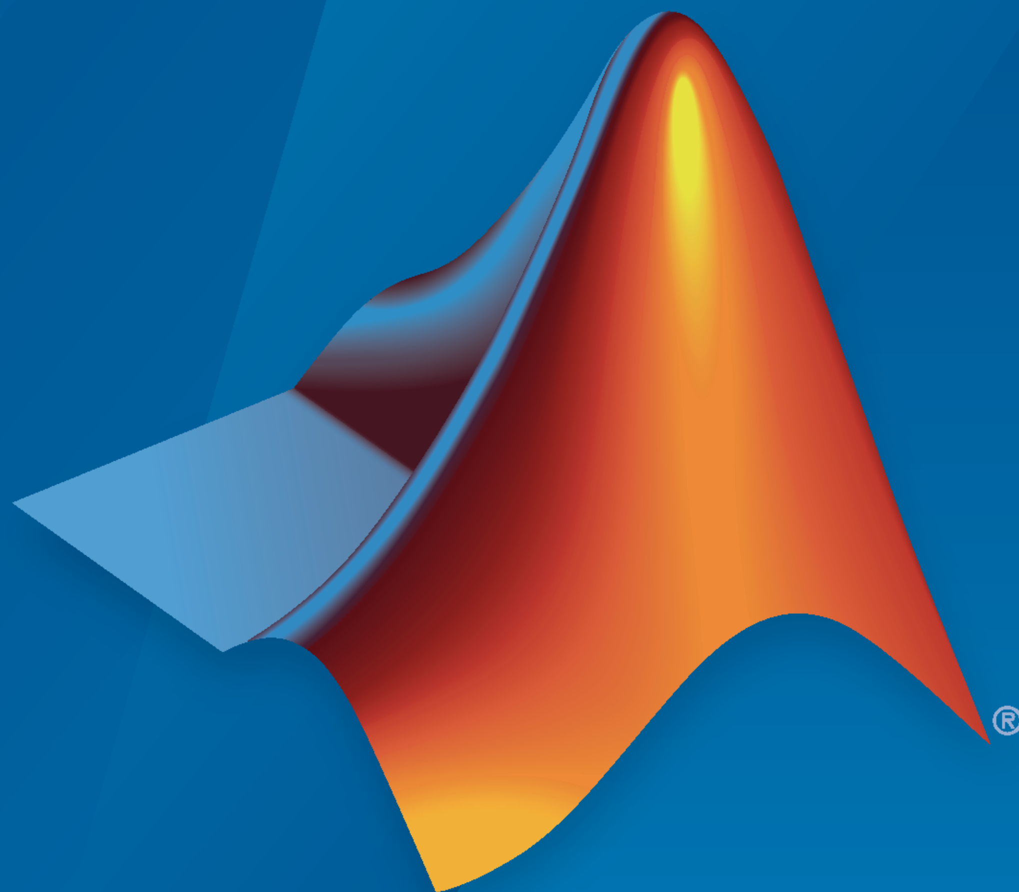
Predictive Maintenance Toolbox™

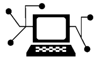
Reference



MATLAB®

R2021**b**

How to Contact MathWorks



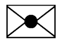
Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000



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

Predictive Maintenance Toolbox™ Reference

© COPYRIGHT 2018–2021 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 2018	Online only	New for Version 1.0 (Release 2018a)
September 2018	Online only	Revised for Version 1.1 (Release 2018b)
March 2019	Online only	Revised for Version 2.0 (Release 2019a)
September 2019	Online only	Revised for Version 2.1 (Release 2019b)
March 2020	Online only	Revised for Version 2.2 (Release 2020a)
September 2020	Online only	Revised for Version 2.2.1 (Release 2020b)
March 2021	Online only	Revised for Version 2.3 (Release 2021a)
September 2021	Online only	Revised for Version 2.4 (Release 2021b)

Contents

1 Functions

2 Objects

Functions

approximateEntropy

Measure of regularity of nonlinear time series

Syntax

```
approxEnt = approximateEntropy(X)
approxEnt = approximateEntropy(X,lag)
approxEnt = approximateEntropy(X,[],dim)
approxEnt = approximateEntropy(X,lag,dim)
approxEnt = approximateEntropy( ___ ,Name,Value)
```

Description

`approxEnt = approximateEntropy(X)` estimates the approximate entropy of the uniformly sampled time-domain signal `X` by reconstructing the phase space. Approximate entropy is a measure to quantify the amount of regularity and unpredictability of fluctuations over a time series.

`approxEnt = approximateEntropy(X,lag)` estimates the approximate entropy for the time delay `lag`.

`approxEnt = approximateEntropy(X,[],dim)` estimates the approximate entropy for the embedding dimension `dim`.

`approxEnt = approximateEntropy(X,lag,dim)` estimates the approximate entropy for the time delay `lag` and the embedding dimension `dim`.

`approxEnt = approximateEntropy( ___ ,Name,Value)` estimates the approximate entropy with additional options specified by one or more `Name,Value` pair arguments.

Examples

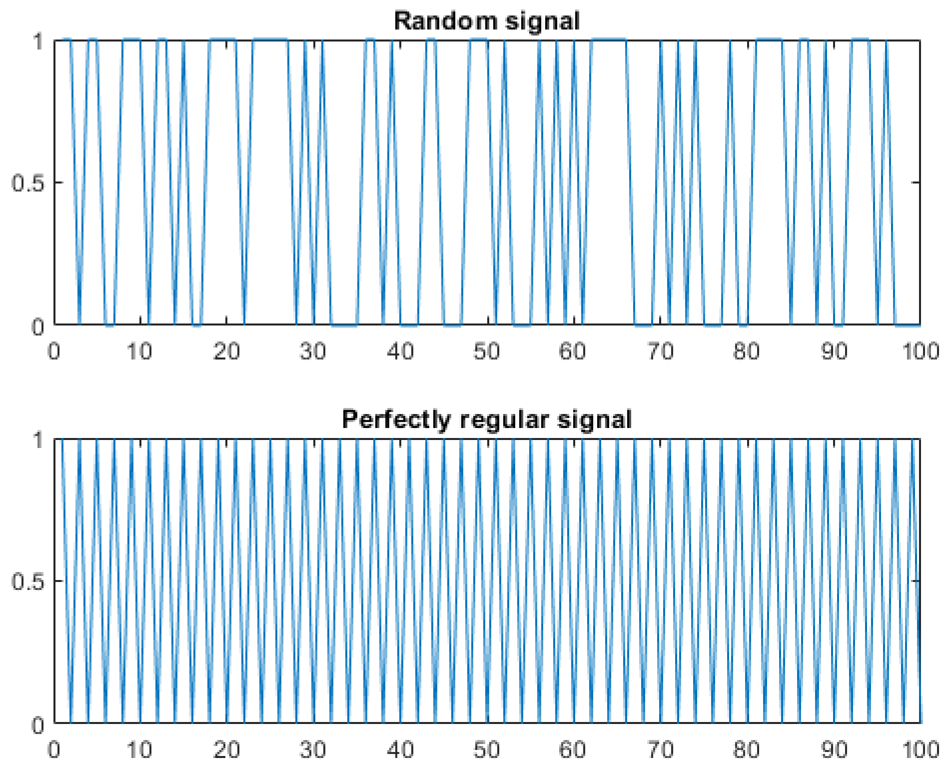
Compute Approximate Entropy of Signals

For this example, generate two signals for comparison — a random signal `xRand` and a perfectly regular signal `xReg. Set` `rng` to `default` for reproducibility of the random signal.

```
rng('default');
xRand = double(randn(100,1)>0);
xReg = repmat([1;0],50,1);
```

Visualize the random and regular signals.

```
figure;
subplot(2,1,1);
plot(xRand);
title('Random signal');
subplot(2,1,2);
plot(xReg);
title('Perfectly regular signal');
```



The plots show that the regular signal is more predictable than the random signal.

Find approximate entropy of the two signals.

```
valueReg = approximateEntropy(xReg)
```

```
valueReg = 5.1016e-05
```

```
valueIrreg = approximateEntropy(xRand)
```

```
valueIrreg = 0.6849
```

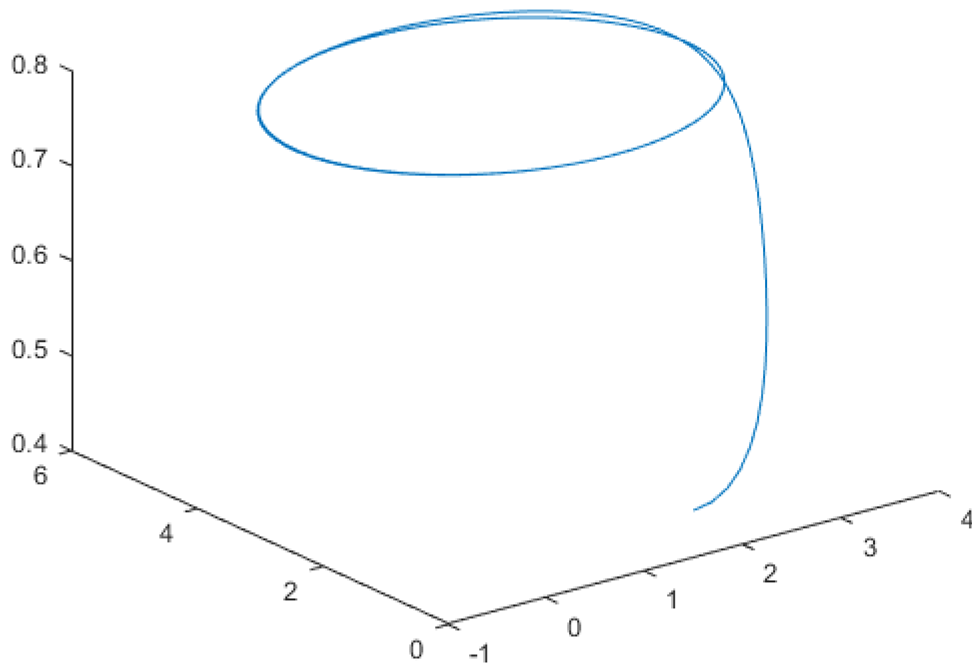
The approximate entropy of the perfectly regular signal is significantly smaller than the random signal. Hence, the perfectly regular signal containing many repetitive patterns has a relatively small value of approximate entropy while the less predictable random signal has a higher value of approximate entropy.

Find Approximate Entropy of Data

In this example, consider the position data of a quadcopter, following a circular path. The file `uavPositionData.mat` contains the x, y and z-direction position data traversed by the copter.

Load the data set and visualize the quadcopter path in 3D.

```
load('uavPositionData.mat','xv','yv','zv');
plot3(xv,yv,zv);
```



For this example, use only x-direction position data for computation. Since `Lag` is unknown, estimate the delay using `phaseSpaceReconstruction`. Set `'Dimension'` to 3. The `Dimension` and `Lag` parameters are required to compute the approximate entropy of the data.

```
dim = 3;
[~,lag] = phaseSpaceReconstruction(xv,[],dim)
```

```
lag = 10
```

Find the approximate entropy using the `Lag` value obtained in the previous step.

```
approxEnt = approximateEntropy(xv,lag,dim)
```

```
approxEnt = 0.0386
```

Since the quadcopter is traversing a pre-defined circular trajectory of fixed radius, the position data is regular and hence, the value of approximate entropy is low.

Input Arguments

X — Uniformly sampled time-domain signal

vector | array | timetable

Uniformly sampled time-domain signal, specified as either a vector, array, or timetable. If X has multiple columns, approximateEntropy computes the approximate entropy by treating X as a multivariate signal.

If X is specified as a row vector, approximateEntropy treats it as a univariate signal.

dim — Embedding dimension
scalar | vector

Embedding dimension, specified as a scalar or vector. dim is equivalent to the 'Dimension' name-value pair.

lag — Time delay
scalar | vector

Time delay, specified as a scalar or vector. lag is equivalent to the 'Lag' name-value pair.

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: ...,'Dimension',3

Dimension — Embedding dimension
2 (default) | scalar | vector

Embedding dimension, specified as the comma-separated pair consisting of 'Dimension' and a scalar or vector. When Dimension is scalar, every column in X is reconstructed using Dimension. When Dimension is a vector having same length as the number of columns in X, the reconstruction dimension for column i is Dimension(i).

Specify Dimension based on the dimension of your system. For more information on embedding dimension, see phaseSpaceReconstruction.

Lag — Delay in phase space reconstruction
1 (default) | scalar | vector

Delay in phase space reconstruction, specified as the comma-separated pair consisting of 'Lag' and a scalar. When Lag is scalar, every column in X is reconstructed using Lag. When Lag is a vector having same length as the number of columns in X, the reconstruction delay for column i is Lag(i).

If the delay is too small, random noise is introduced in the data. In contrast, if the lag is too large, the reconstructed dynamics does not represent the true dynamics of the time series. For more information on calculating optimal delay, see phaseSpaceReconstruction.

Radius — Similarity criterion
0.2*std(X) | 0.2*sqrt(trace(cov(X))) | scalar

Similarity criterion, specified as the comma-separated pair consisting of 'Radius' and a scalar. The similarity criterion, also called radius of similarity, is a tuning parameter that is used to identify a meaningful range in which fluctuations in data are to be considered similar.

The default value of Radius is,

- `0.2*std(X)`, if `X` has a single column.
- `0.2*sqrt(trace(cov(X)))`, if `X` has multiple columns.

Output Arguments

`approxEnt` — Approximate entropy of nonlinear time series

scalar

Approximate entropy of nonlinear time series, returned as a scalar. Approximate entropy is a regularity statistic that quantifies the unpredictability of fluctuations in a time series. A relatively higher value of approximate entropy reflects the likelihood that similar patterns of observations are not followed by additional similar observations.

For example, consider two binary signals `S1` and `S2`,

S1 = [0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1];

S2 = [1 1 0 1 1 1 1 0 1 0 1 0 0 0 0 1];

Signal `S1` is perfectly regular since it alternates between `0` and `1`, that is, you can predict the next value with knowledge of the previous value. Signal `S2` however offers no insight into the next value, even with prior knowledge of the previous value. Hence, signal `S2` is random and less predictable. Therefore, a signal containing highly repetitive patterns has a relatively small value of `approxEnt` while a less predictable signal has a relatively larger value of `approxEnt`.

Use `approximateEntropy` as a measure of regularity to quantify levels of complexity within a time series. The ability to discern levels of complexity within data sets is useful in the field of engineering to estimate component failure by studying their vibration and acoustic signals, or in the clinical domain where, for instance, the chance of a seizure is predicted by observing Electroencephalography (EEG) patterns.[2][3]

Algorithms

Approximate entropy is computed in the following way,

1. The `approximateEntropy` function first generates a delayed reconstruction $Y_{1:N}$ for N data points with embedding dimension m, and lag τ.
2. The software then calculates the number of within range points, at point i, given by,

   $$N_i = \sum_{i = 1, i \neq k}^{N} 1(\|Y_i - Y_k\|_\infty < R)$$

   where **1** is the indicator function, and R is the radius of similarity.
3. The approximate entropy is then calculated as $approxEnt = \Phi_m - \Phi_{m+1}$ where,

   $$\Phi_m = (N - m + 1)^{-1} \sum_{i = 1}^{N - m + 1} \log(N_i)$$

References

[1] Pincus, Steven M. "Approximate entropy as a measure of system complexity." *Proceedings of the National Academy of Sciences*. 1991 88 (6) 2297-2301; doi:10.1073/pnas.88.6.2297.

[2] U. Rajendra Acharya, Filippo Molinari, S. Vinitha Sree, Subhagata Chattopadhyay, Kwan-Hoong Ng, Jasjit S. Suri. "Automated diagnosis of epileptic EEG using entropies." *Biomedical Signal Processing and Control* Volume 7, Issue 4, 2012, Pages 401-408, ISSN 1746-8094.

[3] Caesarendra, Wahyu & Kosasih, P & Tieu, Kiet & Moodie, Craig. "An application of nonlinear feature extraction-A case study for low speed slewing bearing condition monitoring and prognosis." *IEEE/ASME International Conference on Advanced Intelligent Mechatronics: Mechatronics for Human Wellbeing, AIM 2013*.1713-1718. 10.1109/AIM.2013.6584344.

[4] Kantz, H., and Schreiber, T. *Nonlinear Time Series Analysis*. Cambridge: Cambridge University Press, 2003.

See Also

`correlationDimension` | `phaseSpaceReconstruction` | `lyapunovExponent`

Introduced in R2018a

bearingFaultBands

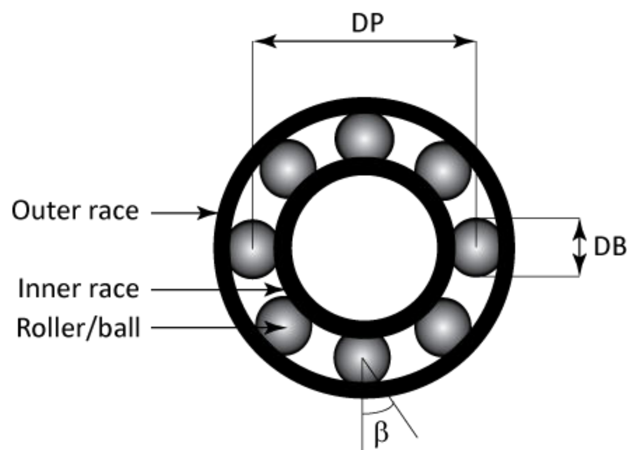
Generate frequency bands around the characteristic fault frequencies of ball or roller bearings for spectral feature extraction

Syntax

```
FB = bearingFaultBands(FR,NB,DB,DP,beta)
FB = bearingFaultBands( ___ ,Name,Value)
[FB,info] = bearingFaultBands( ___ )

bearingFaultBands( ___ )
```

Description

`FB = bearingFaultBands(FR,NB,DB,DP,beta)` generates characteristic fault frequency bands `FB` of a roller or ball bearing using its physical parameters. `FR` is the rotational speed of the shaft or inner race, `NB` is the number of balls or rollers, `DB` is the ball or roller diameter, `DP` is the pitch diameter, and `beta` is the contact angle in degrees. The values in `FB` have the same implicit units as `FR`.



`FB = bearingFaultBands( ___ ,Name,Value)` allows you to specify additional parameters using one or more name-value pair arguments.

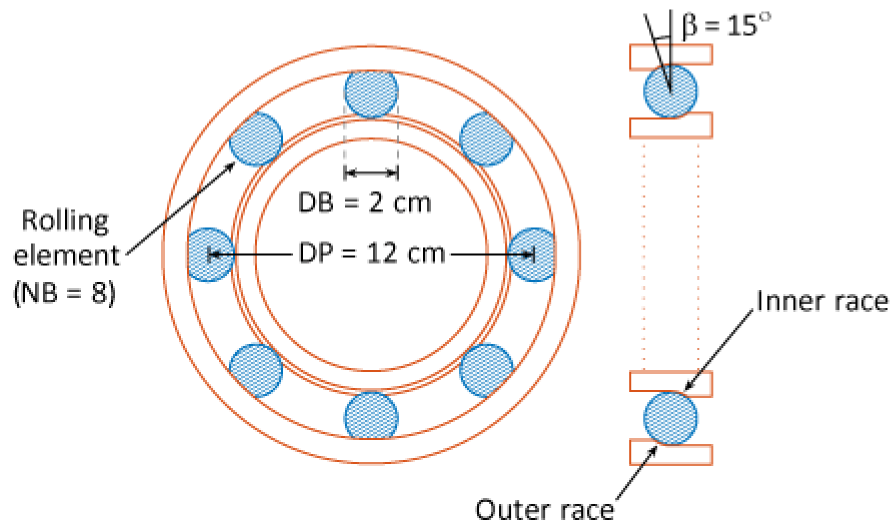
`[FB,info] = bearingFaultBands( ___ )` also returns the structure `info` containing information about the generated fault frequency bands `FB`.

`bearingFaultBands( ___ )` with no output arguments plots a bar chart of the generated fault frequency bands `FB`.

Examples

Frequency Bands Using Bearing Specifications

For this example, consider a bearing with a pitch diameter of 12 cm with 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 Hz. The contact angle of the rolling element is 15 degrees.



With the above physical dimensions of the bearing, construct the frequency bands using `bearingFaultBands`.

```
FR = 25;
NB = 8;
DB = 2;
DP = 12;
beta = 15;
FB = bearingFaultBands(FR,NB,DB,DP,beta)
```

```
FB = 4×2

   82.6512   85.1512
  114.8488  117.3488
   71.8062   74.3062
    9.2377   11.7377
```

FB is returned as a 4x2 array with default frequency band width of 10 percent of FR which is 2.5 Hz. The first column in FB contains the values of $F - \frac{W}{2}$, while the second column contains all the values of $F + \frac{W}{2}$ for each characteristic defect frequency.

Frequency Bands for Roller Bearing

For this example, consider a micro roller bearing with 11 rollers where each roller is 7.5 mm. The pitch diameter is 34 mm and the contact angle is 0 degrees. Assuming a shaft speed of 1800 rpm,

construct frequency bands for the roller bearing. Specify 'Domain' as 'frequency' to obtain the frequency bands FB in the same units as FR.

```
FR = 1800;
NB = 11;
DB = 7.5;
DP = 34;
beta = 0;
[FB1,info1] = bearingFaultBands(FR,NB,DB,DP,beta,'Domain','frequency')
```

```
FB1 = 4×2
10⁴ ×

    0.7626    0.7806
    1.1994    1.2174
    0.3791    0.3971
    0.0611    0.0791


info1 = struct with fields:
        Centers: [7.7162e+03 1.2084e+04 3.8815e+03 701.4706]
         Labels: ["1Fo"    "1Fi"    "1Fb"    "1Fc"]
    FaultGroups: [1 2 3 4]
```

Now, include the sidebands for the inner race and rolling element defect frequencies using the 'Sidebands' name-value pair.

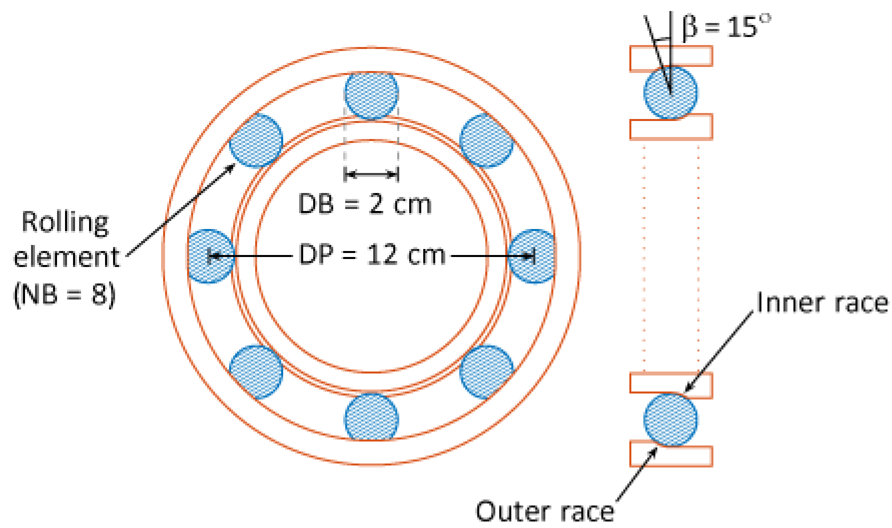
```
[FB2,info2] = bearingFaultBands(FR,NB,DB,DP,beta,'Domain','order','Sidebands',0:1)
```

```
FB2 = 8×2

    4.2368    4.3368
    5.6632    5.7632
    6.6632    6.7632
    7.6632    7.7632
    1.7167    1.8167
    2.1064    2.2064
    2.4961    2.5961
    0.3397    0.4397


info2 = struct with fields:
        Centers: [4.2868 5.7132 6.7132 7.7132 1.7667 2.1564 2.5461 0.3897]
         Labels: ["1Fo"    "1Fi-1Fr"    "1Fi"    "1Fi+1Fr"    ...    ]
    FaultGroups: [1 2 2 2 3 3 3 4]
```

You can use the generated fault bands FB to extract spectral metrics using the faultBandMetrics command.

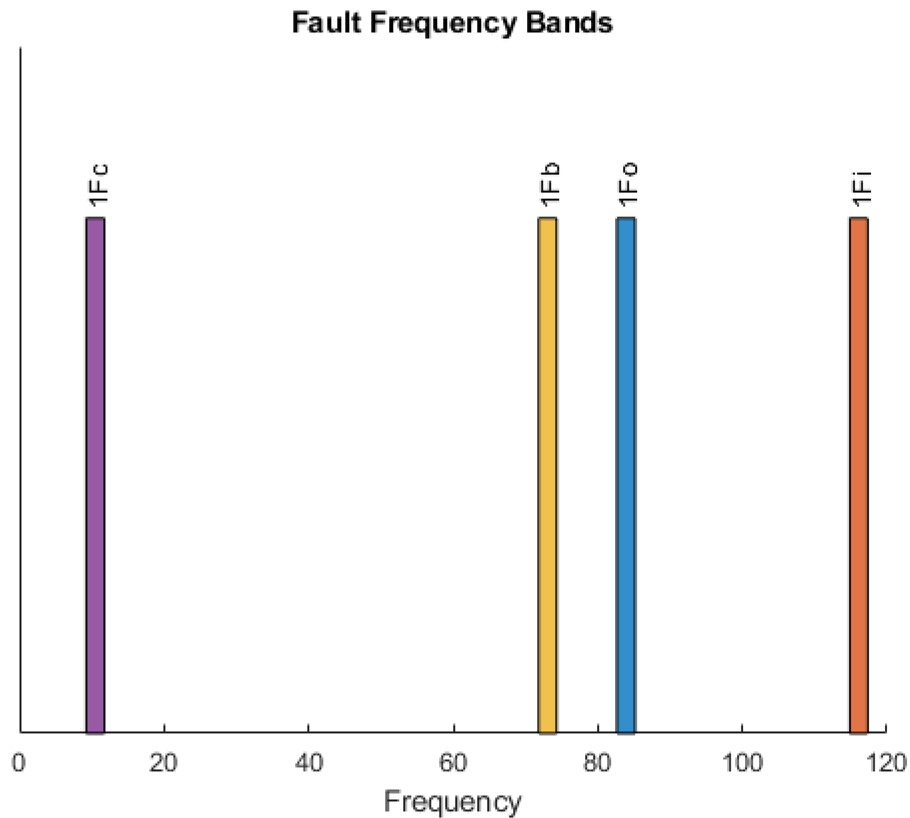
Visualize Frequency Bands Around Characteristic Bearing Frequencies

For this example, consider a damaged bearing with a pitch diameter of 12 cm with 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 Hz. The contact angle of the rolling element is 15 degrees.



With the above physical dimensions of the bearing, visualize the fault frequency bands using `bearingFaultBands`.

```
FR = 25;
NB = 8;
DB = 2;
DP = 12;
beta = 15;
bearingFaultBands(FR,NB,DB,DP,beta)
```



From the plot, observe the following bearing specific vibration frequencies:

- Cage defect frequency, `Fc` at 10.5 Hz.
- Ball defect frequency, `Fb` at 73 Hz.
- Outer race defect frequency, `Fo` at 83.9 Hz.
- Inner race defect frequency, `Fi` at 116.1 Hz.

Frequency Bands and Spectral Metrics of Ball Bearing

For this example, consider a ball bearing with a pitch diameter of 12 cm with 10 rolling elements. Each rolling element has a diameter of 0.5 cm. The outer race remains stationary as the inner race is driven at 25 Hz. The contact angle of the ball is 0 degrees. The dataset `bearingData.mat` contains power spectral density (PSD) and its respective frequency data for the bearing vibration signal in a table.

First, construct the bearing frequency bands including the first 3 sidebands using the physical characteristics of the ball bearing.

```
FR = 25;
NB = 10;
DB = 0.5;
DP = 12;
```

```
beta = 0;
FB = bearingFaultBands(FR,NB,DB,DP,beta,'Sidebands',1:3)
```

```
FB = 14×2

  118.5417  121.0417
   53.9583   56.4583
   78.9583   81.4583
  103.9583  106.4583
  153.9583  156.4583
  178.9583  181.4583
  203.9583  206.4583
  262.2917  264.7917
  274.2708  276.7708
  286.2500  288.7500
      ⋮
```

FB is a 14x2 array which includes the primary frequencies and their sidebands.

Load the PSD data. bearingData.mat contains a table X where PSD is contained in the first column and the frequency grid is in the second column, as cell arrays respectively.

```
load('bearingData.mat','X')
X
```

```
X=1×2 table
          Var1                 Var2
    ________________    ________________

    {12001x1 double}    {12001x1 double}
```

Compute the spectral metrics using the PSD data in table X and the frequency bands in FB.

```
spectralMetrics = faultBandMetrics(X,FB)
```

```
spectralMetrics=1×43 table
    PeakAmplitude1    PeakFrequency1    BandPower1    PeakAmplitude2    PeakFrequency2    BandPow
    ______________    ______________    __________    ______________    ______________    _______

         121               121            314.43          56.438            56.438          144.9
```

spectralMetrics is a 1x43 table with peak amplitude, peak frequency and band power calculated for each frequency range in FB. The last column in spectralMetrics is the total band power, computed across all 14 frequencies in FB.

Input Arguments

FR — Rotational speed of the shaft or inner race

positive scalar

Rotational speed of the shaft or inner race, specified as a positive scalar. FR is the fundamental frequency around which bearingFaultBands generates the fault frequency bands. Specify FR either in Hertz or revolutions per minute.

NB — Number of balls or rollers
positive integer

Number of balls or rollers in the bearing, specified as a positive integer.

DB — Diameter of the ball or roller
positive scalar

Diameter of the ball or roller, specified as a positive integer.

DP — Pitch diameter
positive scalar

Pitch diameter of the bearing, specified as a positive scalar. DP is the diameter of the circle that the center of the ball or roller travels during the bearing rotation.

beta — Contact angle
non-negative scalar

Contact angle in degrees between a plane perpendicular to the ball or roller axis and the line joining the two raceways, specified as a positive scalar.

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: `...,'Harmonics',[1,3,5]`

Harmonics — Harmonics of the fundamental frequency to be included
`1` (default) | vector of positive integers

Harmonics of the fundamental frequency to be included, specified as the comma-separated pair consisting of '`Harmonics`' and a vector of positive integers. The default value is 1. Specify '`Harmonics`' when you want to construct the frequency bands with more harmonics of the fundamental frequency.

Sidebands — Sidebands around the fundamental frequency and its harmonics to be included
`0` (default) | vector of nonnegative integers

Sidebands around the fundamental frequency and its harmonics to be included, specified as the comma-separated pair consisting of '`Sidebands`' and a vector of nonnegative integers. The default value is 0. Specify '`Sidebands`' when you want to construct the frequency bands with sidebands around the fundamental frequency and its harmonics.

Width — Width of the frequency bands centered at the nominal fault frequencies
`10` percent of the fundamental frequency (default) | positive scalar

Width of the frequency bands centered at the nominal fault frequencies, specified as the comma-separated pair consisting of '`Width`' and a positive scalar. The default value is 10 percent of the fundamental frequency. Avoid specifying '`Width`' with a large value so that the fault bands do not overlap.

Domain — Units of the fault band frequencies
'frequency' (default) | 'order'

Units of the fault band frequencies, specified as the comma-separated pair consisting of 'Domain' and either 'frequency' or 'order'. Select:

- 'frequency' if you want FB to be returned in the same units as FR.
- 'order' if you want FB to be returned as number of rotations relative to the inner race rotation, FR.

Folding — Logical value specifying whether negative nominal fault frequencies have to be folded about the frequency origin
false (default) | true

Logical value specifying whether negative nominal fault frequencies have to be folded about the frequency origin, specified as the comma-separated pair consisting of 'Folding' and either true or false. If you set 'Folding' to true, then faultBands folds the negative nominal fault frequencies about the frequency origin by taking their absolute values such that the folded fault bands always fall in the positive frequency intervals. The folded fault bands are computed as $\left[\max\left(0,\ |F| - \frac{W}{2}\right),\ |F| + \frac{W}{2}\right]$, where W is the 'Width' name-value pair and F is one of the nominal fault frequencies.

Output Arguments

FB — Fault frequency bands
Nx2 array

Fault frequency bands, returned as an Nx2 array, where N is the number of fault frequencies. FB is returned in the same units as FR, in either hertz or orders depending on the value of 'Domain'. Use the generated fault frequency bands to extract spectral metrics using faultBandMetrics. The generated fault bands, $\left[F - \frac{W}{2},\ F + \frac{W}{2}\right]$, are centered at:

- Outer race defect frequency, Fo and its harmonics
- Inner race defect frequency, Fi, its harmonics and sidebands at FR
- Rolling element (ball) defect frequency, Fbits harmonics and sidebands at Fc
- Cage (train) defect frequency, Fc and its harmonics

The value W is the width of the frequency bands, which you can specify using the 'Width' name-value pair. For more information on bearing frequencies, see “Algorithms” on page 1-16.

info — Information about the fault frequency bands
structure

Information about the fault frequency bands in FB, returned as a structure with the following fields:

- Centers — Center fault frequencies
- Labels — Labels describing each frequency
- FaultGroups — Fault group numbers identifying related fault frequencies

Algorithms

`bearingFaultBands` computes the different characteristic bearing frequencies as follows:

- Outer race defect frequency, $F_o = \frac{NB}{2}FR\left(1 - \frac{DB}{DP}\cos(\beta)\right)$
- Inner race defect frequency, $F_i = \frac{NB}{2}FR\left(1 + \frac{DB}{DP}\cos(\beta)\right)$
- Rolling element (ball) defect frequency, $F_b = \frac{DP}{2DB}FR\left(1 - \left[\frac{DB}{DP}\cos(\beta)\right]^2\right)$
- Cage (train) defect frequency,$F_c = \frac{FR}{2}\left(1 - \frac{DB}{DP}\cos(\beta)\right)$

References

[1] Chandravanshi, M & Poddar, Surojit. "Ball Bearing Fault Detection Using Vibration Parameters." *International Journal of Engineering Research & Technology*. 2. 2013.

[2] Singh, Sukhjeet & Kumar, Amit & Kumar, Navin. "Motor Current Signature Analysis for Bearing Fault Detection in Mechanical Systems." *Procedia Materials Science*. 6. 171–177. 10.1016/j.mspro.2014.07.021. 2014.

[3] Roque, Antonio & Silva, Tiago & Calado, João & Dias, J. "An approach to fault diagnosis of rolling bearings." *WSEAS Transactions on Systems and Control*. 4. 2009.

See Also

`faultBandMetrics` | `faultBands` | `gearMeshFaultBands`

Introduced in R2019b

bhattacharyyaDistance

One-dimensional Bhattacharyya distance between two independent data groups to measure class separability

Syntax

```
Z = bhattacharyyaDistance(X,I)
```

Description

`bhattacharyyaDistance` is a function used in code generated by **Diagnostic Feature Designer**.

`Z = bhattacharyyaDistance(X,I)` calculates the one-dimensional Bhattacharyya distances between two independent subsets of data set `X` that are grouped according to the logical labels in `I`. The Bhattacharyya distance provides a metric for ranking features according to their ability to separate two classes of data, such as data from healthy and faulty machines. The distance calculation assumes that the data in `X` follows a Gaussian distribution.

Code that is generated by **Diagnostic Feature Designer** uses `bhattacharyyaDistance` when ranking features with this method.

Input Arguments

X — Data samples to group

vector | matrix

Data set containing data samples that can be logically classified into two groups, specified as a vector when you have a single set of samples, such as values for one feature, and a matrix when you have multiple sets of samples.

- When `X` contains a single set of *n* features, such as a set of multiple features extracted from a single data source, `X` is a 1-by-*n* vector.
- When `X` contains *m* sets of *n* features, `X` is an *m*-by-*n* matrix. Each row in `X` represents one data source and must correspond to a single logical class.

`X` must contain at least two rows that correspond to the logical class in `I` of `0` and two rows that correspond to the label `1` to calculate legitimate Bhattacharyya distance values.

For example, suppose that you have a set of five features for each of 20 gearboxes and you are computing the Bhattacharyya distances to assess these features. `X` is a 20-by-5 matrix. Each row represents a gearbox that is either healthy or faulty, as indicated by the associated logical class label of `0` or `1`. At least two gearboxes must be healthy and at least two gearboxes must be faulty. The Bhattacharyya distance indicates how well each feature separates the data for the healthy gearboxes from the data for the faulty gearboxes.

I — Logical classification labels

vector

Logical classification labels that assign the rows in `X` to one of two logical classes, specified as a vector of length *m*, where *m* is the number of rows in `X`.

For example, suppose once more that X is a 20-by-5 matrix corresponding to 20 gearboxes. The first 9 gearboxes are healthy. The remaining 11 gearboxes are faulty. Define the healthy state as 0 and the faulty state as 1. Then I has a length of 20. The first 9 labels in I are equal to 0 and the remaining 11 labels are equal to 1.

Output Arguments

Z — Bhattacharyya distances

scalar | vector

Bhattacharyya distances between labeled groups, returned as a scalar or a vector of length *n*.

- If X is a vector, then Z is a scalar.
- If X is a matrix, then bhattacharyyaDistance calculates the distance separately for each feature. Z is then a vector of length *n*, where *n* is the number of columns in Z.

bhattacharyyaDistance treats NaN entries in X as missing values and ignores them.

References

[1] Theodoridis, Sergios, and Konstantinos Koutroumbas. *Pattern Recognition*, 177–179. 2nd ed. Amsterdam; Boston: Academic Press, 2003.

See Also

correlationWeightedScore | **Diagnostic Feature Designer**

Topics

"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a

compare

Compare test data to historical data ensemble for similarity models

Syntax

```
compare(mdl,data)
compare( ___ ,Name,Value)
```

Description

`compare(mdl,data)` plots the test component degradation data in `data` superimposed on the most similar data sets from the historical ensemble stored in the fitted similarity model `mdl`. The *K* most similar data sets from the ensemble are plotted, where *K* is the `NumNearestNeighbors` property of `mdl`.

`compare( ___ ,Name,Value)` specifies plotting options using one or more name-value pair arguments.

Examples

Compare Test Data to Historical Data

Load training data.

```
load('pairwiseTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component.

Create and train a pairwise similarity model.

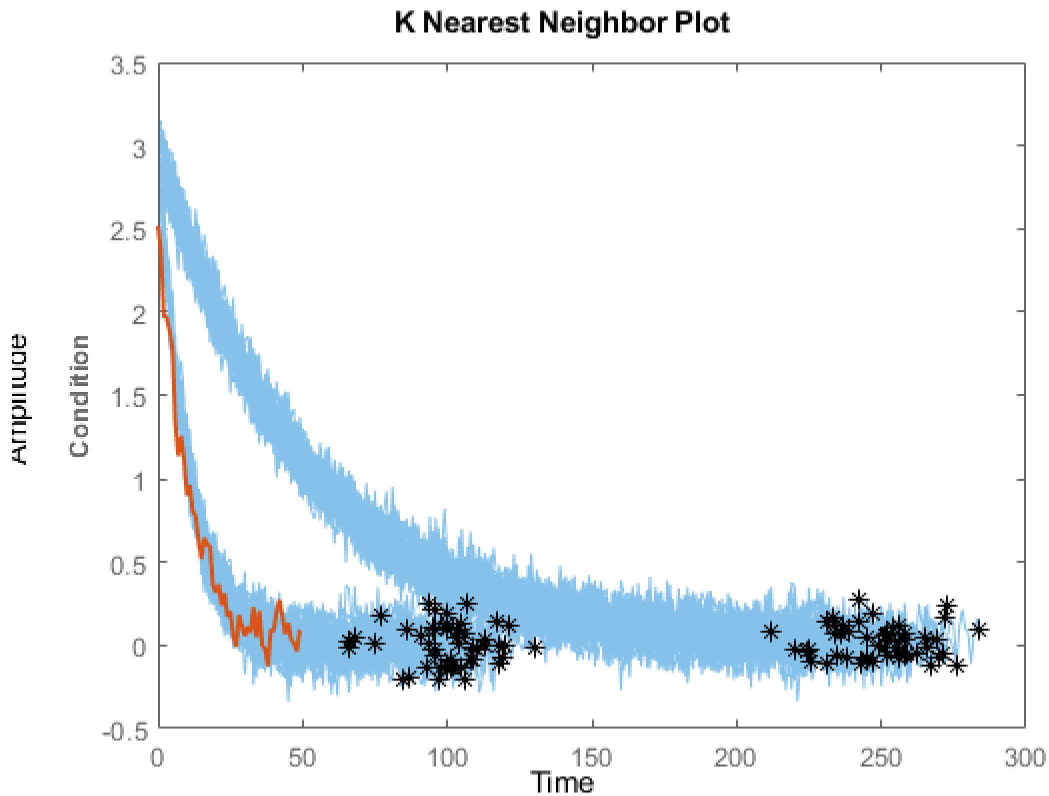
```
mdl = pairwiseSimilarityModel;
fit(mdl,pairwiseTrainTables,"Time","Condition")
```

Load testing data.

```
load('pairwiseTestData.mat')
```

Compare the degradation profile of the test data to the profiles of the historical data ensemble.

```
compare(mdl,pairwiseTestData)
```



Compare Test Data to Most Similar Historical Data

Load training data.

```
load('pairwiseTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component.

Create and train a pairwise similarity model.

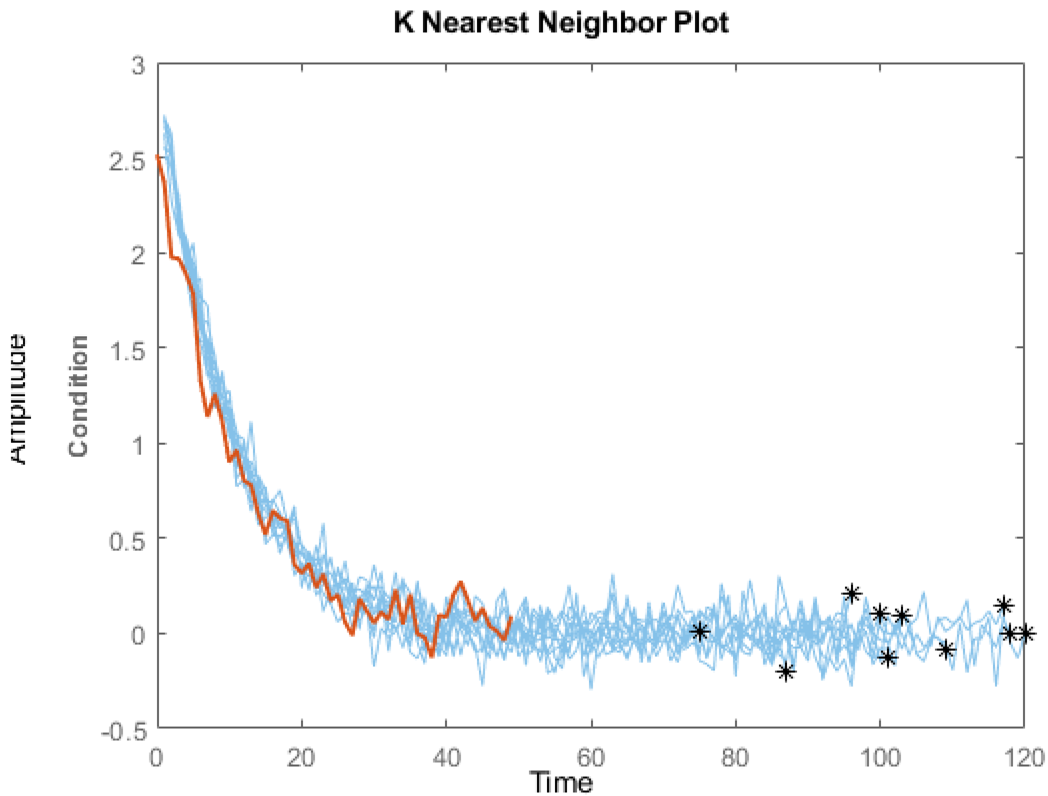
```
mdl = pairwiseSimilarityModel;
fit(mdl,pairwiseTrainTables,"Time","Condition")
```

Load testing data.

```
load('pairwiseTestData.mat')
```

Compare the degradation profile of the test data to the profiles of the `10` most similar members of the historical data ensemble.

```
compare(mdl,pairwiseTestData,'NumNearestNeighbors',10)
```



Input Arguments

mdl — Similarity RUL model
hashSimilarityModel object | pairwiseSimilarityModel object | residualSimilarityModel object

Similarity RUL model, specified as a hashSimilarityModel object, a pairwiseSimilarityModel object, or a residualSimilarityModel object. The model must be fitted using fit before calling compare.

data — Degradation feature measurements
array | table | timetable

Degradation feature profiles for estimating the RUL of similarity models, measured over the life span of a component up to the current life time, specified as one of the following:

- (N+1)-by-M numeric array, where N is the number of features and M is the number of feature measurements. In each row, the first column contains the usage time and the remaining columns contain the corresponding degradation feature measurements. The order of the features must match the order specified in the DataVariables property of mdl.
- table or timetable object — The table must contain variables with names that match the strings in the DataVariables and LifeTimeVariable properties of mdl.

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: `'NumNearestNeighbors','10'` plots ten similar data sets

`NumNearestNeighbors` — Number of nearest neighbors
`Inf` | finite positive integer

Number of nearest neighbors, specified as the comma-separated pair `'NumNearestNeighbors'` and either `Inf` or a finite positive integer. Use this option to select the number of most similar data sets to plot by overriding the `NumNearestNeighbors` property. If `NumNearestNeighbors` is `Inf`, then `compare` plots the degradation data for all the ensemble data sets.

`Threshold` — Degradation data bounds
two-column array

Degradation data bounds, specified as the comma-separated pair `'Threshold'` and a two-column array with *N* rows, where *N* is the number of data variables used by `mdl`. The first column of `Threshold` contains the lower bounds for the variables, and the second column contains the upper bounds. The bounds are rendered as yellow-colored patches.

To disable the bounds for a given variable, specify the lower and upper bounds as `-Inf` and `Inf`, respectively.

Tips

- To select which signals to plot, right-click on the plot area, and select **Data Variable Selector**. In the Data Variable Selector dialog box, the **Select Variables** box shows the variables that are available for plotting.

See Also

Functions
`hashSimilarityModel` | `pairwiseSimilarityModel` | `residualSimilarityModel`

Introduced in R2018a

correlationDimension

Measure of chaotic signal complexity

Syntax

```
corDim = correlationDimension(X)
corDim = correlationDimension(X,lag)
corDim = correlationDimension(X,[],dim)
corDim = correlationDimension(X,lag,dim)
[corDim,rRange,corInt] = correlationDimension( ___ )
___ = correlationDimension( ___ ,Name,Value)
```

```
correlationDimension( ___ )
```

Description

`corDim = correlationDimension(X)` estimates the correlation dimension of the uniformly sampled time-domain signal `X`. Correlation dimension is the measure of dimensionality of the space occupied by a set of random points. `corDim` is estimated as the slope of the correlation integral versus the range of radius of similarity. Use `correlationDimension` as a characteristic measure to distinguish between deterministic chaos and random noise, to detect potential faults.[1]

`corDim = correlationDimension(X,lag)` estimates the correlation dimension of the uniformly sampled time-domain signal `X` for the time delay `lag`.

`corDim = correlationDimension(X,[],dim)` estimates the correlation dimension of the uniformly sampled time-domain signal `X` for the embedding dimension `dim`.

`corDim = correlationDimension(X,lag,dim)` estimates the correlation dimension of the uniformly sampled time-domain signal `X` for the time delay `lag` and embedding dimension `dim`.

`[corDim,rRange,corInt] = correlationDimension( ___ )` additionally estimates the range of radius of similarity and correlation integral of the uniformly sampled time-domain signal `X`. Correlation integral is the mean probability that the states of a system are close at two different time intervals, which reflects self-similarity.

`___ = correlationDimension( ___ ,Name,Value)` estimates the correlation dimension with additional options specified by one or more `Name,Value` pair arguments.

`correlationDimension( ___ )` with no output arguments creates a correlation integral versus neighborhood radius plot.

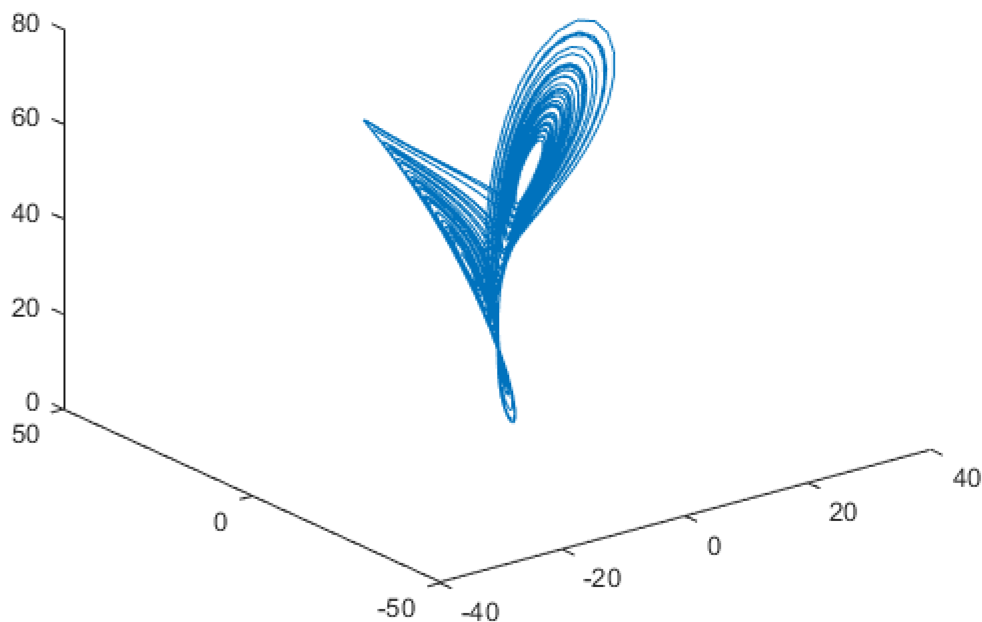
Examples

Visualize and Estimate Correlation Dimension of Data

In this example, consider a Lorenz Attractor describing a unique set of chaotic solutions.

Load the data set and visualize the Lorenz Attractor in 3D.

```
load('lorenzAttractorExampleData.mat','data');
plot3(data(:,1),data(:,2),data(:,3));
```



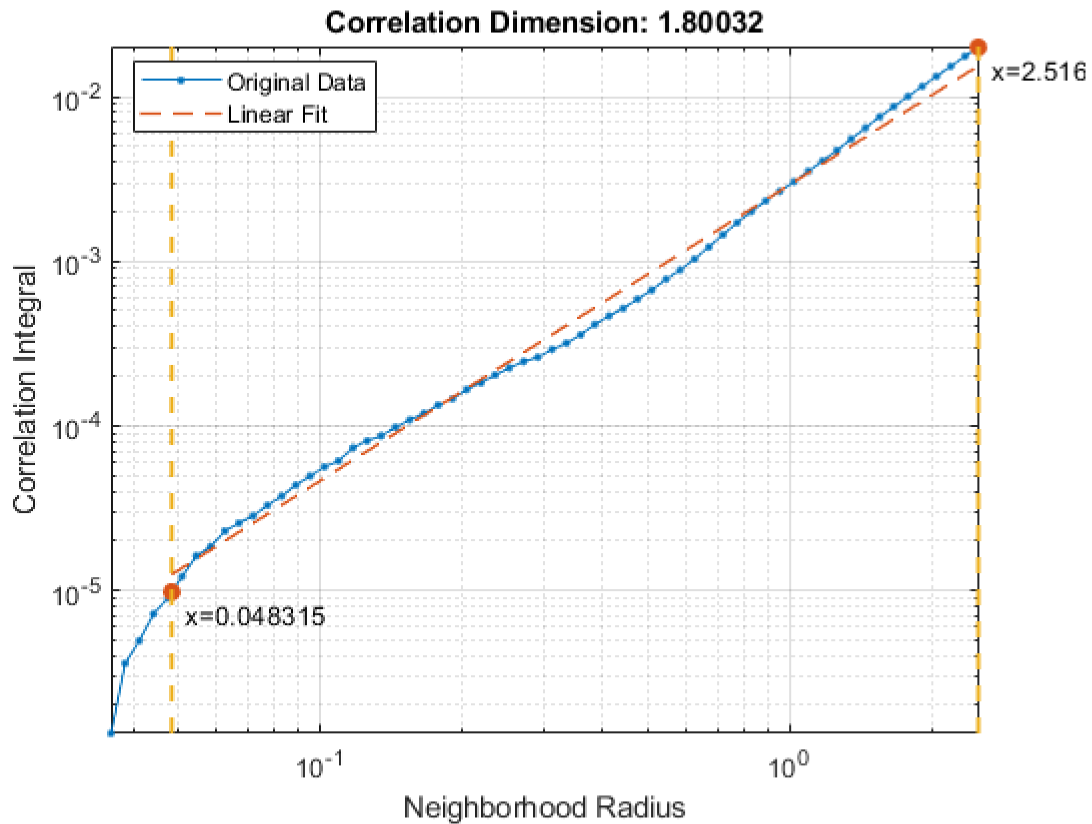
For this example, use only x-direction data of the Lorenz Attractor. Since `lag` is unknown, estimate the delay using `phaseSpaceReconstruction`. Set '`Dimension`' to 3 since the Lorenz Attractor is a three-dimensional system. The `dim` and `lag` parameters are required to create the correlation integral versus the neighborhood radius plot.

```
xdata = data(:,1);
dim = 3;
[~,lag] = phaseSpaceReconstruction(xdata,[],dim)
```

```
lag = 10
```

Create the correlation integral versus neighborhood radius plot for the Lorenz Attractor, using the `lag` value obtained in the previous step. Set an appropriate value for '`NumPoints`' to determine a good resolution for the neighborhood radius.

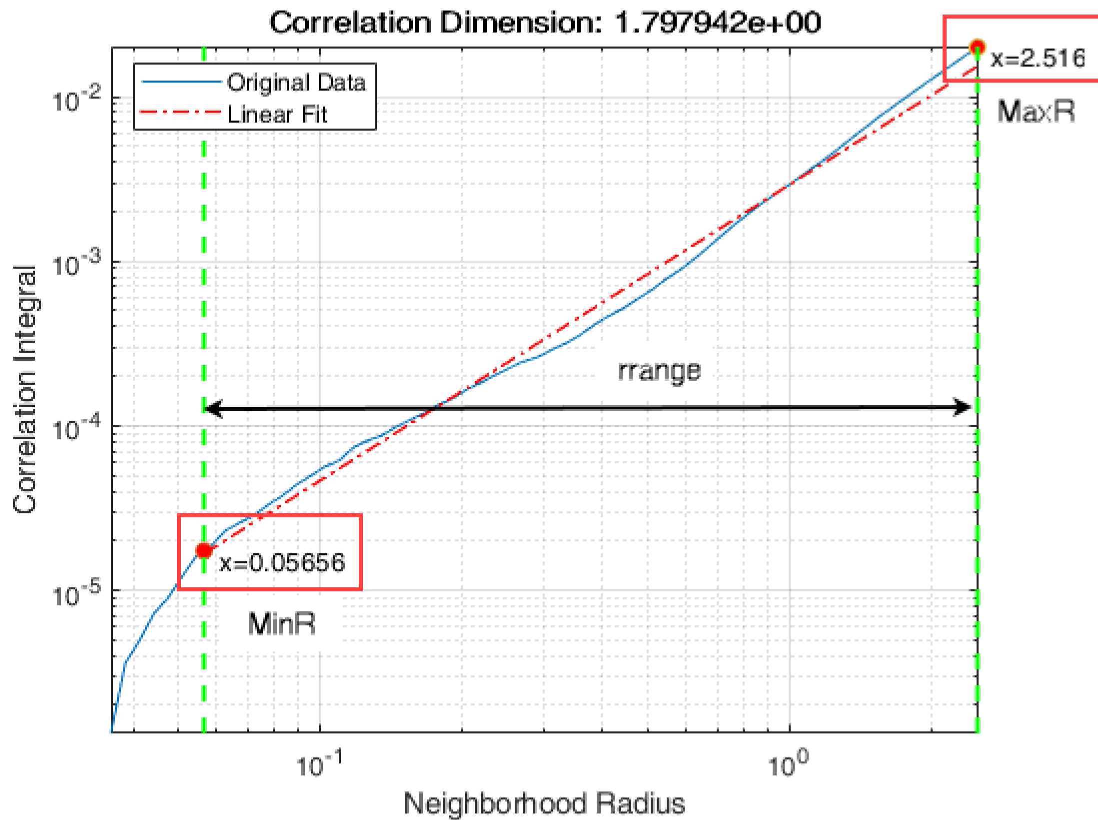
```
Np = 100;
correlationDimension(xdata,lag,dim,'NumPoints',Np);
```



The first dashed, vertical green line (on the left) indicates the value of `MinRadius`, while the second vertical green line (on the right), represents `MaxRadius`. The dashed red line indicates the linear fit line for the correlation integral versus neighborhood radius data, within the computed range of radius.

To compute correlation dimension, you first need to determine the `MinRadius` and `MaxRadius` values needed for accurate estimation.

In the plot, drag the two dashed, vertical green lines to 'best fit' the linear fit line to the original data line to obtain the range of radius.



Note the new values of `MinRadius` and `MaxRadius` after dragging the two vertical lines for an appropriate fit.

Find the correlation dimension of the Lorenz Attractor, using the new `MinRadius` and `MaxRadius` values obtained in the previous step.

```
MinR = 0.05656;
MaxR = 2.516;
corDim = correlationDimension(xdata,[],dim,'MinRadius',MinR,'MaxRadius',MaxR,'NumPoints',Np)
```

```
corDim = 1.7490
```

The value of correlation dimension is directly proportional to the level of chaos in the system, that is, a higher value of `corDim` represents a high level of chaotic complexity in the system.

Input Arguments

X — Uniformly sampled time-domain signal

vector | array | timetable

Uniformly sampled time-domain signal, specified as a vector, array, or timetable. If `X` has multiple columns, `correlationDimension` computes the correlation dimension by treating `X` as a multivariate signal.

If `X` is specified as a row vector, `correlationDimension` treats it as a univariate signal.

dim — Embedding dimension
scalar | vector

Embedding dimension, specified as a scalar or vector. dim is equivalent to the 'Dimension' name-value pair.

lag — Time delay
scalar | vector

Time delay, specified as a scalar or vector. lag is equivalent to the 'Lag' name-value pair.

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: ...,'Dimension',3

Dimension — Embedding dimension
2 (default) | scalar | vector

Embedding dimension, specified as the comma-separated pair consisting of 'Dimension' and a scalar or vector. When Dimension is scalar, every column in X is reconstructed using Dimension. When Dimension is a vector having same length as the number of columns in X, the reconstruction dimension for column i is Dimension(i).

Specify Dimension based on the dimension of your system, that is, the number of states. For more information on embedding dimension, see phaseSpaceReconstruction.

Lag — Delay in phase space reconstruction
1 (default) | scalar | vector

Delay in phase space reconstruction, specified as the comma-separated pair consisting of 'Lag' and either a scalar or vector. When Lag is scalar, every column in X is reconstructed using Lag. When Lag is a vector having same length as the number of columns in X, the reconstruction delay for column i is Lag(i).

If the delay is too small, random noise is introduced in the data. In contrast, if the lag is too large, the reconstructed dynamics does not represent the true dynamics of the time series. For more information on estimating optimal delay, see phaseSpaceReconstruction.

MinRadius — Minimum radius of similarity
MaxRadius/1000 (default) | scalar

Minimum radius of similarity, specified as the comma-separated pair consisting of 'MinRadius' and a scalar. Find the optimal value of MinRadius by adjusting the linear fit of the correlation dimension plot.

MaxRadius — Maximum radius of similarity
0.2*sqrt(trace(cov(X))) (default) | scalar

Maximum radius of similarity, specified as the comma-separated pair consisting of 'MaxRadius' and a scalar. Find the optimal value of MaxRadius by adjusting the linear fit of the correlation dimension plot.

`NumPoints` — Number of points for computation
10 (default) | positive scalar integer

Number of points for computation, specified as the comma-separated pair consisting of '`NumPoints`' and a positive scalar integer. `NumPoints` is the number of points between `MinRadius` and `MaxRadius`. Choose an appropriate value for `NumPoints` based on the resolution required for `rRange`.

`NumPoints` only accepts values greater than 1, and the default value is 10.

Output Arguments

`corDim` — Correlation Dimension
scalar

Correlation dimension, returned as a scalar. `corDim` is a measure of chaotic signal complexity in multidimensional phase space and is the slope of the correlation integral versus the range of radius of similarity. `corDim` is used in fault detection as a characteristic measure to distinguish between deterministic chaos and random noise.

`rRange` — Range of radius of similarity
array

Radius of similarity, returned as an array. `rRange` is the difference between `MaxRadius` and `MinRadius` split into an equal number of points defined by `NumPoints`.

`corInt` — Correlation integral
array

Correlation integral, returned as an array. `corInt` is the mean probability that the states at two different times are close, which reflects self-similarity. `NumPoints` defines the length of `corInt` array.

Algorithms

Correlation dimension is computed in the following way,

1. The `correlationDimension` function first generates a delayed reconstruction $Y_{1:N}$ with embedding dimension m, and lag τ.
2. The software then calculates the number of with-in range points, at point i, given by,

   $$N_i(R) = \sum_{i = 1, i \neq k}^{N} 1(\|Y_i - Y_k\| < R)$$

   where **1** is the indicator function, and R is the radius of similarity, given by, R = exp(linspace(log(r_{min}), log(r_{max}), N)). Here, r_{min} is `MinRadius`, r_{max} is `MaxRadius`, and N is `NumPoints`.
3. The correlation dimension `corDim` is the slope of *C(R)* vs. R where, the correlation integral *C(R)* is defined as,

   $$C(R) = \frac{2}{N(N-1)} \sum_{i = 1}^{N} N_i(R)$$

References

[1] Caesarendra, Wahyu & Kosasih, P & Tieu, Kiet & Moodie, Craig. "An application of nonlinear feature extraction-A case study for low speed slewing bearing condition monitoring and prognosis." *IEEE/ASME International Conference on Advanced Intelligent Mechatronics: Mechatronics for Human Wellbeing, AIM 2013*.1713-1718. 10.1109/AIM.2013.6584344.

[2] Theiler, James. "Efficient algorithm for estimating the correlation dimension from a set of discrete points". American Physical Society. *Physical Review A* 1987/11/1. Volume 36. Issue 9. Pages 44-56.

See Also

`approximateEntropy` | `phaseSpaceReconstruction` | `lyapunovExponent`

Introduced in R2018a

correlationWeightedScore

Adjust feature ranking scores using correlation factor

Syntax

```
[score,idx] = correlationWeightedScore(X,Z,alpha)
```

Description

`correlationWeightedScore` is a function used in code generated by **Diagnostic Feature Designer**.

`[score,idx] = correlationWeightedScore(X,Z,alpha)` weights the original ranking scores in `Z` for the features in `X` according to the correlation between features. Correlation weighting reduces feature redundancy. `correlationWeightedScore` lowers the score of a feature that has a high correlation to a higher ranking feature. The correlation importance factor `alpha` determines how much impact the correlation level has on the feature ranking score.

Code that is generated by **Diagnostic Feature Designer** uses `correlationWeightedScore` when ranking features if the specified correlation importance factor is greater than zero.

Input Arguments

X — Feature set

vector | matrix

Feature set, specified as an *m*-by-1 vector or an *m*-by-*n* matrix, where *m* is the number of data samples and *n* is the number of features. For an ensemble-based feature set, *m* is the number of members in the ensemble.

Z — Original ranking scores

vector

Original ranking scores, computed by a ranking method such as `bhattacharyyaDistance`, and specified as a vector of length *n*, where *n* is the number of features. The length of `Z` must be the same as the width of `X`.

alpha — Correlation importance factor

scalar in the range [0 1]

Correlation importance factor that determines how much impact correlation has on scores.

- If `alpha` is set to `0`, correlation has no impact on the ranking score.
- If `alpha` is set to `1`, correlation has the maximum possible impact on ranking score.

Output Arguments

score — Adjusted ranking scores

vector

Adjusted ranking scores, returned as a vector that is the same size as Z.

idx — Updated ranking order

integer vector

Updated ranking order after the scores are adjusted by correlation weighting, returned as a vector of integers.

References

[1] Theodoridis, Sergios, and Konstantinos Koutroumbas. *Pattern Recognition*, 182–183. 2nd ed. Amsterdam; Boston: Academic Press, 2003.

See Also

bhattacharyyaDistance | relativeEntropy | **Diagnostic Feature Designer**

Topics

"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a

Diagnostic Feature Designer

Interactively extract, visualize, and rank features from measured or simulated data for machine diagnostics and prognostics

Description

The **Diagnostic Feature Designer** app allows you to accomplish the feature design portion of the predictive maintenance workflow using a multifunction graphical interface. You design and compare features interactively and then determine which features are best at discriminating between data from nominal systems and from faulty systems. The most effective features ultimately become your condition indicators for fault diagnosis and prognostics.

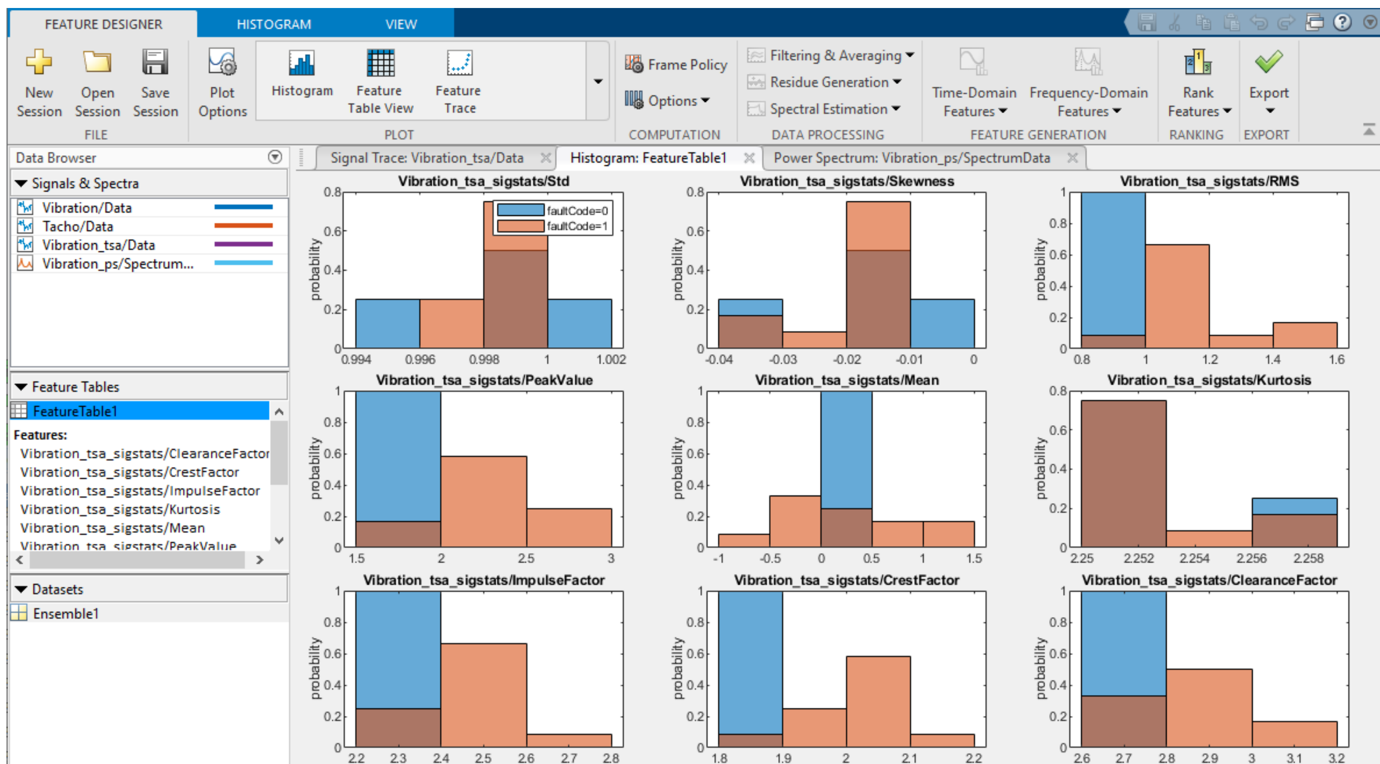
Using this app, you can:

- Import measured or simulated data from individual files, an ensemble file, or an ensemble datastore that references files external to the app.
- Interactively visualize data to plot the ensemble variables you import or that you compute within the app. Group data by condition label in plots so that you can clearly see whether member data comes from nominal or faulty systems.
- Derive new variables such as time-synchronous averaged signals or order spectra. The app executes processing on all ensemble members with one command.
- Generate features from your variables and visualize their effectiveness using histograms. Features include signal statistics, nonlinear metrics, rotating machinery metrics, and spectral metrics.
- Rank your features to determine which ones are best at discriminating behavioral differences in the data.

  - Use supervised ranking with labeled features to determine which features are most likely to discriminate between nominal and faulty behavior.
  - Use unsupervised ranking when your data has no condition variables or labels to determine which features exhibit the best clustering with other features and are most likely to indicate different fault or operating conditions.
  - Use prognostic ranking with features extracted from run-to-failure data to determine which features are most likely to indicate remaining useful life (RUL).
- Export your most effective features directly to **Classification Learner** for more insight into feature effectiveness and for algorithm training.
- Generate code for the features you choose so that you can reproduce, customize, and automate the feature computations in a MATLAB® function.

To get started with the app, you must have data from your systems available in your MATLAB workspace. For information about organizing your data for import into the app, see “Organize System Data for Diagnostic Feature Designer”.

For more information about condition indicators for predictive maintenance, see “Condition Indicators for Monitoring, Fault Detection, and Prediction”.



Open the Diagnostic Feature Designer App

- MATLAB toolstrip: On the **Apps** tab, under **Control System Design and Analysis**, click the app icon.
- MATLAB command prompt: Enter `diagnosticFeatureDesigner`.

Examples

- “Identify Condition Indicators for Predictive Maintenance Algorithm Design”
- “Import and Visualize Ensemble Data in Diagnostic Feature Designer”
- “Process Data and Explore Features in Diagnostic Feature Designer”
- “Rank and Export Features in Diagnostic Feature Designer”
- “Prepare Matrix Data for Diagnostic Feature Designer”
- “Isolate a Shaft Fault Using Diagnostic Feature Designer”
- “Perform Prognostic Feature Ranking for a Degrading System Using Diagnostic Feature Designer”
- “Generate a MATLAB Function in Diagnostic Feature Designer”
- “Apply Generated MATLAB Function to Expanded Data Set”

Parameters

Feature Designer Tab

`New Session` — Import datasets from the MATLAB workspace into app
button

Initiate a new app session by importing source data into the app from your MATLAB workspace. You can import data from tables, timetables, cell arrays, or matrices. You can import data from a single source that combines the data of multiple ensemble members or import individual ensemble members from separate sources. You can also import an ensemble datastore that contains information that allows the app to interact with external data files. Your files can contain actual or simulated time-domain measurement data, spectral models or tables, variable names, condition and operational variables, and features you generated previously. **Diagnostic Feature Designer** combines all your member data into a single ensemble data set. In this data set, each variable is a collective signal or model that contains all the individual member values.

For more information about importing data, see “Import Data into Diagnostic Feature Designer”.

For more information about terms related to data ensembles, see “More About” on page 1-45.

For more information about organizing your data for import into the app, see “Organize System Data for Diagnostic Feature Designer”.

`Plot Options` — Specify default plotting options for all plots that you generate during your app session
button

Specify default plotting options for all plots that you generate during your app session. You can set these options before you generate your first plot, or at any time during your session. New settings apply only to plots that you generate after setting the options and not to plots that you generated earlier. You can temporarily override the **Plot Options** settings for individual plots without changing your specified defaults for subsequent plots. When you click **Plot Options**, you open a dialog that allows you to set options in the following panes.

- **General** — These options apply to all signal and spectrum plots.
  - **Group by** — Group data by a condition variable label. The app uses color to distinguish label groups. For example, if your condition variable is `faultCode` with labels `healthy` and `degraded`, the app uses one color for member data with the `healthy` label and another color for member data with the `degraded` label
  - **Number of curves** — Specify the number of members to plot. Set this option when you have a large number of ensemble members and you want to plot only a subset of the members. Using this option reduces the plotting time and allows you to assess individual member behavior more easily.
- **Spectrum** — These options apply only to spectral plots.
  - **Number of peaks to mark** — Specify the number of peaks to mark. Set this option to limit the number of spectral peaks that are marked to highlight only the most significant peaks.
- **Ensemble Summary** — These options apply only to the ensemble summary plot, which is a special plot that displays the mean and standard deviation of the ensemble as a whole.
  - **Number of standard deviations** — Specify the number of standard deviations that the ensemble summary plot displays.

- **Show min and max boundaries** — Specify whether to display the boundaries of the actual minimum and maximum values of the ensemble.

`Plot` — Generate plot of ensemble variable or feature

plot type

Generate a plot of an ensemble variable or feature table. To generate a plot, first select a variable or feature table from the Data Browser. The plot gallery shows icons for the compatible plot types. The following table describes the plot types for each type of selection.

Input Type	Plot Type	Description	Customize With
Signal	Signal Trace	Ensemble signal data plotted by time or another independent variable that does not represent frequency.	Signal Trace Tab on page 1-0
	Ensemble Summary	Mean, standard deviation, and min/max boundary for the ensemble as a whole.	Ensemble Summary Tab
Spectrum	Power Spectrum	Ensemble signal power plotted by frequency.	Power Spectrum Tab on page 1-0
	Order Spectrum	Ensemble signal power plotted by order, which is the ratio of a specific frequency to the main rotational frequency.	Order Spectrum Tab on page 1-0
Feature Table	Histogram	Feature effectiveness, as visualized by a bar chart with color coding for the condition label. Effective features separate conditions cleanly.	“Histogram Tab” on page 1-0
	Feature Table View	Table containing feature values and their condition labels for each ensemble member.	N/A
	Feature Trace	Feature values for each member. This plot is especially useful for prognostic features (for RUL) computed from frame-based data.	N/A

`Frame Policy` — Specify data handling mode and frame size and rate

button

Specify a frame policy when you want to perform data processing on sequential segments of a signal rather than the full signal at once. A frame policy consists of a frame size and a frame rate. The frame

size is the interval over which the frame data is collected. The frame rate is the time interval between frame start times.

For more information on frame-based processing, see “Data Handling Mode and Frame Policy”.

`Options` — Select independent variable and specify parallel computing
button

Specify options when you want to modify one or both the following settings.

- **Independent Variable** — Independent variable (IV) to use. When you import your data, you can specify more than one independent variable for a signal. For example, if your signal is time based, you might want to also have an independent variable for sample index. After you complete the import, you can change the independent variable that the app uses for a specific plot or computation. When you select **Options** > **Independent Variable**, the app displays a list of the available independent variables. Your selection changes the IV of all the applicable signals or spectra. For more information on specifying an alternate IV, see Specify Sample Index as Alternate IV in “Import Data into Diagnostic Feature Designer”.
- **Use Parallel Computing** — Process ensemble members in parallel. Using parallel computing can significantly decrease processing time for large ensembles. This option is available only when you have Parallel Computing Toolbox™ installed and licensed.

`Data Processing` — Select data processing options by category
filtering & averaging | residue generation | spectral estimation

Select options for processing your data into new signals. Use these new signals as inputs to other processing options or as inputs to feature generation. Most processing options operate on each ensemble member. You can also perform ensemble-level processing to view how the ensemble behaves as a whole. Each option selection opens a new tab for your specifications. Selection of an option also opens a general **Data Processing** tab if that tab is not already open. The **Data Processing** tab provides information about the input signal.

To specify a signal to process, select a variable from the data browser prior to selecting the data processing option. To change the signal after opening the option tab, close the option tab and select a new signal either in the data browser or from the **Signal** list in the **Data Processing** tab.

For more information about the processing options and the parameters that you can set for each option, see:

- **Filtering & Averaging**
  - “Time-Synchronous Signal Averaging”
  - “Ensemble Statistics”
  - “Remove Harmonics”
  - “Filter TSA Signals”
  - “Interpolation”
- **Residue Generation**
  - “Subtract Reference”
- **Spectral Estimation**
  - “Power Spectrum”

- “Order Spectrum”

Time-Domain Features — Compute features in time domain
Signal Features | Rotating Machinery Features | Nonlinear Features

Compute features in the time domain. **Signal Features** apply to any signals. **Rotating Machinery Features** are specialized metrics that apply to gearing. **Nonlinear Features** provide metrics that characterize chaotic behavior in vibration signals. Each selection opens a dialog box for your source signal and feature specifications.

To specify a signal source for your features, select a signal variable from the data browser prior to selecting the time-domain features option. To change the signal after opening the option tab, close the option tab and select a new signal either in the data browser or from the **Signal** menu in the **Time-Domain Features** tab.

For more information about time-domain features options and the parameters that you can set for each option, see:

- “Signal Features”
- “Rotating Machinery Features”
- “Nonlinear Features”

Frequency-Domain Features — Compute features in frequency domain
Spectral Features | Bearing Faults Features | Gear Mesh Faults Features | Custom Faults Features

Compute features in the frequency domain. **Spectral Features** are general metrics that apply to any spectrum, such as the peak amplitude across the full specified frequency range. **Bearing Faults Features**, **Gear Mesh Faults Features**, and **Custom Faults Features** are specialized metrics for rotating machinery that focus on spectral behavior within specific fault bands that bound characteristic frequencies, where faults can occur, of the components of the system.

For more information about the frequency-domain features, see

- “Spectral Features”
- “Spectral Features Based on Fault Bands”

Rank Features — Rank features
feature table

Open the feature ranking tab to perform supervised, unsupervised, or prognostic ranking for the feature table that you select. For more information, see “Feature Ranking Tab” on page 1-0.

Export — Export features and data or generate MATLAB code
Export Features to the MATLAB workspace | Export Features to the Classification Learner | Export a Data Set to the MATLAB Workspace | Generate Function for Features | Generate Function for...

Export features, or your entire data set, to use them or share them outside of the app. Generate code to reproduce your feature computations in a MATLAB function.

- For feature export, both options open a list of features.
  - If you have not yet ranked your features, the app sorts this list by name, and marks all features by export by default. You can refine the selection if you want to export only specific features.

- If you have ranked your features, the app sorts this list by your **Sort by** specification in the "Feature Ranking Tab" on page 1-0 . Use **Select top features** to export only the most highly ranked features, based on the number of features that you specify. You can change the sorting order to alphabetical by selecting `Name` in the **Features sorted by** list. With either sorting order, you can individually select or clear features to export.

  When you export to the MATLAB workspace, you can use command-line techniques with the features. When you export to **Classification Learner**, you open a **Classification Learner** session that uses your selected features as input.

- For code generation, the first option, `Generate Function for Features`, lets you generate MATLAB code with a simple set of specifications for feature table, ranking algorithm, and number of features. Use this option when you want to generate code for features based solely on ranking, or when you want to generate code for all your features.

  The second code generation option, `Generate Function for...`, allows you to customize your selection of items to include in the function. For example, you can filter your selection based on criteria such as input or output text. You can include signals and spectra that are not used in the features you select. Selecting `Generate Function for...` opens a selectable list of all the signals, features, and ranking tables that you have generated. `Generate Function for...` also opens the **Code Generation** tab, which provides filtering capability for the list. Use a filter to view only the items that meet the filter criterion. You can use different filters to select the outputs you want. To review all your selections regardless of filter, click **Sort by Selection**. This option lists all the available outputs with items that you selected on top. For more information, see Code Generation Tab.

  If you have specified frame-based data (see Computation Options), clicking `Generate Function for...` first opens a list with selections for the frame specifications that you have used. The items in your generated code must either all operate on the full signal or all use the same frame specification.

  For more information on how to generate code in the app, see "Automatic Feature Extraction Using Generated MATLAB Code" and "Generate a MATLAB Function in Diagnostic Feature Designer".

For more information about the **Export** options, see:

- "Export Features to MATLAB Workspace"
- "Export Features to Classification Learner"
- "Export a Dataset to the MATLAB Workspace"
- "Generate Function for Features"

Signal Trace, Power Spectrum, and Order Spectrum Plot Tabs

`Panner` — Control plot scale and x-axis range

on (default) | off

Use the **Panner** to focus on data segments in the x-axis range that you specify and to change the plot scale. The **Panner** provides a strip plot beneath the main plot. To focus on a section of the main plot, move the handles. To change the scale of the plot, select one of the options in **Scale**.

`Group By, Number of Curves, Number of Peaks to Mark` — Override default settings for display of condition grouping, number of members to plot, and, for spectral plots, number of peaks to mark
on | off | positive scalar

Use **Group By** and **Number of Curves** when you want to display ensemble label grouping and limit the number of members that you plot. Use **Peaks to Mark** to specify how many spectral peaks to highlight. The default values correspond to the settings in **Plot Options**. When you change these settings in the plot tab, you change them only for the current plot. For more information on these parameters, see “Plot Options” on page 1-0 .

`Normalize Y Axis` — Normalize variables to the same scale
off (default) | on

Use **Normalize Y Axis** when you are plotting multiple variables and want to view the variables on the same [-1, 1] scale. The relative signal or spectrum amplitudes within a variable do not change.

`Show Signal (Spectrum) Information` — Display highlighted variable member name and condition label
on (default) | off

In a signal or spectrum plot, you highlight an individual member by positioning your cursor on the member trace. Select **Show Signal Information** or **Show Spectrum Information** to display both the variable member that you highlight and the condition label for that member in the lower right corner.

`Merge Axes` — Plot multiple variables together in separate plots or in one plot
on (default) | off

Specify how to plot multiple variables together.

- Select to create a single plot that overlays all traces and uses a single y-axis scale.
- Clear to create separate plots displayed vertically, each with a unique y-axis scaling.

`Data Cursors` — Display x and y values of points and distances between two points
off (default) | on

Select **Data Cursors** to display values of key points in your signal. Data Cursors are horizontal and vertical bars that you position over a point of interest, such as a peak value. The cursors display the x and y positions. To display the distance between the cursors, select **Show Signal Information**. To lock the bars so that they move together, select one of the **Lock Spacing** options.

Histogram Tab

`Select Features` — Choose features to plot
button (default)

Click **Select Features** to open a selectable list of features to plot. Use **Select Features**, for example, when you have generated many features but you want to focus on a subset in a single plot panel.

`Group By` — Select condition variable for grouping data
condition variable name

Select the condition variable to base feature histograms on. The feature histograms use color to visualize the separation of data groups with different labels for that condition variable.

Example: `faultCode`

`Bin Settings` — Specify histogram resolution
`auto` (default) | numeric | binning method name

Specify histogram resolution using **Bin Width**, **Bin Method**, **Number of Bins**, and **Bin Limits**. The bin settings apply to all the histograms for the feature table.

The bin settings are not independent. The app histogram algorithm uses an order of precedence to determine what to use:

- The **Binning Method** is the default driver for the bin width.
- A **Bin Width** specification overrides the Binning Method.
- The bin width and the independent **Bin Limits** drive the number of bins. A **Number of Bins** specification has an effect only when the value of **Group By** is `none`.

  For more information on interpreting and customizing histograms, see “Generate and Customize Feature Histograms”.

Feature Ranking Tab

`Supervised Ranking` — Select supervised classification ranking method to apply
`T-Test` | `One-way ANOVA` | `ROC` | ...

Select a supervised classification ranking method to assess how effectively each feature separates data with different condition labels. If you have already ranked your features, you can rank again with a different method and display the resulting rankings together. Each method uses a different statistical approach.

The menu differentiates between two-class and multiclass ranking methods.

- Two-Class Methods — Use when your condition variable (CV) has only two labels, such as `healthy` and `faulty`. The default value for two-class methods is `T-Test`.
- Multiclass methods — Use when your condition variable has two or more labels, such as `healthy`, `faultCode1`, and `faultCode2`. The default value for multiclass methods is `One-way ANOVA`

The default ranking method for two-class condition variables, `T-Test`, is the simplest method, as it only considers whether the means of the two labeled groups are equal or not. `T-Test` is primarily useful for identifying ineffective features to discard.

The table lays out the influence of specific criteria on ranking-method selection.

Criterion	Ranking Method
Condition Variable Type	• Multiclass CV — One-way ANOVA, Kruskal-Wallis • Two-Class CV — T-test, Entropy, Bhattacharyya, Wilcoxon, ROC
Feature Scoring Criterion	• Mean Difference — T-Test (primarily for discarding ineffective features) • Distribution Overlap — All others

Criterion	Ranking Method
Distribution Shape	• Gaussian — T-Test, Entropy, Bhattacharyya, One-way ANOVA • Non-Gaussian — ROC, Wilcoxon, Kruskal-Wallis
Desired Method Basis	• Hypothesis Test — T-Test, One-way ANOVA, Wilcoxon, Kruskal-Wallis • Distance Measurement — Entropy, Bhattacharyya, ROC

Selecting a method activates a new tab with a name that matches the ranking method. For more information on this method-activated tab, see “Ranking Method Tabs” on page 1-0.

For more information on the ranking methods, see:

- One-way ANOVA — `anova1`
- Bhattacharyya — `bhattacharyyaDistance`
- Kruskal-Wallis — `kruskalwallis`
- Entropy — `relativeEntropy`
- ROC — `perfcurve`
- Wilcoxon — `ranksum`
- T-Test — `ttest2`

`Unsupervised Ranking` — Select unsupervised classification ranking algorithm to apply
`Laplacian Score` | `Variance`

Select an unsupervised classification ranking method to assess how effectively each feature performs when you do not have labeled data. The app provides two unsupervised ranking options:

- Laplacian Score — Scores reflect how well features cluster with other features to form distinct groupings.
- Variance — Scores reflect feature variance. Features with low variances tend to add less useful information to a model.

Selecting a method activates a new tab with a name that matches the ranking method. For more information on this tab, see “Ranking Method Tabs” on page 1-0.

For more information on the unsupervised ranking scoring, see:

- Laplacian — `fsulaplacian`
- Variance— `var`

Unsupervised ranking is available in **Diagnostic Feature Designer**, but not in **Classification Learner**. If you plan to export your features to **Classification Learner** to train a model, you must use ensemble data that includes labels.

`Prognostic Ranking` — Select prognostic ranking algorithm to apply
`Monotonicity` | `Trendability` | `Prognosability`

Select a prognostic ranking method to assess how effectively each feature tracks the degradation of your ensemble members when you have run-to-failure data. The top-ranking features are best at predicting the remaining useful life (RUL).

The app provides three prognostic ranking methods, all of which score features on a scale ranging from 0 through 1. One method, `Monotonicity`, is always available. The other two methods, `Trendability` and `Prognosability`, are available only when you are using frame-based data. The smaller data segments in frame-based data allow the tracking of small changes that are induced by degradation.

- Monotonicity characterizes the trend of a feature as the system evolves toward failure. As a system gets progressively closer to failure, a suitable condition indicator has a monotonic positive or negative trend. For more information, see `monotonicity`.
- Trendability provides a measure of similarity between the trajectories of a feature measured in multiple run-to-failure experiments. Trendability of a candidate condition indicator is defined as the smallest absolute correlation between measurements. For more information, see `trendability`.
- Prognosability is a measure of the variability of a feature at failure relative to the range between its initial and final values. A more prognosable feature has less variation at failure relative to the range between its initial and final values. For more information, see `prognosability`.

Selecting a method activates a new tab with a name that matches the ranking method. For more information on this method-activated tab, see **Ranking Method Tab**.

For an example of using prognostic ranking in the app, see "Perform Prognostic Feature Ranking for a Degrading System Using Diagnostic Feature Designer".

`Rank By` — Specify condition variable for classification ranking algorithm to use
condition variable name

Select the condition variable that provides the labels for the classification ranking algorithm to use.

`Sort By` — Specify ranking method to sort results by when displaying results from multiple methods
ranking method

Specify the ranking method to sort by when comparing the results of different ranking methods. When you use a single ranking method, the app displays the results in order of importance, as indicated by the ranking score for that method. When comparing the results for multiple methods, change **Sort By** to change the method that drives the sorting order.

`Delete Scores` — Delete ranking scores from display
no selection (default) | ranking method

Specify this parameter to eliminate ranking scores for a specific method. Use this parameter, for example, when comparing the results of multiple rankings, and you want to simplify the display by eliminating rankings that do not influence your feature selection.

`Export` — Export features from app or generate MATLAB code to reproduce your feature computations in command-line function
`Export features to the MATLAB workspace` | `Export features to the Classification Learner` | `Generate Function for Features` | `Generate Function for...`

Export features to use them or share them outside of the app. Both options open a ranking-sorted selectable list to choose from. When you export to the MATLAB workspace, you can use command-

line techniques with the features. When you export to the **Classification Learner**, you open a **Classification Learner** session that uses your selected features as input.

- "Export Features to MATLAB Workspace"
- "Export Features to Classification Learner"

If you want to export your entire data set from the app, use **Export** from the **Feature Designer** tab.

You can also generate code that reproduces the computations for the variables and features you select. For more information, see the code generation options description in the "Export" on page 1-0 section in the **Feature Designer** tab. When you generate code using `Generate Function for Features` from the **Feature Ranking** tab, **Ranking Method** defaults to the method you specify in **Sort By**.

Ranking Method Tabs

`Correlation Importance` — Reduce ranking of redundant features

0 (default) | scalar in the range [0,1]

The correlation importance setting allows you to screen out features that convey similar information to higher ranked features. This screening provides a more diverse feature set in the upper ranks.

The criterion for the screening is the set of cross-correlation coefficients a feature has with higher ranked features. High cross-correlation between two features implies that both features are separating condition groups similarly and provide redundant information. With the default value of 0, the app does not incorporate feature redundancy into ranking scores. As you increase the correlation importance value, the app increases the influence of feature cross-correlation on the feature ranking score. This increasing influence progressively lowers the score of redundant features.

`Normalization Scheme` — Apply normalization across members for supervised ranking and unsupervised Laplacian ranking

`minmax` (default) | `none` | `meanvar` | `softmax`

The normalization scheme performs independent normalization across the members for every feature. Normalization allows more direct comparisons among features. The app displays the defining equation for the scheme you select directly beneath your selection.

This option is available only for supervised ranking and unsupervised Laplacian ranking methods.

`Laplacian Parameters` — Specify parameters for Laplacian ranking

parameter values

The Laplacian parameters define key values for calculating the Laplacian score <to be updated>.

- **Number of neighbors** — <to be updated>
- **Distance Metric** — <to be updated>
- **Kernel Scale** — <to be updated>

This option is available only for the unsupervised Laplacian ranking method.

`Apply` — Apply parameter settings to new ranking computation

button

Click **Apply** to calculate a ranking with the specified parameters. The **Feature Ranking** tab in the plotting area displays the results both graphically and tabularly. This display also includes the results for the default ranking algorithm, and for any other ranking methods you calculated previously.

Once you calculate a ranking, the app disables **Apply** until you change a parameter. You can calculate a ranking within a tab multiple times. Each time you modify the parameters and calculate a ranking, the new results overwrite the previous results in the plotting-area tab.

`Close` — Close tab and return control to feature ranking tab
button

Once you have completed your ranking within the ranking method tab, close that tab to return control to the **Feature Ranking** tab. The **Feature Ranking** is disabled while any ranking method tab is activated.

Code Generation Tab

`Frame Policy` — Feature table name, frame size, and frame rate
feature table name, `Full Signal`, `None` (default) | feature table name, frame size, frame rate

This property is read-only.

The frame policy information reflects the choice you make when you select **Export > Generate Function for...** in the **Feature Designer** tab.

`Filter or Sort` — Define criteria to refine your code generation choices
empty (default) | string | input, method, or analysis type

Set criteria to refine your options when selecting items for your generated function. All criteria allow you to overwrite selectable options with a string. String matching is case insensitive. Your filters apply to all output items, including signals, features, and ranking tables. Criteria include:

- **Output** — String appearing in the output name, which is the name of the variable, feature, or ranking table to select for the generated function
- **Input** — Input signal from which the output variable or feature was computed or feature table from which the ranking table was computed
- **Method** — Computation that produced the output item, such as `TSA` or `Kurtosis`
- **Analysis Type** — Data processing, feature processing, or feature ranking

To reset a single filter, delete the contents and click anywhere in the app. To reset all filters at once, click **Reset Filters**.

`Sort by Selection` — Display all selected items
button

Display all selected items together. Use **Sort Selected** especially when you have used multiple filter combinations to assemble your codegen selections. All your selections appear together.

`Use Parallel Computing` — Generate code that uses parallel computing
off | on

Specify whether to use parallel computing in the generated code. The default value is the value that is specified in “Options” on page 1-0 . You can specify parallel computing even if you performed your interactive processing without using parallel computing. This approach helps your code to be

more scalable if you plan to run the generated code on a larger ensemble than the ensemble you used to develop the features. You can also turn parallel computing off if you used it when you developed the features.

To take advantage of parallel processing in generated code, the user must have Parallel Computing Toolbox installed and licensed. However, the code will still run in serial mode on systems that do not have the toolbox.

Code — Execute function generation
Generate Function button

Click the **Generate Function** button when you have completed configuring your selections. The app opens a function that contains computations used for all the output items you selected.

For more information about generating code in the app, see “Automatic Feature Extraction Using Generated MATLAB Code”.

Programmatic Use

`diagnosticFeatureDesigner` opens the **Diagnostic Feature Designer** app.

`diagnosticFeatureDesigner(sessionFile)` opens the app and loads a previously saved session. `sessionFile` is the name of a session data file on the MATLAB path. The data includes all of the variables and features that you either imported into the app or computed within the app. The data also includes your app settings and the processing information necessary to generate code.

To save a session, in the **Diagnostic Feature Designer** app, on the **Feature Designer** tab, click **Save Session**.

More About

Data Ensemble

A data ensemble is a collection of datasets, created by measuring or simulating a system under varying conditions. An ensemble can be implemented using independent datasets such as matrices or tables, or in a single collective dataset such as an ensemble table.

For more information on data ensembles and variables, see “Data Ensembles for Condition Monitoring and Predictive Maintenance”.

Ensemble Member

Each dataset within an ensemble is a member. Members of an ensemble all contain the same variables. For example, if your ensemble contains data from a set of similar machines, the dataset corresponding to one of those machines is a member.

Ensemble Table

An ensemble table is an ensemble dataset formatted as a `table`. Each column of the table represents one variable. Each row of the table represents one ensemble member. For information on converting member matrices to an ensemble table, see “Prepare Matrix Data for Diagnostic Feature Designer”.

Ensemble Datastore Object

Large ensembles can be implemented using an ensemble datastore object. These objects contain a list of the member files and information for interacting with them. For more information on ensemble datastore objects, see “Data Ensembles for Condition Monitoring and Predictive Maintenance”.

Data Variable

Data variables make up the main content of the ensemble members, including measured data and derived data that you use for analysis and development of predictive maintenance algorithms. For example, you might represent accelerometer data as the data variable `Vibration`. Data variables can also include derived values, such as the mean value of a signal, or the frequency of the peak magnitude in a signal spectrum.

Independent Variable

Independent variables (IV) are the variables that identify or order the members in an ensemble, such as timestamps, number of operating hours, or machine identifiers. For example, `Time` is a common independent variable.

Condition Variable

Condition variables (CV) are variables that describe the fault condition or operating condition of the ensemble member. Condition variables can record the presence or absence of a fault state, or other operating conditions such as ambient temperature. Frequently condition variables have specific possible values described by labels. For example, a condition variable named `Health` might have two states described by labels `Healthy` and `Degraded`. Condition variables can also be derived values, such as a single scalar value that encodes multiple fault and operating conditions.

See Also

Topics
“Identify Condition Indicators for Predictive Maintenance Algorithm Design”
“Import and Visualize Ensemble Data in Diagnostic Feature Designer”
“Process Data and Explore Features in Diagnostic Feature Designer”
“Rank and Export Features in Diagnostic Feature Designer”
“Prepare Matrix Data for Diagnostic Feature Designer”
“Isolate a Shaft Fault Using Diagnostic Feature Designer”
“Perform Prognostic Feature Ranking for a Degrading System Using Diagnostic Feature Designer”
“Generate a MATLAB Function in Diagnostic Feature Designer”
“Apply Generated MATLAB Function to Expanded Data Set”
“Explore Ensemble Data and Compare Features Using Diagnostic Feature Designer”
“Organize System Data for Diagnostic Feature Designer”
“Interpret Feature Histograms in Diagnostic Feature Designer”
“Import Data into Diagnostic Feature Designer”
“Automatic Feature Extraction Using Generated MATLAB Code”
“Anatomy of App-Generated MATLAB Code”
“Data Ensembles for Condition Monitoring and Predictive Maintenance”

Introduced in R2019a

effectivefs

Effective sampling rate of a time vector

Syntax

```
[Fs,irregular] = effectivefs(T)
```

Description

`effectivefs` is a function used in code generated by **Diagnostic Feature Designer**.

