

Predictive Maintenance Toolbox™

User's Guide



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™ User's Guide

© COPYRIGHT 2018 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

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

1 **Manage System Data**

Data Ensembles for Condition Monitoring and Predictive Maintenance	1-2
Data Ensembles	1-2
Ensemble Variables	1-5
Ensemble Data in Predictive Maintenance Toolbox	1-6
Convert Ensemble Data into Tall Tables	1-11
Processing Ensemble Data	1-12
Generate and Use Simulated Data Ensemble	1-13
File Ensemble Datastore With Measured Data	1-21

2 **Preprocess Data**

Data Preprocessing for Condition Monitoring and Predictive Maintenance	2-2
Basic Preprocessing	2-3
Filtering	2-3
Time-Domain Preprocessing	2-4
Frequency-Domain (Spectral) Preprocessing	2-4
Time-Frequency Preprocessing	2-5

Identify Condition Indicators

3

Condition Indicators for Monitoring, Fault Detection, and Prediction	3-2
Signal-Based Condition Indicators	3-4
Time-Domain Condition Indicators	3-4
Frequency-Domain Condition Indicators	3-6
Time-Frequency Condition Indicators	3-6
Model-Based Condition Indicators	3-8
Static Models	3-9
Dynamic Models	3-9
State Estimators	3-11

Detect and Predict Faults

4

Decision Models for Fault Detection and Diagnosis	4-2
Feature Selection	4-3
Statistical Distribution Fitting	4-4
Machine Learning	4-4
Regression with Dynamic Models	4-5
Control Charts	4-6
Changepoint Detection	4-7
Models for Predicting Remaining Useful Life	4-8
RUL Estimation Using Identified Models or State Estimators	4-9
RUL Estimation Using RUL Estimator Models	4-10
Choose an RUL Estimator	4-11
Similarity Models	4-11
Degradation Models	4-13
Survival Models	4-13

Deploy Predictive Maintenance Algorithms

5

Deploy Predictive Maintenance Algorithms	5-2
Specifications and Requirements	5-2
Design and Prototype	5-3
Implement and Deploy	5-3
Software and System Integration	5-5
Production	5-5

Manage System Data

- “Data Ensembles for Condition Monitoring and Predictive Maintenance” on page 1-2
- “Generate and Use Simulated Data Ensemble” on page 1-13
- “File Ensemble Datastore With Measured Data” on page 1-21

Data Ensembles for Condition Monitoring and Predictive Maintenance

Data analysis is the heart of any condition monitoring and predictive maintenance activity. The data can come from measurements on systems using sensors such as accelerometers, pressure gauges, thermometers, altimeters, voltmeters, and tachometers. You might have access to measured data from:

- Normal system operation
- The system operating in a faulty condition
- Lifetime record of system operation (run-to-failure data)

For algorithm design, you can also use simulated data generated by running a Simulink model of your system under various operating and fault conditions.

Whether using measured data, generated data, or both, you frequently have many signals, ranging over a time span or multiple time spans. You might also have signals from many machines (for example, measurements from 100 separate engines all manufactured to the same specifications). And, you might have data representing both healthy operation and fault conditions. In any case, designing algorithms for predictive maintenance requires organizing and analyzing large amounts of data while keeping track of the systems and conditions the data represents.

Predictive Maintenance Toolbox provides tools called ensemble datastores for creating, labeling, and managing such data sets. Ensemble datastores can help you work with data that is stored locally or in a remote location such as cloud storage using Amazon S3™ (Simple Storage Service), Windows Azure® Blob Storage, and Hadoop® Distributed File System (HDFS™).

Data Ensembles

The main unit for organizing and managing multifaceted data sets in Predictive Maintenance Toolbox is the data ensemble. An ensemble is a collection of data sets, created by measuring or simulating a system under varying conditions.

For example, consider a transmission gear box system in which you have an accelerometer to measure vibration and a tachometer to measure the engine shaft rotation. Suppose that you run the engine for five minutes and record the measured

signals as a function of time. You also record the engine age, measured in miles driven. Those measurements yield the following data set.

Vibration	Tachometer	Age
[time-series data]	[time-series data]	[scalar]

Now suppose that you have a fleet of many identical engines, and you record data from all of them. Doing so yields a family of data sets.

Engine ID	Vibration	Tachometer	Age
01	[time-series data]	[time-series data]	9,500
02	[time-series data]	[time-series data]	48,000
...
N	[time-series data]	[time-series data]	16,700

This family of data sets is an ensemble, and each row in the ensemble is a member of the ensemble.

The members in an ensemble are related, in the sense that they contain the same data variables. For instance, in the illustrated ensemble, all members include the same four variables: an engine identifier, the vibration and tachometer signals, and the engine age. In that example, each member corresponds to a different machine. Your ensemble might also include that set of data variables recorded from the same machine at different times. For instance, the following illustration shows an ensemble that includes multiple data sets from the same engine recorded as the engine ages.

Engine ID	Vibration	Tachometer	Age
01	[time-series data]	[time-series data]	9,500
01	[time-series data]	[time-series data]	21,250
01	[time-series data]	[time-series data]	44,800
02	[time-series data]	[time-series data]	14,000
02	[time-series data]	[time-series data]	48,000
...

Simulated Ensemble Data

In many cases, you have no real failure data from your system, or only limited data from the system in fault conditions. If you have a Simulink model that approximates the behavior of the actual system, you can generate a data ensemble by simulating the model repeatedly under various conditions and logging the simulation data. For instance, you can:

- Vary parameter values that reflect the presence or absence of a fault. For example, use a very low resistance value to model a short circuit.
- Injecting signal faults. Sensor drift and disturbances in the measured signal affect the measured data values. You can simulate such variation by adding an appropriate signal to the model. For example, you can add an offset to a sensor to represent drift, or model a disturbance by injecting a signal at some location in the model.
- Vary system dynamics. The equations that govern the behavior of a component may change for normal and faulty operation. In this case, the different dynamics can be implemented as variants of the same component.

For example, suppose that you have a Simulink model that describes a gear-box system. The model contains a parameter that represents the drift in a vibration sensor. You

simulate this model at different values of sensor drift, and configure the model to log the vibration and tachometer signals for each simulation. These simulations generate an ensemble that covers a range of operating conditions. Each ensemble member corresponds to one simulation, and records the same data variables under a particular set of conditions.

Vibration	Tachometer	Sensor Drift
[time-series data]	[time-series data]	0
[time-series data]	[time-series data]	0.1
[time-series data]	[time-series data]	0.2
[time-series data]	[time-series data]	0.3
...

Ensemble Variables

The variables in your ensemble serve different purposes, and accordingly can be grouped into several types:

- Data variables — The main content of the ensemble members, including measured data and derived data that you use for analysis and development of predictive maintenance algorithms. For example, in the illustrated gear-box ensembles, **Vibration** and **Tachometer** are the data variables. Data variables can also include derived values, such as the mean value of a signal, or the frequency of the peak magnitude in a signal spectrum.
- Independent variables — The variables that identify or order the members in an ensemble, such as timestamps, number of operating hours, or machine identifiers. In the ensemble of measured gear-box data, **Age** is an independent variable.
- Condition variables — The variables that describe the fault condition or operating condition of the ensemble member. Condition variables can record the presence or

absence of a fault state, or other operating conditions such as ambient temperature. In the ensemble of simulated gear-box data, **Sensor Drift** is a condition variables. Condition variables can also be derived values, such as a single scalar value that encodes multiple fault and operating conditions.

Ensemble Data in Predictive Maintenance Toolbox

With Predictive Maintenance Toolbox, you manage and interact with ensemble data using ensemble datastore objects. In MATLAB®, time-series data is often stored as a vector or a timetable. Other data might be stored as scalar values (such as engine age), logical values (such as whether a fault is present or not), strings (such as an identifier), or tables. Your ensemble can contain any data type that is useful to record for your application. In an ensemble, you typically store the data for each member in a separate file. Ensemble datastore objects help you organize, label, and process ensemble data. Which ensemble datastore object you use depends on whether you are working with measured data on disk, or generating simulated data from a Simulink.

- `simulationEnsembleDatastore` objects — Manage data generated from a Simulink model using `generateSimulationEnsemble`.
- `fileEnsembleDatastore` objects — Manage any other ensemble data stored on disk, such as measured data.

The ensemble datastore objects contain information about the data stored on disk and allow you to interact with the data. You do so using commands such as `read`, which extracts data from the ensemble into the MATLAB workspace, and `writeToLastMemberRead`, which writes data to the ensemble.

Last Member Read


When you work with an ensemble, the software keeps track of which ensemble member it has most recently read. When you call `read`, the software selects the next member to read and updates the `LastMemberRead` property of the ensemble to reflect that member. When you next call `writeToLastMemberRead`, the software writes to that member.

For example, consider the ensemble of simulated gear-box data. When you generate this ensemble using `generateSimulationEnsemble`, the data from each simulation run is logged to a separate file on disk. You then create a `simulationEnsembleDatastore` object that points to the data in those files. You can set properties of the ensemble object to separate the variables into groups such as independent variables or condition variables.

Suppose that you now read some data from the ensemble object, `ensemble`.

```
data = read(ensemble);
```

The first time you call `read` on an ensemble, the software designates some member of the ensemble as the first member to read. The software reads selected variables from that member into the MATLAB workspace, into a table called `data`. (The selected variables are the variables you specify in the `SelectedVariables` property of `ensemble`.) The software updates the property `ensemble.LastMemberRead` with the file name of that member.

Last Member Read 

Vibration	Tachometer	Worn Shaft	Sensor Drift
[time-series data]	[time-series data]	No	0
[time-series data]	[time-series data]	No	0.1
[time-series data]	[time-series data]	No	0.2
...

Until you call `read` again, the last-member-read designation stays with the ensemble member to which the software assigned it. Thus, for example, suppose that you process `data` to compute some derived variable, such as the frequency of the peak value in the vibration signal spectrum, `VibPeak`. You can append the derived value to the ensemble member to which it corresponds, which is still the last member read. To do so, first expand the list of data variables in `ensemble` to include the new variable.

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

This operation is equivalent to adding a new column to the ensemble, as shown in the next illustration. The new variable is initially populated in each ensemble by a missing value. (See `missing` for more information.)

Vibration	Tachometer	Worn Shaft	Sensor Drift	Peak
[time-series data]	[time-series data]	No	0	<missing>
[time-series data]	[time-series data]	No	0.1	<missing>
[time-series data]	[time-series data]	No	0.2	<missing>
...

Now, use `writeToLastMemberRead` to fill in the value of the new variable for the last member read.

```
newdata = table(VibPeak, 'VariableNames', {'VibPeak'});  
writeToLastMemberRead(ensemble, newdata);
```

In the ensemble, the new value is present, and the last-member-read designation remains on the same member.

New Value
Written To Last Member Read

Last Member Read →

Vibration	Tachometer	Worn Shaft	Sensor Drift	Peak
[time-series data]	[time-series data]	No	0	0.00
[time-series data]	[time-series data]	No	0.1	<miss
[time-series data]	[time-series data]	No	0.2	<miss
...

The next time you call read on the ensemble, it determines the next member to read, and returns the selected variables from that member. The last-member-read designation advances to that member.

