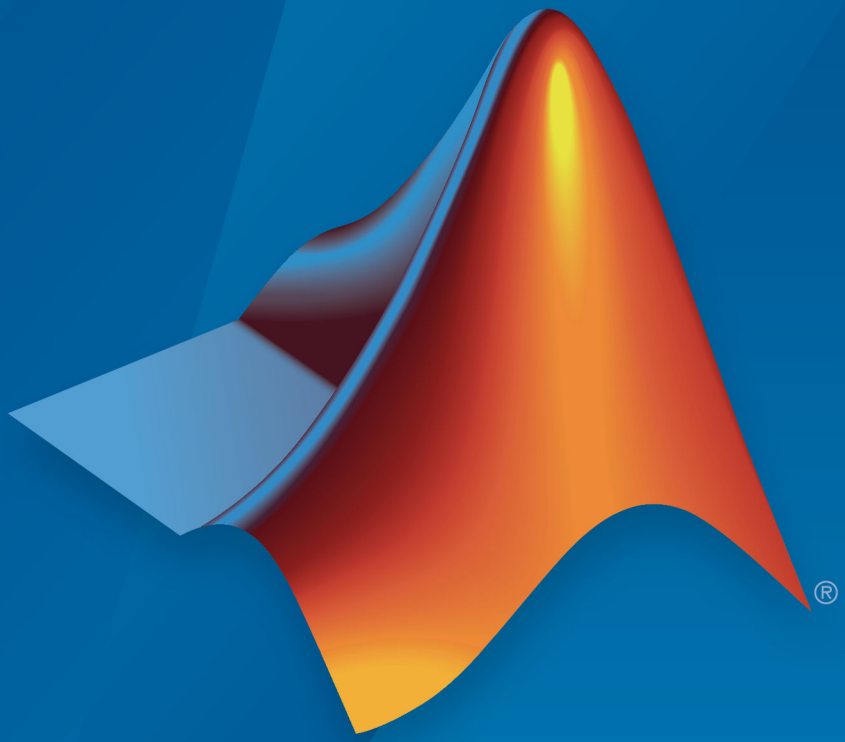
Predictive Maintenance Toolbox™

Reference



MATLAB®

R2018a

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

Predictive Maintenance Toolbox ™ Reference

Revision History

March 2018	Online only	New for Version 1.0 (Release 2018a)

Contents

Functions — Alphabetical List

approximateEntropy

Measure of regularity of nonlinear time series

Syntax

```
approx_entropy = approximateEntropy(X)
approx_entropy = approximateEntropy( ___ ,Name,Value)
```

Description

`approx_entropy = 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.

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

Examples

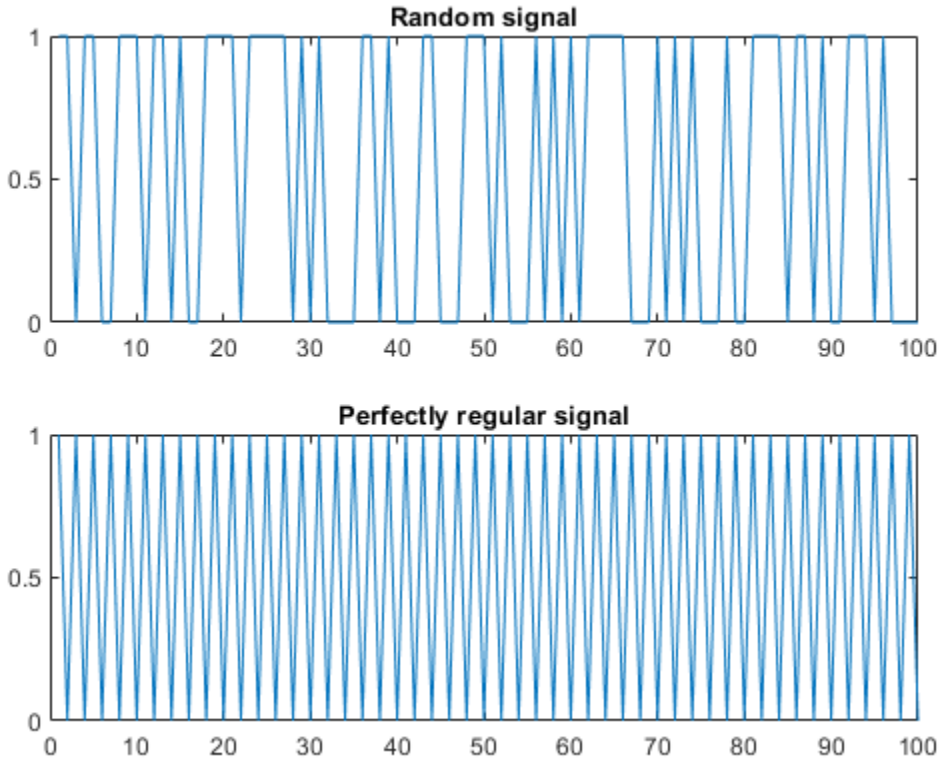
Compute Approximate Entropy of Signals

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

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

Visualize the random and regular signals.

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



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

Find approximate entropy of the two signals.

```
value_reg = approximateEntropy(x_reg)
```

```
value_reg = 2.6141e-04
```

```
value_irreg = approximateEntropy(x_randn)
```

```
value_irreg = 0.7289
```

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.

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.

Name-Value Pair Arguments

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

Example: `...,'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*variance(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*variance(X), if X has a single column.
- 0.2*sqrt(trace(cov(X))), if X has multiple columns.

Output Arguments

approx_entropy — Approximate entropy of nonlinear time series
scalar

Approximate entropy of nonlinear times 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 `approx entropy` while a less predictable signal has a relatively larger value of `approx entropy`.

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,

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

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

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

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` | `lyapunovExponent` | `phaseSpaceReconstruction`

Introduced in R2018a

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.

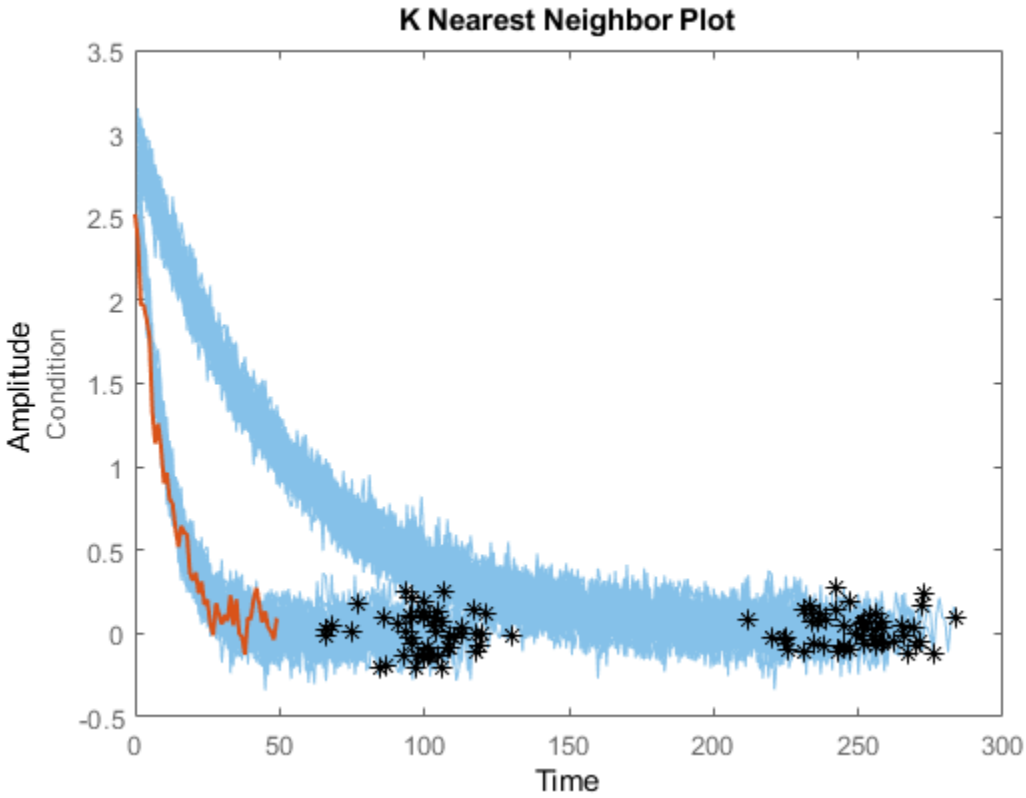
```
mdl = pairwiseSimilarityModel;
fit(mdl,pairwiseTrainTables,"Time","Condition")
```

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.

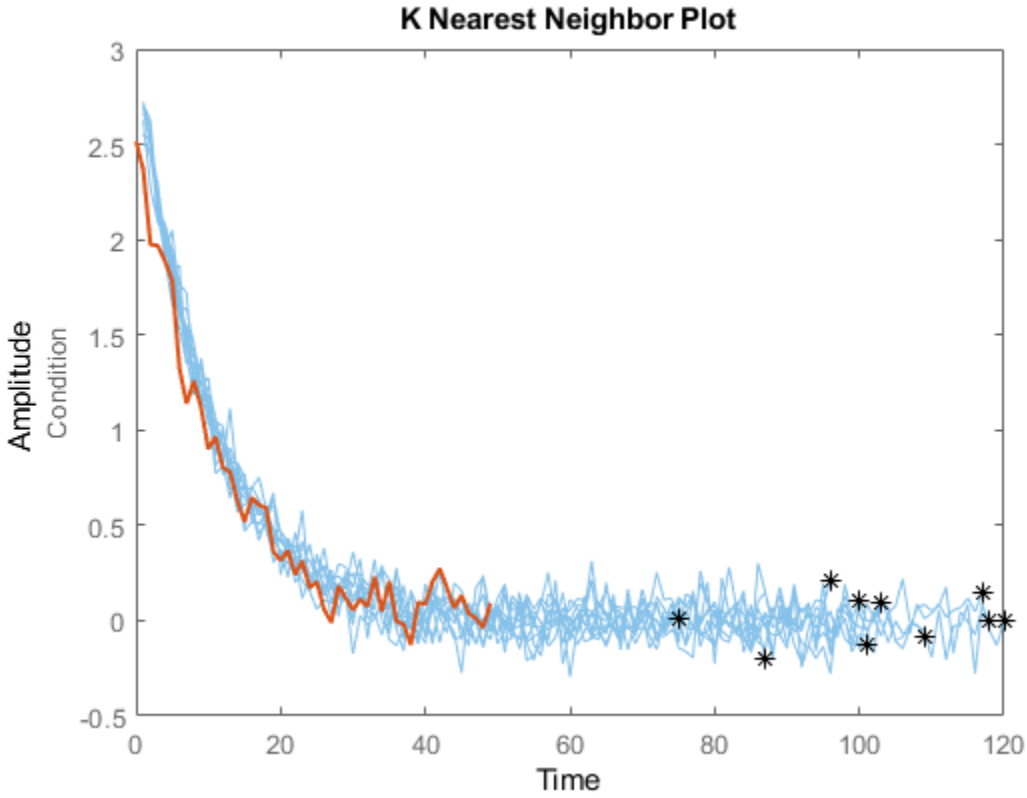
```
mdl = pairwiseSimilarityModel;
fit(mdl,pairwiseTrainTables,"Time","Condition")
```

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, specfied 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 single 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

```
corr_dim = correlationDimension(X)
[corr_dim,rrange,corr_int] = correlationDimension(X)
___ = correlationDimension( ___ ,Name,Value)

correlationDimension(X)
```

Description

`corr_dim = 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. `corr_dim` 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]

`[corr_dim,rrange,corr_int] = correlationDimension(X)` 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(X)` with no output arguments creates a correlation integral versus neighborhood radius plot.

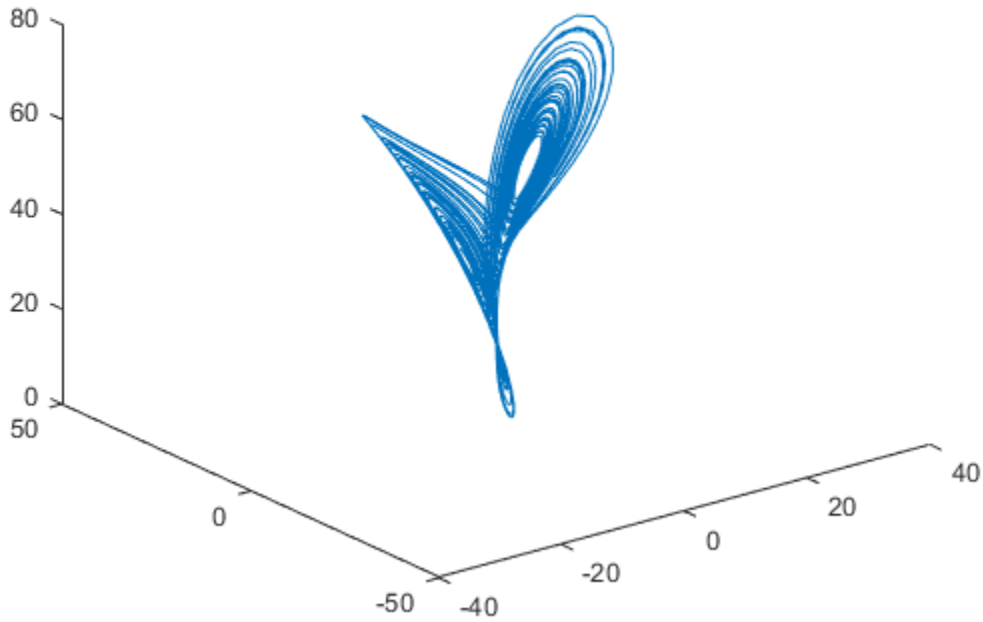
Examples

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));
```



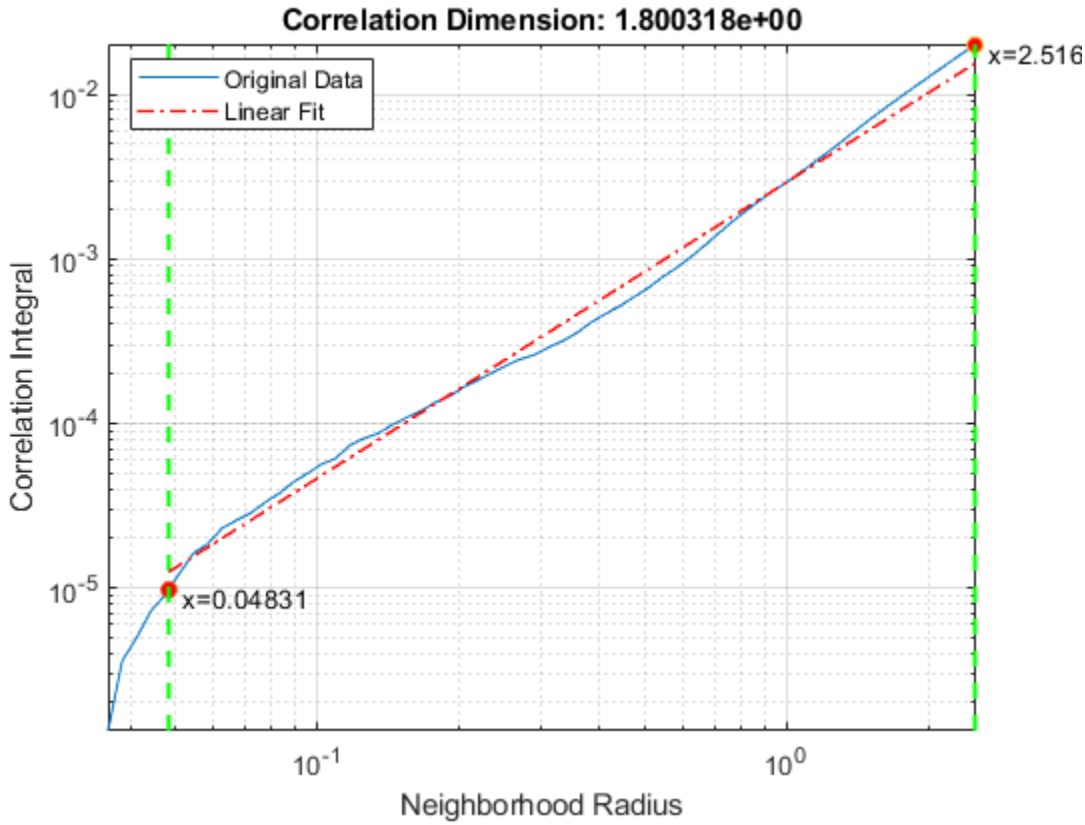
For this example, use only x-direction data of the Lorenz Attractor. Since `Lag` is unknown, estimate the delay using `phaseSpaceReconstruction`. Set `'Dimension'` to 3 since the Lorenz Attractor is a three-dimensional system. The `Dimension` and `Lag` parameters are required to create the correlation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

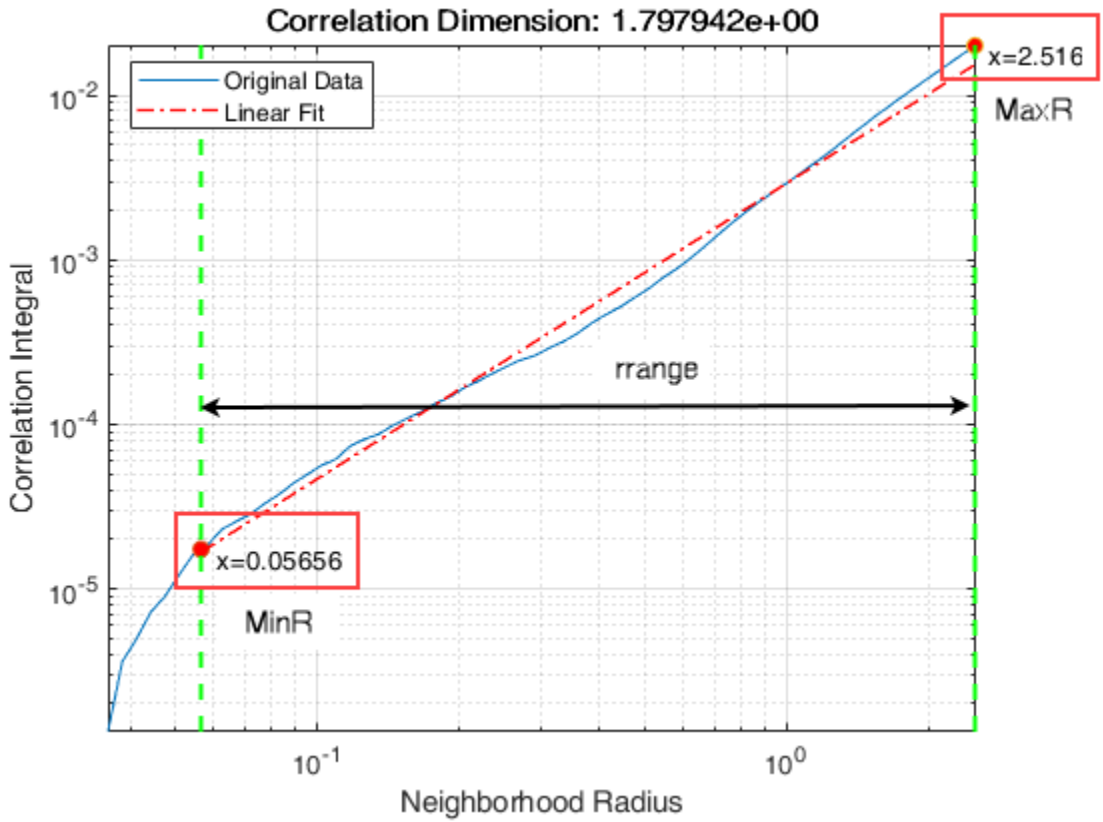
```
Np = 100;
correlationDimension(xdata,'Dimension',dim,'Lag',lag,'NumPoints',Np);
```



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;
corr_dim = correlationDimension(xdata,'Dimension',dim,'MinRadius',MinR,'MaxRadius',MaxR
```

```
corr_dim = 1.7490
```

The value of correlation dimension is directly proportional to the level of chaos in the system, that is, a higher value of `corr_dim` 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.