`[Fs,irregular] = effectivefs(T)` checks the regularity of one-dimensional time array `T` and returns the best approximation `Fs` to the underlying sampling rate. Code that is generated by **Diagnostic Feature Designer** uses `effectivefs` when performing spectral processing and other computations.

Input Arguments

T — Time array

`datetime` array | `duration` array | `numeric` vector

Time array of sampling instants, expressed as a one-dimensional `datetime` array, a one-dimensional `duration` array, or a numeric vector.

Output Arguments

Fs — Effective sampling rate

numeric scalar

Effective sampling rate, returned as a numeric scalar.

- If `T` is a `duration` array, then `Fs` is in cycles per time unit of `T`.
- If `T` is a `datetime` array, then `effectiveFs` determines the best value for the time unit based on the data, and `Fs` is in cycles per time unit.
- If `T` is a numeric vector, then `Fs` is in cycles per second.

irregular — Irregularity indicator

logical

Irregularity indicator, returned as a logical.

- When `irregular` is `true`, the sampling instants in `T` are unevenly spaced. `Fs` represents the effective sampling rate of `T`.
- When `irregular` is `false`, the sampling instants in `T` are evenly spaced. `Fs` represents the true sampling rate of `T`.

See Also

`datetime` | `duration` | `time2num`

Topics
"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a

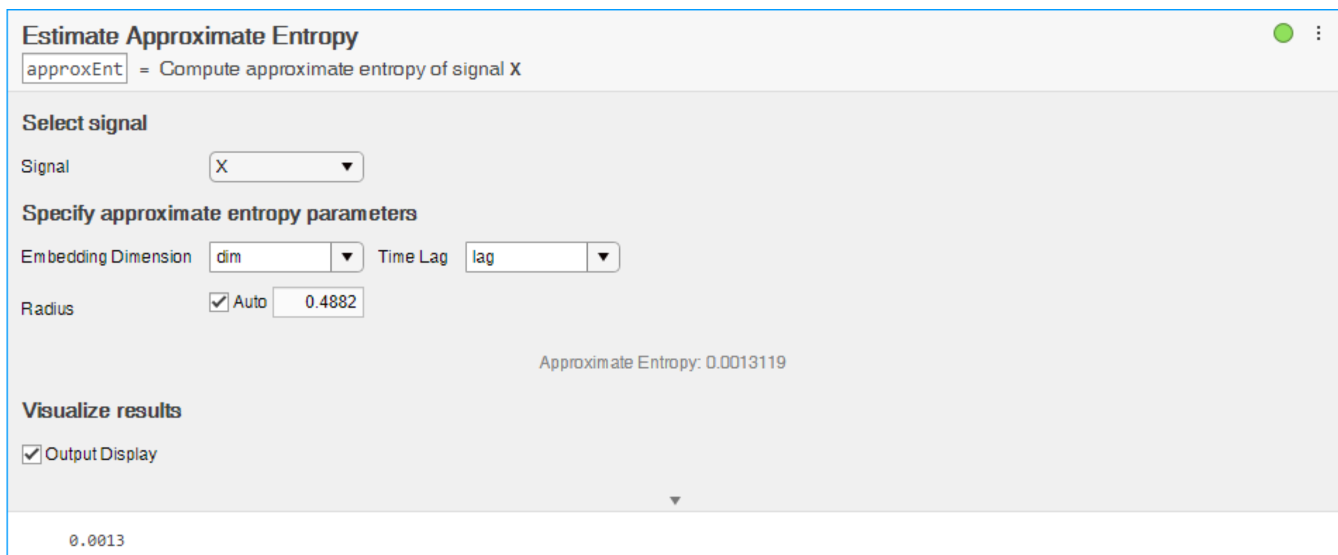
Estimate Approximate Entropy

Interactively estimate the approximate entropy of a uniformly sampled signal in the Live Editor

Description

The **Estimate Approximate Entropy** task lets you interactively estimate the approximate entropy of a uniformly sampled signal. The task automatically generates MATLAB code for your live script. For more information about Live Editor tasks generally, see "Add Interactive Tasks to a Live Script".

Approximate entropy is a regularity statistic that quantifies the unpredictability of fluctuations in a time series. A relatively higher value of approximate entropy reflects the likelihood that similar patterns of observations are not followed by additional similar observations.



Open the Task

To add the **Estimate Approximate Entropy** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task** > **Estimate Approximate Entropy**.
- In a code block in your script, type a relevant keyword, such as `approximate` or `approximate entropy`. Select `Estimate Approximate Entropy` from the suggested command completions.

Examples

Estimate Approximate Entropy in the Live Editor

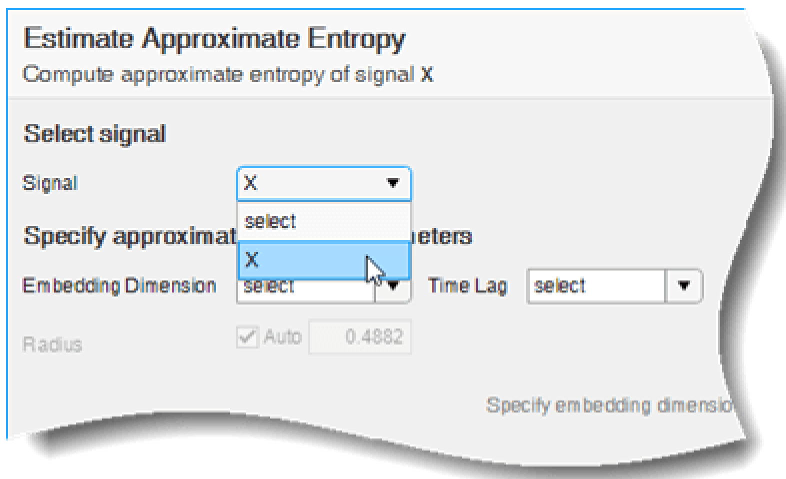
Use the **Estimate Approximate Entropy** task in the Live Editor to interactively estimate the approximate entropy of a uniformly sampled signal. Experiment with different values for lag,

embedding dimension and radius. The task automatically generates code reflecting your selections. Open this example to see a preconfigured script containing the **Estimate Approximate Entropy** task.

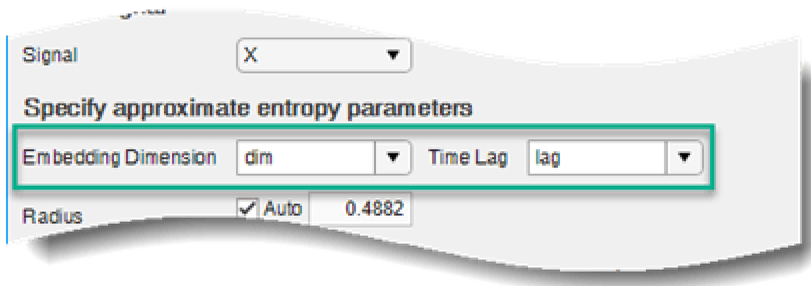
For this example, consider `'approxEntData.mat'` which contains uniformly sampled signal `X`, embedding dimension `dim`, and the time delay `lag`.

```
load('approxEntData.mat','X','dim','lag')
```

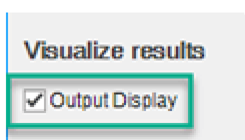
To approximate entropy of the signal `X`, open the **Estimate Approximate Entropy** task in the Live Editor. On the **Live Editor** tab, select **Task** > **Estimate Approximate Entropy**. In the task, select signal `X`.



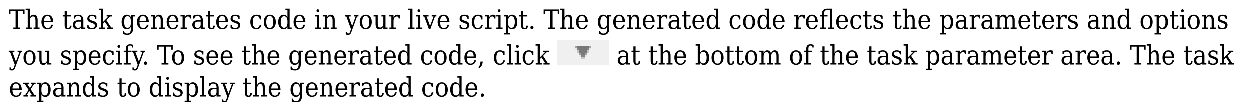
Select `dim` for the **Embedding Dimension** and `lag` for the **Time Lag** dropdown menu respectively. If you do not know the embedding dimension and the time lag for your signal, use the **Reconstruct Phase Space** task to compute the values.

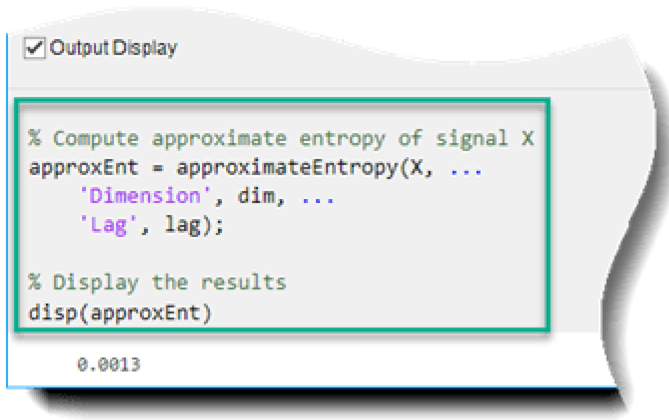


Evaluate whether the value of approximate entropy is affected drastically by changing the **Radius** field and observe the change in value in the Live Editor output. You can toggle displaying the output of the approximate entropy value in the Live Editor output using the **Output Display** option.

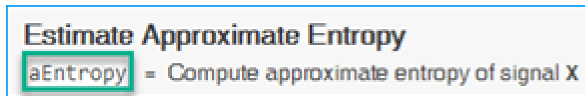


The task generates code in your live script. The generated code reflects the parameters and options you specify. To see the generated code, click ▼ at the bottom of the task parameter area. The task expands to display the generated code.





By default, the generated code uses `approxEnt` as the name of the output variable. To specify a different output variable name, enter a new name in the summary line at the top of the task. For instance, change the name to `aEntropy`.



The task updates the generated code to reflect the new variable name, and the new variable `aEntropy` appears in the MATLAB workspace.

Parameters

Select Signal

`Signal` — Uniformly sampled time-domain signal

array | timetable

Select a uniformly sampled time-domain signal in array or timetable format. If the signal has multiple columns, the **Estimate Approximate Entropy** task computes the approximate entropy by treating it as a multivariate signal. If the signal is a row vector, then the **Estimate Approximate Entropy** task treats it as a univariate signal.

Specify Approximate Entropy Parameters

`Embedding Dimension` — Number of dimensions of phase space vectors

scalar | vector

Specify the number of dimensions of phase space vectors as a scalar or vector from the MATLAB workspace. When you specify the embedding dimension as a scalar, then every column of the uniformly sampled signal is computed using the same embedding dimension value.

If you do not know the value of embedding dimension for your signal, then you can compute it using the **Reconstruct Phase Space** task.

`Time Lag` — Time lag between successive phase vectors
scalar | vector

Specify time lag between successive phase vectors as a scalar or vector from the MATLAB workspace. When you specify the time lag as a scalar, then the **Estimate Approximate Entropy** task uses the same time delay value to estimate the value of approximate entropy for all the columns of the uniformly sampled signal. If you specify the embedding dimension as a vector, then specify the time lag also as a vector of the same length.

If you do not know the value of time lag for your signal, then you can compute it using the **Reconstruct Phase Space** task.

`Radius` — Similarity criterion
`'Auto'` (default) | scalar

Specify similarity criterion as a scalar. The similarity criterion, also called radius of similarity, is a tuning parameter that is used to identify a meaningful range in which fluctuations in data are to be considered similar.

Visualize Results

`Output Display` — Toggle result display in the Live Editor output
on (default) | off

Toggle to display the value of approximate entropy in the Live Editor output.

See Also

Reconstruct Phase Space | `correlationDimension` | `phaseSpaceReconstruction` | `lyapunovExponent` | `approximateEntropy`

Topics

"Add Interactive Tasks to a Live Script"

Introduced in R2019b

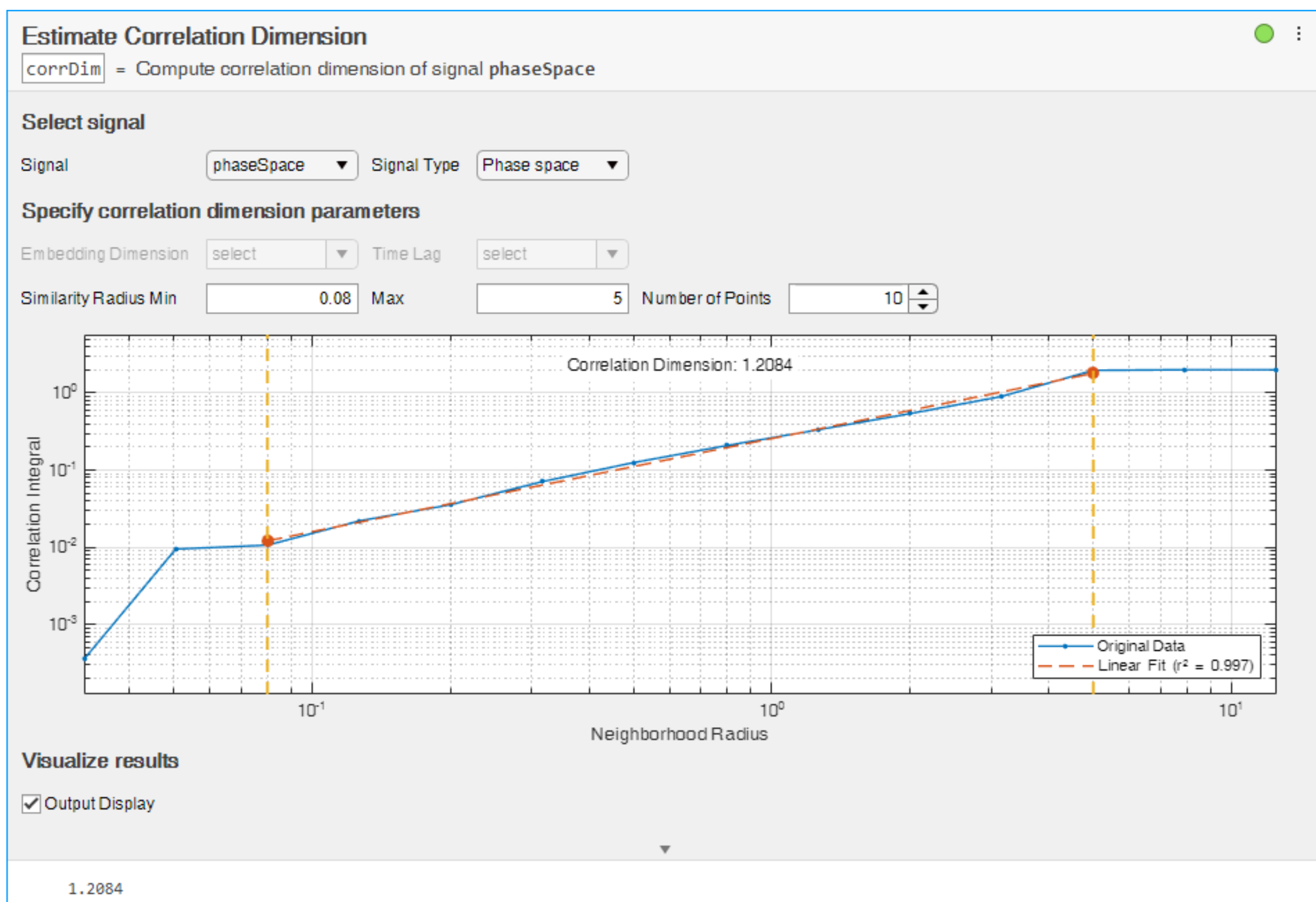
Estimate Correlation Dimension

Estimate the correlation dimension of a uniformly sampled signal in the Live Editor

Description

The **Estimate Correlation Dimension** task lets you interactively estimate the correlation dimension of a uniformly sampled signal. The task automatically generates MATLAB code for your live script. For more information about Live Editor tasks generally, see “Add Interactive Tasks to a Live Script”.

Correlation dimension is the measure of dimensionality of the space occupied by a set of random points. Correlation dimension is estimated as the slope of the correlation integral versus the range of radius of similarity. Use correlation dimension as a characteristic measure to distinguish between deterministic chaos and random noise, to detect potential faults.



Open the Task

To add the **Estimate Correlation Dimension** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task** > **Estimate Correlation Dimension**.
- In a code block in your script, type a relevant keyword, such as `correlation dimension` or `correlation dimension`. Select `Estimate Correlation Dimension` from the suggested command completions.

Examples

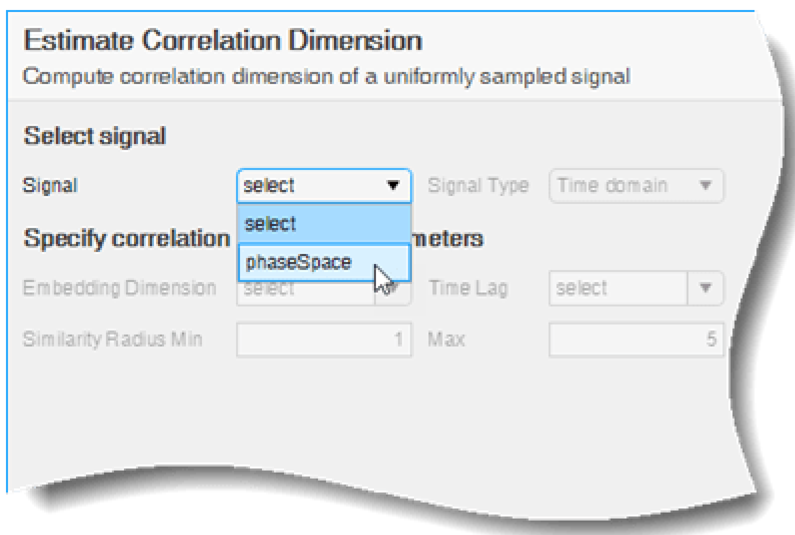
Estimate Correlation Dimension in the Live Editor

Use the **Estimate Correlation Dimension** task in the Live Editor to interactively estimate the correlation dimension of a uniformly sampled signal. Experiment with different values for lag, embedding dimension, similarity radius and number of points to align the linear fit line with the original data plot. The task automatically generates code reflecting your selections. Open this example to see a preconfigured script containing the **Estimate Correlation Dimension** task.

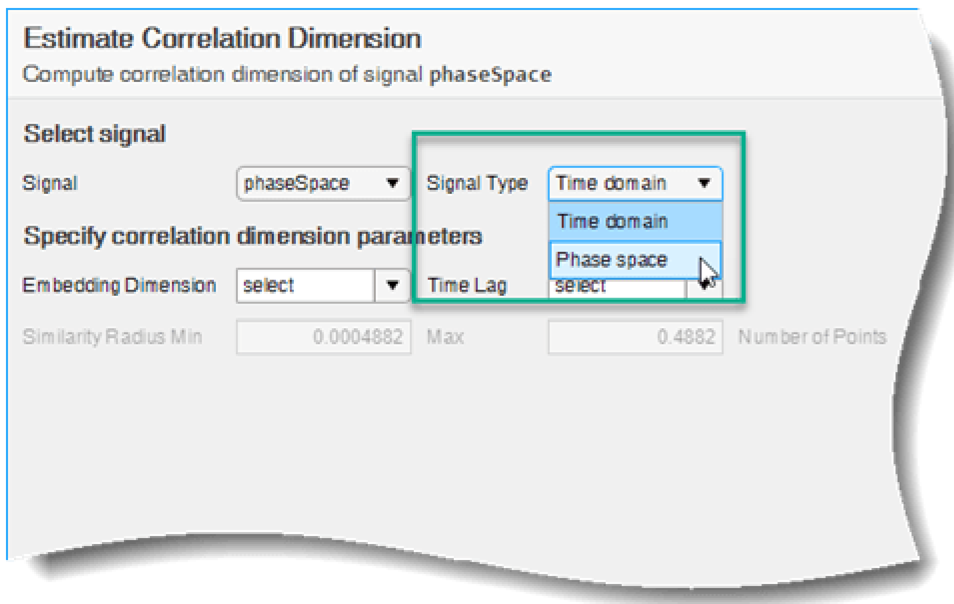
For this example, consider 'corrDimData`.mat`' which contains reconstructed phase space signal `phaseSpace`.

```
load('corrDimData.mat','phaseSpace')
```

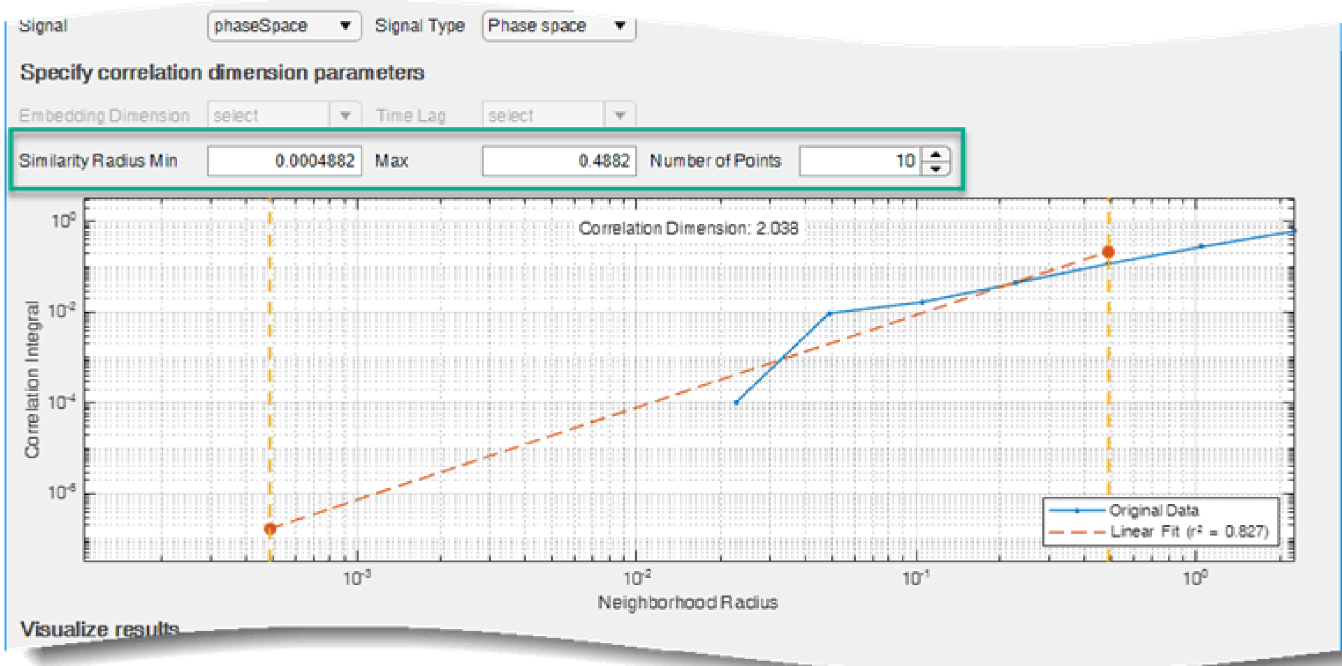
To estimate the correlation dimension of the signal `phaseSpace`, open the **Estimate Correlation Dimension** in the Live Editor. On the **Live Editor** tab, select **Task** > **Estimate Correlation Dimension**. In the task, select signal `phaseSpace`.



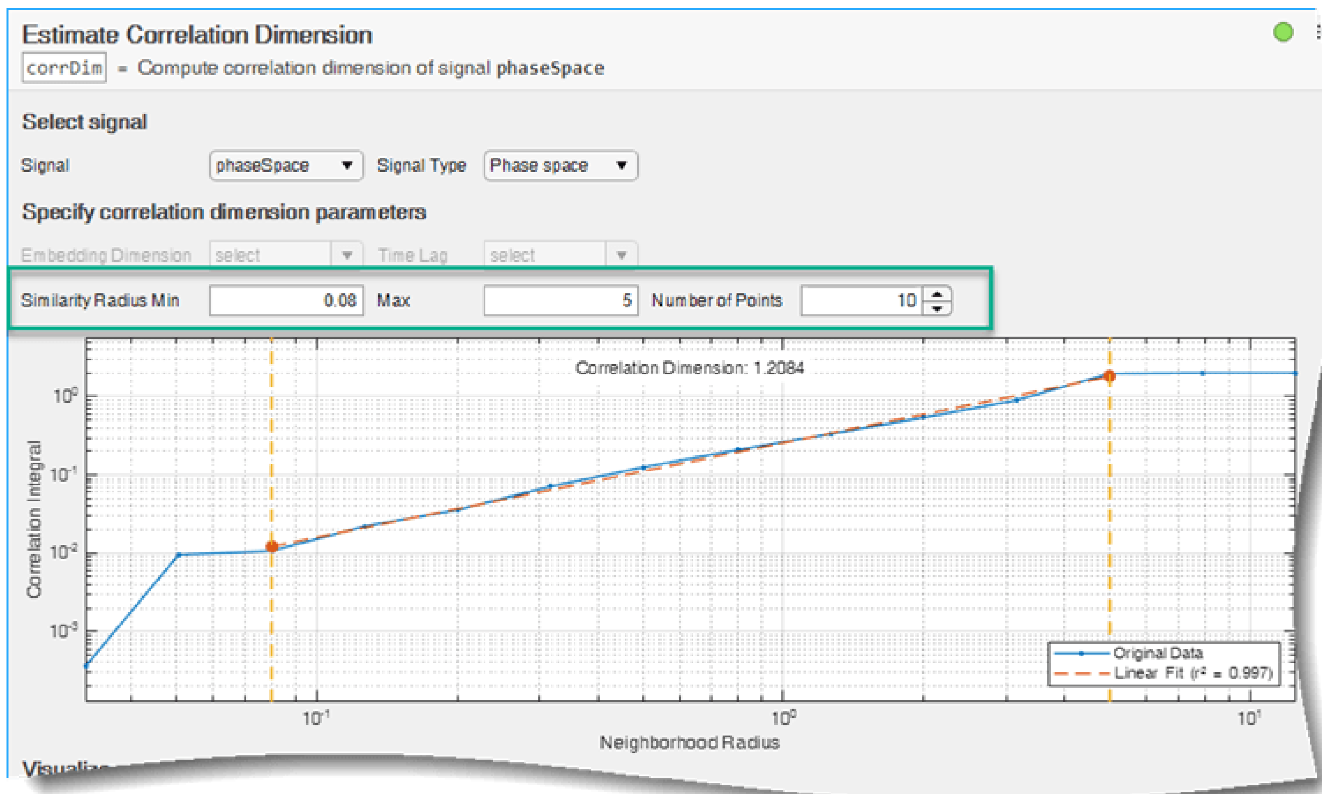
Since the selected signal is a phase space signal, select `Phase space` from the **Signal Type** dropdown menu.



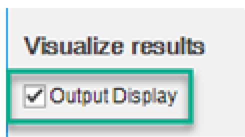
The **Estimate Correlation Dimension** task creates the correlation dimension plot with default values for the similarity radius and the number of points.

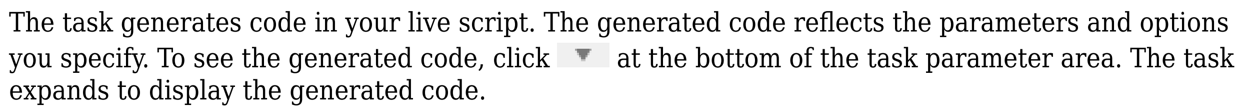


If your linear fit line does not align with the original data line using the default similarity radius values, try different values in the **Similarity Radius Min**, **Similarity Radius Max** and **Number of Points** fields until the alignment is satisfactory. For this example, use the minimum value of 0.08 and maximum value of 5 for the best alignment. The default value of 10 points provides good alignment for the signal `phaseSpace`.



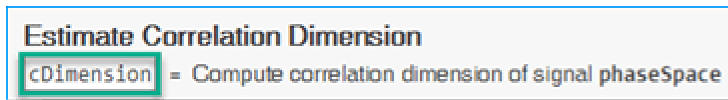
You can toggle displaying the output of the correlation dimension value in the Live Editor output using the **Output Display** option.



The task generates code in your live script. The generated code reflects the parameters and options you specify. To see the generated code, click  at the bottom of the task parameter area. The task expands to display the generated code.

```
Output Display

% Compute correlation dimension of signal phaseSpace
corrDim = correlationDimension(phaseSpace, ...
    'Dimension', 1, ...
    'Lag', 0, ...
    'MinRadius', 0.08, ...
    'MaxRadius', 5, ...
    'NumPoints', 10);

% Display the results
disp(corrDim)

    1.2084
```

By default, the generated code uses `corrDim` as the name of the output variable. To specify a different output variable name, enter a new name in the summary line at the top of the task. For instance, change the name to `cDimension`.



The task updates the generated code to reflect the new variable name, and the new variable `cDimension` appears in the MATLAB workspace. The value of correlation dimension is directly proportional to the level of chaos in the system, that is, a higher value of `cDimension` represents a high level of chaotic complexity in the system.

- "Reconstruct Phase Space and Estimate Condition Indicators Using Live Editor Tasks"

Parameters

Select Signal

`Signal` — Uniformly sampled time-domain signal

array | timetable

Select a uniformly sampled time-domain signal in array or timetable format from the MATLAB workspace. If the signal has multiple columns, the **Estimate Correlation Dimension** task computes the correlation dimension by treating it as a multivariate signal. If the signal is a row vector, then the **Estimate Correlation Dimension** task treats it as a univariate signal.

`Signal Type` — Type of selected signal

'`Time Domain`' | '`Phase space`'

Specify the type of the selected signal as either '`Time Domain`' or '`Phase space`'. If you specify the signal type as:

- '`Time Domain`', then also specify the embedding dimension and time lag for your signal.

- `'Phase space'`, then the **Estimate Correlation Dimension** task automatically infers the embedding dimension and time lag using the phase space information.

Specify Correlation Dimension Parameters

`Embedding Dimension` — Number of dimensions of phase space vectors
scalar | vector

Specify the number of dimensions of phase space vectors as a scalar or vector from the MATLAB workspace. When you specify the embedding dimension as a scalar, then the **Estimate Correlation Dimension** task uses the same embedding dimension value to estimate the value of correlation dimension for all the columns of the uniformly sampled signal.

The `Embedding Dimension` drop down is active only when you specify the signal type as `'Time Domain'`. For phase space signals, the **Estimate Correlation Dimension** task automatically computes the embedding dimension from the phase space data.

If you do not know the value of embedding dimension for your signal, then you can compute it using the **Reconstruct Phase Space** task.

`Time Lag` — Time lag between successive phase vectors
scalar | vector

Specify time lag between successive phase vectors as a scalar or vector from the MATLAB workspace. When you specify the time lag as a scalar, then the **Estimate Correlation Dimension** task uses the same time delay value to estimate the value of correlation dimension for all the columns of the uniformly sampled signal. If you specify the embedding dimension as a vector, then specify the time lag also as a vector of the same length.

The `Time Lag` drop down is active only when you specify the signal type as `'Time Domain'`. For phase space signals, the **Estimate Correlation Dimension** task automatically computes the time lag from the phase space data.

If you do not know the value of time lag for your signal, then you can compute it using the **Reconstruct Phase Space** task.

`Similarity Radius Min` — Minimum radius of similarity
`max radius/1000` (default) | scalar

Specify the minimum radius of similarity to be used to compute the number of with-in range points for correlation dimension estimation. Try different values such that the linear fit line aligns with the original data line in the plot.

`Similarity Radius Max` — Maximum radius of similarity
`0.2*sqrt(trace(cov(signal)))` (default) | scalar

Specify the maximum radius of similarity to be used to compute the number of with-in range points for correlation dimension estimation. Try different values such that the linear fit line aligns with the original data line in the plot.

`Number of Points` — Number of points between the minimum and maximum radius
10 (default) | positive scalar integer

Specify the number of points between the maximum and minimum radius of similarity. Choose an appropriate number of points based on the resolution required to compute the correlation dimension.

Visualize Results

`Output Display` — Toggle result display in the Live Editor output
on (default) | off

Toggle to display the value of correlation dimension in the Live Editor output.

See Also
Reconstruct Phase Space | `correlationDimension` | `phaseSpaceReconstruction` | `lyapunovExponent` | `approximateEntropy`

Topics
"Reconstruct Phase Space and Estimate Condition Indicators Using Live Editor Tasks"
"Add Interactive Tasks to a Live Script"

Introduced in R2019b

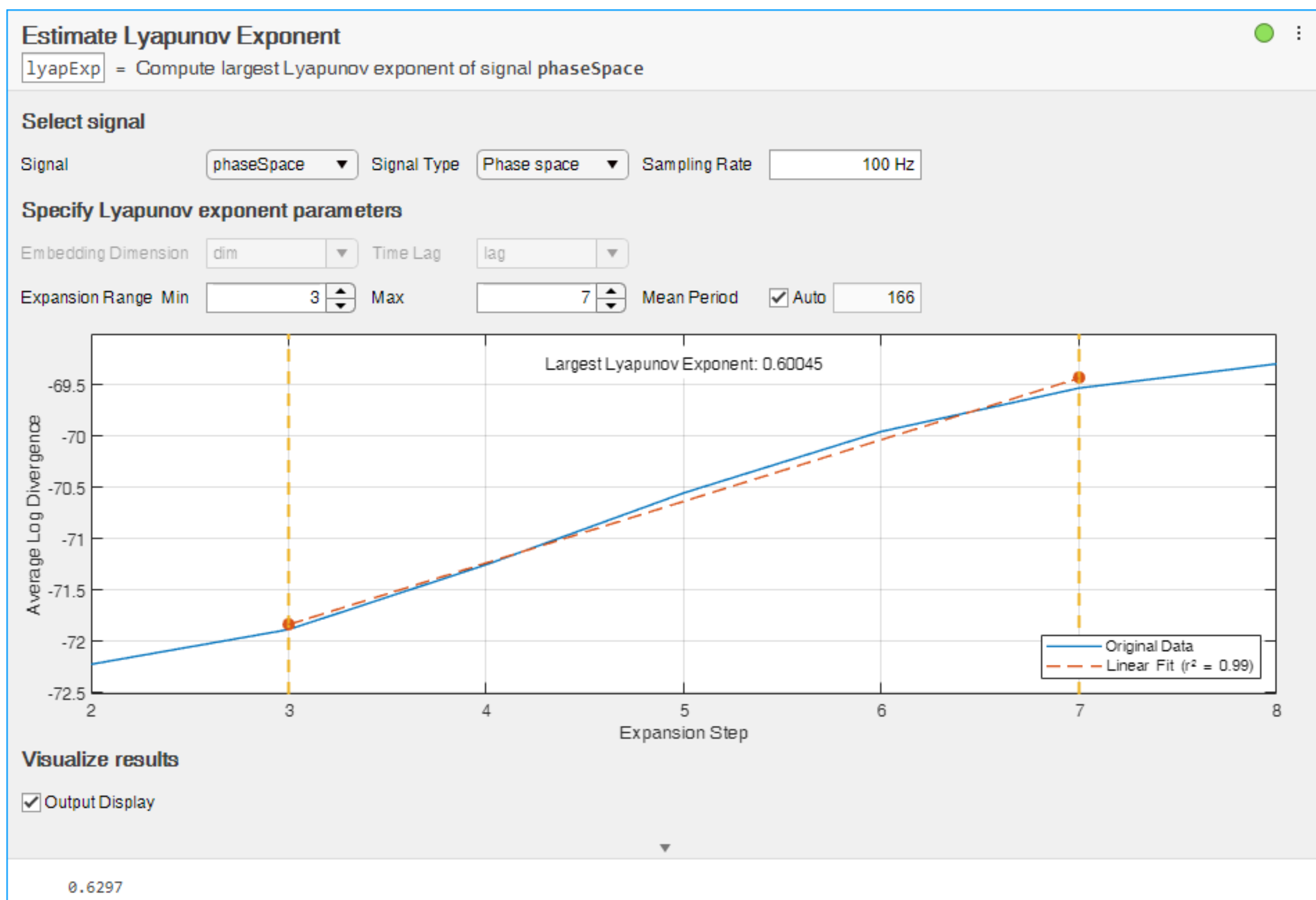
Estimate Lyapunov Exponent

Interactively estimate the Lyapunov exponent of a uniformly sampled signal in the Live Editor

Description

The **Estimate Lyapunov Exponent** task lets you interactively estimate the Lyapunov exponent of a uniformly sampled signal. The task automatically generates MATLAB code for your live script. For more information about Live Editor tasks generally, see “Add Interactive Tasks to a Live Script”.

Use the Lyapunov exponent to characterize the rate of separation of infinitesimally close trajectories in phase space to distinguish different attractors. The Lyapunov exponent is useful in quantifying the level of chaos in a system, which in turn can be used to detect potential faults. A negative Lyapunov exponent indicates convergence, while a positive Lyapunov exponents indicates divergence and chaos.



Open the Task

To add the **Estimate Lyapunov Exponent** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task** > **Estimate Lyapunov Exponent**.
- In a code block in your script, type a relevant keyword, such as `Lyapunov` or `Lyapunov exponent`. Select `Estimate Lyapunov Exponent` from the suggested command completions.

Examples

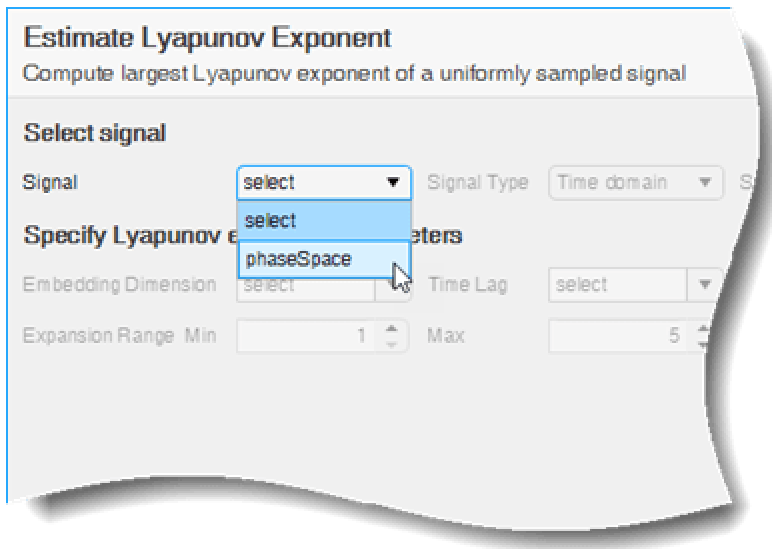
Estimate Lyapunov Exponent in the Live Editor

Use the **Estimate Lyapunov Exponent** task in the Live Editor to interactively estimate the Lyapunov exponent of a uniformly sampled signal. Experiment with different values for lag, embedding dimension, expansion range and mean period to align the linear fit line with the original data plot. The task automatically generates code reflecting your selections. Open this example to see a preconfigured script containing the **Estimate Lyapunov Exponent** task.

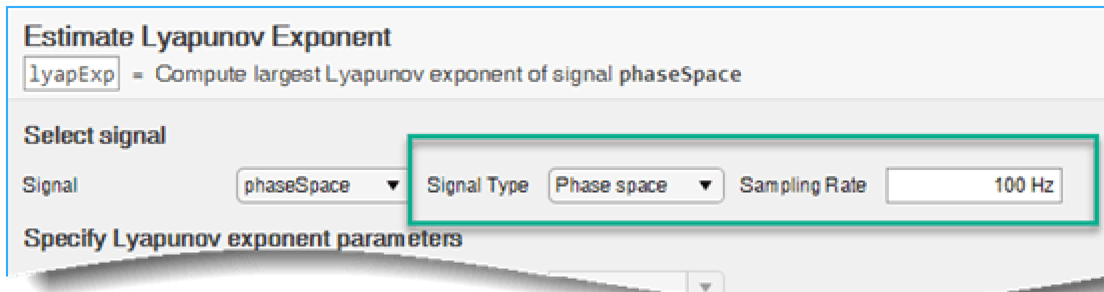
For this example, consider '`lyapExpData.mat`' which contains reconstructed phase space signal `phaseSpace` sampled at 100 Hz.

```
load('lyapExpData.mat','phaseSpace')
```

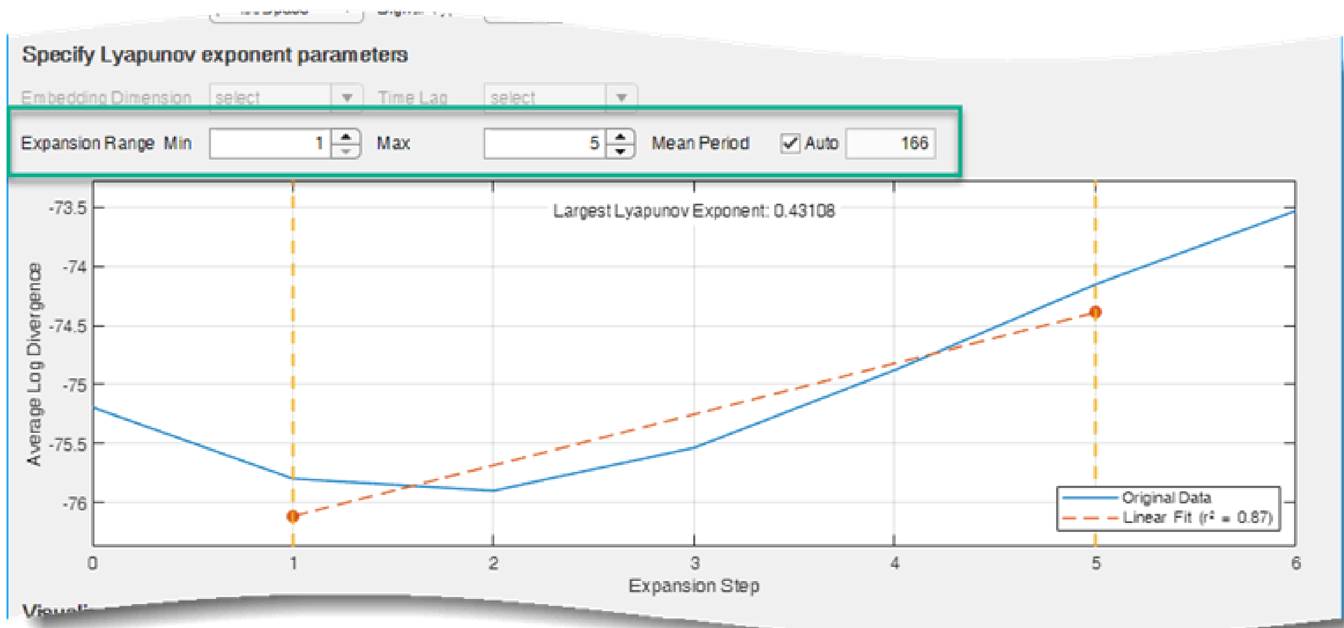
To estimate the Lyapunov exponent of the signal `phaseSpace`, open the **Estimate Lyapunov Exponent** in the Live Editor. On the **Live Editor** tab, select **Task** > **Estimate Lyapunov Exponent**. In the task, select signal `phaseSpace`.



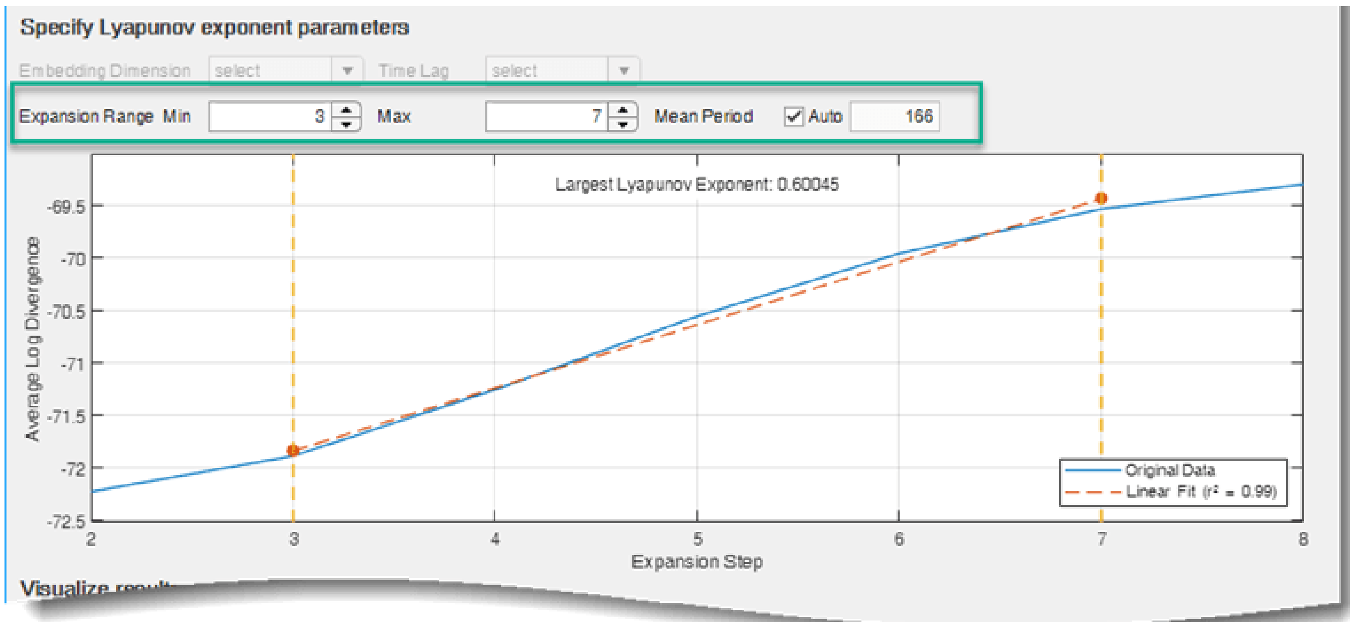
Since the selected signal is a phase space signal, select `Phase space` from the **Signal Type** dropdown menu. The signal was sampled at 100 Hz, hence specify this value in the **Sampling Rate** field.1



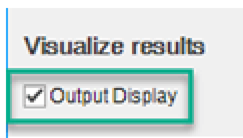
The **Estimate Lyapunov Exponent** task automatically computes the embedding dimension and lag from the phase space data and creates the Lyapunov exponent plot with default values for expansion range and mean period.

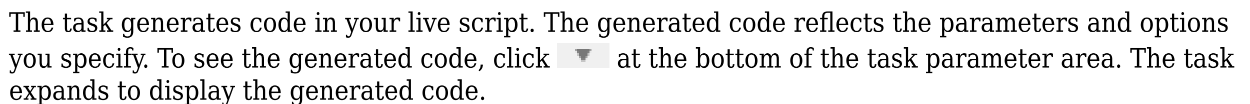


If your linear fit line does not align with the original data line using the default expansion range values, try different values in the **Expansion Range Min**, **Expansion Range Max** and **Mean Period** fields until the alignment is satisfactory. For this example, use the minimum value of 3 and maximum value of 7 for the best alignment. The default mean period value of 166 provides good alignment for the signal `phaseSpace`.



You can toggle displaying the output of the Lyapunov exponent value in the Live Editor output using the **Output Display** option.



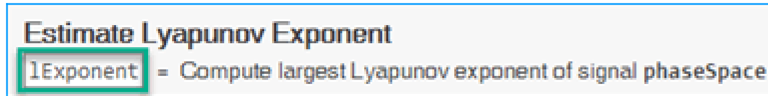
The task generates code in your live script. The generated code reflects the parameters and options you specify. To see the generated code, click  at the bottom of the task parameter area. The task expands to display the generated code.

Output Display

```
% Compute largest Lyapunov exponent of signal phaseSpace
lyapExp = lyapunovExponent(phaseSpace, 100, ...
    'Dimension', 1, ...
    'Lag', 0, ...
    'ExpansionRange', [3,7]);

% Display the results
disp(lyapExp)
```

```
0.6297
```

By default, the generated code uses `lyapExp` as the name of the output variable. To specify a different output variable name, enter a new name in the summary line at the top of the task. For instance, change the name to `lExponent`.



The task updates the generated code to reflect the new variable name, and the new variable `lExponent` appears in the MATLAB workspace. A negative Lyapunov exponent indicates convergence, while positive Lyapunov exponents demonstrate divergence and chaos. The magnitude of `lExponent` is an indicator of the rate of convergence or divergence of the infinitesimally close trajectories.

- "Reconstruct Phase Space and Estimate Condition Indicators Using Live Editor Tasks"

Parameters

Select Signal

`Signal` — Uniformly sampled time-domain signal

array | timetable

Select a uniformly sampled time-domain signal in array or timetable format. If the signal has multiple columns, the **Estimate Lyapunov Exponent** task computes the Lyapunov exponent by treating it as a multivariate signal. If the signal is a row vector, then the **Estimate Lyapunov Exponent** task treats it as a univariate signal.

`Signal Type` — Type of selected signal

'`Time Domain`' | '`Phase space`'

Specify the type of the selected signal as either '`Time Domain`' or '`Phase space`'. If you specify the signal type as:

- '`Time Domain`', then also specify the embedding dimension and time lag for your signal.
- '`Phase space`', then the **Estimate Correlation Dimension** task automatically computes the embedding dimension and time lag using the phase space information.

`Sampling Rate` — Sampling frequency of the data set

2π (default) | scalar

Specify the sampling frequency of the data set as a scalar. The **Estimate Lyapunov Exponent** task uses a value of 2π or `6.283 Hz` by default. When the signal data is in a timetable, the **Estimate Lyapunov Exponent** task infers the sampling rate from the data set.

Specify Lyapunov Exponent Parameters

`Embedding Dimension` — Number of dimensions of phase space vectors

scalar | vector

Specify the number of dimensions of phase space vectors as a scalar or vector from the MATLAB workspace. When you specify the embedding dimension as a scalar, then the **Estimate Lyapunov**

Exponent task uses the same embedding dimension value to estimate the value of Lyapunov exponent for all the columns of the uniformly sampled signal.

The `Embedding Dimension` drop down is active only when you specify the signal type as '`Time Domain`'. For phase space signals, the **Estimate Lyapunov Exponent** task automatically computes the embedding dimension from the phase space data.

If you do not know the value of embedding dimension for your signal, then you can compute it using the **Reconstruct Phase Space** task.

`Time Lag` — Time lag between successive phase vectors
scalar | vector

Specify time lag between successive phase vectors as a scalar or vector from the MATLAB workspace. When you specify the time lag as a scalar, then the **Estimate Lyapunov Exponent** task uses the same time delay value to estimate the value of Lyapunov exponent for all the columns of the uniformly sampled signal. If you specify the embedding dimension as a vector, then specify the time lag also as a vector of the same length.

The `Time Lag` drop down is active only when you specify the signal type as '`Time Domain`'. For phase space signals, the **Estimate Lyapunov Exponent** task automatically computes the time lag from the phase space data.

If you do not know the value of time lag for your signal, then you can compute it using the **Reconstruct Phase Space** task.

`Expansion Range Min` — Minimum expansion step value
`1` (default) | positive scalar integer

Specify the minimum expansion step value used to compute the expansion rate to estimate the Lyapunov exponent. Try different values such that the linear fit line aligns with the original data line in the plot.

`Expansion Range Max` — Maximum expansion step value
`5` (default) | positive scalar integer

Specify the maximum expansion step value used to compute the expansion rate to estimate the Lyapunov exponent. Try different values such that the linear fit line aligns with the original data line in the plot.

`Mean Period` — Threshold value for nearest neighbor computation
`ceil(fs/max(meanfreq(signal,sampling rate)))` (default) | positive scalar integer

Specify the threshold value to compute the nearest neighbor `i*` for a point `i` to estimate the largest Lyapunov exponent. For more information, see `lyapunovExponent`.

Visualize Results

`Output Display` — Toggle result display in the Live Editor output
on (default) | off

Toggle to display the value of Lyapunov exponent in the Live Editor output.

See Also

Reconstruct Phase Space | `correlationDimension` | `phaseSpaceReconstruction` | `lyapunovExponent` | `approximateEntropy`

Topics

"Reconstruct Phase Space and Estimate Condition Indicators Using Live Editor Tasks"
"Add Interactive Tasks to a Live Script"

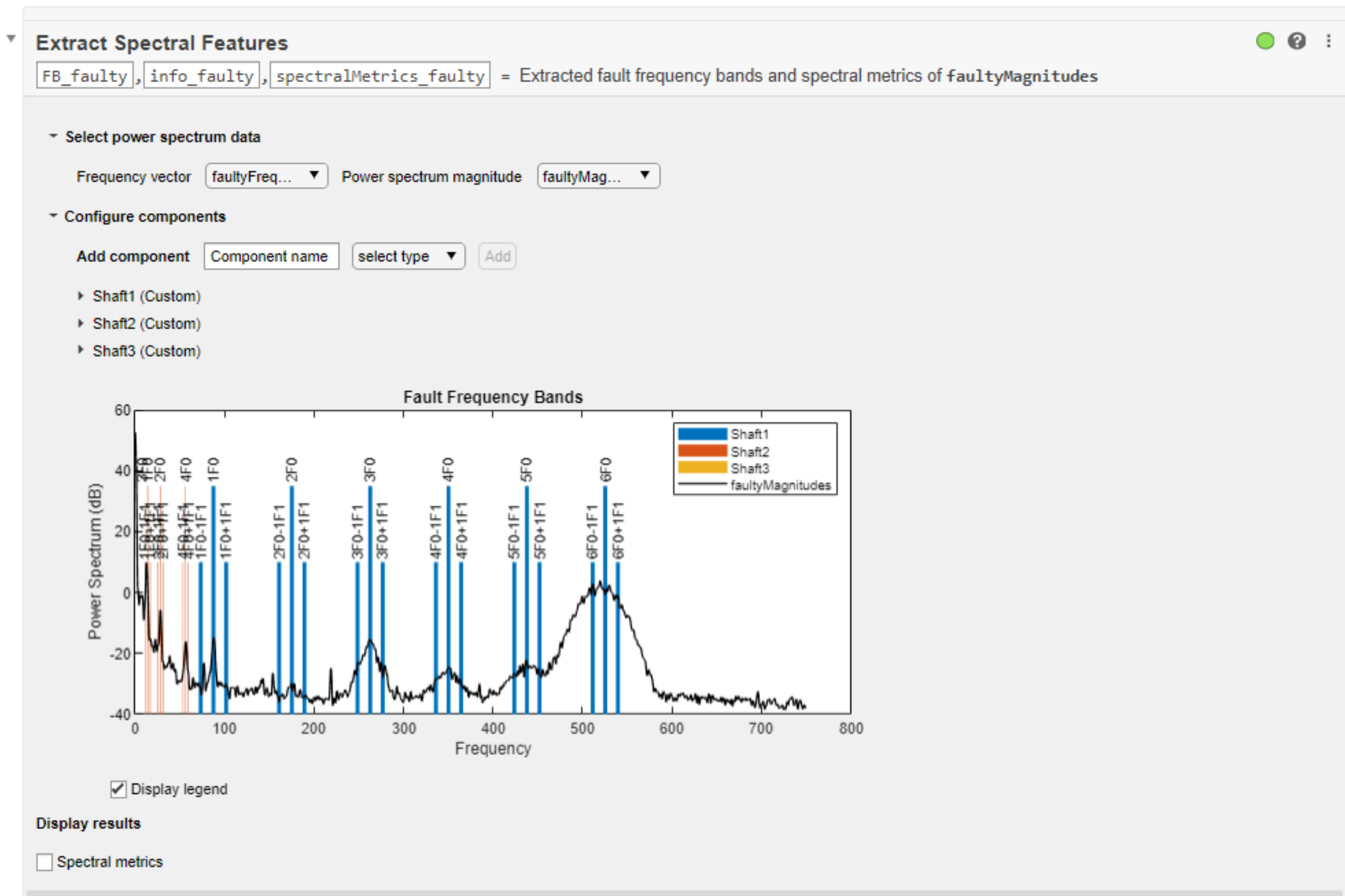
Introduced in R2019b

Extract Spectral Features

Interactively extract spectral fault band metrics in the Live Editor

Description

The **Extract Spectral Features** task lets you interactively extract spectral fault band metrics. The task helps with analyzing and understanding spectral data. Using a comprehensive interface, you can add components to represent various bearings, gear meshes, or other parts of your hardware setup. As you set the physical parameters of these components, the **Extract Spectral Features** Live Editor task will plot fault frequency bands at the characteristic frequencies of the components. You can overlay power spectrum data on the fault band plot to associate various peaks in the data with the components' characteristic frequencies. This can make fault detection and fault isolation easier, as changes in the power spectrum data can easily be traced back to the physical components causing them. In addition to a plot of the characteristic frequencies and the power spectrum data, the task will generate spectral metrics of the data within each characteristic frequency band. The output metrics table containing the peak amplitude, peak frequency, and band power of each band aids in characterizing potential mechanical faults. The task automatically generates MATLAB code for your live script. For more information about Live Editor tasks generally, see “Add Interactive Tasks to a Live Script”.



Open the Task

To add the **Extract Spectral Features** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task** > **Extract Spectral Features**.
- In a code block in your script, type a relevant keyword, such as `fault bands` or `metrics`. Select `Extract Spectral Features` from the suggested command completions.

Parameters

Select power spectrum data

`Frequency vector` — Frequencies corresponding to the power spectrum data

vector

Select a vector of frequencies from the MATLAB workspace that correspond to your power spectrum data.

`Power spectrum magnitude` — Power spectrum magnitude data

vector

Select a vector containing the power spectrum magnitudes from the MATLAB workspace.

Configure components

`Add component` — Bearing, gear mesh or custom hardware representation

'`Bearing`' | '`Gear mesh`' | '`Custom`'

Choose between adding a bearing, gear mesh or a custom component. You can name your component and then click the **Add** button. You can set the physical characteristics of these components using the parameters below. The **Extract Spectral Features** Live Editor task will plot fault frequency bands at the characteristic frequencies of the components.

Bearing Component Parameters

`Enable component` — Toggle to enable or disable component for spectral metrics computation

`on` (default) | `off`

You can toggle this option to enable or disable the component from being included in the spectral metrics computation. Disabling the component will also remove its fault bands from the plot. Use the **Delete** button to permanently remove a component.

`Number of balls` — Number of balls or rollers in the bearing

10 (default) | positive integer

Specify the number of rolling elements in the bearing.

`Pitch diameter` — Pitch diameter of the bearing

35 (default) | positive scalar

Pitch diameter of the bearing is the diameter of the circle that the center of the ball or roller travels during the bearing rotation.

`Rotational speed` — Rotational speed of the shaft or inner race of the bearing
60 (default) | positive scalar

Rotational speed of the shaft or inner race of the bearing. It is the fundamental frequency around which the **Extract Spectral Features** live task generates the fault frequency bands. The units must be consistent with the unit of the frequency vector.

`Contact angle` — Bearing contact angle
0 (default) | non-negative scalar

Contact angle in degrees between a plane perpendicular to the ball or roller axis and the line joining the two raceways.

`Ball diameter` — Diameter of the ball or roller
10 (default) | positive scalar

Diameter of the ball or roller in the bearing.

`Harmonics` — Harmonics of the fundamental frequency to be included
`1` (default) | vector of positive integers

Specify the harmonics of the fundamental frequency to be included in the plot and in the spectral metrics computation.

`Sidebands` — Sidebands around the fundamental frequency and its harmonics to be included
`0` (default) | vector of nonnegative integers

Specify the sidebands around the fundamental frequency and its harmonics to be included in the plot and in the spectral metrics computation.

`Domain` — Units of the fault band frequencies
`'frequency'` (default) | `'order'`

Specify the units of the fault band frequencies as either `'frequency'` or `'order'`. Select:

- `'frequency'` if you have the fault bands in the same units as the **Rotational speed**.
- `'order'` if you have the fault bands as number of rotations relative to the inner race rotation **Rotational speed**.

`Band width` — Width of the frequency bands centered at the nominal fault frequencies
auto (default) | positive scalar

Specify the width of the frequency bands centered at the nominal fault frequencies as a positive scalar. Uncheck the **Auto** option to specify the width value manually.

Gear Mesh Component Parameters

`Enable component` — Toggle to enable or disable component for spectral metrics computation
on (default) | `off`

You can toggle this option to enable or disable the component from being included in the spectral metrics computation. Disabling the component will also remove its fault bands from the plot. Use the **Delete** button to permanently remove a component.

`Input gear teeth` — Number of teeth on the input gear
10 (default) | positive integer

Specify the number of teeth on the input gear as a positive integer.

`Output gear teeth` — Number of teeth on the output gear
40 (default) | positive integer

Specify the number of teeth on the output gear as a positive integer.

`Rotational speed` — Rotational speed of the input gear
60 (default) | positive scalar

Specify the rotational speed of the input gear as a positive scalar. It is the fundamental frequency around which the **Extract Spectral Features** live task generates the fault frequency bands. The units must be consistent with the unit of the frequency vector.

`Harmonics` — Harmonics of the fundamental frequency to be included
`1` (default) | vector of positive integers

Specify the harmonics of the fundamental frequency to be included in the plot and in the spectral metrics computation.

`Sidebands` — Sidebands around the fundamental frequency and its harmonics to be included
`0` (default) | vector of nonnegative integers

Specify the sidebands around the fundamental frequency and its harmonics to be included in the plot and in the spectral metrics computation.

`Domain` — Units of the fault band frequencies
`'frequency'` (default) | `'order'`

Specify the units of the fault band frequencies as either `'frequency'` or `'order'`. Select:

- `'frequency'` if you have the fault bands in the same units as the **Rotational speed**.
- `'order'` if you have the fault bands as number of rotations relative to the **Rotational speed**.

`Band width` — Width of the frequency bands centered at the nominal fault frequencies
auto (default) | positive scalar

Specify the width of the frequency bands centered at the nominal fault frequencies as a positive scalar. Uncheck the **Auto** option to specify the width value manually.

Custom Component Parameters

`Enable component` — Toggle to enable or disable component for spectral metrics computation
on (default) | `off`

You can toggle this option to enable or disable the component from being included in the spectral metrics computation. Disabling the component will also remove its fault bands from the plot. Use the **Delete** button to permanently remove a component.

`Frequency` — Fundamental frequency of interest
60 (default) | positive scalar

Specify fundamental frequency of interest as a positive scalar. The **Extract Spectral Features** live task constructs the fault frequency bands around the fundamental frequency. For instance, to construct fault bands for a faulty induction motor, the mains frequency of 60 Hz is the fundamental frequency of interest. Similarly, to generate fault bands for a faulty gear train, the input shaft frequency is the fundamental frequency.

`Harmonics` — Harmonics of the fundamental frequency to be included

`1` (default) | vector of positive integers

Specify the harmonics of the fundamental frequency to be included in the plot and in the spectral metrics computation.

`Sidebands` — Sidebands around the fundamental frequency and its harmonics to be included

`0` (default) | vector of nonnegative integers

Specify the sidebands around the fundamental frequency and its harmonics to be included in the plot and in the spectral metrics computation.

`Separation type` — Type of separation between successive sidebands

`'additive'` (default) | `'multiplicative'`

Specify the type of separation between successive sidebands as either `'additive'` or `'multiplicative'`. Select:

- `'additive'`, to set the separation between successive sidebands to `0.1*F1` value, where `F1` is the distance of the first sideband from the fundamental frequency.
- `'multiplicative'`, to set the separation between successive sidebands proportional to both the harmonic order and the sideband value.

`Separation` — Separation value between successive sidebands

`Auto` (default) | positive scalar

Specify the separation value between successive sidebands as a positive scalar. Uncheck the **Auto** option to specify the separation value manually.

`Band width` — Width of the frequency bands centered at the nominal fault frequencies

auto (default) | positive scalar

Specify the width of the frequency bands centered at the nominal fault frequencies as a positive scalar. Uncheck the **Auto** option to specify the width value manually.

`Folding` — Toggle to specify whether negative nominal fault frequencies have to be folded about the frequency origin

`off` (default) | `on`

Toggle to specify whether negative nominal fault frequencies have to be folded about the frequency origin. If you turn **Folding** `on`, then the **Extract Spectral Features** live task folds the negative nominal fault frequencies about the frequency origin by taking their absolute values such that the folded fault bands always fall in the positive frequency intervals. The folded fault bands are computed as $\left[\max\left(0,\ |F| - \frac{W}{2}\right),\ |F| + \frac{W}{2}\right]$, where `W` is the **Band width** and `F` is the **Frequency**.

Display results

`Spectral metrics` — Toggle to enable or disable the display of spectral metrics

`off` (default) | `on`

Toggle this option enable or disable the display of spectral metrics. When the option is checked, then the **Extract Spectral Features** live task displays the metrics as a `1xN` table, where `N = 3*size((F+S),1)+1`, that is three metrics per frequency range and the total band power over the frequency range.

The live task returns the following spectral metrics for each frequency range:

- `Peak Amplitude` — Peak amplitude value for each specified frequency range.
- `Peak Frequency` — Peak frequency value for each specified frequency range.
- `Band Power` — Average power of each frequency range. For more information on band power, see `bandpower`.
- `Total Band Power` — Sum of individual band powers for the set of specified frequency ranges.

See Also

`faultBands` | `gearMeshFaultBands` | `bearingFaultBands` | `faultBandMetrics`

Topics

"Analyze Gear Train Data and Extract Spectral Features Using Live Editor Tasks"

Introduced in R2021a

faultBandMetrics

Spectral metrics for the specified fault frequency bands of the power spectral density (PSD)

Syntax

```
spectralMetrics = faultBandMetrics(psd,freqGrid,FB)
spectralMetrics = faultBandMetrics(X,FB)
spectralMetrics = faultBandMetrics(T,FB)
spectralMetrics = faultBandMetrics( ___ ,Name,Value)
[spectralMetrics,info] = faultBandMetrics( ___ )
```

Description

`spectralMetrics = faultBandMetrics(psd,freqGrid,FB)`returns a set of spectral metrics `spectralMetrics` for the power spectral density (PSD) data `psd` defined at the frequencies specified in `freqGrid` for each fault frequency range in `FB`.

The output `spectralMetrics` includes peak amplitude, peak frequency, and band powers for each frequency range specified in `FB` along with the total band power across all frequency bands.

`spectralMetrics = faultBandMetrics(X,FB)` returns a set of spectral metrics `spectralMetrics` for the PSD and frequency grid data specified in the cell array `X`. `faultBandMetrics` assumes that the last column of data in each cell of `X` contains the frequency grid while the first column contains PSD data. If the data is not in the same order, then use the '`SpectrumColumn`' and '`FrequencyColumn`' name-value pair arguments to specify the column numbers or names of the PSD data and the frequency grid, respectively. The output `spectralMetrics` has as many rows as the length of cell array `X`.

`spectralMetrics = faultBandMetrics(T,FB)` returns a set of spectral metrics `spectralMetrics` for the PSD and frequency grid data specified in the dataset `T`.

`T` can be a table/timetable or an ensemble, where a member variable of matrices or tables should contain the PSD data corresponding to one experiment. The last column of data in the member variable should contain the frequency grid and the first column should contain the PSD data.

If `T` is not in the same order, then use the '`SpectrumColumn`' and '`FrequencyColumn`' name-value pair arguments to specify the column numbers or names of the PSD data and the frequency grid, respectively. The output `spectralMetrics` has as many rows as the number of rows in dataset `T`.

`spectralMetrics = faultBandMetrics( ___ ,Name,Value)` allows you to specify additional parameters using one or more name-value pair arguments.

`[spectralMetrics,info] = faultBandMetrics( ___ )` also returns a structure `info` with additional information about the table or `fileEnsembleDatastore` object variables used to compute `spectralMetrics`.

Examples

Frequency Bands and Spectral Metrics of Gear Train

For this example, consider a simple gear set with an 8-toothed pinion on the input shaft meshing with a 42-toothed spur gear on the output shaft. Assume that the input shaft is driven at 20 Hz. The dataset `motorSignal.mat` contains vibration data for the gear mesh sampled at 1500 Hz.

First, construct the gear mesh frequency bands using the physical characteristics of the gear set. Construct the frequency bands with the first 3 sidebands.

```
Ni = 8;
No = 42;
FR = 20;
FB = gearMeshFaultBands(FR,Ni,No,'Sidebands',1:3)
```

```
FB = 15×2

   19.0000   21.0000
    2.8095    4.8095
   79.0000   81.0000
   99.0000  101.0000
  119.0000  121.0000
  139.0000  141.0000
  179.0000  181.0000
  199.0000  201.0000
  219.0000  221.0000
  147.5714  149.5714
      ⋮
```

`FB` is a 15x2 array which includes the primary frequencies and their sidebands.

Load the vibration data and compute PSD and frequency grid using `pspectrum`. Use a frequency resolution of 0.5.

```
load('motorSignal.mat','C');
fs = 1500;
[psd,freqGrid] = pspectrum(C,fs,'FrequencyResolution',0.5);
```

Now, use the frequency bands and PSD data to compute the spectral metrics.

```
spectralMetrics = faultBandMetrics(psd,freqGrid,FB)
```

```
spectralMetrics=1×46 table
    PeakAmplitude1    PeakFrequency1    BandPower1    PeakAmplitude2    PeakFrequency2    BandPo
    ______________    ______________    __________    ______________    ______________    ______

      0.0054125             19          0.0051216        0.55167             4.25           0.418
```

`spectralMetrics` is a 1x46 table with peak amplitude, peak frequency and band power calculated for each frequency range in `FB`. The last column in `spectralMetrics` is the total band power, computed across all 15 frequencies in `FB`.

Frequency Bands and Spectral Metrics of Ball Bearing

For this example, consider a ball bearing with a pitch diameter of 12 cm with 10 rolling elements. Each rolling element has a diameter of 0.5 cm. The outer race remains stationary as the inner race is

driven at 25 Hz. The contact angle of the ball is 0 degrees. The dataset `bearingData.mat` contains power spectral density (PSD) and its respective frequency data for the bearing vibration signal in a table.

First, construct the bearing frequency bands including the first 3 sidebands using the physical characteristics of the ball bearing.

```
FR = 25;
NB = 10;
DB = 0.5;
DP = 12;
beta = 0;
FB = bearingFaultBands(FR,NB,DB,DP,beta,'Sidebands',1:3)
```

```
FB = 14×2

  118.5417  121.0417
   53.9583   56.4583
   78.9583   81.4583
  103.9583  106.4583
  153.9583  156.4583
  178.9583  181.4583
  203.9583  206.4583
  262.2917  264.7917
  274.2708  276.7708
  286.2500  288.7500
      ⋮
```

`FB` is a 14x2 array which includes the primary frequencies and their sidebands.

Load the PSD data. `bearingData.mat` contains a table `X` where PSD is contained in the first column and the frequency grid is in the second column, as cell arrays respectively.

```
load('bearingData.mat','X')
X
```

```
X=1×2 table
          Var1                 Var2
    ________________    ________________

    {12001x1 double}    {12001x1 double}
```

Compute the spectral metrics using the PSD data in table `X` and the frequency bands in `FB`.

```
spectralMetrics = faultBandMetrics(X,FB)
```

```
spectralMetrics=1×43 table
    PeakAmplitude1    PeakFrequency1    BandPower1    PeakAmplitude2    PeakFrequency2    BandPow
    ______________    ______________    __________    ______________    ______________    _______

         121               121            314.43          56.438            56.438          144.9
```

`spectralMetrics` is a 1x43 table with peak amplitude, peak frequency and band power calculated for each frequency range in `FB`. The last column in `spectralMetrics` is the total band power, computed across all 14 frequencies in `FB`.

Compute Fault Band Metrics from Ensemble Datastore

Consider psdData.zip, a collection of 4 data sets where each file contains separate tables for the tachometer, vibration, and power spectrum data of a bearing. It also contains the read file for the ensemble hReadData.m.

Each dataset contains a table spectrum with 4 columns, where the first column F contains the frequency grid data, and the other three columns named Pxx, Pyy and Pzz contain spectral data.

Extract the compressed files, read the data in the table, and create a fileEnsembleDatastore object using the table data. For more information on creating a file ensemble datastore, see fileEnsembleDatastore.

```
unzip psdData.zip;
ens = fileEnsembleDatastore(pwd,'.mat');
% Make sure that the function for reading data is on path
addpath(fullfile(matlabroot,'examples','predmaint','main'))
ens.ReadFcn = @hReadData;
ens.DataVariables = {'tach','vibration','spectrum'};
ens.SelectedVariables = ens.DataVariables;
```

Assuming fault bands FB, compute the spectral metrics. Specify the spectral column, data variable and frequency columns to be used.

```
FB = [10,20;40,50;60,70]
```

```
FB = 3×2

    10    20
    40    50
    60    70
```

```
[spectralMetrics,info] = faultBandMetrics(ens,FB, ...
    'SpectrumColumn','Pxx', ...
    'FrequencyColumn','F', ...
    'DataVariable','spectrum');
size(spectralMetrics)
```

```
ans = 1×2

     4    10
```

The output table spectralMetrics contains 4 rows of metrics where each row corresponds to one data set.

```
info
```

```
info = struct with fields:
     SpectrumColumn: 'Pxx'
    FrequencyColumn: 'F'
       DataVariable: 'spectrum'
```

The structure info contains information about the data variable, frequency column and spectrum column used to compute the metrics.

```
rmpath(fullfile(matlabroot,'examples','predmaint','main')) % Reset path
```

Input Arguments

psd — Power spectral density data

vector | array

Power spectral density (PSD) data, specified as a vector or array. When psd is

- A vector, then faultBandMetrics converts it to a column vector and treats psd as a single channel.
- An array, then specify the PSD data column to be used with the 'SpectrumColumn' name-value pair.faultBandMetrics computes spectral metrics only for the PSD data column you specify.

For more information on computing PSD, see pspectrum.

freqGrid — Frequency grid data

vector

Frequency grid data corresponding to psd, specified as a vector. For more information on computing spectrum frequencies, see pspectrum.

FB — Fault frequency bands

Nx2 array

Fault frequency bands, specified as an Nx2 array, where N is the number of fault frequencies. The frequency bands specified in FB must be contained within the range of the frequency grid freqGrid. Also, the frequency units of the values in FB and the vector freqGrid must be the same.

X — PSD and frequency grid dataset

cell array of matrices | cell array of tables

PSD and frequency grid dataset, specified as a cell array of matrices or tables, where each cell contains the PSD data corresponding to one experiment. faultBandMetrics assumes that the last column of data in each cell contains the frequency grid while the first column contains PSD data. If the data is not in the same order, then use the 'SpectrumColumn' and 'FrequencyColumn' name-value pair arguments to specify the column numbers or names of the PSD data and the frequency grid, respectively.

T — PSD and frequency grid dataset

timetable | table of tables/timetables | fileEnsembleDatastore object

PSD and frequency grid dataset, specified as a timetable, table of tables/timetables or a fileEnsembleDatastore object where each member variable contains the PSD data corresponding to one experiment. faultBandMetrics assumes that the last column of data in the member variable of T contains the frequency grid while the first column contains PSD data. If T is not in the same order, then use the 'SpectrumColumn' and 'FrequencyColumn' name-value pair arguments to specify the column numbers or names of the PSD data and the frequency grid, 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: `...,'SpectrumColumn','Var1'`

`SpectrumColumn` — PSD data column to be used

first column of dataset (default) | integer | string

PSD data column to be used, specified as the comma-separated pair consisting of '`SpectrumColumn`' and an integer or a string. `faultBandMetrics` uses the first column of data by default. If the PSD data is not the first column of your cell array `X` or dataset `T`, use '`SpectrumColumn`' to specify the column numbers or names of the PSD data column.

When you specify '`DataVariable`', you must specify '`SpectrumColumn`' as a column of data in it.

When your dataset is in a cell array of matrices, you can use the values '`Var1`,'`Var2`',`...` to refer to the spectrum data columns.

`FrequencyColumn` — Frequency grid data column to be used

last column of dataset (default) | integer | string

Frequency grid data column to be used, specified as the comma-separated pair consisting of '`FrequencyColumn`' and an integer or string. `faultBandMetrics` uses the last column of data by default. If the frequency grid data is not the last column of your cell array `X` or dataset `T`, use '`FrequencyColumn`' to specify the column numbers or names of the frequency grid data column.

When your dataset is in a cell array of matrices, you can use the values `’Var1’,’Var2’,...` to refer to the spectrum data columns.

`DataVariable` — Data variable containing PSD and frequency grid data

`'Var1'` (default) | string

Data variable containing PSD and frequency grid data, specified as the comma-separated pair consisting of '`DataVariable`' and a string. Use '`DataVariable`' to specify the data variable containing both PSD and frequency grid data when the input dataset is a cell array of tables, a table of tables, tables/timetables of matrices, or a `fileEnsembleDatastore` object. '`DataVariable`' must be valid table variable name.

Output Arguments

`spectralMetrics` — Spectral metrics

table

Spectral metrics, returned as an `n`-by-`m` table, where

- `n` is the number of rows when dataset is a cell array `X`, or the number of members (rows) when the data is in a table or an ensemble `T`
- `m = 3*size(FB,1)+1`, that is three metrics per frequency range in `FB` and the total band power over the frequency range.

`faultBandMetrics` returns the following spectral metrics for each frequency range in `FB`:

- `Peak Amplitude` — Peak amplitude value for each frequency range in `FB`.
- `Peak Frequency` — Peak frequency value for each frequency range in `FB`.
- `Band Power` — Average power of each frequency range in `FB`. For more information on band power, see `bandpower`.
- `Total Band Power` — Sum of individual band powers for the set of frequency ranges in `FB`.

`info` — Data assignment information
structure

Data assignment information, returned as a structure with the following fields:

- `DataVariable` — Data variable being used from `X` or `T`
- `FrequencyColumn` — Frequency grid data column name
- `SpectrumColumn` — PSD data column name

Extended Capabilities

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

Usage notes and limitations:

- Data stored in `fileEnsembleDatastore` and `workspaceEnsemble` objects, as well as data in the form of a tall array are not supported.

See Also

`bandpower` | `pspectrum` | `bearingFaultBands` | `faultBands` | `gearMeshFaultBands`

Topics
"Motor Current Signature Analysis for Gear Train Fault Detection"

Introduced in R2019b

faultBands

Generate fault frequency bands for spectral feature extraction

Syntax

```
FB = faultBands(F0,N0)
FB = faultBands(F0,N0,F1,N1)
___ = faultBands(___,Name,Value)
[FB,info] = faultBands(___)

faultBands(___)
```

Description

`FB = faultBands(F0,N0)` generates fault frequency bands `FB`, using the fundamental frequency of interest `F0` and the array of harmonics `N0`. For instance, to construct fault bands for an induction motor, the mains frequency of 60 Hz is the fundamental frequency of interest.

`FB = faultBands(F0,N0,F1,N1)` constructs fault frequency bands `FB`, using the distance of the first sideband `F1` from the fundamental frequency `F0`. `N1` is the array of the sidebands around `F0`. If `F1` is not specified, then `faultBands` sets `F1` to 10 percent of `F0` by default. `N1` is equivalent to the '`Sidebands`' name-value pair. You can use the '`Type`' name-value pair to specify separation between successive sidebands.

`___ = faultBands(___,Name,Value)` allows you to specify additional parameters using one or more name-value pair arguments.

`[FB,info] = faultBands(___)` also returns the structure `info` containing information about the generated fault frequency bands `FB`.

`faultBands(___)` with no output arguments plots a bar chart of the generated fault frequency bands `FB`.

Examples

Frequency Bands of Electrical Mains Supply

For this example, generate frequency bands for analyzing the signal components around the first 5 harmonics of the mains supply frequency.

With the fundamental frequency of 60 Hz, the frequency of the alternating current in the mains power supply, use `faultBands` to generate the first 5 harmonics of the mains supply.

```
F0 = 60;
N0 = 1:5;
FB = faultBands(F0,N0)
```

```
FB = 5×2
```

```
   58.5000   61.5000
  118.5000  121.5000
  178.5000  181.5000
  238.5000  241.5000
  298.5000  301.5000
```

FB is returned as a 5x2 array with default frequency band width of 5% of F0 which is 3 Hz. The first column in FB contains the values of $F - \frac{W}{2}$, while the second column contains all the values of $F + \frac{W}{2}$ for each harmonic.

Frequency Bands of Faulty Induction Motor

For this example, consider an induction motor with broken rotor bars. Under normal operation with load, the rotor speed always lags the speed of the magnetic field allowing the rotor bars to cut magnetic lines of force and produce useful torque. This difference is called slip. Considering a slip value of 0.03 in the system with broken rotors, construct frequency bands for sideband components around the fundamental frequency of 60 Hz.

```
F0 = 60;
N0 = 1:2;
slip = 0.03;
F1 = 2*slip*F0;
N1 = 1:3;
[FB,info] = faultBands(F0,N0,F1,N1)
```

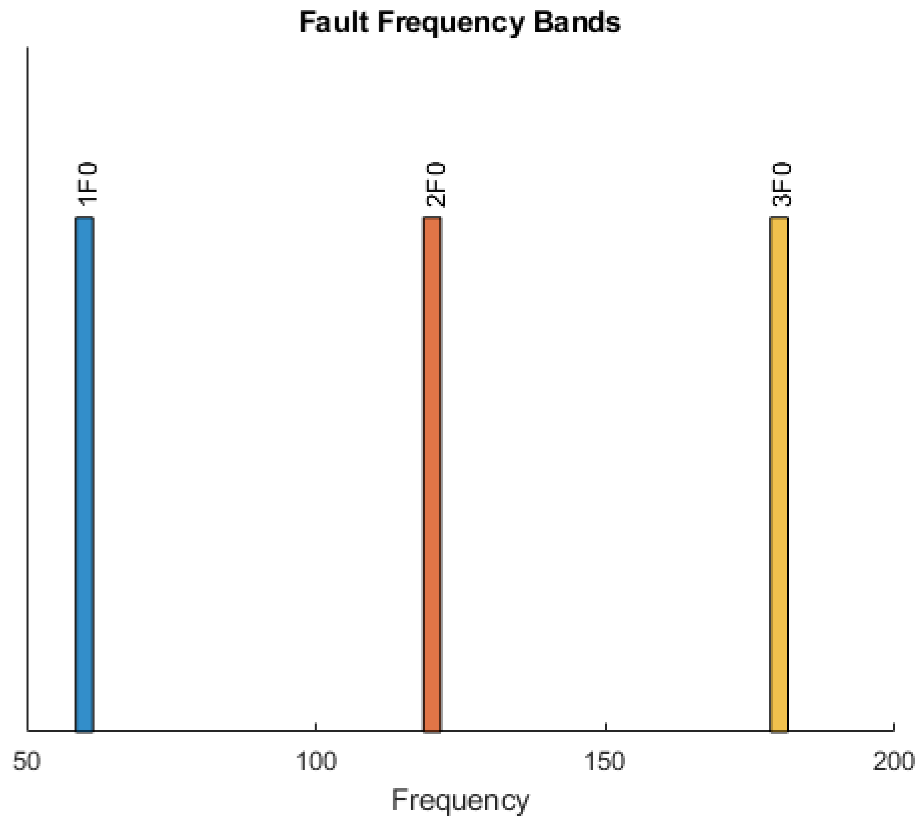
```
FB = 12×2

   47.7000   50.7000
   51.3000   54.3000
   54.9000   57.9000
   62.1000   65.1000
   65.7000   68.7000
   69.3000   72.3000
  107.7000  110.7000
  111.3000  114.3000
  114.9000  117.9000
  122.1000  125.1000
      ⋮

info = struct with fields:
           Centers: [49.2000 52.8000 56.4000 63.6000 67.2000 70.8000 ... ]
            Labels: ["1F0-3F1"    "1F0-2F1"    "1F0-1F1"    ...    ]
    HarmonicGroups: [1 1 1 1 1 1 2 2 2 2 2 2]
```

Visualize Frequency Bands and Harmonics of the Electrical Mains Supply

Construct frequency bands for analyzing the signal components around the first three harmonics of the electrical mains supply frequency.

With the fundamental frequency of 60 Hz, the alternating current in the mains power supply, use `faultBands` to visualize the first 3 harmonics of the mains supply.

```
F0 = 60;
N0 = 1:3;
faultBands(F0,N0)
```

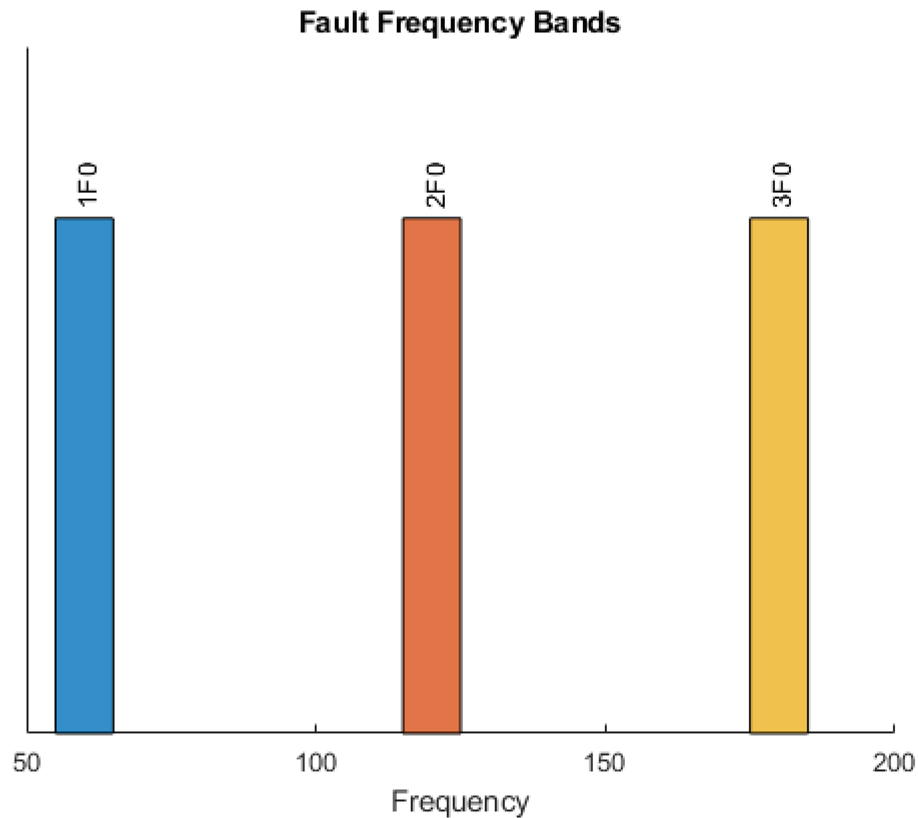


From the plot, observe the following:

- The fundamental frequency, which is also the first harmonic, `1F0` at 60 Hz
- The second harmonic, `2F0` at 120 Hz
- The third harmonic, `3F0` at 180 Hz

To better capture the expected variations of the actual system signals around the nominal fault frequencies, set the widths of each band to 10 Hz.

```
faultBands(F0,N0,'Width',10)
```



Folding Negative Fault Frequencies

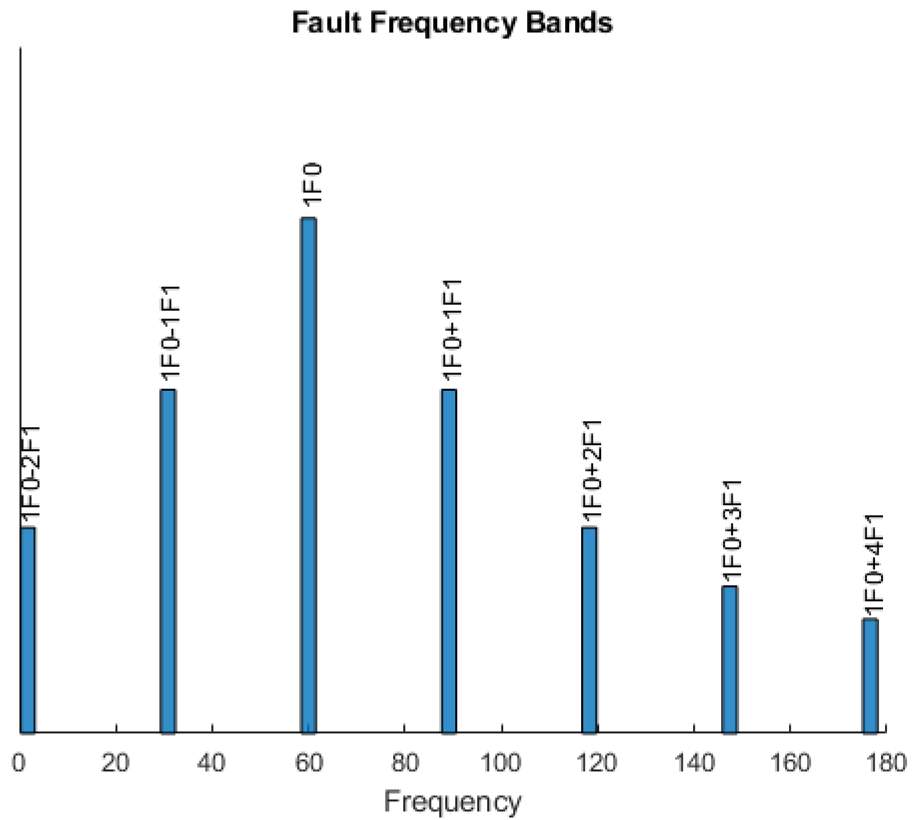
For this example, consider an induction motor with static and dynamic rotor eccentricities. Construct and visualize the frequency bands for the 4 sideband components of an induction motor with 4 pole pairs around the fundamental frequency due to the rotor eccentricities.

```
F0 = 60;
N0 = 1;
slip = 0.029;
polePairs = 4;
F1 = 2*F0*(1-slip)/polePairs
```

```
F1 = 29.1300
```

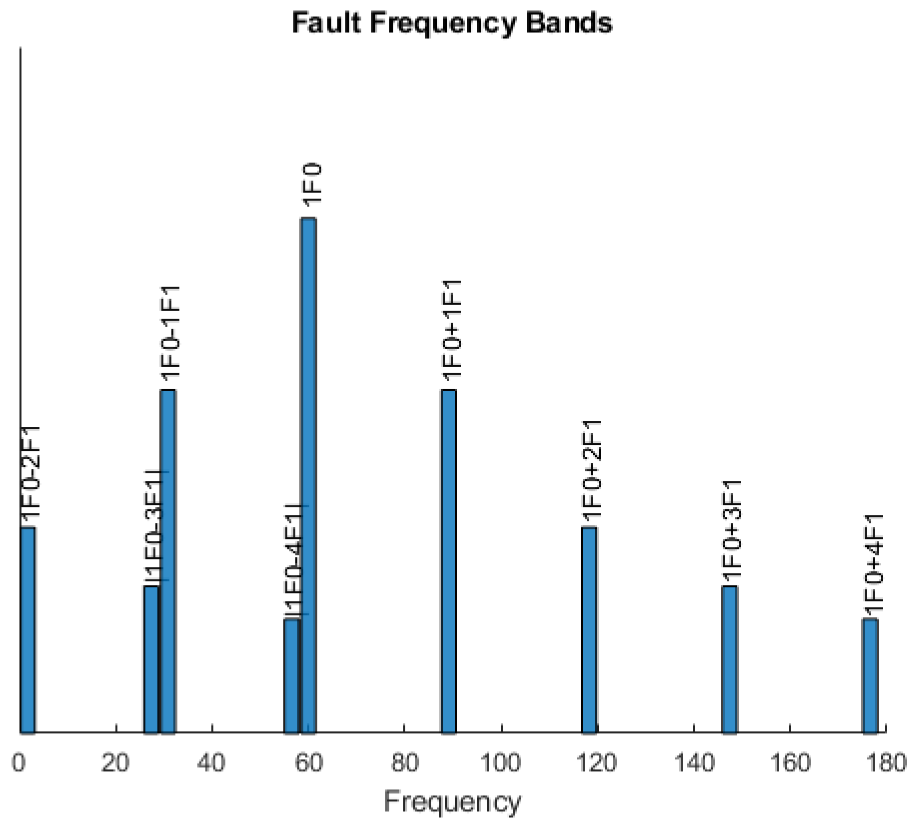
```
N1 = 0:4;
faultBands(F0,N0,F1,N1)
```

```
Warning: Truncated or removed negative fault frequency bands.
```



To avoid truncating negative fault frequency bands, set 'Folding' to true to fold them onto the positive frequency axis.

```
faultBands(F0,N0,F1,N1,'Folding',true)
```



Observe that the sideband frequencies `1F0-3F1` and `1F0-4F1` are now visible on the positive axis.

Input Arguments

F0 — Fundamental frequency of interest

positive scalar

Fundamental frequency of interest, specified as a positive scalar. `faultBands` constructs the fault frequency bands around the fundamental frequency `F0`. For instance, to construct fault bands for a faulty induction motor, the mains frequency of 60 Hz is the fundamental frequency of interest. Similarly, to generate fault bands for a faulty gear train, the input shaft frequency is the fundamental frequency.

You can specify `F0` in either hertz or orders.

N0 — Harmonics of the fundamental frequency

`1` (default) | vector of positive integers

Harmonics of the fundamental frequency, specified as a vector of positive integers. Specify fault bands around the fundamental frequency `F0` and its harmonics by `N0`. `N0` is equivalent to the '`Harmonics`' name-value pair with a default value of 1.

F1 — Distance of the first sideband from the fundamental frequency

`0.1*F0` (default) | positive scalar

Distance of the first sideband from the fundamental frequency, specified as a positive scalar. If F1 is not specified, then faultBands assumes a value of 10 percent of the fundamental frequency for F1.

N1 — Sidebands of the fundamental frequency and its harmonics
vector of nonnegative integers

Sidebands of the fundamental frequency and its harmonics, specified as a vector of nonnegative integers. N1 is equivalent to the 'Sidebands' name-value pair with a default value of 0.

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: ...,'Harmonics',[1,3,5]

Harmonics — Harmonics of the fundamental frequency to be included
1 (default) | vector of positive integers

Harmonics of the fundamental frequency to be included, specified as the comma-separated pair consisting of 'Harmonics' and a vector of positive integers. The default value is 1. Specify 'Harmonics' when you want to construct the frequency bands with more harmonics of the fundamental frequency.

Sidebands — Sidebands around the fundamental frequency and its harmonics to be included
0 (default) | vector of nonnegative integers

Sidebands around the fundamental frequency and its harmonics to be included, specified as the comma-separated pair consisting of 'Sidebands' and a vector of nonnegative integers. The default value is 0. Specify 'Sidebands' when you want to construct the frequency bands with sidebands around the fundamental frequency and its harmonics.

Width — Width of the frequency bands centered at the nominal fault frequencies
5 percent of the fundamental frequency (default) | positive scalar

Width of the frequency bands centered at the nominal fault frequencies, specified as the comma-separated pair consisting of 'Width' and a positive scalar. The default value is 5 percent of the fundamental frequency. Avoid specifying 'Width' with a large value so that the fault bands do not overlap.

Type — Separation value between successive sidebands
'additive' (default) | 'multiplicative'

Separation value between successive sidebands, specified as the comma-separated pair consisting of 'Type' and either 'additive' or 'multiplicative'. Specify 'Type' as:

- 'additive', to set the separation between successive sidebands to F1.
- 'multiplicative', to set the separation between successive sidebands proportional to both the harmonic order and the sideband value.

Folding — Logical value specifying whether negative nominal fault frequencies have to be folded about the frequency origin
false (default) | true

Logical value specifying whether negative nominal fault frequencies have to be folded about the frequency origin, specified as the comma-separated pair consisting of 'Folding' and either true or false. If you set 'Folding' to true, then faultBands folds the negative nominal fault frequencies about the frequency origin by taking their absolute values such that the folded fault bands always fall in the positive frequency intervals. The folded fault bands are computed as $\left[\max\left(0,\ |F| - \frac{W}{2}\right),\ |F| + \frac{W}{2}\right]$, where W is the 'Width' name-value pair and F is one of the nominal fault frequencies.