Last Member Read →

Vibration	Tachometer	Worn Shaft	Sensor Drift	Peak
[time-series data]	[time-series data]	No	0	0.00
[time-series data]	[time-series data]	No	0.1	<miss
[time-series data]	[time-series data]	No	0.2	<miss
...

The hasdata command tells you whether all members of the ensemble have been read. The reset command clears the "read" designation from all members, such that the next

call to `read` operates on the first member of the ensemble. The reset operation clears the `LastMemberRead` property of the ensemble, but it does not change other ensemble properties such as `DataVariables` or `SelectedVariables`. It also does not change any data that you have written back to the ensemble. For an example that shows more interactions with an ensemble of generated data, see “Generate and Use Simulated Data Ensemble” on page 1-13.

Reading Measured Data

Although the previous discussion used a simulated ensemble as an example, the last-member-read designation behaves the same way in ensembles of measured data that you manage with `fileEnsembleDatastore`. However, when you work with measured data, you have to provide information to tell the `read` and `writeToLastMemberRead` commands how your data is stored and organized on disk.

You do so by setting properties of the `fileEnsembleDatastore` object to functions that you write. For instance, set the `DataVariablesFcn` property to the handle of a function that describes how to read the data variables from a data file. You can also provide functions that describe how to read the independent variables and condition variables. When you call `read`, it compares the `SelectedVariables` property of the file ensemble with the `DataVariables`, `IndependentVariables`, and `ConditionVariables` properties to determine which functions to use to read each of the selected variables.

Similarly, you use the `WriteToMemberFcn` property of the `fileEnsembleDatastore` object to provide a function that describes how to write data to a member of the ensemble.

For an example that shows these interactions with an ensemble of measured data, see “File Ensemble Datastore With Measured Data” on page 1-21.

Ensembles and MATLAB Datastores

Ensembles in Predictive Maintenance Toolbox are a specialized kind of MATLAB datastore (see “Getting Started with Datastore” (MATLAB)). The `read` and `writeToLastMemberRead` commands have behavior that is specific to ensemble datastores. Additionally, the following MATLAB datastore commands work with ensemble datastores the same as they do with other MATLAB datastores.

- `hasdata` — Determine whether an ensemble datastore has members that have not yet been read.

- `reset` — Restore an ensemble datastore to the state where no members have yet been read. In this state, there is no current member. Use this command to reread data you have already read from an ensemble.
- `tall` — Convert ensemble datastore to tall table. (See “Tall Arrays” (MATLAB)).
- `progress` — Determine what percentage of an ensemble datastore has been read.
- `partition` — Partition an ensemble datastore into multiple ensemble datastores for parallel computing.
- `numpartitions` — Determine number of datastore partitions.

Convert Ensemble Data into Tall Tables

Some functions, such as many statistical analysis functions, can operate on data in tall tables, which let you work with out-of-memory data that is backed by a datastore. You can convert data from an ensemble datastore into a tall table for use with such analysis commands using the `tall` command.

When working with large ensemble data, such as long time-series signals, you typically process them member-by-member in the ensemble using `read` and `writeToLastMemberRead`. You process the data to compute some feature of the data that can serve as a useful condition indicator for that ensemble member.

Typically, your condition indicator is a scalar value or some other value that takes up less space in memory than the original unprocessed signal. Thus, once you have written such values to your datastore, you can use `tall` and `gather` to extract the condition indicators into memory for further statistical processing, such as training a classifier.

For example, suppose that each member of your ensemble contains time-series vibration data. For each member, you read the ensemble data and compute a condition indicator that is a scalar value derived from a signal-analysis process. You write the derived value back to the member. Suppose that the derived value is in an ensemble variable called `Indicator` and a label containing information about the ensemble member (such as a fault condition) is in a variable called `Label`. To perform further analysis on the ensemble, you can read the condition indicator and label into memory, without reading in the larger vibration data. To do so, set the `SelectedVariables` property of the ensemble to the variables you want to read. Then use `tall` to create a tall table of the selected variables, and `gather` to read the values into memory.

```
ensemble.SelectedVariables = ["Indicator","Label"];  
featureTable = tall(ensemble);  
featureTable = gather(featureTable);
```

The resulting variable `featureTable` is an ordinary table residing in the MATLAB workspace. You can process it with any function that supports the MATLAB table data type.

For examples that illustrate the use of `tall` and `gather` to manipulate ensemble data for predictive maintenance analysis, see:

- “Rolling Element Bearing Fault Diagnosis”
- “Using Simulink to Generate Fault Data”

Processing Ensemble Data

After organizing your data in an ensemble, the next step in predictive maintenance algorithm design is to preprocess the data to clean or transform it. Then you process the data further to extract condition indicators, which are data features that you can use to distinguish healthy from faulty operation. For more information, see:

- “Data Preprocessing for Condition Monitoring and Predictive Maintenance” on page 2-2
- “Condition Indicators for Monitoring, Fault Detection, and Prediction” on page 3-2

See Also

`fileEnsembleDatastore` | `generateSimulationEnsemble` | `read` | `simulationEnsembleDatastore`

More About

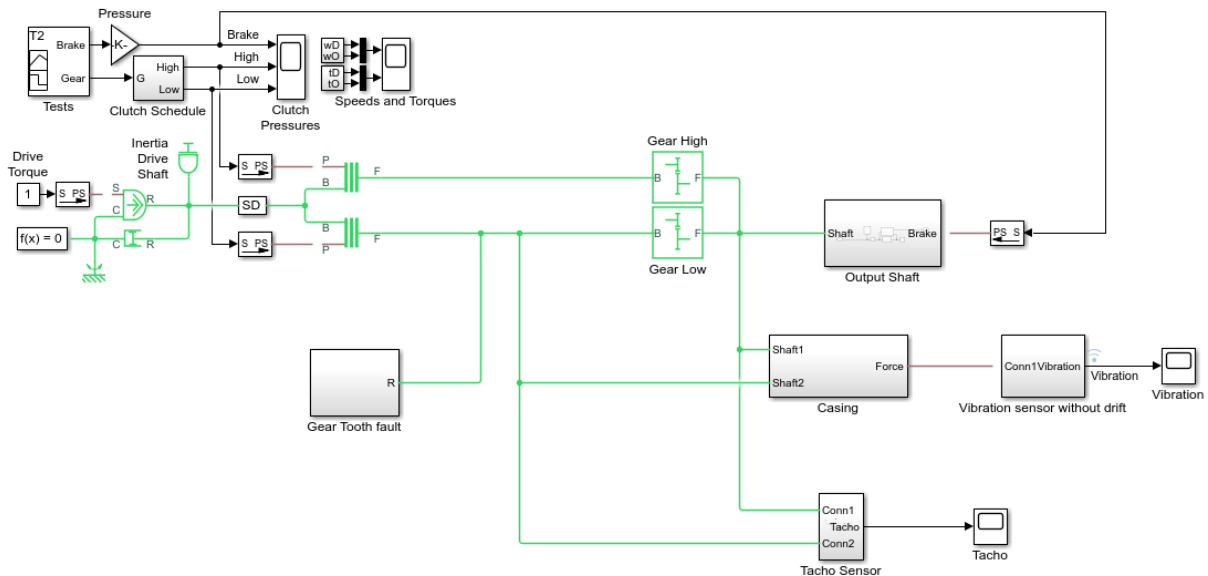
- “Generate and Use Simulated Data Ensemble” on page 1-13
- “File Ensemble Datastore With Measured Data” on page 1-21

Generate and Use Simulated Data Ensemble

This example shows how to generate a data ensemble for predictive-maintenance algorithm design by simulating a Simulink® model of a machine while varying a fault parameter. The example then illustrates some of the ways you interact with a simulation ensemble datastore. The example shows how to read data from the datastore into the MATLAB® workspace, process the data to compute derived variables, and write the new variables back to the datastore.

The model in this example is a simplified version of the gear-box model described in “Using Simulink to Generate Fault Data”. Load the Simulink model.

```
mdl = 'TransmissionCasingSimplified';
open_system(mdl)
```



For this example, only one fault mode is modeled. 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).

Generate the Ensemble of Simulated Data

To generate a simulation ensemble datastore of fault data, you use `generateSimulationEnsemble` to simulate the model at different values of `ToothFaultGain`, ranging from -2 to zero. This function simulates the model once for each entry in an array of `Simulink.SimulationInput` objects that you provide. Each simulation generates a separate member of the ensemble. 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
    tmp = Simulink.SimulationInput mdl;
    tmp = setVariable(tmp, 'ToothFaultGain', toothFaultValues(ct));
    simin(ct) = tmp;
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` function 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 `true` if all the simulations complete without error.

```
mkdir Data
location = fullfile(pwd, 'Data');
[status,E] = generateSimulationEnsemble(simin,location);

[19-Jan-2018 13:25:13] Running SetupFcn...
[19-Jan-2018 13:25:13] Running simulations...
[19-Jan-2018 13:25:31] Completed 1 of 5 simulation runs
[19-Jan-2018 13:25:37] Completed 2 of 5 simulation runs
[19-Jan-2018 13:25:43] Completed 3 of 5 simulation runs
[19-Jan-2018 13:25:48] Completed 4 of 5 simulation runs
[19-Jan-2018 13:25:54] Completed 5 of 5 simulation runs
```

```
status
```

```
status = logical  
      1
```

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.

```
ensemble = simulationEnsembleDatastore(location)
```

```
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]
```

```
ensemble.DataVariables
```

```
ans = 6×1 string array  
  "SimulationInput"  
  "SimulationMetadata"  
  "Tacho"  
  "Vibration"  
  "xFinal"  
  "xout"
```

```
ensemble.SelectedVariables
```

```
ans = 6×1 string array  
  "SimulationInput"  
  "SimulationMetadata"  
  "Tacho"  
  "Vibration"  
  "xFinal"  
  "xout"
```

Read Data from Ensemble Members

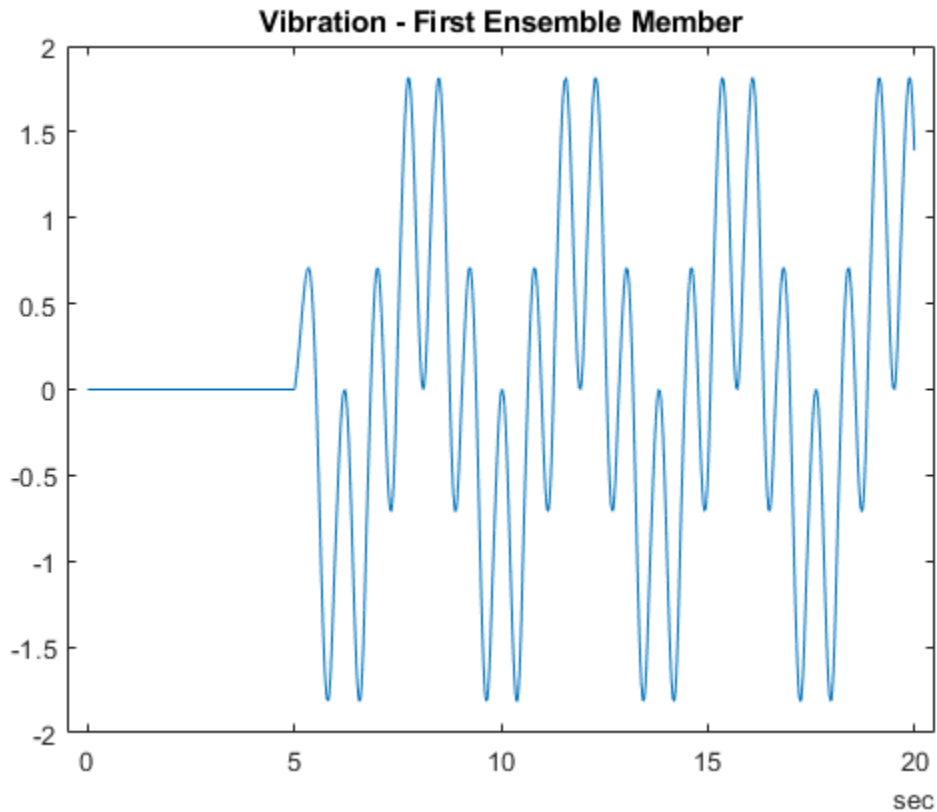
Suppose that for the analysis you want to do, you need only the `Vibration` data and the `Simulink.SimulationInput` object that describes the conditions under which each member was simulated. Set `ensemble.SelectedVariables` to specify the variables you want to read. The `read` command then extracts those variables from the first ensemble member, as determined by the software.

