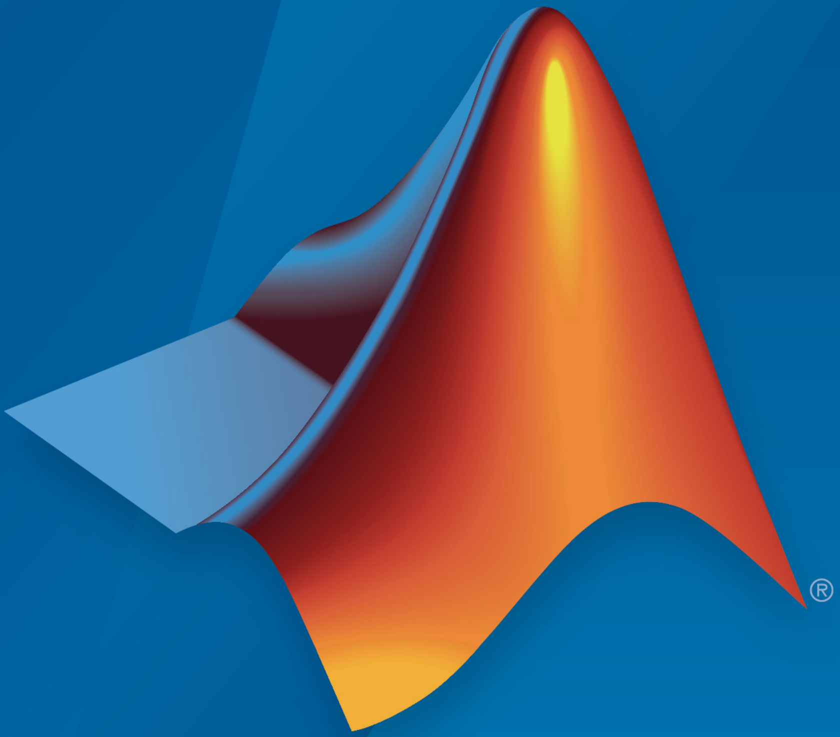
Predictive Maintenance Toolbox™

User's Guide



MATLAB®

R2018b

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

Revision History

March 2018	Online only	New for Version 1.0 (Release 2018a)
September 2018	Online only	Revised for Version 1.1 (Release 2018b)

Contents

3 Identify Condition Indicators

4 Detect and Diagnose Faults

Manage System Data

Data Ensembles for Condition Monitoring and Predictive Maintenance

Data analysis is the heart of any condition monitoring and predictive maintenance activity. Predictive Maintenance Toolbox provides tools called ensemble datastores for creating, labeling, and managing the often large, complex data sets needed for predictive maintenance algorithm design.

The data can come from measurements on systems using sensors such as accelerometers, pressure gauges, thermometers, altimeters, voltmeters, and tachometers. For instance, 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.

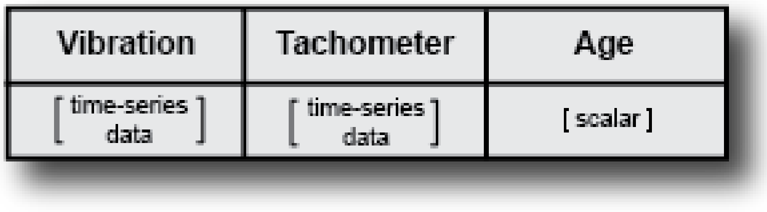
Whether using measured data, generated data, or both, you frequently have many signals, ranging over a time span or multiple time spans. You might also have signals from many machines (for example, measurements from 100 separate engines all manufactured to the same specifications). And you might have 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.

Ensemble datastores can help you work with such data, whether it 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

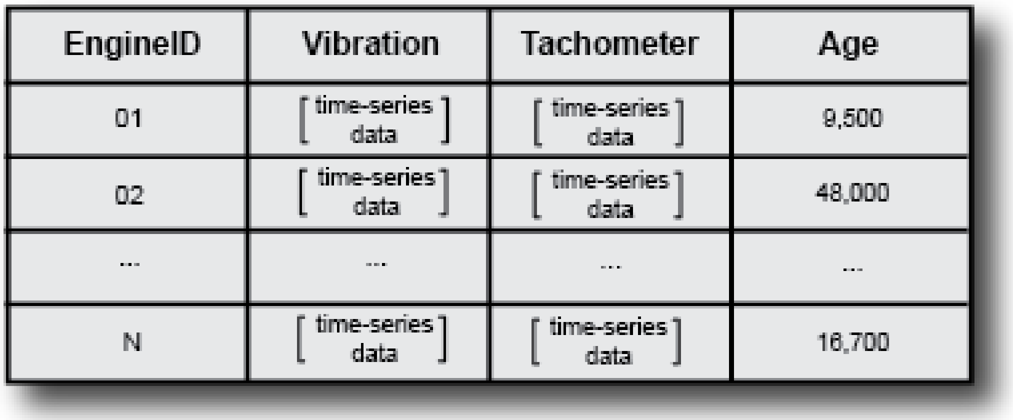
The main unit for organizing and managing multifaceted data sets in Predictive Maintenance Toolbox is the data ensemble. An ensemble is a collection of data sets, created by measuring or simulating a system under varying conditions.