Output Arguments

FB — Fault frequency bands

Nx2 array

Fault frequency bands, returned as an Nx2 array, where N is the number of fault frequencies. FB is returned in the same units as F0, in either Hertz or orders. The generated fault bands, $\left[F - \frac{W}{2},\ F + \frac{W}{2}\right]$, are centered depending on the sideband specification as follows:

- If you do not specify the sidebands, then the fault bands are centered at $F = n_0F_0$, where the integer n0 ranges through the elements of the array of harmonics, N0.
- If you specify sidebands using N1 or the 'Sidebands' name-value pair, then fault bands are centered at:
  - $F = n_0F_0 \pm n_1F_1$, when 'Type' is specified as 'additive'. Here, the integer n1 ranges through the elements of the array of sidebands, N1.
  - $F = n_0(F_0 \pm n_1F_1)$, when 'Type' is specified as 'multiplicative'.

info — Information about the fault frequency bands

structure

Information about the fault frequency bands in FB, returned as a structure with the following fields:

- Centers — Center fault frequencies
- Labels — Labels describing each frequency
- HarmonicGroups — Harmonic group numbers equal to the harmonic order of each frequency band to be able to identify fault bands associated with the nominal fault frequency $F = n_0F_0$, where the integer n0 ranges through the elements of the array of harmonics, N0

See Also

faultBandMetrics | bearingFaultBands | gearMeshFaultBands

Topics

"Motor Current Signature Analysis for Gear Train Fault Detection"

Introduced in R2019b

findIndex

Find the workspace ensemble member indices for members that match a specified variable name and value

Syntax

```
index = findIndex(wensemble,varname,value)
```

Description

`findIndex` is a function used in code generated by **Diagnostic Feature Designer**.

`index = findIndex(wensemble,varname,value)` finds the indices of members that contain the value of the variable `varname`.

For example, when you specify `findIndex(outputEnsemble,'File',filename)`, where `filename` identifies the last file read from an ensemble datastore, `findIndex` finds the index of the workspace ensemble member that is associated with that file name.

Code that is generated by **Diagnostic Feature Designer** uses `writeMember`, `readMember`, and `findIndex` under the following conditions:

- The input data is a file or simulation ensemble datastore.
- The computation option during code generation specified storing results in local memory rather than writing results back to the ensemble datastore.

Explicitly specifying a member index when reading and writing within the local version of the data, which the code manages using a `workspaceEnsemble` object, ensures member synchronization with the original ensemble datastore. This synchronization is necessary when you have sequential member-processing loops, such as when you compute ensemble statistics as a precursor to computing signal residues.

- During the first member-processing loop, which starts with an empty ensemble, no indexing is needed. The code appends each new member result to the end of the ensemble.
- During the second loop, the index enables the code to write updated member results to the correct location within the now-populated ensemble.

For more information about the dual processing loop for ensemble statistics, see “Anatomy of App-Generated MATLAB Code”.

Input Arguments

`wensemble` — Ensemble object
`workspaceEnsemble` object

Ensemble object, specified as a `workspaceEnsemble` object. `wensemble` contains ensemble data and specifies variable names and types.

`varname` — Variable name
`string`

Variable name to match, specified as a string.

Example: 'File'

value — Variable value
number | string

Variable value to match, specified as a string.

Example: filename

Output Arguments

index — Member index
positive integer vector | []

Member index for ensemble members that contain a specified variable name and value, returned as a vector of positive integers with length equal to the number of matching members. If no members contain the specified name-value combination, findIndex returns []. In code generated by Diagnostic Feature Designer, index is either a single integer or [], and identifies the member with the file name that matches the file name in the input argument.

See Also

Diagnostic Feature Designer | fileEnsembleDatastore | simulationEnsembleDatastore | workspaceEnsemble | readMember | writeMember

Topics

"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a

fit

Estimate parameters of remaining useful life model using historical data

Syntax

```
fit(mdl,data)
fit(mdl,data,lifeTimeVariable)
fit(mdl,data,lifeTimeVariable,dataVariables)

fit(mdl,data,lifeTimeVariable,dataVariables,censorVariable)
fit(mdl,data,lifeTimeVariable,dataVariables,censorVariable,encodedVariables)
```

Description

The `fit` function estimates the parameters of a remaining useful life (RUL) prediction model using historical data regarding the health of an ensemble of similar components, such as multiple machines manufactured to the same specifications. Depending on the type of model, you specify the historical health data as a collection of lifespan measurements or degradation profiles. Once you estimate the parameters of your model, you can then predict the remaining useful life of similar components using the `predictRUL` function.

Using `fit`, you can configure the parameters for the following types of estimation models:

- Degradation models
- Survival models
- Similarity models

For a basic example illustrating RUL prediction, see “Update RUL Prediction as Data Arrives”.

For general information on predicting remaining useful life using these models, see “RUL Estimation Using RUL Estimator Models”.

`fit(mdl,data)` fits the parameters of the remaining useful life model `mdl` using the historical data in `data`. This syntax applies only when `data` does not contain `table` or `timetable` data.

`fit(mdl,data,lifeTimeVariable)` fits the parameters of `mdl` using the time variable `lifeTimeVariable` and sets the `LifeTimeVariable` property of `mdl`. This syntax applies only when `data` contains:

- Nontabular data
- Tabular data, and `mdl` does not use data variables

`fit(mdl,data,lifeTimeVariable,dataVariables)` fits the parameters of `mdl` using the data variables in `dataVariables` and sets the `DataVariables` property of `mdl`.

`fit(mdl,data,lifeTimeVariable,dataVariables,censorVariable)` specifies the censor variable for a survival model and sets the `CensorVariable` property of `mdl`. The censor variable indicates which life-time measurements in `data` are not end-of-life values. This syntax applies only when `mdl` is a survival model and `data` contains tabular data.

`fit(mdl,data,lifeTimeVariable,dataVariables,censorVariable,encodedVariables)` specifies the encoded variables for a covariate survival model and sets the `EncodedVariables` property of `mdl`. Encoded variables are usually nonnumeric categorical features that `fit` converts to numeric vectors before fitting. This syntax applies only when `mdl` is a `covariateSurvivalModel` object and `data` contains tabular data.

Examples

Train Linear Degradation Model

Load training data.

```
load('linTrainVectors.mat')
```

The training data is a cell array of column vectors. Each column vector is a degradation feature profile for a component.

Create a linear degradation model with default settings.

```
mdl = linearDegradationModel;
```

Train the degradation model using the training data.

```
fit(mdl,linTrainVectors)
```

Train Reliability Survival Model

Load training data.

```
load('reliabilityData.mat')
```

This data is a column vector of `duration` objects representing battery discharge times.

Create a reliability survival model with default settings.

```
mdl = reliabilitySurvivalModel;
```

Train the survival model using the training data.

```
fit(mdl,reliabilityData,"hours")
```

Train Hash Similarity Model Using Tabular Data

Load training data.

```
load('hashTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component. Each profile consists of life time measurements in the `"Time"` variable and corresponding degradation feature measurements in the `"Condition"` variable.

Create a hash similarity model that uses the following values as hashed features:

```
mdl = hashSimilarityModel('Method',@(x) [mean(x),std(x),kurtosis(x),median(x)]);
```

Train the similarity model using the training data. Specify the names of the life time and data variables.

```
fit(mdl,hashTrainTables,"Time","Condition")
```

Predict RUL Using Covariate Survival Model

Load training data.

```
load('covariateData.mat')
```

This data contains battery discharge times and related covariate information. The covariate variables are:

- Temperature
- Load
- Manufacturer

The manufacturer information is a categorical variable that must be encoded.

Create a covariate survival model, and train it using the training data.

```
mdl = covariateSurvivalModel('LifeTimeVariable',"DischargeTime",'LifeTimeUnit',"hours",...
   'DataVariables',["Temperature","Load","Manufacturer"],'EncodedVariables',"Manufacturer");
fit(mdl,covariateData)
```

```
Successful convergence: Norm of gradient less than OPTIONS.TolFun
```

Suppose you have a battery pack manufactured by maker `B` that has run for `30` hours. Create a test data table that contains the usage time, `DischargeTime`, and the measured ambient temperature, `TestAmbientTemperature`, and current drawn, `TestBatteryLoad`.

```
TestBatteryLoad = 25;
TestAmbientTemperature = 60;
DischargeTime = hours(30);
TestData = timetable(TestAmbientTemperature,TestBatteryLoad,"B",'RowTimes',hours(30));
TestData.Properties.VariableNames = {'Temperature','Load','Manufacturer'};
TestData.Properties.DimensionNames{1} = 'DischargeTime';
```

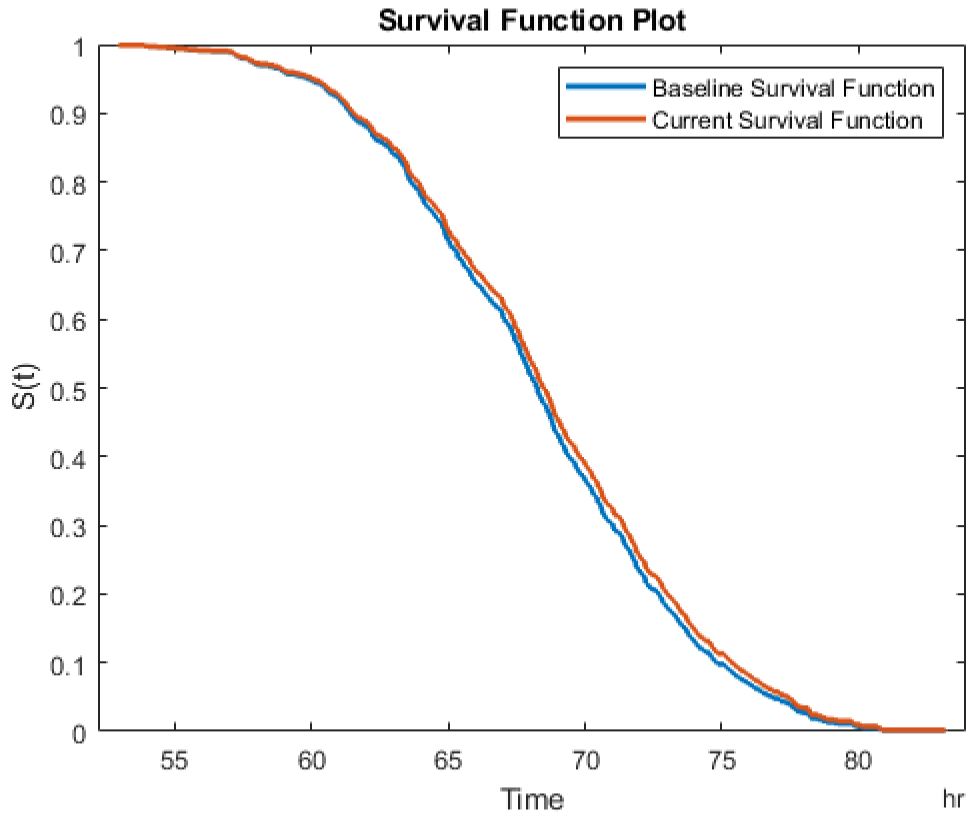
Predict the RUL for the battery.

```
estRUL = predictRUL(mdl,TestData)
```

```
estRUL = duration
   38.332 hr
```

Plot the survival function for the covariate data of the battery.

```
plot(mdl,TestData)
```



Input Arguments

`mdl` — Remaining useful life prediction model

degradation model | survival model | similarity model

Remaining useful life prediction model, specified as one of these models. `fit` updates the parameters of this model using the historical data in `data`.

RUL Model Groups	Prediction Model
Degradation models	`linearDegradationModel`
	`exponentialDegradationModel`
Survival models	`reliabilitySurvivalModel`
	`covariateSurvivalModel`
Similarity models	`hashSimilarityModel`
	`pairwiseSimilarityModel`
	`residualSimilarityModel`

For more information on the different model types and when to use them, see “Models for Predicting Remaining Useful Life”.

`data` — Historical data

column vector | array | table | timetable | cell array

Historical data regarding the health of an ensemble of similar components, such as their degradation profiles or life spans, specified as an array or table of component life times, or a cell array of degradation profiles.

If your historical data is stored in an ensemble datastore object, you must first convert it to a `table` before estimating your model parameters. For more information, see “Data Ensembles for Condition Monitoring and Predictive Maintenance”.

The format of `data` depends on the type of RUL model you specify in `mdl`.

Degradation Model

If `mdl` is a `linearDegradationModel` or `exponentialDegradationModel`, specify `data` as a cell array of component degradation profiles. Each element of the cell array contains the degradation feature profile across the lifetime of a single component. There can be only one degradation feature for your model. You can specify `data` as a cell array of:

- Two-column arrays, where each row contains the usage time in the first column and the corresponding feature measurement in the second column. In this case, the usage time column must contain numeric values; that is, it cannot use, for example, `duration` or `timedate` values.
- `table` objects. Select the variable from the table that contains the feature degradation profile using `dataVariables`, and select the usage time variable, if present, using `lifeTimeVariable`.
- `timetable` objects. Select the variable from the table that contains the feature degradation profile using `dataVariables`, and select the usage time variable using `lifeTimeVariable`.

Survival Model

For survival models, `data` contains the life span measurements for multiple components. Also, for covariate survival models, `data` contains corresponding time-independent covariates, such as the component provider or working regimes. Specify `data` as one of the following:

- Column vector of life span measurements — This case applies only when `mdl` is a `reliabilitySurvivalModel`.
- Array — The first column contains the life span measurements, and the remaining columns contain the covariate values. This case applies only when `mdl` is a `covariateSurvivalModel`.
- `table` or `timetable` — In this case, select the variable from the table that contains the life span measurements using `lifeTimeVariable`. For covariate survival models, select the covariate variables using `dataVariables`. For reliability survival models, `fit` ignores `dataVariables`.

By default, `fit` assumes that all life span measurements are end-of-life values. To indicate that a life span measurement is not an end-of-life value, use censoring. To do so, specify `data` as a `table` or `timetable` that contains a censor variable. The censor variable is a binary variable that is `1` when the corresponding life span measurement is not an end-of-life value. Select the censor variable using `censorVariable`.

Similarity Model

If `mdl` is a `hashSimilarityModel`, `pairwiseSimilarityModel`, or `residualSimilarityModel`, specify `data` as a cell array of degradation profiles. Each element of the cell array contains degradation feature profiles across the lifetime a single component. For similarity models, you can specify multiple degradation features, where each feature is a health indicator for the component. You can specify `data` as a cell array of:

- N-by-(M_i+1) arrays, where N is the number of feature measurements (at different usage times) and M_i is the number of features. The first column contains the usage times and the remaining columns contain the corresponding measurements for degradation features.
- `table` objects. Select the variables from the table that contain the feature degradation profiles using `dataVariables`, and select the corresponding usage time variable, if present, using `lifeTimeVariable`.
- `timetable` objects. Select the variables from the table that contain the feature degradation profiles using `dataVariables`, and select the corresponding usage time variable using `lifeTimeVariable`.

`fit` assumes that all the degradation profiles represent run-to-failure data; that is, the data starts when the component is in a healthy state and end when the component is close to failure or maintenance.

`lifeTimeVariable` — Life time variable
`""` (default) | string

Life time variable, specified as a string. If `data` is a:

- `table`, then `lifeTimeVariable` must match one of the variable names in the table.
- `timetable`, then `lifeTimeVariable` one of the variable names in the table or the dimension name of the time variable , `data.Properties.DimensionNames{1}`.

`table` or `timetable`, then `lifeTimeVariable` must match one of the variable names in the table. If there is no life time variable in the table or if `data` is nontabular, then you can omit `lifeTimeVariable`.

`lifeTimeVariable` must be `""` or a valid MATLAB variable name, and must not match any of the strings in `dataVariables`.

`fit` stores `lifeTimeVariable` in the `LifeTimeVariable` property of the model.

`dataVariables` — Feature data variables
`""` (default) | string | string array

Feature data variables, specified as a string or string array. If `data` is a:

- Degradation model, then `dataVariables` must be a string
- Similarity model or covariate survival model, then `dataVariables` must be a string array
- Reliability survival model, then `fit` ignores `dataVariables`

If `data` is:

- A `table` or `timetable`, then the strings in `dataVariables` must match variable names in the table.
- Nontabular, then `dataVariables` must be `""` or contain the same number of strings as there are data columns in `data`. The strings in `dataVariables` must be valid MATLAB variable names.

`fit` stores `dataVariables` in the `DataVariables` property of the model.

`censorVariable` — Censor variable
`""` (default) | string

Censor variable for survival models, specified as a string. The censor variable is a binary variable that indicates which life time measurements in `data` are not end-of-life values. To use censoring, `data` must be a `table` or `timetable`.

If you specify `censorVariable`, the string must match one of the variable names in `data` and must not match any of the strings in `dataVariables` or `lifeTimeVariable`.

`fit` stores `censorVariable` in the `CensorVariable` property of the model.

`encodedVariables` — Encoded variables
`""` (default) | string | string array

Encoded variables for covariate survival models, specified as a string or string array. Encoded variables are usually nonnumeric categorical features that `fit` converts to numeric vectors before fitting. You can also designate logical or numeric values that take values from a small set to be encoded.

The strings in `encodedVariables` must be a subset of the strings in `dataVariables`.

`fit` stores `encodedVariables` in the `EncodedVariables` property of the model.

See Also

Functions
`predictRUL` | `table` | `timetable`

Topics
"Update RUL Prediction as Data Arrives"
"RUL Estimation Using RUL Estimator Models"
"Data Ensembles for Condition Monitoring and Predictive Maintenance"

Introduced in R2018a

frameintervals

Create frame intervals based on frame settings

Syntax

```
intervals = frameintervals(range,framerate,framesize)
intervals = frameintervals(___,Name,Value)
```

Description

`frameintervals` is a function used in code generated by **Diagnostic Feature Designer**.

`intervals = frameintervals(range,framerate,framesize)` creates frame intervals containing frame start and stop times for all frames within the specified range of data, using the specified frame rate and frame size.

For instance, suppose that your full signal starts at 0 and ends at 30 seconds. You specify contiguous one second frames by setting both `framerate` and `framesize` to `1`. Then `range` is equal to `[0 30]` and `intervals` is returned as a table of 30 intervals that starts with the interval `[0 1]` and ends with the interval `[29 30]`.

Code that is generated by **Diagnostic Feature Designer** uses `frameintervals` when performing frame-based member processing.

`intervals = frameintervals(___,Name,Value)` creates frame intervals using one or more name-value pair arguments. For instance, `frameintervals('FrameUnit','days')` returns frame intervals in the units of days. Specify name-value pair arguments after all other input arguments.

Input Arguments

`range` — Data range

numeric vector | duration vector

Data range over which to create frame intervals, specified as a numeric or duration vector with two elements.

`framerate` — Frame rate

numeric | duration

Frame rate, which represents the distance between the starting points of each successive frame, specified as a numeric or duration value. By default, `frameintervals` interprets the units of `framerate` and `framesize`, and sets the units and data type of `intervals`, according to the data type and units of `range` as the table shows.

range	**framerate, framesize**	**Units of framerate, framesize**	**intervals**	**Units of intervals**
numeric	numeric	same as `range`	numeric	same as `range`

range	framerate, framesize	Units of framerate, framesize	intervals	Units of intervals
duration	numeric	seconds	duration	seconds
duration	duration	same as range	duration	same as range
duration	duration	different from range	duration	seconds

framesize — Frame size
numeric | duration

Frame size, which represents the distance between the start point and end point of each successive frame, specified as a numeric or duration value. By default, `frameintervals` interprets the units of `framesize` and `framerate`, and sets the units and data type of `intervals`, according to the data type and units of `range`. For more information, see the table in `framerate`.

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: `frameintervals('FrameUnit','days')`

FrameUnit — Frame units
'seconds' (default) | string

Frame units, specified as the comma-separated pair consisting of `'FrameUnit'` and a string identifying the unit in which to return `intervals`. When `framerate` and `framesize` are numeric but `range` is duration, `'FrameUnit'` also specifies the units of `framerate` and `framesize`.

VariableNames — Variable names
'seconds' (default) | string

Variable names for `intervals`, specified as the comma-separated pair consisting of `'VariableNames'` and a string array with two strings representing the names for start points and end points.

Output Arguments

intervals — Frame intervals
table

Frame intervals, returned as an *nf*-by-2 table, where *nf* is the number of frames. By default, the data type and units of `intervals` depend on the data type and units of `range` and `framerate`. For more information, see `framerate`. The name-value pair argument `framesize` overrides the default units for `intervals.`

See Also
`readFrameIntervals` | `joindata` | **Diagnostic Feature Designer**

Topics
"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a

gearConditionMetrics

Standard metrics for gear condition monitoring

Syntax

```
gearMetrics = gearConditionMetrics(X)
gearMetrics = gearConditionMetrics(T)
gearMetrics = gearConditionMetrics( ___ ,Name,Value)

gearMetrics = gearConditionMetrics(T,sigVar,diffVar,regVar,resVar)
gearMetrics = gearConditionMetrics( ___ ,'SortBy',sortByValue)

[gearMetrics,info] = gearConditionMetrics( ___ )
```

Description

gearMetrics = gearConditionMetrics(X) returns the gear condition monitoring metrics gearMetrics using the vibration data in cell array X. gearConditionMetrics assumes that each cell element in X contains columns of time-synchronous averaged (TSA), difference, regular, and residual signals, in their respective order. If the signals are not in the same order, then use Name,Value pair arguments.

gearMetrics = gearConditionMetrics(T) computes the gear condition monitoring metrics gearMetrics from vibration dataset T. gearConditionMetrics assumes that T contains columns of TSA, difference, regular, and residual signals, in their respective order. If the signals are not in the same order, then use Name,Value pair arguments.

gearMetrics = gearConditionMetrics(___ ,Name,Value) allows you to specify additional parameters using one or more name-value pair arguments.

gearMetrics = gearConditionMetrics(T,sigVar,diffVar,regVar,resVar) computes the gear condition monitoring metrics gearMetrics from vibration dataset T. Use [] or '' to skip a signal in the computation. For instance, if the data set T contains only the TSA and regular signal, use the syntax in the following way.

```
gearMetrics = gearConditionMetrics(T,sigVar,[],regVar,[])
```

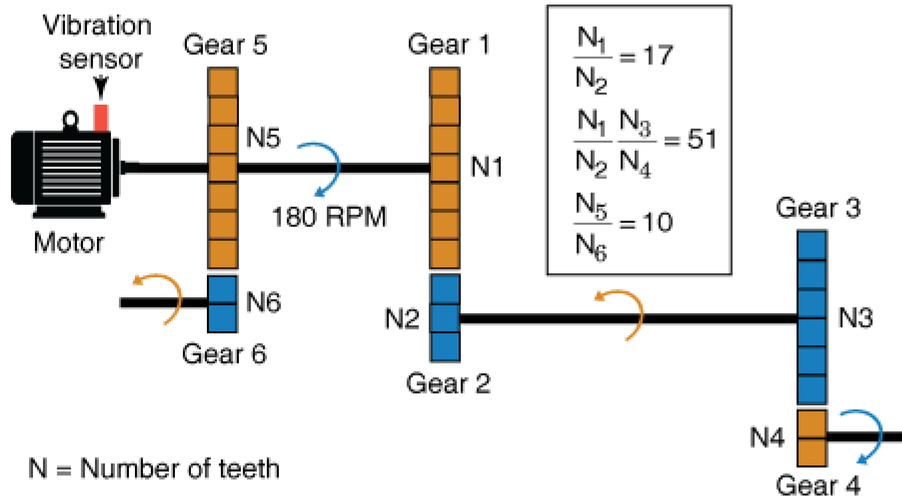
gearMetrics = gearConditionMetrics(___ ,'SortBy',sortByValue) allows you to specify the chronological order of the signal histories using sortByValue. NA4 depends on the chronological order of the vibration data since gearConditionMetrics uses the previous datasets up to the current index to compute the metric.

[gearMetrics,info] = gearConditionMetrics(___) also returns the structure info containing information about the table or fileEnsembleDatastore object variables assigned to various signals.

Examples

Extract Gear Condition Monitoring Metrics from Vibration Signals

Consider a drivetrain with six gears driven by a motor that is fitted with a vibration sensor, as depicted in the figure below. Gear 1 on the motor shaft meshes with gear 2 with a gear ratio of 17:1. The final gear ratio, that is, the ratio between gears 1 and 2 and gears 3 and 4, is 51:1. Gear 5, also on the motor shaft, meshes with gear 6 with a gear ratio of 10:1. The motor is spinning at 180 RPM, and the sampling rate of the vibration sensor is 50 kHz.



Create the dataset.

```
rpm = 180;
fs = 50e3;
t = (0:1/fs:(1/3)-1/fs)'; % sample times
orderList = [17 51];
f = rpm/60*[1 orderList 10];
```

In practice, you would use measured data such as vibration signals obtained from an accelerometer. For this example, generate TSA signal X, which is the simulated data from the vibration sensor mounted on the motor, and then compute the difference, regular, and residual signals. Store the signals in a preallocated table.

```
T = table('Size',[10 4],'VariableTypes',{'cell','cell','cell','cell'},'VariableNames',{'TSA','Di
for k = 1:10
    X = sin(2*pi*f(1)*t) + sin(2*pi*2*f(1)*t) + ... % motor shaft rotation and harmonic
    3*sin(2*pi*f(2)*t) + 3*sin(2*pi*2*f(2)*t) + ... % gear mesh vibration and harmonic for gears
    4*sin(2*pi*f(3)*t) + 4*sin(2*pi*2*f(3)*t) + ... % gear mesh vibration and harmonic for gears
    2*(k/6)*sin(2*pi*10*f(1)*t) + randn(size(t))/5;    % gear mesh vibration for gears 5 and 6 a
  res = tsaresidual(X, fs, rpm, orderList);
  dif = tsadifference(X, fs, rpm, orderList);
  reg = tsaregular(X, fs, rpm, orderList);

  T(k,'TSA') = {X};
  T(k,'Diff') = {dif};
  T(k,'Reg') = {reg};
  T(k,'Res') = {res};
end
T
```

```
T=10×4 table
           TSA                   Diff                  Reg                   Res
```

```
    _________________    _________________    _________________    _________________

    {16666x1 double}    {16666x1 double}    {16666x1 double}    {16666x1 double}
    {16666x1 double}    {16666x1 double}    {16666x1 double}    {16666x1 double}
    {16666x1 double}    {16666x1 double}    {16666x1 double}    {16666x1 double}
    {16666x1 double}    {16666x1 double}    {16666x1 double}    {16666x1 double}
    {16666x1 double}    {16666x1 double}    {16666x1 double}    {16666x1 double}
    {16666x1 double}    {16666x1 double}    {16666x1 double}    {16666x1 double}
    {16666x1 double}    {16666x1 double}    {16666x1 double}    {16666x1 double}
    {16666x1 double}    {16666x1 double}    {16666x1 double}    {16666x1 double}
    {16666x1 double}    {16666x1 double}    {16666x1 double}    {16666x1 double}
    {16666x1 double}    {16666x1 double}    {16666x1 double}    {16666x1 double}
```

T is a 10x4 table, where each element is a cell array.

Compute the gear condition monitoring metrics using the dataset in table T.

```
[gearMetrics1,info1] = gearConditionMetrics(T,'SignalVariable','TSA','DifferenceVariable','Diff',
```

```
gearMetrics1=10×9 table
     RMS      Kurtosis    CrestFactor     FM4       M6A       M8A       FM0      EnergyRatio
    ______    ________    ___________    ______    ______    ______    ______    ___________

    5.1119     2.074        2.4377      2.4633    9.0009     42.31    1.5499      0.060057
    5.1272     2.087        2.4819      1.9331    4.9869    15.634    1.5785       0.10044
    5.1526     2.102        2.4744      1.7084    3.6211    8.8635    1.5881       0.14423
    5.1877    2.1264        2.5443        1.63    3.1749    6.9296    1.6424       0.18889
    5.2385    2.1566        2.5985      1.5861    2.9421    6.0165    1.6937       0.23407
    5.2953    2.1879         2.605      1.5604    2.8046    5.4734    1.7211       0.28052
     5.365    2.2277        2.6551      1.5423    2.7169    5.1619    1.7761       0.32511
    5.4425    2.2574        2.6428      1.5356    2.6796     5.016    1.7945       0.37196
    5.5269    2.2891        2.7112      1.5269    2.6344    4.8502    1.8614       0.41819
    5.6219    2.3214        2.6979      1.5202    2.6015    4.7342    1.8892       0.46377
```

```
info1 = struct with fields:
        SignalVariable: 'TSA'
    DifferenceVariable: 'Diff'
       RegularVariable: 'Reg'
      ResidualVariable: 'Res'
                SortBy: [1x0 char]
```

Observe that the gear metrics are changing due to fault in gear mesh between gears 5 and 6. The NA4 value is highly sensitive to the fault and its propagation as it significantly increases in value over the different data sets.

`info1` contains information about variables that were used to compute the metrics.

Alternatively, you can also compute the metrics using following syntax.

```
[gearMetrics2,info2] = gearConditionMetrics(T,'TSA','Diff','Reg','Res')
```

```
gearMetrics2=10×9 table
     RMS      Kurtosis    CrestFactor     FM4       M6A       M8A       FM0      EnergyRatio
    ______    ________    ___________    ______    ______    ______    ______    ___________
```

```
    5.1119      2.074        2.4377        2.4633    9.0009     42.31    1.5499     0.060057      2
    5.1272      2.087        2.4819        1.9331    4.9869    15.634    1.5785      0.10044      4
    5.1526      2.102        2.4744        1.7084    3.6211    8.8635    1.5881      0.14423      5
    5.1877     2.1264        2.5443          1.63    3.1749    6.9296    1.6424      0.18889      6
    5.2385     2.1566        2.5985        1.5861    2.9421    6.0165    1.6937      0.23407      7
    5.2953     2.1879         2.605        1.5604    2.8046    5.4734    1.7211      0.28052      8
     5.365     2.2277        2.6551        1.5423    2.7169    5.1619    1.7761      0.32511      8
    5.4425     2.2574        2.6428        1.5356    2.6796     5.016    1.7945      0.37196      9
    5.5269     2.2891        2.7112        1.5269    2.6344    4.8502    1.8614      0.41819      9
    5.6219     2.3214        2.6979        1.5202    2.6015    4.7342    1.8892      0.46377      1
```

```
info2 = struct with fields:
        SignalVariable: 'TSA'
    DifferenceVariable: 'Diff'
       RegularVariable: 'Reg'
      ResidualVariable: 'Res'
                SortBy: [1x0 char]
```

Compute Gear Condition Metrics from Ensemble Datastore

Consider `gearData.zip`, a collection of 9 data sets where each file contains separate timetables for the TSA, difference, regular and residual signals.

Extract the compressed files, read the data in the timetables, and create a `fileEnsembleDatastore` object using the timetable data. For more information on creating a file ensemble datastore, see `fileEnsembleDatastore`.

```
unzip gearData.zip;
ens = fileEnsembleDatastore(pwd,'.mat');
% Make sure that the function for reading data is on path
addpath(fullfile(matlabroot,'examples','predmaint','main'))
ens.ReadFcn = @readData;
ens.DataVariables = {'TSA','Diff','Reg','Res'};
ens.SelectedVariables = ens.DataVariables;
```

Compute the gear condition metrics using the data in the ensemble datastore.

```
[gearMetrics,info] = gearConditionMetrics(ens,'SignalVariable','TSA','DifferenceVariable','Diff'
```

```
gearMetrics=9×9 table
     RMS      Kurtosis    CrestFactor     FM4       M6A       M8A       FM0      EnergyRatio
    ______    ________    ___________    ______    ______    ______    ______    ___________

    5.1119     2.0734        2.3417        2.4977    9.3854    45.859    1.4919     0.060189      2
    5.1271      2.086        2.3714        1.9236    4.9222    15.262    1.5155      0.10018      4
    5.1526      2.101        2.3938        1.7199    3.6873    9.1708    1.5398      0.14418      5
    5.1882     2.1247        2.4128        1.6283    3.1667    6.9051    1.5589      0.18951      6
     5.238     2.1572          2.45        1.5816    2.9135    5.8919    1.5994      0.23373      7
    5.2947     2.1888        2.4253        1.5571    2.7877    5.4113    1.5956      0.28007      8
    5.3657      2.226        2.4526        1.5443    2.7251    5.1856    1.6297      0.32562      8
    5.4421     2.2564         2.447        1.5341    2.6718    4.9888    1.6549      0.37177      9
    5.5254     2.2867        2.4349        1.5269    2.6354    4.8572    1.6763      0.41747      9
```

```
info = struct with fields:
        SignalVariable: 'TSA'
    DifferenceVariable: 'Diff'
       RegularVariable: 'Reg'
      ResidualVariable: 'Res'
                SortBy: [1x0 char]
```

The output table contains 9 rows of metrics where each row corresponds to one data set.

```
rmpath(fullfile(matlabroot,'examples','predmaint','main')) % Reset path
```

Input Arguments

X — Vibration dataset

cell array of matrices | cell array of timetables

Vibration dataset, specified as a cell array of matrices or timetables, where each cell contains the signals corresponding to one time in the historical record. Each cell element in `X` contains columns of vibration data representing a combination of TSA, difference, regular, and residual signals.

T — Vibration dataset

timetable | table of vectors | table of tables/timetables | `fileEnsembleDatastore` object

Vibration dataset, specified as a timetable, table of vectors, table of tables/timetables or a `fileEnsembleDatastore` object. Each member (row) of `T` contains the signals corresponding to one time in the historical record. When `T` is a table, each table element contains a signal vector or a table/timetable with a single numeric column variable. The table variables represent TSA, difference, regular, and residual signals.

When `T` is a single timetable, `gearConditionMetrics` interprets it as a single cell of the same timetable. For instance, consider a single timetable `TT`. The command `gearConditionMetrics(TT)` is interpreted as `gearConditionMetrics({T})`.

`sigVar` — TSA signal variable

string | character array

TSA signal variable, specified as a string or character array. `sigVar` is equivalent to the '`SignalVariable`' name-value pair.

`diffVar` — Difference signal variable

string | character array

Difference signal variable, specified as a string or character array. `diffVar` is equivalent to the '`DifferenceVariable`' name-value pair.

`regVar` — Regular signal variable

string | character array

Regular signal variable, specified as a string or character array. `regVar` is equivalent to the '`RegularVariable`' name-value pair.

`resVar` — Residual signal variable

string | character array

Residual signal variable, specified as a string or character array. `resVar` is equivalent to the '`ResidualVariable`' name-value pair.

`sortByValue` — Value of `'SortBy'`
string

Value of `'SortBy'`, specified as a string. For more information, see '`SortBy`'.

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: ...,`'SortBy','FaultCode'`

`SignalVariable` — TSA signal variable
first column of data (default) | string | character array

TSA signal variable, specified as the comma-separated pair consisting of '`SignalVariable`' and a string or character array.

'`SignalVariable`' must be a valid table variable name when the dataset is specified as a table or timetable. When the data is specified as a cell array of matrices, the values `'Var1','Var2',...` can be used to refer to the data columns. If '`SignalVariable`' is not specified, `gearConditionMetrics` assumes that the first data column contains the TSA signal.

The `RMS`, `Kurtosis`, `Crest Factor`, and `FM0` metrics require the TSA signal for computation. If the TSA signal is not available, `gearConditionMetrics` returns `NaN` for these metrics.

`DifferenceVariable` — Difference signal variable
second column of data (default) | string | character array

Difference signal variable, specified as the comma-separated pair consisting of '`DifferenceVariable`' and a string or character array.

'`DifferenceVariable`' must be a valid table variable name when the dataset is specified as a table or timetable. When the data is specified as a cell array of matrices, the values `'Var1','Var2',...` can be used to refer to the data columns. If '`DifferenceVariable`' is not specified, `gearConditionMetrics` assumes that the second data column contains the difference signal.

The `FM4`, `M6A`, `M8A` and `Energy Ratio` metrics require the difference signal for computation. If the difference signal is not available, `gearConditionMetrics` returns `NaN` for these metrics.

For more information on difference signals, see `tsadifference`.

`RegularVariable` — Regular signal variable
third column of data (default) | string | character array

Regular signal variable, specified as the comma-separated pair consisting of '`RegularVariable`' and a string or character array.

'`RegularVariable`' must be a valid table variable name when the dataset is specified as a table or timetable. When the data is specified as a cell array of matrices, the values `'Var1','Var2',...` can be used to refer to the data columns. If '`RegularVariable`' is not specified, `gearConditionMetrics` assumes that the third data column contains the regular signal.

The FM0 and Energy Ratio metrics require the regular signal for computation. If the regular signal is not available, gearConditionMetrics returns NaN for these metrics.

For more information on regular signals, see tsaregular.

ResidualVariable — Residual signal variable
fourth column of data (default) | string | character array

Residual signal variable, specified as the comma-separated pair consisting of 'ResidualVariable' and a string or character array.

'ResidualVariable' must be a valid table variable name when the dataset is specified as a table or timetable. When the data is specified as a cell array of matrices, the values 'Var1','Var2',... can be used to refer to the data columns. If 'ResidualVariable' is not specified, gearConditionMetrics assumes that the fourth data column contains the residual signal.

The NA4 metric requires the residual signal for computation. If the residual signal is not available, gearConditionMetrics returns NaN for NA4.

For more information on residual signals, see tsaresidual.

SortBy — Signal ordering variable
'' (default) | string

Signal ordering variable, specified as the comma-separated pair consisting of 'SortBy' and a string. Use 'SortBy' to order the signal histories in ascending order only when the input dataset T is a table of vectors or table of tables/timetables. gearConditionMetrics sorts the rows in ascending order with respect to 'SortBy' before computing gearMetrics. The value in the specified table column must be a valid input to 'SortBy. For more information, see sort.

If 'SortBy' is not specified or if the dataset is a cell array or fileEnsembleDatastore, then the signal histories are assumed to be in ascending order, that is, older data at the top.

Output Arguments

gearMetrics — Gear condition monitoring metrics
table

Gear condition monitoring metrics, returned as a table, where each row corresponds to its respective member in X or T. gearConditionMetrics returns the following condition monitoring metrics:

Computed from TSA Signal

- Root-Mean Square (RMS) — Indicates the general condition of the gearbox in later stages of degradation. RMS is sensitive to gearbox load and speed changes.
- Kurtosis — Fourth order normalized moment of the signal that indicates major peaks in the amplitude distribution. A signal consisting exclusively of Gaussian distributed noise has an approximate kurtosis value of 3. Kurtosis values are higher for damaged gear trains due to sharp peaks in the amplitude distribution of the signal.
- Crest Factor (CF) — Ratio of signal peak value to RMS value that indicates early signs of damage, especially where vibration signals exhibit impulsive traits.

Computed from Difference Signal

- `FM4` — Describes how peaked or flat the difference signal amplitude is. `FM4` is normalized by the square of the variance, and detects faults isolated to only a finite number of teeth in a gear mesh.
- `M6A` — Describes how peaked or flat the difference signal amplitude is. `M6A` is normalized by the cube of the variance, and indicates surface damage on the rotating machine components.
- `M8A` — An improved version of the `M6A` indicator. `M8A` is normalized by the fourth power of the variance.

Computed from a Mix of Signals

- `FM0` — Compares ratio of peak value of TSA signal to energy of regular signal. `FM0` identifies major anomalies, such as tooth breakage or heavy wear, in the meshing pattern of a gear.
- `Energy Ratio (ER)` — Ratio between energy of the difference signal and the energy of the regular meshing component. `Energy Ratio` indicates heavy wear, where multiple teeth on the gear are damaged.

Computed from a Set of Residual Signals

- `NA4` — An improved version of the `FM4` indicator. `NA4` indicates the onset of damage and continues to react to the damage as it spreads and increases in magnitude.

`gearConditionMetrics` returns `NaN` for metrics when their respective signals are not available for computation. For more information about these metrics, see “Algorithms” on page 1-107.

`info` — Signal assignment information

structure

Signal assignment information, returned as a structure with the following fields:

- `DifferenceVariable` — Difference variable name
- `RegularVariable` — Regular variable name
- `ResidualVariable` — Residual variable name
- `SignalVariable` — TSA signal variable name
- `SortBy` — Signal ordering variable name

Algorithms

Root Mean Square (RMS)

The root mean square (RMS) of the TSA signal is computed using the `rms` command. For a TSA signal x, RMS is computed as,

$$\mathrm{RMS}(x) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}.$$

Here, N is the number of data samples.

RMS is usually a good indicator of the overall condition of gearboxes, but not a good indicator of incipient tooth failure. It is also useful to detect unbalanced rotating elements. RMS of a standard normal distribution is 1.

For more information, see `rms`.

Kurtosis

Kurtosis is a measure of how outlier-prone a distribution is. The kurtosis of a standard normal distribution is 3. Distributions that are more outlier-prone have kurtosis values greater than 3; distributions that are less outlier-prone have kurtosis values less than 3.

`gearConditionMetrics` computes the kurtosis value of the TSA signal using the `kurtosis` command. The kurtosis of a sequence is defined as,

$$\mathrm{Kurtosis}(x) = \frac{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^4}{\left[\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2\right]^2}.$$

Here, $\bar{x}$ is the mean of the TSA signal x.

For more information, see `kurtosis`.

Crest Factor (CF)

`Crest Factor` is the ratio of the positive peak value of the input signal x to the `RMS` value. `gearConditionMetrics` computes the crest factor of the TSA signal using the `peak2rms` command.

The crest factor of a sequence is defined as,

$$\mathrm{CF}(x) = \frac{\mathrm{P}(x)}{\mathrm{RMS}(x)}.$$

Here, $P(x)$ is the peak value of the TSA signal.

The crest factor indicates the relative size of peaks to the effective value of the signal. It is a good indicator of gear damage in its early stages, where vibration signals exhibit impulsive traits.

FM4

The `FM4` indicator is used to detect faults isolated to only a limited number of teeth in a gear mesh. `FM4` is defined as the normalized kurtosis of the difference signal [4]. `FM4` of a standard normal distribution is 3.

`FM4` is computed as,

$$\mathrm{FM4}(d) = \frac{\frac{1}{N}\sum_{i=1}^{N}\left(d_i - \bar{d}\right)^4}{\left[\frac{1}{N}\sum_{i=1}^{N}\left(d_i - \bar{d}\right)^2\right]^2}$$

where, $\bar{d}$ is the mean of the difference signal d.

M6A

The `M6A` indicator is used to detect surface damage on machinery components. `M6A` employs the same theory as the `FM4` metric, but uses the sixth moment of the difference signal normalized by the cube

of the variance. `M6A` of a standard normal distribution is 15. Hence, `M6A` is expected to be more sensitive to peaks in the difference signal. `gearConditionMetrics` uses the `moment` command to compute `M6A`.

`M6A` is computed as,

$$\mathrm{M6A}(d) = \frac{\frac{1}{N}\sum_{i=1}^{N}\left(d_i - \bar{d}\right)^6}{\left[\frac{1}{N}\sum_{i=1}^{N}\left(d_i - \bar{d}\right)^2\right]^3}$$

where, $\bar{d}$ is the mean of the difference signal d.

M8A

The `M8A` indicator is an improved version of `M6A`. It is expected to be more sensitive to peaks in the difference signal since `M6A` is normalized by the fourth power of the variance. `M8A` of a standard normal distribution is 105. It is computed as,

$$\mathrm{M8A}(d) = \frac{\frac{1}{N}\sum_{i=1}^{N}\left(d_i - \bar{d}\right)^8}{\left[\frac{1}{N}\sum_{i=1}^{N}\left(d_i - \bar{d}\right)^2\right]^4}.$$

FM0

`FM0` is useful in detecting major anomalies in the gear meshing pattern. It does so by comparing the maximum peak-to-peak amplitude of the TSA signal to the sum of the amplitudes of the meshing frequencies and their harmonics. `gearConditionMetrics` uses a combination of `peak2peak` and `fft` commands to compute the `FM0` metric.

`FM0` is computed as,

$$\mathrm{FM0}(x) = \frac{PP(x)}{\sum_{i=1}^{N} A(i)}$$

where, *PP(x)* is the peak-to-peak values of the TSA signal. A contains the frequency-domain amplitudes at the mesh frequencies and their harmonics, which represents the energy of the regular signal.

A is computed as,

$$A = \frac{fft(R(t))}{N}$$

where, *R(t)* is the regular signal.

Energy Ratio (ER)

`Energy Ratio` is defined as the ratio of the standard deviations of the difference and regular signals [1]. It is useful as an indicator of heavy uniform wear, where multiple teeth on the gear are damaged.

`Energy Ratio` is computed as,

$$\mathrm{ER}(x) = \frac{\sigma(d)}{\sigma(R)}$$

where, d and R represent the difference and regular signals, respectively.

NA4

`NA4` is an improved version of the `FM4` indicator [3]. `NA4` indicates the onset of damage and continues to react to the damage as it spreads and increases in magnitude.

`NA4` is computed as,

$$\mathrm{NA4}(r,k) = \frac{\frac{1}{N}\sum_{i=1}^{N}(r_{ik} - \bar{r}_k)^4}{\left[\frac{1}{k}\sum_{j=1}^{k}\frac{1}{N}\sum_{i=1}^{N}\left(r_{ij} - \bar{r}_j\right)^2\right]^2}$$

where the normalization is across all vibration data sets up to the current time k using the running average of variances of residual signals.

References

[1] Keller, Jonathan A., and P. Grabill. "Vibration monitoring of UH-60A main transmission planetary carrier fault." *Annual Forum Proceedings-American Helicopter Society*. Vol. 59. No. 2. American Helicopter Society, Inc, 2003.

[2] Večeř, P., Marcel Kreidl, and R. Šmíd. "Condition indicators for gearbox condition monitoring systems." *Acta Polytechnica* pages 35-43, 45.6 (2005).

[3] Zakrajsek, James J., Dennis P. Townsend, and Harry J. Decker. "An analysis of gear fault detection methods as applied to pitting fatigue failure data." *Technical Memorandum 105950*. No. NASA-E-7470. NASA, 1993.

[4] Zakrajsek, James J. "An investigation of gear mesh failure prediction techniques." MS Thesis-Cleveland State University, 1989.

Extended Capabilities

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

Usage notes and limitations:

- Data stored in `fileEnsembleDatastore` and `workspaceEnsemble` objects, as well as data in the form of a tall array are not supported.
- Position arguments are not supported. For instance, the syntax `M = gearConditionMetrics(T,sigvar,difvar,regvar,resvar)` is not supported.
- Sorting a table by a table column with values wrapped in cells is not supported. For instance, the values of the column `Order` must be scalar for the syntax `M = gearConditionMetrics(T,'SortBy','Order')`.

See Also
`tsa` | `tsaresidual` | `tsaregular` | `tsadifference`

Topics
"Condition Indicators for Gear Condition Monitoring"

Introduced in R2019a

gearMeshFaultBands

Construct frequency bands around the characteristic fault frequencies of meshing gears for spectral feature extraction

Syntax

```
FB = gearMeshFaultBands(FR,Ni,No)
FB = gearMeshFaultBands( ___ ,Name,Value)
[FB,info] = gearMeshFaultBands( ___ )

gearMeshFaultBands( ___ )
```

Description

`FB = gearMeshFaultBands(FR,Ni,No)` generates characteristic fault frequency bands `FB` of gear mesh using the rotational speed of the input gear `FR` and the number of teeth on the input `Ni` and output gear `No` respectively. The values in `FB` have the same implicit units as `FR`

`FB = gearMeshFaultBands( ___ ,Name,Value)` allows you to specify additional parameters using one or more name-value pair arguments.

`[FB,info] = gearMeshFaultBands( ___ )` also returns the structure `info` containing information about the generated fault frequency bands `FB`.

`gearMeshFaultBands( ___ )` with no output arguments plots a bar chart of the generated fault frequency bands `FB`.

Examples

Frequency Bands of Pinion and Gear Mesh

For this example, consider a simple gear set with an 8-toothed pinion on the input shaft meshing with a 42-toothed spur gear on the output shaft. Assume that the input shaft is spinning at 20 rpm. Construct the gear mesh frequency bands using the physical characteristics of the gear set.

```
Ni = 8;
No = 42;
FR = 20;
[FB,info] = gearMeshFaultBands(FR,Ni,No)
```

```
FB = 5×2

   19.0000   21.0000
    2.8095    4.8095
   79.0000   81.0000
  159.0000  161.0000
  159.0000  161.0000


info = struct with fields:
        Centers: [20 3.8095 80 160 160]
```

```
        Labels: ["1Fi"    "1Fo"    "1Fa"    "1Fm"    "1Fm"]
   FaultGroups: [1 2 3 4 5]
```

FB is a 5x2 array which includes the primary frequencies 1Fi, 1Fo, 1Fa and 1Fm respectively. The structure info contains the center frequencies and labels of each frequency range in FB.

Frequency Bands and Spectral Metrics of Gear Train

For this example, consider a simple gear set with an 8-toothed pinion on the input shaft meshing with a 42-toothed spur gear on the output shaft. Assume that the input shaft is driven at 20 Hz. The dataset motorSignal.mat contains vibration data for the gear mesh sampled at 1500 Hz.

First, construct the gear mesh frequency bands using the physical characteristics of the gear set. Construct the frequency bands with the first 3 sidebands.

```
Ni = 8;
No = 42;
FR = 20;
FB = gearMeshFaultBands(FR,Ni,No,'Sidebands',1:3)
```

```
FB = 15×2

   19.0000   21.0000
    2.8095    4.8095
   79.0000   81.0000
   99.0000  101.0000
  119.0000  121.0000
  139.0000  141.0000
  179.0000  181.0000
  199.0000  201.0000
  219.0000  221.0000
  147.5714  149.5714
      ⋮
```

FB is a 15x2 array which includes the primary frequencies and their sidebands.

Load the vibration data and compute PSD and frequency grid using pspectrum. Use a frequency resolution of 0.5.

```
load('motorSignal.mat','C');
fs = 1500;
[psd,freqGrid] = pspectrum(C,fs,'FrequencyResolution',0.5);
```

Now, use the frequency bands and PSD data to compute the spectral metrics.

```
spectralMetrics = faultBandMetrics(psd,freqGrid,FB)
```

spectralMetrics=*1×46 table*

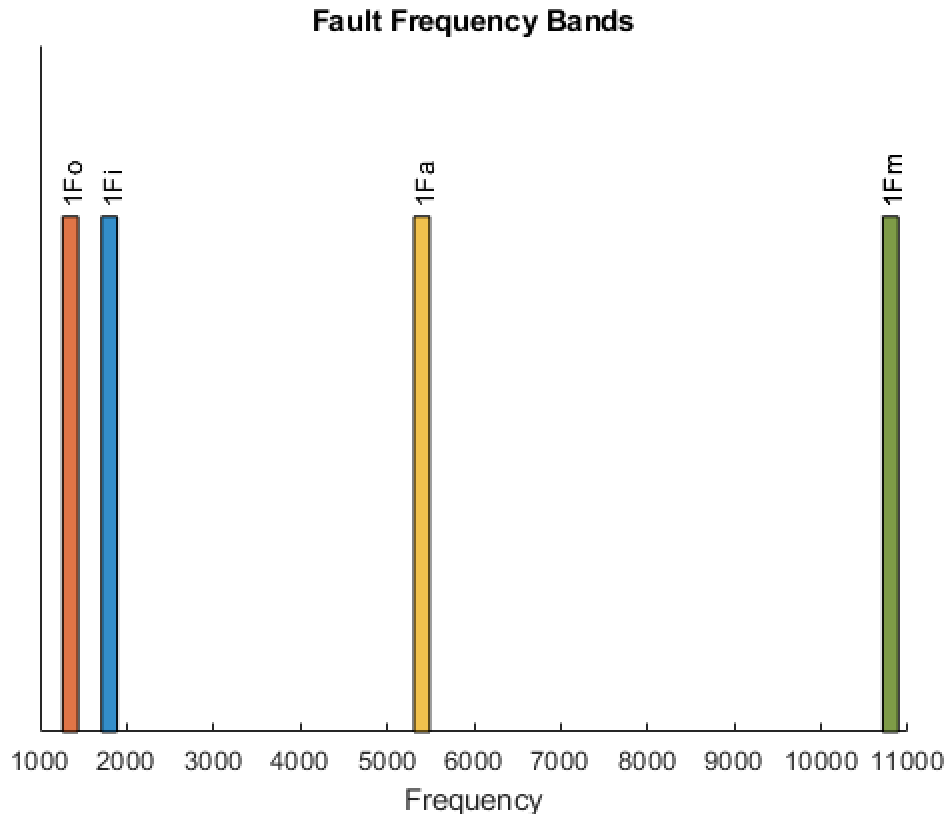
PeakAmplitude1	PeakFrequency1	BandPower1	PeakAmplitude2	PeakFrequency2	BandPo
0.0054125	19	0.0051216	0.55167	4.25	0.418

spectralMetrics is a 1x46 table with peak amplitude, peak frequency and band power calculated for each frequency range in FB. The last column in spectralMetrics is the total band power, computed across all 15 frequencies in FB.

Visualize Frequency Bands for Pinion and Gear Set

For this example, consider a simple pinion and gear set with an input shaft speed of 1800 rpm. Considering that the pinion on the input shaft has 6 teeth and the gear on the output shaft has 8 teeth, visualize the frequency bands for the gear mesh.

```
FR = 1800;
Ni = 6;
No = 8;
gearMeshFaultBands(FR,Ni,No)
```



From the plot, observe the following:

- Output shaft defect frequency, 1Fo at 1350 Hz
- Input shaft defect frequency, 1Fi at 1800 Hz
- Assembly phase defect frequency, 1Fa at 5400 Hz
- Gear mesh defect frequency, 1Fm at 10800 Hz

Input Arguments

FR — Rotational speed of the input gear
positive scalar

Rotational speed of the input gear, specified as a positive scalar. `FR` is the fundamental frequency around which `gearMeshFaultBands` generates the fault frequency bands. Specify `FR` either in Hertz or revolutions per minute.

`Ni` — Number of teeth on the input gear
positive integer

Number of teeth on the input gear, specified as a positive integer.

`No` — Number of teeth on the output gear
positive integer

Number of teeth on the output gear, specified as a positive integer.

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: `...,'Harmonics',[1,3,5]`

`Harmonics` — Harmonics of the fundamental frequency to be included
`1` (default) | vector of positive integers

Harmonics of the fundamental frequency to be included, specified as the comma-separated pair consisting of '`Harmonics`' and a vector of positive integers. The default value is 1. Specify '`Harmonics`' when you want to construct the frequency bands with more harmonics of the fundamental frequency.

`Sidebands` — Sidebands around the fundamental frequency and its harmonics to be included
`0` (default) | vector of nonnegative integers

Sidebands around the fundamental frequency and its harmonics to be included, specified as the comma-separated pair consisting of '`Sidebands`' and a vector of nonnegative integers. The default value is 0. Specify '`Sidebands`' when you want to construct the frequency bands with sidebands around the fundamental frequency and its harmonics.

`Width` — Width of the frequency bands centered at the nominal fault frequencies
`10` percent of the fundamental frequency (default) | positive scalar

Width of the frequency bands centered at the nominal fault frequencies, specified as the comma-separated pair consisting of '`Width`' and a positive scalar. The default value is 10 percent of the fundamental frequency. Avoid specifying '`Width`' with a large value so that the fault bands do not overlap.

`Domain` — Units of the fault band frequencies
`'frequency'` (default) | `'order'`

Units of the fault band frequencies, specified as the comma-separated pair consisting of 'Domain' and either 'frequency' or 'order'. Select:

- 'frequency' if you want FB to be returned in the same units as FR.
- 'order' if you want FB to be returned as number of rotations relative to FR.

Folding — Logical value specifying whether negative nominal fault frequencies have to be folded about the frequency origin

false (default) | true

Logical value specifying whether negative nominal fault frequencies have to be folded about the frequency origin, specified as the comma-separated pair consisting of 'Folding' and either true or false. If you set 'Folding' to true, then faultBands folds the negative nominal fault frequencies about the frequency origin by taking their absolute values such that the folded fault bands always fall in the positive frequency intervals. The folded fault bands are computed as $\left[\max\left(0,\ |F| - \frac{W}{2}\right),\ |F| + \frac{W}{2}\right]$, where W is the 'Width' name-value pair and F is one of the nominal fault frequencies.

Output Arguments

FB — Fault frequency bands

Nx2 array

Fault frequency bands, returned as an Nx2 array, where N is the number of fault frequencies. FB is returned in the same units as FR, in either Hertz or orders. Use the generated fault frequency bands to extract spectral metrics using faultBandMetrics. The generated fault bands, $\left[F - \frac{W}{2},\ F + \frac{W}{2}\right]$, are centered at the characteristic defect frequencies and their harmonics and sidebands for:

- Input shaft defect frequency, Fi
- Output shaft defect frequency, Fo
- Gear mesh defect frequency, Fm
- Assembly phase pass defect frequency, Fa

When you specify the sidebands, gearMeshFaultBands computes the sidebands with respect to the input and output shaft defect frequencies:

- Fault frequency bands for input gear defects with its harmonics and the first sideband at Fi
- Fault frequency bands for output gear defects with its harmonics and the first sideband at Fo

gearMeshFaultBands truncates negative fault frequency bands automatically and generates a warning message.

The value W is the width of the frequency bands, which you can specify using the 'Width' name-value pair.

info — Information about the fault frequency bands

structure

Information about the fault frequency bands in FB, returned as a structure with the following fields:

- `Centers` — Center fault frequencies
- `Labels` — Labels describing each frequency
- `FaultGroups` — Fault group numbers equal to the number of frequencies

Algorithms

`gearMeshFaultBands` computes the different characteristic fault frequencies as follows:

- Input shaft defect frequency, $F_i = FR$
- Output shaft defect frequency, $F_o = \frac{Ni}{No}FR$
- Gear mesh defect frequency, $F_m = NiFR = NoF_o$
- Assembly phase pass defect frequency,$F_a = \frac{F_m}{\mathrm{gcd}(Ni, No)}$

References

[1] Lang, George Fox. "S&V geometry 101." *Sound and Vibration* 33 (1999): 16-26.

See Also

`faultBandMetrics` | `faultBands` | `bearingFaultBands` | `gearConditionMetrics`

Topics

"Motor Current Signature Analysis for Gear Train Fault Detection"

Introduced in R2019b

generateSimulationEnsemble

Generate ensemble data by running a Simulink model

Syntax

```
[status,E] = generateSimulationEnsemble(simin)
[status,E] = generateSimulationEnsemble(simin,location)
[status,E] = generateSimulationEnsemble(simin,location,Name,Value)
```

Description

`[status,E] = generateSimulationEnsemble(simin)` generates data for a simulation ensemble by running the Simulink® model specified by `simin`. This input argument is a vector of `Simulink.SimulationInput` objects that also specifies other parameters to change from simulation to simulation to generate the ensemble. The function writes the simulation data log files to the current folder. Each file contains the corresponding `Simulink.SimulationInput` object and all the variables that the model is configured to log for the simulation. The output arguments indicate whether any simulations generate errors and return any such errors. Use `simulationEnsembleDatastore` to create an ensemble datastore for interacting with the simulated data.

For general information about data ensembles, see “Data Ensembles for Condition Monitoring and Predictive Maintenance”.

`[status,E] = generateSimulationEnsemble(simin,location)` also specifies a path to a location at which to store the simulation results.

`[status,E] = generateSimulationEnsemble(simin,location,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Generate Ensemble of Fault Data

Generate a simulation ensemble datastore of data representing a machine operating under fault conditions by simulating a Simulink® model of the machine while varying a fault parameter.

Load the Simulink model. This model is a simplified version of the gear-box model described in “Using Simulink to Generate Fault Data”. For this example, only one fault mode is modeled, a gear-tooth fault.

```
mdl = 'TransmissionCasingSimplified';
open_system(mdl)
```

The gear-tooth fault is modeled as a disturbance in the `Gear Tooth fault` subsystem. The magnitude of the disturbance is controlled by the model variable `ToothFaultGain`, where `ToothFaultGain = 0` corresponds to no gear-tooth fault (healthy operation). To generate the ensemble of fault data, you use `generateSimulationEnsemble` to simulate the model at different values of `ToothFaultGain`, ranging from -2 to zero. This function uses an array of

`Simulink.SimulationInput` objects to configure the Simulink model for each member in the ensemble. Each simulation generates a separate member of the ensemble in its own data file. Create such an array, and use `setVariable` to assign a tooth-fault gain value for each run.

```
toothFaultValues  = -2:0.5:0; % 5 ToothFaultGain values

for ct = numel(toothFaultValues):-1:1
    simin(ct) = Simulink.SimulationInput(mdl);
    simin(ct) = setVariable(simin(ct),'ToothFaultGain',toothFaultValues(ct));
end
```

For this example, the model is already configured to log certain signal values, `Vibration` and `Tacho` (see “Export Signal Data Using Signal Logging” (Simulink)). `generateSimulationEnsemble` further configures the model to:

- Save logged data to files in the folder you specify.
- Use the `timetable` format for signal logging.
- Store each `Simulink.SimulationInput` object in the saved file with the corresponding logged data.

Specify a location for the generated data. For this example, save the data to a folder called `Data` within your current folder. The indicator `status` is 1 (true) if all the simulations complete without error.

```
mkdir Data
location = fullfile(pwd,'Data');
[status,E] = generateSimulationEnsemble(simin,location);
```

```
[01-Sep-2021 12:59:27] Running simulations...
[01-Sep-2021 12:59:59] Completed 1 of 5 simulation runs
[01-Sep-2021 13:00:29] Completed 2 of 5 simulation runs
[01-Sep-2021 13:00:54] Completed 3 of 5 simulation runs
[01-Sep-2021 13:01:27] Completed 4 of 5 simulation runs
[01-Sep-2021 13:01:44] Completed 5 of 5 simulation runs
```

Inside the `Data` folder, examine one of the files. Each file is a MAT-file containing the following MATLAB® variables:

- `SimulationInput` — The `Simulink.SimulationInput` object that was used to configure the model for generating the data in the file. You can use this to extract information about the conditions (such as faulty or healthy) under which this simulation was run.
- `logsout` — A `Dataset` object containing all the data that the Simulink model is configured to log.
- `PMSignalLogName` — The name of the variable that contains the logged data (`'logsout'` in this example). The `simulationEnsembleDatastore` command uses this name to parse the data in the file.
- `SimulationMetadata` — Other information about the simulation that generated the data logged in the file.

Now you can create the simulation ensemble datastore using the generated data. The resulting `simulationEnsembleDatastore` object points to the generated data. The object lists the data variables in the ensemble, and by default all the variables are selected for reading. Examine the `DataVariables` and `SelectedVariables` properties of the ensemble to confirm these designations.

```
ensemble = simulationEnsembleDatastore(location)
```

```
ensemble = 
  simulationEnsembleDatastore with properties:

           DataVariables: [4x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: [0x0 string]
       SelectedVariables: [4x1 string]
                ReadSize: 1
              NumMembers: 5
          LastMemberRead: [0x0 string]
                   Files: [5x1 string]
```

```
ensemble.DataVariables
```

```
ans = 4x1 string
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
```

```
ensemble.SelectedVariables
```

```
ans = 4x1 string
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
```

You can now use `ensemble` to read and analyze the generated data in the ensemble datastore. See `simulationEnsembleDatastore` for more information.

Input Arguments

`simin` — Simulation configurations

vector of `Simulink.SimulationInput` objects

Simulation configurations, specified as a vector of `Simulink.SimulationInput` objects. The simulation configurations specify parameters for each generated member of the ensemble, such as:

- Simulink model to run
- Values of model variables
- Block parameters
- Model initial state

Thus, for example, you can create a vector of `Simulink.SimulationInput` objects in which all simulation configurations are identical except for the parameters that model the presence and severity of faults in your system. You can then use the vector to generate an ensemble of simulated data representing a range of healthy and faulty operating conditions.

`location` — Folder path

pwd (default) | string | character vector

Folder path at which to store simulation data, specified as a string or a character vector. If you do not provide `location`, the function uses the current folder (the path returned by `pwd`).

In the specified folder, the function writes one MAT-file per simulation. Each file includes the following variables:

- `SimulationInput` — The `Simulink.SimulationInput` object that was used to configure the model for generating the data in this file. You can use this object to extract information about the conditions (such as faulty or healthy) under which this simulation was run.
- `SimulationMetadata` — Other information about the simulation that generated the logged data in the file.
- A `Dataset` object containing all the signal and state data that the Simulink model is configured to log. By default, this variable is called `logsout`, but the name is configurable in the model.
- `PMSignalLogName` — The name of the variable that contains the logged data (`'logsout'` by default). The `simulationEnsembleDatastore` command uses this name to parse the data in the file.

For more information about data logging, see “Export Signal Data Using Signal Logging” (Simulink).

Example: `pwd + "\simResults"`

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: `'UseParallel',true`

`UseParallel` — Whether to run simulations in parallel
`false` (default) | `true`

Whether to run simulations in parallel, specified as the comma-separated pair consisting of `'UseParallel'` and:

- `false` — Do not run simulations in parallel.
- `true` — Use a parallel pool to run multiple simulations in parallel (requires Parallel Computing Toolbox).

`ShowProgress` — Whether to display simulation progress
`true` (default) | `false`

Whether to display simulation progress in the MATLAB command window, specified as the comma-separated pair consisting of `'ShowProgress'` and:

- `true` — Display a simulation progress line each time an individual simulation run completes.
- `false` — Do not display simulation progress.

Output Arguments

`status` — Simulation error status
logical

Simulation error status, returned as a logical value:

- 1 (true) if all simulations run to completion without error
- 0 (false) otherwise

E — Simulation errors
structure array

Simulation errors, returned as a structure array with fields:

- `'SimulationInput'` — `Simulink.SimulationInput` for the simulation run that generated the error
- `'ErrorDiagnostic'` — String containing the error

Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the `'UseParallel'` option to true.

See Also

`simulationEnsembleDatastore` | `Simulink.SimulationInput`

Topics
"Data Ensembles for Condition Monitoring and Predictive Maintenance"

Introduced in R2018a

joindata

Merge two frame tables using an outer join

Syntax

```
table12 = joindata(table1,table2)
table12 = joindata(table1,table2,'Keys',keys)
```

Description

`joindata` is a function used in code generated by **Diagnostic Feature Designer**.

`table12 = joindata(table1,table2)` merges two tables using an outer join with the first two columns as the primary keys to merge. In general, an outer join combines table rows where the key variables have matching values, while also retaining rows where key variables from one input table have no matches in the other input table (see `outerjoin`). `joindata` joins two frame tables. The first two columns of both frame tables contain the segment start and segment end points. The other columns in the tables contain data associated with the frame. The data column names must be unique, that is, data columns in `table2` must not have the same name as the data columns in `table1`.

Code that is generated by **Diagnostic Feature Designer** uses `joindata` when performing frame-based ensemble statistics processing. In the code, `table1` contains existing frame data and `table2` contains newly computed frame results.

`table12 = joindata(table1,table2,'Keys',keys)` uses the columns with the names specified in `keys` as the primary keys for the merge. For example, `'Keys',["TimeStart","TimeEnd"]` specifies that `joindata` use the columns named `"TimeStart"` and `"TimeEnd"` rather than automatically using the first two columns for primary keys.

Examples

Merge Frame Data

Merge two overlapping frame tables.

Create `table1`, a 4-by-3 table that contains values for variable `Var1` in four successive 5-second frames.

```
table1 = table(seconds(0:5:15)', seconds(5:5:20)', [3;4;5;6], ...
    'VariableNames', ["TimeStart", "TimeEnd", "Var1"])
```

```
table1=4×3 table
    TimeStart    TimeEnd    Var1
    _________    _______    ____

      0 sec       5 sec      3
      5 sec      10 sec      4
     10 sec      15 sec      5
```

```
    15 sec      20 sec      6
```

Create `table2`, also a 4-by-3 table, that overlaps the frames in `table1`. `table2` contains the values for `Var2`.

```
table2 = table(seconds(5:5:20)', seconds(10:5:25)', [1;2;3;4], ...
    'VariableNames', ["TimeStart", "TimeEnd", "Var2"])
```

```
table2=4×3 table
    TimeStart    TimeEnd    Var2
    _________    _______    ____

      5 sec      10 sec      1
     10 sec      15 sec      2
     15 sec      20 sec      3
     20 sec      25 sec      4
```

Merge the two tables using `"TimeStart"` and `"TimeEnd"` as the merge keys.

```
table12 = joindata(table1, table2, 'Keys', ["TimeStart", "TimeEnd"])
```

```
table12=5×4 table
    TimeStart    TimeEnd    Var1    Var2
    _________    _______    ____    ____

      0 sec       5 sec       3     NaN
      5 sec      10 sec       4       1
     10 sec      15 sec       5       2
     15 sec      20 sec       6       3
     20 sec      25 sec     NaN       4
```

`table12` is a 5-by-4 table that contains the values for `Var1` and `Var2` for each frame. Missing values are represented by `NaN`.

Input Arguments

table1 — First frame table to merge

table

First frame table to merge, specified as a table with the first two columns representing the segment start and stop points, and the remaining columns containing the corresponding data.

table2 — Second frame table to merge

table

Second frame table to merge, specified as a table with the first two columns representing the segment start and stop point, and the remaining columns containing the corresponding data. Data column names must not match any data column names in `table1`.

keys — Primary keys

string array | cell array

Primary keys for table merge, specified as the comma-separated pair containing `'Keys'` and either a string array with two strings or a cell of two character arrays.

Output Arguments

`table12` — Merged frame data
table

Merged frame data, returned as a table.

See Also

`frameintervals` | `readFrameIntervals` | **Diagnostic Feature Designer**

Topics
"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a

loadRULModelForCoder

Load and reconstruct RUL model from file for use in code generation

Syntax

```
mdl = loadRULModelForCoder(filename)
```

Description

`mdl = loadRULModelForCoder(filename)` reconstructs an RUL model object from properties stored in `filname`, which is a file created using `saveRULModelForCoder`. Use `loadRULModelForCoder` in an entry-point function for code generation to reconstruct the model at compile time. See "Generate Code for Predicting Remaining Useful Life" for more information.

Input Arguments

`filename` — File containing saved model
character vector | string

File containing saved model, specified as a character vector or string. `filename` is a file created using `saveRULModelForCoder`. You can specify a full or relative path in `filename`.

Output Arguments

`mdl` — RUL model
RUL model object

RUL model loaded from file, returned as a `linearDegradationModel`, an `exponentialDegradationModel`, a `reliabilitySurvivalModel`, or a `covariateSurvivalModel` object. The RUL model has the same type and property values as the model used to create `filename`.

Tips

- `loadRULModelForCoder` loads the model at compile time, not at run time. Therefore, any changes you make to the MAT file after compiling are not available at run time. To update the state of a degradation RUL model at run time, use `restoreState`.

Extended Capabilities

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

See Also

`saveRULModelForCoder` | `restoreState` | `linearDegradationModel` | `exponentialDegradationModel` | `reliabilitySurvivalModel` | `covariateSurvivalModel`

Topics
"Generate Code for Predicting Remaining Useful Life"

Introduced in R2021a

lyapunovExponent

Characterize the rate of separation of infinitesimally close trajectories

Syntax

```
lyapExp = lyapunovExponent(X,fs)
lyapExp = lyapunovExponent(X,fs,lag)
lyapExp = lyapunovExponent(X,fs,[],dim)
lyapExp = lyapunovExponent(X,fs,lag,dim)
[lyapExp,estep,ldiv] = lyapunovExponent( ___ )
 ___ = lyapunovExponent( ___ ,Name,Value)

lyapunovExponent( ___ )
```

Description

`lyapExp = lyapunovExponent(X,fs)` estimates the Lyapunov exponent of the uniformly sampled time-domain signal `X` using sampling frequency `fs`. Use `lyapunovExponent` to characterize the rate of separation of infinitesimally close trajectories in phase space to distinguish different attractors. Lyapunov exponent is useful in quantifying the level of chaos in a system, which in turn can be used to detect potential faults.

`lyapExp = lyapunovExponent(X,fs,lag)` estimates the Lyapunov exponent for the time delay `lag`.

`lyapExp = lyapunovExponent(X,fs,[],dim)` estimates the Lyapunov exponent for the embedding dimension `dim`.

`lyapExp = lyapunovExponent(X,fs,lag,dim)` estimates the Lyapunov exponent for the time delay `lag` and embedding dimension `dim`.

`[lyapExp,estep,ldiv] = lyapunovExponent( ___ )` estimates the Lyapunov exponent, expansion step, and the corresponding logarithmic divergence of the uniformly sampled time-domain signal `X`. Use expansion step `estep` and the corresponding logarithmic divergence `ldiv` for signal diagnostics.

`___ = lyapunovExponent( ___ ,Name,Value)` estimates the Lyapunov exponent with additional options specified by one or more `Name,Value` pair arguments.

`lyapunovExponent( ___ )` with no output arguments creates an average logarithmic divergence versus expansion step plot.

Use the generated interactive plot to find an appropriate `ExpansionRange`.

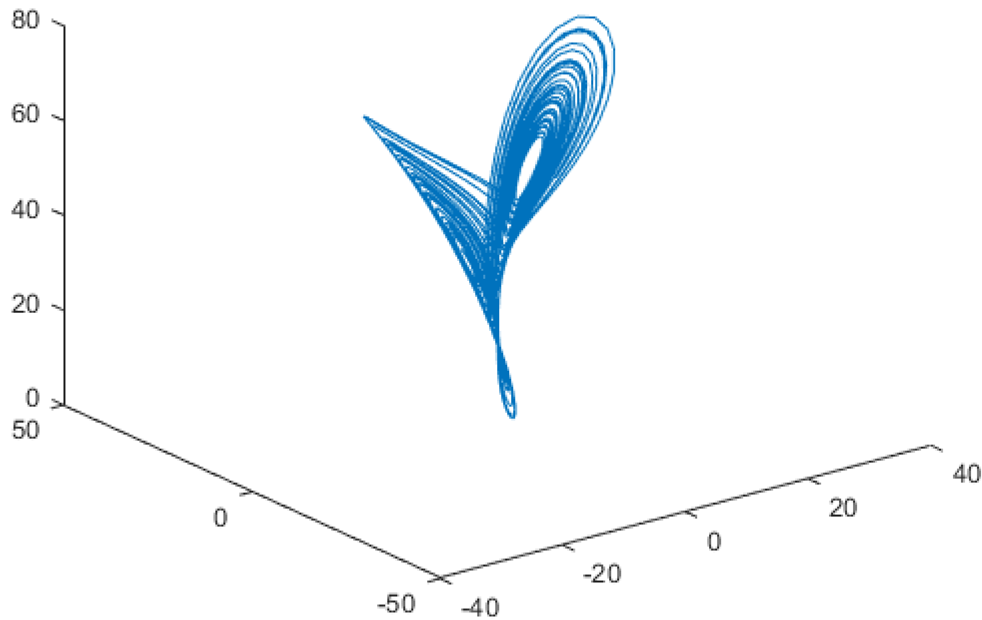
Examples

Visualize and Estimate Largest Lyapunov Exponent

In this example, consider a Lorenz attractor describing a unique set of chaotic solutions.

Load the data set and sampling frequency `fs` to the workspace, and visualize the Lorenz attractor in 3-D.

```
load('lorenzAttractorExampleData.mat','data','fs');
plot3(data(:,1),data(:,2),data(:,3));
```



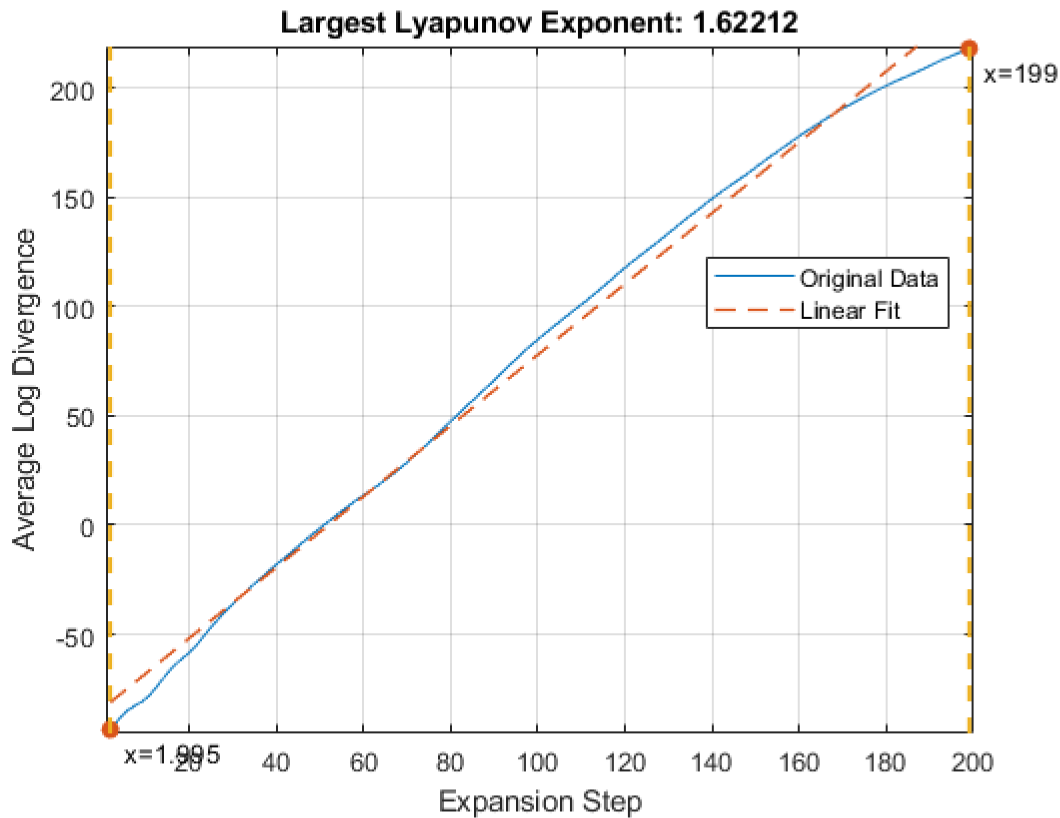
For this example, use the x-direction data of the Lorenz attractor. Since `Lag` is unknown, estimate the delay using `phaseSpaceReconstruction`. Set dimension to 3 since the Lorenz attractor is a three-dimensional system. The `dim` and `lag` parameters are required to create the logarithmic divergence versus expansion step plot.

```
xdata = data(:,1);
dim = 3;
[~,lag] = phaseSpaceReconstruction(xdata,[],dim)
```

```
lag = 10
```

Create the average logarithmic divergence versus expansion step plot for the Lorenz attractor, using the `lag` value obtained in the previous step. Set a sufficiently large expansion range to capture all the expansion steps.

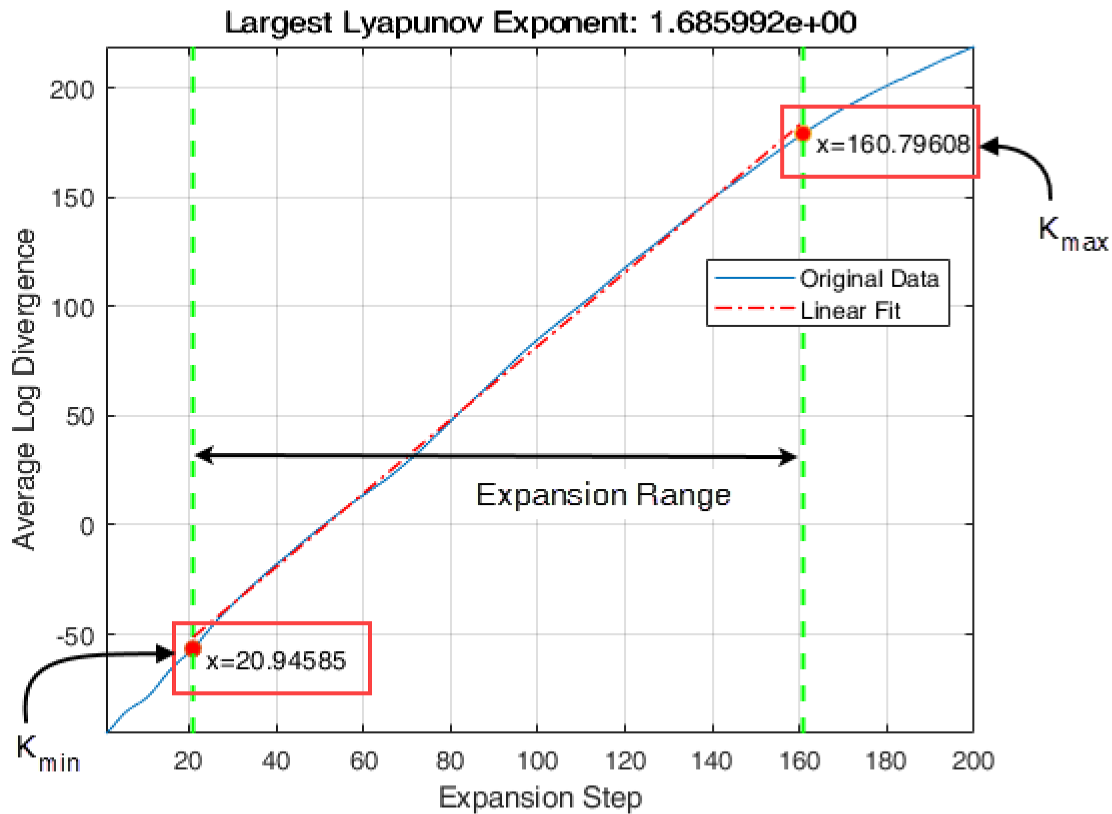
```
eRange = 200;
lyapunovExponent(xdata,fs,lag,dim,'ExpansionRange',eRange)
```



The first dashed, vertical green line (on the left) indicates the minimum number of steps used to estimate the expansion range, while the second vertical green line (on the right), represents the maximum number of steps used. Together, the first and second vertical lines represent the expansion range. The dashed red line indicates the linear fit line for the data, within the expansion range.

To compute the largest Lyapunov exponent, you first need to determine the expansion range needed for accurate estimation.

In the plot, drag the two dashed, vertical green lines to *best fit* the linear fit line to the original data line to obtain the expansion range: K_{min} and K_{max}.



Note the new values of the expansion range after dragging the two vertical lines for an appropriate fit.

Since expansion range can only be specified using whole numbers, round-off K_{min} and K_{max} to the nearest integer. Find the largest Lyapunov exponent of the Lorenz attractor using the new expansion range value.

```
Kmin = 21;
Kmax = 161;
lyapExp = lyapunovExponent(xdata,fs,lag,dim,'ExpansionRange',[Kmin Kmax])
```

```
lyapExp = 1.6834
```

A negative Lyapunov exponent indicates convergence, while positive Lyapunov exponents demonstrate divergence and chaos. The magnitude of `lyapExp` is an indicator of the rate of convergence or divergence of the infinitesimally close trajectories.

Input Arguments

X — Uniformly sampled time-domain signal

vector | array | timetable

Uniformly sampled time-domain signal, specified as a vector, array, or timetable. If X has multiple columns, `lyapunovExponent` computes the largest Lyapunov exponent by treating X as a multivariate signal.

If X is specified as a row vector, `lyapunovExponent` treats it as a univariate signal.

fs — Sampling frequency
scalar

Sampling frequency, specified as a scalar. Sampling frequency or sampling rate is the average number of samples obtained in one second.

If `fs` is not supplied, a normalized frequency of 2π is used to compute the Lyapunov exponent. If X is specified as a timetable, the sampling time is inferred from it.

dim — Embedding dimension
scalar | vector

Embedding dimension, specified as a scalar or vector. `dim` is equivalent to the '`Dimension`' name-value pair.

lag — Time delay
scalar | vector

Time delay, specified as a scalar or vector. `lag` is equivalent to the '`Lag`' name-value pair.

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: ...,`'Dimension',3`

Dimension — Embedding dimension
2 (default) | scalar | vector

Embedding dimension, specified as the comma-separated pair consisting of '`Dimension`' and either a scalar or vector. When `Dimension` is scalar, every column in X is reconstructed using `Dimension`. When `Dimension` is a vector having same length as the number of columns in X, the reconstruction dimension for column `i` is `Dimension(i)`.

Specify `Dimension` based on the dimension of your system, that is, the number of states. For more information on embedding dimension, see `phaseSpaceReconstruction`.

Lag — Delay in phase space reconstruction
1 (default) | scalar | vector

Delay in phase space reconstruction, specified as the comma-separated pair consisting of '`Lag`' and either a scalar or vector. When `Lag` is scalar, every column in X is reconstructed using `Lag`. When `Lag` is a vector having same length as the number of columns in X, the reconstruction delay for column `i` is `Lag(i)`.

The default value of `Lag` is 1.

If the delay is too small, random noise is introduced in the data. In contrast, if the lag is too large, the reconstructed dynamics do not represent the true dynamics of the time series. For more information on estimating optimal delay, see `phaseSpaceReconstruction`.

MinSeparation — Mean period
ceil(fs/max(meanfreq(X,fs))) (default) | positive scalar integer

Mean period, specified as the comma-separated pair consisting of 'MinSeparation' and a positive scalar integer.

MinSeparation is the threshold value used to find the nearest neighbor i^* for a point i to estimate the largest Lyapunov exponent.

The default value of MinSeparation is ceil(fs/max(meanfreq(X,fs))).

ExpansionRange — Range of expansion steps
[1, 5] (default) | 1x2 positive integer array | positive scalar integer

Range of expansion steps, specified as the comma-separated pair consisting of 'ExpansionRange' and either a 1x2 positive integer array or a positive scalar integer.

The minimum and maximum value of ExpansionRate is used to estimate the local expansion rate to calculate the Lyapunov exponent.

If ExpansionRange is specified as a scalar M, then the range is set to be [1, M]. ExpansionRange can only be specified using positive whole numbers and the default value is [1, 5].

Output Arguments

lyapExp — Largest Lyapunov exponent
scalar

Largest Lyapunov exponent, returned as a scalar. lyapExp quantifies the rate of divergence or convergence of close trajectories in phase space.

A negative Lyapunov exponent indicates convergence, while positive Lyapunov exponents demonstrate divergence and chaos. The magnitude of lyapExp is an indicator of the rate of convergence or divergence of the infinitesimally close trajectories.

The ability to discern levels of divergence within data sets is useful in the field of engineering to estimate component failure by studying their vibration and acoustic signals, or to predict when a ship would capsize based on its motion.[2][3]

estep — Expansion step used for estimation
array

Expansion step used for estimation, returned as an array. estep is the difference between the maximum and minimum expansion range split into an equal number of points defined by the maximum value of ExpansionRange.

ldiv — Logarithmic divergence
array

Logarithmic divergence, returned as an array with the same size as estep. The magnitude of each value in ldiv corresponds to the logarithmic convergence or divergence of each point in estep.

Algorithms

Lyapunov exponent is calculated in the following way:

1. The `lyapunovExponent` function first generates a delayed reconstruction $Y_{1:N}$ with embedding dimension m, and lag τ.
2. For a point `i`, the software then finds the nearest neighbor point i^* that satisfies $\min_{i^*} \|Y_i - Y_{i^*}\|$ such that $|i - i^*| > MinSeparation$, where `MinSeparation`, the mean period, is the reciprocal of the mean frequency.
3. From [1], the Lyapunov exponent for the entire expansion range is calculated as,

   $$\lambda(i) = \frac{1}{(K_{\max} - K_{\min} + 1)dt} \sum_{K = K_{\min}}^{K_{\max}} \frac{1}{K} \ln \frac{\|Y_{i+K} - Y_{i^*+K}\|}{\|Y_i - Y_{i^*}\|}$$

   where, K_{min} and K_{max} represent `ExpansionRange`, `dt` is the sampling time and $ldiv = \ln \frac{\|Y_{i+K} - Y_{i^*+K}\|}{\|Y_i - Y_{i^*}\|}$
4. A single value for the Lyapunov exponent is then calculated from the earlier step using the `polyfit` command as,

   $$lyapExp = polyfit([K_{\min}\ K_{\max}], \lambda(i))$$

References

[1] Michael T. Rosenstein , James J. Collins , Carlo J. De Luca. "A practical method for calculating largest Lyapunov exponents from small data sets ". *Physica D* 1993. Volume 65. Pages 117-134.

[2] Caesarendra, Wahyu & Kosasih, P & Tieu, Kiet & Moodie, Craig. "An application of nonlinear feature extraction-A case study for low speed slewing bearing condition monitoring and prognosis." *IEEE/ASME International Conference on Advanced Intelligent Mechatronics: Mechatronics for Human Wellbeing, AIM 2013*.1713-1718. 10.1109/AIM.2013.6584344.

[3] McCue, Leigh & W. Troesch, Armin. (2011). "Use of Lyapunov Exponents to Predict Chaotic Vessel Motions". *Fluid Mechanics and its Applications*. 97. 415-432. 10.1007/978-94-007-1482-3_23.

See Also

`approximateEntropy` | `correlationDimension` | `phaseSpaceReconstruction`

Introduced in R2018a

monotonicity

Quantify monotonic trend in condition indicators

Syntax

```
Y = monotonicity(X)
Y = monotonicity(X,lifetimeVar)
Y = monotonicity(X,lifetimeVar,dataVar)
Y = monotonicity(X,lifetimeVar,dataVar,memberVar)
Y = monotonicity( ___ ,Name,Value)

monotonicity( ___ )
```

Description

`Y = monotonicity(X)` returns the monotonicity of the lifetime data `X`. Use `monotonicity` to quantify the monotonic trend in condition indicators as the system evolves toward failure. The values of `Y` range from 0 to 1, where `Y` is 1 if `X` is perfectly monotonic and 0 if `X` is non-monotonic.

As a system gets progressively closer to failure, a suitable condition indicator typically has a monotonic trend. Conversely, any feature with a non-monotonic trend is a less suitable condition indicator.

`Y = monotonicity(X,lifetimeVar)` returns the monotonicity of the lifetime data `X` using the lifetime variable `lifetimeVar`.

`Y = monotonicity(X,lifetimeVar,dataVar)` returns the monotonicity of the lifetime data `X` using the data variables specified by `dataVar`.

`Y = monotonicity(X,lifetimeVar,dataVar,memberVar)` returns the monotonicity of the lifetime data `X` using the lifetime variable `lifetimeVar`, the data variables specified by `dataVar`, and the member variable `memberVar`.

`Y = monotonicity( ___ ,Name,Value)` estimates the monotonicity with additional options specified by one or more `Name,Value` pair arguments. You can use this syntax with any of the previous input-argument combinations.

`monotonicity( ___ )` with no output arguments plots a bar chart of ranked monotonicity values.

Examples

Monotonicity of Data in Cell Array of Matrices

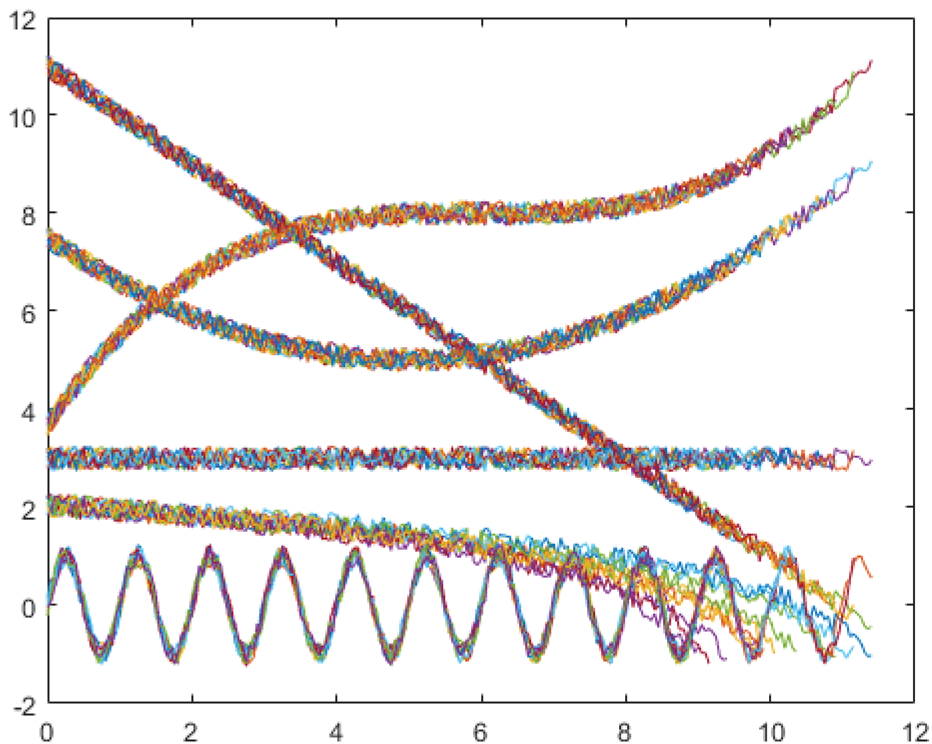
In this example, consider the lifetime data of 10 identical machines with the following 6 potential prognostic parameters—constant, linear, quadratic, cubic, logarithmic, and periodic. The data set `machineDataCellArray.mat` contains `C`, which is a `1x10` cell array of matrices where each element of the cell array is a matrix that contains the lifetime data of a machine. For each matrix in the cell array, the first column contains the time while the other columns contain the data variables.

Load the lifetime data and visualize it against time.

```
load('machineDataCellArray.mat','C')
display(C)
```

```
C=1×10 cell array
  Columns 1 through 4

    {219x7 double}    {189x7 double}    {202x7 double}    {199x7 double}

  Columns 5 through 8

    {229x7 double}    {184x7 double}    {224x7 double}    {208x7 double}

  Columns 9 through 10

    {181x7 double}    {197x7 double}
```

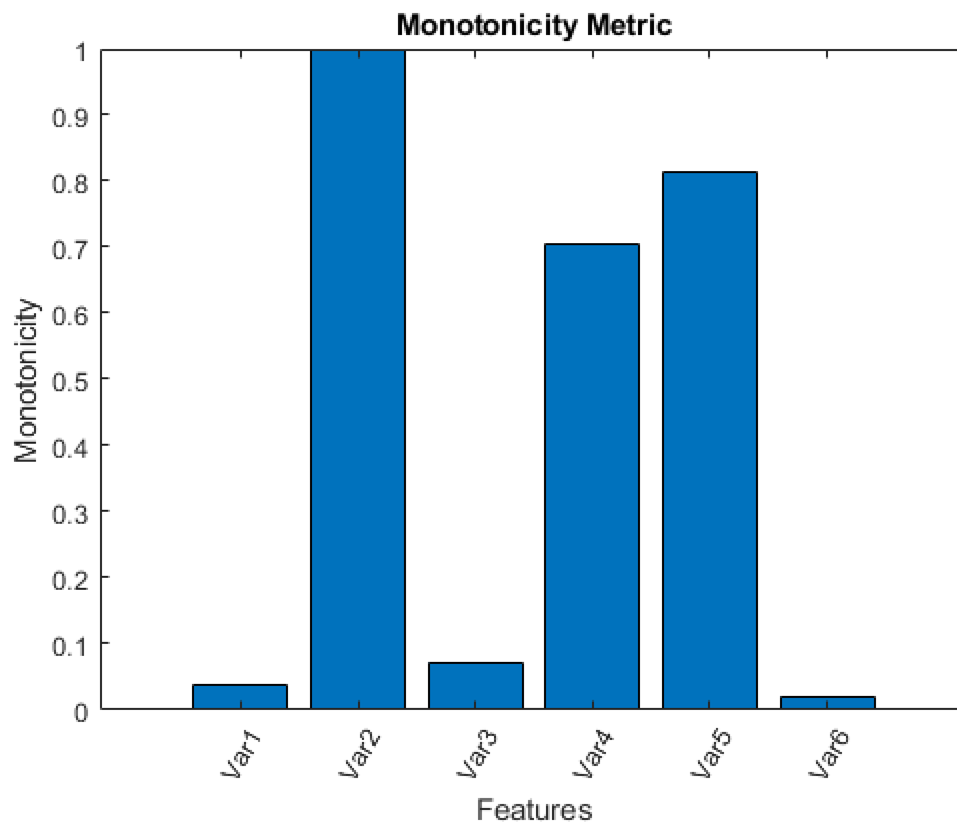
```
for k = 1:length(C)
    plot(C{k}(:,1), C{k}(:,2:end));
    hold on;
end
```



Observe the 6 different condition indicators–constant, linear, quadratic, cubic, logarithmic, and periodic–for all 10 machines on the plot.

Visualize the monotonicity of the potential prognostic features.

```
monotonicity(C)
```



From the histogram plot, observe that the features `Var2`, `Var4` and `Var5` rank better than the others. Hence, these features are more appropriate for remaining useful life predictions since they are the best indicators of machine health.

Monotonicity of Data in Cell Array of Tables

In this example, consider the lifetime data of 10 identical machines with the following 6 potential prognostic parameters—constant, linear, quadratic, cubic, logarithmic, and periodic. The data set `machineDataTable.mat` contains `T`, which is a `1x10` cell array of tables where each element of the cell array contains a table of lifetime data for a machine.

Load and display the data.