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

```
data1=1x2 table  
      Vibration      SimulationInput  
-----  
[20202x1 timetable]  [1x1 Simulink.SimulationInput]
```

`data.Vibration` is a cell array containing one `timetable` row storing 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 `LastMemberRead` property of the ensemble contains the file name of the most recently read member. The next time you call `read` on this ensemble, the software advances to the next member of the ensemble. (See “Data Ensembles for Condition Monitoring and Predictive Maintenance” on page 1-2 for more information.) Read the selected variables from the next member of the ensemble.

```
data2 = read(ensemble)
```

```
data2=1×2 table
```

```
Vibration
```

```
SimulationInput
```

```
[20215×1 timetable]
```

```
[1×1 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.

```
SimInput1 = data1.SimulationInput{1};  
SimInput1.Variables
```

```
ans =  
  Variable with properties:  
  
      Name: 'ToothFaultGain'  
      Value: -2  
  Workspace: 'global-workspace'
```

```
SimInput2 = data2.SimulationInput{1};  
SimInput2.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 Data to Ensemble Member

Suppose that you want to convert the `ToothFaultGain` values for each ensemble member into a binary indicator of whether or not a tooth fault is present. Suppose further that you know from your experience with the system that tooth-fault gain values less than 0.1 in magnitude are small enough to be considered healthy operation. Convert the gain value for the ensemble member you just read into an indicator that is 0 (no fault) for $-0.1 < \text{gain} < 0.1$, and 1 (fault) otherwise.

```
sT = (abs(SimInput2.Variables.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 = 7x1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
    "xFinal"
    "xout"
    "ToothFault"
```

Then, create a table row containing the derived indicator value and the name `ToothFault`. Use `writeToLastMemberRead` to write the value to the last-read member of the ensemble.

```
sdata = table(sT, 'VariableNames', {'ToothFault'});
writeToLastMemberRead(ensemble, sdata);
```

Batch-Process Data from All Ensemble Members

In practice, you want to append the tooth-fault indicator to every member in the ensemble. To do so, reset the ensemble to its unread state, so that the next read begins at the first ensemble member. Then, loop through all the ensemble members, computing `ToothFault` for each member and appending it.

```
reset(ensemble);
sT = false;
while hasdata(ensemble)
    data = read(ensemble);
    SimInputVars = data.SimulationInput{1}.Variables;
    TFGain = SimInputVars.Value;
    sT = (abs(TFGain) < 0.1);
    sdata = table(sT, 'VariableNames', {'ToothFault'});
    writeToLastMemberRead(ensemble, sdata)
end
```

Finally, designate the new tooth-fault indicator as a condition variable in the ensemble. 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"
```

For an example that shows more ways to manipulate and analyze data stored in a `simulationEnsembleDatastore` object, see “Using Simulink to Generate Fault Data”.

See Also

`generateSimulationEnsemble` | `read` | `simulationEnsembleDatastore` | `writeToLastMemberRead`

More About

- “Data Ensembles for Condition Monitoring and Predictive Maintenance” on page 1-2

File Ensemble Datastore With Measured Data

In predictive-maintenance algorithm design, you often work with large sets of data collected from operation of your system under varying conditions. The `fileEnsembleDatastore` object helps you manage and interact with such data. For this example, create a `fileEnsembleDatastore` object that points to some ensemble data on disk. Configure it with functions that read data from and write data to the ensemble.

Structure of the Data Files

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`. In practice you might have many more data files.

Each of these data files contains one data structure, `bearing`. Load and examine the data structure from the first healthy data set.

```
unzip fileEnsData.zip % extract compressed files
load baseline_01.mat
bearing

bearing = struct with fields:
    sr: 97656
    gs: [585936x1 double]
    load: 270
    rate: 25
```

The structure contains a vector of accelerometer data `gs`, the sample rate `sr` at which that data was recorded, and some other data variables.

Create and Configure File Ensemble Datastore

To work with this data for predictive maintenance algorithm design, first create a file ensemble datastore that points to the data 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. For this example, use the following provided functions:

- `readBearingData` — Parse the `bearing` structure in a file, and return a table row containing one table variable for each field in the structure. If the file includes other data variables besides `bearing`, the returned table row also includes those variables.
- `readLabels` — Parse the file name and return a table row containing a fault label in the variable `Label`, and the file name in a variable `Filename`.
- `writeBearingData` — Take a structure and write its variables to a data file as individual stored variables.

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

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

Finally, set properties of the ensemble to identify the four data variables, the condition variables, and the selected variables for reading.

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

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

```
fensemble
```

```
fensemble =  
  fileEnsembleDatastore with properties:  
  
    DataVariablesFcn: @readBearingData  
  ConditionVariablesFcn: @readLabels  
IndependentVariablesFcn: []  
    WriteToMemberFcn: @writeBearingData  
      DataVariables: [4×1 string]  
IndependentVariables: [0×0 string]  
    ConditionVariables: [2×1 string]  
    SelectedVariables: [6×1 string]  
      NumMembers: 5  
    LastMemberRead: [0×0 string]
```

Read Data From Ensemble Member

The functions you assigned tell the `read` and `writeToLastMemberRead` commands how to interact with the data files that make up the ensemble datastore. 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`, and uses `@readLabels` to read selected variables that are in `fensemble.ConditionVariables`. The command reads data from the first ensemble member (determined by the software) into a table row in the MATLAB workspace.

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

```
data=1x6 table
      gs          sr      load      rate      Label      Filename
-----
[146484x1 double] 48828      0      25      "Faulty"      "FaultData_01"
```

Suppose that you want to analyze the accelerometer data `gs` by computing its power spectrum, and then write the power spectrum data back into the ensemble. To do so, first extract the data from the table and compute the spectrum.

```
gsdata = data.gs{1};
sr = data.sr;
[pdata,fpdata] = pspectrum(gsdata,sr);
pdata = 10*log10(pdata); % Convert to dB
```

Write Data to Ensemble Member

You can write the frequency vector `fpdata` and the power spectrum `pdata` to the data file as separate variables. For this example, suppose that you want to add the new variables in a data structure called `Spectrum`.

First, add `Spectrum` to the data variables of the ensemble datastore.

```
fensemble.DataVariables = [fensemble.DataVariables; "Spectrum"];
fensemble.DataVariables
```

```
ans = 5x1 string array
    "gs"
    "sr"
    "load"
    "rate"
```

```
"Spectrum"
```

Next, create the data structure and write it to the file representing the last-read ensemble member. When you call `writeToLastMemberRead`, it uses `fensemble.WriteToMemberFcn` to write the table data to the file.

```
Spectrum = struct('Freq',fpdata,'Power',pdata);  
writeToLastMemberRead(fensemble,'Spectrum',Spectrum);
```

You can add the new variable to `fensemble.SelectedVariables`, or other properties for identifying variables, as needed.

Calling `read` again reads the data from the next file in the ensemble datastore and updates the property `fensemble.LastMemberRead`.

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

```
data=1x6 table  
      gs          sr    load    rate    Label    Filename  
-----  
[146484x1 double] 48828    50     25    "Faulty"  "FaultData_02"
```

You can see that this data is from a different member by the `load` variable in the table. Here, its value is 50, while in the previously read member, it was 0.

Batch-Process Data from All Ensemble Members

You can repeat the processing steps to compute and append the spectrum 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 datastore to a state in which no data has been read. (The reset operation does not change `fensemble.DataVariables`, which already contains `Spectrum`.) 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};  
    sr = data.sr;  
    [pdata,fpdata] = pspectrum(gsdata,sr);  
    Spectrum = struct('Freq',fpdata,'Power',pdata);
```

```

        writeToLastMemberRead(fensemble, 'Spectrum', Spectrum);
end

```

The `hasdata` command returns false when every member of the ensemble has been read. Now, each data file in the ensemble includes the `Spectrum` variable derived from the accelerometer data 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, `Spectrum`, but do not include the unprocessed data, `gs`.

```

reset(fensemble)
fensemble.SelectedVariables = ["load", "Spectrum", "Label"];
data1 = read(fensemble)

```

```

data1=1x3 table
    load      Spectrum      Label
    _____  _____  _____
    0          [1x1 struct]  "Faulty"

```

```

data2 = read(fensemble)

```

```

data2=1x3 table
    load      Spectrum      Label
    _____  _____  _____
    50          [1x1 struct]  "Faulty"

```

```

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

```

See Also

`fileEnsembleDatastore` | `read` | `writeToLastMemberRead`

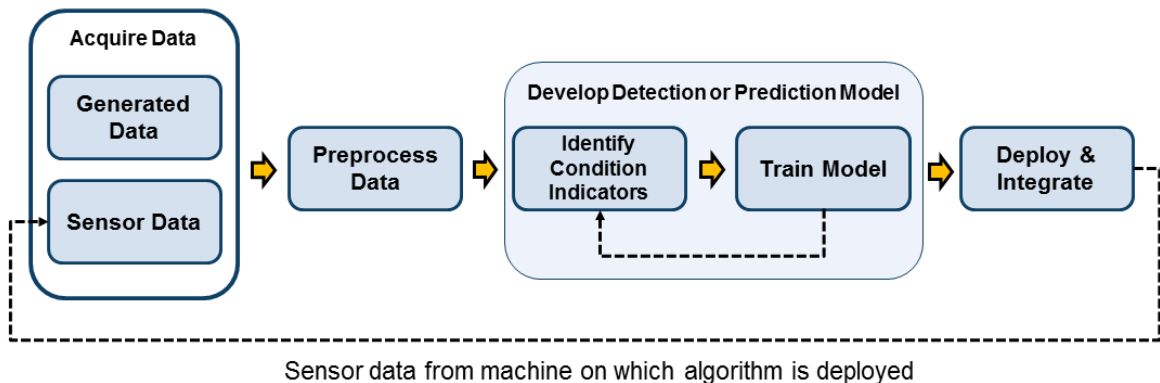
More About

- “Data Ensembles for Condition Monitoring and Predictive Maintenance” on page 1-2

Preprocess Data

Data Preprocessing for Condition Monitoring and Predictive Maintenance

Data preprocessing is the second stage of the workflow for predictive maintenance algorithm development:



Data preprocessing is often necessary to clean the data and convert it into a form from which you can extract condition indicators. Data preprocessing can include:

- Outlier and missing-value removal, offset removal, and detrending.
- Noise reduction, such as filtering or smoothing.
- Transformations between time and frequency domain.
- More advanced signal processing such as short-time Fourier transforms and transformations to the order domain.