Name-Value Pair Arguments

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

Example: `...,'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.

When `NumPoints` = 1, only the `MaxRadius` is used to compute the correlation dimension.

`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

`corr_dim` — Correlation Dimension
scalar

Correlation dimension, returned as a scalar. `corr_dim` 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. `corr_dim` 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`.

`corr_int` — Correlation integral
array

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

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

   where $\mathbf{1}$ 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 `corr_dim` 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` | `lyapunovExponent` | `phaseSpaceReconstruction`

Introduced in R2018a

fit

Estimate parameters of remaining useful life model using historical data

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 life span 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 more information on predicting remaining useful life using these models, see “Models for Predicting Remaining Useful Life”.

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

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

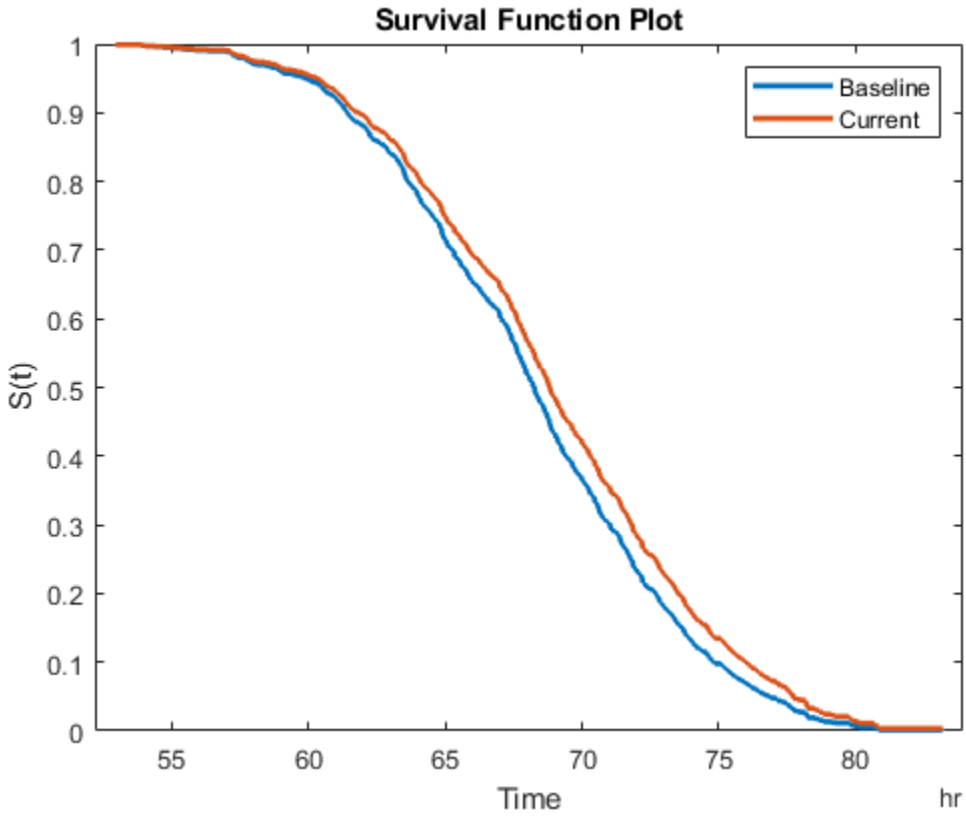
Predict the RUL for the battery.

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

```
estRUL = duration
   38.657 hr
```

Plot the survivor 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 the models in the following table. `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+1)-by-M_i column arrays, where N is the number of features and M_i 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.

- `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 MATLABvariable 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 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
“Models for Predicting Remaining Useful Life”

“Data Ensembles for Condition Monitoring and Predictive Maintenance”

Introduced in R2018a

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 during simulation. The output arguments indicate whether any simulations generate errors and return any such errors. The simulation data logs are stored in the current folder. 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`, as well as state values `xout` and `xfinal` (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);
```

```
[26-Feb-2018 19:09:46] Running SetupFcn...
[26-Feb-2018 19:09:46] Running simulations...
[26-Feb-2018 19:10:11] Completed 1 of 5 simulation runs
[26-Feb-2018 19:10:30] Completed 2 of 5 simulation runs
[26-Feb-2018 19:10:45] Completed 3 of 5 simulation runs
[26-Feb-2018 19:11:00] Completed 4 of 5 simulation runs
[26-Feb-2018 19:11:16] Completed 5 of 5 simulation runs
```

Finally, 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: [6x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: [0x0 string]
       SelectedVariables: [6x1 string]
              NumMembers: 5
          LastMemberRead: [0x0 string]
```

```
ensemble.DataVariables
```

```
ans = 6x1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
    "xFinal"
    "xout"
```

```
ensemble.SelectedVariables
```

```
ans = 6x1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
    "xFinal"
    "xout"
```

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

Simluation 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` — File path

`pwd` (default) | string | character vector

File 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`).

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 single 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™).

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

See Also

`Simulink.SimulationInput` | `simulationEnsembleDatastore`

Topics

“Data Ensembles for Condition Monitoring and Predictive Maintenance”

Introduced in R2018a

lyapunovExponent

Characterize the rate of separation of infinitesimally close trajectories

Syntax

```
lyap_exp = lyapunovExponent(X,fs)
[lyap_exp,estep,ldiv] = lyapunovExponent(X,fs)
[lyap_exp,estep,ldiv] = lyapunovExponent( ___ ,Name,Value)

lyapunovExponent(X,fs)
```

Description

`lyap_exp = 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.

`[lyap_exp,estep,ldiv] = lyapunovExponent(X,fs)` 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.

`[lyap_exp,estep,ldiv] = lyapunovExponent( ___ ,Name,Value)` estimates the Lyapunov exponent with additional options specified by one or more `Name,Value` pair arguments.

`lyapunovExponent(X,fs)` with no output arguments creates an average logarithmic divergence versus expansion step plot.

Use the generated interactive plot to find an appropriate `ExpansionRange`.

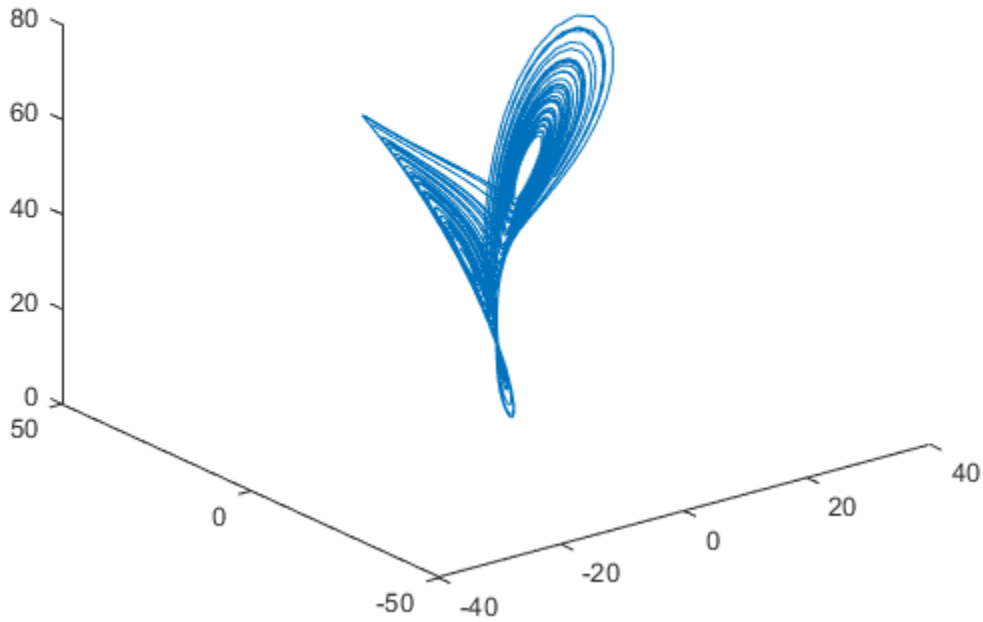
Examples

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));
```



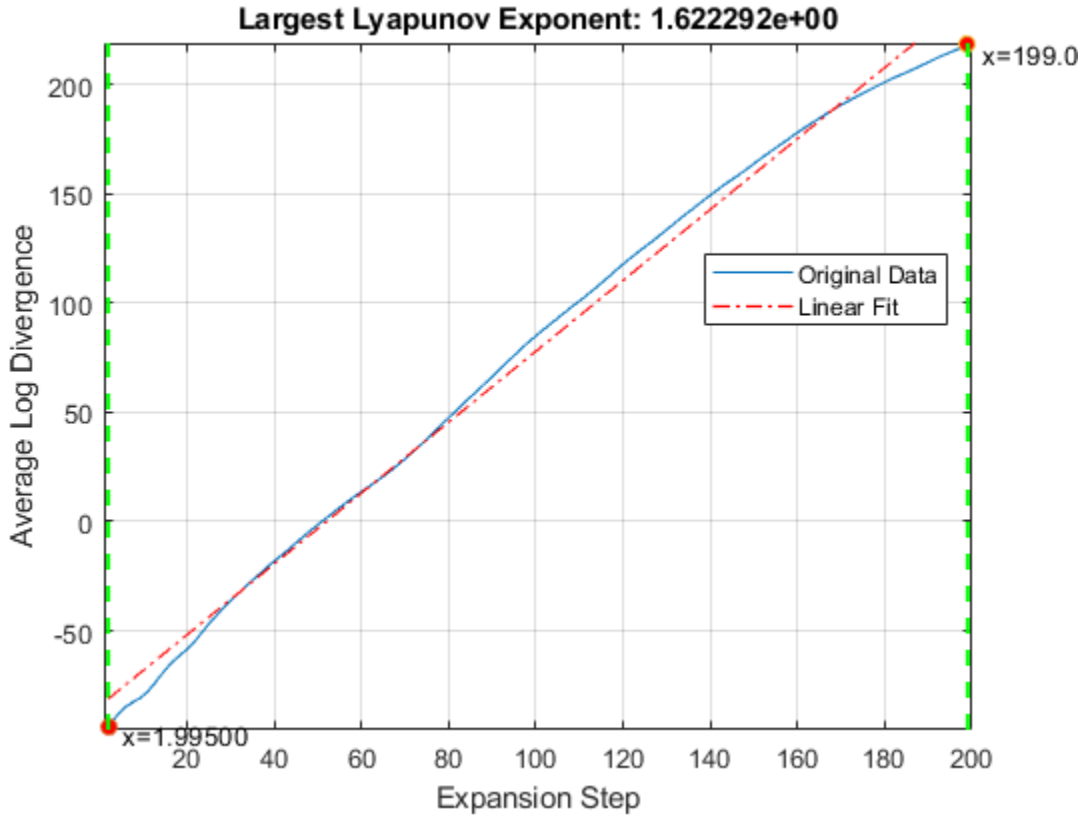
For this example, use the x-direction data of the Lorenz attractor. Since `Lag` is unknown, estimate the delay using `phaseSpaceReconstruction`. Set `Dimension` to 3 since the Lorenz attractor is a three-dimensional system. The `Dimension` and `Lag` parameters are required to create the logarithmic divergence 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.

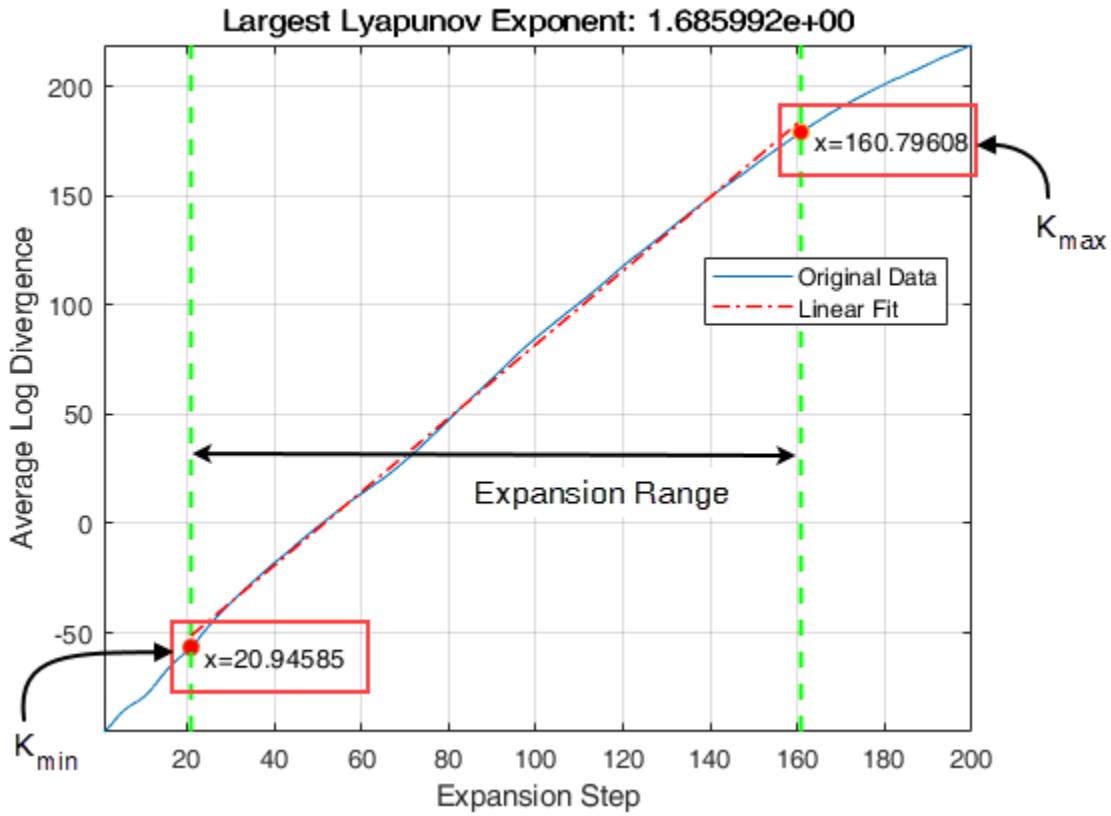
```
ERange = 200;
lyapunovExponent(xdata,fs,'Dimension',dim,'Lag',lag,'ExpansionRange',ERange)
```



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;
lyap_exp = lyapunovExponent(xdata,fs,'Dimension',dim,'Lag',lag,'ExpansionRange',[Kmin K
```

```
lyap_exp = 1.6837
```

A negative Lyapunov exponent indicates convergence, while positive Lyapunov exponents demonstrate divergence and chaos. The magnitude of lyap_exp 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.

Name-Value Pair Arguments

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

Example: `…,'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
`1/max(meanfreq(X))` (default) | scalar

Mean period, specified as the comma-separated pair consisting of '`MinSeparation`' and a scalar.

`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 `1/max(meanfreq(X))`.

`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

`lyap_exp` — Largest Lyapunov exponent

scalar

Largest Lyapunov exponent, returned as a scalar. `lyap_exp` 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 `lyap_exp` 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 $\min_{i^*} \left\| Y_i - Y_{i^*} \right\|$ such that $\left| i - i^* \right| > MinSeparation$, where `MinSeparation`, the mean period, is the reciprocal of the mean frequency.

3 The largest Lyapunov exponent is calculated as,

$$lyap\_\exp = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{K*dt}\sum_{K=K_{\min}}^{K_{\max}}\ln\frac{\left\|Y_{i+K}-Y_{i^*+K}\right\|}{\left\|Y_i-Y_{i^*}\right\|}\right)$$

where, K_{min} and K_{max} represent `ExpansionRange`, `dt` is the sampling time and $ldiv = \ln\frac{\left\|Y_{i+K}-Y_{i^*+K}\right\|}{\left\|Y_i-Y_{i^*}\right\|}$

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

phaseSpaceReconstruction

Convert observed time series to state vectors

Syntax

```
XR = phaseSpaceReconstruction(X,lag,dim)
[XR,est_lag,est_dim] = phaseSpaceReconstruction(X)
[XR,est_lag,est_dim] = phaseSpaceReconstruction(X,lag)
[XR,est_lag,est_dim] = 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,est_lag,est_dim] = phaseSpaceReconstruction(X)` returns reconstructed phase space `XR` along with the estimated delay `est_lag` and embedding dimension `est_dim`.

`[XR,est_lag,est_dim] = phaseSpaceReconstruction(X,lag)` returns the reconstructed phase space `XR` of uniformly sampled time domain signal `X` and embedding dimension `est_dim` using time dealy specified by `lag.`

`[XR,est_lag,est_dim] = phaseSpaceReconstruction(X,[],dim)` returns the reconstructed phase space `XR` of uniformly sampled time domain signal `X` and time delay `est_lag` 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 subaxes of the reconstructed phase space with histogram plots along the diagonal.

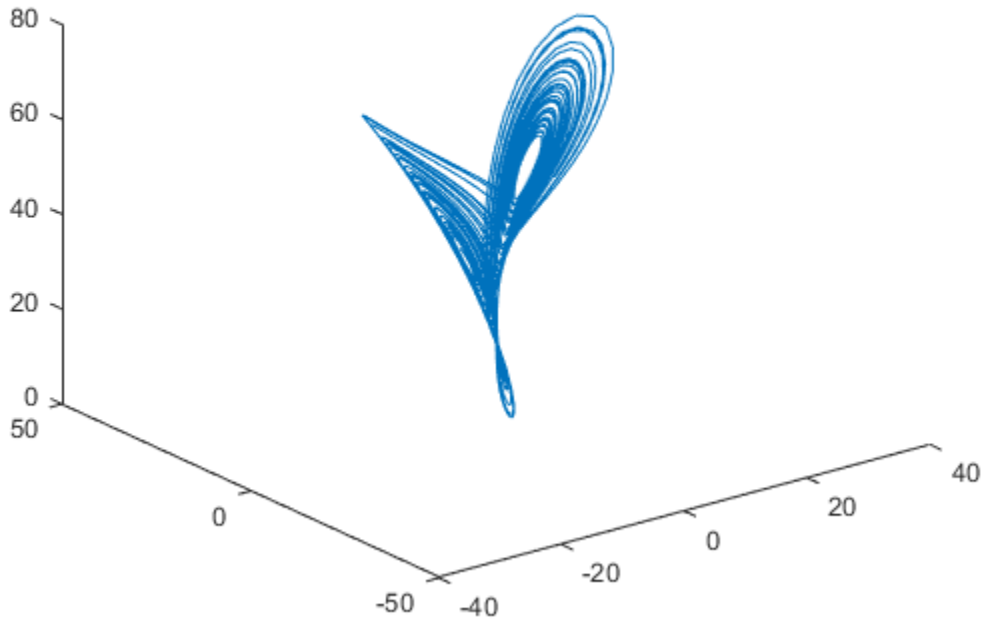
Examples

Reconstruct Data using Phase Space Reconstruction

In this example, assume that you have measurement only in x-direction for a Lorenz Attractor, which 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);
[~,est_lag,est_dim] = phaseSpaceReconstruction(xdata)
```

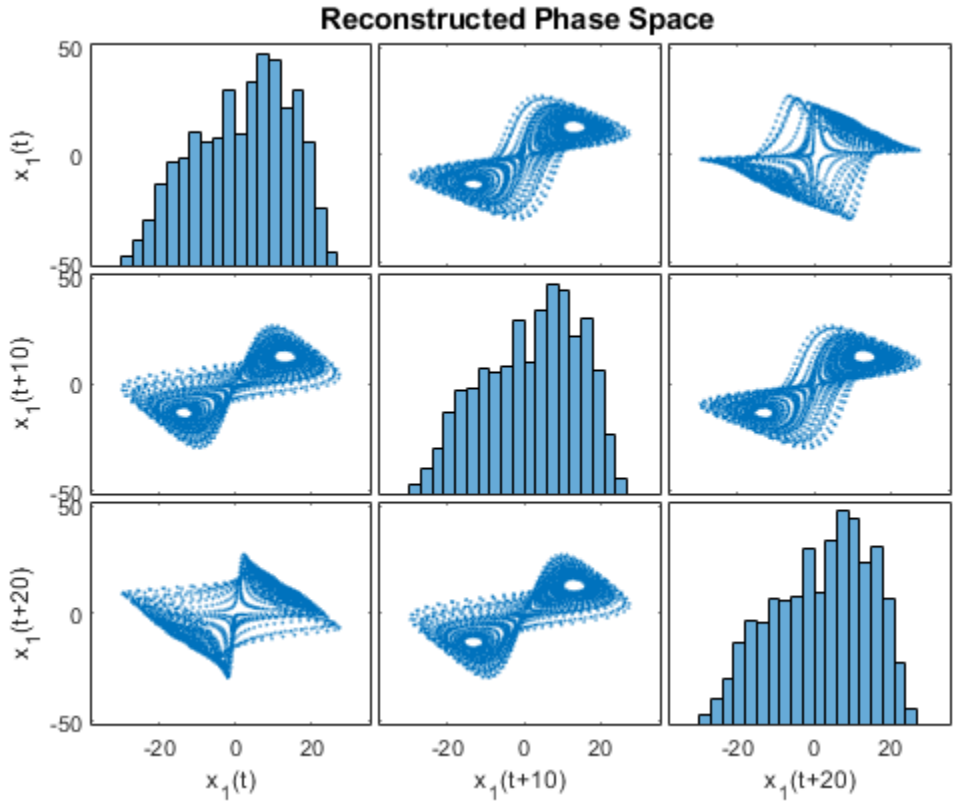
```
est_lag = 10
```

```
est_dim = 3
```

Since the Lorenz Attractor has data in 3 dimensions, the estimated embedding dimension `est_dim` is 3.