```
load('machineDataTable.mat','T');
display(T)
```

```
T=1×10 cell array
  Columns 1 through 4

    {219x7 table}    {189x7 table}    {202x7 table}    {199x7 table}

  Columns 5 through 8
```

```
    {229x7 table}    {184x7 table}    {224x7 table}    {208x7 table}

  Columns 9 through 10

    {181x7 table}    {197x7 table}
```

```
head(T{1},2)
```

```
ans=2×7 table
    Time    Constant    Linear    Quadratic    Cubic     Logarithmic    Periodic
    ____    ________    ______    _________    ______    ___________    ________

       0     3.2029     11.203     7.7029      3.8829      2.2517         0.2029
    0.05     2.8135     10.763     7.2637      3.6006      1.8579        0.12251
```

Note that every table in the cell array contains the lifetime variable `'Time'` and the data variables '`Constant`', '`Linear`', '`Quadratic`', '`Cubic`', '`Logarithmic`', and '`Periodic`'.

Compute monotonicity using Spearman's rank correlation method with `Time` as the lifetime variable.

```
Y = monotonicity(T,'Time','Method','rank')
```

```
Y=1×6 table
    Constant    Linear    Quadratic     Cubic     Logarithmic    Periodic
    ________    ______    _________    _______    ___________    ________

    0.069487      1        0.17777     0.97993      0.99957      0.059208
```

From the resulting table of monotonicity values, observe that the linear, cubic, and logarithmic features have values closer to 1. Hence, these three features are more appropriate for predicting remaining useful life since they are the best indicators of machine health.

Visualize Monotonicity of Lifetime Data in Ensemble Datastore

Consider the lifetime data of 4 machines. Each machine has 4 fault codes for the potential condition indicators—voltage, current, and power. `monotonicityEnsemble.zip` is a collection of 4 files where every file contains a timetable of lifetime data for each machine – `tbl1.mat`, `tbl2.mat`, `tbl3.mat`, and `tbl4.mat`. You can also use files containing data for multiple machines. For each timetable, the organization of the data is as follows:

Time	Voltage	Current	Power	FaultCode	Machine

When you perform calculations on tall arrays, MATLAB® uses either a parallel pool (default if you have Parallel Computing Toolbox™) or the local MATLAB session. To run the example using the local MATLAB session, change the global execution environment by using the `mapreducer` function.

```
mapreducer(0)
```

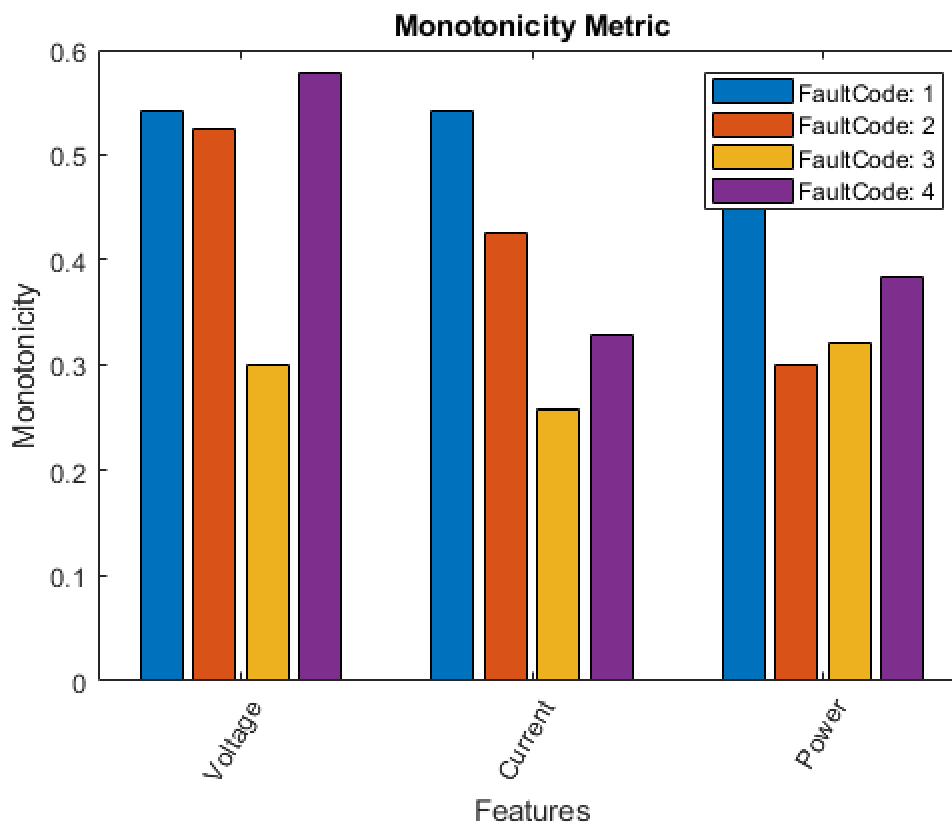
Extract the compressed files, read the data in the timetables, and create a `fileEnsembleDatastore` object using the timetable data. For more information on creating a file ensemble datastore, see `fileEnsembleDatastore`.

```
unzip monotonicityEnsemble.zip;
ens = fileEnsembleDatastore(pwd,'.mat');
ens.DataVariables = {'Voltage','Current','Power','FaultCode','Machine'};
% Make sure that the function for reading data is on path
addpath(fullfile(matlabroot,'examples','predmaint','main'))
ens.ReadFcn = @readtable_data;
ens.SelectedVariables = {'Voltage','Current','Power','FaultCode','Machine'};
```

Visualize the monotonicity of the potential prognostic features with '`Machine`' as the member variable and group the lifetime data by '`FaultCode`'. Grouping the lifetime data ensures that `monotonicity` calculates the metric for each fault code separately.

```
monotonicity(ens,'MemberVariable','Machine','GroupBy','FaultCode');
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 1.1 sec
Evaluation completed in 2.1 sec
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.24 sec
Evaluation completed in 0.53 sec
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.7 sec
Evaluation completed in 0.75 sec
```



`monotonicity` returns a histogram plot with the features ranked by their monotonicity values. A higher monotonicity value indicates a more suitable prognostic parameter. For instance, the candidate feature `Current` has the highest monotonic trend for machines with `FaultCode 1`.

```
rmpath(fullfile(matlabroot,'examples','predmaint','main')) % Reset path
```

Input Arguments

X — Lifetime data

cell array of matrices | cell array of tables and timetables | `fileEnsembleDatastore` object | table | timetable

Lifetime data, specified as a cell array of matrices, cell array of tables and timetables, `fileEnsembleDatastore` object, table, or timetable. Lifetime data contains run-to-failure data of the systems being monitored. The term *lifetime* here refers to the life of the machine defined in terms of the units you use to measure system life. Units of lifetime can be quantities such as the distance traveled (miles), fuel consumed (gallons), or time since the start of operation (days).

If `X` is

- a cell array of matrices or tables, the function assumes that each matrix or table contains columns of lifetime data for a system. Each column of every matrix or table, except the first column, contains data for a prognostic variable. `'Var1','Var2', ...` can be used to refer to the matrix columns that contain the lifetime data. For instance, the file `machineDataCellArray.mat` contains a 1-by-10 cell array of matrices `C`, where each of the 10 matrices contains data for a particular machine.
- a table or timetable, the function assumes that each column, except the first one, contains columns of lifetime data. The table variable names can be used to refer to the columns that contain the lifetime data. If `lifetimeVar` is not specified when `X` is a table, then the first data column is used as the lifetime variable.
- a `fileEnsembleDatastore` object, specify the data variables `dataVar` and member variables `memberVar` to be used. If `lifetimeVar` is not specified, then the first data column is used as the lifetime variable for computation.

Each numerical member in `X` is of type `double`.

`lifetimeVar` — Lifetime variable

string | character vector

Lifetime variable, specified as a string or character vector. `lifetimeVar` measures the lifetime of the systems being monitored and the lifetime data is sorted with respect to `lifetimeVar`. The value of `lifetimeVar` must be a valid ensemble or table variable name.

For a cell array of matrices, the value `'Time'` can be used to refer to the first column of each matrix, which is assumed to contain the lifetime variable. For instance, the file `machineDataCellArray.mat` contains the cell array `C`, where the first column in each matrix contains the lifetime variable while the other columns contain the data variables.

`dataVar` — Data variables

string array | character vector | cell array of character vectors

Data variables, specified as a string array, character vector, or cell array of character vectors. Data variables are the main content of the members of an ensemble. Data variables can include measured data or derived data for the analysis and development of predictive maintenance algorithms.

If `X` is

- a `fileEnsembleDatastore` object, the value of `dataVar` supersedes the `DataVariables` property of the ensemble.
- a cell array of matrices, the value `'Time'` can be used to refer to the first column of each matrix, that is, the lifetime variable `lifetimeVar`. `'Var1','Var2', ...` can be used to refer to the other matrix columns which contain the lifetime data. For instance, the file `machineDataCellArray.mat` contains the cell array `C` where the first column in each matrix contains the lifetime variable. The other columns in the cell array `C` contain the data variables.
- a table, the table variable names can be used to refer to the columns which contain the lifetime data.

The values of `dataVar` must be valid ensemble or table variable names. If `dataVar` is not specified, the computation includes all data columns except the one specified in `lifetimeVar`. For instance, suppose that each entry in a cell array is a table with variables `A`, `B`, `C`, and `D`. Setting `dataVar` to `["A","D"]` uses only `A` and `D` for the computation while `C` and `D` are ignored.

`memberVar` — Member variable
string | character vector

Member variable, specified as a string or character vector. Use `memberVar` to specify the variable for identifying the systems or machines in lifetime data `X`. For instance, in the `fileEnsembleDatastore` object, the fifth column in each timetable contains numbers that identify data from a particular machine. The column name corresponds to the member variable `memberVar`.

`memberVar` is ignored when `X` is specified as a cell array of matrices or tables.

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','rank'`

`LifeTimeVariable` — Lifetime variable
`strings(0)` (default) | string | character vector

Lifetime variable, specified as the comma-separated pair consisting of '`LifeTimeVariable`' and either a string or character vector. If '`LifeTimeVariable`' is not specified, then the first data column is used.

'`LifeTimeVariable`' is equivalent to the input argument `lifetimeVar`.

`DataVariables` — Data variables
`strings(0)` (default) | string array | character vector | cell array of character vectors

Data variables, specified as the comma-separated pair consisting of '`DataVariables`' and either a string array, character vector or cell array of character vectors.

'`DataVariables`' is equivalent to the input argument `dataVar`.

`MemberVariable` — Member variables
[] (default) | string | character vector

Member variables, specified as the comma-separated pair consisting of '`MemberVariable`' and either a string or character vector.

'`MemberVariable`' is equivalent to the input argument `memberVar`.

`GroupBy` — Grouping criterion
[] (default) | string | character vector

Grouping criterion, specified as the comma-separated pair consisting of '`GroupBy`' and either a string or character vector. Use '`GroupBy`' to specify the variables for grouping the lifetime data `X` by operating conditions.

The function computes the metric separately for each group that results from applying the criterion, such as a fault condition, specified by '`GroupBy`'. For instance, in the `fileEnsembleDatastore` object `ens`, the fourth column in each timetable in `ens` contains the variable `'FaultCode'`. The metric is computed for each machine by grouping the data by `'FaultCode'`.

You can only group variables when `X` is defined as a `fileEnsembleDatastore` object, table, timetable, or cell array of tables or timetables.

`WindowSize` — Size of the centered moving average window for data smoothing
[] (default) | scalar | two-element vector

Size of the centered moving average window for data smoothing, specified as the comma-separated pair consisting of '`WindowSize`' and either a scalar or two-element vector. A Savitzky-Golay filter is used for data smoothing. For more information, see `smoothdata`.

If '`WindowSize`' is not specified, the window length is automatically determined from lifetime data `X` using `smoothdata(X,'sgolay')`. Set '`WindowSize`' to 0 to turn off data smoothing.

`Method` — Method to compute monotonicity
`'sign'` (default) | `'rank'`

Method to compute monotonicity, specified as the comma-separated pair consisting of '`Method`' and either `'sign'` or `'rank'`.

- `'sign'`, Use the signum formula.
- `'rank'`, Use Spearman's rank correlation formula.

For more information, see “Algorithms” on page 1-143.

Output Arguments

Y — Monotonicity of lifetime data
vector | table

Monotonicity of lifetime data, returned as a vector or table.

`monotonicity` characterizes the trend of a feature as the system evolves toward failure. As a system gets progressively closer to failure, a suitable condition indicator typically has a monotonic trend.

Conversely, any feature with a non-monotonic trend is a less suitable condition indicator. The values of Y range from 0 to 1.

- Y is 1 if X is perfectly monotonic.
- Y is 0 if X is perfectly non-monotonic.

Selecting appropriate estimation parameters out of all available features is the first step in building a reliable remaining useful life prediction engine. The monotonicity values in Y are useful to determine which condition indicators best track the degradation process of the systems being monitored. The higher the monotonic trend, the more desirable the feature is for prognostics.

When 'GroupBy' is not specified, then Y is returned as a row vector or single-row table. Conversely, when 'GroupBy' is specified, then each row in Y corresponds to one group.

Limitations

- When X is a tall table or tall timetable, `monotonicity` nevertheless loads the complete array into memory using `gather`. If the memory available is inadequate, then `monotonicity` returns an error.

Algorithms

Monotonicity is computed in the following two ways as specified by the 'Method' option.

Signum Formula or Sign Method

When you specify 'Method' as `'sign'`, the computation of monotonicity uses this formula:

$$\text{monotonicity} = \frac{1}{M}\sum_{j=1}^{M}\left|\sum_{k=1}^{N_j-1}\frac{\text{sgn}\left(x_j(k+1)-x_j(k)\right)}{N_j-1}\right|$$

where x_j represents the vector of measurements of a feature on the jth system, M is the number of systems monitored, and N_j is the number of measurements on the jth system.

Spearman's Rank Correlation Coefficient Method

When you specify 'Method' as `'rank'`, the computation of monotonicity uses this formula:

$$\text{monotonicity} = \frac{1}{M}\sum_{j=1}^{M}\left|\text{corr}\left(\text{rank}(x_j),\text{rank}(t_j)\right)\right|$$

where M is the number of systems monitored and t_j is the vector of time points corresponding to the measurement vector x_j.

References

[1] Coble, J., and J. W. Hines. "Identifying Optimal Prognostic Parameters from Data: A Genetic Algorithms Approach." In *Proceedings of the Annual Conference of the Prognostics and Health Management Society*. 2009.

[2] Coble, J. "Merging Data Sources to Predict Remaining Useful Life - An Automated Method to Identify Prognostics Parameters." Ph.D. Thesis. University of Tennessee, Knoxville, TN, 2010.

[3] Lei, Y. *Intelligent Fault Diagnosis and Remaining Useful Life Prediction of Rotating Machinery*. Xi'an, China: Xi'an Jiaotong University Press, 2017.

[4] Lofti, S., J. B. Ali, E. Bechhoefer, and M. Benbouzid. "Wind turbine high-speed shaft bearings health prognosis through a spectral Kurtosis-derived indices and SVR." *Applied Acoustics* Vol. 120, 2017, pp. 1-8.

See Also

`prognosability` | `trendability` | `fileEnsembleDatastore`

Topics

"Wind Turbine High-Speed Bearing Prognosis"
"Feature Selection for Remaining Useful Life Prediction"

Introduced in R2018b

phaseSpaceReconstruction

Convert observed time series to state vectors

Syntax

```
XR = phaseSpaceReconstruction(X,lag,dim)
[XR,eLag,eDim] = phaseSpaceReconstruction(X)
[XR,eLag,eDim] = phaseSpaceReconstruction(X,lag)
[XR,eLag,eDim] = phaseSpaceReconstruction(X,[],dim)
[ ___ ] = phaseSpaceReconstruction( ___ ,Name,Value)
```

```
phaseSpaceReconstruction( ___ )
```

Description

`XR = phaseSpaceReconstruction(X,lag,dim)` returns the reconstructed phase space `XR` of the uniformly sampled time-domain signal `X` with time delay `lag` and embedding dimension `dim` as inputs.

Use `phaseSpaceReconstruction` to verify the system order and reconstruct all dynamic system variables, while preserving system properties. Reconstructing the phase space is useful when limited data is available, or when the phase space dimension and lag is unknown. The nonlinear features `approximateEntropy`, `correlationDimension`, and `lyapunovExponent` use `phaseSpaceReconstruction` as the first step of the computation.

`[XR,eLag,eDim] = phaseSpaceReconstruction(X)` returns reconstructed phase space `XR` along with the estimated delay `eLag` and embedding dimension `eDim`.

`[XR,eLag,eDim] = phaseSpaceReconstruction(X,lag)` returns the reconstructed phase space `XR` of uniformly sampled time domain signal `X` and embedding dimension `eDim` using time delay specified by `lag.`

`[XR,eLag,eDim] = phaseSpaceReconstruction(X,[],dim)` returns the reconstructed phase space `XR` of uniformly sampled time domain signal `X` and time delay `eLag` using embedding dimension specified by `dim`.

`[ ___ ] = phaseSpaceReconstruction( ___ ,Name,Value)` returns the reconstructed phase space `XR` with additional options specified by one or more `Name,Value` pair arguments.

`phaseSpaceReconstruction( ___ )` with no output arguments creates a matrix of sub-axes of the reconstructed phase space with histogram plots along the diagonal.

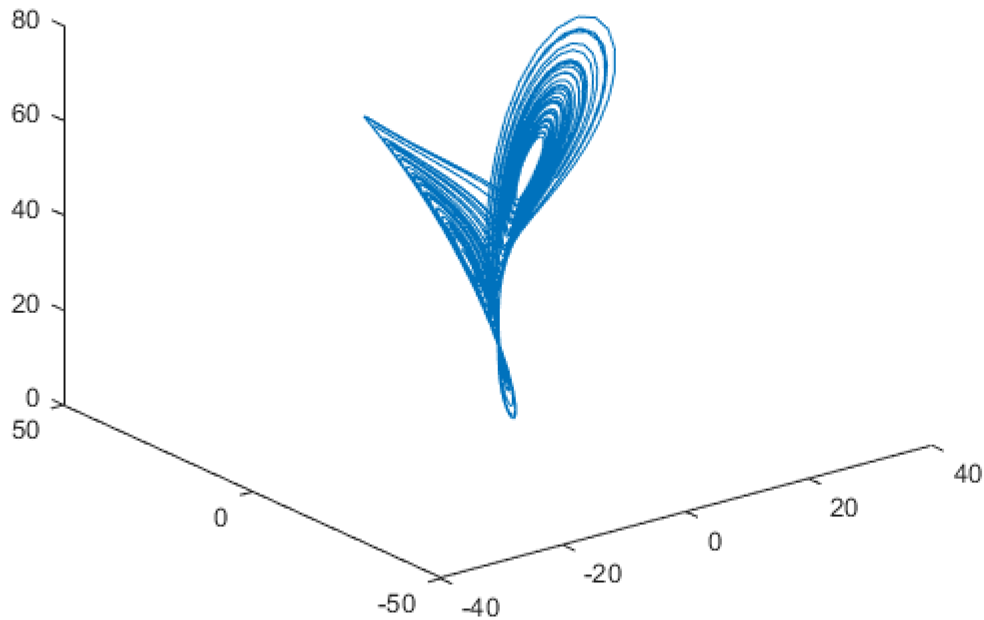
Examples

Reconstruct Data using Phase Space Reconstruction

In this example, assume that you have measurements for a Lorenz Attractor. Your measurements are along the x direction only, but the attractor is a three-dimensional system. Using this limited data, reconstruct the phase space such that the properties of the original system are preserved.

Load the Lorenz Attractor data and visualize its x, y and z measurements on a 3-D plot.

```
load('lorenzAttractorExampleData.mat','data');
plot3(data(:,1),data(:,2),data(:,3));
```



Estimate the lag and dimension using the x-direction measurement.

```
xdata = data(:,1);
[~,eLag,eDim] = phaseSpaceReconstruction(xdata)
```

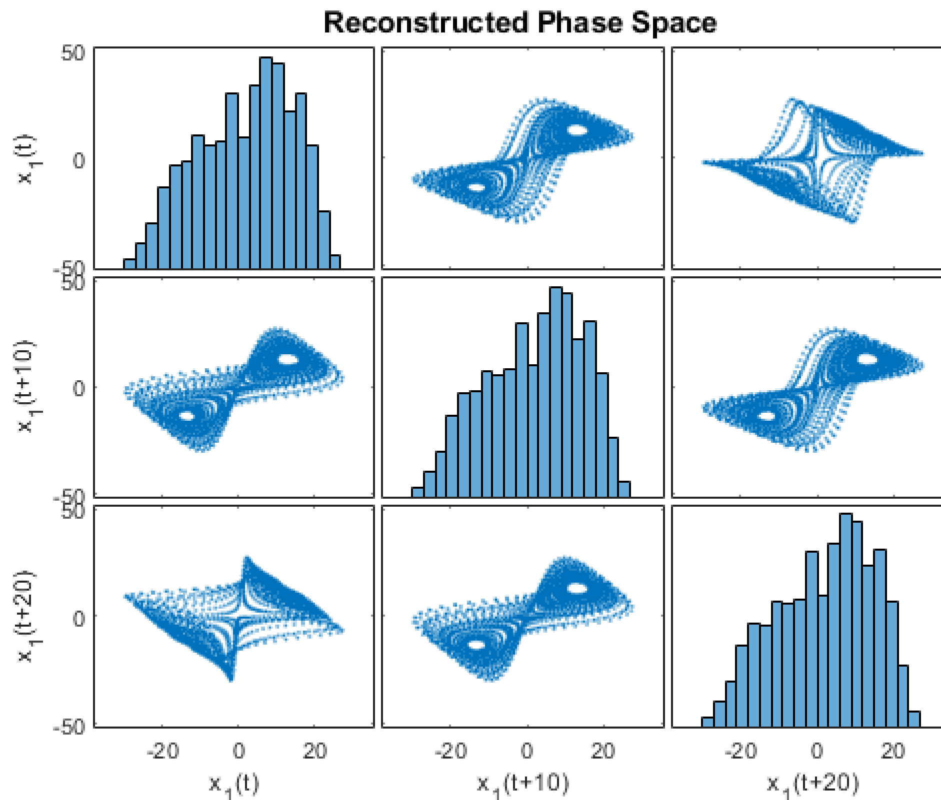
```
eLag = 10
```

```
eDim = 3
```

Since the Lorenz Attractor has data in 3 dimensions, the estimated embedding dimension `eDim` is 3.

Visualize the reconstructed phase space using the estimated lag and embedding dimension.

```
phaseSpaceReconstruction(xdata,eLag,eDim);
```



As observed from the 3x3 phase space plot, the topology of the attractor is recovered. $x_1(t+10)$ and $x_1(t+20)$ are the other two states reconstructed with the estimated lag value of 10. The diagonal plots (1,1), (2,2) and (3,3) represent the histogram of $x_1(t)$, $x_1(t+10)$and $x_1(t+20)$data, respectively.

Input Arguments

X — Uniformly sampled time-domain signal
vector | array | timetable

Uniformly sampled time-domain signal, specified as a vector, array, or timetable. When multiple columns exist in X, each column is treated as an independent time series.

If X is specified as a row vector, `phaseSpaceReconstruction` treats it as a univariate signal.

`dim` — Embedding dimension
scalar | vector

Embedding dimension, specified as a scalar or vector. `dim` is the dimension of the space in which you reconstruct a phase portrait starting from your measurements.

When `dim` is scalar, every column in X is reconstructed using `dim`. When `dim` is a vector having same length as the number of columns in X, the reconstruction dimension for column `i` is `dim(i)`.

`lag` — Delay value used in phase space reconstruction
scalar | vector

Delay value used in phase space reconstruction, specified as a scalar or vector. When lag is scalar, every column in X is reconstructed using lag. When lag is a vector having same length as the number of columns in X, the reconstruction delay for column i is lag(i).

If the time delay is too small, random noise is introduced in the states. In contrast, if the lag is too large, the reconstructed dynamics do not represent the true dynamics of the time series.

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: ...'HistogramBins',12

HistogramBins — Number of bins for discretization

10 (default) | scalar

Number of bins for discretization, specified as the comma-separated pair consisting of 'HistogramBins' and a scalar. HistogramBins is required to compute the Average Mutual Information (AMI) to estimate delay eLag.

Set the value of HistogramBins based on the length of X.

MaxLag — Maximum value of lag

10 (default) | scalar

Maximum value of lag, specified as the comma-separated pair consisting of 'MaxLag' and a scalar. MaxLag is used to estimate delay est_delay using the Average Mutual Information (AMI) algorithm.

PercentFalseNeighbors — Factor to determine embedding dimension

0.1 (default) | scalar

Factor to determine embedding dimension, specified as the comma-separated pair consisting of 'PercentFalseNeighbors' and a scalar. When percentage of false nearest neighbors drops below the tuning parameter PercentFalseNeighbors at a dimension d, d is considered as the embedding dimension.

The default value of PercentFalseNeighbors is 0.1 and permissible values lie within the range 0 through 1.

DistanceThreshold — Distance threshold to determine false neighbors

10 (default) | scalar

Distance threshold to determine false neighbors, specified as the comma-separated pair consisting of 'DistanceThreshold' and a scalar. DistanceThreshold is a tuning parameter to determine the number of points that are false nearest neighbors in the reconstructed phase space.

The default value of DistanceThreshold is 10, and suggested values lie within the range 10 through 50.

MaxDim — Maximum value of embedding dimension

5 (default) | scalar

Maximum value of embedding dimension, specified as the comma-separated pair consisting of 'MaxDim' and a scalar.

Change the value of MaxDim if the number of states of your system exceeds 5.

Output Arguments

XR — Reconstructed phase space

array | timetable

Reconstructed phase space, returned as either an array or timetable. XR contains state vectors based on the embedding dimension and lag value.

eLag — Estimated time delay

scalar

Estimated time delay, returned as a scalar, regardless of the size of X.

eLag is estimated using Average Mutual Information (AMI) algorithm. For more information, see "Algorithms" on page 1-149.

eDim — Estimated embedding dimension

scalar

Estimated embedding dimension, returned as a scalar, regardless of the size of X.

eDim is estimated using False Nearest Neighbor (FNN) algorithm. For more information, see "Algorithms" on page 1-149.

Algorithms

Phase Space Reconstruction

For a uniformly sampled univariate time signal $X_1 = \left(x_{1,1}, x_{1,2}, \ldots, x_{1,N}\right)^T$, phaseSpaceReconstruction computes the delayed reconstruction

$$X_{1,i}^r = \left(x_{1,i}, x_{1,i+\tau_1}, \ldots, x_{1,i+(m_1-1)\tau_1}\right),\ i = 1, 2, \ldots, N - (m_1 - 1)\tau_1$$

where, N is the length of the time series, τ_1 is the lag, and m_1 is the embedding dimension for X_1.

Similarly, for a multivariate time series X given by,

$$X = [X_1, X_2, \ldots, X_S] = \begin{bmatrix} x_{1,1} & \ldots & x_{S,1} \\ \vdots & \ddots & \vdots \\ x_{1,N} & \cdots & x_{S,N} \end{bmatrix}$$

phaseSpaceReconstruction computes the reconstruction for each time series as,

$$X_i^r = \left(X_{1,i}^r, X_{2,i}^r, \ldots, X_{S,i}^r\right),\ i = 1, 2, \ldots, N - (\max\{m_i\} - 1)(\max\{\tau_i\})$$

where S is the number of measurements, and N is the length of the time series.

Delay Estimation

The delay for phase space reconstruction is estimated using Average Mutual Information (AMI). For reconstruction, the time delay is set to be the first local minimum of AMI.

Average Mutual Information is computed as,

$$AMI(T) = \sum_{i=1}^{N} p(x_i, x_{i+T}) \log_2 \left[\frac{p(x_i, x_{i+T})}{p(x_i)p(x_{i+T})} \right]$$

where, N is the length of the time series and T = 1:`MaxLag`.

Embedding Dimension Estimation

The embedding dimension for phase space reconstruction is estimated using False Nearest Neighbor (FNN) algorithm.

- For a point i at dimension d, the points $X^r{}_i$ and its nearest point $X^{r*}{}_i$ in the reconstructed phase space $\{X^r{}_i\}$, $i = 1:N$, are false neighbors if

  $$\sqrt{\frac{R_i^2(d+1) - R_i^2(d)}{R_i^2(d)}} > DistanceThreshold$$

  where, $R_i^2(d) = \left\| X_i^r - X_i^{r*} \right\|^2$ is the distance metric.
- The estimated embedding dimension `d` is the smallest value that satisfies the condition p_{fnn} < `PercentFalseNeighbors` where, p_{fnn} is the ratio of FNN points to total number of points in the reconstructed phase space.

References

[1] Rhodes, Carl & Morari, Manfred. "False Nearest Neighbors Algorithm and Noise Corrupted Time Series." *Physical Review. E*. 55.10.1103/PhysRevE.55.6162.

[2] Kliková, B., and Aleš Raidl. "Reconstruction of phase space of dynamical systems using method of time delay." *Proceedings of the 20th Annual Conference of Doctoral Students* WDS 2011.

[3] I. Vlachos, D. Kugiumtzis, "State Space Reconstruction for Multivariate Time Series Prediction", *Nonlinear Phenomena in Complex Systems*, Vol 11, No 2, pp 241-249, 2008.

[4] Kantz, H., and Schreiber, T. *Nonlinear Time Series Analysis*. Cambridge: Cambridge University Press, Vol. 7, 2004.

See Also

`approximateEntropy` | `lyapunovExponent` | `correlationDimension`

Introduced in R2018a

plot

Plot survival function for covariate survival remaining useful life model

Syntax

```
plot(mdl)
plot(mdl,covariates)
```

Description

`plot(mdl)` plots the baseline survival function of the fitted covariate survival model `mdl` against the life time value for which it was computed. The plot data is stored in the `BaselineCumulativeHazard` property of `mdl`.

`plot(mdl,covariates)` plots the survival function computed for the covariate data in `covariates`. To obtain the survival function, the hazard rate is computed using the covariates and combined with the baseline survival function.

Examples

Train Covariate Survival Model

Load training data.

```
load('covariateData.mat')
```

This data contains battery discharge times and related covariate information. The covariate variables are:

- Temperature
- Load
- Manufacturer

The manufacturer information is a categorical variable that must be encoded.

Create a covariate survival model.

```
mdl = covariateSurvivalModel;
```

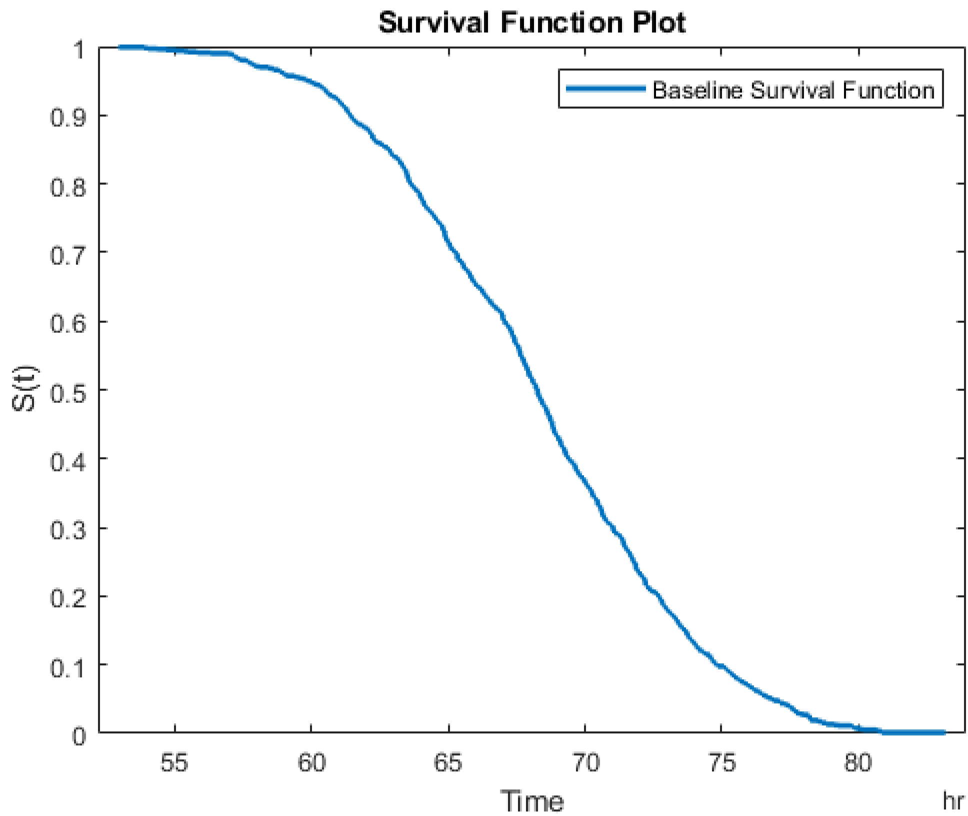
Train the survival model using the training data, specifying the life time variable, data variables, and encoded variable. There is no censor variable for this training data.

```
fit(mdl,covariateData,"DischargeTime",["Temperature","Load","Manufacturer"],[],"Manufacturer")
```

```
Successful convergence: Norm of gradient less than OPTIONS.TolFun
```

Plot the baseline survival function for the model.

```
plot(mdl)
```



Predict RUL Using Covariate Survival Model

Load training data.

```
load('covariateData.mat')
```

This data contains battery discharge times and related covariate information. The covariate variables are:

- Temperature
- Load
- Manufacturer

The manufacturer information is a categorical variable that must be encoded.

Create a covariate survival model, and train it using the training data.

```
mdl = covariateSurvivalModel('LifeTimeVariable',"DischargeTime",'LifeTimeUnit',"hours",...
   'DataVariables',["Temperature","Load","Manufacturer"],'EncodedVariables',"Manufacturer");
fit(mdl,covariateData)
```

```
Successful convergence: Norm of gradient less than OPTIONS.TolFun
```

Suppose you have a battery pack manufactured by maker `B` that has run for `30` hours. Create a test data table that contains the usage time, `DischargeTime`, and the measured ambient temperature, `TestAmbientTemperature`, and current drawn, `TestBatteryLoad`.

```
TestBatteryLoad = 25;
TestAmbientTemperature = 60;
DischargeTime = hours(30);
TestData = timetable(TestAmbientTemperature,TestBatteryLoad,"B",'RowTimes',hours(30));
TestData.Properties.VariableNames = {'Temperature','Load','Manufacturer'};
TestData.Properties.DimensionNames{1} = 'DischargeTime';
```

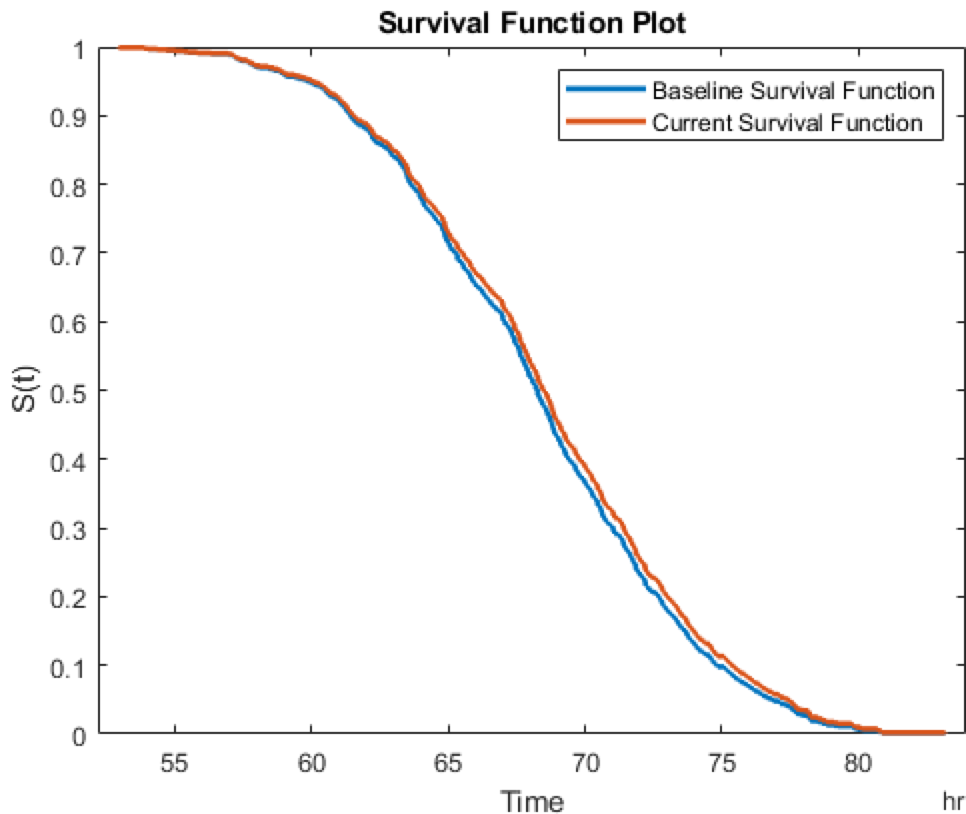
Predict the RUL for the battery.

```
estRUL = predictRUL(mdl,TestData)
```

```
estRUL = duration
   38.332 hr
```

Plot the survival function for the covariate data of the battery.

```
plot(mdl,TestData)
```



Input Arguments

`mdl` — Covariate survival RUL model
`covariateSurvivalModel` object

Covariate survival RUL model, specified as a `covariateSurvivalModel` object.

`plot` plots the data in the `BaselineCumulativeHazard` property of `mdl`, which is a two-column array. The second column contains the baseline survival functions values, and the first column contains the corresponding life time values. The life time values are plotted in the units specified by the `LifeTimeUnits` property of `mdl`.

covariates — Current covariate values
row vector | `table` with one row | `timetable` with one row

Current covariate values for the component, specified as a:

- Row vector whose elements specify the component covariate values only and not the life time values. The number of covariate values must match the number and order of the covariate data columns used when estimating `mdl` using `fit`.
- `table` or `timetable` with one row. The table must contain the variables specified in the `DataVariables` property of `mdl`.

If the covariate data contains encoded variables, then you must specify `covariates` using a `table` or `timetable`.

To obtain the survival function, the hazard rate is computed using the covariates and combined with the baseline survival function. For more information, see “Cox Proportional Hazards Model”.

See Also

Functions
`covariateSurvivalModel` | `predictRUL` | `coxphfit`

Topics
“Cox Proportional Hazards Model”

Introduced in R2018a

predictRUL

Estimate remaining useful life for a test component

Syntax

```
estRUL = predictRUL(mdl,data)
estRUL = predictRUL(mdl,data,bounds)

estRUL = predictRUL(mdl,threshold)

estRUL = predictRUL(mdl,usageTime)

estRUL = predictRUL(mdl,covariates)

estRUL = predictRUL( ___ ,Name,Value)

[estRUL,ciRUL] = predictRUL( ___ )
[estRUL,ciRUL,pdfRUL] = predictRUL( ___ )
[estRUL,ciRUL,pdfRUL,histRUL] = predictRUL( ___ )
```

Description

The `predictRUL` function estimates the remaining useful life (RUL) of a test component given an estimation model and information about its usage time and degradation profile. Before predicting the RUL, you must first configure your estimation model using historical data regarding the health of an ensemble of similar components, such as multiple machines manufactured to the same specifications. To do so, use the `fit` function.

Using `predictRUL`, you can estimate the remaining useful life for the following types of estimation models:

- Degradation models
- Survival models
- Similarity models

For a basic example illustrating RUL prediction, see "Update RUL Prediction as Data Arrives".

For general information on predicting remaining useful life using these models, see "RUL Estimation Using RUL Estimator Models".

`estRUL = predictRUL(mdl,data)` estimates the remaining useful life for a component using similarity model `mdl` and the degradation feature profiles in `data`. `data` contains feature measurements over the life span of the component up to the current life time.

`estRUL = predictRUL(mdl,data,bounds)` estimates the remaining useful life for a component using a similarity model and the feature bounds specified in `bounds`.

`estRUL = predictRUL(mdl,threshold)` estimates the RUL for a component using degradation model `mdl` and the current life time variable value stored in `mdl`. The RUL is the remaining time before the forecasted response of the model reaches the threshold value `threshold`.

`estRUL = predictRUL(mdl,usageTime)` estimates the RUL for a component using reliability survival model `mdl` and the current usage time for the component.

`estRUL = predictRUL(mdl,covariates)` estimates the RUL of a component using covariate survival model `mdl` and the current covariate values for the component.

`estRUL = predictRUL( ___ ,Name,Value)` specifies additional options using one or more name-value pair arguments.

`[estRUL,ciRUL] = predictRUL( ___ )` returns the confidence interval associated with the RUL estimation.

`[estRUL,ciRUL,pdfRUL] = predictRUL( ___ )` returns the probability density function for the RUL estimation.

`[estRUL,ciRUL,pdfRUL,histRUL] = predictRUL( ___ )` returns the histogram of component similarity scores when estimating RUL using a similarity model.

Examples

Train Pairwise Similarity Model

Load training data.

```
load('pairwiseTrainVectors.mat')
```

The training data is a cell array of column vectors. Each column vector is a degradation feature profile for a component.

Create a pairwise similarity model with default settings.

```
mdl = pairwiseSimilarityModel;
```

Train the similarity model using the training data.

```
fit(mdl,pairwiseTrainVectors)
```

Update Linear Degradation Model and Predict RUL

Load observation data.

```
load('linTestData.mat','linTestData1')
```

For this example, assume that the training data is not historical data, but rather real-time observations of the component condition.

Based on knowledge of the degradation feature limits, define a threshold condition indicator value that indicates the end-of-life of a component.

```
threshold = 60;
```

Create a linear degradation model arbitrary prior distribution data and a specified noise variance. Also, specify the life time and data variable names for the observation data.

```
mdl = linearDegradationModel('Theta',1,'ThetaVariance',1e6,'NoiseVariance',0.003,...
                             'LifeTimeVariable',"Time",'DataVariables',"Condition",...
                             'LifeTimeUnit',"hours");
```

Observe the component condition for `50` hours, updating the degradation model after each observation.

```
for i=1:50
    update(mdl,linTestData1(i,:));
end
```

After `50` hours, predict the RUL of the component using the current life time value stored in the model.

```
estRUL = predictRUL(mdl,threshold)
```

```
estRUL = duration
   50.301 hr
```

The estimated RUL is about 50 hours, which indicates a total predicted life span of about 100 hours.

Predict RUL Using Exponential Degradation Model

Load training data.

```
load('expTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component. Each profile consists of life time measurements in the `"Hours"` variable and corresponding degradation feature measurements in the `"Condition"` variable.

Create an exponential degradation model, specifying the life time variable units.

```
mdl = exponentialDegradationModel('LifeTimeUnit',"hours");
```

Train the degradation model using the training data. Specify the names of the life time and data variables.

```
fit(mdl,expTrainTables,"Time","Condition")
```

Load testing data, which is a run-to-failure degradation profile for a test component. The test data is a table with the same life time and data variables as the training data.

```
load('expTestData.mat')
```

Based on knowledge of the degradation feature limits, define a threshold condition indicator value that indicates the end-of-life of a component.

```
threshold = 500;
```

Assume that you measure the component condition indicator every hour for 150 hours. Update the trained degradation model with each measurement. Then, predict the remaining useful life of the component at 150 hours. The RUL is the forecasted time at which the degradation feature will pass the specified threshold.

```
for t = 1:150
 update(mdl,expTestData(t,:))
end
estRUL = predictRUL(mdl,threshold)
```

```
estRUL = duration
   136.45 hr
```

The estimated RUL is around 137 hours, which indicates a total predicted life span of 287 hours.

Predict RUL Using Covariate Survival Model

Load training data.

```
load('covariateData.mat')
```

This data contains battery discharge times and related covariate information. The covariate variables are:

- Temperature
- Load
- Manufacturer

The manufacturer information is a categorical variable that must be encoded.

Create a covariate survival model, and train it using the training data.

```
mdl = covariateSurvivalModel('LifeTimeVariable',"DischargeTime",'LifeTimeUnit',"hours",...
   'DataVariables',["Temperature","Load","Manufacturer"],'EncodedVariables',"Manufacturer");
fit(mdl,covariateData)
```

```
Successful convergence: Norm of gradient less than OPTIONS.TolFun
```

Suppose you have a battery pack manufactured by maker B that has run for 30 hours. Create a test data table that contains the usage time, DischargeTime, and the measured ambient temperature, TestAmbientTemperature, and current drawn, TestBatteryLoad.

```
TestBatteryLoad = 25;
TestAmbientTemperature = 60;
DischargeTime = hours(30);
TestData = timetable(TestAmbientTemperature,TestBatteryLoad,"B",'RowTimes',hours(30));
TestData.Properties.VariableNames = {'Temperature','Load','Manufacturer'};
TestData.Properties.DimensionNames{1} = 'DischargeTime';
```

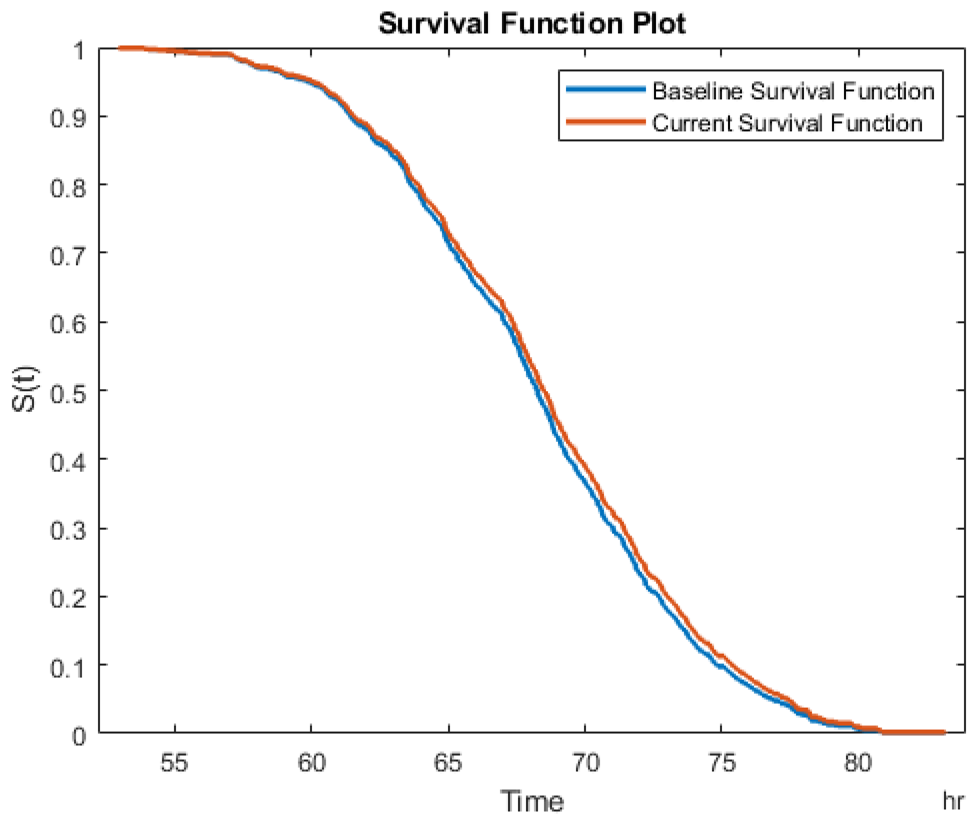
Predict the RUL for the battery.

```
estRUL = predictRUL(mdl,TestData)
```

```
estRUL = duration
   38.332 hr
```

Plot the survival function for the covariate data of the battery.

```
plot(mdl,TestData)
```



Predict RUL Using Reliability Survival Model and View PDF

Load training data.

```
load('reliabilityData.mat')
```

This data is a column vector of `duration` objects representing battery discharge times.

Create a reliability survival model, specifying the life time variable and life time units.

```
mdl = reliabilitySurvivalModel('LifeTimeVariable',"DischargeTime",'LifeTimeUnit',"hours");
```

Train the survival model using the training data.

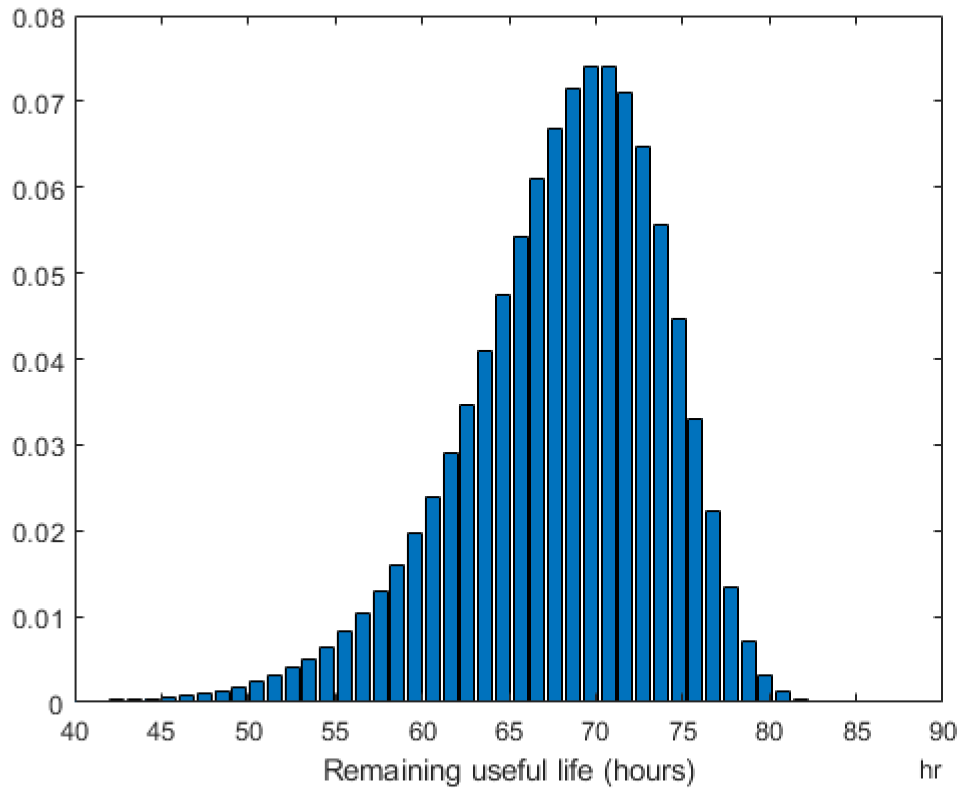
```
fit(mdl,reliabilityData)
```

Predict the life span of a new component, and obtain the probability distribution function for the estimate.

```
[estRUL,ciRUL,pdfRUL] = predictRUL(mdl);
```

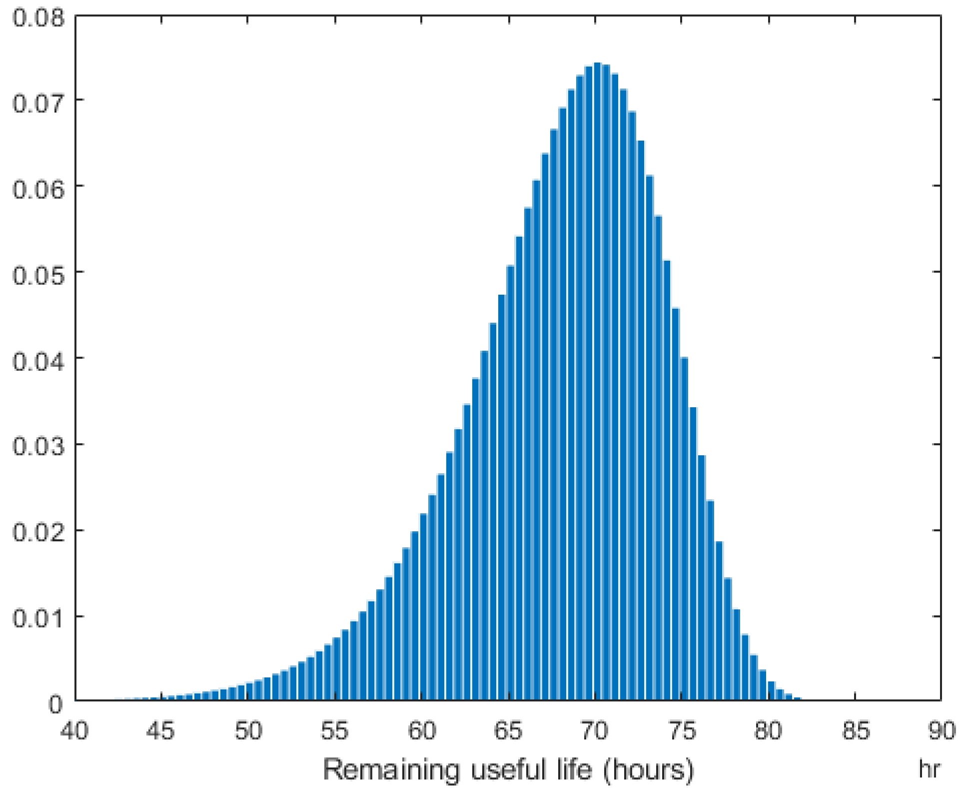
Plot the probability distribution.

```
bar(pdfRUL.RUL,pdfRUL.ProbabilityDensity)
xlabel('Remaining useful life (hours)')
xlim(hours([40 90]))
```



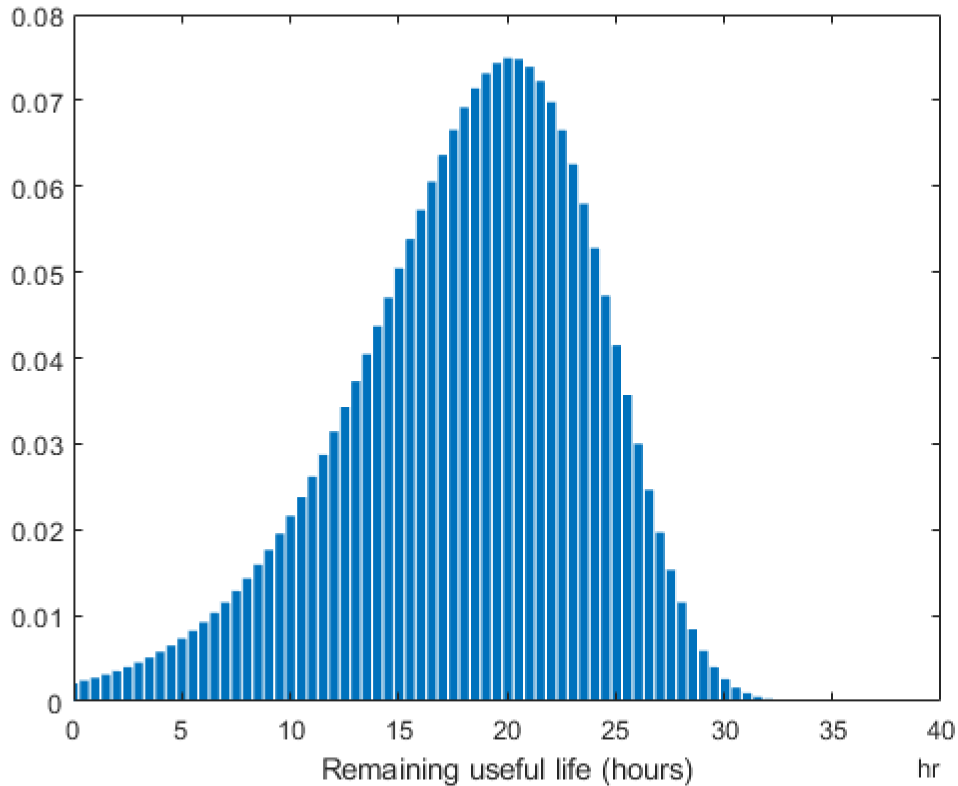
Improve the distribution view by providing the number of bins and bin size for the prediction.

```
[estRUL,ciRUL,pdfRUL] = predictRUL(mdl,'BinSize',0.5,'NumBins',500);
bar(pdfRUL.RUL,pdfRUL.ProbabilityDensity)
xlabel('Remaining useful life (hours)')
xlim(hours([40 90]))
```



Predict the RUL for a component that has been operating for 50 hours.

```
[estRUL,ciRUL,pdfRUL] = predictRUL(mdl,hours(50),'BinSize',0.5,'NumBins',500);
bar(pdfRUL.RUL,pdfRUL.ProbabilityDensity)
xlabel('Remaining useful life (hours)')
xlim(hours([0 40]))
```



Input Arguments

`mdl` — Remaining useful life prediction model
degradation model | survival model | similarity model

Remaining useful life prediction model, specified as one of the following models.

RUL Model Groups	More Information
Degradation models	`linearDegradationModel`
	`exponentialDegradationModel`
Survival models	`reliabilitySurvivalModel`
	`covariateSurvivalModel`
Similarity models	`hashSimilarityModel`
	`pairwiseSimilarityModel`
	`residualSimilarityModel`

For more information on the different model types and when to use them, see “Models for Predicting Remaining Useful Life”.

`data` — Degradation feature measurements
array | `table` | `timetable`

Degradation feature profiles for estimating the RUL using similarity models, measured over the life span of a component up to its current life time, specified as one of the following:

- (*N*+1)-by-*M* numeric array, where *N* is the number of features and *M* is the number of feature measurements. In each row, the first column contains the usage time and the remaining columns contain the corresponding degradation feature measurements. The order of the features must match the order specified in the `DataVariables` property of `mdl`.
- `table` or `timetable` object. The table must contain variables with names that match the strings in the `DataVariables` and `LifeTimeVariable` properties of `mdl`.

`data` applies when `mdl` is a `hashSimilarityModel`, `pairwiseSimilarityModel`, or `residualSimilarityModel`, object.

bounds — Degradation feature bounds
scalar | two-column array

Degradation feature bounds, which indicate the effective life span of a component, specified as an *N*-by-2 array, where *N* is the number of degradation features. For the *i*th feature, `bounds(i,1)` is the lower bound on the feature and `bounds(i,2)` is the upper bound. The order of the features must match the order specified in the `DataVariables` property of `mdl`.

Select `bounds` based on your knowledge of the allowable bounds for the degradation features.

`bounds` applies when `mdl` is a `hashSimilarityModel`, `pairwiseSimilarityModel`, or `residualSimilarityModel` object.

threshold — Data variable threshold
scalar

Data variable threshold limits for degradation models, specified as a scalar value. The remaining useful life is the remaining time before the forecasted response of the model reaches the threshold value.

The sign of the `Theta` property of `mdl` indicates the direction of degradation growth. If `Theta` is:

- Positive, then `threshold` is an upper bound on the degradation feature
- Negative, then `threshold` is a lower bound on the degradation feature

Select `threshold` based on your knowledge of the allowable bounds for the degradation feature.

`threshold` applies when `mdl` is a `linearDegradationModel` or `exponentialDesgradationModel` object.

usageTime — Current usage time
scalar | `duration` object

Current usage time of the component, specified as a scalar value or a `duration` object. The units of `usageTime` must be compatible with the `LifeTimeUnit` property of `mdl`.

covariates — Current covariate values and usage time
row vector | `table` with one row | `timetable` with one row

Current covariate values and usage time for the component, specified as a:

- Row vector whose first column contains the usage time. The remaining columns specify the component covariate values only and not the life time values. The number of covariate values must match the number and order of the covariate data columns used when estimating `mdl` using `fit`.
- `table` or `timetable` with one row. The table must contain the variables specified in the `LifeTimeVariable`, `DataVariables`, and `CensorVariable` properties of `mdl`.

If the covariate data contains encoded variables, then you must specify `covariates` using a `table` or `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: `Alpha,0.2` sets the prediction confidence interval to the `0.2/2` to `1-0.2/2` percentile region.

`Alpha` — Confidence level

`0.1` (default) | scalar value in the range 0 to 1

Confidence level for computing `ciRUL`, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range 0–1. `predictRUL` computes the confidence interval as the `Alpha`/2 to 1-`Alpha`/2 percentile region.

`NumBins` — Number of bins

`200` (default) | positive integer

Number of bins used to evaluate `pdfRUL`, specified as the comma-separated pair consisting of `'NumBins'` and a positive integer. This argument applies when `mdl` is a degradation model or survival model.

`BinSize` — Bin size

`1` (default) | positive scalar | `duration` object

Bin size used to determine the life span for computing `pdfRUL`, specified as the comma-separated pair consisting of `'BinSize'` and either a positive scalar or a `duration` object. This argument applies when `mdl` is a degradation model or reliability survival model.

`Method` — Survival function conversion method

`'empirical'` (default) | `'weibull'`

Survival function conversion method for generating the probability density function of a covariate survival model, specified as the comma-separated pair consisting of `'Method'` and one of the following:

- `'empirical'` — Generate `pdfRUL` by finding the gradient of the empirical cumulative distribution function. The cumulative distribution function is 1-$S(t)$, where $S(t)$ is the survival function.
- `'weibull'` — Generate `pdfRUL` by fitting a Weibull distribution to the survival function.

For more information on survival functions, see `covariateSurvivalModel`.

Output Arguments

estRUL — Estimated remaining useful life
scalar

Estimated remaining useful life of a component, returned as a scalar. The returned value is in the units of the life time variable as indicated by the LifeTimeUnit property of mdl.

ciRUL — Confidence interval
two-element row vector

Confidence interval associated with estRUL, returned as a two-element row vector. Specify the percentile for the confidence interval using Alpha.

pdfRUL — RUL probability density function
timetable | table

RUL probability density function, returned as a timetable if the life time variable of mdl is time-based, or as a table otherwise.

The life span used by predictRUL when computing the probability density function depends on the type of RUL model you specify. If mdl is a:

- Degradation model, then the life span is [usageTime usageTime+BinSize*NumBins].
- Reliability survival model, then the life span is [T T+BinSize*NumBins], where T is the usage time specified in usageTime.
- Covariate survival model, then the life span is linspace(T1,T2,NumBins), where [T1,T2] is the life range of components as determined by the BaslineCumulativeHazard property of mdl.
- Similarity model, then the life span depends on the life spans of the nearest neighbors found by the search algorithm. For example, if the NumNearestNeighbors property of mdl is 10 and the 10 nearest neighbors have life times in the range of 10 months to three years, then the histogram of failure times is found across this range. predictRUL then fits a probability density function to the raw histogram data using a kernel smoothing approach.

histRUL — Raw similarity scores
timetable | table

Raw similarity scores for histogram plotting, returned as a timetable if the life time variable of mdl is time-based, or as a table otherwise. histRUL has the following variables:

- 'RUL' — Remaining useful life values of historical components used to fit the parameters of mdl.
- 'NormalizedDistanceScore' — Similarity scores obtained by comparing the test component to the historical components used to fit the parameters of mdl.

The histogram of the data in histRUL is the unfitted version of pdfRUL. To plot the histogram, at the MATLAB command line, type:

```
bar(histRUL.RUL,histRUL.NormalizedDistanceScore)
```

histRUL is returned when mdl is a hashSimilarityModel, pairwiseSimilarityModel, or residualSimilarityModel object.

Compatibility Considerations

`currentValue` syntax not recommended

Not recommended starting in R2018b

The following syntax is not recommended:

```
estRUL = predictRUL(mdl,currentValue,threshold)
```

For a trained degradation model `mdl`, this syntax estimates the remaining useful life (RUL) based on the current measured value `currentValue` of a condition indicator. A more reliable way to estimate RUL for degradation models is to update the model with each successive measurement of the condition indicator using the `update` command. Then, use the updated model to estimate the RUL.

Update Code

Suppose that you store successive condition indicator measurements in an array `TestData`. The array contains measurements at regular intervals at least up to the time `currentTime` for which `currentValue` is the condition indicator measurement. To update your code, replace:

```
estRUL = predictRUL(mdl,currentValue,threshold)
```

with the following code:

```
for t = 1:CurrentTime
    update(mdl,TestData(t,:))
end
estRUL = predictRUL(mdl,threshold)
```

For an example, see “Predict RUL Using Exponential Degradation Model” on page 1-157.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

- This command supports code generation with MATLAB Coder™ for RUL prediction using `linearDegradationModel` or `exponentialDegradationModel`.
- Before generating code from a function that uses `predictRUL`, you must save the RUL model using `saveRULModelForCoder`. For an example, see “Generate Code for Predicting Remaining Useful Life”.

See Also

`fit` | `update`

Topics

“Update RUL Prediction as Data Arrives”
“RUL Estimation Using RUL Estimator Models”

Introduced in R2018a

prognosability

Measure of variability of condition indicators at failure

Syntax

```
Y = prognosability(X)
Y = prognosability(X,lifetimeVar)
Y = prognosability(X,lifetimeVar,dataVar)
Y = prognosability(X,lifetimeVar,dataVar,memberVar)
Y = prognosability( ___ ,Name,Value)

prognosability( ___ )
```

Description

`Y = prognosability(X)` returns the prognosability of the lifetime data `X`. Use `prognosability` as a measure of the variability of a feature at failure based on the trajectories of the feature measured in several run-to-failure experiments. A more prognosable feature has less variation at failure relative to the range between its initial and final values. The values of `Y` range from 0 to 1, where `Y` is 1 if `X` is perfectly prognosable and 0 if `X` is non-prognosable.

`Y = prognosability(X,lifetimeVar)` returns the prognosability of the lifetime data `X` using the lifetime variable `lifetimeVar`.

`Y = prognosability(X,lifetimeVar,dataVar)` returns the prognosability of the lifetime data `X` using the data variables specified by `dataVar`.

`Y = prognosability(X,lifetimeVar,dataVar,memberVar)` returns the prognosability of the lifetime data `X` using the lifetime variable `lifetimeVar`, the data variables specified by `dataVar`, and the member variable `memberVar`.

`Y = prognosability( ___ ,Name,Value)` estimates the prognosability with additional options specified by one or more `Name,Value` pair arguments. You can use this syntax with any of the previous input-argument combinations.

`prognosability( ___ )` with no output arguments plots a bar chart of ranked prognosability values.

Examples

Prognosability of Data in Cell Array of Matrices

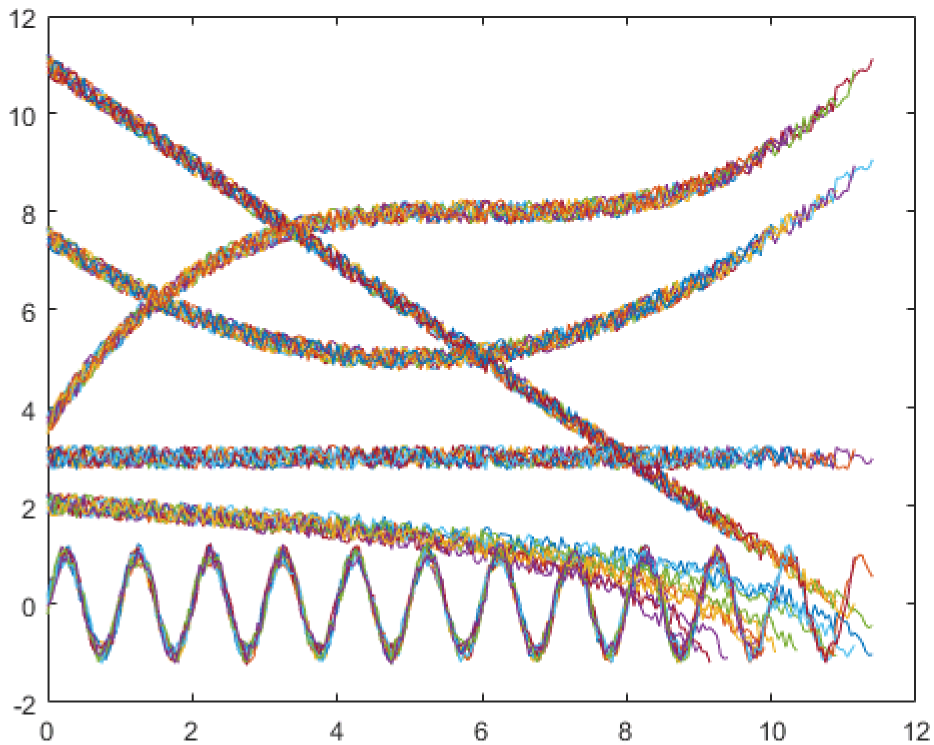
In this example, consider the lifetime data of 10 identical machines with the following 6 potential condition indicators−constant, linear, quadratic, cubic, logarithmic, and periodic. The data set `machineDataCellArray.mat` contains `C` which is a `1x10` cell array of matrices where each element of the cell array is a matrix that contains the lifetime data of a machine. For each matrix in the cell array, the first column contains the time while the other columns contain the data variables.

Load the lifetime data and visualize it against time.

```
load('machineDataCellArray.mat','C')
display(C)
```

```
C=1×10 cell array
  Columns 1 through 4

    {219x7 double}    {189x7 double}    {202x7 double}    {199x7 double}

  Columns 5 through 8

    {229x7 double}    {184x7 double}    {224x7 double}    {208x7 double}

  Columns 9 through 10

    {181x7 double}    {197x7 double}
```

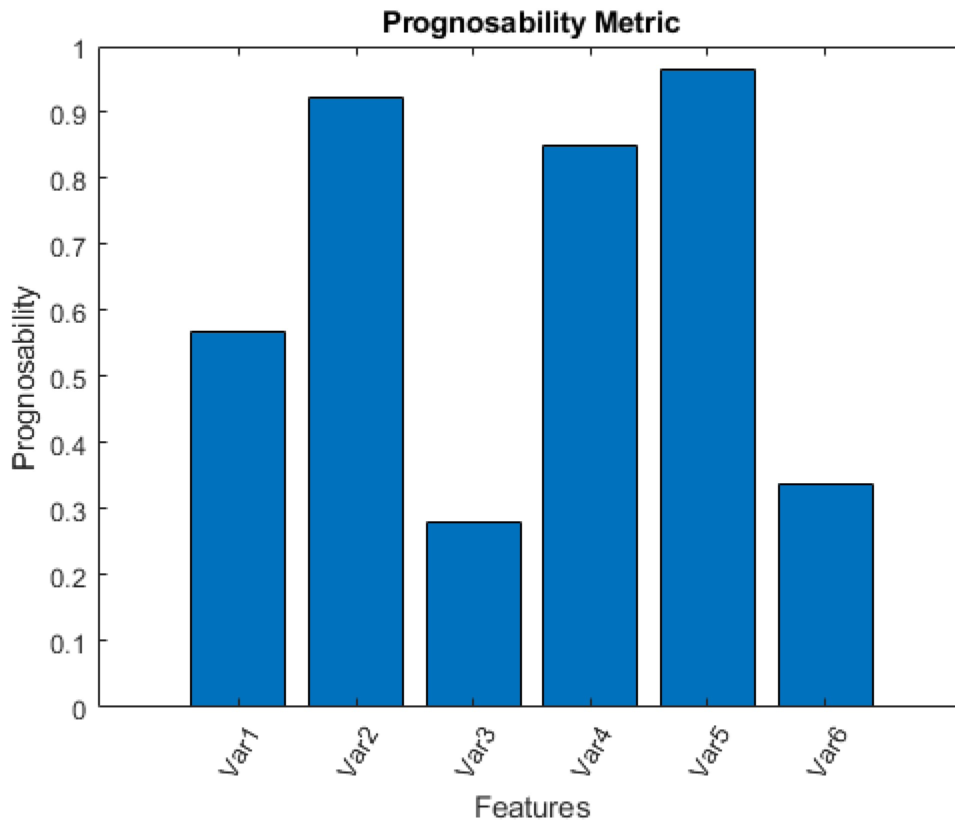
```
for k = 1:length(C)
    plot(C{k}(:,1), C{k}(:,2:end));
    hold on;
end
```



Observe the 6 different condition indicators–constant, linear, quadratic, cubic, logarithmic, and periodic–for all 10 machines on the plot.

Visualize the prognosability of the potential condition indicators.

```
prognosability(C)
```



From the histogram plot, observe that the features `Var2`, `Var4` and `Var5` rank better than the others. Hence, these features are more appropriate for remaining useful life predictions since they are the best indicators of machine health.

Prognosability of Data in Cell Array of Tables

In this example, consider the lifetime data of 10 identical machines with the following 6 potential condition indicators—constant, linear, quadratic, cubic, logarithmic, and periodic. The data set `machineDataTable.mat` contains `T`, which is a `1x10` cell array of tables where each element of the cell array contains a table of lifetime data for a machine.

Load and display the data.

```
load('machineDataTable.mat','T');
display(T)
```

```
T=1×10 cell array
  Columns 1 through 4

    {219x7 table}    {189x7 table}    {202x7 table}    {199x7 table}

  Columns 5 through 8

    {229x7 table}    {184x7 table}    {224x7 table}    {208x7 table}
```

```
  Columns 9 through 10

    {181x7 table}    {197x7 table}
```

```
head(T{1},2)
```

```
ans=2×7 table
    Time    Constant    Linear    Quadratic    Cubic     Logarithmic    Periodic
    ____    ________    ______    _________    ______    ___________    ________

       0     3.2029     11.203     7.7029      3.8829      2.2517        0.2029
    0.05     2.8135     10.763     7.2637      3.6006      1.8579       0.12251
```

Note that every table in the cell array contains the lifetime variable `'Time'` and the data variables `'Constant'`, `'Linear'`, `'Quadratic'`, `'Cubic'`, `'Logarithmic'`, and `'Periodic'`.

Compute prognosability with `Time` as the lifetime variable.

```
Y = prognosability(T,'Time')
```

```
Y=1×6 table
    Constant    Linear     Quadratic     Cubic     Logarithmic    Periodic
    ________    _______    _________    _______    ___________    ________

    0.56697     0.92321     0.28044     0.85048      0.96475      0.33853
```

From the resultant table of prognosability values, observe that the linear, cubic, and logarithmic features have values closer to 1. Hence, these three features are more appropriate for predicting remaining useful life since they are the best indicators of machine health.

Visualize Prognosability of Lifetime Data in Ensemble Datastore

Consider the lifetime data of 4 machines. Each machine has 4 fault codes for the potential condition indicators—voltage, current, and power. `prognosabilityEnsemble.zip` is a collection of 4 files where every file contains a timetable of lifetime data for each machine - `tbl1.mat`, `tbl2.mat`, `tbl3.mat` and `tbl4.mat`. You can also use files containing data for multiple machines. For each timetable, the organization of the data is as follows:

Time	Voltage	Current	Power	FaultCode	Machine

When you perform calculations on tall arrays, MATLAB® uses either a parallel pool (default if you have Parallel Computing Toolbox™) or the local MATLAB session. To run the example using the local MATLAB session, change the global execution environment by using the `mapreducer` function.

```
mapreducer(0)
```

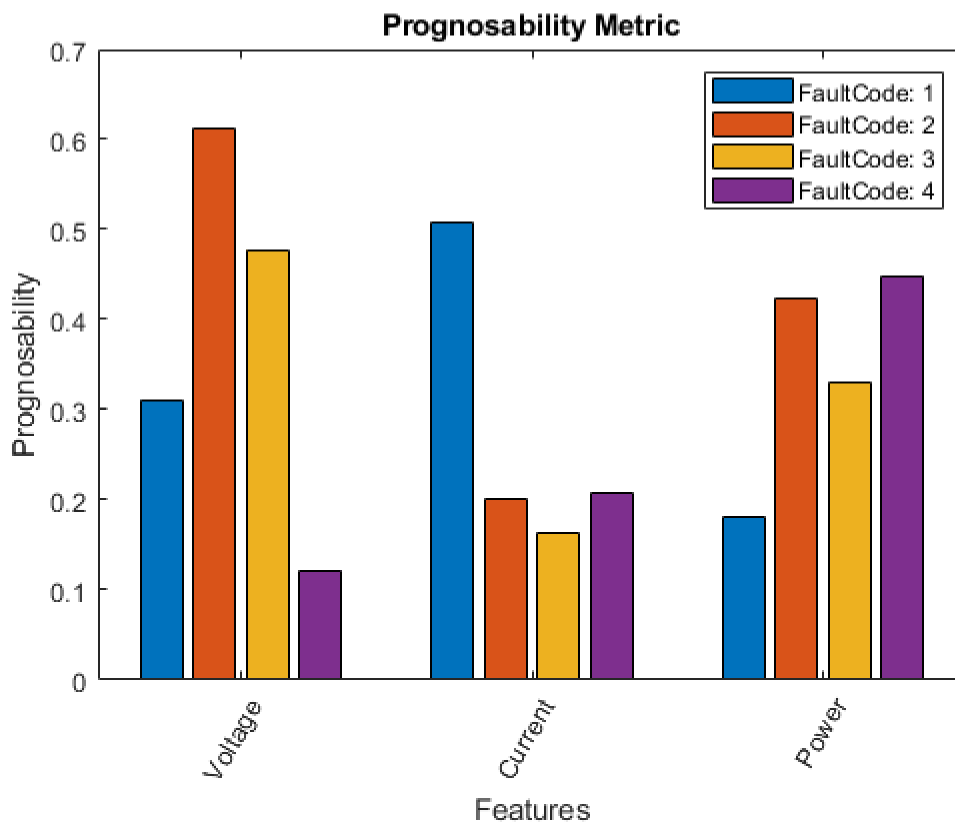
Extract the compressed files, read the data in the timetables, and create a `fileEnsembleDatastore` object using the timetable data. For more information on creating a file ensemble datastore, see `fileEnsembleDatastore`.

```
unzip prognosabilityEnsemble.zip;
ens = fileEnsembleDatastore(pwd,'.mat');
ens.DataVariables = {'Voltage','Current','Power','FaultCode','Machine'};
% Make sure that the function for reading data is on path
addpath(fullfile(matlabroot,'examples','predmaint','main'))
ens.ReadFcn = @readtable_data;
ens.SelectedVariables = {'Voltage','Current','Power','FaultCode','Machine'};
```

Visualize the prognosability of the potential condition indicators with `'Machine'` as the member variable and group the lifetime data by `'FaultCode'`. Grouping the lifetime data ensures that `prognosability` calculates the metric for each fault code separately.

```
prognosability(ens,'MemberVariable','Machine','GroupBy','FaultCode');
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.15 sec
Evaluation completed in 0.35 sec
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.073 sec
Evaluation completed in 0.22 sec
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.29 sec
Evaluation completed in 0.33 sec
```



`prognosability` returns a histogram plot with the features ranked by their prognosability values. A higher prognosability value indicates a more suitable condition indicator. For instance, the candidate feature `Current` has the highest degree of prognosability for machines with `FaultCode 1`.

```
rmpath(fullfile(matlabroot,'examples','predmaint','main')) % Reset path
```

Input Arguments

X — Lifetime data

cell array of matrices | cell array of tables and timetables | `fileEnsembleDatastore` object | table | timetable

Lifetime data, specified as a cell array of matrices, cell array of tables and timetables, `fileEnsembleDatastore` object, table, or timetable. Lifetime data contains run-to-failure data of the systems being monitored. The term *lifetime* here refers to the life of the machine defined in terms of the units you use to measure system life. Units of lifetime can be quantities such as the distance traveled (miles), fuel consumed (gallons), or time since the start of operation (days).

If `X` is

- a cell array of matrices or tables, the function assumes that each matrix or table contains columns of lifetime data for a system. Each column of every matrix or table, except the first column, contains data for a prognostic variable. `'Var1','Var2', ...` can be used to refer to the matrix columns that contain the lifetime data. For instance, the file `machineDataCellArray.mat` contains a 1-by-10 cell array of matrices `C`, where each of the 10 matrices contains data for a particular machine.
- a table or timetable, the function assumes that each column, except the first one, contains columns of lifetime data. The table variable names can be used to refer to the columns that contain the lifetime data. If `lifetimeVar` is not specified when `X` is a table, then the first data column is used as the lifetime variable.
- a `fileEnsembleDatastore` object, specify the data variables `dataVar` and member variables `memberVar` to be used. If `lifetimeVar` is not specified, then the first data column is used as the lifetime variable for computation.

Each numerical member in `X` is of type `double`.

lifetimeVar — Lifetime variable

string | character vector

Lifetime variable, specified as a string or character vector. `lifetimeVar` measures the lifetime of the systems being monitored and the lifetime data is sorted with respect to `lifetimeVar`. The value of `lifetimeVar` must be a valid ensemble or table variable name.

For a cell array of matrices, the value `'Time'` can be used to refer to the first column of each matrix, which is assumed to contain the lifetime variable. For instance, the file `machineDataCellArray.mat` contains the cell array `C`, where the first column in each matrix contains the lifetime variable while the other columns contain the data variables.

dataVar — Data variables

string array | character vector | cell array of character vectors

Data variables, specified as a string array, character vector, or cell array of character vectors. Data variables are the main content of the members of an ensemble. Data variables can include measured data or derived data for the analysis and development of predictive maintenance algorithms.

If `X` is

- a `fileEnsembleDatastore` object, the value of `dataVar` supersedes the `DataVariables` property of the ensemble.
- a cell array of matrices, the value `'Time'` can be used to refer to the first column of each matrix, that is, the lifetime variable `lifetimeVar`. `'Var1','Var2', ...` can be used to refer to the other matrix columns which contain the lifetime data. For instance, the file `machineDataCellArray.mat` contains the cell array `C` where the first column in each matrix contains the lifetime variable. The other columns in the cell array `C` contain the data variables.
- a table, the table variable names can be used to refer to the columns which contain the lifetime data.

The values of `dataVar` must be valid ensemble or table variable names. If `dataVar` is not specified, the computation includes all data columns except the one specified in `lifetimeVar`. For instance, suppose that each entry in a cell array is a table with variables `A`, `B`, `C`, and `D`. Setting `dataVar` to `["A","D"]` uses only `A` and `D` for the computation while `C` and `D` are ignored.

`memberVar` — Member variable
string | character vector

Member variable, specified as a string or character vector. Use `memberVar` to specify the variable for identifying the systems or machines in lifetime data `X`. For instance, in the `fileEnsembleDatastore` object, the fifth column in each timetable contains numbers that identify data from a particular machine. The column name corresponds to the member variable `memberVar`.

`memberVar` is ignored when `X` is specified as a cell array of matrices or tables.

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: `...,'WindowSize',0`

`LifeTimeVariable` — Lifetime variable
`strings(0)` (default) | string | character vector

Lifetime variable, specified as the comma-separated pair consisting of '`LifeTimeVariable`' and either a string or character vector. If '`LifeTimeVariable`' is not specified, then the first data column is used.

'`LifeTimeVariable`' is equivalent to the input argument `lifetimeVar`.

`DataVariables` — Data variables
`strings(0)` (default) | string array | character vector | cell array of character vectors

Data variables, specified as the comma-separated pair consisting of '`DataVariables`' and either a string array, character vector or cell array of character vectors.

'`DataVariables`' is equivalent to the input argument `dataVar`.

`MemberVariable` — Member variables
[] (default) | string | character vector

Member variables, specified as the comma-separated pair consisting of '`MemberVariable`' and either a string or character vector.

'MemberVariable' is equivalent to the input argument memberVar.

GroupBy — Grouping criterion
[] (default) | string | character vector

Grouping criterion, specified as the comma-separated pair consisting of 'GroupBy' and either a string or character vector. Use 'GroupBy' to specify the variables for grouping the lifetime data X by operating conditions.

The function computes the metric separately for each group that results from applying the criterion, such as a fault condition, specified by 'GroupBy'. For instance, in the fileEnsembleDatastore object ens, the fourth column in each timetable in ens contains the variable 'FaultCode'. The metric is computed for each machine by grouping the data by 'FaultCode'.

You can only group variables when X is defined as a fileEnsembleDatastore object, table, timetable, or cell array of tables or timetables.

Output Arguments

Y — Prognosability of lifetime data
vector | table

Prognosability of lifetime data, returned as a vector or table.

Prognosability is the measure of the variability of a feature at failure based on the trajectories of the feature measured in several run-to-failure experiments. A more prognosable feature has less variation at failure relative to the range between its initial and final values. As a system gets progressively closer to failure, a suitable condition indicator is typically highly prognosable. Conversely, any feature that is non-prognosable is a less suitable condition indicator. The values of Y range from 0 to 1.

- Y is 1 if X is perfectly prognosable.
- Y is 0 if X is perfectly non-prognosable.

Selecting appropriate estimation parameters out of all available features is the first step in building a reliable remaining useful life prediction engine. The prognosability values in Y are useful to determine which condition indicators best track the degradation process of systems being monitored. The higher the prognosability, the more desirable the feature is for RUL prediction.

When 'GroupBy' is not specified, then Y is returned as a row vector or single-row table. Conversely, when 'GroupBy' is specified, then each row in Y corresponds to one group.

Limitations

- When X is a tall table or tall timetable, prognosability nevertheless loads the complete array into memory using gather. If the memory available is inadequate, then prognosability returns an error.

Algorithms

The computation of prognosability uses this formula:

$$\text{prognosability} = \exp\left(-\frac{\text{std}_j\left(x_j(N_j)\right)}{\text{mean}_j\left|x_j(1) - x_j(N_j)\right|}\right), \qquad j = 1, \ldots, M$$

where x_j represents the vector of measurements of a feature on the jth system, variable M is the number of systems monitored, and N_j is the number of measurements on the jth system.

References

[1] Coble, J., and J. W. Hines. "Identifying Optimal Prognostic Parameters from Data: A Genetic Algorithms Approach." In *Proceedings of the Annual Conference of the Prognostics and Health Management Society*. 2009.

[2] Coble, J. "Merging Data Sources to Predict Remaining Useful Life - An Automated Method to Identify Prognostics Parameters." Ph.D. Thesis. University of Tennessee, Knoxville, TN, 2010.

[3] Lei, Y. *Intelligent Fault Diagnosis and Remaining Useful Life Prediction of Rotating Machinery*. Xi'an, China: Xi'an Jiaotong University Press, 2017.

[4] Lofti, S., J. B. Ali, E. Bechhoefer, and M. Benbouzid. "Wind turbine high-speed shaft bearings health prognosis through a spectral Kurtosis-derived indices and SVR." *Applied Acoustics* Vol. 120, 2017, pp. 1-8.

See Also

`monotonicity` | `trendability` | `fileEnsembleDatastore`

Topics

"Feature Selection for Remaining Useful Life Prediction"

Introduced in R2018b

read

Read member data from an ensemble datastore

Syntax

```
data = read(ensemble)
[data,info] = read(ensemble)
```

Description

Use this function to read data from ensemble datastores for condition monitoring and predictive maintenance.

`data = read(ensemble)` reads data from a member of the ensemble datastore `ensemble`. The function reads the variables specified in the `SelectedVariables` property of the ensemble datastore and returns them in a table.

If the ensemble has not been read since its creation (or since it was last reset using `reset`), then `read` reads data from the first member of the ensemble, as determined by the software. Otherwise, `read` reads data from the next ensemble member. `read` updates the `LastMemberRead` property of the ensemble to identify the most recently read member. For more information about how ensemble datastores work, see “Data Ensembles for Condition Monitoring and Predictive Maintenance”.

`[data,info] = read(ensemble)` also returns information about the location from which the data is read and the size of the data.

Examples

Extract Subset of Stored Variables from Ensemble Member

In general, you use the `read` command to extract data from a `simulationEnsembleDatastore` object into the MATLAB® workspace. Often, your ensemble contains more variables than you need to use for a particular analysis. Use the `SelectedVariables` property of the `simulationEnsembleDatastore` object to select a subset of variables for reading.

For this example, use the following code to create a `simulationEnsembleDatastore` object using data previously generated by running a Simulink® model at a various fault values (See `generateSimulationEnsemble`.). The ensemble includes simulation data for five different values of a model parameter, `ToothFaultGain`. Because of the volume of data, the `unzip` operation takes a few minutes.

```
unzip simEnsData.zip  % extract compressed files
ensemble = simulationEnsembleDatastore(pwd,'logsout')
```

```
ensemble = 
  simulationEnsembleDatastore with properties:

           DataVariables: [5x1 string]
    IndependentVariables: [0x0 string]
```

```
       ConditionVariables: [0x0 string]
        SelectedVariables: [5x1 string]
                 ReadSize: 1
               NumMembers: 5
           LastMemberRead: [0x0 string]
                    Files: [5x1 string]
```

The model that generated the data, `TransmissionCasingSimplified`, was configured such that the resulting ensemble contains variables including accelerometer data, `Vibration`, and tachometer data, `Tacho`. By default, the `simulationEnsembleDatastore` object designates all these variables as both data variables and selected variables, as shown in the `DataVariables` and `SelectedVariables` properties.

```
ensemble.DataVariables
```

```
ans = 5x1 string
    "PMSignalLogName"
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
```

```
ensemble.SelectedVariables
```

```
ans = 5x1 string
    "PMSignalLogName"
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
```

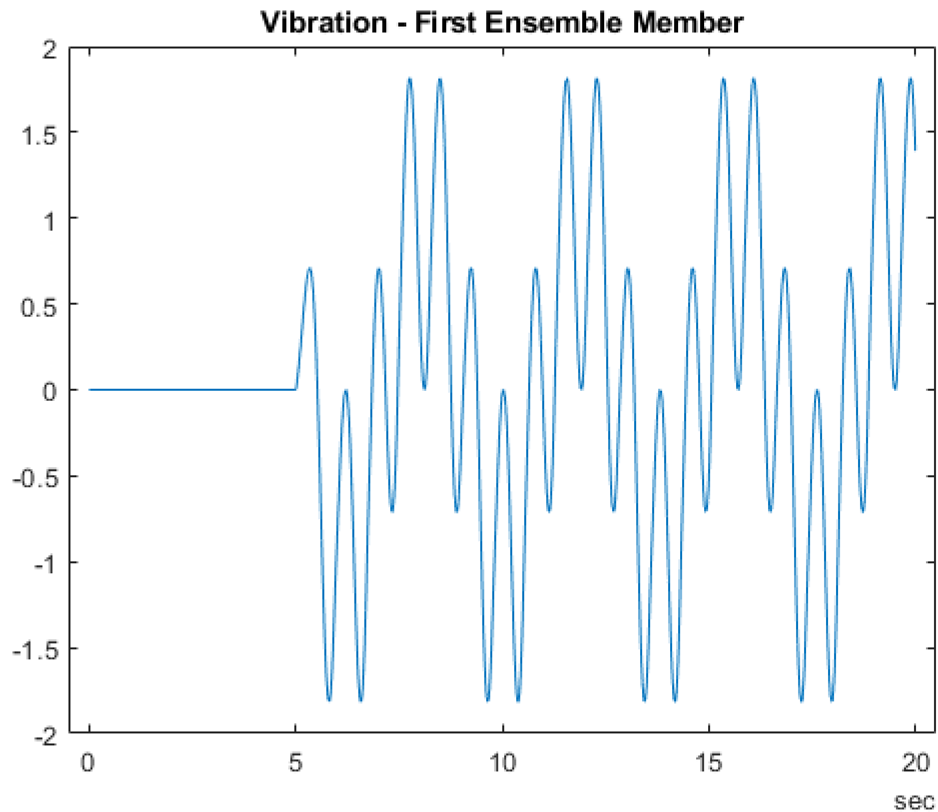
Suppose that for the analysis you want to do, you need only the `Vibration` data and the `Simulink.SimulationInput` object that describes the conditions under which this member data was simulated. Set `ensemble.SelectedVariables` to specify the variables you want to read. The `read` command then extracts those variables from the current ensemble member.

```
ensemble.SelectedVariables = ["Vibration";"SimulationInput"];
data1 = read(ensemble)
```

```
data1=1×2 table
         Vibration                 SimulationInput
    ___________________    ______________________________

    {20202x1 timetable}    {1x1 Simulink.SimulationInput}
```

`data.Vibration` is a cell array containing one `timetable` that stores the simulation times and the corresponding vibration signal. You can now process this data as needed. For instance, extract the vibration data from the table and plot it.

```
vibdata1 = data1.Vibration{1};
plot(vibdata1.Time,vibdata1.Data)
title('Vibration - First Ensemble Member')
```



The next time you call `read` on this ensemble, the last-read member designation advances to the next member of the ensemble (see “Data Ensembles for Condition Monitoring and Predictive Maintenance”). Read the selected variables from the next member of the ensemble.

```
data2 = read(ensemble)
```

```
data2=1×2 table
         Vibration                  SimulationInput
    __________________    ________________________________

    {20215x1 timetable}    {1x1 Simulink.SimulationInput}
```

To confirm that `data1` and `data2` contain data from different ensemble members, examine the values of the varied model parameter, `ToothFaultGain`. For each ensemble, this value is stored in the `Variables` field of the `SimulationInput` variable.

```
data1.SimulationInput{1}.Variables
```

```
ans =
  Variable with properties:

           Name: 'ToothFaultGain'
          Value: -2
      Workspace: 'global-workspace'
    Description: ""
```

```
data2.SimulationInput{1}.Variables
```

```
ans = 
  Variable with properties:

           Name: 'ToothFaultGain'
          Value: -1.5000
      Workspace: 'global-workspace'
    Description: ""
```

This result confirms that `data1` is from the ensemble member with `ToothFaultGain` = –2, and `data2` is from the member with `ToothFaultGain` = –1.5.

Read from and Write to a File Ensemble Datastore

Create a file ensemble datastore for data stored in MATLAB files, and configure it with functions that tell the software how to read from and write to the datastore. (For more details about configuring file ensemble datastores, see “File Ensemble Datastore With Measured Data”.)

```
% Create ensemble datastore that points to datafiles in current folder
unzip fileEnsData.zip  % extract compressed files
location = pwd;
extension = '.mat';
fensemble = fileEnsembleDatastore(location,extension);

% Specify data and condition variables
fensemble.DataVariables = ["gs";"sr";"load";"rate"];
fensemble.ConditionVariables = "label";

% Configure with functions for reading and writing variable data
addpath(fullfile(matlabroot,'examples','predmaint','main')) % Make sure functions are on path
fensemble.ReadFcn = @readBearingData;
fensemble.WriteToMemberFcn = @writeBearingData;
```

The functions tell the `read` and `writeToLastMemberRead` commands how to interact with the data files that make up the ensemble. Thus, when you call the `read` command, it uses `readBearingData` to read all the variables in `fensemble.SelectedVariables`. For this example, `readBearingData` extracts requested variables from a structure, `bearing`, and other variables stored in the file. It also parses the filename for the fault status of the data.

Specify variables to read, and read them from the first member of the ensemble.

```
fensemble.SelectedVariables = ["gs";"load";"label"];
data = read(fensemble)
```

```
data=1×3 table
     label             gs            load
    ________    _______________    ____

    "Faulty"    {5000x1 double}     0
```

You can now process the data from the member as needed. For this example, compute the average value of the signal stored in the variable `gs`. Extract the data from the table returned by `read`.

```
gsdata = data.gs{1};
gsmean = mean(gsdata);
```

You can write the mean value `gsmean` back to the data file as a new variable. To do so, first expand the list of data variables in the ensemble to include a variable for the new value. Call the new variable `gsMean`.

```
fensemble.DataVariables = [fensemble.DataVariables;"gsMean"]
```

```
fensemble = 
  fileEnsembleDatastore with properties:

                 ReadFcn: @readBearingData
        WriteToMemberFcn: @writeBearingData
           DataVariables: [5x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: "label"
       SelectedVariables: [3x1 string]
                ReadSize: 1
              NumMembers: 5
          LastMemberRead: "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\7\tp27f11553\predmaint-ex3416588
                   Files: [5x1 string]
```

Next, write the derived mean value to the file corresponding to the last-read ensemble member. (See “Data Ensembles for Condition Monitoring and Predictive Maintenance”.) When you call `writeToLastMemberRead`, it converts the data to a structure and calls `fensemble.WriteToMemberFcn` to write the data to the file.

```
writeToLastMemberRead(fensemble,'gsMean',gsmean);
```

Calling `read` again advances the last-read-member indicator to the next file in the ensemble and reads the data from that file.

```
data = read(fensemble)
```

```
data=1×3 table
     label            gs            load
    ________    _______________    ____

    "Faulty"    {5000x1 double}     50
```

You can confirm that this data is from a different member by examining the `load` variable in the table. Here, its value is 50, while in the previously read member, it was 0.

You can repeat the processing steps to compute and append the mean for this ensemble member. In practice, it is more useful to automate the process of reading, processing, and writing data. To do so, reset the ensemble to a state in which no data has been read. Then loop through the ensemble and perform the read, process, and write steps for each member.

```
reset(fensemble)
while hasdata(fensemble)
    data = read(fensemble);
    gsdata = data.gs{1};
    gsmean = mean(gsdata);
    writeToLastMemberRead(fensemble,'gsMean',gsmean);
end
```

The `hasdata` command returns `false` when every member of the ensemble has been read. Now, each data file in the ensemble includes the `gsMean` variable derived from the data `gs` in that file. You can use techniques like this loop to extract and process data from your ensemble files as you develop a predictive-maintenance algorithm. For an example illustrating in more detail the use of a file ensemble datastore in the algorithm-development process, see “Rolling Element Bearing Fault Diagnosis”. The example also shows how to use Parallel Computing Toolbox™ to speed up the processing of large data ensembles.