You can perform data preprocessing on arrays or tables of measured or simulated data that you manage with Predictive Maintenance Toolbox ensemble datastores, as described in “Data Ensembles for Condition Monitoring and Predictive Maintenance” on page 1-2. Generally, you preprocess your data before analyzing it to identify a promising condition indicator, a quantity that changes in a predictable way as system performance degrades. (See “Condition Indicators for Monitoring, Fault Detection, and Prediction” on page 3-2.) There can be some overlap between the steps of preprocessing and identifying condition indicators. Typically, though, preprocessing results in a cleaned or transformed signal, on which you perform further analysis to condense the signal information into a condition indicator.

Understanding your machine and the kind of data you have can help determine what preprocessing methods to use. For example, if you are filtering noisy vibration data, knowing what frequency range is most likely to display useful features can help you choose preprocessing techniques. Similarly, it might be useful to transform gearbox vibration data to the order domain, which is used for rotating machines when the rotational speed changes over time. However, that same preprocessing would not be useful for vibration data from a car chassis, which is a rigid body.

Basic Preprocessing

MATLAB includes many functions that are useful for basic preprocessing of data in arrays or tables. These include functions for:

- Data cleaning, such as `fillmissing` and `filloutliers`. Data cleaning uses various techniques for finding, removing, and replacing bad or missing data.
- Smoothing data, such as `smoothdata` and `movmean`. Use smoothing to eliminate unwanted noise or high variance in data.
- Detrending data, such as `detrend`. Removing a trend from the data lets you focus your analysis on the fluctuations in the data about the trend. While trends can be meaningful, others are due to systematic effects, and some types of analyses yield better insight once you remove them. Removing offsets is another, similar type of preprocessing.
- Scaling or normalizing data, such as `rescale`. Scaling changes the bounds of the data, and can be useful, for example, when you are working with data in different units.

Another common type of preprocessing is to extract a useful portion of the signal and discard other portions. For instance, you might discard the first five seconds of a signal that is part of some start-up transient, and retain only the data from steady-state operation. For an example that performs this kind of preprocessing, see “Using Simulink to Generate Fault Data”.

For more information on basic preprocessing commands in MATLAB, see “Preprocessing Data” (MATLAB).

Filtering

Filtering is another way to remove noise or unwanted components from a signal. Filtering is particularly helpful when you know what frequency range in the data is most likely to

display useful features for condition monitoring or prediction. The basic MATLAB function `filter` lets you filter a signal with a transfer function. You can use `designfilt` to generate filters for use with `filter`, such as passband, high-pass and low-pass filters, and other common filter forms. For more information about using these functions, see “Digital and Analog Filters” (Signal Processing Toolbox).

If you have a Wavelet Toolbox™ license, you can use wavelet tools for more complex filter approaches. For instance, you can divide your data into subbands, process the data in each subband separately, and recombine them to construct a modified version of the original signal. For more information about such filters, see “Filter Banks” (Wavelet Toolbox). You can also use the Signal Processing Toolbox™ function `emd` to decompose separate a mixed signal into components with different time-frequency behavior.

Time-Domain Preprocessing

Signal Processing Toolbox provides functions that let you study and characterize vibrations in mechanical systems in the time domain. These functions can be for both preprocessing and extraction of condition indicators. For example:

- `tsa` — Remove noise coherently with time-synchronous averaging and analyze wear using envelope spectra. The example “Using Simulink to Generate Fault Data” uses time-synchronous averaging to preprocess vibration data.
- `ordertrack` — Use order analysis to analyze and visualize spectral content occurring in rotating machinery. Track and extract orders and their time-domain waveforms.
- `rpmtrack` — Track and extract the RPM profile from a vibration signal by computing the RPM as a function of time.
- `envspectrum` — Compute an envelope spectrum. The envelope spectrum removes the high-frequency sinusoidal components from the signal and focuses on the lower-frequency modulations. The example “Rolling Element Bearing Fault Diagnosis” uses an envelope spectrum for such preprocessing.

For more information on these and related functions, see “Vibration Analysis” (Signal Processing Toolbox).

Frequency-Domain (Spectral) Preprocessing

For vibrating or rotating systems, fault development can be indicated by changes in frequency-domain behavior such as the changing of resonant frequencies or the presence of new vibrational components. Signal Processing Toolbox provides many functions for

analyzing such spectral behavior. Often these are useful as preprocessing before performing further analysis for extracting condition indicators. Such functions include:

- `pspectrum` — Compute the power spectrum, time-frequency power spectrum, or power spectrogram of a signal. The spectrogram contains information about how the power distribution changes with time. The example “Multi-Class Fault Detection Using Simulated Data” performs data preprocessing using `pspectrum`.
- `envspectrum` — Compute an envelope spectrum. A fault that causes a repeating impulse or pattern will impose amplitude modulation on the vibration signal of the machinery. The envelope spectrum removes the high-frequency sinusoidal components from the signal and focuses on the lower-frequency modulations. The example “Rolling Element Bearing Fault Diagnosis” uses an envelope spectrum for such preprocessing.
- `orderspectrum` — Compute an average order-magnitude spectrum.
- `modalfrf` — Estimate the frequency-response function of a signal.

For more information on these and related functions, see “Vibration Analysis” (Signal Processing Toolbox).

Time-Frequency Preprocessing

Signal Processing Toolbox includes functions for analyzing systems whose frequency-domain behavior changes with time. Such analysis is called *time-frequency* analysis, and is useful for analyzing and detecting transient or changing signals associated with changes in system performance. These functions include:

- `spectrogram` — Compute a spectrogram using a short-time Fourier transform. The spectrogram describes the time-localized frequency content of a signal and its evolution over time. The example “Condition Monitoring and Prognostics Using Vibration Signals” uses `spectrogram` to preprocess signals and help identify potential condition indicators.
- `hht` — Compute the Hilbert spectrum of a signal. The Hilbert spectrum is useful for analyzing signals that comprise a mixture of signals whose spectral content changes in time. This function computes the spectrum of each component in the mixed signal, where the components are determined by empirical mode decomposition.
- `emd` — Compute the empirical mode decomposition of a signal. This decomposition describes the mixture of signals analyzed in a Hilbert spectrum, and can help you separate a mixed signal to extract a component whose time-frequency behavior changes as system performance degrades. You can use `emd` to generate the inputs for `hht`.

- `kurtogram` — Compute the time-localized spectral kurtosis, which characterizes a signal by differentiating stationary Gaussian signal behavior from nonstationary or non-Gaussian behavior in the frequency domain. As preprocessing for other tools such as envelope analysis, spectral kurtosis can supply key inputs such as optimal band. (See `pkurtosis`.) The example “Rolling Element Bearing Fault Diagnosis” uses spectral kurtosis for preprocessing and extraction of condition indicators.

For more information on these and related functions, see “Time-Frequency Analysis” (Signal Processing Toolbox).

See Also

More About

- “Designing Algorithms for Condition Monitoring and Predictive Maintenance”
- “Data Ensembles for Condition Monitoring and Predictive Maintenance” on page 1-2
- “Condition Indicators for Monitoring, Fault Detection, and Prediction” on page 3-2

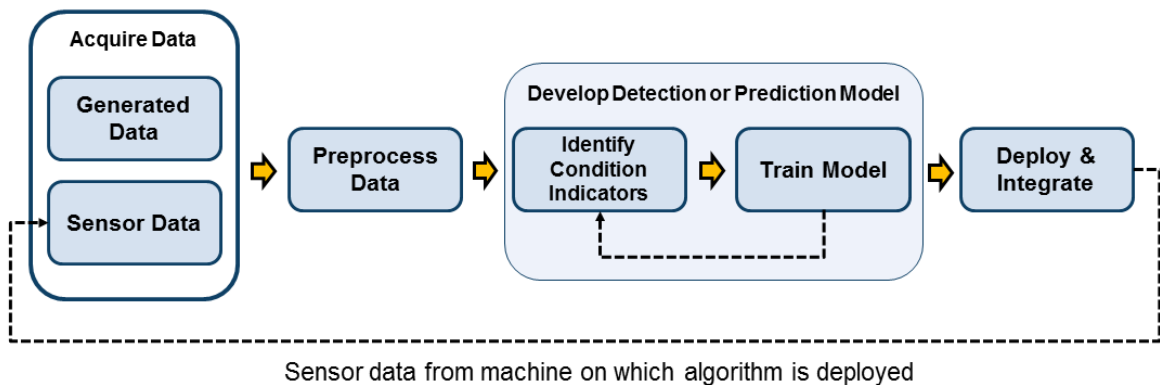
Identify Condition Indicators

Condition Indicators for Monitoring, Fault Detection, and Prediction

A condition indicator is a feature of system data whose behavior changes in a predictable way as the system degrades or operates in different operational modes. A condition indicator can be any feature that is useful for distinguishing normal from faulty operation or for predicting remaining useful life. A useful condition indicator clusters similar system status together, and sets different status apart. Examples of condition indicators include quantities derived from:

- Simple analysis, such as the mean value of the data over time
- More complex signal analysis, such as the frequency of the peak magnitude in a signal spectrum, or a statistical moment describing changes in the spectrum over time
- Model-based analysis of the data, such as the maximum eigenvalue of a state space model which has been estimated using the data
- Combination of both model-based and signal-based approaches, such as using the signal to estimate a dynamic model, simulating the dynamic model to compute a residual signal, and performing statistical analysis on the residual
- Combination of multiple features into a single effective condition indicator

The identification of condition indicators is typically the third step of the workflow for designing a predictive maintenance algorithm, after accessing and preprocessing data.



You use condition indicators extracted from system data taken under known conditions to train a model that can then diagnose or predict the condition of a system based on new data taken under unknown conditions. In practice, you might need to explore your data and experiment with different condition indicators to find the ones that best suit your machine, your data, and your fault conditions. The examples “Fault Diagnosis of Centrifugal Pumps using Residual Analysis” and “Using Simulink to Generate Fault Data” illustrate analyses that test multiple condition indicators and empirically determine the best ones to use.

In some cases, a combination of condition indicators can provide better separation between fault conditions than a single indicator on its own. The example “Rolling Element Bearing Fault Diagnosis” is one in which such a combined indicator is useful. Similarly, you can often train decision models for fault detection and diagnosis using a table containing multiple condition indicators computed for many ensemble members. For an example that uses this approach, see “Multi-Class Fault Detection Using Simulated Data”.