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]

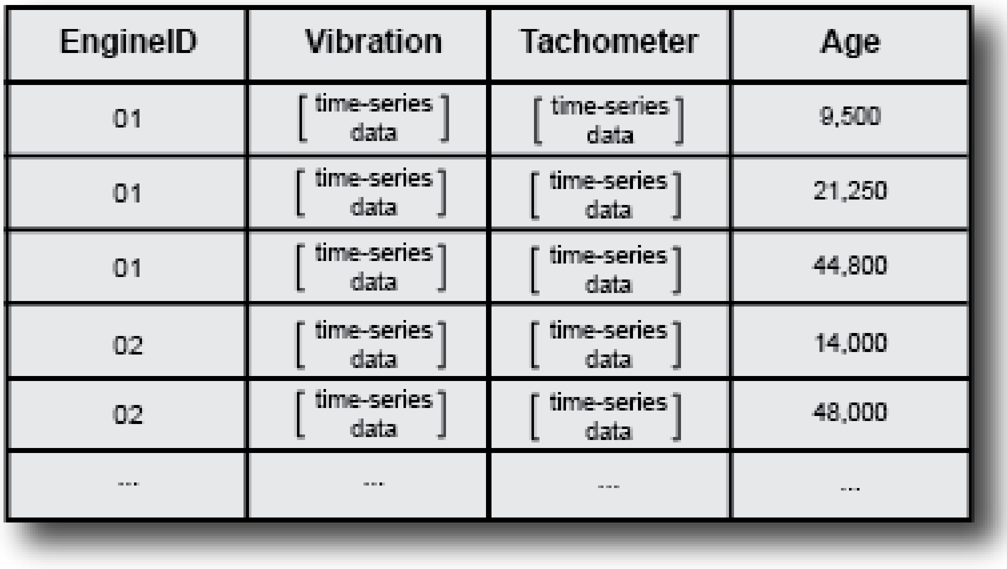
Now suppose that you have a fleet of many identical engines, and you record data from all of them. Doing so yields a family of data sets.



EngineID	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 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.



EngineID	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
...	...	...	...

In practice, the data for each ensemble member is typically stored in a separate data file. Thus, for instance, you might have one file containing the data for engine 01 at 9,500 miles, another file containing the data for engine 01 at 21,250 miles, and so on.

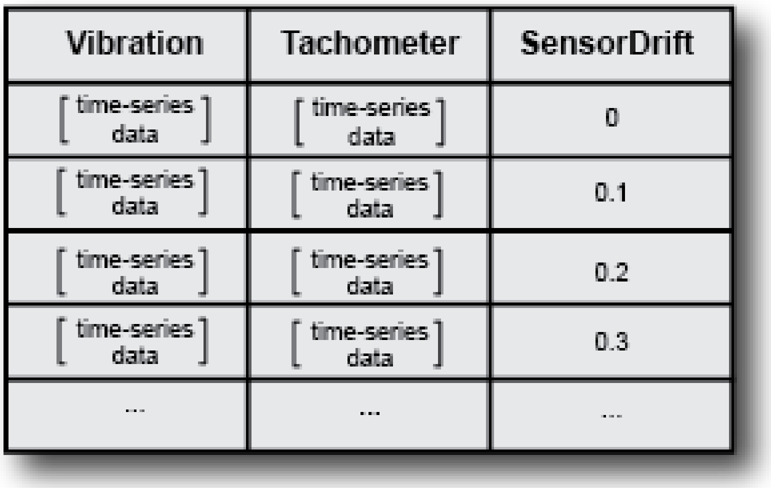
Simulated Ensemble Data

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	SensorDrift
[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
...	...	...

The `generateSimulationEnsemble` command helps you generate such data sets from a model in which you can simulate fault conditions by varying some aspect of the model.

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, SensorDrift is a condition variable. Condition variables can also be derived values, such as a single scalar value that encodes multiple fault and operating conditions.

In practice, your data variables, independent variables, and condition variables are all distinct sets of variables.

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 model.

- 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

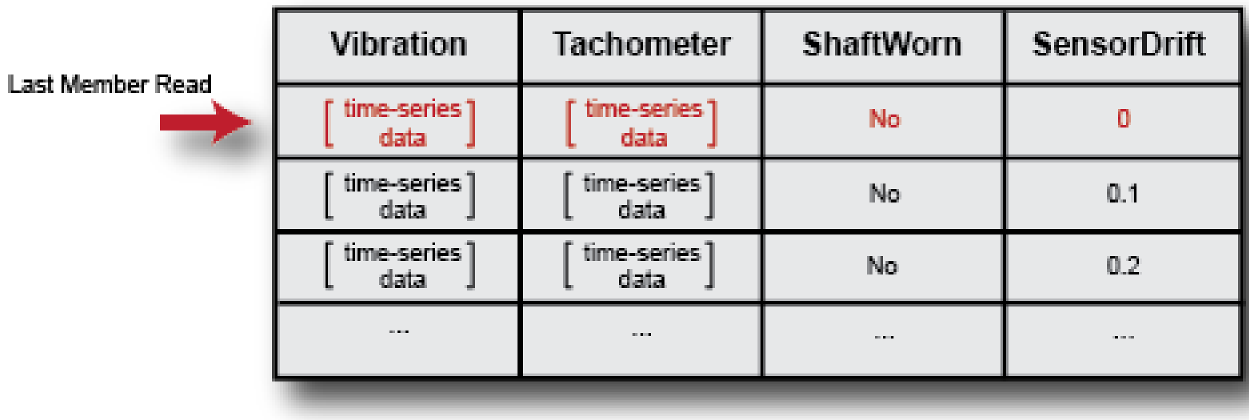
When you work with an ensemble, the software keeps track of which ensemble member it has most recently read. When you call `read`, the software selects the next member to read and updates the `LastMemberRead` property of the ensemble to reflect that member. When you next call `writeToLastMemberRead`,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