To confirm that the derived variable is present in the file ensemble datastore, read it from the first and second ensemble members. To do so, reset the ensemble again, and add the new variable to the selected variables. In practice, after you have computed derived values, it can be useful to read only those values without rereading the unprocessed data, which can take significant space in memory. For this example, read selected variables that include the new variable, `gsMean`, but do not include the unprocessed data, `gs`.

```
reset(fensemble)
fensemble.SelectedVariables = ["label";"load";"gsMean"];
data1 = read(fensemble)
```

```
data1=1×3 table
     label      load     gsMean 
    ________    ____    ________

    "Faulty"     0      -0.22648
```

```
data2 = read(fensemble)
```

```
data2=1×3 table
     label      load     gsMean 
    ________    ____    ________

    "Faulty"     50     -0.22937
```

```
rmpath(fullfile(matlabroot,'examples','predmaint','main')) % Reset path
```

Read Multiple Ensemble Members in One Operation

To read data from multiple ensemble members in one call to the `read` command, use the `ReadSize` property of an ensemble datastore. This example uses `simulationEnsembleDatastore`, but you can use the same technique for `fileEnsembleDatastore`.

Use the following code to create a `simulationEnsembleDatastore` object using data previously generated by running a Simulink model at a various fault values (see `generateSimulationEnsemble`). The ensemble includes simulation data for five different values of a model parameter, `ToothFaultGain`. (Because of the volume of data, the `unzip` operation might take a minute or two.) Specify some of the data variables to read.

```
unzip simEnsData.zip  % extract compressed files
ensemble = simulationEnsembleDatastore(pwd,'logsout');
ensemble.SelectedVariables = ["Vibration";"SimulationInput"];
```

By default, calling `read` on this ensemble datastore returns a single-row table containing the values of the `Vibration` and `SimulationInput` variables for the first ensemble member. Change the `ReadSize` property to read three members at once.

```
ensemble.ReadSize = 3;
data1 = read(ensemble)
```

```
data1=3×2 table
         Vibration                 SimulationInput
    ____________________    ________________________________

    {20202x1 timetable}    {1x1 Simulink.SimulationInput}
    {20215x1 timetable}    {1x1 Simulink.SimulationInput}
    {20204x1 timetable}    {1x1 Simulink.SimulationInput}
```

`read` returns a three-row table, where each row contains data from one of the first, second, and third ensemble members. `read` also updates the `LastReadMember` property of the ensemble datastore to a string array containing the paths of the three corresponding files. Avoid setting `ReadSize` to a value so large as to risk running out of memory while loading the data.

If the ensemble contains three or more additional members, the next `read` operation returns data from the fourth, fifth, and sixth members. Because the ensemble of this example contains only five members total, the next `read` operation returns only two rows.

```
data2 = read(ensemble)
```

```
data2=2×2 table
         Vibration                 SimulationInput
    ____________________    ________________________________

    {20213x1 timetable}    {1x1 Simulink.SimulationInput}
    {20224x1 timetable}    {1x1 Simulink.SimulationInput}
```

Input Arguments

`ensemble` — Ensemble datastore
`fileEnsembleDatastore` object | `simulationEnsembleDatastore` object

Ensemble datastore to read, specified as a:

- `simulationEnsembleDatastore` object — `read` reads the next ensemble member.
- `fileEnsembleDatastore` object — `read` uses the function specified in the `ensemble.ReadFcn` property to read the next ensemble member. (For more information about working with file ensemble datastores, see `fileEnsembleDatastore`.)

In either case, `read` returns a table containing all the variables specified in `ensemble.SelectedVariables`.

Output Arguments

`data` — Selected variables from ensemble member
`table`

Selected variables from the ensemble member, returned as a `table`. The table variables are the selected variables, and the table data are the values read from the ensemble data. By default, `read` reads one ensemble member at a time and returns a single table row.

To read multiple ensemble members at one time, set the `ReadSize` property of `ensemble` to a value greater than 1. For instance, if you set `ReadSize` to 3, then `read` reads the next 3 ensemble members and returns a table with 3 rows. If fewer than `ReadSize` members are unread, then `read` returns a table with as many rows as there are remaining members. For an example, see "Read Multiple Ensemble Members in One Operation" on page 1-181. Avoid setting `ReadSize` to such a large value as to risk running out of memory while loading data.

`info` — Data and member information
structure

Data and ensemble member information, returned as a structure with fields:

- `Size` — Dimensions of the table `data`, returned as a vector. For instance, if your ensemble has four variables specified in `ensemble.SelectedVariables`, then `Info.Size = [1 4]`.
- `FileName` — Path to the data file corresponding to the accessed ensemble member, returned as a string. For example, `"C:\Data\Experiment1\fault1.mat"`. Calling `read` also sets the `LastMemberRead` property of the ensemble to this value. If the `ReadSize` property of `ensemble` is greater than 1, this value is a string vector containing the paths to all the accessed files.

See Also

`simulationEnsembleDatastore` | `fileEnsembleDatastore`

Topics
"Data Ensembles for Condition Monitoring and Predictive Maintenance"

Introduced in R2018a

readFeatureTable

Read feature values, independent variables, and condition variables from an ensemble data set into a table

Syntax

```
ft = readFeatureTable(ensemble)
ft = readFeatureTable(ensemble,framepolid)
ft = readFeatureTable( ___ ,Name,Value)
```

Description

`readFeatureTable` is a function used in code generated by **Diagnostic Feature Designer**.

`ft = readFeatureTable(ensemble)` extracts a feature table `ft` from ensemble `ensemble` for all features computed in full-signal mode. The feature table contains features, independent variables, and condition variables, and is the primary output of code generated by **Diagnostic Feature Designer**.

`ft = readFeatureTable(ensemble,framepolid)` uses the frame size and frame rate defined in `FramePolicyID` to read each frame interval when the function constructs the feature table. This syntax applies to frame-based—also known as segmented—signal and feature computation.

`ft = readFeatureTable( ___ ,Name,Value)` specifies the features and variables to read using one or more name-value pair arguments. For instance, if you use `ft = readFeatureTable(ensemble,'ConditionVariables','FaultCode')`, `ft` contains only the `'FaultCode'` condition variable but still includes all features and independent variables. You can use this syntax with any of the input argument combinations in previous syntaxes.

Input Arguments

`ensemble` — Ensemble of member data
`workspaceEnsemble` object | `fileEnsembleDatastore` object | `simulationEnsembleDatastore` object

Ensemble of member data, specified as a `fileEnsembleDatastore` object, a `simulationEnsembleDatastore` object, or a `workspaceEnsemble` object.

`framepolid` — Frame policy ID
string

Frame policy ID, specified as a string formatted as `FRM_`<frame policy index>. In code generated by **Diagnostic Feature Designer**, the frame policy reflects the choice of frame size and frame rate in segmented data.

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: 'ConditionVariables','FaultCode'

Features — Features to read

string array

Features to read, specified as the comma-separated pair consisting of 'Features' and a string array of paths that point to the selected features. If you do not specify 'Features', the function reads all the features.

IndependentVariables — Independent variables to read

string array

Independent variables to read, specified as the comma-separated pair consisting of 'IndependentVariables' and a string array of paths that point to the selected variables. If you do not specify 'IndependentVariables', the function reads all the independent variables.

ConditionVariables — Condition variables to read

string array

Condition variables to read, specified as the comma-separated pair consisting of 'ConditionVariables' and a string array of paths that point to the selected variables. If you do not specify 'ConditionVariables', the function reads all the condition variables.

IncludeMemberID — Option to return member IDs

false (default) | true

Option to return ensemble member IDs, specified as the comma-separated pair consisting of 'IncludeMemberID' and a logical scalar. When you set 'IncludeMemberID' to true, the feature table ft includes a column of member IDs.

Output Arguments

ft — Feature table

table

Feature table, specified as a table. The table contains features, independent variables, and condition variables for each member. The features and condition variables are scalars. The independent variables are timetables, tables, or cell arrays.

See Also

Diagnostic Feature Designer

Topics

"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a

readFrameIntervals

Extract frame segments from an ensemble member

Syntax

```
frames = readFrameIntervals(memberdata)
frames = readFrameIntervals(memberdata,framepolid)
```

Description

`readframeIntervals` is a function used in code generated by **Diagnostic Feature Designer**.

`frames = readFrameIntervals(memberdata)` extracts the start and stop times for each frame into a table for the first frame policy in a set of frame policies. A frame policy specifies the frame size and frame rate, and enables frame-based rather than full-signal signal processing and feature extraction.

Code that is generated by **Diagnostic Feature Designer** uses `readFrameIntervals` when it processes inputs with both full and framed signals, such as when the code includes ensemble statistics processing and frame-based signal and feature processing.

`frames = readFrameIntervals(memberdata,framepolid)` extracts the frame segments using the specified frame policy ID. For instance, `readFrameIntervals(ensembleStatistics,"FRM_2")` extracts the intervals using the second frame policy.

Input Arguments

`memberdata` — Member data set
table row | cell array row

Member data set, specified as a row within an ensemble data set.

`framepolid` — Frame policy ID
string

Frame policy ID, specified as a string formatted as `FRM_`<frame policy index>. For instance, `"FRM_2"`.

Output Arguments

`frames` — Frame start and stop times
table

Frame start and stop times, specified as an *nf*-by-2 table, where *nf* is the number of frames in the signal.

See Also

`frameintervals` | `joindata` | **Diagnostic Feature Designer**

Topics
"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a

readMember

Return ensemble member data based on the member index

Syntax

```
data = readMember(wensemble,index)
```

Description

`readMember` is a function used in code generated by **Diagnostic Feature Designer**.

`data = readMember(wensemble,index)` reads the workspace ensemble `wensemble` member identified by `index` and returns member data in `data`. `readMember` reads only variables that the `'SelectedVariables'` property of `wensemble` specifies.

Code that is generated by **Diagnostic Feature Designer** uses `writeMember`, `readMember`, and `findIndex` under the following conditions:

- The input data is an ensemble datastore, such as a file or simulation ensemble datastore.
- The computation option during code generation specified storing results in local memory rather than writing results back to the ensemble datastore.

Explicitly specifying a member index when reading and writing within the local version of the data, which the code manages using a `workspaceEnsemble` object, ensures member synchronization with the original ensemble datastore. This synchronization is necessary when you have sequential member-processing loops, such as when you compute ensemble statistics as a precursor to computing signal residues.

- During the first member-processing loop, which starts with an empty ensemble, no indexing is needed. The code appends each new member result to the end of the ensemble.
- During the second loop, the index enables the code to write updated member results to the correct location within the now-populated ensemble.

For more information about the dual processing loop for ensemble statistics, see "Anatomy of App-Generated MATLAB Code".

Input Arguments

`wensemble` — Ensemble object

`workspaceEnsemble` object

Ensemble object, specified as a `workspaceEnsemble` object. `wensemble` contains ensemble data and specifies the variable names and types within the ensemble, such as data variables and condition variables.

`index` — Member index

positive integer

Member index, specified as a positive integer. `index` identifies the ensemble member to read new data from.

Output Arguments

`data` — Member data
single-row table

Member data that `readMember` extracts, returned as a single-row table.

See Also

Diagnostic Feature Designer | `fileEnsembleDatastore` | `simulationEnsembleDatastore` | `workspaceEnsemble` | `findIndex` | `writeMember`

Topics

"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a

readMemberData

Extract data from an ensemble member given a path

Syntax

```
data = readMemberData(memberdata,path)
data = readMemberData(memberdata,path,variablenames)
data = readMemberData( ___ ,'FrameInterval',interval)
```

Description

`readMemberData` is a function used in code generated by **Diagnostic Feature Designer**.

`data = readMemberData(memberdata,path)` reads the value under the path `path` from member data `memberdata`.

Code that is generated by **Diagnostic Feature Designer** uses `readMemberData` when performing member-level processing.

`data = readMemberData(memberdata,path,variablenames)` reads the values of the variable names in `variablenames`. For example, `Vibration = readMemberData(member,"Vibration",["Time","Data"])` reads the `Vibration` variables `Time` and `Data` from the current member row.

`data = readMemberData( ___ ,'FrameInterval',interval)` reads the values contained within a frame interval that is specified by the frame start and stop times. Use this syntax when `path` starts with `FRM`, which indicates that the data to be read is segmented into frames. You can use this syntax with any of the input argument combinations in previous syntaxes.

Input Arguments

`memberdata` — Member data set
table

Member data set, specified as a table. `memberdata` represents one ensemble member read from a multimember ensemble data set.

`path` — path name
scalar string | character array

Path, specified as a scalar string or a character array that represents the highest level variable name. For example, `"Vibration"` is a path that might contain the lower level variable names `time` and `data`.

`variablenames` — Variable names
scalar string | character array | string array | cell array of character arrays

Variable names under a path, specified as a scalar string, a character array, a string array, or a cell array of character arrays. For example, `["Time","Data"]` might be variable names under the path `"Vibration"`.

`interval` — Frame start and stop times
two-element array

Frame start and stop times, specified as an array with two elements. When reading frame-based data, `interval` identifies a specific frame within the frame sequence. For example, if the full signal ranges from 0 to 20 seconds, and the frame size and frame rate specifications are each one second, the first interval is approximately `[0 1]`.

Output Arguments

`data` — Member data values
table

Member data values extracted from the member data, returned as a table. `data` contains the same column names as `variablenames` if `variablenames` is a string array or a cell array of character arrays.

See Also

Diagnostic Feature Designer

Topics
"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a

readState

Get degradation RUL model state for use at runtime

Syntax

```
mdlState = readState(mdl)
```

Description

`mdlState = readState(mdl)` returns a structure containing the properties of the degradation RUL model `mdl`. Use `readState` in an entry-point function for code generation to preserve the values of model parameters, particularly when you update the model at run time. For more information, see "Generate Code that Preserves RUL Model State for System Restart".

Input Arguments

`mdl` — RUL model
`linearDegradationModel` | `exponentialDegradationModel`

RUL model, specified as a `linearDegradationModel` or `exponentialDegradationModel` RUL model object. `readState` creates a structure that contains the current property values of `mdl`.

Output Arguments

`mdlState` — Model state
structure

Model state, returned as a structure. The fields of `mdlState` correspond to the properties of `mdl`, with an extra field that specifies the type of RUL model.

Extended Capabilities

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

See Also

`restoreState` | `saveRULModelForCoder`

Topics
"Generate Code that Preserves RUL Model State for System Restart"

Introduced in R2021a

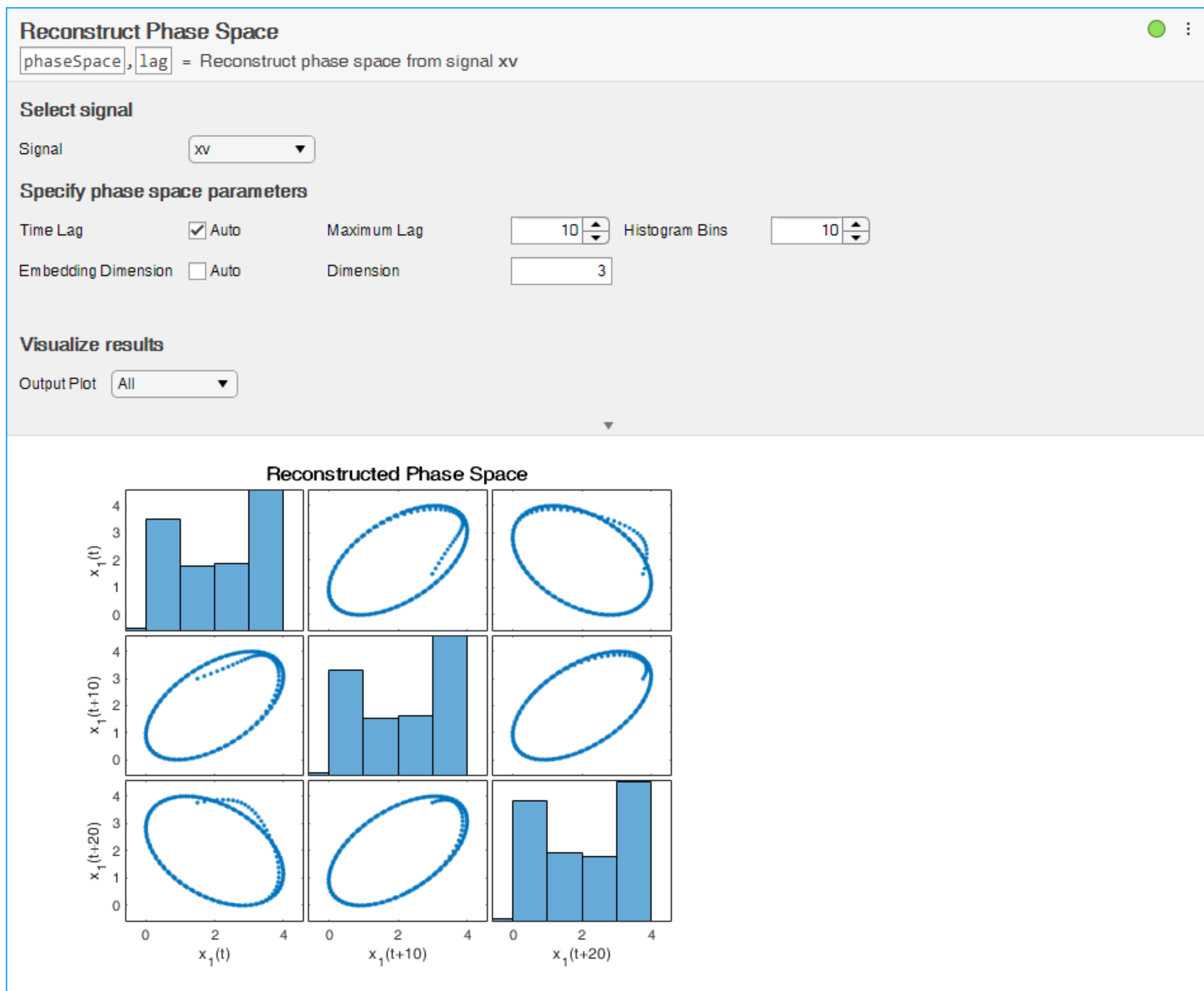
Reconstruct Phase Space

Reconstruct phase space of a uniformly sampled signal in the Live Editor

Description

The **Reconstruct Phase Space** task lets you interactively reconstruct phase space of a uniformly sampled signal. The task automatically generates MATLAB code for your live script. For more information about Live Editor tasks generally, see "Add Interactive Tasks to a Live Script".

Phase space reconstruction is useful to verify the system order and reconstruct all dynamic system variables, while preserving system properties. Reconstructing the phase space is performed when limited data is available, or when the phase space dimension and lag values are unknown. Also, the nonlinear features `approximateEntropy`, `correlationDimension`, and `lyapunovExponent` use phase space reconstruction as the first step of the computation. For more information about phase space reconstruction, see `phaseSpaceReconstruction`.



Open the Task

To add the **Reconstruct Phase Space** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task** > **Reconstruct Phase Space**.
- In a code block in your script, type a relevant keyword, such as `phase` or `phase space`. Select `Reconstruct Phase Space` from the suggested command completions.

Examples

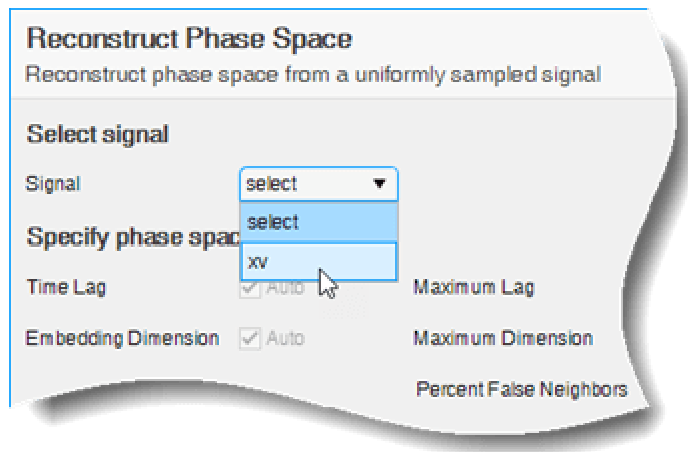
Reconstruct Phase Space in the Live Editor

Use the **Reconstruct Phase Space** task in the Live Editor to interactively reconstruct the phase space of a uniformly sampled signal. Experiment with different values for lag, embedding dimension, histogram bins and distance threshold. The task automatically generates code reflecting your selections. Open this example to see a preconfigured script containing the **Reconstruct Phase Space** task.

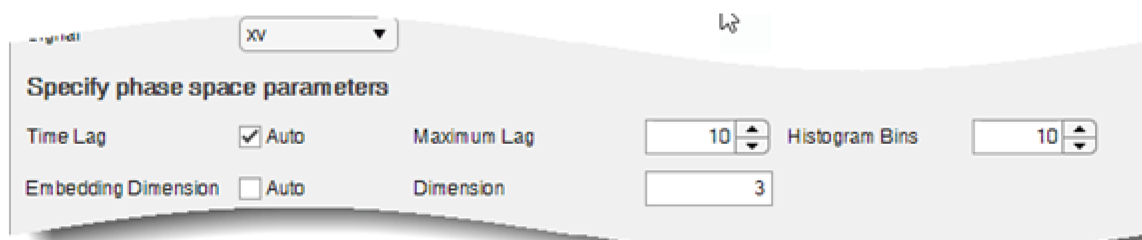
For this example, consider '`uavPositionData.mat`' which contains signal xv which is the x-component of a 3-D path traversed by an unmanned aerial vehicle (UAV). The x, y, and z coordinates define a circle of 2-m radius at 0.75-m altitude.

```
load('uavPositionData.mat','xv')
```

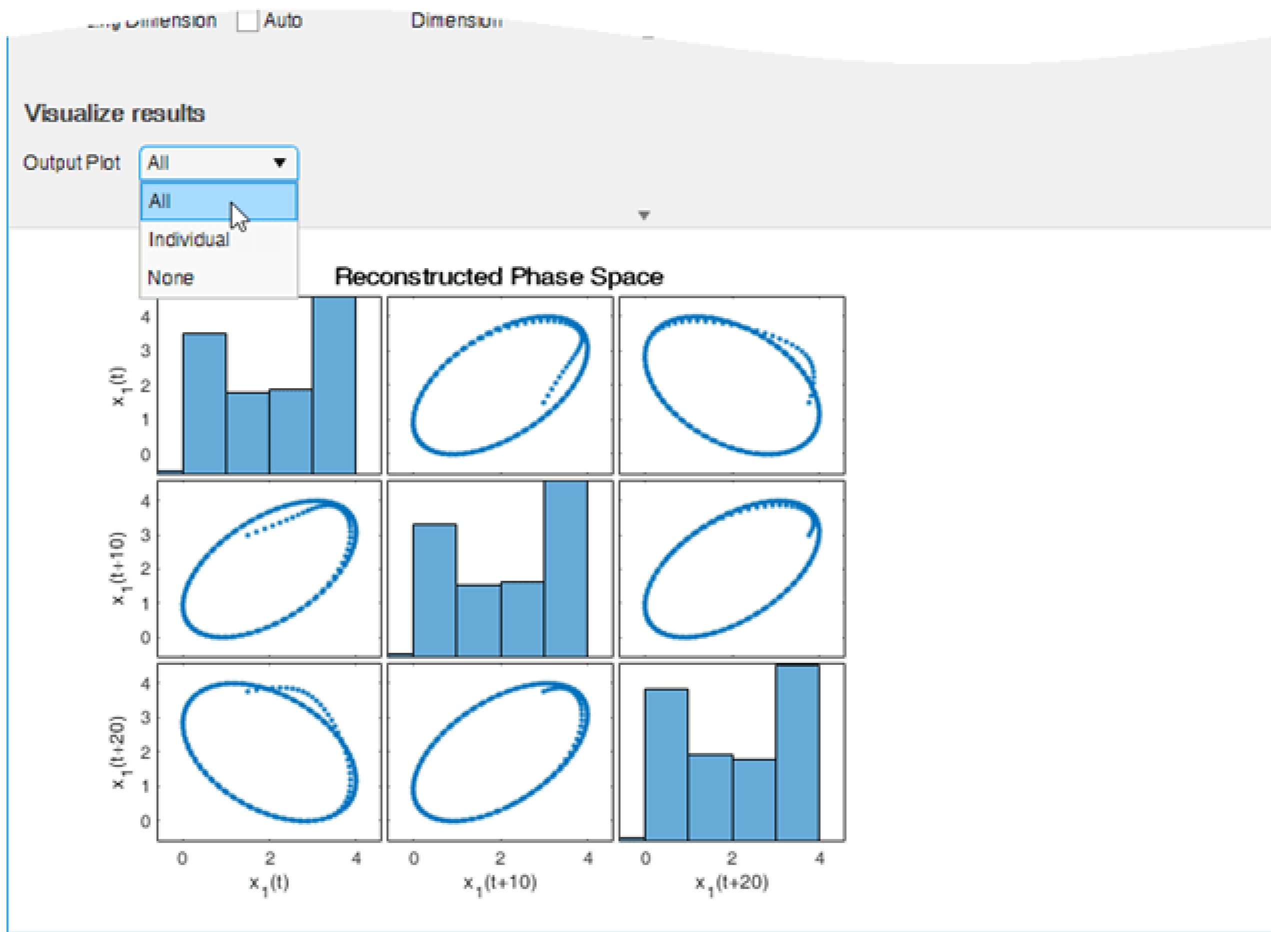
To reconstruct the phase space of the signal `xv`, open the **Reconstruct Phase Space** task in the Live Editor. On the **Live Editor** tab, select **Task > Reconstruct Phase Space**. In the task, select signal `xv`.

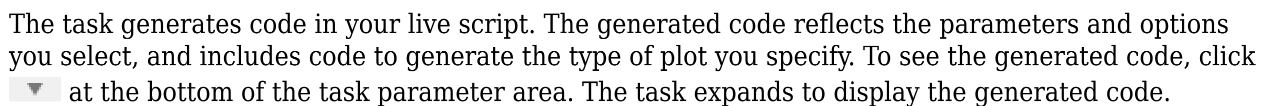


Clear the **Time Lag** check box if you want to use your own values in the **Maximum Lag** and **Histogram Bins** fields. For this example, leave the box checked to calculate the lag using Average Mutual Information (AMI). Since dimension is known, clear the **Embedding Dimension** field and specify dimension as 3.



Evaluate whether the reconstructed phase space preserves the system dynamics with the assigned values by observing the output plots. You can toggle between the display type by choosing between `Individual` or `All` in the **Output Plot** dropdown menu.





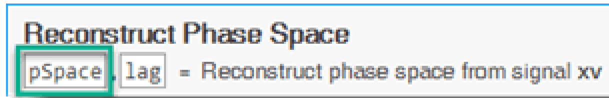
The task generates code in your live script. The generated code reflects the parameters and options you select, and includes code to generate the type of plot you specify. To see the generated code, click ▼ at the bottom of the task parameter area. The task expands to display the generated code.

Output Plot Individual

```
% Reconstruct phase space from signal xv
[phaseSpace, lag] = phaseSpaceReconstruction(xv, [], 3, ...
    'MaxLag', 10, ...
    'HistogramBins', 10);

% Visualize results
phasespaceplot(phaseSpace, lag, 3, size(xv,2), 2);
```

Reconstructed Phase Space

By default, the generated code uses `phaseSpace` as the name of the output variable. To specify a different output variable name, enter a new name in the summary line at the top of the task. For instance, change the name to `pSpace`.



The task updates the generated code to reflect the new variable name, and the new variable `pSpace` appears in the MATLAB workspace. You can use the reconstructed phase space to identify condition indicators like Lyapunov exponent or correlation dimension.

- "Reconstruct Phase Space and Estimate Condition Indicators Using Live Editor Tasks"

Parameters

Select Signal

`Signal` — Uniformly sampled time-domain signal

array | timetable

Select a uniformly sampled time-domain signal in array or timetable format.

Specify Phase Space Parameters

`Time Lag` — Check to use Average Mutual Information (AMI) algorithm to compute time lag

on (default) | off

Check to use Average Mutual Information (AMI) algorithm to compute time lag. Clear to try your own value of **Maximum Lag** and **Histogram Bins**. If the time delay is too small, random noise is introduced in the states. In contrast, if the lag is too large, the reconstructed dynamics do not represent the true dynamics of the time series.

`Maximum Lag` — Maximum value of lags used in the lag estimation

positive scalar

Maximum value of lag used to estimate the time delay using the Average Mutual Information (AMI) algorithm.

`Histogram Bins` — Number of bins for discretization when computing the average mutual information

positive scalar

Number of bins for discretization to compute lag using the AMI algorithm. Set the value of **Histogram Bins** based on the length of your signal.

`Embedding Dimension` — Check to use Percent False Neighbors (PFN) algorithm to compute embedding dimension

on (default) | off

Check to use Percent False Neighbors (PFN) algorithm to automatically compute embedding dimension.

`Maximum Dimension` — Maximum value of embedding dimension used in the dimension estimation
positive scalar

Maximum value of embedding dimension used in the dimension estimation with Percent False Neighbors (PFN) algorithm.

`Distance Threshold` — Distance ratio threshold for determining two points as false neighbors
scalar

Distance ratio threshold for determining two points as false neighbors using Percent False Neighbors (PFN) algorithm. For more information, see `phaseSpaceReconstruction`.

`Percent False Neighbors` — Percent false neighbors threshold for detecting embedding dimension
scalar

Percent false neighbors threshold for detecting embedding dimension using PFN algorithm. To specify percent false neighbors, check the **Embedding Dimension** check box. For more information, see `phaseSpaceReconstruction`.

Visualize Results

`Output Plot` — Number of output plots to display
`Individual` (default) | `All` | `None`

Number of output plots to display. To toggle between the reconstructed plot and the histogram plot, and to go through each plot, select `Individual`. To display both plots in the Live Editor, select `All`. To hide plots, select `None`.

See Also

Estimate Approximate Entropy | **Estimate Correlation Dimension** | **Estimate Lyapunov Exponent** | `correlationDimension` | `phaseSpaceReconstruction` | `lyapunovExponent` | `approximateEntropy`

Topics

"Reconstruct Phase Space and Estimate Condition Indicators Using Live Editor Tasks"
"Add Interactive Tasks to a Live Script"

Introduced in R2019b

refresh

Update a workspace ensemble with partitions of modified or added data computed in parallel processing

Syntax

```
refresh(wensemble,ensarray)
```

Description

`refresh` is a function used in code generated by **Diagnostic Feature Designer**.

`refresh(wensemble,ensarray)` updates the workspace ensemble object `wensemble` with the combined data partitions in `ensarray`.

To enable parallel processing, generated code creates data partitions that allow operations to run simultaneously. In code generated by **Diagnostic Feature Designer**, `refresh` updates `wensemble` at the conclusion of parallel processing when the computation of all variables and features for all partitions is complete. `refresh` reassembles the partitioned results that are stored in `ensarray` and replaces the original contents of `wensemble` with the new values.

Input Arguments

`wensemble` — Ensemble object
`workspaceEnsemble` object

Ensemble object, specified as a `workspaceEnsemble` object. `wensemble` contains ensemble data and specifies the variable names and types within the ensemble, such as data variables and condition variables.

`ensarray` — New or updated ensemble data
cell array of `workspaceEnsemble` partitions

New or updated ensemble data, specified as a cell array of `workspaceEnsemble` parallel-processing partitions.

See Also

Diagnostic Feature Designer | `partition`

Topics

"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a

relativeEntropy

One-dimensional Kullback-Leibler divergence of two independent data groups to measure class separability

Syntax

```
Z = relativeEntropy(X,I)
```

Description

`relativeEntropy` is a function used in code generated by **Diagnostic Feature Designer**.

`Z = relativeEntropy(X,I)` calculates the one-dimensional Kullback-Leibler divergence of two independent subsets of data set `X` that are grouped according to the logical labels in `I`. The relative entropy provides a metric for ranking features according to their ability to separate two classes of data, such as healthy and faulty machines. The entropy calculation assumes that the data in `X` follows a Gaussian distribution.

Code that is generated by **Diagnostic Feature Designer** uses `relativeEntropy` when ranking features with this method.

Input Arguments

X — Data samples to group

vector | matrix

Data set containing data samples that can be logically classified into two groups, specified as a vector when you have a single set of samples, such as values for one feature, and a matrix when you have multiple sets of samples.

- When `X` contains a single set of *n* features, such as a multiple features extracted from a single data source, `X` is a 1-by-*n* vector.
- When `X` contains *m* sets of *n* features, `X` is an *m*-by-*n* matrix. Each row in `X` represents one data source and must correspond to a single logical class.

`X` must contain at least two rows that correspond to the logical class in `I` of `0` and two rows that correspond to the label `1` to calculate legitimate relative entropy values.

For example, suppose that you have a set of five features for each of 20 gearboxes and you are computing the relative entropy to assess these features. `X` is a 20-by-5 matrix. Each row represents a gearbox that is either healthy or faulty, as indicated by the associated logical class label of `0` or `1`. At least two gearboxes must be healthy and at least two gearboxes must be faulty. The relative entropy indicates how well each feature separates the data for the healthy gearboxes from the data for the faulty gearboxes.

I — Logical classification label

vector | matrix

Logical classification label that assigns the rows in `X` to one of two logical classes, specified as a vector of length *m*, where *m* is the number of rows in `X`.

For example, suppose once more that X is a 20-by-5 matrix corresponding to 20 gearboxes. The first 9 gearboxes are healthy. The remaining 11 gearboxes are faulty. Define the healthy state as 0 and the faulty state as 1. Then I has a length of 20. The first 9 labels in I are equal to 0 and the remaining 11 labels are equal to 1.

Output Arguments

Z — Relative entropy

scalar | vector

Relative entropy of two labeled groups, returned as a scalar or a vector.

- If X is a vector, then Z is a scalar.
- If X is a matrix, then `relativeEntropy` calculates the distance separately for each feature. Z is then a vector of length *n*, where *n* is the number of columns in Z.

`relativeEntropy` treats NaN entries in X as missing values and ignores them.

References

[1] Theodoridis, Sergios, and Konstantinos Koutroumbas. *Pattern Recognition*, 175–177. 2nd ed. Amsterdam; Boston: Academic Press, 2003.

See Also

`correlationWeightedScore` | **Diagnostic Feature Designer**

Topics

"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a

restart

Reset remaining useful life degradation model

Syntax

```
restart(mdl)
restart(mdl,resetPrior)
restart( ___ ,Name,Value)
```

Description

`restart(mdl)` resets the internally stored statistics of the degradation process accumulated by the previous calls to `update` and resets the `InitialLifeTimeValue` and `CurrentLifeTimeValue` properties of the model. If the `SlopeDetectionLevel` property of the model is not empty, then the slope test is also restarted, ignoring any previous detections.

`restart(mdl,resetPrior)` sets the prior parameter values in `mdl` to their corresponding posterior values when `resetPrior` is `true`.

`restart( ___ ,Name,Value)` specifies properties of `mdl` using one or more name-value pair arguments.

Examples

Reset Degradation Model

Load training data, which is a degradation feature profile for a component.

```
load('expRealTime.mat')
```

For this example, assume that the training data is not historical data. When there is no historical data, you can update your degradation model in real time using observed data.

Create an exponential degradation model with the following settings:

- θ prior distribution with a mean of `2.4` and a variance of `0.006`
- β prior distribution with a mean of `0.07` and a variance of `3e-5`
- Noise variance of `0.003`

```
mdl = exponentialDegradationModel('Theta',2.4,'ThetaVariance',0.006,...
                                  'Beta',0.07,'BetaVariance',3e-5,...
                                  'NoiseVariance',0.003);
```

Since there is no life time variable in the training data, create an arbitrary life time vector for fitting.

```
lifeTime = [1:length(expRealTime)];
```

Observe the degradation feature for 100 iterations. Update the degradation model after each iteration.

```
for i=1:100
    update(mdl,[lifeTime(i) expRealTime(i)])
end
```

Reset the model, which clears the accumulated statistics from the previous observations and resets the posterior distributions to the prior distributions.

```
restart(mdl)
```

Update Exponential Degradation Model in Real Time

Load training data, which is a degradation feature profile for a component.

```
load('expRealTime.mat')
```

For this example, assume that the training data is not historical data. When there is no historical data, you can update your degradation model in real time using observed data.

Create an exponential degradation model with the following settings:

- Arbitrary θ and β prior distributions with large variances so that the model relies mostly on observed data
- Noise variance of `0.003`

```
mdl = exponentialDegradationModel('Theta',1,'ThetaVariance',1e6,...
                                  'Beta',1,'BetaVariance',1e6,...
                                  'NoiseVariance',0.003);
```

Since there is no life time variable in the training data, create an arbitrary life time vector for fitting.

```
lifeTime = [1:length(expRealTime)];
```

Observe the degradation feature for 10 iterations. Update the degradation model after each iteration.

```
for i=1:10
    update(mdl,[lifeTime(i) expRealTime(i)])
end
```

After observing the model for some time, for example at a steady-state operating point, you can restart the model and save the current posterior distribution as a prior distribution.

```
restart(mdl,true)
```

View the updated prior distribution parameters.

```
mdl.Prior
```

```
ans = struct with fields:
            Theta: 2.3555
    ThetaVariance: 0.0058
             Beta: 0.0722
     BetaVariance: 3.6362e-05
              Rho: -0.8429
```

Input Arguments

mdl — Degradation RUL model
linearDegradationModel object | exponentialDegradationModel object

Degradation RUL model, specified as a linearDegradationModel object or an exponentialDegradationModel object. restart clears the accumulated statistics in mdl and resets the InitialLifeTimeValue and CurrentLifeTimeValue properties of mdl.

resetPrior — Flag for resetting prior parameter values
false (default) | true

Flag for resetting prior parameter information, specified as a logical value. When resetPrior is:

- true, then restart sets the prior parameter values of mdl to their corresponding current posterior parameter values. For example, mdl.Prior.Theta is set to mdl.Theta.
- false or omitted, then restart does not update the prior.

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: nv1,'value'

Theta — Mean value of model θ parameter
scalar

This property is read-only.

Mean value of model θ parameter, specified as the comma-separated pair 'Theta' and a scalar. Use this argument to set the Theta property of mdl and the corresponding field of the Prior property of mdl.

ThetaVariance — Variance of model θ parameter
nonnegative scalar

This property is read-only.

Variance of the θ parameter in the degradation model, specified as the comma-separated pair 'ThetaVariance' and a nonnegative scalar. Use this argument to set the ThetaVariance property of mdl and the corresponding field of the Prior property of mdl.

Beta — Mean value of model β parameter
scalar

This property is read-only.

Mean value of model β parameter, specified as the comma-separated pair 'Beta' and a scalar. Use this argument to set the Beta property of mdl and the corresponding field of the Prior property of mdl.

This argument applies only when mdl is an exponentialDegradationModel.

`BetaVariance` — Variance of model β parameter
nonnegative scalar

This property is read-only.

Variance of model β parameter, specified as the comma-separated pair `'BetaVariance'` and a nonnegative scalar. Use this argument to set the `BetaVariance` property of `mdl` and the corresponding field of the `Prior` property of `mdl`.

This argument applies only when `mdl` is an `exponentialDegradationModel`.

Rho — Correlation between θ and β
scalar value in the range [-1,1]

This property is read-only.

Correlation between θ and β, specified as the comma-separated pair `'Rho'` and a scalar value in the range [-1,1]. Use this argument to set the `Rho` property of `mdl` and the corresponding field of the `Prior` property of `mdl`.

This argument applies only when `mdl` is an `exponentialDegradationModel`.

`NoiseVariance` — Model additive noise variance
nonnegative scalar

Model additive noise variance, specified as the comma-separated pair `'NoiseVariance'` and a nonnegative scalar. Use this argument to set the `NoiseVariance` property of `mdl`.

`SlopeDetectionLevel` — Slope detection level
scalar value in the range [0,1] | `[]`

Slope detection level for determining the start of the degradation process, specified as the comma-separated pair `'SlopeDetectionLevel'` and a scalar in the range [0,1]. Use this argument to set the `SlopeDetectionLevel` property of `mdl`.

To disable the slope detection test, set `SlopeDetectionLevel` to `[]`.

`UseParallel` — Flag for using parallel computing
`false` (default) | `true`

Flag for using parallel computing when fitting prior values from data, specified as the comma-separated pair `'UseParallel'` and either `true` or `false`. Use this argument to set the `UseParallel` property of `mdl`.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

- This command supports code generation with MATLAB Coder. Before generating code that uses an RUL model, you must save the model using `saveRULModelForCoder`. For an example, see “Generate Code for Predicting Remaining Useful Life”.

See Also

Functions
`linearDegradationModel` | `exponentialDegradationModel` | `update`

Topics
"Models for Predicting Remaining Useful Life"

Introduced in R2018a

restoreState

Restore degradation RUL model state at runtime

Syntax

`restoreState(mdl,mdlState)`

Description

`restoreState(mdl,mdlState)` updates the properties of the degradation RUL model `mdl` according to the values specified in the structure `mdlState`. Create `mdlState` using the `readState` command. Use `readState` and `restoreState` in an entry-point function for code generation to preserve the values of model parameters, particularly when you update the model at run time. For more information, see "Generate Code that Preserves RUL Model State for System Restart".

Input Arguments

`mdl` — RUL model
`linearDegradationModel` | `exponentialDegradationModel`

RUL model to update, specified as a `linearDegradationModel` or `exponentialDegradationModel` RUL model object.

`mdlState` — Model state
structure

Model state, specified as a structure. The fields of `mdlState` correspond to the properties of `mdl`, with an extra field that specifies the type of RUL model. Create `mdlState` using the `readState` command.

Extended Capabilities

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

See Also

`readState` | `loadRULModelForCoder`

Topics
"Generate Code that Preserves RUL Model State for System Restart"

Introduced in R2021a

saveRULModelForCoder

Save RUL model for use in code generation

Syntax

```
saveRULModelForCoder(mdl,filename)
```

Description

`saveRULModelForCoder(mdl,filename)` saves an RUL model to a MAT file for use in code generation. Use the file with `loadModelForCoder` in an entry-point function for code generation to reconstruct the model at compile time. See "Generate Code for Predicting Remaining Useful Life" for more information.

Input Arguments

`mdl` — RUL model

`linearDegradationModel` | `exponentialDegradationModel` | `reliabilitySurvivalModel` | `covariateSurvivalModel`

RUL model to save, specified as a `linearDegradationModel`, a `exponentialDegradationModel`, a `reliabilitySurvivalModel`, or a `covariateSurvivalModel` object. `saveRULModelForCoder` stores the properties of `mdl` in the MAT file `filename`.

`filename` — Name of file

character vector | string

Name of file in which to save the RUL model, specified as character vector or a string. You can specify a full or relative path in `filename`. The function creates a MAT file.

See Also

`loadRULModelForCoder` | `readState` | `linearDegradationModel` | `exponentialDegradationModel` | `reliabilitySurvivalModel` | `covariateSurvivalModel`

Topics

"Generate Code for Predicting Remaining Useful Life"

Introduced in R2021a

subset

Create new ensemble datastore from subset of existing ensemble datastore

Syntax

```
sens = subset(ens,idx)
```

Description

`sens = subset(ens,idx)` creates a new ensemble datastore `sens` from a subset of the existing ensemble datastore `ens` by extracting the ensemble members that correspond to the indices in `idx`.

Use `subset` when you want to perform ensemble operations on a specific ensemble member or group of ensemble members, and when using a sequence of `read` commands with the source ensemble does not provide the ensemble members that you want to process. For example, you can use `subset` to:

- Extract only ensemble members with a specific fault condition.
- Perform preliminary processing and feature generation on a smaller ensemble that contains a similar distribution of conditions to the larger ensemble.
- Extract a single ensemble member with specific characteristics to isolate and explore member behavior.

Specify which members you want to extract using the index vector `idx`. You can then operate on your extracted ensemble using the same techniques that you use for any data ensemble.

Examples

Extract Specific Member from Ensemble Datastore

Extract the ensemble member that you identify from an ensemble datastore and use a single read command to obtain the contents.

For this example, use the following code to create a `simulationEnsembleDatastore` object using data previously generated by running a Simulink® model at a various fault values (see `generateSimulationEnsemble`). The ensemble includes simulation data for five different values of a model parameter, `ToothFaultGain`. Because of the volume of data, the `unzip` operation takes a few minutes.

```
unzip simEnsData.zip
ens = simulationEnsembleDatastore(pwd,'logsout')
```

```
ens = 
  simulationEnsembleDatastore with properties:

           DataVariables: [5x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: [0x0 string]
       SelectedVariables: [5x1 string]
                ReadSize: 1
```

```
              NumMembers: 5
          LastMemberRead: [0x0 string]
                   Files: [5x1 string]
```

```
ems_nm = ens.NumMembers
```

```
ems_nm = 5
```

The ensemble contains five files.

Extract the fourth ensemble member into a new, single-member ensemble `sens`.

```
idx = 4;
sens = subset(ens,idx);
sens_nm = sens.NumMembers
```

```
sens_nm = 1
```

`sens` contains one member. View the file name to confirm the member index.

```
sens.Files
```

```
ans = 
"C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\7\tp27f11553\predmaint-ex43507974\TransmissionCasingSimpli
```

Reset `sens` to the first member and read the contents.

```
reset(sens)
m4 = read(sens)
```

```
m4=1×5 table
    PMSignalLogName           SimulationInput                        SimulationMetadata
    _______________    _________________________________    ___________________________________    __

      {'logsout'}      {1x1 Simulink.SimulationInput}    {1x1 Simulink.SimulationMetadata}    {20
```

`m4` contains the data for the extracted member.

Create Subset of Ensemble Datastore

Create a simulation ensemble datastore from a subset of an existing simulation ensemble datastore.

Create a `simulationEnsembleDatastore` object using data previously generated by running a Simulink® model at various fault values.

```
unzip simEnsData.zip
ens = simulationEnsembleDatastore(pwd,'logsout');
ens_nm = ens.NumMembers
```

```
ens_nm = 5
```

The ensemble contains five files. View the file names.

```
ens.Files
```

```
ans = 5x1 string
    "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\7\tp27f11553\predmaint-ex46856662\TransmissionCasingS
    "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\7\tp27f11553\predmaint-ex46856662\TransmissionCasingS
    "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\7\tp27f11553\predmaint-ex46856662\TransmissionCasingS
    "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\7\tp27f11553\predmaint-ex46856662\TransmissionCasingS
    "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\7\tp27f11553\predmaint-ex46856662\TransmissionCasingS
```

Extract the first, third, and fifth files into a new ensemble.

```
idx = [1 3 5];
sens = subset(ens,idx);
sens_nm = sens.NumMembers
```

```
sens_nm = 3
```

The new ensemble contains three members. View the file names.

```
sens.Files
```

```
ans = 3x1 string
    "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\7\tp27f11553\predmaint-ex46856662\TransmissionCasingS
    "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\7\tp27f11553\predmaint-ex46856662\TransmissionCasingS
    "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\7\tp27f11553\predmaint-ex46856662\TransmissionCasingS
```

The new ensemble contains the three files that you indexed.

Input Arguments

ens — Source ensemble datastore
`fileEnsembleDatastore` object | `simulationEnsembleDatastore` object

Source ensemble datastore from which to extract members, specified as a `fileEnsembleDatastore` or a `simulationEnsembleDatastore` object. For an example of extracting a member from an ensemble datastore, see “Extract Specific Member from Ensemble Datastore” on page 1-209.

idx — Indices of source ensemble members
numeric vector | integer vector | logical vector

Indices of source ensemble members to extract, specified as a numeric vector, an integer vector, or a logical vector. The number of elements in the vector must not exceed the number of members in `ens`. For numeric or integer vectors, all indices must be positive. For logical vectors, the number of elements must be equal to the number of ensemble members in `ens`. For an example of creating and using an index vector, see “Create Subset of Ensemble Datastore” on page 1-210.

Output Arguments

sens — Extracted ensemble datastore
`fileEnsembleDatastore` object | `simulationEnsembleDatastore` object

Extracted ensemble datastore, returned as a `fileEnsembleDatastore` or a `simulationEnsembleDatastore` object.

See Also

`fileEnsembleDatastore` | `simulationEnsembleDatastore` | `generateSimulationEnsemble` | `read` | `reset`

Topics

“Data Ensembles for Condition Monitoring and Predictive Maintenance”
“Ensemble Data in Predictive Maintenance Toolbox”

Introduced in R2021a

tfmoment

Joint moment of the time-frequency distribution of a signal

Syntax

```
momentJ = tfmoment(xt,order)
momentJ = tfmoment(x,fs,order)
momentJ = tfmoment(x,ts,order)
momentJ = tfmoment(p,fp,tp,order)
momentJ = tfmoment( ___ ,Name,Value)
```

Description

Time-frequency moments provide an efficient way to characterize signals whose frequencies change in time (that is, are nonstationary). Such signals can arise from machinery with degraded or failed hardware. Classical Fourier analysis cannot capture the time-varying frequency behavior. Time-frequency distribution generated by short-time Fourier transform (STFT) or other time-frequency analysis techniques can capture the time-varying behavior, but directly treating these distributions as features carries a high computational burden, and potentially introduces unrelated and undesirable feature characteristics. In contrast, distilling the time-frequency distribution results into low-dimension time-frequency moments provides a method for capturing the essential features of the signal in a much smaller data package. Using these moments significantly reduces the computational burden for feature extraction and comparison — a key benefit for real-time operation [1], [2].

The Predictive Maintenance Toolbox™ implements the three branches of time-frequency moment:

- Conditional spectral moment — `tfsmoment`
- Conditional temporal moment — `tftmoment`
- Joint time-frequency moment — `tfmoment`

`momentJ = tfmoment(xt,order)` returns the “Joint Time-Frequency Moments” on page 1-219 of `timetable` `xt` as a vector with one or more components. Each `momentJ` scalar element represents the joint moment for one of the orders you specify in `order`. The data in `xt` can be nonuniformly sampled.

`momentJ = tfmoment(x,fs,order)` returns the joint time-frequency moment of time-series vector `x`, sampled at rate `Fs`. The moment is returned as a vector, in which each scalar element represents the joint moment corresponding to one of the orders you specify in `order`. With this syntax, `x` must be uniformly sampled.

`momentJ = tfmoment(x,ts,order)` returns the joint time-frequency moment of `x` sampled at the time instants specified by `ts` in seconds.

- If `ts` is a scalar `duration`, then `tfmoment` applies it uniformly to all samples.
- If `ts` is a vector, then `tfmoment` applies each element to the corresponding sample in `x`. Use this syntax for nonuniform sampling.

`momentJ = tfmoment(p,fp,tp,order)` returns the joint time-frequency moment of a signal whose power spectrogram is `p`. `fp` contains the frequencies corresponding to the spectral estimate

contained in p. tp contains the vector of time instants corresponding to the centers of the windowed segments used to compute short-time power spectrum estimates. Use this syntax when:

- You already have the power spectrogram you want to use.
- You want to customize the options for pspectrum, rather than accept the default pspectrum options that tfmoment applies. Use pspectrum first with the options you want, and then use the output p as input for tfmoment. This approach also allows you to plot the power spectrogram.

momentJ = tfmoment(___ ,Name,Value) specifies additional properties using name-value pair arguments. Options include moment centralization, frequency-limit specification, and time-limit specification.

You can use Name,Value with any of the input-argument combinations in previous syntaxes.

Examples

Find the Joint Time-Frequency Moments of a Time Series

Find the joint time-frequency moments of a time series using multiple moment specifications. Compute the same moment using a specified power spectrogram input.

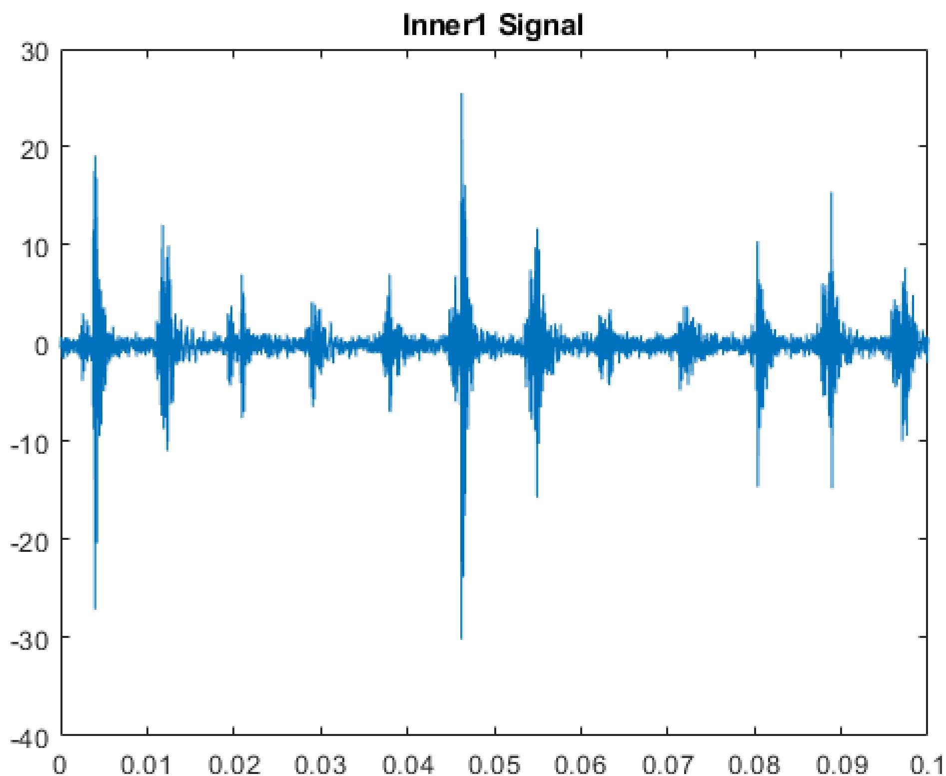
This example is adapted from “Rolling Element Bearing Fault Diagnosis”, which provides a more comprehensive treatment of the data sources and history.

Load the data, which contains vibration measurements for a faulty machine. x_inner1 and sr_inner1 contain the data vector and sample rate.

```
load tfmoment_data.mat x_inner1 sr_inner1
```

Examine the data. Construct a time vector from the sample rate, and plot the data. Then zoom in to an 0.1 s section so that the behavior can be seen more clearly.

```
t_inner1 = (0:length(x_inner1)-1)/sr_inner1; % Construct time vector of [0 1/sr 2/sr ...] matchi
figure
plot(t_inner1,x_inner1)
title ('Inner1 Signal')
hold on
xlim([0 0.1]) % Zoom in to an 0.1 s section
hold off
```



The plot shows periodic impulsive variations in the acceleration measurements over time.

Find the joint moment of second order for both time and frequency

```
order = [2,2];
momentJ = tfmoment(x_inner1,sr_inner1,order)
```

```
momentJ = 3.6253e+08
```

The resulting moment has only one element, representing the [2,2] time-frequency pair.

Now include the fourth moment for time and frequency. You can also mix orders within a pair. Include a joint moment with a second order for time and a fourth order for frequency. The order matrix contains two columns — the first for time and the second for frequency. Each row contains the order pair to compute.

```
order = [2,2;2,4;4,4];
momentJ = tfmoment(x_inner1,t_inner1,order);
momentJ(1)
```

```
ans = 3.6253e+08
```

```
momentJ(2)
```

```
ans = 7.9495e+16
```

```
momentJ(3)
```

```
ans = 4.0886e+17
```

You can also take the moment using an existing spectrogram. Load the data for a spectrogram which was computed using the same signal and default options. Input this to tfmoment, using the 3-row `order` matrix already computed.

```
load tfmoment_data.mat p_inner1_def f_p_def t_p_def
momentJ = tfmoment(p_inner1_def,f_p_def,t_p_def,order);
momentJ(1)
```

```
ans = 3.6261e+08
```

```
momentJ(2)
```

```
ans = 7.9513e+16
```

```
momentJ(3)
```

```
ans = 4.0896e+17
```

The joint moments distill a large amount of time and frequency data into a small set of single data points. They represent important, and concise, features that you can use in multiple ways in your application. Possibilities include comparison with health-regime limits and computing moments of segmented data over a period of time to assess long-term degradation.

Input Arguments

`xt` — Time-series signal

timetable

Time-series signal for which `tfmoment` returns the moments, specified as a `timetable` that contains a single variable with a single column. `xt` must contain increasing, finite row times. If the timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”. `xt` can be nonuniformly sampled, with the `pspectrum` constraint that the median time interval and the mean time interval must obey:

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

`order` — Moment orders to return

positive integer matrix

Moment orders to return, specified as an n-by-2 matrix with real positive integers.

- The first column provides the orders of time.
- The second column provides the orders of frequency.

Example: `momentJ = tfmoment(x,[2,2])` specifies the second-order joint moment (variance) of the time-frequency distribution of `x`.

Example: `momentJ = tfmoment(x,[2,2;4,4])` specifies the second and fourth moment orders for both time and frequency of the time-frequency distribution of `x`.

You can specify any order and number of orders, but low-order moments carry less computational burden and are better suited to real-time applications. You can also use a different order for time than you use for frequency. The first four moment orders correspond to the statistical moments of a data set:

1. Mean
2. Variance
3. Skewness (degree of asymmetry about the mean)
4. Kurtosis (length of outlier tails in the distribution — a normal distribution has a kurtosis of 3)

For an example, see "Find the Joint Time-Frequency Moments of a Time Series" on page 1-214.

x — Time-series signal
vector

Time-series signal from which `tfmoment` returns the moments, specified as a vector.

For an example of a time-series input, see "Find the Joint Time-Frequency Moments of a Time Series" on page 1-214.

fs — Sample rate
positive scalar

Sample rate of `x`, specified as positive scalar in hertz when `x` is uniformly sampled.

ts — Sample-time values
`duration` scalar | vector | `duration` vector | `datetime` vector

Sample-time values, specified as one of the following:

- `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`.

`ts` can be nonuniform, with the `pspectrum` constraint that the median time interval and the mean time interval must obey:

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

p — Power spectrogram or spectrum of signal
matrix | vector

Power spectrogram or spectrum of a signal, specified as a matrix (spectrogram) or a column vector (spectrum). `p` contains an estimate of the short-term, time-localized power spectrum of a time-series signal. If you specify `p`, then `tfmoment` uses `p` rather than generate its own power spectrogram. For an example, see "Find the Joint Time-Frequency Moments of a Time Series" on page 1-214.

fp — Frequencies for p
vector

Frequencies for power spectrogram or spectrum `p` when `p` is supplied explicitly to `tfmoment`, specified as a vector in hertz. The length of `fp` must be equal to the number of rows in `p`.

tp — Time information for p
vector | `duration` vector | `datetime` vector | `duration` scalar

Time information for power spectrogram or spectrum `p` when `p` is supplied explicitly to `tfmoment`, specified as one of the following:

- Vector of time points, whose data type can be numeric, `duration`, or `datetime`. The length of vector `tp` must be equal to the number of columns in `p`.
- `duration` scalar that represents the time interval in `p`. The scalar form of `tp` can be used only when `p` is a power spectrogram matrix.
- For the special case where `p` is a column vector (power spectrum), `tp` can be a numeric, `duration`, or `datetime` scalar representing the time point of the spectrum.

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: `'Centralize',false,'FrequencyLimits',[10 100]` computes the joint time-frequency moment for the portion of the signal ranging from 10 Hz to 100 Hz.

`Centralize` — Centralize-moment option
`true` (default) | `false`

Centralize-moment option, specified as the comma-separated pair consisting of `'Centralize'` and a logical.

- If `Centralize` is `true`, then `tfmoment` returns the centralized conditional moment by subtracting the conditional mean (which is the first moment) in the computation.
- If `Centralize` is `false`, then `tfmoment` returns the noncentralized moment, preserving any data offset.

Example: `momentJ = tfmoment(x,[2,2],'Centralize',false)`.

`FrequencyLimits` — Frequency limits
full frequency band (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. This specification allows you to exclude a band of data at either end of the spectral range.

`TimeLimits` — Time Limits
full time band (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 `ts`, and of the data types:

- Numeric or `duration` when `fs` or a scalar `ts` are specified, or when `ts` is a single, double, or `duration` vector
- Numeric, `duration`, or `datetime` when `ts` is specified as a `datetime` vector

This specification allows you to extract a temporal section of data from a longer data set.

Output Arguments

`momentJ` — Conditional joint moment
vector

Conditional joint moment returned as a vector, the scalar elements of which each represents the joint moment of one of the specified time-frequency order pairs.

`momentJ` is always a vector, regardless of whether the input data is `timetable` `xt`, time-series vector `x`, or spectrogram data `p`.

More About

Joint Time-Frequency Moments

The joint time-frequency moments of a nonstationary signal comprise a set of time-varying parameters that characterize the signal spectrum as it evolves in time. They are related to the conditional temporal moments and the joint time-frequency moments. The joint time-frequency moment is an integral function of frequency, given time, and marginal distribution. The conditional temporal moment is an integral function of time, given frequency, and marginal distribution. The calculation of the joint time-frequency moment is a double integral that varies both time and frequency [1], [2].

Each moment is associated with a specific order, with the first four orders being the statistical properties of 1) mean, 2) variance, 3) skewness, and 4) kurtosis.

`tfmoment` computes the joint time-frequency moments of the time-frequency distribution for a signal `x`, for the orders specified in `order`. The function performs these steps:

1. Compute the spectrogram power spectrum, $P(t,f)$, of the input using the `pspectrum` function and uses it as a time-frequency distribution. If the syntax used supplies an existing $P(t,f)$, then `tfmoment` uses that instead.
2. Estimate the joint time-frequency moment $\langle t^n \omega^m \rangle$ of the signal using, for the noncentralized case:

   $$\langle t^n \omega^m \rangle = \int \int t^n \omega^m P(t,\omega) dt d\omega,$$

   where m is the order and $P(t)$ is the marginal distribution.

   For the centralized joint time-frequency moment $\mu_{t,\omega}^{n,m}(t)$, the function uses

   $$\mu_{t,\omega}^{n,m}(t) = \frac{1}{P(\omega)} \int \int \left(t - \langle t^1 \rangle_\omega\right)^n \left(\omega - \langle \omega^1 \rangle_t\right)^m P(t,\omega) dt d\omega,$$

   where $\langle t^1 \rangle_\omega$ and $\langle \omega^1 \rangle_t$ are the first temporal and spectral time-frequency moments.

References

[1] Loughlin, P. J. "What Are the Time-Frequency Moments of a Signal?" *Advanced Signal Processing Algorithms, Architectures, and Implementations XI, SPIE Proceedings*. Vol. 4474, November 2001.

[2] Loughlin, P., F. Cakrak, and L. Cohen. "Conditional Moment Analysis of Transients with Application to Helicopter Fault Data." *Mechanical Systems and Signal Processing*. Vol 14, Issue 4, 2000, pp. 511–522.

See Also

`tfsmoment` | `tftmoment` | `pspectrum`

Introduced in R2018a

tfsmoment

Conditional spectral moment of the time-frequency distribution of a signal

Syntax

```
momentS = tfsmoment(xt,order)
momentS = tfsmoment(x,fs,order)
momentS = tfsmoment(x,ts,order)
momentS = tfsmoment(p,fp,tp,order)
momentS = tfsmoment( ___ ,Name,Value)
```

```
[momentS,t] = tfsmoment( ___ )
```

```
tfsmoment( ___ )
```

Description

Time-frequency moments provide an efficient way to characterize signals whose frequencies change in time (that is, are nonstationary). Such signals can arise from machinery with degraded or failed hardware. Classical Fourier analysis cannot capture the time-varying frequency behavior. Time-frequency distribution generated by short-time Fourier transform (STFT) or other time-frequency analysis techniques can capture the time-varying behavior, but directly treating these distributions as features carries a high computational burden, and potentially introduces unrelated and undesirable feature characteristics. In contrast, distilling the time-frequency distribution results into low-dimension time-frequency moments provides a method for capturing the essential features of the signal in a much smaller data package. Using these moments significantly reduces the computational burden for feature extraction and comparison — a key benefit for real-time operation [1], [2].

The Predictive Maintenance Toolbox implements the three branches of time-frequency moment:

- Conditional spectral moment — `tfsmoment`
- Conditional temporal moment — `tftmoment`
- Joint time-frequency moment — `tfmoment`

`momentS = tfsmoment(xt,order)` returns the conditional spectral moment on page 1-238 of `timetable` `xt` as a `timetable`. The `momentS` variables provide the spectral moments for the orders you specify in `order`. The data in `xt` can be nonuniformly sampled.

`momentS = tfsmoment(x,fs,order)` returns the conditional spectral moment of time-series vector `x`, sampled at rate `Fs`. The moment is returned as a matrix, in which each column represents a spectral moment corresponding each element in `order`. With this syntax, `x` must be uniformly sampled.

`momentS = tfsmoment(x,ts,order)` returns the conditional spectral moment of `x` sampled at the time instants specified by `ts` in seconds.

- If `ts` is a scalar `duration`, then `tfsmoment` applies it uniformly to all samples.
- If `ts` is a vector, then `tfsmoment` applies each element to the corresponding sample in `x`. Use this syntax for nonuniform sampling.

`momentS = tfsmoment(p,fp,tp,order)` returns the conditional spectral moment of a signal whose power spectrogram is `p`. `fp` contains the frequencies corresponding to the spectral estimate contained in `p`. `tp` contains the vector of time instants corresponding to the centers of the windowed segments used to compute short-time power spectrum estimates. Use this syntax when:

- You already have the power spectrum or spectrogram you want to use.
- You want to customize the options for `pspectrum`, rather than accept the default `pspectrum` options that `tfsmoment` applies. Use `pspectrum` first with the options you want, and then use the output `p` as input for `tfsmoment`. This approach also allows you to plot the power spectrogram.

`momentS = tfsmoment( ___ ,Name,Value)` specifies additional properties using name-value pair arguments. Options include moment centralization and frequency-limit specification.

You can use `Name,Value` with any of the input-argument combinations in previous syntaxes.

`[momentS,t] = tfsmoment( ___ )` returns time vector `t`.

You can use `t` with any of the input-argument combinations in previous syntaxes.

`tfsmoment( ___ )` with no output arguments plots the conditional spectral moment. The plot x-axis is time, and the plot y-axis is the corresponding spectral moment.

You can use this syntax with any of the input-argument combinations in previous syntaxes.

Examples

Plot the Conditional Spectral Moment of a Time Series Vector

Plot the second-order conditional spectral moment (variance) of a time series using the plot-only approach and the return-data approach. Visualize the moment differently by plotting the histogram. Compare the moments for data arising from faulty and healthy machine conditions.

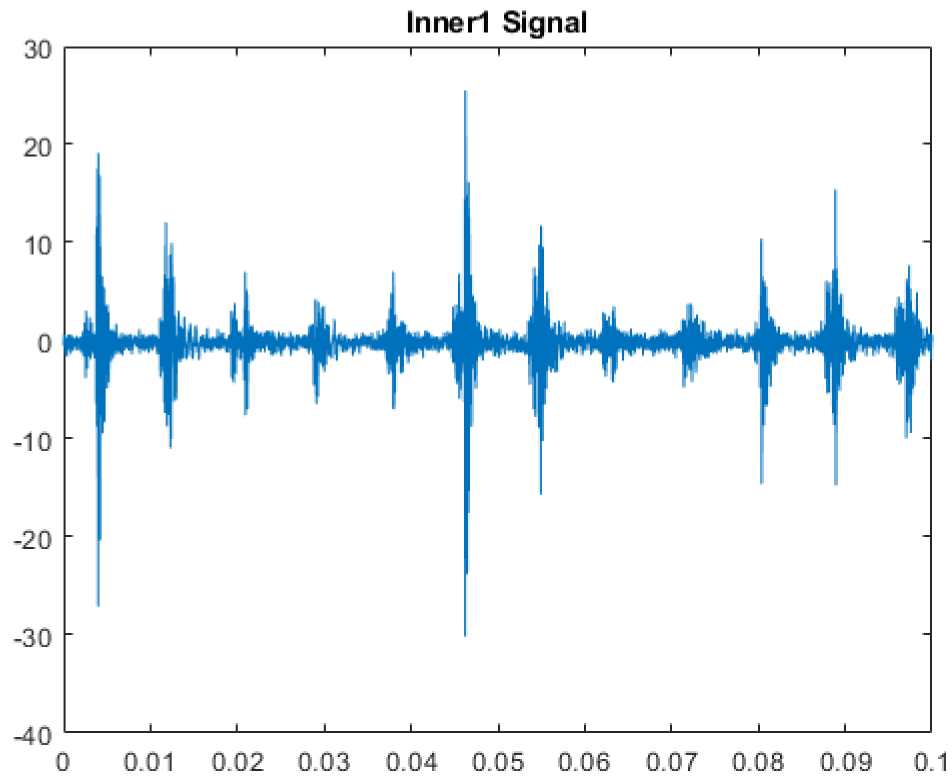
This example is adapted from “Rolling Element Bearing Fault Diagnosis”, which provides a more comprehensive treatment of the data sources and history.

Load the data, which contains vibration measurements for two conditions. `x_inner1` and `sr_inner1` contain the data vector and sample rate for a faulty condition. `x_baseline` and `sr_baseline` contain the data vector and sample rate for a healthy condition.

```
load tfmoment_data.mat x_inner1 sr_inner1 x_baseline1 sr_baseline1
```

Examine the faulty-condition data. Construct a time vector from the sample rate, and plot the data. Then zoom in to an 0.1-s section so that the behavior can be seen more clearly.

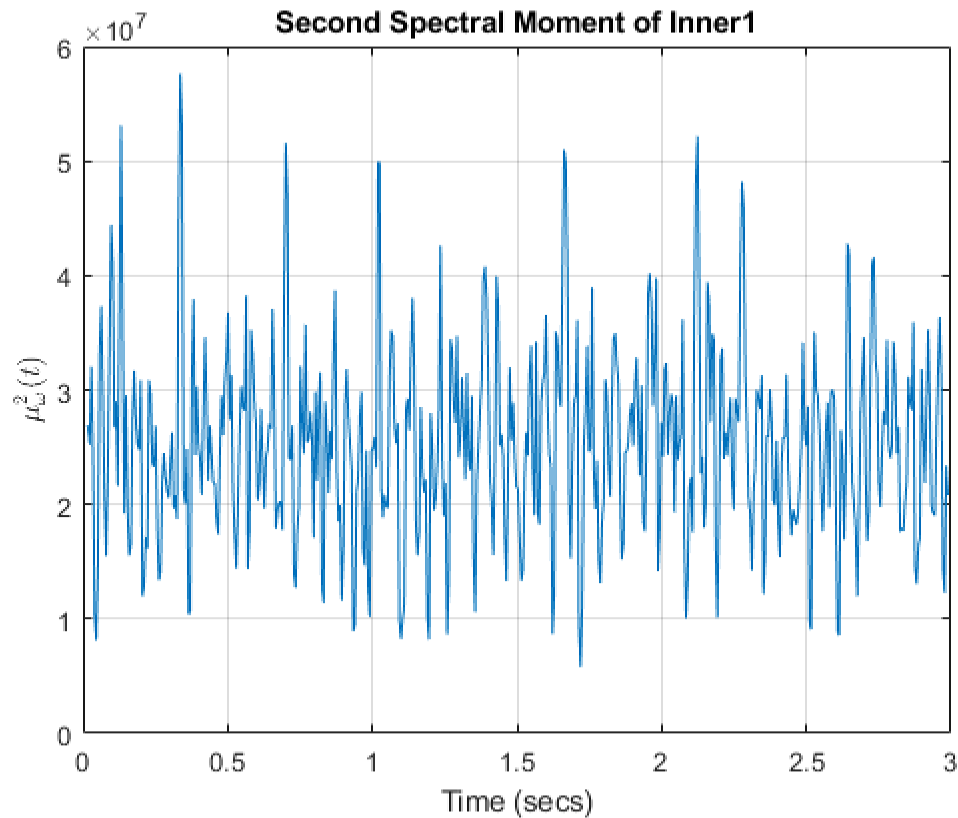
```
t_inner1 = (0:length(x_inner1)-1)/sr_inner1; % Construct time vector of [0 1/sr 2/sr ...] matchi
figure
plot(t_inner1,x_inner1)
title ('Inner1 Signal')
hold on
xlim([0 0.1]) % Zoom in to an 0.1 s section
hold off
```



The plot shows periodic impulsive variations in the acceleration measurements over time.

Plot the second spectral moment (`order=2`), using the `tfsmoment` syntax with no output arguments.

```
order = 2;
figure
tfsmoment(x_inner1,t_inner1,order)
title('Second Spectral Moment of Inner1')
```

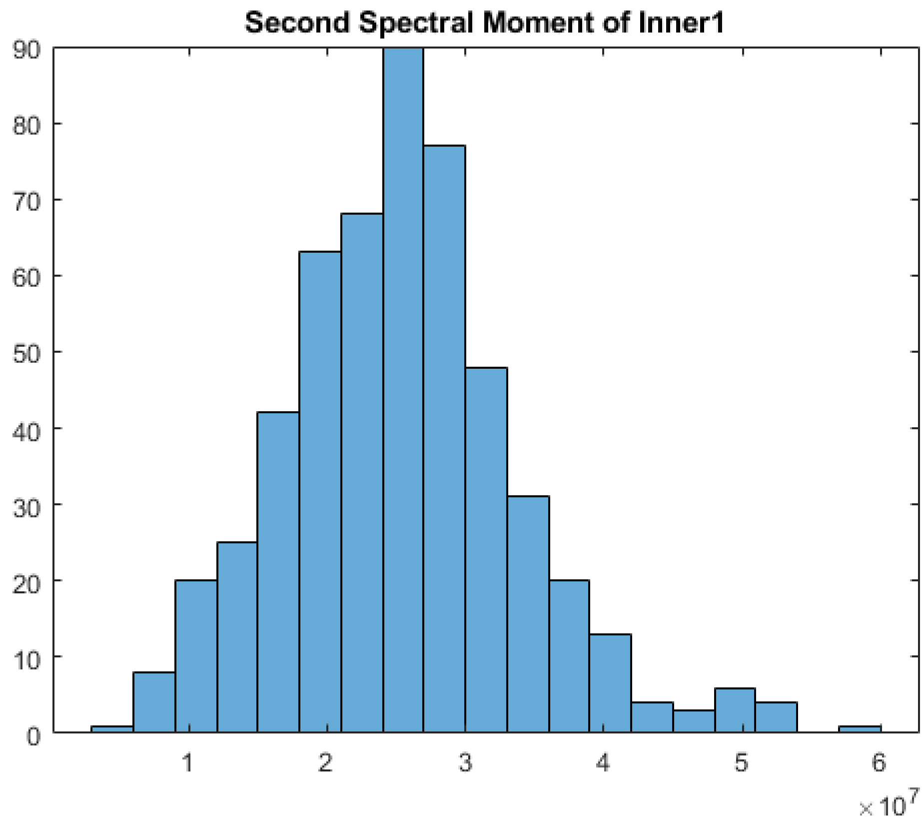


The plot illustrates the changes in the variance of the `x_inner1` spectrum over time. You are limited to this visualization (moment versus time) because `tfsmoment` returned no data. Now use `tfmoment` again to compute the second spectral moment, this time using the syntax that returns both the moment values and the associated time vector. You can use the sample rate directly in the syntax (`sr_inner1`), rather than the time vector you constructed (`t_inner1`).

```
[momentS_inner1,t1_inner1] = tfsmoment(x_inner1,sr_inner1,order);
```

You can now plot moment versus time as you did before, using `moment_inner1` and `t1_inner1`, with the same result as earlier. But you can also perform additional analysis and visualization of the moment vector, since `tfsmoment` returned the data. A histogram can provide concise information on the signal characteristics.

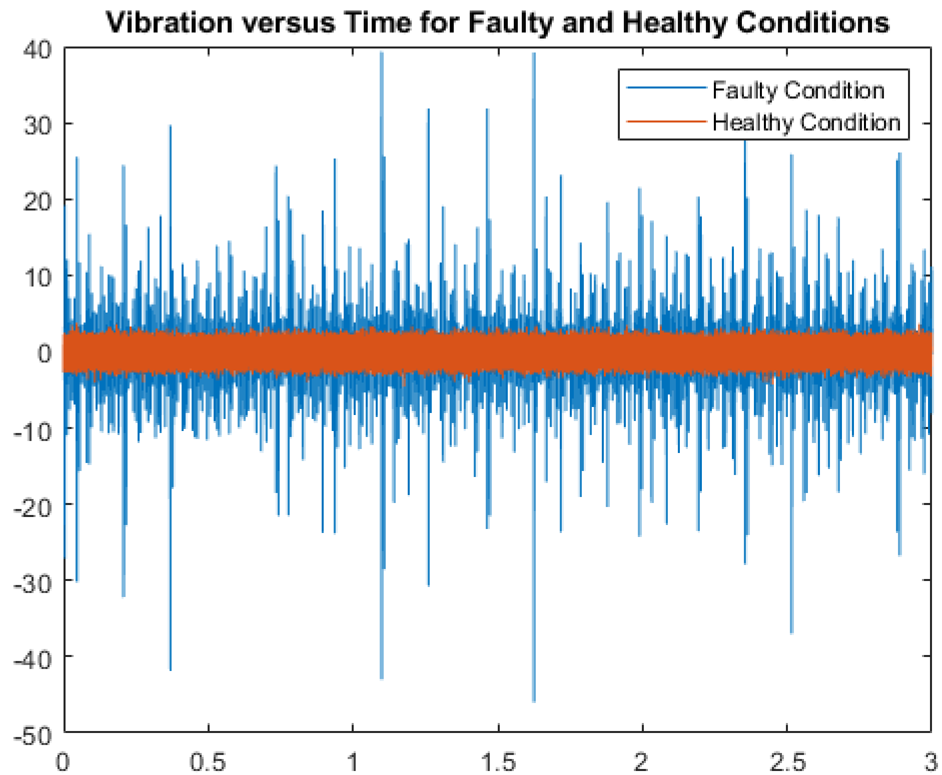
```
figure
histogram(momentS_inner1)
title('Second Spectral Moment of Inner1')
```



On its own, the histogram does not reveal obvious fault information. However, you can compare it to the histogram produced by the healthy-condition data.

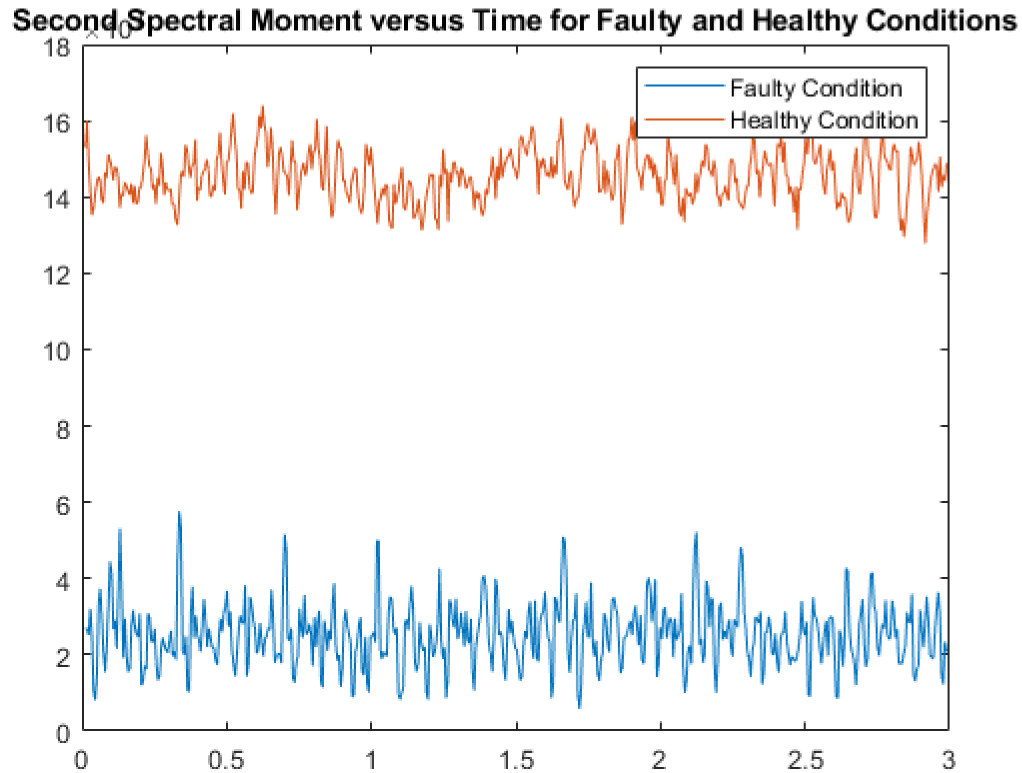
First, compare the inner and baseline time series directly using the same time-vector construction for the `baseline1` data as previously for the `inner1` data.

```
t_baseline1 = (0:length(x_baseline1)-1)/sr_baseline1;

figure
plot(t_inner1,x_inner1)
hold on
plot(t_baseline1,x_baseline1)
hold off
legend('Faulty Condition','Healthy Condition')
title('Vibration versus Time for Faulty and Healthy Conditions')
```



Calculate the second spectral moment of the `baseline1` data. Compare the `baseline1` and `inner1` time histories.

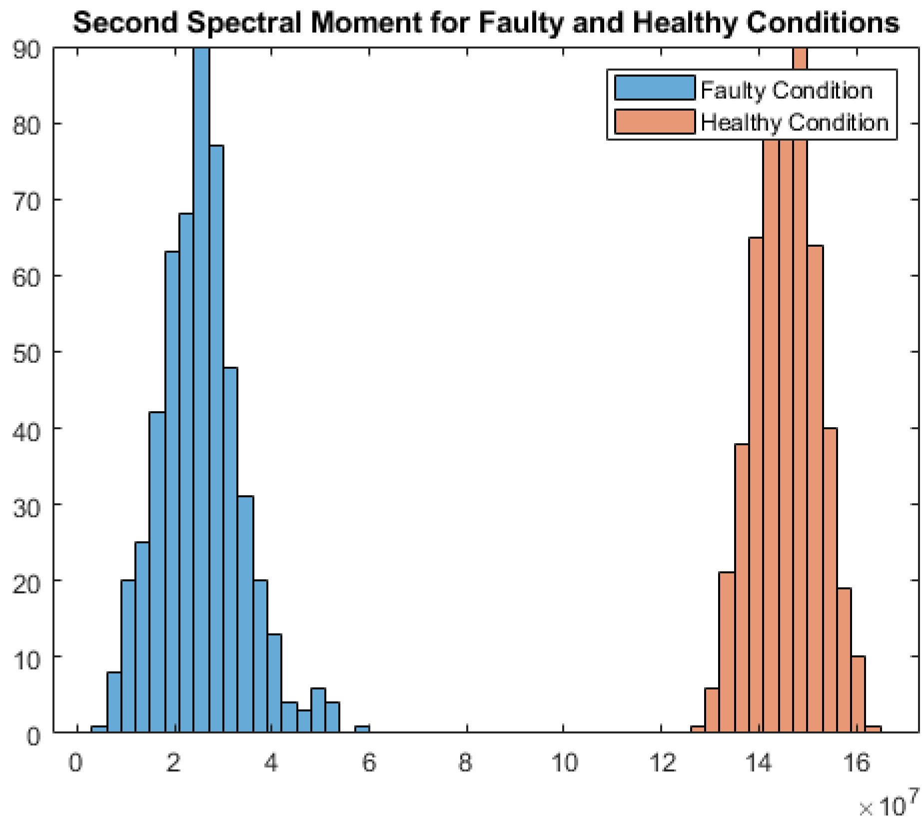
```
[momentS_baseline1,t1_baseline1] = tfsmoment(x_baseline1,sr_baseline1,2);

figure
plot(t1_inner1,momentS_inner1)
hold on
plot(t1_baseline1,momentS_baseline1)
hold off
legend('Faulty Condition','Healthy Condition')
title('Second Spectral Moment versus Time for Faulty and Healthy Conditions')
```



The moment plot shows behavior different from the earlier vibration plot. The vibration data for the faulty case is much noisier with higher-magnitude spikes than for the healthy case, although both appear to be zero mean. However, the spectral variance (second spectral moment) is significantly lower for the faulty case. The moment of the faulty case is still more noisy than the healthy case.

Plot the histograms.

```
figure
histogram(momentS_inner1);
hold on
histogram(momentS_baseline1);
hold off
legend('Faulty Condition','Healthy Condition')
title('Second Spectral Moment for Faulty and Healthy Conditions')
```



The moment behaviors distinguish the faulty condition from the healthy condition in both plots. The histogram provides distinct distribution characteristics — center point along x-axis, spread, and peak histogram bin.

Determine Multiple Orders of Conditional Spectral Moment for a Time Series

Determine the first four conditional spectral moments of a time-series data set, and extract the moments that you want to visualize with a histogram.

Load the data, which contains vibration measurements (`x_inner1`) and sample rate(`sr_inner1`) for machinery. Then use `tfsmoment` to compute the first four moments. These moments represent the statistical quantities of: 1) Mean; 2) Variance; 3) Skewness; and 4) Kurtosis.

You can specify the moment designators as a vector within the `order` argument.

```
load tfmoment_data.mat x_inner1 sr_inner1
momentS_inner1 = tfsmoment(x_inner1,sr_inner1,[1 2 3 4]);
```

Compare the dimensions of the input vector and the output matrix.

```
xsize = size(x_inner1)
```

```
xsize = 1×2
```

```
       146484            1
```

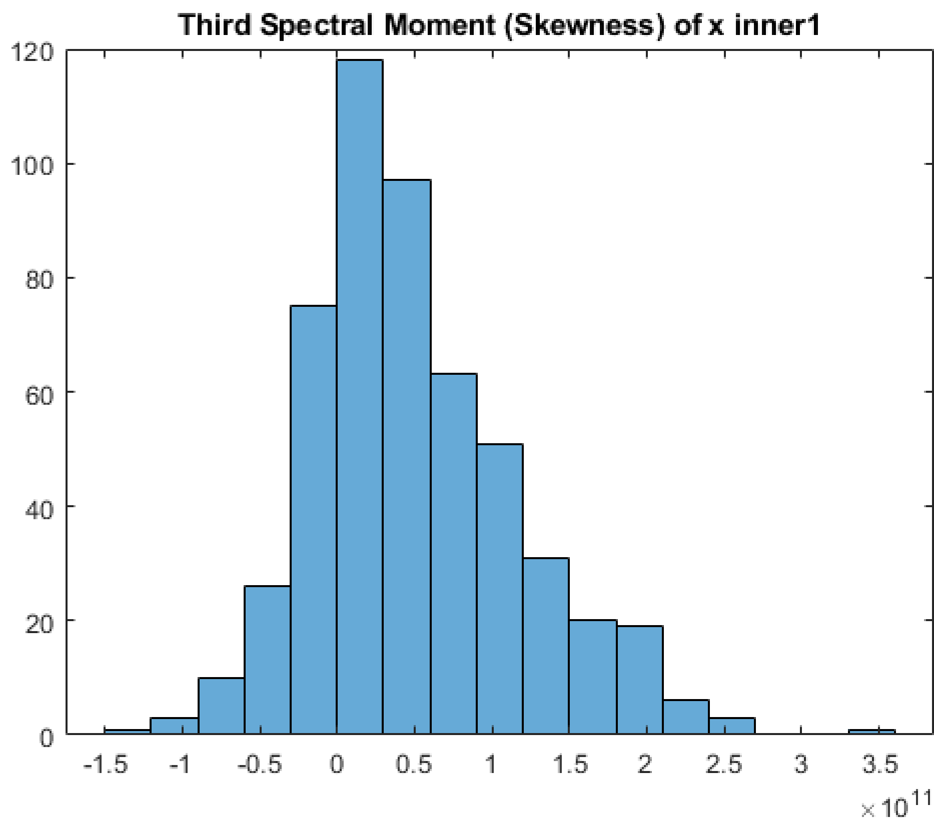
```
msize = size(momentS_inner1)
```

```
msize = 1×2

   524     4
```

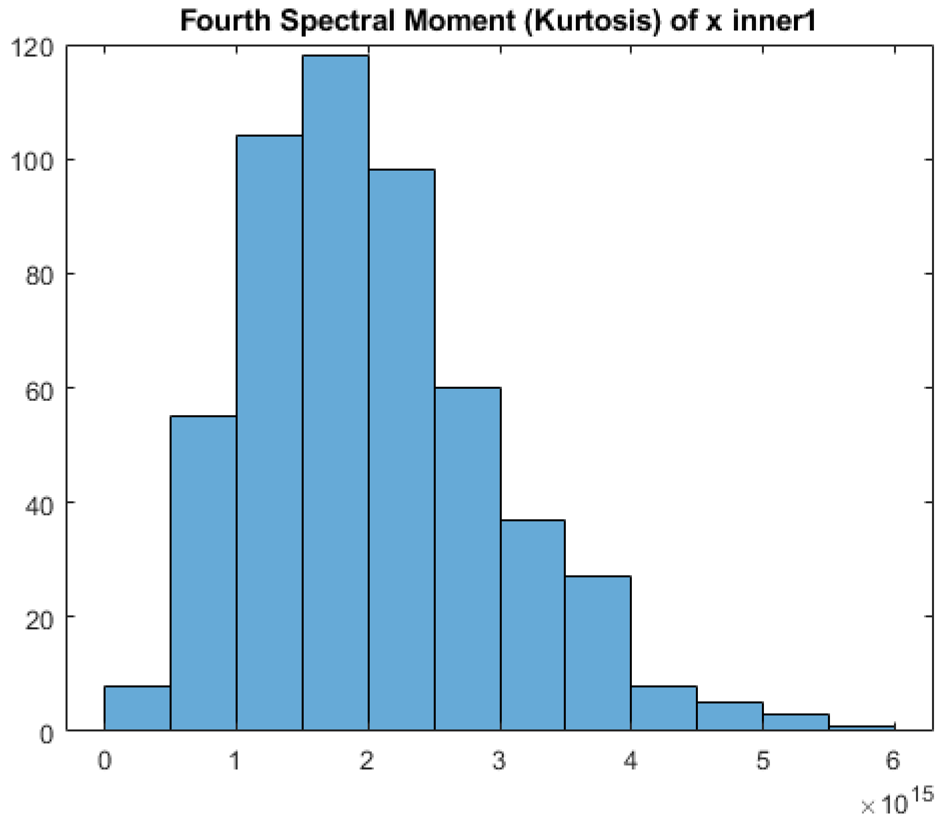
The data vector `x_inner` is considerably longer than the vectors in the moment matrix `momentS_inner1 because the spectrogram computation produces optimally-sized lower-resolution time windows. In this case, tfsmoment` returns a moment matrix containing four columns, one column for each moment order.

Plot the histograms for the third (skewness) and fourth (kurtosis) moments. The third and fourth columns of `momentS_inner1` provide these.

```
momentS_3 = momentS_inner1(:,3);
momentS_4 = momentS_inner1(:,4);
figure
histogram(momentS_3)
title('Third Spectral Moment (Skewness) of x inner1')
```




```
figure
histogram(momentS_4)
title('Fourth Spectral Moment (Kurtosis) of x inner1')
```



The plots are similar, but each has some unique characteristics with respect to number of bins and slope steepness.

Use a Customized Power Spectrogram to Compute the Conditional Spectral Moment

By default, `tfsmoment` calls the function `pspectrum` internally to generate the power spectrogram that `tfsmoment` uses for the moment computation. You can also import an existing power spectrogram for `tfsmoment` to use instead. This capability is useful if you already have a power spectrogram as a starting point, or if you want to customize the `pspectrum` options by generating the spectrogram explicitly first.

Input a power spectrogram that has been generated with customized options. Compare the resulting spectral-moment histogram with one that `tfsmoment` generates using its `pspectrum` default options.

Load the data, which includes two power spectrums and the associated frequency and time vectors.

The `p_inner1_def` spectrum was created using the default `pspectrum` options. It is equivalent to what `tfsmoment` computes internally when an input spectrum is not provided in the syntax.

The `p_inner1_MinThr` spectrum was created using the `MinThreshold` `pspectrum` option. This option puts a lower bound on nonzero values to screen out low-level noise. For this example, the threshold was set to screen out noise below the 0.5% level.

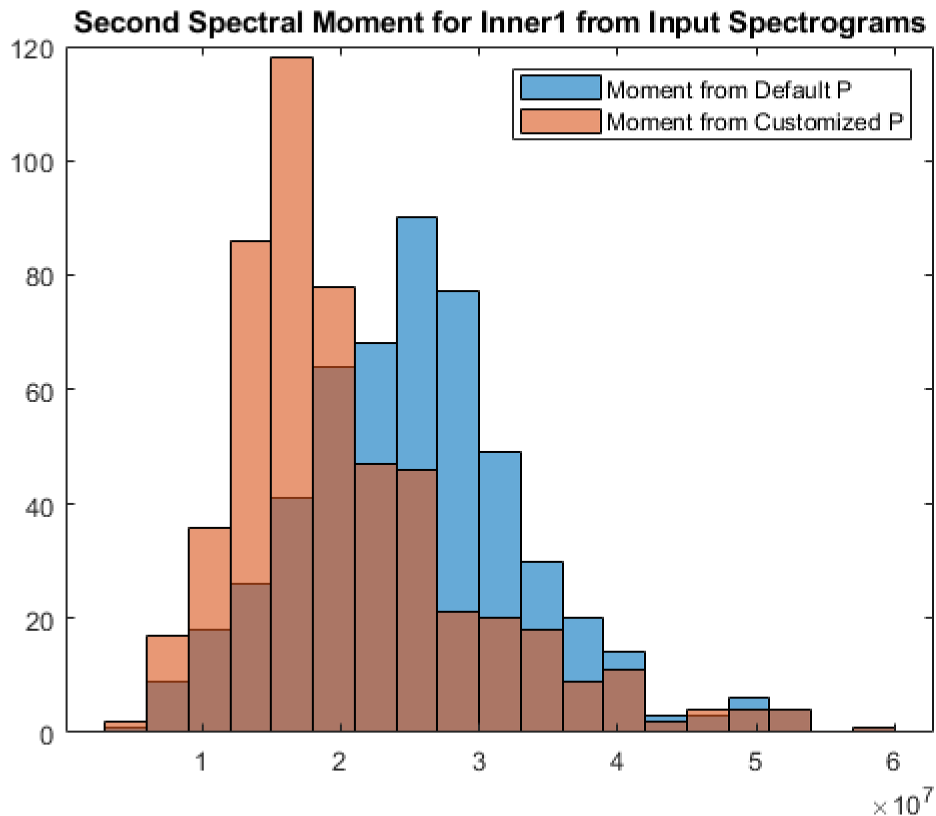
```
load tfmoment_data.mat p_inner1_def f_p_def t_p_def ...
    p_inner1_MinThr f_p_MinThr t_p_MinThr
load tfmoment_data.mat x_inner1 x_baseline1
```

Determine the second spectral moments (variance) for both cases.

```
moment_p_def = tfsmoment(p_inner1_def,f_p_def,t_p_def,2);
moment_p_MinThr = tfsmoment(p_inner1_MinThr,f_p_MinThr,t_p_MinThr,2);
```

Plot the histograms together.

```
figure
histogram(moment_p_def);
hold on
histogram(moment_p_MinThr);
hold off
legend('Moment from Default P','Moment from Customized P')
title('Second Spectral Moment for Inner1 from Input Spectrograms')
```



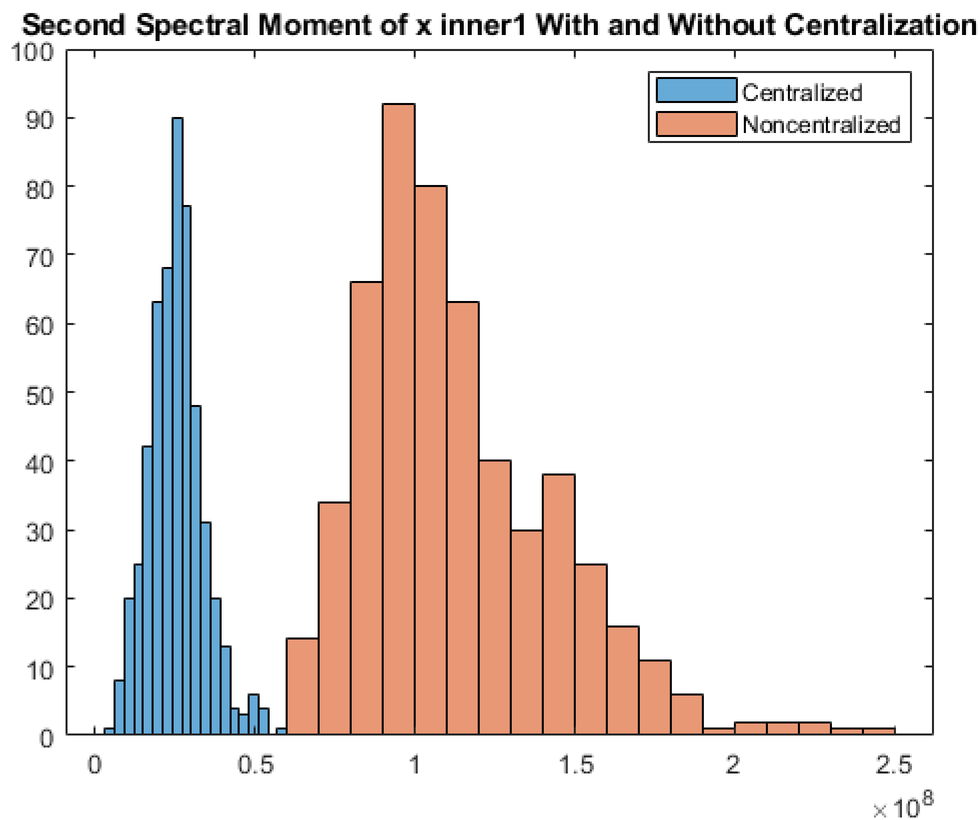
The histograms have the same overall spread, but the thresholded moment histogram has a higher peak bin at a lower moment magnitude level than the default moment. This example is for illustration purposes only, but does show the impact that preprocessing in the spectrum computation stage can have.