Predictive Maintenance Toolbox and other toolboxes include many functions that can be useful for extracting condition indicators. For more information about different types of condition indicators and their uses, see:

- “Signal-Based Condition Indicators” on page 3-4
- “Model-Based Condition Indicators” on page 3-8

You can extract condition indicators from vectors or timetables of measured or simulated data that you manage with Predictive Maintenance Toolbox ensemble datastores, as described in “Data Ensembles for Condition Monitoring and Predictive Maintenance” on page 1-2. It is often useful to preprocess such data first, as described in “Data Preprocessing for Condition Monitoring and Predictive Maintenance” on page 2-2.

See Also

More About

- “Signal-Based Condition Indicators” on page 3-4
- “Model-Based Condition Indicators” on page 3-8
- “Designing Algorithms for Condition Monitoring and Predictive Maintenance”

Signal-Based Condition Indicators

A signal-based condition indicator is a quantity derived from processing signal data. The condition indicator captures some feature of the signal that changes in a reliable way as system performance degrades. In designing algorithms for predictive maintenance, you use such a condition indicator to distinguish healthy from faulty machine operation. Or, you can use trends in the condition indicator to identify degrading system performance indicative of wear or other developing fault condition.

Signal-based condition indicators can be extracted using any type of signal processing, including time-domain, frequency-domain, and time-frequency analysis. Examples of signal-based condition indicators include:

- The mean value of a signal that changes as system performance changes
- A quantity that measures chaotic behavior in a signal, the presence of which might be indicative of a fault condition
- The peak magnitude in a signal spectrum, or the frequency at which the peak magnitude occurs, if changes in such frequency-domain behavior are indicative of changing machine conditions

In practice, you might need to explore your data and experiment with different condition indicators to find the ones that best suit your machine, your data, and your fault conditions. There are many functions that you can use for signal analysis to generate signal-based condition indicators. The following sections summarize some of them. You can use these functions on signals in arrays or timetables, such as signals extracted from an ensemble datastore. (See “Data Ensembles for Condition Monitoring and Predictive Maintenance” on page 1-2.)

Time-Domain Condition Indicators

Simple Time-Domain Features

For some systems, simple statistical features of time signals can serve as condition indicators, distinguishing fault conditions from healthy conditions. For example, the average value of a particular signal (`mean`) or its standard deviation (`std`) might change as system health degrades. Or, you can try higher-order moments of the signal such as `skewness` and `kurtosis`. With such features, you can try to identify threshold values that distinguish healthy operation from faulty operation, or look for abrupt changes in the value that mark changes in system state.

Other functions you can use to extract simple time-domain features include:

- `peak2peak` — Difference between maximum and minimum values in a signal.
- `envelope` — Signal envelope.
- `dtw` — Distance between two signals, computed by dynamic time warping.
- `rainflow` — Cycle counting for fatigue analysis.

Nonlinear Features in Time-Series Data

In systems that exhibit chaotic signals, certain nonlinear properties can indicate sudden changes in system behavior. Such nonlinear features can be useful in analyzing vibration and acoustic signals from systems such as bearings, gears, and engines. They can reflect changes in phase space trajectory of the underlying system dynamics that occur even before the occurrence of a fault condition. Thus, monitoring a system's dynamic characteristics using nonlinear features can help identify potential faults earlier, such as when a bearing is slightly worn.

Predictive Maintenance Toolbox includes several functions for computing nonlinear signal features. These quantities represent different ways of characterizing the level of chaos in a system. Increase in chaotic behavior can indicate a developing fault condition.

- `lyapunovExponent` — Compute the largest Lyapunov exponent, which characterizes the rate of separation of nearby phase-space trajectories.
- `approximateEntropy` — Estimate the approximate entropy of a time-domain signal. The approximate entropy quantifies the amount of regularity or irregularity in a signal.
- `correlationDimension` — Estimate the correlation dimension of a signal, which is a measure of the dimensionality of the phase space occupied by the signal. Changes in correlation dimension indicate changes in the phase-space behavior of the underlying system.

The computation of these nonlinear features relies on the `phaseSpaceReconstruction` function, which reconstructs the phase space containing all dynamic system variables.

The example “Using Simulink to Generate Fault Data” uses both simple time-domain features and these nonlinear features as candidates for diagnosing different fault conditions. The example computes all features for every member of a simulated data ensemble, and uses the resulting feature table to train a classifier.

Frequency-Domain Condition Indicators

For some systems, spectral analysis can generate signal features that are useful for distinguishing healthy and faulty states. Some functions you can use to compute frequency-domain condition indicators include:

- `meanfreq` — Mean frequency of the power spectrum of a signal.
- `powerbw` — 3-dB power bandwidth of a signal.
- `findpeaks` — Values and locations of local maxima in a signal. If you preprocess the signal by transforming it into the frequency domain, `findpeaks` can give you the frequencies of spectral peaks.

The example “Condition Monitoring and Prognostics Using Vibration Signals” uses such frequency-domain analysis to extract condition indicators.

For a list of functions you can use for frequency-domain feature extraction, see “Identify Condition Indicators”.

Time-Frequency Condition Indicators

Time-Frequency Spectral Properties

The time-frequency spectral properties are another way to characterize changes in the spectral content of a signal over time. Available functions for computing condition indicators based on time-frequency spectral analysis include:

- `pkurtosis` — Compute spectral kurtosis, which characterizes a signal by differentiating stationary Gaussian signal behavior from nonstationary or non-Gaussian behavior in the frequency domain. Spectral kurtosis takes small values at frequencies where stationary Gaussian noise only is present, and large positive values at frequencies where transients occur. Spectral kurtosis can be a condition indicator on its own. You can use `kurtogram` to visualize the spectral kurtosis, before extracting features with `pkurtosis`. As preprocessing for other tools such as envelope analysis, spectral kurtosis can supply key inputs such as optimal bandwidth.
- `pentropy` — Compute spectral entropy, which characterizes a signal by providing a measure of its information content. Where you expect smooth machine operation to result in a uniform signal such as white noise, higher information content can indicate mechanical wear or faults.

The example “Rolling Element Bearing Fault Diagnosis” uses spectral features of fault data to compute a condition indicator that distinguishes two different fault states in a bearing system.

Time-Frequency Moments

Time-frequency moments provide an efficient way to characterize nonstationary signals, signals whose frequencies change in time. Classical Fourier analysis cannot capture the time-varying frequency behavior. Time-frequency distributions generated by short-time Fourier transform or other time-frequency analysis techniques can capture the time-varying behavior. Time-frequency moments provide a way to characterize such time-frequency distributions more compactly. There are three types of time-frequency moments:

- `tfsmoment` — Conditional spectral moment, which is the variation of the spectral moment over time. Thus, for example, for the second conditional spectral moment, `tfsmoment` returns the instantaneous variance of the frequency at each point in time.
- `tftmoment` — Conditional temporal moment, which is the variation of the temporal moment with frequency. Thus, for example, for the second conditional temporal moment, `tftmoment` returns the variance of the signal at each frequency.
- `tfmoment` — Joint time-frequency moment. This scalar quantity captures the moment over both time and frequency.

You can also compute the instantaneous frequency as a function of time using `instfreq`.

See Also

More About

- “Condition Indicators for Monitoring, Fault Detection, and Prediction” on page 3-2
- “Model-Based Condition Indicators” on page 3-8

Model-Based Condition Indicators

A model-based condition indicator is a quantity derived from fitting system data to a model and performing further processing using the model. The condition indicator captures aspects of the model that change as system performance degrades. Model based-condition indicators can be useful when:

- It is difficult to identify suitable condition indicators using features from signal analysis alone. This situation can occur when other factors affect the signal apart from the fault condition of the machine. For instance, the signals you measure might vary depending upon one or more input signals elsewhere in the system.
- You have knowledge of the system or underlying processes such that you can model some aspect of the system's behavior. For instance, you might know from system knowledge that there is a system parameter, such as a time constant, that will change as the system degrades.
- You want to do some forecasting or simulation of future system behavior based upon current system conditions. (See “Models for Predicting Remaining Useful Life” on page 4-8.)

In such cases, it can be useful and efficient to fit the data to some model and use condition indicators extracted from the model rather than from direct analysis of the signal. Model-based condition indicators can be based on any type of model that is suitable for your data and system, including both static and dynamic models. Condition indicators you extract from models can be quantities such as:

- Model parameters, such as the coefficients of a linear fit. A change in such a parameter value can be indicative of a fault condition.
- Statistical properties of model parameters, such as the variance. A model parameter that falls outside the statistical range expected from healthy system performance can be indicative of a fault.
- Dynamic properties, such as system state values obtained by state estimation, or the pole locations or damping coefficient of an estimated dynamic model.
- Quantities derived from simulation of a dynamic model.

In practice, you might need to explore different models and experiment with different condition indicators to find the ones that best suit your machine, your data, and your fault conditions. There are many approaches that you can take to identifying model-based condition indicators. The following sections summarize common approaches.

Static Models

When you have data obtained from steady-state system operation, you can try fitting the data to a static model, and using parameters of that model to extract condition indicators. For example, suppose that you generate an ensemble of data by measuring some characteristic curve in different machines, at different times, or under different conditions. You can then fit a polynomial model to the characteristic curves, and use the resulting polynomial coefficients as condition indicators.

The example “Fault Diagnosis of Centrifugal Pumps using Steady State Experiments” takes this approach. The data in that example describes the characteristic relation between pump head and flow rate, measured in an ensemble of pumps during healthy steady-state operation. The example performs a simple linear fit to describe this characteristic curve. Because there is some variation in the best-fit parameters across the ensemble, the example uses the resulting parameters to determine a distribution and confidence region for the fit parameters. Performing the same fit with a test data set yields parameters, and comparison of these parameters with the distribution yields the likelihood of a fault.

You can also use static models to generate grouped distributions of healthy and faulty data. When you obtain a new point from test data, you can use hypothesis tests to determine which distribution the point most likely belongs to.

Dynamic Models

For dynamic systems, changes in measured signals (outputs) depend on changes in signals elsewhere in the system (inputs). You can use a dynamic model of such a system to generate condition indicators. Some dynamic models are based on both input and output data, while others can be fit based on time-series output data alone. You do not necessarily need a known model of the underlying dynamic processes to perform such model fitting. However, system knowledge can help you choose the type or structure of model to fit.

Some functions you can use for model fitting include:

- `ssest` — Estimate a state-space model from time-domain input-output data or frequency-response data.
- `ar` — Estimate a least-squares autoregressive (AR) model from time-series data.
- `nlarx` — Model nonlinear behavior using dynamic nonlinearity estimators such as wavelet networks, tree-partitioning, and sigmoid networks.

There are also recursive estimation functions that let you fit models in real time as you collect the data, such as `recursiveARX`. The example “Detect Abrupt System Changes Using Identification Techniques” illustrates this approach.

For more functions you can use for model fitting, see “Identify Condition Indicators”.

Condition Indicators Based on Model Parameters or Dynamics

Any parameter of a model might serve as a useful condition indicator. As with static models, changes in model parameters or values outside of statistical confidence bounds can be indicative of fault conditions. For example, if you identify a state-space model using `ssest`, the pole locations or damping coefficients might change as a fault condition develops. You can use linear analysis functions such as `damp`, `pole`, and `zero` to extract dynamics from the estimated model.