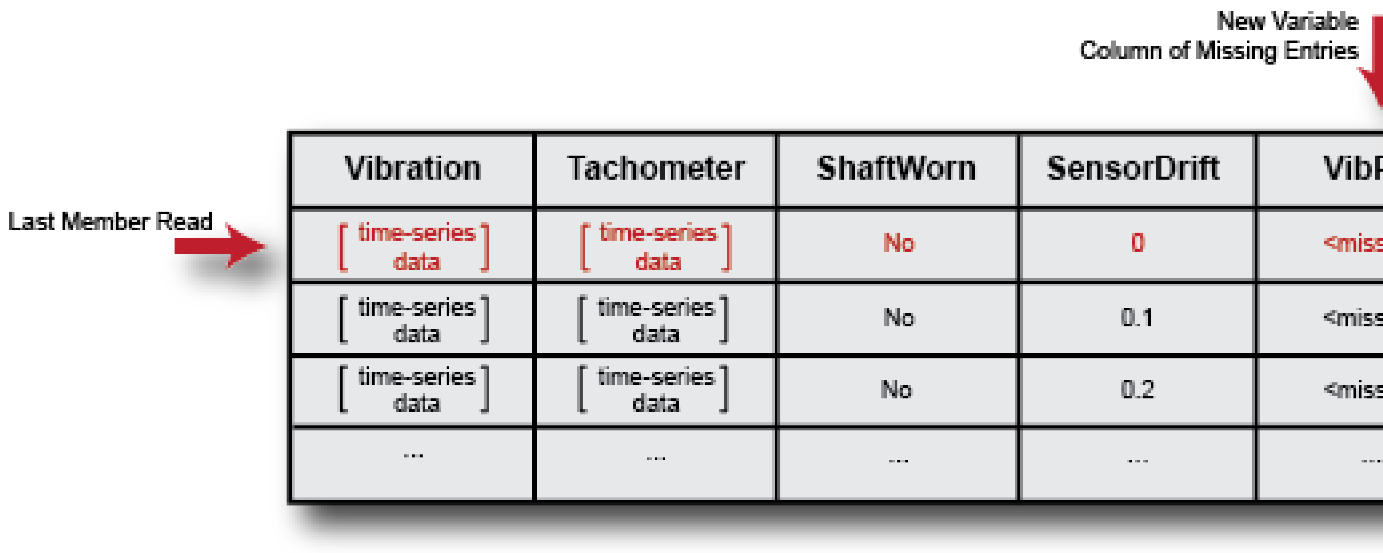


Vibration	Tachometer	ShaftWorn	SensorDrift
[time-series data]	[time-series data]	No	0
[time-series data]	[time-series data]	No	0.1
[time-series data]	[time-series data]	No	0.2
...	...	...	...

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.)