Calculate a Conditional Spectral Moment that is not Centralized

By default, `tfsmoment` centralizes the moment as part of its calculation. That is, it subtracts the sensor-data mean (which is the first moment) from the sensor data as part of the “Conditional Spectral Moments” on page 1-238. If you wish to preserve the offset, you can set the input argument `Centralize` to `false`.

Load the data, which contains vibration measurements x and sample rate sr for machinery. Calculate the 2nd moment (`order` = 2) both with centralization (default), and without centralization (`Centralize` = `false`). Plot the histograms together.

```
load tfmoment_data.mat x_inner1 sr_inner1
momentS_centr = tfsmoment(x_inner1,sr_inner1,2);
momentS_nocentr = tfsmoment(x_inner1,sr_inner1,2,'Centralize',false);

figure
histogram(momentS_centr)
hold on
histogram(momentS_nocentr);
hold off
legend('Centralized','Noncentralized')
title('Second Spectral Moment of x inner1 With and Without Centralization')
```



The noncentralized distribution is offset to the right.

Find the Conditional Spectral Moments of Data Measurements in a Timetable

Real-world measurements often come packaged as part of a time-stamped table that records actual time and readings rather than relative times. You can use the `timetable` format for capturing this data. This example shows how `tfsmoment` operates with a `timetable` input, in contrast to the data vector inputs used for the other `tfsmoment` examples, such as “Plot the Conditional Spectral Moment of a Time Series Vector” on page 1-222.

Load the data, which consists of a single `timetable` `xt_inner1` containing measurement readings and time information for a piece of machinery. Examine the properties of the `timetable`.

```
load tfmoment_tdata.mat xt_inner1;
xt_inner1.Properties
```

```
ans = 
  TimetableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Time'  'Variables'}
           VariableNames: {'x_inner1'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowTimes: [146484x1 duration]
               StartTime: 0 sec
              SampleRate: 4.8828e+04
                TimeStep: 2.048e-05 sec
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

This table consists of dimensions `Time` and the `Variables`, where the only variable is `x_inner1`.

Find the second and fourth conditional spectral moments for the data in the `timetable`. Examine the properties of the resulting moment `timetable`.

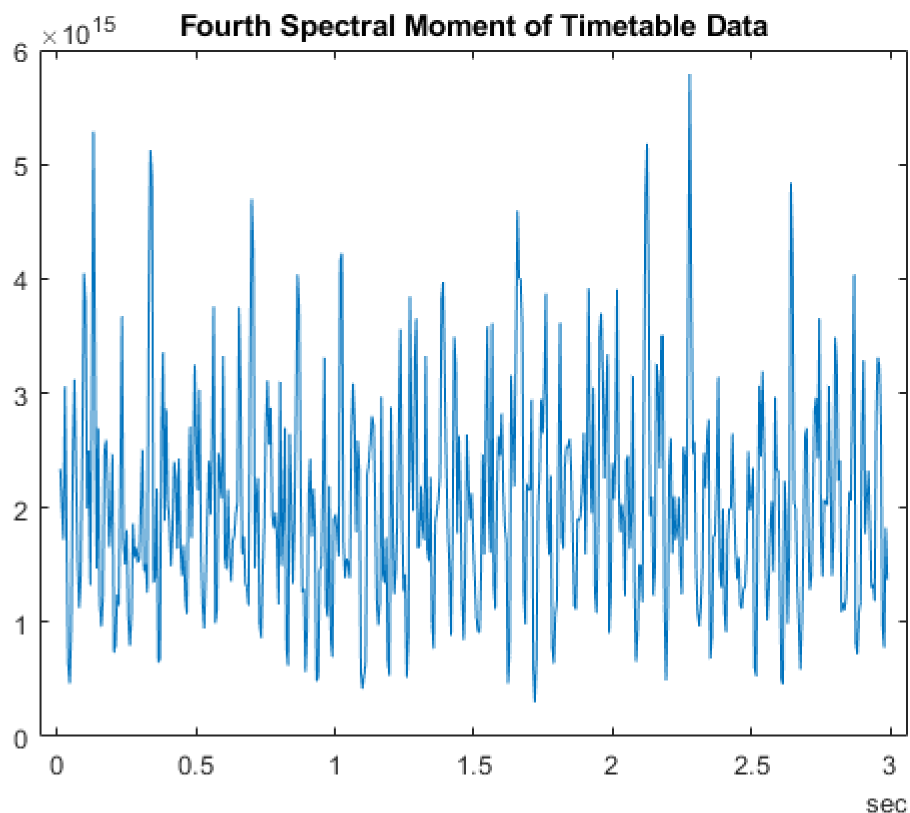
```
order = [2 4];
momentS_xt_inner1 = tfsmoment(xt_inner1,order);
momentS_xt_inner1.Properties
```

```
ans = 
  TimetableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Time'  'Variables'}
           VariableNames: {'CentralSpectralMoment2'  'CentralSpectralMoment4'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowTimes: [524x1 duration]
               StartTime: 0.011725 sec
              SampleRate: 175.6403
                TimeStep: 0.0056935 sec
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

The returned timetable represents the moments in the variable 'CentralSpectralMoment2' and 'CentralSpectralMoment4', providing information not only on what specific moment was calculated, but also on whether it was centralized.

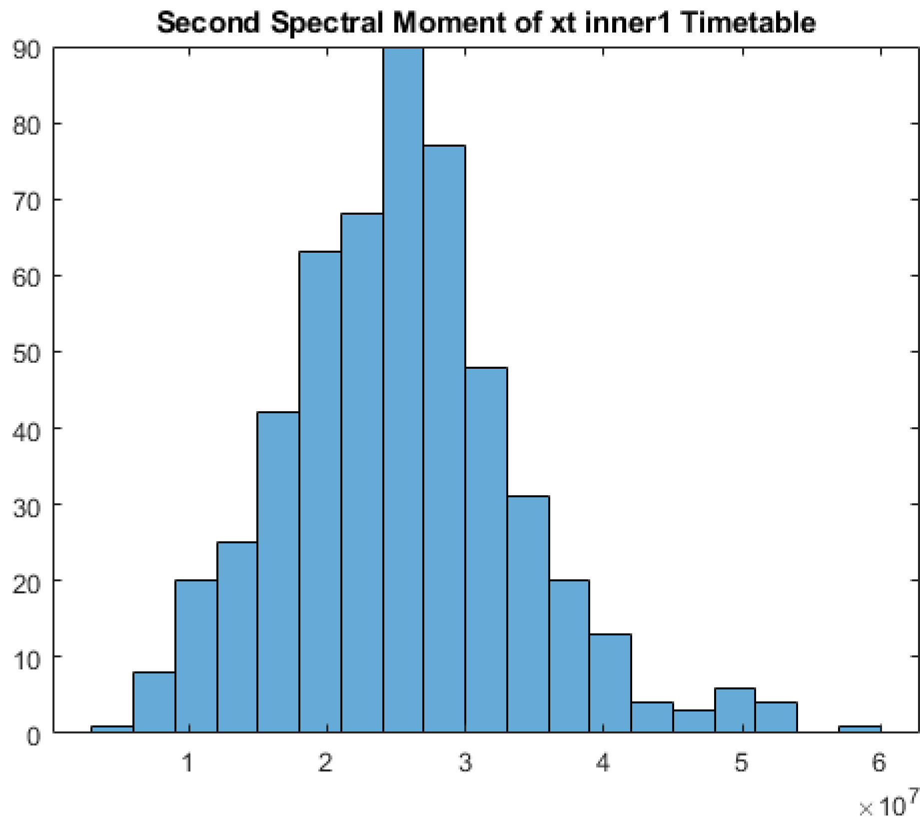
You can access the time and moment information directly from the timetable properties. Compute the second and fourth moments. Plot the fourth moment.

```
tt_inner1 = momentS_xt_inner1.Time;
momentS_inner1_2 = momentS_xt_inner1.CentralSpectralMoment2;
momentS_inner1_4 = momentS_xt_inner1.CentralSpectralMoment4;

figure
plot(tt_inner1,momentS_inner1_4)
title('Fourth Spectral Moment of Timetable Data')
```



As is illustrated in "Plot the Conditional Spectral Moment of a Time Series Vector" on page 1-222, a histogram is a very useful visualization for moment data. Plot the histogram, directly referencing the CentralSpectralMoment2 variable property.

```
figure
histogram(momentS_xt_inner1.CentralSpectralMoment2)
title('Second Spectral Moment of xt inner1 Timetable')
```



Input Arguments

xt — Signal Timetable
timetable

Signal Timetable for which `tfsmoment` returns the moments, specified as a `timetable` that contains a single variable with a single column. `xt` must contain increasing, finite row times. If the timetable has missing or duplicate time points, you can fix it using the tips in “Clean Timetable with Missing, Duplicate, or Nonuniform Times”. `xt` can be nonuniformly sampled, with the `pspectrum` constraint that the median time interval and the mean time interval must obey.

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

For an example of timetable input, see “Find the Conditional Spectral Moments of Data Measurements in a Timetable” on page 1-232

order — Moment orders to return
integer scalar | integer vector

Moment orders to return, specified as one of the following:

- Integer — Compute one moment
- Vector — Compute multiple moments at once.

Example: `momentS = tfsmoment(x,2)` specifies the second-order spectral moment (variance) of the time-frequency distribution of `x`.

Example: `momentS = tfsmoment(x,[1 2 3 4])` specifies the first four moment orders of the time-frequency distribution of `x`.

You can specify any order and number of orders, but low-order moments carry less computational burden and are better suited to real-time applications. The first four moment orders correspond to the statistical moments of a data set:

1. Mean
2. Variance
3. Skewness (degree of asymmetry about the mean)
4. Kurtosis (length of outlier tails in the distribution — a normal distribution has a kurtosis of 3)

For examples, see:

- Timetable data input — “Find the Conditional Spectral Moments of Data Measurements in a Timetable” on page 1-232
- Time-series vector data input — “Determine Multiple Orders of Conditional Spectral Moment for a Time Series” on page 1-228

x — Time-series signal
vector

Time-series signal from which `tfsmoment` returns the moments, specified as a vector.

For an example of a time-series input, see “Plot the Conditional Spectral Moment of a Time Series Vector” on page 1-222

fs — Sample rate
positive scalar

Sample rate of `x`, specified as positive scalar in hertz when `x` is uniformly sampled.

ts — Sample-time values
`duration` scalar | vector | `duration` vector | `datetime` vector

Sample-time values, specified as one of the following:

- `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`.

`ts` can be nonuniform, with the `pspectrum` constraint that the median time interval and the mean time interval must obey:

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

p — Power spectrogram or spectrum of signal
vector | matrix

Power spectrogram or spectrum of a signal, specified as a matrix (spectrogram) or a column vector (spectrum). `p` contains an estimate of the short-term, time-localized power spectrum of a time-series

signal. If you specify `p`, `tfsmoment` uses `p` rather than generate its own power spectrogram. For an example, see “Use a Customized Power Spectrogram to Compute the Conditional Spectral Moment” on page 1-230.

`fp` — Frequencies for `p`

vector

Frequencies for power spectrogram or spectrum `p` when `p` is supplied explicitly to `tfsmoment`, specified as a vector in hertz. The length of `fp` must be equal to the number of rows in `p`.

`tp` — Time information for `p`

vector | `duration` vector | `datetime` vector | `duration` scalar

Time information for power spectrogram or spectrum `p` when `p` is supplied explicitly to `tfsmoment`, specified as one of the following:

- Vector of time points, whose data type can be numeric, `duration`, or `datetime`. The length of vector `tp` must be equal to the number of columns in `p`.
- `duration` scalar that represents the time interval in `p`. The scalar form of `tp` can be used only when `p` is a power spectrogram matrix.
- For the special case where `p` is a column vector (power spectrum), `tp` can be a numeric, `duration`, or `datetime` scalar representing the time point of the spectrum.

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: `'Centralize',false,'FrequencyLimits',[10 100]` computes the noncentralized conditional spectral moment for the portion of the signal ranging from 10 Hz to 100 Hz.

`Centralize` — Centralize-moment option

`true` (default) | `false`

Centralize-moment option, specified as the comma-separated pair consisting of `'Centralize'` and a logical.

- If `Centralize` is `true`, then `tfsmoment` returns the centralized conditional moment by subtracting the conditional mean (which is the first moment) in the computation.
- If `Centralize` is `false`, then `tfsmoment` returns the noncentralized moment, preserving any data offset.

For an example, see “Calculate a Conditional Spectral Moment that is not Centralized” on page 1-231.

`FrequencyLimits` — Frequency limits

full frequency band (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. This specification allows you to exclude a band of data at either end of the spectral range.

Output Arguments

momentS — Conditional spectral moment
timetable array | matrix

Conditional spectral moment returned as a `timetable` or a matrix.

- If you use `timetable` data `xt`, then `momentS` is a `timetable` array, containing variables which are the spectral moments for the orders specified in `order`. For an example, see “Find the Conditional Spectral Moments of Data Measurements in a Timetable” on page 1-232.
- If you use vector data `x`, or spectrogram data `p`, then `momentS` is an array whose columns represent the spectral moments. For an example, see “Determine Multiple Orders of Conditional Spectral Moment for a Time Series” on page 1-228.

t — Times of moment estimates
double vector

Times of moment estimates in seconds. `t` results from the time windowing that the internal spectrogram computation computes. The spectrogram windows require less time resolution than the original sample vector. Therefore, the returned `t` vector is more compact than the input data vectors, as is `momentS`. If time information has been provided by sample rate or sample time,`t` starts from the center of the first time window. If time information has been provided in `duration` or `datetime` format, `t` preserves the start-time offset.

More About

Conditional Spectral Moments

The conditional spectral moments of a nonstationary signal comprise a set of time-varying parameters that characterize the signal spectrum as it evolves in time. They are related to the conditional temporal moments and the joint time-frequency moments. The conditional spectral moment is an integral function of frequency, given time, and marginal distribution. The conditional temporal moment is an integral function of time, given frequency, and marginal distribution. The calculation of the joint time-frequency moment is a double integral that varies both time and frequency [1], [2].

Each moment is associated with a specific order, with the first four orders being the statistical properties of 1) mean, 2) variance, 3) skewness, and 4) kurtosis.

`tfsmoment` computes the conditional spectral moments of the time-frequency distribution for a signal `x`, for the orders specified in `order`. The function performs these steps:

1. Compute the spectrogram power spectrum, $P(t,f)$, of the input using the `pspectrum` function and uses it as a time-frequency distribution. If the syntax used supplies an existing $P(t,f)$, then `tfsmoment` uses that instead.
2. Estimate the conditional spectral moment $\langle \omega^m \rangle_t$ of the signal using, for the noncentralized case:

   $$\langle \omega^m \rangle_t = \frac{1}{P(t)} \int \omega^m P(t,\omega) d\omega,$$

   where m is the order and $P(t)$ is the marginal distribution.

   For the centralized conditional spectral moment $\mu_\omega^m(t)$, the function uses

$$\mu_{\omega}^{m}(t) = \frac{1}{P(t)}\int\left(\omega - \left\langle \omega^{1}\right\rangle_{t}\right)^{m} P(t,\omega)d\omega\,.$$

References

[1] Loughlin, P. J. "What Are the Time-Frequency Moments of a Signal?" *Advanced Signal Processing Algorithms, Architectures, and Implementations XI, SPIE Proceedings*. Vol. 4474, November 2001.

[2] Loughlin, P., F. Cakrak, and L. Cohen. "Conditional Moment Analysis of Transients with Application to Helicopter Fault Data." *Mechanical Systems and Signal Processing*. Vol 14, Issue 4, 2000, pp. 511–522.

See Also

`tftmoment` | `tfmoment` | `pspectrum`

Introduced in R2018a

tftmoment

Conditional temporal moment of the time-frequency distribution of a signal

Syntax

```
momentT = tftmoment(xt,order)
momentT = tftmoment(x,fs,order)
momentT = tftmoment(x,ts,order)
momentT = tftmoment(p,fp,tp,order)
momentT = tftmoment( ___ ,Name,Value)

[momentT,f] = tftmoment( ___ )

tftmoment( ___ )
```

Description

Time-frequency moments provide an efficient way to characterize signals whose frequencies change in time (that is, are nonstationary). Such signals can arise from machinery with degraded or failed hardware. Classical Fourier analysis cannot capture the time-varying frequency behavior. Time-frequency distribution generated by short-time Fourier transform (STFT) or other time-frequency analysis techniques can capture the time-varying behavior, but directly treating these distributions as features carries a high computational burden, and potentially introduces unrelated and undesirable feature characteristics. In contrast, distilling the time-frequency distribution results into low-dimension time-frequency moments provides a method for capturing the essential features of the signal in a much smaller data package. Using these moments significantly reduces the computational burden for feature extraction and comparison — a key benefit for real-time operation [1], [2].

The Predictive Maintenance Toolbox implements the three branches of time-frequency moment:

- Conditional spectral moment — `tfsmoment`
- Conditional temporal moment — `tftmoment`
- Joint time-frequency moment — `tfmoment`

`momentT = tftmoment(xt,order)` returns the conditional temporal moment on page 1-250 of `timetable` `xt` as a matrix. The `momentT` variables provide the temporal moments for the orders you specify in `order`. The data in `xt` can be nonuniformly sampled.

`momentT = tftmoment(x,fs,order)` returns the conditional temporal moment of time-series vector `x`, sampled at rate `fs`. The moment is returned as a matrix, in which each column represents a temporal moment corresponding to each element in `order`. With this syntax, `x` must be uniformly sampled.

`momentT = tftmoment(x,ts,order)` returns the conditional temporal moment of `x` sampled at the time instants specified by `ts` in seconds.

- If `ts` is a scalar `duration`, then `tftmoment` applies it uniformly to all samples.
- If `ts` is a vector, then `tftmoment` applies each element to the corresponding sample in `x`. Use this syntax for nonuniform sampling.

`momentT = tftmoment(p,fp,tp,order)` returns the conditional temporal moment of a signal whose power spectrogram is `p`. `fp` contains the frequencies corresponding to the temporal estimate contained in `p`. `tp` contains the vector of time instants corresponding to the centers of the windowed segments used to compute short-time power spectrum estimates. Use this syntax when:

- You already have the power spectrogram you want to use.
- You want to customize the options for `pspectrum`, rather than accept the default `pspectrum` options that `tftmoment` applies. Use `pspectrum` first with the options you want, and then use the output `p` as input for `tftmoment`. This approach also allows you to plot the power spectrogram.

`momentT = tftmoment( ___ ,Name,Value)` specifies additional properties using name-value pair arguments. Options include moment centralization and time-limit specification.

You can use `Name,Value` with any of the input-argument combinations in previous syntaxes.

`[momentT,f] = tftmoment( ___ )` returns the frequency vector `f` associated with the moment matrix in `momentT`.

You can use `f` with any of the input-argument combinations in previous syntaxes.

`tftmoment( ___ )` with no output arguments plots the conditional temporal moment. The plot x-axis is frequency, and the plot y-axis is the corresponding temporal moment.

You can use this syntax with any of the input-argument combinations in previous syntaxes.

Examples

Plot the Conditional Temporal Moments of a Time Series Vector

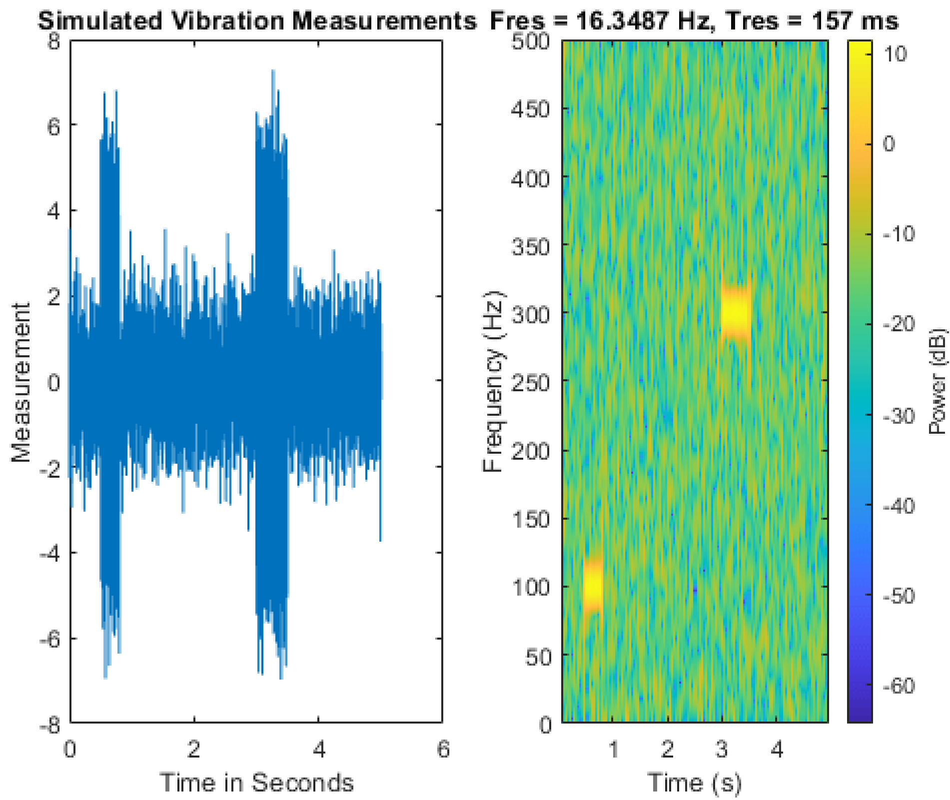
Plot the conditional temporal moments of a time series using a plot-only approach and a return-data approach.

Load and plot the data, which consists of simulated vibration measurements for a system with a fault that causes periodic resonances. `x` is the vector of measurements, and `fs` is the sampling frequency.

```
load tftmoment_example x fs

ts=0:1/fs:(length(x)-1)/fs;
figure
subplot(1,2,1)
plot(ts,x)
xlabel('Time in Seconds')
ylabel('Measurement')
title('Simulated Vibration Measurements')
```

Use the function `pspectrum` with the 'spectrogram' option to show the frequency content versus time.

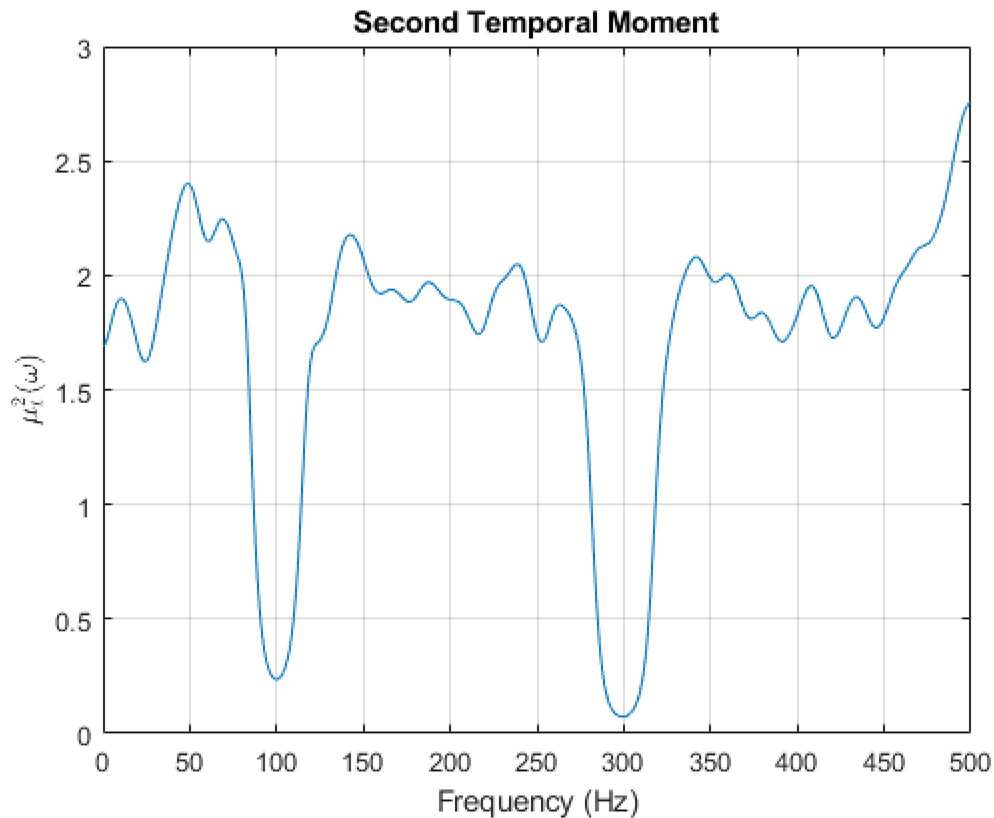
```
subplot(1,2,2)
pspectrum(x,ts,'spectrogram')
```



The spectrogram shows that the first burst is at 100 Hz, and the second burst is at 300 Hz. The 300-Hz burst is stronger than the 100-Hz burst by 70 dB.

Plot the second temporal moment (variance), using the plot-only approach with no output arguments and specifying `fs`.

```
figure
order = 2;
tftmoment(x,fs,order);title('Second Temporal Moment')
```



There are two distinct features in the plot at 100 and 300 Hz corresponding to the induced resonances shown by the spectrogram. The moments are much closer in magnitude than the spectral results were.

Now find the first four temporal moments, using the timeline `ts` that you already constructed. This time, use the form that returns both the moment vectors and the associated frequency vectors. Embed the order array as part of the input argument.

```
[momentT,f] = tftmoment(x,ts,[1 2 3 4]);
```

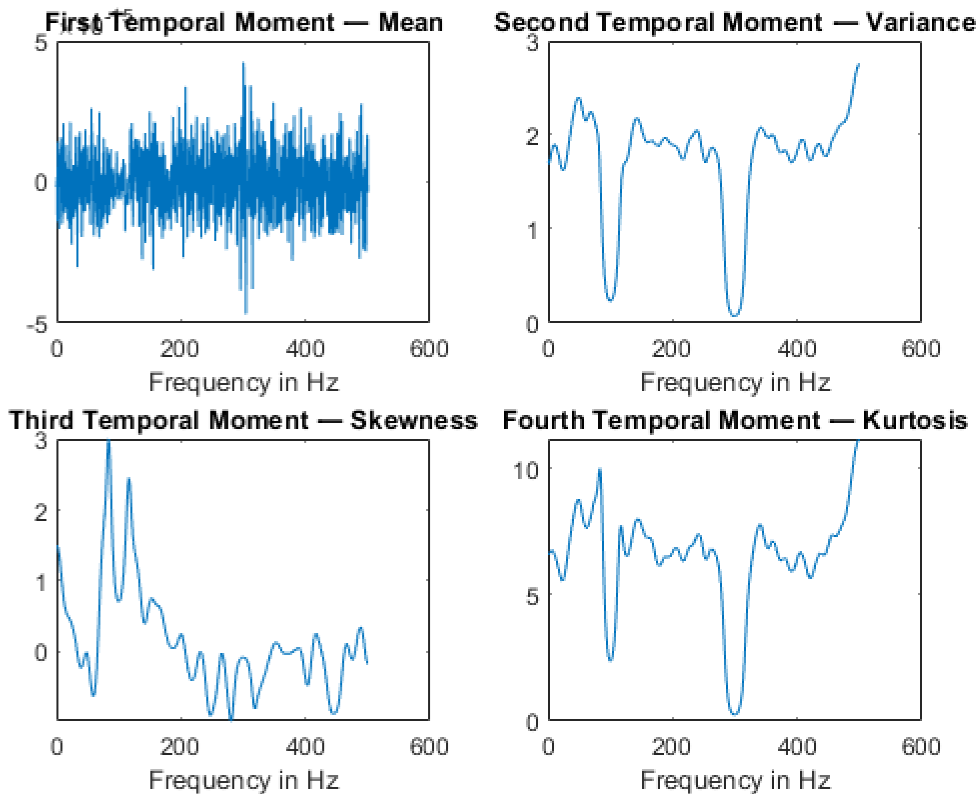
Each column of `momentT` contains the moment corresponding to one of the input orders.

```
momentT_1 = momentT(:,1);
momentT_2 = momentT(:,2);
momentT_3 = momentT(:,3);
momentT_4 = momentT(:,4);
```

Plot the four moments separately to compare the shapes.

```
figure
subplot(2,2,1)
plot(f,momentT_1)
title('First Temporal Moment — Mean')
xlabel('Frequency in Hz')

subplot(2,2,2)
plot(f,momentT_2)
```

```
title('Second Temporal Moment — Variance')
xlabel('Frequency in Hz')

subplot(2,2,3)
plot(f,momentT_3)
title('Third Temporal Moment — Skewness')
xlabel('Frequency in Hz')

subplot(2,2,4)
plot(f,momentT_4)
title('Fourth Temporal Moment — Kurtosis')
xlabel('Frequency in Hz')
```



For the data in this example, the second and fourth temporal moments show the clearest features for the faulty resonance.

Use an Existing Power Spectrogram to Compute the Conditional Temporal Moment

By default, `tfsmoment` calls the function `pspectrum` internally to generate the power spectrogram that `tftmoment` uses for the moment computation. You can also import an existing power spectrogram for `tftmoment` to use instead. This capability is useful if you already have a power spectrogram as a starting point, or if you want to customize the `pspectrum` options by generating the spectrogram explicitly first.

Input a power spectrogram that has already been generated using default options. Compare the resulting temporal-moment plot with one that `tftmoment` generates using its own `pspectrum` default options. The results should be the same.

Load the data, which consists of simulated vibration measurements for a system with a fault that causes periodic resonances. `p` is the previously computed spectrogram, `fp` and tp are the frequency and time vectors associated with p, x is the original vector of measurements,and fs is the sampling frequency,.

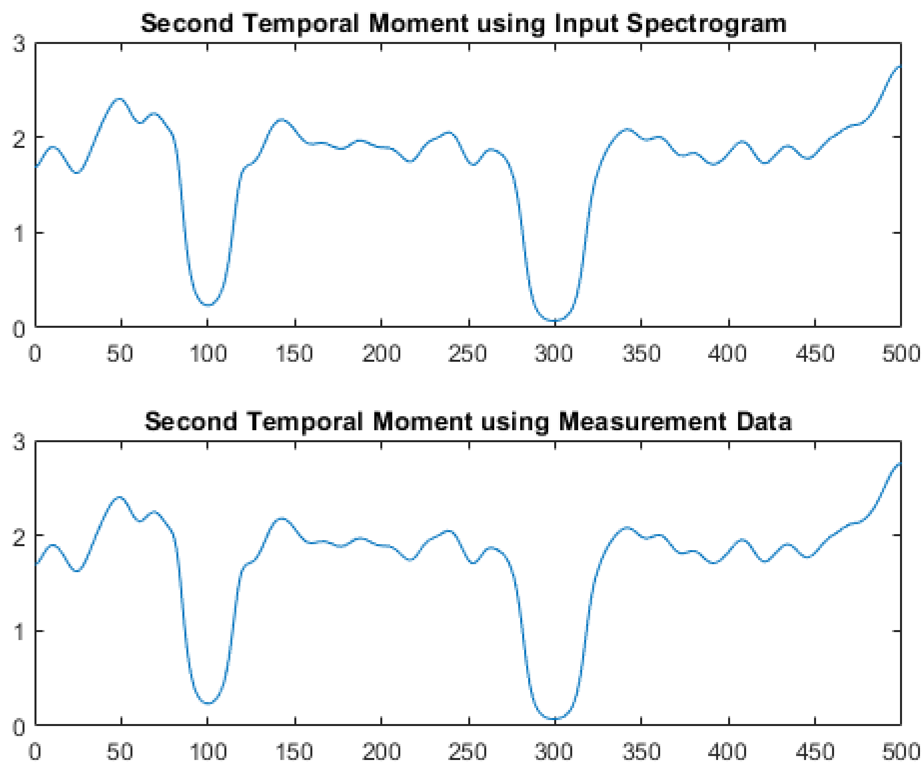
```
load tftmoment_example p fp tp x fs
```

Determine the second temporal moment using the spectrogram and its associated frequency and time vectors. Plot the moment.

```
[momentT_p,f_p] = tftmoment(p,fp,tp,2);
figure
subplot(2,1,1)
plot(f_p,momentT_p)
title('Second Temporal Moment using Input Spectrogram ')
```

Now find and plot the second temporal moments using the original data and sampling rate.

```
[momentT,f] = tftmoment(x,fs,2);
subplot(2,1,2)
plot(f,momentT)
title('Second Temporal Moment using Measurement Data')
```



As expected, the plots match since the default `pspectrum` options were used for both. This result demonstrates the equivalence between the two approaches when there is no customization.

Find the Conditional Temporal Moments of Data Measurements in a Timetable

Real-world measurements often come packaged as part of a time-stamped table that records actual time and readings rather than relative times. You can use the `timetable` format for capturing this data. This example shows how `tftmoment` operates with a timetable input, in contrast to the data vector inputs used for the other `tftmoment` examples, such as “Plot the Conditional Temporal Moments of a Time Series Vector” on page 1-241.

Load the data, which consists of a single timetable (`xt_inner1`) containing measurement readings and time information for a piece of machinery. Examine the properties of the timetable.

```
load tfmoment_tdata.mat xt_inner1;
xt_inner1.Properties
```

```
ans = 
  TimetableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Time'  'Variables'}
           VariableNames: {'x_inner1'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowTimes: [146484x1 duration]
               StartTime: 0 sec
              SampleRate: 4.8828e+04
                TimeStep: 2.048e-05 sec
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

This table consists of dimensions `Time` and the `Variables`, where the only variable is `x_inner1`.

Find the second and fourth conditional temporal moments (`order = [2 4]`) for the data in the timetable.

```
order = [2 4];
[momentT_xt_inner1,f] = tftmoment(xt_inner1,order);
size(momentT_xt_inner1)
```

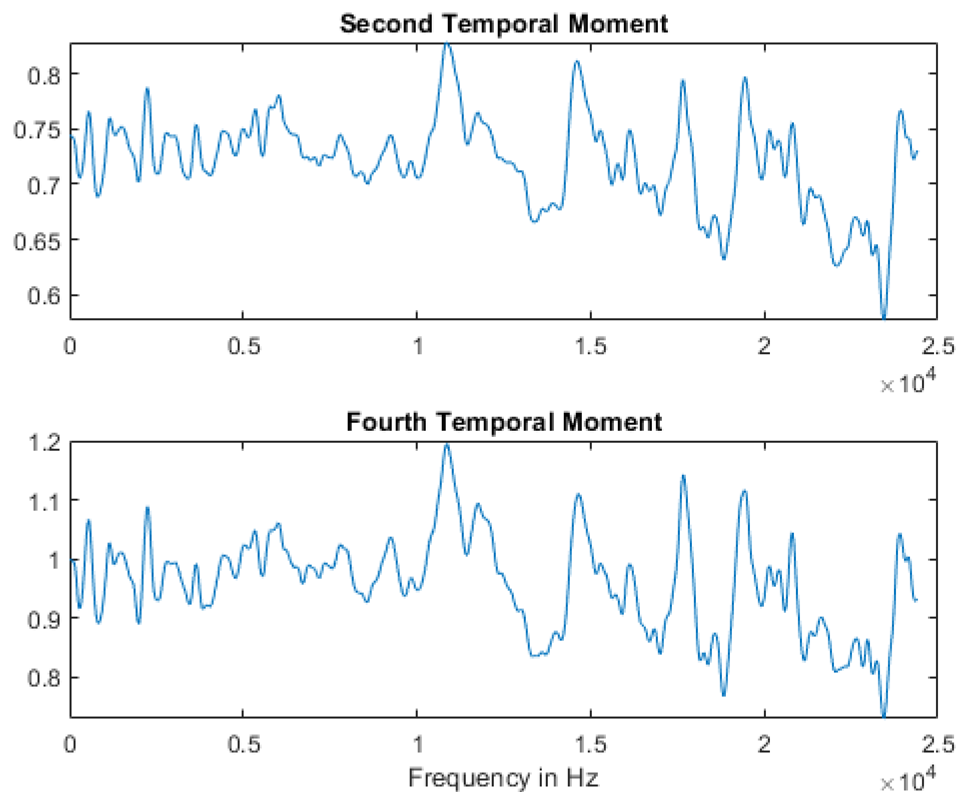
```
ans = 1×2

        1024           2
```

The temporal moments are represented by the columns of `momentT_xt_inner1`, just as they would be for a moment taken from a time series vector input.

Plot the moments versus returned frequency vector f.

```
momentT_inner1_2 = momentT_xt_inner1(:,1);
momentT_inner1_4 = momentT_xt_inner1(:,2);
```

```
figure
subplot(2,1,1)
plot(f,momentT_inner1_2)
title("Second Temporal Moment")

subplot(2,1,2)
plot(f,momentT_inner1_4)
title("Fourth Temporal Moment")
xlabel('Frequency in Hz')
```



Input Arguments

xt — Time-series signal
timetable

Time-series signal for which `tftmoment` returns the moments, specified as a `timetable` that contains a single variable with a single column. `xt` must contain increasing, finite row times. If the timetable has missing or duplicate time points, you can fix it using the tips in "Clean Timetable with Missing, Duplicate, or Nonuniform Times". `xt` can be nonuniformly sampled, with the `pspectrum` constraint that the median time interval and the mean time interval must obey:

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

For an example of `timetable` input, see “Find the Conditional Temporal Moments of Data Measurements in a Timetable” on page 1-246

order — Moment orders to return
integer scalar | integer vector

Moment orders to return, specified as one of the following:

- Integer — Compute one moment.
- Vector — Compute multiple moments at once.

Example: `momentT = tftmoment(x,2)` specifies the second-order temporal moment (variance) of the time-frequency distribution of `x`.

Example: `momentT = tftmoment(x,[1 2 3 4])` specifies the first four moment orders of the time-frequency distribution of `x`.

You can specify any order and number of orders, but low-order moments carry less computational burden and are better suited to real-time applications. The first four moment orders correspond to the statistical moments of a data set:

1. Mean ("group delay" for temporal data)
2. Variance
3. Skewness (degree of asymmetry about the mean)
4. Kurtosis (length of outlier tails in the distribution — a normal distribution has a kurtosis of 3)

For examples, see:

- Timetable data input — “Find the Conditional Temporal Moments of Data Measurements in a Timetable” on page 1-246
- Time-series vector data input — “Plot the Conditional Temporal Moments of a Time Series Vector” on page 1-241

x — Time-series signal
vector

Time-series signal from which `tftmoment` returns the moments, specified as a vector.

For an example of a time-series input, see “Plot the Conditional Temporal Moments of a Time Series Vector” on page 1-241

fs — Sample rate
positive scalar

Sample rate of `x`, specified as positive scalar in hertz when `x` is uniformly sampled.

ts — Sample-time values
`duration` scalar | vector | `duration` vector | `datetime` vector

Sample-time values, specified as one of the following:

- `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`.

ts can be nonuniform, with the pspectrum constraint that the median time interval and the mean time interval must obey:

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

p — Power spectrogram or spectrum of signal
matrix | vector

Power spectrogram or spectrum of a signal, specified as a matrix (spectrogram) or a column vector (spectrum). p contains an estimate of the short-term, time-localized power spectrum of a time-series signal. If you specify p, then tftmoment uses p rather than generate its own power spectrogram. For an example, see “Use a Customized Power Spectrogram to Compute the Conditional Spectral Moment” on page 1-230.

fp — Frequencies for p
vector

Frequencies for power spectrogram or spectrum p when p is supplied explicitly to tftmoment, specified as a vector in hertz. The length of fp must be equal to the number of rows in p.

tp — Time information for p
vector | duration vector | datetime vector | duration scalar

Time information for power spectrogram or spectrum p when p is supplied explicitly to tftmoment, specified as one of the following:

- Vector of time points, whose data type can be numeric, duration, or datetime. The length of vector tp must be equal to the number of columns in p.
- duration scalar that represents the time interval in p. The scalar form of tp can be used only when p is a power spectrogram matrix.
- For the special case where p is a column vector (power spectrum), tp can be a numeric, duration, or datetime scalar representing the time point of the spectrum.

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: 'Centralize',false,'TimeLimits',[20 100] computes the noncentralized conditional temporal moment for the portion of the signal ranging from 20 sec to 100 sec.

Centralize — Centralize-moment option
true (default) | false

Centralize-moment option, specified as the comma-separated pair consisting of 'Centralize' and a logical.

- If Centralize is true, then tftmoment returns the centralized conditional moment by subtracting the conditional mean (which is the first moment) in the computation.
- If Centralize is false, then tftmoment returns the noncentralized moment, preserving any data offset.

Example: momentT = tftmoment(x,2,'Centralize',false).

`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 `ts`, and of the data types:

- Numeric or `duration` when `fs` or a scalar `ts` are specified, or when `ts` is a numeric or `duration` vector
- Numeric, `duration`, or `datetime` when `ts` is specified as a `datetime` vector

This specification allows you to extract a temporal section of data from a longer data set.

Output Arguments

`momentT` — Conditional temporal moment
matrix

Conditional temporal moment returned as a matrix whose columns represent the temporal moments.

`momentT` is a matrix with one or more columns, regardless of whether the input data is `timetable` `xt`, time-series vector `x`, or spectrogram data `p`.

`f` — Frequencies of moment estimates
double vector

Frequencies of moment estimates in hertz, specified as a double vector. For an example, see “Plot the Conditional Temporal Moments of a Time Series Vector” on page 1-241

More About

Conditional Temporal Moments

The conditional temporal moments of a nonstationary signal comprise a set of time-varying parameters that characterize the group delay as it evolves in time. They are related to the conditional spectral moment on page 1-238 and the joint time-frequency moments. The conditional spectral moment is an integral function of frequency, given time, and marginal distribution. The conditional temporal moment is an integral function of time, given frequency, and marginal distribution. The joint time-frequency moment is a double integral that varies both time and frequency [1], [2].

Each moment is associated with a specific order, with the first four orders being the statistical properties of 1) mean, 2) variance, 3) skewness, and 4) kurtosis.

`tftmoment` computes the conditional temporal moments of the time-frequency distribution for a signal `x`, for the orders specified in `order`. The function performs these steps:

1. Compute the spectrogram power spectrum, $P(t,f)$, of the input using the `pspectrum` function and uses it as a time-frequency distribution. If the syntax used supplies an existing $P(t,f)$, then `tftmoment` uses that instead.
2. Estimate the conditional temporal moment $\langle t^n \rangle_\omega$ of the signal using, for the non-centralized case:

$$\langle t^n \rangle_\omega = \frac{1}{P(\omega)} \int t^n P(t,\omega) dt,$$

where m is the order and $P(t)$ is the marginal distribution.

For the centralized conditional temporal moment $\mu_t^n(\omega)$, the function uses

$$\mu_t^n(\omega) = \frac{1}{P(\omega)} \int \left(t - \left\langle t^1 \right\rangle_\omega\right)^n P(t, \omega) dt \,.$$

References

[1] Loughlin, P. J. "What are the time-frequency moments of a signal?" *Advanced Signal Processing Algorithms, Architectures, and Implementations XI, SPIE Proceedings*. Vol. 4474, November 2001.

[2] Loughlin, P., F. Cakrak, and L. Cohen. "Conditional moment analysis of transients with application to helicopter fault data." *Mechanical Systems and Signal Processing*. Vol 14, Issue 4, 2000, pp. 511–522.

See Also

`tfsmoment` | `tfmoment` | `pspectrum`

Introduced in R2018a

time2num

Convert `duration` or `datetime` array into numeric vector with the specified time unit

Syntax

```
[x,origUnit] = time2num(T,targetUnit)
```

Description

`time2num` is a function used in code generated by **Diagnostic Feature Designer**.

`[x,origUnit] = time2num(T,targetUnit)` converts the time array `T` from its original data type and unit into a numeric vector `x` expressed in the unit of `targetUnit`. For instance, suppose that `T` is a `datetime` vector that contains timestamps for data measurements, and you want to convert `T` into a numeric vector `x` that expresses the time in units of hours. Use `x = time2num(T,"hours")`.

If `x` is already a numeric vector, `time2num` sets `x` to `T` and ignores `targetUnit`.

Code that is generated by **Diagnostic Feature Designer** uses `time2num` when performing spectral processing and other computations.

Input Arguments

T — Time array
`datetime` array | `duration` array | `numeric` vector

Time array of sampling instants, expressed as a one-dimensional `datetime` array, a one-dimensional `duration` array, or a numeric vector.

`targetUnit` — Time unit
`"seconds"` | `"minutes"` | `"hours"` | `"days"` | `"years"` | `" "`

Time unit corresponding to the converted numeric vector, specified as a string. `targetUnit` can be one of the following:

- `"seconds"`
- `"minutes"`
- `"hours"`
- `"days"`
- `"years"`
- `""`

If you omit `targetUnit` or set `targetUnit` to `""`, then `time2num` derives the original time unit from `T`.

- If `T` is a `duration` array, then `time2num` sets `timeUnit` to the unit of the `duration` array.

- If `T` is a `datetime` array, then `time2num` determines the best value for `timeUnit` based on sample time. For instance, if the timestamps in `T` are 100 seconds apart, `time2num` sets `timeUnit` to `"minutes"`.
- If `T` is a numeric array, then `time2num` ignores `targetUnit` and sets `x` to `T`.

Example: `tNumeric = time2num(Tacho.Time,"seconds")`

Output Arguments

x — Sampling instants

numeric vector

Sampling instants, returned as a numeric vector. The starting point `x(1)` depends on the data type of `T`.

- If `T` is a `datetime` array, then `x(1)` is 0.
- If `T` is a `duration` array or a numeric vector, then `x(1)` is `T(1)`, converted to the unit in `targetUnit` if `targetUnit` is specified.
- If `T` is a numeric vector, then `x(1)` is `T(1)`.

origUnit — Original unit

string

Original unit of `T`, returned as a string.

See Also

`datetime` | `duration` | `effectivefs`

Introduced in R2020a

trendability

Measure of similarity between trajectories of condition indicators

Syntax

```
Y = trendability(X)
Y = trendability(X,lifetimeVar)
Y = trendability(X,lifetimeVar,dataVar)
Y = trendability(X,lifetimeVar,dataVar,memberVar)
Y = trendability( ___ ,Name,Value)

trendability( ___ )
```

Description

`Y = trendability(X)` returns the trendability of the lifetime data `X`. Use `trendability` as measure of similarity between the trajectories of a feature measured in several run-to-failure experiments. A more trendable feature has trajectories with the same underlying shape. The values of `Y` range from 0 to 1, where `Y` is 1 if `X` is perfectly trendable and 0 if `X` is non-trendable.

`Y = trendability(X,lifetimeVar)` returns the trendability of the lifetime data `X` using the lifetime variable `lifetimeVar`.

`Y = trendability(X,lifetimeVar,dataVar)` returns the trendability of the lifetime data `X` using the data variables specified by `dataVar`.

`Y = trendability(X,lifetimeVar,dataVar,memberVar)` returns the trendability of the lifetime data `X` using the lifetime variable `lifetimeVar`, the data variables specified by `dataVar`, and the member variable `memberVar`.

`Y = trendability( ___ ,Name,Value)` estimates the trendability with additional options specified by one or more `Name,Value` pair arguments. You can use this syntax with any of the previous input-argument combinations.

`trendability( ___ )` with no output arguments plots a bar chart of ranked trendability values.

Examples

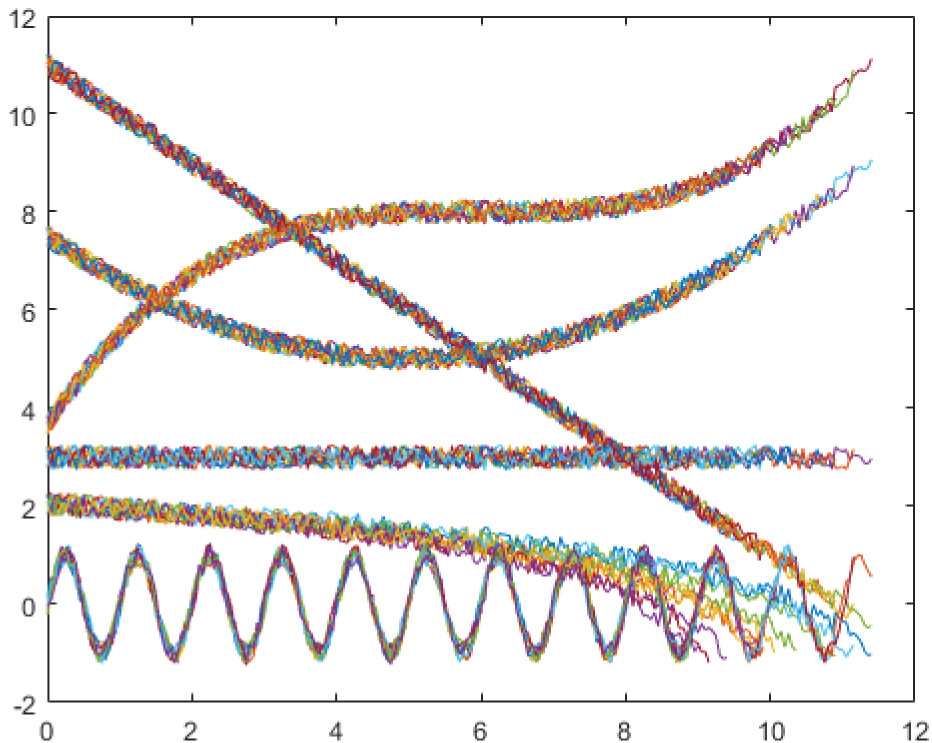
Trendability of Data in Cell Array of Matrices

In this example, consider the lifetime data of 10 identical machines with the following 6 potential prognostic parameters—constant, linear, quadratic, cubic, logarithmic, and periodic. The data set `machineDataCellArray.mat` contains `C` which is a `1x10` cell array of matrices where each element of the cell array is a matrix that contains the lifetime data of a machine. For each matrix in the cell array, the first column contains the time while the other columns contain the data variables.

Load the lifetime data and visualize it against time.

```
load('machineDataCellArray.mat','C')
display(C)

C=1×10 cell array
  Columns 1 through 4

    {219x7 double}    {189x7 double}    {202x7 double}    {199x7 double}

  Columns 5 through 8

    {229x7 double}    {184x7 double}    {224x7 double}    {208x7 double}

  Columns 9 through 10

    {181x7 double}    {197x7 double}
```

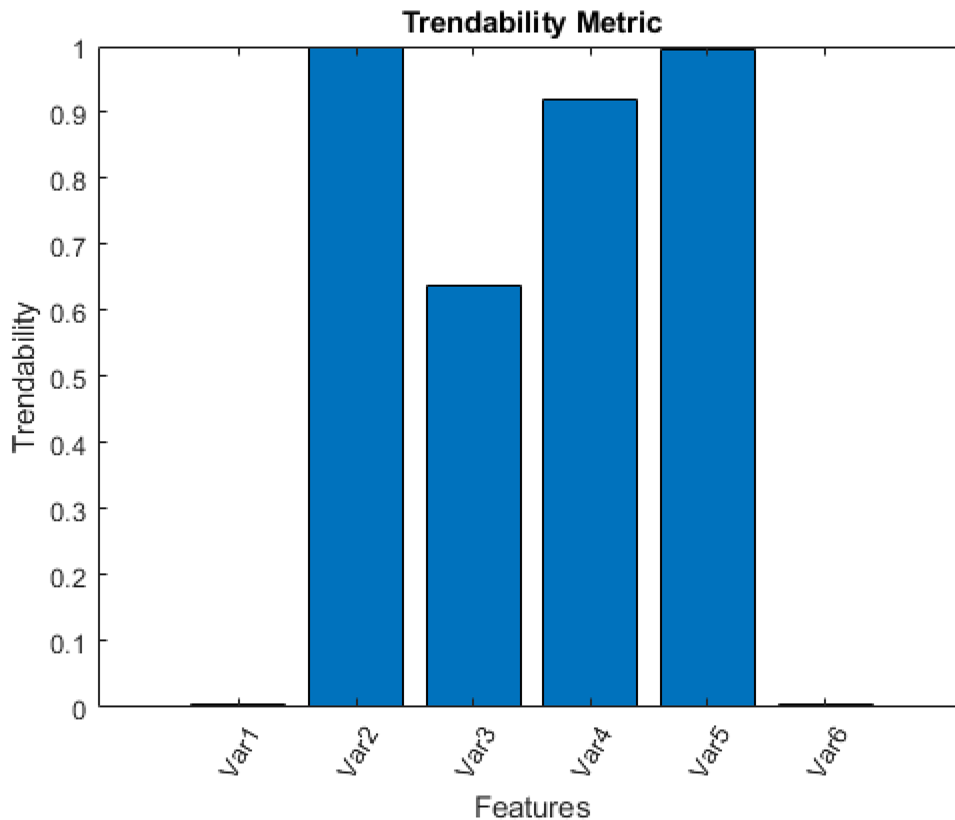
```
for k = 1:length(C)
    plot(C{k}(:,1), C{k}(:,2:end));
    hold on;
end
```



Observe the 6 different condition indicators–constant, linear, quadratic, cubic, logarithmic, and periodic–for all 10 machines on the plot.

Visualize the trendability of the potential prognostic features.

```
trendability(C)
```



From the histogram plot, observe that the features `Var2` and `Var5` have trendability values of 1. Hence, these features are more appropriate for remaining useful life predictions since they are the best indicators of machine health.

Trendability of Data in Cell Array of Tables

In this example, consider the lifetime data of 10 identical machines with the following 6 potential prognostic parameters—constant, linear, quadratic, cubic, logarithmic, and periodic. The data set `machineDataTable.mat` contains `T`, which is a `1x10` cell array of tables where each element of the cell array contains a table of lifetime data for a machine.

Load and display the data.

```
load('machineDataTable.mat','T');
display(T)
```

```
T=1×10 cell array
  Columns 1 through 4

    {219x7 table}    {189x7 table}    {202x7 table}    {199x7 table}

  Columns 5 through 8

    {229x7 table}    {184x7 table}    {224x7 table}    {208x7 table}
```

```
  Columns 9 through 10

    {181x7 table}    {197x7 table}
```

```
head(T{1},2)
```

```
ans=2×7 table
    Time    Constant    Linear    Quadratic    Cubic     Logarithmic    Periodic
    ____    ________    ______    _________    ______    ___________    ________

       0     3.2029     11.203     7.7029      3.8829      2.2517        0.2029
    0.05     2.8135     10.763     7.2637      3.6006      1.8579       0.12251
```

Note that every table in the cell array contains the lifetime variable `'Time'` and the data variables `'Constant'`, `'Linear'`, `'Quadratic'`, `'Cubic'`, `'Logarithmic'`, and `'Periodic'`.

Compute trendability with `Time` as the lifetime variable.

```
Y = trendability(T,'Time')
```

```
Y=1×6 table
    Constant     Linear     Quadratic     Cubic     Logarithmic    Periodic
    _________    _______    _________    _______    ___________    _________

    0.0035529    0.99984     0.63753     0.92057      0.99582      0.0041995
```

From the resultant table of trendability values, observe that the linear, cubic, and logarithmic features have values closer to 1. Hence, these three features are more appropriate for predicting remaining useful life since they are the best indicators of machine health.

Visualize Trendability of Lifetime Data in Ensemble Datastore

Consider the lifetime data of 4 machines. Each machine has 4 fault codes for the potential condition indicators—voltage, current, and power. `trendabilityEnsemble.zip` is a collection of 4 files where every file contains a timetable of lifetime data for each machine - `tbl1.mat`, `tbl2.mat`, `tbl3.mat` and `tbl4.mat`. You can also use files containing data for multiple machines. For each timetable, the organization of the data is as follows:

Time	Voltage	Current	Power	FaultCode	Machine

When you perform calculations on tall arrays, MATLAB® uses either a parallel pool (default if you have Parallel Computing Toolbox™) or the local MATLAB session. To run the example using the local MATLAB session, change the global execution environment by using the `mapreducer` function.

```
mapreducer(0)
```

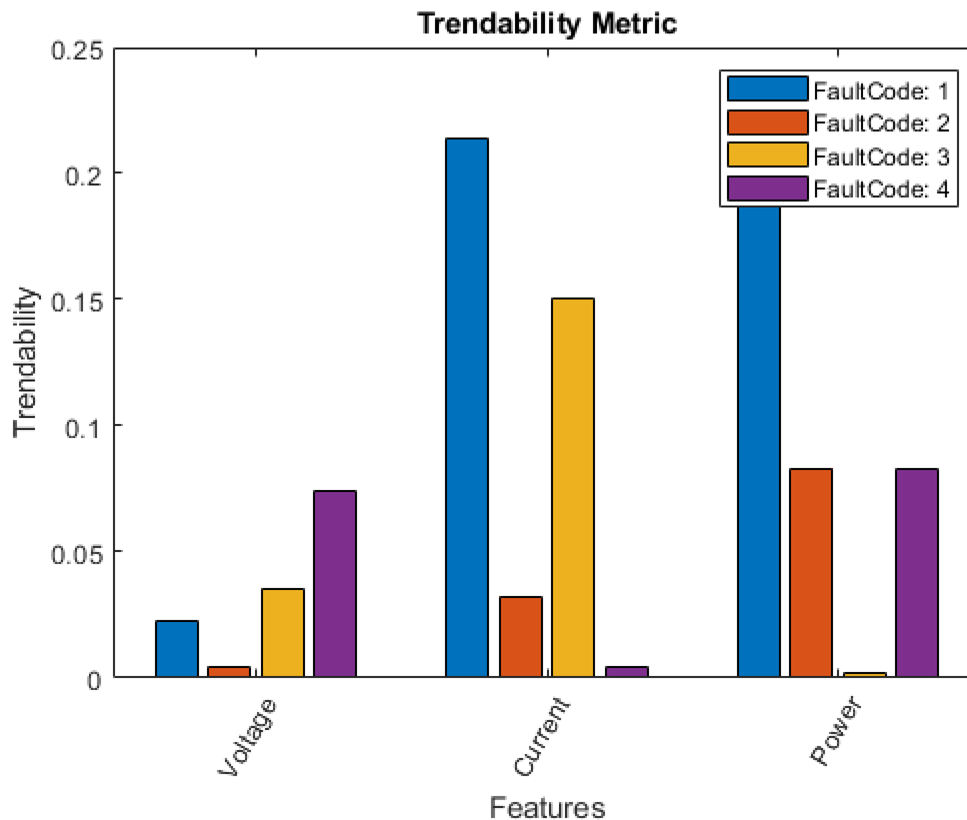
Extract the compressed files, read the data in the timetables, and create a `fileEnsembleDatastore` object using the timetable data. For more information on creating a file ensemble datastore, see `fileEnsembleDatastore`.

```
unzip trendabilityEnsemble.zip;
ens = fileEnsembleDatastore(pwd,'.mat');
ens.DataVariables = {'Voltage','Current','Power','FaultCode','Machine'};
% Make sure that the function for reading data is on path
addpath(fullfile(matlabroot,'examples','predmaint','main'))
ens.ReadFcn = @readtable_data;
ens.SelectedVariables = {'Voltage','Current','Power','FaultCode','Machine'};
```

Visualize the trendability of the potential prognostic features with `'Machine'` as the member variable and group the lifetime data by `'FaultCode'`. Grouping the lifetime data ensures that `trendability` calculates the metric for each fault code separately.

```
trendability(ens,'MemberVariable','Machine','GroupBy','FaultCode');
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.13 sec
Evaluation completed in 0.32 sec
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.057 sec
Evaluation completed in 0.19 sec
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.21 sec
Evaluation completed in 0.24 sec
```



`trendability` returns a histogram plot with the features ranked by their trendability values. A higher trendability value indicates a more suitable prognostic parameter. For instance, the candidate feature `Current` has the highest degree of trendability for machines with `FaultCode` 1.

```
rmpath(fullfile(matlabroot,'examples','predmaint','main')) % Reset path
```

Input Arguments

X — Lifetime data

cell array of matrices | cell array of tables and timetables | `fileEnsembleDatastore` object | table | timetable

Lifetime data, specified as a cell array of matrices, cell array of tables and timetables, `fileEnsembleDatastore` object, table, or timetable. Lifetime data contains run-to-failure data of the systems being monitored. The term *lifetime* here refers to the life of the machine defined in terms of the units you use to measure system life. Units of lifetime can be quantities such as the distance traveled (miles), fuel consumed (gallons), or time since the start of operation (days).

If `X` is

- a cell array of matrices or tables, the function assumes that each matrix or table contains columns of lifetime data for a system. Each column of every matrix or table, except the first column, contains data for a prognostic variable. `'Var1','Var2', ...` can be used to refer to the matrix columns that contain the lifetime data. For instance, the file `machineDataCellArray.mat` contains a 1-by-10 cell array of matrices `C`, where each of the 10 matrices contains data for a particular machine.
- a table or timetable, the function assumes that each column, except the first one, contains columns of lifetime data. The table variable names can be used to refer to the columns that contain the lifetime data. If `lifetimeVar` is not specified when `X` is a table, then the first data column is used as the lifetime variable.
- a `fileEnsembleDatastore` object, specify the data variables `dataVar` and member variables `memberVar` to be used. If `lifetimeVar` is not specified, then the first data column is used as the lifetime variable for computation.

Each numerical member in `X` is of type `double`.

lifetimeVar — Lifetime variable

string | character vector

Lifetime variable, specified as a string or character vector. `lifetimeVar` measures the lifetime of the systems being monitored and the lifetime data is sorted with respect to `lifetimeVar`. The value of `lifetimeVar` must be a valid ensemble or table variable name.

For a cell array of matrices, the value `'Time'` can be used to refer to the first column of each matrix, which is assumed to contain the lifetime variable. For instance, the file `machineDataCellArray.mat` contains the cell array `C`, where the first column in each matrix contains the lifetime variable while the other columns contain the data variables.

dataVar — Data variables

string array | character vector | cell array of character vectors

Data variables, specified as a string array, character vector, or cell array of character vectors. Data variables are the main content of the members of an ensemble. Data variables can include measured data or derived data for the analysis and development of predictive maintenance algorithms.

If `X` is

- a `fileEnsembleDatastore` object, the value of `dataVar` supersedes the `DataVariables` property of the ensemble.
- a cell array of matrices, the value `'Time'` can be used to refer to the first column of each matrix, that is, the lifetime variable `lifetimeVar`. `'Var1','Var2', ...` can be used to refer to the other matrix columns which contain the lifetime data. For instance, the file `machineDataCellArray.mat` contains the cell array `C` where the first column in each matrix contains the lifetime variable. The other columns in the cell array `C` contain the data variables.
- a table, the table variable names can be used to refer to the columns which contain the lifetime data.

The values of `dataVar` must be valid ensemble or table variable names. If `dataVar` is not specified, the computation includes all data columns except the one specified in `lifetimeVar`. For instance, suppose that each entry in a cell array is a table with variables `A`, `B`, `C`, and `D`. Setting `dataVar` to `["A","D"]` uses only `A` and `D` for the computation while `C` and `D` are ignored.

`memberVar` — Member variable
string | character vector

Member variable, specified as a string or character vector. Use `memberVar` to specify the variable for identifying the systems or machines in lifetime data `X`. For instance, in the `fileEnsembleDatastore` object, the fifth column in each timetable contains numbers that identify data from a particular machine. The column name corresponds to the member variable `memberVar`.

`memberVar` is ignored when `X` is specified as a cell array of matrices or tables.

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','rank'`

`LifeTimeVariable` — Lifetime variable
`strings(0)` (default) | string | character vector

Lifetime variable, specified as the comma-separated pair consisting of '`LifeTimeVariable`' and either a string or character vector. If '`LifeTimeVariable`' is not specified, then the first data column is used.

'`LifeTimeVariable`' is equivalent to the input argument `lifetimeVar`.

`DataVariables` — Data variables
`strings(0)` (default) | string array | character vector | cell array of character vectors

Data variables, specified as the comma-separated pair consisting of '`DataVariables`' and either a string array, character vector or cell array of character vectors.

'`DataVariables`' is equivalent to the input argument `dataVar`.

`MemberVariable` — Member variables
[] (default) | string | character vector

Member variables, specified as the comma-separated pair consisting of '`MemberVariable`' and either a string or character vector.

'MemberVariable' is equivalent to the input argument memberVar.

GroupBy — Grouping criterion
[] (default) | string | character vector

Grouping criterion, specified as the comma-separated pair consisting of 'GroupBy' and either a string or character vector. Use 'GroupBy' to specify the variables for grouping the lifetime data X by operating conditions.

The function computes the metric separately for each group that results from applying the criterion, such as a fault condition, specified by 'GroupBy'. For instance, in the fileEnsembleDatastore object ens, the fourth column in each timetable in ens contains the variable 'FaultCode'. The metric is computed for each machine by grouping the data by 'FaultCode'.

You can only group variables when X is defined as a fileEnsembleDatastore object, table, timetable, or cell array of tables or timetables.

WindowSize — Size of the centered moving average window for data smoothing
[] (default) | scalar | two-element vector

Size of the centered moving average window for data smoothing, specified as the comma-separated pair consisting of 'WindowSize' and either a scalar or two-element vector. A Savitzky-Golay filter is used for data smoothing. For more information, see smoothdata.

If 'WindowSize' is not specified, the window length is automatically determined from lifetime data X using smoothdata(X,'sgolay'). Set 'WindowSize' to 0 to turn off data smoothing.

Output Arguments

Y — Trendability of lifetime data
vector | table

Trendability of lifetime data, returned as a vector or table.

Trendability is the measure of similarity between the trajectories of a feature measured in several run-to-failure experiments. A more trendable feature has trajectories with the same underlying shape. As a system gets progressively closer to failure, a suitable condition indicator is typically highly trendable. Conversely, any feature that is non-trendable is a less suitable condition indicator. The values of Y range from 0 to 1.

- Y is 1 if X is perfectly trendable.
- Y is 0 if X is perfectly non-trendable.

Selecting appropriate estimation parameters out of all available features is the first step in building a reliable remaining useful life prediction engine. The trendability values in Y are useful to determine which condition indicators best track the degradation process of systems being monitored. The higher the trendability, the more desirable the feature is for prognostics.

When 'GroupBy' is not specified, then Y is returned as a row vector or single-row table. Conversely, when 'GroupBy' is specified, then each row in Y corresponds to one group.

Limitations

- When X is a tall table or tall timetable, `trendability` nevertheless loads the complete array into memory using `gather`. If the memory available is inadequate, then `trendability` returns an error.

Algorithms

The computation of trendability uses this formula:

$$\text{trendability} = \min_{j,k} \left|\text{corr}\left(x_j, x_k\right)\right|, \qquad j,k = 1,\ldots,M$$

where x_j represents the vector of measurements of a feature on the jth system and the variable M is the number of systems monitored.

When x_j and x_k have different lengths, the shorter vector is resampled to match the length of the longer vector. To facilitate this process, their time vectors are first normalized to percent lifetime, that is, [0%, 100%].

References

[1] Coble, J., and J. W. Hines. "Identifying Optimal Prognostic Parameters from Data: A Genetic Algorithms Approach." In *Proceedings of the Annual Conference of the Prognostics and Health Management Society*. 2009.

[2] Coble, J. "Merging Data Sources to Predict Remaining Useful Life - An Automated Method to Identify Prognostics Parameters." Ph.D. Thesis. University of Tennessee, Knoxville, TN, 2010.

[3] Lei, Y. *Intelligent Fault Diagnosis and Remaining Useful Life Prediction of Rotating Machinery*. Xi'an, China: Xi'an Jiaotong University Press, 2017.

[4] Lofti, S., J. B. Ali, E. Bechhoefer, and M. Benbouzid. "Wind turbine high-speed shaft bearings health prognosis through a spectral Kurtosis-derived indices and SVR." *Applied Acoustics* Vol. 120, 2017, pp. 1-8.

See Also

`prognosability` | `monotonicity` | `fileEnsembleDatastore`

Topics

"Feature Selection for Remaining Useful Life Prediction"

Introduced in R2018b

tsadifference

Difference signal of a time-synchronous averaged signal

Syntax

```
Y = tsadifference(X,fs,rpm,orderList)
Y = tsadifference(X,t,rpm,orderList)
Y = tsadifference(XT,rpm,orderList)
[Y,S] = tsadifference( ___ )
 ___ = tsadifference( ___ )

tsadifference( ___ )
```

Description

`Y = tsadifference(X,fs,rpm,orderList)` computes the difference signal `Y` of the time-synchronous averaged (TSA) signal vector `X` using sampling rate `fs`, the rotational speed `rpm`, and the orders to be filtered `orderList`. `Y` is computed by removing the regular signal, the value of '`NumSidebands`', and their respective harmonics from `X`. For more information on regular signal, see `tsaregular`.

You can use `Y` to further extract condition indicators of rotating machinery for predictive maintenance. For example, extracting the FM4 indicator from `Y` is useful to detect faults isolated to only a limited number of teeth in a gear mesh.

`Y = tsadifference(X,t,rpm,orderList)` computes the difference signal `Y` of the TSA signal vector `X` with corresponding time values from `t`.

`Y = tsadifference(XT,rpm,orderList)` computes the difference signal `Y` of the TSA signal stored in the timetable `XT`. `XT` must contain a single numeric column variable.

`[Y,S] = tsadifference( ___ )` returns the amplitude spectrum `S` of the difference signal `Y`. `S` is the amplitude spectrum computed using the normalized fast Fourier transform (FFT) of `Y`.

`___ = tsadifference( ___ )` allows you to specify additional parameters using one or more name-value pair arguments. You can use this syntax with any of the previous input and output arguments.

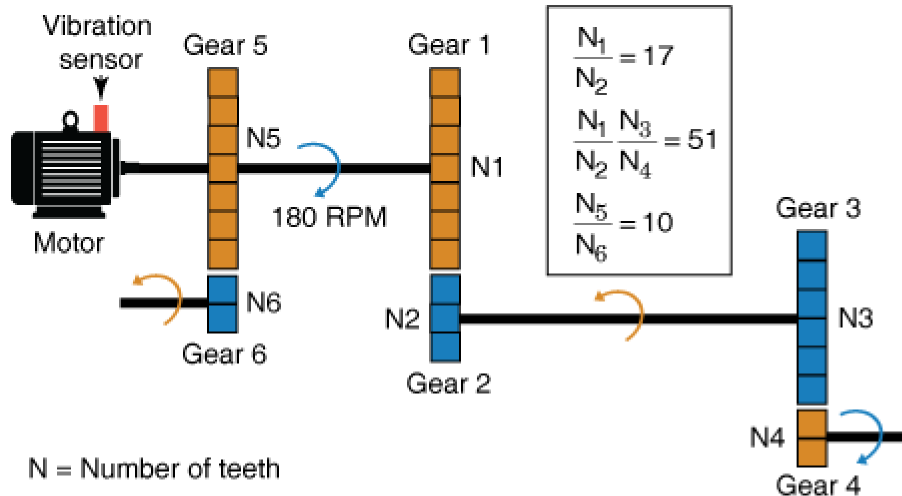
`tsadifference( ___ )` with no output arguments plots the time-domain and frequency-domain plots of the raw and difference TSA signals.

Examples

Find and Visualize the Difference Signal of a Compound TSA Signal

Consider a drivetrain with six gears driven by a motor that is fitted with a vibration sensor, as depicted in the figure below. Gear 1 on the motor shaft meshes with gear 2 with a gear ratio of 17:1. The final gear ratio, that is, the ratio between gears 1 and 2 and gears 3 and 4, is 51:1. Gear 5, also on the motor shaft, meshes with gear 6 with a gear ratio of 10:1. The motor is spinning at 180 RPM,

and the sampling rate of the vibration sensor is 50 KHz. To obtain the signal containing just the meshing components for gears 5 and 6, filter out the components of the shaft rotation, gears 1 and 2 and, 3 and 4 by specifying their gear ratios of 17 and 51 in `orderList`. The signal components corresponding to the shaft rotation (order = 1) is always implicitly included in the computation.




```
rpm = 180;
fs = 50e3;
t = (0:1/fs:(1/3)-1/fs)';                              % sample times
orderList = [17 51];
f = rpm/60*[1 orderList 10];
```

In practice, you would use measured data such as vibration signals obtained from an accelerometer. For this example, generate TSA signal X, which is the simulated data from the vibration sensor mounted on the motor.

```
X = sin(2*pi*f(1)*t) + sin(2*pi*2*f(1)*t) + ...      % motor shaft rotation and harmonic
    3*sin(2*pi*f(2)*t) + 3*sin(2*pi*2*f(2)*t) + ... % gear mesh vibration and harmonic for gears
    4*sin(2*pi*f(3)*t) + 4*sin(2*pi*2*f(3)*t) + ... % gear mesh vibration and harmonic for gears
    2*sin(2*pi*10*f(1)*t);                          % gear mesh vibration for gears 5 and 6
```

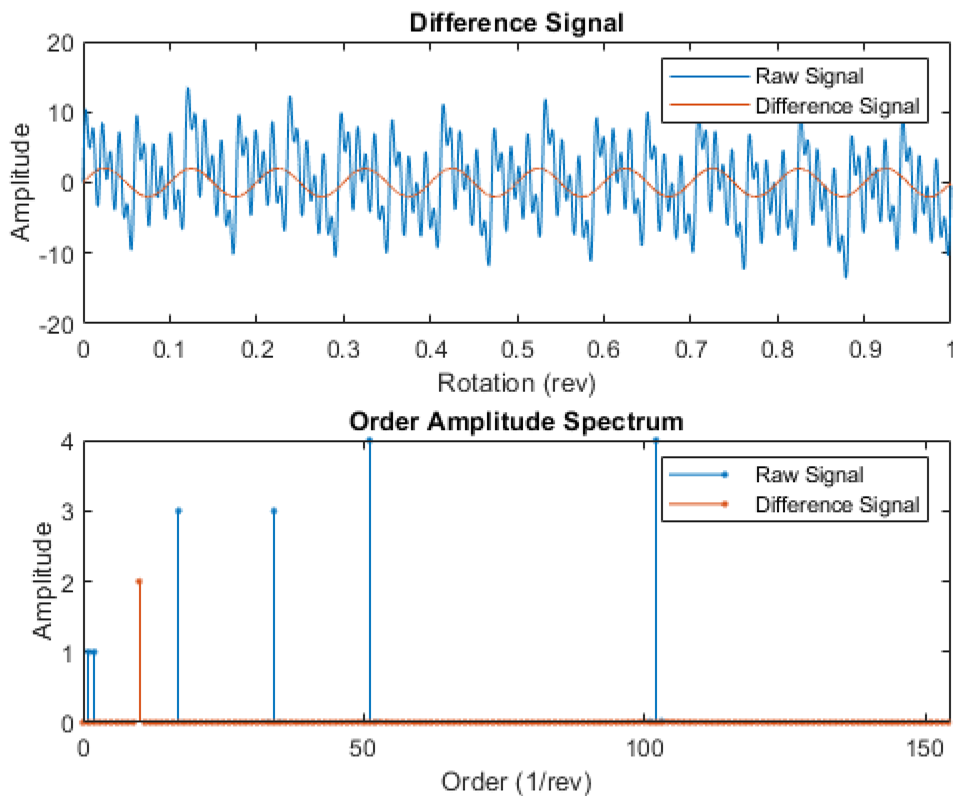
Compute the difference signal of the TSA signal using the sample time, rpm, and the mesh orders to be filtered out.

```
Y = tsadifference(X,t,rpm,orderList);
```

The output Y is a vector containing the gear mesh signal and harmonics for gears 5 and 6.

Visualize the difference signal, the raw TSA signal, and their amplitude spectrum on a plot.

```
tsadifference(X,fs,rpm,orderList)
```



From the amplitude spectrum plot, observe the following components:

- The filtered component at the 17th order and its harmonic at the 34th order
- The second filtered component at the 51st order and its harmonic at the 102nd order
- The residual mesh components for gears 5 and 6 at the 10th order
- The filtered shaft component at the 1st and 2nd orders
- The amplitudes on the spectrum plot match the amplitudes of individual signals

Compute Difference Signal and Amplitude Spectrum of a TSA Signal

In this example, `sineWavePhaseMod.mat` contains the data of a phase modulated sine wave. `XT` is a timetable with the sine wave data and `rpm` used is 60 RPM. The sine wave has a frequency of 32 Hz. To filter out the unmodulated sine wave and the sidebands of the phase modulating signal, use 32 as the `orderList`.

Load the data and the required variables.

```
load('sineWavePhaseMod.mat','XT','rpm','orders')
head(XT,4)
```

```
ans=4×1 timetable
          Time             Data
    ______________     _______
```

```
    0 sec                 0
    0.00097656 sec     0.2011
    0.0019531 sec     0.39399
    0.0029297 sec     0.57078
```

Note that the time values in XT are strictly increasing, equidistant, and finite.

Compute the difference signal and its amplitude spectrum. Set the value of `'Domain'` to `'frequency'` since the orders are in Hz.

```
[Y,S] = tsadifference(XT,rpm,orders,'Domain','frequency')
```

```
Y=1024×1 timetable
         Time             Data
    ______________    __________

    0 sec             2.2849e-15
    0.00097656 sec      0.046525
    0.0019531 sec       0.091185
    0.0029297 sec        0.13219
    0.0039062 sec         0.1679
    0.0048828 sec        0.19688
    0.0058594 sec        0.21799
    0.0068359 sec        0.23039
    0.0078125 sec         0.2336
    0.0087891 sec        0.22751
    0.0097656 sec        0.21239
    0.010742 sec         0.18888
    0.011719 sec         0.15793
    0.012695 sec         0.12081
    0.013672 sec        0.079041
    0.014648 sec        0.034303
          ⋮
```

```
S = 1024×1 complex

  -0.0000 + 0.0000i
   0.0000 + 0.0000i
   0.0000 + 0.0000i
   0.0000 + 0.0000i
   0.0000 + 0.0000i
  -0.0000 - 0.0000i
  -0.0000 + 0.0000i
   0.0000 + 0.0000i
  -0.0000 - 0.0000i
   0.0000 + 0.0000i
      ⋮
```

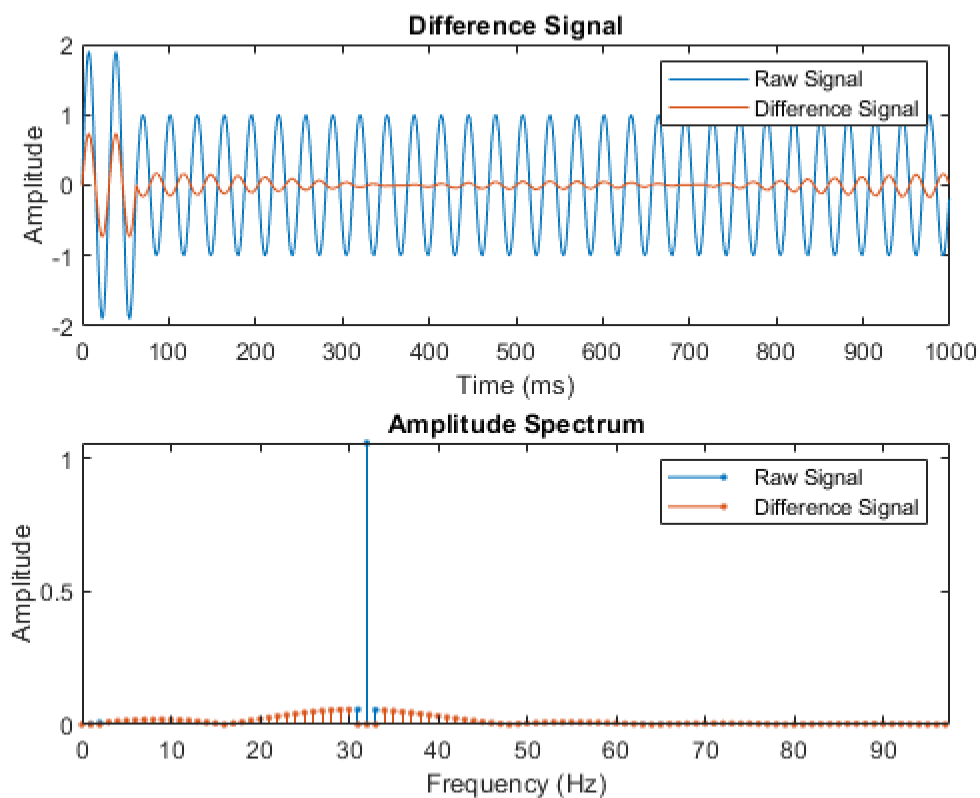
The output Y is a timetable that contains the difference signal, while S is a vector that contains the amplitude spectrum of the difference signal Y.

Visualize the Difference Signal and Amplitude Spectrum of a TSA Signal

In this example, `sineWaveRectangularPulse.mat` contains the data of a sine wave modulated by a rectangular pulse. `X` is a vector with the modulated sine wave data obtained at a shaft speed of 60 RPM. The unmodulated sine wave has a frequency of 32 Hz and amplitude of 1.0 units.

Load the data, and plot the difference signal of the modulated TSA signal `X`. To obtain the difference signal, filter out the unmodulated sine wave and the sidebands of the modulation signal by specifying the frequency of 32 Hz in `orderList`. Set the value of `'Domain'` to `'frequency'`.

```
load('sineWaveRectangularPulse.mat','X','t','rpm','orderList')
tsadifference(X,t,rpm,orderList,'Domain','frequency');
```



From the plot, observe the waveform and amplitude spectrum of the difference and raw signals, respectively. Observe that the difference signal contains everything except:

- Unmodulated sine wave at 32 Hz
- First-order sidebands of the unmodulated sine wave at 31 Hz and 33 Hz, respectively

Input Arguments

X — Time-synchronous averaged (TSA) signal

vector

Time-synchronous averaged (TSA) signal, specified as a vector. The time-synchronous averaged signal is computed from a long and relatively periodic raw signal through synchronization, resampling, and averaging. For more information on TSA signals, see `tsa`.

Time-synchronous averaging is a convenient method of background noise reduction in a spectrum of complex signals. It is effective in concentrating useful information that can be extracted from a time-domain signal for predictive maintenance. The synchronization typically requires a tachometer pulse signal in addition to the raw sensor data. The TSA signal depicts measurements at equally spaced angular positions over a single revolution of a shaft of interest.

XT — Time-synchronous averaged signal
timetable

Time synchronous averaged (TSA) signal, specified as a timetable. XT must contain a single numeric column variable corresponding to the TSA signal. Time values in XT must be strictly increasing, equidistant, and finite.

`fs` — Sampling frequency of the TSA signal
positive scalar

Sampling frequency of the TSA signal in Hertz, specified as a positive scalar.

`t` — Sample times of the TSA signal
positive scalar | vector of positive values

Sample times of the TSA signal, specified as a positive scalar or a vector of positive values.

If `t` is:

- A positive scalar, it contains the time interval or duration between samples. You must specify `t` as a `duration` variable.
- A vector of positive values, it contains sample times corresponding to elements in X. The time values must be strictly increasing, equidistant, and finite. You can specify `t` as a `double` or `duration` variable.

`rpm` — Rotational speed of the shaft
positive scalar

Rotational speed of the shaft, specified as a positive scalar. `tsadifference` uses a bandwidth equal to the shaft speed and the value of 'NumSidebands' around the frequencies of interest to compute Y from the TSA signal. Specify `rpm` in revolutions per minute. The signal components corresponding to this frequency, that is, `order = 1` are always filtered out.

`orderList` — Orders to be filtered out of the TSA signal
vector of positive integers

Orders to be filtered out of the TSA signal, specified as a vector of positive integers. Select the orders and harmonics to be filtered out of the TSA signal by observing them on the amplitude spectrum plot. For instance, specify `orderList` as the known mesh orders in a gear train to filter out the known components and their harmonics. For more information, see “Visualize the Difference Signal and Amplitude Spectrum of a TSA Signal” on page 1-266. Specify the units of `orderList` by selecting the appropriate value for 'Domain'.

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: `...,'NumSidebands',2`

`NumHarmonics` — Number of shaft and gear meshing frequency harmonics to be filtered

2 (default) | positive integer

Number of shaft and gear meshing frequency harmonics to be filtered, specified as the comma-separated pair consisting of '`NumHarmonics`' and a positive integer. Modify '`NumHarmonics`' if your TSA signal contains more than two known harmonics of components to be filtered.

`NumSidebands` — Number of sidebands to be filtered from the `orderList` frequencies and their harmonics

1 (default) | nonnegative integer

Number of sidebands to be filtered from the `orderList` frequencies and their harmonics, specified as the comma-separated pair consisting of '`NumSidebands`' and a nonnegative integer. The width of sidebands is determined using `2*(rpm/60)*(NumSidebands+0.5)`. Modify '`NumSidebands`' based on the number of sidebands to be filtered from `X` as observed in the amplitude spectrum plot.

`NumRotations` — Number of shaft rotations in the TSA signal

1 (default) | positive integer

Number of shaft rotations in the TSA signal, specified as the comma-separated pair consisting of '`NumRotations`' and a positive integer. Modify '`NumRotations`' if your input `X` or `XT` contains data for more than one rotation of the driver gear shaft. The function uses '`NumRotations`' to determine the number of rotations to be shown on the x-axis of the plot. The filtering results in `Y` are not affected by this value.

`Domain` — Units of the `orderList` values

`'order'` (default) | `'frequency'`

Units of the `orderList` values, specified as the comma-separated pair consisting of '`Domain`' and one of the following:

- `'frequency'`, if the orders in `orderList` are specified as frequencies in units of Hertz.
- `'order'`, if the orders in `orderList` are specified as number of rotations relative to the value of `rpm`. For example, if the rotational speed of the driven gear is defined as a factor of the driver gear rpm, specify '`Domain`' as `'order'`. Also, choose `'order'` if you are comparing data obtained from machines operating at different speeds.

Output Arguments

Y — Difference signal of the TSA signal

vector | timetable

Difference signal of the TSA signal, returned as:

- A vector, when the TSA signal is specified as a vector `X`.

- A timetable, when the TSA signal is specified as a timetable XT.

The difference signal is computed by removing the regular signal, the first-order sidebands, the value of 'NumSidebands', and their respective harmonics from X. You can use Y to further extract condition indicators of rotating machinery for predictive maintenance. For example, extracting the FM4 indicator from Y is useful to detect faults isolated to only a limited number of teeth in a gear mesh. For more information on how Y is computed, see “Algorithms” on page 1-270.

For more information on regular signal, see `tsaregular`.

S — Amplitude spectrum of the difference signal
vector

Amplitude spectrum of the difference signal, returned as a vector. S is the normalized fast Fourier transform of the signal Y. S is the same length as the input TSA signal X. For more information on how S is computed, see “Algorithms” on page 1-270.

Algorithms

Difference Signal

The difference signal is computed from the TSA signal by filtering the following from the signal spectrum:

- Shaft frequency and its harmonics
- Gear meshing frequencies and their harmonics
- First-order sidebands at the gear meshing frequencies and their harmonics
- Optionally, the sidebands specified in 'NumSidebands' at the gear meshing frequencies and their harmonics

`tsadifference` uses a bandwidth equal to three times the shaft speed and the value of 'NumSidebands', around the frequencies of interest, to compute Y from the TSA signal.

Amplitude Spectrum

The amplitude spectrum of the difference signal is computed as follows,

$$\mathrm{S} = \frac{\mathrm{fft}(Y)}{\mathrm{length}(Y) * 2}$$

Here, Y is the difference signal.

References

[1] McFadden, P.D. "Examination of a Technique for the Early Detection of Failure in Gears by Signal Processing of the Time Domain Average of the Meshing Vibration." *Aero Propulsion Technical Memorandum 434*. Melbourne, Australia: Aeronautical Research Laboratories, Apr. 1986.

[2] Večeř, P., Marcel Kreidl, and R. Šmíd. "Condition Indicators for Gearbox Monitoring Systems." *Acta Polytechnica* 45.6 (2005), pages 35-43.

[3] Zakrajsek, J. J., Townsend, D. P., and Decker, H. J. "An Analysis of Gear Fault Detection Methods as Applied to Pitting Fatigue Failure Data." *Technical Memorandum 105950*. NASA, Apr. 1993.

[4] Zakrajsek, James J. "An investigation of gear mesh failure prediction techniques." National Aeronautics and Space Administration Cleveland OH Lewis Research Center, 1989. No. NASA-E-5049.

See Also

`tsaresidual` | `tsaregular`

Introduced in R2018b

tsaregular

Regular signal of a time-synchronous averaged signal

Syntax

```
Y = tsaregular(X,fs,rpm,orderList)
Y = tsaregular(X,t,rpm,orderList)
Y = tsaregular(XT,rpm,orderList)
[Y,S] = tsaregular( ___ )
___ = tsaregular( ___ ,Name,Value)

tsaregular( ___ )
```

Description

`Y = tsaregular(X,fs,rpm,orderList)` computes the regular signal `Y` of the time-synchronous averaged (TSA) signal vector `X` using sampling rate `fs`, the rotational speed `rpm`, and the orders to be retained `orderList`. `Y` is computed by retaining the primary frequency, the components in `orderList`, and their respective harmonics from `X`. You can use `Y` to further extract condition indicators of rotating machinery for predictive maintenance. For example, extracting the FM0 indicator from `Y` is useful in identifying major changes such as gear tooth breakage or heavy wear in a gear box.

`Y = tsaregular(X,t,rpm,orderList)` computes the regular signal `Y` of the TSA signal vector `X` with corresponding time values from `t`.

`Y = tsaregular(XT,rpm,orderList)` computes the regular signal `Y` of the TSA signal stored in the timetable `XT`. `XT` must contain a single numeric column variable.

`[Y,S] = tsaregular( ___ )` returns the amplitude spectrum `S` of the regular signal `Y`. `S` is the amplitude spectrum computed using the normalized fast Fourier transform (FFT) of `Y`.

`___ = tsaregular( ___ ,Name,Value)` allows you to specify additional parameters using one or more name-value pair arguments. You can use this syntax with any of the previous input and output arguments.

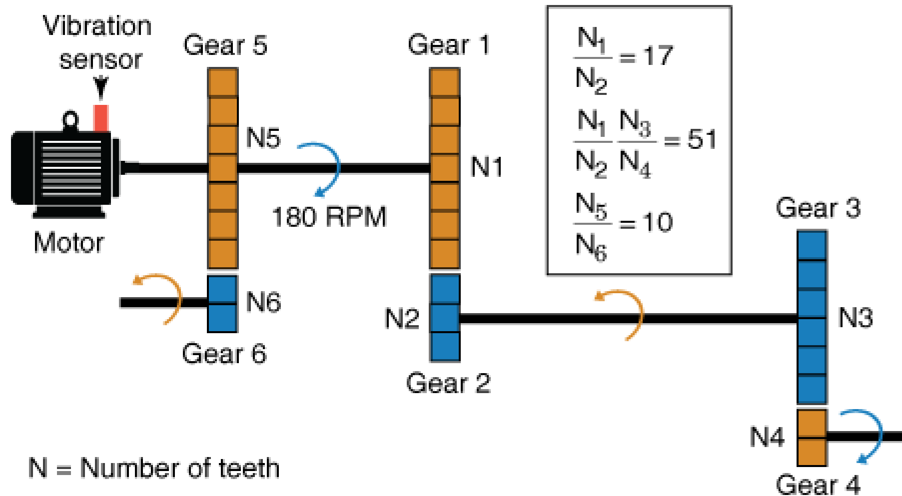
`tsaregular( ___ )` with no output arguments plots the time-domain and frequency-domain plots of the raw and regular TSA signals.

Examples

Find and Visualize the Regular Signal of a Compound TSA Signal

Consider a drivetrain with six gears driven by a motor that is fitted with a vibration sensor, as depicted in the figure below. Gear 1 on the motor shaft meshes with gear 2 with a gear ratio of 17:1. The final gear ratio, that is, the ratio between gears 1 and 2 and gears 3 and 4, is 51:1. Gear 5, also on the motor shaft, meshes with gear 6 with a gear ratio of 10:1. The motor is spinning at 180 RPM, and the sampling rate of the vibration sensor is 50 KHz. To retain the signal containing the meshing

components of the gears 1 and 2, gears 3 and 4 and, the shaft rotation, specify their gear ratios of 17 and 51 in `orderList`. The signal components corresponding to the shaft rotation (order = 1) is always implicitly included in the computation.




```
rpm = 180;
fs = 50e3;
t = (0:1/fs:(1/3)-1/fs)';                             % sample times
orderList = [17 51];
f = rpm/60*[1 orderList 10];
```

In practice, you would use measured data such as vibration signals obtained from an accelerometer. For this example, generate TSA signal X, which is the simulated data from the vibration sensor mounted on the motor.

```
X = sin(2*pi*f(1)*t) + sin(2*pi*2*f(1)*t) + ...      % motor shaft rotation and harmonic
    3*sin(2*pi*f(2)*t) + 3*sin(2*pi*2*f(2)*t) + ... % gear mesh vibration and harmonic for gears
    4*sin(2*pi*f(3)*t) + 4*sin(2*pi*2*f(3)*t) + ... % gear mesh vibration and harmonic for gears
    2*sin(2*pi*10*f(1)*t);                          % gear mesh vibration for gears 5 and 6
```

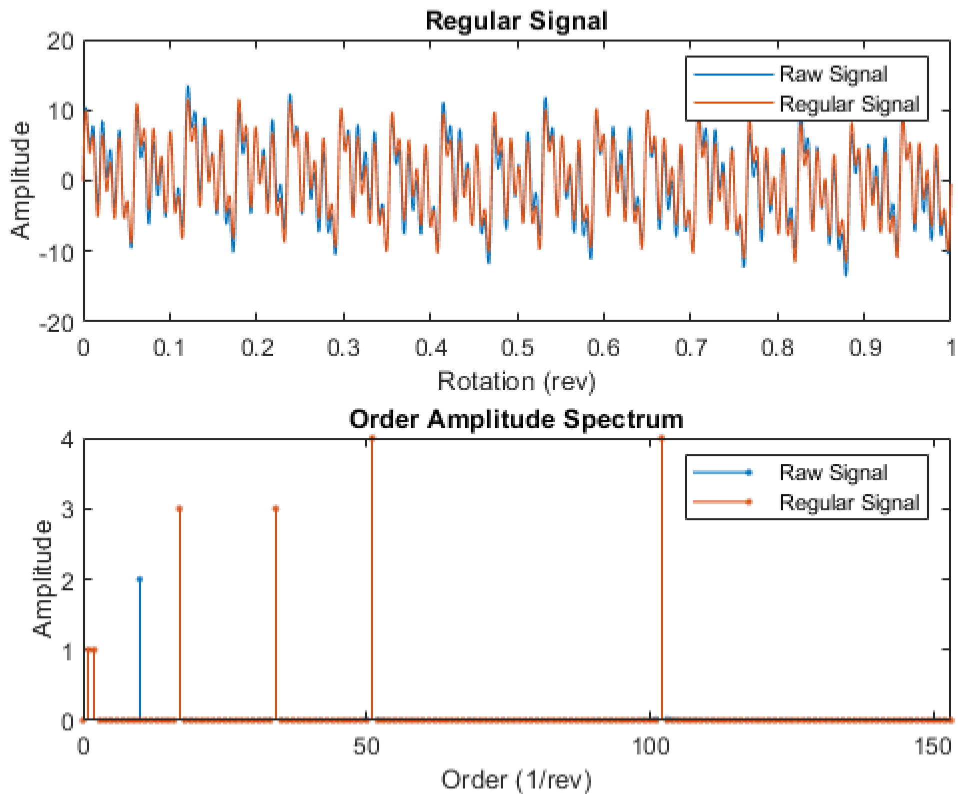
Compute the regular signal of the TSA signal using the sample time, rpm, and the mesh orders to be retained.

