \right) I_0(ax) \, dx
\]

where \( a \) and \( b \) are nonnegative real numbers. In this expression, \( I_0 \) is the modified Bessel function of the first kind of zero order.

\[ Q = \text{marcumq}(a,b,m) \] computes the generalized Marcum Q, defined by

\[
Q(a, b) = \frac{1}{a^{m-1}} \int_{b}^{\infty} x^m \exp \left( -\frac{x^2 + a^2}{2} \right) I_{m-1}(ax) \, dx
\]

where \( a \) and \( b \) are nonnegative real numbers, and \( m \) is a positive integer. In this expression, \( I_{m-1} \) is the modified Bessel function of the first kind of order \( m-1 \).

If any of the inputs is a scalar, it is expanded to the size of the other inputs.

Algorithms

marcumq uses the algorithm developed in [3]. The paper describes two error criteria: a relative error criterion and an absolute error criterion. marcumq utilizes the absolute error criterion.
References


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,'design_flag')

Description

[b,a] = maxflat(n,m,Wn) is a lowpass Butterworth filter with numerator and
denominator coefficients b and a of orders n and m, respectively. Wn is the normalized
cutoff frequency at which the magnitude response of the filter is equal to 1/√2
(approximately –3 dB). Wn must be between 0 and 1, where 1 corresponds to the Nyquist
frequency.

b = maxflat(n,'sym',Wn) is a symmetric FIR Butterworth filter. n must be even, and
Wn is restricted to a subinterval of [0,1]. The function raises an error if Wn is specified
outside of this subinterval.

[b,a,b1,b2] = maxflat(n,m,Wn) returns two polynomials b1 and b2 whose product
is equal to the numerator polynomial b (that is, b = conv(b1,b2)). b1 contains all the
zeros at z = -1, and b2 contains all the other zeros.

[b,a,b1,b2,sos,g] = maxflat(n,m,Wn) returns the second-order sections
representation of the filter as the filter matrix sos and the gain g.

[...] = maxflat(n,m,Wn,'design_flag') enables you to monitor the filter design,
where 'design_flag' is

• 'trace' for a textual display of the design table used in the design
• 'plots' for plots of the filter's magnitude, group delay, and zeros and poles
• 'both' for both the textual display and plots

Examples

Generalized Butterworth Filter

Design a generalized Butterworth filter with normalized cutoff frequency $0.2\pi$ rad/s. Specify a numerator order of 10 and a denominator order of 2. Visualize the frequency response of the filter.

```matlab
n = 10;
m = 2;
Wn = 0.2;

[b,a] = maxflat(n,m,Wn);
fvtool(b,a)
```
Algorithms

The method consists of the use of formulae, polynomial root finding, and a transformation of polynomial roots.
References


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(____,freqlrange)
[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(____,freqlrange) 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 freqlrange 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.

\[
x_2 = 2\cdot \text{chirp}(t, 200e3, n\text{Samp}/F_s, 300e3);
\]
\[
x_2 = x_2 + \text{randn(size}(x_2))\cdot \text{std}(x_2)/\text{db2mag}(\text{SNR});
\]

Concatenate the chirps to produce a two-channel signal. Estimate the mean frequency of each channel.

\[
y = \text{meanfreq}([x \ x_2], F_s)
\]
\[
y = 1\times 2
\]
\[
10^5 \times
\]
Plot the PSDs of the two channels and annotate their mean frequencies.

```matlab
meanfreq([x x2],Fs);
```

Add the two channels to form a new signal. Plot the PSD and annotate the mean frequency.

```matlab
meanfreq(x+x2,Fs)
```
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.

\[
\text{nSamp} = 1024; \\
\text{Fs} = 1024e3; \\
\text{SNR} = 40; \\
\]

\[
\text{ans} = 2.1496e+05 \\
\]
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.

```matlab
[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.

\[ x_2 = 2 \times \sin(2\pi t \times 257.321\text{e}3); \]
\[ x_2 = x_2 + \text{randn(size}(x2)) \times \text{std}(x2)/\text{db2mag}(\text{SNR}); \]

Concatenate the sinusoids to produce a two-channel signal. Estimate the PSD of each channel and use the result to determine the mean frequency.

\[ \text{[Pyy,f]} = \text{periodogram}([x \ x_2],\text{kaiser(nSamp,38)},[],\text{Fs}); \]
\[ y = \text{meanfreq}(\text{Pyy,f}) \]
\[ y = 1x2 \]
\[ 10^5 \times \]
\[ 1.0013 \quad 2.5732 \]

Annotate the mean frequencies of the two channels on a plot of the PSDs.

\[ \text{meanfreq}(\text{Pyy,f}); \]
Add the two channels to form a new signal. Estimate the PSD and annotate the mean frequency.

```matlab
[Pzz,f] = periodogram(x+x2,kaiser(nSamp,38),[],Fs);
meanfreq(Pzz,f);
```
Mean Frequency of Bandlimited Signals

Generate a signal whose PSD resembles the frequency response of an 88th-order bandpass FIR filter with normalized cutoff frequencies $0.25\pi$ rad/sample and $0.45\pi$ rad/sample.

```matlab
d = fir1(88,[0.25 0.45]);
```

Compute the mean frequency of the signal between $0.3\pi$ rad/sample and $0.6\pi$ rad/sample. Plot the PSD and annotate the mean frequency and measurement interval.

```matlab
meanfreq(d,[],[0.3 0.6]*pi);
```
Output the mean frequency and the band power of the measurement interval. Specifying a sample rate of $2\pi$ is equivalent to leaving the rate unset.

$$[\text{mnf}, \text{power}] = \text{meanfreq}(d, 2\pi, [0.3, 0.6] \cdot \pi);$$

```matlab
fprintf('Mean = %.3f\pi, power = %.1f\% of total 
', ...
    mnf/pi, power/bandpower(d)*100)
```

Mean = $0.373\pi$, power = 75.6\% of total

Add a second channel with normalized cutoff frequencies $0.5\pi$ rad/sample and $0.8\pi$ rad/sample and an amplitude that is one-tenth that of the first channel.

```matlab
d = [d;fir1(88, [0.5, 0.8])/10]';
```
Compute the mean frequency of the signal between $0.3\pi$ rad/sample and $0.9\pi$ rad/sample. Plot the PSD and annotate the mean frequency of each channel and the measurement interval.

```matlab
meanfreq(d,[],[0.3 0.9]*pi);
```

Output the mean frequency of each channel. Divide by $\pi$.

```matlab
mnf = meanfreq(d,[],[0.3 0.9]*pi)/pi
```

```
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:
```matlab
[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.

**See Also**

*findpeaks | medfreq | periodogram | plomb | pwelch*

**Introduced in R2015a**
medfilt1

1-D median filtering

Syntax

\[ y = \text{medfilt1}(x) \]
\[ y = \text{medfilt1}(x,n) \]
\[ y = \text{medfilt1}(x,n,\text{blksz},\text{dim}) \]
\[ y = \text{medfilt1}(x,n,[]\text{,dim}) \]
\[ y = \text{medfilt1}(___,\text{nanflag},\text{padding}) \]

Description

\[ y = \text{medfilt1}(x) \] 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 \( x \).

\[ y = \text{medfilt1}(x,n) \] applies an \( n \)th-order one-dimensional median filter to \( x \).

\[ y = \text{medfilt1}(x,n,\text{blksz},\text{dim}) \] or \( y = \text{medfilt1}(x,n,[]\text{,dim}) \) specifies the dimension, \( \text{dim} \), along which the filter operates. \( \text{blksz} \) is required for backward compatibility and is ignored.

\[ y = \text{medfilt1}(___,\text{nanflag},\text{padding}) \] specifies how NaN values are treated over each segment, using any input arguments from previous syntaxes. This syntax also specifies \( \text{padding} \), the type of filtering performed at the signal edges.

\( \text{nanflag} \) and \( \text{padding} \) 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.

\[
\begin{align*}
fs &= 100; \\
t &= 0:1/fs:1; \\
x &= \sin(2\pi t \cdot 3) + 0.25\sin(2\pi t \cdot 40);
\end{align*}
\]

Use a 10th-order median filter to smooth the signal. Plot the result.

\[
\begin{align*}
y &= \text{medfilt1}(x,10); \\
\text{plot}(t,x,t,y) \\
\text{legend}('Original','Filtered') \\
\text{legend}('boxoff')
\end{align*}
\]
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.

```matlab
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 \texttt{medfilt1} with the default settings. Plot the filtered signal. By default, the filter assigns NaN to the median of any segment with missing samples.

\texttt{y = medfilt1(x);}  
\texttt{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.

```matlab
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.

```matlab
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.

```matlab
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, \text{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

\textbf{dim} — Dimension to filter along

positive integer scalar

Dimension to filter along, specified as a positive integer scalar. By default, \text{medfilt1} operates along the first nonsingleton dimension of \( x \). In particular, if \( x \) is a matrix, the function filters its columns so that \( y(:,i) = \text{medfilt1}(x(:,i),n) \).

Data Types: double

\textbf{nanflag} — NaN condition

'\text{includenan}' (default) | '\text{omitnan}'

NaN condition, specified as '\text{includenan}' or '\text{omitnan}'.

- '\text{includenan}' — Returns the filtered signal so that the median of any segment containing NaNs is also NaN.
- '\text{omitnan}' — Returns the filtered signal so that the median of any segment containing NaNs is the median of the non-NaN values. If all elements of a segment are NaNs, the result is NaN.

\textbf{padding} — Endpoint filtering

'\text{zeropad}' (default) | '\text{truncate}'

Endpoint filtering, specified as '\text{zeropad}' or '\text{truncate}'.

- '\text{zeropad}' — Considers the signal to be zero beyond the endpoints.
- '\text{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


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.

\[
x_2 = 2 \times \text{chirp}(t,200e3,nSamp/Fs,300e3);
\]
\[
x_2 = x_2 + \text{randn(size}(x_2)) \times \text{std}(x_2)/\text{db2mag}(\text{SNR});
\]

Concatenate the chirps to produce a two-channel signal. Estimate the median frequency of each channel.

\[
y = \text{medfreq}([x \ x_2],Fs)
\]
\[
y = 1 	imes 2 \\
10^5 	imes
\]
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)
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.

```matlab
nSamp = 1024;
Fs = 1024e3;
SNR = 40;
```

ans = 2.3756e+05
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.

\[ x_2 = 2 \sin(2\pi t \times 257.321e3); \]
\[ x_2 = x_2 + \text{randn(size}(x_2) * \text{std}(x_2)/\text{db2mag}(\text{SNR}); \]

Concatenate the sinusoids to produce a two-channel signal. Estimate the PSD of each channel and use the result to determine the median frequency.

\[ [P_{yy}, f] = \text{periodogram}([x \ x_2], \text{kaiser}(n\text{Samp}, 38), [], \text{Fs}); \]
\[ y = \text{medfreq}(P_{yy}, f) \]
\[ y = 1 \times 2 \]
\[ 10^5 \times \]
\[ 1.0012 \quad 2.5731 \]

Annotate the median frequencies of the two channels on a plot of the PSDs.

\[ \text{medfreq}(P_{yy}, f); \]
Add the two channels to form a new signal. Estimate the PSD and annotate the median frequency.

\[ [Pzz,f] = \text{periodogram}(x+x2,\text{kaiser}(nSamp,38),[],Fs); \]
\[ \text{medfreq}(Pzz,f); \]
Median Frequency of Bandlimited Signals

Generate a signal whose PSD resembles the frequency response of an 88th-order bandpass FIR filter with normalized cutoff frequencies $0.25\pi$ rad/sample and $0.45\pi$ rad/sample.

```matlab
d = fir1(88,[0.25 0.45]);
```

Compute the median frequency of the signal between $0.3\pi$ rad/sample and $0.6\pi$ rad/sample. Plot the PSD and annotate the median frequency and measurement interval.

```matlab
medfreq(d,[],[0.3 0.6]*pi);
```
Output the median frequency and the band power of the measurement interval. Specifying a sample rate of $2\pi$ is equivalent to leaving the rate unset.

```matlab
[mdf,power] = medfreq(d,2*pi,[0.3 0.6]*pi);
fprintf('Mean = %.3f*pi, power = %.1f% of total 
', mdf/pi,power/bandpower(d)*100)
```

Mean = 0.371*pi, power = 77.4% of total

Add a second channel with normalized cutoff frequencies $0.5\pi$ rad/sample and $0.8\pi$ rad/sample and an amplitude that is one-tenth that of the first channel.

```matlab
d = [d;fir1(88,[0.5 0.8])/10]';
```
Compute the median frequency of the signal between $0.3\pi$ rad/sample and $0.9\pi$ rad/sample. Plot the PSD and annotate the median frequency of each channel and the measurement interval.

```matlab
medfreq(d,[],[0.3 0.9]*pi);
```

Output the median frequency of each channel. Divide by $\pi$.

```matlab
mdf = medfreq(d,[],[0.3 0.9]*pi)/pi
mdf = 1x2
     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 \texttt{freq} has the same units as \texttt{fs}.
• If you do not specify a sample rate, then \texttt{freq} has units of rad/sample.

\texttt{power — Band power}

\texttt{scalar | vector}

Band power, returned as a scalar or vector.

\textbf{See Also}

\texttt{findpeaks|meanfreq|periodogram|plomb|pwelch}

\textit{Introduced in R2015a}
midcross

Mid-reference level crossing for bilevel waveform

Syntax

C = midcross(X)
C = midcross(X,FS)
C = midcross(X,T)
[C,MIDLEV] = midcross(...)
C = midcross(X,Name,Value)
midcross(...)

Description

C = midcross(X) returns a vector, C, of time instants where each transition of the input signal, X, crosses the 50% reference level. The sample instants correspond to the indices of the input vector. Because midcross uses interpolation to determine the crossing instant, C may contain values that do not correspond to sampling instants. To determine the transitions, midcross estimates the state levels of X by a histogram method. midcross identifies all intervals which cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-1509.

C = midcross(X,FS) specifies the sample rate, FS, in hertz as a positive scalar. The first sample instant corresponds to t=0. Because midcross uses interpolation to determine the crossing instant, C may contain values that do not correspond to sampling instants.

C = midcross(X,T) specifies the sample instants, T, as a vector with the same number of elements as X. Because midcross uses interpolation to determine the crossing instant, C may contain values that do not correspond to sampling instants.

[C,MIDLEV] = midcross(...) returns the waveform value corresponding to the mid-reference level.
C = midcross(X,Name,Value) returns the time instants corresponding to mid-reference level crossings with additional options specified by one or more Name,Value pair arguments.

midcross(...) plots the signal and marks the location of the mid-crossings (mid-reference level instants) and the associated reference levels. midcross also plots the state levels with upper and lower state boundaries.

**Input Arguments**

X
Bilevel waveform. X is a real-valued row or column vector.

FS
Sample rate in hertz.

T
Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

**MidPercentReferenceLevel**
Mid-reference level as a percentage of the waveform amplitude.

**Default:** 50

**StateLevels**
Low and high state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low state level. The second element is the high state level. If you do not specify low-
and high-state levels, `midcross` estimates the state levels from the input waveform using the histogram method.

**Tolerance**

Tolerance levels (lower- and upper-state boundaries) expressed as a percentage. See "State-Level Tolerances" on page 1-1509.

**Default:** 2

**Output Arguments**

`c`

Time instants of the mid-reference level crossings.

`MIDLEV`

Mid-reference level.

**Examples**

**Mid-Reference Level Instant of Bilevel Waveform**

Assuming a sampling interval of 1, compute the mid-reference level instant of a bilevel waveform. Plot the result.

```matlab
load('transitionex.mat','x')
midcross(x)
```
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. \texttt{midcross} uses interpolation to identify the mid-reference level crossing.

\textbf{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)
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.

```matlab
load('transitionex.mat','x','t')
 [~,midlev] = midcross(x,t)
 midlev = 1.1388
```

Annotate the result on a plot of the waveform.

```matlab
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);
More About

Mid-Reference Level

The mid-reference level in a bilevel waveform with low-state level, \( S_1 \), and high-state level, \( S_2 \), is

\[
S_1 + \frac{1}{2}(S_2 - S_1)
\]
**Mid Reference Level Instant**

Let \( y_{50\%} \) denote the mid-reference level.

Let \( t_{50\%}^- \) and \( t_{50\%}^+ \) denote the two consecutive sampling instants corresponding to the waveform values nearest in value to \( y_{50\%} \).

Let \( y_{50\%}^- \) and \( y_{50\%}^+ \), denote the waveform values at \( t_{50\%}^- \) and \( t_{50\%}^+ \).

The mid-reference level instant is

\[
  t_{50\%} = t_{50\%}^- + \left( \frac{t_{50\%}^+ - t_{50\%}^-}{y_{50\%}^+ - y_{50\%}^-} \right)(y_{50\%}^+ - y_{50\%}^-)
\]

**State-Level Tolerances**

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the \( \alpha\% \) region for the low state is defined as

\[
  S_1 \pm \frac{\alpha}{100}(S_2 - S_1),
\]

where \( S_1 \) is the low-state level and \( S_2 \) is the high-state level. Replace the first term in the equation with \( S_2 \) to obtain the \( \alpha\% \) tolerance region for the high state.

The following figure illustrates lower and upper 2\% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.
References


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.

fn = modalfit(frf,f,fs,mnum,Name,Value) specifies additional options using name-value pair 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 pair 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 $\Delta t = 1/F_s$.

\[
F_s = 1; \\
dt = 1/Fs; \\
N = 3000; \\
t = dt*(0:N-1); \\
b = 0.01;
\]

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 $2 \times 2$ identity, and the continuous-time state-space matrices are

1-1512
\[ A_c = \begin{bmatrix} 0 & 1 \\ -1 & -b \end{bmatrix}, \quad B_c = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \]

\[ Ac = [0 \ 1; -1 \ -b]; \]
\[ A = \expm(Ac*dt); \]

\[ Bc = [0;1]; \]
\[ B = Ac/(A-eye(2))*Bc; \]

\[ C = [1 \ 0]; \]
\[ D = 0; \]

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 \texttt{modalfrf} estimate with the definition.

[b,a] = ss2tf(A,B,C,D);

nfs = 2048;
\[
fz = 0:1/nfs:1/2-1/nfs; 
z = \exp(2j*\pi*fz); 
ztf = \text{polyval}(b,z)./\text{polyval}(a,z);
\]

\[
\text{plot}(f,20*\log10(\text{abs(frf)))) 
\text{hold on} 
\text{plot}(fz*Fs,20*\log10(\text{abs(ztf)))) 
\text{hold off} 
\text{grid} 
\text{ylim([-60 40])}
\]

Estimate the natural frequency and the damping ratio for the vibration mode.

\[
[fn,dr] = \text{modalfit}(frf,f,Fs,1,'FitMethod','PP')
\]
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

```matlab
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.

```matlab
[fn,dr,ms,ofrf] = modalfit(frf,f,fs,24,'FitMethod','lsrf');
```

Compare the reconstructed FRF array to the measured one.

```matlab
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))
```
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 \( i \)th 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

\[
A = \exp(A_c\Delta t), \quad B = A_c^{-1}(A - I)B_c, \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 0 & 0 \end{bmatrix},
\]

\( I \) is the \( 4 \times 4 \) identity, and the continuous-time state-space matrices are

\[
A_c = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad B_c = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.
\]
\[
A_c = \begin{bmatrix}
0 & 1 & 0 & 0 \\
-2k & -2b & k & b \\
0 & 0 & 0 & 1 \\
k/\mu & b/\mu & -2k/\mu & -2b/\mu
\end{bmatrix}, \quad B_c = \begin{bmatrix}
0 & 0 \\
1 & 0 \\
0 & 0 \\
0 & 1/\mu
\end{bmatrix}.
\]

Set \( k = 400 \), \( b = 0.1 \), and \( \mu = 1/10 \).

\[
k = 400; \\
b = 0.1; \\
m = 1/10;
\]

\[
A_c = [0 \ 1 \ 0 \ 0;-2k \ -2b \ k \ b;0 \ 0 \ 0 \ 1;k/m \ b/m \ -2k/m \ -2b/m]; \\
A = \expm(Ac*dt); \\
Bc = [0 \ 0;1 \ 0;0 \ 0;0 \ 1/m]; \\
B = A \backslash (A\text{-}eye(4))*Bc; \\
C = [1 \ 0 \ 0 \ 0;0 \ 0 \ 1 \ 0]; \\
D = zeros(2);
\]

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.

\[
\text{rng default} \\
u = \text{randn}(2,N); \\
y = [0;0]; \\
x = [0;0;0;0]; \\
\text{for} \ kk = 1:N \\
\quad y(:,kk) = C*x + D*u(:,kk); \\
\quad x = A*x + B*u(:,kk); \\
\text{end}
\]

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.

\[
\text{wind} = \text{hann}(15000); \\
\text{nove} = 9000; \\
[FRF,f] = \text{modalfrf(u',y',Fs,wind,nove,'Sensor','dis.');}
\]

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.

```matlab
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.

\[
[f_{n}, d_{r}, m_{s}] = \text{modalfit}(FRF, f, Fs, 2, 'FitMethod', 'pp');
\]

\[
f_{n} =
\]

\[
f_{n}(\cdot, :, 1) =
\]

\[
\begin{bmatrix}
3.8466 & 3.8466 \\
3.8495 & 3.8495
\end{bmatrix}
\]

\[
f_{n}(\cdot, :, 2) =
\]
3.8492    3.8490
3.8552    14.4684

Compare the natural frequencies to the theoretical predictions for the undamped system.

\[
\text{undamped} = \frac{\sqrt{\text{eig}([2k -k;-k/m 2k/m])}}{2\pi}
\]

\[
\text{undamped} = 2 \times 1
\]

\[
3.8470
14.4259
\]

**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.

```matlab
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.

```matlab
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.

```matlab
[fn,dr,ms,ofrf] = modalfit(frf,f,fs,6,'PhysFreq',phfr);
```
Plot the reconstructed frequency-response functions and compare them to the original ones.

```matlab
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

\texttt{frf} – Frequency-response functions

vector | matrix | 3-D array

Frequency-response functions, specified as a vector, matrix, or 3-D array. \texttt{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: \texttt{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

**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 comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

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 the comma-separated pair consisting of 'Feedthrough' and 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 the comma-separated pair consisting of 'FitMethod' and 'lsce', 'lsrf', or 'pp'.

- 'lsce' — “Least-Squares Complex Exponential Method” on page 1-1531. 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 1-1533. 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 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

**PhysFreq — Natural frequencies for physical modes**

vector

Natural frequencies for physical modes to include in the analysis, specified as the comma-separated pair consisting of 'PhysFreq' and 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 the comma-separated pair consisting of 'DriveIndex' and 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.

\textbf{dr — Damping ratios}

\textit{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.

\textbf{ms — Mode-shape vectors}

\textit{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.

\textbf{oifr — Reconstructed frequency-response functions}

\textit{vector | matrix | 3-D array}

Reconstructed frequency-response functions, returned as a vector, matrix, or 3-D array with the same size as frf.

\section*{Algorithms}

\subsection*{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

\begin{align*}
s_i(n) &= A_ie^{-b_in/f_s}\cos(2\pi f_in/f_s + \phi_i) \\
&= \frac{1}{2}A_ie^{j\phi_i}\exp(-(b_i/f_s - j2\pi f_i/f_s)n) + \frac{1}{2}A_ie^{-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-,
\end{align*}

\chaptermark{modalfit}
where:

- $f_s$ is the sample rate.
- $f_i$ is the sinusoid frequency.
- $b_i$ is the damping coefficient.
- $A_i$ and $\phi_i$ 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):

$$h(0) = a_1 x_1^0 + a_2 x_2^0 + \cdots + a_N x_N^0$$
$$h(1) = a_1 x_1^1 + a_2 x_2^1 + \cdots + a_N x_N^1$$
$$\vdots$$
$$h(N-1) = a_1 x_1^{N-1} + a_2 x_2^{N-1} + \cdots + a_N x_N^{N-1}.$$  

The poles are the roots of a polynomial with coefficients $c_0, c_1, \ldots, c_{N-1}$:

$$x_i^N + c_{N-1} x_i^{N-1} + \cdots + c_1 x_i^1 + c_0 x_i^0 = 0.$$  

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:

\[
H(f) = \frac{1}{(2\pi)^2f^2 + j2\zeta rf_rf + f_r^2} \Rightarrow H(f)f_r^2 + j2\zeta rf_rfH(f) - \frac{1}{(2\pi)^2m} = f^2H(f),
\]

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

\[
\begin{bmatrix}
H(f_{p-k}) & j2f_{p-k}H(f_{p-k}) & -1 \\
\vdots & \vdots & \vdots \\
H(f_p) & j2f_pH(f_p) & -1 \\
\vdots & \vdots & \vdots \\
H(f_{p+k}) & j2f_{p+k}H(f_{p+k}) & -1 
\end{bmatrix}
\begin{bmatrix}
f_r^2 \\
\zeta_rf_r \\
d 
\end{bmatrix}
= \begin{bmatrix}
f_r^2 - f_r^2H(f_{p-k}) \\
\vdots \\
\vdots \\
f_r^2 - f_r^2H(f_{p+k}) 
\end{bmatrix}
\]
References


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

```matlab
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

```matlab
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 frequency-response function matrix, frf, is
computed in terms of dynamic flexibility, and the system response, y, contains
acceleration measurements.

frf = modalfrf(x,y,fs,window,noverlap) specifies overlap samples of overlap
between adjoining segments.

frf = modalfrf(____,Name,Value) specifies options using name-value pair
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 pair 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 pair 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.

```matlab
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 X_rand, the input excitation signal, and Y_rand, 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 $b = 0.01$.

![Diagram of a one-dimensional discrete-time oscillating system]

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

$$
\begin{align*}
\mathbf{x}(k+1) &= A\mathbf{x}(k) + B\mathbf{u}(k), \\
\mathbf{y}(k) &= C\mathbf{x}(k) + D\mathbf{u}(k),
\end{align*}
$$

where $\mathbf{x} = [r \ v]^T$ is the state vector, $r$ and $v$ are respectively the displacement and velocity of the mass, $\mathbf{u}$ is the driving force, and $\mathbf{y} = r$ 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 $2 \times 2$ identity, and the continuous-time state-space matrices are

$$
A_c = \begin{bmatrix} 0 & 1 \\ -1 & -b \end{bmatrix}, \quad B_c = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.
$$
\( A_c = \begin{bmatrix} 0 & 1 \\ -1 & -b \end{bmatrix}; \)
\( A = \exp(A_c \cdot dt); \)

\( B_c = \begin{bmatrix} 0 \\ 1 \end{bmatrix}; \)
\( B = A_c \cdot (A - \text{eye}(2)) \cdot B_c; \)

\( C = \begin{bmatrix} 1 & 0 \end{bmatrix}; \)
\( D = 0; \)

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.

\texttt{rng default}
\texttt{u = randn(1,N)/2;}
\texttt{u(2001:end) = 0;}

\texttt{y = 0;}
\texttt{x = [0;0];}
\texttt{for k = 1:N}
\texttt{\hspace{1cm} y(k) = C*x + D*u(k);}
\texttt{\hspace{1cm} x = A*x + B*u(k);}
\texttt{end}

\texttt{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;
Estimate the natural frequency and the damping ratio for the vibration mode.

\[ [fn, dr] = \text{modalfit}(\text{frf}, f, Fs, 1, 'FitMethod', 'PP') \]
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 \(i\)th 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 \(4 \times 4\) 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 \(\mu = 1/10\).

```matlab
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.

```matlab
rng default
u = randn(2,N);

y = [0;0];
x = [0;0;0;0];
for kk = 1:N
```

1-1545
\[ y(:,kk) = C\cdot x + D\cdot u(:,kk); \]
\[ x = A\cdot x + B\cdot u(:,kk); \]
\[ \text{end} \]

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.

\[
\text{wind} = \text{hann}(15000); \\
\text{nove} = 9000; \\
[\text{FRF},f] = \text{modalfrf}(u',y',\text{Fs},\text{wind},\text{nove},'\text{Sensor}','\text{dis}');
\]

Compute the theoretical transfer function as the Z-transform of the time-domain transfer function, evaluated at the unit circle.

\[
\text{nfs} = 2048; \\
fz = 0:1/\text{nfs}:1/2-1/\text{nfs}; \\
z = \exp(2j\pi*fz); \\
[\text{b1,a1}] = \text{ss2tf}(A,B,C,D,1); \\
[\text{b2,a2}] = \text{ss2tf}(A,B,C,D,2);
\]

\[
\text{frf}(1,:,1) = \text{polyval}(\text{b1}(1,:),z)./\text{polyval}(\text{a1},z); \\
\text{frf}(1,:,2) = \text{polyval}(\text{b1}(2,:),z)./\text{polyval}(\text{a1},z); \\
\text{frf}(2,:,1) = \text{polyval}(\text{b2}(1,:),z)./\text{polyval}(\text{a2},z); \\
\text{frf}(2,:,2) = \text{polyval}(\text{b2}(2,:),z)./\text{polyval}(\text{a2},z);
\]

Plot the estimates and overlay the theoretical predictions.

\[
\text{for } jk = 1:2 \\
\quad \text{for } kj = 1:2 \\
\quad\quad \text{subplot}(2,2,2*(jk-1)+kj) \\
\quad\quad \text{plot}(f,20*\text{log10}(|\text{FRF}(:,jk,kj)|)) \\
\quad\quad \text{hold on} \\
\quad\quad \text{plot}(fz*\text{Fs},20*\text{log10}(|\text{frf}(jk,:,kj)|)) \\
\quad\quad \text{hold off} \\
\quad\quad \text{axis}([0 \text{ Fs/2} -100 0]) \\
\quad\quad \text{title}(\text{sprintf}('Input %d, Output %d',jk,kj)) \\
\quad \end{for} \\
\end{for}
\]
Plot the estimates by using the syntax of `modalfrf` with no output arguments.

```matlab
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.

\[
[f_n, d_r, m_s] = \text{modalfit}(\text{FRF}, f, F_s, 2, 'FitMethod', 'pp');
\]

\[
f_n = 
\]
\[
f_n(\cdot,\cdot,1) = 
\]
\[
\begin{bmatrix}
3.8466 & 3.8466 \\
3.8495 & 3.8495 \\
\end{bmatrix}
\]
Compare the natural frequencies to the theoretical predictions for the undamped system.

\[
\text{undamped} = \frac{\sqrt{\text{eig}([2k -k;-k/m 2k/m])}}{2/\pi}
\]

\[
\text{undamped} = 2\times1
3.8470
14.4259
\]

**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.

```matlab
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.

```matlab
[frf,f] = modalfrf(X,Y,fs,1000,'Estimator','subspace','Order',36);
```

Visualize the stabilization diagram for the system. Identify up to 15 physical modes.

```matlab
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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Sensor','vel','Est','H1'` specifies that the response signal consists of velocity measurements and that the estimator of choice is H1.

Estimator — Estimator

'`H1' (default) | 'H2' | 'Hv' | 'subspace'`

Estimator, specified as the comma-separated pair consisting of `Estimator` and `'H1'`, `'H2'`, `'Hv'`, or `'subspace'`. See “Transfer Function” on page 1-2929 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 $H_2$:

$$H_v = (H_1 H_2)^{1/2}$$

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 the comma-separated pair consisting of 'Feedthrough' and 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 the comma-separated pair consisting of 'Measurement' and '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 the comma-separated pair consisting of 'Order' and 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 the comma-separated pair consisting of 'Sensor' and 'acc', 'dis', or 'vel'.

- 'acc' — The response signal voltage is proportional to acceleration.
- 'dis' — The response signal voltage is proportional to displacement.
- 'vel' — The response signal voltage is proportional to velocity.

### 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.

**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


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 ith element contains a length-i vector of natural frequencies of stable poles. Poles that are not stable are returned as NaNs. 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.

```matlab
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.

```matlab
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.

\[
\text{modalsd}(\text{frf}, f, \text{fs}, '\text{MaxModes}', 20, '\text{SCriteria}', [1e-4 \ 0.002])
\]

Restrict the frequency range to between 0 and 500 Hz. Relax the stability criteria to 0.5% for frequency and 10% for damping.

\[
\text{modalsd}(\text{frf}, f, \text{fs}, '\text{MaxModes}', 20, '\text{SCriteria}', [5e-3 \ 0.1], '\text{FreqRange}', [0 \ 500])
\]
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.

```matlab
modalsd(frf,f,fs,'MaxModes',10,'FreqRange',[100 350],'FitMethod','lsrf')
```
Input Arguments

\textbf{frf — Frequency-response functions}

\texttt{vector | matrix | 3-D array}

Frequency-response functions, specified as a vector, matrix, or 3-D array. \texttt{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: \texttt{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 comma-separated pairs of **Name, Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as **Name1, Value1,...,NameN, ValueN**.

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 [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.

**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 ith row contain natural frequencies. Poles that are nonphysical or not stable in frequency are returned as NaNs.

### References


See Also
modalfit | modalfrf | tfest

Introduced in R2017a
modulate

Modulation for communications simulation

Syntax

```matlab
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.

```matlab
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.

```matlab
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**

'm' (default) | 'amdsb-tc' | 'amssb' | fm | 'pm' | 'pwm' | 'pm' | '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 \( f_c \).

\[ y = x \cdot \cos(2\pi f_c t) \]

- **amdsb-tc** — Amplitude modulation, double sideband, transmitted carrier. Subtracts scalar \( \text{opt} \) from \( x \) and multiplies the result by a sinusoid of frequency \( f_c \).

\[ y = (x - \text{opt}) \cdot \cos(2\pi f_c t) \]

If you do not specify the \( \text{opt} \) parameter, *modulate* uses a default of \( \min(\min(x)) \) so that the message signal \( (x - \text{opt}) \) is entirely nonnegative and has a minimum value of 0.

- **amssb** — Amplitude modulation, single sideband. Multiplies \( x \) by a sinusoid of frequency \( f_c \) and adds the result to the Hilbert transform of \( x \) multiplied by a phase-shifted sinusoid of frequency \( f_c \).

\[ y = x \cdot \cos(2\pi f_c t) + \text{imag(hilbert}(x)) \cdot \sin(2\pi f_c t) \]

This effectively removes the upper sideband.

- **fm** — Frequency modulation. Creates a sinusoid with instantaneous frequency that varies with the message signal \( x \).

\[ y = \cos(2\pi f_c t + \text{opt} \cdot \text{cumsum}(x)) \]

\text{cumsum} \ is a rectangular approximation of the integral of \( x \). *modulate* uses \( \text{opt} \) as the constant of frequency modulation. If you do not specify the \( \text{opt} \) parameter, *modulate* uses a default of \( \text{opt} = (f_c/fs) \cdot 2\pi/(\max(\max(x))) \) so the maximum frequency excursion from \( f_c \) is \( f_c \) Hz.

- **pm** — Phase modulation. Creates a sinusoid of frequency \( f_c \) 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 \( \pi \) 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 \( \text{length}(x) \times fs/fc \).

- **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 \( \text{length}(x) \times fs/fc \).

- **qam** — Quadrature amplitude modulation. Creates a quadrature amplitude modulated signal from signals `x` and `opt`.

\[ y = x \times \cos(2\pi fc t) + opt \times \sin(2\pi fc t) \]

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**

\[
cxy = \text{mscohere}(x,y)
\]

\[
cxy = \text{mscohere}(x,y,\text{window})
\]

\[
cxy = \text{mscohere}(x,y,\text{window},\text{noverlap})
\]

\[
cxy = \text{mscohere}(x,y,\text{window},\text{noverlap},\text{nfft})
\]

\[
cxy = \text{mscohere}(\_,\_,'\text{mimo}')
\]

\[
[cxy,w] = \text{mscohere}(\_)
\]

\[
[cxy,f] = \text{mscohere}(\_,\text{fs})
\]

\[
[cxy,w] = \text{mscohere}(x,y,\text{window},\text{noverlap},w)
\]

\[
[cxy,f] = \text{mscohere}(x,y,\text{window},\text{noverlap},f,\text{fs})
\]

\[
[\_] = \text{mscohere}(x,y,\_,\_,'\text{freqrange})
\]

\[
\text{mscohere}(\_)
\]

**Description**

\[
cxy = \text{mscohere}(x,y)
\] 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 \text{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
mth output signal. See “Magnitude-Squared Coherence” on page 1-1591 for more information.

- If \( x \) and \( y \) are matrices of equal size, then \( \text{mscohere} \) operates column-wise: \( cxy(:,n) = \text{mscohere}(x(:,n),y(:,n)) \). To obtain a multiple coherence matrix, append 'mimo' to the argument list.

\[
\text{cxy} = \text{mscohere}(x,y,\text{window})
\]
uses \( \text{window} \) 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.

\[
\text{cxy} = \text{mscohere}(x,y,\text{window},\text{nooverlap})
\]
uses \( \text{nooverlap} \) samples of overlap between adjoining segments.

\[
\text{cxy} = \text{mscohere}(x,y,\text{window},\text{nooverlap},\text{nfft})
\]
uses \( \text{nfft} \) sampling points to calculate the discrete Fourier transform.

\[
\text{cxy} = \text{mscohere}(\_,\_,'\text{mimo}')
\]
computes a multiple coherence matrix for matrix inputs. This syntax can include any combination of input arguments from previous syntaxes.

\[
[\text{cxy},w] = \text{mscohere}(\_)
\]
returns a vector of normalized frequencies, \( w \), at which the magnitude-squared coherence is estimated.

\[
[\text{cxy},f] = \text{mscohere}(\_,\_,'\text{fs}')
\]
returns a vector of frequencies, \( f \), expressed in terms of the sample rate, \( \text{fs} \), at which the magnitude-squared coherence is estimated. \( \text{fs} \) must be the sixth numeric input to \( \text{mscohere} \). To input a sample rate and still use the default values of the preceding optional arguments, specify these arguments as empty, \([]\).

\[
[\text{cxy},w] = \text{mscohere}(x,y,\text{window},\text{nooverlap},w)
\]
returns the magnitude-squared coherence estimate at the normalized frequencies specified in \( w \).

\[
[\text{cxy},f] = \text{mscohere}(x,y,\text{window},\text{nooverlap},f,\_)
\]
returns the magnitude-squared coherence estimate at the frequencies specified in \( f \).

\[
[\_\_] = \text{mscohere}(x,y,\_\_,'\text{freqrange}')
\]
returns the magnitude-squared coherence estimate over the frequency range specified by \( \text{freqrange} \). Valid options for \( \text{freqrange} \) are 'onesided', 'twosided', and 'centered'.

\[
\text{mscohere}(\_\_\_)
\]
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.

\[ r = \text{randn}(16384,1); \]

To create the first sequence, bandpass filter the signal. Design a 16th-order filter that passes normalized frequencies between \(0.2\pi\) and \(0.4\pi\) rad/sample. Specify a stopband attenuation of 60 dB. Filter the original signal.

\[
\text{dx} = \text{designfilt}('\text{bandpassiir}', '\text{FilterOrder}', 16, \ldots \\
\quad '\text{StopbandFrequency1}', 0.2, '\text{StopbandFrequency2}', 0.4, \ldots \\
\quad '\text{StopbandAttenuation}', 60); \\
x = \text{filter}(\text{dx}, r); 
\]

To create the second sequence, design a 16th-order filter that stops normalized frequencies between \(0.6\pi\) and \(0.8\pi\) rad/sample. Specify a passband ripple of 0.1 dB. Filter the original signal.

\[
\text{dy} = \text{designfilt}('\text{bandstopiir}', '\text{FilterOrder}', 16, \ldots \\
\quad '\text{PassbandFrequency1}', 0.6, '\text{PassbandFrequency2}', 0.8, \ldots \\
\quad '\text{PassbandRipple}', 0.1); \\
y = \text{filter}(\text{dy}, r); 
\]

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.

\[
[cxy, fc] = \text{mscohere}(x, y, \text{hamming}(512), 500, 2048); 
\]

Plot the coherence function and overlay the frequency responses of the filters.

\[
[qx, f] = \text{freqz}(\text{dx}); \\
qy = \text{freqz}(\text{dy}); \\
\text{plot}(fc/pi, cxy) \\
\text{hold on} \\
\text{plot}(f/pi, \text{abs}(qx), f/pi, \text{abs}(qy)) \\
\text{hold off} 
\]
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 0.3\( \pi \) and designed using a rectangular window.

\[
\begin{align*}
h & = \text{fir1}(30, 0.3, \text{rectwin}(31)); \\
x & = \text{randn}(16384, 2); \\
y & = \text{sum}(\text{filter}(h, 1, x), 2);
\end{align*}
\]
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.

\[
\text{noverlap} = 512; \\
nfft = 1024; \\
\text{mscohere}(x,y,\text{hann}(nfft),\text{noverlap},nfft,'mimo')
\]

Compare the coherence estimate to the frequency response of the filter. The drops in coherence correspond to the zeros of the frequency response.

\[
[H,f] = \text{freqz}(h);
\]
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`.

```matlab
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.

\[ [C_{xy,f}] = \text{mscohere}(\text{oupt}, \text{inpt}(:,[1 2]), \text{hamming}(100), [], [], \text{fs}); \]
\[ \text{plot}(f, C_{xy}) \]
Find the phase differences by computing the angle of the cross-spectrum at the points of maximum coherence.

\[
P_{xy} = \text{cpsd}(\text{oupt}, \text{inpt}(:, [1 2]), \text{hamming}(100), [], [], \text{fs});
\]
\[
[-, \text{mxx}] = \text{max}(C_{xy});
\]
\[
\text{for } k = 1:2
\]
\[
\quad \text{fprintf('Phase lag %d = %5.2f*pi
', k, \text{angle}(P_{xy}(\text{mxx}(k), k)))/\text{pi})}
\]
\[
\text{end}
\]

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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
ln = ax.Children;
ln.Color = [0.8 0 0];
ln.LineWidth = 1.5;
```
Alternatively, use `set` and `get` to modify the line properties.

```matlab
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: \(\cos(\pi/4(0:159))+\text{randn}(1,160)\) 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 `ceil(N/4.5)`, 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 = ceil(log2 N)` 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 \([0,\pi]\) rad/sample. If \( nfft \) is odd, \( cxy \) has \((nfft + 1)/2\) rows and the interval is \([0,\pi)\) 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, \(c_{xy}\) has \(n_{fft}\) rows and is computed over the interval \([0, 2\pi]\) 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, \(c_{xy}\) has \(n_{fft}\) rows and is computed over the interval \((-\pi, \pi]\) rad/sample for even \(n_{fft}\) and \((-\pi, \pi)\) rad/sample for odd \(n_{fft}\). If you specify \(fs\), the corresponding intervals are \((-fs/2, fs/2]\) cycles/unit time for even \(n_{fft}\) and \((-fs/2, fs/2)\) cycles/unit time for odd \(n_{fft}\).

**Output Arguments**

\(c_{xy}\) — **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 \(y\):

\[
C_{xy}(f) = \frac{|P_{xy}(f)|^2}{P_{xx}(f)P_{yy}(f)}. 
\]
For multi-input/multi-output systems, the multiple-coherence function becomes

$$C_{Xy_i}(f) = \frac{P_{Xy_i}^T(f)P_{XY}^{-1}(f)P_{Xy_i}(f)}{P_{y_iy_i}(f)}$$

$$= \begin{bmatrix} P_{x_1y_i}(f) & \ldots & P_{x_my_i}(f) \\ P_{x_2y_1}(f) & \ldots & P_{x_2y_m}(f) \\ \vdots & \ddots & \vdots \\ P_{x_my_1}(f) & \ldots & P_{x_my_m}(f) \end{bmatrix}^{-1} \begin{bmatrix} P_{x_1y_i}(f) \\ \vdots \\ P_{x_my_i}(f) \end{bmatrix} \frac{1}{P_{y_iy_i}(f)}$$

for the $i$th output signal, where:

- $X$ corresponds to the array of $m$ inputs.
- $P_{x_iy_i}$ 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.
- $P_{y_iy_i}$ is the power spectral density of the output.
- The dagger ($\dagger$) 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**


**Extended Capabilities**

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

**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

\[
\begin{align*}
\text{w} &= \text{nuttallwin}(N) \\
\text{w} &= \text{nuttallwin}(N, \text{SFLAG})
\end{align*}
\]

Description

\(w = \text{nuttallwin}(N)\) returns a Nuttall defined \(N\)-point, 4-term symmetric Blackman-Harris window in the column vector \(w\). The window is minimum in the sense that its maximum sidelobes are minimized. The coefficients for this window differ from the Blackman-Harris window coefficients computed with \(\text{blackmanharris}\) and produce slightly lower sidelobes.

\(w = \text{nuttallwin}(N, \text{SFLAG})\) uses \(\text{SFLAG}\) window sampling. \(\text{SFLAG}\) can be 'symmetric' or 'periodic'. The default is 'symmetric'. You can find the equations defining the symmetric and periodic windows in “Algorithms” on page 1-1595.

Examples

Nuttall and Blackman-Harris Windows

Compare 64-point Nuttall and Blackman-Harris windows. Plot them using \(\text{wvtool}\).

\[
\begin{align*}
L &= 64; \\
w &= \text{blackmanharris}(L); \\
y &= \text{nuttallwin}(L); \\
wvtool(w,y)
\end{align*}
\]
Compute the maximum difference between the two windows.

\[
\text{max}(\text{abs}(y-w))
\]

\[
\text{ans} = 0.0099
\]

**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 \( n = 0, 1, 2, \ldots, N-1 \).
The equation for the periodic Nuttall defined four-term Blackman-Harris window is

\[ w(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N}\right) + a_2 \cos\left(\frac{4\pi n}{N}\right) - a_3 \cos\left(\frac{6\pi n}{N}\right) \]

where \( n = 0, 1, 2, \ldots, N-1 \). 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


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(____,freqrange,p)

[bw,flo,fhi,power] = obw(____)

obw(____)

Description

bw = obw(x) returns the 99% occupied bandwidth, bw, of the input signal, x.

bw = obw(x,fs) returns the occupied bandwidth in terms of the sample rate, fs.

bw = obw(pxx,f) returns the 99% occupied bandwidth of the power spectral density (PSD) estimate, pxx. The frequencies, f, correspond to the estimates in pxx.

bw = obw(sxx,f,rbw) computes the occupied bandwidth of the power spectrum estimate, sxx. The frequencies, f, correspond to the estimates in sxx. rbw is the resolution bandwidth used to integrate each power estimate.

bw = obw(____,freqrange,p) specifies the frequency interval over which to compute the occupied bandwidth. 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.

```matlab
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).

```matlab
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.

\[
x_2 = 2*\text{chirp}(t,200e3,n\text{Samp}/F_s,300e3);
\]

\[
x_2 = x_2 + \text{randn(size}(x_2)) \ast \text{std}(x_2)/\text{db2mag}(\text{SNR});
\]

Concatenate the chirps to produce a two-channel signal. Estimate the occupied bandwidth of each channel.

\[
y = \text{obw}([x \ x_2],F_s)
\]

\[
y = 1 \times 2 \\
10^5 \times
\]
Annotate the occupied bandwidths of the two channels on a plot of the PSDs.
\[
\text{obw}([x \ x2], \text{Fs});
\]

Add the two channels to form a new signal. Plot the PSD and annotate the occupied bandwidth.
\[
\text{obw}(x+x2, \text{Fs});
\]
**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.

```matlab
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.

```matlab
[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.

```matlab
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.

```matlab
[Pyy,f] = periodogram([x x2],kaiser(nSamp,38),[],Fs);
y = obw(Pyy,f)
```

```
y = 1x2
10^3 x

7.2001    7.3777
```

Annotate the occupied bandwidths of the two channels on a plot of the PSDs.

```matlab
obw(Pyy,f);
```
Add the two channels to form a new signal. Estimate the PSD and annotate the occupied bandwidth.

```matlab
[Pzz,f] = periodogram(x+x2,kaiser(nSamp,38),[],Fs);

obw(Pzz,f);
```
Occupied Bandwidth of Bandlimited Signals

Generate a signal whose PSD resembles the frequency response of an 88th-order bandpass FIR filter with normalized cutoff frequencies $0.25\pi$ rad/sample and $0.45\pi$ rad/sample.

```matlab
d = fir1(88, [0.25 0.45]);
```

Compute the 99% occupied bandwidth of the signal between $0.2\pi$ rad/sample and $0.6\pi$ rad/sample. Plot the PSD and annotate the occupied bandwidth and measurement interval.
Output the occupied bandwidth, its lower and upper bounds, and the occupied band power. Specifying a sample rate of $2\pi$ is equivalent to leaving the rate unset.

```matlab
[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 $0.5\pi$ rad/sample and $0.8\pi$ rad/sample and an amplitude that is one-tenth that of the first channel.

\[ d = [d; \text{fir1}(88,[0.5 0.8])/10]'; \]

Compute the 50% occupied bandwidth of the signal between $0.3\pi$ rad/sample and $0.9\pi$ rad/sample. Plot the PSD and annotate the occupied bandwidth and measurement interval.

\[ \text{obw}(d,[],[0.3 0.9]*\pi,50); \]

Output the occupied bandwidth of each channel. Divide by $\pi$.

\[ \text{bw} = \text{obw}(d,[],[0.3 0.9]*\pi,50)/\pi \]
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

**freqrange — Frequency range**

two-element vector

Frequency range, specified as a two-element vector of real values. If you do not specify freqrange, then obw uses the entire bandwidth of the input signal.

Data Types: single | double

**p — Power percentage**

99 (default) | positive scalar
Power percentage, specified as a positive scalar between 0 and 100. obw computes the difference in frequency between the points where the integrated power crosses the \( \frac{1}{2} (100 - p) \) and \( \frac{1}{2} (100 + p) \) percentages of the total power in the spectrum.

Data Types: single | double

**Output Arguments**

**bw — Occupied bandwidth**

scalar | vector

Occupied bandwidth, returned as a scalar or vector.

- If you specify a sample rate, then bw has the same units as fs.
- If you do not specify a sample rate, then bw has units of rad/sample.

**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.

**See Also**

bandpower | periodogram | plomb | powerbw | pwelch

Introduced in R2015a
**orderspectrum**

Average spectrum versus order for vibration signal

**Syntax**

```matlab
spec = orderspectrum(x,fs,rpm)
[spec,order] = orderspectrum(x,fs,rpm)
[spec,order] = orderspectrum(map,order)
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.

`orderspectrum(____)` with no output arguments plots the RMS amplitude of the order spectrum, scaled linearly, on the current figure.

**Examples**

1-1612
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

\[
\text{plot}(t, \text{rpm}) \\
\text{xlabel('Time (s)')} \\
\text{ylabel('RPM')}
\]

Compute the average order spectrum of the signal. Use the default order resolution.

\[
\text{orderspectrum(vib,fs,rpm)}
\]
Use `rpmordermap` to repeat the computation with a finer order resolution. The lower orders are resolved more clearly.

```matlab
[map,order] = rpmordermap(vib,fs,rpm,0.005);
orderspectrum(map,order)
```
Compute the power level for each estimated order. Display the result in decibels.

```matlab
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: \( \cos(\pi/4*(0:159)) + \text{randn}(1,160) \) 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

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.

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

**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**


**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;
\( x = [a_1 \ a_2 \ a_3 \ a_4] \times \cos([o_1 \ o_2 \ o_3 \ o_4]' \times \phi) \)

Extract and visualize the magnitudes of the orders.

\( \text{ordertrack}(x, fs, rpm, [o_1 \ o_2 \ o_3 \ o_4]) \)

---

**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.

```matlab
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 $4\sqrt{2}$.

```matlab
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.

```matlab
rs = [1+1./(1+linspace(-10,10,nsamp).^4)'/2 ones(nsamp,1)];
x = sum(rs.*x,2);
```

Visualize the orders using `rpmfreqmap`.

```matlab
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.

```matlab
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.

```matlab
load('helidata.mat')

vib = detrend(vib);
```
Compute the order-RPM map. Specify an order resolution of 0.005.

\[
[\text{map,order,rpm,time,}\text{res}] = \text{rpmOrderMap(vib,fs,rpm,0.005)};
\]

Compute and plot the average order spectrum of the signal. Find the three highest peaks of the spectrum.

\[
[\text{spectrumspecorder}] = \text{orderspectrum(map,order)};
\]

\[
[-,pkords] = \text{findpeaks(spectrumspecorder,'SortStr','descend','Npeaks',3)};
\]

findpeaks(spectrumspecorder,'SortStr','descend','Npeaks',3)

![Graph showing the average order spectrum with three highest peaks marked.]

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`

```markdown
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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Decouple',true,'Amplitude','peak'` extracts the specified orders simultaneously and returns the peak amplitude of each order.

```markdown
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.

```markdown
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**


**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**

```matlab
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.

```matlab
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.

```matlab
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: \( \cos(\pi/4 \cdot (0:159)) + \text{randn}(1,160) \) 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

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.

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`.

Data Types: double | single

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. 
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

\( \text{xrec} \) — Reconstructed time-domain order waveforms matrix

Reconstructed time-domain order waveforms, returned as a matrix with one waveform in each column.

References


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 the greatest absolute deviations larger than the final state levels of each transition in the bilevel waveform, X. The overshoots, OS, are expressed as a percentage of the difference between the state levels. The length of OS corresponds to the number of transitions detected in the input signal. The sample instants in X correspond to the vector indices. To determine the transitions, overshoot estimates the state levels of the input waveform by a histogram method. overshoot identifies all intervals which cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-1652.

OS = overshoot(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.

OS = overshoot(X,T) specifies the sample instants, T, as a vector with the same number of elements as X.

[OS,OSLEV,OSINST] = overshoot(...) returns the levels, OSLEV, and sample instants,OSINST, of the overshoots for each transition.
[...] = overshoot(...,Name,Value) returns the greatest deviations larger than the final state level with additional options specified by one or more Name,Value pair arguments.

overshoot(...) plots the bilevel waveform and marks the location of the overshoot of each transition as well as the lower and upper reference-level instants and the associated reference levels. The state levels and associated lower and upper-state boundaries are also plotted.

Input Arguments

X

Bilevel waveform. X is a real-valued row or column vector.

FS

Sample rate in hertz.

T

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

Name-Value Pair Arguments

PercentReferenceLevels

Reference levels as a percentage of the waveform amplitude. The lower-state level is defined to be 0 percent. The upper-state level is defined to be 100 percent. The value of 'PercentReferenceLevels' is a two-element real row vector whose elements correspond to the lower and upper percent reference levels.

Default: [10 90]

Region

Specifies the region over which to compute the overshoot. Valid values for 'Region' are 'Preshoot' or 'Postshoot'. If you specify 'Preshoot', the end of the pretransition aberration region is defined as the last instant where the signal exits the first state. If you
specify 'Postshoot', the start of the posttransition aberration region is defined as the instant when the signal enters the second state.

**Default:** 'Postshoot'

**SeekFactor**

Aberration region duration. Specifies the duration of the region over which to compute the overshoot for each transition as a multiple of the corresponding transition duration. If the edge of the waveform is reached, or a complete intervening transition is detected before the duration aberration region duration elapses, the duration is truncated to the edge of the waveform or the start of the intervening transition.

**Default:** 3

**StateLevels**

Lower and upper state levels. Specifies the levels to use for the lower and upper state levels as a two-element real row vector whose first and second elements correspond to the lower and upper state levels of the input waveform.

**Tolerance**

Specifies the tolerance that the initial and final levels of each transition must be within the respective state levels. The 'Tolerance' value is a scalar expressed as the percentage of the difference between the upper and lower state levels.

**Default:** 2

**Output Arguments**

**OS**

Overshoots expressed as a percentage of the state levels. The overshoot percentages are computed based on the greatest deviation from the final state level in each transition. By default overshoots are computed for posttransition aberration regions. See “Overshoot” on page 1-1651.

**OSLEV**

Level of the pretransition or posttransition overshoot.
OSINST

Sample instants of pretransition or posttransition overshoots. If you specify the sample rate or sampling instants, the overshoot instants are in seconds. If you do not specify the sample rate or sampling instants, the overshoot instants are the indices of the input vector.

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.

```matlab
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.

```matlab
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.
\[\text{os, oslev, osinst} = \text{overshoot}(x, t)\]

\[\text{os} = 6.1798\]
\[\text{oslev} = 2.4276\]
\[\text{osinst} = 5.2500 \times 10^{-06}\]

\text{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.

```matlab
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.

```matlab
[os,oslev,osinst] = overshoot(x,t,'Region','Preshoot')
```

```matlab
os = 4.8050
oslev = 0.1020
osinst = 4.7500e-06
```

```matlab
overshoot(x,t,'Region','Preshoot');
```
More About

Overshoot

For a positive-going (positive-polarity) pulse, overshoot expressed as a percentage is

\[
100 \frac{(O - S_2)}{(S_2 - S_1)}
\]

where \(O\) is the maximum deviation greater the high-state level, \(S_2\) is the high state, and \(S_1\) is the low state.
For a negative-going (negative-polarity) pulse, overshoot expressed as a percentage is

\[
100 \frac{(O - S_1)}{(S_2 - S_1)}
\]

The following figure illustrates the calculation of overshoot for a positive-going transition.

The red dashed lines indicate the estimated state levels. The double-sided black arrow depicts the difference between the high and low-state levels. The solid black line indicates the difference between the overshoot value and the high-state level.

**State-Level Tolerances**

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the \( \alpha \% \) region for the low state is defined as
where \( S_1 \) is the low-state level and \( S_2 \) is the high-state level. Replace the first term in the equation with \( S_2 \) to obtain the \( \alpha \% \) tolerance region for the high state.

The following figure illustrates lower and upper 2\% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.
References


See Also
settlingtime | statelevels

Introduced in R2012a
parzenwin

Parzen (de la Vallée Poussin) window

Syntax

\[ w = \text{parzenwin}(L) \]

Description

\[ w = \text{parzenwin}(L) \] returns the \( L \)-point Parzen (de la Vallée Poussin) window in a column vector, \( w \). Parzen windows are piecewise-cubic approximations of Gaussian windows. Parzen window sidelobes fall off as \( 1/\omega^4 \). See “Algorithms” on page 1-1656 for the equation that defines the Parzen window.

Examples

Parzen and Gaussian Windows

Compare 64-point Parzen and Gaussian windows. Display the result using \text{wvtool}.

\[
gw = \text{gausswin}(64);
pw = \text{parzenwin}(64);
wvtool(gw,pw)
\]
Algorithms

The following equation defines the $N$-point Parzen window over the interval

$$-\frac{(N-1)}{2} \leq n \leq \frac{(N-1)}{2}:$$

$$w(n) = \begin{cases} 
1 - 6\left(\frac{|n|}{N/2}\right)^2 + 6\left(\frac{|n|}{N/2}\right)^3 & 0 \leq |n| \leq (N - 1)/4 \\
2\left(1 - \frac{|n|}{N/2}\right)^3 & (N - 1)/4 < |n| \leq (N - 1)/2 
\end{cases}$$
References


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 \([0, 2\pi]\).

\[
[pxx, f] = \text{pburg}(\_, fs)\]
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 \([0, fs]\).

\[
[pxx, w] = \text{pburg}(x, order, w)\]
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 \(nfft\).

\[
[pxx, f] = \text{pburg}(x, order, f, fs)\]
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).

\[
[\_, \_] = \text{pburg}(x, order, \_, \_, \text{freqrange})\]
returns the AR PSD estimate over the frequency range specified by \(\text{freqrange}\). Valid options for \(\text{freqrange}\) are: 'onesided', 'twosided', or 'centered'.

\[
[\_, pxxc] = \text{pburg}(\_, \_,'\text{ConfidenceLevel}',\text{probability})\]
returns the probability \(\times 100\%\) confidence intervals for the PSD estimate in \(pxxc\).

\text{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.
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
```

```matlab
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)')
```
```matlab
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.

\texttt{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.

$$F_s = 1000;$$
$$t = 0:1/F_s:1-1/F_s;$$
$$f = [100;200;300];$$
\[ x = \cos(2\pi \cdot f \cdot t) + \text{randn}(\text{length}(t), 3); \]

Estimate the PSD of the signal using Burg's method with a 12th-order autoregressive model. Use the default DFT length. Plot the estimate.

\[ \text{morder} = 12; \]

\[ \text{pburg}(x, \text{morder}, [], \text{Fs}) \]
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: \( \cos(\pi/4*(0:159)) + \text{randn}(1,160) \) is a single-channel row-vector signal.

Example: \( \cos(\pi./(4;2)*(0:159))' + \text{randn}(160,2) \) 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, \( \text{pxx} \) has length \( (\text{nfft}/2+1) \) if \( \text{nfft} \) is even, and \( (\text{nfft}+1)/2 \) if \( \text{nfft} \) is odd. For a complex-valued input signal, \( x \), the PSD estimate always has length \( \text{nfft} \). If \( \text{nfft} \) is specified as empty, the default \( \text{nfft} \) 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.

• '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, $\text{pxxc}$, contains the lower and upper bounds of the probability $\times 100\%$ interval estimate for the true PSD.

**Output Arguments**

$p\text{xx} — \text{PSD estimate}$

vector | matrix

PSD estimate, returned as a real-valued, nonnegative column vector or matrix. Each column of $p\text{xx}$ 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

$w — \text{Normalized frequencies}$

vector

Normalized frequencies, returned as a real-valued column vector. If $p\text{xx}$ is a one-sided PSD estimate, $w$ spans the interval $[0,\pi]$ if $n\text{fft}$ is even and $[0,\pi)$ if $n\text{fft}$ is odd. If $p\text{xx}$ 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 $n\text{fft}$ and $(-\pi,\pi)$ for odd $n\text{fft}$.

Data Types: double

$f — \text{Cyclical frequencies}$

vector

Cyclical frequencies, returned as a real-valued column vector. For a one-sided PSD estimate, $f$ spans the interval $[0,fs/2]$ when $n\text{fft}$ is even and $[0,fs/2)$ when $n\text{fft}$ 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 $n\text{fft}$ and $(-fs/2,fs/2)$ cycles/unit time for odd length $n\text{fft}$.

Data Types: double | single

$\text{pxxc} — \text{Confidence bounds}$

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, \( pxxc(m,2\times n-1) \) is the lower confidence bound and \( pxxc(m,2\times n) \) is the upper confidence bound corresponding to the estimate \( pxx(m,n) \). The coverage probability of the confidence intervals is determined by the value of the probability input.

Data Types: single | double

See Also

pcov | pmcov | 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 \([0, n]\) when \(nfft\) is even and \([0, n)\) when \(nfft\) is odd. For complex-valued signals, \(w\) always spans the interval \([0, 2\pi]\).

\[
[pxx,f] = \text{pcov}(\_\_\_,fs)\]

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 \([0,fs]\).

\[
[pxx,w] = \text{pcov}(x,order,w)\]

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 \(nfft\).

\[
[pxx,f] = \text{pcov}(x,order,f,fs)\]

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).

\[
[\_\_\_] = \text{pcov}(x,order,\_\_\_,freqrange)\]

returns the AR PSD estimate over the frequency range specified by \(freqrange\). Valid options for \(freqrange\) are: 'onesided', 'twosided', or 'centered'.

\[
[\_\_,pxxc] = \text{pcov}(\_\_,\text{'ConfidenceLevel'},probability)\]

returns the probability \(\times 100\%\) confidence intervals for the PSD estimate in \(pxxc\).

\(\text{pcov}(\_\_)\) 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.
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.

```matlab
rng default

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)')
```
```matlab
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.

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

Estimate the PSD of the signal using the covariance method with a 12th-order autoregressive model. Use the default DFT length. Plot the estimate.

\[
morder = 12;
\]
\[
\text{pcov}(x,morder,[],Fs)
\]
**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: \( \cos\left(\frac{\pi}{4}*(0:159)\right)+\text{randn}(1,160) \) is a single-channel row-vector signal.  
Example: \( \cos\left(\pi/\left[4;2\right]*(0:159)\right)'+\text{randn}(160,2) \) 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.

- '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
See Also
pb urg | pm cov | pyulear

Introduced before R2006a
peak2peak

Maximum-to-minimum difference

Syntax

\[
y = \text{peak2peak}(x) \\
y = \text{peak2peak}(x,\text{dim})
\]

Description

\[y = \text{peak2peak}(x)\] returns the difference between the maximum and minimum values in \(x\).

\[y = \text{peak2peak}(x,\text{dim})\] computes the maximum-to-minimum differences of \(x\) along dimension \(\text{dim}\).

Examples

Peak-to-Peak Difference of Sinusoid

Compute the maximum-to-minimum difference of a 100 Hz sinusoid sampled at 1 kHz.

\[
t = 0:0.001:1-0.001; \\
x = \cos(2\pi*100*t); \\
y = \text{peak2peak}(x)
\]

\[y = 2\]
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.

```matlab
n = 0:99;
x = exp(1j*pi/4*n);
y = peak2peak(x)
y = 0.0000 + 1.4142i
```

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.

```matlab
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)'*(1:4);
y = peak2peak(x)
y = 1x4
    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.

```matlab
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


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” (MATLAB).

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
max | min | peak2rms | rms | rssq
Introduced in R2012a
**peak2rms**

Peak-magnitude-to-RMS ratio

**Syntax**

\[ y = \text{peak2rms}(x) \]
\[ y = \text{peak2rms}(x,\text{dim}) \]

**Description**

\[ y = \text{peak2rms}(x) \] returns the ratio of the largest absolute value in \( x \) to the root-mean-square (RMS) value of \( x \).

\[ y = \text{peak2rms}(x,\text{dim}) \] computes the peak-magnitude-to-RMS ratio of \( x \) along dimension \( \text{dim} \).

**Examples**

**Peak-Magnitude-to-RMS Ratio of Sinusoid**

Compute the peak-magnitude-to-RMS ratio of a 100 Hz sinusoid sampled at 1 kHz.

\[ t = 0:0.001:1-0.001; \]
\[ x = \cos(2\pi100t); \]
\[ y = \text{peak2rms}(x) \]
\[ y = 1.4142 \]
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.

\[ n = 0:99; \]
\[ x = \exp(1j*\pi/4*n); \]
\[ y = \text{peak2rms}(x) \]
\[ y = 1 \]

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.

\[ t = 0:0.001:1-0.001; \]
\[ x = \cos(2*\pi*100*t)'*(1:4); \]
\[ y = \text{peak2rms}(x) \]
\[ y = 1\times4 \]
\[ \begin{array}{cccc}
1.4142 & 1.4142 & 1.4142 & 1.4142 \\
\end{array} \]

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 \texttt{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 \( \ell \)-infinity norm and RMS values are computed along the specified dimension.

References


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” (MATLAB).

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
max | mean | min | 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.

```matlab
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 \( x' \times x \) 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 $\lambda_{\text{min}}$, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold $\lambda_{\text{min}} \cdot p(2)$ 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 double the number of input signals. If the inputs are complex sinusoids, set `p` equal to the number of inputs.