Visualize the reconstructed phase space using the estimated lag and embedding dimension.

```
phaseSpaceReconstruction(xdata,est_lag,est_dim);
```



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 reconstruced 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 single 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 `est_lag`.

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.

`est_lag` — Estimated time delay
scalar

Estimated time delay, returned as a scalar, regardless of the size of `X`.

`est_lag` is estimated using Average Mutual Information (AMI) algorithm. For more information, see “Algorithms” on page 1-53.

`est_dim` — Estimated embedding dimension
scalar

Estimated embedding dimension, returned as a scalar, regardless of the size of `X`.

`est_dim` is estimated using False Nearest Neighbor (FNN) algorithm. For more information, see “Algorithms” on page 1-53.

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,

$$X = [X_1, X_2, ..., X_S] = \begin{bmatrix} x_{1,1} & \dots & x_{S,1} \\ \vdots & \ddots & \vdots \\ x_{1,N} & \cdots & x_{S,N} \end{bmatrix}$$

`phaseSpaceReconstruction` computes the reconstruction for each time series as,

$$X_i^r = \left(X_{1,i}^r, X_{2,i}^r, ..., X_{S,i}^r \right),\ i = 1, 2, ..., N - \left(\max\{m_i\} - 1 \right) \left(\max\{\tau_i\} \right)$$

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` | `correlationDimension` | `lyapunovExponent`

Introduced in R2018a

plot

Plot survivor function for covariate survival remaining useful life model

Syntax

```
plot(mdl)
plot(mdl,covariates)
```

Description

`plot(mdl)` plots the baseline survivor 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 survivor function computed for the covariate data in `covariates`. To obtain the survivor function, the hazard rate is computed using the covariates and combined with the baseline survivor 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;
```

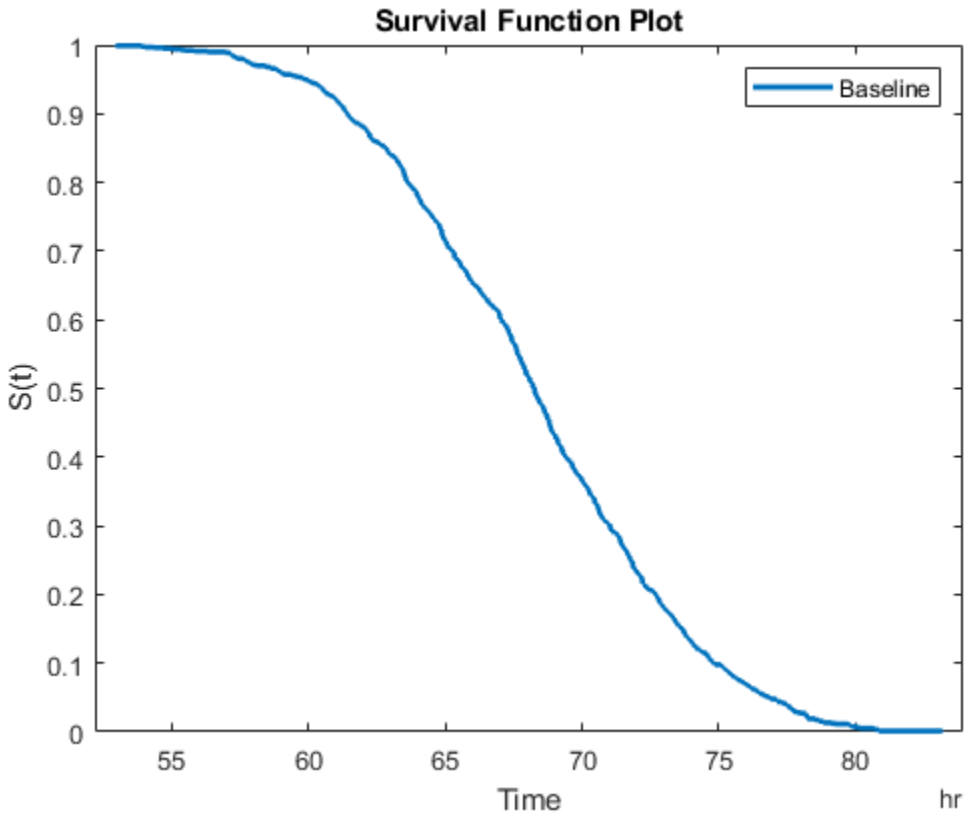
Train the survival model using the training data, specifying the life time variable, data variables, and encoded variable. There is no censor variable for this training data.

```
fit(mdl,covariateData,"DischargeTime",["Temperature","Load","Manufacturer"],[],"Manufac
```

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

Plot the baseline survivor 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',"Manufactu
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(TestBatteryLoad,TestAmbientTemperature,'B','RowTimes',hours(30));
TestData.Properties.VariableNames = {'Temperature','Load','Manufacturer'};
TestData.Properties.DimensionNames{1} = 'DischargeTime';
```

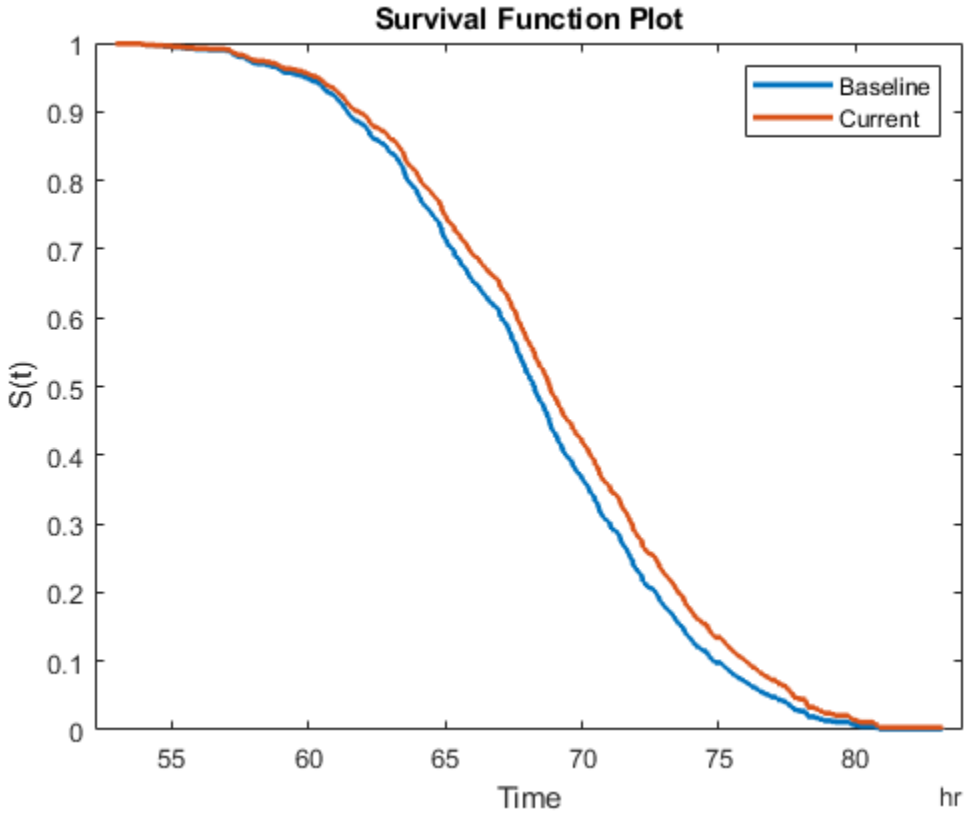
Predict the RUL for the battery.

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

```
estRUL = duration
   38.657 hr
```

Plot the survivor 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 survivor 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 survivor function, the hazard rate is computed using the covariates and combined with the baseline survivor function. For more information, see "Cox Proportional Hazards Model" (Statistics and Machine Learning Toolbox).

See Also

Functions
`covariateSurvivalModel` | `coxphfit` | `predictRUL`

Topics

"Cox Proportional Hazards Model" (Statistics and Machine Learning Toolbox)

Introduced in R2018a

predictRUL

Estimate remaining useful life for a test component

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 more information on predicting remaining useful life using these models, see “Models for Predicting Remaining Useful Life”.

Syntax

```
estRUL = predictRUL(mdl,data)
estRUL = predictRUL(mdl,data,bounds)

estRUL = predictRUL(mdl,threshold)
estRUL = predictRUL(mdl,currentValue,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

`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,currentValue,threshold)` estimates the RUL for a component using a degradation model and the current usage time and degradation feature measurement in `currentValue`.

`estRUL = predictRUL(mdl,usageTime)` estimates the RUL for a component using residual 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
   59.406 hr
```

The estimated RUL is about 60 hours, which indicates a total predicted life span of 110 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 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 after 150 hours. Predict the remaining useful life of the component at this time using the trained exponential degradation model. The RUL is the forecasted time at which the degradation feature will pass the specified threshold.

```
estRUL = predictRUL(mdl,expTestData(150,:),threshold)
```

```
estRUL = duration
   129.73 hr
```

The estimated RUL is around `130` hours, which indicates a total predicted life span of `280` 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',"Manufactu
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(TestBatteryLoad,TestAmbientTemperature,'B','RowTimes',hours(30));
TestData.Properties.VariableNames = {'Temperature','Load','Manufacturer'};
TestData.Properties.DimensionNames{1} = 'DischargeTime';
```

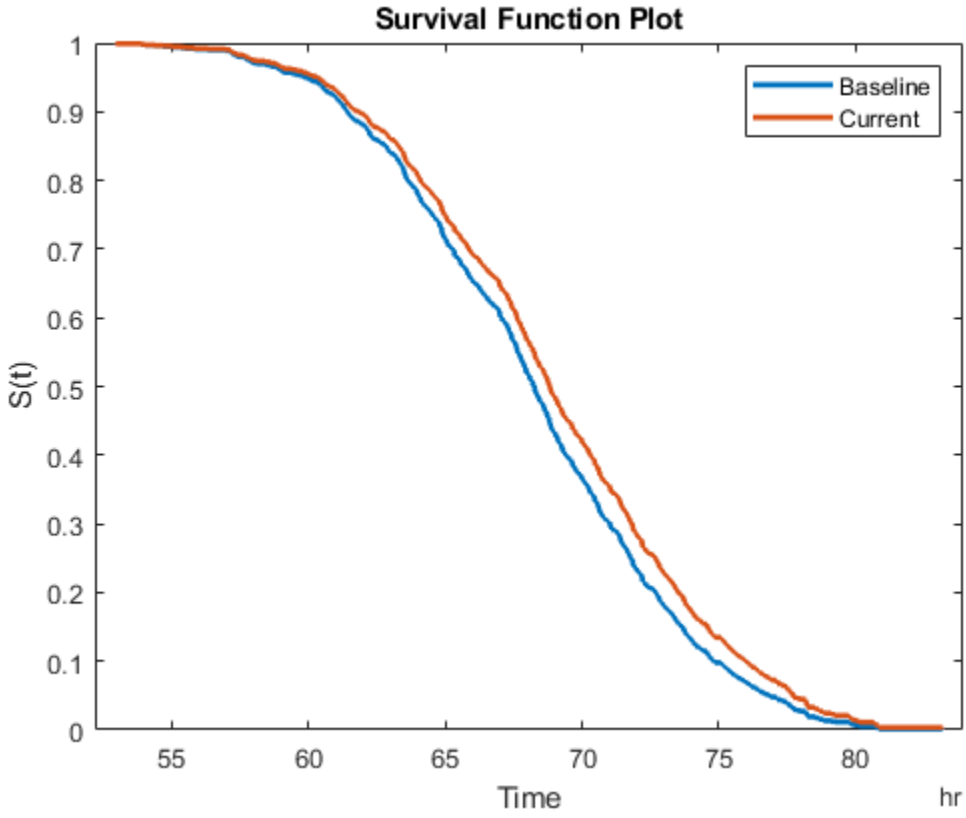
Predict the RUL for the battery.

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

```
estRUL = duration
   38.657 hr
```

Plot the survivor 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.

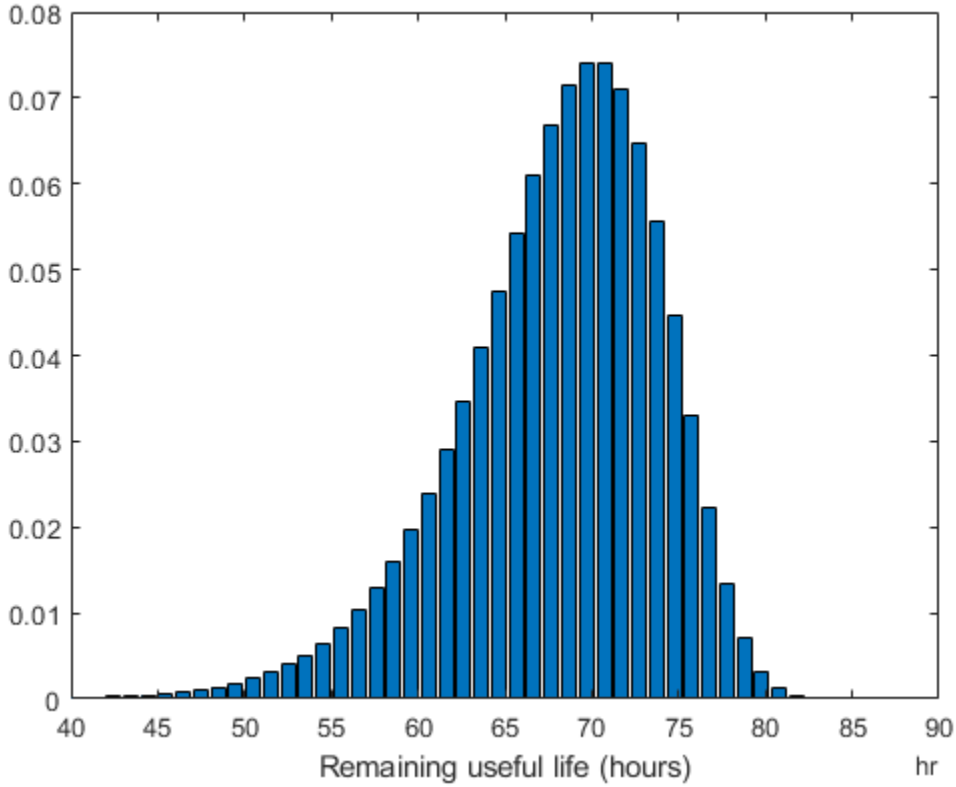
```
fit(mdl,reliabilityData)
```

Predict the life span of a new component and obtain the probability distribution function for the estimate.

```
[estRUL,ciRUL,pdfRUL] = predictRUL(mdl);
```

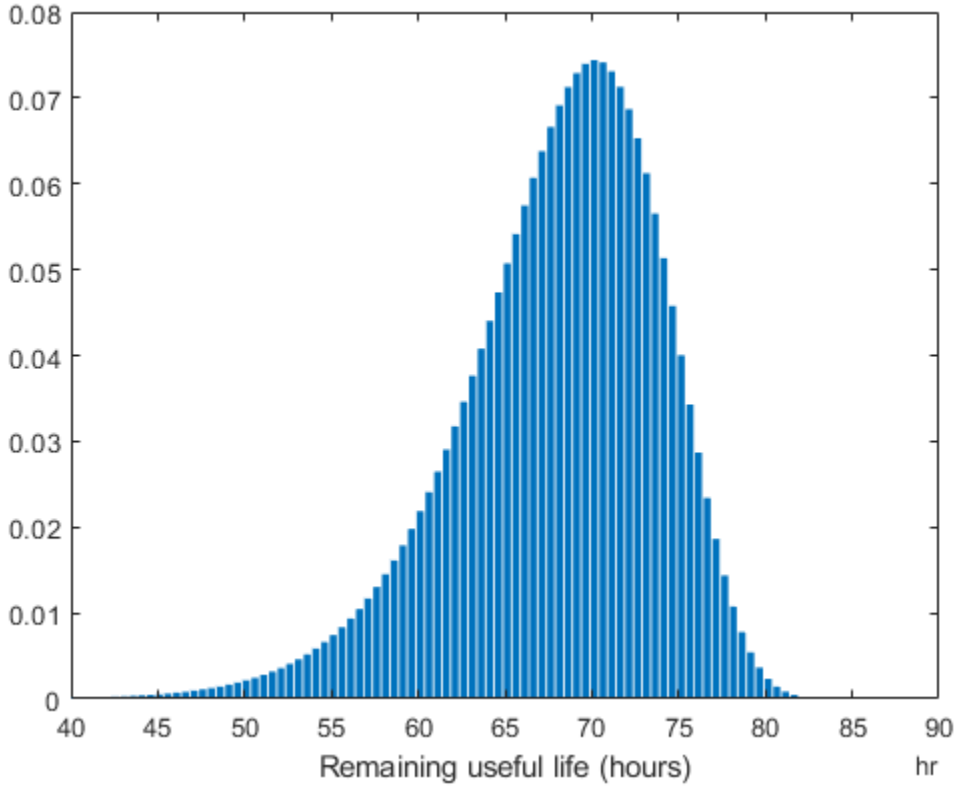
Plot the probability distribution.

```
bar(pdfRUL.RUL,pdfRUL.ProbabilityDensity)
xlabel('Remaining useful life (hours)')
xlim(hours([40 90]))
```



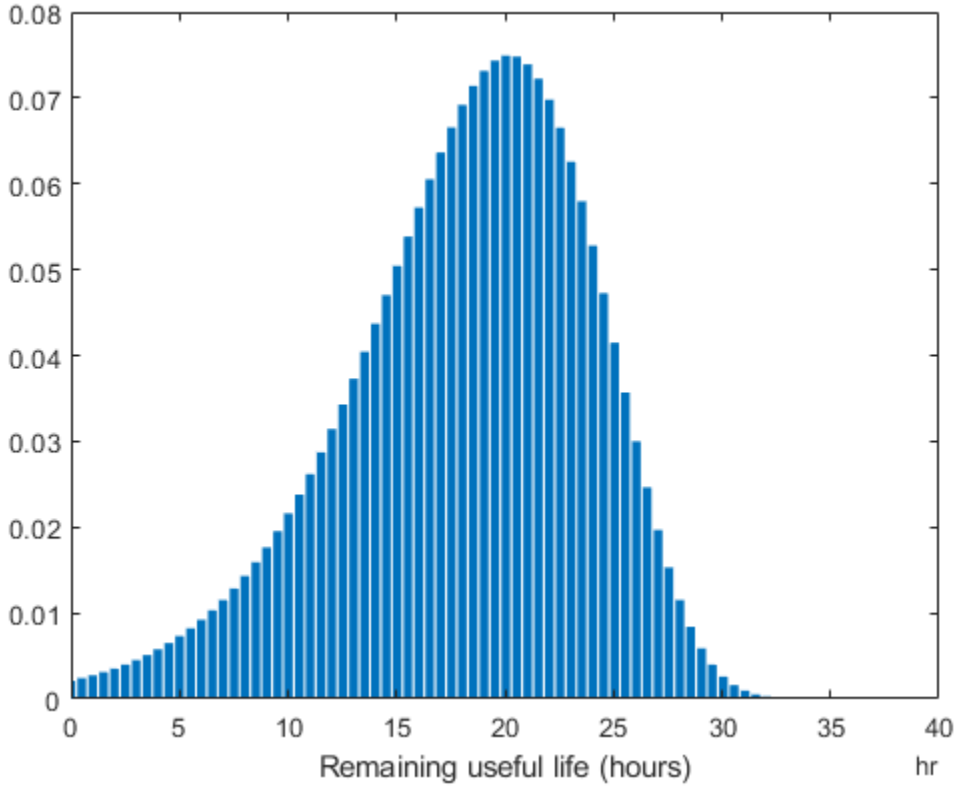
Improve the distribution view by providing the number of bins and bin size for the prediction.