```
Y = tsaregular(X,t,rpm,orderList);
```

The output Y is a vector containing everything except the gear mesh signal and harmonics for gears 5 and 6.

Visualize the regular signal, the raw TSA signal, and their amplitude spectrum on a plot.

```
tsaregular(X,fs,rpm,orderList)
```



From the amplitude spectrum plot, observe the following components:

- The retained component at the 17th order and its harmonic at the 34th order
- The second retained component at the 51st order and its harmonic at the 102nd order
- The filtered mesh components for gears 5 and 6 at the 10th order
- The retained shaft component at the 1st and 2nd orders
- The amplitudes on the spectrum plot match the amplitudes of individual signals

Compute Regular Signal and Amplitude Spectrum of a TSA Signal

In this example, `sineWavePhaseMod.mat` contains the data of a phase modulated sine wave. `XT` is a timetable with the sine wave data and `rpm` used is 60 RPM. The sine wave has a frequency of 32 Hz and to recover the unmodulated sine wave, use 32 as the `orderList`.

Load the data and the required variables.

```
load('sineWavePhaseMod.mat','XT','rpm','orders')
head(XT,4)
```

```
ans=4×1 timetable
         Time            Data
    ______________    ________
```

```
    0 sec                    0
    0.00097656 sec     0.2011
    0.0019531 sec     0.39399
    0.0029297 sec     0.57078
```

Note that the time values in XT are strictly increasing, equidistant, and finite.

Compute the regular signal and its amplitude spectrum. Set the value of `'Domain'` to `'frequency'` since the orders are in Hz.

```
[Y,S] = tsaregular(XT,rpm,orders,'Domain','frequency')
```

```
Y=1024×1 timetable
         Time           Data
    ______________    __________

    0 sec             -2.552e-15
    0.00097656 sec       0.14928
    0.0019531 sec        0.29283
    0.0029297 sec        0.42512
    0.0039062 sec        0.54108
    0.0048828 sec        0.63624
    0.0058594 sec        0.70695
    0.0068359 sec        0.75049
    0.0078125 sec         0.7652
    0.0087891 sec        0.75049
    0.0097656 sec        0.70695
    0.010742 sec         0.63624
    0.011719 sec         0.54108
    0.012695 sec         0.42512
    0.013672 sec         0.29283
    0.014648 sec         0.14928
      ⋮

S = 1024×1 complex

   0.0000 + 0.0000i
   0.0000 - 0.0000i
   0.0000 + 0.0000i
   0.0000 + 0.0000i
  -0.0000 - 0.0000i
  -0.0000 - 0.0000i
  -0.0000 - 0.0000i
  -0.0000 + 0.0000i
   0.0000 + 0.0000i
  -0.0000 - 0.0000i
      ⋮
```

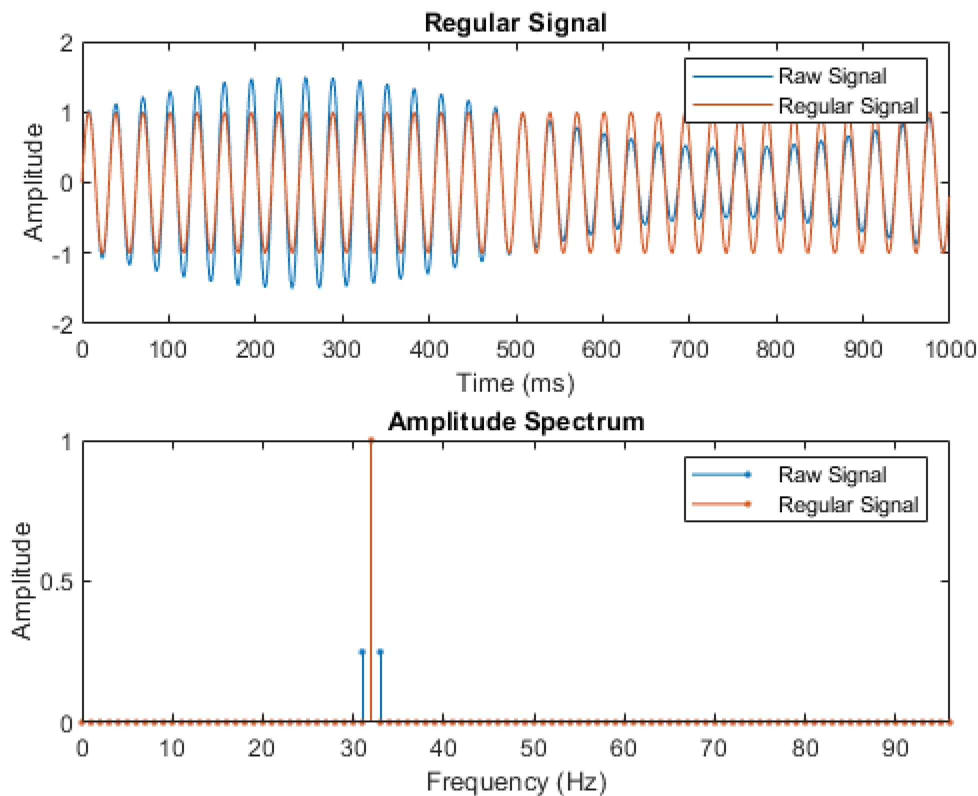
The output Y is a timetable that contains the regular signal, that is, the unmodulated sine wave, while S is a vector that contains the amplitude spectrum of the regular signal Y.

Visualize the Regular Signal and Amplitude Spectrum of a TSA Signal

In this example, `sineWaveAmpMod.mat` contains the data of an amplitude modulated sine wave. `X` is a vector with the amplitude modulated sine wave data obtained at a shaft speed of 60 RPM. The unmodulated sine wave has a frequency of 32 Hz and amplitude of 1.0 units.

Load the data, and plot the regular signal of the amplitude modulated TSA signal `X`. To retain the unmodulated signal, specify the frequency of 32 Hz in `orderList`. Set the value of `'Domain'` to `'frequency'`.

```
load('sineWaveAmpMod.mat','X','t','rpm','orderList')
tsaregular(X,t,rpm,orderList,'Domain','frequency');
```



From the plot, observe the waveform and amplitude spectrum of the regular and raw signals, respectively. Observe that the regular signal contains the unmodulated sine wave with an amplitude of 1.0 units and frequency of 32 Hz.

Input Arguments

X — Time-synchronous averaged (TSA) signal

vector

Time-synchronous averaged (TSA) signal, specified as a vector. The time-synchronous averaged signal is computed from a long and relatively periodic raw signal through synchronization, resampling, and averaging. For more information on TSA signals, see `tsa`.

Time-synchronous averaging is a convenient method of background noise reduction in a spectrum of complex signals. It is effective in concentrating useful information that can be extracted from a time-domain signal for predictive maintenance. The synchronization typically requires a tachometer pulse signal in addition to the raw sensor data. The TSA signal depicts measurements at equally spaced angular positions over a single revolution of a shaft of interest.

XT — Time-synchronous averaged signal

timetable

Time synchronous averaged (TSA) signal, specified as a timetable. `XT` must contain a single numeric column variable corresponding to the TSA signal. Time values in `XT` must be strictly increasing, equidistant, and finite.

`fs` — Sampling frequency of the TSA signal

positive scalar

Sampling frequency of the TSA signal in Hertz, specified as a positive scalar.

`t` — Sample times of the TSA signal

positive scalar | vector of positive values

Sample times of the TSA signal, specified as a positive scalar or a vector of positive values.

If `t` is:

- A positive scalar, it contains the time interval or duration between samples. You must specify `t` as a `duration` variable.
- A vector of positive values, it contains sample times corresponding to elements in `X`. The time values must be strictly increasing, equidistant, and finite. You can specify `t` as a `double` or `duration` variable.

`rpm` — Rotational speed of the shaft

positive scalar

Rotational speed of the shaft, specified as a positive scalar. `tsaregular` uses a bandwidth equal to the shaft speed and the value of '`NumSidebands`' around the frequencies of interest to compute `Y` from the TSA signal. Specify `rpm` in revolutions per minute. The signal components corresponding to this frequency, that is, `order = 1` are always retained.

`orderList` — Orders to be retained from the TSA signal

vector of positive integers

Orders to be retained from the TSA signal, specified as a vector of positive integers. Select the orders and harmonics to be retained from the TSA signal by observing them on the amplitude spectrum plot. For instance, specify `orderList` as the known mesh orders in a gear train to retain the desired components and their harmonics. For more information, see “Find and Visualize the Regular Signal of a Compound TSA Signal” on page 1-272. Specify the units of `orderList` by selecting the appropriate value for '`Domain`'.

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: ...,'NumSidebands',2

NumHarmonics — Number of shaft and gear meshing frequency harmonics to be filtered
2 (default) | positive integer

Number of shaft and gear meshing frequency harmonics to be filtered, specified as the comma-separated pair consisting of 'NumHarmonics' and a positive integer. Modify 'NumHarmonics' if your TSA signal contains more than two known harmonics of components to be filtered.

NumSidebands — Number of sidebands to be retained from the orderList frequencies and their harmonics
0 (default) | nonnegative integer

Number of sidebands to be retained from the orderList frequencies and their harmonics, specified as the comma-separated pair consisting of 'NumSidebands' and a nonnegative integer. The width of sidebands is determined using 2*(rpm/60)*(NumSidebands+0.5). Modify 'NumSidebands' based on the number of sidebands to be retained from X as observed in the amplitude spectrum plot.

NumRotations — Number of shaft rotations in the TSA signal
1 (default) | positive integer

Number of shaft rotations in the TSA signal, specified as the comma-separated pair consisting of 'NumRotations' and a positive integer. Modify 'NumRotations' if your input X or XT contains data for more than one rotation of the driver gear shaft. The function uses 'NumRotations' to determine the number of rotations to be shown on the x-axis of the plot. The filtering results in Y are not affected by this value.

Domain — Units of the orderList values
'order' (default) | 'frequency'

Units of the orderList values, specified as the comma-separated pair consisting of 'Domain' and one of the following:

- 'frequency', if the orders in orderList are specified as frequencies in units of Hertz.
- 'order', if the orders in orderList are specified as number of rotations relative to the value of rpm. For example, if the rotational speed of the driven gear is defined as a factor of the driver gear rpm, specify 'Domain' as 'order'. Also, choose 'order' if you are comparing data obtained from machines operating at different speeds.

Output Arguments

Y — Regular signal of the TSA signal
vector | timetable

Regular signal of the TSA signal, returned as:

- A vector, when the TSA signal is specified as a vector X.
- A timetable, when the TSA signal is specified as a timetable XT.

Y is computed by retaining the primary frequency, the components in orderList, the first-order sidebands in 'NumSidebands', and their respective harmonics from X. You can use Y to further extract condition indicators of rotating machinery for predictive maintenance. For example, extracting the FM0 indicator from Y is useful in identifying major changes such as gear tooth

breakage or heavy wear in a gear box. For more information on how Y is computed, see “Algorithms” on page 1-279.

S — Amplitude spectrum of the regular signal
vector

Amplitude spectrum of the regular signal, returned as a vector. S is the normalized fast Fourier transform of the signal Y. S is the same length as the input TSA signal X. For more information on how S is computed, see “Algorithms” on page 1-279.

Algorithms

Regular Signal

The regular signal is computed from the TSA signal by retaining the following from the signal spectrum:

- Shaft frequency and its harmonics
- Gear meshing frequencies and their harmonics
- Optionally, the sidebands specified in 'NumSidebands' at the gear meshing frequencies and their harmonics

`tsaregular` uses a bandwidth equal to the shaft speed times the value of 'NumSidebands', around the frequencies of interest, to compute Y from the TSA signal. The regular signal is related to the residual signal through the equation $Y_{regular} = X - Y_{residual}$. If the first-order sidebands are retained in the regular signal, then, $Y_{regular} = X - Y_{difference}$.

Amplitude Spectrum

The amplitude spectrum of the regular signal is computed as follows,

$$\mathrm{S} = \frac{\mathrm{fft}(Y)}{\mathrm{length}(Y) * 2}$$

Here, Y is the regular signal.

References

[1] McFadden, P.D. "Examination of a Technique for the Early Detection of Failure in Gears by Signal Processing of the Time Domain Average of the Meshing Vibration." *Aero Propulsion Technical Memorandum 434*. Melbourne, Australia: Aeronautical Research Laboratories, Apr. 1986.

[2] Večeř, P., Marcel Kreidl, and R. Šmíd. "Condition Indicators for Gearbox Monitoring Systems." *Acta Polytechnica* 45.6 (2005), pages 35-43.

[3] Zakrajsek, J. J., Townsend, D. P., and Decker, H. J. "An Analysis of Gear Fault Detection Methods as Applied to Pitting Fatigue Failure Data." *Technical Memorandum 105950*. NASA, Apr. 1993.

[4] Zakrajsek, James J. "An investigation of gear mesh failure prediction techniques." National Aeronautics and Space Administration Cleveland OH Lewis Research Center, 1989. No. NASA-E-5049.

See Also

`tsaresidual` | `tsadifference`

Introduced in R2018b

tsaresidual

Residual signal of a time-synchronous averaged signal

Syntax

```
Y = tsaresidual(X,fs,rpm,orderList)
Y = tsaresidual(X,t,rpm,orderList)
Y = tsaresidual(XT,rpm,orderList)
[Y,S] = tsaresidual( ___ )
___ = tsaresidual( ___ ,Name,Value)

tsaresidual( ___ )
```

Description

`Y = tsaresidual(X,fs,rpm,orderList)` computes the residual signal `Y` of the time-synchronous averaged (TSA) signal vector `X` using sampling rate `fs`, the rotational speed `rpm`, and the orders to be filtered `orderList`. The residual signal is computed by removing the components in `orderList` and their harmonics from `X`. You can use `Y` to further extract condition indicators of rotating machinery for predictive maintenance. For example, extracting the root-mean-squared value of the residual signal is useful in identifying changes over time which indicate potential machine faults.

`Y = tsaresidual(X,t,rpm,orderList)` computes the residual signal `Y` of the TSA signal vector `X` with corresponding time values in vector `t`.

`Y = tsaresidual(XT,rpm,orderList)` computes the residual signal `Y` of the TSA signal stored in the timetable `XT`. `XT` must contain a single numeric column variable.

`[Y,S] = tsaresidual( ___ )` returns the amplitude spectrum `S` of the residual signal `Y`. `S` is the amplitude spectrum computed using the normalized fast Fourier transform (FFT) of `Y`.

`___ = tsaresidual( ___ ,Name,Value)` allows you to specify additional parameters using one or more name-value pair arguments. You can use this syntax with any of the previous input and output arguments.

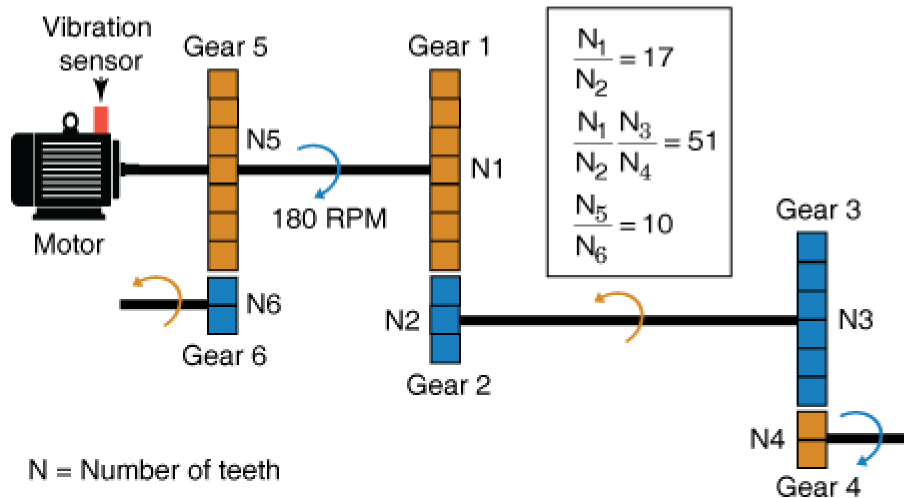
`tsaresidual( ___ )` with no output arguments plots the time-domain and frequency-domain plots of the raw and residual TSA signals.

Examples

Find and Visualize the Residual Signal of a Compound TSA Signal

Consider a drivetrain with six gears driven by a motor that is fitted with a vibration sensor, as depicted in the figure below. Gear 1 on the motor shaft meshes with gear 2 with a gear ratio of 17:1. The final gear ratio, that is, the ratio between gears 1 and 2 and gears 3 and 4, is 51:1. Gear 5, also on the motor shaft, meshes with gear 6 with a gear ratio of 10:1. The motor is spinning at 180 RPM, and the sampling rate of the vibration sensor is 50 KHz. To obtain the signal containing just the meshing components for gears 5 and 6, filter out the signal components due to the gears 1 and 2 and,

3 and 4 by specifying their gear ratios of 17 and 51 in `orderList`. The signal components corresponding to the shaft rotation (order = 1) is always implicitly included in the computation.




```
rpm = 180;
fs = 50e3;
t = (0:1/fs:(1/3)-1/fs)';                           % sample times
orderList = [17 51];
f = rpm/60*[1 orderList 10];
```

In practice, you would use measured data such as vibration signals obtained from an accelerometer. For this example, generate TSA signal X, which is the simulated data from the vibration sensor mounted on the motor.

```
X = sin(2*pi*f(1)*t) + sin(2*pi*2*f(1)*t) + ...     % motor shaft rotation and harmonic
    3*sin(2*pi*f(2)*t) + 3*sin(2*pi*2*f(2)*t) + ... % gear mesh vibration and harmonic for gears
    4*sin(2*pi*f(3)*t) + 4*sin(2*pi*2*f(3)*t) + ... % gear mesh vibration and harmonic for gears
    2*sin(2*pi*10*f(1)*t);                          % gear mesh vibration for gears 5 and 6
```

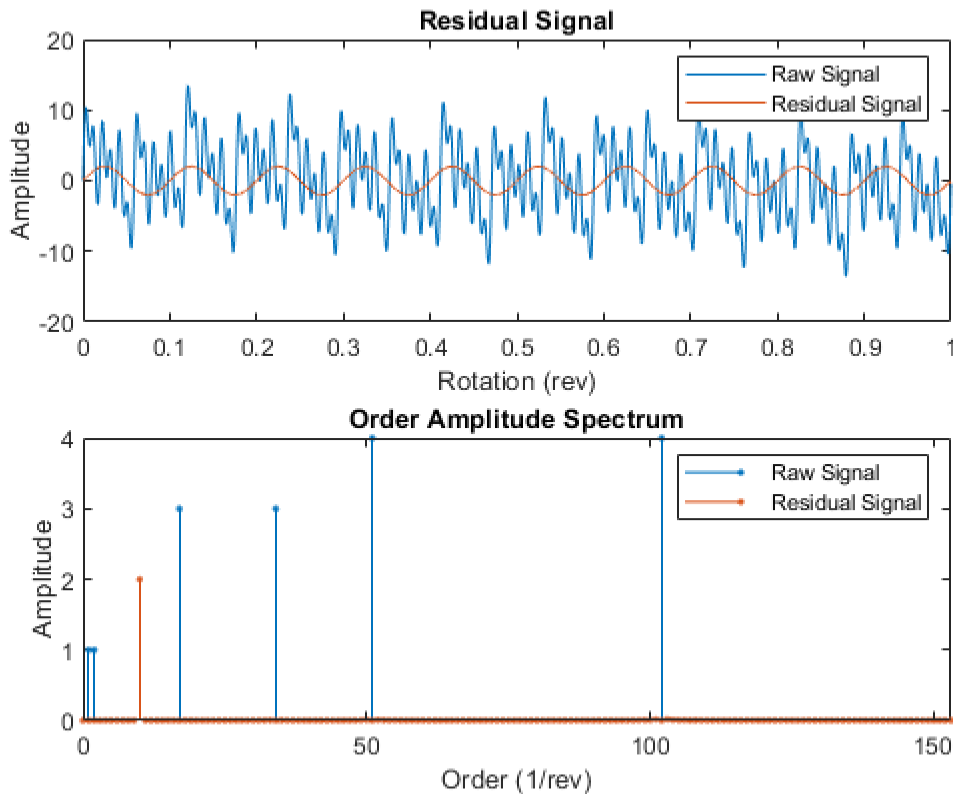
Compute the residual of the TSA signal using the sample time, rpm, and the mesh orders to be filtered out.

```
Y = tsaresidual(X,t,rpm,orderList);
```

The output Y is a vector containing the gear mesh signal and harmonics for gears 5 and 6.

Visualize the residual signal, the raw TSA signal, and their amplitude spectrum on a plot.

```
tsaresidual(X,fs,rpm,orderList)
```



From the amplitude spectrum plot, observe the following components:

- The filtered component at the 17th order and its harmonic at the 34th order
- The second filtered component at the 51st order and its harmonic at the 102nd order
- The residual mesh components for gears 5 and 6 at the 10th order
- The filtered shaft component at the 1st and 2nd orders
- The amplitudes on the spectrum plot match the amplitudes of individual signals

Compute Residual Signal and Amplitude Spectrum of a TSA Signal

In this example, `sineWavePhaseMod.mat` contains the data of a phase modulated sine wave. `XT` is a timetable with the sine wave data and `rpm` used is 60 RPM. The sine wave has a frequency of 32 Hz, and to filter out the unmodulated sine wave, use 32 as the `orderList`.

Load the data and the required variables.

```
load('sineWavePhaseMod.mat','XT','rpm','orders')
head(XT,4)
```

```
ans=4×1 timetable
         Time            Data
    ______________    ________
```

```
    0 sec                   0
    0.00097656 sec     0.2011
    0.0019531 sec     0.39399
    0.0029297 sec     0.57078
```

Note that the time values in XT are strictly increasing, equidistant, and finite.

Compute the residual signal and its amplitude spectrum. Set the value of `'Domain'` to `'frequency'` since the orders are in Hz.

```
[Y,S] = tsaresidual(XT,rpm,orders,'Domain','frequency')
```

```
Y=1024×1 timetable
         Time             Data
    ______________    _________

    0 sec             2.552e-15
    0.00097656 sec     0.051822
    0.0019531 sec       0.10116
    0.0029297 sec       0.14566
    0.0039062 sec       0.18317
    0.0048828 sec       0.21188
    0.0058594 sec       0.23039
    0.0068359 sec       0.23776
    0.0078125 sec        0.2336
    0.0087891 sec       0.21803
    0.0097656 sec       0.19174
    0.010742 sec         0.1559
    0.011719 sec        0.11215
    0.012695 sec       0.062503
    0.013672 sec      0.0092782
    0.014648 sec      -0.045032
      ⋮
```

```
S = 1024×1 complex

  -0.0000 + 0.0000i
   0.0000 + 0.0000i
   0.0000 + 0.0000i
   0.0000 + 0.0000i
   0.0000 + 0.0000i
  -0.0000 - 0.0000i
  -0.0000 + 0.0000i
   0.0000 + 0.0000i
  -0.0000 - 0.0000i
   0.0000 + 0.0000i
      ⋮
```

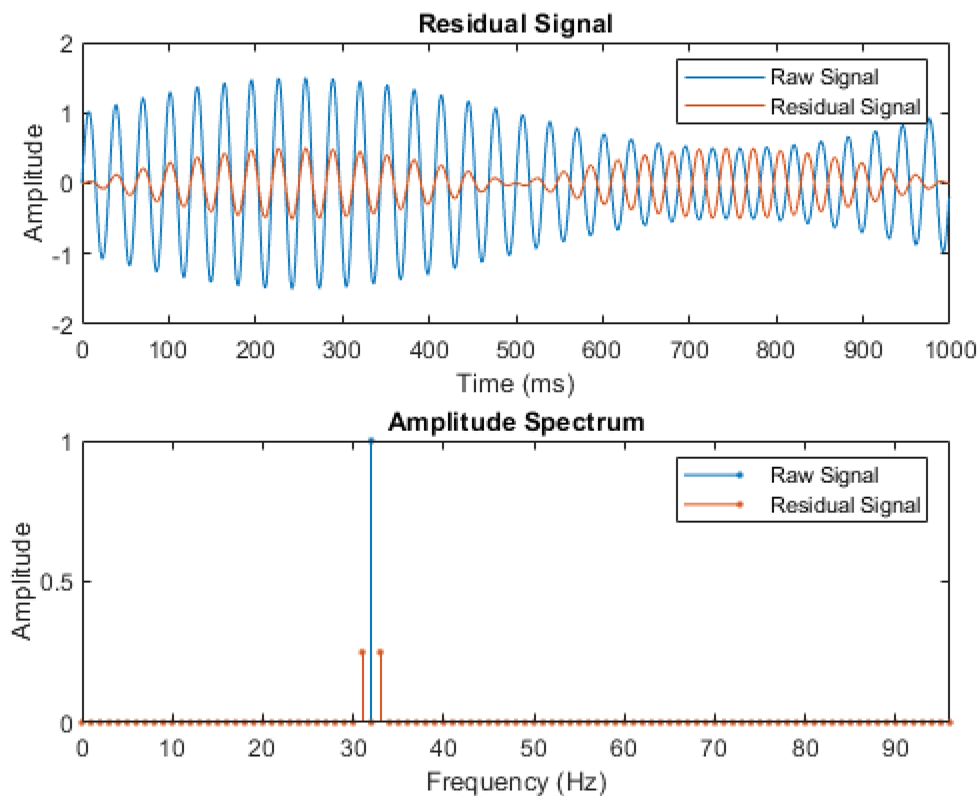
The output Y is a timetable that contains the residual signal, that is, the phase modulation signal, while S is a vector that contains the amplitude spectrum of the residual signal Y.

Visualize the Residual and Amplitude Spectrum of a TSA Signal

In this example, `sineWaveAmpMod.mat` contains the data of an amplitude modulated sine wave. `X` is a vector with the amplitude modulated sine wave data obtained at a shaft speed of 60 RPM. The unmodulated sine wave has a frequency of 32 Hz and amplitude of 1.0 units.

Load the data, and plot the residual signal of the amplitude modulated TSA signal `X`. To obtain the residual signal, filter out the unmodulated sine wave by specifying the frequency of 32 Hz in `orderList`. Set the value of `'Domain'` to `'frequency'`.

```
load('sineWaveAmpMod.mat','X','t','rpm','orderList')
tsaresidual(X,t,rpm,orderList,'Domain','frequency');
```



From the plot, observe the waveform and amplitude spectrum of the residual and raw signals, respectively.

Input Arguments

X — Time-synchronous averaged (TSA) signal
vector

Time-synchronous averaged (TSA) signal, specified as a vector. The time-synchronous averaged signal is computed from a long and relatively periodic raw signal through synchronization, resampling, and averaging. For more information on TSA signals, see `tsa`.

Time-synchronous averaging is a convenient method of background noise reduction in a spectrum of complex signals. It is effective in concentrating useful information that can be extracted from a time-domain signal for predictive maintenance. The synchronization typically requires a tachometer pulse signal in addition to the raw sensor data. The TSA signal depicts measurements at equally spaced angular positions over a single revolution of a shaft of interest.

XT — Time-synchronous averaged signal
timetable

Time synchronous averaged (TSA) signal, specified as a timetable. `XT` must contain a single numeric column variable corresponding to the TSA signal. Time values in `XT` must be strictly increasing, equidistant, and finite.

fs — Sampling frequency of the TSA signal
positive scalar

Sampling frequency of the TSA signal in Hertz, specified as a positive scalar.

t — Sample times of the TSA signal
positive scalar | vector of positive values

Sample times of the TSA signal, specified as a positive scalar or a vector of positive values.

If `t` is:

- A positive scalar, it contains the time interval or duration between samples. You must specify `t` as a `duration` variable.
- A vector of positive values, it contains sample times corresponding to elements in `X`. The time values must be strictly increasing, equidistant, and finite. You can specify `t` as a `double` or `duration` variable.

rpm — Rotational speed of the shaft
positive scalar

Rotational speed of the shaft, specified as a positive scalar. `tsaresidual` uses a bandwidth equal to the shaft speed around the frequencies of interest to filter out the undesired frequency components from the TSA signal. The signal components corresponding to this frequency, that is, `order = 1` are always filtered out.

Specify `rpm` in revolutions per minute.

orderList — Orders to be filtered out of the TSA signal
vector of positive integers

Orders to be filtered out of the TSA signal, specified as a vector of positive integers. Select the orders and harmonics to be filtered out of the TSA signal by observing them on the amplitude spectrum plot. For instance, specify `orderList` as the known mesh orders in a gear train to filter out the known components and their harmonics. For more information, see "Find and Visualize the Residual Signal of a Compound TSA Signal" on page 1-281. Specify the units of `orderList` by selecting the appropriate value for '`Domain`'.

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: `...,'NumRotations',5`

`NumHarmonics` — Number of shaft and gear meshing frequency harmonics to be filtered

2 (default) | positive integer

Number of shaft and gear meshing frequency harmonics to be filtered, specified as the comma-separated pair consisting of '`NumHarmonics`' and a positive integer. Modify '`NumHarmonics`' if your TSA signal contains more than two known harmonics of components to be filtered.

`NumRotations` — Number of shaft rotations in the TSA signal

1 (default) | positive integer

Number of shaft rotations in the TSA signal, specified as the comma-separated pair consisting of '`NumRotations`' and a positive integer. Modify '`NumRotations`' if your input `X` or `XT` contains data for more than one rotation of the driver gear shaft. The function uses '`NumRotations`' to determine the number of rotations to be shown on the x-axis of the plot. The filtering results in `Y` are not affected by this value.

`Domain` — Units of the `orderList` values

`'order'` (default) | `'frequency'`

Units of the `orderList` values, specified as the comma-separated pair consisting of '`Domain`' and one of the following:

- `'frequency'`, if the orders in `orderList` are specified as frequencies in units of Hertz.
- `'order'`, if the orders in `orderList` are specified as number of rotations relative to the value of `rpm`. For example, if the rotational speed of the driven gear is defined as a factor of the driver gear rpm, specify '`Domain`' as `'order'`. Also, choose `'order'` if you are comparing data obtained from machines operating at different speeds.

Output Arguments

Y — Residual signal of the TSA signal

vector | timetable

Residual signal of the TSA signal, returned as:

- A vector, when the TSA signal is specified as a vector `X`
- A timetable, when the TSA signal is specified as a timetable `XT`

The residual signal is computed by removing the components in `orderList` and the shaft signal along with their respective harmonics from `X`. You can use `Y` to further extract condition indicators of rotating machinery for predictive maintenance. For example, extracting the root-mean-squared value of the residual signal is useful in identifying changes over time, which indicate potential machine faults. For more information on how `Y` is computed, see “Algorithms” on page 1-288.

S — Amplitude spectrum of the residual signal

vector

Amplitude spectrum of the residual signal, returned as a vector. S is the normalized fast Fourier transform of the signal Y. S has the same length as the input TSA signal X. For more information on how S is computed, see “Algorithms” on page 1-288.

Algorithms

Residual Signal

The residual signal is computed from the TSA signal by removing the following from the signal spectrum:

- Shaft frequency and its harmonics
- Gear meshing frequencies and their harmonics

The frequencies are removed by computing the discrete Fourier transform (DFT) and setting the spectrum values to zero at the specified frequencies. `tsaresidual` uses a bandwidth equal to the shaft speed around the frequencies of interest to filter out the undesired frequency components, as mentioned in [4].

Amplitude Spectrum

The amplitude spectrum of the residual signal is computed as follows,

$$\mathrm{S} = \frac{\mathrm{fft}(Y)}{\mathrm{length}(Y) * 2}$$

Here, Y is the residual signal.

References

[1] McFadden, P.D. "Examination of a Technique for the Early Detection of Failure in Gears by Signal Processing of the Time Domain Average of the Meshing Vibration." *Aero Propulsion Technical Memorandum 434*. Melbourne, Australia: Aeronautical Research Laboratories, Apr. 1986.

[2] Večeř, P., Marcel Kreidl, and R. Šmíd. "Condition Indicators for Gearbox Monitoring Systems." *Acta Polytechnica* 45.6 (2005), pages 35-43.

[3] Zakrajsek, J. J., Townsend, D. P., and Decker, H. J. "An Analysis of Gear Fault Detection Methods as Applied to Pitting Fatigue Failure Data." *Technical Memorandum 105950*. NASA, Apr. 1993.

[4] Zakrajsek, James J. "An investigation of gear mesh failure prediction techniques." National Aeronautics and Space Administration Cleveland OH Lewis Research Center, 1989. No. NASA-E-5049.

See Also

`tsadifference` | `tsaregular`

Introduced in R2018b

update

Update posterior parameter distribution of degradation remaining useful life model

Syntax

```
update(mdl,data)
```

Description

`update(mdl,data)` updates the posterior estimate of the parameters of the degradation remaining useful life (RUL) model `mdl` using the latest degradation measurements in `data`.

Examples

Update Exponential Degradation Model in Real Time

Load training data, which is a degradation feature profile for a component.

```
load('expRealTime.mat')
```

For this example, assume that the training data is not historical data. When there is no historical data, you can update your degradation model in real time using observed data.

Create an exponential degradation model with the following settings:

- Arbitrary θ and β prior distributions with large variances so that the model relies mostly on observed data
- Noise variance of `0.003`

```
mdl = exponentialDegradationModel('Theta',1,'ThetaVariance',1e6,...
                                  'Beta',1,'BetaVariance',1e6,...
                                  'NoiseVariance',0.003);
```

Since there is no life time variable in the training data, create an arbitrary life time vector for fitting.

```
lifeTime = [1:length(expRealTime)];
```

Observe the degradation feature for 10 iterations. Update the degradation model after each iteration.

```
for i=1:10
    update(mdl,[lifeTime(i) expRealTime(i)])
end
```

After observing the model for some time, for example at a steady-state operating point, you can restart the model and save the current posterior distribution as a prior distribution.

```
restart(mdl,true)
```

View the updated prior distribution parameters.

```
mdl.Prior
```

```
ans = struct with fields:
            Theta: 2.3555
    ThetaVariance: 0.0058
             Beta: 0.0722
     BetaVariance: 3.6362e-05
              Rho: -0.8429
```

Input Arguments

mdl — Degradation RUL model
linearDegradationModel object | exponentialDegradationModel object

Degradation RUL model, specified as a linearDegradationModel object or an exponentialDegradationModel object. update updates the posterior estimates of the degradation model parameters based on the latest degradation feature measurements in data.

For a linearDegradationModel, the updated parameters are Theta and ThetaVariance.

For an exponentialDegradationModel, the updated parameters are Theta, ThetaVariance, Beta, BetaVariance, and Rho.

update also sets the following properties of mdl:

- InitialLifeTimeValue — The first time you call update, this property is set to the life time value in the first row of data.
- CurrentLifeTimeValue — Each time you call update, this property is set to the life time value in the last row of data.
- CurrentMeasurement — Each time you call update, this property is set to the feature measurement value in the last row of data.

data — Degradation feature measurements
two-column array | table object

Degradation feature measurements, specified as one of the following:

- Two-column array — The first column contains life time values and the second column contains the corresponding degradation feature measurement.
- table or timetable object that contains variables with names that match the LifeTimeVariable and DataVariables properties of mdl.

Extended Capabilities

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

Usage notes and limitations:

- This command supports code generation with MATLAB Coder. Before generating code that uses an RUL model, you must save the model using saveRULModelForCoder. When updating the model at run time, it is also useful to store the model state using readState. For an example, see “Generate Code that Preserves RUL Model State for System Restart”.

See Also

Functions
`linearDegradationModel` | `exponentialDegradationModel` | `predictRUL` | `fit`

Topics
"Update RUL Prediction as Data Arrives"
"RUL Estimation Using RUL Estimator Models"

Introduced in R2018a

writeMember

Write data to a specific workspace ensemble member

Syntax

```
writeMember(wensemble,data)
writeMember(wensemble,data,index)
```

Description

`writeMember` is a function used in code generated by **Diagnostic Feature Designer**.

`writeMember(wensemble,data)` creates a new member in workspace ensemble `wensemble`, and appends `data` to the data set that `wensemble` references.

`writeMember(wensemble,data,index)` writes `data` to the ensemble member that `index` identifies. `writeMember` overwrites the data in existing variables and creates additional columns for new variables.

Code that is generated by **Diagnostic Feature Designer** uses `writeMember`, `readMember`, and `findIndex` under the following conditions:

- The input data is an ensemble datastore, such as a file or simulation ensemble datastore.
- The computation option during code generation specified storing results in local memory rather than writing results back to the ensemble datastore.

Explicitly specifying a member index when reading and writing within the local version of the data, which the code manages using a `workspaceEnsemble` object, ensures member synchronization with the original ensemble datastore. This synchronization is necessary when you have sequential member-processing loops, such as when you compute ensemble statistics as a precursor to computing signal residues.

- During the first member-processing loop, which starts with an empty ensemble, no indexing is needed. The code appends each new member result to the end of the ensemble.
- During the second loop, the index enables the code to write updated member results to the correct location within the now-populated ensemble.

For more information about the dual processing loop for ensemble statistics, see “Anatomy of App-Generated MATLAB Code”.

Input Arguments

`wensemble` — Ensemble object
`workspaceEnsemble` object

Ensemble object, specified as a `workspaceEnsemble` object. `wensemble` contains ensemble data and specifies the variable names and types within the ensemble, such as data variables and condition variables.

`data` — Member data
single-row table

Member data, specified as a single-row table.

`index` — Member index
positive integer

Member index, specified as a positive integer. `index` identifies the ensemble member to write new data to. If you omit `index`, `writeMember` appends `data` as a new ensemble member in `wensemble`.

See Also

Diagnostic Feature Designer | `fileEnsembleDatastore` | `simulationEnsembleDatastore` | `workspaceEnsemble` | `findIndex` | `readMember`

Topics

"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a

writeToLastMemberRead

Write data to member of an ensemble datastore

Syntax

```
writeToLastMemberRead(ensemble,Name,Value)
writeToLastMemberRead(ensemble,data)
```

Description

`writeToLastMemberRead(ensemble,Name,Value)` writes the data specified one or more `Name,Value` pair arguments to the last-read member of an ensemble datastore. The last-read member is the member most recently accessed using the `read` command. (See “Data Ensembles for Condition Monitoring and Predictive Maintenance”.) Each `Name` must match an entry in the property `ensemble.DataVariables`. The function writes the corresponding `Value` to the ensemble datastore.

- If `ensemble` is a `simulationEnsembleDatastore` object, then `writeToLastMemberRead` writes the data to the MAT-file corresponding to the last-read ensemble member (`ensemble.LastMemberRead`).
- If `ensemble` is a `fileEnsembleDatastore` object, then `writeToLastMemberRead` uses the function identified by the property `ensemble.WriteToMemberFcn` to write the data. If that property is not defined, then `writeToLastMemberRead` generates an error.

This syntax is not available when the `ReadSize` property of `ensemble` is greater than 1. Use `writeToLastMemberRead(ensemble,data)` instead.

`writeToLastMemberRead(ensemble,data)` writes the data in a table to the last-read ensemble member. The table variables must match entries in the property `ensemble.DataVariables`.

Examples

Append Derived Data to Ensemble Members

You can process data in an ensemble datastore and add derived variables to the ensemble members. For this example, process a variable value to compute a label that indicates whether the ensemble member contains data obtained with a fault present. You then add that label to the ensemble.

For this example, use the following code to create a `simulationEnsembleDatastore` object using data previously generated by running a Simulink® model at a various fault values. (See `generateSimulationEnsemble`.) The ensemble includes simulation data for five different values of a model parameter, `ToothFaultGain`. The model was configured to log the simulation data to a variable named `logsout` in the MAT-files that are stored for this example in `simEnsData.zip`. Because of the volume of data, the `unzip` operation might take a minute or two.

```
unzip simEnsData.zip  % extract compressed files
ensemble = simulationEnsembleDatastore(pwd,'logsout')
```

```
ensemble = 
  simulationEnsembleDatastore with properties:
```

```
          DataVariables: [5x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: [0x0 string]
       SelectedVariables: [5x1 string]
                ReadSize: 1
              NumMembers: 5
          LastMemberRead: [0x0 string]
                   Files: [5x1 string]
```

Read the data from the first member in the ensemble. The software determines which ensemble is the first member, and updates the property `ensemble.LastMemberRead` to reflect the name of the corresponding file.

```
data = read(ensemble)
```

```
data=1×5 table
    PMSignalLogName          SimulationInput                      SimulationMetadata
    _______________    ______________________________    __________________________________    ___

      {'logsout'}      {1x1 Simulink.SimulationInput}    {1x1 Simulink.SimulationMetadata}    {2(
```

By default, all the variables stored in the ensemble data are designated as `SelectedVariables`. Therefore, the returned table row includes all ensemble variables, including a variable `SimulationInput`, which contains the `Simulink.SimulationInput` object that configured the simulation for this ensemble member. That object includes the `ToothFaultGain` value used for the ensemble member, stored in a data structure in its `Variables` property. Examine that value. (For more information about how the simulation configuration is stored, see `Simulink.SimulationInput` (Simulink).)

```
data.SimulationInput{1}
```

```
ans =
  SimulationInput with properties:

          ModelName: 'TransmissionCasingSimplified'
       InitialState: [0x0 Simulink.op.ModelOperatingPoint]
      ExternalInput: []
    ModelParameters: [0x0 Simulink.Simulation.ModelParameter]
    BlockParameters: [0x0 Simulink.Simulation.BlockParameter]
          Variables: [1x1 Simulink.Simulation.Variable]
          PreSimFcn: []
         PostSimFcn: []
         UserString: ''
```

```
Inputvars = data.SimulationInput{1}.Variables;
Inputvars.Name
```

```
ans =
'ToothFaultGain'
```

```
Inputvars.Value
```

```
ans = -2
```

Suppose that you want to convert the `ToothFaultGain` values for each ensemble member into a binary indicator of whether or not a tooth fault is present. Suppose further that you know from your experience with the system that tooth-fault gain values less than 0.1 in magnitude are small enough to be considered healthy operation. Convert the gain value for this ensemble into an indicator that is 0 (no fault) for –0.1 < gain < 0.1, and 1 (fault) otherwise.

```
sT = abs(Inputvars.Value) < 0.1;
```

To append the new tooth-fault indicator to the corresponding ensemble data, first expand the list of data variables in the ensemble to include a variable for the indicator.

```
ensemble.DataVariables = [ensemble.DataVariables; "ToothFault"];
ensemble.DataVariables
```

```
ans = 6x1 string
    "PMSignalLogName"
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
    "ToothFault"
```

This operation is conceptually equivalent to adding a column to the table of ensemble data. Now that `DataVariables` contains the new variable name, assign the derived value to that column of the member using `writeToLastMemberRead`.

```
writeToLastMemberRead(ensemble,'ToothFault',sT);
```

In practice, you want to append the tooth-fault indicator to every member in the ensemble. To do so, reset the ensemble datastore to its unread state, so that the next read operation starts at the first ensemble member. Then, loop through all the ensemble members, computing `ToothFault` for each member and appending it. The `reset` operation does not change `ensemble.DataVariables`, so `"ToothFault"` is still present in that list.

```
reset(ensemble);

sT = false;
while hasdata(ensemble)
    data = read(ensemble);
    InputVars = data.SimulationInput{1}.Variables;
    TFGain = InputVars.Value;
    sT = abs(TFGain) < 0.1;
    writeToLastMemberRead(ensemble,'ToothFault',sT);
end
```

Finally, designate the new tooth-fault indicator as a condition variable in the ensemble datastore. You can use this designation to track and refer to variables in the ensemble data that represent conditions under which the member data was generated.

```
ensemble.ConditionVariables = {"ToothFault"};
ensemble.ConditionVariables
```

```
ans = 
"ToothFault"
```

You can add the new variable to ensemble.SelectedVariables when you want to read it out for further analysis. For an example that shows more ways to manipulate and analyze data stored in a simulationEnsembleDatastore object, see “Using Simulink to Generate Fault Data”.

Read from and Write to a File Ensemble Datastore

Create a file ensemble datastore for data stored in MATLAB files, and configure it with functions that tell the software how to read from and write to the datastore. (For more details about configuring file ensemble datastores, see “File Ensemble Datastore With Measured Data”.)

```
% Create ensemble datastore that points to datafiles in current folder
unzip fileEnsData.zip  % extract compressed files
location = pwd;
extension = '.mat';
fensemble = fileEnsembleDatastore(location,extension);

% Specify data and condition variables
fensemble.DataVariables = ["gs";"sr";"load";"rate"];
fensemble.ConditionVariables = "label";

% Configure with functions for reading and writing variable data
addpath(fullfile(matlabroot,'examples','predmaint','main')) % Make sure functions are on path
fensemble.ReadFcn = @readBearingData;
fensemble.WriteToMemberFcn = @writeBearingData;
```

The functions tell the read and writeToLastMemberRead commands how to interact with the data files that make up the ensemble. Thus, when you call the read command, it uses readBearingData to read all the variables in fensemble.SelectedVariables. For this example, readBearingData extracts requested variables from a structure, bearing, and other variables stored in the file. It also parses the filename for the fault status of the data.

Specify variables to read, and read them from the first member of the ensemble.

```
fensemble.SelectedVariables = ["gs";"load";"label"];
data = read(fensemble)
```

```
data=1×3 table
     label            gs            load
    ________    _______________    ____

    "Faulty"    {5000x1 double}     0
```

You can now process the data from the member as needed. For this example, compute the average value of the signal stored in the variable gs. Extract the data from the table returned by read.

```
gsdata = data.gs{1};
gsmean = mean(gsdata);
```

You can write the mean value gsmean back to the data file as a new variable. To do so, first expand the list of data variables in the ensemble to include a variable for the new value. Call the new variable gsMean.

```
fensemble.DataVariables = [fensemble.DataVariables;"gsMean"]
```

```
fensemble = 
  fileEnsembleDatastore with properties:

                 ReadFcn: @readBearingData
        WriteToMemberFcn: @writeBearingData
           DataVariables: [5x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: "label"
       SelectedVariables: [3x1 string]
                ReadSize: 1
              NumMembers: 5
          LastMemberRead: "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\7\tp27f11553\predmaint-ex3416588
                   Files: [5x1 string]
```

Next, write the derived mean value to the file corresponding to the last-read ensemble member. (See "Data Ensembles for Condition Monitoring and Predictive Maintenance".) When you call `writeToLastMemberRead`, it converts the data to a structure and calls `fensemble.WriteToMemberFcn` to write the data to the file.

```
writeToLastMemberRead(fensemble,'gsMean',gsmean);
```

Calling `read` again advances the last-read-member indicator to the next file in the ensemble and reads the data from that file.

```
data = read(fensemble)
```

```
data=1×3 table
     label            gs           load
    ________    _______________    ____

    "Faulty"    {5000x1 double}     50
```

You can confirm that this data is from a different member by examining the `load` variable in the table. Here, its value is 50, while in the previously read member, it was 0.

You can repeat the processing steps to compute and append the mean for this ensemble member. In practice, it is more useful to automate the process of reading, processing, and writing data. To do so, reset the ensemble to a state in which no data has been read. Then loop through the ensemble and perform the read, process, and write steps for each member.

```
reset(fensemble)
while hasdata(fensemble)
    data = read(fensemble);
    gsdata = data.gs{1};
    gsmean = mean(gsdata);
    writeToLastMemberRead(fensemble,'gsMean',gsmean);
end
```

The `hasdata` command returns `false` when every member of the ensemble has been read. Now, each data file in the ensemble includes the `gsMean` variable derived from the data `gs` in that file. You can use techniques like this loop to extract and process data from your ensemble files as you develop a predictive-maintenance algorithm. For an example illustrating in more detail the use of a file ensemble datastore in the algorithm-development process, see "Rolling Element Bearing Fault Diagnosis". The example also shows how to use Parallel Computing Toolbox™ to speed up the processing of large data ensembles.

To confirm that the derived variable is present in the file ensemble datastore, read it from the first and second ensemble members. To do so, reset the ensemble again, and add the new variable to the selected variables. In practice, after you have computed derived values, it can be useful to read only those values without rereading the unprocessed data, which can take significant space in memory. For this example, read selected variables that include the new variable, `gsMean`, but do not include the unprocessed data, `gs`.

```
reset(fensemble)
fensemble.SelectedVariables = ["label";"load";"gsMean"];
data1 = read(fensemble)
```

```
data1=1×3 table
     label      load     gsMean 
    ________    ____    ________

    "Faulty"     0      -0.22648
```

```
data2 = read(fensemble)
```

```
data2=1×3 table
     label      load     gsMean 
    ________    ____    ________

    "Faulty"     50     -0.22937
```

```
rmpath(fullfile(matlabroot,'examples','predmaint','main')) % Reset path
```

Input Arguments

ensemble — Ensemble datastore
`simulationEnsembleDatastore` object | `fileEnsembleDatastore` object

Ensemble datastore to add data variables to, specified as a:

- `simulationEnsembleDatastore` object
- `fileEnsembleDatastore` object

`writeToLastMemberRead` writes the data to the last-read member of the specified ensemble, identified by the `LastMemberRead` property of the ensemble. The last-read ensemble member is the member most recently accessed using the `read` command. (See “Data Ensembles for Condition Monitoring and Predictive Maintenance”.)

data — New data
`table`

New data to write to the current ensemble member, specified as a `table`. For example, suppose that you have calculated two values that you want to add to the current member: a vector stored as the MATLAB workspace variable `Afilt`, and a scalar stored as `Amean`. Use the following command to construct `data`.

```
data = table(Afilt,Amean,'VariableNames',{'Afilt','Amean'});
```

The number of rows in the table must match the `ReadSize` property of `ensemble`. By default, `ReadSize` = 1, and you write a single table row to a single ensemble member. When you configure

ensemble to read multiple members at once, you must write to the same number of members. For instance, if ReadSize = 3, then data is a three-row table.

Limitations

- When you use a simulationEnsembleDatastore to manage data at a remote location, such as cloud storage using Amazon S3™ (Simple Storage Service), Windows Azure® Blob Storage, and Hadoop® Distributed File System (HDFS™), you cannot use writeToLastMemberRead to add data to the ensemble datastore.

See Also

simulationEnsembleDatastore | fileEnsembleDatastore | read

Topics

"Data Ensembles for Condition Monitoring and Predictive Maintenance"

Introduced in R2018a

2

Objects

covariateSurvivalModel

Proportional hazard survival model for estimating remaining useful life

Description

Use `covariateSurvivalModel` to estimate the remaining useful life (RUL) of a component using a proportional hazard survival model. This model describes the survival probability of a test component using historical information about the life span of components and associated covariates. Covariates are environmental or explanatory variables, such as the component manufacturer or operating conditions. Covariate survival models are useful when the only data you have is the failure times and associated covariates for an ensemble of similar components, such as multiple machines manufactured to the same specifications. For more information on the survival model, see “Proportional Hazard Survival Model” on page 2-9.

To configure a `covariateSurvivalModel` object for a specific type of component, use `fit`, which estimates model coefficients using a collection of failure-time data and associated covariates. After you configure the parameters of your covariate survival model, you can then predict the remaining useful life of similar components using `predictRUL`. For a basic example illustrating RUL prediction, see “Update RUL Prediction as Data Arrives”.

If you have only life span measurements and do not have covariate information, use a `reliabilitySurvivalModel`.

For general information on predicting remaining useful life, see “Models for Predicting Remaining Useful Life”.

Creation

Syntax

```
mdl = covariateSurvivalModel
mdl = covariateSurvivalModel(initModel)
mdl = covariateSurvivalModel(___,Name,Value)
```

Description

`mdl = covariateSurvivalModel` creates a covariate survival model for estimating RUL and initializes the model with default settings.

`mdl = covariateSurvivalModel(initModel)` creates a covariate survival model and initializes the model parameters using an existing `covariateSurvivalModel` object `initModel`.

`mdl = covariateSurvivalModel(___,Name,Value)` specifies user-settable model properties using name-value pairs. For example, `covariateSurvivalModel('LifeTimeUnit',"days")` creates a covariate survival model with that uses days as a lifetime unit. You can specify multiple name-value pairs. Enclose each property name in quotes.

Input Arguments

initModel — Covariate survival model
covariateSurvivalModel object

Covariate survival model, specified as a covariateSurvivalModel object.

Properties

BaselineCumulativeHazard — Baseline hazard rate function
two-column array

This property is read-only.

Baseline hazard rate of the survival model, specified as a two-column array and estimated by the fit function. The second column contains the baseline survival functions values, and the first column contains the corresponding lifetime values.

For more information on the survival model, see “Proportional Hazard Survival Model” on page 2-9.

EncodingMethod — Encoding method
"dummy" (default) | "binary"

Encoding method for the categorical features in EncodedVariables, specified as one of the following:

- "dummy" — For a categorical feature with *N* categories, encode the variable using (*N* - 1) bits.
- "binary" — Binary encoding

You can specify EncodingMethod:

- Using a name-value pair when you create the model
- Using dot notation after model creation

Standardize — Flag for standardizing covariate features
false (default) | true

Flag for standardizing covariate features when calculating Cox regression parameters, specified as a logical value. When Standardize is true, numeric covariate variables are standardized such that covariate X becomes (X-mean(X))/std(X).

Standardization does not affect encoded categorical variables.

You can specify Standardize:

- Using a name-value pair when you create the model
- Using dot notation after model creation

Ties — Method for handling tied failure times
"breslow" (default) | "efron"

Method for handling tied failure times, specified as either "breslow" or "efron". For more information on these methods, see “Cox Proportional Hazards Model”.

You can specify `Ties`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`Options` — Numerical and display settings
structure

Numerical and display settings for Cox regression, specified as a structure created using `statset('coxphfit')`. You can modify the options in the structure using dot notation.

You can specify `Options`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`ParameterValues` — Covariate multiplying coefficients
vector

This property is read-only.

Covariate multiplying coefficients of the survival model, specified as a scalar and estimated by the `fit` function. For more information on the survival model, see “Proportional Hazard Survival Model” on page 2-9.

`ParameterCovariance` — Covariance of covariate multiplying coefficients
array

This property is read-only.

Covariance of the covariate multiplying coefficients, specified as a positive array with size equal to the number of coefficients and estimated by the `fit` function.

`ParameterNames` — Covariate multiplying coefficient names
string array

This property is read-only.

Covariate multiplying coefficient names specified as a string array and assigned when the model is trained using the `fit` function.

Coefficients corresponding to numeric covariates have the same name as the corresponding data variable in `DataVariables`. For encoded variables, the coefficient names contain the name of the corresponding encoded variable from `EncodedVariables` and a representation of the encoded bit order.

`CensorVariable` — Censor variable
`""` (default) | string

Censor variable, specified as a string that contains a valid MATLAB variable name. The censor variable is a binary variable that indicates which life-time measurements in `data` are not end-of-life values.

`CensorVariable` must not match any of the strings in `DataVariables` or `LifeTimeVariable`.

You can specify `CensorVariable`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function
- Using dot notation after model creation

`LifeTimeVariable` — Lifetime variable

"" (default) | string

Lifetime variable, specified as a string that contains a valid MATLAB variable name. For survival models, the lifetime variable contains the historical life span measurements of components.

You can specify `LifeTimeVariable`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function
- Manually using dot notation

`LifeTimeUnit` — Lifetime variable units

"" (default) | string

Lifetime variable units, specified as a string.

The units of the lifetime variable do not need to be time-based. The life of the test component can be measured in terms of a usage variable, such as distance traveled (miles) or fuel consumed (gallons).

`DataVariables` — Covariate data variable

"" (default) | string | string array

Covariate data variables, specified as a string or string array. The strings in `DataVariables` must be valid MATLAB variable names. Covariates are also called environmental or explanatory variables.

You can specify `DataVariables`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function
- Using dot notation after model creation

`EncodedVariables` — Encoded covariate variables

"" (default) | string | string array

Encoded covariate variables, specified as a string or string array. The strings in `EncodedVariables` must be valid MATLAB variable names. Encoded variables are usually nonnumeric categorical features that `fit` converts to numeric vectors before fitting. You can also designate logical or numeric values that take values from a small set to be encoded.

To specify the method of encoding, use `EncodingMethod`.

You can specify `EncodedVariables`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function

- Using dot notation after model creation

The strings in `EncodedVariables` must be a subset of the strings in `DataVariables`.

`UserData` — Additional model information
[ ] (default) | any data type or format

Additional model information for bookkeeping purposes, specified as any data type or format. The model does not use this information.

You can specify `UserData`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

Object Functions

predictRUL	Estimate remaining useful life for a test component
fit	Estimate parameters of remaining useful life model using historical data
plot	Plot survival function for covariate survival remaining useful life model

Examples

Train Covariate Survival Model

Load training data.

```
load('covariateData.mat')
```

This data contains battery discharge times and related covariate information. The covariate variables are:

- Temperature
- Load
- Manufacturer

The manufacturer information is a categorical variable that must be encoded.

Create a covariate survival model.

```
mdl = covariateSurvivalModel;
```

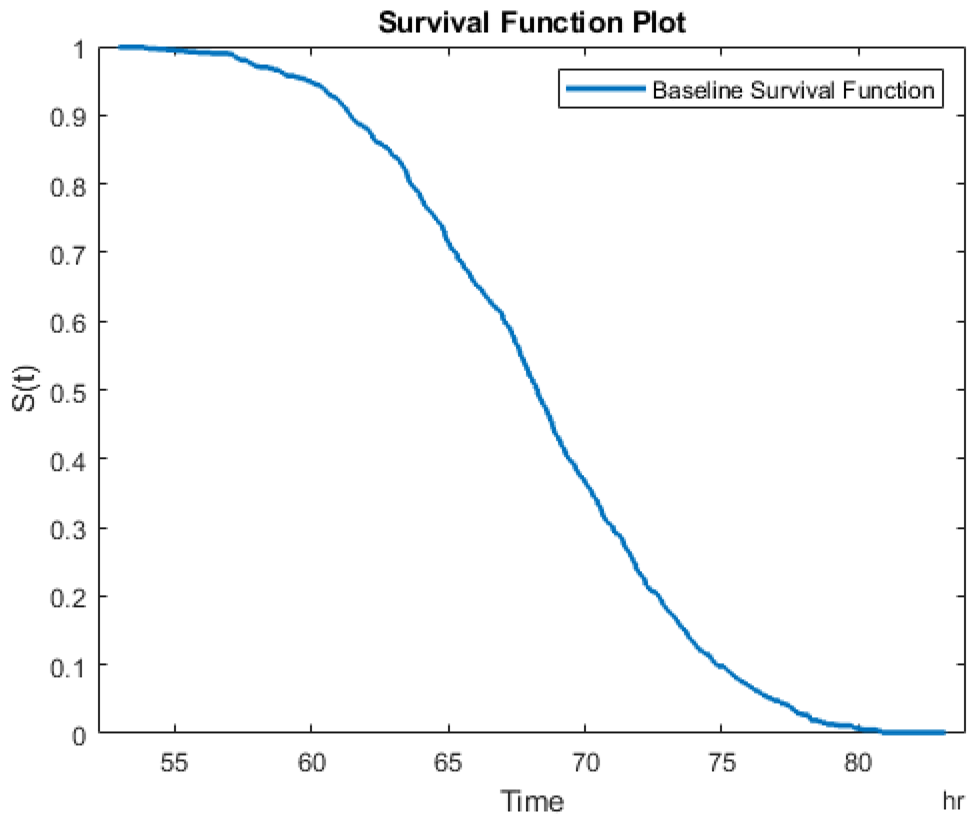
Train the survival model using the training data, specifying the life time variable, data variables, and encoded variable. There is no censor variable for this training data.

```
fit(mdl,covariateData,"DischargeTime",["Temperature","Load","Manufacturer"],[],"Manufacturer")
```

```
Successful convergence: Norm of gradient less than OPTIONS.TolFun
```

Plot the baseline survival function for the model.

```
plot(mdl)
```



Predict RUL Using Covariate Survival Model

Load training data.

```
load('covariateData.mat')
```

This data contains battery discharge times and related covariate information. The covariate variables are:

- Temperature
- Load
- Manufacturer

The manufacturer information is a categorical variable that must be encoded.

Create a covariate survival model, and train it using the training data.

```
mdl = covariateSurvivalModel('LifeTimeVariable',"DischargeTime",'LifeTimeUnit',"hours",...
    'DataVariables',["Temperature","Load","Manufacturer"],'EncodedVariables',"Manufacturer");
fit(mdl,covariateData)
```

```
Successful convergence: Norm of gradient less than OPTIONS.TolFun
```

Suppose you have a battery pack manufactured by maker `B` that has run for `30` hours. Create a test data table that contains the usage time, `DischargeTime`, and the measured ambient temperature, `TestAmbientTemperature`, and current drawn, `TestBatteryLoad`.

```
TestBatteryLoad = 25;
TestAmbientTemperature = 60;
DischargeTime = hours(30);
TestData = timetable(TestAmbientTemperature,TestBatteryLoad,"B",'RowTimes',hours(30));
TestData.Properties.VariableNames = {'Temperature','Load','Manufacturer'};
TestData.Properties.DimensionNames{1} = 'DischargeTime';
```

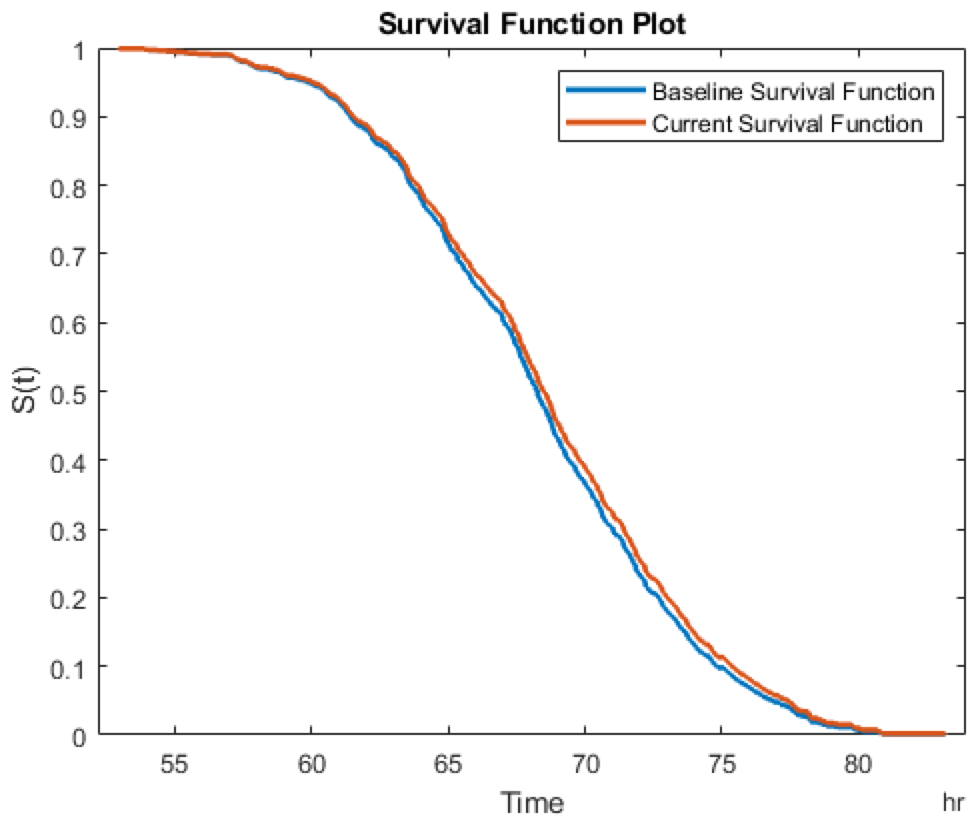
Predict the RUL for the battery.

```
estRUL = predictRUL(mdl,TestData)
```

```
estRUL = duration
   38.332 hr
```

Plot the survival function for the covariate data of the battery.

```
plot(mdl,TestData)
```



Algorithms

Proportional Hazard Survival Model

The covariateSurvivalModel object implements the following proportional hazard survival model:

$$h(X,t) = h_0(t)e^{b^T X}$$

where:

- X is a vector covariate values.
- b is a vector of covariate multiplying coefficients. These coefficients correspond to the ParameterValues property of the model.
- $h_0(t)$ is the baseline hazard rate function, which corresponds to the BaselineCumulativeHazard property of the model.
- $h(X,t)$ is the hazard rate at time t for X.

To find the parameters of this model, the fit function uses the coxphfit function.

For more information on proportional hazard models, see "Cox Proportional Hazards Model".

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

- The predictRUL command supports code generation with MATLAB Coder for this RUL model type.
- Before generating code that uses this model, you must save the model using saveRULModelForCoder. For an example, see "Generate Code for Predicting Remaining Useful Life".
- For a covariateSurvivalModel, you cannot generate code for predictRUL using covariate data specified as string or character-vector values in table or timetable form. Instead, convert the string or character-vector data to a different data type, or delete the data from the table.
- In addition to the read-only properties, the following properties of survival models cannot be changed at run time:
  - LifeTimeVariable
  - LifeTimeUnit
  - DataVariables
  - CensorVariable
  - EncodedVariables (covariateSurvivalModel only)
  - EncodingMethod (covariateSurvivalModel only)

See Also

Functions
`coxphfit` | `reliabilitySurvivalModel` | `fit` | `predictRUL`

Topics
"Update RUL Prediction as Data Arrives"
"RUL Estimation Using RUL Estimator Models"
"Cox Proportional Hazards Model"

Introduced in R2018a

exponentialDegradationModel

Exponential degradation model for estimating remaining useful life

Description

Use `exponentialDegradationModel` to model an exponential degradation process for estimating the remaining useful life (RUL) of a component. Degradation models estimate the RUL by predicting when a monitored signal will cross a predefined threshold. Exponential degradation models are useful when the component experiences cumulative degradation. For more information on the degradation model, see “Exponential Degradation Model” on page 2-19.

To configure an `exponentialDegradationModel` object for a specific type of component, you can:

- Estimate the model parameters using historical data regarding the health of an ensemble of similar components, such as multiple machines manufactured to the same specifications. To do so, use `fit`.
- Specify the model parameters when you create the model based on your knowledge of the component degradation process.

Once you configure the parameters of your degradation model, you can then predict the remaining useful life of similar components using `predictRUL`. For a basic example illustrating RUL prediction with a degradation model, see “Update RUL Prediction as Data Arrives”.

For general information on predicting remaining useful life, see “Models for Predicting Remaining Useful Life”.

Creation

Syntax

```
mdl = exponentialDegradationModel
mdl = exponentialDegradationModel(Name,Value)
```

Description

`mdl = exponentialDegradationModel` creates an exponential degradation model for estimating RUL and initializes the model with default settings.

`mdl = exponentialDegradationModel(Name,Value)` specifies user-settable model properties using name-value pairs. For example, `exponentialDegradationModel('NoiseVariance',0.5)` creates an exponential degradation model with a model noise variance of `0.5`. You can specify multiple name-value pairs. Enclose each property name in quotes.

Properties

`Theta` — Current mean value of the θ parameter
scalar

This property is read-only.

Current mean value of the θ parameter in the degradation model, specified as a scalar. For more information on the degradation model, see "Exponential Degradation Model" on page 2-19.

You can specify `Theta` using a name-value pair argument when you:

- Create the model.
- Reset the model using the `restart` function.

Otherwise, the value of `Theta` changes when you use the `update` function.

`ThetaVariance` — Current variance of the θ parameter
nonnegative scalar

This property is read-only.

Current variance of the θ parameter in the degradation model, specified as a nonnegative scalar. For more information on the degradation model, see "Exponential Degradation Model" on page 2-19.

You can specify `ThetaVariance` using a name-value pair argument when you:

- Create the model.
- Reset the model using the `restart` function.

Otherwise, the value of `ThetaVariance` changes when you use the `update` function.

`Beta` — Current mean value of the β parameter
scalar

This property is read-only.

Current mean value of the β parameter in the degradation model, specified as a scalar. For more information on the degradation model, see "Exponential Degradation Model" on page 2-19.

You can specify `Beta` using a name-value pair argument when you:

- Create the model.
- Reset the model using the `restart` function.

Otherwise, the value of `Beta` changes when you use the `update` function.

`BetaVariance` — Current variance of the β parameter
nonnegative scalar

This property is read-only.

Current variance of the β parameter in the degradation model, specified as a nonnegative scalar. For more information on the degradation model, see "Exponential Degradation Model" on page 2-19.

You can specify `BetaVariance` using a name-value pair argument when you:

- Create the model.
- Reset the model using the `restart` function.

Otherwise, the value of `BetaVariance` changes when you use the `update` function.

Rho — Current correlation between θ and β
0 (default) | scalar value in the range [-1,1]

This property is read-only.

Current correlation between θ and β, specified as a scalar value in the range [-1,1]. For more information on the degradation model, see “Exponential Degradation Model” on page 2-19.

You can specify `Rho` using a name-value pair argument when you:

- Create the model.
- Reset the model using the `restart` function.

Otherwise, the value of `Rho` changes when you use the `update` function.

`Phi` — Current intercept value
scalar

Current intercept value ϕ in the degradation model, specified as a scalar. For more information on the degradation model, see “Exponential Degradation Model” on page 2-19.

You can specify `Phi` using a name-value pair argument when you create the model. Otherwise, the value of `Phi` changes when you estimate the model prior using the `fit` function.

`Prior` — Prior information about model parameters
structure

Prior information about model parameters, specified as a structure with the following fields:

- `Theta` — Mean value of θ
- `ThetaVariance` — Variance of θ
- `Beta` — Mean value of β
- `BetaVariance` — Variance of β
- `Rho` — Correlation between θ and β.

You can specify the fields of `Prior`:

- When you create the model. When you specify `Theta`, `ThetaVariance`, `Beta`, `BetaVariance`, or `Rho` at model creation using name-value pairs, the corresponding field of `Prior` is also set.
- Using the `fit` function. In this case, the prior values are derived from the data used to fit the model.
- Using the `restart` function. In this case, the current values of `Theta`, `ThetaVariance`, `Beta`, `BetaVariance`, and `Rho` are copied to the corresponding fields of `Prior`.
- Using dot notation after model creation.

For more information on the degradation model, see “Exponential Degradation Model” on page 2-19.

`NoiseVariance` — Variance of additive noise
1 (default) | nonnegative scalar

Variance of additive noise ε in the degradation model, specified as a nonnegative scalar. For more information on the degradation model, see "Exponential Degradation Model" on page 2-19.

You can specify `NoiseVariance`:

- Using a name-value pair when you create the model
- Using a name-value pair with the `restart` function
- Using dot notation after model creation

`SlopeDetectionLevel` — Slope detection level
`0.05` (default) | scalar value in the range [0,1] | `[]`

Slope detection level for determining the start of the degradation process, specified as a scalar in the range [0,1]. This value corresponds to the alpha value in a t-test of slope significance.

To disable the slope detection test, set `SlopeDetectionLevel` to `[]`.

You can specify `SlopeDetectionLevel`:

- Using a name-value pair when you create the model
- Using a name-value pair with the `restart` function
- Using dot notation after model creation

`SlopeDetectionInstant` — Slope detection time
`[]` (default) | scalar

This property is read-only.

Slope detection time, which is the instant when a significant slope is detected, specified as a scalar. The `update` function sets this value when `SlopeDetectionLevel` is not empty.

`CurrentMeasurement` — Latest degradation feature value
scalar

This property is read-only.

Latest degradation feature value supplied to the `update` function, specified as a scalar.

`InitialLifeTimeValue` — Initial lifetime variable value
scalar | `duration` object

This property is read-only.

Initial lifetime variable value when the `update` function is first called on the model, specified as a scalar.

When the model detects a slope, the `InitialLifeTime` value is changed to match the `SlopeDetectionInstant` value.

`CurrentLifeTimeValue` — Current lifetime variable value
scalar | `duration` object

This property is read-only.

Latest lifetime variable value supplied to the `update` function, specified as a scalar.

LifeTimeVariable — Lifetime variable
"" (default) | string

Lifetime variable, specified as a string that contains a valid MATLAB variable name or "".

When you train the model using the fit function, if your training data is a:

- table, then LifeTimeVariable must match one of the variable names in the table
- timetable, then LifeTimeVariable one of the variable names in the table or the dimension name of the time variable, data.Properties.DimensionNames{1}

You can specify LifeTimeVariable:

- Using a name-value pair when you create the model
- As an argument when you call the fit function
- Using dot notation after model creation

LifeTimeUnit — Lifetime variable units
"" (default) | string

Lifetime variable units, specified as a string.

The units of the lifetime variable do not need to be time-based. The life of the test component can be measured in terms of a usage variable, such as distance traveled (miles) or fuel consumed (gallons).

DataVariables — Degradation variable name
"" (default) | string

Degradation variable name, specified as a string that contains a valid MATLAB variable name. Degradation models have only one data variable.

You can specify DataVariables:

- Using a name-value pair when you create the model
- As an argument when you call the fit function
- Using dot notation after model creation

UseParallel — Flag for using parallel computing
false (default) | true

Flag for using parallel computing when fitting prior values from data, specified as either true or false.

You can specify UseParallel:

- Using a name-value pair when you create the model
- Using a name-value pair with the restart function
- Using dot notation after model creation

UserData — Additional model information
[] (default) | any data type or format

Additional model information for bookkeeping purposes, specified as any data type or format. The model does not use this information.

You can specify `UserData`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

Object Functions

fit	Estimate parameters of remaining useful life model using historical data
predictRUL	Estimate remaining useful life for a test component
update	Update posterior parameter distribution of degradation remaining useful life model
restart	Reset remaining useful life degradation model

Examples

Train Exponential Degradation Model

Load training data.

```
load('expTrainVectors.mat')
```

The training data is a cell array of column vectors. Each column vector is a degradation feature profile for a component.

Create an exponential degradation model with default settings.

```
mdl = exponentialDegradationModel;
```

Train the degradation model using the training data.

```
fit(mdl,expTrainVectors)
```

Create Exponential Degradation Model with Known Priors

Create an exponential degradation model, and configure it with a known prior distribution.

```
mdl = exponentialDegradationModel('Theta',0.5,'ThetaVariance',0.003,...
                                  'Beta',0.3,'BetaVariance',0.002,...
                                  'Rho',0.1);
```

The specified prior distribution parameters are stored in the `Prior` property of the model.

```
mdl.Prior
```

```
ans = struct with fields:
            Theta: 0.5000
    ThetaVariance: 0.0030
             Beta: 0.3000
     BetaVariance: 0.0020
              Rho: 0.1000
```

The current posterior distribution of the model is also set to match the specified prior distribution. For example, check the posterior value of the correlation parameter.

```
mdl.Rho
```

```
ans = 0.1000
```

Train Exponential Degradation Model Using Tabular Data

Load training data.

```
load('expTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component. Each profile consists of life time measurements in the `"Time"` variable and corresponding degradation feature measurements in the `"Condition"` variable.

Create a exponential degradation model with default settings.

```
mdl = exponentialDegradationModel;
```

Train the degradation model using the training data. Specify the names of the life time and data variables.

```
fit(mdl,expTrainTables,"Time","Condition")
```

Predict RUL Using Exponential Degradation Model

Load training data.

```
load('expTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component. Each profile consists of life time measurements in the `"Hours"` variable and corresponding degradation feature measurements in the `"Condition"` variable.

Create an exponential degradation model, specifying the life time variable units.

```
mdl = exponentialDegradationModel('LifeTimeUnit',"hours");
```

Train the degradation model using the training data. Specify the names of the life time and data variables.

```
fit(mdl,expTrainTables,"Time","Condition")
```

Load testing data, which is a run-to-failure degradation profile for a test component. The test data is a table with the same life time and data variables as the training data.

```
load('expTestData.mat')
```

Based on knowledge of the degradation feature limits, define a threshold condition indicator value that indicates the end-of-life of a component.

```
threshold = 500;
```

Assume that you measure the component condition indicator every hour for 150 hours. Update the trained degradation model with each measurement. Then, predict the remaining useful life of the

component at 150 hours. The RUL is the forecasted time at which the degradation feature will pass the specified threshold.

```
for t = 1:150
 update(mdl,expTestData(t,:))
end
estRUL = predictRUL(mdl,threshold)
```

```
estRUL = duration
   136.45 hr
```

The estimated RUL is around 137 hours, which indicates a total predicted life span of 287 hours.

Update Exponential Degradation Model and Predict RUL

Load observation data.

```
load('expTestData.mat')
```

For this example, assume that the training data is not historical data, but rather real-time observations of the component condition.

Based on knowledge of the degradation feature limits, define a threshold condition indicator value that indicates the end-of-life of a component.

```
threshold = 500;
```

Create an exponential degradation model arbitrary prior distribution data and a specified noise variance. Also, specify the life time and data variable names for the observation data.

```
mdl = exponentialDegradationModel('Theta',1,'ThetaVariance',1e6,...
                                  'Beta',1,'BetaVariance',1e6,...
                                  'NoiseVariance',0.003,...
                                  'LifeTimeVariable',"Time",'DataVariables',"Condition",...
                                  'LifeTimeUnit',"hours");
```

Observe the component condition for 100 hours, updating the degradation model after each observation.

```
for i=1:100
    update(mdl,expTestData(i,:));
end
```

After 100 hours, predict the RUL of the component using the current life time value stored in the model. Also, obtain the confidence interval associated with the estimated RUL.

```
estRUL = predictRUL(mdl,threshold)
```

```
estRUL = duration
   221.38 hr
```

The estimated RUL is about 234 hours, which indicates a total predicted life span of 334 hours.

Algorithms

Exponential Degradation Model

The `exponentialDegradationModel` object implements the following continuous-time exponential degradation model [1]:

$$S(t) = \phi + \theta(t)e^{\left(\beta(t)t + \varepsilon(t) - \frac{\sigma^2}{2}\right)}$$

where:

- ϕ is the model intercept, which is constant. You can initialize ϕ as the lower or upper bound on the feasible region of the degradation variable using `Phi`. If the sign of θ is:
  - Positive, then ϕ is a lower bound.
  - Negative, then ϕ is an upper bound.
- $\theta(t)$ is a random variable modeled as a lognormal distribution with mean `Theta` and variance `ThetaVariance`.
- $\beta(t)$ is a random variable modeled as a Gaussian distribution with mean `Beta` and variance `BetaVariance`.
- $\varepsilon(t)$ is the model additive noise and is modeled as a normal distribution with zero mean and variance `NoiseVariance`.
- σ^2 is equal to `NoiseVariance`.

References

[1] Gebraeel, Nagi. "Sensory-Updated Residual Life Distributions for Components with Exponential Degradation Patterns." *IEEE Transactions on Automation Science and Engineering.* Vol. 3, Number 4, 2006, pp. 382–393.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

- The `predictRUL`, `update`, and `restart` commands support code generation with MATLAB Coder for this RUL model type.
- Before generating code that uses this model, you must save the model using `saveRULModelForCoder`. For an example, see “Generate Code for Predicting Remaining Useful Life”.
- In addition to its read-only properties, the following properties cannot be changed at run time:
  - `LifeTimeVariable`
  - `LifeTimeUnit`
  - `DataVariables`

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To evaluate these models in parallel, set the `UseParallel` property to `true`.

See Also

Functions
`linearDegradationModel` | `fit` | `predictRUL` | `update`

Topics
"Update RUL Prediction as Data Arrives"
"RUL Estimation Using RUL Estimator Models"

Introduced in R2018a

fileEnsembleDatastore

Manage ensemble data in custom file format

Description

A `fileEnsembleDatastore` object is a datastore specialized for use in developing algorithms for condition monitoring and predictive maintenance using measured data.

An ensemble is a collection of member data stored in a collection of files. The `fileEnsembleDatastore` object specifies the data variables, independent variables, and condition variables in the ensemble. You provide functions that tell the `fileEnsembleDatastore` object how to read each type of variable from the collection of files. Therefore, you can use `fileEnsembleDatastore` to manage ensemble data stored in any file format or configuration of variables.

The data for a `fileEnsembleDatastore` object can be stored at any location supported by MATLAB datastores, including remote locations, such as cloud storage using Amazon S3 (Simple Storage Service), Windows Azure Blob Storage, and Hadoop Distributed File System (HDFS).

For a detailed example illustrating the use of a file ensemble datastore, see “File Ensemble Datastore With Measured Data”. For general information about data ensembles in Predictive Maintenance Toolbox, see “Data Ensembles for Condition Monitoring and Predictive Maintenance”.

Creation

Syntax

```
fensemble = fileEnsembleDatastore(location,extension)
fensemble = fileEnsembleDatastore(location,extension,Name,Value)
```

Description

`fensemble = fileEnsembleDatastore(location,extension)` creates a `fileEnsembleDatastore` object that points to data at the file path specified by `location` and having the specified file extension. Set properties of the object to specify the functions for reading from and writing to the ensemble datastore.

`fensemble = fileEnsembleDatastore(location,extension,Name,Value)` specifies additional properties on page 2-22 of the object using one or more name-value pair arguments. For example, using `'ConditionVariables',["FaultCond";"ID"]` specifies the condition variables when you create the object.

Input Arguments

`location` — Files or folders
string | character vector | string array | cell array of character vectors

Files or folders from which to read ensemble data, specified as a string, character vector, string array, or cell array of character vectors. If the files are not in the current folder, then `location` must contain full or relative paths.

If you specify a folder, then `fileEnsembleDatastore` uses all files in that folder with the extension specified by `extension`. Alternatively, specify an explicit list of files to include. You can also use the wildcard character (*) when specifying `location`. This character indicates that all matching files or all files in the matching folders are included in the datastore.

The file path can be any location supported by MATLAB datastores, including an IRI path pointing to a remote location, such as cloud storage using Amazon S3 (Simple Storage Service), Windows Azure Blob Storage, and Hadoop Distributed File System (HDFS). For more information about working with remote data in MATLAB, see “Work with Remote Data”.

Example: `pwd + "\simResults"`

Example: `{'C:\dir\data\file1.xls','C:\dir\data\file2.xlsx'}`

Example: `"../dir/data/*.mat"`

extension — File extension
string | character vector | string vector

File extension for files in the datastore, specified as a string or a character vector, such as `".mat"` or `'.csv'`.

If the datastore contains files having more than one extension, specify them as a string vector, such as `[".xls",".xlsx"]`. The functions that you supply for the `ReadFcn` and `WriteToMemberFcn` properties must be able to interact with all specified file types.

Properties

ReadFcn — Function for reading all selected variables
[ ] (default) | function handle

Function for reading all selected variables from the ensemble, specified as a handle to a function you provide. You write a function that instructs the software how to read variables from a data file containing a member of your ensemble. The function has:

- Two inputs, a file name (string), and the names of signals (string vector) to load from the file
- One output, a table row with table variables for each independent variable

When you specify `ReadFcn`, the software uses this function to read all selected variables from the ensemble, regardless of whether they are named in `DataVariables`, `IndependentVariables`, or `ConditionVariables`.

For example, suppose that you write the following function, `readVars`, for reading variables from your files. This function creates a table containing the variables in a data file that match those in the input string vector, `variables`.

```
function data = readVars(filename,variables)
data = table();
mfile = matfile(filename); % Allows partial loading
for ct=1:numel(variables)
    val = mfile.(variables{ct});
```

```
    if numel(val) > 1
        val = {val};
    end
    data.(variables{ct}) = val;
end
end
```

Save the function in a MATLAB file in the current folder or on the path. Then, if you create a `fileEnsembleDatastore` called `fensemble`, set `ReadFcn` as follows.

```
fensemble.ReadFcn = @readVars;
```

When you call `read(fensemble)`, the software uses `readVars` to read all the variables in the `SelectedVariables` property of the ensemble datastore. You must set this property to read data from a `fileEnsembleDatastore` member. Otherwise, `read` generates an error.

WriteToMemberFcn — Function for adding data

[ ] (default) | function handle

Function for writing data to the last-read member of the ensemble, specified as a handle to a function you provide. You write a function that instructs the software how to write variables to a data file containing a member of your ensemble. The function has:

- Two inputs, a file name (string), and a data structure whose field names are the data variables to write, and whose values are the corresponding values
- No outputs

For example, suppose that you write the following function, `writeNewData`, for writing data to your files. This function writes an input data structure named `data` to the specified data file.

```
function writeNewData(filename,data)
save(filename, '-append', '-struct', 'data');
end
```

Store `writeNewData` in a MATLAB file in the current folder or on the path. Then, if you create a `fileEnsembleDatastore` called `fensemble`, set `WriteToMemberFcn` as follows:

```
fensemble.WriteToMemberFcn = @writeNewData;
```

When you call the `writeToLastMemberRead` command on `fensemble`, the software uses `writeNewData` to add the new data to the data file of the last-read ensemble member. You must set this property to add data to a `fileEnsembleDatastore` member. Otherwise, `writeToLastMemberRead` generates an error.

DataVariables — Data variables in the ensemble

[ ] (default) | string array

Data variables in the ensemble, specified as a string array. Data variables are the main content of the members of an ensemble. Data variables can include measured data or derived data for analysis and development of predictive maintenance algorithms. For example, your data variables might include measured or simulated vibration signals and derived values such as mean vibration value or peak vibration frequency. In practice, your data variables, independent variables, and condition variables are all distinct sets of variables.

You can also specify `DataVariables` using a cell array of character vectors, such as `{'Vibration';'Tacho'}`, but the variable names are always stored as a string array,

`["Vibration";"Tacho"]`. If you specify a matrix of variable names, the matrix is flattened to a column vector.

`IndependentVariables` — Independent variables in the ensemble

[ ] (default) | string array

Independent variables in the ensemble, specified as a string array. You typically use independent variables to order the members of an ensemble. Examples are timestamps, number of operating hours, or miles driven. Set this property to the names of such variables in your ensemble. In practice, your data variables, independent variables, and condition variables are all distinct sets of variables.

You can also specify `IndependentVariables` using a cell array of character vectors, such as `{'Time';'Age'}`, but the variable names are always stored as a string array, `["Time";"Age"]`. If you specify a matrix of variable names, the matrix is flattened to a column vector.

`ConditionVariables` — Condition variables in the ensemble

[ ] (default) | string array

Condition variables in the ensemble, specified as a string array. Use condition variables to label the members in a ensemble according to the fault condition or other operating condition under which the ensemble member was collected. In practice, your data variables, independent variables, and condition variables are all distinct sets of variables.

You can also specify `ConditionVariables` using a cell array of character vectors, such as `{'GearFault';'Temperature'}`, but the variable names are always stored as a string array, `["GearFault";"Temperature"]`. If you specify a matrix of variable names, the matrix is flattened to a column vector.

`SelectedVariables` — Variables to read

[ ] (default) | string array

Variables to read from the ensemble, specified as a string array. Use this property to specify which variables are extracted to the MATLAB workspace when you use the `read` command to read data from the current member ensemble. `read` returns a table row containing a table variable for each name specified in `SelectedVariables`. For example, suppose that you have an ensemble, `fensemble`, that contains six variables, and you want to read only two of them, `Vibration` and `FaultState`. Set the `SelectedVariables` property and call `read`:

```
fensemble.SelectedVariables = ["Vibration";"FaultState"];
data = read(fensemble)
```

`SelectedVariables` can be any combination of the variables in the `DataVariables`, `ConditionVariables`, and `IndependentVariables` properties. If `SelectedVariables` is empty, `read` generates an error.

You can specify `SelectedVariables` using a cell array of character vectors, such as `{'Vibration';'Tacho'}`, but the variable names are always stored as a string array, `["Vibration";"Tacho"]`. If you specify a matrix of variable names, the matrix is flattened to a column vector.

`ReadSize` — Number of members to read

1 (default) | positive integer

Number of members to read from the ensemble datastore at once, specified as a positive integer that is smaller than the total number of members in the ensemble. By default, the `read` command returns

a one-row table containing data from one ensemble member. To read data from multiple members in a single `read` operation, set this property to an integer value greater than one. For example, if `ReadSize` = 3, then `read` returns a three-row table where each row contains data from a different ensemble member. If fewer than `ReadSize` members are unread, then `read` returns a table with as many rows as there are remaining members.

The ensemble datastore property `LastMemberRead` contains the names of all files read during the most recent `read` operation. Thus, for instance, if `ReadSize` = 3, then a `read` operation sets `LastMemberRead` to a string vector containing three file names.

When you use `writeToLastMemberRead`, specify the data to write as a table with a number of rows equal to `ReadSize`. The `writeToLastMemberRead` command updates the members specified by `LastMemberRead`, writing one table row to each specified file.

Changing the `ReadSize` property also resets the ensemble to its unread state. For instance, suppose that you read some ensemble members one at a time (`ReadSize` = 1), and then change `ReadSize` to 3. The next `read` operation returns data from the first three ensemble members.

`NumMembers` — Number of members in ensemble
positive integer

This property is read-only.

Number of members in the ensemble, specified as a positive integer.

`LastMemberRead` — File name of last ensemble member read
`""` (default) | string | string array

This property is read-only.

File name of last ensemble member read into the MATLAB workspace, specified as a string. When you use the `read` command to read data from an ensemble datastore, the software determines which ensemble member to read next, and reads data from the corresponding file. The `LastMemberRead` property contains the path to the most recently read file. When the ensemble datastore has not yet been read, or has been reset, `LastMemberRead` is an empty string.

When you call `writeToLastMemberRead` to add data back to the ensemble datastore, that function writes to the file specified in `LastMemberRead`.

By default, `read` reads data from one ensemble member at a time (the `ReadSize` property of the ensemble datastore is 1). When `ReadSize` > 1, `LastMemberRead` is a string array containing the paths to all files read in the most recent `read` operation.

`Files` — List of files in ensemble datastore
string vector

This property is read-only.

List of files in the ensemble datastore, specified as a column string vector of length `NumMembers`. Each entry contains the full path to a file in the datastore. The files are in the order in which the `read` command reads ensemble members.

Example: `["C:\Data\Data_01.csv"; "C:\Data\Data_02.csv"; "C:\Data\Data_03.csv"]`

Object Functions

The `read`, `writeToLastMemberRead`, and `subset` functions are specialized for Predictive Maintenance Toolbox ensemble data. Other functions, such as `reset` and `hasdata`, are identical to those used with `datastore` objects in MATLAB. To transfer all the member data into a table or cell array with a single command, use `readall`. To extract specific ensemble members into a smaller or more specialized ensemble datastore, use `subset`. To partition an ensemble datastore, use the `partition(ds,n,index)` syntax of the `partition` function.

read	Read member data from an ensemble datastore
writeToLastMemberRead	Write data to member of an ensemble datastore
subset	Create new ensemble datastore from subset of existing ensemble datastore
reset	Reset datastore to initial state
hasdata	Determine if data is available to read
progress	Determine how much data has been read
readall	Read all data in datastore
numpartitions	Number of datastore partitions
partition	Partition a datastore
tall	Create tall array
isPartitionable	Determine whether datastore is partitionable
isShuffleable	Determine whether datastore is shuffleable

Examples

Create and Configure File Ensemble Datastore

Create a file ensemble datastore for data stored in MATLAB® files, and configure it with functions that tell the software how to read from and write to the datastore.

For this example, you have two data files containing healthy operating data from a bearing system, `baseline_01.mat` and `baseline_02.mat`. You also have three data files containing faulty data from the same system, `FaultData_01.mat`, `FaultData_02.mat`, and `FaultData_03.mat`.

```
unzip fileEnsData.zip  % extract compressed files
location = pwd;
extension = '.mat';
fensemble = fileEnsembleDatastore(location,extension);
```

Before you can interact with data in the ensemble, you must create functions that tell the software how to process the data files to read variables into the MATLAB workspace and to write data back to the files. Save these functions to a location on the file path. For this example, use the following supplied functions:

- `readBearingData` — Extract requested variables from a structure, `bearing`, and other variables stored in the file. This function also parses the filename for the fault status of the data. The function returns a table row containing one table variable for each requested variable.
- `writeBearingData` — Take a structure and write its variables to a data file as individual stored variables.

```
fensemble.ReadFcn = @readBearingData;
fensemble.WriteToMemberFcn = @writeBearingData;
```

Finally, set properties of the ensemble to identify data variables, condition variables, and selected variables for reading. For this example, the variables in the data file are `gs`, `sr`, `load`, and `rate`. Suppose that you only need to read the fault label, `gs`, and `sr`. Set these variables as the selected variables.

```
fensemble.DataVariables = ["gs";"sr";"load";"rate"];
fensemble.ConditionVariables = ["label"];
fensemble.SelectedVariables = ["label";"gs";"sr"];
```

Examine the ensemble. The functions and the variable names are assigned to the appropriate properties.

```
fensemble
```

```
fensemble = 
  fileEnsembleDatastore with properties:

                 ReadFcn: @readBearingData
        WriteToMemberFcn: @writeBearingData
           DataVariables: [4x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: "label"
       SelectedVariables: [3x1 string]
                ReadSize: 1
              NumMembers: 5
          LastMemberRead: [0x0 string]
                   Files: [5x1 string]
```

These functions that you assigned tell the `read` and `writeToLastMemberRead` commands how to interact with the data files that make up the ensemble. For example, when you call the `read` command, it uses `readBearingData` to read all the variables in `fensemble.SelectedVariables`. For a more detailed example, see “File Ensemble Datastore With Measured Data”.

Read from and Write to a File Ensemble Datastore

Create a file ensemble datastore for data stored in MATLAB files, and configure it with functions that tell the software how to read from and write to the datastore. (For more details about configuring file ensemble datastores, see “File Ensemble Datastore With Measured Data”.)

```
% Create ensemble datastore that points to datafiles in current folder
unzip fileEnsData.zip  % extract compressed files
location = pwd;
extension = '.mat';
fensemble = fileEnsembleDatastore(location,extension);

% Specify data and condition variables
fensemble.DataVariables = ["gs";"sr";"load";"rate"];
fensemble.ConditionVariables = "label";

% Configure with functions for reading and writing variable data
addpath(fullfile(matlabroot,'examples','predmaint','main')) % Make sure functions are on path
fensemble.ReadFcn = @readBearingData;
fensemble.WriteToMemberFcn = @writeBearingData;
```

The functions tell the `read` and `writeToLastMemberRead` commands how to interact with the data files that make up the ensemble. Thus, when you call the `read` command, it uses `readBearingData`

to read all the variables in `fensemble.SelectedVariables`. For this example, `readBearingData` extracts requested variables from a structure, `bearing`, and other variables stored in the file. It also parses the filename for the fault status of the data.

Specify variables to read, and read them from the first member of the ensemble.

```
fensemble.SelectedVariables = ["gs";"load";"label"];
data = read(fensemble)
```

```
data=1×3 table
     label            gs            load
    ________    _______________    ____

    "Faulty"    {5000x1 double}     0
```

You can now process the data from the member as needed. For this example, compute the average value of the signal stored in the variable `gs`. Extract the data from the table returned by `read`.

```
gsdata = data.gs{1};
gsmean = mean(gsdata);
```

You can write the mean value `gsmean` back to the data file as a new variable. To do so, first expand the list of data variables in the ensemble to include a variable for the new value. Call the new variable `gsMean`.

```
fensemble.DataVariables = [fensemble.DataVariables;"gsMean"]
```

```
fensemble = 
  fileEnsembleDatastore with properties:

                 ReadFcn: @readBearingData
        WriteToMemberFcn: @writeBearingData
           DataVariables: [5x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: "label"
       SelectedVariables: [3x1 string]
                ReadSize: 1
              NumMembers: 5
          LastMemberRead: "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\7\tp27f11553\predmaint-ex3416588
                   Files: [5x1 string]
```

Next, write the derived mean value to the file corresponding to the last-read ensemble member. (See “Data Ensembles for Condition Monitoring and Predictive Maintenance”.) When you call `writeToLastMemberRead`, it converts the data to a structure and calls `fensemble.WriteToMemberFcn` to write the data to the file.

```
writeToLastMemberRead(fensemble,'gsMean',gsmean);
```

Calling `read` again advances the last-read-member indicator to the next file in the ensemble and reads the data from that file.

```
data = read(fensemble)
```

```
data=1×3 table
     label            gs            load
    ________    _______________    ____
```

```
    "Faulty"    {5000x1 double}     50
```

You can confirm that this data is from a different member by examining the `load` variable in the table. Here, its value is 50, while in the previously read member, it was 0.

You can repeat the processing steps to compute and append the mean for this ensemble member. In practice, it is more useful to automate the process of reading, processing, and writing data. To do so, reset the ensemble to a state in which no data has been read. Then loop through the ensemble and perform the read, process, and write steps for each member.