**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 you 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)

<table>
<thead>
<tr>
<th>Input Data Type</th>
<th>Length of S and wo</th>
<th>Range of the Corresponding Normalized Frequencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>129</td>
<td>[0, π]</td>
</tr>
<tr>
<td>Complex</td>
<td>256</td>
<td>[0, 2π)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Input Data Type</th>
<th>nfft Even or Odd</th>
<th>Length of S and w</th>
<th>Range of w</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>Even</td>
<td>(nfft/2 + 1)</td>
<td>[0, π]</td>
</tr>
<tr>
<td>Real</td>
<td>Odd</td>
<td>(nfft + 1)/2</td>
<td>[0, π]</td>
</tr>
<tr>
<td>Complex</td>
<td>Even or odd</td>
<td>nfft</td>
<td>[0, 2π)</td>
</tr>
</tbody>
</table>

**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 \( f_0 \) if \( nfft \) and \( fs \) are specified.

### S and Frequency Vector Characteristics with \( fs \) Specified

<table>
<thead>
<tr>
<th>Input Data Type</th>
<th>nfft Even/Odd</th>
<th>Range of ( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>Even</td>
<td>([0, fs/2])</td>
</tr>
<tr>
<td>Real</td>
<td>Odd</td>
<td>([0, fs/2))</td>
</tr>
<tr>
<td>Complex</td>
<td>Even or odd</td>
<td>([0, fs))</td>
</tr>
</tbody>
</table>

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 \( nwin \)

<table>
<thead>
<tr>
<th>form of ( x )</th>
<th>Form of ( nwin )</th>
<th>Windowed Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data vector</td>
<td>Scalar</td>
<td>Length is ( nwin ).</td>
</tr>
<tr>
<td>Data vector</td>
<td>Vector of coefficients</td>
<td>Length is ( \text{length}(nwin) ).</td>
</tr>
<tr>
<td>Data matrix</td>
<td>Scalar</td>
<td>Data is not windowed.</td>
</tr>
<tr>
<td>Data matrix</td>
<td>Vector of coefficients</td>
<td>( \text{length}(nwin) ) must be the same as the column length of ( x ), and ( noverlap ) is not used.</td>
</tr>
</tbody>
</table>

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 \( \text{size}(v,2) \). The dimension of the signal subspace is \( \text{size}(v,1) - \text{size}(v,2) \).

### \( e \) — 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

\[ P_{ev}(f) = \frac{1}{\sum_{k=p+1}^{N} \left| v_k^H e(f) \right|^2 / \lambda_k} \]

where \( N \) is the dimension of the eigenvectors and \( v_k \) is the \( k \)th eigenvector of the correlation matrix of the input signal. The integer \( p \) is the dimension of the signal subspace, so the eigenvectors \( v_k \) used in the sum correspond to the smallest eigenvalues \( \lambda_k \) of the correlation matrix. The eigenvectors used span the noise subspace. The vector \( e(f) \) consists of complex exponentials, so the inner product \( v_k^H e(f) \) amounts to a Fourier transform. This is used for computation of the pseudospectrum. The FFT is computed for each \( v_k \) and then the squared magnitudes are summed and scaled.

References


See Also

corrmtx | pburg | peig | periodogram | pmtm | 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 1-1715 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 pair 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. You can use these output arguments with any of the input arguments in previous syntaxes.

pentropy( ___ ) either:
- Plots the spectral entropy against time (Instantaneous is true.)
- Outputs the scalar value for spectral entropy (Instantaneous is false.)

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`.

```matlab
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.

```matlab
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.

```matlab
[se,te] = pentropy(x1,fs);
```

```matlab
subplot(2,1,1)
plot(t,x1)
ylabel('Speech Signal')
xlabel('Time')
substitue(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.

```matlab
[p,fp,tp] = pspectrum(x1,fs,'FrequencyResolution',20,'spectrogram');
```

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.

```matlab
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.

```matlab
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.

```matlab
fs = 100;
t = 0:1/fs:10;
```
```matlab
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`.

```matlab
se = pentropy(x,fs,'Instantaneous',false)
se = 0.9035
```
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” (MATLAB). `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 1-1702.

**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 — 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 \texttt{sampx} represents a time vector, time samples can be nonuniform, with the \texttt{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 1-1702.

\textbf{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 \texttt{p}, then \texttt{pentropy} uses \texttt{p} rather than generate its own spectrogram or power spectrogram. \texttt{fp} and \texttt{tp}, which provide the frequency and time information, must accompany \texttt{p}. Each element of \texttt{p} at the \texttt{i}'th row and the \texttt{j}'th column represents the signal power at the frequency bin centered at \texttt{fp(i)} and the time instance \texttt{tp(j)}.

For an example, see “Plot Spectral Entropy of Speech Signal” on page 1-1705.

\textbf{fp — Frequencies for spectrogram p}

vector

Frequencies for spectrogram or power spectrogram \texttt{p} when \texttt{p} is supplied explicitly to \texttt{pentropy}, specified as a vector in hertz. The length of \texttt{fp} must be equal to the number of rows in \texttt{s}.

\textbf{tp — Time information for spectrogram p}

vector | duration array | datetime array

Time information for power spectrogram or spectrum \texttt{p} when \texttt{p} is supplied explicitly to \texttt{pentropy}, specified as one of the following:

- Vector of time points, whose data type can be numeric, duration, or datetime. The length of vector \texttt{tp} must be equal to the number of columns in \texttt{p}.
- \texttt{duration} scalar that represents the time interval in \texttt{p}. The scalar form of \texttt{tp} can be used only when \texttt{p} is a power spectrogram matrix.
- For the special case where \texttt{p} is a column vector (power spectrum), \texttt{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), \( t_p \) can be a single/double/duration/datetime scalar representing the time point of the spectrum.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

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 the comma-separated pair consisting of 'Instantaneous' and 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 1-1708.

**Scaled — Scale by white noise option**

true (default) | false

Scale by white noise option, specified as the comma-separated pair consisting of 'Scaled' and 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 1-1715. 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 the comma-separated pair consisting of 'FrequencyLimits' and 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 1-1705.

**Timelimits — Time Limits**
full timespan (default) | [t1 t2]

Time limits, specified as the comma-separated pair consisting of 'Timelimits' and 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 1-1715, 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.

For an example, see “Plot Spectral Entropy of Signal” on page 1-1702.
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 [1], [2]. 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**


**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 \( \text{sum}([1 5; 2 6; 3 7; 4 8],2) \).

[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 \([0,fs/2]\) 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 \( \times 100\% \) 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 π/4 rad/sample with additive N(0, 1) white noise.

Create a sine wave with an angular frequency of π/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.

```matlab
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 \(\frac{\pi}{4}\) radians/sample with additive \(N(0, 1)\) white noise.

Create a sine wave with an angular frequency of \(\frac{\pi}{4}\) 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 $320/2+1$.

```matlab
n = 0:319;
x = cos(pi/4*n)+randn(size(n));
nfft = length(x);
periodogram(x,[],nfft)
```

![Periodogram Power Spectral Density Estimate](image)
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.

```matlab
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 = 1x2

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.

```matlab
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.

```matlab
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.

```matlab
[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.

```matlab
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 0.01². 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.

```matlab
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 = \frac{\text{abs}(\text{max}(\text{pre}) - \text{ffts})}{\text{ffts}} \times 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
',f(idx))
The maximum power occurs at 100.0 Hz
fprintf('The power estimate is %2.2f
',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 PSD estimate using the window, `window`. The length of the window and the number of points in the DFT is the same as the length of `inputData`.

```matlab
type periodogram_data

function [pxx,f] = periodogram_data(inputData,window)
    %#codegen
    nfft = length(inputData);
```
[pxx,f] = periodogram(inputData,window,nfft);
end

The %#codegen directive indicates that the MATLAB code is intended for code generation. Use codegen to generate a MEX-file periodogram_mex. The -args option defines the input specifications for the MEX file. inputData is a 1024x1 random vector. Specify a Hamming window of length 1024. To view the code generation report, you can add the flag -report at the end of the following line of code.

codegen periodogram_data -args {randn(1024,1),hamming(1024)} -o periodogram_mex

Plot the periodogram of a noisy sinusoid using both periodogram_mex and periodogram_data.m.

t = linspace(0,1,1024);
x = 2*cos(2*pi*50*t')+randn(size(t'));

[pxMex,fMex] = periodogram_mex(x,hann(length(x)));

[pxMat,fMat] = periodogram_data(x,hamming(length(x)));
plot(fMex,10*log10(pxMex))
hold on
plot(fMat,10*log10(pxMat),'--')
hold off
grid on

title('Periodogram Power Spectral Density Estimate')
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Power/frequency (dB/(rad/sample))')

legend('Periodogram with MEX function','Periodogram with MATLAB 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: \( \cos(\pi/4*(0:159)) + \text{randn}(1,160) \) is a single-channel row-vector signal.
Example: \( \cos(\pi./[4;2]*(0:159))' + \text{randn}(160,2) \) 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
'onethsided' | '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.
- '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 $\times 100\%$ interval estimate for the true PSD.

**Output Arguments**

**pxx — PSD estimate**
vector | matrix

PSD estimate, returned as a real-valued, nonnegative column vector or matrix. Each column of `pxx` is the PSD estimate of the corresponding column of `x`. The units of the PSD estimate are in squared magnitude units of the time series data per unit frequency. For example, if the input data is in volts, the PSD estimate is in units of squared volts per unit frequency. For a time series in volts, if you assume a resistance of 1 $\Omega$ and specify the 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,$\pi$] 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`
**Confidence bounds**

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`

**Reassigned PSD estimate**

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`.

**Center-of-energy frequencies**

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 `f_s` samples per unit time, the periodogram is defined as

\[
\hat{P}(f) = \frac{\Delta t}{N} \left| \sum_{n=0}^{N-1} x_n e^{-j2\pi f n} \right|^2, \quad -1/2\Delta t < f \leq 1/2\Delta t,
\]

where \(\Delta t\) is the sampling interval. For a one-sided periodogram, the values at all frequencies except 0 and the Nyquist, \(1/2\Delta t\), are multiplied by 2 so that the total power is conserved.
If the frequencies are in radians/sample, the periodogram is defined as

$$\hat{P}(\omega) = \frac{1}{2\pi N} \left| \sum_{n=0}^{N-1} x_n e^{-j\omega n} \right|^2, \quad -\pi < \omega \leq \pi.$$ 

The frequency range in the preceding equations has variations depending on the value of the freqrange argument. See the description of freqrange in “Input Arguments” on page 1-1739.

The integral of the true PSD, $P(f)$, over one period, $1/\Delta t$ for cyclical frequency and $2\pi$ for normalized frequency, is equal to the variance of the wide-sense stationary random process:

$$\sigma^2 = \int_{-1/2\Delta t}^{1/2\Delta t} P(f) \, df.$$ 

For normalized frequencies, replace the limits of integration appropriately.

## Modified Periodogram

The modified periodogram multiplies the input time series by a window function. A suitable window function is nonnegative and decays to zero at the beginning and end points. Multiplying the time series by the window function tapers the data gradually on and off and helps to alleviate the leakage in the periodogram. See “Bias and Variability in the Periodogram” for an example.

If $h_n$ is a window function, the modified periodogram is defined by

$$\hat{P}(f) = \frac{\Delta t}{N} \left| \sum_{n=0}^{N-1} h_n x_n e^{-j2\pi fn} \right|^2, \quad -1/2\Delta t < f \leq 1/2\Delta t,$$

where $\Delta t$ is the sampling interval.

If the frequencies are in radians/sample, the modified periodogram is defined as

$$\hat{P}(\omega) = \frac{1}{2\pi N} \left| \sum_{n=0}^{N-1} h_n x_n e^{-j\omega n} \right|^2, \quad -\pi < \omega \leq \pi.$$
The frequency range in the preceding equations has variations depending on the value of the \texttt{freqrange} argument. See the description of \texttt{freqrange} in “Input Arguments” on page 1-1739.

**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**


**Extended Capabilities**

**C/C++ Code Generation**

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

**See Also**

**Apps**

- Signal Analyzer

**Functions**

- \texttt{bandpower}
- \texttt{pburg}
- \texttt{pcov}
- \texttt{plomb}
- \texttt{pmcov}
- \texttt{pmtm}
- \texttt{pspectrum}
- \texttt{pwelch}
- \texttt{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**

\[
[\phi, w] = \text{phasedelay}(b, a, n)
\]
\[
[\phi, w] = \text{phasedelay}(sos, n)
\]
\[
[\phi, w] = \text{phasedelay}(d, n)
\]
\[
[\phi, w] = \text{phasedelay}(\ldots, n, 'whole')
\]
\[
\phi = \text{phasedelay}(\ldots, w)
\]
\[
[\phi, f] = \text{phasedelay}(\ldots, n, fs)
\]
\[
[\phi, f] = \text{phasedelay}(\ldots, n, 'whole', fs)
\]
\[
\phi = \text{phasedelay}(\ldots, f, fs)
\]
\[
[\phi, w, s] = \text{phasedelay}(\ldots)
\]
\[
[\phi, f, s] = \text{phasedelay}(\ldots)
\]
\[
\text{phasedelay}(\ldots)
\]

**Description**

\[
[\phi, w] = \text{phasedelay}(b, a, n) \]
returns the \(n\)-point phase delay response vector, \(\phi\), and the \(n\)-point frequency vector in radians/sample, \(w\), of the filter defined by numerator coefficients, \(b\), and denominator coefficients, \(a\). The phase delay response is evaluated at \(n\) equally spaced points around the upper half of the unit circle. If \(n\) is omitted, it defaults to 512. For best results, set \(n\) to a value greater than the filter order.

\[
[\phi, w] = \text{phasedelay}(sos, n) \]
returns the \(n\)-point phase delay response for the second-order sections matrix, \(sos\). \(sos\) is a \(K\)-by-6 matrix, where the number of sections, \(K\), must be greater than or equal to 2. If the number of sections is less than 2, \text{phasedelay} considers the input to be a numerator vector, \(b\). Each row of \(sos\) corresponds to the coefficients of a second order (biquad) filter. The \(i\)th row of the \(sos\) matrix corresponds to \([bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]\).

\[
[\phi, w] = \text{phasedelay}(d, n) \]
returns the \(n\)-point phase delay response of the digital filter, \(d\). Use \text{designfilt} to generate \(d\) based on frequency-response specifications.
[\phi,w] = \text{phasedelay}(\ldots,n,'\text{whole}') \text{ uses } n \text{ equally spaced points around the whole unit circle.}

\phi = \text{phasedelay}(\ldots,w) \text{ returns the phase delay response at frequencies specified, in radians/sample, in vector } w. \text{ The frequencies are normally between } 0 \text{ and } \pi. \text{ } w \text{ must contain at least two elements.}

[\phi,f] = \text{phasedelay}(\ldots,n,fs) \text{ and } [\phi,f] = \text{phasedelay}(\ldots,n,'\text{whole}',fs) \text{ return the phase delay vector } f \text{ (in Hz), using the sampling frequency } fs \text{ (in Hz). } f \text{ must contain at least two elements.}

\phi = \text{phasedelay}(\ldots,f,fs) \text{ returns the phase delay response at the frequencies specified in vector } f \text{ (in Hz), using the sampling frequency } fs \text{ (in Hz).}

[\phi,w,s] = \text{phasedelay}(\ldots) \text{ and } [\phi,f,s] = \text{phasedelay}(\ldots) \text{ return plotting information, where } s \text{ is a structure with fields you can change to display different frequency response plots.}

\text{phasedelay}(\ldots) \text{ with no output arguments plots the phase delay response versus frequency.}

\textbf{Note} \text{ If the input to phasedelay is single precision, the phase delay response is calculated using single-precision arithmetic. The output, } \phi, \text{ is single precision.}

\textbf{Examples}

\textbf{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 = \text{fircls1}(54,0.3,Ap,As);
\text{phasedelay}(b)
Repeat the example using `designfilt`. Keep in mind that this function expresses the ripples in decibels.

```matlab
Apd = 40*log10((1+Ap)/(1-Ap));
Asd = -20*log10(As);
d = designfilt('lowpassfir','FilterOrder',54,'CutoffFrequency',0.3, ... 'PassbandRipple',Apd,'StopbandAttenuation',Asd);
phasedelay(d)
```
Phase Delay Response of an Elliptic Filter

Design an elliptic filter of order 10 and normalized passband frequency 0.4. Specify a passband ripple of 0.5 dB and a stopband attenuation of 20 dB. Display the phase delay response of the filter over the complete unit circle.

\[ [b,a] = \text{ellip}(10,0.5,20,0.4); \]
\[ \text{phasedelay}(b,a,512,'\text{whole}') \]
Repeat the example using `designfilt`.

```matlab
d = designfilt('lowpassiir','DesignMethod','ellip','FilterOrder',10, ... 'PassbandFrequency',0.4, ... 'PassbandRipple',0.5,'StopbandAttenuation',20);
phasedelay(d,512,'whole')
```
See Also

FVTool | designfilt | digitalFilter | freqz | grpdelay | phasez

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 = phasez(..., w)
[phi, f] = phasez(..., n, fs)
phi = phasez(... f, fs)
[phi, w, s] = phasez(....)
phasez(...)

Description

[phi, w] = phasez(b, a, n) returns the n-point unwrapped phase response vector, phi, in radians and the frequency vector, w, in radians/sample for the filter coefficients specified in b and a. The values of the frequency vector, w, range from 0 to \( \pi \). If n is omitted, the length of the phase response vector defaults to 512. For best results, set n to a value greater than the filter order.

[phi, w] = phasez(sos, n) returns the unwrapped phase response for the second order sections matrix, sos. sos is a K-by-6 matrix, where the number of sections, K, must be greater than or equal to 2. If the number of sections is less than 2, phasez considers the input to be the numerator vector, b. Each row of sos corresponds to the coefficients of a second-order (biquad) filter. The i\text{th} row of the sos matrix corresponds to [bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)].

[phi, w] = phasez(d, n) returns the unwrapped phase response for the digital filter, d. Use designfilt to generate d based on frequency-response specifications.

[phi, w] = phasez(..., n, 'whole') returns frequency and unwrapped phase response vectors evaluated at n equally-spaced points around the unit circle from 0 to 2\( \pi \) radians/sample.
phi = phasez(...,w) returns the unwrapped phase response in radians at frequencies specified in w (radians/sample). The frequencies are normally between 0 and π. The vector w must have at least two elements.

[phi,f] = phasez(...,n,fs) return the unwrapped phase vector phi in radians and the frequency vector in hertz. The frequency vector ranges from 0 to the Nyquist frequency, fs/2. If the 'whole' option is used, the frequency vector ranges from 0 to the sampling frequency.

phi = phasez(...f,fs) return the phase response in radians at the frequencies specified in the vector f (in hertz) using the sampling frequency fs (in hertz). The vector f must have at least two elements.

[phi,w,s] = phasez(...) return plotting information, where s is a structure array with fields you can change to display different frequency response plots.

phasez(...) with no output arguments plots the phase response of the filter. If you input the filter coefficients or second order sections matrix, the current figure window is used. If you input a digitalFilter, the step response is displayed in FVTool.

**Note** If the input to phasez is single precision, the phase response is calculated using single-precision arithmetic. The output, phi, is single precision.

### Examples

#### Phase Response of an FIR Filter

Use designfilt to design an FIR filter of order 54, normalized cutoff frequency 0.3π 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.

```matlab
Nf = 54;
Fc = 0.3;
Ap = 0.7;
As = 42;

d = designfilt('lowpassfir','CutoffFrequency',Fc,'FilterOrder',Nf,...
              'PassbandRipple',Ap,'StopbandAttenuation',As,...
              'DesignMethod','cls');

phasez(d)
```
Design the same filter using **fircls1**. Keep in mind that **fircls1** uses linear units to measure the ripple and attenuation.

```
pAp = 10^(Ap/40);
Apl = (pAp-1)/(pAp+1);

pAs = 10^(As/20);
Asl = 1/pAs;

b = fircls1(Nf,Fc,Apl,Asl);
phasez(b)
```
Phase Response of an Equiripple Filter

Design a lowpass equiripple filter with normalized passband frequency $0.45\pi$ rad/s, normalized stopband frequency $0.55\pi$ rad/s, passband ripple 1 dB, and stopband attenuation 60 dB. Display the phase response of the filter.

```matlab
d = designfilt('lowpassfir', ...
    'PassbandFrequency',0.45,'StopbandFrequency',0.55, ...
    'PassbandRipple',1,'StopbandAttenuation',60, ...
    'DesignMethod','equiripple');
phasez(d)
```
Phase Response of an Elliptic Filter

Design an elliptic lowpass IIR filter with normalized passband frequency $0.4\pi$ rad/s, normalized stopband frequency $0.5\pi$ rad/s, passband ripple 1 dB, and stopband attenuation 60 dB. Display the phase response of the filter.

```matlab
d = designfilt('lowpassiir', ...    'PassbandFrequency',0.4,'StopbandFrequency',0.5, ...    'PassbandRipple',1,'StopbandAttenuation',60, ...    'DesignMethod','ellip');

phasez(d)
```
See Also
FVTool | designfilt | digitalFilter | freqz | grpdelay | phasedelay

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 1-1773 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 threshold-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 = 1x2

    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.

```plaintext
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.

```plaintext
pkurtosis(x,fs)
title('Spectral Kurtosis with Default Window Size')
```
Now compute the optimal window size using kurtogram.

\texttt{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 $w_0$ into $pkurtosis$.

\[
w_0 = 256;
\]
\[
pkurtosis(x,fs,w_0)
\]
\[
title('Spectral Kurtosis with Optimum Window Size of 256')
\]
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 \( \text{pkurtosis} \) 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” (MATLAB). `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 \( \text{pkurtosis} \) treats \( s \) as a short-time Fourier transform (STFT) of the original signal (spectrogram).
- If \( s \) is real, then \( \text{pkurtosis} \) 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 \( s \), \( \text{pkurtosis} \) uses \( s \) rather than generate its own spectrogram or power spectrogram. For an example, see “Plot Spectral Kurtosis Using a Customized Spectrogram” on page 1-1768.

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 \( \text{pkurtosis} \), 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 \text{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 1-1760.

Output Arguments

sk — Spectral kurtosis
double vector
Spectral Kurtosis on page 1-1773, 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 1-1760.

**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 [1], [2].

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, $S(t,f)$:
\[ S(t, f) = \int_{-\infty}^{+\infty} x(t)w(t-\tau)e^{-2\pi ft}d\tau, \]

where \( w(t) \) is the window function used in STFT. \( K(f) \) is calculated as:

\[ K(f) = \frac{\langle |S(t, f)|^4 \rangle}{\langle |S(t, f)|^2 \rangle^2} - 2, \quad f \neq 0, \]

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 \( s_\alpha \) given a confidence level \( \alpha \) is:

\[ s_\alpha = \Phi^{-1}(\alpha) \frac{2}{\sqrt{M}}, \]

where \( \Phi^{-1} \) 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 frequency. The window size must be shorter than the spacing between transient impulses, but longer than the individual transient impulses.

**References**


**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 NaNs or NaTs. 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 NaNs.
[pxx,f] = plomb(__,fmax) estimates the PSD up to a maximum frequency, fmax, using any of the input arguments from previous syntaxes. If the signal is sampled at N non-NaN instants, and Δt is the time difference between the first and the last of them, then pxx is returned at round(fmax/fmin) points, where fmin = 1/(4 × N × ts) is the smallest frequency at which pxx is computed and the average sample time is ts = Δt/(N – 1). fmax defaults to 1/(2 × ts), which for uniformly sampled signals corresponds to the Nyquist frequency.

[pxx,f] = plomb(__,fmax,ofac) specifies an integer oversampling factor, ofac. The use of ofac to interpolate or smooth a spectrum resembles the zero-padding technique for FFT-based methods. pxx is again returned at round(fmax/fmin) frequency points, but the minimum frequency considered in this case is 1/(ofac × N × ts). ofac defaults to 4.

[pxx,fvec] = plomb(__,fvec) estimates the PSD of x at the frequencies specified in fvec. fvec must have at least two elements. The second output argument is the same as the input fvec.

You cannot specify a maximum frequency or an oversampling factor if you use this syntax.

[___] = plomb(__,spectrumtype) specifies the normalization of the periodogram.

• Set spectrumtype to 'psd', or leave it unspecified, to obtain pxx as a power spectral density.
• Set spectrumtype to 'power' to get the power spectrum of the input signal.
• Set spectrumtype to 'normalized' to get the standard Lomb-Scargle periodogram, which is scaled by two times the variance of x.

[___,pth] = plomb(__,'Pd',pdvec) returns the power-level threshold, pth, such that a peak with a value larger than pth has a probability pdvec of being a true signal peak and not the result of random fluctuations. pdvec can be a vector. Every element of pdvec must be greater than 0 and smaller than 1. Each row of pth corresponds to an element of pdvec. pth has the same number of channels as x. This option is not available if you specify the output frequencies in fvec.

[pxx,w] = plomb(x) returns the PSD estimate of x evaluated at a set of evenly spaced normalized frequencies, w, spanning the Nyquist interval. Use NaNs to specify missing samples. All of the above options are available for normalized frequencies. To access them, specify an empty array as the second input.
plomb( ___ ) with no output arguments plots the Lomb-Scargle periodogram PSD estimate in the current figure window.

**Examples**

**Irregularly Sampled Signal and Signal with Missing Samples**

The Lomb-Scargle method can handle signals that have been sampled unevenly or have missing samples.

Generate a two-channel sinusoidal signal sampled at 1 kHz for about 0.5 s. The sinusoid frequencies are 175 Hz and 400 Hz. Embed the signal in white noise with variance $\sigma^2 = 1/4$.

```matlab
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.

```matlab
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.

```plaintext
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.

```matlab
sinMiss = sigOrig;
misfrac = 0.1;
nTime = length(t)*2;

sinMiss(randperm(nTime,round(nTime*misfrac))) = NaN;
plomb(sinMiss,t)
```
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.

```matlab
tirr = t + (1/2 - rand(size(t))) / Fs / 2;
tirr(1) = 0;

sinIrreg = sin(2*pi*[f0;f1]*tirr)' + wgn;
plomb(sinIrreg, tirr)
```

```matlab
axis(axisLim)
title('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.

```matlab
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 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.

```matlab
[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.

```matlab
[pk,f0] = findpeaks(pxx,f,'MinPeakHeight',10);
plot(f,pxx,f0,pk,'o')
xlabel('Frequency (day^-1)')
```
Determine Callisto's orbital period (in days) as the inverse of the frequency of maximum energy. The result differs by less than 1% from the value published by NASA.

\[ \text{Period} = \frac{1}{f_0} \]

\[ \text{Period} = 16.6454 \]

\[ \text{NASA} = 16.6890184; \]
\[ \text{PercentDiscrep} = \frac{(\text{Period} - \text{NASA})}{\text{NASA}} \times 100 \]

\[ \text{PercentDiscrep} = -0.2613 \]
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.

```plaintext
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.

```matlab
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.
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 = 3x1

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.

```matlab
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.

\texttt{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.

\texttt{tn = sort(100*rand(1,100));}
\texttt{n = sin(2*pi*f0*tn) + wgn;}

![Lomb-Scargle Power Spectral Density Estimate](image-url)
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
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

plomb(n,tn,fvec)
legend('ofac = 2','df = 0.001')
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 $\text{pwgn}$ and a histogram of a set of exponentially distributed random numbers generated using the distribution function $f(z|1) = e^{-z}$. To normalize the histograms, recall that the total number of periodogram samples is $N \times \text{ofac}/2$. Specify a bin width of 0.25. Overlay a plot of the theoretical distribution function.

dx = 0.25;
br = 0:dx:5;

Nf = N*ofac/2;

hpwgn = histcounts(pwgn,br)';
hRand = histcounts(-log(rand(Nf,1)),br)';

bend = br(1:end-1);

bar(bend,[hpwgn hRand]/Nf/dx,'histc')
hold on
plot(br,exp(-br))
legend('wgn','Empirical pdf','Theoretical pdf')
hold off
Embed in the noise a sinusoidal signal of frequency 0.1 Hz. Use a signal-to-noise ratio of $\xi = 0.01$. Specify the sinusoid amplitude, $x_0$, using the relation $x_0 = \sigma\sqrt{2\xi}$. Compute the power spectrum of the signal and plot its histogram alongside the empirical and theoretical distribution functions.

```matlab
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);`
```
Repeat the calculation using $\xi = 1$. The distribution now differs noticeably.

```matlab
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 \( \frac{0.6}{2\pi} \approx 0.096 \) Hz. Embed the signal in white noise of variance 0.902. Reset the random number generator for repeatable results.

```matlab
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.

\[
P_{fa} = \left[ \frac{50}{100}, \frac{10}{100}, \frac{1}{100}, \frac{0.01}{100} \right] = \left[ 0.50, 0.10, 0.01, 0.0001 \right];
\]

\[
P_d = 1 - P_{fa};
\]

\[
[pxx,f,pth] = plomb(X,1:Nsamp,'normalized','Pd',Pd);
\]

\[
plot(f,pxx,f,pth*ones(size(f')))
\]

\[
xlabel('f')
\]

\[
text(0.3*[1 1 1 1],pth-.5,[repmat('P_{fa} = ','[4 1]) num2str(Pfa')])
\]
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 $1/100$.

```matlab
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.

\[ [p,f] = \text{plomb}(x); \]
\[ [pper,fper] = \text{periodogram}(x); \]

Plot the PSD estimates in decibels. Verify that the results are equivalent.

\[ \text{plot}(f/\pi, \text{pow2db}(p)) \]
\[ \text{hold on} \]
\[ \text{plot}(fper/\pi, \text{pow2db}(pper)) \]

\[ \text{axis}([0 1 -40 20]) \]
\[ \text{xlabel}('\omega / \pi') \]
\[ \text{ylabel}('PSD') \]
\[ \text{legend}('plomb','periodogram') \]
Estimate the Lomb-Scargle PSD of a three-channel signal composed of sinusoids. Specify the frequencies as $\frac{2\pi}{3}$ rad/sample, $\frac{\pi}{2}$ rad/sample, and $\frac{2\pi}{5}$ 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.

```matlab
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.

\[ x_3(\text{randperm(numel(x3),0.25*:numel(x3)))} = \text{NaN}; \]

\[ \text{plomb(x3)} \]
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.

```matlab
t = (0:1023)';
x = sin(2*pi*t/5);
x = x + randn(size(x))/10;
```

![Lomb-Scargle Power Spectral Density Estimate](image)
```matlab
[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.

```matlab
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.

```matlab
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 plobm computes the PSD estimate independently for each column and returns it in the corresponding column of pxx. x can contain NaNs. NaNs are treated as missing data and are excluded from the spectrum computation.

Data Types: single | double
**t — Time instants**
nonnegative real vector | 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 NaNs or NaTs. 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 \textit{pth} corresponds to an element of \textit{pdvec}. \textit{pth} has the same number of channels as \textit{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 \(k = 1, \ldots, N\). The Lomb-Scargle periodogram is defined by [1]

\[
P_{LS}(f) = \frac{1}{2\sigma^2} \left\{ \left[ \sum_{k=1}^{N} (x_k - \bar{x}) \cos(2\pi f (t_k - \tau)) \right]^2 + \left[ \sum_{k=1}^{N} (x_k - \bar{x}) \sin(2\pi f (t_k - \tau)) \right]^2 \right\},
\]

where

\[
\bar{x} = \frac{1}{N} \sum_{k=1}^{N} x_k
\]

and

\[
\sigma^2 = \frac{1}{N-1} \sum_{k=1}^{N} (x_k - \bar{x})^2
\]

are respectively the mean and the variance of the data.

The time offset, \(\tau\), is chosen as

\[
\tan(2(2\pi f)\tau) = \frac{\sum_{k=1}^{N} \sin(2(2\pi f) t_k)}{\sum_{k=1}^{N} \cos(2(2\pi f) t_k)}
\]

to guarantee the time invariance of the computed spectrum. Any shift \(t_k \rightarrow t_k + T\) in the time measurements results in an identical shift in the offset: \(\tau \rightarrow \tau + T\). Moreover, the choice ensures that “a maximum in the periodogram occurs at the same frequency which
minimizes the sum of squares of the residuals of the fit of a sine wave to the data." [2] The offset depends only on the measurement times and vanishes when the times are equally spaced.

If the input signal consists of white Gaussian noise, then $P_{LS}(f)$ follows an exponential probability distribution with unit mean [3].

References


Extended Capabilities

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

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**

```matlab
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 $[0, 2\pi)$.

$$[pxx, f] = \text{pmcov}(\_\_\_, fs)$$ 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 $[0, fs)$.

$$[pxx, w] = \text{pmcov}(x, order, w)$$ 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 $nfft$.

$$[pxx, f] = \text{pmcov}(x, order, f, fs)$$ 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).

$$[\_\_] = \text{pmcov}(x, order, \_\_, freqrange)$$ returns the AR PSD estimate over the frequency range specified by freqrange. Valid options for freqrange are: 'onesided', 'twosided', or 'centered'.

$$[\_\_, pxxc] = \text{pmcov}(\_\_, 'ConfidenceLevel', probability)$$ returns the probability $\times 100\%$ 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.
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
```matlab
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.

\[
F_s = 1000; \\
t = 0:1/Fs:1-1/Fs; \\
f = [100;200;300]; \\
x = \cos(2\pi f^t)' + \text{randn(length(t),3)};
\]

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: \( \cos(\pi/4 \cdot (0:159)) + \text{randn}(1,160) \) is a single-channel row-vector signal.
Example: \( \cos(\pi \cdot [4;2] \cdot (0:159))' + \text{randn}(160,2) \) 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 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.
- '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

\( \mathbf{w} \) — Normalized frequencies
vector

Normalized frequencies, returned as a real-valued column vector. If \( \mathbf{pxx} \) is a one-sided PSD estimate, \( \mathbf{w} \) spans the interval \([0, \pi]\) if \( \mathbf{nfft} \) is even and \([0, \pi)\) if \( \mathbf{nfft} \) is odd. If \( \mathbf{pxx} \) is a two-sided PSD estimate, \( \mathbf{w} \) spans the interval \([0, 2\pi]\). For a DC-centered PSD estimate, \( \mathbf{w} \) spans the interval \((-\pi, \pi]\) for even \( \mathbf{nfft} \) and \((-\pi, \pi)\) for odd \( \mathbf{nfft} \).

Data Types: double

\( \mathbf{f} \) — Cyclical frequencies
vector

Cyclical frequencies, returned as a real-valued column vector. For a one-sided PSD estimate, \( \mathbf{f} \) spans the interval \([0, \mathbf{fs}/2]\) when \( \mathbf{nfft} \) is even and \([0, \mathbf{fs}/2)\) when \( \mathbf{nfft} \) is odd. For a two-sided PSD estimate, \( \mathbf{f} \) spans the interval \([0, \mathbf{fs})\). For a DC-centered PSD estimate, \( \mathbf{f} \) spans the interval \((-\mathbf{fs}/2, \mathbf{fs}/2]\) cycles/unit time for even length \( \mathbf{nfft} \) and \((-\mathbf{fs}/2, \mathbf{fs}/2)\) cycles/unit time for odd length \( \mathbf{nfft} \).

Data Types: double | single

\( \mathbf{pxxc} \) — 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, \( \mathbf{pxx} \). \( \mathbf{pxxc} \) has twice as many columns as \( \mathbf{pxx} \). Odd-numbered columns contain the lower bounds of the confidence intervals, and even-numbered columns contain the upper bounds. Thus, \( \mathbf{pxxc}(m, 2n-1) \) is the lower confidence bound and \( \mathbf{pxxc}(m, 2n) \) is the upper confidence bound corresponding to the estimate \( \mathbf{pxx}(m, n) \). The coverage probability of the confidence intervals is determined by the value of the probability input.

Data Types: single | double
See Also
pburg | pcov | pyulear

Introduced before R2006a
pmtm
Multitaper power spectral density estimate

Syntax

pxx = pmtm(x)
pxx = pmtm(x,nw)
pxx = pmtm(x,nw,nfft)

[pxx,w] = pmtm( ___ )
[pxx,f] = pmtm( ___ ,fs)

[pxx,w] = pmtm(x,nw,w)
[pxx,f] = pmtm(x,nw,f,fs)

[ ___ ] = pmtm( ___ ,method)
[ ___ ] = pmtm(x,e,v)
[ ___ ] = pmtm(x,dpss_params)

[ ___ ] = pmtm( ___ ,[DropLastTaper'],dropflag)
[ ___ ] = pmtm( ___ ,freqrange)
[ ___ ,pxxc] = pmtm( ___ ,[ConfidenceLevel'],probability)

pmtm( ___ )

Description

pxx = pmtm(x) returns Thomson’s multitaper power spectral density (PSD) estimate, pxx, of the input signal, x. When x is a vector, it is treated as a single channel. When x is a matrix, the PSD is computed independently for each column and stored in the corresponding column of pxx. The tapers are the discrete prolate spheroidal (DPSS), or Slepian, sequences. The time-halfbandwidth, nw, product is 4. By default, pmtm uses the first 2 × nw - 1 DPSS sequences. If x is real-valued, pxx is a one-sided PSD estimate. If x is complex-valued, pxx is a two-sided PSD estimate. The number of points, nfft, in the
discrete Fourier transform (DFT) is the maximum of 256 or the next power of two greater than the signal length.

\[ pxx = \text{pmtm}(x, nw) \] use the time-halfbandwidth product, \( nw \), to obtain the multitaper PSD estimate. The time-halfbandwidth product controls the frequency resolution of the multitaper estimate. \text{pmtm} uses \( 2 \times nw - 1 \) Slepian tapers in the PSD estimate.

\[ pxx = \text{pmtm}(x, nw, nfft) \] uses \( nfft \) points in the DFT. If \( nfft \) is greater than the signal length, \( x \) is zero-padded to length \( nfft \). If \( nfft \) is less than the signal length, the signal is wrapped modulo \( nfft \).

\[ [pxx,w] = \text{pmtm}(____) \] returns the normalized frequency vector, \( w \). If \( pxx \) is a one-sided PSD estimate, \( w \) spans the interval \([0,\pi]\) if \( nfft \) is even and \([0,\pi)\) if \( nfft \) is odd. If \( pxx \) is a two-sided PSD estimate, \( w \) spans the interval \([0,2\pi)\).

\[ [pxx,f] = \text{pmtm}(____, fs) \] 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 \([0,fs]\). \( fs \) must be the fourth input to \text{pmtm}. To input a sample rate and still use the default values of the preceding optional arguments, specify these arguments as empty, [ ].

\[ [pxx,w] = \text{pmtm}(x, nw, w) \] returns the two-sided multitaper PSD estimates at the normalized frequencies specified in \( w \). The vector \( w \) must contain at least two elements, because otherwise the function interprets it as \( nfft \).

\[ [pxx,f] = \text{pmtm}(x, nw, f, fs) \] returns the two-sided multitaper 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).

\[ [____] = \text{pmtm}(____, method) \] combines the individual tapered PSD estimates using the method, \( method \). \( method \) can be one of: 'adapt' (default), 'eigen', or 'unity'.

\[ [____] = \text{pmtm}(x, e, v) \] uses the tapers in the N-by-K matrix \( e \) with concentrations \( v \) in the frequency band \([-w,w]\). \( N \) is the length of the input signal, \( x \). Use \text{dpss} to obtain the Slepian tapers and corresponding concentrations.

\[ [____] = \text{pmtm}(x, dpss\_params) \] uses the cell array, \( dpss\_params \), to pass input arguments to \text{dpss} except the number of elements in the sequences. The number of
elements in the sequences is the first input argument to dpss and is not included in dpss_params. An example of this usage is \( \text{pxx} = \text{pmtm} \left( \text{randn}(1000,1),\{2.5,3\} \right) \).

\[
[\_\_] = \text{pmtm}(\_,\_\_,'\text{DropLastTaper}',\text{dropflag})
\]

specifies whether pmtm drops the last taper in the computation of the multitaper PSD estimate. \text{dropflag} is a logical. The default value of \text{dropflag} is true and the last taper is not used in the PSD estimate.

\[
[\_\_\_\_] = \text{pmtm}(\_,\_\_,\_\_,\text{freqrange})
\]

returns the multitaper PSD estimate over the frequency range specified by \text{freqrange}. Valid options for \text{freqrange} are 'onesided', 'twosided', and 'centered'.

\[
[\_\_,\text{pxxc}] = \text{pmtm}(\_,\_\_,\_\_,\_\_,\_\_\_,\text{ConfidenceLevel}',\text{probability})
\]

returns the probability \( \times 100\% \) confidence intervals for the PSD estimate in \text{pxxc}.

\text{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.

\[
n = 0:319;
x = \cos(\pi/4*n)+\text{randn(size}(n));
\text{pxx} = \text{pmtm}(x);
\]

Plot the multitaper PSD estimate.

\[
\text{pmtm}(x)
\]
Specify Time-Halfbandwidth Product

Obtain the multitaper PSD estimate with a specified time-halfbandwidth product.

Create a sine wave with an angular frequency of $\pi/4$ rad/sample with additive $N(0,1)$ white noise. The signal is 320 samples in length. Obtain the multitaper PSD estimate with a time-halfbandwidth product of 2.5. The resolution bandwidth is $[-2.5\pi/320, 2.5\pi/320]$ rad/sample. The default number of DFT points is 512. Because the signal is real-valued, the PSD estimate is one-sided and there are $512/2 + 1$ points in the PSD estimate.
\[ n = 0:319; \]
\[ x = \cos(\pi/4*n) + \text{randn(size(n))}; \]
\[ \text{pmtm}(x,2.5) \]

**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 $320/2+1$.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
pmtm(x,3,length(x),fs,'unity')
```
DPSS Sequences and Their Frequency-Domain Concentrations

This example examines the frequency-domain concentrations of the DPSS sequences. The example produces a multitaper PSD estimate of an input signal by precomputing the Slepian sequences and selecting only those with more than 99% of their energy concentrated in the resolution bandwidth.

The signal is a 100 Hz sine wave in additive $N(0,1)$ white noise. The signal duration is 2 s.
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.

\[
[e,v] = \text{dpss}(\text{length}(x), 3.5, 10);
\]

stem(1:length(v), v, 'filled')
ylim([0 1.2])
title('Proportion of Energy in \([-w,w]\) of k-th Slepian Sequence')
Determine the number of Slepian sequences with energy concentrations greater than 99%. Using the selected DPSS sequences, obtain the multitaper PSD estimate. Set 'DropLastTaper' to false to use all the selected tapers.

```matlab
hold on
plot(1:length(v),0.99*ones(length(v),1))
```

![Proportion of Energy in [-w,w] of k-th Slepian Sequence](image)

```matlab
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.
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.

```matlab
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.

\[ \text{pmtm}(x, 3.5, \text{length}(x), fs, 'centered') \]

**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.

```matlab
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.

```matlab
[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: \( \cos(\pi/4 \times (0:159)) + \text{randn}(1,160) \) is a single-channel row-vector signal.

Example: \( \cos(\pi./[4;2] \times (0:159))' + \text{randn}(160,2) \) is a two-channel signal.

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

\( nw \) — Time-halfbandwidth product

4 (default) | positive scalar

Time-halfbandwidth product, specified as a positive scalar. In multitaper spectral estimation, the user specifies the resolution bandwidth of the multitaper estimate \([-W,W]\) where \( W = k/N \Delta t \) for some small \( k > 1 \). Equivalently, \( W \) is some small multiple of the frequency resolution of the DFT. The time-halfbandwidth product is the product of the resolution halfbandwidth and the number of samples in the input signal, \( N \). The number of Slepian tapers whose Fourier transforms are well-concentrated in \([-W,W]\) (eigenvalues close to unity) is \( 2NW - 1 \).

\( nfft \) — Number of DFT points

\( \max(256,2^{\text{nextpow2}(\text{length}(x))}) \) (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

**method — Weights on individual tapered PSD estimates**

'adapt' (default) | 'eigen' | 'unity''

Weights on individual tapered PSD estimates, specified as one of 'adapt', 'eigen', or 'unity'. The default is Thomson’s adaptive frequency-dependent weights, 'adapt'. The calculation of these weights is detailed on pp. 368-370 in [1]. The 'eigen' method weights each tapered PSD estimate by the eigenvalue (frequency concentration) of the corresponding Slepian taper. The 'unity' method weights each tapered PSD estimate equally.

**e — DPSS (Slepian) sequences**

matrix

DPSS (Slepian) sequences, specified as a N-by-K matrix where N is the length of the input signal, $x$. The matrix $e$ is the output of dpss.

**v — Eigenvalues for DPSS (Slepian) sequences**

vector

Eigenvalues for DPSS (Slepian) sequences, specified as a column vector. The eigenvalues for the DPSS sequences indicate the proportion of the sequence energy concentrated in the resolution bandwidth, $[-W, W]$. The eigenvalues range lie in the interval (0,1) and generally the first $2NW-1$ eigenvalues are close to 1 and then decrease toward 0.

**dpss_params — Input arguments for dpss**

cell array

Input arguments for dpss, specified as a cell array. The first input argument to dpss is the length of the DPSS sequences and is omitted from dpss_params. The length of the DPSS sequences is obtained from the length of the input signal, $x$. 

1-1841
Example: \{3.5,5\}

**dropflag** — Flag indicating whether to drop or keep the last DPSS sequence

*true* (default) | *false*

Flag indicating whether to drop or keep the last DPSS sequence, specified as a logical. The default is *true* and `pmtm` drops the last taper. In a multitaper estimate, the first \(2NW - 1\) DPSS sequences have eigenvalues close to unity. If you use less than \(2NW - 1\) sequences, it is likely that all the tapers have eigenvalues close to 1 and you can specify `dropflag` as *false* to keep the last taper.

**freqrange** — Frequency range for PSD estimate

*'onesided'* | *'twosided'* | *'centered'*

Frequency range for the PSD estimate, specified as a one of *'onesided'*, *'twosided'*, or *'centered'*'. The default is *'onesided'* for real-valued signals and *'twosided'* for complex-valued signals. The frequency ranges corresponding to each option are

- *'onesided'* — returns the one-sided PSD estimate of a real-valued input signal, \(x\). If \(nfft\) is even, \(pxx\) has length \(nfft/2 + 1\) and is computed over the interval \([0,\pi]\) rad/sample. If \(nfft\) is odd, the length of \(pxx\) is \((nfft + 1)/2\) and the interval is \([0,\pi]\) rad/sample. When \(fs\) is optionally specified, the corresponding intervals are \([0,fs/2]\) cycles/unit time and \([0,fs/2]\) cycles/unit time for even and odd length \(nfft\) respectively.

- *'twosided'* — returns the two-sided PSD estimate for either the real-valued or complex-valued input, \(x\). In this case, \(pxx\) has length \(nfft\) and is computed over the interval \([0,2\pi]\) rad/sample. When \(fs\) is optionally specified, the interval is \([0,fs]\) cycles/unit time.

- *'centered'* — returns the centered two-sided PSD estimate for either the real-valued or complex-valued input, \(x\). In this case, \(pxx\) has length \(nfft\) and is computed over the interval \((-\pi,\pi]\) rad/sample for even length \(nfft\) and \((-\pi,\pi]\) rad/sample for odd length \(nfft\). When \(fs\) is optionally specified, the corresponding intervals are \((-fs/2, fs/2]\) cycles/unit time and \((-fs/2, fs/2]\) cycles/unit time for even and odd length \(nfft\) respectively.

**probability** — Confidence interval for PSD estimate

0.95 (default) | scalar in the range \((0,1)\)

Coverage probability for the true PSD, specified as a scalar in the range \((0,1)\). The output, \(pxxc\), contains the lower and upper bounds of the probability \(\times 100\%\) interval estimate for the true PSD.
Output Arguments

**pxx — PSD estimate**

vector | matrix

PSD estimate, returned as a real-valued, nonnegative column vector or matrix. Each column of pxx is the PSD estimate of the corresponding column of x. The units of the PSD estimate are in squared magnitude units of the time series data per unit frequency. For example, if the input data is in volts, the PSD estimate is in units of squared volts per unit frequency. For a time series in volts, if you assume a resistance of 1 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 $pxc(m, 2*n)$ 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**

**Discrete Prolate Spheroidal (Slepian) Sequences**

The derivation of the Slepian sequences proceeds from the discrete-time — continuous frequency concentration problem. For all $\ell^2$ sequences index-limited to 0,1,...,$N - 1$, the problem seeks the sequence having the maximal concentration of its energy in a frequency band $[-W, W]$ with $|W| < 1/2\Delta t$.

This amounts to finding the eigenvalues and corresponding eigenvectors of an $N$-by-$N$ self-adjoint positive semi-definite operator. Therefore, the eigenvalues are real and nonnegative and eigenvectors corresponding to distinct eigenvalues are mutually orthogonal. In this particular problem, the eigenvalues are bounded by 1 and the eigenvalue is the measure of the sequence’s energy concentration in the frequency interval $[-W, W]$.

The eigenvalue problem is given by

$$\sum_{n = 0}^{N - 1} \frac{\sin(2\pi W(n - m))}{n(n - m)} g_n = \lambda_k(N, W) g_m \quad m = 0, 1, 2, ..., N - 1$$

The 0th-order DPSS sequence, $g_0$ is the eigenvector corresponding to the largest eigenvalue. The 1-st order DPSS sequence, $g_1$ is the eigenvector corresponding to the next largest eigenvalue and is orthogonal to the 0-th order sequence. The 2nd-order DPSS sequence, $g_2$, is the eigenvector corresponding to the third largest eigenvalue and is orthogonal to the 0-th order and 1-st order DPSS sequences. Because the operator is $N$-by-$N$, there are $N$ eigenvectors. However, it can be shown that for a given sequence length $N$ and a specified bandwidth $[-W, W]$, there are approximately $2NW - 1$ DPSS sequences with eigenvalues very close to unity.

**Multitaper Spectral Estimation**

The periodogram is not a consistent estimator of the true power spectral density of a wide-sense stationary process. To produce a consistent estimate of the PSD, the
multitaper method averages modified periodograms obtained using a family of mutually orthogonal tapers (windows). In addition to mutual orthogonality, the tapers also have optimal time-frequency concentration properties. Both the orthogonality and time-frequency concentration of the tapers is critical to the success of the multitaper technique. See “Discrete Prolate Spheroidal (Slepian) Sequences” on page 1-1844 for a brief description of the Slepian sequences used in Thomson’s multitaper method.

The multitaper method uses $K$ modified periodograms with each one obtained using a different Slepian sequence as the window. Let

$$S_k(f) = \Delta t \left| \sum_{n=0}^{N-1} g_{k,n} x_n e^{-i2\pi fn\Delta t} \right|^2$$

denote the modified periodogram obtained with the $k$-th Slepian sequence, $g_{k,n}$.

In the simplest form, the multitaper method simply averages the $K$ modified periodograms to produce the multitaper PSD estimate.

$$S^{(MT)}(f) = \frac{1}{K} \sum_{k=0}^{K-1} S_k(f)$$

Note the difference between the multitaper PSD estimate and Welch’s method. Both methods reduce the variability in the periodogram by averaging over approximately uncorrelated estimates of the PSD. However, the two approaches differ in how they produce these uncorrelated PSD estimates. The multitaper method uses the entire signal in each modified periodogram. The orthogonality of the Slepian tapers decorrelates the different modified periodograms. Welch’s overlapped segment averaging approach uses segments of the signal in each modified periodogram and the segmenting decorrelates the different modified periodograms.

The preceding equation corresponds to the 'unity' option in `pmtm`. However, as explained in “Discrete Prolate Spheroidal (Slepian) Sequences” on page 1-1844, the Slepian sequences do not possess equal energy concentration in the frequency band of interest. The higher the order of the Slepian sequence, the less concentrated the sequence energy is in the band $[-W,W]$ with the concentration given by the eigenvalue. Consequently, it can be beneficial to use the eigenvalues to weight the $K$ modified periodograms prior to averaging. This corresponds to the 'eigen' option in `pmtm`.

Using the sequence eigenvalues to produce a weighted average of modified periodograms accounts for the frequency concentration properties of the Slepian sequences. However, it does not account for the interaction between the power spectral density of the random
process and the frequency concentration of the Slepian sequences. Specifically, frequency regions where the random process has little power are less reliably estimated in the modified periodograms using higher order Slepian sequences. This argues for an frequency-dependent adaptive process, which accounts not only for the frequency concentration of the Slepian sequence, but also for the power distribution in the time series. This adaptive weighting corresponds to the 'adapt' option in `pmtm` and is the default for computing the multitaper estimate.

**References**


**See Also**

dpss | periodogram | pWelch

*Introduced before R2006a*
psecurity

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(___,frequrange)
[___,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.
[\text{S,fo}] = \text{pms}\text{usic}(x,p,nfft,fs) returns the pseudospectrum computed at the frequencies specified in vector \text{fo} (in Hz). Supply the sample rate \text{fs} in Hz.

[\text{S,fo}] = \text{pms}\text{usic}(x,p,fi,fs) returns the pseudospectrum computed at the frequencies specified in the vector \text{fi}. The vector \text{fi} must have two or more elements, because otherwise the function interprets it as \text{nfft}.

[\text{S,fo}] = \text{pms}\text{usic}(x,p,nfft,fs,n\text{win},noverlap) returns the pseudospectrum \text{S} by segmenting the input data \text{x} using the window \text{nwin} and overlap length \text{noverlap}.

[\_\_] = \text{pms}\text{usic}(\_\_\_,\text{freqrange}) specifies the range of frequency values to include in \text{fo} or \text{wo}.

[\_\_,\text{v},\text{e}] = \text{pms}\text{usic}(\_\_) returns the matrix \text{v} of noise eigenvectors along with the associated eigenvalues in the vector \text{e}.

\text{pms}\text{usic}(\_\_) with no output arguments plots the pseudospectrum in the current figure window.

**Examples**

**pms\text{usic with no Sampling Specified}**

This example analyzes a signal vector, \text{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.

\[
\begin{align*}
n &= 0:199; \\
x &= \cos(0.257*pi*n) + \sin(0.2*pi*n) + 0.01*randn(size(n)); \\
pms\text{usic}(x,4) & \quad \text{% Set } p \text{ to } 4 \text{ because there are two real inputs}
\end{align*}
\]
Specifying Sampling Frequency and Subspace Dimensions

This example analyzes the same signal vector, \( x \), with an eigenvalue cutoff of 10% above the minimum. Setting \( p(1) = \text{Inf} \) forces the signal/noise subspace decision to be based on the threshold parameter, \( p(2) \). Specify the eigenvectors of length 7 using the \( \text{nwin} \) argument, and set the sampling frequency, \( \text{fs} \), to 8 kHz:

```matlab
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
```
Entering a Correlation Matrix

Supply a positive definite correlation matrix, R, for estimating the spectral density. Use the default 256 samples.

```matlab
R = toeplitz(cos(0.1*pi*(0:6))) + 0.1*eye(7);
pmusic(R,4,'corr')
```
Entering a Signal Data Matrix Generated from \texttt{corrmtx}

Enter a signal data matrix, \(X_m\), generated from data using \texttt{corrmtx}.

\begin{verbatim}
n = 0:699;
x = \cos(0.257\pi(n)) + 0.1\times\text{randn(size(n))};
Xm = \text{corrmtx}(x,7,'modified');
\text{pmusic}(Xm,2)
\end{verbatim}
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.

```matlab
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 \( \lambda_{\text{min}} \), the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold \( \lambda_{\text{min}} \cdot p(2) \) 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 double 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. 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 you 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)

<table>
<thead>
<tr>
<th>Input Data Type</th>
<th>Length of S and w0</th>
<th>Range of the Corresponding Normalized Frequencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>129</td>
<td>[0, π]</td>
</tr>
<tr>
<td>Complex</td>
<td>256</td>
<td>[0, 2π)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Input Data Type</th>
<th>nfft Even or Odd</th>
<th>Length of S and w</th>
<th>Range of w</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>Even</td>
<td>(nfft/2 )+ 1</td>
<td>[0, π]</td>
</tr>
<tr>
<td>Real</td>
<td>Odd</td>
<td>(nfft + 1)/2</td>
<td>[0, π]</td>
</tr>
<tr>
<td>Complex</td>
<td>Even or odd</td>
<td>nfft</td>
<td>[0, 2π]</td>
</tr>
</tbody>
</table>

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 \( f_0 \) if \( nfft \) and \( fs \) are specified.

**S and Frequency Vector Characteristics with fs Specified**

<table>
<thead>
<tr>
<th>Input Data Type</th>
<th>nfft Even/Odd</th>
<th>Range of f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>Even</td>
<td>([0, fs/2])</td>
</tr>
<tr>
<td>Real</td>
<td>Odd</td>
<td>([0, fs/2))</td>
</tr>
<tr>
<td>Complex</td>
<td>Even or odd</td>
<td>([0, fs))</td>
</tr>
</tbody>
</table>

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 \( nwin \)**

<table>
<thead>
<tr>
<th>form of x</th>
<th>Form of nwin</th>
<th>Windowed Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data vector</td>
<td>Scalar</td>
<td>Length is ( nwin ).</td>
</tr>
<tr>
<td>Data vector</td>
<td>Vector of coefficients</td>
<td>( \text{Length is length}(nwin) ).</td>
</tr>
<tr>
<td>Data matrix</td>
<td>Scalar</td>
<td>Data is not windowed.</td>
</tr>
<tr>
<td>Data matrix</td>
<td>Vector of coefficients</td>
<td>( \text{length}(nwin) ) must be the same as the column length of ( x ), and ( noverlap ) is not used.</td>
</tr>
</tbody>
</table>

See the Eigenvector Length Depending on Input Data and Syntax for related information on this syntax.

**\( v \) — Noise eigenvector**

Noise eigenvectors, returned as a matrix. The columns of \( v \) span the noise subspace of dimension \( \text{size}(v,2) \). The dimension of the signal subspace is \( \text{size}(v,1)-\text{size}(v,2) \).

**\( e \) — Estimated eigenvalues**

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 $\lambda_j$ of the signal's correlation matrix. The smallest of these eigenvalues is used in conjunction with the threshold parameter $p(2)$ to affect the dimension of the noise subspace in some cases.

The length $n$ of the eigenvectors computed by `pmusic` is the sum of the dimensions of the signal and noise subspaces. This eigenvector length depends on your input (signal data or correlation matrix) and the syntax you use.

The following table summarizes the dependency of the eigenvector length on the input argument.

**Eigenvector Length Depending on Input Data and Syntax**

<table>
<thead>
<tr>
<th>Form of Input Data x</th>
<th>Comments on the Syntax</th>
<th>Length n of Eigenvectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row or column vector</td>
<td>$\text{nwin}$ is specified as a scalar integer.</td>
<td>$nwin$</td>
</tr>
<tr>
<td>Row or column vector</td>
<td>$\text{nwin}$ is specified as a vector.</td>
<td>$\text{length(nwin)}$</td>
</tr>
<tr>
<td>Row or column vector</td>
<td>$\text{nwin}$ is not specified.</td>
<td>$2 \times p(1)$</td>
</tr>
<tr>
<td>$l$-by-$m$ matrix</td>
<td>If $\text{nwin}$ is specified as a scalar, it is not used. If $\text{nwin}$ is specified as a vector, $\text{length(nwin)}$ must equal $m$.</td>
<td>$m$</td>
</tr>
<tr>
<td>$m$-by-$m$ nonnegative definite matrix</td>
<td><code>'corr'</code> is specified and $\text{nwin}$ is not used.</td>
<td>$m$</td>
</tr>
</tbody>
</table>

You should specify $\text{nwin} > p(1)$ or $\text{length(nwin)} > p(1)$ 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 signal's correlation matrix are estimated if you do not supply the correlation matrix.

The MUSIC pseudospectrum estimate is given by

\[ P_{\text{MUSIC}}(f) = \frac{1}{e^H(f)\left(\sum_{k=p+1}^{N} v_k v_k^H\right)e(f)} = \frac{1}{\sum_{k=p+1}^{N} \left|v_k^H e(f)\right|^2} \]

where \( N \) is the dimension of the eigenvectors and \( v_k \) is the \( k \)th eigenvector of the correlation matrix. The integer \( p \) is the dimension of the signal subspace, so the eigenvectors \( v_k \) used in the sum correspond to the smallest eigenvalues and also span the noise subspace. The vector \( e(f) \) consists of complex exponentials, so the inner product

\[ v_k^H e(f) \]

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


See Also
corrmxtpburgpeigperiodogrampmtmpronypwelchrooteigrootmusic

Introduced before R2006a
poctave

Generate octave spectrum

Syntax

\[ p = \text{poctave}(x,fs) \]
\[ p = \text{poctave}(xt) \]
\[ p = \text{poctave}(\_,\text{Name,Value}) \]
\[ p = \text{poctave}(\text{pxx,fs,}\text{f,}\text{Name,Value,'psd'}) \]
\[ [p,\text{cf}] = \text{poctave}(\_) \]
\[ \text{poctave}(\_) \]

Description

\( p = \text{poctave}(x,fs) \) 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 [1]. 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 \( p \).

\( p = \text{poctave}(xt) \) returns the octave spectrum of a signal stored in the MATLAB timetable \( xt \).

\( p = \text{poctave}(\_,\text{Name,Value}) \) specifies additional options for any of the previous syntaxes using name-value pair arguments.

\( p = \text{poctave}(\text{pxx,fs,}\text{f,}\text{Name,Value,'psd'}) \) performs octave smoothing by converting a power spectral density, \( \text{pxx} \), 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 \( \text{pxx} \).

\( [p,\text{cf}] = \text{poctave}(\_) \) also returns the center frequencies of the octave bands over which the octave spectrum is computed.
poctave( ___ ) with no output arguments plots the octave spectrum in the current figure.

**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.

```matlab
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 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.

```matlab
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.

```plaintext
poctave(pn,fs,opts{:},'Weighting','A')
hold off
legend('Pink noise','C-weighted','A-weighted','Location','SouthWest')
```
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.

```matlab
poctave(pxx,fs,f,'BandsPerOctave',6,'FilterOrder',8,'FrequencyLimits',[200 fs/2],'psd')
legend('White','Pink')
```
**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 independently for each column and returns the result in the corresponding column of `p`.

Example: \( \sin(2\pi \cdot (0:127)/16) + \text{randn}(1,128)/100 \) specifies a noisy sinusoid.

Example: \([2\ 1].*\sin(2\pi \cdot (0:127)')/[16\ 64]\) specifies a two-channel sinusoid.
Data Types: single | double

**fs — Sample rate**

positive scalar

Sample rate, specified as a positive scalar expressed in Hz. 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” (MATLAB).

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.

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 $2\pi$ 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 $2\pi$ Hz and the frequencies at which it is computed.
**Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as 

**Name1**, **Value1**, ..., **NameN**, **ValueN**.

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 the comma-separated pair consisting of 'BandsPerOctave' and 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 the comma-separated pair consisting of 'FilterOrder' and a positive even integer.

Data Types: single | double

**FrequencyLimits — Frequency band**

[3  fs/2] (default) | two-element vector

Frequency band, specified as the comma-separated pair consisting of 'FrequencyLimits' and a two-element vector. The lower value of the vector must be at least 3 Hz. The higher value of the vector must be smaller than or equal to the Nyquist frequency. For some combinations of octave-band center frequencies and widths, the higher value of the vector must be at least $\max(3,3*fs/48e3)$ to ensure filter stability.

Data Types: single | double

**Weighting — Frequency weighting**

'none' (default) | 'A' | 'C' | vector | matrix | 1-by-2 cell array | digitalFilter object

Frequency weighting, specified as the comma-separated pair consisting of 'Weighting' and one of the following:
• '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
Output Arguments

**p — Octave spectrum**

vector | matrix

Octave spectrum, returned as a vector or matrix.

**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.

References


See Also

pspectrum

Introduced in R2018a
**poly2ac**

Convert prediction filter polynomial to autocorrelation sequence

**Syntax**

```matlab
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.

```matlab
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
```

1-1875
Tips

You can apply this function to both real and complex polynomials.

References


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.

```matlab
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


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 = 5x1

0.3090
0.9801
0.0031
0.0081
-0.0082

r0 = 5.6032

**Limitations**

If \( \text{abs}(k(i)) == 1 \) for any \( i \), finding the reflection coefficients is an ill-conditioned problem. `poly2rc` returns some NaNs and provides a warning message in those cases.

**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.

\[
\text{stable} = \text{all}(\text{abs}(\text{poly2rc}(a))<1)
\]

**Algorithms**

`poly2rc` implements this recursive relationship:

\[
k(n) = a_n(n)
\]

\[
a_{n-1}(m) = \frac{a_n(m) - k(n)a_n(n-m)}{1 - k(n)^2}, \quad m = 1, 2, \ldots, n - 1
\]

This relationship is based on Levinson’s recursion [1]. To implement it, `poly2rc` loops through \( a \) in reverse order after discarding its first element. For each loop iteration \( i \), the function:

1. Sets \( k(i) \) equal to \( a(i) \)
2. Applies the second relationship above to elements 1 through \( i \) of the vector \( a \).

\[
a = (a - k(i) \times \text{fliplr}(a))/(1 - k(i)^2);
\]
References


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**

\[ b = \text{polyscale}(a,\alpha) \]

**Description**

\[ b = \text{polyscale}(a,\alpha) \]

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.

\[
\begin{align*}
pp &= [1 \ 0 \ 0 \ 0 \ 0 \ 0 \ -1]; \\
zplane(pp,1)
\end{align*}
\]
Scale the roots of \( p \) in and out of the unit circle. Plot the results.

```matlab
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.

$A_o = \text{lpc}(\text{mtlb}(1000:1100),12);$
$A_x = \text{polyscale}(A_o,0.85);$
Perform bandwidth expansion of the signal by scaling the roots of the autoregressive polynomial by 0.85. Plot the zeros, poles, and frequency responses of the models.

```matlab
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:

\[ A(z) = a(1) + a(2)z^{-1} + ... + a(m+1)z^{-m}. \]

Examples

Convert Linear-Phase Filter to Minimum-Phase

Use the window method to design a 25th-order FIR filter with normalized cutoff frequency 0.4π 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)
\texttt{h\_minphase = logical}
\texttt{0}

Use \texttt{polystab} to convert the linear-phase filter into a minimum-phase filter. Plot the phase responses of the filters.