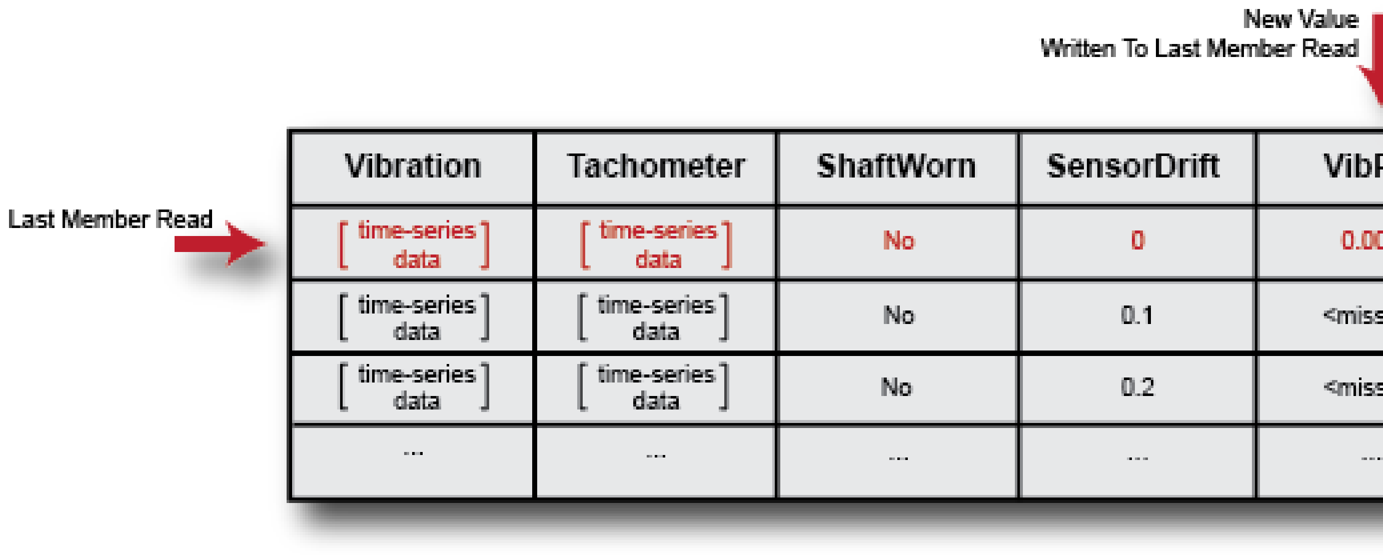


Vibration	Tachometer	ShaftWorn	SensorDrift	VibP
[time-series data]	[time-series data]	No	0	<miss
[time-series data]	[time-series data]	No	0.1	<miss
[time-series data]	[time-series data]	No	0.2	<miss
...	...	...	...	...

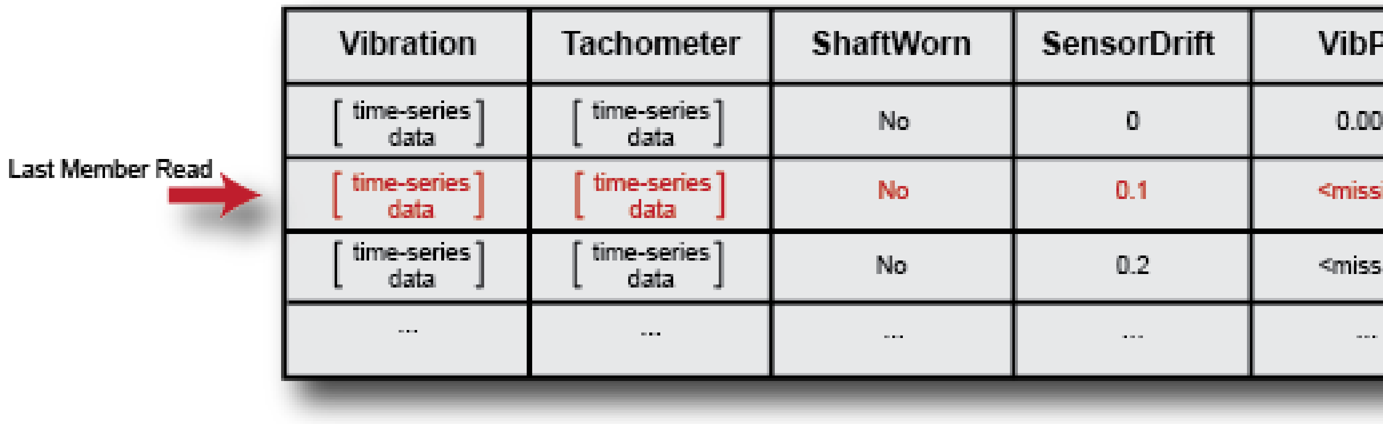
Now, use writeToLastMemberRead to fill in the value of the new variable for the last member read.

```
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.



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.



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 1-14.

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. Set the `ReadFcn` property to the handle of a function that describes how to read the data variables from a data file. When you call `read`, it uses this function to access the next ensemble file, and to read from it the variables specified in the `SelectedVariables` property of the ensemble datastore. 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 examples that show these interactions with an ensemble of measured data on disk, see:

- “File Ensemble Datastore With Measured Data” on page 1-23
- “File Ensemble Datastore Using Data in Text Files” on page 1-29

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.

Reading from Multiple Ensemble Members

By default, the `read` command returns data from one ensemble member at a time. To process data from more than one ensemble member at a time, set the `ReadSize` of the ensemble datastore object to a value greater than 1. For instance, if you set `ReadSize` to 3, then each call to `read` returns a table with three rows, and designates three ensemble members as last member read. For details, see the `fileEnsembleDatastore` and `simulationEnsembleDatastore` reference pages.

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" on page 4-8
- "Using Simulink to Generate Fault Data" on page 1-35

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 1-14

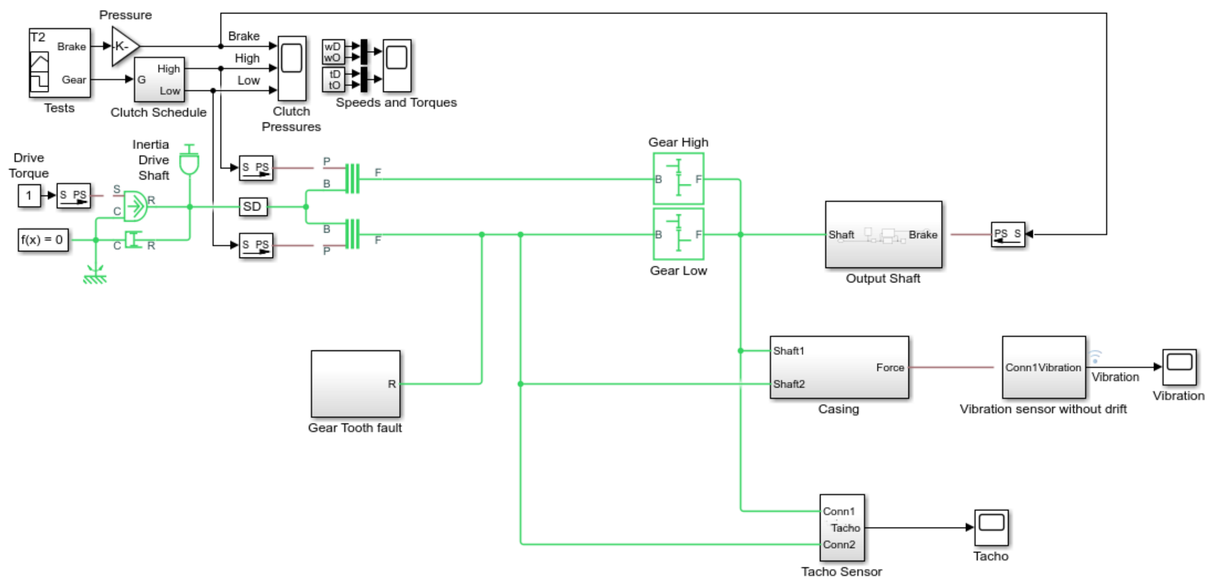
- “File Ensemble Datastore With Measured Data” on page 1-23
- “File Ensemble Datastore Using Data in Text Files” on page 1-29

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” on page 1-35. 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` (see “Export Signal Data Using Signal Logging” (Simulink)). The `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. If all the simulations complete without error, the function returns `true` in the indicator output, `status`.

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