```
[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 models in the following table.

RUL Model Groups	More Information
Degradation models	`linearDegradationModel`

RUL Model Groups	More Information
	`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.

`currentValue` — Current usage time and degradation feature measurement
row vector | `table` | `timetable`

Current usage time and degradation feature measurement for estimating the remaining useful life of degradation models, specified as one of the following:

- Row vector of the form [*T X*], where *T* is the current usage time and *X* is the degradation feature measurement.
- `table` or `timetable` with one row — The table must contain variables with names that match the strings in the `DataVariables` and `LifeTimeVariable` properties of `mdl`.

`currentValue` applies when `mdl` is a `linearDegradationModel` or `exponentialDesgradationModel` 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 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 single 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
`100` value (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 `currentValue`.
- 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.

See Also

`fit`

Topics

“Models for Predicting Remaining Useful Life”

Introduced in R2018a

read

Read member data from an ensemble datastore

Use this function to read data from ensemble datastores for condition monitoring and predictive maintenance.

Syntax

```
data = read(ensemble)
[data,info] = read(ensemble)
```

Description

`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 row.

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: [6x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: [0x0 string]
       SelectedVariables: [6x1 string]
             NumMembers: 5
          LastMemberRead: [0x0 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 = 6x1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
    "xFinal"
    "xout"
```

```
ensemble.SelectedVariables
```

```
ans = 6x1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
    "xFinal"
    "xout"
```

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.

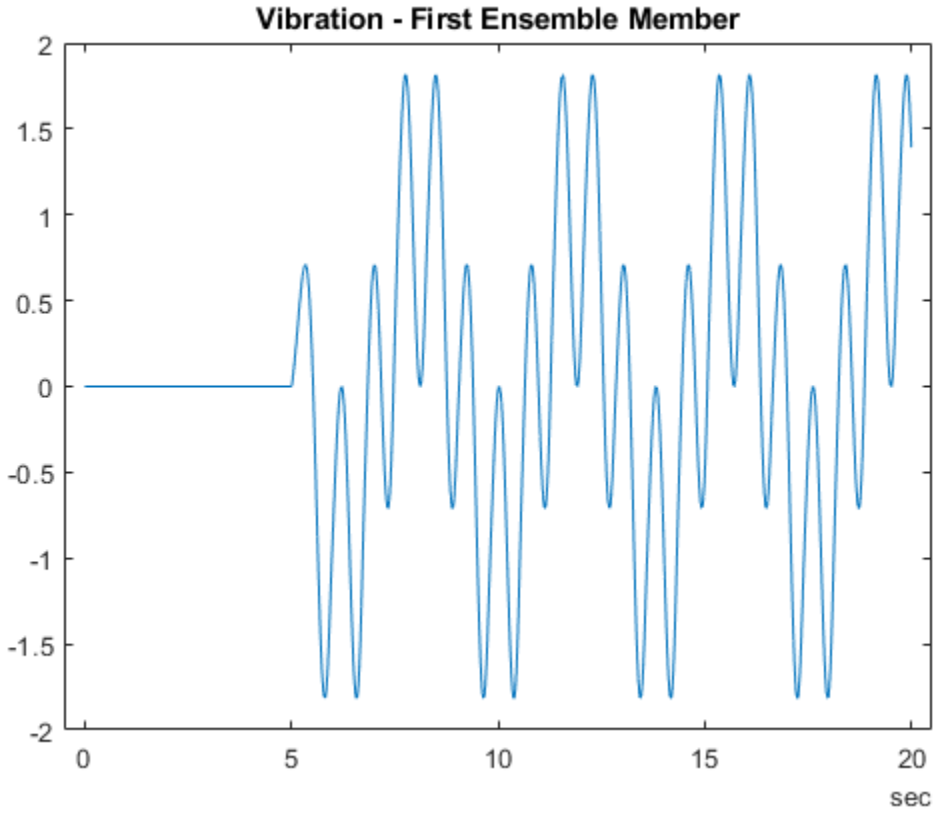
```
ensemble.SelectedVariables = ["Vibration";"SimulationInput"];
data1 = read(ensemble)
```

```
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” for more information.) 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'
```

```
data2.SimulationInput{1}.Variables
```

```
ans =
  Variable with properties:

         Name: 'ToothFaultGain'
        Value: -1.5000
    Workspace: 'global-workspace'
```

This result confirms that `data1` is from the ensemble with `ToothFaultGain` = –2, and `data2` is from the ensemble 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”.) Because of the volume of data, the `unzip` operation takes a few minutes.

```
% Create ensmeble datastore that points to datafiles in current folder
unzip fileEnsData.zip  % extract compressed files
location = pwd;
extension = '.mat';
fensemble = fileEnsembleDatastore(location,extension);

% Configure with functions for reading and writing variable data
```

```
addpath(fullfile(matlabroot,'examples','predmaint','main')) % Make sure functions are o
fensemble.DataVariablesFcn = @readBearingData;
fensemble.WriteToMemberFcn = @writeBearingData;

% Specify data and selected variables
fensemble.DataVariables = ["gs";"sr";"load";"rate"];
fensemble.SelectedVariables = ["gs";"load"];
```

Read the first member of the ensemble. The functions that you assigned tell the `read` and `writeToLastMemberRead` commands how to interact with the data files that make up the ensemble. Thus, when you call `read`, it reads all the variables named in `fensemble.SelectedVariables`. The read command uses `@readBearingData` to read selected variables that are in `fensemble.DataVariables`. For this example, `@readBearingData` extracts the data variables from a structure, `bearing`, that is stored in the file.

```
data = read(fensemble)
```

```
data=1×2 table
           gs            load
    _________________    ____

    [146484×1 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:

           DataVariablesFcn: @readBearingData
     ConditionVariablesFcn: []
   IndependentVariablesFcn: []
```

```
           WriteToMemberFcn: @writeBearingData
              DataVariables: [5×1 string]
        IndependentVariables: [0×0 string]
          ConditionVariables: [0×0 string]
           SelectedVariables: [2×1 string]
                  NumMembers: 5
             LastMemberRead: '\\fs-21-ah\home$\clevy\Documents\MATLAB\examples\predmai
```

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 uses `fensemble.WriteToMemberFcn` to write the table data to the file. In this example, `WriteToMemberFcn` is `writeBearingData`, a simple function that takes a data structure and adds it to whatever other data is already present in the data file.

```
newData = struct('gsMean',gsmean);
writeToLastMemberRead(fensemble,'gsMean',newData);
```

Calling read again advances the last-read-member indictor to the next file in the ensemble and reads the data from that file.

```
data = read(fensemble)
```

```
data=1×2 table
           gs             load
    ___________________    ____

    [146484×1 double]       50
```

You can see 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);
    newData = struct('gsMean',gsmean);
    writeToLastMemberRead(fensemble,'gsMean',newData);
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”.

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 = ["load";"gsMean"];
data1 = read(fensemble)
```

```
data1=1×2 table
    load       gsMean
    ____    ___________

     0      [1x1 struct]
```

```
data2 = read(fensemble)
```

```
data2=1×2 table
    load       gsMean
    ____    ___________

     50     [1x1 struct]
```

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

Input Arguments

`ensemble` — Ensemble datastore

`fileEnsembleDatastore` object | `simulationEnsembleDatastore` object

Ensemble datastore to read, specified as a:

- `simulationEnsembleDatastore` object.
- `fileEnsembleDatastore` object.

If `ensemble` is a `simulationEnsembleDatastore` object, then `read` returns a table row containing all the variables specified in `ensemble.SelectedVariables`.

If `ensemble` is a `fileEnsembleDatastore` object, then `read` uses the functions specified in the object properties as follows. In particular, `read` uses:

- `ensemble.DataVariablesFcn` to read variables specified in both the `SelectedVariables` and `DataVariables` properties of `ensemble`
- `ensemble.IndependentVariablesFcn` to read variables specified in both the `SelectedVariables` and `IndependentVariables` properties of `ensemble`
- `ensemble.ConditionVariablesFcn` to read variables specified in both the `SelectedVariables` and `ConditionVariables` properties of `ensemble`

Although you can specify different functions to process the data files for these different types of variables, `read` combines all the variables into one table row, `data`, containing all the variables specified in `ensemble.SelectedVariables`. For more information about working with file ensemble datastores, see `fileEnsembleDatastore`.

Output Arguments

`data` — Selected variables from ensemble member

`table` row

Selected variables from the ensemble member, returned as a `table` row. The table variables are the selected variables, and the table data are the values read from the ensemble data.

`info` — Data and member information

structure

Data and ensemble member information, returned as a structure with fields:

- `Size` — Dimensions of the table row `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.

See Also

`fileEnsembleDatastore` | `simulationEnsembleDatastore`

Topics

“Data Ensembles for Condition Monitoring and Predictive Maintenance”

Introduced in R2018a

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: 2.3567
    ThetaVariance: 0.0058
             Beta: 0.0721
     BetaVariance: 3.6363e-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 single 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`.

See Also

Functions
`exponentialDegradationModel` | `linearDegradationModel` | `update`

Topics
“Models for Predicting Remaining Useful Life”

Introduced in R2018a

tfmoment

Joint moment of the time-frequency distribution of a signal

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`

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

`momentJ = tfmoment(xt,order)` returns the “Joint Time-Frequency Moments” on page 1-101 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.

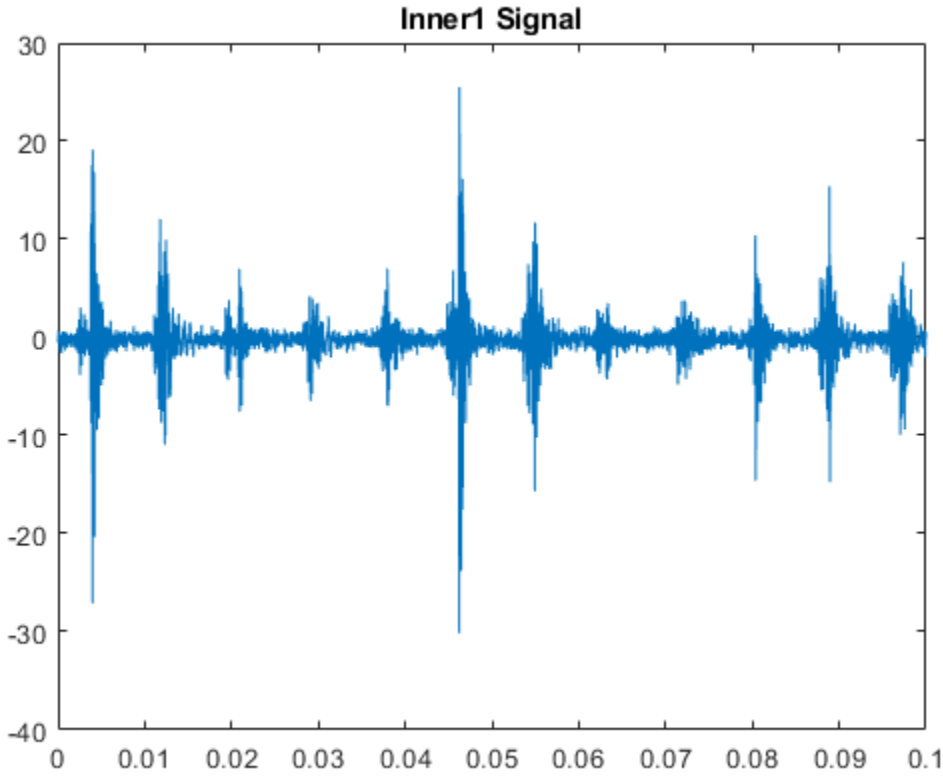
This example is adapted from “Rolling Element Bearing Fault Diagnosis”, which provides a more comprehensive treatment of the data sources and history.

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 ..
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 = 3.6261e+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 = 3.6261e+08
```

```
momentJ(2)
```

```
ans = 7.9513e+16
```

```
momentJ(3)
```

```
ans = 4.0896e+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” (MATLAB). `xt` can be nonuniformly sampled, with the `pspectrum` constraint that the median time interval and the mean time interval must obey:

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

`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 1-94.

`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 1-94.

`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 1-94.

`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 single 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`.

Definitions

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 $\left\langle t^n \omega^m \right\rangle$ of the signal using, for the noncentralized case:

   $$\left\langle t^n \omega^m \right\rangle = \int\int t^n \omega^m P(t,\omega) dt d\omega,$$

   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

   $$\mu_{t,\omega}^{n,m}(t) = \frac{1}{P(\omega)} \int\int \left(t - \left\langle t^1 \right\rangle_\omega \right)^n \left(\omega - \left\langle \omega^1 \right\rangle_t \right)^m P(t,\omega) dt d\omega,$$

   where $\left\langle t^1 \right\rangle_\omega$ and $\left\langle \omega^1 \right\rangle_t$ 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

Introduced in R2018a

tfsmoment

Conditional spectral moment of the time-frequency distribution of a signal

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`

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

`momentS = tfsmoment(xt,order)` returns the conditional spectral moment on page 1-126 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.

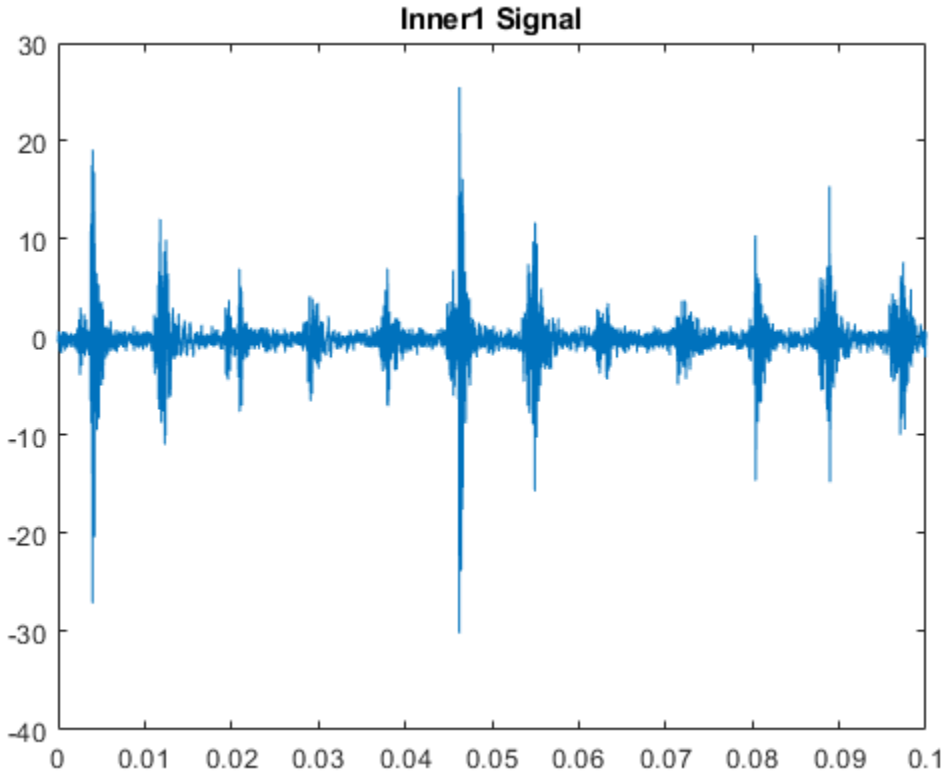
This example is adapted from “Rolling Element Bearing Fault Diagnosis”, which provides a more comprehensive treatment of the data sources and history.

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.

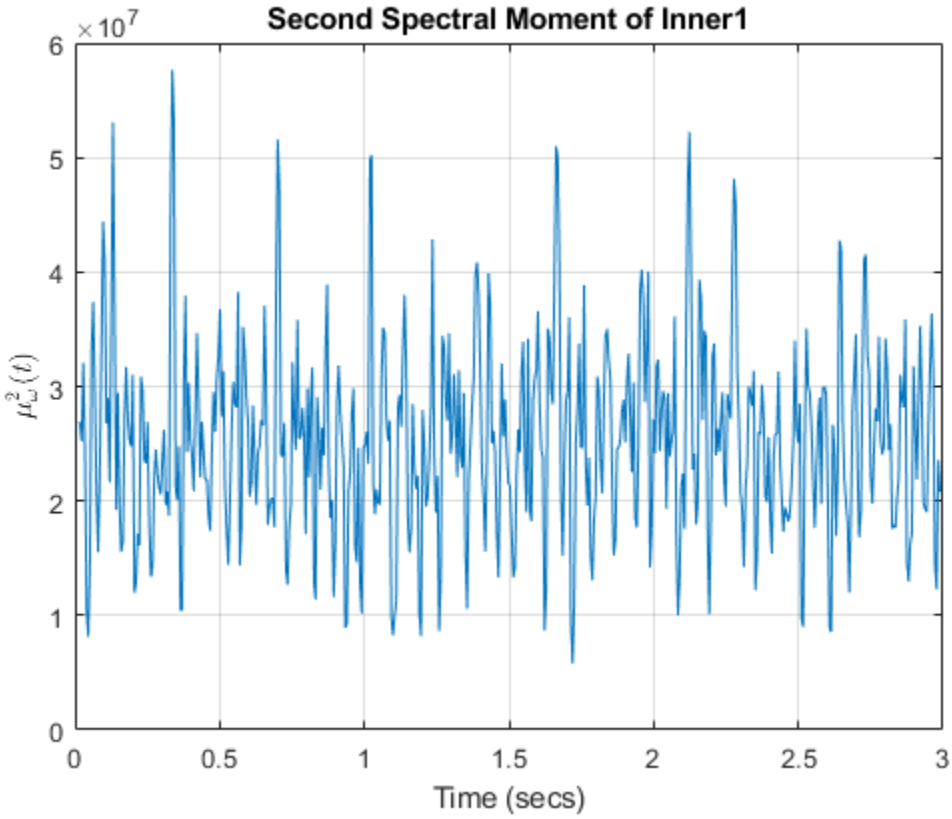
```
t_inner1 = (0:length(x_inner1)-1)/sr_inner1; % Construct time vector of [0 1/sr 2/sr .
figure
plot(t_inner1,x_inner1)
title ('Inner1 Signal')
hold on
xlim([0 0.1]) % Zoom in to an 0.1 s section
hold off
```