\texttt{hmin = polystab(h)*norm(h)/norm(polystab(h));}

\texttt{hmin\_linphase = islinphase(hmin)}

\texttt{hmin\_linphase = logical}
\texttt{0}

\texttt{hmin\_minphase = isminphase(hmin)}

\texttt{hmin\_minphase = logical}
\texttt{1}

\texttt{hfvt = fvtool(h,1,hmin,1,'Analysis','phase');}
\texttt{legend(hfvt,'h','hmin')}
Verify that the two filters have identical magnitude responses.

```matlab
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:

```matlab
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**

\[ ydb = \text{pow2db}(y) \]

**Description**

\[ ydb = \text{pow2db}(y) \] 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 \( \text{pwelch} \). Express the estimate in decibels and plot it.

\[
\begin{align*}
n &= 0:1024-1; \\
x &= \cos(2\pi n/3) + \text{randn}(\text{size}(n)); \\
[pxx,w] &= \text{pwelch}(x, 'power'); \\
dB &= \text{pow2db}(pxx); \\
\text{plot}(w/pi,dB) \\
\text{xlabel}('\omega / \pi') \\
\text{ylabel}'Power (dB)'
\end{align*}
\]
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™.

See Also

db | db2mag | db2pow | mag2db

Introduced in R2007b
powerbw

Power bandwidth

Syntax

\[ \text{bw} = \text{powerbw}(x) \]
\[ \text{bw} = \text{powerbw}(x,fs) \]

\[ \text{bw} = \text{powerbw}(\text{pxx},f) \]
\[ \text{bw} = \text{powerbw}(\text{sxx},f,\text{rbw}) \]

\[ \text{bw} = \text{powerbw}(\_\_\_\_,\text{freqrange},r) \]

\[ [\text{bw},\text{flo},\text{fhi},\text{power}] = \text{powerbw}(\_\_\_) \]

\[ \text{powerbw}(\_\_\_) \]

Description

\( \text{bw} = \text{powerbw}(x) \) returns the 3-dB (half-power) bandwidth, \( \text{bw} \), of the input signal, \( x \).

\( \text{bw} = \text{powerbw}(x,fs) \) returns the 3-dB bandwidth in terms of the sample rate, \( fs \).

\( \text{bw} = \text{powerbw}(\text{pxx},f) \) returns the 3-dB bandwidth of the power spectral density (PSD) estimate, \( \text{pxx} \). The frequencies, \( f \), correspond to the estimates in \( \text{pxx} \).

\( \text{bw} = \text{powerbw}(\text{sxx},f,\text{rbw}) \) computes the 3-dB bandwidth of the power spectrum estimate, \( \text{sxx} \). The frequencies, \( f \), correspond to the estimates in \( \text{sxx} \). \( \text{rbw} \) is the resolution bandwidth used to integrate each power estimate.

\( \text{bw} = \text{powerbw}(\_\_\_\_,\text{freqrange},r) \) 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. \( \text{freqrange} \) must lie within the target band.
If you also specify $r$, the function computes the difference in frequency between the points where the spectrum drops below the reference level by $r$ dB or reaches an endpoint.

$$[bw, flo, fhi, power] = \text{powerbw}(\_\_\_\_\_\_)$$
also returns the lower and upper bounds of the power bandwidth and the power within those bounds.

\text{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.

```matlab
nSamp = 1024;
Fs = 1024e3;
SNR = 40;

% Time vector
% 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).

```
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.

\[
x_2 = 2*\text{chirp}(t,200\text{e}3,\text{nSamp}/\text{Fs},300\text{e}3);
\]
\[
x_2 = x_2 + \text{randn(size}(x_2))\text{*std}(x_2)/\text{db2mag}(\text{SNR});
\]

Concatenate the chirps to produce a two-channel signal. Estimate the 3-dB bandwidth of each channel.

\[
y = \text{powerbw}([x \ x_2],\text{Fs})
\]
\[
y = 1\times 2
\]
\[
10^4 \times
\]
Annotate the 3-dB bandwidths of the two channels on a plot of the PSDs.

```matlab
powerbw([x x2],Fs);
```

Add the two channels to form a new signal. Plot the PSD and annotate the 3-dB bandwidth.

```matlab
powerbw(x+x2,Fs)
```
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.

```matlab
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.

\[ x_2 = 2 \times \sin(2 \pi t \times 257.321 e3); \]
\[ x_2 = x_2 + \text{randn(size}(x_2)) \times \text{std}(x_2) / \text{db2mag}(\text{SNR}); \]

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.

\[ [P_{yy}, f] = \text{periodogram}([x \ x_2], \text{kaiser}(\text{nSamp}, 38), [], \text{Fs}); \]
\[ y = \text{powerbw}(P_{yy}, f) \]
\[ y = 1 \times 2 \]
\[ 10^3 \times \]
\[ 3.1753 \quad 3.3015 \]

Annotate the 3-dB bandwidths of the two channels on a plot of the PSDs.

```
powerbw(P_{yy}, f);
```
Add the two channels to form a new signal. Estimate the PSD and annotate the 3-dB bandwidth.

\[
[Pzz, f] = \text{periodogram}(x + x2, \text{kaiser}(nSamp, 38), [], Fs);
\]

\[
\text{powerbw}(Pzz, f);
\]
Bandwidth of Bandlimited Signals

Generate a signal whose PSD resembles the frequency response of an 88th-order bandpass FIR filter with normalized cutoff frequencies $0.25\pi$ rad/sample and $0.45\pi$ rad/sample.

```
d = fir1(88,[0.25 0.45]);
```

Compute the 3-dB occupied bandwidth of the signal. Specify as a reference level the average power in the band between $0.2\pi$ rad/sample and $0.6\pi$ rad/sample. Plot the PSD and annotate the bandwidth.
Output the bandwidth, its lower and upper bounds, and the band power. Specifying a sample rate of $2\pi$ is equivalent to leaving the rate unset.

```matlab
[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 \(0.5\pi\) rad/sample and \(0.8\pi\) rad/sample and an amplitude that is one-tenth that of the first channel.

\[
d = [d; \text{fir1}(88,[0.5 0.8])/10]'
\]

Compute the 6-dB bandwidth of the two-channel signal. Specify as a reference level the maximum power level of the spectrum.

\[
\text{powerbw}(d,[],[],6);
\]

Output the 6-dB bandwidth of each channel and the lower and upper bounds.

\[
[bw,flo,fhi] = \text{powerbw}(d,[],[],6);
bds = [bw;flo;fhi];
\]
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 $2\pi$ 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`

**freqrange — Frequency range**

two-element vector

Frequency range, specified as a two-element vector of real values. If you specify `freqrange`, then the reference level is the average power level in the reference band. If
you do not specify `freqrange`, then the reference level is the maximum power level of the spectrum.

Data Types: `single` | `double`

**r — Power level drop**

10 \log_{10} 2 (default) | positive real scalar

Power level drop, specified as a positive real scalar expressed in dB.

Data Types: `single` | `double`

### Output Arguments

**bw — Power bandwidth**

scalar | vector

Power bandwidth, returned as a scalar or vector.

- If you specify a sample rate, then `bw` has the same units as `fs`.
- If you do not specify a sample rate, then `bw` has units of rad/sample.

**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.
See Also
bandpower | obw | periodogram | plomb | pwelch

Introduced in R2015a
**prony**

Prony method for filter design

**Syntax**

\[
[\text{Num}, \text{Den}] = \text{prony}(\text{impulse}\_\text{resp}, \text{num}\_\text{ord}, \text{denom}\_\text{ord})
\]

**Description**

\[ [\text{Num}, \text{Den}] = \text{prony}(\text{impulse}\_\text{resp}, \text{num}\_\text{ord}, \text{denom}\_\text{ord}) \]

returns the numerator \( \text{Num} \) and denominator \( \text{Den} \) coefficients for a causal rational system function with impulse response \( \text{impulse}\_\text{resp} \). The system function has numerator order \( \text{num}\_\text{ord} \) and denominator order \( \text{denom}\_\text{ord} \). The lengths of \( \text{Num} \) and \( \text{Den} \) are \( \text{num}\_\text{ord}+1 \) and \( \text{denom}\_\text{ord}+1 \). If the length of \( \text{impulse}\_\text{resp} \) is less than the largest order (\( \text{num}\_\text{ord} \) or \( \text{denom}\_\text{ord} \)), \( \text{impulse}\_\text{resp} \) is padded with zeros. Enter 0 in \( \text{num}\_\text{ord} \) for an all-pole system function. For an all-zero system function, enter a 0 for \( \text{denom}\_\text{ord} \).

**Examples**

**Filter Responses via Prony's Method**

Fit a 4th-order IIR model to the impulse response of a lowpass filter. Plot the original and Prony-designed impulse responses.

\[
d = \text{designfilt}(\text{'lowpassiir'}, \text{'NumeratorOrder'}, 4, \text{'DenominatorOrder'}, 4, \text{...}
\]

\[
\text{'HalfPowerFrequency'}, 0.2, \text{'DesignMethod'}, \text{'butter'})
\]

\[
\text{impulse}\_\text{resp} = \text{filter}(d, [1 \text{ zeros}(1,31)]);
\]

\[
\text{denom}\_\text{order} = 4;
\]

\[
\text{num}\_\text{order} = 4;
\]

\[
[\text{Num}, \text{Den}] = \text{prony}(\text{impulse}\_\text{resp}, \text{num}\_\text{order}, \text{denom}\_\text{order});
\]

\[
\text{subplot}(2,1,1)
\]

\[
\text{stem}(\text{impz}(\text{Num}, \text{Den}, \text{length}(\text{impulse}\_\text{resp})))
\]
Fit a 10th-order FIR model to the impulse response of a highpass filter. Plot the original and Prony-designed frequency responses.

```matlab
d = designfilt('highpassfir', 'FilterOrder', 10, 'CutoffFrequency', .8);
impulse_respond = filter(d,[1 zeros(1,31)]);
num_order = 10;
denom_order = 0;
```
More About

System Function

The system function is the z-transform of the impulse response $h[n]$:
\[ H(z) = \sum_{n = -\infty}^{\infty} h[n] z^{-n} \]

A rational system function is a ratio of polynomials in \( z^{-1} \). By convention the numerator polynomial is \( B(z) \) and the denominator is \( A(z) \). The following equation describes a causal rational system function of numerator order \( q \) and denominator order \( p \):

\[
H(z) = \frac{\sum_{k=0}^{q} b[k] z^{-k}}{1 + \sum_{l=1}^{p} a[l] z^{-l}}
\]

where \( a[0] = 1 \).

References


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')
p = 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.
\[ p = \text{pspectrum}(\_,\text{type}) \] 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.

\[ p = \text{pspectrum}(\_,\text{Name},\text{Value}) \] specifies additional options using name-value pair arguments. Options include the frequency resolution bandwidth and the percent overlap between adjoining segments.

\[ [p,f] = \text{pspectrum}(\_) \] returns the frequencies corresponding to the spectral estimates contained in \( p \).

\[ [p,f,t] = \text{pspectrum}(\_,\text{'spectrogram'}) \] also returns a vector of time instants corresponding to the centers of the windowed segments used to compute short-time power spectrum estimates.

\[ [p,f,pwr] = \text{pspectrum}(\_,\text{'persistence'}) \] 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.

### Examples

#### Spectrum of Sinusoid

Compute the power spectrum of a noisy sinusoid. Specify a sinusoid frequency of 200 Hz. Sample the sinusoid at 1 kHz for 296 milliseconds. Embed the signal in white Gaussian noise of variance 0.1². Store the signal and its time information in a MATLAB® timetable.

\[
\begin{align*}
Fs &= 1000; \\
t &= (0:1/Fs:0.296)'; \\
x &= \cos(2\pi t \times 200) + 0.1 \times \text{randn(size(t))}; \\
xTable &= \text{timetable(seconds(t)},x);
\end{align*}
\]

Compute the spectrum of the signal. Express the spectrum in decibels and plot it.

\[
\begin{align*}
[pxx,f] &= \text{pspectrum}(xTable); \\
\text{plot}(f,\text{pow2db}(pxx)) \\
\text{grid on}
\end{align*}
\]
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 two signals, each sampled at 3 kHz for 1 second. The first 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. The second signal, also embedded in white noise, is a chirp with sinusoidally varying frequency content.

```matlab
fs = 3000;
t = 0:1/fs:1-1/fs;
x1 = chirp(t,300,t(end),1300,'quadratic',0,'convex') + ...
```

![pspectrum](image-url)
randn(size(t))/100;

x2 = exp(2j*pi*100*cos(2*pi*2*t)) + randn(size(t))/100;

Compute and plot the two-sided power spectrum of the first 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)
```

Compute the spectrogram of the second signal. For complex signals, the spectrogram is two-sided by default. Display the spectrogram as a waterfall plot.
Window Leakage and Tone Resolution

Generate a two-channel signal sampled at 100 Hz for 2 seconds.
The first channel consists of a 20 Hz tone and a 21 Hz tone. Both tones have unit amplitude.

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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
fs = 1e3; t = 0:1/fs:2; y = chirp(t,100,1,200,'quadratic'); [sp,fp,tp] = pspectrum(y,fs,'spectrogram');
```
Compute the reassigned spectrogram. Specify a frequency resolution of 10 Hz. Visualize the result using the `pspectrum` function with no output arguments.

```matlab
pspectrum(y,fs,'spectrogram','FrequencyResolution',10,'Reassign',true)
```
Recompute the spectrogram using a time resolution of 0.2 second.

```matlab
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.

```matlab
fs = 4e3;
t = 0:1/fs:0.5-1/fs;
ver = [697 770 852 941];
hor = [1209 1336 1477];
```

Fres = 12.8337 Hz, Tres = 200 ms
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” (MATLAB).

• 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: \( \cos(\pi/(4:2)*(0:159))' + \text{randn}(160,2) \) 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**

2\( \pi \) (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. See “Spectrum Computation” on page 1-1942 for more information.
- 'spectrogram' — Compute the spectrogram of the input. Use this option to analyze how the frequency content of a signal changes over time. See “Spectrogram Computation” on page 1-1945 for more information.
- '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. See “Persistence Spectrum Computation” on page 1-1948 for more information.

**Note** The 'spectrogram' and 'persistence' options do not support multichannel input.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Leakage',1,'Reassigned',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 0.5] (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, \texttt{pspectrum} computes the spectrum over the whole Nyquist range:

- If the specified frequency band contains a region that falls outside the Nyquist range, then \texttt{pspectrum} truncates the frequency band.
- If the specified frequency band lies completely outside of the Nyquist range, then \texttt{pspectrum} throws an error.

See “Spectrum Computation” on page 1-1942 for more information about the Nyquist range.

If \( x \) is nonuniformly sampled, then \texttt{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 \texttt{'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 \( 0.2\pi \) to \( 0.7\pi \) rad/sample.

\textbf{FrequencyResolution — Frequency resolution bandwidth}

real numeric scalar

Frequency resolution bandwidth, specified as the comma-separated pair consisting of \texttt{'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 \texttt{'TimeResolution'}. The default value of this argument depends on the size of the input data. See “Spectrogram Computation” on page 1-1945 for details.

Example: \( \pi/100 \) computes the spectrum of a signal with no time information with a frequency resolution of \( \pi/100 \) rad/sample.

\textbf{Leakage — Spectral leakage}

0.5 (default) | real numeric scalar between 0 and 1

Spectral leakage, specified as the comma-separated pair consisting of \texttt{'Leakage'} and a real numeric scalar between 0 and 1. \texttt{'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 $10 \log_{10}(p) \leq \text{MinThreshold}$ 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 1-1945 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 1-1945 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 multiples 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 \( x \).

• '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 \texttt{pspectrum} is a timetable, then \( t \) has the same format as the time values of the input timetable.
- If the input to \texttt{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 \texttt{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 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, $\beta$. `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 $\beta$, use the
'Leakage' name-value pair. If you set 'Leakage' to $\ell$, then $\ell$ and $\beta$ 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

$$RBW_{\text{theory}} = \frac{\text{ENBW}}{t_{\text{max}} - t_{\text{min}}}.$$  

- $t_{\text{max}} - t_{\text{min}}$, 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

$$RBW_{\text{performance}} = 4 \times \frac{f_{\text{span}}}{4096 - 1}.$$
where \( f_{\text{span}} \) is the width of the frequency band specified using 'FrequencyLimits'. If 'FrequencyLimits' is not specified, then \texttt{pspectrum} uses the sample rate as \( f_{\text{span}} \).

\( \text{RBW}_{\text{performance}} \) cannot be adjusted.

To compute the spectrum of a signal, the function chooses the larger of the two values, called the target resolution bandwidth:

\[
\text{RBW} = \max(\text{RBW}_{\text{theory}}, \text{RBW}_{\text{performance}}).
\]

- If the resolution bandwidth is \( \text{RBW}_{\text{theory}} \), then \texttt{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 and applies zero-padding when the time limits on the axes exceed the signal duration. See \texttt{periodogram} for more details.
- If the resolution bandwidth is \( \text{RBW}_{\text{performance}} \), then \texttt{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 \texttt{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 \( f_{\text{Nyquist}} \) 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, \texttt{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, \texttt{pspectrum} follows these steps:

\begin{enumerate}
\item 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.
\item Window each segment and compute its spectrum to get the short-time Fourier transform.
\item Use the segment spectra to construct the spectrogram:
\begin{itemize}
\item If called with output arguments, concatenate the spectra to form a matrix.
\item 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.
\end{itemize}
\end{enumerate}

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 \texttt{pspectrum} function, you can control the length of the segments and the amount of overlap between adjoining segments using the \texttt{'TimeResolution'} and \texttt{'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

\[
\left(1 - \frac{1}{2 \times \text{ENBW} - 1}\right) \times 100,
\]

where ENBW is the equivalent noise bandwidth of the spectral window. See \texttt{enbw} and “Spectrum Computation” on page 1-1942 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 $N$:

<table>
<thead>
<tr>
<th>Signal Length ($N$)</th>
<th>Divisor ($d$)</th>
<th>Segment Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 samples – 63 samples</td>
<td>2</td>
<td>1 sample – 32 samples</td>
</tr>
<tr>
<td>64 samples – 255 samples</td>
<td>8</td>
<td>8 samples – 32 samples</td>
</tr>
<tr>
<td>256 samples – 2047 samples</td>
<td>8</td>
<td>32 samples – 256 samples</td>
</tr>
<tr>
<td>2048 samples – 4095 samples</td>
<td>16</td>
<td>128 samples – 256 samples</td>
</tr>
<tr>
<td>4096 samples – 8191 samples</td>
<td>32</td>
<td>128 samples – 256 samples</td>
</tr>
<tr>
<td>8192 samples – 16383 samples</td>
<td>64</td>
<td>128 samples – 256 samples</td>
</tr>
<tr>
<td>16384 samples – $N$ samples</td>
<td>128</td>
<td>128 samples – $\lceil N / 128 \rceil$ samples</td>
</tr>
</tbody>
</table>

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 \([s_0 \ s_1 \ s_2 \ s_3 \ s_4 \ s_5 \ s_6]\). Because \([7/2] = [3.5] = 4\), 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.

<table>
<thead>
<tr>
<th>Number of Overlapping Samples</th>
<th>Resulting Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(s_0 \ s_1 \ s_2 \ s_3)</td>
</tr>
<tr>
<td></td>
<td>(s_4 \ s_5 \ s_6 )</td>
</tr>
<tr>
<td>1</td>
<td>(s_0 \ s_1 \ s_2 \ s_3)</td>
</tr>
<tr>
<td></td>
<td>(s_3 \ s_4 \ s_5 \ s_6)</td>
</tr>
<tr>
<td>2</td>
<td>(s_0 \ s_1 \ s_2 \ s_3)</td>
</tr>
<tr>
<td></td>
<td>(s_2 \ s_3 \ s_4 \ s_5)</td>
</tr>
<tr>
<td></td>
<td>(s_4 \ s_5 \ s_6 )</td>
</tr>
<tr>
<td>3</td>
<td>(s_0 \ s_1 \ s_2 \ s_3)</td>
</tr>
<tr>
<td></td>
<td>(s_1 \ s_2 \ s_3 \ s_4)</td>
</tr>
<tr>
<td></td>
<td>(s_2 \ s_3 \ s_4 \ s_5)</td>
</tr>
<tr>
<td></td>
<td>(s_3 \ s_4 \ s_5 \ s_6)</td>
</tr>
</tbody>
</table>

*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 \(\beta\) 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 1-1942, except that the lower limit of the resolution bandwidth is

\[
\text{RBW}_{\text{performance}} = 4 \times \frac{f_{\text{span}}}{1024 - 1}.
\]

**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 of a signal 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 1-1945 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.
References


See Also

Apps
Signal Analyzer

Functions
periodogram | p wel ch | spectrogram

Introduced in R2017b
**pulseperiod**

Period of bilevel pulse

**Syntax**

```matlab
P = pulseperiod(X)
P = pulseperiod(X,FS)
P = pulseperiod(X,T)
[P,INITCROSS] = pulseperiod(...)
[P,INITCROSS,FINALCROSS] = pulseperiod(...)
[P,INITCROSS,FINALCROSS,NEXTCROSS] = pulseperiod(...)
[P,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = pulseperiod(...)
[P,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = pulseperiod(...,
Name,Value)
pulseperiod(...)
```

**Description**

`P = pulseperiod(X)` returns a vector, P, containing the difference between the mid-reference level instants of the initial transition of each positive-polarity pulse and the next positive-going transition in the bilevel waveform, X. If `pulseperiod` does not find two positive-polarity transitions, P is empty. To determine the transitions for each pulse, `pulseperiod` estimates the state levels of the input waveform by a histogram method and identifies all regions which cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-1957. Because `pulseperiod` uses interpolation to determine the mid-reference level instants, P may contain values that do not correspond to sampling instants of the bilevel waveform, X.

`P = pulseperiod(X,FS)` specifies the sample rate in hertz as a positive scalar. The first sample instant in X corresponds to t=0. Because `pulseperiod` uses interpolation to determine the mid-reference level instants, P may contain values that do not correspond to sampling instants of the bilevel waveform, X.
P = pulseperiod(X,T) specifies the sampling instants in a vector equal in length to X. Because pulseperiod uses interpolation to determine the mid-reference level instants, P may contain values that do not correspond to sampling instants of the bilevel waveform, X.

[P,INITCROSS] = pulseperiod(...) returns the mid-reference level instants of the first transition of each pulse.

[P,INITCROSS,FINALCROSS] = pulseperiod(...) returns the mid-reference level instants of the final transition of each pulse.

[P,INITCROSS,FINALCROSS,NEXTCROSS] = pulseperiod(...) returns the mid-reference level instants of next detected transition after each pulse.

[P,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = pulseperiod(...) returns the mid-reference level, MIDLEV.

[P,INITCROSS,FINALCROSS,NEXTCROSS,MIDLEV] = pulseperiod(..., Name,Value) returns the pulse periods with additional options specified by one or more Name,Value pair arguments.

pulseperiod(...) plots the signal and darkens every other identified pulse. It marks the location of the mid crossings, and their associated reference level. The state levels and their associated lower and upper boundaries (adjustable by the Name,Value pair with name 'Tolerance') are also plotted.

**Input Arguments**

X
Bilevel waveform. If the waveform, X, does not contain at least two transitions, pulseperiod outputs an empty matrix.

FS
Sample rate in hertz.

T
Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.
Name-Value Pair Arguments

MidPercentReferenceLevel
Mid-reference level as a percentage of the waveform amplitude.

**Default:** 50

Polarity
Pulse polarity. Specify the polarity as 'positive' or 'negative'. If you specify 'positive', pulseperiod looks for pulses whose initial transition is positive-going (positive polarity). If you specify 'negative', pulseperiod looks for pulses whose initial transition is negative-going (negative polarity).

**Default:** 'positive'

StateLevels
Low- and high-state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low and high-state levels, pulseperiod estimates the state levels from the input waveform using the histogram method.

Tolerance
Tolerance levels (lower and upper state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-1957.

**Default:** 2

Output Arguments

P
Pulse period in seconds. The pulse period is defined as the time between the mid-reference level instants of two consecutive transitions.

INITCROSS
Mid-reference level instant of initial transition.
**FINALCROSS**

Mid-reference level instant of final transition.

**NEXTCROSS**

Mid-reference level instant of the first pulse transition after the final transition of the preceding pulse.

**MIDLEV**

Waveform value that corresponds to the mid-reference level.

**Examples**

**Pulse Period of Bilevel Waveform**

Compute the pulse period of a bilevel waveform with two positive-polarity transitions. The sample rate is 4 MHz.

```matlab
load('pulseex.mat','x','t')
p = pulseperiod(x,t)
p = 5.0030e-06
```

Annotate the pulse period on a plot of the waveform.

```matlab
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.

```matlab
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
```
More About

Mid-Reference Level

The mid-reference level in a bilevel waveform with low-state level, $S_1$, and high-state level, $S_2$, is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

Mid-Reference Level Instant

Let $y_{50\%}$ denote the mid-reference level.

Let $t_{50\%}$ and $t_{50\%^+}$ denote the two consecutive sampling instants corresponding to the waveform values nearest in value to $y_{50\%}$.

Let $y_{50\%}$ and $y_{50\%^+}$ denote the waveform values at $t_{50\%}$ and $t_{50\%^+}$.

The mid-reference level instant is

$$t_{50\%} = t_{50\%} + \left(\frac{t_{50\%^+} - t_{50\%^-}}{y_{50\%^+} - y_{50\%^-}}\right)(y_{50\%^+} - y_{50\%^-})$$

Pulse Polarity

If the initial transition of a pulse is positive-going, the pulse has positive polarity. The following figure shows a positive-polarity pulse.
Equivalently, a positive-polarity (positive-going) pulse has a terminating state more positive than the originating state.

If the initial transition of a pulse is negative-going, the pulse has negative polarity. The following figure shows a negative-polarity pulse.

Equivalently, a negative-polarity (negative-going) pulse has a originating state more positive than the terminating state.

**State-Level Tolerances**

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the $\alpha\%$ region for the low state is defined as

$$S_l \pm \frac{\alpha}{100}(S_h - S_l),$$

where $S_1$ is the low-state level and $S_2$ is the high-state level. Replace the first term in the equation with $S_2$ to obtain the $\alpha\%$ tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.
References


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, S, between the mid-reference level instants of the final negative-going transitions of every positive-polarity pulse and the next positive-going transition. X is a bilevel waveform. To determine the transitions that compose each pulse, pulsesep estimates the state levels of X by a histogram method. pulsesep identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-1966. Because pulsesep uses interpolation to determine the mid-reference level instants, S may contain values that do not correspond to sampling instants of the bilevel waveform, X.

S = pulsesep(X,FS) specifies the sample rate, FS, in Hz as a positive scalar. The first time instant corresponds to t=0. Because pulsesep uses interpolation to determine the mid-reference level instants, S may contain values that do not correspond to sampling instants of the bilevel waveform, X.

S = pulsesep(X,T) specifies the sampling instants, T, in a vector equal in length to X. Because pulsesep uses interpolation to determine the mid-reference level instants, S
may contain values that do not correspond to sampling instants of the bilevel waveform, \(X\).

\([S, \text{INITCROSS}] = \text{pulsesep}(\ldots)\) returns the mid-reference level instants, \(\text{INITCROSS}\), of the first positive-polarity transitions.

\([S, \text{INITCROSS}, \text{FINALCROSS}] = \text{pulsesep}(\ldots)\) returns the mid-reference level instants, \(\text{FINALCROSS}\), of the final transition of each pulse.

\([S, \text{INITCROSS}, \text{FINALCROSS}, \text{NEXTCROSS}] = \text{pulsesep}(\ldots)\) returns the mid-reference level instants, \(\text{NEXTCROSS}\), of the next detected transition after each pulse.

\([S, \text{INITCROSS}, \text{FINALCROSS}, \text{NEXTCROSS}, \text{MIDLEV}] = \text{pulsesep}(\ldots)\) returns the mid-reference level, \(\text{MIDLEV}\).

\([S, \text{INITCROSS}, \text{FINALCROSS}, \text{NEXTCROSS}, \text{MIDLEV}] = \text{pulsesep}(\ldots, \text{Name,Value})\) returns the pulse separations with additional options specified by one or more \text{Name,Value} pair arguments.

\(\text{pulsesep}(\ldots)\) plots the signal and darkens the regions between each pulse where pulse separation is computed. It marks the location of the mid crossings, and their associated reference level. The state levels and their associated lower and upper boundaries (adjustable by the \text{Name,Value} pair with name 'Tolerance') are also plotted.

**Input Arguments**

\(X\)

Bilevel waveform. If the waveform, \(X\), does not contain at least two transitions, \text{pulsesep} outputs an empty matrix.

\(FS\)

Sample rate in hertz.

\(T\)

Vector of sample instants. The length of \(T\) must equal the length of the bilevel waveform, \(X\).
Name-Value Pair Arguments

MidPercentReferenceLevel
Mid-reference level as a percentage of the waveform amplitude.

Default: 50

Polarity
Pulse polarity. Specify the polarity as 'positive' or 'negative'. If you specify 'positive', pulsesep looks for pulses with positive-going (positive polarity) initial transitions. If you specify 'negative', pulsesep looks for pulses with negative-going (negative polarity) initial transitions. See “Pulse Polarity” on page 1-1965.

Default: 'positive'

StateLevels
Low- and high-state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low- and high-state levels, pulsesep estimates the state levels from the input waveform using the histogram method.

Tolerance
Tolerance levels (lower- and upper-state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-1966.

Default: 2

Output Arguments

S
Pulse separations in seconds. The pulse separation is defined as the time between the mid-reference level instants of the final transition of one pulse and the initial transition of the next pulse. See “Pulse Separation” on page 1-1967.
**INITCROSS**
Mid-reference level instants of initial transition.

**FINALLCROSS**
Mid-reference level instants of final transition.

**NEXTCROSS**
Mid-reference level instants of the initial transition after the final transition of the preceding pulse.

**MIDLEV**
Waveform value that corresponds to the mid-reference level.

**Examples**

**Pulse Separation in Bilevel Waveform**
Compute the pulse separation in a bilevel waveform with two positive-polarity transitions. The 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.

```matlab
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.

\[ \text{pulsesep}(x,t) \]

\[ \text{ans} = 3.5014\times 10^{-6} \]
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. The following figure shows a positive-polarity pulse.
Equivalently, a positive-polarity (positive-going) pulse has a terminating state more positive than the originating state.

If the pulse has an initial negative-going transition, the pulse has negative polarity. The following figure shows a negative-polarity pulse.

Equivalently, a negative-polarity (negative-going) pulse has an originating state more positive than the terminating state.

**State-Level Tolerances**

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the $\alpha\%$ region for the low state is defined as

\[ S_1 \pm \frac{\alpha}{100} (S_2 - S_1), \]

where $S_1$ is the low-state level and $S_2$ is the high-state level. Replace the first term in the equation with $S_2$ to obtain the $\alpha\%$ tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.
Pulse Separation

Pulse separation is the time difference between the mid-reference level instant of the final transition of one pulse and the mid-reference level instant of the initial transition of the next pulse. The following figure illustrates pulse separation.
References


See Also
dutycycle | pulseperiod | pulsedwidth | statelevels

Introduced in R2012a
pulsewidth

Bilevel waveform pulse width

Syntax

\[ W = \text{pulsewidth}(X) \]
\[ W = \text{pulsewidth}(X,FS) \]
\[ W = \text{pulsewidth}(X,T) \]
\[ [W,\text{INITCROSS}] = \text{pulsewidth}(...) \]
\[ [W,\text{INITCROSS,FINALCROSS}] = \text{pulsewidth}(...) \]
\[ [W,\text{INITCROSS,FINALCROSS,MIDLEV}] = \text{pulsewidth}(...) \]
\[ W = \text{pulsewidth}(...,\text{Name,Value}) \]
\[ \text{pulsewidth}(...) \]

Description

\( W = \text{pulsewidth}(X) \) returns a vector, \( W \), containing the time differences between the mid-reference level instants of the initial and final transitions of each positive-polarity pulse in the bilevel waveform, \( X \). To determine the transitions, \( \text{pulsewidth} \) estimates the low- and high-state levels of \( X \) by a histogram method. \( \text{pulsewidth} \) identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-1977. Because \( \text{pulsewidth} \) uses interpolation to determine the mid-reference level instants, \( W \) may contain values that do not correspond to sampling instants of the bilevel waveform, \( X \).

\( W = \text{pulsewidth}(X,FS) \) specifies the sample rate, \( FS \), in hertz as a positive scalar. The first sample in the waveform corresponds to \( t=0 \). Because \( \text{pulsewidth} \) uses interpolation to determine the mid-reference level instants, \( W \) may contain values that do not correspond to sampling instants of the bilevel waveform, \( X \).

\( W = \text{pulsewidth}(X,T) \) specifies the sample instants, \( T \), as a vector with the same number of elements as \( X \). Because \( \text{pulsewidth} \) uses interpolation to determine the mid-reference level instants, \( W \) may contain values that do not correspond to sampling instants of the bilevel waveform, \( X \).
[W, INITCROSS] = pulsewidth(...) returns a column vector, INITCROSS, whose elements correspond to the mid-reference level instants of the initial transition of each pulse.

[W, INITCROSS, FINALCROSS] = pulsewidth(...) returns a column vector, FINALCROSS, whose elements correspond to the mid-reference level instants of the final transition of each pulse.

[W, INITCROSS, FINALCROSS, MIDLEV] = pulsewidth(...) returns the waveform value, MIDLEV, which corresponds to the mid-reference level.

W = pulsewidth(...,Name,Value) returns the pulse widths with additional options specified by one or more Name,Value pair arguments.

pulsewidth(...) plots the signal and darkens the regions of each pulse where pulse width is computed. It marks the location of the mid crossings, and their associated reference level. The state levels and their associated lower and upper boundaries (adjustable by the Name,Value pair with name 'Tolerance') are also plotted.

Input Arguments
X
Bilevel waveform. X is a real-valued row or column vector.

FS
Sample rate in hertz.

T
Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

Name-Value Pair Arguments
MidPercentReferenceLevel
Mid-reference level as percentage of the waveform amplitude. See “Mid-Reference Level” on page 1-1976.
**Polarity**

Pulse polarity. Specify the polarity as 'positive' or 'negative'. If you specify 'positive', pulsewidth looks for pulses with positive-going (positive polarity) initial transitions. If you specify 'negative', pulsewidth looks for pulses with negative-going (negative polarity) initial transitions. See “Pulse Polarity” on page 1-1976.

**Default: 'positive'**

**StateLevels**

Low- and high-state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low- and high-state levels, pulsewidth estimates the state levels from the input waveform using the histogram method.

**Tolerance**

Tolerance levels (lower and upper state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-1977.

**Default: 2**

**Output Arguments**

**W**

Pulse widths in seconds. The pulse width is the time difference between the initial and final transitions of a pulse. The times of the initial and final transitions are referred to as transition occurrence instants in [1] on page 1-1978.

**INITCROSS**

Mid-reference level instants of the initial transition

**FINALCROSS**

Mid-reference level instants of the final transition
MIDLEV

Waveform value corresponding to the mid-reference level

Examples

Pulse Width of Bilevel Waveform

Compute the pulse width of a bilevel waveform sampled at 4 MHz.

```matlab
load('pulseex.mat','x','t')
w = pulsewidth(x,t)
w = 1.5016e-06
```

Plot the waveform and annotate the pulse width.

```matlab
pulsewidth(x,t);
```
First and Second Transition Times for Bilevel Waveform

Compute the initial and final transition occurrences for a bilevel waveform sampled at 4 MHz.

```matlab
load('pulseex.mat', 'x', 't');
fs = 4e6;
[w,initcross,finalcross] = pulsewidth(x,fs);
```

Plot the result, annotated with the transition occurrences.
Specify State Levels for Bilevel Waveform

Specify the state levels for the bilevel waveform instead of estimating the levels from the data. Use the 'StateLevels' name-value pair to enter the low-state level as 0 and the high-state level as 5.
load('pulseex.mat', 'x', 't')
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];
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 a positive-going initial transition, the pulse has positive polarity. The following figure shows a positive-polarity pulse.
Equivalently, a positive-polarity (positive-going) pulse has a terminating state more positive than the originating state.

If the pulse has a negative-going initial transition, the pulse has negative polarity. The following figure shows a negative-polarity pulse.

Equivalently, a negative-polarity (negative-going) pulse has a originating state more positive than the terminating state.

**State-Level Tolerances**

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the $\alpha\%$ region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1),$$

where $S_1$ is the low-state level and $S_2$ is the high-state level. Replace the first term in the equation with $S_2$ to obtain the $\alpha\%$ tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.
References


See Also
dutycycle | pulseperiod | pulsesep | statelevels

Introduced in R2012a
pulstran

Pulse train

Syntax

\[ y = \text{pulstran}(t,d,\text{func}) \]
\[ y = \text{pulstran}(t,d,\text{func},fs) \]
\[ y = \text{pulstran}(t,d,p) \]
\[ y = \text{pulstran}(\_\_\_\_,\text{intfunc}) \]

Description

\[ y = \text{pulstran}(t,d,\text{func}) \] generates a pulse train based on samples of a continuous function, \( \text{func} \).

\[ y = \text{pulstran}(t,d,\text{func},fs) \] uses a sample rate of \( fs \).

\[ y = \text{pulstran}(t,d,p) \] generates a pulse train that is the sum of multiple delayed interpolations of the prototype pulse in vector \( p \).

\[ y = \text{pulstran}(\_\_\_\_,\text{intfunc}) \] specifies alternative interpolation methods. See \text{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.

\[
\begin{align*}
t &= 0:1/1e3:60; \\
d &= [0:2:60;\sin(2*\pi*0.05*(0:2:60))]';
\end{align*}
\]
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.

```matlab
fs = 1e3;
t = 0:1/fs: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.

```matlab
fs = 1e7;
tc = gauspuls('cutoff',10e3,0.5,[],-40);
t = -tc:1/fs:tc;
x = gauspuls(t,10e3,0.5);
plot(t,x)
```
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.

```matlab
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])
```

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.

```matlab
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')
```

![Graph showing waveforms generated by `pulstran` function]
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.

```matlab
y = pulstran(t,dd,fnx,30);

plot(t,y)
xlabel('Time (s)')
ylabel('Waveform')
```

![Graph of waveform over time](image)
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.

```matlab
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.

```matlab
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');
```

1-1989
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:

```matlab
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Continuity</th>
<th>Comments</th>
</tr>
</thead>
</table>
| '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' |
| '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' |
Method Description Continuity Comments

'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.

\[
pxx = \text{pwelch}(x, \text{window})
\]

uses the input vector or integer, \( \text{window} \), to divide the signal into segments. If \( \text{window} \) is a vector, \text{pwelch} \) divides the signal into segments equal in length to the length of \( \text{window} \). The modified periodograms are computed using the signal segments multiplied by the vector, \( \text{window} \). If \( \text{window} \) is an integer, the signal is divided into segments of length \( \text{window} \). The modified periodograms are computed using a Hamming window of length \( \text{window} \).

\[
pxx = \text{pwelch}(x, \text{window}, \text{noverlap})
\]

uses \( \text{noverlap} \) samples of overlap from segment to segment. \( \text{noverlap} \) must be a positive integer smaller than \( \text{window} \) if \( \text{window} \) is an integer. \( \text{noverlap} \) must be a positive integer less than the length of \( \text{window} \) if \( \text{window} \) is a vector. If you do not specify \( \text{noverlap} \), or specify \( \text{noverlap} \) as empty, the default number of overlapped samples is 50% of the window length.

\[
pxx = \text{pwelch}(x, \text{window}, \text{noverlap}, \text{nfft})
\]

specifies the number of discrete Fourier transform (DFT) points to use in the PSD estimate. The default \( \text{nfft} \) is the greater of 256 or the next power of 2 greater than the length of the segments.

\[
[pxx, w] = \text{pwelch}(__) \text{ returns the normalized frequency vector, } w. \text{ If } pxx \text{ is a one-sided PSD estimate, } w \text{ spans the interval } [0, \pi] \text{ if } nfft \text{ is even and } [0, \pi) \text{ if } nfft \text{ is odd. If } pxx \text{ is a two-sided PSD estimate, } w \text{ spans the interval } [0, 2\pi).
\]

\[
[pxx, f] = \text{pwelch}(__, fs) \text{ returns a frequency vector, } f, \text{ in cycles per unit time. The sample rate, } fs, \text{ is the number of samples per unit time. If the unit of time is seconds, then } f \text{ is in cycles/sec (Hz). For real-valued signals, } f \text{ spans the interval } [0, fs/2] \text{ when } nfft \text{ is even and } [0, fs/2) \text{ when } nfft \text{ is odd. For complex-valued signals, } f \text{ spans the interval } [0, fs). \text{ fs must be the fifth input to } \text{pwelch}. \text{ To input a sample rate and still use the default values of the preceding optional arguments, specify these arguments as empty, []}.
\]

\[
[pxx, w] = \text{pwelch}(x, \text{window}, \text{noverlap}, w) \text{ returns the two-sided Welch PSD estimates at the normalized frequencies specified in the vector, } w. \text{ The vector } w \text{ must contain at least two elements, because otherwise the function interprets it as } nfft.
\]

\[
[pxx, f] = \text{pwelch}(x, \text{window}, \text{noverlap}, f, fs) \text{ returns the two-sided Welch PSD estimates at the frequencies specified in the vector, } f. \text{ The vector } f \text{ must contain at least two elements, because otherwise the function interprets it as } nfft. \text{ The frequencies in } f \text{ are in cycles per unit time. The sample rate, } fs, \text{ is the number of samples per unit time. If the unit of time is seconds, then } f \text{ is in cycles/sec (Hz).}
\[ \text{[___]} = \text{pwelch}(x, \text{window, ___}, \text{freqrange}) \] returns the Welch PSD estimate over the frequency range specified by \text{freqrange}. Valid options for \text{freqrange} are: 'onesided', 'twosided', or 'centered'.

\[ \text{[___]} = \text{pwelch}(x, \text{window, ___}, \text{trace}) \] returns the maximum-hold spectrum estimate if \text{trace} is specified as 'maxhold' and returns the minimum-hold spectrum estimate if \text{trace} is specified as 'minhold'.

\[ \text{[___,pxxc]} = \text{pwelch}(___, \text{'ConfidenceLevel'}, \text{probability}) \] returns the probability \( \times 100\% \) confidence intervals for the PSD estimate in \text{pxxc}.

\[ \text{[___]} = \text{pwelch}(___, \text{spectr umtype}) \] returns the PSD estimate if \text{spectr umtype} is specified as 'psd' and returns the power spectrum if \text{spectr umtype} is specified as 'power'.

\text{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.

\text{rng default}

\text{n = 0:319;}
\text{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.
Repeat the computation.

- Divide the signal into sections of length \( nsc = \lceil N_x/4.5 \rceil \). 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 \( p = \lceil \log_2 nsc \rceil \).
Verify that the two approaches give identical results.

\[
\begin{align*}
\text{Nx} &= \text{length}(x); \\
\text{nsc} &= \text{floor}(\text{Nx}/4.5); \\
\text{nov} &= \text{floor}(\text{nsc}/2); \\
\text{nff} &= \max(256,2^{\text{nextpow2}(\text{nsc})}); \\
\text{t} &= \text{pwelch}(x,\text{hamming}(\text{nsc}),\text{nov},\text{nff}); \\
\text{maxerr} &= \max(\text{abs}(\text{abs}(\text{t}(:))-\text{abs}(\text{pxx}(:)))) \\
\text{maxerr} &= 0
\end{align*}
\]

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.

\[
\begin{align*}
\text{ns} &= 8; \\
\text{ov} &= 0.5; \\
\text{lsc} &= \text{floor}(\text{Nx}/(\text{ns}-(\text{ns}-1)*\text{ov})); \\
\text{t} &= \text{pwelch}(x,\text{lsc},\text{floor}(\text{ov}\times\text{lsc}),\text{nff}); \\
\text{maxerr} &= \max(\text{abs}(\text{abs}(\text{t}(:))-\text{abs}(\text{pxx}(:)))) \\
\text{maxerr} &= 0
\end{align*}
\]

**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.

\sil{rng default}

\[
\begin{align*}
\text{n} &= 0:511; \\
\text{x} &= \cos(\pi/3\times\text{n})+\text{randn}(.size(\text{n}))
\end{align*}
\]

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.

```matlab
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 overlapping 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.

```matlab
segmentLength = 100;
noverlap = 25;
pxx = pwelch(x,segmentLength,noverlap);
```

```matlab
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.

```matlab
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.

```matlab
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.

\[
[pxx,f] = \text{pwelch}(x,500,300,500,fs);
\]

\[
\text{plot}(f,10*\log10(pxx))\]

\[
\text{xlabel('Frequency (Hz)'})
\]

\[
\text{ylabel('PSD (dB/Hz)'})
\]
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.

```matlab
[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.

```matlab
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 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: \( \cos\left(\pi/4 \cdot (0:159)\right) + \text{randn}(1,160) \) is a single-channel row-vector signal.

Example: \( \cos\left(\pi \cdot [4;2] \cdot (0:159)\right) + \text{randn}(160,2) \) 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 \( \text{window} \) is a vector, \( \text{pwelch} \) divides \( x \) into overlapping segments of length equal to the length of \( \text{window} \), and then multiplies each signal segment with the vector specified in \( \text{window} \). If \( \text{window} \) is an integer, \( \text{pwelch} \) 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 \( \text{noverlap} \) number of overlapping samples, \( x \) is truncated accordingly. If you specify \( \text{window} \) as empty, the default Hamming window is used to obtain eight segments of \( x \) with \( \text{noverlap} \) 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 \( \text{window} \). If you omit \( \text{noverlap} \) or specify \( \text{noverlap} \) 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, \( \text{pxx} \) has length \( (\text{nfft}/2 + 1) \) if \( \text{nfft} \) is even, and \((\text{nfft} + 1)/2\) if \( \text{nfft} \) is odd. For a complex-valued input signal, \( x \), the PSD estimate always has length \( \text{nfft} \). If \( \text{nfft} \) is specified as empty, the default \( \text{nfft} \) is used.

If \( \text{nfft} \) is greater than the segment length, the data is zero-padded. If \( \text{nfft} \) is less than the segment length, the segment is wrapped using \( \text{datawrap} \) to make the length equal to \( \text{nfft} \).

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 \([0,\pi]\) rad/sample. If nfft is odd, the length of pxx is (nfft + 1)/2 and the interval is \([0,\pi]\) rad/sample. When fs is optionally specified, the corresponding intervals are \([0,fs/2]\) cycles/unit time and \([0,fs/2]\) cycles/unit time for even and odd length nfft respectively.
- 'twosided' — returns the two-sided PSD estimate for either the real-valued or complex-valued input, x. In this case, pxx has length nfft and is computed over the interval \([0,2\pi]\) rad/sample. When fs is optionally specified, the interval is \([0,fs]\) cycles/unit time.
'centered' — returns the centered two-sided PSD estimate for either the real-valued or complex-valued input, \( x \). In this case, \( pxx \) has length \( nfft \) and is computed over the interval \(( -\pi, \pi ] \) rad/sample for even length \( nfft \) and \(( -\pi, \pi ) \) rad/sample for odd length \( nfft \). When \( fs \) is optionally specified, the corresponding intervals are \((-fs/2, fs/2] \) cycles/unit time and \((-fs/2, fs/2) \) cycles/unit time for even and odd length \( nfft \) respectively.

**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. \( \text{pwelch} \) 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. \( \text{pwelch} \) 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. \( \text{pwelch} \) 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 \( \times 100\% \) interval estimate for the true PSD.
Output Arguments

**pxx — PSD estimate**

vector | matrix

PSD estimate, returned as a real-valued, nonnegative column vector or matrix. Each column of pxx is the PSD estimate of the corresponding column of x. The units of the PSD estimate are in squared magnitude units of the time series data per unit frequency. For example, if the input data is in volts, the PSD estimate is in units of squared volts per unit frequency. For a time series in volts, if you assume a resistance of 1 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 \( pxxc(m, 2n) \) 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.

Extended Capabilities

C/C++ Code Generation

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

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

```matlab
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).

\[
[\_\_, pxxc] = pyulear(\_\_, 'ConfidenceLevel', probability)\]
returns the \(probability \times 100\%\) confidence intervals for the PSD estimate in \(pxxc\).

\texttt{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.
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.

```matlab
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)')
```

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.

$$Fs = 1000;$$

$$t = 0:1/Fs:1-1/Fs;$$

$$f = [100;200;300];$$

$$x = \cos(2\pi f t) + \text{randn}(\text{length}(t),3);$$

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.

$$morder = 12;$$

$$\text{pyulear}(x,morder,[]),Fs)$$
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 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) cycles/unit time for even and odd length nfft respectively.

- 'twosided' — returns the two-sided PSD estimate for either the real-valued or complex-valued input, x. In this case, pxx has length nfft and is computed over the interval [0,2π) 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 \([0,\pi]\) 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
See Also
pbureg | 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(___)

Description

c = rainflow(x) returns cycle counts for the load time history, x, according to the ASTM E 1049 standard. See “Algorithms” on page 1-2043 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.

\[
[c, \text{hist}, \text{edges}, \text{rmm}, \text{idx}] = \text{rainflow}(Y, fs);
\]

\[
T = \text{array2table}(c, 'VariableNames', \{'\text{Count}', '\text{Range}', '\text{Mean}', '\text{Start}', '\text{End}'\})
\]
Display a histogram of cycle counts as a function of stress range.

```matlab
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.

```matlab
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.

```matlab
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)
```matlab
[c,hist,edges,rmm,idx] = rainflow(Z,t);
TT = array2table(c,'VariableNames',{'Count','Range','Mean','Start','End'});
```

<table>
<thead>
<tr>
<th>Count</th>
<th>Range</th>
<th>Mean</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>3</td>
<td>-0.5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0.5</td>
<td>4</td>
<td>-1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>0.5</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>0.5</td>
<td>9</td>
<td>0.5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>0.5</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>15</td>
</tr>
</tbody>
</table>

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.

```matlab
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.
Cycle Counts of Identified Reversals

Generate a set of extrema resembling load reversals. Plot the data.

```matlab
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.

```matlab
[C,hist,edges] = rainflow(X,'ext');
```

Display a histogram of cycle counts as a function of stress range.

```matlab
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” (MATLAB).

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 1-2043 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.

\( \text{rmm, rmr} \) — Histogram bin edges

vectors

Histogram bin edges, returned as vectors. \( \text{rmm} \) and \( \text{rmr} \) contain the bin edges of the rows and columns of \( rm \), respectively.

\( \text{idx} \) — 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:
Start

Set \( Z \) to the first reversal in the history

Read the next reversal

Out of data?

YES

Count each remaining range as 1/2 cycle

NO

<3 reversals?

YES

End

NO

Form \( X \) and \( Y \) using the 3 latest nondiscarded reversals: \( Y:(1-2), \ X:(2-3) \)

\( r(X) < r(Y) ? \)

YES

Y includes \( Z \)?

YES

1. Count \( Y \) as 1/2 cycle
2. Discard the first reversal of \( Y \)
3. Set \( Z \) to the second reversal of \( Y \)

NO

NO

1. Count \( Y \) as 1 cycle
2. Discard both points of \( Y \)
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 the following reversal sequence:

<table>
<thead>
<tr>
<th>Step</th>
<th>Z</th>
<th>Reversals</th>
<th>Three Reversals?</th>
<th>Y</th>
<th>r(Y)</th>
<th>X</th>
<th>r(X)</th>
<th>r(X) &lt; r(Y)?</th>
<th>Z in Y?</th>
<th>Actions</th>
</tr>
</thead>
</table>
| 1    | A  | A, B, C   | Yes              | AB  | 3    | BC  | 4    | No            | Yes    | 1 Count AB as \( \frac{1}{2} \) cycle.  
2 Discard A.  
3 Set Z to B. |
<p>| 2    | B  | B, C      | No               | —   | —    | —   | —    | —             | —      | Read D.                   |</p>
<table>
<thead>
<tr>
<th>Step</th>
<th>Z</th>
<th>Reversals</th>
<th>Three Reversals?</th>
<th>Y</th>
<th>r((Y))</th>
<th>X</th>
<th>r((X))</th>
<th>r((X)) &lt; r((Y))?</th>
<th>Z in Y?</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>B</td>
<td>B, C, D</td>
<td>Yes</td>
<td>BC</td>
<td>4</td>
<td>CD</td>
<td>8</td>
<td>No</td>
<td>Yes</td>
<td>1 Count BC as (\frac{1}{2}) cycle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 Discard B.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 Set Z to C.</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>C, D</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Read E.</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>C, D, E</td>
<td>Yes</td>
<td>CD</td>
<td>8</td>
<td>DE</td>
<td>6</td>
<td>Yes</td>
<td></td>
<td>Read F.</td>
</tr>
<tr>
<td>6</td>
<td>C</td>
<td>C, D, E, F</td>
<td>Yes</td>
<td>DE</td>
<td>6</td>
<td>EF</td>
<td>4</td>
<td>Yes</td>
<td></td>
<td>Read G.</td>
</tr>
<tr>
<td>7</td>
<td>C</td>
<td>C, D, E, F, G</td>
<td>Yes</td>
<td>EF</td>
<td>4</td>
<td>FG</td>
<td>7</td>
<td>No</td>
<td>No</td>
<td>1 Count EF as 1 cycle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 Discard E and F.</td>
</tr>
<tr>
<td>8</td>
<td>C</td>
<td>C, D, G</td>
<td>Yes</td>
<td>CD</td>
<td>8</td>
<td>DG</td>
<td>9</td>
<td>No</td>
<td>Yes</td>
<td>1 Count CD as (\frac{1}{2}) cycle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 Discard C.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 Set Z to D.</td>
</tr>
<tr>
<td>9</td>
<td>D</td>
<td>D, G</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Read H.</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
<td>D, G, H</td>
<td>Yes</td>
<td>DG</td>
<td>9</td>
<td>GH</td>
<td>8</td>
<td>Yes</td>
<td></td>
<td>Read J.</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
<td>D, G, H, J</td>
<td>Yes</td>
<td>GH</td>
<td>8</td>
<td>HJ</td>
<td>7</td>
<td>Yes</td>
<td></td>
<td>Read K.</td>
</tr>
<tr>
<td>12</td>
<td>D</td>
<td>D, G, H, J, K</td>
<td>Yes</td>
<td>HJ</td>
<td>7</td>
<td>JK</td>
<td>4</td>
<td>Yes</td>
<td></td>
<td>Read L.</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>D, G, H, J, K, L</td>
<td>Yes</td>
<td>JK</td>
<td>4</td>
<td>KL</td>
<td>3</td>
<td>Yes</td>
<td></td>
<td>Read M.</td>
</tr>
<tr>
<td>14</td>
<td>D</td>
<td>D, G, H, J, K, L, M</td>
<td>Yes</td>
<td>KL</td>
<td>3</td>
<td>LM</td>
<td>5</td>
<td>No</td>
<td>No</td>
<td>1 Count KL as 1 cycle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 Discard K and L.</td>
</tr>
<tr>
<td>15</td>
<td>D</td>
<td>D, G, H, J, M</td>
<td>Yes</td>
<td>HJ</td>
<td>7</td>
<td>JM</td>
<td>5</td>
<td>Yes</td>
<td></td>
<td>Read N.</td>
</tr>
<tr>
<td>16</td>
<td>D</td>
<td>D, G, H, J, M, N</td>
<td>Yes</td>
<td>JM</td>
<td>5</td>
<td>MN</td>
<td>1</td>
<td>Yes</td>
<td></td>
<td>Read P.</td>
</tr>
</tbody>
</table>
Now collect the results.

<table>
<thead>
<tr>
<th>Cycle Count</th>
<th>Range</th>
<th>Mean</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>½</td>
<td>3</td>
<td>-0.5</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>½</td>
<td>4</td>
<td>-1</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>½</td>
<td>8</td>
<td>1</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>-0.5</td>
<td>K</td>
<td>L</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2.5</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0.5</td>
<td>H</td>
<td>J</td>
</tr>
<tr>
<td>½</td>
<td>9</td>
<td>0.5</td>
<td>D</td>
<td>G</td>
</tr>
<tr>
<td>½</td>
<td>10</td>
<td>1</td>
<td>G</td>
<td>P</td>
</tr>
</tbody>
</table>

Compare this to the result of running rainflow on the sequence:

1-2047
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


Extended Capabilities

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

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.
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination</td>
<td>'current' (default), 'new' or</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>SubsystemName</td>
<td></td>
</tr>
<tr>
<td>Blockname</td>
<td>'filter' (default)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MapCoeffstoPorts</td>
<td>'off' (default) or 'on'</td>
<td>Specify whether to map the coefficients of the filter to the ports of the block.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MapStates</td>
<td>'off' (default) or 'on'</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OverwriteBlock</td>
<td>'off' or 'on'</td>
<td>Specify whether to overwrite an existing block with the same name or create a new block.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OptimizeZeros</td>
<td>'off' (default) or 'on'</td>
<td>Specify whether to remove zero-gain blocks.</td>
</tr>
<tr>
<td>Property Name</td>
<td>Property Values</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OptimizeOnes</td>
<td><code>'off'</code> (default) or <code>'on'</code></td>
<td>Specify whether to replace unity-gain blocks with direct connections.</td>
</tr>
<tr>
<td>OptimizeNegOnes</td>
<td><code>'off'</code> (default) or <code>'on'</code></td>
<td>Specify whether to replace negative unity-gain blocks with a sign change at the nearest sum block.</td>
</tr>
<tr>
<td>OptimizeDelayChains</td>
<td><code>'off'</code> (default) or <code>'on'</code></td>
<td>Specify whether to replace cascaded chains of delay blocks with a single integer delay block to provide an equivalent delay.</td>
</tr>
<tr>
<td>CoeffNames</td>
<td>{'Num'}, {'Num','Den'}, {'Num','Den','g'}, {'K'}</td>
<td>Specify the coefficient variable names as a cell array. MapCoeffsToPorts must be set to <code>'on'</code> for this property to apply.</td>
</tr>
<tr>
<td>InputProcessing</td>
<td>'columnsaschannels' (default), 'elementsaschannels' or 'inherited'</td>
<td>Specify frame-based ('columnsaschannels') or sample-based ('elementsaschannels') processing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The Inherited (this choice will be removed - see release notes) option will be removed in a future release.</td>
</tr>
<tr>
<td>RateOption</td>
<td>'enforcesinglerate' (default) or 'allowmultirate'</td>
<td>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'.</td>
</tr>
</tbody>
</table>
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.

```matlab
[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 \( r0 \).

\[
k = [0.3090 \ 0.9800 \ 0.0031 \ 0.0082 \ -0.0082];
\]

\[
r0 = 0.1;
\]

\[
a = rc2ac(k, r0)
\]

\[
a = 6 \times 1
\]

\[
0.1000
-0.0309
-0.0791
0.0787
0.0294
-0.0950
\]
References


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
\textbf{rc2is}

Convert reflection coefficients to inverse sine parameters

\textbf{Syntax}

\texttt{isin = rc2is(k)}

\textbf{Description}

\texttt{isin = rc2is(k)} returns a vector of inverse sine parameters, \texttt{isin}, from a vector of reflection coefficients, \texttt{k}.

\textbf{Examples}

\textbf{Compute Inverse Sine Parameters}

Define a vector, \texttt{k}, of reflection coefficients and determine the corresponding inverse sine parameters.

\begin{verbatim}
  k = [0.3090 0.9801 0.0031 0.0082 -0.0082];
  isin = rc2is(k)

  isin = 1x5
  0.2000   0.8728   0.0020   0.0052  -0.0052
\end{verbatim}

\textbf{References}

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 = 1x5

0.6389 4.6002 0.0062 0.0164 -0.0164

References

See Also
lar2rc

Introduced before R2006a
rc2poly

Convert reflection coefficients to prediction filter polynomial

Syntax

```matlab
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.

```matlab
k = [0.3090 0.9800 0.0031 0.0082 -0.0082];
a = rc2poly(k)
a = 1x6

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


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)
rceps(x)

Description

[y,ym] = rceps(x) returns both the real cepstrum y and a minimum phase reconstructed version ym of the input sequence.

rceps(x) returns the real cepstrum of the real sequence x.

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 $\Delta$ samples and attenuated by a known factor $\alpha$: $y(n) = x(n) + \alpha x(n - \Delta)$. Specify a time lag of 0.23 s and an attenuation factor of 0.5.

timelag = 0.23;
delta = round(Fs*timelag);
alpha = 0.5;
orig = [mtlb;zeros(delta,1)];
echo = [zeros(delta,1);mtlb]*alpha;
mtEcho = orig + echo;

Plot the original, the echo, and the resulting signal.

```
t = (0:length(mtEcho)-1)/Fs;

subplot(2,1,1)
plot(t,[orig echo])
legend('Original','Echo')

subplot(2,1,2)
plot(t,mtEcho)
legend('Total')
xlabel('Time (s)')
```
% To hear, type soundsc(mtEcho,Fs)

Compute 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.

```matlab
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)
```

1 Functions — Alphabetical List

1-2064
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.

\( y_m \) — 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** \( \text{rcsps} \) only works on real data.

\( \text{rcsps} \) is an implementation of algorithm 7.2 in [2], that is,

\[
y = \text{real}(\text{ifft}(\log(\text{abs}(\text{fft}(x)))))
\]

Appropriate windowing in the cepstral domain forms the reconstructed minimum phase signal:

\[
w = [1;2*\text{ones}(n/2-1,1);\text{ones}(1-\text{rem}(n,2),1);\text{zeros}(n/2-1,1)];
\]
\[
y_m = \text{real}(\text{ifft}(\exp(\text{fft}(w.*y))))
\]

References


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

\[
b = \text{rcosdesign}(\beta, \text{span}, \text{sps})
\]

\[
b = \text{rcosdesign}(\beta, \text{span}, \text{sps}, \text{shape})
\]

Description

\[
b = \text{rcosdesign}(\beta, \text{span}, \text{sps})\]
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 \(\text{span}\) symbols, and each symbol period contains \(\text{sps}\) samples. The order of the filter, \(\text{sps} * \text{span}\), must be even. The filter energy is 1.

\[
b = \text{rcosdesign}(\beta, \text{span}, \text{sps}, \text{shape})\]
returns a square-root raised cosine filter when you set \(\text{shape}\) to 'sqrt' and a normal raised cosine FIR filter when you set \(\text{shape}\) 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 \(\text{shape}\) parameter.

\[
h = \text{rcosdesign}(0.25, 6, 4);
\]

\[
\text{mx} = \max(\text{abs}(h - \text{rcosdesign}(0.25, 6, 4, 'sqrt'))) \]

\[
\text{mx} = 0
\]

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.
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:

```matlab
rf = 0.25;
span = 4;
sps = 3;

h1 = rcosdesign(rf,span,sps,'normal');
fvtool(h1,'impulse')

h2 = rcosdesign(rf,span,sps,'sqrt');
fvtool(h2,'impulse')
```
Convolve the square-root filter with itself. Truncate the impulse response outward from the maximum so it has the same length as $h_1$. Normalize the response using the maximum. Then, compare the convolved square-root filter to the normal filter.

```matlab
h3 = conv(h2,h2);
p2 = ceil(length(h3)/2);
m2 = ceil(p2-length(h1)/2);
M2 = floor(p2+length(h1)/2);
ct = h3(m2:M2);

stem([h1/max(abs(h1));ct/max(abs(ct))], 'filled')
xlabel('Samples')
ylabel('Normalized amplitude')
legend('h1','h2 * h2')
```
The convolved response does not coincide with the normal filter because of its finite length. Increase span to obtain closer agreement between the responses and better compliance with the Nyquist criterion.

Pass a Signal through a Raised Cosine Filter

This example shows how to pass a signal through a square-root, raised cosine filter. Specify the filter parameters.
rolloff = 0.25; % Rolloff factor
span = 6; % Filter span in symbols
sps = 4; % Samples per symbol

Generate the square-root, raised cosine filter coefficients.
b = rcosdesign(rolloff, span, sps);

Create a vector of bipolar data.
d = 2*randi([0 1], 100, 1) - 1;

Upsample and filter the data for pulse shaping.
x = upfirdn(d, b, sps);

Add noise.
r = x + randn(size(x))*0.01;

Filter and downsample the received signal for matched filtering.
y = upfirdn(r, b, 1, sps);

Interpolate and Decimate Using RRC Filter

This example shows how to interpolate and decimate signals using square-root, raised cosine filters designed with the rcosdesign function. This example requires the Communications Toolbox™ software.

Define the square-root raised cosine filter parameters. Define the signal constellation parameters.

rolloff = 0.25; % Filter rolloff
span = 6; % Filter span
sps = 4; % Samples per symbol
M = 4; % Size of the signal constellation
k = log2(M); % Number of bits per symbol

Generate the coefficients of the square-root raised cosine filter using the rcosdesign function.

rrcFilter = rcosdesign(rolloff, span, sps);
Generate 10000 data symbols using the `randi` function.

```matlab
data = randi([0 M-1], 10000, 1);
```

Apply PSK modulation to the data symbols. Because the constellation size is 4, the modulation type is QPSK.

```matlab
modData = pskmod(data, M, pi/4);
```

Using the `upfirdn` function, upsample and filter the input data.

```matlab
txSig = upfirdn(modData, rrcFilter, sps);
```

Convert the Eb/N0 to SNR and then pass the signal through an AWGN channel.

```matlab
EbNo = 7;
snr = EbNo + 10*log10(k) - 10*log10(sps);
rxSig = awgn(txSig, snr, 'measured');
```

Filter and downsample the received signal. Remove a portion of the signal to account for the filter delay.

```matlab
rxFilt = upfirdn(rxSig, rrcFilter, 1, sps);
rxFilt = rxFilt(span+1:end-span);
```

Create a scatterplot of the modulated data using the first 5000 symbols.

```matlab
hScatter = scatterplot(sqrt(sps)* ...    
    rxSig(1:sps*5000), ...    
    sps, 0, 'g.');
hold on    
scatterplot(rxFilt(1:5000), 1, 0, 'kx', hScatter)    
title('Received Signal, Before and After Filtering')    
legend('Before Filtering', 'After Filtering')    
axis([-3 3 -3 3]) % Set axis ranges    
hold off
```
Input Arguments

**beta — Rolloff factor**

real nonnegative scalar

Rolloff factor, specified as a real nonnegative scalar not greater than 1. The rolloff factor determines the excess bandwidth of the filter. Zero rolloff corresponds to a brick-wall filter and unit rolloff to a pure raised cosine.

Data Types: double | single
span — Number of symbols
positive scalar

Number of symbols, specified as a positive integer scalar.
Data Types: double | single

sps — Samples per symbol
positive integer scalar

Number of samples per symbol (oversampling factor), specified as a positive integer scalar.
Data Types: double | single

shape — Shape of the raised cosine window
'sqrt' (default) | 'normal'

Shape of the raised cosine window, specified as 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


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

Introduced in R2013b
**rectpuls**

Sampled aperiodic rectangle

**Syntax**

```matlab
y = rectpuls(t)
y = rectpuls(t,w)
```

**Description**

```matlab
y = rectpuls(t) returns a continuous, aperiodic, unit-height rectangular pulse at the sample times indicated in array t, centered about t = 0.
```

```matlab
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.