```
[27-Aug-2018 14:29:47] Running SetupFcn...
[27-Aug-2018 14:29:47] Running simulations...
[27-Aug-2018 14:30:29] Completed 1 of 5 simulation runs
[27-Aug-2018 14:30:41] Completed 2 of 5 simulation runs
[27-Aug-2018 14:30:52] Completed 3 of 5 simulation runs
[27-Aug-2018 14:31:05] Completed 4 of 5 simulation runs
[27-Aug-2018 14:31:19] Completed 5 of 5 simulation runs
```

```
status
```

```
status = logical
   1
```

Inside the `Data` folder, examine one of the files. Each file is a MAT-file containing the following MATLAB® variables:

- `SimulationInput` — The `Simulink.SimulationInput` object that was used to configure the model for generating the data in the file. You can use this to extract information about the conditions (such as faulty or healthy) under which this simulation was run.
- `logsout` — A `Dataset` object containing all the data that the Simulink model is configured to log.
- `PMSignalLogName` — The name of the variable that contains the logged data (`'logsout'` in this example). The `simulationEnsembleDatastore` command uses this name to parse the data in the file.
- `SimulationMetadata` — Other information about the simulation that generated the data logged in the file.

Now you can create the simulation ensemble datastore using the generated data. The resulting `simulationEnsembleDatastore` object points to the generated data. The object lists the data variables in the ensemble, and by default all the variables are selected for reading.

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

```
ensemble = 
  simulationEnsembleDatastore with properties:

           DataVariables: [4x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: [0x0 string]
       SelectedVariables: [4x1 string]
                ReadSize: 1
              NumMembers: 5
          LastMemberRead: [0x0 string]
                   Files: [5x1 string]
```

```
ensemble.DataVariables
```

```
ans = 4x1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
```

```
ensemble.SelectedVariables
```

```
ans = 4x1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
```

Read Data from Ensemble Members

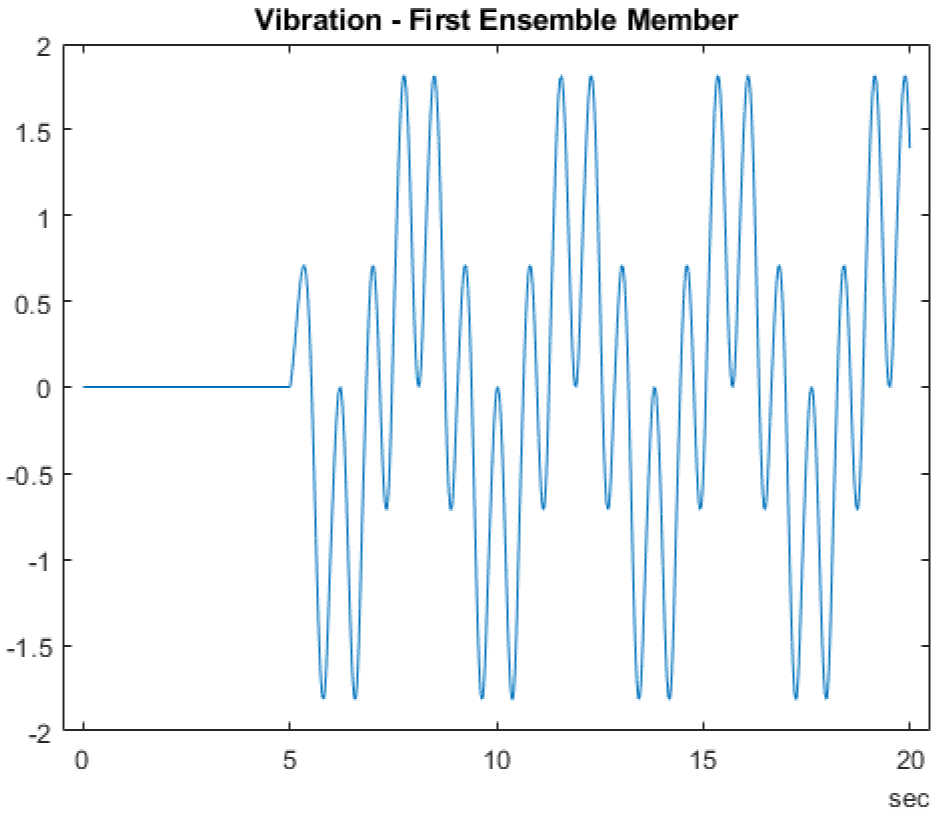
Suppose that for the analysis you want to do, you need only the `Vibration` data and the `Simulink.SimulationInput` object that describes the conditions under which each member was simulated. Set `ensemble.SelectedVariables` to specify the variables you want to read. The `read` command then extracts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

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