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')
```

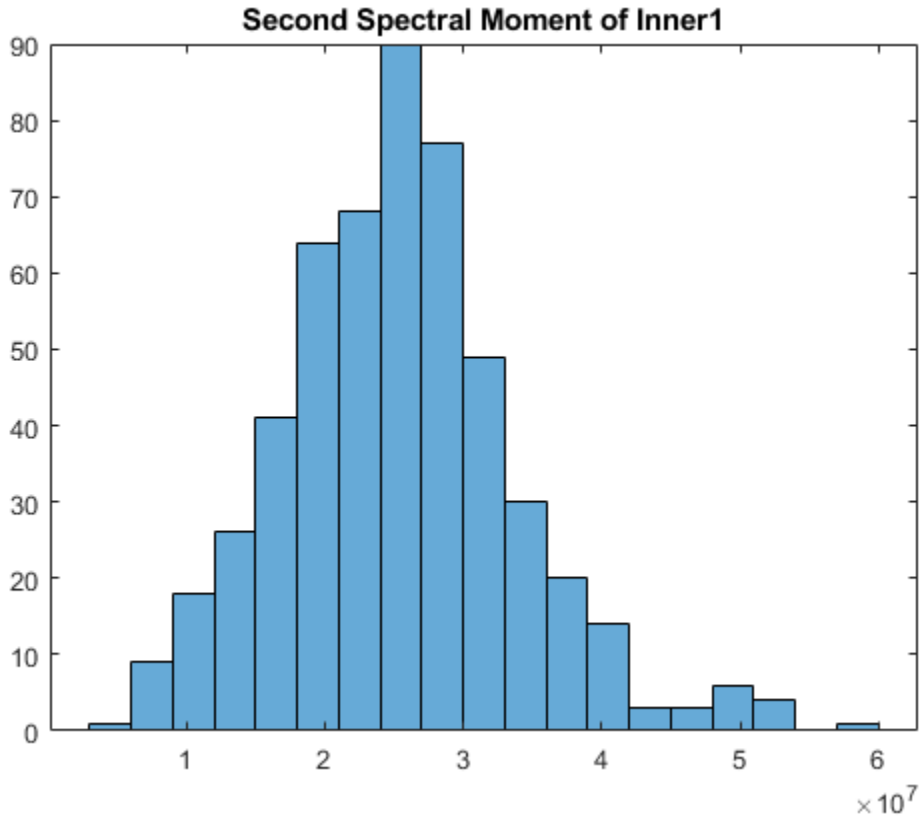


The plot illustrates the changes in the variance of the `x_inner1` spectrum over time. You are limited to this visualization (moment versus time) because `tfsmoment` returned no data. Now use `tfmoment` again to compute the second spectral moment, this time using the syntax that returns both the moment 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')
```



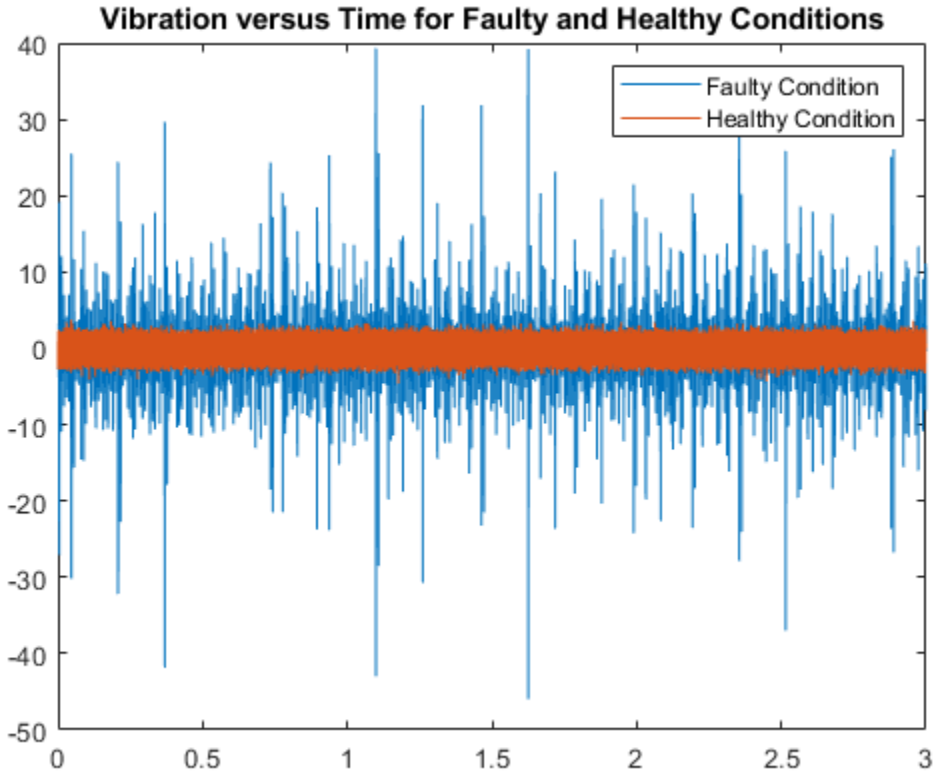
On its own, the histogram does not reveal obvious fault information. However, you can compare it to the histogram produced by the healthy-condition data.

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')
```

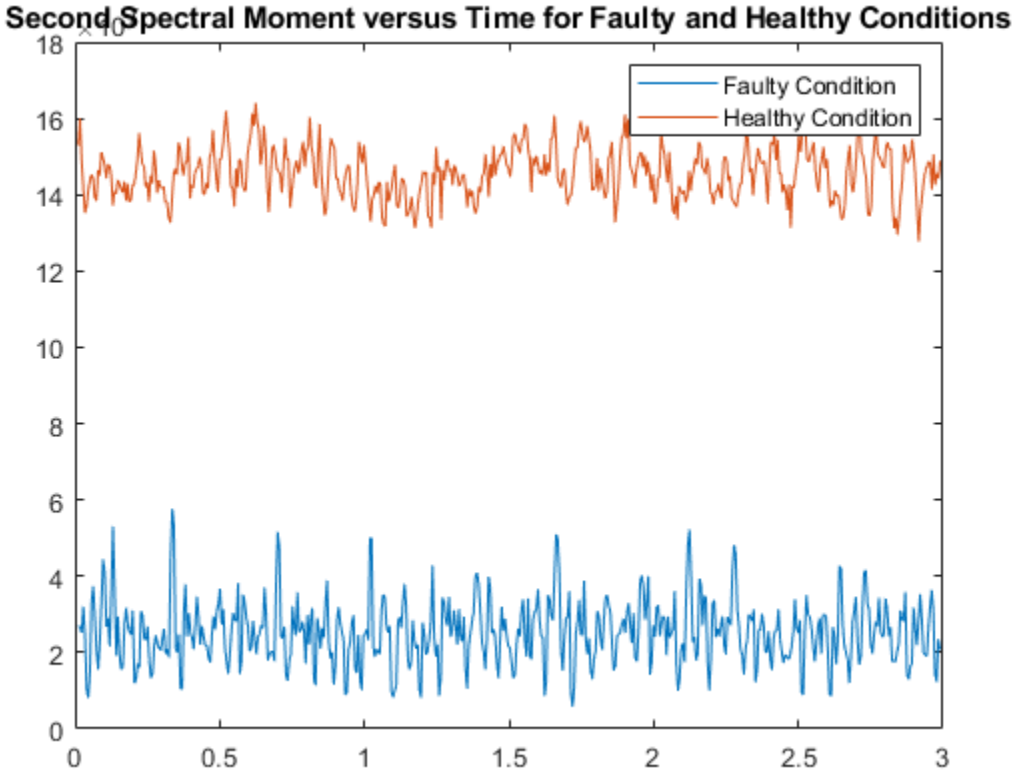


Calculate the second spectral moment of the `baseline1` data. Compare the `baseline1` and `inner1` time histories.

```
[momentS_baseline1,t1_baseline1] = tfsmoment(x_baseline1,sr_baseline1,2);

figure
plot(t1_inner1,momentS_inner1)
hold on
```

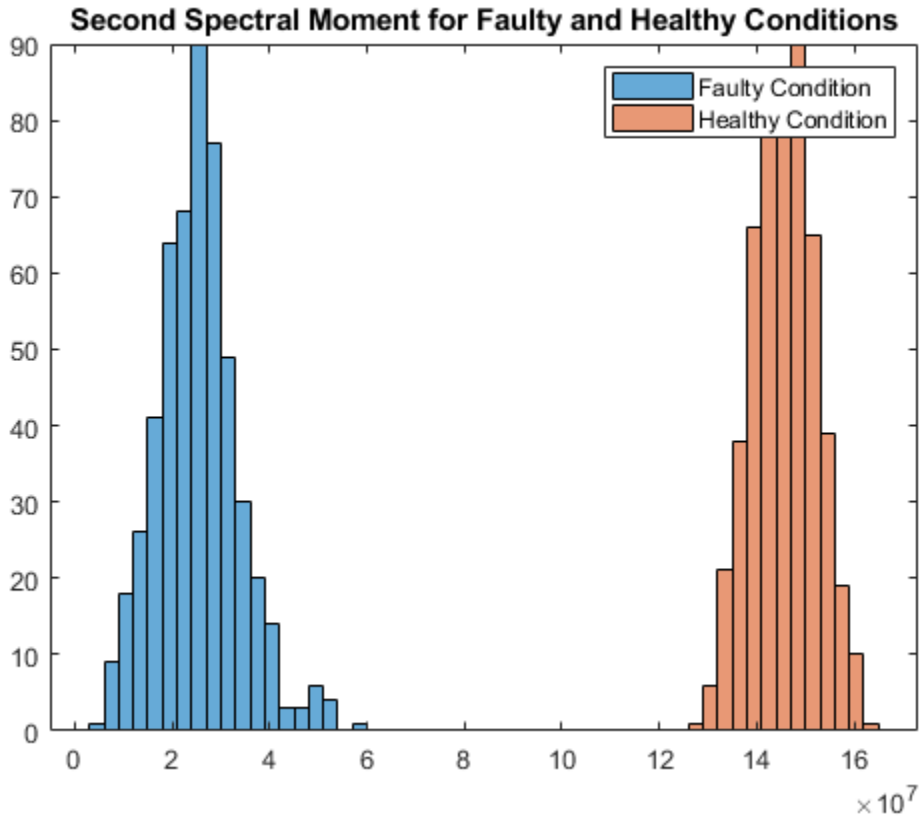
```
plot(t1_baseline1,momentS_baseline1)
hold off
legend('Faulty Condition','Healthy Condition')
title('Second Spectral Moment versus Time for Faulty and Healthy Conditions')
```



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)
```

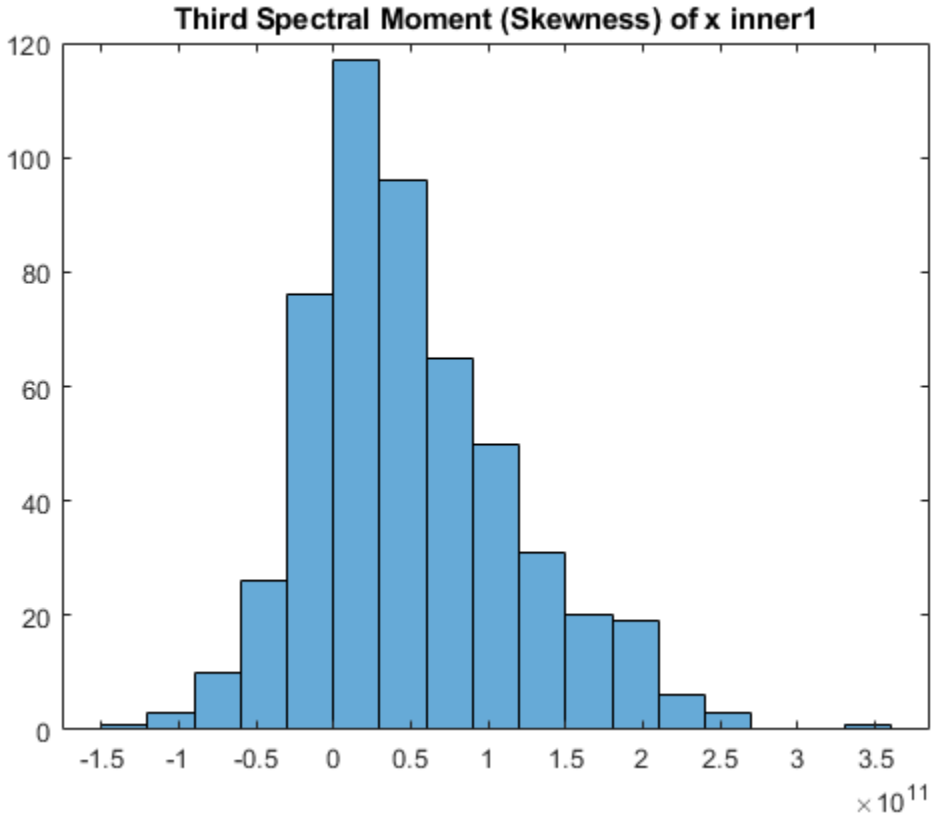
```
msize = 1×2

   524     4
```

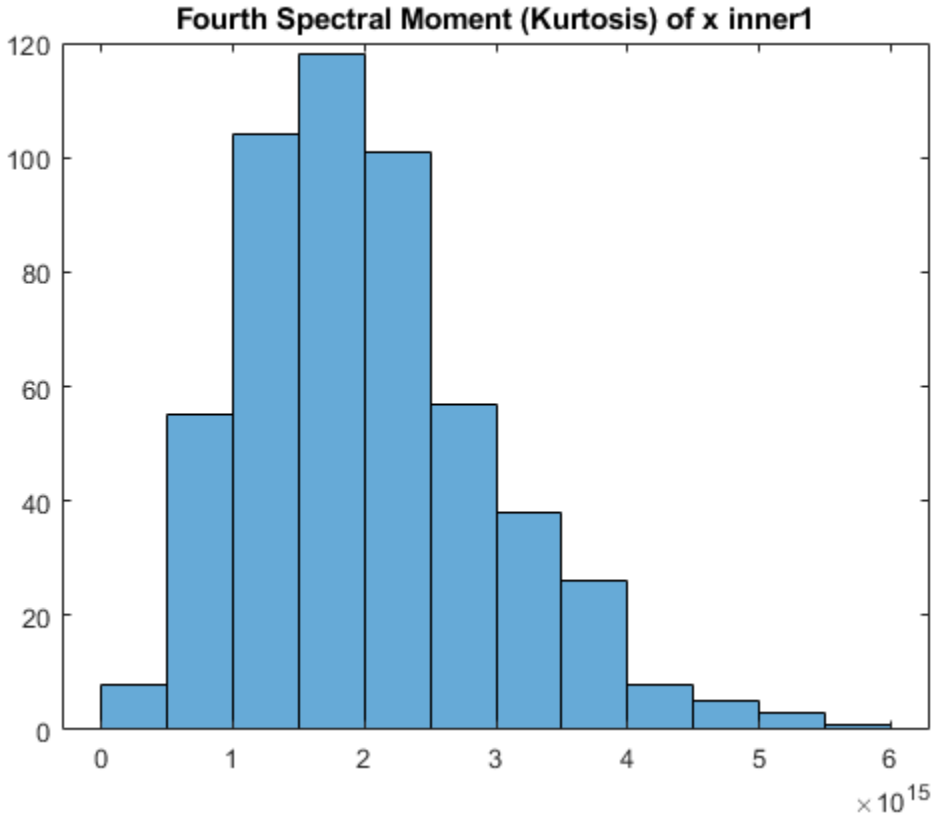
The data vector `x_inner` is considerably longer than the vectors in the moment matrix `momentS_inner1` because the spectrogram computation produces optimally-sized lower-resolution time windows. In this case, `tfsmoment` returns a moment matrix containing four columns, one column for each moment order.

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.

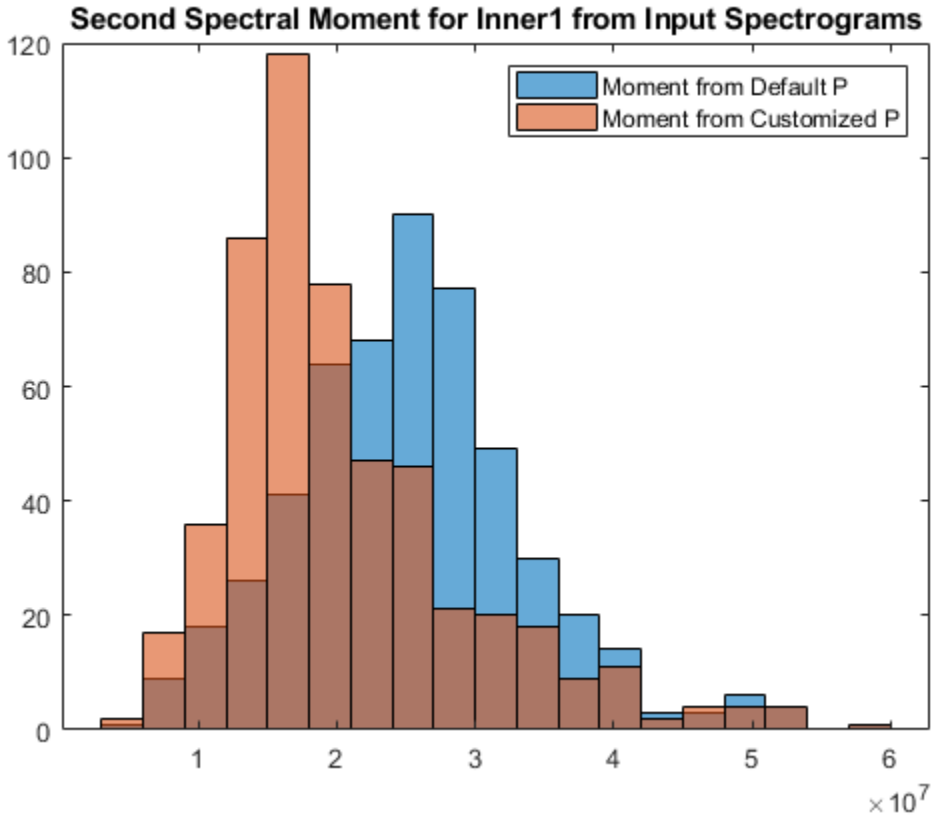
```
load tfmoment_data.mat p_inner1_def f_p_def t_p_def ...
    p_inner1_MinThr f_p_MinThr t_p_MinThr
load tfmoment_data.mat x_inner1 x_baseline1
```

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.

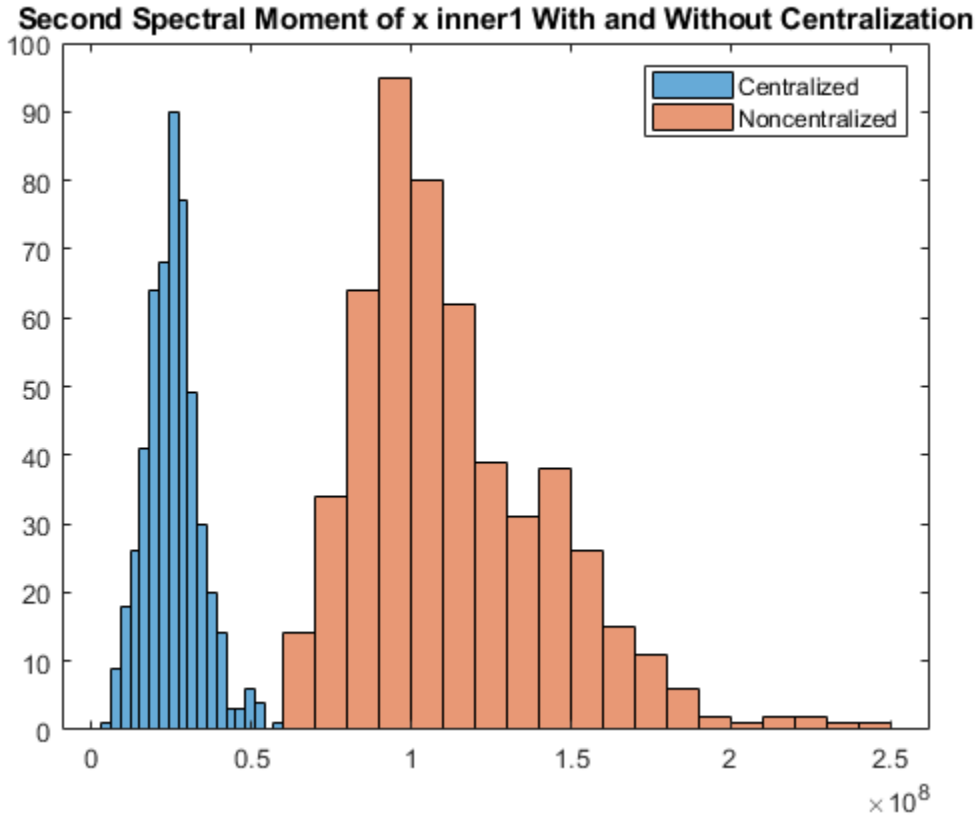
Calculate a Conditional Spectral Moment that is not Centralized

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 1-126. 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 1-105.