```matlab
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.

```matlab
tpast = -45e-3;
xpast = rectpuls(t-tpast,w);
```
- One displaced 60 ms into the future and half as wide.

\[
t_{\text{futr}} = 60e^{-3}; \\
x_{\text{futr}} = \text{rectpuls}(t-t_{\text{futr}}, w/2);
\]

Plot the original pulse and the two copies on the same axes.

\[
\text{plot}(t,x,t,x_{\text{past}},t,x_{\text{futr}}) \\
\text{ylim}([-0.2 \ 1.2])
\]
Input Arguments

t — Sample times
  vector

Sample times of unit rectangular pulse, specified by a vector.
Data Types: single | double

w — Rectangle width
1 (default) | positive number

Rectangle width, specified by 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 | tri脉

Introduced before R2006a
rectwin

Rectangular window

Syntax

\[ w = \text{rectwin}(L) \]

Description

\[ w = \text{rectwin}(L) \] returns a rectangular window of length \( L \).

Examples

Rectangular Window

Create a 64-point rectangular window. Display the result using \( \text{wvtool} \).

\[
\begin{align*}
L &= 64; \\
\text{wvtool} & (\text{rectwin}(L))
\end{align*}
\]
**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

![Figure 1: Window Visualization Tool](image)
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.

\[
w = \text{ones}(L,1);
\]

**References**


**Extended Capabilities**

**C/C++ Code Generation**

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

**See Also**

**Apps**

Window Designer

**Functions**

`barthannwin`, `blackman`, `blackmanharris`, `flattopwin`, `hamming`, `hann`, `wvtool`

Introduced before R2006a
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 array
   "MoanRegions"
   "TrillRegions"
```
Input Arguments

ls — 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.

```matlab
removeMembers(lss,2)
```

```matlab
lss
```

```matlab
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 point 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.

```matlab
clear all
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.

```matlab
nm = lss.NumMembers;
```
Define a point label associated with the signal maximum.

```matlab
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.

```matlab
addLabelDefinitions(lss,themax)
```

Find the maxima of the signals and add their values to the labeled set.

```matlab
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

<table>
<thead>
<tr>
<th>WhaleType</th>
<th>MoanRegions</th>
<th>TrillRegions</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member{1}</td>
<td>blue</td>
<td>{3x2 table}</td>
<td>{1x3 table}</td>
</tr>
</tbody>
</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',sidx)
removeRegionValue(lss,midx,lblname,'LabelRowIndex',ridx,'SublabelRowIndex',sidx)

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.

```matlab
code
load whales
lss
```

```matlab
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.
bbledefs = getLabelValues(lss)

<table>
<thead>
<tr>
<th>WhaleType</th>
<th>MoanRegions</th>
<th>TrillRegions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member{1}</td>
<td>blue</td>
<td>{3x2 table}</td>
</tr>
<tr>
<td>Member{2}</td>
<td>blue</td>
<td>{3x2 table}</td>
</tr>
</tbody>
</table>

idx = 2;

Retrieve the signal and the time information. Plot the signal.

[lbs,info] = getLabeledSignal(lss,idx)

<table>
<thead>
<tr>
<th>Signal</th>
<th>WhaleType</th>
<th>MoanRegions</th>
<th>TrillRegions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member{2}</td>
<td>{76579x1 double}</td>
<td>blue</td>
<td>{3x2 table}</td>
</tr>
</tbody>
</table>

info = struct with fields:
    TimeInformation: "sampleRate"
    SampleRate: 4000

fs = info.SampleRate;
sg = getSignal(lss,idx);
t = (0:length(sg)-1)/fs;
plot(t,sg)

Highlight the moans and trills of the signal.

mvals = getLabelValues(lss,idx,'MoanRegions');
tvals = getLabelValues(lss,idx,'TrillRegions');

hold on
[X,Y] = meshgrid([mvals.ROILimits;tvals.ROILimits],ylim);
plot(X,Y,:k)
topts = {'HorizontalAlignment','center','FontWeight','bold', ...
'FontSize',12,'Color',[139 69 19]/255);
text((X(1,1:4)+X(1,5:end))/2,Y(2,5:end)-0.1, ...
Remove the second moan from the labels. Plot the signal again. Highlight the moans and trills.

```
removeRegionValue(lss, idx, 'MoanRegions', 'LabelRowIndex', 2)
```

```
plot(t, sg)
mvals = getLabelValues(lss, idx, 'MoanRegions');
hold on
[X,Y] = meshgrid([mvals.ROILimits; tvals.ROILimits], ylim);
plot(X, Y, ':k')
```
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
resample

Resample uniform or nonuniform data to new fixed rate

Syntax

\[
\begin{align*}
y &= \text{resample}(x,p,q) \\
y &= \text{resample}(x,p,q,n) \\
y &= \text{resample}(x,p,q,n,beta) \\
y &= \text{resample}(x,p,q,b) \\
[y,b] &= \text{resample}(x,p,q,\text{___}) \\
y &= \text{resample}(x,tx) \\
y &= \text{resample}(x,tx,fs) \\
y &= \text{resample}(x,tx,fs,p,q) \\
y &= \text{resample}(x,tx,\text{___},\text{method}) \\
[y,ty] &= \text{resample}(x,tx,\text{___}) \\
[y,ty,b] &= \text{resample}(x,tx,\text{___})
\end{align*}
\]

Description

\[
y = \text{resample}(x,p,q) \text{ resamples the input sequence, } x, \text{ at } p/q \text{ times the original sample rate. If } x \text{ is a matrix, then } \text{resample} \text{ treats each column of } x \text{ as an independent channel. } \text{resample} \text{ applies an antialiasing FIR lowpass filter to } x \text{ and compensates for the delay introduced by the filter.}
\]

\[
y = \text{resample}(x,p,q,n) \text{ uses an antialiasing filter of order } 2 \times n \times \text{max}(p,q).
\]

\[
y = \text{resample}(x,p,q,n,beta) \text{ specifies the shape parameter of the Kaiser window used to design the lowpass filter.}
\]

\[
y = \text{resample}(x,p,q,b) \text{ filters } x \text{ using the filter coefficients specified in } b.
\]

\[
[y,b] = \text{resample}(x,p,q,\text{___}) \text{ also returns the coefficients of the filter applied to } x \text{ during the resampling.}
\]

\[
y = \text{resample}(x,tx) \text{ resamples the values, } x, \text{ of a signal sampled at the instants specified in vector } tx. \text{ The function interpolates } x \text{ linearly onto a vector of uniformly}
\]
spaced instants with the same endpoints and number of samples as \( tx \). NaNs are treated as missing data and are ignored.

\[
y = \text{resample}(x,tx,fs)
\]

uses a polyphase antialiasing filter to resample the signal at the uniform sample rate specified in \( fs \).

\[
y = \text{resample}(x,tx,fs,p,q)
\]

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 \( fs \times q/p \) is at least twice as large as the highest frequency component of \( x \).

\[
y = \text{resample}(x,tx,\_,\_,\text{method})
\]

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'.

**Note:** If \( x \) is not slowly varying, consider using \( \text{interp1} \) with the 'pchip' interpolation method.

\[
[y,ty] = \text{resample}(x,tx,\_,\_)
\]

returns in \( ty \) the instants that correspond to the resampled signal.

\[
[y,ty,b] = \text{resample}(x,tx,\_,\_)
\]

returns in \( b \) the coefficients of the antialiasing filter.

**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.

\[
\begin{align*}
fs &= 10; \\
t1 &= 0:1/fs:1; \\
x &= t1; \\
y &= \text{resample}(x,3,2); \\
t2 &= (0:(\text{length}(y)-1))*2/(3*fs);
\end{align*}
\]

\[
\text{plot}(t1,x,'*',t2,y,'o') \\
xlabel('Time (s)') \\
ylabel('Signal')
\]
When filtering, \texttt{resample} assumes that the input sequence, \texttt{x}, is zero before and after the samples it is given. Large deviations from zero at the endpoints of \texttt{x} can result in unexpected values for \texttt{y}.

Show these deviations by resampling a triangular sequence and a vertically shifted version of the sequence with nonzero endpoints.

\begin{verbatim}
x = [1:10 9:-1:1; 10:-1:1 2:10]';
y = resample(x,3,2);
\end{verbatim}
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.

```matlab
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.

```matlab
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)
```
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.

\[
n = 1; \\
\text{beta} = 0; \\
[y, b] = \text{resample}(x, \text{ups}, \text{dns}, n, \text{beta}); \\
fo = \text{filtord}(b)
\]
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.
\( n = 5; \)
\[
[y,b] = \text{resample}(x,\text{ups},\text{dns},n,\text{beta});
\]
\[
fo = \text{filtord}(b)
\]
\[
fo = 40
\]
\[
\text{stem}(tu,y)
\]
\[
\text{hold on}
\]
\[
\text{stairs}(t,x,'--')
\]
\[
\text{hold off}
\]
\[
\text{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 $\beta = 20$.

\begin{verbatim}
beta = 20;
y = resample(x,ups,dns,n,beta);

stem(tu,y)
hold on
stairs(t,x,'--')
hold off
legend('n = 5, \beta = 20')
\end{verbatim}
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.

```matlab
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.

```matlab
tz = 0:9:360-9;
[z,bz] = resample(x,2,3);

plot(tx,x,'+-',tz,z,'o:');
legend('original','resampled')
```
Plot the impulse response of the new lowpass filter.

impz(bz)
Resample a Nonuniformly Sampled Data Set

Use the data recorded by Galileo Galilei in 1610 to determine the orbital period of Callisto, the outermost of Jupiter's four largest satellites.

Galileo observed the satellites' motion for six weeks, starting on 15 January. The observations have several gaps because Jupiter was not visible on cloudy nights. Generate a `datetime` array of observation times.

\[
t = [0\ 2\ 3\ 7\ 8\ 9\ 10\ 11\ 12\ 17\ 18\ 19\ 20\ 24\ 25\ 26\ 27\ 28\ 29\ 31\ 32\ 33\ 35\ 37\ \ldots\ 41\ 42\ 43\ 44\ 45]' + 1;
\]
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.

```matlab
fs = 1/86400;
[ys,tys] = resample(yg,obsv,fs,3,1,'spline');
plot(t,yg,'o')
hold on
plot(ys,'.-')
hold off
ax = gca;
```
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})')

**Input Arguments**

x — Input signal  
vector | matrix

Input signal, specified as a vector or matrix. If x is a matrix, then its columns are treated as independent channels. x can contain NaNs. NaNs are treated as missing data and are excluded from the resampling.
Example: \( \cos\left(\frac{\pi}{4} \cdot (0:159)\right) + \text{randn}(1,160) \) is a single-channel row-vector signal.

Example: \( \cos\left(\frac{\pi}{[4;2]} \cdot (0:159)\right) + \text{randn}(160,2) \) 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 \( n = 0 \), \text{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, \text{resample} designs the filter using \text{firls} with a Kaiser window. When compensating for the delay, \text{resample} assumes \( b \) has odd length and linear phase.

Example: \text{fir1}(4,0.5) \) specifies a 4th-order lowpass filter with normalized cutoff frequency 0.5\( \pi \) rad/sample.

**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.

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.

**Output Arguments**

**y — Resampled signal**  
vector | matrix

Resampled signal, returned as a vector or matrix. If x is a signal of length N and you specify p and q, then y is of length \( N \times p/q \).

**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.
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:

C and C++ code generation for resample requires DSP System Toolbox software.

The upsampling and downsampling factors must be specified as constants. Expressions or variables are allowed if their values do not change.

Variable-size inputs are not supported.

See Also

decimate | downsample | firls | interp | interp1 | intfilt | kaiser | spline | upfirdn | upsample

Topics

“Reconstruct a Signal from Irregularly Sampled Data”

Introduced before R2006a
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.

```matlab
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 `setLabelDefinitionsHierarchy` to see a list of labels and sublabels. Use `setLabelValue` to add data to the set.

```matlab
getLabelNames(lss)

ans = 3x1 string array
    "WhaleType"
    "MoanRegions"
    "TrillRegions"
```

Get the label values corresponding to the trill regions for the second signal in the set.

```matlab
idx = 2;
getLabelValues(lss,idx,'TrillRegions')

ans=1x2 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=1x3 table

<table>
<thead>
<tr>
<th>WhaleType</th>
<th>MoanRegions</th>
<th>TrillRegions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member{2}</td>
<td>blue</td>
<td>{3x2 table}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{0x3 table}</td>
</tr>
</tbody>
</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`
residuez

Z-transform partial-fraction expansion

Syntax

\[ [\text{ro}, \text{po}, \text{ko}] = \text{residuez}(\text{bi}, \text{ai}) \]
\[ [\text{bo}, \text{ao}] = \text{residuez}(\text{ri}, \text{pi}, \text{ki}) \]

Description

\[ [\text{ro}, \text{po}, \text{ko}] = \text{residuez}(\text{bi}, \text{ai}) \]
finds the residues, poles, and direct terms of a partial fraction expansion of the ratio of numerator and denominator polynomials, \( b \) and \( a \).

\[ [\text{bo}, \text{ao}] = \text{residuez}(\text{ri}, \text{pi}, \text{ki}) \]
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.

\[
[bn, an] = \text{residuez}(r, p, k)
\]

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 \( z \).

\[
B(z) = b_0 + b_1z^{-1} + b_2z^{-2} + \cdots + b_mz^{-m} \\
A(z) = a_0 + a_1z^{-1} + a_2z^{-2} + \cdots + a_nz^{-n}
\]

If there are multiple roots and \( a > n - 1 \),

\[
\frac{B(z)}{A(z)} = \frac{r(1)}{1 - p(1)z^{-1}} + \frac{r(n)}{1 - p(n)z^{-1}} + k(1) + k(2)z^{-1} + \cdots + k(m-n+1)z^{-(m-n)}
\]

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

\( n = \text{length}(a) - 1 = \text{length}(r) = \text{length}(p) \)

If \( p(j) = \ldots = p(j+s-1) \) is a pole of multiplicity \( s \), then the expansion includes terms of the form

\[
\frac{r(j)}{1 - p(j)z^{-1}} + \frac{r(j + 1)}{(1 - p(j)z^{-1})^2} + \ldots + \frac{r(j + s - 1)}{(1 - p(j)z^{-1})^s}
\]

**ko — Direct term**

Row vector

Direct terms, returned as a row vector. The direct term coefficient vector \( k \) is empty if \( \text{length}(b) \) is less than \( \text{length}(a) \); otherwise:

\( \text{length}(k) = \text{length}(b) - \text{length}(a) + 1 \)

**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 \( \text{length}(b) > \text{length}(a) - 1 \).
• The poles using \( p = \text{roots}(a) \).
• 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 \( 1/(1 - p_jz^{-1}) \) 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 \( \backslash \). \( h \) 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.

\[
\frac{1}{1 - p_jz^{-1}}, \frac{1}{(1 - p_jz^{-1})^2}, \ldots, \frac{1}{(1 - p_jz^{-1})^{s_j}}
\]

The vector \( h \) and matrices \( S1 \) and \( S2 \) have \( n + \text{xtra} \) rows, where \( n \) is the total number of roots and the internal parameter \( \text{xtra} \), 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 \texttt{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 \texttt{residuez}.

**References**


**See Also**

\texttt{convmtx | deconv | poly | prony | residue | roots | ss2tf | tf2ss | tf2zp | tf2zpk | zp2ss}
Introduced before R2006a
risetime

Rise time of positive-going bilevel waveform transitions

Syntax

\[
\begin{align*}
R &= \text{risetime}(X) \\
R &= \text{risetime}(X,FS) \\
R &= \text{risetime}(X,T) \\
[R,LT,UT] &= \text{risetime}(\_\_) \\
[R,LT,UT,LL,UL] &= \text{risetime}(\_\_) \\
[\_\_] &= \text{risetime}(\_\_,\text{Name,Value}) \\
\text{risetime}(\_\_) &= \text{risetime}(\_\_) 
\end{align*}
\]

Description

\( R = \text{risetime}(X) \) 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, \( \text{risetime} \) estimates the state levels of the input waveform by a histogram method. \( \text{risetime} \) 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 1-2147. Because \( \text{risetime} \) uses interpolation, \( R \) may contain values that do not correspond to sampling instants of the bilevel waveform, \( X \).

\( R = \text{risetime}(X,FS) \) 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 \( \text{risetime} \) uses interpolation, \( R \) may contain values that do not correspond to sampling instants of the bilevel waveform, \( X \).

\( R = \text{risetime}(X,T) \) specifies the sample instants, \( T \), as a vector with the same number of elements as \( X \).

\( [R,LT,UT] = \text{risetime}(\_\_) \) 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')
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 and annotate the rise time.

risetime(x);
Rise Time with 20% and 80% Reference Levels

Determine the rise time in a 2.3 V clock waveform sampled at 4 MHz. Compute the rise time using the 20% and 80% reference levels.

Load the 2.3 V clock data with sampling instants. Determine the rise time using the 20% and 80% reference levels. Plot the annotated waveform.

```matlab
load('transitionex.mat','x','t')
risetime(x,'PercentReferenceLevels',[20 80])
```
Rise Time, Reference-Level Instants, and Reference Levels

Determine the rise time, reference-level instants, and reference levels in a 2.3 V clock waveform sampled at 4 MHz.

Load the 2.3 V clock waveform along with the sampling instants.

load('transitionex.mat','x','t')

Determine the rise time, reference-level instants, and reference levels.
[R,lt,ut,ll,ul] = risetime(x,t);

Plot the waveform 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 in as a positive real integer in Hz.

**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 comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

**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 1-687 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 1-687 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 rise times are in seconds. If you do not specify a sample rate or sample instants, 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 lower 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. \textit{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.

\textbf{UL — Upper reference level}

real numeric scalar

Upper reference-level in waveform amplitude units, returned as a real numeric scalar. \textit{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 10\%. You can change the default reference levels by specifying the 'PercentReferenceLevels' name-value pair.

\section*{More About}

\subsection*{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**

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the $\alpha\%$ region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1),$$

where $S_1$ is the low-state level and $S_2$ is the high-state level. Replace the first term in the equation with $S_2$ to obtain the $\alpha\%$ tolerance region for the high state.

The following figure illustrates lower and upper 2\% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.
References


See Also
falltime | slewrate | statelevels

Introduced in R2012a
rlevinson

Reverse Levinson-Durbin recursion

Syntax

$r = \text{rlevenson}(a,\text{efinal})$
$[r,u] = \text{rlevenson}(a,\text{efinal})$
$[r,u,k] = \text{rlevenson}(a,\text{efinal})$
$[r,u,k,e] = \text{rlevenson}(a,\text{efinal})$

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 $r(i)$.

\[
\begin{bmatrix}
  r(1) & r(2)^* & \cdots & r(p)^* \\
  r(2) & r(1) & \cdots & r(p-1)^* \\
  \vdots & \ddots & \ddots & \vdots \\
  r(p) & \cdots & r(2) & r(1)
\end{bmatrix}
\begin{bmatrix}
  a(2) \\
  a(3) \\
  \vdots \\
  a(p+1)
\end{bmatrix}
= \begin{bmatrix}
  -r(2) \\
  -r(3) \\
  \vdots \\
  -r(p+1)
\end{bmatrix}
\]

$r = \text{rlevenson}(a,\text{efinal})$ 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. $e_{\text{final}}$ is the scalar prediction error power, which is equal to the variance of the prediction error signal, $\sigma^2(e)$.

$$[r,u] = rlevinson(a,e_{\text{final}})$$

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.
where $a_i(j)$ is the $j$th coefficient of the $i$th order prediction filter polynomial (i.e., step $i$ in the recursion). For example, the 5th order prediction filter polynomial is

$$a_5 = u(5:-1:1,5)'.$$

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 $u$.

$$k = \text{conj}(u(1,2:\text{end}))$$

$$[r,u,k,e] = rlevinson(a,efinal)$$

returns a vector of length $p + 1$ containing the prediction errors from each iteration of the reverse Levinson-Durbin recursion: $e(1)$ 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 $0.2^2$.

$$\begin{align*}
Fs &= 1000; \\
t &= (0:50e3-1)'/Fs; \\
x &= \sin(2\pi\times50\times t) + \sin(2\pi\times55\times t) + 0.2\times\text{randn}(50e3,1);
\end{align*}$$

Estimate the autoregressive model parameters.

$$\begin{align*}
[a,e] &= \text{arcov}(x,100); \\
[r,u,k] &= \text{rlevinson}(a,e);
\end{align*}$$

Estimate the power spectral density for orders 1, 5, 25, 50, and 100.

$$\begin{align*}
N &= [1 \ 5 \ 25 \ 50 \ 100]; \\
nFFT &= 8096; \\
P &= \text{zeros}(nFFT,5);
end
\end{align*}$$

$$\begin{align*}
\text{for } idx &= 1:\text{numel}(N) \\
\text{order} &= N(idx); \\
\text{ARtest} &= \text{flipud}(u(:,\text{order})); \\
P(:,idx) &= 1./\text{abs}(\text{fft(ARtest,nFFT)})^2;
\end{align*}$$

Plot the PSD estimates.

$$\begin{align*}
F &= (0:1/nFFT:1/2-1/nFFT)*Fs; \\
\text{plot}(F, 10\times\text{log10}(P(1:\text{length(\text{P})}/2,:))) \\
\text{grid}
\end{align*}$$

$$\begin{align*}
\text{legend}([\text{repmat('Order = ',}[5 \ 1]) \times\text{num2str}(N')]) \\
\text{xlabel('Frequency (Hz)')} \\
\text{ylabel('dB')} \\
\text{xlim([35 \ 70])}
\end{align*}$$
References

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
**rms**

Root-mean-square level

**Syntax**

\[
y = \text{rms}(x) \\
y = \text{rms}(x,\text{dim})
\]

**Description**

\(y = \text{rms}(x)\) returns the root-mean-square (RMS) level of the input, \(x\). If \(x\) is a row or column vector, \(y\) is a real-valued scalar. For matrices, \(y\) contains the RMS levels computed along the first array dimension of \(x\) with size greater than 1. For example, if \(x\) is an \(N\)-by-\(M\) matrix with \(N > 1\), then \(y\) is a 1-by-\(M\) row vector containing the RMS levels of the columns of \(x\).

\(y = \text{rms}(x,\text{dim})\) computes the RMS level of \(x\) along the dimension \(\text{dim}\).

**Examples**

**RMS Level of Sinusoid**

Compute the RMS level of a 100 Hz sinusoid sampled at 1 kHz.

\[
t = 0:0.001:1-0.001; \\
x = \cos(2\pi\times100\times t); \\
y = \text{rms}(x) \\
y = 0.7071
\]
**RMS Levels 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 RMS levels of the columns.

```matlab
t = 0:0.001:1-0.001;
x = cos(2*pi*100*t)'*(1:4);
y = rms(x)
y = 1x4
    0.7071    1.4142    2.1213    2.8284
```

**RMS Levels 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.

```matlab
t = 0:0.001:1-0.001;
x = (1:4)'*cos(2*pi*100*t);
y = rms(x,2)
y = 4x1
    0.7071
    1.4142
    2.1213
    2.8284
```
Input Arguments

**x — Input array**

vector | matrix | N-D array | gpuArray object

Input array, specified as a vector, matrix, N-D array, or gpuArray object. By default, `rms` acts along the first array dimension of `X` with size greater than 1.

See “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox) and “GPU Support by Release” (Parallel Computing Toolbox) for details on `gpuArray` objects.

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

**dim — Dimension along which to compute RMS levels**

integer scalar

Dimension along which to compute RMS levels, specified as an integer scalar.

Data Types: single | double

Output Arguments

**y — Root-mean-square level**

real-valued scalar | real-valued vector | real-valued N-D array

Root-mean-square level, returned as a real-valued scalar, vector, N-D array, or gpuArray object. If `x` is a vector, then `y` is a real-valued scalar. If `x` is a matrix, then `y` contains the RMS levels computed along dimension `dim`. By default, `dim` is the first array dimension of `x` with size greater than 1.

More About

**Root-Mean-Square Level**

The root-mean-square level of a vector `x` is
\[ x_{RMS} = \sqrt{\frac{1}{N} \sum_{n=1}^{N} |x_n|^2}, \]

with the summation performed along the specified dimension.

**References**


**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” (MATLAB).

**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**

- mean
- peak2peak
- peak2rms
- rssq
- std

**Introduced in R2012a**
**rooteig**

**Frequency and power content using eigenvector method**

**Syntax**

```
[w,pow] = rooteig(x,p)
[w,pow] = rooteig(___,'corr')
[f,pow] = rooteig(___,'corr','corr')
[f,pow] = rooteig(___,'corr',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(___,'corr',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.

```matlab
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)
```

\[ W = \begin{bmatrix} 0.7883 \\ 1.5674 \\ 1.0429 \end{bmatrix} \]

\[ P = \begin{bmatrix} 4.1748 \\ 1.0572 \\ 1.2419 \end{bmatrix} \]

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 \( x' \times x \) is an estimate of the correlation matrix.

  For complex-valued input data \( x \), \( \text{pow} \) and \( w \) have the same length. For real-valued input data \( x \), the length of the corresponding power vector \( \text{pow} \) is \( 0.5 \times \text{length}(w) \).

  **Note** You can use the output of \texttt{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 $\lambda_{\text{min}}$, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold $\lambda_{\text{min}}*p(2)$ 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.

 See Also

corrmtx | peig | pmusic | rootmusic

Introduced before R2006a
rootmusic

Root MUSIC algorithm

Syntax

\[ w = \text{rootmusic}(x,p) \]
\[ \left[w, pow\right] = \text{rootmusic}(x,p) \]
\[ \left[w, pow\right] = \text{rootmusic}(\_\_,'corr') \]
\[ \left[f, pow\right] = \text{rootmusic}(\_\_,'fs') \]

Description

\( w = \text{rootmusic}(x,p) \) 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.

\[ \left[w, pow\right] = \text{rootmusic}(x,p) \] returns the vector of frequencies \( w \) and the corresponding signal power in the vector \( pow \).

\[ \left[w, pow\right] = \text{rootmusic}(\_\_,'corr') \] 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 \( p \).

\[ \left[f, pow\right] = \text{rootmusic}(\_\_,'fs') \] 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.

```matlab
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 \( x \), \( \text{pow} \) and \( w \) have the same length. For real-valued input data \( x \), the length of the corresponding power vector \( \text{pow} \) is \( 0.5 \times \text{length}(w) \).

**Note** You can use the output of \( \text{corrmtx} \) 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 \( \lambda_{\text{min}} \), the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold \( \lambda_{\text{min}} \times p(2) \) 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.

\( f_s \) — Sample rate

1 (default) | positive scalar | []

Sample rate, specified as a positive scalar. You can supply the sample rate \( f_s \) in Hz. If you specify \( f_s \) 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.

\( \text{pow} \) — 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 $f_s$ in Hz. If you specify $f_s$ 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.

**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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
plot(t,rpm)
xlabel('Time (s)')
ylabel('RPM')
```
Compute the frequency-RPM map. Specify a resolution bandwidth of 2.5 Hz.

\[ \text{[map, freq, rpmOut, time]} = \text{rpmfreqmap(vib, fs, rpm, 2.5)}; \]

Visualize the map.

\text{imagesc(time, freq, map)}
\text{ax = gca;}\n\text{ax.YDir} = \text{'normal'};
\text{xlabel('Time (s)')}\n\text{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.

```matlab
rpmfreqmap(vib,fs,rpm,1.5);
```

![Frequency Map](image)
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.

\[
\begin{align*}
fs &= 600; \\
t_1 &= 15; \\
t &= 0:1/fs:t_1; \\
f_0 &= 10; \\
f_1 &= 40; \\
rpm &= 60*linspace(f_0,f_1,length(t));
\end{align*}
\]

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.

\[
\begin{align*}
o_1 &= 1; \\
o_2 &= 2.5; \\
o_6 &= 6; \\
x &= 0.5*\text{chirp}(t,o_1*f_0,t_1,o_1*f_1)+\text{chirp}(t,o_2*f_0,t_1,o_2*f_1) + ... \\
&\quad 0.8*\text{chirp}(t,o_6*f_0,t_1,o_6*f_1,\text{'quadratic'});
\end{align*}
\]

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.

\[
\begin{align*}
[\text{map},fr,rp] &= \text{rpmfreqmap}(x,fs,rpm,6, ... \\
&\quad \text{'Amplitude','peak','Window','flattopwin'});
\end{align*}
\]

Draw the frequency-RPM map as a waterfall plot.

\[
\begin{align*}
[FR,RP] &= \text{meshgrid}(fr,rp); \\
\text{waterfall}(FR,RP,map'); \\
\text{view}(-6,60) \\
\text{xlabel}('Frequency (Hz)') \\
\text{ylabel}('RPM') \\
\text{zlabel}('Amplitude')
\end{align*}
\]
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: \( \cos\left(\frac{\pi}{4} \times (0:159)\right) + \text{randn}(1,160) \) 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*

Rotational speeds, specified as a vector of positive values expressed in revolutions per minute. \( \text{rpm} \) must have the same length as \( x \).

- If you have a tachometer pulse signal, use \text{tachorpm} to extract \( \text{rpm} \) directly.
- If you do not have a tachometer pulse signal, use \text{rpmtrack} to extract \( \text{rpm} \) 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

**res — Resolution bandwidth**

\( \frac{\text{fs}}{128} \) (default) | positive scalar

Resolution bandwidth of the frequency-RPM map, specified as a positive scalar. If \( \text{res} \) is not specified, then \text{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 comma-separated pairs of \text{Name,Value} arguments. \text{Name} is the argument name and \text{Value} is the corresponding value. \text{Name} must appear inside quotes.
You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

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**


**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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
plot(t,rpm)
xlabel('Time (s)')
ylabel('RPM')
```
Compute the order-RPM map. Specify an order resolution of 0.015.

\[
[\text{map}, \text{order}, \text{rpmOut}, \text{time}] = \text{rpmordermap}(\text{vib}, \text{fs}, \text{rpm}, 0.015);
\]

Visualize the map.

\[
\text{imagesc}(\text{time}, \text{order}, \text{map})
\]
\[
\text{ax} = \text{gca};
\]
\[
\text{ax.YDir} = \text{'normal'};
\]
\[
\text{xlabel}('\text{Time (s)}')
\]
\[
\text{ylabel}('\text{Order}')
\]

Repeat the computation using a finer order resolution. Plot the map using the built-in functionality of \text{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.

```matlab
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.

```matlab
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.

```matlab
[map,or,rp] = rpmordermap(x,fs,rpm,0.25, ... 
   'Amplitude','peak','Window',{"chebwin",80});
```

Draw the order-RPM map as a waterfall plot.

```matlab
[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.

```matlab
load helidata.mat
rpmordermap(detrend(vib),fs,rpm,0.005)
```
See “Algorithms” on page 1-2198 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: \( \cos(\pi/4*(0:159)) + \text{randn}(1,160) \) 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

Rotational speeds, specified as a vector of positive values expressed in revolutions per minute. \( \text{rpm} \) must have the same length as \( x \).

- If you have a tachometer pulse signal, use \( \text{tachorpm} \) to extract \( \text{rpm} \) directly.
- If you do not have a tachometer pulse signal, use \( \text{rpmtrack} \) to extract \( \text{rpm} \) 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

**res — Order resolution**

\((15 \times \text{fs})/(16 \times \max(\text{rpm})) \) (default) | positive scalar

Order resolution of the order-RPM map, specified as a positive scalar. If \( \text{res} \) is not specified, then \( \text{rpmordermap} \) 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 comma-separated pairs of **Name, Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes.
You can specify several name and value pair arguments in any order as
Name1,Value1,...,NameN,ValueN.

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 1-2198 for more information.

**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, $\beta$. 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:

   \[ \phi(t) = \int_0^t \frac{\text{RPM}(\tau)}{60} d\tau. \]

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:

   \[ O_{\text{max}} = \frac{f_s/2}{\max(\text{RPM}/60)}. \]
To capture this highest order accurately, it is necessary to sample the signal at twice $O_{\text{max}}$ at least. For better results, \texttt{rpmordermap} oversamples by an extra factor of 4. The resulting phase-domain sample rate, $f_p$, is

$$f_p = 4 \times 2O_{\text{max}} = 4 \times 2 \frac{f_s/2}{\text{max}(\text{RPM}/60)}.$$  

The default order resolution, $r$, is

$$r = \frac{f_p}{256} = \frac{4 \times 60}{256} \frac{2 \times f_s/2}{\text{max}(\text{RPM})} = \frac{15}{16} \frac{f_s}{\text{max}(\text{RPM})}.$$  

4 Uses \texttt{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

$$N_{\text{overlap}} = \min\left(\left\lfloor \frac{p_{\text{overlap}}}{100} \times L \right\rfloor, L - 1 \right)$$

samples of overlap between adjoining segments, where $p_{\text{overlap}}$ is specified using the \texttt{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

$$r = \frac{k f_p}{L},$$

where $k$ is the equivalent noise bandwidth of the window, as implemented in \texttt{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 \texttt{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

See Also
orderspectrum | ordertrack | orderwaveform | rpmfreqmap | rpmtrack |
spectrogram | tachorpm

Introduced in R2015b
rpmtrack

Track and extract RPM profile from vibration signal

Syntax

```matlab
rpm = rpmtrack(x,fs,order,p)
rpm = rpmtrack(xt,order,p)
rpm = rpmtrack(___,'Name',Value)
[rpm,tout] = 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 pair 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.

```matlab
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.

```matlab
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.

```matlab
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)
Use the `rpmtrack` function to 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.

```matlab
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
  - Min: 0 sec
  - Median: 15 sec
  - Max: 30 sec
  - TimeStep: 0.001 sec

Variables:

- **rpm**: 30001x1 double
  - Min: 2.2204e-16
  - Median: 4327.2
  - Max: 8879.8

Plot the reconstructed RPM profile and the points used in the reconstruction.

```matlab
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.

```matlab
xrc = orderwaveform(vib,fs,rpmest.rpm,ol);
figure
```
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:

\[
\phi = 2\pi f_0 T \left(1 - e^{-t/T}\right),
\]

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 0.1².

\[
x = a*cos(phi) + randn(size(phi))/10; \\
y = a*sin(phi) + randn(size(phi))/10;
\]

plot(t,x,t,y)
Use the `rpmtrack` function to determine the RPM profile. Type

```matlab
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
% MATLAB Code from rpmtrack GUI
% Generated by MATLAB 9.4 and Signal Processing Toolbox 8.0
% Generated on 18-Dec-2017 19:00:35

% Set sample rate
fs = 1000.0000;
% Set order of ridge of interest
order = 1.0000;
% Set ridge points on ridge of interest
ridgePoints = [...
    0.4501 179.8246;...
```
% 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: \( \cos(\pi/4*(0:159))+\text{randn}(1,160) \) specifies a noisy sinusoid sampled at 2\( \pi \) 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” (MATLAB).

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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN.`
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 the comma-separated pair consisting of 'Method' and either 'stft' or 'fsst'.

- 'stft' — Use the short-time Fourier transform to compute a power spectrogram time-frequency map. See `pspectrum` for more details about the short-time Fourier transform.
- 'fsst' — Use the synchrosqueezed Fourier transform to compute a time-frequency map. See `fsst` for more details about the synchrosqueezed Fourier transform.

**FrequencyResolution — Frequency resolution bandwidth**
numeric scalar

Frequency resolution bandwidth used to compute the time-frequency map, specified as the comma-separated pair consisting of 'FrequencyResolution' and a numeric scalar expressed in Hz.

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 the comma-separated pair consisting of 'PowerPenalty' and a numeric scalar expressed in dB.

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 the comma-separated pair consisting of 'FrequencyPenalty' and 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 the comma-separated pair consisting of 'StartTime' and 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 the comma-separated pair consisting of 'EndTime' and 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


See Also

Functions
ordertrack | rpmfreqmap | rpmordermap | tachorpm

Introduced in R2018a
rssq

Root-sum-of-squares level

Syntax

\[ Y = \text{rssq}(X) \]
\[ Y = \text{rssq}(X,\text{DIM}) \]

Description

\( Y = \text{rssq}(X) \) returns the root-sum-of-squares (RSS) level, \( Y \), of the input, \( X \). If \( X \) is a row or column vector, \( Y \) is a real-valued scalar. For matrices, \( Y \) contains the RSS levels computed along the first nonsingleton dimension. For example, if \( Y \) is an \( N \)-by-\( M \) matrix with \( N > 1 \), \( Y \) is a 1-by-\( M \) row vector containing the RSS levels of the columns of \( Y \).

\( Y = \text{rssq}(X,\text{DIM}) \) computes the RSS level of \( X \) along the dimension, \( \text{DIM} \).

Input Arguments

\( X \)

Real- or complex-valued input vector or matrix. By default, \( \text{rssq} \) acts along the first nonsingleton dimension of \( X \).

\( \text{DIM} \)

Dimension for root-sum-of-squares (RSS) level. The optional \( \text{DIM} \) input argument specifies the dimension along which to compute the RSS level.

Default: First nonsingleton dimension
Output Arguments

Y

Root-sum-of-squares level. For vectors, Y is a real-valued scalar. For matrices, Y contains the RSS levels computed along the specified dimension, DIM. By default, DIM is the first nonsingleton dimension.

Examples

RSS Level of Sinusoid

Compute the RSS level of a 100 Hz sinusoid sampled at 1 kHz.

t = 0:0.001:1-0.001;
x = cos(2*pi*100*t);
y = rssq(x)
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 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 = rssq(x,2)
y = 4×1
    22.3607
    44.7214
    67.0820
    89.4427

More About

Root-Sum-of-Squares Level

The root-sum-of-squares (RSS) level of a vector, \( X \), is

\[
X_{RSS} = \sqrt{\sum_{n=1}^{N} |X_n|^2},
\]

with the summation performed along the specified dimension. The RSS is also referred to as the \( \ell^2 \) norm.
References


See Also
mean | peak2peak | peak2rms | rms | std

Introduced in R2012a
sawtooth

Sawtooth or triangle wave

Syntax

\[ x = \text{sawtooth}(t) \]
\[ x = \text{sawtooth}(t,x_{\text{max}}) \]

Description

\[ x = \text{sawtooth}(t) \] generates a sawtooth wave with period \(2\pi\) for the elements of the time array \(t\). \text{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 \(2\pi\) and to increase linearly with time with a slope of \(1/\pi\) at all other times.

\[ x = \text{sawtooth}(t,x_{\text{max}}) \] generates a modified triangle wave with the maximum location at each period controlled by \(x_{\text{max}}\). Set \(x_{\text{max}}\) 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.

\[ T = 10*(1/50); \]
\[ fs = 1000; \]
\[ t = 0:1/fs:T-1/fs; \]
\[ x = \text{sawtooth}(2*pi*50*t); \]
\[ \text{plot}(t,x) \]
\[ \text{grid on} \]
Plot the power spectrum of the wave.

\texttt{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.

\[
T = 10 \times \frac{1}{50};
\]

\[
fs = 1000;
\]

\[
t = 0:1/fs:T-1/fs;
\]

\[
x = \text{sawtooth}(2\pi \times 50 \times t, 1/2);
\]
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. \( \text{xmax} \) determines the point between 0 and \( 2\pi \) at which the wave reaches its maximum. The function increases from \(-1\) to 1 on the interval 0 to \( 2\pi \times \text{xmax} \), then decreases linearly from 1 to \(-1\) on the interval \( 2\pi \times \text{xmax} \) to \( 2\pi \). The shape then repeats with a period of \( 2\pi \).

Example: \( \text{xmax} = 0.5 \) specifies a standard triangle wave, symmetric about time \( \pi \) 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**

\[
k = \text{schurrc}(r)
\]

\[
[k,e] = \text{schurrc}(r)
\]

**Description**

\( k = \text{schurrc}(r) \) 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, \( \text{schurrc} \) 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 \( r \).

\( [k,e] = \text{schurrc}(r) \) 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 \texttt{mtlb.mat}. Use the Schur algorithm to compute the reflection coefficients of a lattice prediction filter for the sequence.

\[
\text{load } \texttt{mtlb}
\]

\[
r = \text{xcorr}(	exttt{mtlb}(1:5),'unbiased');
\]

\[
k = \text{schurrc}(r(5:end))
\]
k = 4×1

-0.7583
0.1384
0.7042
-0.3699

References


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, S, from the mid-reference level instant to
the time instant each transition enters and remains within a 2% tolerance region of the
final state over the duration, D. D is a positive scalar. Because settlingtime uses
interpolation to determine the mid-reference level instant, S may contain values that do
not correspond to sampling instants. The length of S is equal to the number of detected
transitions in the input signal, X. If for any transition, the level of the waveform does not
remain within the lower and upper tolerance boundaries, the requested duration is not
present, or an intervening transition is detected, settlingtime marks the
corresponding element in S as NaN. See “Settle Seek Duration” on page 1-2240 for cases
in which settlingtime returns a NaN. To determine the transitions, settlingtime
estimates the state levels of the input waveform by a histogram method. settlingtime
identifies all regions that cross the upper-state boundary of the low state and the lower-
state boundary of the high state. The low-state and high-state boundaries are expressed
as the state level plus or minus a multiple of the difference between the state levels. See
“State-Level Tolerances” on page 1-2239.

S = settlingtime(X,FS,D) specifies the sample rate for the bilevel waveform, X in
hertz. The first sample instant in X is equal to \( t = 0 \). Because settlingtime uses
interpolation to determine the mid-reference level instant, S may contain values that do
not correspond to sampling instants.
\( S = \text{settlingtime}(X, T, D) \) specifies the sample instants, \( T \), as a vector with the same number of elements as \( X \).

\([S, SLEV, SINST] = \text{settlingtime}(...)\) returns vectors, \( SLEV \), and \( SINST \), whose elements correspond to the levels and sample instants of the settling points for each transition.

\([S, SLEV, SINST] = \text{settlingtime}(..., \text{Name}, \text{Value})\) returns the settling times, levels, and corresponding sample instants with additional options specified by one or more \( \text{Name}, \text{Value} \) pair arguments.

\( \text{settlingtime}(...) \) plots the signal and darkens the regions of each transition where settling time is computed. The plot marks the location of the settling time of each transition, the mid-crossings, and the associated reference levels. The plot also displays the state levels with the corresponding lower and upper tolerance boundaries.

**Input Arguments**

\( X \)

Bilevel waveform. \( X \) is a real-valued row or column vector.

\( D \)

Settle-seek duration. \( D \) is a positive scalar, which defines the duration after the mid-reference level instant that \( \text{settlingtime} \) looks for a settling time. If no settling time occurs in \( D \) seconds after the mid-reference level instant, \( \text{settlingtime} \) returns a \( \text{NaN} \). See “Settling Time” on page 1-2238 and “Settle Seek Duration” on page 1-2240.

\( FS \)

Sample rate in hertz.

\( T \)

Vector of sample instants. The length of \( T \) must equal the length of the bilevel waveform, \( X \).
Name-Value Pair Arguments

MidPercentReferenceLevel

Mid-reference level as a percentage of the waveform amplitude. See “Mid-Reference Level” on page 1-2238.

Default: 50

StateLevels

Low and high-state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low and high-state levels, settlingtime estimates the state levels from the input waveform using the histogram method.

Tolerance

Tolerance levels (lower and upper state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-2239.

Default: 2

Output Arguments

S

The time from the mid-reference level instant to the time instant each transition enters and remains within a 2% tolerance region of the final state over duration, D.

SLEV

Waveform values at the settling points.

SINST

Time instants of the settling points.

Examples
Determine Settling Point and Settling Level

Determine the settling point and corresponding waveform value for a bilevel waveform. Specify a settle-seek duration of 10 seconds.

```matlab
load('transitionex.mat', 'x')
[s,slev,sinst] = settlingtime(x,10);
```

Plot the waveform and annotate the settling point.

```matlab
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.

```matlab
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)
```
\[ \text{ans} = \begin{bmatrix} 10^{-6} \\ 0.4725 \\ 0.1181 \\ 0.4725 \end{bmatrix} \]
More About

Settling Time

The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the 2%-tolerance region around the state level. The settling time is illustrated in the following figure. The low- and high-state levels are the dashed black lines. The 2% tolerances above and below the state levels are shown by the red dashed lines and the settling time is indicated by the yellow circle.

Mid-Reference Level

The mid-reference level in a bilevel waveform with low-state level, $S_1$, and high-state level, $S_2$, is

$$S_1 + \frac{1}{2}(S_2 - S_1)$$

Mid-Reference Level Instant

Let $y_{50\%}$ denote the mid reference level.
Let $t_{50\%}$ and $t_{50\%}$, denote the two consecutive sampling instants corresponding to the waveform values nearest in value to $y_{50\%}$.

Let $y_{50\%}$ and $y_{50\%}$, denote the waveform values at $t_{50\%}$ and $t_{50\%}$.

The mid-reference level instant is

$$t_{50\%} = t_{50\%} + \left( \frac{t_{50\%} - t_{50\%}}{y_{50\%} - y_{50\%}} \right) (y_{50\%} - y_{50\%})$$

**State-Level Tolerances**

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the $\alpha\%$ region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100} (S_2 - S_1),$$

where $S_1$ is the low-state level and $S_2$ is the high-state level. Replace the first term in the equation with $S_2$ to obtain the $\alpha\%$ tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.
Settle Seek Duration

The settle seek duration defines the interval of time after the mid-reference level instant that settlingtime looks for a settling point. If settlingtime does not find a settling point within the settle seek duration, settlingtime returns NaN for the settling time. The following figure illustrates a settle seek duration of 10 samples.
settlingtime may fail to find a settling point in the specified settle seek duration if any one of the following conditions occurs:

- The last waveform value in the settle seek interval is not within the upper- and lower-state boundaries determined by the specified tolerance. The following figure illustrates this condition for a settle seek duration of 8 samples and a 2% tolerance region.
In the preceding figure, you see that the last sample in the settle seek interval exceeds the upper state boundary. In this example, reducing or increasing the settle seek duration can result in a valid settling time.

- There is an insufficient number of waveform samples for the specified settle seek duration. The following figure illustrates this condition for a settle seek duration of 20 samples. The settle seek duration extends beyond the final sample of the waveform.
• An intervening transition is detected before the end of the specified settle seek duration. The following figure illustrates this condition for a settle seek duration of 22 samples. An intervening transition is detected before the end of the 22-sample settle seek duration.
References


See Also
falltime | midcross | pulsewidth | risetime | statelevels

Introduced in R2012a
seqperiod

Compute period of sequence

Syntax

p = seqperiod(x)
[p,num] = seqperiod(x)

Description

p = seqperiod(x) returns the integer p that corresponds to the period of the sequence in a vector x. The period p is computed as the minimum length of a subsequence x(1:p) of x that repeats itself continuously every p samples in x. The length of x does not have to be a multiple of p, so that an incomplete repetition is permitted at the end of x. If the sequence x is not periodic, then p = length(x).

- If x is a matrix, then seqperiod checks for periodicity along each column of x. The resulting output p is a row vector with the same number of columns as x.
- If x is a multidimensional array, then seqperiod checks for periodicity along the first nonsingleton dimension of x. In this case:
  - p is a multidimensional array of integers with a leading singleton dimension.
  - The lengths of the remaining dimensions of p correspond to those of the dimensions of x after the first nonsingleton one.

[p,num] = seqperiod(x) also returns the number num of repetitions of x(1:p) in x. num might not be an integer.

Examples

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.

Introduced before R2006a
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.

```matlab
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.

```matlab
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.

```matlab
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.

\[ \text{lbldefs} = \text{getLabelValues}(\text{lss}) \]

\[
\begin{array}{cccc}
\text{WhaleType} & \text{MoanRegions} & \text{TrillRegions} & \text{Maximum} \\
\hline
\text{Member}\{1\} & \text{blue} & \{3\times2 \text{ table}\} & \{1\times3 \text{ table}\} & \{[0.2850]\} \\
\text{Member}\{2\} & \text{blue} & \{3\times2 \text{ table}\} & \{1\times3 \text{ table}\} & \{[0.3791]\}
\end{array}
\]

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'.

- **lss** - Labeled signal set
  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.

**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.

\textbf{sridx — Sublabel row index}

\texttt{positive integer}

Sublabel row index, specified as a positive integer. This argument applies only when a label and sublabel pair has been specified in \texttt{lblname} and the sublabel is of type ROI or point.

\textbf{See Also}

\texttt{Signal Labeler | labeledSignalSet | signalLabelDefinition}

\texttt{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 idx.

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 array
   "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” table of a labeled signal set.
See Also
Signal Labeler | labeledSignalSet | signalLabelDefinition

Introduced in R2019a
sfdr

Spurious free dynamic range

Syntax

\[
\begin{align*}
  r &= \text{sfdr}(x) \\
  r &= \text{sfdr}(x, fs) \\
  r &= \text{sfdr}(x, fs, msd) \\
  r &= \text{sfdr}(sxx, f, 'power') \\
  r &= \text{sfdr}(sxx, f, msd, 'power') \\
  [r, spurpow, spurfreq] &= \text{sfdr}(\_\_\_) \\
  \text{sfdr}(\_\_) 
\end{align*}
\]

Description

\(r = \text{sfdr}(x)\) returns the spurious free dynamic range (SFDR), \(r\), in dB of the real sinusoidal signal, \(x\). \text{sfdr} computes the power spectrum using a modified periodogram and a Kaiser window with \(\beta = 38\). The mean is subtracted from \(x\) before computing the power spectrum. The number of points used in the computation of the discrete Fourier transform (DFT) is the same as the length of the signal, \(x\).

\(r = \text{sfdr}(x, fs)\) 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.

\(r = \text{sfdr}(x, fs, msd)\) 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.

\(r = \text{sfdr}(sxx, f, 'power')\) 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 $3.16 \times 10^{-4}$.

deltat = 1e-8;
fs = 1/deltat;
t = 0:deltat:1e-5-deltat;
x = cos(2*pi*10e6*t)+3.16e-4*cos(2*pi*20e6*t);
r = sfdr(x,fs)

r = 70.0063

Display the spectrum of the signal. Annotate the fundamental, the DC value, the spur, and the SFDR.

sfdr(x,fs);
Minimum Spur Distance

Obtain the SFDR for a 10 MHz tone with amplitude 1 sampled at 100 MHz. There is a spur at the 1st harmonic (20 MHz) with an amplitude of $3.16 \times 10^{-4}$ and another spur at 25 MHz with an amplitude of $10^{-5}$. Skip the first harmonic by using a minimum spur distance of 11 MHz.

```matlab
deltat = 1e-8;
fs = 1/deltat;
t = 0:deltat:1e-5-deltat;
```
\[
x = \cos(2\pi \times 10^6 t) + 3.16 \times 10^{-4} \cos(2\pi \times 20^6 t) + \ldots
\]
\[
0.1 \times 10^{-5} \cos(2\pi \times 25^6 t);
\]
\[
r = \text{sfdr}(x, fs, 11^6);
\]

\[
r = 120.0000
\]

Display the spectrum of the signal. Annotate the fundamental, the DC value, the spurs, and the SFDR.

\[
\text{sfdr}(x, fs, 11^6);
\]
**SFDR from Periodogram**

Obtain the power spectrum of a 10 MHz tone with amplitude 1 sampled at 100 MHz. There is a spur at the 1st harmonic (20 MHz) with an amplitude of $3.16 \times 10^{-4}$. Use the one-sided power spectrum and a vector of corresponding frequencies in Hz to compute the SFDR.

```matlab
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.

```matlab
sfdr(sxx,f,'power');```
**Frequency and Power of Largest Spur**

Determine the frequency in MHz for the largest spur. The input signal is a 10 MHz tone with amplitude 1 sampled at 100 MHz. There is a spur at the first harmonic (20 MHz) with an amplitude of $3.16 \times 10^{-4}$.

```matlab
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
```
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`

\( \text{spurpow} \) — power of largest spur  
real-valued scalar

Power in dB of the largest spur, specified as a real-valued scalar.

Data Types: `double`

\( \text{spurfreq} \) — 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, \( \text{sfdr} \) 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, \( \text{sfdr} \) performs a periodogram using a Kaiser window with large sidelobe attenuation. To find the fundamental frequency, the algorithm searches the periodogram for the largest nonzero spectral component. It then computes the central
moment of all adjacent bins that decrease monotonically away from the maximum. To be
detectable, the fundamental should be at least in the second frequency bin. If a harmonic
lies within the monotonically decreasing region in the neighborhood of another, its power
is considered to belong to the larger harmonic. This larger harmonic may or may not be
the fundamental. The algorithm ignores all the power that decreases monotonically away
from the DC bin.

sfdr fails if the fundamental is not the highest spectral component in the signal.

Ensure that the frequency components are far enough apart to accommodate for the
sidelobe width of the Kaiser window. If this is not feasible, you can use the 'power' flag
and compute a periodogram with a different window.

See Also
bandpower | enbw | periodogram

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.

```matlab
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 4th-order polynomials.
order = 4;
framel en = 21;

b = sgolay(order,framel en);

Compute the steady-state portion of the signal by convolving it with the center row of b.

yc enter = conv(x,b((framel en+1)/2,:),'valid');

Compute the transients. Use the last rows of b for the startup and the first rows of b for the terminal.

yb egin = b(end:-1:(framel en+3)/2,:) * x(framelen:-1:1);
yend = b((framel en-1)/2:-1:1,:) * x(end:-1:end-(framel en-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 5th-order polynomials. Divide the columns by powers of $dt$ to scale the derivatives correctly.

```matlab
[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.

```matlab
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.

**framelen** — Frame length
positive odd integer

Frame length, specified as a positive odd integer. The value of `framelen` must be greater than `order`. If `order=framelen-1`, then the designed filter produces no smoothing.
weights — Weighing vector
real positive vector

Weighing vector, specified as a real positive vector. The weighing vector has the same length as framelen and are 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-framel en matrix. In a smoothing filter implementation (for example, sgolayfilt), the last (framel en-1)/2 rows (each an FIR filter) are applied to the signal during the startup transient, and the first (framel en-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 \( p^{\text{th}} \) order derivative, \( x_p \), of its middle value from \( x_p((\text{framen}+1)/2) = (\text{factorial}(p)) \times g(:,p+1)' \times x \).

Algorithms

Savitzky-Golay smoothing filters (also called digital smoothing polynomial filters or least squares smoothing filters) are typically used to “smooth out” a noisy signal whose frequency span (without noise) is large. In this type of application, Savitzky-Golay smoothing filters perform much better than standard averaging FIR filters, which tend to filter out a significant portion of the signal's high frequency content along with the noise.

You can implement data smoothing to measure a variable that is both slowly varying and also corrupted by random noise. Since nearby points measure nearly the same underlying value, you can replace each data point by a local average of the surrounding data points. Savitzky-Golay filters are optimal in the sense that they minimize the least-squares error in fitting a polynomial to each frame of noisy data.
References


Extended Capabilities

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

See Also
filter | fir1 | firls | sgolayfilt

Introduced before R2006a
sgolayfilt

Savitzky-Golay filtering

Syntax

\[
y = \text{sgolayfilt}(x, \text{order}, \text{framelen}) \\
y = \text{sgolayfilt}(x, \text{order}, \text{framelen}, \text{weights}) \\
y = \text{sgolayfilt}(x, \text{order}, \text{framelen}, \text{weights}, \text{dim})
\]

Description

\( y = \text{sgolayfilt}(x, \text{order}, \text{framelen}) \) applies a Savitzky-Golay finite impulse response (FIR) smoothing filter of polynomial order \( \text{order} \) and frame length \( \text{framelen} \) to the data in vector \( x \). If \( x \) is a matrix, then \text{sgolayfilt} \) operates on each column.

\( y = \text{sgolayfilt}(x, \text{order}, \text{framelen}, \text{weights}) \) specifies a weighting vector to use during the least-squares minimization.

\( y = \text{sgolayfilt}(x, \text{order}, \text{framelen}, \text{weights}, \text{dim}) \) specifies the dimension along which the filter operates.

Examples

Steady-State and Transient Savitzky-Golay Filters

Generate a random signal and smooth it using \text{sgolayfilt}. Specify a polynomial order of 3 and a frame length of 11. Plot the original and smoothed signals.

\[
\begin{align*}
\text{order} &= 3; \\
\text{framelen} &= 11; \\
\text{lx} &= 34; \\
x &= \text{randn}(%lx,1); \\
y &= \text{sgolayfilt}(x, \text{order}, \text{framelen}, \text{weights}) \\
\end{align*}
\]
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.

```matlab
sgf = sgolayfilt(x,order,framelen);
plot(x,':')
hold on
plot(sgf,'.-')
legend('signal','sgolay')
```

```matlab
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 \((\text{framelen}-1)/2\) rows of \(B\) by the first \(\text{framelen}\) samples of the signal.

\[ y_{beg} = B(1:m,:) \times x(1:\text{framelen}); \]
To determine the terminal transient, matrix multiply the final \((\text{framelen}-1)/2\) rows of \(B\) by the final \text{framelen} samples of the signal.

\[
y_{\text{end}} = B(\text{framelen}-m+1: \text{framelen}, :) \times x(\text{lx}-\text{framelen}+1: \text{lx})\;
\]

Concatenate the transients and the steady-state portion to generate the complete signal.

```matlab
cmplt = steady;
cmplt(1:m) = ybeg;
cmplt(lx-m+1:lx) = yend;

plot(cmplt)
legend('signal', 'sgolay', 'steady', 'complete')
hold off
```

![Graph with legend](image)
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®."

```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.

```matlab
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.

```matlab
kmtlb = sgolayfilt(mtlb, rd, fl, kaiser(fl, 38));
```

```matlab
subplot(2,1,2)
hold on
plot(t, kmtlb)
axis([0.2 0.22 -3 2])
hold off
```
Input Arguments

**x** — Input signal
type: vector | matrix

Input signal, specified as a vector or matrix.

Data Types: single | double

**order** — Polynomial order
type: positive integer

Data Type: single | double
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.

References


Extended Capabilities

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

See Also
filter | medfilt1 | sgolay | sosfilt

Introduced before R2006a
**shiftdata**

Shift data to operate on specified dimension

**Syntax**

\[ [x, \text{perm}, \text{nshifts}] = \text{shiftdata}(x, \text{dim}) \]

**Description**

\[ [x, \text{perm}, \text{nshifts}] = \text{shiftdata}(x, \text{dim}) \] shifts data \( x \) to permute dimension \( \text{dim} \) to the first column using the same permutation as the built-in \text{filter} function. The vector \( \text{perm} \) returns the permutation vector that is used.

If \( \text{dim} \) is missing or empty, then the first nonsingleton dimension is shifted to the first column, and the number of shifts is returned in \( \text{nshifts} \).

\text{shiftdata} is meant to be used in tandem with \text{unshiftdata}, which shifts the data back to its original shape. These functions are useful for creating functions that work along a certain dimension, like \text{filter}, \text{goertzel}, \text{sgolayfilt}, and \text{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. \text{unshiftdata} shifts \( x \) back to its original shape.

Create a 3-by-3 magic square.

\( x = \text{magic}(3) \)

\[
\begin{array}{ccc}
8 & 1 & 6 \\
\end{array}
\]
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.

$$[x, perm, nshifts] = \text{shiftdata}(x, 2)$$

$x = \begin{bmatrix} 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$

$perm = \begin{bmatrix} 2 & 1 \end{bmatrix}$

$nshifts = [\ ]$

Shift the matrix back to its original shape.

$y = \text{unshiftdata}(x, perm, nshifts)$

$y = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$

**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.
Define \texttt{dim} as empty to shift the first nonsingleton dimension of \texttt{x} to the first column. \texttt{shiftdata} returns \texttt{x} as a column vector, along with \texttt{perm}, the permutation vector, and \texttt{nshifts}, the number of shifts.

\[ [x, \texttt{perm}, \texttt{nshifts}] = \texttt{shiftdata}(x,[]) \]

\texttt{x} = 5×1

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\texttt{perm} =

\[
[ ]
\]

\texttt{nshifts} = 1

Using \texttt{unshiftdata}, restore \texttt{x} to its original shape.

\texttt{y} = \texttt{unshiftdata}(x,\texttt{perm},\texttt{nshifts})

\texttt{y} = 1×5

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\textbf{See Also}

\texttt{permute} | \texttt{shiftdim} | \texttt{unshiftdata}

\textbf{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, duplicate, extract, and rename 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
• Label signals interactively for analysis or for machine learning and deep learning applications

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.

You need a Wavelet Toolbox™ license to use the scalogram view.

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**

Implement a basic digital music synthesizer and use it to play a traditional song in a three-voice arrangement. Specify a sample rate of 2 kHz. Save the song as a MATLAB® timetable.

```matlab
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 0 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 2 0 2 2 3 0 5 5 3 0 5 0 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)
tune = timetable(seconds((0:length(song)-1)'/fs),song);
```

Open **Signal Analyzer** and drag the timetable from the Workspace browser to the Signal table. Click **Display Grid ▼** to create a two-by-two grid of displays. Select the top two displays and the lower left display and click the **Spectrum** button to add a spectrum view. Select the lower right display, click **Time-Frequency** to add a spectrogram view, and click **Time** to remove the time view. Drag the song to all four displays. Select the lower right display, and in the **Spectrogram** tab, specify a time resolution of 0.31 second (310 ms) and 0% overlap between adjoining segments. Set the **Power Limits** to −50 dB and −10 dB.
On the **Analyzer** tab, click **Duplicate** three times to create three copies of the song. Rename the copies as high, medium, and low by double-clicking the **Name** column in the Signal table. Move the copies to the top two and lower left displays.

Preprocess the duplicate signals using filters.

1. Select the high signal by clicking its name in the Signal table. On the **Analyzer** tab, click **Highpass**. On the **Highpass** tab that appears, enter a passband frequency of 450 Hz and increase the steepness to 0.95. Click **Highpass**.

2. Select the medium signal by clicking its name in the Signal table. On the **Analyzer** tab, click **Preprocessing ▼** and select **Bandpass**. On the **Bandpass** tab that appears, enter 230 Hz and 450 Hz as the lower and upper passband frequencies, respectively. Increase the steepness to 0.95. Click **Bandpass**.

3. Select the low signal by clicking its name in the Signal table. On the **Analyzer** tab, click **Lowpass**. On the **Lowpass** tab that appears, enter a passband frequency of 230 Hz and increase the steepness to 0.95. Click **Lowpass**.
On each of the three displays containing filtered signals:

1. Remove the original signal by clearing the check box next to its name.
2. On the Display tab, click Time-Frequency to add a spectrogram view and click Time to remove the time view.
3. On the Spectrogram tab, specify a time resolution of 0.31 second and 0% overlap between adjoining segments. Set the Power Limits to −50 dB and −10 dB.
Select the three filtered signals by clicking their Name column in the Signal table. On the Analyzer tab, click Export and save the signals to a MAT-file called music.mat. In MATLAB, load the file to the workspace. Plot the spectra of the three signals.

```matlab
load music
pspectrum(low)
hold on
pspectrum(medium)
pspectrum(high)
hold off
```
Modulation and Demodulation Using Complex Envelope

This example simulates the different steps of a basic communication process. Communication systems work by modulating chunks of information into a higher carrier frequency, transmitting the modulated signals through a noisy physical channel, receiving the noisy waveforms, and demodulating the received signals to reconstruct the initial information.
All the information carried in a real-valued signal \( s(t) \) can be represented by a corresponding lowpass complex envelope:

\[
 s(t) = \text{Re}\left\{g(t)e^{j2\pi f_c t}\right\} = i(t)\cos(2\pi f_c t) + q(t)\sin(2\pi f_c t). 
\]

In this equation:

- \( f_c \) is the carrier frequency.
- \( \text{Re} \) represents the real part of a complex-valued quantity.
- \( g(t) = i(t) + jq(t) \) is the complex envelope of \( s(t) \).
- \( i(t) \) is the inphase component of the complex envelope.
- \( q(t) \) is the quadrature component of the complex envelope.

The complex envelope is modulated to the carrier frequency and sent through the channel. At the receiver, the noisy waveform is demodulated using the carrier frequency. The phase variation due to the carrier frequency is predictable and thus does not convey any information. The complex envelope does not include the phase variation and can be sampled at a lower rate.

Generate a signal whose complex envelope consists of a sinusoid and a chirp. The inphase component is a sinusoid with a frequency of 19 Hz. The quadrature component is a quadratic chirp whose frequency ranges from 61 Hz to 603 Hz. The signal is sampled at 2 kHz for 1 second.

```matlab
fs = 2e3;
t = (0:1/fs:1-1/fs)';
inph = sin(2*pi*19*t);
quad = chirp(t-0.6,61,t(end),603,'quadratic');

env = inph + 1j*quad;
g = timetable(env,'SampleRate',fs);
```

Open Signal Analyzer and drag the complex envelope from the Workspace browser to the Signal table. The display shows the inphase and quadrature components of the envelope as lines of the same hue and saturation, but different luminosity. The first line color represents the inphase component and the second line color represents the quadrature component. Click Spectrum ▼ on the Display tab and select Spectrum. The app displays a set of axes with the signal spectrum. The complex envelope has a two-sided
spectrum, displayed as a line of the same color of the inphase component of the complex envelope.

On the **Display** tab, click **Panner** to activate the panner. Use the panner to create a zoom window between 300 ms and 720 ms. Drag the zoom window so that it is centered at 0 Hz. The spectrum has an impulse at 0.19 kHz and a wider tapering profile at higher frequencies. The negative-frequency region of the spectrum is a mirror image of the positive-frequency region.
Modulate the signal using a carrier frequency of 200 Hz. Multiply by $\sqrt{2}$ so that the power of the modulated signal equals the power of the original signal. Add white Gaussian noise such that the signal-to-noise ratio is 40 dB.

$$fc = 200;$$
$$mod = \sqrt{2} \cdot \text{real(env.*exp(2j*pi*fc*t))};$$

$$SNR = 40;$$
$$mod = mod + \text{randn(size(mod))*std(mod)/db2mag(SNR)};$$
$$s = \text{timetable(mod,'SampleRate',fs);}$$

Click Display Grid ▼ to add a second display. Drag the modulated signal to the Signal table and enter the time information. The modulation has moved the spectrum to positive frequencies centered on the carrier frequency.
Calculate the analytic signal and demodulate the signal by multiplying the analytic signal with a complex-valued negative exponential of frequency 200 Hz.

\[
\text{dem} = \text{hilbert}(\text{mod}).*\exp(-2j\pi f_c t)/\sqrt{2}.
\]
Click **Display Grid ▼** to create a three-by-one grid of displays. Drag the demodulated signal to the Signal table. Add time information to the complex envelope by clicking **Time Values** in the **Analyzer** tab. The two-sided spectrum shows the recovered inphase and quadrature components of the baseband signal.

![Image of three-by-one display grid with data cursors and extracted signal](image)

Click **Display Grid ▼** to create a one-by-one grid of displays and plot the demodulated signal. Click **Data Cursors ▼** and select **Two**. Place the time-domain cursors at 300 ms and 900 ms, so they enclose the spectral peaks. Click **Extract Signals ▼** and select **Between Time Cursors**. Check the **Preserve Start Time** box. Clear the display and plot the extracted signal. The app extracts both inphase and quadrature components of the demodulated signal in the region of interest. Select the extracted signal by clicking its **Name** column in the Signal table. On the **Analyzer** tab, click **Export** and save the signal to a MAT-file called **dem_ROI.mat**.
Load the `dem_ROI` file to the workspace. Compute the demodulated inphase and quadrature components by taking the real and imaginary parts of the extracted signal. Store the time information of the extracted signal in a time variable `t_dem`.