```
data1=1×2 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
    ___________________    ______________________________

    [20215x1 timetable]    [1x1 Simulink.SimulationInput]
```

To confirm that `data1` and `data2` contain data from different ensemble members, examine the values of the varied model parameter, `ToothFaultGain`. For each ensemble, this value is stored in the `Variables` field of the `SimulationInput` variable.

```
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 < 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 = 5x1 string array
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
    "ToothFault"
```

Then, use `writeToLastMemberRead` to write a value for new variable to the last-read member of the ensemble.

```
writeToLastMemberRead(ensemble,'ToothFault',sT);
```

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);
    writeToLastMemberRead(ensemble,'ToothFault',sT);
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"
```

Now, each ensemble member contains the original unprocessed data and an additional variable indicating the fault condition under which the data was collected. In practice, you might compute and append other values derived from the raw vibration data, to identify potential condition indicators that you can use for fault detection and diagnosis.

For a more detailed example that shows more ways to manipulate and analyze data stored in a `simulationEnsembleDatastore` object, see “Using Simulink to Generate Fault Data” on page 1-35.

Read Multiple Members at Once

If it is efficient or useful for the processing you want to do, you can configure the ensemble to read data from multiple members at once. To do so, use the `ReadSize` property. The `read` command uses this property to determine how many ensemble members to read at one time. For example, configure the ensemble to read two members at a time.

```
ensemble.ReadSize = 2;
```

Changing the value of `ReadSize` also resets the ensemble to its unread state. Thus, the next read operation reads the first two ensemble members. `read` returns a table with a number of rows equal to `ReadSize`.

```
ensemble.SelectedVariables = ["Vibration";"ToothFault"];
data3 = read(ensemble)
```

```
data3=2×2 table
         Vibration          ToothFault
    ___________________    __________

    [20202x1 timetable]      false
    [20215x1 timetable]      false
```

The `LastMemberRead` property of the ensemble contains the file names of all ensemble members that were read in this operation.

```
ensemble.LastMemberRead
```

```
ans = 2x1 string array
    "C:\TEMP\Bdoc18b_943130_7372\ib632619\14\tpf2dcb597\predmaint-ex54897023\Data\Trans
    "C:\TEMP\Bdoc18b_943130_7372\ib632619\14\tpf2dcb597\predmaint-ex54897023\Data\Trans
```

When you append data to an ensemble datastore that has `ReadSize` > 1, you must write to the same number of ensemble members as you read. Thus, for instance, when `ReadSize` = 2, supply a two-row table to `writeToLastMemberRead`.

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

```
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 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 in the current folder.

```
fensemble = fileEnsembleDatastore(pwd,'.mat');
```

Before you can interact with data in the ensemble, you must create functions that tell the software how to process the data files to read variables into the MATLAB® workspace and to write data back to the files. For this example, use the following provided functions:

- `readBearingData` — Extract requested variables from a structure, `bearing`, and other variables stored in the file. This function also parses the file name for the fault status of the data. The function returns a table row containing one table variable for each requested variable.
- `writeBearingData` — Take a structure and write its variables to a data file as individual stored variables.

Assign these functions to the `ReadFcn` and `WriteToMemberFcn` properties of the ensemble datastore, respectively.

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

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

Finally, set properties of the ensemble to identify data variables and condition variables.

```
fensemble.DataVariables = ["gs";"sr";"load";"rate"];
fensemble.ConditionVariables = ["label";"file"];
```

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

```
fensemble
```

```
fensemble = 
  fileEnsembleDatastore with properties:

                 ReadFcn: @readBearingData
        WriteToMemberFcn: @writeBearingData
           DataVariables: [4x1 string]
    IndependentVariables: [0x0 string]
      ConditionVariables: [2x1 string]
       SelectedVariables: [0x0 string]
                ReadSize: 1
              NumMembers: 5
          LastMemberRead: [0x0 string]
                   Files: [5x1 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. Thus, when you call the `read` command, it uses `readBearingData` to read all the variables in `fensemble.SelectedVariables`.