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 = struct with fields:
              Description: ''
                 UserData: []
           DimensionNames: {'Time'  'Variables'}
            VariableNames: {'x_inner1'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                 RowTimes: [146484x1 duration]
```

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
```

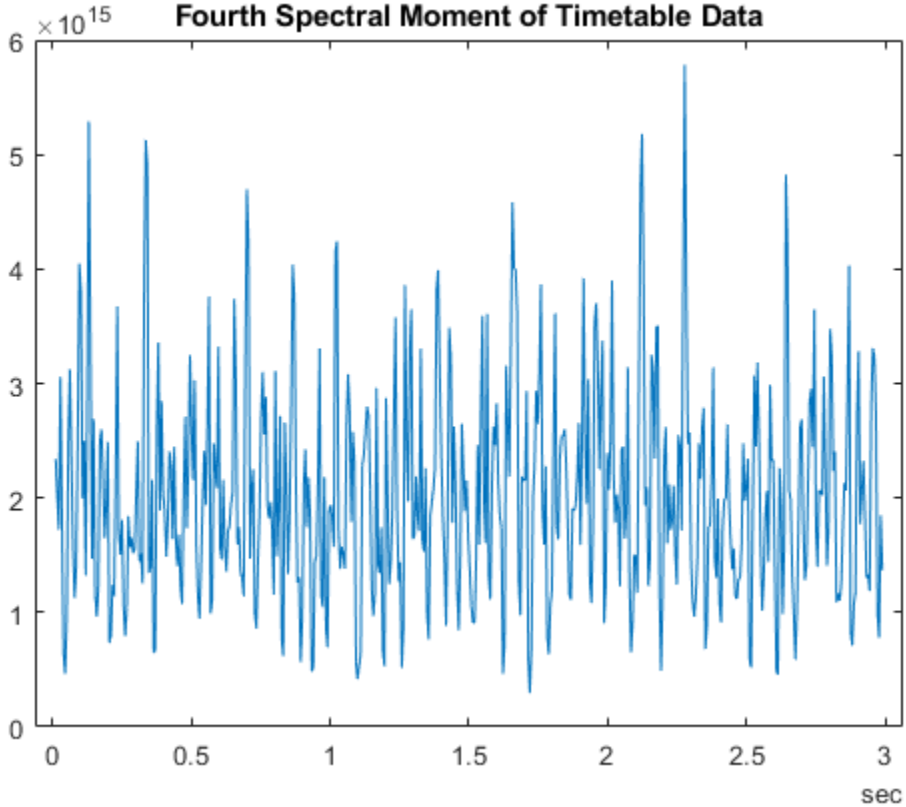
```
ans = struct with fields:
              Description: ''
                 UserData: []
           DimensionNames: {'Time'  'Variables'}
            VariableNames: {'CentralSpectralMoment2'  'CentralSpectralMoment4'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                 RowTimes: [524x1 duration]
```

The returned `timetable` represents the moments in the variable '`CentralSpectralMoment2`' and '`CentralSpectralMoment4`', providing information not only on what specific moment was calculated, but 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.

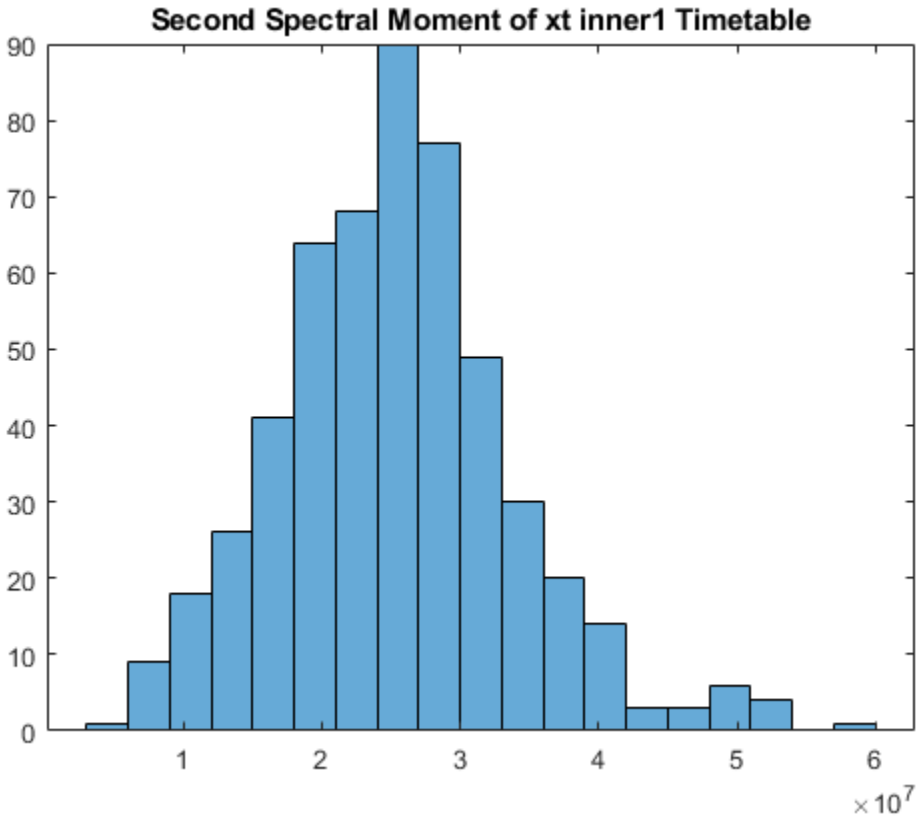
```
tt_inner1 = momentS_xt_inner1.Time;
momentS_inner1_2 = momentS_xt_inner1.CentralSpectralMoment2;
momentS_inner1_4 = momentS_xt_inner1.CentralSpectralMoment4;

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 1-105, 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” (MATLAB). `xt` can be nonuniformly sampled, with the `pspectrum` constraint that the median time interval and the mean time interval must obey.

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

For an example of timetable input, see “Find the Conditional Spectral Moments of Data Measurements in a Timetable” on page 1-118

`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 1-118
- Time-series vector data input — “Determine Multiple Orders of Conditional Spectral Moment for a Time Series” on page 1-111

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 1-105

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 1-114.

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 single 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 1-116.

`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 1-118.
- 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 1-111.

`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.

Definitions

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 $\left\langle \omega^m \right\rangle_t$ of the signal using, for the noncentralized case:

   $$\left\langle \omega^m \right\rangle_t = \frac{1}{P(t)} \int \omega^m P(t,\omega) d\omega,$$

   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

Introduced in R2018a

tftmoment

Conditional temporal moment of the time-frequency distribution of a signal

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 Toolboximplements the three branches of time-frequency moment:

- Conditional spectral moment — `tfsmoment`
- Conditional temporal moment — `tftmoment`
- Joint time-frequency moment — `tfmoment`

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

`momentT = tftmoment(xt,order)` returns the conditional temporal moment on page 1-143 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.

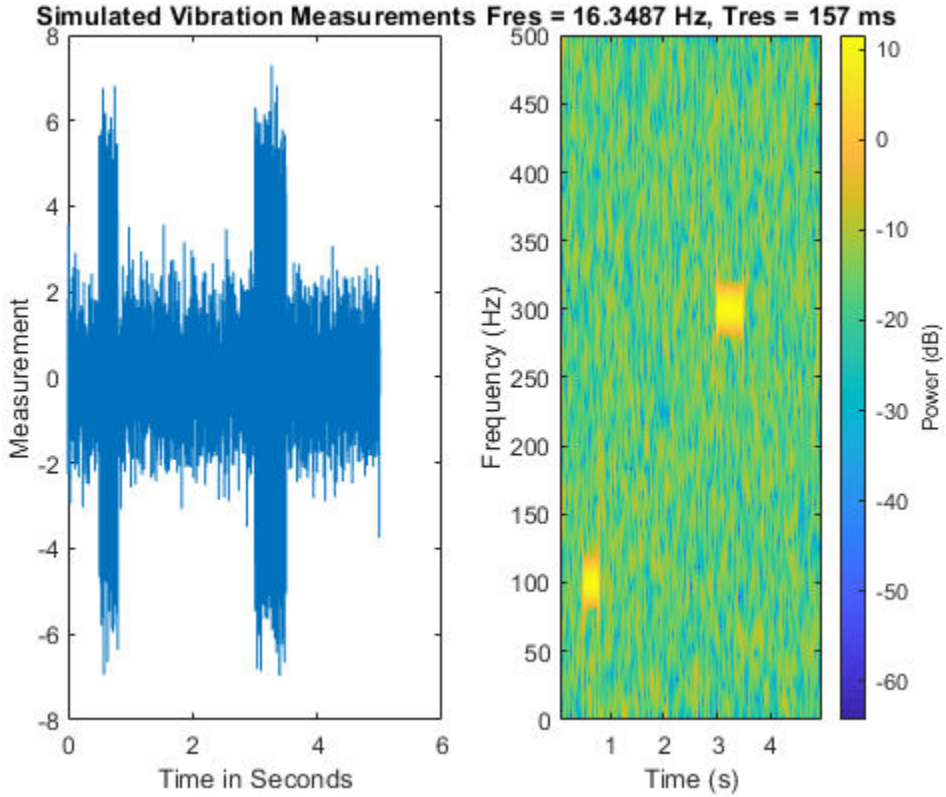
Load and plot the data, which consists of simulated vibration measurements for a system with a fault that causes periodic resonances. `x` is the vector of measurements, and `fs` is the sampling frequency.

```
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.

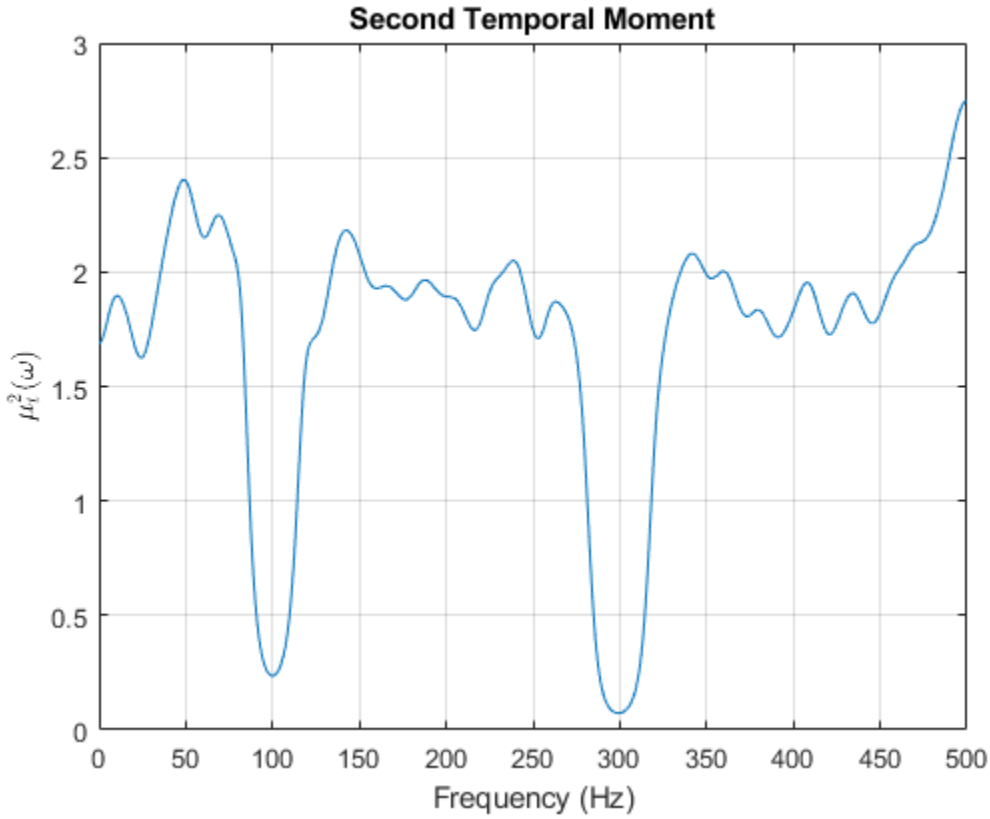
```
subplot(1,2,2)
pspectrum(x,ts,'spectrogram')
```



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);
```

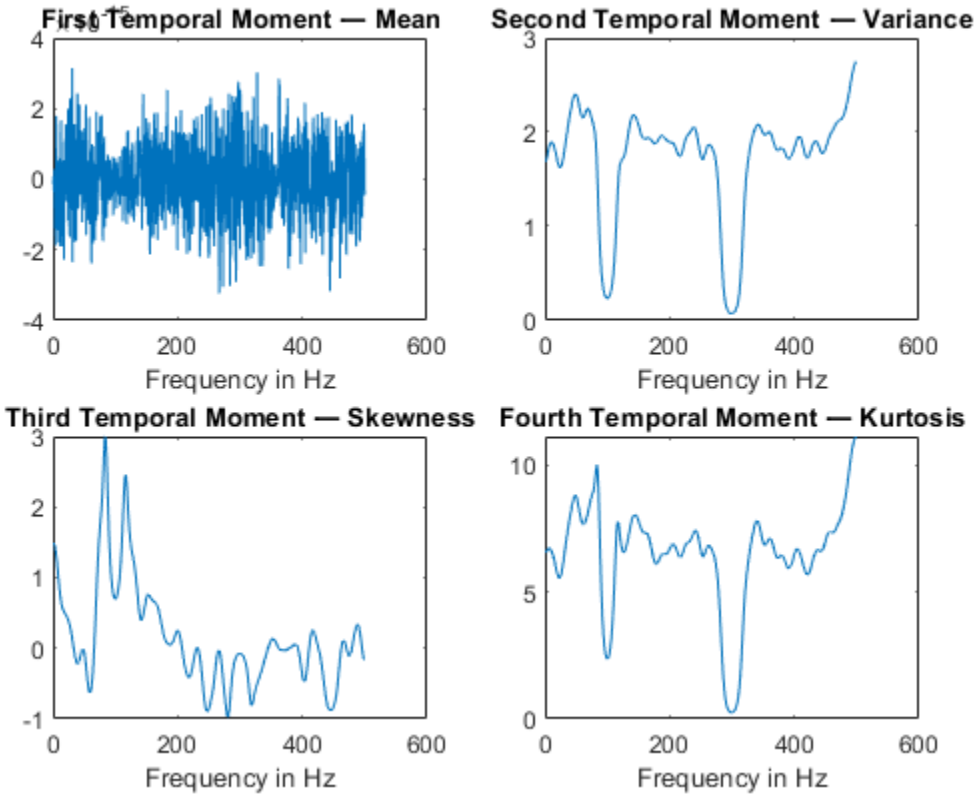
Plot the four moments separately to compare the shapes.

```
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,.

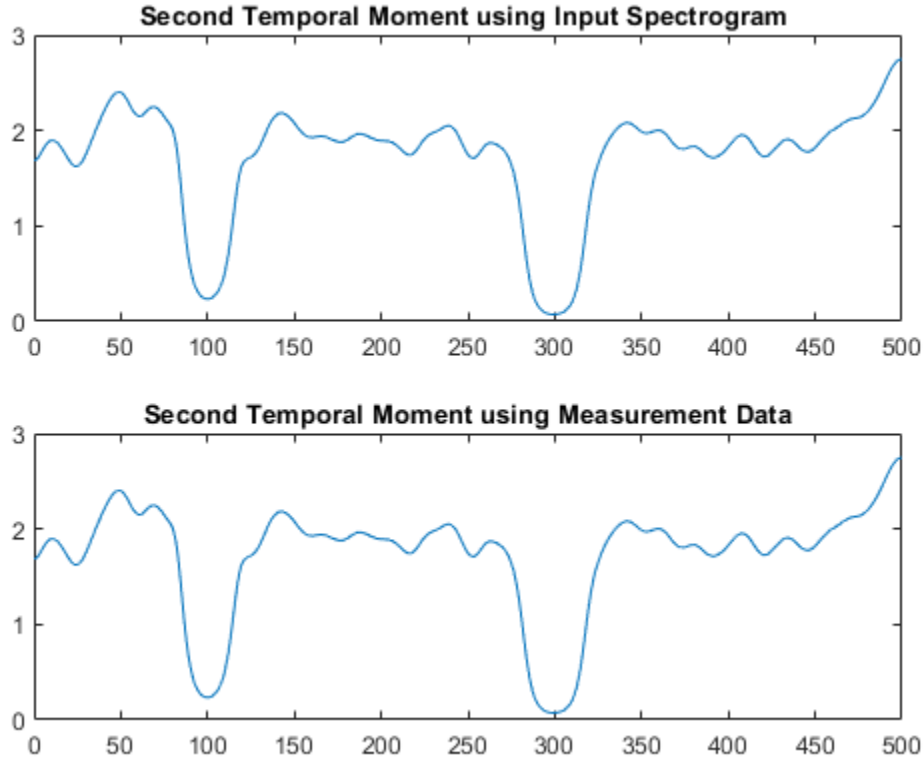
```
load tftmoment_example p fp tp x fs
```

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 1-130.

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 = struct with fields:
             Description: ''
                UserData: []
          DimensionNames: {'Time'  'Variables'}
           VariableNames: {'x_inner1'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowTimes: [146484x1 duration]
```

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.

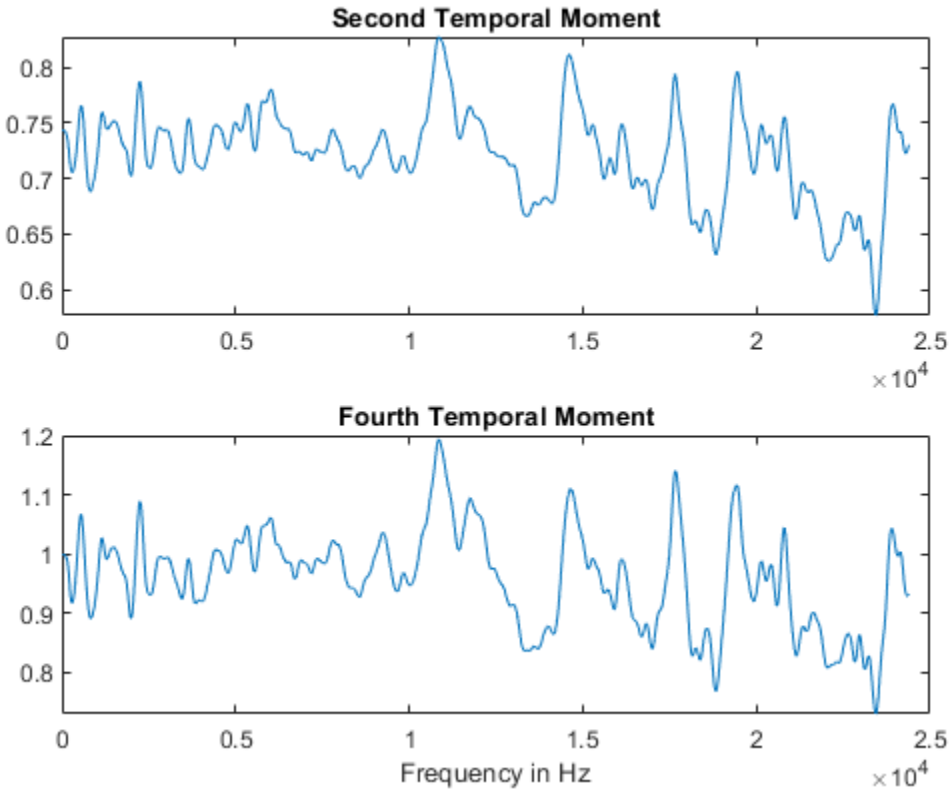
Plot the moments versus returned frequency vector f.

```
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” (MATLAB). `xt` can be nonuniformly sampled, with the `pspectrum` constraint that the median time interval and the mean time interval must obey:

$$\frac{1}{100} < \frac{\text{Median time interval}}{\text{Mean time interval}} < 100.$$

For an example of `timetable` input, see “Find the Conditional Temporal Moments of Data Measurements in a Timetable” on page 1-136

`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 1-136
- Time-series vector data input — “Plot the Conditional Temporal Moments of a Time Series Vector” on page 1-130

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 1-130

`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 1-114.