```matlab
load dem_ROI
inph_dem = real(dem_ROI);
quad_dem = imag(dem_ROI);
t_dem = 0.3+(0:length(dem_ROI)-1)/fs;
```

Compare the transmitted waveforms and the extracted regions of interest. Also compare their spectra.

```matlab
subplot(2,1,1)
plot(t,inph,t_dem,inph_dem,'--')
legend('Transmitted Inphase Signal','Received Inphase Signal')
```

```matlab
subplot(2,1,2)
plot(t,quad,t_dem,quad_dem,'--')
legend('Transmitted Quadrature Signal','Received Quadrature Signal')
```
```matlab
figure
subplot(2,1,1)
psspectrum(inph,fs)
hold on
psspectrum(inph_dem,fs)
legend('Transmitted Inphase Signal','Received Inphase Signal')
hold off

subplot(2,1,2)
psspectrum(quad,fs)
hold on
psspectrum(quad_dem,fs)
legend('Transmitted Quadrature Signal','Received Quadrature Signal')
hold off
```
Find and Track Ridges Using Reassigned Spectrogram

Load a datafile containing an echolocation pulse emitted by a big brown bat (*Eptesicus fuscus*) and measured with a sampling interval of 7 microseconds. Create a MATLAB® timetable using the signal and the time information.

```matlab
load batsignal
t = (0:length(batsignal)-1)*DT;
sg = timetable(seconds(t)',batsignal);
```
Open **Signal Analyzer** and drag the timetable from the Workspace browser to the Signal table. Click **Display Grid ▼** to create two side-by-side displays. Select each display and click the **Time-Frequency** button to add a spectrogram view.

Drag the timetable to both displays.

Select the **Spectrogram** tab. On the display at right, check **Reassign**. For each display:

- Set the time resolution to 280 microseconds and specify 85% overlap between adjoining segments.
- Use the **Leakage** slider to increase the leakage until the RBW is about 4.5 kHz.
- Set the power limits to –45 dB and –20 dB.
The reassigned spectrogram clearly shows three time-frequency ridges. To track the ridges, select the display at right. On the Display tab, click Generate Script and select Spectrogram Script. The script appears in the Editor.

% Compute spectrogram
% Generated by MATLAB(R) 9.7 and Signal Processing Toolbox 8.2.
% Generated on: 26-Dec-2018 17:21:44

% Parameters
timeLimits = seconds([0 0.002793]); % seconds
frequencyLimits = [0 71428.57]; % Hz
leakage = 0.9;
timeResolution = 0.00028; % seconds
overlapPercent = 85;
reassignFlag = true;

%%
% Index into signal time region of interest
sg_batsignal_ROI = sg(:, 'batsignal');
sg_batsignal_ROI = sg_batsignal_ROI(timerange(timeLimits(1), timeLimits(2), 'closed'), 1);

% Compute spectral estimate
% Run the function call below without output arguments to plot the results
[P, F, T] = pspectrum(sg_batsignal_ROI, ...
    'spectrogram', ..., 
    'FrequencyLimits', frequencyLimits, ..., 
    'Leakage', leakage, ..., 
    'TimeResolution', timeResolution, ..., 
    'OverlapPercent', overlapPercent, ..., 
    'Reassign', reassignFlag);

Run the script. Plot the reassigned spectrogram.

mesh(seconds(T), F, P)
xlabel('Time')
ylabel('Frequency')
axis tight
view(2)
colormap pink
Use the `tfridge` function to track the ridges.

```matlab
[fridge,~,lridge] = tfridge(P,F,0.01,'NumRidges',3,'NumFrequencyBins',10);
hold on
plot3(seconds(T),fridge,P(lridge),':','linewidth',3)
hold off
```
Thanks to Curtis Condon, Ken White, and Al Feng of the Beckman Center at the University of Illinois for the bat data and permission to use it in this example.

**Declip Saturated Signals Using Your Own Function**

Sensors can return clipped readings if the data are larger than a given saturation point. To reconstruct the readings, you can fit a polynomial through the points adjacent to the saturated intervals. Write a function that performs the reconstruction and integrate it into Signal Analyzer.
Generate a three-channel signal sampled at 1 kHz for 14 seconds. The signal has several peaks of varying sizes and shapes. A sensor that reads the signal saturates at 0.1 V.

```matlab
fs = 1000;
t = 0:1/fs:14-1/fs;
sig = [chirp(t-1,0.1,17,'quadratic',1).*sin(2*pi*t/5);
    chirp(t-2,2,2.1,'quadratic',100).*exp(-(t-6.5).^2/20).*sin(2*pi*t*2);
    0.85*besselj(0,5*(sin(2*pi*(t+1.5).^2/20).^2)).*sin(2*pi*t/9)]';
sigsat = sig;
stv = 0.1;
sigsat(sigsat >= stv) = stv;
```

Open **Signal Analyzer** and drag the original signal and the saturated signal to the Signal table. Drag each original and saturated channel to its own display.

Write a function that uses a polynomial to reconstruct the signal peaks:
• The first input argument, \( x \), is the input signal. This argument must be a vector and is treated as a single channel.
• The second input argument, \( t_{\text{In}} \), is a vector of time values. The vector must have the same length as the signal. If the input signal has no time information, the function reads this argument as an empty array.
• Use `varargin` to specify additional input arguments. If you do not have additional input arguments, you can omit `varargin`. Enter the additional arguments as an ordered comma-separated list in the `Preprocess` tab.
• The first output argument, \( y \), is the preprocessed signal.
• The second output argument, \( t_{\text{Out}} \), is a vector of output time values. If the input signal has no time information, \( t_{\text{Out}} \) is returned as an empty array.
• To implement your algorithm, you can use any MATLAB® or Signal Processing Toolbox™ function.

```matlab
function [y,tOut] = declip(x,tIn,varargin)
% Declip saturated signal by fitting a polynomial

% Initialize the output signal
y = x;

% For signals with no time information, use sample numbers as abscissas
if isempty(tIn)
    tOut = [];
    t = (1:length(x))';
else
    t = tIn;
    tOut = t;
end

% Specify the degree of the polynomial as an optional input argument
% and provide a default value of 4
if nargin<3
    ndx = 4;
else
    ndx = varargin{1};
end

% To implement your algorithm, you can use any MATLAB or Signal Processing Toolbox function
```
% Find the intervals where the signal is saturated and generate an
% array containing the interval endpoints
idx = find(x==max(x));
fir = [true;diff(idx)==1];
ide = [idx(fir) idx(fir([2:end 1]))];
% For each interval, fit a polynomial of degree ndx over the ndx+1 points
% before the interval and the ndx+1 points after the interval
for k = 1:size(ide,1)
    bef = ide(k,1); aft = ide(k,2);
    intv = [bef-1:(-ndx:0) aft+1:(ndx)];
    [pp,~,mu] = polyfit(t(intv),x(intv),ndx);
    y(bef:aft) = polyval(pp,t(bef:aft),[],mu);
end

end

Add the function to Signal Analyzer as a custom preprocessing function. On the
Analyzer tab, click Preprocessing ▼ and select Add Custom Function. Input the
function name and description. Paste the text of your function in the editor window that
appears. Save the file. The function appears in the preprocessing gallery.
Demonstrate that the function you created reconstructs the saturated regions.

1. Select the first channel of the saturated signal in the Signal table.
2. On the Analyzer tab, click **Preprocessing ▼** and select **declip**.
3. On the **Preprocessing** tab that appears, click **Preprocess**.

Verify that the preprocessing function works when the signals have time information.

1. Select **sig** and **sigsat** in the Signal table. Do not select individual channels.
2. On the Analyzer tab, click **Time Values**, select **Sample Rate and Start Time**, and specify $f_s$ as the sample rate.
3. The signal in the top panel, including the reconstructed regions, has time information.
Check that the function works when you specify optional inputs.

1 Select the second and third channels of the saturated signal in the Signal table.
2 On the **Preprocessing** tab, enter 8 in the **Arguments** field and click **Preprocess**. The preprocessing function uses a polynomial of degree 8 to reconstruct the saturated regions.
Compute Envelope Spectrum of Vibration Signal

Use Signal Analyzer to compute the envelope spectrum of a bearing vibration signal and look for defects. Generate MATLAB® scripts and functions to automate the analysis.

Generate Bearing Vibration Data

A bearing with the dimensions shown in the figure is driven at \( f_0 = 25 \) cycles per second. An accelerometer samples the bearing vibrations at 10 kHz.
Generate vibration signals from two defective bearings using the `bearingdata` function at the end of the example. In one of the signals, `xBPFO`, the bearing has a defect in the outer race. In the other signal, `xBPFI`, the bearing has a defect in the inner race. For more details on modeling and diagnosing defects in bearings, see “Vibration Analysis of Rotating Machinery” and `envspectrum`.

```matlab
[t,xBPFO,xBPFI,bpfi] = bearingdata;
```

**Compute Envelope Spectrum Using Signal Analyzer**

Open **Signal Analyzer** and drag the BPFO signal to a display. Add time information to the signal by selecting it in the Signal table and clicking the **Time Values** button on the **Analyzer** tab. Select the **Sample Rate** and **Start Time** option and enter the 10 kHz sample rate.
On the **Display** tab, click **Spectrum** to open a spectrum view. The spectrum of the vibration signal shows BPFO harmonics modulated by the 3 kHz impact frequency. At the low end of the spectrum, the driving frequency and its orders obscure other features.

Select the signal and, on the **Analyzer** tab, click **Duplicate** to generate a copy of it. Give the new signal the name *envspec* and drag it to the display. Compute the envelope spectrum of the signal using the Hilbert transform:

1. Remove the DC value of the signal. On the **Analyzer** tab, click **Preprocessing ▼** and select **Detrend**. On the **Detrend** tab that appears, select **Constant** as the method. Click **Detrend**.

2. Bandpass-filter the detrended signal. On the **Analyzer** tab, click **Preprocessing ▼** and select **Bandpass**. On the **Bandpass** tab that appears, enter 2250 Hz and 3750 Hz as the lower and upper passband frequencies, respectively. Click **Bandpass**.
3 Compute the envelope of the filtered signal. On the Analyzer tab, click Preprocessing ▼ and select Envelope. On the Envelope tab that appears, select Hilbert as the method. Click Envelope.

4 Remove the DC value of the envelope using Detrend.

The envelope spectrum appears in the spectrum view of the display. The envelope spectrum clearly displays the BPFO harmonics.

**Steps to Create an Integrated Analysis Script**

The computation of the envelope spectrum can get tedious if it has to be repeated for many different bearings. Signal Analyzer can generate MATLAB® scripts and functions to help you automate the computation.
As an exercise, repeat the previous analysis for the BPFI signal. **Signal Analyzer** generates two components useful for the automation:

1. A function that preprocesses the signal by detrending it, filtering it, and computing its envelope
2. A script that computes the envelope spectrum

To create the integrated analysis script, put the preprocessing function and the plotting script together unchanged in a single file. (Alternatively, you can save functions in separate files.)

- If you save the script and the function in a single MATLAB® script, keep in mind that functions must appear at the end.
- You must add the keyword `end` at the end of each function.

### 1. Create Preprocessing Function

Initially, create the function that reproduces the preprocessing steps. Select the `envspec` signal. On the **Analyzer** tab, click **Generate Function**. The function, called `preprocess` by default, appears in the Editor. Save the generated function at the end of your integrated analysis script. The function expects a second argument specifying the time information. Preprocess the BPFI signal using the function.

```matlab
eenvspec = preprocess(xBPFI,t);
```

### 2. Create Spectrum Script

In the app, remove the unprocessed signal from the display by clearing the check box next to its name. On the **Display** tab, click **Generate Script ▼** and select **Spectrum Script**. The script appears in the Editor. Include the generated code in your integrated analysis script. When you run the analysis script, the generated spectrum script computes the envelope spectrum of the preprocessed BPFI signal.

```matlab
% Compute power spectrum

% Generated by MATLAB(R) 9.6 and Signal Processing Toolbox 8.2.
% Generated on: 12-Nov-2018 15:13:34

% Parameters
timeLimits = [0 0.9999]; % seconds
frequencyLimits = [0 5000]; % Hz
```
%%
% Index into signal time region of interest
envspec_ROI = envspec(:);
sampleRate = 10000; % Hz
startTime = 0; % seconds
minIdx = ceil(max((timeLimits(1)-startTime)*sampleRate,0))+1;
maxIdx = floor(min((timeLimits(2)-startTime)*sampleRate,length(envspec_ROI)-1))+1;
envspec_ROI = envspec_ROI(minIdx:maxIdx);

% Compute spectral estimate
% Run the function call below without output arguments to plot the results
[Penvspec_ROI, Fenvspec_ROI] = pspectrum(envspec_ROI,sampleRate, ...
    'FrequencyLimits',frequencyLimits);

3. Plot Envelope Spectrum

Plot the envelope spectrum. Compare the peak locations to the frequencies of the first ten BPFI harmonics. You can also plot the envelope spectrum using the pspectrum command with no output arguments.

plot(Fenvspec_ROI,(Penvspec_ROI))
hold on
[X,Y] = meshgrid((1:10)*bpfi,ylim);
plot(X,Y,':k')
hold off
xlim([0 10*bpfi])
Function Code

Signal Preprocessing Function

The signal preprocessing function generated by the app combines detrending, bandpass filtering, and envelope computation.

```matlab
function y = preprocess(x,tx)
% Preprocess input x
% This function expects an input vector x and a vector of time values
% tx. tx is a numeric vector in units of seconds.

% Generated by MATLAB(R) 9.6 and Signal Processing Toolbox 8.2.
% Generated on: 12-Nov-2018 15:09:44
```
```
y = detrend(x,'constant');
Fs = 1/mean(diff(tx)); % Average sample rate
y = bandpass(y,[2250 3750],Fs,'Steepness',0.85,'StopbandAttenuation',60);
[y,~] = envelope(y);
y = detrend(y,'constant');
end

Bearing Data Generating Function

The bearing has pitch diameter \( p = 12 \) cm and a bearing contact angle \( \theta = 0 \). Each of the \( n = 8 \) rolling elements has a diameter \( d = 2 \) cm. The outer race remains stationary as the inner race is driven at \( f_0 = 25 \) cycles per second. An accelerometer samples the bearing vibrations at 10 kHz.

```function [t,xBPFO,xBPFI,bpfi] = bearingdata

p = 0.12;
d = 0.02;
n = 8;
th = 0;
f0 = 25;
fs = 10000;
```

For a healthy bearing, the vibration signal is a superposition of several orders of the driving frequency, embedded in white Gaussian noise.

```matlab
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);
xHealthy = z + randn(size(z))/10;
```

A defect in the outer race causes a series of 5 millisecond impacts that over time result in bearing wear. The impacts occur at the ball pass frequency outer race (BPFO) of the bearing,

\[
BPFO = \frac{1}{2}nf_0\left[1 - \frac{d}{p}\cos \theta \right].
\]

Model the impacts as a periodic train of 3 kHz exponentially damped sinusoids. Add the impacts to the healthy signal to generate the BPFO vibration signal.

```matlab
bpfo = n*f0/2*(1-d/p*cos(th));
```
tmp = 0:1/fs:5e-3:1/fs;
xmp = sin(2*pi*3000*tmp).*exp(-1000*tmp);

xBPFO = xHealthy + pulstran(t,0:1/bpfo:1,xmp,fs)/4;

If the defect is instead in the inner race, the impacts occur at a frequency

\[ \text{BPFI} = \frac{1}{2} n f_0 \left[ 1 + \frac{d}{p} \cos \theta \right]. \]

Generate the BPFI vibration signal by adding the impacts to the healthy signals.

bpfi = n*f0/2*(1+d/p*cos(th));

xBPFI = xHealthy + pulstran(t,0:1/bpfi:1,xmp,fs)/4;

end

• “Find Delay Between Correlated Signals”
• “Plot Signals from the Command Line”
• “Resolve Tones by Varying Window Leakage”
• “Analyze Signals with Inherent Time Information”
• “Spectrogram View of Dial Tone Signal”
• “Find Interference Using Persistence Spectrum”
• “Scalogram of Hyperbolic Chirp”
• “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.
See Also

Apps
Filter Designer | Signal Labeler

Functions
periodogram | pspectrum | pwelch | spectrogram

Topics
“Find Delay Between Correlated Signals”
“Plot Signals from the Command Line”
“Resolve Tones by Varying Window Leakage”
“Analyze Signals with Inherent Time Information”
“Spectrogram View of Dial Tone Signal”
“Find Interference Using Persistence Spectrum”
“Scalogram of Hyperbolic Chirp”
“Resample and Filter a Nonuniformly Sampled Signal”
“Extract Regions of Interest from Whale Song”
“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” property set to `name` and other properties set to default values.

`sld = signalLabelDefinition(name,Name,Value)` sets “Properties” using name-value pairs. You can specify multiple name-value pairs. 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
class 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'

Type of label, specified as one of the following:

- 'attribute' — Define signal characteristics.
- 'roi' — Define signal characteristics over regions of interest.
- 'point' — Define signal characteristics over points of interest.

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'. Use the “Categories” property to specify the array of categories when this property is set to 'categorical'.

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” 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.

**Note**  Sublabels cannot have sublabels.

Example: 'Sublabels',
[signalLabelDefinition("negative"),signalLabelDefinition("positive")]

**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. 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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
load labelwhalesignals
lbldefs = [dWhaleType dMoans dTrills];
```
lss = labeledSignalSet({whale1 whale2},lbldefs,...
    'SampleRate',Fs,'Description','Characterize wave 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    DefaultValue             Sublabels             Tag            Description
    ____________    ___________    _____________    ____________    __________________    ____________    ___________________________    ___    ____________________________
   "WhaleType"       "attribute"    "categorical"    {3x1 string}       {"["N/A""]}       {0x0 double}    {0x0 double               }    ""     "Whale type"
   "MoanRegions"     "roi"          "logical"        {"["N/A""]}       {0x0 double}       {0x0 double}    {0x0 double               }    ""     "Regions where moans occur"
   "TrillRegions"    "roi"          "logical"        {"["N/A""]}       {0x0 double}       {0x0 double}    {1x1 signalLabelDefinition}    ""     "Regions where trills occur"

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
    ___________    ___________    ___________
   Member{1}      blue        {0x2 table}    {0x3 table}
Visualize Regions and Points

Visualize the whale songs to identify the trill and moan regions.

```matlab
subplot(2,1,1)
plot((0:length(whale1)-1)/Fs,whale1)
ylabel('Whale 1')
hold on

subplot(2,1,2)
plot((0:length(whale2)-1)/Fs,whale2)
ylabel('Whale 2')
hold on
```
Add the moan and trill regions to the labeled set. For ROI labels, specify the ROI limits in seconds and the label values. Label the different regions in the plots using an auxiliary function.

```matlab
moanRegionsWhale1 = [6.1 7.7; 11.4 13.1; 16.5 18.1];
setLabelValue(lss,1,'MoanRegions',moanRegionsWhale1,[true true true]);

trillRegionWhale1 = [1.4 3.1];
setLabelValue(lss,1,'TrillRegions',trillRegionWhale1,true);

subplot(2,1,1)
labelIntervals([moanRegionsWhale1;trillRegionWhale1])

moanRegionsWhale2 = [2.5 3.5; 5.8 8; 15.4 16.7];
```
setLabelValue(lss,2,'MoanRegions',moanRegionsWhale2,[true true true]);

trillRegionWhale2 = [11.1 13];
setLabelValue(lss,2,'TrillRegions',trillRegionWhale2,true);

subplot(2,1,2)
labelIntervals([moanRegionsWhale2;trillRegionWhale2])

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`.

```matlab
getLabelValues(lss)
ans=2×3 table
   WhaleType    MoanRegions    TrillRegions
  ___________    ___________    ____________
 Member{1}      blue       {3x2 table}    {1x3 table}
 Member{2}      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
________    _____    ___________
   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]}  

function labelIntervals(moansTrills)
% Auxiliary function to label moan and trill regions in plots
    [X,Y] = meshgrid(moansTrills,ylim);
    plot(X,Y,'k:');
    topts = {'HorizontalAlignment','center','FontWeight','bold', ...  
    'FontSize',12,'Color',[139 69 19]/255};
    text((X(1,1:4)+X(1,5:end))/2,Y(2,5:end)-0.1, ...  
    ['moan' 'moan' 'moan' 'trill'],topts{:})
end

See Also
Signal Labeler | labeledSignalSet

Introduced in R2018b
Signal Labeler

Label signals for analysis or machine and deep learning applications

Description

**Signal Labeler** 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 signals with attributes, regions, and points of interest
- Use logical, categorical, numerical, or string-valued labels
- Automatically label signal peaks or apply custom labeling functions
- Add, edit, and delete labels or sublabels
- Display selected subsets of signals and labels

**Signal Labeler** saves data to **Signal Analyzer** as labeledSignalSet objects. You can use labeledSignalSet objects to train a network, classifier, or analyze data and report statistics.

Open the Signal Labeler

In the **Signal Analyzer** app, on the **Analyzer** tab, click **Label**.

Examples

**Label Signal Attributes, Regions of Interest, and Points**

Recordings of whale songs contain 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 a few selected signal peaks.
Load Unlabeled Data

Start by loading a data set that includes two recordings of whale songs. The signals are called \texttt{whale1} and \texttt{whale2} and are sampled at 4 kHz. \texttt{whale1} consists of a trill followed by three moans. \texttt{whale2} consists of two moans, a trill, and another moan.

load labelwhalesignals

% To hear, type soundsc(whale1,Fs), pause(22), soundsc(whale2,Fs)

Bring the signals into \textbf{Signal Labeler}:

1. Start Signal Analyzer and drag the signals to the Signal table.
2. Select the signals and add time information. On the \textbf{Analyzer} tab, click \textbf{Time Values}, select \textbf{Sample Rate} and \textbf{Start Time}, and enter the sample rate, \texttt{Fs}.
3. With the signals selected, click \textbf{Label} on the \textbf{Analyzer} tab.

Add Signal Label Definitions

Define labels to attach to the signals. Labels can be of three types:

- \textit{Attribute} labels define signal characteristics.
- \textit{Region-of-interest} (ROI) labels define signal characteristics over regions of interest.
- \textit{Point} labels define signal characteristics over points of interest.

Each label can have one of four data types:

- \textit{Logical} labels can be either true or false.
- \textit{Categorical} labels can belong to any one of a set of categories that you specify.
- \textit{Numeric} labels can have any numeric value.
- \textit{String} labels can have any value represented by a string.

Any label can have any number of \textit{sublabels}. Sublabels themselves cannot have sublabels.

For the whale song signals:

1. Define a categorical attribute label to store whale types. Call it \texttt{WhaleType}. The possible categories are blue whale, humpback whale, and white whale.
2. Define a logical region-of-interest (ROI) label that is true for moan regions. Call it \texttt{MoanRegions}.
Define a logical ROI label that is true for trill regions. Call it TrillRegions.

Define a numeric point label to capture trill peaks. Call it TrillPeaks. Set this label as a sublabel of the TrillRegions label.

To define each label, click Add Definition on the Label tab. To define the sublabel, select the TrillRegions label in the Label Definitions browser, click Add Definition ▼, and select Add sublabel definition.

Enter the following values in the fields in the dialog box that appears for each signal label or sublabel definition. Leave the Default field empty in each case.

<table>
<thead>
<tr>
<th>Label Name</th>
<th>Label Type</th>
<th>Label Description</th>
<th>Data Type</th>
<th>Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>WhaleType</td>
<td>Attribute</td>
<td>Whale type</td>
<td>categorical</td>
<td>blue, humpback, white</td>
</tr>
<tr>
<td>MoanRegions</td>
<td>ROI</td>
<td>Regions where moans occur</td>
<td>logical</td>
<td>- - -</td>
</tr>
<tr>
<td>TrillRegions</td>
<td>ROI</td>
<td>Regions where trills occur</td>
<td>logical</td>
<td>- - -</td>
</tr>
<tr>
<td>TrillPeaks</td>
<td>Point</td>
<td>Trill peaks</td>
<td>numeric</td>
<td>- - -</td>
</tr>
</tbody>
</table>

You can export the signal definitions you created to a MAT-file by clicking Export. A dialog box appears that prompts you for a file name. At any point you can import signal definitions stored in a MAT-file by clicking Import.

**Label Signal Attributes**

The songs in the data are from two blue whales. Set the WhaleType values for both signals:

1. Select WhaleType on the Label Definitions browser.
2. Click Label ▼ and select Label Signals.
3. In the dialog box that appears, verify that both whale1 and whale2 are selected and that the Value field is set to blue. (If you do not specify a default value in a categorical signal label definition, Signal Labeler sets the label to the first category specified in the definition.)
4. Click OK.

Plot the whale1 signal by selecting the check box next to its name. Signal attributes appear both in the Labeled Signal Set browser and under the time plot.
Label Signal Regions

Visualize the whale songs and label the trill and moan regions.

- Trill regions have distinct bursts of sound punctuated by silence. `whale1` has a trill centered at about 2 seconds.
- Moan regions are sustained low-frequency wails. `whale1` has moans centered at about 7 seconds, 12 seconds, and 17 seconds.

Label the signals one at a time:

1. On the **Plot** column of the **Labeled Signal Set** browser, check the box next to the signal name to plot the signal.
To label a moan, on the **Label Definitions** browser, select the **MoanRegions** label definition.

Click **Label ▼** and select **Label Plotted**. A shaded region appears, framed by an animated dashed line. (The animated frame indicates the region is active.) Move and resize the active region until it encloses a moan region. For better label placement, you can go to the **Display** tab and choose a zoom action or activate the panner.

Click the check mark next to the **Label** button, press **Enter**, or double-click to label the ROI. The region changes to a gradient of the signal color. If you do not specify a default value in a logical label definition, **Signal Labeler** sets the label to **true**.

Repeat the procedure for the other two moans.

To label a trill, on the **Label Definitions** browser, select the **TrillRegions** label definition. Label the trill region using steps 3 and 4.

Before labeling the second whale song signal, remove the first whale song signal from the plot by clearing the check box next to its name in the **Labeled Signal Set** browser. If you have the two signals plotted when you label a region or point, **Signal Labeler** associates the label with both signals.

The label viewer axes show the locations and widths of the regions of interest. They also show the value assigned to each region.
Label Signal Points

Trill regions have distinct peaks that correspond to bursts of sound. Label three peaks in each trill region. Because trill peaks are sublabels, each one must be associated with a particular TrillRegions label.

Label the signals one at a time:

1. On the Plot column of the Labeled Signal Set browser, check the box next to the signal name to plot the signal. Also check the box corresponding to the trill region whose peaks you want to label.
2. On the Label Definitions browser, select TrillPeaks.
3. On the toolstrip, under Value, enter 1, corresponding to the first peak.
4 On the **Labeled Signal Set** browser, select the trill region. The trill region becomes active and is framed by an animated dashed line.

5 Click **Label ▼** and select **Label Plotted**. The trill region is framed by a solid line, and an animated dashed (active) line appears for the point being labeled.

6 Move the active line until it crosses the signal at a peak of your choice. For better label placement, you can go to the **Display** tab and choose a zoom action or activate the panner.

7 Click the check mark next to the **Label** button, press **Enter**, or double-click to label the peak. The dashed line changes to a solid line of the same color as the signal.

8 Repeat for two more peaks, entering 2 and 3 to identify them.

9 Before labeling trill peaks for the second whale song signal, remove the first whale song signal from the plot by clearing the check box next to its name in the **Labeled Signal Set** browser.

The label viewer axes show the locations of the points of interest and the value assigned to each point.

Plot the two signals to see a summary of their labels in the **Label Viewer**. Expand the labeled signal set hierarchy in the **Labeled Signal Set** browser to see details for all the labels. (To expand the hierarchy, right-click any signal in the browser and select **Expand All**.) For each signal, plot the first moan region and the third trill peak that you labeled.
Edit Signal Label Values

At any point, you can edit any signal label using the **Labeled Signal Set** browser. To edit an attribute label, select it, right-click, and select **Edit**. For example, if you discover that the second whale is actually a white whale, you can select the **WhaleType** attribute for **whale2**, right-click, select **Edit**, and, on the dialog box that appears, select **white** from the drop-down menu.

If you want to edit the value of an ROI label or a point label, you can modify the value in the dialog box. To modify the location of an ROI label or a point label, you can modify the location fields in the dialog box. Alternatively, you can:

1. Plot the label by checking the box next to its name.
2 Select the label.
3 Move the region or point in the time plot when it becomes active.

**Export Labeled Signal Set**

Export labeled signals by saving your labeling and exporting the new `labeledSignalSet` object. Click the **Save Labels** button on the toolstrip. In the dialog box that appears, give the name `whalesongs` to the labeled signal set. Clicking the **OK** button gets you back to **Signal Analyzer**. See “Signal Labeler Import and Export Behavior” for more information on how **Signal Labeler** exports labeled signal sets.
On the Signal table, select `whalesongs` and right-click to export it to a file called `Whale_Songs.mat`.

### Examine Labeled Signal Set

Load into the MATLAB® workspace the MAT-file you created in the “Label Signal Attributes, Regions of Interest, and Points” example. Verify that the labeled signal set contains the definitions that you added using **Signal Labeler**.

```matlab
load Whale_Songs

labelDefinitionsSummary(whalesongs)
```

```
ans=3×9 table
LabelName      LabelType    LabelDataType    Categories     ValidationFunction    DefaultValue             Sublabels             Tag            Description
______________    ___________    _____________    ____________    __________________    ____________    ___________________________    ___    ____________________________
"WhaleType"       "attribute"    "categorical"    {3x1 string}       {["N/A"    ]}       {0x0 double}    {0x0 double               }    ""     "Whale type" "MoanRegions"     "roi"          "logical"        {["N/A"    ]}       {0x0 double}    {0x0 double               }    ""     "Regions where moans occur" "TrillRegions"    "roi"          "logical"        {["N/A"    ]}       {0x0 double}    {1x1 signalLabelDefinition}    ""     "Regions where trills occur"
```

Verify that `TrillPeaks` is a sublabel of `TrillRegions`.

```matlab
labelDefinitionsHierarchy(whalesongs)
```

```
ans =
'aWhaleType
   Sublabels: []
'MoanRegions
   Sublabels: []
'TrillRegions
   Sublabels: TrillPeaks'
```

Retrieve the second member of the set. Retrieve the names of the timetable variables.

```matlab
song = getSignal(whalesongs,2);

summary(song)
```

RowTimes:
Time: 76579x1 duration
Values:
  Min          0 sec
  Median       9.5722 sec
  Max          19.144 sec
  TimeStep     0.00025 sec

Variables:

  whale2: 76579x1 double

Values:
  Min          -0.37326
  Median       0
  Max          0.37914

Plot the signal.

t = song.Time;
sng = song.whale2;

plot(t,sng)
Visualize Labeled Regions

Display and identify the regions of interest that you labeled. For more details, see the code for the `labelIntervals` function at the end of the example.

```matlab
mvals = getLabelValues(whalesongs,2,'MoanRegions');
tvals = getLabelValues(whalesongs,2,'TrillRegions');

cmap = lines;
hold on

tmoan = mvals.ROILimits;
for kj = 1:size(tmoan,1)
```

1-2347
tv = find(seconds(t)>tmoan(kj,1) & seconds(t)<tmoan(kj,2));
plot(t(tv),sng(tv),'Color',cmap(2,:))
end

ttrill = tvals.ROIlimits;
for kj = 1:size(ttrill,1)
    tv = find(seconds(t)>ttrill(kj,1) & seconds(t)<ttrill(kj,2));
    plot(t(tv),sng(tv),'Color',cmap(3,:))
end

labelIntervals(mvals,tvals,cmap(4,:))

hold off
**Visualize Labeled Points**

Display and identify the trill peaks that you labeled.

```matlab
pk = getLabelValues(whalesongs,2,{'TrillRegions','TrillPeaks'});

locs = zeros(size(pk,1),1);
for kj = 1:length(locs)
    locs(kj) = find(seconds(t) == pk.Location(kj));
end

hold on
plot(t(locs),sng(locs)+0.01,'v','MarkerSize',8,'Color',[0.929,0.694,0.125])
text(t(locs)+seconds(0.2),sng(locs)+0.05,int2str(cell2mat(pk.Value)), ...    'HorizontalAlignment','center')
hold off
```
This helper function displays and identifies regions of interest.

```matlab
function labelIntervals(mvals,tvals,clr)
    [X,Y] = meshgrid(seconds([mvals.ROILimits;tvals.ROILimits]),ylim);
    plot(X,Y,':k')
    topts = {'HorizontalAlignment','center','FontWeight','bold', ...
             'FontSize',12,'Color',clr};
    text((X(1,1:4)+X(1,5:end))/2,Y(2,5:end)-0.1, ... 
         ['moan' 'moan' 'moan' 'trill'],topts{:})
end
```
Label QRS Complexes and R Peaks of ECG Signals Using Deep Network

This example shows how to use custom automated labeling functions in Signal Labeler to label QRS complexes and R peaks of electrocardiogram (ECG) signals. One custom function uses a previously trained recurrent deep learning network to identify and locate the QRS complexes. Another custom function uses a simple peak finder to locate the R peaks. In the example, the network labels the QRS complexes of two signals that are completely independent of the network training and testing process.

The QRS complex, which consists of three deflections in the ECG waveform, reflects the depolarization of the right and left ventricles of the heart. The QRS is also the highest-amplitude segment of the human heartbeat. Study of the QRS complex can help assess the overall health of a person's heart and the presence of abnormalities. In particular, by locating R peaks within the QRS complexes and looking at the time intervals between consecutive peaks, a diagnostician can compute the heart-rate variability of a patient and detect cardiac arrhythmia.

The deep learning network in this example was introduced in “Waveform Segmentation Using Deep Learning”, where it was trained using ECG signals from the publicly available QT Database. The data consists of roughly 15 minutes of ECG recordings from a total of 105 patients, sampled at 250 Hz. To obtain each recording, the examiners placed two electrodes on different locations on a patient's chest, which resulted in a two-channel signal. The database provides signal region labels generated by an automated expert system. The added labels make it possible to use the data to train a deep network. See “Waveform Segmentation Using Deep Learning” for more details.

Load, Resample, and Import Data into Signal Labeler

The signals labeled in this example are from the MIT-BIH Arrhythmia Database. Each signal in the database was sampled at 360 Hz and was annotated by two cardiologists, allowing for verification of the results.

Load two of the MIT database signals, corresponding to records 200 and 203. Resample the signals to 250 Hz, the sample rate of the QT Database data.

```matlab
load mit200
y200 = resample(ecgsig,25,36);

load mit203
y203 = resample(ecgsig,25,36);
```
Start **Signal Analyzer** and drag the signals to the Signal table. Select the signals. Add time information: on the **Analyzer** tab, click **Time Values**, select **Sample Rate** and **Start Time**, and specify a sample rate of 250 Hz. On the **Analyzer** tab, click **Label**. The signals appear in the **Labeled Signal Set** browser.

**Define Labels**

Define labels to attach to the signals.

1. Define a categorical region-of-interest (ROI) label for the QRS complexes. Click **Add Definition** on the **Label** tab. Specify the **Label Name** as **QRSregions**, select a **LabelType** of **ROI**, enter the **Data Type** as categorical, and add two **Categories**, **QRS** and **n/a**, each on its own line.

2. Define a numerical point label for the R peaks and set it as a sublabel of **QRSregions**. Click **QRSregions** in the **Label Definitions** browser to select it. Click **Add Definition ▼** and select **Add sublabel definition**. Specify the **Label Name** as **Rpeaks**, select a **LabelType** of **Point**, and enter the **Data Type** as numeric.
Create Custom Autolabeling Functions

Create two custom functions, one to locate and label the QRS complexes and another to locate and label the R peak within each QRS complex. (Code for the findQRS function on page 1-0, computeFSST on page 1-0, p2qrs on page 1-0, and findRpeaks on page 1-0 functions appears later in the example.) To create each function, in the Analyzer tab, click Automate Value ▼ and select Add Custom Function. Signal Labeler shows a dialog box asking for the name, description, and label type of the function to add.

1 For the function that locates the QRS complexes, enter findQRS in the Name field and select ROI as the Label Type. You can leave the Description field empty or you can enter your own description.
2 For the function that locates the R peaks, enter `findRpeaks` in the **Name** field and select **Point** as the **Label Type**. You can leave the **Description** field empty or you can enter your own description.

If you already have written the functions, and the functions are in the current folder or in the MATLAB® path, **Signal Labeler** adds the functions to the gallery. If you have not written the functions, **Signal Labeler** opens blank templates in the Editor for you to type or paste the code. Save the files. The functions appear in the gallery.

---

**Label QRS Complexes and R Peaks**

Find and label the QRS complexes of the input signals.

1 In the **Labeled Signal Set** browser, select the check box next to `y200`. 
2 Select QRS regions in the Label Definitions browser.
3 On the Automate Value gallery, select findQRS.
4 Click Auto-Label and click OK in the dialog box that appears.

Signal Labeler locates and labels the QRS complexes for all signals but displays only those of the signal whose check box you selected. The QRS complexes appear as shaded regions in the plot and in the label viewer axes. Activate the panner by clicking Panner on the Display tab and zoom in on a region of the labeled signal.

Find and label the R peaks corresponding to the QRS complexes.
1 Select Rpeaks in the Label Definitions browser.
2 Go back to the Label tab. On the Automate Value gallery, select findRpeaks.
3 Click **Auto-Label** and click **OK** in the dialog box that appears.

The labels and their numeric values appear in the plot and in the label viewer axes.

**Export Labeled Signals and Compute Heart-Rate Variability**

Export the labeled signals to compare the heart-rate variability for each patient. On the **Label** tab, click **Save Labels**. In the dialog box that appears, give the name `heartrates` to the labeled signal set. Click **OK** to return to **Signal Analyzer**. In the Signal table, select `heartrates` and right-click to export it to a file called `HeartRates.mat`.

Load the labeled signal set. For each signal in the set, compute the heart-rate variability as the standard deviation of the time differences between consecutive heartbeats. Plot a histogram of differences and display the heart-rate variability.
load HeartRates

nms = getMemberNames(heartrates);

for k = 1:heartrates.NumMembers
    v = getLabelValues(heartrates,k,{'QRSregions','Rpeaks'});
    hr = diff(cellfun(@(x)x.Location,v));

    subplot(2,1,k)
    histogram(hr,0.5:.025:1.5)
    legend([\'\textit{hrv = ' num2str(std(hr))\}])
    ylabel(nms(k))
    ylim([0 6])
end
findQRS Function: Find QRS Complexes

The findQRS function finds and labels the QRS complexes of the input signals.

The function uses two auxiliary functions, computeFSST and p2qrs. (Code for both auxiliary functions appears later in the example.) You can either store the functions in separate files in the same directory or nest them inside findQRS by inserting them before the final end statement.

Between calls to computeFSST and p2qrs, findQRS uses the classify function and the trained deep network net to identify the QRS regions. Before calling classify, findQRS converts the data into the format expected by net, as explained in “Waveform Segmentation Using Deep Learning”:
• Each signal must be sampled at 250 Hz and partitioned into a stack of 2-by-\( N \) cell arrays, where each row corresponds to a channel and \( N \) is a multiple of 5000. The actual partitioning and stacking is done in the \texttt{computeFSST} function.

• Each of the resampled MIT signals has 6945 samples, a number that is not a multiple of 5000. To keep all the data in each signal, pad the signal with random numbers. Later in the process, the \texttt{p2qrs} function labels the random numbers as not belonging to QRS complexes and discards them.

\begin{verbatim}
function [labelVals,labelLocs] = findQRS(x,t,parentLabelVal,parentLabelLoc,varargin)
    % This is a template for creating a custom function for automated labeling
    %
    % x is a matrix where each column contains data corresponding to a
    % channel. If the channels have different lengths, then x is a cell array
    % of column vectors.
    %
    % t is a matrix where each column contains time corresponding to a
    % channel. If the channels have different lengths, then t is a cell array
    % of column vectors.
    %
    % parentLabelVal is the parent label value associated with the output
    % sublabel or empty when output is not a sublabel.
    % parentLabelLoc contains an empty vector when the parent label is an
    % attribute, a vector of ROI limits when parent label is an ROI or a point
    % location when parent label is a point.
    %
    % labelVals must be a column vector with numeric, logical or string output
    % values.
    % labelLocs must be an empty vector when output labels are attributes, a
    % two column matrix of ROI limits when output labels are ROIs, or a column
    % vector of point locations when output labels are points.

    labelVals = [];
    labelLocs = [];

    Fs = 250;

    load('trainedQTSegmentationNetwork','net')

    for kj = 1:size(x,2)
        sig = x(:,kj);

        % Create 10000-sample signal expected by the deep network
\end{verbatim}
sig = [sig;randn(10000-length(sig),1)/100]';

% Resize input and compute synchrosqueezed Fourier transforms
mitFSST = computeFSST(sig,Fs);

% Use trained network to predict which points belong to QRS regions
netPreds = classify(net,mitFSST,'MiniBatchSize',50);

% Convert stack of cell arrays into a single vector
Location = [1:length(netPreds{1}) length(netPreds{1})+(1:length(netPreds{2}'))];
Value = [netPreds{1} netPreds{2}'];

% Label QRS complexes as regions of interest and discard non-QRS data
[Locs,Vals] = p2qrs(table(Location,Value));

labelVals = [labelVals;Vals];
labelLocs = [labelLocs;Locs/Fs];

end

% Insert computeFSST and p2qrs here if you want to nest them inside
% queryQRS instead of including them as separate functions in the folder.

end

**computeFSST Function: Resize Input and Compute Synchrosqueezed Fourier Transforms**

This function reshares the input data into the form expected by net and then uses the fsst function to compute the Fourier synchrosqueezed transform (FSST) of the input. In “Waveform Segmentation Using Deep Learning”, the network performs best when given as input a time-frequency map of each training or testing signal. The FSST results in a set of features particularly useful for recurrent networks because the transform has the same time resolution as the original input. The function:

- Specifies a Kaiser window of length 128 to provide adequate frequency resolution.
- Extracts data over the frequency range from 0.5 Hz to 40 Hz.
- Subtracts the mean of each signal and divides by the standard deviation.
• Treat the real and imaginary parts of the FSST as separate features.

```matlab
function signalsFsst = computeFSST(xd,Fs)

targetLength = 5000;
signalsOut = {};

for sig_idx = 1:size(xd,1)
    current_sig = xd(sig_idx,:)';

    % Compute the number of targetLength-sample chunks in the signal
    numSigs = floor(length(current_sig)/targetLength);

    % Truncate to a multiple of targetLength
    current_sig = current_sig(1:numSigs*targetLength);

    % Create a matrix with as many columns as targetLength signals
    xM = reshape(current_sig,targetLength,numSigs);

    % Vertically concatenate into cell arrays
    signalsOut = [signalsOut; mat2cell(xM.',ones(numSigs,1))];

end

signalsFsst = cell(size(signalsOut));

for idx = 1:length(signalsOut)
    [s,f] = fsst(signalsOut{idx},Fs,kaiser(128));

    % Extract data over the frequency range from 0.5 Hz to 40 Hz
    f_indices = (f > 0.5) & (f < 40);
    signalsFsst{idx} = [real(s(f_indices,:)); imag(s(f_indices,:))];

    signalsFsst{idx} = (signalsFsst{idx}-mean(signalsFsst{idx},2)) ./ std(signalsFsst{idx},[],2);

end
end
```
**p2qrs Function: Label QRS Complexes as Regions of Interest**

The deep network outputs a categorical array that labels every point of the input signal as belonging a P region, a QRS complex, a T region, or to none of those. This function converts those point labels to QRS region-of-interest labels.

- To perform the conversion, the function assigns integer numerical values to the categories and uses the `findchangepts` function to find the points where the numerical array changes value.
- Each of those changepoints is the left endpoint of a categorical region, and the point that precedes it in the array is the right endpoint of the preceding region.
- The algorithm adds $1 \times 10^{-6}$ to the right endpoints to prevent one-sample regions from having zero duration.
- The `df` parameter selects as regions of interest only those QRS complexes whose duration is greater than `df` samples.

```matlab
function [locs,vals] = p2qrs(k)

fc = 1e-6;
df = 20;

cat2num = categorical(k.Value);
levs = 1:length(cat2num);
for jk = levs
    cat2num(k.Value == cat2num{jk}) = levs(jk);
end
chpt = findchangepts(cat2num,'MaxNumChanges',length(cat2num));
locs = [1;chpt'] [chpt'-1;length(cat2num)]+fc;
vals = categorical(cat2num(locs(:,1))',levs,cat2num);
locs = locs+round(k.Location(1))-1;
qrs = find(vals=='QRS' & diff(locs,[],2)>df);
vals = categorical(string(vals(qrs)),['QRS' 'n/a']);
locs = locs(qrs,:);
end
```
**findRpeaks Function: Find R Peaks**

This function locates the most prominent peak of the QRS regions of interest found by
findQRS. The function applies the MATLAB® islocalmax function to the absolute value
of the signal in the intervals located by findQRS.

```matlab
function [labelVals,labelLocs] = findRpeaks(x,t,parentLabelVal,parentLabelLoc,varargin)

Fs = 250;
if isempty(t)
    t = (0:length(x)-1)'/Fs;
end

labelVals = zeros(size(parentLabelLoc,1),1);
labelLocs = zeros(size(parentLabelLoc,1),1);

for kj = 1:size(parentLabelLoc,1)
    tvals = t>=parentLabelLoc(kj,1) & t<=parentLabelLoc(kj,2);
    ti = t(tvals);
    xi = x(tvals);
    lc = islocalmax(abs(xi),'MaxNumExtrema',1);
    labelVals(kj) = xi(lc);
    labelLocs(kj) = ti(lc);
end
end
```

**References**

boundaries in multilead ECG signals: Validation with the CSE database." *Computers and
Biomedical Research.* Vol. 27, No. 1, 1994, pp. 45–60.

Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H.
Eugene Stanley. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New
pp. e215–e220. [Circulation Electronic Pages: http://circ.ahajournals.org/content/101/23/
e215.full].

for Evaluation of Algorithms for Measurement of QT and Other Waveform Intervals in the
Label Spoken Words in Audio Signals Using External API

This example shows how to label spoken words in Signal Labeler. The example uses the IBM® Watson Speech to Text API and Audio Toolbox™ software. See “Speech-to-Text Transcription” (Audio Toolbox) for instructions about:

1. Downloading the Audio Toolbox speech2text extended functionality, available from MATLAB® Central.
2. Setting up the IBM Watson Speech API, offered through IBM Cloud Services. You must create an IBM Cloud account, a Speech to Text service instance, and go to the service dashboard and copy your credentials – API Key and URL values. See the Getting Started Tutorial in the IBM documentation for more details.

Load Speech Data

Load an audio data file containing the sentence "Oak is strong, and also gives shade" spoken by a male voice. The signal is sampled at 44,100 Hz.

\[
[y,fs] = \text{audioread('oak.m4a');}
\]

% To hear, type soundsc(y,fs)

1. Start Signal Analyzer and drag the signal to the Signal table. Select the signal.
2. Add time information: on the Analyzer tab, click Time Values, select Sample Rate and Start Time, and specify fs as the sample rate.
3. On the Analyzer tab, click Label. The signal appears in the Labeled Signal Set browser.

Define Label

Define a label to attach to the signal. Click Add Definition on the Label tab. Specify the Label Name as Words, select a Label Type of ROI, and enter the Data Type as string.

Create Custom Autolabeling Function

Create a custom function to label the words spoken in the audio file. (Code for the stt function appears later in the example.)
1. Go to the directory where you have stored the speech2text P-code files and the JSON file that stores your IBM Cloud credentials.

2. To create the function, in the Analyzer tab, click Automate Value ▼ and select Add Custom Function. Signal Labeler shows a dialog box asking you to enter the name, description, and label type of the function to add. Enter stt in the Name field and select ROI as the Label Type. You can leave the Description field empty or you can enter your own description.

3. Copy the function code and paste it in the empty template that appears. Save the file. The function appears in the gallery.

**Locate and Identify Spoken Words**

Locate and identify the words spoken in the input signal.
1 In the Labeled Signal Set browser, select the check box next to \( y \).
2 Select Words in the Label Definitions browser.
3 On the Automated Value gallery, select stt.
4 Click Auto-Label and click OK in the dialog box that appears.

Signal Labeler locates and labels the spoken words.

Export Labeled Signal

Export the labeled signal. On the Label tab, click Save Labels. In the dialog box that appears, give the name transcribedAudio to the labeled signal set. Clicking OK returns you to Signal Analyzer. On the Signal table, select transcribedAudio and right-click to export it to a file called Transcription.mat.
Load the labeled signal set. The set has only one member. Get the names of the labels, and use the name to obtain and display the transcribed words.

```
load Transcription

ln = getLabelNames(transcribedAudio);

v = getLabelValues(transcribedAudio,1,ln)
```

```
v=7×2 table
    ROILimits       Value
    __________    ________
       0.09    0.56    "oak"
       0.59    0.97    "is"
          1    1.78    "strong"
       1.94    2.19    "and"
       2.22    2.67    "also"
       2.67    3.22    "gives"
       3.25    3.91    "shade"
```

Rearrange the words so that the sentence reads "Oak gives shade, and also is strong."

```
Plot the signal using a different color for each word.

k = v([1 6:7 4:5 2:3],:);

s = getSignal(transcribedAudio,1);

sent = [];
sgs = NaN(height(s),height(k));
lgd = [];

for kj = 1:height(k)
    lm = length(sent);
    word = s.y(timerange(seconds(k.ROILimits(kj,1)),seconds(k.ROILimits(kj,2))));
    sent = [sent;word];
    sgs(lm+(1:length(word)),kj) = word;
    lgd = [lgd;(length(sent)-length(word)/2)/fs];
end

sgs(length(sent)+1:end,:) = [];
% To hear, type soundsc(sent,fs)
```
plot((0:length(sgs)-1)/fs,sgs)
text(lgd,-0.7*ones(size(lgd)),k.Value,'HorizontalAlignment','center')
axis tight

**stt Function: Locate and Identify Spoken Words**

This function uses the IBM Watson Speech API and the Audio Toolbox `speech2text` extended functionality to extract spoken words from an audio file.

```matlab
function [labelVals,labelLocs] = stt(x,t,parentLabelVal,parentLabelLoc,varargin)
aspeechObjectIBM = speechClient('IBM','timestamps',true,'model','en-US_NarrowbandModel',
fs = 1/(t(2)-t(1));
```
tixt = speech2text(aspeechObjectIBM,x,fs);

numLabels = numel(tixt.TimeStamps{:});
labelVals = strings(numLabels,1);
labelLocs = zeros(numLabels,2);

for idx = 1:numLabels
    labelVals(idx) = tixt.TimeStamps{:}{idx}{1};
    labelLocs(idx,1) = tixt.TimeStamps{:}{idx}{2};
    labelLocs(idx,2) = tixt.TimeStamps{:}{idx}{3};
end

end

**Tips**

- When saving labels, **Signal Labeler** converts all signals with time information to timetables. This conversion results in a deeper hierarchy of nested channels in the saved `labeledSignalSet`. See “Signal Labeler Import and Export Behavior” for more details.

- **Signal Labeler** averages channel colors when it renders labels corresponding to multichannel signals. For best results, customize the line color so that it is the same for all channels in a given signal. Perform the customization in **Signal Analyzer** before entering **Signal Labeler**.

  Compare the label color for `diff`, a signal whose channels have different colors, to the label color for `equal`, a signal whose channels all have the same color. Any labels for `equal` are rendered in the shade of blue that all the channels share. Labels for `diff` are rendered in a shade of brown that does not match any of the channel colors.
• When you reduce the width of the **Signal Labeler** window, the attribute table under the time plot shows a horizontal scrollbar. In Apple macOS systems, the scrollbar is hidden by default. If you have a trackpad and scroll horizontally, the scrollbar appears. If you use a mouse and want the scrollbar to be visible always, you can modify the system behavior in the System Preferences.

### See Also

**Apps**

Signal Analyzer

**Functions**

labeledSignalSet | signalLabelDefinition

**Topics**

“Automatically Label Signals”

“Signal Labeler Import and Export Behavior”

“Signal Labeler Keyboard Shortcuts”

**Introduced in R2019a**
sigwin

Signal processing window object

Syntax

\[ w = \text{sigwin} \cdot \text{window} \]

Description

**Note** The use of \text{sigwin} \cdot \text{window} is not recommended. Use the corresponding function instead. See “Windows” on page 1-2371 for the functional forms.

\[ w = \text{sigwin} \cdot \text{window} \] returns a window object, \( w \), of type \text{window}. Each window type takes one or more inputs. If you specify a \text{sigwin} \cdot \text{window} with no inputs, a default window of length 64 is created.

**Note** You must specify a \text{window} type with \text{sigwin}.

Windows

\text{window} for \text{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.

<table>
<thead>
<tr>
<th>Window</th>
<th>Window Object</th>
<th>Corresponding Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified Bartlett-Hann Window</td>
<td>\text{sigwin}.barthannwin</td>
<td>barthannwin</td>
</tr>
<tr>
<td>Bartlett Window</td>
<td>\text{sigwin}.bartlett</td>
<td>bartlett</td>
</tr>
<tr>
<td>Blackman Window</td>
<td>\text{sigwin}.blackman</td>
<td>blackman</td>
</tr>
<tr>
<td>Blackman-Harris Window</td>
<td>\text{sigwin}.blackmanharris</td>
<td>blackmanharris</td>
</tr>
<tr>
<td>Window</td>
<td>Window Object</td>
<td>Corresponding Function</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Bohman Window</td>
<td>sigwin.bohmanwin</td>
<td>bohmanwin</td>
</tr>
<tr>
<td>Dolph-Chebyshev Window</td>
<td>sigwin.chebwin</td>
<td>chebwin</td>
</tr>
<tr>
<td>Flat Top Window</td>
<td>sigwin.flattopwin</td>
<td>flattopwin</td>
</tr>
<tr>
<td>Gaussian Window</td>
<td>sigwin.gausswin</td>
<td>gausswin</td>
</tr>
<tr>
<td>Hamming Window</td>
<td>sigwin.hamming</td>
<td>hamming</td>
</tr>
<tr>
<td>Hann (Hanning) Window</td>
<td>sigwin.hann</td>
<td>hann</td>
</tr>
<tr>
<td>Kaiser Window</td>
<td>sigwin.kaiser</td>
<td>kaiser</td>
</tr>
<tr>
<td>Nuttall defined 4-term</td>
<td>sigwin.nuttallwin</td>
<td>nuttallwin</td>
</tr>
<tr>
<td>Blackman-Harris Window</td>
<td>sigwin.nuttallwin</td>
<td>nuttallwin</td>
</tr>
<tr>
<td>Parzen Window</td>
<td>sigwin.parzenwin</td>
<td>parzenwin</td>
</tr>
<tr>
<td>Rectangular Window</td>
<td>sigwin.rectwin</td>
<td>rectwin</td>
</tr>
<tr>
<td>Taylor Window</td>
<td>sigwin.taylorwin</td>
<td>taylorwin</td>
</tr>
<tr>
<td>Triangular Window</td>
<td>sigwin.triang</td>
<td>triang</td>
</tr>
<tr>
<td>Tukey Window</td>
<td>sigwin.tukeywin</td>
<td>tukeywin</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate</td>
<td>Returns a column vector of values representing the window.</td>
</tr>
<tr>
<td>info</td>
<td>Returns information about the window object.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>winwrite</td>
<td>Writes an ASCII file that contains window weights for a single window object or a vector of window objects. Default filename is untitled.wf.</td>
</tr>
<tr>
<td></td>
<td>winwrite(Hd,filename) writes to a disk file named filename in the current working directory. The .wf extension is added automatically.</td>
</tr>
</tbody>
</table>

**Viewing Object Parameters**

As with any object, you can use `get` to view a `sigwin` object's parameters. To see a specific parameter,

```matlab
get(w,'parameter')
```

or to see all parameters for an object,

```matlab
get(w)
```

**Changing Object Parameters**

To set specific parameters,

```matlab
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.

```matlab
w = sigwin.bartlett
```
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`.

\[ w = \text{sigwin.chebwin}(128,100) \]
w =
    Name: 'Chebyshev'
    Length: 128
    SidelobeAtten: 100

wvtool(w)

**Save Window Object as Vector**

Create a Hamming window of length 12. Visualize the result.

```matlab
H = sigwin.hamming(12);
wvtool(H)
```
Save the window values in a column vector:

\[ d = \text{generate}(H)' \]

\[ d = 1 \times 12 \]

\[ 0.0800 \quad 0.1530 \quad 0.3489 \quad 0.6055 \quad 0.8412 \quad 0.9814 \quad 0.9814 \quad 0.8412 \quad 0.6055 \quad 0.3489 \quad 0.1530 \quad 0.0800 \]

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 $N$:

$$
w(x) = 0.62 - 0.48|x| + 0.38 \cos \pi x, \quad -\frac{1}{2} \leq x \leq \frac{1}{2}
$$

where $x$ is an $N$-point linearly spaced vector over the interval $[1/2, 1/2]$.

**Construction**

$H = \text{sigwin.barthannwin}$ returns a modified Bartlett-Hann window object $H$ of length 64.

$H = \text{sigwin.barthannwin}($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 \textit{Length} rounds the length to the nearest integer. Entering a 1 for \textit{Length} results in a window with a single value of 1.

Methods

\begin{itemize}
  \item generate \hspace{2em} Generates modified Bartlett-Hann window
  \item info \hspace{2em} Display information about modified Bartlett-Hann window object
  \item winwrite \hspace{2em} Save modified Bartlett-Hann window object values in ASCII file
\end{itemize}

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects (MATLAB) 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.

\begin{verbatim}
H = sigwin.barthannwin(16);
win = generate(H)
win = 16x1
 0
 0.0649
 0.1897
 0.3586
\end{verbatim}
wininfo = info(H)

wininfo = 3x23 char array
    'Bartlett-Hanning Window'
    '-----------------------'
    'Length : 16'

wvtool(H)
References


See Also

barthannwin | window | wvtool

Topics

“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
**generate**

*Class:* sigwin.bart hannwin  
*Package:* sigwin

Generates modified Bartlett-Hann window

**Syntax**

```matlab
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**


```matlab
H = sigwin.bart hannwin(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 | wvttool
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


H = sigwin.barthannwin(16);
win = generate(H)

win = 16×1

0
0.0649
0.1897
0.3586
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 filename 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**


```
H = sigwin.barthannwin(16);
win = generate(H)
win = generate(H)
```

`win = 16×1`
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:

$$
w(n) = \begin{cases} 
\frac{2n}{N-1} & 0 \leq n \leq N/2 - 1 \\
2 - \frac{2n}{N-1} & N/2 \leq n \leq N - 1 
\end{cases}
$$

For $N$ odd, the equation for the Bartlett window is:

$$
w(n) = \begin{cases} 
\frac{2n}{N-1} & 0 \leq n \leq (N - 1)/2 \\
2 - \frac{2n}{N-1} & (N - 1)/2 + 1 \leq n \leq N - 1 
\end{cases}
$$

Construction

H = sigwin.bartlett returns a Bartlett window object H of length 64.

H = sigwin.bartlett(Length) returns a Bartlett window object H of length Length. Length must be a positive integer. Entering a positive noninteger value for Length rounds the length to the nearest integer. Entering a 1 for Length results in a window with a single value of 1.
Properties

Length

Bartlett window length. The length requires a positive integer. Entering a positive noninteger value for *Length* rounds the length to the nearest integer. Entering a 1 for *Length* results in a window with a single value of 1.

Methods

generate  Generates Bartlett window
info       Display information about Bartlett window object
winwrite   Save Bartlett window object values in ASCII file

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects (MATLAB) 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.

\[
H = \text{sigwin.bartlett}(16);
\]
\[
\text{win} = \text{generate}(H)
\]
\[
\text{win} = 16\times1
\]

\[
\begin{array}{c}
0 \\
0.1333 \\
0.2667 \\
0.4000
\end{array}
\]
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


See Also

bartlett | window | wvtool

Topics

“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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.

\[
H = \text{sigwin.bartlett}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16\times1
\]

\[
\begin{array}{c}
0 \\
0.1333 \\
0.2667 \\
0.4000 \\
0.5333 \\
0.6667 \\
0.8000
\end{array}
\]
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.

```matlab
H = sigwin.bartlett(16);
win = generate(H)
```

```
win = 16×1

0
```
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 $N$:

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

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

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

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

Construction

$H = \text{sigwin.blackman}$ 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 (MATLAB) 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.

\[
H = \text{sigwin.blackman}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16\times1
\]

\[
\begin{array}{c}
0 \\
0.0168 \\
0.0771 \\
0.2008 \\
0.3940 \\
0.6300 \\
0.8492 \\
0.9822 \\
0.9822 \\
0.8492 \\
\vdots
\end{array}
\]

\[
\text{wininfo} = \text{info}(H)
\]

\[
\text{wininfo} = 4\times26 \text{ char array}
\]

\[
\begin{array}{c}
'Blackman Window' \\
'---------' \\
'Length' : 16 \\
'Sampling Flag' : symmetric
\end{array}
\]

\[
\text{wvtool}(H)
\]
References


See Also

blackman | window | wvtool

Topics

“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
**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
```
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.

\[
H = \text{sigwin.blackman}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16 \times 1
\]

\[
\begin{bmatrix}
0 \\
0.0168 \\
0.0771 \\
0.2008
\end{bmatrix}
\]
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 filename extension wf.

winwrite(H,'filename') saves the values of the Blackman window object H in the current folder as a column vector in the ASCII file 'filename' with filename extension wf.

Examples

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
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 $N$:

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

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

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

The following table lists the coefficients:

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>0.35875</td>
</tr>
<tr>
<td>$a_1$</td>
<td>0.48829</td>
</tr>
<tr>
<td>$a_2$</td>
<td>0.14128</td>
</tr>
<tr>
<td>$a_3$</td>
<td>0.01168</td>
</tr>
</tbody>
</table>
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 1-2420 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 (MATLAB) 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.

```matlab
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
: 
```

```matlab
wininfo = info(H)
```

```
wininfo = 4x26 char array
'Blackman-Harris Window'    
'----------------------'    
'Length : 16'    
'Sampling Flag : symmetric'
```

wvtool(H)
References


See Also

blackmanharris | window | wvtool

Topics

“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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.

\[
H = \text{sigwin.blackmanharris}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16\times1
\]

0.0001
0.0036
0.0267
0.1030
0.2680
0.5206
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.

\[
H = \text{sigwin.blackmanharris}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16 \times 1
\]

0.0001
0.0036
0.0267
0.1030
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**

```matlab
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.

```matlab
H = sigwin.blackmanharris(16);  
win = generate(H)  
win = 16×1
0.0001```

1-2431
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 \( N \):

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

where \( x \) is a length \( N \) vector of linearly spaced values generated using \texttt{linspace}. The first and last elements of the Bohman window are forced to be identically zero.

Construction

\( \text{H} = \text{sigwin.bohmanwin} \) returns a Bohman window object \( \text{H} \) of length 64.

\( \text{H} = \text{sigwin.bohmanwin}(\text{Length}) \) returns a Bohman window object \( \text{H} \) of length \( \text{Length} \). \( \text{Length} \) is a positive integer. Entering a positive noninteger value for \( \text{Length} \) rounds the length to the nearest integer. Entering a 1 for \( \text{Length} \) 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 \textit{Length} rounds the length to the nearest integer. Entering a 1 for \textit{Length} results in a window with a single value of 1.

Methods

- \texttt{generate} \hspace{1em} Generates Bohman window
- \texttt{info} \hspace{1em} Display information about Bohman window object
- \texttt{winwrite} \hspace{1em} 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 (MATLAB) 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.

\[
H = \text{sigwin.bohmanwin}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16\times1
\]

\[
\begin{array}{c}
0 \\
0.0077 \\
0.0581 \\
0.1791 \\
\end{array}
\]
wininfo = info(H)

wininfo = 3x13 char array
'Bohman Window'
'-------------'
'Length : 16 '

wvtool(H)
References


See Also

bohmanwin | window | wvtool

Topics

“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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
wininfo = info(H)

wininfo = 3x13 char array
'Bohman Window'
'-------------'
'Length : 16'

wvtool(H)
See Also
bohmanwin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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 = 16x1

0
0.0077
0.0581
0.1791
wininfo = info(H)

wininfo = 3x13 char array
    'Bohman Window'
    '-------------'
    'Length : 16'

wvtool(H)
See Also
bohmanwin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
**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.

```matlab
H = sigwin.bohmanwin(16);
win = generate(H)
```

```
win = 16×1
     0
```
wininfo = info(H)

wininfo = 3x13 char array
    'Bohman Window'
    '-------------'
    'Length : 16 '

wvtool(H)
See Also
bohmanwin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
sigwin.chebwin class

Package: sigwin

Construct Dolph-Chebyshev window object

Description

Note The use of sigwin.chebwin is not recommended. Use chebwin instead.

sigwin.chebwin creates a handle to a Dolph-Chebyshev window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The Dolph-Chebyshev window is constructed in the frequency domain by taking samples of the window's Fourier transform:

$$
\hat{W}(k) = (-1)^k \frac{\cos[N\cos^{-1}[\beta\cos(\pi k/N)]]}{\cosh[N\cosh^{-1}(\beta)]}, \quad 0 \leq k \leq N - 1
$$

where

$$
\beta = \cos[1/N\cosh^{-1}(10^\alpha)]
$$

$\alpha$ determines the level of the sidelobe attenuation. The level of the sidelobe attenuation is equal to $-20\alpha$. For example, 100 dB of attenuation results from setting $\alpha = 5$

The discrete-time Dolph-Chebyshev window is obtained by taking the inverse DFT of $\hat{W}(k)$ and scaling the result to have a peak value of 1.

Construction

H = sigwin.chebwin returns a Dolph-Chebyshev window object H of length 64 with relative sidelobe attenuation of 100 dB.
H = sigwin.chebwin(Length) returns a Dolph-Chebyshev window object H of length Length with relative sidelobe attenuation of 100 dB. Length requires a positive integer. Entering a positive noninteger value for Length rounds the length to the nearest integer. A window length of 1 results in a window with a single value equal to 1.