Specify variables to read, and read them from the first member of the ensemble. The `read` command reads data from the first ensemble member into a table row in the MATLAB workspace. The software determines which ensemble member to read first.

```
fensemble.SelectedVariables = ["file";"label";"gs";"sr";"load";"rate"];
data = read(fensemble)
```

```
data=1×6 table
     label           file                  gs               sr       load     rate 
    ________    ______________    _________________    _____    ____    ____

    "Faulty"    "FaultData_01"    [146484x1 double]    48828      0        25  
```

Write Data to Ensemble Member

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
```

You can write the frequency vector `fpdata` and the power spectrum `pdata` to the data file as separate variables. First, add the new variables to the list of data variables in the ensemble datastore.

```
fensemble.DataVariables = [fensemble.DataVariables;"freq";"spectrum"];
fensemble.DataVariables
```

```
ans = 6x1 string array
    "gs"
    "sr"
    "load"
    "rate"
    "freq"
    "spectrum"
```

Next, write the new values to the file corresponding to the last-read ensemble member. When you call `writeToLastMemberRead`, it converts the data to a structure and calls `fensemble.WriteToMemberFcn` to write the data to the file.

```
writeToLastMemberRead(fensemble,'freq',fpdata,'spectrum',pdata);
```

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=1×6 table
     label           file                  gs               sr      load    rate
    _________    _______________    ___________________    _____    ____    ____

    "Faulty"     "FaultData_02"     [146484x1 double]      48828     50      25
```

You can confirm 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 contains the two new variables you already added.) 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);
    writeToLastMemberRead(fensemble,'freq',fpdata,'spectrum',pdata);
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` and `freq` variables 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” on page 4-8. That example also shows the use of Parallel Computing Toolbox™ to speed up the processing of a larger ensemble.

To confirm that the derived variables are present in the file ensemble datastore, read them from the first and second ensemble members. To do so, reset the ensemble again, and add the new variables 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 variables but do not include the unprocessed data, `gs`.

```
reset(fensemble)
fensemble.SelectedVariables = ["label","load","freq","spectrum"];
data1 = read(fensemble)
```

```
data1=1×4 table
     label      load          freq              spectrum     
    ________    ____    ________________    ________________

    "Faulty"     0      [4096x1 double]     [4096x1 double]
```

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

```
data2=1×4 table
     label      load          freq              spectrum     
    ________    ____    ________________    ________________

    "Faulty"     50     [4096x1 double]     [4096x1 double]
```

```
rmpath(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
- “File Ensemble Datastore Using Data in Text Files” on page 1-29

File Ensemble Datastore Using Data in Text Files

In predictive maintenance algorithm design, you frequently have system data in a plain text format such as comma-separated values (CSV). This example shows how to create and use a `fileEnsembleDatastore` object to manage an ensemble of data stored in such a format.

Ensemble Data

Extract the compressed data for the example.

```
unzip fleetdata.zip  % extract compressed files
```

The ensemble consists of ten files, `fleetdata_01.txt, ..., fleetdata_10.txt`, each containing data for one car in a fleet of cars. Each file contains five unlabeled columns of data, corresponding to daily readings of the following values:

- Odometer reading at the end of the day, in miles
- Fuel consumed that day, in gallons
- Maximum rpm for the day
- Maximum engine temperature for the day, in degrees Celsius
- Engine light status at the end of the day (0 = off, 1 = on)

Each file contains data for between about 80 and about 120 days of operation. The data sets were artificially manufactured for this example and do not correspond to real fleet data.

Configure the Ensemble Datastore

Create a `fileEnsembleDatastore` object to manage the data.

```
location = pwd;
extension = '.txt';
fensemble = fileEnsembleDatastore(location,extension);
```

Configure the ensemble datastore to use the provided function `readFleetData.m` to read data from the files.

```
addpath(fullfile(matlabroot,'examples','predmaint','main')) % Make sure functions are o
fensemble.ReadFcn = @readFleetData;
```

Because the columns in the data files are unlabeled, the function `readFleetData` attaches a predefined label to the corresponding data. Configure the ensemble data variables to match the labels defined in `readFleetData`.

```
fensemble.DataVariables = ["Odometer";"FuelConsump";"MaxRPM";"MaxTemp";"EngineLight"];
```

The function `readFleetData` also parses the file name to return an ID of the car from which the data was collected, a number from 1 through 10. This ID is the ensemble independent variable.

```
fensemble.IndependentVariables = ["ID"];
```

Specify all data variables and the independent variable as selected variables for reading from the ensemble datastore.

```
fensemble.SelectedVariables = [fensemble.IndependentVariables;fensemble.DataVariables]
fensemble
```

```
fensemble = 
  fileEnsembleDatastore with properties:

                 ReadFcn: @readFleetData
        WriteToMemberFcn: []
           DataVariables: [5x1 string]
    IndependentVariables: "ID"
      ConditionVariables: [0x0 string]
       SelectedVariables: [6x1 string]
                ReadSize: 1
              NumMembers: 10
          LastMemberRead: [0x0 string]
                   Files: [10x1 string]
```

Read Ensemble Data

When you call `read` on the ensemble datastore, it uses `readFleetData` to read the selected variables from the first ensemble member.