Another approach is `modalfit`, which identifies dynamic characteristics by separating a signal into multiple modes with distinct frequency-response functions.

Sometimes, you understand some of your system dynamics and can represent them using differential equations or model structures with unknown parameters. For instance, you might be able to derive a model of your system in terms of physical parameters such as time constants, resonant frequencies, or damping coefficients, but the precise values of such parameters are unknown. In this case, you can use linear or nonlinear grey-box models to estimate parameter values, and track how those parameter values change with different fault conditions. Some functions for you can use for grey-box estimation include `pem` and `nlarx`.

A Simulink model can also serve as a grey-box model for parameter estimation. You can use Simulink to model your system under both healthy and faulty conditions using physically meaningful parameters, and estimate the values of those parameters based on system data (for instance, using the tools in Simulink Design Optimization™).

Condition Indicators Based on Residuals

Another way to use a dynamic model is to simulate the model and compare the result to the real data on which the model was based. The difference between system data and the results of simulating an estimated model is called the residual signal. The example “Fault Diagnosis of Centrifugal Pumps using Residual Analysis” analyzes the residual signal of an estimated `nlarx` model. The example computes several statistical and spectral features of the residual signal. It tests these candidate condition indicators to determine which provide the clearest distinction between healthy operation and several different faulty states.

Another residual-based approach is to identify multiple models for ensemble data representing different healthy and fault conditions. For test data, you then compute the residuals for each of these models. The model that yields the smallest residual signal (and therefore the best fit) indicates which healthy or fault condition most likely applies to the test data.

For residual analysis of an identified model obtained using commands such as `nlarx`, `ar`, or `ssest`, use:

- `sim` — Simulate the model response to an input signal.
- `resid` — Compute the residuals for the model.

As in the case parameter-based condition indicators, you can also use Simulink to construct models for residual analysis. The example “Fault Detection Using Data Based Models” also illustrates the residual-analysis approach, using a model identified from simulated data.

State Estimators

The values of system states can also serve as condition indicators. System states correspond to physical parameters, and abrupt or unexpected changes in state values can therefore indicate fault conditions. State estimators such as `unscentedKalmanFilter`, `extendedKalmanFilter`, and `particleFilter` let you track the values of system states in real time, to monitor for such changes. The following examples illustrate the use of state estimators for fault detection:

- “Fault Detection Using an Extended Kalman Filter”
- “Nonlinear State Estimation of a Degrading Battery System”

See Also

More About

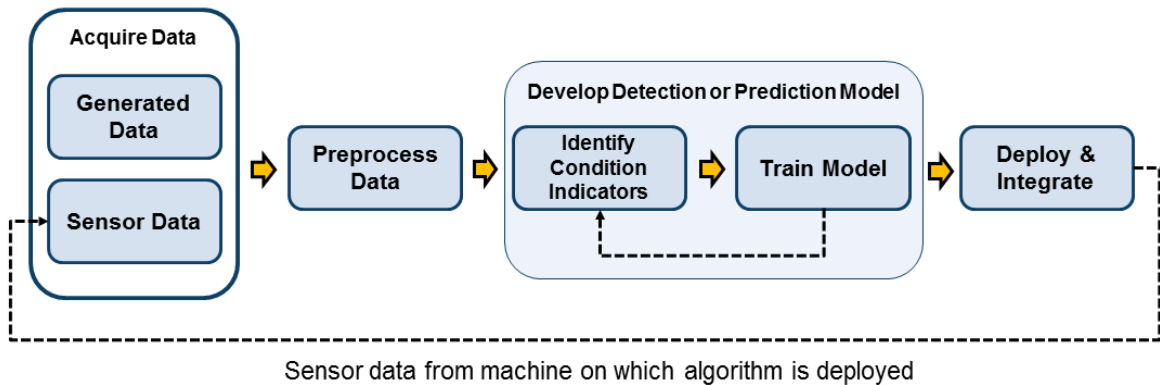
- “Condition Indicators for Monitoring, Fault Detection, and Prediction” on page 3-2
- “Signal-Based Condition Indicators” on page 3-4

Detect and Predict Faults

- “Decision Models for Fault Detection and Diagnosis” on page 4-2
- “Models for Predicting Remaining Useful Life” on page 4-8

Decision Models for Fault Detection and Diagnosis

Condition monitoring includes discriminating between faulty and healthy states (fault detection) or, when a fault state is present, determining the source of the fault (fault diagnosis). To design an algorithm for condition monitoring, you use condition indicators extracted from system data to train a decision model that can analyze indicators extracted from test data to determine the current system state. Thus, this step in the algorithm-design process is the next step after identifying condition indicators.



(For information about using condition indicators for fault prediction, see “Models for Predicting Remaining Useful Life” on page 4-8.)

Some examples of decision models for condition monitoring include:

- A threshold value or set of bounds on a condition-indicator value that indicates a fault when the indicator exceeds it
- A probability distribution that describes the likelihood that any particular value of the condition indicator is indicative of any particular type of fault
- A classifier that compares the current value of the condition indicator to values associated with fault states, and returns the likelihood that one or another fault state is present

In general, when you are testing different models for fault detection or diagnosis, you construct a table of values of one or more condition indicators. The condition indicators are features that you extract from data in an ensemble representing different healthy and

faulty operating conditions. (See “Condition Indicators for Monitoring, Fault Detection, and Prediction” on page 3-2.) It is useful to partition your data into a subset that you use for training the decision model (the training data) and a disjoint subset that you use for validation (the validation data). Compared to training and validation with overlapping data sets, using completely separate training and validation data generally gives you a better sense of how the decision model will perform with new data.

When designing your algorithm, you might test different fault detection and diagnosis models using different condition indicators. Thus, this step in the design process is likely iterative with the step of extraction condition indicators, as you try different indicators, different combinations of indicators, and different decision models.

Statistics and Machine Learning Toolbox™ and other toolboxes include functionality that you can use to train decision models such as classifiers and regression models. Some common approaches are summarized here.

Feature Selection

Feature selection techniques help you reduce large data sets by eliminating features that are irrelevant to the analysis you are trying to perform. In the context of condition monitoring, irrelevant features are those that do not separate healthy from faulty operation or help distinguish between different fault states. In other words, feature selection means identifying those features that are suitable to serve as condition indicators because they change in a detectable, reliable way as system performance degrades. Some functions for feature selection include:

- `pca` — Perform principal component analysis, which finds the linear combination of independent data variables that account for the greatest variation in observed values. For instance, suppose that you have ten independent sensor signals for each member of your ensemble from which you extract many features. In that case, principal component analysis can help you determine which features or combination of features are most effective for separating the different healthy and faulty conditions represented in your ensemble. The example “Wind Turbine High-Speed Bearing Prognosis” uses this approach to feature selection.
- `sequentialfs` — For a set of candidate features, identify the features that best distinguish between healthy and faulty conditions, by sequentially selecting features until there is no improvement in discrimination.
- `fscnca` — Perform feature selection for classification using neighborhood component analysis. The example “Using Simulink to Generate Fault Data” uses this function to

weight a list of extracted condition indicators according to their importance in distinguishing among fault conditions.

For more functions relating to feature selection, see “Dimensionality Reduction and Feature Extraction” (Statistics and Machine Learning Toolbox).

Statistical Distribution Fitting

When you have a table of condition indicator values and corresponding fault states, you can fit the values to a statistical distribution. Comparing validation or test data to the resulting distribution yields the likelihood that the validation or test data corresponds to one or the other fault states. Some functions you can use for such fitting include:

- `ksdensity` — Estimate a probability density for sample data.
- `histfit` — Generate a histogram from data, and fit it to a normal distribution. The example “Fault Diagnosis of Centrifugal Pumps using Steady State Experiments” uses this approach.
- `ztest` — Test likelihood that data comes from a normal distribution with specified mean and standard deviation.

For more information about statistical distributions, see “Probability Distributions” (Statistics and Machine Learning Toolbox).

Machine Learning

There are several ways to apply machine-learning techniques to the problem of fault detection and diagnosis. Classification is a type of supervised machine learning in which an algorithm “learns” to classify new observations from examples of labeled data. In the context of fault detection and diagnosis, you can pass condition indicators derived from an ensemble and their corresponding fault labels to an algorithm-fitting function that trains the classifier.

For instance, suppose that you compute a table of condition-indicator values for each member in an ensemble of data that spans different healthy and faulty conditions. You can pass this data to a function that fits a classifier model. This training data trains the classifier model to take a set of condition-indicator values extracted from a new data set, and guess which healthy or faulty condition applies to the data. In practice, you use a portion of your ensemble for training, and reserve a disjoint portion of the ensemble for validating the trained classifier.

Statistics and Machine Learning Toolbox includes many functions that you can use to train classifiers. These functions include:

- `fitcsvm` — Train a binary classification model to distinguish between two states, such as the presence or absence of a fault condition. The examples “Using Simulink to Generate Fault Data” use this function to train a classifier with a table of feature-based condition indicators. The example “Fault Diagnosis of Centrifugal Pumps using Steady State Experiments” also uses this function, with model-based condition indicators computed from statistical properties of the parameters obtained by fitting data to a static model.
- `fitcecoc` — Train a classifier to distinguish among multiple states. This function reduces a multiclass classification problem to a set of binary classifiers. The example “Multi-Class Fault Detection Using Simulated Data” uses this function.
- `fitctree` — Train a multiclass classification model by reducing the problem to a set of binary decision trees.
- `fitclinear` — Train a classifier using high-dimensional training data. This function can be useful when you have a large number of condition indicators that you are not able to reduce using functions such as `fscnca`.

Other machine-learning techniques include k-means clustering (`kmeans`), which partitions data into mutually exclusive clusters. In this technique, a new measurement is assigned to a cluster by minimizing the distance from the data point to the mean location of its assigned cluster. Tree bagging is another technique that aggregates an ensemble of decision trees for classification. The example “Fault Diagnosis of Centrifugal Pumps using Steady State Experiments” uses a `TreeBagger` classifier.

For more general information about machine-learning techniques for classification, see “Classification” (Statistics and Machine Learning Toolbox).

Regression with Dynamic Models

Another approach to fault detection and diagnosis is to use model identification. In this approach, you estimate dynamic models of system operation in healthy and faulty states. Then, you analyze which model is more likely to explain the live measurements from the system. This approach is useful when you have some information about your system that can help you select a model type for identification. To use this approach, you:

- 1 Collect or simulate data from the system operating in a healthy condition and in known faulty, degraded, or end-of-life conditions.

- 2 Identify a dynamic model representing the behavior in each healthy and fault condition.
- 3 Use clustering techniques to draw a clear distinction between the conditions.
- 4 Collect new data from a machine in operation and identify a model of its behavior. You can then determine which of the other models, healthy or faulty, is most likely to explain the observed behavior.

The example “Fault Detection Using Data Based Models” uses this approach. Functions you can use for identifying dynamic models include:

- `ssest`
- `arx`, `armax`, `ar`
- `nlrx`

You can use functions like `forecast` to predict the future behavior of the identified model.