```
reset(fensemble)
while hasdata(fensemble)
    data = read(fensemble);
    gsdata = data.gs{1};
    gsmean = mean(gsdata);
    writeToLastMemberRead(fensemble,'gsMean',gsmean);
end
```

The `hasdata` command returns `false` when every member of the ensemble has been read. Now, each data file in the ensemble includes the `gsMean` variable derived from the data `gs` in that file. You can use techniques like this loop to extract and process data from your ensemble files as you develop a predictive-maintenance algorithm. For an example illustrating in more detail the use of a file ensemble datastore in the algorithm-development process, see “Rolling Element Bearing Fault Diagnosis”. The example also shows how to use Parallel Computing Toolbox™ to speed up the processing of large data ensembles.

To confirm that the derived variable is present in the file ensemble datastore, read it from the first and second ensemble members. To do so, reset the ensemble again, and add the new variable to the selected variables. In practice, after you have computed derived values, it can be useful to read only those values without rereading the unprocessed data, which can take significant space in memory. For this example, read selected variables that include the new variable, `gsMean`, but do not include the unprocessed data, `gs`.

```
reset(fensemble)
fensemble.SelectedVariables = ["label";"load";"gsMean"];
data1 = read(fensemble)
```

```
data1=1×3 table
     label      load     gsMean 
    ________    ____    ________

    "Faulty"     0      -0.22648
```

```
data2 = read(fensemble)
```

```
data2=1×3 table
     label      load     gsMean 
    ________    ____    ________

    "Faulty"     50     -0.22937
```

```
rmpath(fullfile(matlabroot,'examples','predmaint','main')) % Reset path
```

Compatibility Considerations

DataVariablesFcn, IndependentVariablesFcn, and ConditionVariablesFcn properties will be removed
Not recommended starting in R2018b

The DataVariablesFcn, IndependentVariablesFcn, and ConditionVariablesFcn properties will be removed in a future release. Use the ReadFcn property instead.

The ReadFcn property, introduced in R2018b, lets you specify one function to read all variable types from your ensemble datastore. Formerly, you had to designate functions separately for data variables, independent variables, and condition variables. An advantage of using ReadFcn is that the read operation needs to access each member file only once to read all the variables. With separate functions for each variable type, read opens the file up to three times to read all variable types. Thus, designating a single ReadFcn is a more efficient way to access the datastore.

Update Code

To update your code to use the new property:

1. Rewrite your fileEnsembleDatastore read functions into one new function that reads variables of all types. (See “Create and Configure File Ensemble Datastore” on page 2-26 for an example of such a function.)
2. Set DataVariablesFcn, IndependentVariablesFcn, and ConditionVariablesFcn to [] to clear them.
3. Set ReadFcn to the new function.

See Also

generateSimulationEnsemble | simulationEnsembleDatastore

Topics

“Data Ensembles for Condition Monitoring and Predictive Maintenance”
“File Ensemble Datastore With Measured Data”
“File Ensemble Datastore Using Data in Text Files”

Introduced in R2018a

hashSimilarityModel

Hashed-feature similarity model for estimating remaining useful life

Description

Use `hashSimilarityModel` to estimate the remaining useful life (RUL) of a component using a hashed-feature similarity model. This model is useful when you have run-to-failure degradation path histories for an ensemble of similar components, such as multiple machines manufactured to the same specifications, and the data set is large. The hashed-feature similarity model transforms the historical degradation path data for each ensemble member into a series of *hashed-features*, such as the mean, power, minimum, or maximum values for the data. You can then compute the hashed features of a test component and compare them to the hashed features of the ensemble data members.

To configure a `hashSimilarityModel` object, use `fit`, which computes and stores the hashed feature values of the ensemble data members. Once you configure the parameters of your similarity model, you can then predict the remaining useful life of similar components using `predictRUL`. For similarity models, the RUL of the test component is estimated as the median statistic of the lifetime span of the most similar components minus the current lifetime value of the test component. For a basic example illustrating RUL prediction, see “Update RUL Prediction as Data Arrives”.

For general information on predicting remaining useful life, see “Models for Predicting Remaining Useful Life”.

Creation

Syntax

```
mdl = hashSimilarityModel
mdl = hashSimilarityModel(initModel)
mdl = hashSimilarityModel( ___ ,Name,Value)
```

Description

`mdl = hashSimilarityModel` creates a hashed-feature similarity model for estimating RUL and initializes the model with default settings.

`mdl = hashSimilarityModel(initModel)` creates a hashed-feature similarity model and initializes the model parameters using an existing `hashSimilarityModel` object `initModel`.

`mdl = hashSimilarityModel( ___ ,Name,Value)` specifies user-settable model properties using name-value pairs. For example, `hashSimilarityModel('LifeTimeUnit',"days")` creates a hashed-feature similarity model with that uses days as a lifetime unit. You can specify multiple name-value pairs. Enclose each property name in quotes.

Input Arguments

initModel — Hashed-feature similarity model
hashSimilarityModel object

Hashed-feature similarity model, specified as a hashSimilarityModel object.

Properties

HashTable — Hashed feature values
N-by-*M* array

This property is read-only.

Hashed feature values generated by the fit function, specified as *N*-by-*M* array, where *M* is the number of ensemble members and *N* is the number of hashed features. HashTable(i,j) contains the hashed feature value of jth feature computed for the ith data member.

To specify the method for computing the hashed features, use the Method property of the model.

RegimeSplit — Breakpoints for splitting historical data into multiple regimes
row vector of doubles (default) | [] | row vector of duration objects | row vector of datetime objects

Breakpoints for splitting historical data into multiple regimes, specified as a row vector of double values, duration objects, or datetime objects. The row vector of breakpoints must:

- Be in increasing order
- Have units and a format that is compatible with the training data used with the fit function

To use a single regime, specify RegimeSplit as [].

A separate hash table is generated for each regime. The RUL prediction is based on the similarity to the hashed features in the regime to which the test data belongs. If you change the value of RegimeSplit, then you must retrain your model using fit.

You can specify RegimeSplit:

- Using a name-value pair when you create the model
- Using dot notation after model creation

LifeSpan — Ensemble member life spans
double vector (default) | vector of duration objects

This property is read-only.

Ensemble member life spans, specified as a double vector or duration object vector and computed from the ensemble member degradation profiles by the fit function.

NumNearestNeighbors — Number of nearest neighbors for RUL estimation
Inf (default) | finite positive integer

Number of nearest neighbors for RUL estimation, specified as Inf or a finite positive integer. If NumNearestNeighbors is Inf, then predictRUL uses all the ensemble members during estimation.

You can specify `NumNearestNeighbors`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`Method` — Hashed feature computation method
`"minmaxstd"` (default) | function handle

Hashed feature computation method, specified as one of the following:

- `"minmaxstd"` — Extract the minimum, maximum, and standard deviation of the data. This option omits observations that contain `NaN`. When you use this method, `HashTable` is *M*-by-3, where *M* is the number of ensemble members.
- Function handle — Use a custom function that takes degradation data as a column vector, `table`, or `timetable`, and returns a row vector of features. For example:

  ```
  mdl.Method = @(x) [mean(x),std(x),kurtosis(x),median(x)]
  ```

You can specify `Method`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`Distance` — Distance computation method
`"euclidian"` (default) | `"absolute"` | function handle

Distance computation method, specified as one of the following:

- `"euclidian"` — Use the 2-norm of the difference between hash vectors.
- `"absolute"` — Use the 1-norm of the difference between hash vectors.
- Function handle — Use a custom function of the form:

  ```
  D = distanceFunction(xTest,xEnsemble)
  ```

  Here,

  - `xTest` is a column vector of length *N* that contains test component hashed features, where *N* is the number of hashed features.
  - `xEnsemble` is an *M*-by-*N* array of ensemble component hashed features, where *M* is the number of ensemble components. `xEnsemble(i,:)` contains the hashed features for the `i`th ensemble member.
  - `D` is a row vector of length *M*, where `D(i)` is the distance between the test feature vector and the feature vector of the `i`th ensemble member.

You can specify `Distance`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`IncludeTies` — Flag to include ties
`true` (default) | `false`

Flag to include ties, specified as `true` or `false`. When `IncludeTies` is `true`, the model includes all neighbors whose distance to the test component data is less than the *K*th smallest distance, where *K* is equal to `NumNearestNeigbors`.

You can specify `IncludeTies`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`Standardize` — Flag for standardizing feature data

`false` (default) | `true`

Flag for standardizing feature data before generating hashed features, specified as `true` or `false`. When `Standardize` is `true`, the feature data is standardized such that feature `X` becomes `(X-mean(X))/std(X)`.

You can specify `Standardize`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`LifeTimeVariable` — Lifetime variable

`""` (default) | string

Lifetime variable, specified as a string that contains a valid MATLAB variable name or `""`.

When you train the model using the `fit` function, if your training data is a:

- `table`, then `LifeTimeVariable` must match one of the variable names in the table
- `timetable`, then `LifeTimeVariable` one of the variable names in the table or the dimension name of the time variable, `data.Properties.DimensionNames{1}`

You can specify `LifeTimeVariable`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function
- Using dot notation after model creation

`LifeTimeUnit` — Lifetime variable units

`""` (default) | string

Lifetime variable units, specified as a string.

The units of the lifetime variable do not need to be time-based. The life of the test component can be measured in terms of a usage variable, such as distance traveled (miles) or fuel consumed (gallons).

`DataVariables` — Degradation variable names

`""` (default) | string | string array

Degradation variable names, specified as a string or string array. The strings in `DataVariables` must be valid MATLAB variable name.

You can specify `DataVariables`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function
- Using dot notation after model creation

`UseParallel` — Flag for using parallel computing
`false` (default) | `true`

Flag for using parallel computing for hash table generation by the `fit` function, specified as either `true` or `false`.

You can specify `UseParallel`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`UserData` — Additional model information
`[]` (default) | any data type or format

Additional model information for bookkeeping purposes, specified as any data type or format. The model does not use this information.

You can specify `UserData`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

Object Functions

predictRUL	Estimate remaining useful life for a test component
fit	Estimate parameters of remaining useful life model using historical data
compare	Compare test data to historical data ensemble for similarity models

Examples

Train Hash Similarity Model

Load training data.

```
load('hashTrainVectors.mat')
```

The training data is a cell array of column vectors. Each column vector is a degradation feature profile for a component.

Create a hash similarity model with default settings. By default, the hashed features used by the model are the signal maximum, minimum, and standard deviation values.

```
mdl = hashSimilarityModel;
```

Train the similarity model using the training data.

```
fit(mdl,hashTrainVectors)
```

Train Hash Similarity Model Using Tabular Data

Load training data.

```
load('hashTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component. Each profile consists of life time measurements in the "Time" variable and corresponding degradation feature measurements in the "Condition" variable.

Create a hash similarity model that uses the following values as hashed features:

```
mdl = hashSimilarityModel('Method',@(x) [mean(x),std(x),kurtosis(x),median(x)]);
```

Train the similarity model using the training data. Specify the names of the life time and data variables.

```
fit(mdl,hashTrainTables,"Time","Condition")
```

Predict RUL Using Hash Similarity Model

Load training data.

```
load('hashTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component. Each profile consists of life time measurements in the "Time" variable and corresponding degradation feature measurements in the "Condition" variable.

Create a hash similarity model that uses hours as a life time unit and the following values as hashed features:

- Mean
- Standard deviation
- Kurtosis
- Median

```
mdl = hashSimilarityModel('Method',@(x) [mean(x),std(x),kurtosis(x),median(x)],...
                          'LifeTimeUnit',"hours");
```

Train the similarity model using the training data. Specify the names of the life time and data variables.

```
fit(mdl,hashTrainTables,"Time","Condition")
```

Load testing data. The test data contains the degradation feature measurements for a test component up to the current life time.

```
load('hashTestData.mat')
```

Predict the RUL of the test component using the trained similarity model.

```
estRUL = predictRUL(mdl,hashTestData)
```

```
estRUL = duration
   175.69 hr
```

The estimated RUL for the component is around 176 hours.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To evaluate these models in parallel, set the `UseParallel` property to `true`.

See Also

Functions

`pairwiseSimilarityModel` | `residualSimilarityModel` | `fit` | `predictRUL`

Topics

"Update RUL Prediction as Data Arrives"
"RUL Estimation Using RUL Estimator Models"

Introduced in R2018a

linearDegradationModel

Linear degradation model for estimating remaining useful life

Description

Use `linearDegradationModel` to model a linear degradation process for estimating the remaining useful life (RUL) of a component. Degradation models estimate the RUL by predicting when a monitored signal will cross a predefined threshold. Linear degradation models are useful when the monitored signal is a log scale signal or when the component does not experience cumulative degradation. For more information on the degradation model, see “Linear Degradation Model” on page 2-45.

To configure a `linearDegradationModel` object for a specific type of component, you can:

- Estimate the model prior parameters using historical data regarding the health of an ensemble of similar components, such as multiple machines manufactured to the same specifications. To do so, use `fit`.
- Specify the model prior parameters when you create the model based on your knowledge of the component degradation process.

Once you configure the parameters of your degradation model, you can then predict the remaining useful life of similar components using `predictRUL`. For a basic example illustrating RUL prediction with a degradation model, see “Update RUL Prediction as Data Arrives”.

For general information on predicting remaining useful life, see “Models for Predicting Remaining Useful Life”.

Creation

Syntax

```
mdl = linearDegradationModel
mdl = linearDegradationModel(Name,Value)
```

Description

`mdl = linearDegradationModel` creates a linear degradation model for estimating RUL and initializes the model with default settings.

`mdl = linearDegradationModel(Name,Value)` specifies user-settable model properties using name-value pairs. For example, `linearDegradationModel('NoiseVariance',0.5)` creates a linear degradation model with a model noise variance of `0.5`. You can specify multiple name-value pairs. Enclose each property name in quotes.

Properties

`Theta` — Current mean value of slope parameter
scalar

This property is read-only.

Current mean value of slope parameter θ in the degradation model, specified as a scalar. For more information on the degradation model, see “Linear Degradation Model” on page 2-45.

You can specify `Theta` using a name-value pair argument when you:

- Create the model.
- Reset the model using the `restart` function.

Otherwise, the value of `Theta` changes when you use the `update` function.

`ThetaVariance` — Current variance of slope parameter
nonnegative scalar

This property is read-only.

Current variance of slope parameter θ in the degradation model, specified as a nonnegative scalar. For more information on the degradation model, see “Linear Degradation Model” on page 2-45.

You can specify `ThetaVariance` using a name-value pair argument when you:

- Create the model.
- Reset the model using the `restart` function.

Otherwise, the value of `ThetaVariance` changes when you use the `update` function.

`Phi` — Current intercept value
scalar

Current intercept value ϕ for the degradation model, specified as a scalar. For more information on the degradation model, see “Linear Degradation Model” on page 2-45.

You can specify `Phi` using a name-value pair argument when you create the model. Otherwise, the value of `Phi` changes when you estimate the model prior using the `fit` function.

`Prior` — Prior information about model parameters
structure

Prior information about model parameters, specified as a structure with the following fields:

- `Theta` — Mean value of slope parameter
- `ThetaVariance` — Variance of slope parameter

You can specify the fields of `Prior`:

- When you create the model. When you specify `Theta` or `ThetaVariance` at model creation using name-value pairs, the corresponding field of `Prior` is also set.
- Using the `fit` function. In this case, the prior values are derived from the data used to fit the model.

- Using the restart function. In this case, the current values of Theta and ThetaVariance are copied to the corresponding fields of Prior.
- Using dot notation after model creation.

For more information on the degradation model, see “Linear Degradation Model” on page 2-45.

NoiseVariance — Variance of additive noise

1 (default) | nonnegative scalar

Variance of additive noise ε in the degradation model, specified as a nonnegative scalar. For more information on the degradation model, see “Linear Degradation Model” on page 2-45.

You can specify NoiseVariance:

- Using a name-value pair when you create the model
- Using a name-value pair with the restart function
- Using dot notation after model creation

SlopeDetectionLevel — Slope detection level

0.05 (default) | scalar value in the range [0,1] | []

Slope detection level for determining the start of the degradation process, specified as a scalar in the range [0,1]. This value corresponds to the alpha value in a t-test of slope significance.

To disable the slope detection test, set SlopeDetectionLevel to [].

You can specify SlopeDetectionLevel:

- Using a name-value pair when you create the model
- Using a name-value pair with the restart function
- Using dot notation after model creation

SlopeDetectionInstant — Slope detection time

[] (default) | scalar

This property is read-only.

Slope detection time, which is the instant when a significant slope is detected, specified as a scalar. The update function sets this value when SlopeDetectionLevel is not empty.

CurrentMeasurement — Latest degradation feature value

scalar

This property is read-only.

Latest degradation feature value supplied to the update function, specified as a scalar.

InitialLifeTimeValue — Initial lifetime variable value

scalar | duration object

This property is read-only.

Initial lifetime variable value when the update function is first called on the model, specified as a scalar.

When the model detects a slope, the `InitialLifeTime` value is changed to match the `SlopeDetectionInstant` value.

`CurrentLifeTimeValue` — Current lifetime variable value

scalar | `duration` object

This property is read-only.

Latest lifetime variable value supplied to the `update` function, specified as a scalar.

`LifeTimeVariable` — Lifetime variable

`""` (default) | string

Lifetime variable, specified as a string that contains a valid MATLAB variable name or `""`.

When you train the model using the `fit` function, if your training data is a:

- `table`, then `LifeTimeVariable` must match one of the variable names in the table
- `timetable`, then `LifeTimeVariable` one of the variable names in the table or the dimension name of the time variable, `data.Properties.DimensionNames{1}`

You can specify `LifeTimeVariable`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function
- Using dot notation after model creation

`LifeTimeUnit` — Lifetime variable units

`""` (default) | string

Lifetime variable units, specified as a string.

The units of the lifetime variable do not need to be time-based. The life of the test component can be measured in terms of a usage variable, such as distance traveled (miles) or fuel consumed (gallons).

`DataVariables` — Degradation variable name

`""` (default) | string

Degradation variable name, specified as a string that contains a valid MATLAB variable name. Degradation models have only one data variable.

You can specify `DataVariables`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function
- Using dot notation after model creation

`UseParallel` — Flag for using parallel computing

`false` (default) | `true`

Flag for using parallel computing when fitting prior values from data, specified as either `true` or `false`.

You can specify `UseParallel`:

- Using a name-value pair when you create the model
- Using a name-value pair with the `restart` function
- Using dot notation after model creation

`UserData` — Additional model information
[ ] (default) | any data type or format

Additional model information for bookkeeping purposes, specified as any data type or format. The model does not use this information.

You can specify `UserData`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

Object Functions

fit	Estimate parameters of remaining useful life model using historical data
predictRUL	Estimate remaining useful life for a test component
update	Update posterior parameter distribution of degradation remaining useful life model
restart	Reset remaining useful life degradation model

Examples

Train Linear Degradation Model

Load training data.

```
load('linTrainVectors.mat')
```

The training data is a cell array of column vectors. Each column vector is a degradation feature profile for a component.

Create a linear degradation model with default settings.

```
mdl = linearDegradationModel;
```

Train the degradation model using the training data.

```
fit(mdl,linTrainVectors)
```

Create Linear Degradation Model with Known Priors

Create a linear degradation model and configure it with a known prior distribution.

```
mdl = linearDegradationModel('Theta',0.25,'ThetaVariance',0.002);
```

The specified prior distribution parameters are stored in the `Prior` property of the model.

```
mdl.Prior
```

```
ans = struct with fields:
            Theta: 0.2500
```

```
    ThetaVariance: 0.0020
```

The current posterior distribution of the model is also set to match the specified prior distribution. For example, check the posterior value of the slope variance.

```
mdl.ThetaVariance
```

```
ans = 0.0020
```

Train Linear Degradation Model Using Tabular Data

Load training data.

```
load('linTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component. Each profile consists of life time measurements in the "Time" variable and corresponding degradation feature measurements in the "Condition" variable.

Create a linear degradation model with default settings.

```
mdl = linearDegradationModel;
```

Train the degradation model using the training data. Specify the names of the life time and data variables.

```
fit(mdl,linTrainTables,"Time","Condition")
```

Predict RUL Using Linear Degradation Model

Load training data.

```
load('linTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component. Each profile consists of life time measurements in the "Time" variable and corresponding degradation feature measurements in the "Condition" variable.

Create a linear degradation model, specifying the life time variable units.

```
mdl = linearDegradationModel('LifeTimeUnit',"hours");
```

Train the degradation model using the training data. Specify the names of the life time and data variables.

```
fit(mdl,linTrainTables,"Time","Condition")
```

Load testing data, which is a run-to-failure degradation profile for a test component. The test data is a table with the same life time and data variables as the training data.

```
load('linTestData.mat','linTestData1')
```

Based on knowledge of the degradation feature limits, define a threshold condition indicator value that indicates the end-of-life of a component.

```
threshold = 60;
```

Assume that you measure the component condition indicator after 48 hours. Predict the remaining useful life of the component at this time using the trained linear degradation model. The RUL is the forecasted time at which the degradation feature will pass the specified threshold.

```
estRUL = predictRUL(mdl,linTestData1(48,:),threshold)
```

```
estRUL = duration
   112.64 hr
```

The estimated RUL is around 113 hours, which indicates a total predicted life span of around 161 hours.

Update Linear Degradation Model and Predict RUL

Load observation data.

```
load('linTestData.mat','linTestData1')
```

For this example, assume that the training data is not historical data, but rather real-time observations of the component condition.

Based on knowledge of the degradation feature limits, define a threshold condition indicator value that indicates the end-of-life of a component.

```
threshold = 60;
```

Create a linear degradation model arbitrary prior distribution data and a specified noise variance. Also, specify the life time and data variable names for the observation data.

```
mdl = linearDegradationModel('Theta',1,'ThetaVariance',1e6,'NoiseVariance',0.003,...
                             'LifeTimeVariable',"Time",'DataVariables',"Condition",...
                             'LifeTimeUnit',"hours");
```

Observe the component condition for `50` hours, updating the degradation model after each observation.

```
for i=1:50
    update(mdl,linTestData1(i,:));
end
```

After `50` hours, predict the RUL of the component using the current life time value stored in the model.

```
estRUL = predictRUL(mdl,threshold)
```

```
estRUL = duration
   50.301 hr
```

The estimated RUL is about 50 hours, which indicates a total predicted life span of about 100 hours.

Algorithms

Linear Degradation Model

The `linearDegradationModel` object implements the following continuous-time linear degradation model [1]:

$$S(t) = \phi + \theta(t)t + \varepsilon(t)$$

where:

- ϕ is the model intercept, which is constant. You can initialize ϕ as the nominal value of the degradation variable using `Phi`.
- $\theta(t)$ is the model slope and is modeled as a random variable with a normal distribution with mean `Theta` and variance `ThetaVariance`.
- $\varepsilon(t)$ is the model additive noise and is modeled as a normal distribution with zero mean and variance `NoiseVariance`.

References

[1] Chakraborty, S., N. Gebraeel, M. Lawley, and H. Wan. "Residual-Life Estimation for Components with Non-Symmetric Priors." *IIE Transactions*. Vol. 41, Number 4, 2009, pp. 372–387.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

- The `predictRUL`, `update`, and `restart` commands support code generation with MATLAB Coder for this RUL model type.
- Before generating code that uses this model, you must save the model using `saveRULModelForCoder`. For an example, see "Generate Code for Predicting Remaining Useful Life".
- In addition to its read-only properties, the following properties cannot be changed at run time:
  - `LifeTimeVariable`
  - `LifeTimeUnit`
  - `DataVariables`

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To evaluate these models in parallel, set the `UseParallel` property to `true`.

See Also

Functions

`exponentialDegradationModel` | `fit` | `predictRUL` | `update`

Topics
"Update RUL Prediction as Data Arrives"
"RUL Estimation Using RUL Estimator Models"

Introduced in R2018a

pairwiseSimilarityModel

Pairwise comparison-based similarity model for estimating remaining useful life

Description

Use `pairwiseSimilarityModel` to estimate the remaining useful life (RUL) of a component using a pairwise comparison-based similarity model. This model compares the degradation profile of a test component directly to the degradation path histories for an ensemble of similar components, such as multiple machines manufactured to the same specifications. The similarity of the test component to the ensemble members is a function of the distance between the degradation profile and the ensemble member profile, which is computed using correlation or dynamic time warping.

To configure a `pairwiseSimilarityModel` object, use `fit`. Once you configure the parameters of your similarity model, you can then predict the remaining useful life of similar components using `predictRUL`. For similarity models, the RUL of the test component is estimated as the median statistic of the lifetime span of the most similar components minus the current lifetime value of the test component. For a basic example illustrating RUL prediction, see “Update RUL Prediction as Data Arrives”.

For general information on predicting remaining useful life, see “Models for Predicting Remaining Useful Life”.

Creation

Syntax

```
mdl = pairwiseSimilarityModel
mdl = pairwiseSimilarityModel(initModel)
mdl = pairwiseSimilarityModel( ___ ,Name,Value)
```

Description

`mdl = pairwiseSimilarityModel` creates a pairwise comparison-based similarity model for estimating RUL and initializes the model with default settings.

`mdl = pairwiseSimilarityModel(initModel)` creates a pairwise comparison-based similarity model and initializes the model parameters using an existing `pairwiseSimilarityModel` object `initModel`.

`mdl = pairwiseSimilarityModel( ___ ,Name,Value)` specifies user-settable model properties using name-value pairs. For example, `hashSimilarityModel('LifeTimeUnit',"days")` creates a pairwise comparison-based similarity model that uses days as a lifetime unit. You can specify multiple name-value pairs. Enclose each property name in quotes.

Input Arguments

`initModel` — Pairwise comparison-based similarity model
`pairwiseSimilarityModel` object

Pairwise comparison-based similarity model, specified as a `pairwiseSimilarityModel` object.

Properties

`Method` — Time series distance computation method

`"correlation"` (default) | `"dtw"`

Time series distance computation method, specified as one of the following:

- `"correlation"` — Measure distance using correlation
- `"dtw"` — Compute distance using dynamic time warping. For more information, see `dtw`.

You can specify `Method`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`Distance` — Distance formula for `"dtw"`

`"euclidian"` (default) | `"absolute"`

Distance formula for `"dtw"` distance computation method, specified as one of the following:

- `"euclidian"` — Use the 2-norm of the difference between residuals.
- `"absolute"` — Use the 1-norm of the difference between residuals.

You can specify `Distance`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`HistorySpan` — Lifetime span of historical data

`Inf` (default) | positive scalar | `duration` object

Lifetime span of historical data for computing similarity, specified as a positive scalar or `duration` object. When computing similarity, the model uses historical data from lifetime (*t*-`HistorySpan`) to lifetime *t*, where *t* is the current lifetime.

You can specify `HistorySpan`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`WithinRangeRatio` — Factor determining ensemble member exclusion rule

`1` (default) | scalar from 0 through 1

Factor determining ensemble member exclusion rule for similarity computation, specified as a scalar from 0 through 1. `WithinRangeRatio` is used when the length of the test data and the length of the ensemble member data do not match, which happens near end-of-lifetime values of historical data. When `WithinRangeRatio` is `1`, then there is no exclusion of ensemble members.

Suppose that the length of the shorter data is P and the length of the longer data is Q. Then, a similarity test is performed only if Q(1-`WithinRangeRatio`) <= P <= Q. Otherwise, the ensemble member is ignored.

You can specify `WithinRangeRatio`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`LifeSpan` — Ensemble member life spans
double vector (default) | vector of `duration` objects

This property is read-only.

Ensemble member life spans, specified as a double vector or `duration` object vector and computed from the ensemble member degradation profiles by the `fit` function.

`NumNearestNeighbors` — Number of nearest neighbors for RUL estimation
`Inf` (default) | finite positive integer

Number of nearest neighbors for RUL estimation, specified as `Inf` or a finite positive integer. If `NumNearestNeighbors` is `Inf`, then `predictRUL` uses all the ensemble members during estimation.

You can specify `NumNearestNeighbors`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`IncludeTies` — Flag to include ties
`true` (default) | `false`

Flag to include ties, specified as `true` or `false`. When `IncludeTies` is `true`, the model includes all neighbors whose distance to the test component data is less than the Kth smallest distance, where K is equal to `NumNearestNeigbors`.

You can specify `IncludeTies`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`Standardize` — Flag for standardizing feature data
`false` (default) | `true` | `'time-varying'`

Flag for standardizing feature data before computing distance, specified as `true`, `false`, or `'time-varying'`.

When `Standardize` is `true`, the feature data is standardized such that feature `X` becomes `(X-mean(X))/std(X)`.

When `Standardize` is `'time-varying'`, the feature data is standardized such that feature $X(t)$ becomes $(X(t) - M(t)) / S(t)$. Here, $M(t)$ and $S(t)$ are running estimates of the mean and standard deviation of the data.

You can specify `Standardize`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`LifeTimeVariable` — Lifetime variable
`""` (default) | string

Lifetime variable, specified as a string that contains a valid MATLAB variable name or `""`.

When you train the model using the `fit` function, if your training data is a:

- `table`, then `LifeTimeVariable` must match one of the variable names in the table
- `timetable`, then `LifeTimeVariable` one of the variable names in the table or the dimension name of the time variable, `data.Properties.DimensionNames{1}`

You can specify `LifeTimeVariable`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function
- Using dot notation after model creation

`LifeTimeUnit` — Lifetime variable units
`""` (default) | string

Lifetime variable units, specified as a string.

The units of the lifetime variable do not need to be time-based. The life of the test component can be measured in terms of a usage variable, such as distance traveled (miles) or fuel consumed (gallons).

`DataVariables` — Degradation variable names
`""` (default) | string | string array

Degradation variable names, specified as a string or string array. The strings in `DataVariables` must be valid MATLAB variable names.

You can specify `DataVariables`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function
- Using dot notation after model creation

`UseParallel` — Flag for using parallel computing
`false` (default) | `true`

Flag for using parallel computing for nearest-neighbor searching, specified as either `true` or `false`.

You can specify `UseParallel`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`UserData` — Additional model information
`[]` (default) | any data type or format

Additional model information for bookkeeping purposes, specified as any data type or format. The model does not use this information.

You can specify `UserData`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

Object Functions

predictRUL	Estimate remaining useful life for a test component
fit	Estimate parameters of remaining useful life model using historical data
compare	Compare test data to historical data ensemble for similarity models

Examples

Train Pairwise Similarity Model

Load training data.

```
load('pairwiseTrainVectors.mat')
```

The training data is a cell array of column vectors. Each column vector is a degradation feature profile for a component.

Create a pairwise similarity model with default settings.

```
mdl = pairwiseSimilarityModel;
```

Train the similarity model using the training data.

```
fit(mdl,pairwiseTrainVectors)
```

Train Pairwise Similarity Model Using Tabular Data

Load training data.

```
load('pairwiseTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component. Each profile consists of life time measurements in the `"Time"` variable and corresponding degradation feature measurements in the `"Condition"` variable.

Create a pairwise similarity model that computes distance using dynamic time warping with an absolute distance metric.

```
mdl = pairwiseSimilarityModel('Method',"dtw",'Distance',"absolute");
```

Train the similarity model using the training data. Specify the names of the life time and data variables.

```
fit(mdl,pairwiseTrainTables,"Time","Condition")
```

Predict RUL Using Pairwise Similarity Model

Load training data.

```
load('pairwiseTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component. Each profile consists of life time measurements in the `"Time"` variable and corresponding degradation feature measurements in the `"Condition"` variable.

Create a pairwise similarity model that computes distance using dynamic time warping with an absolute distance metric and uses hours as a life time unit.

```
mdl = pairwiseSimilarityModel('Method',"dtw",'Distance',"absolute",'LifeTimeUnit',"hours");
```

Train the similarity model using the training data. Specify the names of the life time and data variables.

```
fit(mdl,pairwiseTrainTables,"Time","Condition")
```

Load testing data. The test data contains the degradation feature measurements for a test component up to the current life time.

```
load('pairwiseTestData.mat')
```

Predict the RUL of the test component using the trained similarity model.

```
estRUL = predictRUL(mdl,pairwiseTestData)
```

```
estRUL = duration
   93.671 hr
```

The estimated RUL for the component is around 94 hours.

Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To evaluate these models in parallel, set the `UseParallel` property to `true`.

See Also

Functions
`hashSimilarityModel` | `residualSimilarityModel` | `fit` | `predictRUL`

Topics
"Update RUL Prediction as Data Arrives"
"RUL Estimation Using RUL Estimator Models"

Introduced in R2018a

reliabilitySurvivalModel

Probabilistic failure-time model for estimating remaining useful life

Description

Use `reliabilitySurvivalModel` to estimate the remaining useful life (RUL) of a component using a probability distribution of component failure times. Reliability survival models are useful when the only data you have are the failure times for an ensemble of similar components, such as multiple machines manufactured to the same specifications.

To configure a `reliabilitySurvivalModel` object for a specific type of component, use `fit`, which estimates the probability distribution coefficients from a collection of failure-time data. Once you configure the parameters of your reliability survival model, you can then predict the remaining useful life of similar components using `predictRUL`. For a basic example illustrating RUL prediction, see "Update RUL Prediction as Data Arrives".

For general information on predicting remaining useful life, see "Models for Predicting Remaining Useful Life".

Creation

Syntax

```
mdl = reliabilitySurvivalModel
mdl = reliabilitySurvivalModel(distribution)
mdl = reliabilitySurvivalModel(initModel)
mdl = reliabilitySurvivalModel(___,Name,Value)
```

Description

`mdl = reliabilitySurvivalModel` creates a reliability survival model for estimating RUL model that uses a Weibull distribution and initializes the model with default settings.

`mdl = reliabilitySurvivalModel(distribution)` creates a reliability survival model that uses the specified probability distribution function and sets the `Distribution` property of the model.

`mdl = reliabilitySurvivalModel(initModel)` creates a reliability survival model and initializes the model parameters using an existing `reliabilitySurvivalModel` object `initModel`.

`mdl = reliabilitySurvivalModel(___,Name,Value)` specifies user-settable model properties using name-value pairs. For example, `reliabilitySurvivalModel('LifeTimeUnit',"days")` creates a reliability survival model that uses days as a lifetime unit. You can specify multiple name-value pairs. Enclose each property name in quotes.

Input Arguments

`initModel` — Reliability survival model
`reliabilitySurvivalModel` object

Reliability survival model, specified as a `reliabilitySurvivalModel` object.

Properties

`Distribution` — Probability distribution function
`"Weibull"` (default) | `"Normal"` | `"Poisson"` | `"Kernel"` | `"Rayleigh"` | `"Gamma"` | ...

Probability distribution function used to model the lifetime distribution, specified as one of the following:

Distribution String	Distribution Object
"BirnbaumSaunders"	BirnbaumSaundersDistribution
"Exponential"	ExponentialDistribution
"Gamma"	GammaDistribution
"GeneralizedPareto"	GeneralizedParetoDistribution
"HalfNormal"	HalfNormalDistribution
"InverseGaussian"	InverseGaussianDistribution
"Kernel"	KernelDistribution
"Logistic"	LogisticDistribution
"Loglogistic"	LoglogisticDistribution
"Lognormal"	LognormalDistribution
"Nakagami"	NakagamiDistribution
"Normal"	NormalDistribution
"Poisson"	PoissonDistribution
"Rayleigh"	RayleighDistribution
"Stable"	StableDistribution
"Weibull"	WeibullDistribution

To configure the parameters of the probability distribution function, use the `fit` function.

`ParameterValues` — Distribution coefficients
vector

This property is read-only.

Distribution coefficients estimated by the `fit` function, specified as a vector. For more information on the coefficients of each distribution function, see the corresponding distribution object listed in `Distribution`. For more information on model fitting, see `fitdist`.

`ParameterCovariance` — Covariance of the distribution coefficients
array

This property is read-only.

Covariance of the distribution coefficients estimated by the `fit` function, specified as a positive array with size equal to the number of coefficients. For more information on the coefficients of each distribution function, see the corresponding distribution object listed in `Distribution`.

`ParameterNames` — Distribution coefficient names

string array

This property is read-only.

Distribution coefficient names assigned when the model is trained using the `fit` function, specified as string array. For more information on the coefficients of each distribution function, see the corresponding distribution object listed in `Distribution`.

`CensorVariable` — Censor variable

`""` (default) | string

Censor variable, specified as a string that contains a valid MATLAB variable name. The censor variable is a binary variable that indicates which life-time measurements in `data` are not end-of-life values.

`CensorVariable` must not match any of the strings in `DataVariables` or `LifeTimeVariable`.

You can specify `CensorVariable`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function
- Using dot notation after model creation

`LifeTimeVariable` — Lifetime variable

`""` (default) | string

Lifetime variable, specified as a string that contains a valid MATLAB variable name. For survival models, the lifetime variable contains the historical life span measurements of components.

You can specify `LifeTimeVariable`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function
- Manually using dot notation

`LifeTimeUnit` — Lifetime variable units

`""` (default) | string

Lifetime variable units, specified as a string.

The units of the lifetime variable do not need to be time-based. The life of the test component can be measured in terms of a usage variable, such as distance traveled (miles) or fuel consumed (gallons).

`DataVariables` — Data variables

`""` (default)

Data variables, specified as an empty string. This property is ignored for reliability survival models.

`UserData` — Additional model information

`[]` (default) | any data type or format

Additional model information for bookkeeping purposes, specified as any data type or format. The model does not use this information.

You can specify `UserData`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

Object Functions

predictRUL	Estimate remaining useful life for a test component
fit	Estimate parameters of remaining useful life model using historical data

Examples

Train Reliability Survival Model

Load training data.

```
load('reliabilityData.mat')
```

This data is a column vector of `duration` objects representing battery discharge times.

Create a reliability survival model with default settings.

```
mdl = reliabilitySurvivalModel;
```

Train the survival model using the training data.

```
fit(mdl,reliabilityData,"hours")
```

Predict RUL Using Reliability Survival Model and View PDF

Load training data.

```
load('reliabilityData.mat')
```

This data is a column vector of `duration` objects representing battery discharge times.

Create a reliability survival model, specifying the life time variable and life time units.

```
mdl = reliabilitySurvivalModel('LifeTimeVariable',"DischargeTime",'LifeTimeUnit',"hours");
```

Train the survival model using the training data.

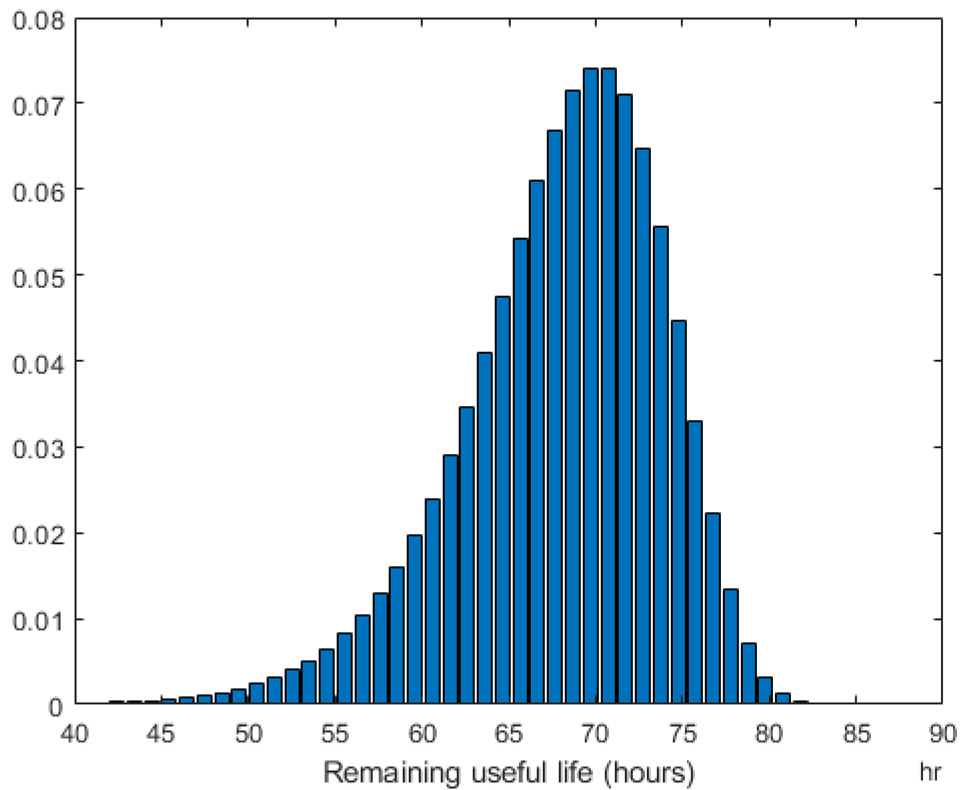
```
fit(mdl,reliabilityData)
```

Predict the life span of a new component, and obtain the probability distribution function for the estimate.

```
[estRUL,ciRUL,pdfRUL] = predictRUL(mdl);
```

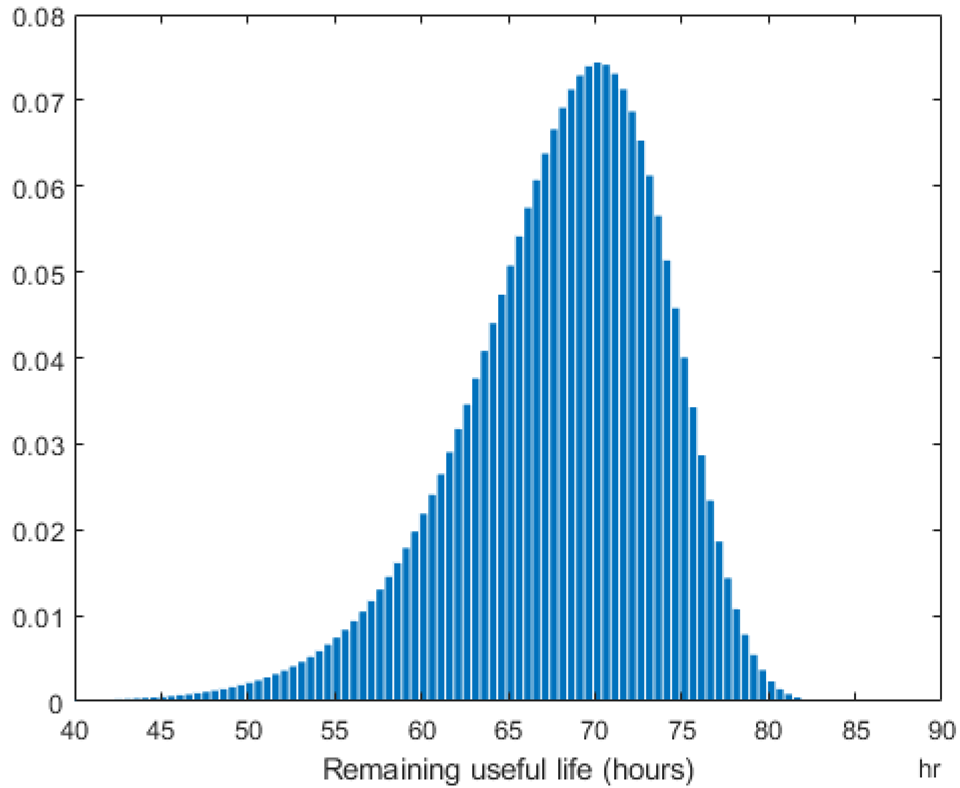
Plot the probability distribution.

```
bar(pdfRUL.RUL,pdfRUL.ProbabilityDensity)
xlabel('Remaining useful life (hours)')
xlim(hours([40 90]))
```



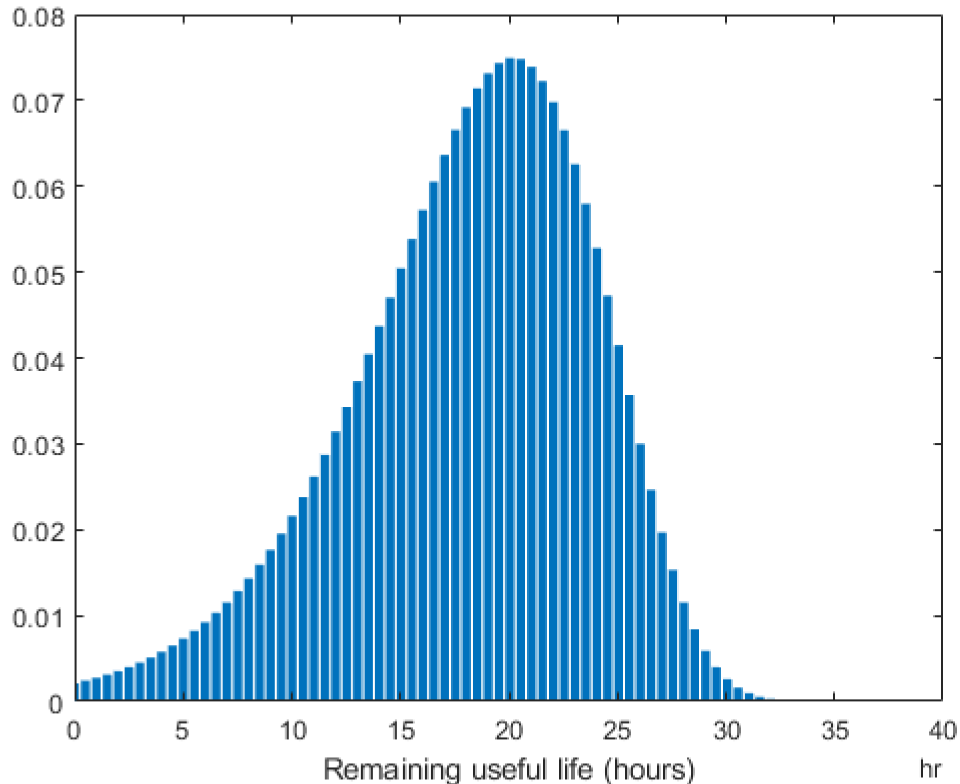
Improve the distribution view by providing the number of bins and bin size for the prediction.

```
[estRUL,ciRUL,pdfRUL] = predictRUL(mdl,'BinSize',0.5,'NumBins',500);
bar(pdfRUL.RUL,pdfRUL.ProbabilityDensity)
xlabel('Remaining useful life (hours)')
xlim(hours([40 90]))
```



Predict the RUL for a component that has been operating for 50 hours.

```
[estRUL,ciRUL,pdfRUL] = predictRUL(mdl,hours(50),'BinSize',0.5,'NumBins',500);
bar(pdfRUL.RUL,pdfRUL.ProbabilityDensity)
xlabel('Remaining useful life (hours)')
xlim(hours([0 40]))
```



Extended Capabilities

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

Usage notes and limitations:

- The `predictRUL` command supports code generation with MATLAB Coder for this RUL model type.
- Before generating code that uses this model, you must save the model using `saveRULModelForCoder`. For an example, see "Generate Code for Predicting Remaining Useful Life".
- For a `covariateSurvivalModel`, you cannot generate code for `predictRUL` using covariate data specified as string or character-vector values in `table` or `timetable` form. Instead, convert the string or character-vector data to a different data type, or delete the data from the table.
- In addition to the read-only properties, the following properties of survival models cannot be changed at run time:
  - `LifeTimeVariable`
  - `LifeTimeUnit`
  - `DataVariables`
  - `CensorVariable`

- `EncodedVariables` (`covariateSurvivalModel` only)
- `EncodingMethod` (`covariateSurvivalModel` only)

See Also

Functions
`covariateSurvivalModel` | `fit` | `predictRUL`

Topics
"Update RUL Prediction as Data Arrives"
"RUL Estimation Using RUL Estimator Models"

Introduced in R2018a

residualSimilarityModel

Residual comparison-based similarity model for estimating remaining useful life

Description

Use `residualSimilarityModel` to estimate the remaining useful life (RUL) of a component using a residual comparison-based similarity model. This model is useful when you have degradation profiles for an ensemble of similar components, such as multiple machines manufactured to the same specifications, and you know the dynamics of the degradation process. The historical data for each member of the data ensemble is fitted with a model of identical structure. The degradation data of the test component is used to compute 1-step prediction errors, or residuals, for each ensemble model. The magnitudes of these errors indicate how similar the test component is to the corresponding ensemble members.

To configure a `residualSimilarityModel` object, use `fit`, which trains and stores the degradation model for each data ensemble member. Once you configure the parameters of your similarity model, you can then predict the remaining useful life of similar components using `predictRUL`. For similarity models, the RUL of the test component is estimated as the median statistic of the lifetime span of the most similar components minus the current lifetime value of the test component. For a basic example illustrating RUL prediction, see “Update RUL Prediction as Data Arrives”.

For general information on predicting remaining useful life, see “Models for Predicting Remaining Useful Life”.

Creation

Syntax

```
mdl = residualSimilarityModel
mdl = residualSimilarityModel(initModel)
mdl = residualSimilarityModel( ___ ,Name,Value)
```

Description

`mdl = residualSimilarityModel` creates a residual comparison-based similarity model for estimating RUL and initializes the model with default settings.

`mdl = residualSimilarityModel(initModel)` creates a residual comparison-based similarity model and initializes the model parameters using an existing `residualSimilarityModel` object `initModel`.

`mdl = residualSimilarityModel( ___ ,Name,Value)` specifies user-settable model properties using name-value pairs. For example, `residualSimilarityModel('LifeTimeUnit',"days")` creates a residual comparison-based similarity model that uses days as a lifetime unit. You can specify multiple name-value pairs. Enclose each property name in quotes.

Input Arguments

`initModel` — Residual comparison-based similarity model
`residualSimilarityModel` object

Residual comparison-based similarity model, specified as a `residualSimilarityModel` object.

Properties

`Method` — Type of model
`"arma2"` (default) | `"linear"` | `"arima2"` | `"poly2"` | `"exp1"` | ...

Type of model trained using the `fit` function and used for residual generation, specified as one of the following:

- `"linear"` — Line with offset term
- `"poly2"` — Second-order polynomial
- `"poly3"` — Third-order polynomial
- `"exp1"` — Exponential with offset term
- `"exp2"` — Sum of two exponentials
- `"arma2"` — Second-order ARMA model
- `"arma3"` — Third-order ARMA model
- `"arima2"` — Second-order ARMA model with noise integration
- `"arima3"` — Third-order ARMA model with noise integration

Select the model type based on your knowledge of the dynamics of the component degradation process.

You can specify `Method`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

For more information on estimating ARMA and polynomial models, see `armax` and `polyfit`, respectively.

`Distance` — Distance computation method
`"euclidian"` (default) | `"absolute"` | function handle

Distance computation method, specified as one of the following:

- `"euclidian"` — Use the 2-norm of the residual signal.
- `"absolute"` — Use the 1-norm of the residual signal.
- Function handle — Use a custom function of the form:

  ```
  D = distanceFunction(r)
  ```

  where,

  - `r` is the residual, specified as a column vector.

- `D` is the distance, returned as nonnegative scalar.

You can specify `Distance`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`Models` — Parameters of the fitted models

cell array

This property is read-only.

Parameters of the fitted models for each member of the training data ensemble, specified as a cell array and assigned by the `fit` function. The content of `Models` depends on the type of model used for regression, as specified in `Method`.

Method	Model Structure	Models Cell Content
"linear"	$at + b$	Row vector — [a b]
"poly2"	$at^2 + bt + c$	Row vector — [a b c]
"poly3"	$at^3 + bt^2 + ct + d$	Row vector — [a b c d]
"exp1"	$ae^{bt}+c$	Row vector — [a b c]
"exp2"	$ae^{bt}+ce^{dt}$	Row vector — [a b c d]
"arma2"	Second-order ARMA model: $A(q)S(t) = C(q)e(t)$ where • $A(q) = [1\ a_1q^{-1}\ a_2q^{-2}]$ • $C(q) = [1\ c_1q^{-1}]$ • $S(t)$ is the degradation feature	Structure with fields: • A — Row vector [1 a_1 a_2] • C — Row vector [1 c_1]
"arma3"	Similar to "arma2", but with $A(q)$ third-order and $C(q)$ second-order	Structure with fields: • A — Row vector [1 a_1 a_2 a_2] • C — Row vector [1 c_1 c_2]
"arima2"	Similar to "arma2", but with an additional noise integrator: $A(q)S(t) = \frac{C(q)}{1 - q^{-1}}e(t)$	Structure with fields: • A — Row vector [1 a_1 a_2] • C — Row vector [1 c_1]
"arima3"	Similar to "arma3", but with an additional noise integrator	Structure with fields: • A — Row vector [1 a_1 a_2 a_2] • C — Row vector [1 c_1 c_2]

For more information on estimating ARMA and polynomial models, see `armax` and `polyfit`, respectively.

`ModelMSE` — Mean squared error of the estimation for each model

vector

This property is read-only.

Mean squared error of the estimation for each model in `Models`, specified as a vector and assigned by the `fit` function.

`LifeSpan` — Ensemble member life spans
double vector (default) | vector of `duration` objects

This property is read-only.

Ensemble member life spans, specified as a double vector or `duration` object vector and computed from the ensemble member degradation profiles by the `fit` function.

`NumNearestNeighbors` — Number of nearest neighbors for RUL estimation
`Inf` (default) | finite positive integer

Number of nearest neighbors for RUL estimation, specified as `Inf` or a finite positive integer. If `NumNearestNeighbors` is `Inf`, then `predictRUL` uses all the ensemble members during estimation.

You can specify `NumNearestNeighbors`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`IncludeTies` — Flag to include ties
`true` (default) | `false`

Flag to include ties, specified as `true` or `false`. When `IncludeTies` is `true`, the model includes all neighbors whose distance to the test component data is less than the *K*th smallest distance, where *K* is equal to `NumNearestNeigbors`.

You can specify `IncludeTies`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`Standardize` — Flag for standardizing residuals
`false` (default) | `true`

Flag for standardizing residuals before computing distance, specified as `true` or `false`.

When `Standardize` is `true`, the residuals are scaled by the inverse square root of the estimated mean squared errors in `ModelMSE`.

You can specify `Standardize`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`LifeTimeVariable` — Lifetime variable
`""` (default) | string

Lifetime variable, specified as a string that contains a valid MATLAB variable name or `""`.

When you train the model using the `fit` function, if your training data is a:

- `table`, then `LifeTimeVariable` must match one of the variable names in the table
- `timetable`, then `LifeTimeVariable` one of the variable names in the table or the dimension name of the time variable, `data.Properties.DimensionNames{1}`

You can specify `LifeTimeVariable`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function
- Using dot notation after model creation

`LifeTimeUnit` — Lifetime variable units
`""` (default) | string

Lifetime variable units, specified as a string.

The units of the lifetime variable do not need to be time-based. The life of the test component can be measured in terms of a usage variable, such as distance traveled (miles) or fuel consumed (gallons).

`DataVariables` — Degradation variable names
`""` (default) | string | string array

Degradation variable names, specified as a string or string array. The strings in `DataVariables` must be valid MATLAB variable names.

You can specify `DataVariables`:

- Using a name-value pair when you create the model
- As an argument when you call the `fit` function
- Using dot notation after model creation

`UseParallel` — Flag for using parallel computing
`false` (default) | `true`

Flag for using parallel computing for nearest-neighbor searching, specified as either `true` or `false`.

You can specify `UseParallel`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

`UserData` — Additional model information
`[]` (default) | any data type or format

Additional model information for bookkeeping purposes, specified as any data type or format. The model does not use this information.

You can specify `UserData`:

- Using a name-value pair when you create the model
- Using dot notation after model creation

Object Functions

predictRUL	Estimate remaining useful life for a test component
fit	Estimate parameters of remaining useful life model using historical data
compare	Compare test data to historical data ensemble for similarity models

Examples

Train Residual Similarity Model

Load training data.

```
load('residualTrainVectors.mat')
```

The training data is a cell array of column vectors. Each column vector is a degradation feature profile for a component.

Create a residual similarity model with default settings.

```
mdl = residualSimilarityModel;
```

Train the similarity model using the training data.

```
fit(mdl,residualTrainVectors)
```

Train Residual Similarity Model Using Tabular Data

Load training data.

```
load('residualTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component. Each profile consists of life time measurements in the `"Time"` variable and corresponding degradation feature measurements in the `"Condition"` variable.

Create a residual similarity model that fits the data with a third-order ARMA model and uses an absolute distance metric.

```
mdl = residualSimilarityModel('Method',"arma3",'Distance',"absolute");
```

Train the similarity model using the training data. Specify the names of the life time and data variables.

```
fit(mdl,residualTrainTables,"Time","Condition")
```

Predict RUL Using Residual Similarity Model

Load training data.

```
load('residualTrainTables.mat')
```

The training data is a cell array of tables. Each table is a degradation feature profile for a component. Each profile consists of life time measurements in the "Time" variable and corresponding degradation feature measurements in the "Condition" variable.

Create a residual similarity model that fits the data with a third-order ARMA model and uses hours as the life time unit.

```
mdl = residualSimilarityModel('Method',"arma3",'LifeTimeUnit',"hours");
```

Train the similarity model using the training data. Specify the names of the life time and data variables.

```
fit(mdl,residualTrainTables,"Time","Condition")
```

Load testing data. The test data contains the degradation feature measurements for a test component up to the current life time.

```
load('residualTestData.mat')
```

Predict the RUL of the test component using the trained similarity model.

```
estRUL = predictRUL(mdl,residualTestData)
```

```
estRUL = duration
   85.73 hr
```

The estimated RUL for the component is around 86 hours.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To evaluate these models in parallel, set the UseParallel property to true.

See Also

Functions

pairwiseSimilarityModel | hashSimilarityModel | fit | predictRUL

Topics

"Update RUL Prediction as Data Arrives"
"RUL Estimation Using RUL Estimator Models"

Introduced in R2018a

simulationEnsembleDatastore

Manage ensemble data generated by `generateSimulationEnsemble` or by logging simulation data in Simulink

Description

A `simulationEnsembleDatastore` object is a datastore specialized for use in developing algorithms for condition monitoring and predictive maintenance using simulated data.

This object specifies the data variables, independent variables, and condition variables stored in a collection of MATLAB data files (MAT-files). The data files contain `Simulink.SimulationData.Dataset` variables that are the result of logging data during Simulink model simulation.

For a detailed example illustrating the use of a simulated ensemble datastore, see “Generate and Use Simulated Data Ensemble”. For general information about data ensembles in Predictive Maintenance Toolbox, see “Data Ensembles for Condition Monitoring and Predictive Maintenance”.

Creation

To create a `simulationEnsembleDatastore` object:

1. Generate and log simulation data from a Simulink model. You can do so using `generateSimulationEnsemble` or any other means of logging simulation to disk.
2. Create a `simulationEnsembleDatastore` object that points to the generated simulation data using the `simulationEnsembleDatastore` command (described below).

If you have simulation data previously generated with `generateSimulationEnsemble` or other means, you can use the creation function `simulationEnsembleDatastore` to create a new simulation ensemble datastore object at any time.

Syntax

```
ensemble = simulationEnsembleDatastore(location)
ensemble = simulationEnsembleDatastore(location,signallog)
ensemble = simulationEnsembleDatastore(location,signallog,Name,Value)
```

Description

`ensemble = simulationEnsembleDatastore(location)` creates a simulation ensemble from data previously generated using `generateSimulationEnsemble` in the folder specified by `location`. The function identifies ensemble variables in the generated data from information stored in the generated MAT-files. The function populates the `DataVariables` and `SelectedVariables` properties of `ensemble` with the names of these ensemble variables.

`ensemble = simulationEnsembleDatastore(location,signallog)` uses `signallog` to determine which variable in the MAT-files contains logged signals. Use the variable name specified in

the `Signal logging` configuration parameter of the Simulink model from which the data is generated. Specifying this variable allows the ensemble to treat those signals as ensemble data variables, rather than the `signallog` variable itself. The other variables in the MAT-file are also returned as ensemble data variables.

`ensemble = simulationEnsembleDatastore(location,signallog,Name,Value)` specifies additional properties on page 2-69 of the object using one or more name-value pair arguments. For example, using `'IndependentVariables',["Age";"ID"]` specifies the independent variables when you create the object.

Input Arguments

`location` — File path
string | character vector

File path to the location in which to store simulation data, specified as a string or a character vector. The file path can be any location supported by MATLAB datastores, including an IRI path pointing to a remote location. However, when you use a `simulationEnsembleDatastore` to manage remote data, you cannot use `writeToLastMemberRead` to add data to the ensemble datastore. For more information about working with remote data in MATLAB, see “Work with Remote Data”

Example: `pwd + "\simResults"`

`signallog` — Variable name of logged signals
string | character vector

Variable name of logged signals, specified as a string or a character vector. This input argument tells `simulationEnsembleDatastore` which data variable in the stored MAT-files contains the logged simulation data. This variable name is specified in the `Signal logging` configuration parameter of the Simulink model from which the data is generated. When you use `generateSimulationEnsemble` to generate simulation data for the ensemble, each generated MAT-file contains a variable, `PMSignalLogName`, specifying the variable name of the logged signals.

Example: `"logsout"`

Properties

`DataVariables` — Data variables in the ensemble
string array of logged signal names (default) | string array

Data variables in the ensemble, specified as a string array. Data variables are the main content of the members of an ensemble. Data variables can include measured data or derived data for analysis and development of predictive maintenance algorithms. For example, your data variables might include measured or simulated vibration signals and derived values such as mean vibration value or peak vibration frequency. In practice, your data variables, independent variables, and condition variables are all distinct sets of variables.

`simulationEnsembleDatastore` sets the initial value of `DataVariables` to the names of all the logged signals in the data generated with `generateSimulationEnsemble`. `simulationEnsembleDatastore` also adds the variables `SimulationInput` and `SimulationMetadata` to `DataVariables`. These variables contain information about how the simulation was performed.

You can also specify `DataVariables` using a cell array of character vectors, such as `{'Vibration';'Tacho'}`, but the variable names are always stored as a string array,

["Vibration";"Tacho"]. If you specify a matrix of variable names, the matrix is flattened to a column vector.

IndependentVariables — Independent variables in the ensemble

[ ] (default) | string array

Independent variables in the ensemble, specified as a string array. You typically use independent variables to order the members of an ensemble. Examples are timestamps, number of operating hours, or miles driven. Set this property to the names of such variables in your ensemble. In practice, your data variables, independent variables, and condition variables are all distinct sets of variables.

You can also specify IndependentVariables using a cell array of character vectors, such as {'Time';'Age'}, but the variable names are always stored as a string array, ["Time";"Age"]. If you specify a matrix of variable names, the matrix is flattened to a column vector.

ConditionVariables — Condition variables in the ensemble

[ ] (default) | string array

Condition variables in the ensemble, specified as a string array. Use condition variables to label the members in a ensemble according to the fault condition or other operating condition under which the ensemble member was collected. In practice, your data variables, independent variables, and condition variables are all distinct sets of variables.

You can also specify ConditionVariables using a cell array of character vectors, such as {'GearFault';'Temperature'}, but the variable names are always stored as a string array, ["GearFault";"Temperature"]. If you specify a matrix of variable names, the matrix is flattened to a column vector.

SelectedVariables — Variables to read

string array of logged signal names (default) | string array

Variables to read from the ensemble, specified as a string array. Use this property to specify which variables are extracted to the MATLAB workspace when you use the read command to read data from the ensemble. read returns a table row containing a table variable for each name specified in SelectedVariables. For example, suppose that you have an ensemble, ensemble, that contains six variables, and you want to read only two of them, Vibration and FaultState. Set the SelectedVariables property and call read.

```
ensemble.SelectedVariables = ["Vibration";"FaultState"];
data = read(ensemble)
```

SelectedVariables can be any combination of the variables in the DataVariables, ConditionVariables, and IndependentVariables properties. If SelectedVariables is empty, read generates an error.

simulationEnsembleDatastore sets the initial value of SelectedVariables to the names of all the logged signals in the data generated generateSimulationEnsemble.

You can specify SelectedVariables using a cell array of character vectors, such as {'Vibration';'Tacho'}, but the variable names are always stored as a string array, ["Vibration";"Tacho"]. If you specify a matrix of variable names, the matrix is flattened to a column vector.

ReadSize — Number of members to read

1 (default) | positive integer

Number of members to read from the ensemble datastore at once, specified as a positive integer that is smaller than the total number of members in the ensemble. By default, the `read` command returns a one-row table containing data from one ensemble member. To read data from multiple members in a single `read` operation, set this property to an integer value greater than one. For example, if `ReadSize` = 3, then `read` returns a three-row table where each row contains data from a different ensemble member. If fewer than `ReadSize` members are unread, then `read` returns a table with as many rows as there are remaining members.

The ensemble datastore property `LastMemberRead` contains the names of all files read during the most recent `read` operation. Thus, for instance, if `ReadSize` = 3, then a `read` operation sets `LastMemberRead` to a string vector containing three file names.

When you use `writeToLastMemberRead`, specify the data to write as a table with a number of rows equal to `ReadSize`. The `writeToLastMemberRead` command updates the members specified by `LastMemberRead`, writing one table row to each specified file.

Changing the `ReadSize` property also resets the ensemble to its unread state. For instance, suppose that you read some ensemble members one at a time (`ReadSize` = 1), and then change `ReadSize` to 3. The next `read` operation returns data from the first three ensemble members.

`NumMembers` — Number of members in ensemble
positive integer

This property is read-only.

Number of members in the ensemble, specified as a positive integer.

`LastMemberRead` — File name of last ensemble member read
`""` (default) | string | string array

This property is read-only.

File name of last ensemble member read into the MATLAB workspace, specified as a string. When you use the `read` command to read data from an ensemble datastore, the software determines which ensemble member to read next, and reads data from the corresponding file. The `LastMemberRead` property contains the path to the most recently read file. When the ensemble datastore has not yet been read, or has been reset, `LastMemberRead` is an empty string.

When you call `writeToLastMemberRead` to add data back to the ensemble datastore, that function writes to the file specified in `LastMemberRead`.

By default, `read` reads data from one ensemble member at a time (the `ReadSize` property of the ensemble datastore is 1). When `ReadSize` > 1, `LastMemberRead` is a string array containing the paths to all files read in the most recent `read` operation.

`Files` — List of files in ensemble datastore
string vector

This property is read-only.

List of files in the ensemble datastore, specified as a column string vector of length `NumMembers`. Each entry contains the full path to a file in the datastore. The files are in the order in which the `read` command reads ensemble members.

Example: `["C:\Data\Data_01.csv"; "C:\Data\Data_02.csv"; "C:\Data\Data_03.csv"]`

Object Functions

The `read` and `writeToLastMemberRead` functions are specialized for Predictive Maintenance Toolbox ensemble data. Other functions, such as `reset` and `hasdata`, are identical to those used with `datastore` objects in MATLAB. To extract specific ensemble members into a smaller or more specialized ensemble datastore, use `subset`. To transfer all the member data into a table or cell array with a single command, use `readall`. To partition an ensemble datastore, use the `partition(ds,n,index)` syntax of the `partition` function.

read	Read member data from an ensemble datastore
writeToLastMemberRead	Write data to member of an ensemble datastore
subset	Create new ensemble datastore from subset of existing ensemble datastore
reset	Reset datastore to initial state
hasdata	Determine if data is available to read
progress	Determine how much data has been read
readall	Read all data in datastore
numpartitions	Number of datastore partitions
partition	Partition a datastore
tall	Create tall array
isPartitionable	Determine whether datastore is partitionable
isShuffleable	Determine whether datastore is shuffleable

Examples

Generate Ensemble of Fault Data

Generate a simulation ensemble datastore of data representing a machine operating under fault conditions by simulating a Simulink® model of the machine while varying a fault parameter.

Load the Simulink model. This model is a simplified version of the gear-box model described in “Using Simulink to Generate Fault Data”. For this example, only one fault mode is modeled, a gear-tooth fault.

```
mdl = 'TransmissionCasingSimplified';
open_system(mdl)
```

The gear-tooth fault is modeled as a disturbance in the `Gear Tooth fault` subsystem. The magnitude of the disturbance is controlled by the model variable `ToothFaultGain`, where `ToothFaultGain = 0` corresponds to no gear-tooth fault (healthy operation). To generate the ensemble of fault data, you use `generateSimulationEnsemble` to simulate the model at different values of `ToothFaultGain`, ranging from -2 to zero. This function uses an array of `Simulink.SimulationInput` objects to configure the Simulink model for each member in the ensemble. Each simulation generates a separate member of the ensemble in its own data file. Create such an array, and use `setVariable` to assign a tooth-fault gain value for each run.

```
toothFaultValues  = -2:0.5:0; % 5 ToothFaultGain values

for ct = numel(toothFaultValues):-1:1
    simin(ct) = Simulink.SimulationInput(mdl);
    simin(ct) = setVariable(simin(ct),'ToothFaultGain',toothFaultValues(ct));
end
```

For this example, the model is already configured to log certain signal values, `Vibration` and `Tacho` (see “Export Signal Data Using Signal Logging” (Simulink)). `generateSimulationEnsemble` further configures the model to:

- Save logged data to files in the folder you specify.
- Use the `timetable` format for signal logging.
- Store each `Simulink.SimulationInput` object in the saved file with the corresponding logged data.

Specify a location for the generated data. For this example, save the data to a folder called `Data` within your current folder. The indicator `status` is 1 (true) if all the simulations complete without error.

```
mkdir Data
location = fullfile(pwd,'Data');
[status,E] = generateSimulationEnsemble(simin,location);
```

```
[01-Sep-2021 12:59:27] Running simulations...
[01-Sep-2021 12:59:59] Completed 1 of 5 simulation runs
[01-Sep-2021 13:00:29] Completed 2 of 5 simulation runs
[01-Sep-2021 13:00:54] Completed 3 of 5 simulation runs
[01-Sep-2021 13:01:27] Completed 4 of 5 simulation runs
[01-Sep-2021 13:01:44] Completed 5 of 5 simulation runs
```

Inside the `Data` folder, examine one of the files. Each file is a MAT-file containing the following MATLAB® variables:

- `SimulationInput` — The `Simulink.SimulationInput` object that was used to configure the model for generating the data in the file. You can use this to extract information about the conditions (such as faulty or healthy) under which this simulation was run.
- `logsout` — A `Dataset` object containing all the data that the Simulink model is configured to log.
- `PMSignalLogName` — The name of the variable that contains the logged data (`'logsout'` in this example). The `simulationEnsembleDatastore` command uses this name to parse the data in the file.
- `SimulationMetadata` — Other information about the simulation that generated the data logged in the file.

Now you can create the simulation ensemble datastore using the generated data. The resulting `simulationEnsembleDatastore` object points to the generated data. The object lists the data variables in the ensemble, and by default all the variables are selected for reading. Examine the `DataVariables` and `SelectedVariables` properties of the ensemble to confirm these designations.

```
ensemble = simulationEnsembleDatastore(location)
```

```
ensemble = 
  simulationEnsembleDatastore with properties:

           DataVariables: [4x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: [0x0 string]
       SelectedVariables: [4x1 string]
                ReadSize: 1
              NumMembers: 5
```

```
          LastMemberRead: [0x0 string]
                   Files: [5x1 string]
```

```
ensemble.DataVariables
```

```
ans = 4x1 string
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
```

```
ensemble.SelectedVariables
```

```
ans = 4x1 string
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
```

You can now use `ensemble` to read and analyze the generated data in the ensemble datastore. See `simulationEnsembleDatastore` for more information.

Extract Subset of Stored Variables from Ensemble Member

In general, you use the `read` command to extract data from a `simulationEnsembleDatastore` object into the MATLAB® workspace. Often, your ensemble contains more variables than you need to use for a particular analysis. Use the `SelectedVariables` property of the `simulationEnsembleDatastore` object to select a subset of variables for reading.

For this example, use the following code to create a `simulationEnsembleDatastore` object using data previously generated by running a Simulink® model at a various fault values (See `generateSimulationEnsemble`.). The ensemble includes simulation data for five different values of a model parameter, `ToothFaultGain`. Because of the volume of data, the `unzip` operation takes a few minutes.

```
unzip simEnsData.zip  % extract compressed files
ensemble = simulationEnsembleDatastore(pwd,'logsout')
```

```
ensemble = 
  simulationEnsembleDatastore with properties:

           DataVariables: [5x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: [0x0 string]
       SelectedVariables: [5x1 string]
                ReadSize: 1
              NumMembers: 5
          LastMemberRead: [0x0 string]
                   Files: [5x1 string]
```

The model that generated the data, `TransmissionCasingSimplified`, was configured such that the resulting ensemble contains variables including accelerometer data, `Vibration`, and tachometer

data, Tacho. By default, the simulationEnsembleDatastore object designates all these variables as both data variables and selected variables, as shown in the DataVariables and SelectedVariables properties.

```
ensemble.DataVariables
```

```
ans = 5x1 string
    "PMSignalLogName"
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
```

```
ensemble.SelectedVariables
```

```
ans = 5x1 string
    "PMSignalLogName"
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
```

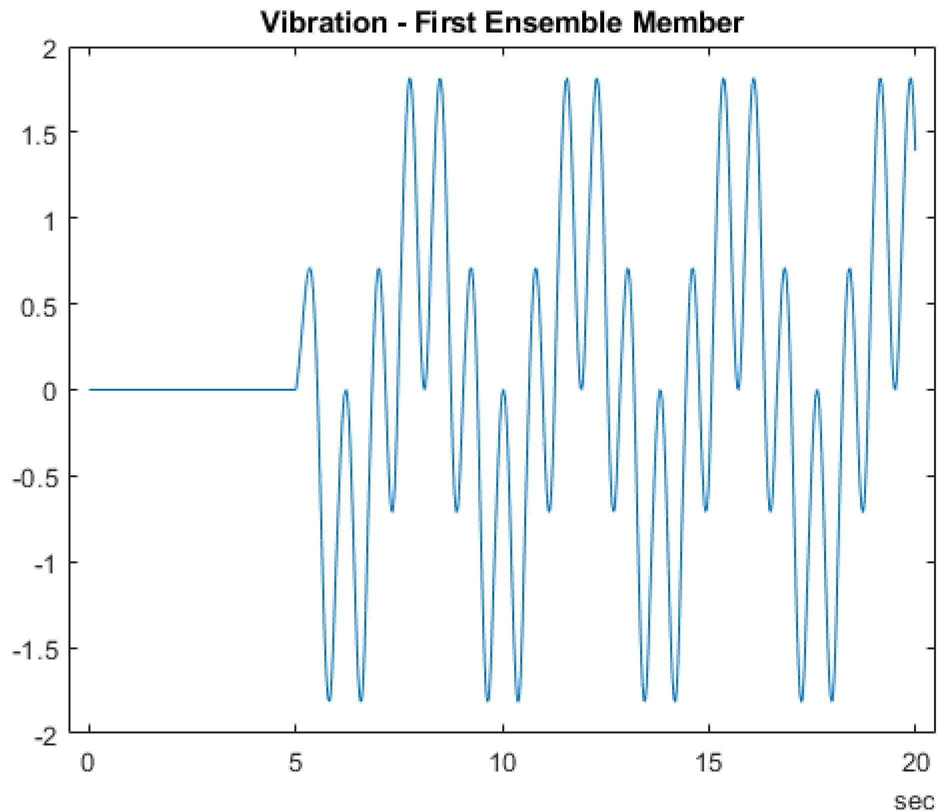
Suppose that for the analysis you want to do, you need only the Vibration data and the Simulink.SimulationInput object that describes the conditions under which this member data was simulated. Set ensemble.SelectedVariables to specify the variables you want to read. The read command then extracts those variables from the current ensemble member.

```
ensemble.SelectedVariables = ["Vibration";"SimulationInput"];
data1 = read(ensemble)
```

```
data1=1×2 table
         Vibration                 SimulationInput
    ___________________    ______________________________

    {20202x1 timetable}    {1x1 Simulink.SimulationInput}
```

data.Vibration is a cell array containing one timetable that stores the simulation times and the corresponding vibration signal. You can now process this data as needed. For instance, extract the vibration data from the table and plot it.

```
vibdata1 = data1.Vibration{1};
plot(vibdata1.Time,vibdata1.Data)
title('Vibration - First Ensemble Member')
```



The next time you call `read` on this ensemble, the last-read member designation advances to the next member of the ensemble (see "Data Ensembles for Condition Monitoring and Predictive Maintenance"). Read the selected variables from the next member of the ensemble.

```
data2 = read(ensemble)
```

```
data2=1×2 table
        Vibration                  SimulationInput
    __________________    ______________________________

    {20215x1 timetable}    {1x1 Simulink.SimulationInput}
```

To confirm that `data1` and `data2` contain data from different ensemble members, examine the values of the varied model parameter, `ToothFaultGain`. For each ensemble, this value is stored in the `Variables` field of the `SimulationInput` variable.

```
data1.SimulationInput{1}.Variables
```

```
ans =
  Variable with properties:

           Name: 'ToothFaultGain'
          Value: -2
      Workspace: 'global-workspace'
    Description: ""
```

```
data2.SimulationInput{1}.Variables
```

```
ans = 
  Variable with properties:

           Name: 'ToothFaultGain'
          Value: -1.5000
      Workspace: 'global-workspace'
    Description: ""
```

This result confirms that `data1` is from the ensemble member with `ToothFaultGain` = –2, and `data2` is from the member with `ToothFaultGain` = –1.5.

Append Derived Data to Ensemble Members

You can process data in an ensemble datastore and add derived variables to the ensemble members. For this example, process a variable value to compute a label that indicates whether the ensemble member contains data obtained with a fault present. You then add that label to the ensemble.

For this example, use the following code to create a `simulationEnsembleDatastore` object using data previously generated by running a Simulink® model at a various fault values. (See `generateSimulationEnsemble`.) The ensemble includes simulation data for five different values of a model parameter, `ToothFaultGain`. The model was configured to log the simulation data to a variable named `logsout` in the MAT-files that are stored for this example in `simEnsData.zip`. Because of the volume of data, the `unzip` operation might take a minute or two.

```
unzip simEnsData.zip  % extract compressed files
ensemble = simulationEnsembleDatastore(pwd,'logsout')
```

```
ensemble = 
  simulationEnsembleDatastore with properties:

           DataVariables: [5x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: [0x0 string]
       SelectedVariables: [5x1 string]
                ReadSize: 1
              NumMembers: 5
          LastMemberRead: [0x0 string]
                   Files: [5x1 string]
```

Read the data from the first member in the ensemble. The software determines which ensemble is the first member, and updates the property `ensemble.LastMemberRead` to reflect the name of the corresponding file.

```
data = read(ensemble)
```

```
data=1×5 table
    PMSignalLogName          SimulationInput                      SimulationMetadata              
    _______________    ______________________________    _________________________________    ___

      {'logsout'}      {1x1 Simulink.SimulationInput}    {1x1 Simulink.SimulationMetadata}    {20
```

By default, all the variables stored in the ensemble data are designated as `SelectedVariables`. Therefore, the returned table row includes all ensemble variables, including a variable `SimulationInput`, which contains the `Simulink.SimulationInput` object that configured the simulation for this ensemble member. That object includes the `ToothFaultGain` value used for the ensemble member, stored in a data structure in its `Variables` property. Examine that value. (For more information about how the simulation configuration is stored, see `Simulink.SimulationInput` (Simulink).)

```
data.SimulationInput{1}
```

```
ans = 
  SimulationInput with properties:

          ModelName: 'TransmissionCasingSimplified'
       InitialState: [0x0 Simulink.op.ModelOperatingPoint]
      ExternalInput: []
    ModelParameters: [0x0 Simulink.Simulation.ModelParameter]
    BlockParameters: [0x0 Simulink.Simulation.BlockParameter]
          Variables: [1x1 Simulink.Simulation.Variable]
          PreSimFcn: []
         PostSimFcn: []
         UserString: ''
```

```
Inputvars = data.SimulationInput{1}.Variables;
Inputvars.Name
```

```
ans = 
'ToothFaultGain'
```

```
Inputvars.Value
```

```
ans = -2
```

Suppose that you want to convert the `ToothFaultGain` values for each ensemble member into a binary indicator of whether or not a tooth fault is present. Suppose further that you know from your experience with the system that tooth-fault gain values less than 0.1 in magnitude are small enough to be considered healthy operation. Convert the gain value for this ensemble into an indicator that is 0 (no fault) for –0.1 < gain < 0.1, and 1 (fault) otherwise.

```
sT = abs(Inputvars.Value) < 0.1;
```

To append the new tooth-fault indicator to the corresponding ensemble data, first expand the list of data variables in the ensemble to include a variable for the indicator.

```
ensemble.DataVariables = [ensemble.DataVariables; "ToothFault"];
ensemble.DataVariables
```

```
ans = 6x1 string
    "PMSignalLogName"
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
    "ToothFault"
```

This operation is conceptually equivalent to adding a column to the table of ensemble data. Now that `DataVariables` contains the new variable name, assign the derived value to that column of the member using `writeToLastMemberRead`.

```
writeToLastMemberRead(ensemble,'ToothFault',sT);
```

In practice, you want to append the tooth-fault indicator to every member in the ensemble. To do so, reset the ensemble datastore to its unread state, so that the next read operation starts at the first ensemble member. Then, loop through all the ensemble members, computing `ToothFault` for each member and appending it. The `reset` operation does not change `ensemble.DataVariables`, so `"ToothFault"` is still present in that list.

```
reset(ensemble);

sT = false;
while hasdata(ensemble)
    data = read(ensemble);
    InputVars = data.SimulationInput{1}.Variables;
    TFGain = InputVars.Value;
    sT = abs(TFGain) < 0.1;
    writeToLastMemberRead(ensemble,'ToothFault',sT);
end
```

Finally, designate the new tooth-fault indicator as a condition variable in the ensemble datastore. You can use this designation to track and refer to variables in the ensemble data that represent conditions under which the member data was generated.

```
ensemble.ConditionVariables = {"ToothFault"};
ensemble.ConditionVariables
```

```
ans = 
"ToothFault"
```

You can add the new variable to `ensemble.SelectedVariables` when you want to read it out for further analysis. For an example that shows more ways to manipulate and analyze data stored in a `simulationEnsembleDatastore` object, see “Using Simulink to Generate Fault Data”.

Read Multiple Ensemble Members in One Operation

To read data from multiple ensemble members in one call to the `read` command, use the `ReadSize` property of an ensemble datastore. This example uses `simulationEnsembleDatastore`, but you can use the same technique for `fileEnsembleDatastore`.

Use the following code to create a `simulationEnsembleDatastore` object using data previously generated by running a Simulink model at a various fault values (see `generateSimulationEnsemble`). The ensemble includes simulation data for five different values of a model parameter, `ToothFaultGain`. (Because of the volume of data, the `unzip` operation might take a minute or two.) Specify some of the data variables to read.

```
unzip simEnsData.zip  % extract compressed files
ensemble = simulationEnsembleDatastore(pwd,'logsout');
ensemble.SelectedVariables = ["Vibration";"SimulationInput"];
```

By default, calling `read` on this ensemble datastore returns a single-row table containing the values of the `Vibration` and `SimulationInput` variables for the first ensemble member. Change the `ReadSize` property to read three members at once.

```
ensemble.ReadSize = 3;
data1 = read(ensemble)
```

```
data1=3×2 table
         Vibration                  SimulationInput
    ___________________    ______________________________

    {20202x1 timetable}    {1x1 Simulink.SimulationInput}
    {20215x1 timetable}    {1x1 Simulink.SimulationInput}
    {20204x1 timetable}    {1x1 Simulink.SimulationInput}
```

`read` returns a three-row table, where each row contains data from one of the first, second, and third ensemble members. `read` also updates the `LastReadMember` property of the ensemble datastore to a string array containing the paths of the three corresponding files. Avoid setting `ReadSize` to a value so large as to risk running out of memory while loading the data.

If the ensemble contains three or more additional members, the next `read` operation returns data from the fourth, fifth, and sixth members. Because the ensemble of this example contains only five members total, the next `read` operation returns only two rows.

```
data2 = read(ensemble)
```

```
data2=2×2 table
         Vibration                  SimulationInput
    ___________________    ______________________________

    {20213x1 timetable}    {1x1 Simulink.SimulationInput}
    {20224x1 timetable}    {1x1 Simulink.SimulationInput}
```

See Also

`generateSimulationEnsemble` | `fileEnsembleDatastore`

Topics

"Generate and Use Simulated Data Ensemble"
"Data Ensembles for Condition Monitoring and Predictive Maintenance"

Introduced in R2018a

workspaceEnsemble

Manage ensemble data stored in the MATLAB workspace using code generated by Diagnostic Feature Designer

Description

A `workspaceEnsemble` object is an ensemble object specialized for use in code generated by **Diagnostic Feature Designer**. The `workspaceEnsemble` object is similar to a `fileEnsembleDatastore` object, as both specify the data variables, independent variables, and condition variables in the ensemble. Unlike a file ensemble datastore, however, a workspace ensemble operates on data in memory rather than in external files.

When you import a table or a cell array into the app and generate code after you have completed your interactive feature design, that code includes the creation of a workspace ensemble. This ensemble contains variables that are identical to those in your initial import, and can manage any input data sets that include the same variables. For example, suppose that you import a 20-member table into the app, extract a feature, and generate a function. The workspace ensemble in that function is compatible with a 2000-member table, as long as the table includes the same variables.

For more information about data ensembles, see “Data Ensembles for Condition Monitoring and Predictive Maintenance”.

Creation

Syntax

```
wsensemble = workspaceEnsemble(Name,Value)
wsensemble = workspaceEnsemble(data,Name,Value)
```

Description

`wsensemble = workspaceEnsemble(Name,Value)` creates an empty `workspaceEnsemble` object `wsensemble` with properties specified by name-value pair arguments.

`wsensemble = workspaceEnsemble(data,Name,Value)` creates a `workspaceEnsemble` object `wsensemble` from the data set `data`.

Input Arguments

`data` — Input data set
table | cell array of tables

Input data set, specified as a table or a cell array of tables.

- If `data` is a table, each row represents the data of one ensemble member.
- If `data` is a cell array of tables, each table in the cell represents the data of one ensemble member.

Properties

`DataVariables` — Data variables
string | cell array

Data variables in the ensemble, specified as a string or cell array. Data variables are the main content of the members of an ensemble. Data variables can include measured data or derived data. For example, your data variables might include measured vibration or simulated vibration signals and derived values such mean vibration value or peak vibration frequency.

Example: `outputEnsemble = workspaceEnsemble(inputData,'DataVariables',["Vibration";"Tacho"])`

`IndependentVariables` — Independent variables
string | cell array

Independent variables in the ensemble, specified as a string or cell array. Typically, independent variables order the members of an ensemble. Examples are timestamps or the number of operating cycles.

Example: `outputEnsemble = workspaceEnsemble(inputData,'IndependentVariables',"Time")`

`ConditionVariables` — Condition variables
string | cell array

Condition variables in the ensemble, specified as a string or cell array. Condition variables label the members in an ensemble according to the fault condition or other operating condition under which the ensemble member was collected.

Example: `outputEnsemble = workspaceEnsemble(inputData,'ConditionVariables',"faultCode")`

`SelectedVariables` — Selected variables
string | cell array

Variables to read from the ensemble, specified as a string or cell array. `SelectedVariables` identifies which variables in `data` to read and operate on.

Example: `outputEnsemble.SelectedVariables = ["Vibration","Tacho"]`

`ReadSize` — Number of members to read
1 (default) | positive integer

Number of members to read from the workspace ensemble at once when you use the `read` command, specified as a positive integer that is smaller than the total number of members in the ensemble. By default, the `read` command returns a one-row table containing data from one ensemble member. To read data from multiple members in a single `read` operation, set this property to an integer value greater than one. For example, if `ReadSize` is 3, then `read` returns a three-row table where each row contains data from a different ensemble member. If fewer than `ReadSize` members are unread, then `read` returns a table with as many rows as there are remaining members.

Changing the `ReadSize` property also resets the ensemble to its unread state. For instance, suppose that you set `ReadSize` to `1` to read some ensemble members one at a time, and then change `ReadSize` to `3`. The next `read` operation returns data from the first three ensemble members.

Object Functions

refresh	Update a workspace ensemble with partitions of modified or added data computed in parallel processing
writeMember	Write data to a specific workspace ensemble member
readMember	Return ensemble member data based on the member index
findIndex	Find the workspace ensemble member indices for members that match a specified variable name and value

Examples

Create and Read a Workspace Ensemble

Create a `workspaceEnsemble` object from an ensemble table and read its contents.

Load the ensemble table `dataTable` and view the first three members.

```
load dfd_Tutorial dataTable
head(dataTable,3)
```

```
ans=3×3 table
        Vibration                Tacho           faultCode
    __________________    __________________    _________

    {6000x1 timetable}    {6000x1 timetable}        0
    {6000x1 timetable}    {6000x1 timetable}        1
    {6000x1 timetable}    {6000x1 timetable}        1
```

The table contains 16 members, each of which contain timetables with vibration and tacho data along with a scalar fault code.

Create a Workspace Ensemble

Create a workspace ensemble `wensemble` from `dataTable`.

```
wensemble = workspaceEnsemble(dataTable,'DataVariables',["Vibration";"Tacho"],...
           'ConditionVariables',"faultCode")
```

```
wensemble = 
  workspaceEnsemble with properties:

           DataVariables: [2x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: "faultCode"
       SelectedVariables: [3x1 string]
                ReadSize: 1
              NumMembers: 16
          LastMemberRead: [0x0 string]
```

Confirm the data condition variable selections.

```
dv = wensemble.DataVariables
```

```
dv = 2x1 string
    "Vibration"
```

```
    "Tacho"
```

```
cv = wensemble.ConditionVariables
```

```
cv = 
"faultCode"
```

Read Workspace Ensemble Members

Inspect the data variables in the workspace ensemble for the first two members.

By default, reading the ensemble returns all ensemble variables. To select a subset of variables to read, specify `SelectedVariables`.

```
wensemble.SelectedVariables = ["Vibration","Tacho"];
```

Use `read` to get the contents of the next unread member. Each time you read a member, the software marks that member as read, and the next `read` command returns the following member. You can use a succession of `read` commands to loop through an ensemble. To start at the first member, use `reset`.

```
reset(wensemble)
m1 = read(wensemble)
```

```
m1=1×2 table
        Vibration                Tacho       
    __________________    __________________

    {6000x1 timetable}    {6000x1 timetable}
```

```
m2 = read(wensemble);
```

`m1` and `m2` are both tables containing vibration and tacho data. `m1` contains the data for the first member. `m2` contains the data for the second member.

Examine the vibration samples for both members. Extract the vibration signals from `m1` and `m2` and display the first three samples of each signal.

```
m1vib = readMemberData(m1,'Vibration');
m2vib = readMemberData(m2,'Vibration');
head(m1vib,3)
```

```
ans=3×1 timetable
      Time         Data  
    _________    ________

    0 sec        -0.66925
    0.005 sec    -0.61623
    0.01 sec     -0.56666
```

```
head(m2vib,3)
```

```
ans=3×1 timetable
      Time        Data  
    _________    _______
```

```
    0 sec        -1.6231
    0.005 sec    -1.5892
    0.01 sec     -1.5534
```

Each `read` command returns a unique result.

Manage Variables and Features in a Workspace Ensemble

This example illustrates some of the basic commands used in code that **Diagnostic Feature Designer** generates. The example shows how to use these commands to create a workspace ensemble from a table, perform member-by-member computations for a new feature, and create a feature table and an ensemble table from the workspace ensemble.

Interacting with a workspace ensemble is similar to interacting with a file ensemble datastore or a simulation ensemble datastore. Many of the commands are the same. Unlike the ensemble datastores, which allow interaction with external files, the workspace ensemble datastore enables interaction with data in memory.

Create a Workspace Ensemble from a Table

Load the ensemble table `dataTable`, which contains 16 members, each of which contain timetables with vibration and tacho data along with a scalar fault code.

```
load dfd_Tutorial dataTable
```

Create a workspace ensemble `wensemble` from `dataTable`, specifying the data variables and condition variables corresponding to the variables in `dataTable`.

```
wensemble = workspaceEnsemble(dataTable,'DataVariables',["Vibration";"Tacho"],...
           'ConditionVariables',"faultCode")
```

```
wensemble = 
  workspaceEnsemble with properties:

           DataVariables: [2x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: "faultCode"
       SelectedVariables: [3x1 string]
                ReadSize: 1
              NumMembers: 16
          LastMemberRead: [0x0 string]
```

Processing the data and extracting features requires only `Vibration` and `Tacho`. Specify `SelectedVariables` to contain `Vibration` and `Tacho`.

```
wensemble.SelectedVariables = ["Vibration","Tacho"];
```

Compute Mean of Vibration Signal for First Ensemble Member

The mean of the vibration signal represents a scalar feature for each member. Compute this feature for the first member, using an approach that scales to a loop that processes multiple members.

Reset the ensemble and read the first member.

```
reset(wensemble)
m = read(wensemble)
```

```
m=1×2 table
         Vibration                Tacho
    __________________    __________________

    {6000x1 timetable}    {6000x1 timetable}
```

Extract the vibration data from the timetable.

```
mvibd = readMemberData(m,'Vibration',"Data");
```

Compute the mean value of the vibration.

```
m_mean = mean(mvibd)
```

```
m_mean = 0.0218
```

Append the results to member table `m`.

```
m = [m,table(m_mean,'VariableNames',"Data_Mean")]
```

```
m=1×3 table
         Vibration                Tacho            Data_Mean
    __________________    __________________    _________

    {6000x1 timetable}    {6000x1 timetable}    0.021809
```

Add New Feature to Ensemble Variables

To incorporate the updated member into `wensemble`, you must first specify the new `Data_Mean` feature as an ensemble variable. Add `Data_Mean` to the set of ensemble data variables `dv` using dot notation.

```
dv = wensemble.DataVariables;
wensemble.DataVariables = [dv;"Data_Mean"];
```

Append Updated Member Table to Workspace Ensemble

Append the updated member table to the ensemble using the `writeToLastMemberRead` command.

```
writeToLastMemberRead(wensemble,m)
```

Loop through Remaining Ensemble Members

Perform the same member-specific steps for the remaining ensemble members.

```
while hasdata(wensemble)
    m = read(wensemble);
    mvibd = readMemberData(m,'Vibration',"Data");
    m_mean = mean(mvibd);
    m = [m,table(m_mean,'VariableNames',"Data_Mean")];
    writeToLastMemberRead(wensemble,m)
end
```

Create Feature Table and Ensemble Table from Workspace Ensemble

Extract the feature table from `wensemble` with the `readFeatureTable` command. View the first three rows.

```
ft = readFeatureTable(wensemble);
head(ft,3)
```

```
ans=3×2 table
    faultCode    Data_Mean
    _________    __________

        0         0.021809
        1       -0.0092964
        1         -0.46431
```

The feature table contains the condition variable `FaultCode` and the data variable `Data_Mean`.

Set the `SelectedVariables` property to include all variables so that the resulting ensemble table contains all your information.

```
wensemble.SelectedVariables = ["Vibration";"Tacho";"Data_Mean";"faultCode"]
```

```
wensemble = 
  workspaceEnsemble with properties:

           DataVariables: [3x1 string]
    IndependentVariables: [0x1 string]
      ConditionVariables: "faultCode"
       SelectedVariables: [4x1 string]
                ReadSize: 1
              NumMembers: 16
          LastMemberRead: [0x0 string]
```

Use the datastore command `readall` to convert the workspace ensemble into an ensemble table.

```
tensemble = readall(wensemble)
```

```
tensemble=16×4 table
        Vibration                 Tacho             Data_Mean     faultCode
    __________________    __________________    __________    _________

    {6000x1 timetable}    {6000x1 timetable}      0.021809        0    
    {6000x1 timetable}    {6000x1 timetable}    -0.0092964        1    
    {6000x1 timetable}    {6000x1 timetable}      -0.46431        1    
    {6000x1 timetable}    {6000x1 timetable}        0.4922        1    
    {6000x1 timetable}    {6000x1 timetable}        0.3923        1    
    {6000x1 timetable}    {6000x1 timetable}      -0.12383        1    
    {6000x1 timetable}    {6000x1 timetable}       0.42548        1    
    {6000x1 timetable}    {6000x1 timetable}       -0.4598        1    
    {6000x1 timetable}    {6000x1 timetable}      0.062685        0    
    {6000x1 timetable}    {6000x1 timetable}      0.059155        0    
    {6000x1 timetable}    {6000x1 timetable}      0.037965        0    
    {6000x1 timetable}    {6000x1 timetable}       0.53982        1    
    {6000x1 timetable}    {6000x1 timetable}       0.52377        1    
    {6000x1 timetable}    {6000x1 timetable}        1.0357        1    
    {6000x1 timetable}    {6000x1 timetable}        1.0592        1    
```

```
    {6000x1 timetable}    {6000x1 timetable}      -0.94084        1
```

The table includes the original signals and the new feature.

See Also

Apps
Diagnostic Feature Designer

Functions
`read` | `readMemberData` | `readFeatureTable` | `writeToLastMemberRead` | `reset` | `readall` | `writeMember` | `readMember` | `findIndex`

Objects
`fileEnsembleDatastore` | `simulationEnsembleDatastore`

Topics
"Condition Indicators for Monitoring, Fault Detection, and Prediction"
"Automatic Feature Extraction Using Generated MATLAB Code"
"Anatomy of App-Generated MATLAB Code"

Introduced in R2020a