`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 single 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 1-130

Definitions

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 1-126 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 $\left\langle t^n \right\rangle_\omega$ of the signal using, for the non-centralized case:

   $$\left\langle t^n \right\rangle_\omega = \frac{1}{P(\omega)} \int t^n P(t,\omega) dt,$$

   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

Introduced in R2018a

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: 2.3567
    ThetaVariance: 0.0058
             Beta: 0.0721
     BetaVariance: 3.6363e-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`.

See Also

Functions
`exponentialDegradationModel` | `linearDegradationModel` | `update`

Topics
“Models for Predicting Remaining Useful Life”

Introduced in R2018a

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.

`writeToLastMemberRead(ensemble,data)` writes the data in a table row 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 takes several minutes.

```
unzip simEnsData.zip  % extract compressed files
ensemble = simulationEnsembleDatastore(pwd,'logsout')
```

```
ensemble =
  simulationEnsembleDatastore with properties:

           DataVariables: [6×1 string]
    IndependentVariables: [0×0 string]
      ConditionVariables: [0×0 string]
       SelectedVariables: [6×1 string]
              NumMembers: 5
         LastMemberRead: [0×0 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=1×6 table
           SimulationInput                    SimulationMetadata                  Tacho
    ______________________________    ____________________________________    ___________

    [1×1 Simulink.SimulationInput]    [1×1 Simulink.SimulationMetadata]    [20202×1 tim
```

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.

```
data.SimulationInput
```

```
ans = 1×1 cell array
    {1×1 Simulink.SimulationInput}
```

```
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 furtther 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.

```
ensemble.DataVariables = [ensemble.DataVariables; "ToothFault"];
ensemble.DataVariables
```

```
ans = 7×1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
    "xFinal"
    "xout"
    "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 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”.) Because of the volume of data, the `unzip` operation takes a few minutes.

```
% Create ensmeble datastore that points to datafiles in current folder
unzip fileEnsData.zip  % extract compressed files
location = pwd;
extension = '.mat';
fensemble = fileEnsembleDatastore(location,extension);

% Configure with functions for reading and writing variable data
addpath(fullfile(matlabroot,'examples','predmaint','main')) % Make sure functions are
fensemble.DataVariablesFcn = @readBearingData;
fensemble.WriteToMemberFcn = @writeBearingData;

% Specify data and selected variables
fensemble.DataVariables = ["gs";"sr";"load";"rate"];
fensemble.SelectedVariables = ["gs";"load"];
```

Read the first member of the ensemble. The functions that you assigned tell the `read` and `writeToLastMemberRead` commands how to interact with the data files that make up the ensemble. Thus, when you call `read`, it reads all the variables named in `fensemble.SelectedVariables`. The read command uses `@readBearingData` to read selected variables that are in `fensemble.DataVariables`. For this example, `@readBearingData` extracts the data variables from a structure, `bearing`, that is stored in the file.

```
data = read(fensemble)
```

```
data=1×2 table
           gs            load
    ________________    ____

    [146484×1 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:

           DataVariablesFcn: @readBearingData
      ConditionVariablesFcn: []
    IndependentVariablesFcn: []
           WriteToMemberFcn: @writeBearingData
              DataVariables: [5×1 string]
       IndependentVariables: [0×0 string]
         ConditionVariables: [0×0 string]
          SelectedVariables: [2×1 string]
                 NumMembers: 5
             LastMemberRead: '\\fs-21-ah\home$\clevy\Documents\MATLAB\examples\predmain
```

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 uses `fensemble.WriteToMemberFcn` to write the table data to the file. In this example, `WriteToMemberFcn` is `writeBearingData`, a simple function that takes a data structure and adds it to whatever other data is already present in the data file.

```
newData = struct('gsMean',gsmean);
writeToLastMemberRead(fensemble,'gsMean',newData);
```

Calling read again advances the last-read-member indictor to the next file in the ensemble and reads the data from that file.

```
data = read(fensemble)
```

```
data=1×2 table
           gs             load
    ____________________    ____

    [146484×1 double]       50
```

You can see 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);
    newData = struct('gsMean',gsmean);
    writeToLastMemberRead(fensemble,'gsMean',newData);
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”.

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 = ["load";"gsMean"];
data1 = read(fensemble)
```

```
data1=1×2 table
    load       gsMean
    ____    ___________

     0      [1x1 struct]
```

```
data2 = read(fensemble)
```

```
data2=1×2 table
    load       gsMean
```

```
    ____    ____________

     50     [1x1 struct]
```

```
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` row

New data to write to the current ensemble member, specified as a `table` row. 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'});
```

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

`fileEnsembleDatastore` | `read` | `simulationEnsembleDatastore`

Topics

“Data Ensembles for Condition Monitoring and Predictive Maintenance”

Introduced in R2018a

Objects — Alphabetical List

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 are 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 2-11.

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. Once you configure the parameters of your covariate survival model, you can then predict the remaining useful life of similar components using `predictRUL`.

If you only have life span measurements and do not have covariate information, use a `reliabilitySurvivalModel`.

For more 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 life time 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 survivor functions values, and the first column contains the corresponding life time values.

For more information on the survival model, see “Proportional Hazard Survival Model” on page 2-11.

`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” (Statistics and Machine Learning Toolbox).

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 2-11.

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

Life time variable, specified as a string that contains a valid MATLAB variable name. For survival models, the life time 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` — Life time variable units
`""` (default) | value

Life time variable units, specified as a string.

The units of the life time 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 survivor 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;
```

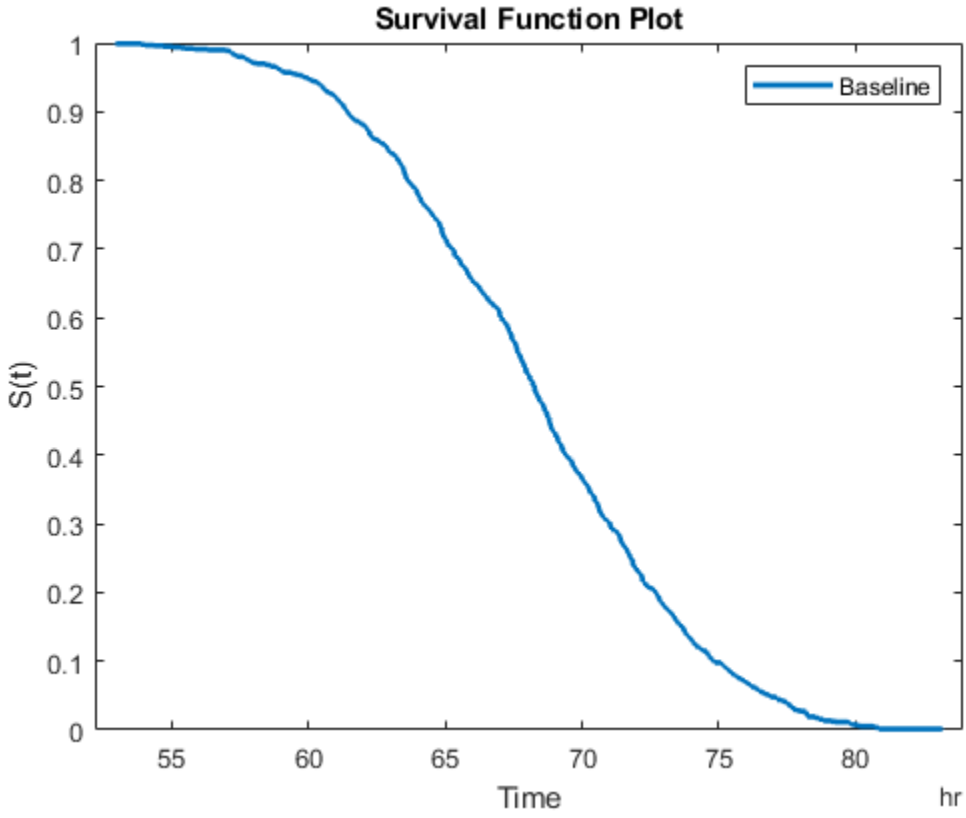
Train the survival model using the training data, specifying the life time variable, data variables, and encoded variable. There is no censor variable for this training data.

```
fit(mdl,covariateData,"DischargeTime",["Temperature","Load","Manufacturer"],[],"Manufac
```

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

Plot the baseline survivor 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',"Manufactu
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(TestBatteryLoad,TestAmbientTemperature,'B','RowTimes',hours(30));
TestData.Properties.VariableNames = {'Temperature','Load','Manufacturer'};
TestData.Properties.DimensionNames{1} = 'DischargeTime';
```

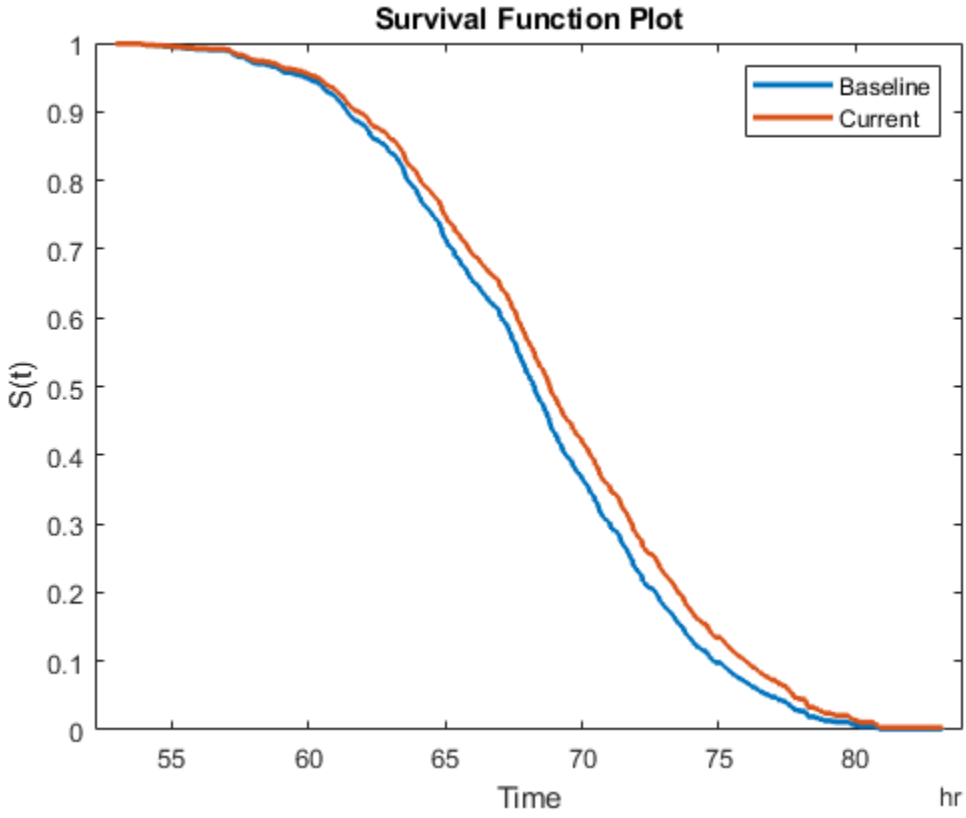
Predict the RUL for the battery.

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

```
estRUL = duration
   38.657 hr
```

Plot the survivor 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 proportional hazard models, see “Cox Proportional Hazards Model” (Statistics and Machine Learning Toolbox).

See Also

Functions
`coxphfit` | `reliabilitySurvivalModel`

Topics

“Models for Predicting Remaining Useful Life”
“Cox Proportional Hazards Model” (Statistics and Machine Learning Toolbox)

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 2-24.

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 more 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 2-24.

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 2-24.

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 2-24.

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 2-24.

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 2-24.

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 2-24.

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 ofβ
- `Rho` — Correlation between θ and β.

You can specify the fields of `Prior`:

- When you create the model. When you specify `Theta`, `ThetaVariance`, `Theta`, `ThetaVariance`, 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`, `Theta`, `ThetaVariance`, 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 2-24.

`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 2-24.

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.

`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 life time variable value
scalar | `duration` object

This property is read-only.

Initial life time 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 life time variable value
scalar | `duration` object

This property is read-only.

Latest life time variable value supplied to the `update` function, specified as a scalar.

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

Life time 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` — Life time variable units
`""` (default) | value

Life time variable units, specified as a string.

The units of the life time 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 ditribution 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 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 after 150 hours. Predict the remaining useful life of the component at this time using the trained exponential degradation model. The RUL is the forecasted time at which the degradation feature will pass the specified threshold.

```
estRUL = predictRUL(mdl,expTestData(150,:),threshold)
```

```
estRUL = duration
   129.73 hr
```

The estimated RUL is around `130` hours, which indicates a total predicted life span of `280` 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
   234.56 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:

$$S(t) = \phi + \theta(t)e^{\left(\beta(t)t+\varepsilon(t)-\frac{\sigma}{2}\right)}$$

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 bivariate Gaussian distribution with mean `Theta` and variance `ThetaVariance`.
- $\beta(t)$ is a random variable modeled as a bivariate 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 is equal to `NoiseVariance`.

See Also

Functions
`linearDegradationModel`

Topics
“Models for Predicting Remaining Useful Life”

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 custom 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 different types of variables.

`fensemble = fileEnsembleDatastore(location,extension,Name,Value)` specifies additional properties on page 2-26 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 from which to read ensemble data, specified as a string or a character vector, and specifying the directory where all the data is stored. 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 “Read Remote Data” (MATLAB).

Example: `pwd + "\simResults"`

`extension` — File extension

string | character vector

File extension for files in the data store, specified as a string or a character vector.

Example: `".mat",".csv"`

Properties

`DataVariablesFcn` — Function for reading data variables

`[]` (default) | function handle

Function for reading data variables from the ensemble, specified as a handle to a function you provide. You write a function that instructs the software how to read data 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 data variable

The function handle you use for data variables can be the same as or different from the ones you use for condition variables and independent variables.

For example, suppose that you write the following function, `readDataVars`, for reading data variables from your files. This function creates a data table containing the variables in a data file that match those in the input string, `variables`.

```
function data = readDataVars(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 this property as follows.

```
fensemble.DataVariablesFcn = @readDataVars;
```

When you call `read(ensemble)`, the software uses `readDataVars` to read any variables in the `SelectedVariables` property of the ensemble that are also named in `DataVariables`.

Your function can include any MATLAB command for reading data from files, such as `csvread` (comma-separated values), `xlsread` Microsoft® Excel® spreadsheets, or other data import commands.

You must set this property to read data variables from a `fileEnsembleDatastore` object. Otherwise, `read(ensemble)` generates an error.

`ConditionVariablesFcn` — Function for reading condition variables
`[ ]` (default) | function handle

Function for reading condition variables from the ensemble, specified as a handle to a function you provide. You write a function that instructs the software how to read

condition 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 condition variable

The function handle you use for condition variables can be the same as or different from the ones you use for data variables and independent variables.

For example, suppose that you write a function called `readCondVars` for reading condition variables from your files. Similarly to the `DataVariablesFcn` property, store the function in a MATLAB file in the current folder or on the path. Then, if you create a `fileEnsembleDatastore` called `fensemble`, set `ConditionVariablesFcn` as follows.

```
fensemble.ConditionVariablesFcn = @readCondVars;
```

When you call `read(ensemble)`, the software uses `readCondVars` to read any variables in the `SelectedVariables` property of the ensemble that are also named in `ConditionVariables`.

You must set this property to read condition variables from a `fileEnsembleDatastore` object. Otherwise, `read(ensemble)` generates an error.

`IndependentVariablesFcn` — Function for reading independent variables
`[ ]` (default) | function handle

Function for reading independent variables from the ensemble, specified as a handle to a function you provide. You write a function that instructs the software how to read independent 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

The function handle you use for independent variables can be the same as or different from the ones you use for condition variables and data variables.

For example, suppose that you write a function called `readIndVars` for reading independent variables from your files. Similarly to the `DataVariablesFcn` property, store the function in a MATLAB file in the current folder or on the path. Then, if you

create a `fileEnsembleDatastore` called `fensemble`, set `IndependentVariablesFcn` as follows.

```
fensemble.IndependentVariablesFcn = @readIndVars;
```

When you call `read(ensemble)`, the software uses `readDataVars` to read any variables in the `SelectedVariables` property of the ensemble that are also named in `IndependentVariables`.

You must set this property to read independent variables from a `fileEnsembleDatastore` object. Otherwise, `read(ensemble)` 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 filen ame (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 to the specified data file.

```
function writeNewData(filename,data)
save(filename, '-append', '-struct', 'structData');
end
```

Store `writeNewData` in a MATLAB file in the current folder or on the path. Then, if you create a `fileEnsembleDatastore` called `fensemble`, set `AddVariablesFcn` as follows:

```
fensemble.AddVariablesFcn = @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.

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.

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.

You can also specify `ConditionVariables` using a cell array of character vectors, such as `{'Gear Fault';'Temperature'}`, but the variable names are always stored as a string array, `["Gear Fault";"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, `ensemble`, that contains six variables, and you want to read only two of them, `Vibration` and `Fault State`. Set the `SelectedVariables` property and call `read`:

```
ensemble.SelectedVariables = ["Vibration";"Fault State"];
data = read(ensemble)
```

`SelectedVariables` must be a subset 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.

`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

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. When you call `writeToLastMemberRead` to add data back to the ensemble datastore, that function writes to the last member read. The `LastMemberRead` property contains the path to the file to which `writeToLastMemberRead` writes.

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.

read	Read member data from an ensemble datastore
writeToLastMemberRead	Write data to member of an ensemble datastore
reset	Reset datastore to initial state
hasdata	Determine if data is available to read
progress	Determine how much data has been read
numpartitions	Number of datastore partitions
partition	Partition a datastore
tall	Create tall array

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`. (Because of the volume of data, the `unzip` operation takes several minutes.) In practice you might have many more data files.

```
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 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 functions:

- `readBearingData` — Parse the data in the file, and return a table row containing a table variable for each data variable in the file.

- `writeBearingData` — Write a table row to a data file.

For more information about these functions, see “File Ensemble Datastore With Measured Data”.

```
addpath(fullfile(matlabroot,'examples','predmaint','main')) % Make sure functions are
```

```
fensemble.DataVariablesFcn = @readBearingData;
fensemble.WriteToMemberFcn = @writeBearingData;
```

Finally, set properties of the ensemble to identify the data variables and the 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 `gs` and `sr`. Set these two variables as the selected variables.

```
fensemble.DataVariables = ["gs";"sr";"load";"rate"];
fensemble.SelectedVariables = ["gs";"sr"];
```

Examine the ensemble. The functions and the variable names are assigned to the appropriate properties.

```
fensemble
```

```
fensemble =
  fileEnsembleDatastore with properties:

           DataVariablesFcn: @readBearingData
      ConditionVariablesFcn: []
    IndependentVariablesFcn: []
          WriteToMemberFcn: @writeBearingData
             DataVariables: [4x1 string]
       IndependentVariables: [0x0 string]
         ConditionVariables: [0x0 string]
          SelectedVariables: [2x1 string]
                 NumMembers: 5
            LastMemberRead: [0x0 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 `read`, it reads all the variables named in `fensemble.SelectedVariables`. The read command uses `@readBearingData` to read selected variables that are in `fensemble.DataVariables.` For more information, see “File Ensemble Datastore With Measured Data”.

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

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”.) Because of the volume of data, the `unzip` operation takes a few minutes.

```
% Create ensmeble datastore that points to datafiles in current folder
unzip fileEnsData.zip  % extract compressed files
location = pwd;
extension = '.mat';
fensemble = fileEnsembleDatastore(location,extension);

% Configure with functions for reading and writing variable data
addpath(fullfile(matlabroot,'examples','predmaint','main')) % Make sure functions are
fensemble.DataVariablesFcn = @readBearingData;
fensemble.WriteToMemberFcn = @writeBearingData;

% Specify data and selected variables
fensemble.DataVariables = ["gs";"sr";"load";"rate"];
fensemble.SelectedVariables = ["gs";"load"];
```

Read the first member of the ensemble. The functions that you assigned tell the `read` and `writeToLastMemberRead` commands how to interact with the data files that make up the ensemble. Thus, when you call `read`, it reads all the variables named in `fensemble.SelectedVariables`. The read command uses `@readBearingData` to read selected variables that are in `fensemble.DataVariables`. For this example, `@readBearingData` extracts the data variables from a structure, `bearing`, that is stored in the file.