Control Charts

Statistical process control (SPC) methods are techniques for monitoring and assessing the quality of manufactured goods. SPC is used in programs that define, measure, analyze, improve, and control development and production processes. In the context of predictive maintenance, control charts and control rules can help you determine when a condition-indicator value indicates a fault. For instance, suppose you have a condition indicator that indicates a fault if it exceeds a threshold, but also exhibits some normal variation that makes it difficult to identify when the threshold is crossed. You can use control rules to define the threshold condition as occurring when a specified number of sequential measurements exceeds the threshold, rather than just one.

- `controlchart` — Visualize a control chart.
- `controlrules` — Define control rules and determine whether they are violated.
- `cusum` — Detect small changes in the mean value of data.

For more information about statistical process control, see “Statistical Process Control” (Statistics and Machine Learning Toolbox).

Changepoint Detection

Another way to detect fault conditions is to track the value of a condition indicator over time and detect abrupt changes in the trend behavior. Such abrupt changes can be indicative of a fault. Some functions you can use for such changepoint detection include:

- `findchangepts` — Find abrupt changes in a signal.
- `findpeaks` — Find peaks in a signal.
- `pdist`, `pdist2`, `mahal` — Find the distance between measurements or sets of measurements, according to different definitions of distance.
- `segment` — Segment data and estimate AR, ARX, ARMA, or ARMAX models for each segment. The example “Fault Detection Using Data Based Models” uses this approach.

See Also

More About

- “Condition Indicators for Monitoring, Fault Detection, and Prediction” on page 3-2
- “Models for Predicting Remaining Useful Life” on page 4-8

Models for Predicting Remaining Useful Life

The remaining useful life (RUL) of a machine is the expected life or usage time remaining before the machine requires repair or replacement. Predicting remaining useful life from system data is a central goal of predictive-maintenance algorithms.

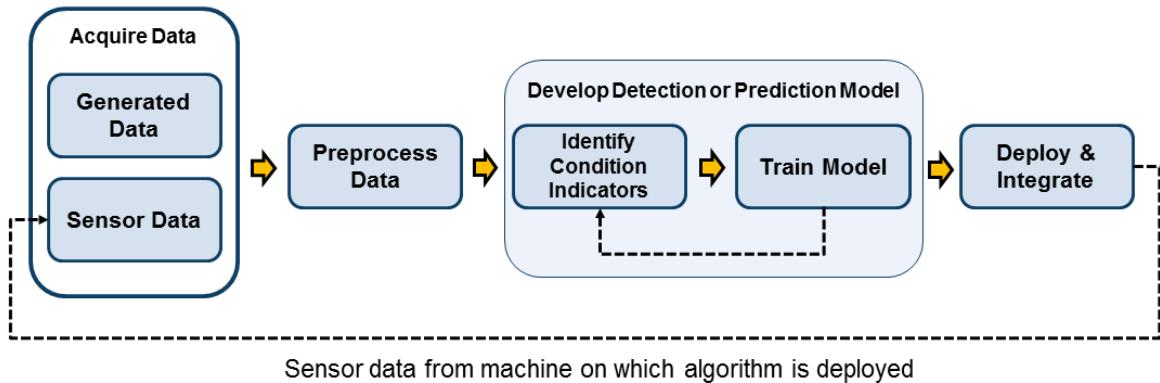
The term life time or usage time here refers to the life of the machine defined in terms of whatever quantity you use to measure system life. Units of life time can be quantities such as the distance travelled (miles), fuel consumed (gallons), repetition cycles performed, or time since the start of operation (days). Similarly time evolution can mean the evolution of a value with any such quantity.

Typically, you estimate the RUL of a system by developing a model that can perform the estimation based upon the time evolution or statistical properties of condition indicator values, such as:

- A model that fits the time evolution of a condition indicator and predicts how long it will be before the condition indicator crosses some threshold value indicative of a fault condition.
- A model that compares the time evolution of a condition indicator to measured or simulated time series from systems that ran to failure. Such a model can compute the most likely time-to-failure of the current system.

Predictions from such models are statistical estimates with associated uncertainty. They provide a probability distribution of the RUL of the test machine. The model you use can be a dynamic model such as those you obtain using System Identification Toolbox™ commands. Predictive Maintenance Toolbox also includes some specialized models designed for computing RUL from different types of measured system data.

Developing a model for RUL prediction is the next step in the algorithm-design process after identifying promising condition indicators (see “Condition Indicators for Monitoring, Fault Detection, and Prediction” on page 3-2). Because the model you develop uses the time evolution of condition indicator values to predict RUL, this step is often iterative with the step of identifying condition indicators.



RUL Estimation Using Identified Models or State Estimators

When you have an identified dynamic model that describes some aspect of system behavior, you can use that model to forecast future behavior. You can identify such a dynamic model from system data. Or, if you have system data that represents the operation of your machines with time or usage, you can extract condition indicators from that data and track the behavior of the condition indicators with time or usage. You can then identify a model that describes the behavior of the condition indicator, and use that model to predict future values of a condition indicator. If you know, for example, that your system needs repair when some condition indicator exceeds some threshold, you can identify a model of the time evolution of that condition indicator. You can then propagate the model forward in time to determine how long it will be before the condition indicator reaches the threshold value.

Some functions you can use for identification of dynamic models include:

- `ssest` — Estimate a state-space model from time-domain input-output data or frequency-response data.
- `arx`, `armax`, `ar` — Estimate an autoregressive or moving-average (AR or ARMA) model from time-series data.
- `nlarx` — Model nonlinear behavior using dynamic nonlinearity estimators such as wavelet networks, tree-partitioning, and sigmoid networks.

You can use functions like `forecast` to predict the future behavior of the identified model. The example “Condition Monitoring and Prognostics Using Vibration Signals” uses this approach to RUL prediction.

There are also recursive estimators that let you fit models in real-time as you collect and process the data, such as `recursiveARX` and `recursiveAR`.

RUL estimation with state estimators such as `unscentedKalmanFilter`, `extendedKalmanFilter`, and `particleFilter` works in a similar way. You perform state estimation on some time-varying data, and predict future state values to determine the time until some state value associated with failure occurs.

RUL Estimation Using RUL Estimator Models

Predictive Maintenance Toolbox includes some specialized models designed for computing RUL from different types of measured system data. These models are useful when you have historical data and information such as:

- Run-to-failure histories of machines similar to the one you want to diagnose
- A known threshold value of some condition indicator that indicates failure
- Data about how much time or how much usage it took for similar machines to reach failure (life time)

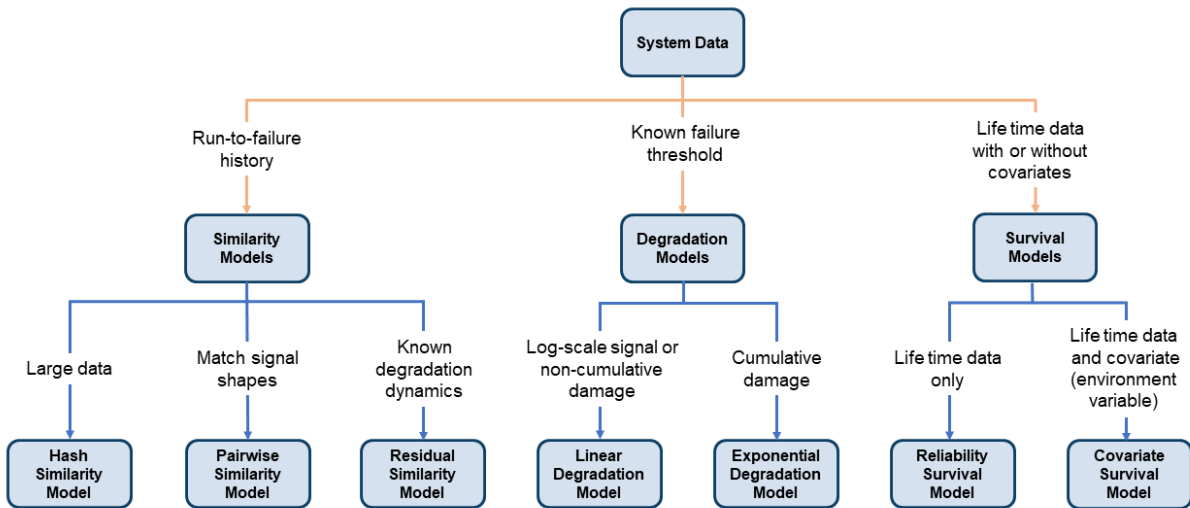
RUL estimation models provide methods for training the model using historical data and using it for performing prediction of the remaining useful life. The term life time here refers to the life of the machine defined in terms of whatever quantity you use to measure system life. Similarly time evolution can mean the evolution of a value with usage, distance traveled, number of cycles, or other quantity that describes life time.

The general workflow for using RUL estimation models is:

- 1** Choose the best type of RUL estimation model for the data and system knowledge you have. Create and configure the corresponding model object.
- 2** Train the estimation model using the historical data you have. To do so, use the `fit` command.
- 3** Using test data of the same type as your historical data, estimate the RUL of the test component. To do so, use the `predictRUL` command. You can also use the test data recursively to update some model types, such as degradation models, to help keep the predictions accurate. To do so, use the `update` commands.

Choose an RUL Estimator

There are three families of RUL estimation models. Choose which family and which model to use based on the data and system information you have available, as shown in the following illustration.



Similarity Models

Similarity models base the RUL prediction of a test machine on known behavior of similar machines from a historical database. Such models compare a trend in test data or condition-indicator values to the same information extracted from other, similar systems.

Similarity models are useful when:

- You have run-to-failure data from similar systems (components). Run-to-failure data is data that starts during healthy operation and ends when the machine is in a state close to failure or maintenance.
- The run-to-failure data shows similar degradation behaviors. That is, the data changes in some characteristic way as the system degrades.

Thus you can use similarity models when you can obtain degradation profiles from your data ensemble. The degradation profiles represent the evolution of one or more condition

indicators for each machine in the ensemble (each component), as the machine transitions from a healthy state to a faulty state.

Predictive Maintenance Toolbox includes three types of similarity models. All three types estimate RUL by determining the similarity between the degradation history of a test data set and the degradation history of data sets in the ensemble. For similarity models, `predictRUL` estimates the RUL of the test component as the median life span of most similar components minus the current life time value of the test component. The three models differ in the ways they define and quantify the notion of similarity.

- Hashed-feature similarity model (`hashSimilarityModel`) — This model transforms historical degradation data from each member of your ensemble into fixed-size, condensed, information such as the mean, total power, maximum or minimum values, or other quantities.

When you call `fit` on a `hashSimilarityModel` object, the software computes these hashed features and stores them in the similarity model. When you call `predictRUL` with data from a test component, the software computes the hashed features and compares the result to the values in the table of historical hashed features.

The hashed-feature similarity model is useful when you have large amounts of degradation data, because it reduces the amount of data storage necessary for prediction. However, its accuracy depends on the accuracy of the hash function that the model uses. If you have identified good condition indicators in your data, you can use the `Method` property of the `hashSimilarityModel` object to specify the hash function to use those features.