H = sigwin.chebwin(Length,SidelobeAtten) returns a Dolph-Chebyshev window object with relative sidelobe attenuation of atten_param dB.

Properties

Length
Dolph-Chebyshev window length.

SidelobeAtten
The attenuation parameter in dB. The attenuation parameter is a positive real number that determines the relative sidelobe attenuation of the window.

Methods

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 (MATLAB) 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.

\[
H = \text{sigwin.chebwin}(16, 40);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16 \times 1
\]

\[
\begin{array}{c}
0.1138 \\
0.1964 \\
0.3319 \\
0.4926 \\
0.6613 \\
0.8163 \\
0.9353 \\
1.0000 \\
1.0000 \\
0.9353 \\
\vdots
\end{array}
\]

\[
\text{wininfo} = \text{info}(H)
\]

\[
\text{wininfo} = 4 \times 26 \text{ char array}
\]

\[
\begin{array}{c}
'\text{Chebyshev Window}' \\
'\text{---------}' \\
'\text{Length} : 16' \\
'\text{Sidelobe Attenuation} : 40'
\end{array}
\]

\[
\text{wvtool}(H)
\]
References


See Also
chebwin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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.

\[
H = \text{sigwin.chebwin}(16,40);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16 \times 1
\]

0.1138
0.1964
0.3319
0.4926
0.6613
0.8163
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 (MATLAB)
Property Attributes (MATLAB)
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.

\[
H = \text{sigwin.chebwin}(16, 40);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16\times1
0.1138
\]
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 (MATLAB)
Property Attributes (MATLAB)
**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.

```matlab
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
:

\text{wininfo = info(H)}

\text{wininfo} = 4\times26 \text{ char array}
\begin{align*}
    &\text{'Chebyshev Window'} \\
    &\text{'----------------'} \\
    &\text{'Length'} : 16' \\
    &\text{'Sidelobe Attenuation'} : 40'
\end{align*}

\text{wvtool(H)}
See Also
chebwin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
**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 (MATLAB) 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 $N$:

$$w(n) = a_0 - a_1 \cos \frac{2\pi n}{N-1} + a_2 \cos \frac{4\pi n}{N-1} - a_3 \cos \frac{6\pi n}{N-1} + a_4 \cos \frac{8\pi n}{N-1}, \quad 0 \leq n \leq M - 1,$$

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

The second half of the symmetric flat top window $M \leq n \leq N - 1$ is obtained by flipping the first half around the midpoint. The symmetric option is the preferred method when using a flat top window in FIR filter design by the window method.
The periodic flat top window is constructed by extending the desired window length by one sample, constructing a symmetric window, and removing the last sample. The periodic version is the preferred method when using a flat top window in spectral analysis because the discrete Fourier transform assumes periodic extension of the input vector.

The coefficients are listed in the following table:

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>0.21557895</td>
</tr>
<tr>
<td>$a_1$</td>
<td>0.41663158</td>
</tr>
<tr>
<td>$a_2$</td>
<td>0.277263158</td>
</tr>
<tr>
<td>$a_3$</td>
<td>0.083578947</td>
</tr>
<tr>
<td>$a_4$</td>
<td>0.006947368</td>
</tr>
</tbody>
</table>

References


See Also

flattopwin | window | wvtool

Topics

“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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.

```matlab
H = sigwin.flattopwin(16);
win = generate(H)
```

```
win =  16×1
   0.0004   0.0061   0.0314   0.0677   0.0316   0.1982   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 (MATLAB)
Property Attributes (MATLAB)
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.

\[
H = \text{sigwin.flattopwin}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16 \times 1
\]

\[
\begin{bmatrix}
-0.0004 \\
-0.0061 \\
-0.0314 \\
-0.0677
\end{bmatrix}
\]
-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 (MATLAB)
Property Attributes (MATLAB)
**winwrite**

**Class:** sigwin.flattopwin  
**Package:** sigwin

Save flat top window in ASCII file

**Syntax**

```matlab
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.

```matlab
H = sigwin.flattopwin(16);
win = generate(H)

win = 16×1
    -0.0004
```
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 (MATLAB)
Property Attributes (MATLAB)
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 $N$:

$$w(x) = e^{-\frac{1}{2}\left(\frac{a^2x^2}{M^2}\right)}, \quad -M \leq x \leq M$$

where $M=(N-1)/2$ and $x$ is a linearly spaced vector of length $N$.

Equating $\alpha$ with the usual standard deviation of a Gaussian value, $\sigma$, note:

$$\alpha = \frac{(N-1)}{2\sigma}$$

Construction

H = sigwin.gausswin returns a Gaussian window object H of length 64 and dispersion parameter $alpha$ of 2.5.

H = sigwin.gausswin(Length) returns a Gaussian window object H of length $Length$ and dispersion parameter $alpha$ of 2.5. $Length$ requires a positive integer. Entering a positive noninteger value for $Length$ rounds the length to the nearest integer. Entering a 1 for $Length$ results in a window with a single value of 1.
H = sigwin.gausswin(Length,Alpha) returns a Gaussian window object with
dispersion parameter alpha. alpha requires a nonnegative real number and is inversely
proportional to the standard deviation of a Gaussian value.

Properties

Length

Gaussian window length. The window length requires a positive integer. Entering a
positive noninteger value for Length rounds the length to the nearest integer. Entering a
1 for Length results in a window with a single value of 1.

Alpha

Width of Gaussian window. Alpha is inversely proportional to the standard deviation of a
Gaussian. Larger values of Alpha produce Gaussian windows with inflection points closer
to the peak value, or narrower windows. In the frequency domain, larger values of Alpha
produce a Gaussian window with increased spread of the main lobe in frequency but
decreased sidelobe energy.

Methods

generate Generates Gaussian window
info Display information about Gaussian window object
winwrite Save Gaussian window in ASCII file

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects
(MATLAB) in the MATLAB Programming Fundamentals documentation.

Examples
Gaussian Windows

Generate two Gaussian windows of length \( N = 64 \). Specify \( \alpha \) values of 4 and 2. Show information about the window objects. Display the windows.

\[
H4 = \text{sigwin.gausswin}(64,4);
H2 = \text{sigwin.gausswin}(64,2);
\]

\[
wvt = \text{wvtool}(H4,H2);
\text{legend(wvt.CurrentAxes,}\backslash\alpha=4,\backslash\alpha=2')
\]

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.
\begin{verbatim}
H = sigwin.gausswin(16,3);

win = generate(H)
win = 16x1
   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)
\end{verbatim}
References


See Also
gausswin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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 \( \alpha \) values of 4 and 2. Show information about the window objects. Display the windows.

\[
\begin{align*}
H4 &= \text{sigwin.gausswin}(64,4); \\
H2 &= \text{sigwin.gausswin}(64,2);
\end{align*}
\]

\[
\begin{align*}
wvt &= \text{wvtool}(H4,H2); \\
\text{legend}(wvt.\text{CurrentAxes},'\backslash\alpha=4','\backslash\alpha=2')
\end{align*}
\]
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.

```matlab
H = sigwin.gausswin(16,3);
win = generate(H)
```

```matlab
type win

win = 16×1

0.0111
0.0340
0.0889
0.1979
0.3753
0.6065
```
wininfo = info(H)

wininfo = 4x15 char array
  'Gaussian Window'
  '---------------'
  'Length : 16  '  
  'Alpha   : 3   '  

wvtool(H)
See Also
gausswin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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 \( \alpha \) values of 4 and 2. Show information about the window objects. Display the windows.

\[
\begin{align*}
H4 &= \text{sigwin.gausswin}(64,4);
H2 &= \text{sigwin.gausswin}(64,2);
\end{align*}
\]

\[
\begin{align*}
wvt &= \text{wvtool}(H4,H2);
\text{legend}(wvt.CurrentAxes,'\alpha=4','\alpha=2')
\end{align*}
\]
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.

```matlab
H = sigwin.gausswin(16,3);
win = generate(H)
```

```matlab
win = 16×1
    0.0111    0.0340    0.0889    0.1979    0.3753    0.6065
```
wininfo = info(H)

wininfo = 4x15 char array
    'Gaussian Window'
    '---------------'
    'Length : 16'
    'Alpha : 3'

wvtool(H)
See Also
gausswin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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 $\alpha$ 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.

```matlab
H = sigwin.gausswin(16,3);
win = generate(H)

win = 16×1
    0.0111    0.0340    0.0889    0.1979    0.3753    0.6065```

The window with $\alpha = 4$ has a wider mainlobe and less sidelobe energy.
wininfo = info(H)

wininfo = 4x15 char array
  'Gaussian Window'
  '---------------'
  'Length : 16'
  'Alpha  : 3'

wvtool(H)
See Also
gausswin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
sigwin.hamming class

Package: sigwin

Construct Hamming window object

Description

**Note** The use of `sigwin.hamming` is not recommended. Use `hamming` instead.

`sigwin.hamming` creates a handle to a Hamming window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

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

$$w(n) = 0.54 - 0.46\cos\frac{2\pi n}{N - 1}, \quad 0 \leq n \leq M - 1$$

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

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

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

**Construction**

$H = \text{sigwin.hamming}$ returns a symmetric Hamming window object $H$ of length 64.

$H = \text{sigwin.hamming}(\text{Length})$ returns a symmetric Hamming window object with length $\text{Length}$. $\text{Length}$ 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.

\[ H = \text{sigwin.hamming}(\text{Length}, \text{SamplingFlag}) \]

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 (MATLAB) 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.

\[
\begin{align*}
Hs &= \text{sigwin.hamming}(64, '\text{symmetric}'); \\
Hp &= \text{sigwin.hamming}(63, '\text{periodic}')
\end{align*}
\]

\[
\begin{align*}
Hp &= \\
&\quad \text{Name: 'Hamming'} \\
&\quad \text{SamplingFlag: 'periodic'} \\
&\quad \text{Length: 63}
\end{align*}
\]

\[
\begin{align*}
wvt &= \text{wvtool}(Hs,Hp); \\
\text{legend}(wvt.\text{CurrentAxes}, '\text{Symmetric}', '\text{Periodic}')
\end{align*}
\]
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


See Also

hamming | window | wvtool

Topics

“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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
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.

\[
H = \text{sigwin.hamming}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16 \times 1
\]

\[
\begin{align*}
0.0800 \\
0.1198 \\
0.2322 \\
0.3979
\end{align*}
\]
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 (MATLAB)
Property Attributes (MATLAB)
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'
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.

```matlab
H = sigwin.hamming(16);
win = generate(H)

win = 16×1
```
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 (MATLAB)
Property Attributes (MATLAB)
**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')
```
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.

\[
H = \text{sigwin.hamming}(16);
\]

\[
\text{win} = \text{generate}(H)
\]
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 (MATLAB)
Property Attributes (MATLAB)
sigwin.hann class

Package: sigwin

Construct Hann (Hanning) window object

Description

*Note* The use of `sigwin.hann` is not recommended. Use `hann` instead.

`sigwin.hann` creates a handle to a Hann window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The symmetric Hann window of length \( N \) is defined as:

\[
w(n) = \frac{1}{2} \left(1 - \cos \frac{2\pi n}{N-1}\right), \quad 0 \leq n \leq M - 1
\]

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

The second half of the symmetric Hann window \( M \leq n \leq N - 1 \) is obtained by flipping the first half around the midpoint. The symmetric option is the preferred method when using a Hann window in FIR filter design.

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

Construction

\( H = \text{sigwin.hann} \) returns a symmetric Hann window object \( H \) of length 64.

\( H = \text{sigwin.hann}(\text{Length}) \) 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 (MATLAB) 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.

```matlab
Hs = sigwin.hann(64,'symmetric');
Hp = sigwin.hann(63,'periodic')
```

```matlab
Hp =
    Name: 'Hann'
    SamplingFlag: 'periodic'
    Length: 63
```

```matlab
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.

```matlab
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
:

\[
\text{wininfo} = \text{info}(H)
\]

\[
\text{wininfo} = 4\times26 \text{ char array}
\]

\[
\begin{array}{l}
\text{'Hann Window'} \\
\text{-----------} \\
\text{'Length : 16'} \\
\text{'Sampling Flag : symmetric'}
\end{array}
\]

\[
\text{wvtool}(H)
\]
References


See Also

hann | window | wvtool

Topics

“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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 (MATLAB)
Property Attributes (MATLAB)
**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'
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.

\[
H = \text{sigwin.hann}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16 \times 1
\]
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 (MATLAB)
Property Attributes (MATLAB)
**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')
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.

```matlab
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 (MATLAB)
  Property Attributes (MATLAB)
sigwin.kaiser class

Package: sigwin

Construct Kaiser window object

Description

Note The use of sigwin.kaiser is not recommended. Use kaiser instead.

sigwin.kaiser creates a handle to a Kaiser window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

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

$$w(x) = I_0\left(\beta \sqrt{1 - \frac{4x^2}{(N - 1)^2}}\right) / I_0(\beta), \quad -(N - 1)/2 \leq x \leq (N - 1)/2$$

where $x$ is linearly spaced $N$-point vector and $I_0()$ is the modified zeroth-order Bessel function of the first kind. $\beta$ is the attenuation parameter.

Construction

H = sigwin.kaiser returns a Kaiser window object H of length 64 and attenuation parameter beta of 0.5.

H = sigwin.kaiser(Length) returns a Kaiser window object H of length Length and attenuation parameter beta of 0.5. Length requires a positive integer. Entering a positive noninteger value for Length rounds the length to the nearest integer. Entering a 1 for Length results in a window with a single value of 1.

H = sigwin.kaiser(Length,Beta) returns a Kaiser window object with real-valued attenuation parameter beta.
Properties

Length
Kaiser window length. The window length requires a positive integer. Entering a positive noninteger value for Length rounds the length to the nearest integer. Entering a 1 for Length results in a window with a single value of 1.

Beta
Attenuation parameter. Beta requires a real number. Larger absolute values of Beta result in greater stopband attenuation, or equivalently greater attenuation between the main lobe and first side lobe.

Methods

- generate: Generates Kaiser window
- info: Display information about Kaiser window object
- winwrite: Save Kaiser window in ASCII file

Copy Semantics
Handle. To learn how copy semantics affect your use of the class, see Copying Objects (MATLAB) 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,'eta = 5','eta = 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.

\[
H = \text{sigwin.kaiser}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16\times1
\]

\[
0.9403
0.9550
\]
wininfo = info(H)

wininfo = 4x13 char array
    'Kaiser Window'
    '-------------'
    'Length    : 16'
    'Beta      : 0.5'

wvtool(H)
References


See Also

kaiser | window | wvtool

Topics

“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
generate

Class: sigwin.kaiser
Package: sigwin

Generates Kaiser window

Syntax

\text{win} = \text{generate}(\text{H})

Description

\text{win} = \text{generate}(\text{H}) \text{ 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.

\text{H05} = \text{sigwin.kaiser}(128,5);
\text{H15} = \text{sigwin.kaiser}(128,15);

\text{wvt} = \text{wvtool}(\text{H05},\text{H15});
\text{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.

\[
H = \text{sigwin.kaiser}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16 \times 1
\]

\[
\begin{align*}
0.9403 \\
0.9550 \\
0.9677 \\
0.9783 \\
0.9868 \\
0.9933 \\
0.9976 \\
0.9997
\end{align*}
\]
wininfo = info(H)

```
wininfo = 4x13 char array
    'Kaiser Window'
    '-------------'
    'Length : 16'
    'Beta   : 0.5'
```

calls `wvtool(H)`
See Also
kaiser | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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.

\[
H05 = \text{sigwin.kaiser}(128,5);
H15 = \text{sigwin.kaiser}(128,15);
\]

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.

\[
H = \text{sigwin.kaiser}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16 \times 1
\]

0.9403
0.9550
0.9677
0.9783
0.9868
0.9933
0.9976
0.9997
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 (MATLAB)
Property Attributes (MATLAB)
**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.

```matlab
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 = 16x1

0.9403
0.9550
0.9677
0.9783
0.9868

Figure 1: Window Visualization Tool

Leakage Factor: 0 %  Relative sidelobe attenuation: -114.1 dB  Mainlobe width (-3dB): 0.033203
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 (MATLAB)
Property Attributes (MATLAB)
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 1-2552 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 (MATLAB) 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.

```matlab
Hs = sigwin.nuttallwin(64);
Hp = sigwin.nuttallwin(63);
```
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.

```matlab
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 $N$.

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

The following equation defines the periodic Nuttall defined four-term Blackman-Harris window of length $N$.

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

The following table lists the coefficients:
<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>0.3635819</td>
</tr>
<tr>
<td>$a_1$</td>
<td>0.4891775</td>
</tr>
<tr>
<td>$a_2$</td>
<td>0.1365995</td>
</tr>
<tr>
<td>$a_3$</td>
<td>0.0106411</td>
</tr>
</tbody>
</table>

**References**


**See Also**

nuttallwin | window | wvtool

**Topics**

“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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.

\[
\begin{align*}
H &= \text{sigwin.nuttallwin}(16); \\
\text{win} &= \text{generate}(H)
\end{align*}
\]

\[
\begin{align*}
\text{win} &= 16\times1 \\
&= \begin{bmatrix}
0.0004 \\
0.0048 \\
0.0306 \\
0.1105 \\
0.2778 \\
0.5292 \\
0.7983
\end{bmatrix}
\end{align*}
\]
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 (MATLAB)
Property Attributes (MATLAB)
info

**Class:** sigwin.nuttallwin  
**Package:** sigwin

Display information about Nuttall defined four-term Blackman-Harris window object

**Syntax**

```matlab
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.

```matlab
Hs = sigwin.nuttallwin(64);  
Hp = sigwin.nuttallwin(63);  
Hp.SamplingFlag = '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.

```matlab
H = sigwin.nuttallwin(16);
win = generate(H)
```

```
win = 16x1

0.0004
0.0048
0.0306
0.1105
```
wininfo = info(H)

wininfo = 4x26 char array
    'Nuttall Window         '    
    '--------------         '    
    'Length         : 16    '    
    'Sampling Flag : symmetric'
See Also
nuttallwin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
**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);
```
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.

```matlab
H = sigwin.nuttallwin(16);
win = generate(H)
```

```
win = 16×1
     0.0004    0.0048
```
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 (MATLAB)
Property Attributes (MATLAB)
sigwin.parzenwin class

Package: sigwin

Construct Parzen window object

Description

**Note** The use of `sigwin.parzenwin` is not recommended. Use `parzenwin` instead.

`sigwin.parzenwin` creates a handle to a Parzen window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

The following equation defines the $N$-point Parzen window over the interval $-\frac{(N-1)}{2} \leq n \leq \frac{(N-1)}{2}$:

$$w(n) = \begin{cases} 
1 - 6\left(\frac{|n|}{N/2}\right)^2 + 6\left(\frac{|n|}{N/2}\right)^3, & 0 \leq |n| \leq (N - 1)/4 \\
2\left(1 - \frac{|n|}{N/2}\right)^3, & (N - 1)/4 < |n| \leq (N - 1)/2 
\end{cases}$$

Construction

`H = sigwin.parzenwin` returns a Parzen window object $H$ of length 64.

`H = sigwin.parzenwin(Length)` returns a Parzen window object $H$ of length `Length`. `Length` requires a positive integer. Entering a positive noninteger value for `Length` rounds the length to the nearest integer. Entering a 1 for `Length` results in a window with a single value of 1.
Properties

Length

Length requires a positive integer. Entering a positive noninteger value for Length rounds the length to the nearest integer. Entering a 1 for Length results in a window with a single value of 1.

Methods

generate | Generate Parzen window
info | Display information about Parzen window object
winwrite | Save Parzen window in ASCII file

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects (MATLAB) 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
wininfo = info(H)

wininfo = 3x13 char array
'Parzen Window'
'-------------'
'Length : 16 '
References


See Also

parzenwin | window | wvtool

Topics

“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
**generate**

**Class:** sigwin.parzenwin  
**Package:** sigwin

Generate Parzen window

**Syntax**

```matlab
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.

```matlab
H = sigwin.parzenwin(16);
win = generate(H)
```

```
win = 16×1
   0.0005  
   0.0132  
   0.0610  
   0.1675  
   0.3540  
   0.5972  
   0.8286 
```
wininfo = info(H)

wininfo = 3x13 char array
'Parzen Window'
'-------------'
'Length : 16'

wvtool(H)
See Also
parzenwin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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.

$H = \text{sigwin.parzenwin}(16)$;

$\text{win} = \text{generate}(H)$

$\text{win} = 16\times1$

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 (MATLAB)
Property Attributes (MATLAB)
**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
```

```
wininfo = info(H)
```

```
wininfo = 3x13 char array
  'Parzen Window'
  '-------------'
  'Length : 16'
```

```
wvtool(H)
```
See Also
parzenwin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
sigwin.rectwin class

Package: sigwin

Construct rectangular window object

Description

Note The use of sigwin.rectwin is not recommended. Use rectwin instead.

sigwin.rectwin creates a handle to a rectangular window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

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

$$w(n) = 1, \quad 0 \leq n \leq N - 1$$

Construction

$H = \text{sigwin.rectwin}$ returns a rectangular window object $H$ of length 64.

$H = \text{sigwin.rectwin}(\text{Length})$ returns a rectangular window object $H$ of length $\text{Length}$. $\text{Length}$ requires a positive integer. Entering a positive noninteger value for $\text{Length}$ rounds the length to the nearest integer. Entering a 1 for $\text{Length}$ 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 $\text{Length}$ rounds the length to the nearest integer. Entering a 1 for $\text{Length}$ 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 (MATLAB) 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.

\[
H = \text{sigwin.rectwin}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16\times1
\]

\[
\begin{array}{c}
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
\end{array}
\]

\[
\text{wininfo} = \text{info}(H)
\]
wininfo = 3x18 char array
'Rectangular Window'
'------------------'
'Length : 16    '

wvtool(H)

References

See Also
rectwin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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.

\[
H = \text{sigwin.rectwin}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16 \times 1
\]

1
1
1
1
1
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 (MATLAB)
Property Attributes (MATLAB)
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
wininfo = info(H)

wininfo = 3x18 char array
    'Rectangular Window'
    '------------------'
    'Length : 16      '

wvtool(H)
See Also
rectwin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
**winwrite**

**Class:** sigwin.rectwin  
**Package:** sigwin

Save rectangular window in ASCII file

**Syntax**

```matlab
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.

```matlab
H = sigwin.rectwin(16);  
win = generate(H)  
win = 16×1
```

1
wininfo = info(H)

\[
\begin{array}{l}
\text{wininfo = 3x18 char array} \\
\text{'Rectangular Window'} \\
\text{'------------------'} \\
\text{'Length : 16'}
\end{array}
\]

wvtool(H)
See Also
rectwin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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 (MATLAB) 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.
Generate a Taylor window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.

```matlab
H = sigwin.taylorwin(16);
win = generate(H)
```

```
win = 16×1
0.3931
0.5021
```

Generate a Taylor window of length $N = 16$. Return its values as a column vector. Show information about the window object. Display the window.
0.6912
0.9174
1.1409
1.3330
1.4737
1.5485
1.5485
1.4737
:

```matlab
wininfo = info(H)
```

```matlab
wininfo = 5x47 char array
'Taylor Window
'-----------------
'Length : 16'
'Number of nearly constant-level sidelobes : 4'
'Maximum sidelobe level : 30'
```

```matlab
wvtool(H)
```
References


See Also
taylorwin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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.

```matlab
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.

```matlab
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
```
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 (MATLAB)
Property Attributes (MATLAB)
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.

\[
H = \text{sigwin.taylorwin}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16\times1
\]

\[
0.3931 \\
0.5021 \\
0.6912 \\
0.9174 \\
1.1409 \\
1.3330 \\
1.4737 \\
1.5485
\]
```
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 (MATLAB)
Property Attributes (MATLAB)
**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.

\[
H = \text{sigwin.taylorwin}(16);
\]

\[
\text{win} = \text{generate}(H)
\]

\[
\text{win} = 16 \times 1
\]

\[
\begin{array}{c}
0.3931 \\
0.5021 \\
0.6912 \\
0.9174 \\
1.1409 \\
1.3330 \\
1.4737 \\
1.5485
\end{array}
\]
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 (MATLAB)
Property Attributes (MATLAB)
sigwin.triang class

Package: sigwin

Construct triangular window object

Description

Note The use of sigwin.triang is not recommended. Use triang instead.

sigwin.triang is a triangular window object.

sigwin.triang creates a handle to a triangular window object for use in spectral analysis and FIR filtering by the window method. Object methods enable workspace import and ASCII file export of the window values.

For $L$ odd, the triangular window is defined as:

$$w(n) = \begin{cases} \frac{2n}{L+1} & 1 \leq n \leq \frac{L+1}{2} \\ 2 - \frac{2n}{L+1} \frac{L+1}{2} + 1 \leq n \leq L \end{cases}$$

For $L$ even, the triangular window is defined as:

$$w(n) = \begin{cases} \frac{(2n-1)}{L} & 1 \leq n \leq \frac{L}{2} \\ 2 - \frac{(2n-1)}{L} \frac{L}{2} + 1 \leq n \leq L \end{cases}$$

Construction

$H = \text{sigwin.triang}$ 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 (MATLAB) 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


See Also

triang | window | wvtool

Topics

“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
**generate**

Class: sigwin.triang
Package: sigwin

Generates triangular window

**Syntax**

\[ \text{win} = \text{generate}(H) \]

**Description**

\[ \text{win} = \text{generate}(H) \]

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.

\[ \text{H} = \text{sigwin.triang}(16); \]

\[ \text{win} = \text{generate}(\text{H}) \]

\[ \text{win} = 16 \times 1 \]

0.0625
0.1875
0.3125
0.4375
0.5625
0.6875
0.8125
wininfo = info(H)

wininfo = 3x17 char array
    'Triangular Window'
    '-----------------'
    'Length : 16'

wvtool(H)
See Also
triang | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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 (MATLAB)
Property Attributes (MATLAB)
**winwrite**

**Class:** sigwin.triang  
**Package:** sigwin

Save triangular window in ASCII file

**Syntax**

```matlab
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.

```matlab
H = sigwin.triang(16);  
win = generate(H)  
win = 16×1
     0.0625
```

1-2622
wininfo = info(H)

wininfo = 3x17 char array
    'Triangular Window'
    '-----------------'
    'Length : 16     '
See Also
triang | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
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:

$$ w(x) = \begin{cases} 
\frac{1}{2}\left\{1 + \cos\left(\frac{2\pi}{\alpha}[x - \alpha/2]\right)\right\} & 0 \leq x < \frac{\alpha}{2} \\
1 & \frac{\alpha}{2} \leq x < 1 - \frac{\alpha}{2} \\
\frac{1}{2}\left\{1 + \cos\left(\frac{2\pi}{\alpha}[x - 1 + \alpha/2]\right)\right\} & 1 - \frac{\alpha}{2} \leq x \leq 1 
\end{cases} $$

where $x$ is a $N$-point linearly spaced vector generated using `linspace`. The parameter $\alpha$ is the ratio of cosine-tapered section length to the entire window length with $0 \leq \alpha \leq 1$. For example, setting $\alpha=0.5$ produces a Tukey window where $1/2$ of the entire window length consists of segments of a phase-shifted cosine with period $2\alpha=1$. If you specify $\alpha \leq 0$, an $N$-point rectangular window is returned. If you specify $\alpha \geq 1$, a von Hann window (`sigwin.hann`) is returned.

**Construction**

$H = \text{sigwin.tukeywin}$ returns a Tukey or cosine-tapered window object $H$ of length 64 with $\textit{Alpha}$ parameter equal to 0.5.
H = sigwin.tukeywin(Length) returns a Tukey window object H of length Length with Alpha parameter equal to 0.5. Length requires a positive integer. Entering a positive noninteger value for Length rounds the length to the nearest integer.

H = sigwin.tukeywin(Length,Alpha) returns a Tukey window object with the ratio of the tapered section length to the entire window length Alpha. Alpha defaults to 0.5. As Alpha approaches zero, the Tukey window approaches a rectangular window. As Alpha approaches one, the Tukey window approaches a Hann window.

Properties

Length

Tukey window length. The window length must be a positive integer. Entering a positive noninteger value for Length rounds the length to the nearest integer. Entering a 1 for Length results in a window with a single value of 1.

Alpha

The ratio of tapered window section to constant section. As a ratio, Alpha satisfies the inequality \(0 \leq \alpha \leq 1\). As Alpha approaches zero, the Tukey window approaches a rectangular window. As Alpha approaches one, the Tukey window approaches a Hann window. Specifying Alpha less than zero or greater than one replaces Alpha with 0 and 1 respectively.

Methods

generate Generates Tukey window
info Display information about Tukey window object
winwrite Save Tukey window in ASCII file

Copy Semantics

Handle. To learn how copy semantics affect your use of the class, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.
Examples

Tukey Windows

Generate two Tukey windows of length $N = 64$:

- The first window has $\alpha = 1/4$. $\alpha$ is the ratio of tapered window section length to constant section length.
- The second window has $\alpha = 3/4$.

Display the two windows.

```matlab
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.

```matlab
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
```

![Image of Window Visualization Tool](image)
wininfo = info(H)

wininfo = 4x13 char array
    'Tukey Window'
    '------------'
    'Length : 16'
    'Alpha  : 0.5'

wvtool(H)
References


See Also
tukeywin | window | wvtool

Topics
“Windows”
Class Attributes (MATLAB)
Property Attributes (MATLAB)
**generate**

**Class:** sigwin.tukeywin  
**Package:** sigwin

Generates Tukey window

**Syntax**

```matlab
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 `α = 1/4`. `α` is the ratio of tapered window section length to constant section length.
- The second window has `α = 3/4`.

Display the two windows.

```matlab
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.

```matlab
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
```

![Window Visualization Tool](image)

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.

```matlab
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
```
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 (MATLAB)
Property Attributes (MATLAB)
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 α = 1/4. α is the ratio of tapered window section length to constant section length.
• The second window has α = 3/4.

Display the two windows.

H14 = sigwin.tukeywin(64,1/4);
H34 = sigwin.tukeywin(64,3/4);
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.

$H = \text{sigwin.tukeywin}(16);$  

$\text{win} = \text{generate}(H)$  

$\text{win} = 16\times1$  

0 0.1654 0.5523 0.9045 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 (MATLAB)
Property Attributes (MATLAB)
**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 \(\alpha = 1/4\). \(\alpha\) 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);
```
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.

$$H = \text{sigwin.tukeywin}(16);$$

$$\text{win} = \text{generate}(H)$$

$$\text{win} = 16 \times 1$$

$$\begin{array}{c}
0 \\
0.1654 \\
0.5523 \\
0.9045 \\
\end{array}$$
1.0000
1.0000
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 (MATLAB)
Property Attributes (MATLAB)
**sinad**

Signal to noise and distortion ratio

**Syntax**

\[ r = \text{sinad}(x) \]
\[ r = \text{sinad}(x,fs) \]
\[ r = \text{sinad}(pxx,f,'psd') \]
\[ r = \text{sinad}(sxx,f,rbw,'power') \]
\[ [r,totdistpow] = \text{sinad}(\_\_\_) \]
\[ \text{sinad}(\_\_) \]

**Description**

\( r = \text{sinad}(x) \) 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 \( \beta = 38 \).

\( r = \text{sinad}(x,fs) \) specifies the sample rate \( fs \) of the input signal \( x \). If you do not specify \( fs \), then the sample rate defaults to 1.

\( r = \text{sinad}(pxx,f,'psd') \) 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 \( pxx \).

\( r = \text{sinad}(sxx,f,rbw,'power') \) specifies the input as a one-sided power spectrum. \( rbw \) is the resolution bandwidth over which each power estimate is integrated.

\([r,totdistpow] = \text{sinad}(\_\_)\) 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.

```matlab
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.

```matlab
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));
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
```
[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

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.

See Also

`sfdr` | `snr` | `thd` | `toi`

Topics

“Analyzing Harmonic Distortion”

Introduced in R2013b
sinc

Sinc function

Syntax

\( y = \text{sinc}(x) \)

Description

\( y = \text{sinc}(x) \) returns an array, \( y \), whose elements are the “sinc” on page 1-2653 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.

```matlab
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

\[
\text{sinc}(t) = \begin{cases} 
\frac{\sin(\pi t)}{\pi t} & t \neq 0, \\
1 & t = 0.
\end{cases}
\]

This analytic expression corresponds to the continuous inverse Fourier transform of a rectangular pulse of width 2\pi and height 1:

\[
\text{sinc}(t) = \frac{1}{2\pi} \int_{-\pi}^{\pi} e^{j\omega t} d\omega.
\]

The space of functions bandlimited in the frequency range \(\omega = (-\pi, \pi]\) is spanned by the countably infinite set of sinc functions shifted by integers. Thus, you can reconstruct any such bandlimited function \(g(t)\) from its samples at integer spacings:

\[
g(t) = \sum_{n=-\infty}^{\infty} g(n)\text{sinc}(t - n).
\]
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” (MATLAB).

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

Introduced before R2006a
single

Cast coefficients of digital filter to single precision

**Syntax**

\[ f2 = \text{single}(f1) \]

**Description**

\( f2 = \text{single}(f1) \) 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 \text{designfilt} to design a 5th-order FIR lowpass filter. Specify a normalized passband frequency of \( 0.2\pi \) rad/sample and a normalized stopband frequency of \( 0.55\pi \) rad/sample. Cast the filter coefficients to single precision.

```matlab
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:
```matlab
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. The slew rate is the slope of the line connecting the 10% and 90% reference levels. The sample instants of X are the indices of the vector. To determine the transitions, slewrate estimates the state levels of the input waveform by a histogram method. slewrate 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 1-2666.

S = slewrate(X,Fs) specifies the sample rate, Fs, in hertz. The first time instant in X corresponds to t=0.

S = slewrate(X,T) specifies the sample instants in the vector, T. The length of T must equal the length of X.

[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%.

[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 pair 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 indicates the state levels and associated lower and upper tolerances.

**Input Arguments**

**X**

Bilevel waveform as a real-valued column or row vector. If the input waveform does not have at least one transition, slewrate returns an empty matrix.

**Fs**

Sampling rate in hertz.

**T**

Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

**Name-Value Pair Arguments**

**PercentReferenceLevels**

Percent reference levels. See “Percent Reference Levels” on page 1-2666 for a definition.

Default: [10,90]

**StateLevels**

Low- and high-state levels. StateLevels is a 1-by-2 real-valued vector. The first element is the low-state level. The second element is the high-state level. If you do not specify low- and high-state levels, slewrate estimates the state levels from the input waveform using the histogram method.
**Tolerance**

Tolerance levels (lower and upper state boundaries) expressed as a percentage. See “State-Level Tolerances” on page 1-2666.

**Default:** 2

**Output Arguments**

**S**

Slew rates as real-valued scalars. A positive slew rate indicates that the upper-percent reference level occurs later than the lower-percent reference level. A negative slew rate indicates that the upper-percent reference level occurs before the lower-percent reference level.

**LT**

Time instants when signal crosses the lower percent reference level. If you do not specify the lower percent reference levels with the 'PercentReferenceLevels' name-value pair, the lower percent reference level is 10%.

**UT**

Time instants when signal crosses the upper-percent reference level. If you do not specify the upper-percent reference levels with the 'PercentReferenceLevels' name-value pair, the upper-percent reference level is 90%.

**LL**

Waveform values at the lower-reference level.

**UL**

Waveform values at the upper-reference level.

**Examples**
Slew Rate For One-Transition Waveform

Use `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.

```matlab
load('transitionex.mat','x','t')
slewrate(x,t)
```

![Graph of Slew Rate](image.png)

```
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.

```matlab
load('transitionex.mat','x')
fs = 4e6;
y = [x;flipud(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.

```matlab
slewrate(y,t);
```
Lower and Upper Transition Times

Return the lower- and upper-transition times for a three-transition waveform sampled at 4 MHz.

```matlab
load('transitionex.mat','x')
fs = 4e6;

y = [x;flip(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.

```matlab
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]);
More About

Percent Reference Levels

If \( S_1 \) is the low state, \( S_2 \) is the high state, and \( U \) is the upper-percent reference level. The waveform value corresponding to the upper-percent reference level is

\[
S_1 + \frac{U}{100}(S_2 - S_1).
\]

If \( L \) is the lower-percent reference level, the waveform value corresponding to the lower percent reference level is

\[
S_1 + \frac{L}{100}(S_2 - S_1).
\]

Slew Rate

The slew rate is the slope of a line connecting the upper- and lower-percent reference levels. Let \( t_L \) denote the time instant when the waveform crosses the lower reference level and \( t_U \) denote the time instant when the waveform crosses the upper percent reference level. Using the definitions for the upper and lower percent reference levels given in “Percent Reference Levels” on page 1-2666, 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

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the \( \alpha \% \) region for the low state is defined as

\[
S_1 \pm \frac{\alpha}{100}(S_2 - S_1),
\]
where $S_1$ is the low-state level and $S_2$ is the high-state level. Replace the first term in the equation with $S_2$ to obtain the $\alpha\%$ tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.

![State Levels ± 2% Tolerance](image)

**References**

See Also
falltime | midcross | pulsewidth | risetime | settlingtime | statelevels

Introduced in R2012a


**snr**

Signal-to-noise ratio

**Syntax**

\[ r = \text{snr}(x,y) \]
\[ r = \text{snr}(x) \]
\[ r = \text{snr}(x,fs,n) \]
\[ r = \text{snr}(pxx,f,'psd') \]
\[ r = \text{snr}(pxx,f,n,'psd') \]
\[ r = \text{snr}(sxx,f,rbw,'power') \]
\[ r = \text{snr}(sxx,f,rbw,n,'power') \]
\[ r = \text{snr}(__,'aliased') \]
\[ [r,\text{noisepow}] = \text{snr}(__) \]
\[ \text{snr}(__) \]

**Description**

\[ r = \text{snr}(x,y) \] returns the signal-to-noise ratio (SNR) in decibels of a signal, \( x \), by computing the ratio of its summed squared magnitude to that of the noise, \( y \). \( y \) must have the same dimensions as \( x \). Use this form when the input signal is not necessarily sinusoidal and you have an estimate of the noise.

\[ r = \text{snr}(x) \] returns the SNR in dBc 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.

\[ r = \text{snr}(x,fs,n) \] 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 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

Compute the signal-to-noise ratio (SNR) of a 20 ms rectangular pulse sampled for 2 s at 10 kHz in the presence of Gaussian noise. Set the random number generator to the default settings for reproducible results.
	rng default
Tpulse = 20e-3;
Fs = 10e3;
**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 0.1².

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 = 1x2

17.0178   16.9897

THD = thd(x);
defTHD = 10*log10(powharm/powfund);
TH = [THD defTHD]
\[ TH = 1 \times 2 \]

\[
\begin{array}{cc}
-7.9546 & -7.9588 \\
\end{array}
\]

\[
\text{SINAD} = \text{sinad}(x); \\
\text{defSINAD} = 10 \times \log_{10}(\text{powfund}/(\text{powharm} + \text{varnoise})); \\
\text{SI} = [\text{SINAD} \ \text{defSINAD}] \\
\]

\[ SI = 1 \times 2 \]

\[
\begin{array}{cc}
7.4571 & 7.4473 \\
\end{array}
\]

**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\(^2\).

\[
\begin{align*}
\text{Fi} &= 2500; \\
\text{Fs} &= 48e3; \\
\text{N} &= 1024; \\
x &= \sin(2\pi\text{Fi}/\text{Fs}*(1:N)) + 0.001*\text{randn}(1,\text{N}); \\
\text{SNR} &= \text{snr}(x,\text{Fs}) \\
\text{SNR} &= 57.7103 \\
\end{align*}
\]

Plot the spectrum and annotate the SNR.

\[
\text{snr}(x,\text{Fs});
\]
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.

```matlab
rng default
Fi = 2500;
Fs = 48e3;
N = 1024;
```

SNR: 57.71 dB
\[
x = \sin(2\pi \frac{Fi}{Fs} (1:N)) + 0.00001 \cdot \text{randn}(1,N);
\]
\[
w = \text{kaiser}([\text{numel}(x),38]);
\]
\[
[Pxx, F] = \text{periodogram}(x,w,\text{numel}(x),Fs);
\]
\[
\text{SNR} = \text{snr}(Pxx,F,'\text{psd}')
\]
\[
\text{SNR} = 97.7446
\]

**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.

\[
\text{rng default}
\]
\[
Fi = 2500;
\]
\[
Fs = 48e3;
\]
\[
N = 1024;
\]
\[
x = \sin(2\pi \frac{Fi}{Fs} (1:N)) + 0.00001 \cdot \text{randn}(1,N);
\]
\[
w = \text{kaiser}([\text{numel}(x),38]);
\]
\[
[Sxx, F] = \text{periodogram}(x,w,\text{numel}(x),Fs,'\text{power}');
\]
\[
\text{rbw} = \text{enbw}(w,Fs);
\]
\[
\text{SNR} = \text{snr}(Sxx,F,\text{rbw},'\text{power}')
\]
\[
\text{SNR} = 97.7446
\]

Plot the spectrum of the signal and annotate the SNR.

\[
\text{snr}(Sxx,F,\text{rbw},'\text{power}');
\]
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.

```matlab
Fs = 10000;
f = 2100;

t = 0:1/Fs:1;
x = tanh(sin(2*pi*f*t)+0.1) + 0.001*randn(1,length(t));
```
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.

```matlab
snr(x,Fs,7);
```
Repeat the computation, but now treat the aliased harmonics as part of the signal.

```matlab
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 $0.1^2$.

```matlab
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 = 1x2

    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**

- **x** — **Real-valued input signal**
  
  real vector

  Real-valued input signal, specified as a row or column vector.

  Data Types: double | single
y — Noise estimate
declare y as real vector

Estimate of the noise in the input signal, specified as a real-valued row or column vector. It must have the same dimensions as x.

Data Types: double | single

fs — Sample rate
declare fs as 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
declare n as 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
declare pxx as 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
declare f as 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.

See Also
snr | sfdr | sinad | thd | toi

Topics
“Analyzing Harmonic Distortion”

Introduced in R2013b
sos2cell

Convert second-order sections matrix to cell array

Syntax

\[
c = \text{sos2cell}(m)
\]
\[
c = \text{sos2cell}(m,g)
\]

Description

\[
c = \text{sos2cell}(m)
\]
changes an \(L\)-by-6 second-order section matrix \(m\) generated by \(\text{tf2sos}\) into a 1-by-\(L\) cell array of 1-by-2 cell arrays, \(c\). You can use \(c\) to specify a quantized filter with \(L\) cascaded second-order sections.

The matrix \(m\) should have the form

\[
m = [b_1 \ a_1; b_2 \ a_2; \ldots; b_L \ a_L]
\]

where both \(b_i\) and \(a_i\), with \(i = 1, \ldots, L\), are 1-by-3 row vectors. The resulting \(c\) is a 1-by-\(L\) cell array of cells of the form

\[
c = \{ \{b_1 \ a_1\} \{b_2 \ a_2\} \ldots \{b_L \ a_L\} \}
\]

\[
c = \text{sos2cell}(m,g)
\]
with the optional gain term \(g\), prepends the constant value \(g\) to \(c\). When you use the added gain term in the command, \(c\) is a 1-by-\(L\) cell array of cells of the form

\[
c = \{\{g,1\} \{b_1,a_1\} \{b_2,a_2\} \ldots \{b_L,a_L\}\}
\]

Examples
**Second-Order-Section Cell Array of Elliptic Filter**

Generate a lowpass elliptic filter of order 4 with 0.5 dB of passband ripple and 20 dB of stopband attenuation. The passband edge is 0.6 times the Nyquist frequency. Convert the transfer function to a matrix of second-order sections.

\[
[b, a] = \text{ellip}(4, 0.5, 20, 0.6);
\]
\[
m = \text{tf2sos}(b, a);
\]

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`.

\[
c = \text{sos2cell}(m);
\]
\[
\text{compare} = [c{1}{2}; m(1,4:6)]
\]

```plaintext
\[
\text{compare} = 2\times3
\]
\[
\begin{array}{ccc}
1.0000 & 0.1677 & 0.2575 \\
1.0000 & 0.1677 & 0.2575 \\
\end{array}
\]
```

**See Also**

`cell2sos` | `tf2sos`

**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

sos2ss converts a second-order section representation of a digital filter to an equivalent state-space representation.

\[[A,B,C,D] = sos2ss(sos)\] converts sos, a system expressed in second-order section form, to a single-input, single-output state-space representation:

\[
x(n+1) = Ax(n) + Bu(n),
\]
\[
y(n) = Cx(n) + Du(n).
\]

The discrete transfer function in second-order section form is given by

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

sos is a \(L \times 6\) matrix organized as

\[
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{b matrix}

The entries of sos must be real for proper conversion to state space. The returned matrix A is of size \(2L \times 2L\), B is a \(2L \times 1\) column vector, C is a \(1 \times 2L\) row vector, and D is a \(1 \times 1\) scalar.
\[[A,B,C,D] = \text{sos2ss}(sos,g)\] converts to state space a system \(sos\) in second-order section form with gain \(g\):

\[H(z) = g \prod_{k=1}^{L} H_k(z).\]

**Examples**

**State-Space Representation of a Second-Order Section System**

Compute the state-space representation of a simple second-order section system with a gain of 2.

\[sos = \begin{bmatrix} 1 & 1 & 1 & 0 & -1 \\ -2 & 3 & 1 & 1 & 10 & 1 \end{bmatrix};\]

\[[A,B,C,D] = \text{sos2ss}(sos,2)\]

\(A = 4 \times 4\)

\[
\begin{bmatrix}
-10 & 0 & 10 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\]

\(B = 4 \times 1\)

\[
\begin{bmatrix}
1 \\
0 \\
0 \\
0
\end{bmatrix}
\]

\(C = 1 \times 4\)

\[
\begin{bmatrix}
42 & 4 & -32 & -2
\end{bmatrix}
\]

\(D = -4\)
Algorithms

`sos2ss` first converts from second-order sections to transfer function using `sos2tf`, and then from transfer function to state-space using `tf2ss`.

See Also
`sos2tf | sos2zp | ss2sos | tf2ss | zp2ss`

Introduced before R2006a
**sos2tf**

Convert digital filter second-order section data to transfer function form

**Syntax**

\[
[b, a] = \text{sos2tf}(sos)
\]

\[
[b, a] = \text{sos2tf}(sos, g)
\]

**Description**

`sos2tf` converts a second-order section representation of a given digital filter to an equivalent transfer function representation.

\[
[b, a] = \text{sos2tf}(sos)
\]

returns the numerator coefficients \(b\) and denominator coefficients \(a\) of the transfer function that describes a discrete-time system given by \(sos\) in second-order section form. The second-order section format of \(H(z)\) is given by

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

\(sos\) is an \(L\)-by-6 matrix that contains the coefficients of each second-order section stored in its rows.

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

Row vectors \(b\) and \(a\) contain the numerator and denominator coefficients of \(H(z)\) stored in descending powers of \(z\).

\[
H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2z^{-1} + \ldots + b_{n+1}z^{-n}}{a_1 + a_2z^{-1} + \ldots + a_{m+1}z^{-m}}.
\]
[b,a] = sos2tf(sos,g) returns the transfer function that describes a discrete-time system given by sos in second-order section form with gain g.

\[ H(z) = g \prod_{k=1}^{L} H_k(z). \]

**Examples**

**Transfer Function Representation of a Second-Order Section System**

Compute the transfer function representation of a simple second-order section system.

```
sos = [1 1 1 1 0 -1; -2 3 1 1 10 1];
[b,a] = sos2tf(sos)
```

```
b = 1x5
   -2   1   2   4   1
```

```
a = 1x5
   1   10   0  -10  -1
```

**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

\[
[z,p,k] = \text{sos2zp}(sos)
\]
\[
[z,p,k] = \text{sos2zp}(sos,g)
\]

Description

sos2zp converts a second-order section representation of a given digital filter to an equivalent zero-pole-gain representation.

\[
[z,p,k] = \text{sos2zp}(sos) \quad \text{returns the zeros } z, \text{ poles } p, \text{ and gain } k \text{ of the system given by } sos \text{ in second-order section form. The second-order section format of } H(z) \text{ is given by}
\]
\[
H(z) = \prod_{k=1}^{L} H_k(z) = \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}.
\]

sos is an \(L\)-by-6 matrix that contains the coefficients of each second-order section in its rows.

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

Column vectors \(z\) and \(p\) contain the zeros and poles of the transfer function \(H(z)\).

\[
H(z) = k \frac{(z - z_1)(z - z_2)\cdots(z - z_n)}{(p - p_1)(p - p_2)\cdots(p - p_m)}
\]

where the orders \(n\) and \(m\) are determined by the matrix sos.
[z, p, k] = sos2zp(sos, g) returns the zeros z, poles p, and gain k of the system given by sos in second-order section form with gain g.

\[ H(z) = g \prod_{k=1}^{L} H_k(z) \]

**Examples**

**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

** Algorithms**

sos2zp finds the poles and zeros of each second-order section by repeatedly calling tf2zp.
See Also
sos2ss | sos2tf | ss2zp | tf2zp | tf2zpk | zp2sos

Introduced before R2006a
sosfilt

Second-order (biquadratic) IIR digital filtering

Syntax

\[ y = \text{sosfilt}(sos,x) \]
\[ y = \text{sosfilt}(sos,x,\text{dim}) \]

Description

\[ y = \text{sosfilt}(sos,x) \] applies the second-order section digital filter \( sos \) to the input signal \( x \).

\[ y = \text{sosfilt}(sos,x,\text{dim}) \] operates along the dimension \( \text{dim} \).

Examples

Second-Order Section Filtering

Load \text{chirp}.mat. The file contains a signal, \( y \), that has most of its power above \( \text{Fs}/4 \), or half the Nyquist frequency. The sample rate is 8192 Hz.

\texttt{load chirp}

\texttt{t = (0:length(y)-1)/Fs;}

Design a seventh-order Butterworth highpass filter to attenuate the components of the signal below \( \text{Fs}/4 \). Use a normalized cutoff frequency of \( 0.48\pi \) rad/sample. Express the filter coefficients in terms of second-order sections.

\[ [\text{zhi},\text{phi},\text{khi}] = \text{butter}(7,0.48,'\text{high}'); \]
\[ \text{soshi} = \text{zp2sos(zhi,phi,khi)}; \]
\[ \text{freqz(soshi)} \]
Filter the signal. Display the original and highpass-filtered signals. Use the same y-axis scale for both plots.

```matlab
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')
```

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.

```matlab
[zlo, plo, klo] = butter(7, 0.48);
soslo = zp2sos(zlo, plo, klo);
outlo = sosfilt(soslo, y);
```

```matlab
subplot(2, 1, 1)
plot(t, y)
title('Original Signal')
```

```matlab
xlabel('Time (s)')
ylim(ys)
```

```matlab
subplot(2, 1, 2)
plot(t, outlo)
title('Highpass-Filtered 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: \([b,a] = \text{butter}(3,1/32); \text{sos} = \text{tf2sos}(b,a)\) specifies a third-order Butterworth filter with a normalized 3 dB frequency of \( \pi/32 \) rad/sample.

Data Types: \text{single} | \text{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: \text{single} | \text{double}
Complex Number Support: Yes

\(\text{dim}\) — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. By default, \(\text{sosfilt}\) operates along the first array dimension of \(x\) with size greater than 1.

Data Types: \text{single} | \text{double}

\section*{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


Extended Capabilities

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

See Also
filter | medfilt1 | sgolayfilt

Introduced before R2006a
spectrogram
Spectrogram using short-time Fourier transform

Syntax

\[
s = \text{spectrogram}(x)
\]
\[
s = \text{spectrogram}(x, \text{window})
\]
\[
s = \text{spectrogram}(x, \text{window}, \text{noverlap})
\]
\[
s = \text{spectrogram}(x, \text{window}, \text{noverlap}, \text{nfft})
\]

\[
[s, w, t] = \text{spectrogram}(\_\_)
\]
\[
[s, f, t] = \text{spectrogram}(\_\_, \text{fs})
\]

\[
[s, w, t] = \text{spectrogram}(x, \text{window}, \text{noverlap}, w)
\]
\[
[s, f, t] = \text{spectrogram}(x, \text{window}, \text{noverlap}, f, \text{fs})
\]

\[
[\_\_, \text{ps}] = \text{spectrogram}(\_\_)
\]
\[
[\_\_] = \text{spectrogram}(\_\_, \text{'reassigned'})
\]
\[
[\_\_, \text{ps}, \text{fc}, \text{tc}] = \text{spectrogram}(\_\_)
\]

\[
[\_\_] = \text{spectrogram}(\_\_, \text{freqrange})
\]
\[
[\_\_] = \text{spectrogram}(\_\_, \text{Name}, \text{Value})
\]
\[
[\_\_] = \text{spectrogram}(\_\_, \text{spectrumbtype})
\]

\[
\text{spectrogram}(\_\_)
\]
\[
\text{spectrogram}(\_\_, \text{freqloc})
\]

Description

\[
s = \text{spectrogram}(x) \text{ returns the short-time Fourier transform of the input signal, } x.
\]
Each column of \( s \) contains an estimate of the short-term, time-localized frequency content of \( x \).

\[
s = \text{spectrogram}(x, \text{window}) \text{ uses } \text{window} \text{ to divide the signal into segments and perform windowing.}
\]
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'.

1-2702
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.

\[
s = \text{spectrogram}(x);
\]

\[
\text{spectrogram}(x,'yaxis')
\]
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 $p = \lfloor \log_2 nsc \rfloor$.

Verify that the two approaches give identical results.

```matlab
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.

```matlab
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.

```matlab
fs = 1000;
t = 0:1/fs:2;
x = chirp(t,100,1,200,'quadratic');
```
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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 \( \left\lfloor \frac{100}{2} + 1 \right\rfloor = 51 \) frequencies.

```
spectrogram(y,100,80,100,fs,'yaxis')
```

Track the chirp frequency by finding the time-frequency ridge with highest energy across the \( \left\lfloor \frac{(2000 - 80)}{(100 - 80)} \right\rfloor = 96 \) time points. Overlay the instantaneous frequency on the spectrogram plot.

```
 [~,f,t,p] = spectrogram(y,100,80,100,fs);
[fridge,~,lr] = tfridge(p,f);
```
Spectrogram of Complex Signal

Generate 512 samples of a chirp with sinusoidally varying frequency content.

```matlab
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.
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.

```matlab
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 \( \lceil (\text{length}(x) - 120)/(128 - 120) \rceil = 235 \) 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.

```matlab
[~,~,~,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.

```matlab
plot(tc(pxx>0),fc(pxx>0),'.')
```
Spectrogram Reassignment and Thresholding

Generate a signal sampled at 1024 Hz for 2 seconds.

\[ n_{\text{samp}} = 2048; \]
\[ F_s = 1024; \]
\[ t = (0:n_{\text{samp}}-1)'/F_s; \]

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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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`.

```matlab
colormap bone
view(-45,65)
```
**Input Arguments**

- **x — Input signal**
  
  Vector

  Input signal, specified as a row or column vector.

  Example: \( \cos\left(\frac{\pi}{4} \cdot (0:159)\right) + \text{randn}(1,160) \) 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 \( \text{window} \) 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 \( N + 1 \).

Data Types: single | double

noverlap — Number of overlapped samples
positive integer | []

Number of overlapped samples, specified as a positive integer.

- If window is scalar, then noverlap must be smaller than window.
- If window is a vector, then noverlap must be smaller than the length of window.

If you specify noverlap as empty, then spectrogram uses a number that produces 50% overlap between segments. If the segment length is unspecified, the function sets noverlap to \( \lceil N_x/4.5 \rceil \), where \( N_x \) is the length of the input signal and the \([\] \) symbols denote the floor function.

Data Types: double | single

nfft — Number of DFT points
positive integer scalar | []
Number of DFT points, specified as a positive integer scalar. If you specify `nfft` as empty, then `spectrogram` sets the parameter to \( \text{max}(256,2^p) \), where \( p = \lceil \log_2 N_w \rceil \), the \( \lceil \rceil \) symbols denote the ceiling function, and

- \( N_w = \text{window} \) if `window` is a scalar.
- \( N_w = \text{length}(`\text{window}`) \) 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, because otherwise the function interprets it as `nfft`. Normalized frequencies are in rad/sample.

Example: `pi./[2 4]`
Data Types: `double` | `single`

`f` — Cyclical frequencies
vector

Cyclical frequencies, specified as a vector. `f` must have at least two elements, because otherwise the function interprets it as `nfft`. 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 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 comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes.
You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

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) \leq \text{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
  - \( k = \lceil (N_x - \text{nooverlap})/(\text{window} - \text{nooverlap}) \rceil \) if window is a scalar.
  - \( k = \lceil (N_x - \text{nooverlap})/(\text{length(window)} - \text{nooverlap}) \rceil \) if window is a vector.
- If x is real and nfft is even, then s has \( (\text{nfft}/2 + 1) \) rows.
- If x is real and nfft is odd, then s has \( (\text{nfft} + 1)/2 \) rows.
- If x is complex, then s has nfft rows.
s is not affected by the 'reassigned' option.

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.

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

**ps — Power spectral density or power spectrum**
matrix

Power spectral density (PSD) or power spectrum, returned as a matrix.

- If x is real, then ps contains the one-sided modified periodogram estimate of the PSD or power spectrum of each segment.
- If x is complex, 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.

Data Types: double | single

**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 \( \mathbf{fc} \) 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, \texttt{spectrogram} adds \texttt{eps} to the short-time Fourier transform when you call it with no output arguments.

**References**


**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” (MATLAB).

See Also

Apps
Signal Analyzer

Functions
goertzel | periodogram | pspectrum | pwelch

Topics
“Formant Estimation with LPC Coefficients”

Introduced before R2006a
**spectrum**

Spectral estimation

**Syntax**

\[ Hs = \text{spectrum}.	ext{estmethod}(\text{input1},...) \]

**Description**

**Note** The use of `spectrum.estmethod` is not recommended. Use the corresponding function instead. See Spectrum Estimation Methods for the functional forms.

\[ Hs = \text{spectrum}.	ext{estmethod}(\text{input1},...) \] returns a spectral estimation object \( Hs \) of type \text{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 \text{estmethod} with `spectrum`. 
## Spectrum Estimation Methods

<table>
<thead>
<tr>
<th>spectrum.estmethod</th>
<th>Description</th>
<th>Corresponding Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>spectrum.burg</td>
<td>Burg</td>
<td>pburg</td>
</tr>
<tr>
<td>spectrum.cov</td>
<td>Covariance</td>
<td>pcov</td>
</tr>
<tr>
<td>spectrum.eigenvector</td>
<td>Eigenvector</td>
<td>peig</td>
</tr>
<tr>
<td>spectrum.mcov</td>
<td>Modified covariance</td>
<td>pmcov</td>
</tr>
<tr>
<td>spectrum.mtm</td>
<td>Thompson multitaper</td>
<td>pmtm</td>
</tr>
<tr>
<td>spectrum.music</td>
<td>Multiple Signal Classification</td>
<td>pmusic</td>
</tr>
<tr>
<td>spectrum.periodogram</td>
<td>Periodogram</td>
<td>periodogram</td>
</tr>
<tr>
<td>spectrum.welch</td>
<td>Welch</td>
<td>pwelch</td>
</tr>
<tr>
<td>spectrum.yulear</td>
<td>Yule-Walker</td>
<td>pyulear</td>
</tr>
</tbody>
</table>

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 1-2748).

## 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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,n] if the number of FFT points (NFFT) is even, and [0,n) if NFFT is odd. For complex signals, the range of W is [0,2n). 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.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For example, ( \text{Hmss} = \text{msspectrum}(\text{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, ( \text{fvect} ).</td>
</tr>
<tr>
<td></td>
<td>( \text{msspectrum}(...) ) with no output arguments plots the mean-square spectrum in dB.</td>
</tr>
</tbody>
</table>
### 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.
**Method** | **Description**
--- | ---

\[ H_{mss} = \text{msspectrum}(\ldots, \text{'Centerdc'}, \text{true}) \] 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.

\[ \text{Hopts.FreqPoints} = \text{'User Defined'} \]

(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.

\[ \text{Hopts.FrequencyVector} = F \]

(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.)

\[ H_{mms} = \text{msspectrum}(\ldots, \text{'ConfLevel'}, p) \] specifies the confidence level \( p \) for computing the confidence interval, which is an estimate of the error in the calculated mean-squared spectrum. The confidence level (\( p \)) is between 0 and 1. For example,

\[ H_{mss} = \text{msspectrum}(Hs, X, \text{'ConfLevel'}, 0.95) \] returns the 95% confidence interval.
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.
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 psdopts 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.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pseudospectrum</td>
<td>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.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>pseudo-</td>
<td><strong>Hopts = pseudospectrumopts(Hs)</strong> returns an object that contains options for the spectrum object Hs.</td>
</tr>
<tr>
<td>spectrumopts</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Hopts = pseudospectrumopts(Hs,X)</strong> 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.</td>
</tr>
<tr>
<td></td>
<td><strong>Hmps = pseudospectrum (...,'SpectrumRange', 'whole')</strong> 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.</td>
</tr>
<tr>
<td></td>
<td><strong>Hmss = pseudospectrum(Hs,X,'NormalizedFrequency',true)</strong> returns a pseudospectrum object with frequency values normalized between 0 and 1. Default is true.</td>
</tr>
<tr>
<td></td>
<td><strong>Hps = pseudospectrum(Hs,X,'Fs',Fs)</strong> returns a pseudospectrum object computed as a function of frequency, where Fs is the sampling frequency in Hz.</td>
</tr>
<tr>
<td></td>
<td><strong>Hps = pseudospectrum(...,'NFFT',nfft)</strong> 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'.</td>
</tr>
<tr>
<td></td>
<td><strong>Hps = pseudospectrum(...,'Centerdc',true)</strong> shifts the data and frequency values so that the DC component is at the center of the spectrum. The default value is false.</td>
</tr>
<tr>
<td></td>
<td>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'</td>
</tr>
</tbody>
</table>

1-2747
### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Note that the default for FreqPoints is 'All', which causes pseudospectrum to use the NFFT property as described above.)</td>
</tr>
<tr>
<td>powerest</td>
<td>Note that powerest is available only for music and eigenvector spectrum objects.</td>
</tr>
</tbody>
</table>

```matlab
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.
```

```matlab
[POW,W]=powerest(Hs,X) returns POW and a vector W of the frequencies in rad/sample of the sinusoids in X.
```

```matlab
[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

```matlab
get(Hs,'property')
```

where 'property' is the specific property name.

To see all properties for an object, use

```matlab
get(Hs)
```

### Changing Object Properties

To set specific properties, use

```matlab
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.

```matlab
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.

```matlab
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

pburn | pcov | pmcov | pyulear

Introduced before R2006a
**spectrum.eigenvector**

Eigenvector spectrum

**Syntax**

```matlab
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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSinusoids</td>
<td>2</td>
<td>Number of complex sinusoids</td>
</tr>
<tr>
<td>SegmentLength</td>
<td>4</td>
<td>Length of each of the time-based segments into which the input signal is divided.</td>
</tr>
<tr>
<td>OverlapPercent</td>
<td>50</td>
<td>Percent overlap between segments</td>
</tr>
<tr>
<td>WindowName</td>
<td>'Rectangular'</td>
<td>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).</td>
</tr>
<tr>
<td>SubspaceThreshold</td>
<td>0</td>
<td>Threshold is the cutoff for signal and noise separation. The threshold is multiplied by $\lambda_{\text{min}}$, the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold ($\lambda_{\text{min}} \times \text{threshold}$) are assigned to the noise subspace.</td>
</tr>
<tr>
<td>InputType</td>
<td>'Vector'</td>
<td>Type of input that will be used with this spectrum object. Valid values are 'Vector', 'DataMatrix' and 'CorrelationMatrix'.</td>
</tr>
</tbody>
</table>
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.

```matlab
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


See Also

peig pmusic

Introduced before R2006a
**spectrum.mcov**

Modified covariance spectrum

**Syntax**

\[
\text{Hs} = \text{spectrum.mcov} \\
\text{Hs} = \text{spectrum.mcov(order)}
\]

**Description**

**Note** The use of \text{spectrum.mcov} is not recommended. Use \text{pmcov} instead.

\[
\text{Hs} = \text{spectrum.mcov}
\]

returns a default modified covariance spectrum object, \( \text{Hs} \), 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.

\[
\text{Hs} = \text{spectrum.mcov(order)}
\]

returns a spectrum object, \( \text{Hs} \) with the specified order. The default value for order is 4.

**Note** See \text{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.

\[
\begin{align*}
\text{x} &= \text{randn(100,1)}; \\
\text{x} &= \text{filter}(1,[1 \ 1/2 \ 1/3 \ 1/4 \ 1/5],\text{x}); \quad \% \ 4\text{th order AR filter} \\
\text{Hs} &= \text{spectrum.mcov}; \quad \% \ 4\text{th order AR model} \\
\text{psd}(\text{Hs},\text{x},'\text{NFFT}',512)
\end{align*}
\]
See Also
pburg | pcov | pmcov | pyulear

Introduced before R2006a
**spectrum.mtm**

Thomson multitaper spectrum

**Syntax**

\[
Hs = \text{spectrum.mtm}
\]

\[
Hs = \text{spectrum.mtm}(\text{TimeBW})
\]

\[
Hs = \text{spectrum.mtm}(	ext{DPSS},\text{Concentrations})
\]

\[
Hs = \text{spectrum.mtm}(\ldots,\text{CombineMethod})
\]

**Description**

**Note** The use of `spectrum.mtm` is not recommended. Use `pmtm` instead.

`Hs = \text{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:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeBW</td>
<td>4</td>
<td>Product of time and bandwidth for the discrete prolate spheroidal sequences (or Slepian sequences) used as data windows</td>
</tr>
<tr>
<td>CombineMethod</td>
<td>'adaptive'</td>
<td>Algorithm for combining the individual spectral estimates. Valid values are 'adaptive' — adaptive (nonlinear) 'unity' — unity weights (linear) 'eigenvector' — Eigenvalue weights (linear)</td>
</tr>
</tbody>
</table>
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.