```
data = read(fensemble)
```

```
data=1×2 table
           gs            load
    ____________________    ____

    [146484×1 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:

           DataVariablesFcn: @readBearingData
      ConditionVariablesFcn: []
    IndependentVariablesFcn: []
           WriteToMemberFcn: @writeBearingData
              DataVariables: [5×1 string]
       IndependentVariables: [0×0 string]
         ConditionVariables: [0×0 string]
          SelectedVariables: [2×1 string]
                 NumMembers: 5
            LastMemberRead: '\\fs-21-ah\home$\clevy\Documents\MATLAB\examples\predmai
```

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 uses `fensemble.WriteToMemberFcn` to write the table data to the file. In this example, `WriteToMemberFcn` is `writeBearingData`, a simple function that takes a data structure and adds it to whatever other data is already present in the data file.

```
newData = struct('gsMean',gsmean);
writeToLastMemberRead(fensemble,'gsMean',newData);
```

Calling read again advances the last-read-member indictor to the next file in the ensemble and reads the data from that file.

```
data = read(fensemble)
```

```
data=1×2 table
           gs            load
```

```
    ________________    ____

    [146484×1 double]    50
```

You can see 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);
    newData = struct('gsMean',gsmean);
    writeToLastMemberRead(fensemble,'gsMean',newData);
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”.

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 = ["load";"gsMean"];
data1 = read(fensemble)

data1=1×2 table
    load       gsMean
```

```
    ____    ____________

     0      [1x1 struct]
```

```
data2 = read(fensemble)
```

```
data2=1×2 table
    load       gsMean
    ____    ____________

     50     [1x1 struct]
```

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

See Also

`generateSimulationEnsemble` | `simulationEnsembleDatastore`

Topics

“Data Ensembles for Condition Monitoring and Predictive Maintenance”
“File Ensemble Datastore With Measured Data”

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 life time span of the most similar components minus the current life time value of the test component.

For more 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 life time 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 `j`th feature computed for the `i`th 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` — Life time variable

`""` (default) | string

Life time 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` — Life time variable units

`""` (default) | value

Life time variable units, specified as a string.

The units of the life time 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.

See Also

Functions
`pairwiseSimilarityModel` | `residualSimilarityModel`

Topics
“Models for Predicting Remaining Useful Life”

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 2-56.

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 more 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 2-56.

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 2-56.

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 2-56.

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 2-56.

`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 2-56.

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 life time variable value

scalar | `duration` object

This property is read-only.

Initial life time 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 life time variable value

scalar | `duration` object

This property is read-only.

Latest life time variable value supplied to the `update` function, specified as a scalar.

`LifeTimeVariable` — Life time variable

`""` (default) | string

Life time 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` — Life time variable units

`""` (default) | value

Life time variable units, specified as a string.

The units of the life time 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 ditribution 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 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
   69.988 hr
```

The estimated RUL is around `70` hours, which indicates a total predicted life span of `118` 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
   59.406 hr
```

The estimated RUL is about `60` hours, which indicates a total predicted life span of `110` hours.

Algorithms

Linear Degradation Model

The`linearDegradationModel` object implements the following continuous-time linear degradation model:

$$S(t) = \phi + \theta(t)t + \varepsilon(t)$$

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`.

See Also

Functions
`exponentialDegradationModel`

Topics
"Models for Predicting Remaining Useful Life"

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 life time span of the most similar components minus the current life time value of the test component.

For more 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 life time 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` — Life time span of historical data
`Inf` (default) | positive scalar | `duration` object

Life time span of historical data for computing similarity, specified as a positive scalar or `duration` object. When computing similarity, the model uses historical data from life time (*t*-`HistorySpan`) to life time *t*, where *t* is the current life time.

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-life time 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` — Life time variable

" " (default) | string

Life time 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` — Life time variable units

" " (default) | value

Life time variable units, specified as a string.

The units of the life time 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 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',"hou
```

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.

See Also

Functions

`hashSimilarityModel` | `residualSimilarityModel`

Topics

“Models for Predicting Remaining Useful Life”

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 more 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 life time 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 life time distribution, specified as one of the following:

Distribution String	Distribution Object
`"BirnbaumSaunders"`	`BirnbaumSaundersDistribution`
`"Exponential"`	`ExponentialDistribution`
`"Gamma"`	`GammaDistribution`
`"GeneralizedPareto"`	`GeneralizedParetoDistribution`
`"HalfNormal"`	`HalfNormalDistribution`
`"InverseGaussian"`	`InverseGaussianDistribution`

Distribution String	Distribution Object
"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` — Life time variable

" " (default) | string

Life time variable, specified as a string that contains a valid MATLAB variable name. For survival models, the life time 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` — Life time variable units

" " (default) | value

Life time variable units, specified as a string.

The units of the life time 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.

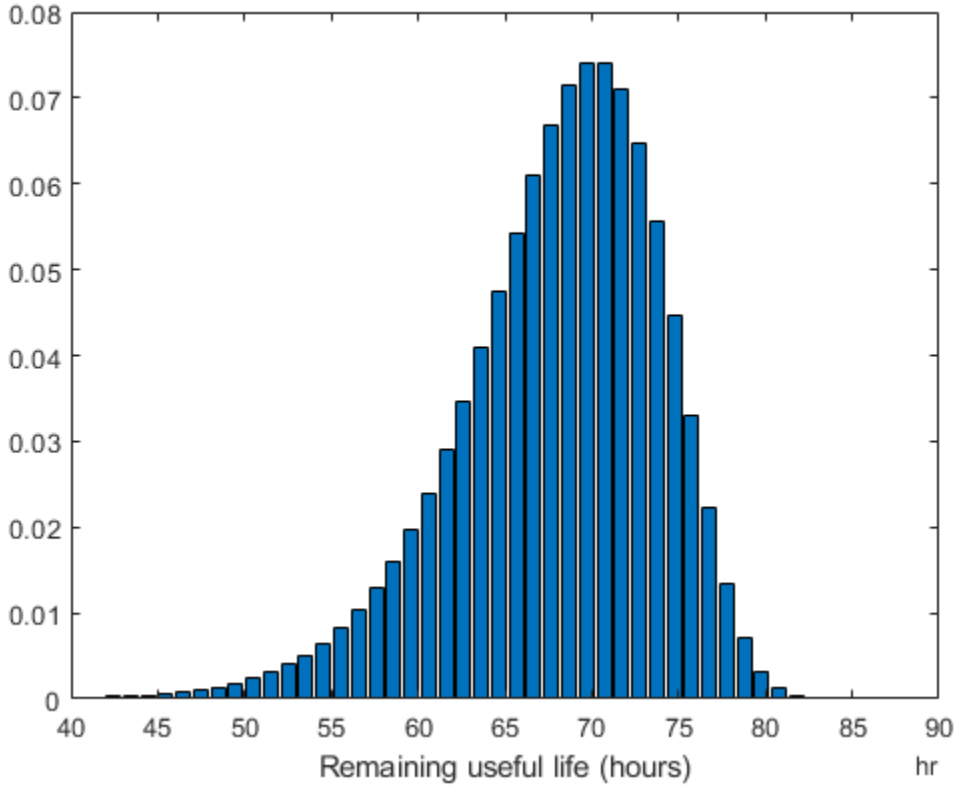
```
fit(mdl,reliabilityData)
```

Predict the life span of a new component and obtain the probability distribution function for the estimate.

```
[estRUL,ciRUL,pdfRUL] = predictRUL(mdl);
```

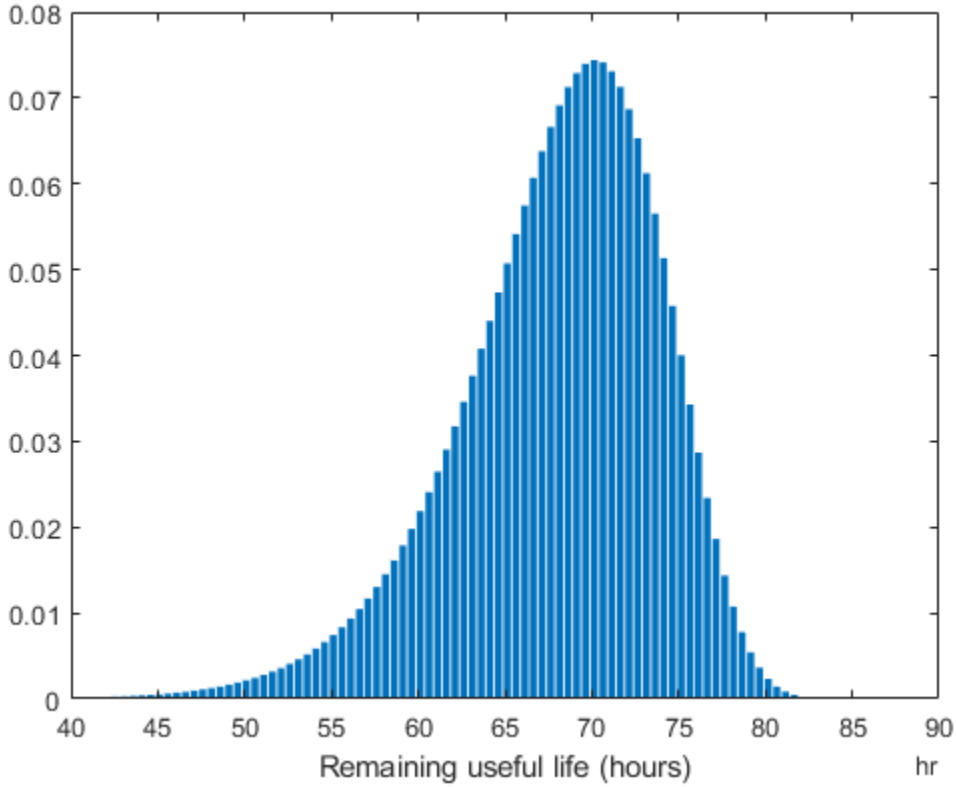
Plot the probability distribution.

```
bar(pdfRUL.RUL,pdfRUL.ProbabilityDensity)
xlabel('Remaining useful life (hours)')
xlim(hours([40 90]))
```



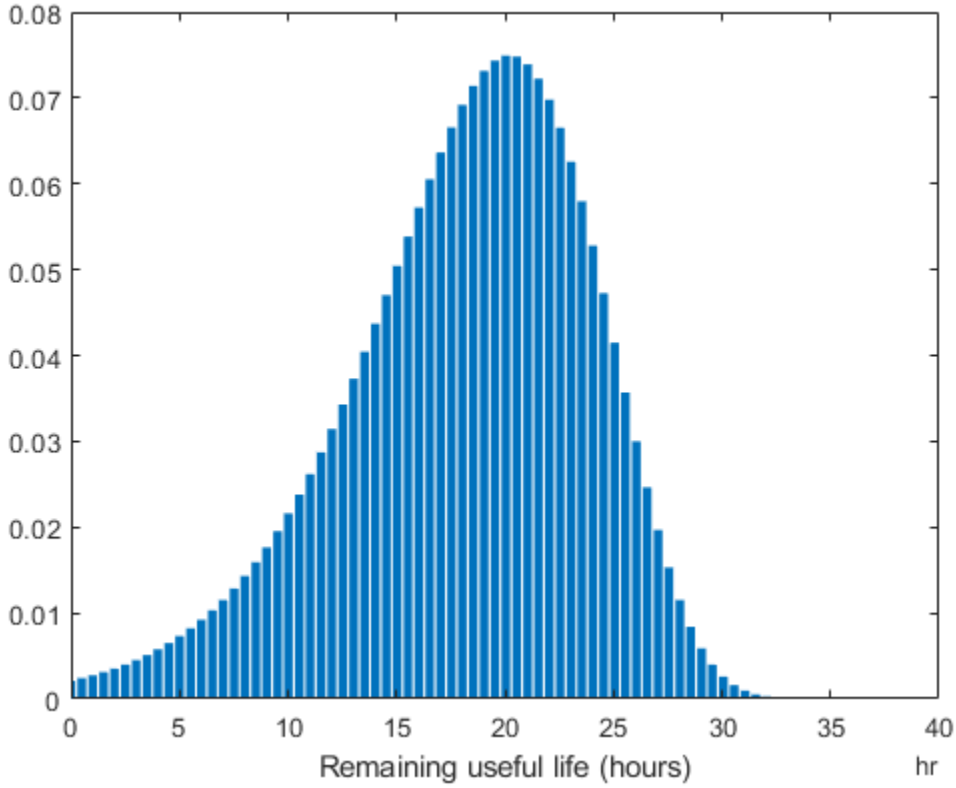
Improve the distribution view by providing the number of bins and bin size for the prediction.

```
[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]))
```



See Also

Functions
`covariateSurvivalModel`

Topics
"Models for Predicting Remaining Useful Life"

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 life time span of the most similar components minus the current life time value of the test component.

For more 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, `hashSimilarityModel('LifeTimeUnit',"days")` creates a residual comparison-based similarity model that uses days as a life time 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 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 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` — Life time variable

`""` (default) | string

Life time 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` — Life time variable units

`""` (default) | value

Life time variable units, specified as a string.

The units of the life time 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 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.

See Also

Functions

`hashSimilarityModel` | `pairwiseSimilarityModel`

Topics

“Models for Predicting Remaining Useful Life”

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 2-86 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 “Read Remote Data” (MATLAB)

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.

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.

`simulationEnsembleDatastore` sets the initial value of `DataVariables` to the names of all the logged signals in the data generated `generateSimulationEnsemble`.

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.

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.

You can also specify `ConditionVariables` using a cell array of character vectors, such as `{'Gear Fault';'Temperature'}`, but the variable names are always stored as a string array, `["Gear Fault";"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 `Fault State`. Set the `SelectedVariables` property and call `read`.

```
ensemble.SelectedVariables = ["Vibration";"Fault State"];
data = read(ensemble)
```

`SelectedVariables` must be a subset 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.

`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

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. When you call `writeToLastMemberRead` to add data back to the ensemble datastore, that function writes to the last member read. The `LastMemberRead` property contains the path to the file to which `writeToLastMemberRead` writes.

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.

read	Read member data from an ensemble datastore
writeToLastMemberRead	Write data to member of an ensemble datastore
reset	Reset datastore to initial state
hasdata	Determine if data is available to read
progress	Determine how much data has been read
numpartitions	Number of datastore partitions
partition	Partition a datastore
tall	Create tall array

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`, as well as state values `xout` and `xfinal` (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);
```

```
[26-Feb-2018 19:09:46] Running SetupFcn...
[26-Feb-2018 19:09:46] Running simulations...
[26-Feb-2018 19:10:11] Completed 1 of 5 simulation runs
[26-Feb-2018 19:10:30] Completed 2 of 5 simulation runs
[26-Feb-2018 19:10:45] Completed 3 of 5 simulation runs
[26-Feb-2018 19:11:00] Completed 4 of 5 simulation runs
[26-Feb-2018 19:11:16] Completed 5 of 5 simulation runs
```

Finally, 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: [6x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: [0x0 string]
       SelectedVariables: [6x1 string]
             NumMembers: 5
         LastMemberRead: [0x0 string]
```

```
ensemble.DataVariables
```

```
ans = 6x1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
    "xFinal"
    "xout"
```

```
ensemble.SelectedVariables
```

```
ans = 6x1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
```

```
    "xFinal"
    "xout"
```

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: [6x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: [0x0 string]
       SelectedVariables: [6x1 string]
              NumMembers: 5
         LastMemberRead: [0x0 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 = 6x1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
    "xFinal"
    "xout"
```

```
ensemble.SelectedVariables
```

```
ans = 6x1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
    "xFinal"
    "xout"
```

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.

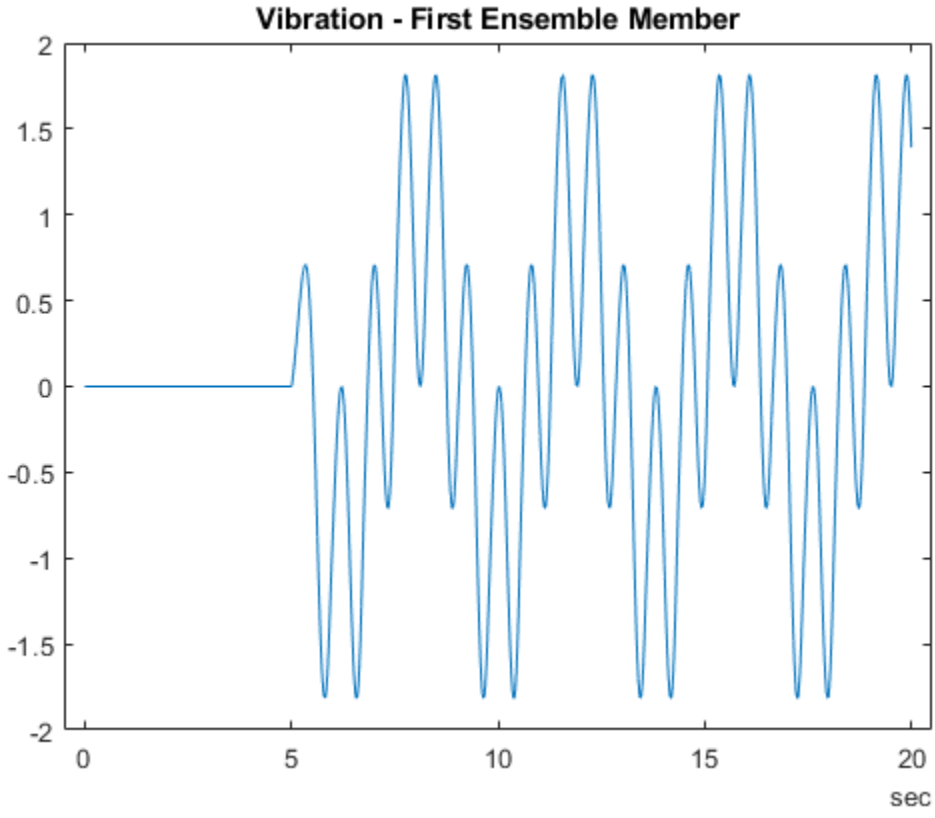
```
ensemble.SelectedVariables = ["Vibration";"SimulationInput"];
data1 = read(ensemble)
```

```
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” for more information.) 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'
```

```
data2.SimulationInput{1}.Variables
```

```
ans =
  Variable with properties:

         Name: 'ToothFaultGain'
        Value: -1.5000
    Workspace: 'global-workspace'
```

This result confirms that `data1` is from the ensemble with `ToothFaultGain` = –2, and `data2` is from the ensemble 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 takes several minutes.

```
unzip simEnsData.zip  % extract compressed files
ensemble = simulationEnsembleDatastore(pwd,'logsout')

ensemble =
  simulationEnsembleDatastore with properties:

           DataVariables: [6×1 string]
    IndependentVariables: [0×0 string]
      ConditionVariables: [0×0 string]
       SelectedVariables: [6×1 string]
              NumMembers: 5
         LastMemberRead: [0×0 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=1×6 table
           SimulationInput                  SimulationMetadata                  Tacho
    ____________________________________    ____________________________________    ____________

    [1×1 Simulink.SimulationInput]    [1×1 Simulink.SimulationMetadata]    [20202×1 tim
```

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.

```
data.SimulationInput

ans = 1×1 cell array
    {1×1 Simulink.SimulationInput}


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 furtther 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.

```
ensemble.DataVariables = [ensemble.DataVariables; "ToothFault"];
ensemble.DataVariables
```

```
ans = 7×1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
    "xFinal"
    "xout"
    "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 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”.

- “Generate and Use Simulated Data Ensemble”

See Also

`fileEnsembleDatastore` | `generateSimulationEnsemble`

Topics

“Generate and Use Simulated Data Ensemble”
“Data Ensembles for Condition Monitoring and Predictive Maintenance”

Introduced in R2018a