- Pairwise similarity model (`pairwiseSimilarityModel`) — Pairwise similarity estimation determines RUL by finding the components whose historical degradation paths are most correlated to that of the test component. In other words, it computes the distance between different time series, where distance is defined as correlation, dynamic time warping (`dtw`), or a custom metric that you provide. By taking into account the degradation profile as it changes over time, pairwise similarity estimation can give better results than the hash similarity model.
- Residual similarity model (`residualSimilarityModel`) — Residual-based estimation fits prior data to model such as an ARMA model or a model that is linear or exponential in usage time. It then computes the residuals between data predicted from the ensemble models and the data from the test component. You can view the residual similarity model as a variation on the pairwise similarity model, where the magnitudes of the residuals is the distance metric. The residual similarity approach is useful when your knowledge of the system includes a form for the degradation model.

For an example that uses a similarity model for RUL estimation, see “Similarity-Based Remaining Useful Life Estimation”.

Degradation Models

Degradation models extrapolate past behavior to predict the future condition. This type of RUL calculation fits a linear or exponential model to degradation profile of a condition indicator, given the degradation profiles in your ensemble. It then uses the degradation profile of the test component to statistically compute the remaining time until the indicator reaches some prescribed threshold. These models are most useful when there is a known value of your condition indicator that indicates failure. The two available degradation model types are:

- Linear degradation model (`linearDegradationModel`) — Describes the degradation behavior as a linear stochastic process with an offset term. Linear degradation models are useful when your system does not experience cumulative degradation.
- Exponential degradation model (`exponentialDegradationModel`) — Describes the degradation behavior as an exponential stochastic process with an offset term. Exponential degradation models are useful when the test component experiences cumulative degradation.

After you create a degradation model object, initialize the 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 `fit`. You can then predict the remaining useful life of similar components using `predictRUL`.

Degradation models only work with a single condition indicator. However, you can use principal-component analysis or other fusion techniques to generate a fused condition indicator that incorporates information from more than one condition indicator. Whether you use a single indicator or a fused indicator, look for an indicator that shows a clear increasing or decreasing trend, so that the modeling and extrapolation are reliable.

For an example that takes this approach and estimates RUL using a degradation model, see “Wind Turbine High-Speed Bearing Prognosis”.

Survival Models

Survival analysis is a statistical method used to model time-to-event data. It is useful when you do not have complete run-to-failure histories, but instead have:

- Only data about the life span of similar components. For example, you might know how many miles each engine in your ensemble ran before needing maintenance, or how many hours of operation each machine in your ensemble ran before failure. In this case, you use `reliabilitySurvivalModel`. Given the historical information on failure times of a fleet of similar components, this model estimates the probability distribution of the failure times. The distribution is used to estimate the RUL of the test component.
- Both life spans and some other variable data (covariates) that correlates with the RUL. Covariates, also called environmental variables or explanatory variables, comprise information such as the component provider, regimes in which the component was used, or manufacturing batch. In this case, use `covariateSurvivalModel`. This model is a proportional hazard survival model which uses the life spans and covariates to compute the survival probability of a test component.

See Also

`covariateSurvivalModel` | `exponentialDegradationModel` | `fit` | `hashSimilarityModel` | `linearDegradationModel` | `pairwiseSimilarityModel` | `predictRUL` | `reliabilitySurvivalModel` | `residualSimilarityModel`

More About

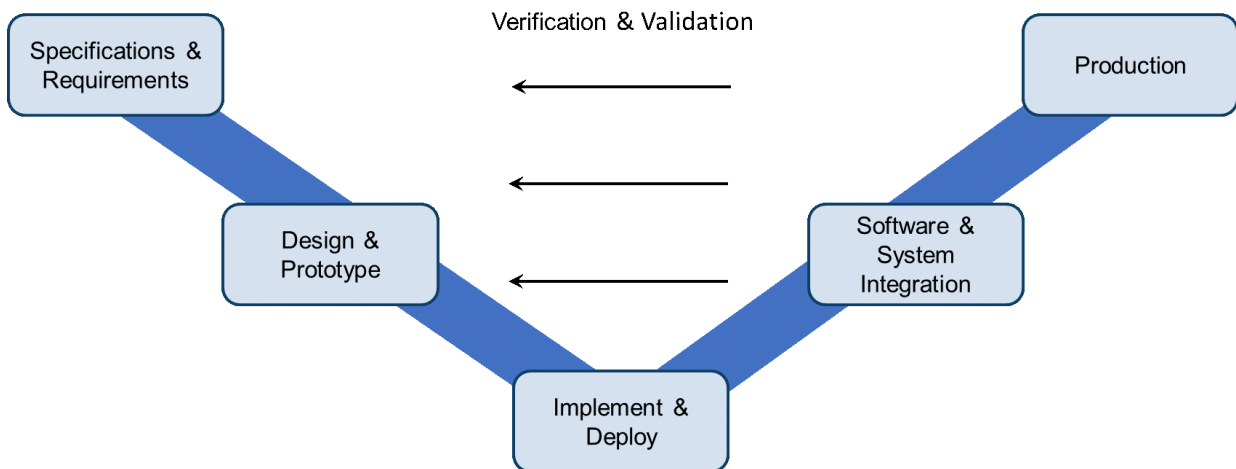
- “Condition Indicators for Monitoring, Fault Detection, and Prediction” on page 3-2
- “Similarity-Based Remaining Useful Life Estimation”
- “Wind Turbine High-Speed Bearing Prognosis”

Deploy Predictive Maintenance Algorithms

Deploy Predictive Maintenance Algorithms

Deployment or integration of a predictive maintenance algorithm is typically the final stage of the algorithm-development workflow. How you ultimately deploy the algorithm can also be a consideration in earlier stages of algorithm design. For example, whether the algorithm runs on embedded hardware, as a stand-alone executable, or as a web application can have impact on requirements and other aspects of the complete predictive-maintenance system design. MathWorks® products support several phases of the process for developing, deploying, and validating predictive maintenance algorithms.

The design-V, a conceptual diagram often used in the context of Model-Based Design, is also relevant when considering the design and deployment of a predictive maintenance algorithm. The design-V highlights the key deployment and implementation phases:



Specifications and Requirements

Developing specifications and requirements includes considerations both from the predictive maintenance algorithm perspective and the deployment perspective. Predictive maintenance algorithm requirements come from an understanding of the system coupled with mathematical analysis of the process, its signals, and expected faults. Deployment requirements can include requirements on:

- Memory and computational power.

- Operating mode. For instance, the algorithm might be a batch process that runs at some fixed time interval such as once a day. Or, it might be a streaming process that runs every time new data is available.
- Maintenance or update of the algorithm. For example, the deployed algorithm might be fixed, changing only changes through occasional updates. Or, you might develop an algorithm that adapts and automatically updates as new data is available.
- Where the algorithm runs, such as whether the algorithm must run in a cloud, or be offered as a web service.

Design and Prototype

This phase of the design-V includes data management, design of data preprocessing, identification of condition indicators, and training of a classification model for fault detection or a model for estimating remaining useful life. (See “Designing Algorithms for Condition Monitoring and Predictive Maintenance”, which provides an overview of the algorithm-design process.) In the design phase, you often use historic or synthesized data to test and tune the developed algorithm.

Implement and Deploy

Once you have developed a candidate algorithm, the next phase is to implement and deploy the algorithm. MathWorks products support many different application needs and resource constraints, ranging from standalone applications to web services.

- MATLAB Compiler™ — Create standalone applications or shared libraries to execute algorithms developed using Predictive Maintenance Toolbox. You can use MATLAB Compiler to deploy MATLAB code in many ways, including as a standalone Windows® application, a shared library, an Excel® add-in, a Microsoft® .NET assembly, or a generic COM component. Such applications or libraries run outside the MATLAB environment using the MATLAB Runtime, which is freely distributable. The MATLAB Runtime can be packaged and installed with your application, or downloaded during the installation process. For more information about deployment with MATLAB Compiler, see “Getting Started with MATLAB Compiler” (MATLAB Compiler).
- MATLAB Production Server™ — Integrate your algorithms into web, database, and enterprise applications. MATLAB Production Server leverages the MATLAB Compiler to run your applications on dedicated servers or a cloud. You can package your predictive maintenance algorithms using MATLAB Compiler SDK™, which extends the functionality of MATLAB Compiler to let you build C/C++ shared libraries, Microsoft .NET assemblies, Java® classes, or Python® packages from MATLAB

programs. Then, you can deploy the generated libraries to MATLAB Production Server without recoding or creating custom infrastructure.

- **ThingSpeak™** — This Internet of Things (IoT) analytics platform service lets you aggregate, visualize, and analyze live data streams in the cloud. For diagnostics and prognostics algorithms that run at intervals of 5 minutes or longer, you can use the ThingSpeak IoT platform to visualize results and monitor the condition of your system. You can also use ThingSpeak as a quick and easy prototyping platform before deployment using the MATLAB Production Server. You can transfer diagnostic data using ThingSpeak web services and use its charting tools to create dashboards for monitoring progress and generating failure alarms. ThingSpeak can communicate directly with desktop MATLAB or MATLAB code embedded in target devices.
- **MATLAB Coder™** and **Simulink Coder** — Generate C/C++ code from MATLAB or Simulink. Many Signal Processing Toolbox, Statistics and Machine Learning Toolbox, and System Identification Toolbox functions support MATLAB Coder. For example, you can generate code from algorithms that use the System Identification Toolbox state estimation (such as `extendedKalmanFilter`) and recursive parameter estimation (such as `recursiveAR`) functionality. See “Functions and Objects Supported for C/C++ Code Generation — Category List” (MATLAB Coder) for a more comprehensive list.

One choice you often have to make is to whether to deploy your algorithm on an embedded system or on the cloud.

A cloud implementation can be useful when you are gathering and storing large amounts of data on the cloud. Removing the need to transfer data between the cloud and local machines that are running the prognostics and health monitoring algorithm makes the maintenance process more effective. Results calculated on the cloud can be made available through tweets, email notifications, web apps, and dashboards. For cloud implementations, you can use ThingSpeak or MATLAB Production Server.

Alternatively, the algorithm can run on embedded devices that are closer to the actual equipment. The main benefits of doing this are that the amount of information sent is reduced as data is transmitted only when needed, and updates and notifications about equipment health are immediately available without any delay. For embedded implementations, you can use MATLAB Compiler, MATLAB Coder, or Simulink Coder to generate code that runs on a local machine.

A third option is to use a combination of the two. The preprocessing and feature extraction parts of the algorithm can be run on embedded devices, while the predictive model can run on the cloud and generate notifications as needed. In systems such as oil drills and aircraft engines that are run continuously and generate huge amounts of data,

storing all the data on board or transmitting it is not always viable because of cellular bandwidth and cost limitations. Using an algorithm that operates on streaming data or on batches of data lets you store and send data only when needed.

Software and System Integration

After you have developed a deployment candidate, you test and validate algorithm performance under real-life conditions. This phase can include designing tests for verification, software-in-the-loop testing, or hardware-in-the-loop testing. This phase is critical to validate both the requirements and the developed algorithm. It often leads to revisions in the requirements, the algorithm, or the implementation, iterating on earlier phases in the design-V.

Production

Finally, you put the algorithm into production. Often this phase includes performance monitoring and further iteration on the design requirements and algorithm as you gain operational experience.

More About

- “Designing Algorithms for Condition Monitoring and Predictive Maintenance”