```matlab
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**

```matlab
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.

The use of `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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSinusoids</td>
<td>2</td>
<td>Number of complex sinusoids</td>
</tr>
<tr>
<td>SegmentLength</td>
<td>4</td>
<td>Length of each of the time-based segments into which the input signal is divided.</td>
</tr>
<tr>
<td>OverlapPercent</td>
<td>50</td>
<td>Percent overlap between segments</td>
</tr>
<tr>
<td>WindowName</td>
<td>'Rectangular'</td>
<td>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).</td>
</tr>
<tr>
<td>SubspaceThreshold</td>
<td>0</td>
<td>Threshold is the cutoff for signal and noise separation. The threshold is multiplied by (\lambda_{\text{min}}), the smallest estimated eigenvalue of the signal's correlation matrix. Eigenvalues below the threshold ((\lambda_{\text{min}} \times \text{threshold})) are assigned to the noise subspace.</td>
</tr>
</tbody>
</table>
**Property Name** | **Default Value** | **Description**
---|---|---
InputType | 'Vector' | Type of input that will be used with this spectrum object. Valid values are 'Vector', 'DataMatrix' and 'CorrelationMatrix'.

$Hs = \text{spectrum.music}(\text{NSinusoids})$ 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.

$Hs = \text{spectrum.music}(\text{NSinusoids}, \text{SegmentLength})$ returns a spectrum object, $Hs$, with the specified segment length.

$Hs = \text{spectrum.music}(\text{NSinusoids}, \text{SegmentLength},... \text{OverlapPercent})$ returns a spectrum object, $Hs$, with the specified overlap between segments.

$Hs = \text{spectrum.music}(\text{NSinusoids}, \text{SegmentLength},... \text{OverlapPercent}, \text{WindowName})$ 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})

$Hs = \text{spectrum.music}(\text{NSinusoids}, \text{SegmentLength},... \text{OverlapPercent}, \text{WindowName}, \text{SubspaceThreshold})$ returns a spectrum object, $Hs$, with the specified subspace threshold.

$Hs = \text{spectrum.music}(\text{NSinusoids}, \text{SegmentLength},... \text{OverlapPercent}, \text{WindowName}, \text{SubspaceThreshold}, \text{InputType})$ 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.

\[ n = 0:99; \]
\[ s = \exp(1i\pi/2*n) + 2*\exp(1i\pi/4*n) + \exp(1i\pi/3*n) + \text{randn}(1,100); \]

\[ Hs = \text{spectrum.music}(3,20); \]
\[ \text{pseudospectrum}(Hs,s) \]
References


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.

```matlab
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


See Also
periodogram | pmem | 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.
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{WindowName,winparam}</td>
<td>'Hamming', SamplingFlag: symmetric</td>
<td>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.)</td>
</tr>
<tr>
<td>Property Name</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>WindowName</td>
<td>'Hamming',</td>
<td>Valid windowname options are:</td>
</tr>
<tr>
<td></td>
<td>symmetric</td>
<td>'Bartlett'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Bartlett-Hann'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Blackman'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Blackman-Harris'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Bohman'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Chebyshev'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Flat Top'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Gaussian'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Hamming'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Hann'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Kaiser'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Nuttall'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Parzen'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Rectangular'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Triangular'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Tukey'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'User Defined'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Window names must be enclosed in single quotes, such as <code>spectrum.welch('tukey')</code> or <code>spectrum.welch({'tukey',0.7})</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See <code>window</code> and the corresponding window function page for window parameter information. You can use <code>set</code> to change the value of the additional window parameter or to define the MATLAB expression and parameters for a user-defined window (see <code>spectrum</code> for information on using <code>set</code>).</td>
</tr>
<tr>
<td>Property Name</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SegmentLength</td>
<td>64</td>
<td>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.</td>
</tr>
<tr>
<td>OverlapPercent</td>
<td>50%</td>
<td>Percent overlap between segments</td>
</tr>
</tbody>
</table>

\[
Hs = \text{spectrum.welch(}\text{WindowName}) \text{ returns a spectrum object, } Hs, \text{ using Welch's method with the specified window and the default values for all other parameters. To specify parameters for a window, use a cell array formatted as } \text{spectrum.welch(}\{\text{WindowName,winparam}\}).\]

\[
Hs = \text{spectrum.welch(}\text{WindowName,SegmentLength}) \text{ returns a spectrum object, } Hs \text{ with the specified segment length.}\]

\[
Hs = \text{spectrum.welch(}\text{WindowName,SegmentLength,OverlapPercent}) \text{ returns a spectrum object, } Hs \text{ with the specified percentage overlap between segments.}\]

\textbf{Note} See \text{pwelch} for more information on the Welch algorithm.

\section*{Examples}

\subsection*{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


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
pb urg | p cov | p m cov | p yulear

Introduced before R2006a
sptool

Open interactive digital signal processing tool

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

Note  SPTool will be removed in a future release.

- 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.

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 1-2786
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 1-2787
- “Signal Display” on page 1-2789
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.

![Signal Browser Window](image)

- **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 1-2791 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:

- \( s = \text{sptool('Signals')} \) returns a structure array of all the signals.
- \( f = \text{sptool('Filters')} \) returns a structure array of all the filters.
- \( s = \text{sptool('Spectra')} \) returns a structure array of all the spectra.
- \([s,\text{ind}] = \text{sptool(____)}\) returns an index vector indicating which of the elements of \( s \) are currently selected in SPTool.
- \( s = \text{sptool(____,0)} \) 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:

- \( \text{struc} = \text{sptool('create',paramlist)} \) creates in the workspace a component structure, \( \text{struc} \), defined by \( \text{paramlist} \).
  \( \text{sptool('load',struc)} \) loads \( \text{struc} \) into SPTool.
- \( \text{struc} = \text{sptool('load',paramlist)} \) loads the component structure defined by \( \text{paramlist} \) 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 \( 0.5\pi \) rad/sample. Specify the filter in state-space representation, label it \textbf{Butterworth} within SPTool, and set it to filter digital signals sampled at 1 kHz.

\[
[z,p,k] = \text{butter}(5,0.5); \\
\text{struc} = \text{sptool('create','Filter','zpk',z,p,k,1e3,'Butterworth');} \\
\text{sptool('load',struc)}
\]
**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 sinusoidal frequency of $\pi/4$ rad/sample. Label the spectrum PSD within SPTool.

```matlab
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.

```matlab
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:

<table>
<thead>
<tr>
<th>Component</th>
<th>paramlist Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signals</td>
<td>component_name, data, fs, label</td>
</tr>
<tr>
<td>Filters</td>
<td>component_name, form, filter_params, fs, label</td>
</tr>
<tr>
<td>Spectra</td>
<td>component_name, data, f, label</td>
</tr>
</tbody>
</table>

The parameters are defined as follows:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>component_name</td>
<td>Specify as one of 'Signal', 'Filter', or 'Spectrum'. If omitted, component_name defaults to 'Signal'.</td>
</tr>
<tr>
<td>form</td>
<td>Form or structure of a filter. Specify as one of 'tf', 'ss', 'sos', or 'zpk'.</td>
</tr>
<tr>
<td>data</td>
<td>Vector of doubles representing a signal or spectrum.</td>
</tr>
<tr>
<td>filter_params</td>
<td>Filter representation.</td>
</tr>
<tr>
<td></td>
<td>• Specify num and den when form is 'tf'.</td>
</tr>
<tr>
<td></td>
<td>• Specify an SOS matrix when form is 'sos'.</td>
</tr>
<tr>
<td></td>
<td>• Specify z, p, and k when form is 'zpk'.</td>
</tr>
<tr>
<td></td>
<td>• Specify A, B, C, and D when form is 'ss'.</td>
</tr>
<tr>
<td>fs</td>
<td>Optional parameter that specifies the sample rate. If omitted, fs defaults to 1.</td>
</tr>
<tr>
<td>f</td>
<td>Frequency vector. This parameter applies only if component_name is 'Spectrum'.</td>
</tr>
<tr>
<td>label</td>
<td>Optional parameter that specifies the variable name of the component within SPTool. If omitted, label defaults to:</td>
</tr>
<tr>
<td></td>
<td>• 'sig' if component_name is 'Signal'</td>
</tr>
<tr>
<td></td>
<td>• 'filt' if component_name is 'Filter'</td>
</tr>
<tr>
<td></td>
<td>• 'spec' if component_name is 'Spectrum'</td>
</tr>
</tbody>
</table>
See Also

Apps
Filter Designer | Signal Analyzer

Functions
FVTool | findpeaks

Introduced before R2006a
square

Square wave

Syntax

\[ x = \text{square}(t) \]
\[ x = \text{square}(t,duty) \]

Description

\[ x = \text{square}(t) \] generates a square wave with period \( 2\pi \) for the elements of the time array \( t \). \text{square} is similar to the \text{sine} function but creates a square wave with values of \(-1\) and \(1\).

\[ x = \text{square}(t,duty) \] generates a square wave with specified duty cycle \( duty \). The \textit{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 \(3\pi\). Generate a square wave with a period of \(2\pi\).

\[ t = \text{linspace}(0,3\pi)'; \]
\[ x = \text{square}(t); \]

Plot the square wave and overlay a sine. Normalize the x-axis by \( \pi \). 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.

\[ \text{plot}(t/pi,x,'.-',t/pi,\text{sin}(t)) \]
\[ \text{xlabel('t / \pi')} \]
\[ \text{grid on} \]
Repeat the calculation, but now evaluate \texttt{\texttt{square}(2*t)} at 121 equally spaced numbers between $-\pi$ and $2\pi$. 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 $2\pi$.

\begin{verbatim}
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
\end{verbatim}
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.

\[
t = 0:1/1e3:0.07; \\
y = \text{square}(2\pi\times30\times t, 37) + \text{randn(size(t))}/10;
\]

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
Convert digital filter to state-space representation

**Syntax**

\[ [A,B,C,D] = \text{ss}(d) \]

**Description**

\[ [A,B,C,D] = \text{ss}(d) \] converts a digital filter, \( d \), to its state-space representation.

The state-space representation of a filter is given by

\[
\begin{align*}
    x(k + 1) &= Ax(k) + Bu(k), \\
    y(k) &= Cx(k) + Du(k),
\end{align*}
\]

where \( x \) is the state vector, \( u \) is the input vector, and \( y \) is the output vector.

**Examples**

**State-Space Representation of a Lowpass IIR Filter**

Design a lowpass IIR filter of order 6. Specify a normalized passband frequency of \( 0.2\pi \) rad/sample. Compute the state-space representation of the filter.

\[
\begin{align*}
    d &= \text{designfilt('lowpassiir','FilterOrder',6,'PassbandFrequency',0.2);} \\
    [A,B,C,D] &= \text{ss}(d)
\end{align*}
\]

\[
A = \begin{bmatrix}
    1.5640 & -0.9294 & 0 & 0 & 0 & 0 \\
    1.0000 & 0 & 0 & 0 & 0 & 0 \\
    0.1795 & 0.0036 & 1.6097 & -0.8112 & 0 & 0 \\
    0 & 0 & 1.0000 & 0 & 0 & 0 \\
    0.0020 & 0.0000 & 0.0408 & 0.0021 & 1.6956 & -0.7409
\end{bmatrix}
\]

1-2809
B = \begin{bmatrix} 
0.0913 \\
0 \\
0.0046 \\
0 \\
0.0001 \\
0 
\end{bmatrix}

C = \begin{bmatrix} 
0.0020 & 0.0000 & 0.0408 & 0.0021 & 3.6956 & 0.2591 
\end{bmatrix}

D = 5.2030e-05

### Input Arguments

d — Digital filter
digitalFilter object

Digital filter, specified as a digitalFilter object. Use designfilt to generate a
digital filter based on frequency-response specifications.

Example: d =
designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)
specifies a third-order Butterworth filter with normalized 3 dB frequency 0.5\pi rad/sample.

### Output Arguments

A — State matrix
matrix

State matrix, returned as a matrix.

Data Types: double
**B — Input-to-state matrix**
matrix

Input-to-state matrix, returned as a matrix.
Data Types: double

**C — State-to-output matrix**
matrix

State-to-output matrix, returned as a matrix.
Data Types: double

**D — Feedthrough matrix**
matrix

Feedthrough matrix, returned as a matrix.
Data Types: double

**See Also**
designfilt | digitalFilter | tf | zpk

**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**

*ss2sos* converts a state-space representation of a given digital filter to an equivalent second-order section representation.

\[ [sos,g] = \text{ss2sos}(A,B,C,D) \]

finds a matrix \( sos \) in second-order section form with gain \( g \) that is equivalent to the state-space system represented by input arguments \( A, B, C, \) and \( D \).

**Note** The input state-space system must be single-output and real.

\( sos \) is an \( L \)-by-6 matrix

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

whose rows contain the numerator and denominator coefficients \( b_{ik} \) and \( a_{ik} \) of the second-order sections of \( H(z) \).
\[
H(z) = g \prod_{k=1}^{L} H_k(z) = g \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}
\]

\[ [\text{sos}, g] = \text{ss2sos}(A, B, C, D, \text{iu}) \]

specifies a scalar \( \text{iu} \) that determines which input of the state-space system \( A, B, C, D \) is used in the conversion. The default for \( \text{iu} \) is 1.

\[ [\text{sos}, g] = \text{ss2sos}(A, B, C, D, 'order') \]

and

\[ [\text{sos}, g] = \text{ss2sos}(A, B, C, D, \text{iu}, 'order') \]

specify the order of the rows in sos, where 'order' is

- 'down', to order the sections so the first row of sos contains the poles closest to the unit circle
- 'up', to order the sections so the first row of sos contains the poles farthest from the unit circle (default)

The zeros are always paired with the poles closest to them.

\[ [\text{sos}, g] = \text{ss2sos}(A, B, C, D, \text{iu}, 'order', 'scale') \]

specifies the desired scaling of the gain and the numerator coefficients of all second-order sections, where 'scale' is

- 'none', to apply no scaling (default)
- 'inf', to apply infinity-norm scaling
- 'two', to apply 2-norm scaling

Using infinity-norm scaling in conjunction with up-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling in conjunction with down-ordering minimizes the peak round-off noise.

**Note** Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

\[ \text{sos} = \text{ss2sos}(\ldots) \]

embeds the overall system gain, \( g \), in the first section, \( H_1(z) \), so that

\[
H(z) = \prod_{k=1}^{L} H_k(z)
\]
Embedding the gain in the first section when scaling a direct-form II structure is not recommended and may result in erratic scaling. To avoid embedding the gain, use `ss2sos` with two outputs.

### Examples

#### Second-Order Section Form of a Filter

Design a 5th-order Butterworth lowpass filter using the `butter` function. Specify a cutoff frequency of $0.2\pi$ rad/sample. Express the output in state-space form. Convert the state-space result to second-order sections. Visualize the frequency response of the filter.

```matlab
[A,B,C,D] = butter(5,0.2);
sos = ss2sos(A,B,C,D)
sos = 3x6
    0.0013    0.0013         0    1.0000   -0.5095         0
    1.0000    2.0027    1.0027    1.0000   -1.0966    0.3554
    1.0000    1.9989    0.9990    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

$$A = \begin{bmatrix} \cos\Delta t & \sin\Delta t \\ -\sin\Delta t & \cos\Delta t \end{bmatrix}, \quad B = \begin{bmatrix} 1 - \cos\Delta t \\ \sin\Delta t \end{bmatrix}, \quad C = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \quad D = 1.$$

$A = [\cos(dt) \ \sin(dt); -\sin(dt) \ \cos(dt)];$
$B = [1-cos(dt);sin(dt)];$
$C = [-1 0];$
$D = 1;$

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.

$u = [1 \ \text{zeros}(1,N-1)];$
$x = [0;0];$
\textbf{for} \ k = 1:N
    \quad y(k) = C*x + D*u(k);
    \quad x = A*x + B*u(k);$
\textbf{end}
Plot the acceleration of the mass as a function of time.

```matlab
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.

```matlab
sos = ss2sos(A,B,C,D);
yt = sosfilt(sos,u);
stem(t,yt,'filled')
```
The result is the same in both cases.

**Algorithms**

`ss2sos` uses a four-step algorithm to determine the second-order section representation for an input state-space system:

1. It finds the poles and zeros of the system given by A, B, C, and D.
2. It uses the function `zp2sos`, which first groups the zeros and poles into complex conjugate pairs using the `cplxpair` function. `zp2sos` then forms the second-order sections by matching the pole and zero pairs according to the following rules:
a  Match the poles closest to the unit circle with the zeros closest to those poles.
b  Match the poles next closest to the unit circle with the zeros closest to those poles.
c  Continue until all of the poles and zeros are matched.

ss2sos groups real poles into sections with the real poles closest to them in absolute value. The same rule holds for real zeros.

3 It orders the sections according to the proximity of the pole pairs to the unit circle. ss2sos normally orders the sections with poles closest to the unit circle last in the cascade. You can tell ss2sos to order the sections in the reverse order by specifying the 'down' flag.

4 ss2sos scales the sections by the norm specified in the 'scale' argument. For arbitrary $H(\omega)$, the scaling is defined by

$$\|H\|_p = \left[ \frac{1}{2\pi} \int_0^{2\pi} |H(\omega)|^p d\omega \right]^{1/p}$$

where $p$ can be either $\infty$ or 2. See the references for details. This scaling is an attempt to minimize overflow or peak round-off noise in fixed point filter implementations.

References


See Also
cplxpair | sos2ss | ss2tf | ss2zp | tf2sos | zp2sos
Introduced before R2006a
**ss2zp**

Convert state-space filter parameters to zero-pole-gain form

**Syntax**

\[ [z,p,k] = \text{ss2zp}(A,B,C,D) \]
\[ [z,p,k] = \text{ss2zp}(A,B,C,D,ni) \]

**Description**

\[ [z,p,k] = \text{ss2zp}(A,B,C,D) \]

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.

\[ [z,p,k] = \text{ss2zp}(A,B,C,D,ni) \]

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:

\[ b = [2 \ 3 \ 0]; \]
\[ a = [1 \ 0.4 \ 1]; \]
\[ [z, p, k] = \text{tf2zp}(b, a) \]

\[ z = 2 \times 1 \]
\[ \begin{array}{c}
0 \\
-1.5000
\end{array} \]

\[ p = 2 \times 1 \text{ complex} \]
\[ \begin{array}{c}
-0.2000 + 0.9798i \\
-0.2000 - 0.9798i
\end{array} \]

\[ k = 2 \]

Express the system in state-space form and determine the zeros, poles, and gain using \text{ss2zp}.

\[ [A, B, C, D] = \text{tf2ss}(b, a); \]
\[ [z, p, k] = \text{ss2zp}(A, B, C, D, 1) \]

\[ z = 2 \times 1 \]
\[ \begin{array}{c}
0 \\
-1.5000
\end{array} \]

\[ p = 2 \times 1 \text{ complex} \]
\[ \begin{array}{c}
-0.2000 + 0.9798i \\
-0.2000 - 0.9798i
\end{array} \]

\[ k = 2 \]
**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 \( ni = 1, \ldots, r \) 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:

\[ z = \text{eig}([A B; C D], \text{diag}([\text{ones}(1,n) 0])); \]

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


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,HISTOGRAM] = statelevels(____)
[LEVELS,HISTOGRAM,BINLEVELS] = statelevels(____)
statelevels(____)

Description

LEVELS = statelevels(X) estimates the low and high state levels in the bilevel waveform X using the histogram method. See “Algorithms” on page 1-2835.

LEVELS = statelevels(X,NBINS) specifies the number of bins to use in the histogram as a positive scalar. If unspecified, NBINS defaults to 100.

LEVELS = statelevels(X,NBINS,METHOD) estimates state levels using the mean or mode of the subhistograms. Valid entries for METHOD are 'mean' and 'mode'. METHOD defaults to 'mode'.

[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)
```

![](signal.png)

![](histogram.png)

```matlab
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.

```matlab
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)`
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.

```matlab
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.

```matlab
load('clockex.mat','x','t')
[levs,histog,bilevs] = statelevels(x,4)
```

```matlab
levs = 1×2
  0.2427    2.0428
```

```matlab
histog = 4×1
  50
  0
  0
  50
```

```matlab
bilevs = 4×1
  0.2427
  0.8427
  1.4428
  2.0428
```

Plot the waveform and annotate the levels.

```matlab
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 1-2835.

**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**

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the $\alpha \%$ region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1),$$

where $S_1$ is the low-state level and $S_2$ is the high-state level. Replace the first term in the equation with $S_2$ to obtain the $\alpha \%$ tolerance region for the high state.

The following figure illustrates lower and upper 2\% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.
Algorithms

`statelevels` uses the histogram method to estimate the states of a bilevel waveform. The histogram method is described in [1] on page 1-2836. 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, \( i_{\text{low}} \), and highest-indexed histogram bin, \( i_{\text{high}} \), with nonzero counts.
5 Divide the histogram into two subhistograms:

The indices of the lower histogram bins are $i_{\text{low}} \leq i \leq \frac{1}{2}(i_{\text{high}} - i_{\text{low}})$.

The indices of the upper histogram bins are $i_{\text{low}} + \frac{1}{2}(i_{\text{high}} - i_{\text{low}}) \leq i \leq i_{\text{high}}$.

6 Compute the state levels by determining the mode or mean of the lower and upper histograms.

References


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 of the filter with numerator coefficients, b, and denominator coefficients, a. stepz chooses the number of samples and returns the response in the column vector h and sample times in the column vector t (where t = [0:n-1]', and n = length(t) is computed automatically).

[h,t] = stepz(sos) returns the step response of the second order sections matrix, sos. sos is a K-by-6 matrix, where the number of sections, K, must be greater than or equal to 2. If the number of sections is less than 2, stepz considers the input to be the numerator vector, b. Each row of sos corresponds to the coefficients of a second-order (biquad) filter. The ith row of the sos matrix corresponds to [bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)].

[h,t] = stepz(d) returns the step response of the digital filter, d. Use designfilt to generate d based on frequency-response specifications.

[h,t] = stepz(...,n) computes the first n samples of the step response when n is an integer (t = [0:n-1]'). If n is a vector of integers, the step response is computed only at those integer values with 0 denoting the time origin.

[h,t] = stepz(...,n,fs) computes n samples and produces a vector t of length n so that the samples are spaced 1/fs units apart. fs is assumed to be in Hz.
stepz(...) with no output arguments plots the step response of the filter. If you input the filter coefficients or second order sections matrix, the current figure window is used. If you input a digitalFilter, the step response is displayed in FVTool.

**Note** If the input to stepz is single precision, the step response is calculated using single-precision arithmetic. The output, h, is single precision.

**Examples**

**Step Response of a Butterworth Filter**

Create a third-order Butterworth filter with normalized half-power frequency 0.4π 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`.

```matlab
d = designfilt('lowpassiir', 'FilterOrder', 3, 'HalfPowerFrequency', 0.4);
stepz(d)
```
Step Response of an Elliptic Filter

Design a fourth-order lowpass elliptic filter with normalized passband frequency $0.4\pi$ rad/sample. Specify a passband ripple of 0.5 dB and a stopband attenuation of 20 dB. Plot the first 50 samples of the filter's step response.

```matlab
[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`.

```matlab
d = designfilt('lowpassiir','FilterOrder',4,'PassbandFrequency',0.4, ... 'PassbandRipple',0.5,'StopbandAttenuation',20, ... 'DesignMethod','ellip');
stepz(d,50)
```
 Algorithms

stepz filters a length n step sequence using

\[
\text{filter}(b,a,\text{ones}(1,n))
\]

and plots the results using \text{stem}.

To compute n in the auto-length case, \text{stepz} either uses \( n = \text{length}(b) \) for the FIR case or first finds the poles using \( p = \text{roots}(a) \), if \text{length}(a) is greater than 1.
If the filter is unstable, n is chosen to be the point at which the term from the largest pole reaches $10^6$ times its original value.

If the filter is stable, n is chosen to be the point at which the term due to the largest amplitude pole is $5 \times 10^{-5}$ of its original amplitude.

If the filter is oscillatory (poles on the unit circle only), stepz computes five periods of the slowest oscillation.

If the filter has both oscillatory and damped terms, n is chosen to equal five periods of the slowest oscillation or the point at which the term due to the pole of largest nonunit amplitude is $5 \times 10^{-5}$ times its original amplitude, whichever is greater.

stepz also allows for delays in the numerator polynomial. The number of delays is incorporated into the computation for the number of samples.

**See Also**
designfilt | digitalFilter | freqz | grpdelay | impz | phasez | zplane

**Introduced before R2006a**
**stft**

Short-time Fourier transform

**Syntax**

\[
s = \text{stft}(x) \\
s = \text{stft}(x,fs) \\
s = \text{stft}(x,ts) \\
s = \text{stft}(\_,\text{Name,Value}) \\
[s,f] = \text{stft}(\_) \\
[s,f,t] = \text{stft}(\_) \\
\text{stft}(\_)
\]

**Description**

\( s = \text{stft}(x) \) returns the short-time Fourier transform (STFT) of \( x \).

\( s = \text{stft}(x,fs) \) returns the STFT of \( x \) using sample rate \( fs \).

\( s = \text{stft}(x,ts) \) returns the STFT of \( x \) using sample time \( ts \).

\( s = \text{stft}(\_,\text{Name,Value}) \) 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.

\( [s,f] = \text{stft}(\_) \) returns the frequencies \( f \) at which the STFT is evaluated.

\( [s,f,t] = \text{stft}(\_) \) returns the times at which the STFT is evaluated.

\( \text{stft}(\_) \) with no output arguments plots the magnitude of the STFT in the current figure window. The STFT is plotted as two-sided and centered.
Examples

3D STFT Visualization

Generate two seconds of a voltage controlled oscillator output, controlled by a sinusoid sampled at 10 kHz.

```matlab
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.

```matlab
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`.

```matlab
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.

```matlab
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 of Multichannel Signals**

Generate a three-channel signal consisting of three different chirps sampled at 1 kHz for one second.
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.

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.

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.

```matlab
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(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.

```matlab
function helperGraphicsOpt(ChannelId)
ax = gca;
ax.XDir = 'reverse';
str = ['Input Channel: ',num2str(ChannelId)];
ax.Title.String = str;
ax.YLabel.String = 'Frequency (Hz)';
ax.XLabel.String = 'Time (S)';
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 \((\text{length}(x) - \text{noverlap}) / (\text{length}(\text{window}) - \text{noverlap})\) 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” (MATLAB).
- 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 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**

2\(\pi\) (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 comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: stft('Window',win,'OverlapLength',50,'FFTLength',128) windows the data using the window win, with 50 samples overlap between adjoining segments and 128 point FFT.

**Window — Spectral window**
hann(128,'periodic') (default) | vector

Spectral window, 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 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. 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

**Centered — Frequency range**

true (default) | false

Frequency range, specified as the comma-separated pair consisting of Centered and true or false. If this option is set to true, then the spectrum is centered and is computed over the interval $-\pi$ to $\pi$. Otherwise, the spectrum is computed over the interval 0 to $2\pi$.

**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 $s$ down the rows and the frequency dimension along the columns. Set this value to acrosscolumns, if you want the time dimension of $s$ 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 | 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 $k = \lceil (N_x-L)/(M-L) \rceil$, $M$ is the length of 'Window', $L$ is the 'OverlapLength', and the $\lceil \rceil$ 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 rows in the STFT matrix equals the number of DFT points, and the number of columns 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 STFT matrix is given by \( \mathbf{X}(f) = [X_1(f) \ X_2(f) \ X_3(f) \ \cdots \ X_k(f)] \) such that the \( m \)th 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 \( k = \frac{\text{length}(x) - \text{noverlap}}{\text{length}(\text{window}) - \text{noverlap}} \) 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**


**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 or tall timetable with a single-channel signal.

• **OutputTimeDimension** must be always specified and set to 'downrows'.

For more information, see “Tall Arrays” (MATLAB).

**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.
See Also

Functions
iscola | istft | pspectrum

Introduced in R2019a
stmcb

Compute linear model using Steiglitz-McBride iteration

Syntax

[b,a] = stmcb(h,nb,na)
[b,a] = stmcb(y,x,nb,na)
[b,a] = stmcb(h,nb,na,niter)
[b,a] = stmcb(y,x,nb,na,niter)
[b,a] = stmcb(h,nb,na,niter,ai)
[b,a] = stmcb(y,x,nb,na,niter,ai)

Description

Steiglitz-McBride iteration is an algorithm for finding an IIR filter with a prescribed time-domain impulse response. It has applications in both filter design and system identification (parametric modeling).

[b,a] = stmcb(h,nb,na) finds the coefficients b and a of the system b(z)/a(z) with approximate impulse response h, exactly nb zeros, and exactly na poles.

[b,a] = stmcb(y,x,nb,na) finds the system coefficients b and a of the system that, given x as input, has y as output. x and y must be the same length.

[b,a] = stmcb(h,nb,na,niter) and
[b,a] = stmcb(y,x,nb,na,niter) use niter iterations. The default for niter is 5.

[b,a] = stmcb(h,nb,na,niter,ai) and
[b,a] = stmcb(y,x,nb,na,niter,ai) use the vector ai as the initial estimate of the denominator coefficients. If ai is not specified, stmcb uses the output argument from [b,ai] = prony(h,0,na) as the vector ai.

stmcb returns the IIR filter coefficients in length nb+1 and na+1 row vectors b and a. The filter coefficients are ordered in descending powers of z.
\[
H(z) = \frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(nb + 1)z^{-nb}}{a(1) + a(2)z^{-1} + \cdots + a(na + 1)z^{-na}}
\]

**Examples**

**Steiglitz-McBride Approximation of a Filter**

Approximate the impulse response of an IIR filter with a system of lower order.

Specify a 6th-order Butterworth filter with normalized 3-dB frequency \(0.2\pi\) rad/sample.

```matlab
d = designfilt('lowpassiir','FilterOrder',6, ... 
  'HalfPowerFrequency',0.2,'DesignMethod','butter');
```

Use the Steiglitz-McBride iteration to approximate the filter with a 4th-order system.

```matlab
h = impz(d);
[bb,aa] = stmcb(h,4,4);
```

Plot the frequency responses of the two systems.

```matlab
hfvt = fvtool(d,bb,aa,'Analysis','freq');
legend(hfvt,'Butterworth','Steiglitz-McBride')
```
Diagnostics

If \( x \) and \( y \) have different lengths, \texttt{stmcb} produces this error message:

Input signal \( X \) and output signal \( Y \) must have the same length.
Algorithms

stmcb attempts to minimize the squared error between the impulse response \( h \) of \( b(z)/a(z) \) and the input signal \( x \).

\[
\min_{a, b} \sum_{i=0}^{\infty} |x(i) - h(i)|^2
\]

stmcb iterates using two steps:

1. It prefilters \( h \) and \( x \) using \( 1/a(z) \).
2. It solves a system of linear equations for \( b \) and \( a \) using \( \backslash \).

stmcb repeats this process \( niter \) times. No checking is done to see if the \( b \) and \( a \) coefficients have converged in fewer than \( niter \) iterations.

References


See Also

aryule | levinson | lpc | prony

Introduced before R2006a
strips

Strip plot

**Syntax**

\[
\begin{align*}
\text{strips}(x) \\
\text{strips}(x,n) \\
\text{strips}(x,\text{sd},\text{fs}) \\
\text{strips}(x,\text{sd},\text{fs},\text{scale})
\end{align*}
\]

**Description**

\text{strips}(x) \text{ plots vector } x \text{ in horizontal strips of length 250. If } x \text{ is a matrix, } \text{strips}(x) \text{ plots each column of } x. \text{ The left-most column (column 1) is the top horizontal strip.}

\text{strips}(x,n) \text{ plots vector } x \text{ in strips that are each } n \text{ samples long.}

\text{strips}(x,\text{sd},\text{fs}) \text{ plots vector } x \text{ in strips of duration } \text{sd} \text{ seconds, given a sampling frequency of } \text{fs} \text{ samples per second.}

\text{strips}(x,\text{sd},\text{fs},\text{scale}) \text{ scales the vertical axes.}

If \(x\) is a matrix, \text{strips}(x,n), \text{strips}(x,\text{sd},\text{fs}), \text{ and } \text{strips}(x,\text{sd},\text{fs},\text{scale}) \text{ plot the different columns of } x \text{ on the same strip plot.}

\text{strips} \text{ ignores the imaginary part of complex-valued } x.

**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.
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
Plot the signal in 0.18 second long strips.

`strips(mtlb, 0.18, Fs)`

See Also

`plot`, `stem`

Introduced before R2006a
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.

\[
\text{lssnew} = \text{subset}(\text{lss},2)
\]

\[
\text{lssnew} =
\text{labeledSignalSet with properties:}
\]

- \( \text{Source: \{[76579x1 double]\}} \)
- \( \text{NumMembers: 1} \)
- \( \text{TimeInformation: "sampleRate"} \)
- \( \text{SampleRate: 4000} \)
- \( \text{Labels: \{1x3 table\}} \)
- \( \text{Description: "Characterize wave song regions"} \)

Use \text{labelDefinitionsHierarchy} to see a list of labels and sublabels. Use \text{setLabelValue} to add data to the set.

**Input Arguments**

- **lss** — Labeled signal set
  
  \text{labeledSignalSet} object

  Labeled signal set, specified as a \text{labeledSignalSet} object.

  Example: \text{labeledSignalSet}({\text{randn}(100,1)} \text{randn}(10,1)},\text{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 \text{midxvect} specifies a member row number as it appears in the “Labels” on page 1-0 table of the \text{labeledSignalSet} object \text{lss}.

  Example: \text{[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
taylorwin

Taylor window

Syntax

\[
w = \text{taylorwin}(L) \\
w = \text{taylorwin}(L,nbar) \\
w = \text{taylorwin}(L,nbar,sll)
\]

Description

\( w = \text{taylorwin}(L) \) returns an \( L \)-point Taylor window.

\( w = \text{taylorwin}(L,nbar) \) returns an \( L \)-point Taylor window with a number (\( \text{nbar} \)) of nearly constant-level sidelobes adjacent to the mainlobe.

\( w = \text{taylorwin}(L,nbar,sll) \) returns an \( L \)-point Taylor window with a maximum sidelobe level of \( \text{sll} \) 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 \text{wvtool}.

\[
w = \text{taylorwin}(64,4,-35); \\
\text{wvtool}(w)
\]
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**


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 in R2006a
**tachorpm**

Extract RPM signal from tachometer pulses

**Syntax**

```matlab
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.

```matlab
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.

```matlab
tachorpm(Yn,fs,'FitPoints',600)
axis([0.47 0.65 1320 1570])
```
Choose a moderate number of points to obtain a better result.

tachorm(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.

```matlab
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);
```
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)))
```matlab
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.
Data Types: double | single

**fs — Sample rate**
positive scalar

Sample rate, specified as a positive scalar expressed in Hz.

Data Types: double | single

### Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name,Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as **Name1,Value1,...,NameN,ValueN**.

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.

Data Types: double | single

**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 **tachrpm** computes the levels automatically using the histogram method, as in the **statelevels** function.

Data Types: double | single

**OutputFs — Output sample rate**
fs (default) | real scalar
Output sample rate, specified as the comma-separated pair consisting of 'OutputFs' and a real scalar.

Data Types: double | single

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.

Data Types: single | double

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'.

Data Types: double | single

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 $\text{RPM} = \frac{60}{\Delta t}$.
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**


**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 = 1x7

0.0003  -0.0019    0.0048   -0.0064    0.0048   -0.0019    0.0003

a = 1x7

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

\[
\begin{align*}
[sos,g] &= \text{tf2sos}(b,a) \\
[sos,g] &= \text{tf2sos}(b,a,'order') \\
[sos,g] &= \text{tf2sos}(b,a,'order','scale') \\
sos &= \text{tf2sos}(...) 
\end{align*}
\]

Description

tf2sos converts a transfer function representation of a given digital filter to an equivalent second-order section representation.

\[
[sos,g] = \text{tf2sos}(b,a)
\]

finds a matrix \(sos\) in second-order section form with gain \(g\) that is equivalent to the digital filter represented by transfer function coefficient vectors \(a\) and \(b\).

\[
H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \cdots + b_{n+1} z^{-n}}{a_1 + a_2 z^{-1} + \cdots + a_{m+1} z^{-m}}
\]

\(sos\) is an \(L\)-by-6 matrix

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

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

\[
H(z) = g \prod_{k=1}^{L} H_k(z) = g \prod_{k=1}^{L} \frac{b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 + a_{1k} z^{-1} + a_{2k} z^{-2}}.
\]
[sos,g] = tf2sos(b,a,'order') specifies the order of the rows in sos, where 'order' is:

- 'down', to order the sections so the first row of sos contains the poles closest to the unit circle
- 'up', to order the sections so the first row of sos contains the poles farthest from the unit circle (default)

[sos,g] = tf2sos(b,a,'order','scale') specifies the desired scaling of the gain and numerator coefficients of all second-order sections, where 'scale' is:

- 'none', to apply no scaling (default)
- 'inf', to apply infinity-norm scaling
- 'two', to apply 2-norm scaling

Using infinity-norm scaling in conjunction with up-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling in conjunction with down-ordering minimizes the peak round-off noise.

**Note** Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

`sos = tf2sos(...)` embeds the overall system gain, g, in the first section, \( H_1(z) \), so that

\[
H(z) = \prod_{k=1}^{L} H_k(z)
\]

**Note** Embedding the gain in the first section when scaling a direct-form II structure is not recommended and may result in erratic scaling. To avoid embedding the gain, use `ss2sos` with two outputs.

**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.

```matlab
[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

denums = [conv(ss(1,4:6),ss(2,4:6)) ;dn]
denums = 2×5
  1.0000    0.0000    0.4860    0.0000    0.0177
  1.0000    0.0000    0.4860    0.0000    0.0177
```

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:
   - Match the poles closest to the unit circle with the zeros closest to those poles.
   - Match the poles next closest to the unit circle with the zeros closest to those poles.
   - Continue until all of the poles and zeros are matched.
tf2sos groups real poles into sections with the real poles closest to them in absolute value. The same rule holds for real zeros.

3 It orders the sections according to the proximity of the pole pairs to the unit circle. tf2sos normally orders the sections with poles closest to the unit circle last in the cascade. You can tell tf2sos to order the sections in the reverse order by specifying the 'down' flag.

4 tf2sos scales the sections by the norm specified in the 'scale' argument. For arbitrary $H(\omega)$, the scaling is defined by

$$\|H\|_p = \left[ \frac{1}{2\pi} \int_0^{2\pi} |H(\omega)|^p d\omega \right]^{1/p}$$

where $p$ can be either $\infty$ or 2. See the references for details on the scaling. This scaling is an attempt to minimize overflow or peak round-off noise in fixed point filter implementations.

References


See Also
cplxpair | sos2tf | ss2sos | tf2ss | tf2zp | tf2zpk | zp2sos

Introduced before R2006a
tf2ss

Convert transfer function filter parameters to state-space form

Syntax

\[ [A,B,C,D] = \text{tf2ss}(b,a) \]

Description

\[ [A,B,C,D] = \text{tf2ss}(b,a) \] 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

\[
H(s) = \frac{2s + 3}{s^2 + 2s + 1}
\]

\[
= \frac{s^2 + 0.4s + 1}{s(s + 0.4)}
\]

Convert it to state-space form using tf2ss.

\[
b = \begin{bmatrix} 0 & 2 & 3 \\ 1 & 2 & 1 \end{bmatrix};
\]

\[
a = \begin{bmatrix} 1 & 0.4 & 1 \end{bmatrix};
\]

\[
[A,B,C,D] = \text{tf2ss}(b,a)
\]

\[
A = 2\times2
\]

\[
\begin{bmatrix}
-0.4000 & -1.0000 \\
1.0000 & 0
\end{bmatrix}
\]
**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 \( \Delta t = 1/F_s \).

\[
\begin{align*}
B & = 2 \times 1 \\
& = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\
C & = 2 \times 2 \\
& = \begin{bmatrix} 2.0000 & 3.0000 \\ 1.6000 & 0 \end{bmatrix} \\
D & = 2 \times 1 \\
& = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\
\end{align*}
\]

\[
\begin{align*}
{F_s} & = 5; \\
{dt} & = 1/F_s; \\
N & = 50; \\
t & = {dt}*(0:N-1); \\
u & = [1 \ zeros(1,N-1)]; \\
\end{align*}
\]

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.

\[
bf = [1 -(1+\cos(dt)) \cos(dt)]; \\
af = [1 -2*\cos(dt) 1]; \\
yf = \text{filter}(bf,af,u); \\
\text{stem}(t,yf,'o') \\
\text{xlabel('t')}
\]
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.

\[
[A,B,C,D] = tf2ss(bf,af);
\]

\[
x = [0;0];
\]

\[
for \ k = 1:N
    \quad y(k) = C*x + D*u(k);
    \quad x = A*x + B*u(k);
end
\]

\[
hold \ on
stem(t,y,'*')
hold \ off
legend('tf','ss')
Input Arguments

\textbf{b} — Transfer function numerator coefficients
vector | matrix

Transfer function numerator coefficients, specified as a vector or matrix. If \textbf{b} is a matrix, then each row of \textbf{b} corresponds to an output of the system.

- For discrete-time systems, \textbf{b} contains the coefficients in descending powers of \( z \).
- For continuous-time systems, \textbf{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-\( 1 \).
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 \( y \):

\[
\begin{align*}
x(k + 1) &= Ax(k) + Bu(k), \\
y(k) &= Cx(k) + Du(k).
\end{align*}
\]

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 \( y \):

\[
\begin{align*}
\dot{x} &= Ax + Bu, \\
y &= Cx + Du.
\end{align*}
\]

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**

\[ [z,p,k] = \text{tf2zp}(b,a) \]

**Description**

\[ [z,p,k] = \text{tf2zp}(b,a) \]
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.
\[ H(s) = \frac{2s^2 + 3s}{s^2 + \frac{1}{\sqrt{2}}s + \frac{1}{4}} = \frac{2(s - 0)(s - \left( -\frac{3}{2} \right))}{(s - \left( -\frac{1}{2\sqrt{2}} \right)(1 - j))(s - \left( -\frac{1}{2\sqrt{2}} \right)(1 + j))} \]

Find the zeros, poles, and gain of the system. Use `eqtflength` to ensure the numerator and denominator have the same length.

\[
\begin{align*}
\text{b} & = [2 \ 3]; \\
\text{a} & = [1 \ \frac{1}{\sqrt{2}} \ \frac{1}{4}]; \\
[b,a] & = \text{eqtflength}(b,a); \\
[z,p,k] & = \text{tf2zp}(b,a)
\end{align*}
\]

\[
\begin{align*}
z & = 2 \times 1 \\
&= \begin{pmatrix} 0 \\ -1.5000 \end{pmatrix}
\end{align*}
\]

\[
p = 2 \times 1 \text{ complex} \\
&= \begin{pmatrix} -0.3536 + 0.3536i \\ -0.3536 - 0.3536i \end{pmatrix}
\]

\[k = 2\]

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.

See Also
sos2zp | ss2zp | tf2sos | tf2ss | tf2zpk | zp2tf

Introduced before R2006a
tf2zpk

Convert transfer function filter parameters to zero-pole-gain form

Syntax

\[ [z, p, k] = \text{tf2zpk}(b, a) \]

Description

\[ [z, p, k] = \text{tf2zpk}(b, a) \]

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

\[
H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \cdots + b_{n-1} z^{-(n-1)} + b_n z^{-n}}{a_1 + a_2 z^{-1} + \cdots + a_{m-1} z^{-(m-1)} + a_m z^{-m}}
\]

d 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 \( \text{tf2zpk} \) when working with transfer functions expressed in inverse powers (\( 1 + z^{-1} + z^{-2} \)). A similar function, \( \text{tf2zp} \), 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 \( 0.4\pi \) 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.

tf2zpk
fvtool(b,a,'polezero')
text(real(z)-0.1,imag(z)-0.1,'fZeros','color',[0 0.4 0])
text(real(p)-0.1,imag(p)-0.1,'fPoles','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
text data

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
text data

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
text data

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

\( k \) — Gains
text data

Gains of the system, returned as a column vector. \( k \) contains the gains for each numerator transfer function.

**See Also**

sos2zp | ss2zp | tf2sos | tf2ss | tf2zp | zp2tf

Introduced before R2006a
tfestimate
Transfer function estimate

Syntax

\[
\begin{align*}
\text{txy} &= \text{tfestimate}(x,y) \\
\text{txy} &= \text{tfestimate}(x,y,\text{window}) \\
\text{txy} &= \text{tfestimate}(x,y,\text{window},\text{noverlap}) \\
\text{txy} &= \text{tfestimate}(x,y,\text{window},\text{noverlap},\text{nfft}) \\
\text{txy} &= \text{tfestimate}(\_,'\text{mimo}') \\
[\text{txy},w] &= \text{tfestimate}(\_) \\
[\text{txy},f] &= \text{tfestimate}(\_,\text{fs}) \\
[\text{txy},w] &= \text{tfestimate}(x,y,\text{window},\text{noverlap},w) \\
[\text{txy},f] &= \text{tfestimate}(x,y,\text{window},\text{noverlap},f,\text{fs}) \\
[\_] &= \text{tfestimate}(x,y,\_,\text{freqrange}) \\
[\_] &= \text{tfestimate}(\_,'\text{Estimator}',\text{est}) \\
\text{tfestimate}(\_) \\
\end{align*}
\]

Description

\(\text{txy} = \text{tfestimate}(x,y)\) finds a transfer function estimate, \(\text{txy}\), 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 \(\text{txy}\) is a multi-input/multi-output (MIMO) transfer function that combines all input and output signals. \(\text{txy}\) 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 1-2929 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.

\[ txy = tfestimate(x,y,\text{window}) \]
uses window to divide x and y into segments and perform windowing.

\[ txy = tfestimate(x,y,\text{window},\text{noverlap}) \]
uses noverlap samples of overlap between adjoining segments.

\[ txy = tfestimate(x,y,\text{window},\text{noverlap},\text{nfft}) \]
uses nfft sampling points to calculate the discrete Fourier transform.

\[ txy = tfestimate(\_,\text{'}mimo\text{'}) \]
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, [ ].

\[ [txy,w] = tfestimate(x,y,\text{window},\text{noverlap},w) \]
returns the transfer function estimate at the normalized frequencies specified in w.

\[ [txy,f] = tfestimate(x,y,\text{window},\text{noverlap},f,fs) \]
returns the transfer function estimate at the frequencies specified in f.

\[ [\_] = tfestimate(x,y,\_,\_,f\text{reqrange}) \]
returns the transfer function estimate over the frequency range specified by freqrange. Valid options for freqrange are 'onesided', 'twosided', and 'centered'.

\[ [\_] = tfestimate(\_,\text{'}Estimator\text{'},est) \]
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 \( 0.2\pi \) 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.

\[
\begin{align*}
h &= \text{fir1}(30, 0.2, \text{rectwin}(31)); \\
x &= \text{randn}(16384, 1); \\
y &= \text{filter}(h, 1, x);
\end{align*}
\]

\[
\begin{align*}
fs &= 500; \\
tfestimate(x, y, 1024, [], [], fs)
\end{align*}
\]
Use `fvtool` to verify that the transfer function approximates the frequency response of the filter:

```matlab
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.

\[ [Txy,f] = \text{tfestimate}(x,y,1024,[[],[],fs]); \]

\[ \text{plot}(f,\text{mag2db}\,(\text{abs}(Txy))) \]
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 $b = 0.01$. 

![Graph of SISO Transfer Function](image-url)
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

$$A = \exp(Ac\Delta t), \quad B = A_c^{-1}(A-I)B_c, \quad C = [-1 \ -b], \quad D = 1,$$

$I$ is the $2 \times 2$ identity, and the continuous-time state-space matrices are

$$Ac = \begin{bmatrix} 0 & 1 \\ -1 & -b \end{bmatrix}, \quad Bc = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

$$Ac = [0 \ 1;-1 \ -b];$$
$$A = \expm(Ac*dt);$$

$$Bc = [0;1];$$
$$B = Ac\backslash(A\text{-eye}(\text{size}(A)))\text{*}Bc;$$

$$C = [-1 \ -b];$$
$$D = 1;$$
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.

```matlab
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.

\[ \text{nfs} = 2048; \]
\[ \text{wind} = \text{kaiser}(N,15); \]

\[ [\text{txy,ft}] = \text{tfestimate}(u,y,\text{wind},[],\text{nfs},\text{Fs}); \]

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 \text{tfestimate} coincides with this definition.
\[ [b,a] = \text{ss2tf}(A,B,C,D); \]
\[ fz = 0:1/nfs:1/2-1/nfs; \]
\[ z = \exp(2j\pi f_z); \]
\[ frf = \text{polyval}(b,z)/\text{polyval}(a,z); \]

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 \texttt{tfestimate}.  

1-2917
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.

![Diagram of a mechanical system with masses $m_1$ and $m_2$, springs, and damping elements.]

Generate 30000 time samples, equivalent to 600 seconds. Define the sampling interval $\Delta t = 1/F_s$.

$F_s = 50$;  
$dt = 1/F_s$;  
$N = 30000$;  
$t = dt*(0:N-1)$;

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 $i$th mass, $u = [u_1 \ u_2]^T$ is the vector of input driving forces, and $y = [a_1 \ a_2]^T$ is the output vector. The state-space matrices are

$$A = \exp(A_c\Delta t), \quad B = A_c^{-1}(A - I)B_c, \quad C = \begin{bmatrix} -2k & -2b & k & b \\ k/\mu & b/\mu & -2k/\mu & -2b/\mu \end{bmatrix}, \quad D = \begin{bmatrix} 1 & 0 \\ 0 & 1/\mu \end{bmatrix}.$$ 

$I$ is the $4 \times 4$ 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 $\mu = 1/10$. 

1-2919
k = 400;
b = 0;
m = 1/10;

Ac = [0 1 0 0; -2*k -2*b k b; 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 \( \omega \) are the eigenvalues of the modal matrix.

\[
\sqrt{\text{eig}(k*[2 -1;-1/m 2/m]))/(2*pi)}
\]

\[
\text{ans} = 2\times1
\]

\[
\begin{align*}
3.8470 \\
14.4259
\end{align*}
\]

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.

\[ b = 0.1; \]

\[
Ac = \begin{bmatrix}
0 & 1 & 0 & 0; \\
-2*k & -2*b & k & b; 0 & 0 & 1; k/m & b/m & -2*k/m & -2*b/m
\end{bmatrix};
\]

\[
A = \expm(Ac*dt);
\]

\[
B = Ac\backslash(A-\text{eye}(4))*Bc;
\]

\[
C = \begin{bmatrix}
-2*k & -2*b & k & b; k/m & b/m & -2*k/m & -2*b/m
\end{bmatrix};
\]

\[
x = [0;0;0;0];
\]

\[
\text{for } \text{kk} = 1:N
\]
\[
\quad y(:,\text{kk}) = Cx + D*u(:,\text{kk});
\]
\[
\quad x = Ax + Bu(:,\text{kk});
\]
\[
\text{end}
\]

\[
\text{clf}
\]

\[
\text{tfestimate}(u',y',\text{wind},\text{nov},[],\text{Fs},\text{'mimo'},\text{'Estimator'},\text{'H2'})
\]

\[
\text{legend('I1, O1','I1, O2','I2, O1','I2, O2')}
\]
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,:,:) = polyval(b1(1,:),z)./polyval(a1,z);
frf(1,:,:) = polyval(b1(2,:),z)./polyval(a1,z);
frf(2,:,:) = polyval(b2(1,:),z)./polyval(a2,z);
frf(2,:,:) = polyval(b2(2,:),z)./polyval(a2,z);

plot(fz*Fs,20*log10(abs(reshape(permute(frf,[2 1 3]),[nfs/2 4]))))
Input Arguments

x — Input signal
vector | matrix

Input signal, specified as a vector or matrix.
Example: \( \cos(\pi/4*(0:159))+\text{randn}(1,160) \) 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

\( \text{window} \) — Window  
integer | vector | []

Window, specified as an integer or as a row or column vector. Use \( \text{window} \) to divide the signal into segments.

- If \( \text{window} \) is an integer, then \( \text{tfestimate} \) divides \( x \) and \( y \) into segments of length \( \text{window} \) and windows each segment with a Hamming window of that length.
- If \( \text{window} \) is a vector, then \( \text{tfestimate} \) divides \( x \) and \( y \) into segments of the same length as the vector and windows each segment using \( \text{window} \).

If the length of \( x \) and \( y \) cannot be divided exactly into an integer number of segments with \( \text{noverlap} \) overlapping samples, then the signals are truncated accordingly.

If you specify \( \text{window} \) as empty, then \( \text{tfestimate} \) uses a Hamming window such that \( x \) and \( y \) are divided into eight segments with \( \text{noverlap} \) overlapping samples.

For a list of available windows, see “Windows”.

Example: \( \text{hann}(N+1) \) and \( (1-\cos(2\pi*(0:N)'/N))/2 \) both specify a Hann window of length \( N + 1 \).

Data Types: single | double

\( \text{noverlap} \) — 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. If the segment length is unspecified, the function sets `noverlap` to \( \lfloor \frac{N}{4.5} \rfloor \), 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 `tfestimate` 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 transfer function estimate
'onesided' | 'twosided' | 'centered'

Frequency range for the transfer function estimate, specified as 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 1-2929 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
\[
H_1(f) = P_{YX}(f)P_{XX}^{-1}(f) = \begin{bmatrix}
P_{y1x1}(f) & P_{y1x2}(f) & \cdots & P_{y1xm}(f) \\
P_{y2x1}(f) & P_{y2x2}(f) & \cdots & P_{y2xm}(f) \\
\vdots & \vdots & \ddots & \vdots \\
P_{ynx1}(f) & P_{ynx2}(f) & \cdots & P_{ynxm}(f)
\end{bmatrix}^{-1}
\]

\[
\begin{bmatrix}
P_{x1x1}(f) & P_{x1x2}(f) & \cdots & P_{x1xm}(f) \\
P_{x2x1}(f) & P_{x2x2}(f) & \cdots & P_{x2xm}(f) \\
\vdots & \vdots & \ddots & \vdots \\
P_{xm1}(f) & P_{xm2}(f) & \cdots & P_{xm1m}(f)
\end{bmatrix}
\]

for \( m \) inputs and \( n \) outputs, where:

- \( P_{yik} \) is the cross power spectral density of the \( k \)th input and the \( i \)th output.
- \( P_{xik} \) is the cross power spectral density of the \( k \)th and \( i \)th inputs.

For two inputs and two outputs, the estimator is the matrix

\[
H_1(f) = \begin{bmatrix}
P_{y1x1}(f)P_{x2x2}(f) - P_{y1x2}(f)P_{x2x1}(f) & P_{y1x2}(f)P_{x1x1}(f) - P_{y1x1}(f)P_{x1x2}(f) \\
P_{y2x1}(f)P_{x2x2}(f) - P_{y2x2}(f)P_{x2x1}(f) & P_{y2x2}(f)P_{x1x1}(f) - P_{y2x1}(f)P_{x1x2}(f)
\end{bmatrix} \begin{bmatrix}
P_{x1x1}(f)P_{x2x2}(f) - P_{x1x2}(f)P_{x2x1}(f) \\
P_{x1x1}(f)P_{x2x2}(f) - P_{x1x2}(f)P_{x2x1}(f)
\end{bmatrix}^{-1}.
\]

- 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:

- \( P_{y_ky_i} \) is the cross power spectral density of the \( k \)th and \( i \)th outputs.
- \( P_{x_ky_i} \) is the complex conjugate of the cross power spectral density of the \( i \)th input and the \( k \)th output.

**Algorithms**

tfestimate uses Welch's averaged periodogram method. See pwelch for details.

**References**


**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.

\[
S = S + \text{rand}(	ext{size}(S))/10;
\]

\[
\text{mesh}(T,F,S)
\]
\[
\text{view}(-30,60)
\]
\[
\text{xlabel}('\text{Time}')
\]
\[
\text{ylabel}('\text{Frequency}')
\]
Extract the ridge and plot the result.

\[
[f\text{ridge},~,l\text{ridge}] = \text{tfridge}(S,f);
\]

\[
r\text{vals} = S(l\text{ridge});
\]

\[
\text{hold on}
\]

\[
\text{plot3}(t,f\text{ridge},r\text{vals},'k','\text{linewidth}',4)
\]

\[
\text{hold off}
\]
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.

```plaintext
tenval = 1;
fridge = tfridge(sst,f,penval,'NumRidges',2);
```

```plaintext
mesh(t,f,abs(sst))
view(2)
xlabel('Time (s)')
ylabel('Frequency (Hz)')
hold on
```
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)')
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.

```matlab
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.

```matlab
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.

```matlab
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**

\( \text{tfm} \) — 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:

   \[
   \begin{pmatrix}
   1 & 4 & 4 \\
   2 & 2 & 2 \\
   5 & 5 & 4
   \end{pmatrix}
   \]

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

   \[
   \text{original value} + \text{penalty} \times \text{distance}
   \]

   \[
   \begin{align*}
   1 + 2 \times 0 &= 1 \\
   2 + 2 \times 1 &= 4 \\
   5 + 2 \times 4 &= 13
   \end{align*}
   \]

   \[
   \begin{pmatrix}
   1 & 4 \\
   4 & 2 \\
   13 & 5
   \end{pmatrix}
   \]

   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

\[ \text{original value} + \text{penalty} \times \text{distance} \]

\[
\begin{align*}
1 + 2 \times 1 &= 3 \\
2 + 2 \times 0 &= 2 \\
5 + 2 \times 1 &= 7
\end{align*}
\]

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

\[
\begin{array}{ccc}
1 & 5_{(1)} & 4 \\
2 & 4_{(2)} & 2 \\
5 & 9_{(2)} & 4 \\
\end{array}
\]

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

\[
\begin{align*}
5 + 2 \times 0 &= 5 \\
4 + 2 \times 1 &= 6 \\
9 + 2 \times 4 &= 17
\end{align*}
\]

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

\[
\begin{array}{ccc}
1 & 5_{(1)} & 9_{(1)} \\
2 & 4_{(2)} & 6_{(2)} \\
5 & 9_{(2)} & 10_{(2)} \\
\end{array}
\]

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

\[ r = \text{thd}(x) \]
\[ r = \text{thd}(x,fs,n) \]
\[ r = \text{thd}(pxx,f,'psd') \]
\[ r = \text{thd}(pxx,f,n,'psd') \]
\[ r = \text{thd}(sxx,f,rbw,'power') \]
\[ r = \text{thd}(sxx,f,rbw,n,'power') \]
\[ r = \text{thd}(___,'aliased') \]
\[ [r,harmpow,harmfreq] = \text{thd}(___) \]
\text{thd}(___)

Description

\[ r = \text{thd}(x) \] 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 \( \beta = 38 \).

\[ r = \text{thd}(x,fs,n) \] specifies the sample rate, \( fs \), and the number of harmonics (including the fundamental) to use in the THD calculation.

\[ r = \text{thd}(pxx,f,'psd') \] 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 \( pxx \).

\[ r = \text{thd}(pxx,f,n,'psd') \] specifies the number of harmonics (including the fundamental) to use in the THD calculation.
\( r = \text{thd}(sxx,f,rbw,'power') \) specifies the input as a one-sided power spectrum. \( rbw \) is the resolution bandwidth over which each power estimate is integrated.

\( r = \text{thd}(sxx,f,rbw,n,'power') \) specifies the number of harmonics (including the fundamental) to use in the THD calculation.

\( r = \text{thd}(\_\_\_,\text{'aliased'}) \) 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.

\([r,\text{harmpow},\text{harmfreq}] = \text{thd}(\_\_\_) \) returns the powers (in dB) and frequencies of the harmonics, including the fundamental.

\( \text{thd}(\_\_\_) \) 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 \( \text{thd} \).

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 \( \text{thd} \).

\[
\begin{align*}
  t &= 0:0.001:1-0.001; \\
  x &= 2\cos(2\pi\times100\times t)+0.01\cos(2\pi\times200\times t)+0.005\cos(2\pi\times300\times t); \\
  \text{tharmdist} &= 10\log_{10}((0.01^2+0.005^2)/2^2)
\end{align*}
\]

\( \text{tharmdist} = -45.0515 \)

\( r = \text{thd}(x) \)

\( r = -45.0515 \)
**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.

\[
t = 0:0.001:1-0.001;
\]
\[
x = 2*\cos(2*\pi*100*t)+0.01*\cos(2*\pi*200*t)+ ...
0.005*\cos(2*\pi*300*t)+0.0025*\sin(2*\pi*400*t);
\]
\[
r = \text{thd}(x,1000,3)
\]
\[
r = -45.0515
\]

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.

\[
t = 0:0.001:1-0.001;
\]
\[
fs = 1000;
\]
\[
x = 2*\cos(2*\pi*100*t)+0.01*\cos(2*\pi*200*t)+ ...
0.005*\cos(2*\pi*300*t)+0.0025*\sin(2*\pi*400*t);
\]
\[
[pxx,f] = \text{periodogram}(x,\text{rectwin(length}(x)),\text{length}(x),fs);
\]
\[
r = \text{thd}(pxx,f,3,'\text{psd}')
\]
\[
r = -45.0515
\]
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.

```matlab
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. 

```matlab
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.

\texttt{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.

```matlab
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);
```

![THD: -22.54 dB](image)

**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.
[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.

```matlab
fs = 5e4;
f0 = 2.5e3;
N = 1024;
t = (0:N-1)/fs;

t = 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

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 $2\pi$ 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.

\textbf{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**

\textbf{r} — Total harmonic distortion in dBc
real-valued scalar

Total harmonic distortion in dBc, returned as a real-valued scalar.

\textbf{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 \textbf{harmpow} is a scalar or a vector depends on the number of harmonics you specify as the input argument \textit{n}.

\textbf{harmfreq} — Frequencies of the harmonics
nonnegative scalar or vector

Frequencies of the harmonics, returned as a nonnegative scalar or vector. Whether \textbf{harmfreq} is a scalar or a vector depends on the number of harmonics you specify as the input argument \textit{n}.

**More About**

**Distortion Measurement Functions**

The functions \texttt{thd}, \texttt{sfdr}, \texttt{sinad}, and \texttt{snr} measure the response of a weakly nonlinear system stimulated by a sinusoid.
When given time-domain input, `thd` performs a periodogram using a Kaiser window with large sidelobe attenuation. To find the fundamental frequency, the algorithm searches the periodogram for the largest nonzero spectral component. It then computes the central moment of all adjacent bins that decrease monotonically away from the maximum. To be detectable, the fundamental should be at least in the second frequency bin. Higher harmonics are at integer multiples of the fundamental frequency. If a harmonic lies within the monotonically decreasing region in the neighborhood of another, its power is considered to belong to the larger harmonic. This larger harmonic may or may not be the fundamental.

`thd` fails if the fundamental is not the highest spectral component in the signal.

Ensure that the frequency components are far enough apart to accommodate for the sidelobe width of the Kaiser window. If this is not feasible, you can use the `'power'` flag and compute a periodogram with a different window.

**See Also**

`sfdr` | `sinad` | `snr` | `toi`

**Topics**

“Analyzing Harmonic Distortion”

**Introduced in R2013b**
** toi 

Third-order intercept point 

** Syntax **

\[
oip3 = \text{toi}(x) \\
oip3 = \text{toi}(x,fs) \\
oip3 = \text{toi}(\text{pxx},f,\text{'psd'}) \\
oip3 = \text{toi}(\text{sxx},f,\text{rbw},\text{'power'})
\]

\[
[oip3,fundpow,fundfreq,imodpow,imodfreq] = \text{toi}(___)
\]

** Description **

\(\text{oip3} = \text{toi}(x)\) 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 \(\beta = 38\).

\(\text{oip3} = \text{toi}(x,fs)\) specifies the sample rate, \(fs\). The default value of \(fs\) is 1.

\(\text{oip3} = \text{toi}(\text{pxx},f,\text{'psd'})\) specifies the input as a one-sided power spectral density (PSD), \(\text{pxx}\), of a real signal. \(f\) is a vector of frequencies that corresponds to the vector of \(\text{pxx}\) estimates.

\(\text{oip3} = \text{toi}(\text{sxx},f,\text{rbw},\text{'power'})\) specifies the input as a one-sided power spectrum, \(\text{sxx}\), of a real signal. \(\text{rbw}\) is the resolution bandwidth over which each power estimate is integrated.

\([\text{oip3},\text{fundpow},\text{fundfreq},\text{imodpow},\text{imodfreq}] = \text{toi}(___)\) also returns the power, \(\text{fundpow}\), and frequencies, \(\text{fundfreq}\), of the two fundamental sinusoids. It also returns the power, \(\text{imodpow}\), and frequencies, \(\text{imodfreq}\), 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.

```matlab
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
10³ x
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.

```matlab
rng default
f1 = 5e3;
f2 = 6e3;
Fs = 48e3;
N = 1000;
x = sin(2*pi*f1/Fs*(1:N))+sin(2*pi*f2/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.

```matlab
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 = 1x2
   -22.9133  -22.9132

Ffund = 1x2
10^3 x
   5.0000   6.0000

Pim3 = 1x2
   -71.4868  -71.5299

Fim3 = 1x2
10^3 x
   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.

```matlab
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')
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.

ans = 1.3844
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 = 1x2
    -2.8350   -2.8201
Ffund = 1x2
    5.0000    7.0001
Pim3 = 1x2
    -43.1362  -43.5211
Fim3 = 1x2
    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**


**See Also**

sfdr | sinad | snr | thd

**Introduced in R2013b**
triang

Triangular window

Syntax

\[ w = \text{triang}(L) \]

Description

\[ w = \text{triang}(L) \] returns an \( L \)-point triangular window.

Examples

Triangular Window

Create a 200-point triangular window. Display the result using \text{wvtool}.

\[
\begin{align*}
L &= 200; \\
w &= \text{triang}(L); \\
wvtool(w)
\end{align*}
\]
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:

$$ w(n) = \begin{cases} 2n \frac{L + 1}{L + 1} & 1 \leq n \leq \frac{L + 1}{2} \\ 2 - 2n \frac{L + 1}{L + 1} & + 1 \leq n \leq L \end{cases} $$

For $L$ even:

$$ w(n) = \begin{cases} \frac{(2n - 1)}{L} & 1 \leq n \leq \frac{L}{2} \\ 2 - \frac{(2n - 1)}{L} \frac{L}{2} & + 1 \leq n \leq L \end{cases} $$

## References


## 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

**tripuls**

Sampled aperiodic triangle

**Syntax**

\[
y = \text{tripuls}(T) \\
y = \text{tripuls}(T,w) \\
y = \text{tripuls}(T,w,s)
\]

**Description**

\[
y = \text{tripuls}(T) \text{ returns a continuous, aperiodic, symmetric, unity-height triangular pulse at the times indicated in array } T, \text{ centered about } T=0 \text{ and with a default width of 1.}
\]

\[
y = \text{tripuls}(T,w) \text{ generates a triangular pulse of width } w.
\]

\[
y = \text{tripuls}(T,w,s) \text{ generates a triangular pulse with skew } s, \text{ where } -1 \leq s \leq 1. \text{ When } s \text{ is 0, a symmetric triangular pulse is generated.}
\]

**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.

\[
fs = 10e3; \\
t = -0.1:1/fs:0.1; \\
w = 40e-3; \\
x = \text{tripuls}(t,w);
\]

Generate two variations of the same pulse:
• One displaced 45 ms into the past and skewed 45% to the left.

```matlab
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.

```matlab
tfutr = 60e-3;
sfutr = 1;
xfutr = tripuls(t-tfutr,w/2,sfutr);
```

Plot the original pulse and the two copies on the same axes.

```matlab
plot(t,x,t,xpast,t,xfutr)
ylim([-0.2 1.2])
```
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**

```matlab
ta = tsa(x,fs,tp)
ta = tsa(x,t,tp)
ta = tsa(xt,tp)
```

```matlab
ta = tsa(___,Name,Value)
```

```matlab
[ta,t,p,rpm] = tsa(___
```

```matlab
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.

```matlab
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.

```matlab
fs = 1e3;
t = 0:1/fs:2;
x = exp(-2*t').*chirp(t',2,1,28,'quadratic');
```
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.

```matlab
plot(t,x)

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.

\[
[xta, xt, xp, xrpm] = tsa(tx, lc);
\]

\[
\text{whos } x^*
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2001x1</td>
<td>16008</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>xp</td>
<td>9x1</td>
<td>1192</td>
<td>timetable</td>
<td></td>
</tr>
<tr>
<td>xrpm</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>xt</td>
<td>9x1</td>
<td>74</td>
<td>duration</td>
<td></td>
</tr>
<tr>
<td>xta</td>
<td>9x1</td>
<td>1190</td>
<td>timetable</td>
<td></td>
</tr>
</tbody>
</table>
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.

```matlab
dtb = datetime(datevec(ts));
dtt = timetable(dtb,x);

nr = 15;
tsa(dtt,lc,'NumRotations',nr)
```
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:

\[
\phi = 2\pi f_0 T (1 - e^{-t/T}),
\]

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.

\[
x = a \cos(\phi) + \text{randn(size(phi))}/200;
\]

\[
y = a \sin(\phi) + \text{randn(size(phi))}/200;
\]

\[\text{plot}(t,x,t,y)\]

Determine the synchronizing signal. Use the \texttt{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.

\[
[\text{rpm},~,\text{tp}] = \text{tachorpm}(x(t<2.5),fs);
\]

\[
\text{tachorpm}(x(t<2.5),fs)
\]
Compute and plot the time-synchronous average signal, which corresponds to a period of a sinusoid. Perform the averaging in the frequency domain.

```matlab
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” (MATLAB).

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 comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

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**

Extended Capabilities

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

See Also
tachorpm

Topics
“Vibration Analysis of Rotating Machinery”

Introduced in R2017b
tukeywin

Tukey (tapered cosine) window

Syntax

\[ w = \text{tukeywin}(L,r) \]

Description

\[ w = \text{tukeywin}(L,r) \] 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 \texttt{wvtool}.

\begin{verbatim}
L = 128;
t0 = tukeywin(L,0); % Equivalent to a rectangular window
t25 = tukeywin(L,0.25); % r = 0.5
t5 = tukeywin(L);
t75 = tukeywin(L,0.75);
t1 = tukeywin(L,1); % Equivalent to a Hann window
wvtool(t0,t25,t5,t75,t1)
\end{verbatim}
**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 2\( r = 1 \). If you specify \( r \leq 0 \), an L-point rectangular window is returned. If you specify \( r \geq 1 \), an L-point von Hann window is returned.

Data Types: `single` | `double`

### Output Arguments

\( \mathbf{w} \) — Tukey window

column vector

Tukey window, returned as a column vector.

### Algorithms

The following equation defines the \( L \)-point Tukey window:

\[
\begin{align*}
\mathbf{w}(x) &= \begin{cases} \\
\frac{1}{2} \left(1 + \cos\left(\frac{2\pi}{r}[x - r/2]\right)\right), & 0 \leq x < \frac{r}{2} \\
1, & \frac{r}{2} \leq x < 1 - \frac{r}{2} \\
\frac{1}{2} \left(1 + \cos\left(\frac{2\pi}{r}[x - 1 + r/2]\right)\right), & 1 - \frac{r}{2} \leq x \leq 1 \\
\end{cases}
\end{align*}
\]

where \( x \) is an \( L \)-point linearly spaced vector generated using `linspace`. The parameter \( r \) is the ratio of cosine-tapered section length to the entire window length with \( 0 \leq r \leq 1 \). For example, setting \( r = 0.5 \) produces a Tukey window where 1/2 of the entire window length consists of segments of a phase-shifted cosine with period 2\( r = 1 \). If you specify \( r \leq 0 \), an \( L \)-point rectangular window is returned. If you specify \( r \geq 1 \), an \( L \)-point von Hann window is returned.

### References

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**

$$y = \text{udecode}(u, n)$$
$$y = \text{udecode}(u, n, v)$$
$$y = \text{udecode}(u, n, v, 'SaturateMode')$$

**Description**

$y = \text{udecode}(u, n)$ 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 ($\text{uint8}$, $\text{uint16}$, $\text{uint32}$, $\text{int8}$, $\text{int16}$, $\text{int32}$). 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 $[-1, 1]$.

$y = \text{udecode}(u, n, v)$ decodes $u$ such that the output $y$ has values in the range $[-v, v]$, where the default value for $v$ is 1.

$y = \text{udecode}(u, n, v, 'SaturateMode')$ 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 \(u = \text{mod}(u+2^n/2, 2^n)-(2^n/2)\)).
  • Entries in unsigned inputs \( u \) whose values are outside of the range \([0, 2^n-1]\) are wrapped back into the required range before decoding using modulo \(2^n\) arithmetic (calculated using \(u = \text{mod}(u, 2^n)\)).

Examples

Use udecode to Decode Integers

Create a vector of 8-bit signed integers. Decode with three bits.

\[
\begin{align*}
u &= \text{int8}([-1 1 2 -5]); \\
y\text{sat} &= \text{udecode}(u,3)
\end{align*}
\]

\[
y\text{sat} = 1\times4 \\
-0.2500 & 0.2500 & 0.5000 & -1.0000
\]

Notice the last entry in \( u \) saturates to 1, the default peak input magnitude. Change the peak input magnitude to 6.

\[
\begin{align*}
y\text{satv} &= \text{udecode}(u,3,6) \\
y\text{satv} = 1\times4 \\
-1.5000 & 1.5000 & 3.0000 & -6.0000
\end{align*}
\]

The last input entry still saturates. Wrap the overflows.

\[
\begin{align*}
y\text{wrap} &= \text{udecode}(u,3,6,'wrap') \\
y\text{wrap} = 1\times4
\end{align*}
\]
Add more quantization levels.

\[ y_{\text{prec}} = \text{udecode}(u, 5) \]

\[ y_{\text{prec}} = 1 \times 4 \]

\[ \begin{array}{cccc}
-0.0625 & 0.0625 & 0.1250 & -0.3125 \\
\end{array} \]

## 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


## See Also

uencode

Introduced before R2006a
uencode

Quantize and encode floating-point inputs to integer outputs

Syntax

\[ y = \text{uencode}(u, n) \]
\[ y = \text{uencode}(u, n, v) \]
\[ y = \text{uencode}(u, n, v, 'SignFlag') \]

Description

\( y = \text{uencode}(u, n) \) quantizes the entries in a multidimensional array of floating-point numbers \( u \) and encodes them as integers using \( 2^n \)-level quantization. \( n \) must be an integer between 2 and 32 (inclusive). Inputs can be real or complex, double- or single-precision. The output \( y \) and the input \( u \) are arrays of the same size. The elements of the output \( y \) are unsigned integers with magnitudes in the range \([0, 2^n - 1]\). Elements of the input \( u \) outside of the range \([-1, 1]\) are treated as overflows and are saturated.

- For entries in the input \( u \) that are less than -1, the value of the output of \text{uencode} is 0.
- For entries in the input \( u \) that are greater than 1, the value of the output of \text{uencode} is \( 2^n - 1 \).

\( y = \text{uencode}(u, n, v) \) allows the input \( u \) to have entries with floating-point values in the range \([-v, v]\) before saturating them (the default value for \( v \) is 1). Elements of the input \( u \) outside of the range \([-v, v]\) are treated as overflows and are saturated:

- For input entries less than \(-v\), the value of the output of \text{uencode} is 0.
- For input entries greater than \(v\), the value of the output of \text{uencode} is \( 2^n - 1 \).

\( y = \text{uencode}(u, n, v, 'SignFlag') \) maps entries in a multidimensional array of floating-point numbers \( u \) whose entries have values in the range \([-v, v]\) to an integer output \( y \). Input entries outside this range are saturated. The integer type of the output depends on the number of quantization levels \( 2^n \) and the value of \text{'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 \([0, 2^n - 1]\).

The output data types are optimized for the number of bits as shown in the table below.

<table>
<thead>
<tr>
<th>n</th>
<th>Unsigned Integer</th>
<th>Signed Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 to 8</td>
<td>uint8</td>
<td>int8</td>
</tr>
<tr>
<td>9 to 16</td>
<td>uint16</td>
<td>int16</td>
</tr>
<tr>
<td>17 to 32</td>
<td>uint32</td>
<td>int32</td>
</tr>
</tbody>
</table>

**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., \(2^3 - 1\)).

```matlab
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.

\[ u = \text{-2:0.5:2}; \]
\[ y = \text{uencode}(u,5,1) \]

\[ y = \text{1x9 uint8 row vector} \]

0 0 0 8 16 24 31 31 31

Specify you want signed output.

\[ u = \text{-2:0.5:2}; \]
\[ y = \text{uencode}(u,5,2,'\text{signed}') \]
y = 1x9 int8 row vector

-16 -12 -8 -4 0 4 8 12 15

Algorithms

tencode maps the floating-point input value to an integer value determined by the requirement for $2^n$ levels of quantization. This encoding adheres to the definition for uniform encoding specified in ITU-T Recommendation G.701. The input range $[-v, v]$ is divided into $2^n$ evenly spaced intervals. Input entries in the range $[-v, v]$ are first quantized according to this subdivision of the input range, and then mapped to one of $2^n$ integers. The range of the output depends on whether or not you specify that you want signed integers.

References


See Also

tdecode

Introduced before R2006a
unshiftdata

Inverse of shiftdata

Syntax

\[ y = \text{unshiftdata}(x,\text{perm},\text{nshifts}) \]

Description

\[ y = \text{unshiftdata}(x,\text{perm},\text{nshifts}) \] restores the orientation of the data that was shifted with \text{shiftdata}. The permutation vector is given by \text{perm}, and \text{nshifts} is the number of shifts that was returned from \text{shiftdata}.

\text{unshiftdata} is meant to be used in tandem with \text{shiftdata}. These functions are useful for creating functions that work along a certain dimension, like \text{filter}, \text{goertzel}, \text{sgolayfilt}, and \text{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. \text{unshiftdata} shifts \( x \) back to its original shape.

Create a 3-by-3 magic square.

\[
\begin{bmatrix}
8 & 1 & 6 \\
3 & 5 & 7 \\
4 & 9 & 2
\end{bmatrix}
\]
Shift the matrix \( x \) to work along the second dimension. The permutation vector, \( \text{perm} \), and the number of shifts, \( nshifts \), are returned along with the shifted matrix.

\[
[x, \text{perm}, nshifts] = \text{shiftdata}(x, 2)
\]

\[
\begin{bmatrix}
8 & 3 & 4 \\
1 & 5 & 9 \\
6 & 7 & 2
\end{bmatrix}
\]

\[
\text{perm} = \begin{bmatrix}
2 & 1
\end{bmatrix}
\]

\[
nshifts = []
\]

Shift the matrix back to its original shape.

\[
y = \text{unshiftdata}(x, \text{perm}, nshifts)
\]

\[
\begin{bmatrix}
8 & 1 & 6 \\
3 & 5 & 7 \\
4 & 9 & 2
\end{bmatrix}
\]

**Rearrange Array to Operate on First Nonsingleton Dimension**

This example shows how \text{shiftdata} and \text{unshiftdata} work when you define \text{dim} as empty.

Define \( x \) as a row vector.

\[
x = 1:5
\]

\[
x = 1 \times 5
\]
Define \texttt{dim} as empty to shift the first nonsingleton dimension of \texttt{x} to the first column. \texttt{shiftdata} returns \texttt{x} as a column vector, along with \texttt{perm}, the permutation vector, and \texttt{nshifts}, the number of shifts.

\[
\begin{bmatrix}
x, \texttt{perm}, \texttt{nshifts}
\end{bmatrix} = \texttt{shiftdata}(x,[]) \\
x = 5\times1 \\
\begin{bmatrix}
1 \\
2 \\
3 \\
4 \\
5
\end{bmatrix}
\]

\texttt{perm} =

\[
\begin{bmatrix}
\end{bmatrix}
\]

\texttt{nshifts} = 1

Using \texttt{unshiftdata}, restore \texttt{x} to its original shape.

\[
y = \texttt{unshiftdata}(x,\texttt{perm},\texttt{nshifts}) \\
y = 1\times5 \\
\begin{bmatrix}
1 & 2 & 3 & 4 & 5
\end{bmatrix}
\]

\textbf{See Also}
\texttt{ipermute} | \texttt{shiftdata} | \texttt{shiftdim}

\textit{Introduced in R2012b}
upfirdn

Upsample, apply FIR filter, and downsample

Syntax

\[
y_{out} = \text{upfirdn}(x_{in}, h) \\
y_{out} = \text{upfirdn}(x_{in}, h, p) \\
y_{out} = \text{upfirdn}(x_{in}, h, p, q)
\]

Description

\[y_{out} = \text{upfirdn}(x_{in}, h)\] filters the input signal \(x_{in}\) with an FIR filter with impulse response \(h\). No upsampling or downsampling is implemented with this syntax.

\[y_{out} = \text{upfirdn}(x_{in}, h, p)\] specifies the integer upsampling factor \(p\).

\[y_{out} = \text{upfirdn}(x_{in}, h, p, q)\] 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 \text{rat} function to find the numerator \(L\) and the denominator \(M\) of the rational factor.

\[
\begin{align*}
F_{dat} &= 48e3; \\
F_{cd} &= 44.1e3; \\
[L, M] &= \text{rat}(F_{cd}/F_{dat})
\end{align*}
\]

\[
\begin{align*}
L &= 147 \\
M &= 160
\end{align*}
\]
Generate a 1.5 kHz sinusoid sampled at $f_{DAT}$ for 0.25 seconds. Plot the first millisecond of the signal.

```matlab
t = 0:1/Fdat:0.25-1/Fdat;
x = sin(2*pi*1.5e3*t);
plot(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 $L$. 

1-3015
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.

\[ y = \text{upfirdn}(x,h,L,M); \]
\[ \text{delay} = \text{floor}(((\text{filtord}(d)-1)/2-(L-1))/L); \]
\[ y = y(\text{delay}+1:end); \]
\[ t\_res = (0:(\text{length}(y)-1))/Fcd; \]

Overlay the resampled signal on the plot.

\[ \text{stem}(t\_res,y,\text{'*'}); \]
\[ \text{legend('Original','Resampled','Location','southeast');} \]
\[ \text{hold off} \]
Input Arguments

\( \text{xin} \) — Input signal
vector | matrix

Input signal, specified by a vector or matrix. If \( \text{xin} \) is a vector, then it represents a single signal. If \( \text{xin} \) is a matrix, then each column is filtered independently.

\( \text{h} \) — 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.

\( p \) — Upsampling factor
1 (default) | positive integer

Upsampling factor, specified by a positive integer.

\( q \) — Downsampling factor
1 (default) | positive integer

Downsampling factor, specified by a positive integer.

Output Arguments

\( yout \) — Output signal
vector | matrix

Output signal, specified as a vector or matrix. If \( yout \) is a vector, then it represents one signal. If \( yout \) is a matrix, then each column is a separate output. The length of the output, \( yout \) is \( \lceil((\text{length}(xin)-1)*p+\text{length}(h))/q \rceil \).

**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 \( x_{in} \). The resulting \( y_{out} \) is a matrix with the same number of columns as \( x_{in} \).

3 \( x_{in} \) 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 \( x_{in} \). The resulting \( y_{out} \) is a matrix with the same number of columns as \( h \).

4 \( x_{in} \) 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 \( x_{in} \) and \( h \). The resulting \( y_{out} \) is a matrix with the same number of columns as \( x_{in} \) and \( h \).

### Algorithms

\texttt{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.

A more accurate flops count is computed in the program, but the exact count is still approximate. For long signals \( x(n) \), the formula is often exact.

\texttt{upfirdn} performs a cascade of three operations:

1. Upsample the input data in the matrix \( x_{in} \) 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 \texttt{firpm} or \texttt{fir1}.

**Note** The function \texttt{resample} performs an FIR design using \texttt{firls}, followed by rate changing implemented with \texttt{upfirdn}. 1-3019
References


Extended Capabilities

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

Usage notes and limitations:

C and C++ code generation for upfirdn requires DSP System Toolbox software.

Filter coefficients, upsampling factor, and downsampling factor must be constants. Expressions or variables are allowed if their values do not change.

Variable-size inputs are not supported.

See Also
conv | decimate | downsample | filter | interp | intfilt | resample | upsample

Introduced before R2006a
**upsample**

Increase sample rate by integer factor

**Syntax**

\[ y = \text{upsample}(x,n) \]
\[ y = \text{upsample}(x,n,\text{phase}) \]

**Description**

\[ y = \text{upsample}(x,n) \] 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.

\[ y = \text{upsample}(x,n,\text{phase}) \] 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.

\[
\begin{align*}
x &= [1 \ 2 \ 3 \ 4]; \\
y &= \text{upsample}(x,3) \\
y &= \begin{bmatrix}
1 & 0 & 0 & 2 & 0 & 0 & 3 & 0 & 0 & 4 & 0 & 0 \\
\end{bmatrix}
\end{align*}
\]

Increase the sample rate of the sequence by a factor of 3 and add a phase offset of 2.

\[
\begin{align*}
x &= [1 \ 2 \ 3 \ 4]; \\
y &= \text{upsample}(x,3,2) \\
y &= \begin{bmatrix}
1 & 0 & 0 & 2 & 0 & 0 & 3 & 0 & 0 & 4 & 0 & 0 \\
\end{bmatrix}
\end{align*}
\]
Increase the sample rate of a matrix by a factor of 3.

```matlab
x = [1 2; 3 4; 5 6];
y = upsample(x,3)
```

```matlab
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™.

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 the greatest deviations below the final state levels of each transition in the bilevel waveform, X. The undershoots, US, are expressed as a percentage of the difference between the state levels. See “Undershoot” on page 1-3031. The length of US corresponds to the number of transitions detected in the input signal. The sample instants in X correspond to the vector indices. To determine the transitions, undershoot estimates the state levels of the input waveform by a histogram method. undershoot identifies all regions that cross the upper-state boundary of the low state and the lower-state boundary of the high state. The low-state and high-state boundaries are expressed as the state level plus or minus a multiple of the difference between the state levels. See “State-Level Tolerances” on page 1-3032.

US = undershoot(X,FS) specifies the sample rate, FS, 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.

US = undershoot(X,T) specifies the sample instants, T, as a vector with the same number of elements as X.

[US,USLEV,USINST] = undershoot(...) returns the levels, USLEV, and sample instants, USINST, of the undershoots for each transition.
... = undershoot(...,Name,Value) returns the greatest deviations below the final state level with additional options specified by one or more Name,Value pair arguments.

undershoot(...) plots the bilevel waveform and marks the location of the undershoot of each transition as well as the lower- and upper reference-level instants and the associated reference levels. undershoot also plots the state levels and associated lower- and upper-state boundaries.

Input Arguments

X
Bilevel waveform. X is a real-valued row or column vector.

FS
Sample rate in hertz.

T
Vector of sample instants. The length of T must equal the length of the bilevel waveform, X.

Name-Value Pair Arguments

PercentReferenceLevels
Reference levels as a percentage of the waveform amplitude. The lower-state level is defined to be 0 percent. The upper-state level is defined to be 100 percent. The value of 'PercentReferenceLevels' is a 2-element real row vector whose elements correspond to the lower and upper percent reference levels.

Default: [10 90]

Region
Specify the region over which to perform the undershoot computation. Valid values for 'Region' are 'Preshoot' or 'Postshoot'. If you specify 'Preshoot', the end of the pretransition aberration region is defined as the last instant when the signal exits the first...
state. If you specify 'Postshoot', the start of the posttransition aberration region is defined as the instant when the signal enters the second state.

**Default:** 'Postshoot'

**SeekFactor**

Aberration region duration. Specifies the duration of the region over which to compute the undershoot for each transition as a multiple of the corresponding transition duration. The edge of the waveform may be reached, or a complete intervening transition may be detected, before the duration aberration region duration elapses. In such cases, the duration is truncated to the edge of the waveform or the start of the intervening transition.

**Default:** 3

**StateLevels**

Lower- and upper-state levels. Specify the levels to use for the lower- and upper-state levels as a 2-element real row vector whose first and second elements correspond to the lower- and upper-state levels of the input waveform.

**Tolerance**

Specify the tolerance that the initial and final levels of each transition must be within the respective state levels. The 'Tolerance' value is a scalar expressing a percentage of the difference between the upper- and lower-state levels. See “State-Level Tolerances” on page 1-3032.

**Default:** 2

**Output Arguments**

**US**

Undershoots expressed as a percentage of the state levels. The undershoot percentages are computed based on the greatest deviation from the final state level in each transition. By default undershoots are computed for posttransition aberration regions. See “Undershoot” on page 1-3031.
**USLEV**

Level of the pretransition or posttransition undershoot.

**USINST**

Sample instants of pretransition or posttransition undershoots. If you specify the sample rate or sampling instants, the undershoot instants are in seconds. If you do not specify the sample rate or sampling instants, the undershoot instants are the indices of the input vector.

**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]);
```
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.

\[ \text{load('transitionex.mat','x','t')} \]

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.

\[
\text{load('transitionex.mat','x','t')}
\]

Determine the maximum percent undershoot, the level of the undershoot in volts, and the sampling instant where the maximum undershoot occurs. Plot the result.

\[
\text{[us,uslev,usinst] = undershoot(x,t,'Region','Preshoot')}
\]

\[
\text{us = 6.1798}
\]

\[
\text{uslev = -0.1500}
\]

\[
\text{usinst = 5.0000e-06}
\]

\[
\text{undershoot(x,t,'Region','Preshoot');}
\]
More About

Undershoot

For a positive-going (positive-polarity) pulse, undershoot expressed as a percentage is

\[ \frac{100}{S_2 - S_1} \frac{(S_2 - U)}{(S_2 - S_1)} \]

where \( U \) is the greatest deviation below the high-state level, \( S_2 \) is the high state, and \( S_1 \) is the low state.
For a negative-going (negative-polarity) pulse, undershoot expressed as a percentage is

\[ \frac{100(S_1 - U)}{(S_2 - S_1)} \]

The following figure illustrates the calculation of undershoot for a positive-going transition.

The red dashed lines indicate the estimated state levels. The double-sided black arrow depicts the difference between the high- and low-state levels. The solid black line indicates the difference between the high-state level and the undershoot value.

**State-Level Tolerances**

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and the low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the \( \alpha \% \) region for the low state is defined as
\[ S_1 \pm \frac{\alpha}{100}(S_2 - S_1), \]

where \( S_1 \) is the low-state level and \( S_2 \) is the high-state level. Replace the first term in the equation with \( S_2 \) to obtain the \( \alpha\% \) tolerance region for the high state.

The following figure illustrates lower and upper 2\% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The red dashed lines indicate the estimated state levels.
References


See Also
overshoot | settlingtime | statelevels

Introduced in R2012a
validstructures

Structures for specification object with design method

Syntax

filtstruct = validstructures(D,'Systemobject',true)
filtstruct = validstructures(D,METHOD,'Systemobject',true)

Description

filtstruct = validstructures(D,'Systemobject',true) returns a structure of cell arrays, filtstruct, which contains a set of valid filter structures for the filter specification object, D. When you set 'Systemobject' to true, validstructures returns a list of structures that support filter System objects. Each field in filtstruct lists a set of filter structures for the design method specified.

filtstruct = validstructures(D,METHOD,'Systemobject',true) returns the valid structures for the filter specification object, D, and the design method, METHOD, in a cell array of character vectors.

Examples

Valid Filter Structures

Design a default lowpass filter specification object. Return all valid design methods and structures in a structure array. Display the fieldnames to see all valid design methods. Display the valid filter structures for the equiripple field.

D = fdesign.lowpass;
filtstruct = validstructures(D);

fn = fieldnames(filtstruct)

fn = 8x1 cell array
{ 'butter'   }
strs = eval([{'filtstruct.' fn{5}}])

strs = 1x4 cell array
    {'dffir'}    {'dffirt'}    {'dfsymfir'}    {'fftfir'}

Create a highpass filter of order 50 with a 3-dB frequency of 0.2. Obtain the available structures for a Butterworth design.

D = fdesign.highpass('N,F3dB',50,0.2);
C = validstructures(D,'butter')

C = 1x6 cell array
Columns 1 through 4
    {'df1sos'}    {'df2sos'}    {'df1tsos'}    {'df2tsos'}
Columns 5 through 6
    {'cascadeallpass'}    {'cascadewdfallpass'}

See Also
    design | designmethods | designopts | fdesign

Introduced in R2009a
vco

Voltage controlled oscillator

Syntax

\[
y = \text{vco}(x, f_c, f_s) \\
y = \text{vco}(x, [F_{min} \ F_{max}], f_s)
\]

Description

\( y = \text{vco}(x, f_c, f_s) \) creates a signal that oscillates at a frequency determined by the real input vector or array \( x \) with sampling frequency \( f_s \). \( f_c \) is the carrier or reference frequency; when \( x \) is 0, \( y \) is an \( f_c \) Hz cosine with amplitude 1 sampled at \( f_s \) Hz. \( x \) ranges from -1 to 1, where \( x = -1 \) corresponds to 0 frequency output, \( x = 0 \) corresponds to \( f_c \), and \( x = 1 \) corresponds to \( 2*f_c \). Output \( y \) is the same size as \( x \).

\( y = \text{vco}(x, [F_{min} \ F_{max}], f_s) \) scales the frequency modulation range so that ±1 values of \( x \) yield oscillations of \( F_{min} \) Hz and \( F_{max} \) Hz respectively. For best results, \( F_{min} \) and \( F_{max} \) should be in the range 0 to \( f_s/2 \).

By default, \( f_s \) is 1 and \( f_c \) is \( f_s/4 \).

If \( x \) is a matrix, \( \text{vco} \) produces a matrix whose columns oscillate according to the columns of \( x \).

Examples

Spectrogram of Sawtooth Signal

Generate two seconds of a signal sampled at 10 kHz whose instantaneous frequency is a triangle function of time.
fs = 10000;
t = 0:1/fs:2;
x = vco(sawtooth(2*pi*t,0.75),[0.1 0.4]*fs,fs);

Plot the spectrogram of the generated signal.
spectrogram(x,kaiser(256,5),220,512,fs,'yaxis')

Diagnostics
If any values of $x$ lie outside [-1, 1], vco gives the following error message:
X outside of range $[-1,1]$.

**Algorithms**

`vco` performs FM modulation using the `modulate` function.

**See Also**

demod | modulate

**Introduced before R2006a**
**window**

Window function gateway

**Syntax**

```matlab
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 `@`.

```matlab
@barthannwin
@bartlett
@blackman
@blackmanharris
@bohmanwin
@chebwin
@flattopwin
@gausswin
@hamming
@hann
@kaiser
@nuttallwin
@parzenwin
@rectwin
@taylorwin
@triang
@tukeywin
```
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.

\[ w = \text{window}(\text{fhandle}, n, \text{winopt}) \]

returns the window specified by its function handle, \( \text{fhandle} \), and its \( \text{winopt} \) value or sampling descriptor. For chebwin, kaiser, and tukeywin, you must enter a \( \text{winopt} \) value. For the other windows listed in the following table, \( \text{winopt} \) values are optional.

<table>
<thead>
<tr>
<th>Window</th>
<th>winopt Description</th>
<th>winopt Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackman</td>
<td>window sampling</td>
<td>'periodic' or 'symmetric'</td>
</tr>
<tr>
<td>chebwin</td>
<td>sidelobe attenuation relative to mainlobe</td>
<td>numeric</td>
</tr>
<tr>
<td>flattopwin</td>
<td>window sampling</td>
<td>'periodic' or 'symmetric'</td>
</tr>
<tr>
<td>gausswin</td>
<td>alpha value (reciprocal of standard deviation)</td>
<td>numeric</td>
</tr>
<tr>
<td>hamming</td>
<td>window sampling</td>
<td>'periodic' or 'symmetric'</td>
</tr>
<tr>
<td>hann</td>
<td>window sampling</td>
<td>'periodic' or 'symmetric'</td>
</tr>
<tr>
<td>kaiser</td>
<td>beta value</td>
<td>numeric</td>
</tr>
<tr>
<td>taylorwin</td>
<td>1. number of sidelobes</td>
<td>1. integer greater than or equal to 1</td>
</tr>
<tr>
<td></td>
<td>2. maximum sidelobe level in dB relative to mainlobe peak</td>
<td>2. negative value</td>
</tr>
<tr>
<td>tukeywin</td>
<td>ratio of taper to constant sections</td>
<td>numeric</td>
</tr>
</tbody>
</table>

### 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 | taylorwin | triang | tukeywin

Introduced before R2006a
window (filter design method)
FIR filter using windowed impulse response

Syntax

\[
\begin{align*}
  h &= \text{window}(d, 'window', \text{fcnhdl}) \\
  h &= \text{window}(d, \text{win})
\end{align*}
\]

Description

Note This is a description of the overloaded method used in conjunction with \texttt{fdesign} to design a filter from a filter specification object. To access the window function gateway see \texttt{window}.

\[
\begin{align*}
  h &= \text{window}(d, 'window', \text{fcnhdl}) \text{ designs an FIR filter using the specifications in filter specification object } d. \\
  \text{fcnhdl} & \text{ is a handle to a filter design function that returns a window vector, such as the \texttt{hamming} or \texttt{blackman} functions. } \text{fcnarg} & \text{ is an optional argument that returns a window. You pass the function to window. Refer to example 1 in the following section to see the function argument used to design the filter.} \\
  h &= \text{window}(d, \text{win}) \text{ designs a filter using the vector you supply in win. The length of vector win must be the same as the impulse response of the filter, which is equal to the filter order plus one.}
\end{align*}
\]

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.

```plaintext
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.

```matlab
whos bohwin
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bohwin</td>
<td>128x1</td>
<td>1024</td>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>

**Kaiser Windows**

Use **Window Designer** to see how the behavior of a Kaiser window depends on the window length and the shape parameter, $\beta$.

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.
Select **kaiser6** from the **Name** list. Leave **Beta** set to 6 and set **Length** to 40. Click **Apply**.

Select **kaiser3** from the **Name** list. Click the **Turn Legend On** button.

![Window Designer GUI](image)
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.
• “Generalized Cosine Windows”
• “Kaiser Window”
• “Chebyshev Window”
See Also

**Apps**
Filter Designer | Signal Analyzer

**Functions**
wvtool

**Topics**
“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 π/2. Compute the Wigner-Ville distribution of the sum of the two signals.

```matlab
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 \texttt{wvd} with no output arguments.

\texttt{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.

```matlab
fs = 1000;
t = (0:1/fs:1.5)';
x = cos(2*pi*t*200);
```

Compute the Wigner-Ville distribution of the signal.

```matlab
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.

```matlab
x = x + vco(cos(2*pi*t),[250 450],fs);
x_t = timetable(seconds(t),x);
wvd(x_t)
```
Set to zero the distribution elements with amplitude less than 0.

\[ \text{wvd}(xt,'\text{MinThreshold}',0) \]
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.

\[ fs = 1000; \]
\[ t = 0:1/fs:1-1/fs; \]
\[ x = \text{chirp}(t,100,1,400,'\text{quadratic}') + \text{chirp}(t,350,1,50); \]

Compute the Wigner-Ville distribution of the signal.

\[ \text{wvd}(x,fs) \]

Compute the smoothed pseudo Wigner-Ville distribution of the signal. Specify 201 frequency points and 502 time points.

\[ \text{wvd}(x,fs,'\text{smoothedPseudo}', '\text{NumFrequencyPoints}',201, '\text{NumTimePoints}',502) \]
Increase the number of time points so the quadratic chirp becomes visible.

\[ \text{wvd}(x,fs,'\text{smoothedPseudo}',\text{NumFrequencyPoints},201,\text{NumTimePoints},522) \]
Increase the frequency points and time points to get a sharper image.

\[
\text{wvd}(x, fs, 'smoothedPseudo', 'NumFrequencyPoints', 1000, 'NumTimePoints', 1502)
\]
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.

```matlab
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.

\texttt{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” (MATLAB).
If the input signal has odd length, the function appends a zero to make the length even.

Example: \( \cos(\pi/8*(0:159))' + \text{randn}(160,1)/10 \) specifies a sinusoid embedded in white noise.

Example: \( \text{timetable(seconds(0:5)',rand(6,1))} \) 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, \text{wvd} uses Kaiser windows with shape factor \( \beta = 20 \).

- The default length of \text{twin} is the smallest odd integer greater than or equal to \( \text{round(length(x)/10)} \).
- The default length of \text{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 \( 2*\text{ceil(length(x)/2)} \).

Example: \text{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 \text{thresh}.
Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

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. The number of time points must be at least 2*length(twin) and cannot be greater than the default. 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

\[
WVD_x(t, f) = \int_{-\infty}^{\infty} x(t + \frac{\tau}{2})x^*(t - \frac{\tau}{2})e^{-j2\pi f \tau} d\tau.
\]

For a discrete signal with \( N \) samples, the distribution becomes

\[
WVD_x(n, k) = \sum_{m = -N}^{N} x(n + m/2)x^*(n - m/2) e^{-j2\pi km/N}.
\]

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:
\[ \text{SPWVD}_{x,H}^{g}(t, f) = \int_{-\infty}^{\infty} g(t) H(f) x(t + \frac{T}{2}) x^*(t - \frac{T}{2}) e^{-j2\pi f \tau} d\tau. \]

**References**


**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**

*Functions*

fsst | psspectrum | spectrogram | xwvd

*Introduced in R2018b*
wvtool

Open Window Visualization Tool

Syntax

wvtool(WindowVector)

wvtool(WindowVector1,...,WindowVectorN)

H = wvtool(...)

Description

wvtool(WindowVector) opens the Window Visualization Tool (WVTool) with time and frequency domain plots of the window vector specified in WindowVector. WindowVector must be a real-valued row or column vector. By default, the frequency domain plot is the magnitude squared of the Fourier transform of the window vector in decibels (dB). You can generate window vectors for a number of common window functions using the Signal Processing Toolbox software. See window for a list of supported window functions.

wvtool(WindowVector1,...,WindowVectorN) opens WVTool with time and frequency domain plots of the window vectors specified in WindowVector1,..., WindowVectorN.

H = wvtool(...) returns the figure handle, H.

Note A related tool, Window Designer, is available for designing and analyzing windows.
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 menus items, wvtool contains these wvtool-specific menu commands:

File menu:

- **Export** — Exports the displayed plot(s) to a graphic file.

Edit menu:
• **Copy figure** — Copies the displayed plot(s) to the clipboard (available only on Windows platforms).

• **Copy options** — Displays the Preferences dialog box (available only on Windows platforms).

• **Figure, Axes, and Current Object Properties** — Displays the Property Editor.

**View menu:**

• **Time domain** — Check to show the time domain plot.

• **Frequency domain** — Check to show the frequency domain plot.

• **Legend** — Toggles the window name legend on and off. This option is also available with the **Legend** toolbar button.

• **Analysis Parameters** — Controls the response plot parameters, including number of points, range, x- and y-axis units, sampling frequency, and normalized magnitude.

  You can also access the Analysis Parameters by right-clicking the x-axis label of a plot in the Window Viewer panel.

• **Insert** menu:

  You use the **Insert** menu to add labels, titles, arrows, lines, text, and axes to your plots.

**Tools menu:**

• **Edit Plot** — Turns on plot editing mode

• **Zoom In** — Zooms in along both x- and y-axes.

• **Zoom X** — Zooms in along the x-axis only. Drag the mouse in the x direction to select the zoom area.

• **Zoom Y** — Zooms in along the y-axis only. Drag the mouse in the y direction to select the zoom area.

• **Full View** — Returns to full view.

**Examples**

**Display and Compare Windows**

Use `wvtool` to display and compare 64-point Hamming, Hann, and Gaussian windows.
Compare 128-point Kaiser windows with different values of $\beta$.

\[
\text{wvtool}(\text{kaiser}(128, 1.5), \text{kaiser}(128, 4.5))
\]
See Also

Apps
Filter Designer | Window Designer

Introduced before R2006a
**xcorr2**

2-D cross-correlation

**Syntax**

\[ c = \text{xcorr2}(a,b) \]
\[ c = \text{xcorr2}(a) \]

**Description**

\[ c = \text{xcorr2}(a,b) \] returns the cross-correlation of matrices \( a \) and \( b \) with no scaling. \text{xcorr2} is the two-dimensional version of \text{xcorr}.

\[ c = \text{xcorr2}(a) \] is the autocorrelation matrix of input matrix \( a \). This syntax is equivalent to \text{xcorr2}(a,a).

**Examples**

**Output Matrix Size and Element Computation**

Create two matrices, \( M1 \) and \( M2 \).

\[
M1 = \begin{bmatrix}
17 & 24 & 1 & 8 & 15 \\
23 & 5 & 7 & 14 & 16 \\
4 & 6 & 13 & 20 & 22 \\
10 & 12 & 19 & 21 & 3 \\
11 & 18 & 25 & 2 & 9 \\
\end{bmatrix};
\]

\[
M2 = \begin{bmatrix}
8 & 1 & 6 \\
3 & 5 & 7 \\
4 & 9 & 2 \\
\end{bmatrix};
\]

\( M1 \) is 5-by-5 and \( M2 \) is 3-by-3, so their cross-correlation has size \((5+3-1)\times(5+3-1)\), 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 \( M_2 \) 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 \( M_2 \) 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

\[1 \times 8 + 7 \times 3 + 13 \times 4 + 8 \times 1 + 14 \times 5 + 20 \times 9 + 15 \times 6 + 16 \times 7 + 22 \times 2 = 585.\]

\[[r2,c2] = \text{size}(M2);\]

\[CC = \text{sum(sum}(M1(0+(1:r2),2+(1:c2)).*M2))\]

\[CC = 585\]

Verify the result using $\text{xcorr2}$. 

1-3080
Two-Dimensional Cross-Correlation of Arbitrary Complex Matrices

Given a matrix $X$ of size $M \times N$ and a matrix $\mathcal{H}$ of size $P \times Q$, their two-dimensional cross-correlation, $C = X \ast \mathcal{H}$, is a matrix of size $(M + P - 1) \times (N + Q - 1)$ with elements

$$C(k, l) = \text{Tr} \left\{ \tilde{X} \tilde{\mathcal{H}}^\dagger_{kl} \right\} \quad 1 \leq k \leq M + P - 1, \quad 1 \leq l \leq N + Q - 1.$$

$\text{Tr}$ is the trace and the dagger denotes Hermitian conjugation. The matrices $\tilde{X}$ and $\tilde{\mathcal{H}}_{kl}$ have size $(M + 2(P - 1)) \times (N + 2(Q - 1))$ and nonzero elements given by

$$\tilde{X}(m, n) = X(m - P + 1, n - Q + 1), \quad P \leq m \leq M + P - 1, \quad Q \leq n \leq N + Q - 1$$

and

$$\tilde{\mathcal{H}}_{kl}(p, q) = \mathcal{H}(p - k + 1, q - l + 1), \quad k \leq p \leq P + k - 1, \quad l \leq q \leq Q + l - 1.$$

Calling $\text{xcorr2}$ is equivalent to this procedure for general complex matrices of arbitrary size.

Create two complex matrices, $X$ of size $7 \times 22$ and $\mathcal{H}$ of size $6 \times 17$.

```matlab
X = randn([7 22])+1j*randn([7 22]);
H = randn([6 17])+1j*randn([6 17]);
```

```matlab
[M,N] = size(X);
m = 1:M;
n = 1:N;

[P,Q] = size(H);
p = 1:P;
q = 1:Q;
```
Initialize \( \tilde{X} \) and \( C \).

\[
X_t = \text{zeros}([M+2*(P-1) \ N+2*(Q-1)]);
X_t(m+P-1,n+Q-1) = X;
C = \text{zeros}([M+P-1 \ N+Q-1]);
\]

Compute the elements of \( C \) by looping over \( k \) and \( l \). Reset \( \mathcal{H}_{kl} \) to zero at each step. Save time and memory by summing element products instead of multiplying and taking the trace.

\[
\text{for } k = 1:M+P-1 \\
\quad \text{for } l = 1:N+Q-1 \\
\quad \quad \mathcal{H}_{kl} = \text{zeros}([M+2*(P-1) \ N+2*(Q-1)]);
\quad \quad \mathcal{H}_{kl}(p+k-1,q+l-1) = H;
\quad \quad C(k,l) = \text{sum}(%
\text{sum}(X_t.*\text{conj}(\mathcal{H}_{kl}));
\quad \end{end}
\end{for}
\]

\[
\text{max(max(abs(C-xcorr2(X,H))))}
\]

\[
\text{ans} = 1.4648e-14
\]

The answer coincides to machine precision with the output of \texttt{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 \texttt{xcorr2}.

Load a black-and-white test image into the workspace. Display it with \texttt{imagesc}.

\[
\text{load durer}
\text{img} = X;
\text{White} = \text{max(max(img))};
\]

\[
\text{imagesc(img)}
\text{axis image off}
\text{colormap gray}
\text{title('Original')}
\]
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 \texttt{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.

\begin{verbatim}
    nimg = img - mean(mean(img));
nSec = nimg(szx,szy);
crr = xcorr2(nimg,nSec);
\end{verbatim}

The maximum of the cross-correlation corresponds to the estimated location of the lower-right corner of the section. Use \texttt{ind2sub} to convert the one-dimensional location of the maximum to two-dimensional coordinates.

\begin{verbatim}
    [ssr,snd] = max(crr(:));
    [ij,ji] = ind2sub(size(crr),snd);
\end{verbatim}
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],[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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
template = gpuArray(template);
offsetTemplate = gpuArray(offsetTemplate);
```

Compute the cross-correlation on the GPU.

```matlab
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.

```matlab
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
\[
C(k, l) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X(m, n) \overline{H(m-k, n-l)},
\]

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:

```matlab
X = ones(2,3);
H = [1 2; 3 4; 5 6];  % H is 3 by 2
C = xcorr2(X,H)
```

\[
C =
\begin{bmatrix}
6 & 11 & 11 & 5 \\
10 & 18 & 18 & 8 \\
6 & 10 & 10 & 4 \\
2 & 3 & 3 & 1
\end{bmatrix}
\]

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 \( X(1,1) \times H(3,2) = 6 \). 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(____,frequrange)
[____] = 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,overlap) uses overlap samples of overlap between adjoining segments.

s = xspectrogram(x,y,window,overlap,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,overlap,w) returns the cross-spectrogram at the normalized frequencies specified in w.

[s,f,t] = xspectrogram(x,y,window,overlap,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 pair arguments. Options include the minimum threshold and output time dimension.

[___] = xspectrogram(____,spectrumpetype) returns short-term cross power spectral density estimates if spectrumpetype is specified as 'psd' and returns short-term cross power spectrum estimates if spectrumpetype 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'.

1-3095
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.

```
xscpectrogram(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.

$x_2 = \text{chirp}(t,50e3,t(\text{end}),350e3)$;
$x_2 = x_2 + \text{randn}(\text{size}(x_2)) \times \text{std}(x_2) / \text{db2mag}(\text{SNR})$;

$\text{xspectrogram}(x_1,x_2,\text{kaiser}(500,5),450,256,Fs,'yaxis')$
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.
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.

```matlab
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.

```matlab
[~,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.

\[
[tfr,~,lridge] = \text{tfridge}(c,f);
\]

\text{hold on}

\text{plot}(xt,tfr,'k','\text{linwidth}',2)

\text{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.

\[
\text{pshft} = \text{angle}(c(\text{lridge})) \times \frac{180}{\pi};
\]

\[
\text{mean(pshft)}
\]

\[
\text{ans} = -23.0000
\]
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.

```matlab
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.

```matlab
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.

```plaintext
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 $0.1\pi$ that increases to $0.8\pi$ by the end of the measurement.

• The 16th-order filter passes normalized frequencies between $0.2\pi$ and $0.4\pi$ rad/sample and has a stopband attenuation of 60 dB.

```matlab
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 $0.9\pi$ that decreases to $0.1\pi$ by the end of the measurement.

• The 16th-order filter stops normalized frequencies between $0.6\pi$ and $0.8\pi$ rad/sample and has a passband ripple of 1 dB.

```matlab
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.

```matlab
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.

\[
x\text{spectrogram}(x, y, \text{hamming}(512), 500, 2048, \text{''yaxis''})
\]
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: \( \cos\left(\pi/4*(0:159)\right)+\text{randn}(1,160) \) 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 \([N/4.5]\), 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 \texttt{xspectrogram} sets the DFT length to $\max(256, 2^p)$, where $p = \lceil \log_2 N_w \rceil$ and

- $N_w = \text{window}$ if window is a scalar.
- $N_w = \text{length(window)}$ if window is a vector.

Data Types: \texttt{single} \mid \texttt{double}

\textit{w} — Normalized frequencies
\texttt{vector}

Normalized frequencies, specified as a vector. \texttt{w} must have at least two elements. Normalized frequencies are in rad/sample.

Example: \texttt{pi./[2 4]}

Data Types: \texttt{double} \mid \texttt{single}

\textit{f} — Frequencies
\texttt{vector}

Frequencies, specified as a vector. \texttt{f} must have at least two elements. The units of \texttt{f} are specified by the sample rate, \texttt{fs}.

Data Types: \texttt{double} \mid \texttt{single}

\textit{fs} — Sample rate
\texttt{1 Hz (default) \mid 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: \texttt{double} \mid \texttt{single}

\textit{freqrange} — Frequency range for cross-spectrum estimate
\texttt{'onesided' \mid 'twosided' \mid '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 \texttt{nfft} is even, then \texttt{s} has \texttt{nfft}/2 + 1 rows and is computed over the interval $[0, \pi]$ rad/
sample. If nfft is odd, then s has (nfft + 1)/2 rows and the interval is [0, n) 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, 2n) 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**

'spd' (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 comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.
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 the comma-separated pair consisting of `MinThreshold` and a real scalar expressed in decibels. `xspectrogram` sets to zero those elements of `s` such that $10 \log_{10}(s) \leq \text{thresh}$.

**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 — 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:
  - $k = \left\lfloor \frac{(N - \text{nooverlap})}{(\text{window} - \text{nooverlap})} \right\rfloor$ if `window` is a scalar.
  - $k = \left\lfloor \frac{(N - \text{nooverlap})}{(\text{length}(\text{window}) - \text{nooverlap})} \right\rfloor$ 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**


**See Also**
cpsd | msscohere | spectrogram
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(____)
oxwvd(____)

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.

```matlab
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 0.012.

\[
fs = 1e3;
t = (0:1/fs:1-1/fs)';
\]

\[
mu = 0.064;
sigma = 0.01;
fsin = 50;
\]

\[
xr = \exp(-(t-mu).^2/(2*sigma^2)).*\sin(2*\pi*fsin*t);
\]

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.

\[
f0 = 400;
f1 = 100;
\]

\[
xa = \text{zeros(size}(t));
xa(t>0.4 & t<=0.9) = \text{chirp}((0:1/fs:0.5-1/fs)',f0.5,f1);
\]

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.

\[
w = \text{wvd}(xa+xr,fs,'\text{smoothedPseudo}');
\]

\[
wvd(xa+xr,fs,'\text{smoothedPseudo}')
\]
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.

```matlab
[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.

\[
d = w + c;
\]

\[
d = \text{abs} \left( \text{real}(d) \right);
\]

```matlab
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.

\[
ff = tfridge(c,fc);
\]

\[
tv = \text{sum}(c) > 0;
\]

\[
ff = ff(tv);
\]

\[
tc = tc(tv);
\]

\[
\text{hold on}
\]

\[
\text{plot}(tc,ff,'r--','\text{linewidth}',2)
\]

\[
\text{hold off}
\]
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.

\[ t_{\text{Est}} = 2 \cdot t_c - \mu; \]
\[ f_{\text{Est}} = 2 \cdot f_f - f_{\text{sin}}; \]

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” (MATLAB).

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*π (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 $\text{round}(\text{length}(x)/10)$.
- The default length of `fwin` is the smallest odd integer greater than or equal to $\text{nf}/4$.

Each window must have a length smaller than or equal to $2\times\text{ceil}(\text{length}(x)/2)$.

Example: `kaiser(65,0.5)` specifies a 65-sample Kaiser window with a shape factor of 0.5.

### nf — Number of frequency points

$2\times\text{ceil}(\text{length}(x)/2)$ (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 \((\text{length}(\text{fwin})+1)/2\) and cannot be greater than the default.

\texttt{thresh} — Minimum nonzero value  
-\texttt{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 \texttt{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 \(4 \times \text{ceil}(\text{length}(x)/2)\).
More About

Cross Wigner-Ville Distribution

For continuous signals $x(t)$ and $y(t)$, the cross Wigner-Ville distribution is defined as

$$\text{XWVD}_{x,y}(t,f) = \int_{-\infty}^{\infty} x\left(t + \frac{T}{2}\right)y^\ast\left(t - \frac{T}{2}\right)e^{-j2nf\tau} d\tau.$$  

For a discrete signal with $N$ samples, the distribution becomes

$$\text{XWVD}_{x,y}(n,k) = \sum_{m = -N}^{N} x(n + m/2) y^\ast(n - m/2) e^{-j2nk \frac{m}{N}}.$$  

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:

$$\text{XSPWVD}^g_H_{x,y}(t,f) = \int_{-\infty}^{\infty} g(t) H(f) x\left(t + \frac{T}{2}\right)y^\ast\left(t - \frac{T}{2}\right)e^{-j2nf\tau} d\tau.$$  

References


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

Functions
cpsd | fsst | pspectrum | wvd | xspectrogram

Introduced in R2018b
**yulewalk**

Recursive digital filter design

**Syntax**

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

**Description**

yulewalk designs recursive IIR digital filters using a least-squares fit to a specified frequency response.

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

returns row vectors, \( b \) and \( a \), containing the \( n+1 \) coefficients of the order \( n \) IIR filter whose frequency-magnitude characteristics approximately match those given in vectors \( f \) and \( m \):

- \( f \) is a vector of frequency points, specified in the range between 0 and 1, where 1 corresponds to half the sample frequency (the Nyquist frequency). The first point of \( f \) must be 0 and the last point 1. All intermediate points must be in increasing order. Duplicate frequency points are allowed, corresponding to steps in the frequency response.
- \( m \) is a vector containing the desired magnitude response at the points specified in \( f \).
- \( f \) and \( m \) must be the same length.
- `plot(f,m)` displays the filter shape.

The output filter coefficients are ordered in descending powers of \( z \).

\[
\frac{B(z)}{A(z)} = \frac{b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \cdots + a(n+1)z^{-n}}
\]

When specifying the frequency response, avoid excessively sharp transitions from passband to stopband. You may need to experiment with the slope of the transition region to get the best filter design.
Examples

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.

\[
\begin{align*}
f & = [0 \ 0.6 \ 0.6 \ 1]; \\
m & = [1 \ 1 \ 0 \ 0]; \\
[b,a] & = \text{yulewalk}(8,f,m); \\
h, w & = \text{freqz}(b,a,128); \\
p\text{lot}(w/pi,\text{abs}(h),f,m,\text{'--'}) \\
\text{xlabel 'Radian frequency ($\omega/\pi$)'}; \ \text{ylabel 'Magnitude'} \\
\text{legend('Yule-Walker','Ideal')}, \ \text{legend boxoff}
\end{align*}
\]
**Algorithms**

`yulewalk` performs a least-squares fit in the time domain. It computes the denominator coefficients using modified Yule-Walker equations, with correlation coefficients computed by inverse Fourier transformation of the specified frequency response. To compute the numerator, `yulewalk` takes the following steps:

1. Computes a numerator polynomial corresponding to an additive decomposition of the power frequency response.
2. Evaluates the complete frequency response corresponding to the numerator and denominator polynomials.
Uses a spectral factorization technique to obtain the impulse response of the filter.
Obtains the numerator polynomial by a least-squares fit to this impulse response.

References


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 | firpm | maxflat

Introduced before R2006a
zerophase

Zero-phase response of digital filter

Syntax

\[
\begin{align*}
[Hr,w] &= \text{zerophase}(b,a) \\
[Hr,w] &= \text{zerophase}(\text{sos}) \\
[Hr,w] &= \text{zerophase}(d) \\
[Hr,w] &= \text{zerophase}(...,
\end{align*}
\]

\[
\begin{align*}
nfft\right) \\
[Hr,w] &= \text{zerophase}(...,
\end{align*}
\]

\[
\begin{align*}
[Hr,w,\phi] &= \text{zerophase}(...) \\
\text{zerophase}(...) \\
\end{align*}
\]

Description

\[
[Hr,w] = \text{zerophase}(b,a) \text{ returns the zero-phase response } H_r \text{, and the frequency vector } w \text{ (in radians/sample) at which } H_r \text{ is computed, given a filter defined by numerator } b \text{ and denominator } a. \text{ For FIR filters where } a=1, \text{ you can omit the value } a \text{ from the command. The zero-phase response is evaluated at 512 equally spaced points on the upper half of the unit circle.}
\]

The zero-phase response, \( H_r(\omega) \), is related to the frequency response, \( H(e^{j\omega}) \), by

\[
H(e^{j\omega}) = H_r(\omega)e^{j\phi(\omega)},
\]

where \( \phi(\omega) \) is the continuous phase.

**Note** The zero-phase response is always real, but it is not the equivalent of the magnitude response. The former can be negative while the latter cannot be negative.

\[
[Hr,w] = \text{zerophase}(\text{sos}) \text{ returns the zero-phase response for the second order sections matrix, } \text{sos}. \text{ sos is a K-by-6 matrix, where the number of sections, } K, \text{ must be}
\]
greater than or equal to 2. If the number of sections is less than 2, `zerophase` considers the input to be the numerator vector, \( b \). Each row of \( \text{sos} \) corresponds to the coefficients of a second order (biquad) filter. The \( i \)th row of the \( \text{sos} \) matrix corresponds to \([b_i(1) b_i(2) b_i(3) a_i(1) a_i(2) a_i(3)]\).

\[
[Hr,w] = \text{zerophase}(d) \quad \text{returns the zero-phase response for the digital filter, } d.\text{ Use } \text{designfilt} \text{ to generate } d \text{ based on frequency-response specifications.}
\]

\[
[Hr,w] = \text{zerophase}(\ldots,nfft) \quad \text{returns the zero-phase response } H_r \text{ and frequency vector } w \text{ (radians/sample), using } nfft \text{ frequency points on the upper half of the unit circle. For best results, set } nfft \text{ to a value greater than the filter order.}
\]

\[
[Hr,w] = \text{zerophase}(\ldots,nfft,'\text{whole}') \quad \text{returns the zero-phase response } H_r \text{ and frequency vector } w \text{ (radians/sample), using } nfft \text{ frequency points around the whole unit circle.}
\]

\[
[Hr,w] = \text{zerophase}(\ldots,w) \quad \text{returns the zero-phase response } H_r \text{ and frequency vector } w \text{ (radians/sample) at frequencies in vector } w. \text{ The vector } w \text{ must have at least two elements.}
\]

\[
[Hr,f] = \text{zerophase}(\ldots,f,fs) \quad \text{returns the zero-phase response } H_r \text{ and frequency vector } f \text{ (Hz), using the sampling frequency } fs \text{ (in Hz), to determine the frequency vector } f \text{ (in Hz) at which } H_r \text{ is computed. The vector } f \text{ must have at least two elements.}
\]

\[
[Hr,w,\phi] = \text{zerophase}(\ldots) \quad \text{returns the zero-phase response } H_r, \text{ frequency vector } w \text{ (rad/sample), and the continuous phase component, } \phi. \text{ (Note that this quantity is not equivalent to the phase response of the filter when the zero-phase response is negative.)}
\]

`zerophase(...)` plots the zero-phase response versus frequency. If you input the filter coefficients or second order sections matrix, the current figure window is used. If you input a `digitalFilter`, the step response is displayed in `FVTool`.

**Note** If the input to `zerophase` is single precision, the zero-phase response is calculated using single-precision arithmetic. The output, \( H_r \), 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.

```matlab
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)
```

![Zero-phase Response](image)
Design the same filter using fircls1, which uses linear units to measure the ripple and attenuation. Display the zero-phase response.

\[
p_{Ap} = 10^{(Ap/40)}; \\
A_{pl} = (p_{Ap}-1)/(p_{Ap}+1); \\
p_{As} = 10^{(As/20)}; \\
A_{sl} = 1/p_{As};
\]

\[
b = \text{fircls1}(N_f,F_c,A_{pl},A_{sl}); \\
\text{zerophase}(b)
\]
**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.

```matlab
d = designfilt('lowpassiir','FilterOrder',10,'PassbandFrequency',0.4, ... 'PassbandRipple',0.5,'StopbandAttenuation',20,'DesignMethod','ellip');
zerophase(d,512,'whole')
```

Create the same filter using `ellip`. Plot its zero-phase response.
[b,a] = ellip(10,0.5,20,0.4);
zerophase(b,a,512,'whole')

See Also
FVTool | designfilt | digitalFilter | freqs | freqz | 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

zp2sos converts a discrete-time zero-pole-gain representation of a given digital filter to an equivalent second-order section representation.

[sos,g] = zp2sos(z,p,k) creates a matrix sos in second-order section form with gain g equivalent to the discrete-time zero-pole-gain filter represented by input arguments z, p, and k. Vectors z and p contain the zeros and poles of the filter's transfer function \( H(z) \), not necessarily in any particular order.

\[
H(z) = k \frac{(z - z_1)(z - z_2)\cdots(z - z_n)}{(z - p_1)(z - p_2)\cdots(z - p_m)}
\]

where \( n \) and \( m \) are the lengths of \( z \) and \( p \), respectively, and \( k \) is a scalar gain. The zeros and poles must be real or complex conjugate pairs. sos is an \( L \)-by-6 matrix

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

whose rows contain the numerator and denominator coefficients \( b_{ik} \) and \( a_{ik} \) of the second-order sections of \( H(z) \).
\[ H(z) = g \prod_{k=1}^{L} H_k(z) = g \prod_{k=1}^{L} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}} \]

The number \( L \) of rows of the matrix \( \text{sos} \) is the closest integer greater than or equal to the maximum of \( n/2 \) and \( m/2 \).

\([\text{sos}, g] = \text{zp2sos}(z, p, k, 'order')\) specifies the order of the rows in \( \text{sos} \), where 'order' is

- 'down', to order the sections so the first row of \( \text{sos} \) contains the poles closest to the unit circle
- 'up', to order the sections so the first row of \( \text{sos} \) contains the poles farthest from the unit circle (default)

\([\text{sos}, g] = \text{zp2sos}(z, p, k, 'order', 'scale')\) specifies the desired scaling of the gain and the numerator coefficients of all second-order sections, where 'scale' is

- 'none', to apply no scaling (default)
- 'inf', to apply infinity-norm scaling
- 'two', to apply 2-norm scaling

Using infinity-norm scaling in conjunction with up-ordering minimizes the probability of overflow in the realization. Using 2-norm scaling in conjunction with down-ordering minimizes the peak round-off noise.

**Note** Infinity-norm and 2-norm scaling are appropriate only for direct-form II implementations.

\([\text{sos}, g] = \text{zp2sos}(z, p, k, 'order', 'scale', \text{zeroflag})\) specifies whether to keep together real zeros that are the negatives of each other instead of ordering them according to proximity to poles. Setting \( \text{zeroflag} \) to true keeps the zeros together and results in a numerator with a middle coefficient equal to zero. The default for \( \text{zeroflag} \) is false.

\( \text{sos} = \text{zp2sos}(\ldots) \) embeds the overall system gain, \( g \), in the first section, \( H_1(z) \), so that

\[ H(z) = \prod_{k=1}^{L} H_k(z) \]
Note Embedding the gain in the first section when scaling a direct-form II structure is not recommended and may result in erratic scaling. To avoid embedding the gain, use `ss2sos` with two outputs.

Examples

Second-Order Sections from Zero-Pole-Gain Parameters

Design a 5th-order Butterworth lowpass filter using the function `butter` with output expressed in zero-pole-gain form. Specify the cutoff frequency to be one-fifth of the Nyquist frequency. Convert the result to second-order sections. Visualize the magnitude response.

```matlab
[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)`
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:
Match the poles closest to the unit circle with the zeros closest to those poles.

Match the poles next closest to the unit circle with the zeros closest to those poles.

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.

It orders the sections according to the proximity of the pole pairs to the unit circle. zp2sos normally orders the sections with poles closest to the unit circle last in the cascade. You can tell zp2sos to order the sections in the reverse order by specifying the down flag.

zp2sos scales the sections by the norm specified in the 'scale' argument. For arbitrary $H(\omega)$, the scaling is defined by

$$\|H\|_p = \left[ \frac{1}{2\pi} \int_0^{2\pi} |H(\omega)|^p d\omega \right]^{1/p}$$

where $p$ can be either $\infty$ or 2. See the references for details on the scaling. This scaling is an attempt to minimize overflow or peak round-off noise in fixed point filter implementations.

References


See Also
cplxpair | filternorm | sos2zp | ss2sos | tf2sos | zp2ss | zp2tf
Introduced before R2006a
**zp2ss**

Convert zero-pole-gain filter parameters to state-space form

**Syntax**

\[ [A, B, C, D] = \text{zp2ss}(z, p, k) \]

**Description**

\text{zp2ss} converts a zero-pole-gain representation of a given system to an equivalent state-space representation.

\[ [A, B, C, D] = \text{zp2ss}(z, p, k) \]
finds a single input, multiple output, state-space representation

\[
\begin{align*}
\dot{x} &= Ax + Bu \\
y &= Cx + Du
\end{align*}
\]

given a system in factored transfer function form.

\[
H(s) = \frac{Z(s)}{P(s)} = k \frac{(s - z_1)(s - z_2)\cdots(s - z_n)}{(s - p_1)(s - p_2)\cdots(s - p_n)}
\]

Column vector \( p \) specifies the pole locations, and matrix \( z \) the zero locations with as many columns as there are outputs. The gains for each numerator transfer function are in vector \( k \). The \( A, B, C, \) and \( D \) matrices are returned in controller canonical form.

\( \text{Inf} \) values may be used as place holders in \( z \) if some columns have fewer zeros than others.

**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.

\[ z = [0 \ 0]; \]
\[ p = \text{roots}([1 0.01 1]) \]

\[ p = 2\times1 \text{ complex} \]

\[ \begin{array}{c}
-0.0050 + 1.0000i \\
-0.0050 - 1.0000i \\
\end{array} \]

\[ k = 1; \]

Use \texttt{zp2ss} to find the state-space matrices.

\[ [A,B,C,D] = \text{zp2ss}(z,p,k) \]

\[ A = 2\times2 \]

\[ \begin{array}{cc}
-0.0100 & -1.0000 \\
1.0000 & 0 \\
\end{array} \]

\[ B = 2\times1 \]

\[ \begin{array}{c}
1 \\
0 \\
\end{array} \]

\[ C = 1\times2 \]
D = 1

Algorithms

zp2ss, for single-input systems, groups complex pairs together into two-by-two blocks down the diagonal of the A matrix. This requires the zeros and poles to be real or complex conjugate pairs.

See Also
sos2ss | ss2zp | tf2ss |zp2sos |zp2tf

Introduced before R2006a
**zp2tf**

Convert zero-pole-gain filter parameters to transfer function form

**Syntax**

\[ [b,a] = \text{zp2tf}(z,p,k) \]

**Description**

\[ [b,a] = \text{zp2tf}(z,p,k) \]

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

\[
Y(s) = \frac{s^2 U(s)}{s^2 + 0.01 s + 1}.
\]
The system has unit gain, a double zero at \( s = 0 \), and two complex-conjugate poles.

\[
k = 1; \\
z = [0 \ 0]'; \\
p = \text{roots([1 0.01 1])}
\]

\[
p = 2\times1 \text{ complex} \\
-0.0050 + 1.0000i \\
-0.0050 - 1.0000i
\]

Use \texttt{zp2tf} to find the transfer function.

\[
[b,a] = \text{zp2tf}(z,p,k)
\]

\[
b = 1\times3 \\
1 \ 0 \ 0
\]

\[
a = 1\times3 \\
1.0000 \ 0.0100 \ 1.0000
\]

**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 \texttt{Inf} values as placeholders in \( z \) if some columns have fewer zeros than others.

Example: \([1 \ (1+1j)/2 \ (1-1j)/2]'

Data Types: \texttt{single} | \texttt{double}

*p* — *Poles*

*column vector*

1-3152
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

\[ [z,p,k] = \text{zpk}(d) \]

Description

\[ [z,p,k] = \text{zpk}(d) \] 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.

\[
\text{hpFilt} = \text{designfilt('highpassiir','FilterOrder',8,} \\
\text{\quad 'PassbandFrequency',75e3,'PassbandRipple',0.2,} \\
\text{\quad 'SampleRate',200e3);}
\]

\[
[z,p,k] = \text{zpk(hpFilt)}
\]

\[
z = 8 \times 1
\]

\[
1
\]

\[
1
\]

\[
1
\]

\[
1
\]

\[
1
\]

\[
1
\]

\[
1
\]

\[
1
\]

1-3155
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.

\[ [z,p,k] = \text{ellip}(4,3,30,200/500); \]
\[ \text{zplane}(z,p); \]
\[ \text{grid} \]
\[ \text{title('4th-Order Elliptic Lowpass Digital Filter')} \]
Create the same filter using `designfilt`. Use `zplane` to plot the poles and zeros. Note that this syntax of `zplane` calls `fvtool`.

```matlab
d = designfilt('lowpassiir','FilterOrder',4,'PassbandFrequency',200, ... 'PassbandRipple',3,'StopbandAttenuation',30, ... 'DesignMethod','ellip','SampleRate',1000);
zplane(d)
```

**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 \textit{zplane} plots the poles and zeros in the columns of \( z \) and \( p \) in different colors.

Data Types: \texttt{single} | \texttt{double}
Complex Number Support: Yes

\textbf{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 0.5\( \pi \) rad/sample.

Data Types: \texttt{single} | \texttt{double}
Complex Number Support: Yes

\textbf{d — Digital filter}

digitalFilter object

Digital filter, specified as a \texttt{digitalFilter} object. Use \texttt{designfilt} to generate a digital filter based on frequency-response specifications.

Example: \( d = \)
\texttt{designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)}
specifies a third-order Butterworth filter with normalized 3-dB frequency 0.5\( \pi \) rad/sample.

\section*{Output Arguments}

\textbf{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

  ```matlab
  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**