```
data1 = read(fensemble)
```

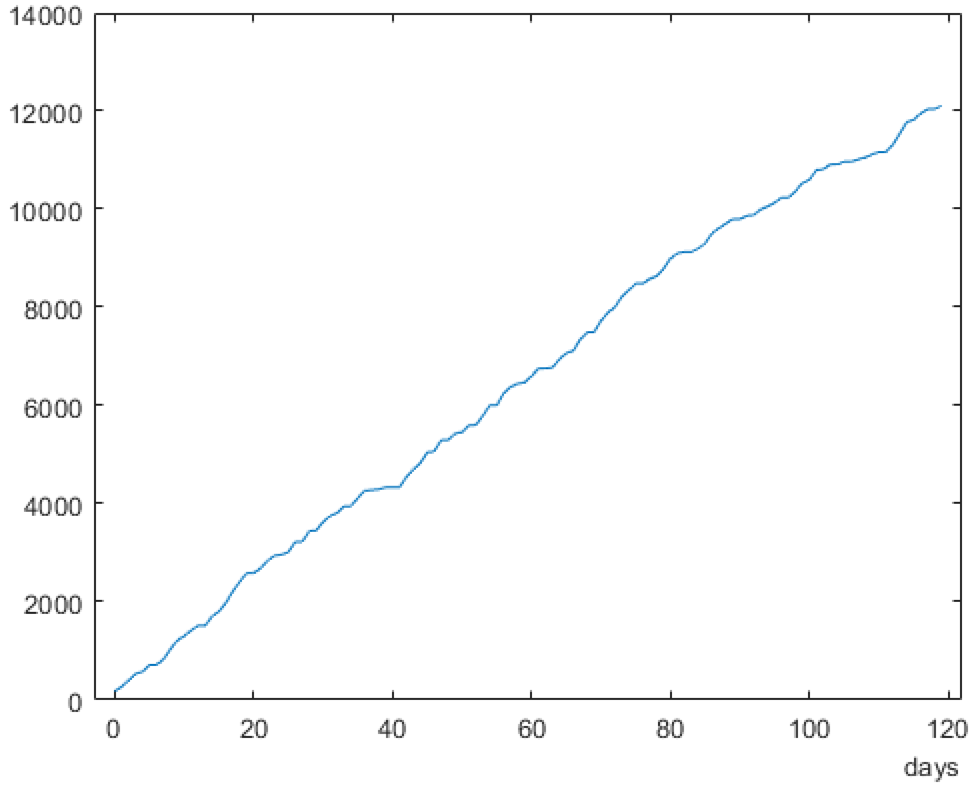
```
data1=1×6 table
    ID        Odometer            FuelConsump             MaxRPM                MaxTemp
    __    _________________    _________________    _________________    ______________
    1     [120x1 timetable]    [120x1 timetable]    [120x1 timetable]    [120x1 timetab
```

Examine and plot the odometer data.

```
odo1 = data1.Odometer{1}
```

```
odo1=120×1 timetable
     Time        Var1 
    ________    ______

    0 days      180.04
    1 day       266.76
    2 days      396.01
    3 days      535.19
    4 days      574.31
    5 days      714.82
    6 days      714.82
    7 days      821.44
    8 days      1030.5
    9 days      1213.4
    10 days     1303.4
    11 days     1416.9
    12 days     1513.5
    13 days     1513.5
    14 days     1697.1
    15 days     1804.6
      ⋮
```

```
plot(odo1.Time,odo1.Var1)
```



Compute the average gas mileage for this member of the fleet. This value is the odometer reading on the last day, divided by the total fuel consumed.

```
fuelConsump1 = data1.FuelConsump{1}.Var1;
totalConsump1 = sum(fuelConsump1);
totalMiles1 = odo1.Var1(end);
mpg1 = totalMiles1/totalConsump1
```

```
mpg1 = 22.3086
```

Batch-Process Data from All Ensemble Members

If you call `read` again, it reads data from the next ensemble member and advances the `LastMemberRead` property of `fensemble` to reflect the file name of that ensemble. You can repeat the processing steps to compute the average gas mileage for that member. In practice, it is more useful to automate the process of reading and processing the data. To do so, reset the ensemble datastore to a state in which no data has been read. Then loop through the ensemble and perform the read and process steps for each member, returning a table that contains each car's ID and average gas mileage. (If you have Parallel Computing Toolbox™, you can use it to speed up the processing of larger data ensembles.)

```
reset(fensemble)
mpgData = [];
while hasdata(fensemble)
    data = read(fensemble);
    odo = data.Odometer{1}.Var1;
    fuelConsump = data.FuelConsump{1}.Var1;
    totalConsump = sum(fuelConsump);
    mpg = odo(end)/totalConsump1;
    ID = data.ID;
    mpgData = [mpgData;ID,mpg];
end
mpgTable = array2table(mpgData,'VariableNames',{'ID','mpg'})
```

```
mpgTable=10×2 table
    ID     mpg  
    __    ______

     1    22.309
     2    19.327
     3    20.816
     4    27.464
     5    18.848
     6    22.517
     7    27.018
     8    27.284
     9    17.149
    10     26.37
```

See Also

fileEnsembleDatastore | read

More About

- “File Ensemble Datastore With Measured Data” on page 1-23
- “Data Ensembles for Condition Monitoring and Predictive Maintenance” on page 1-2

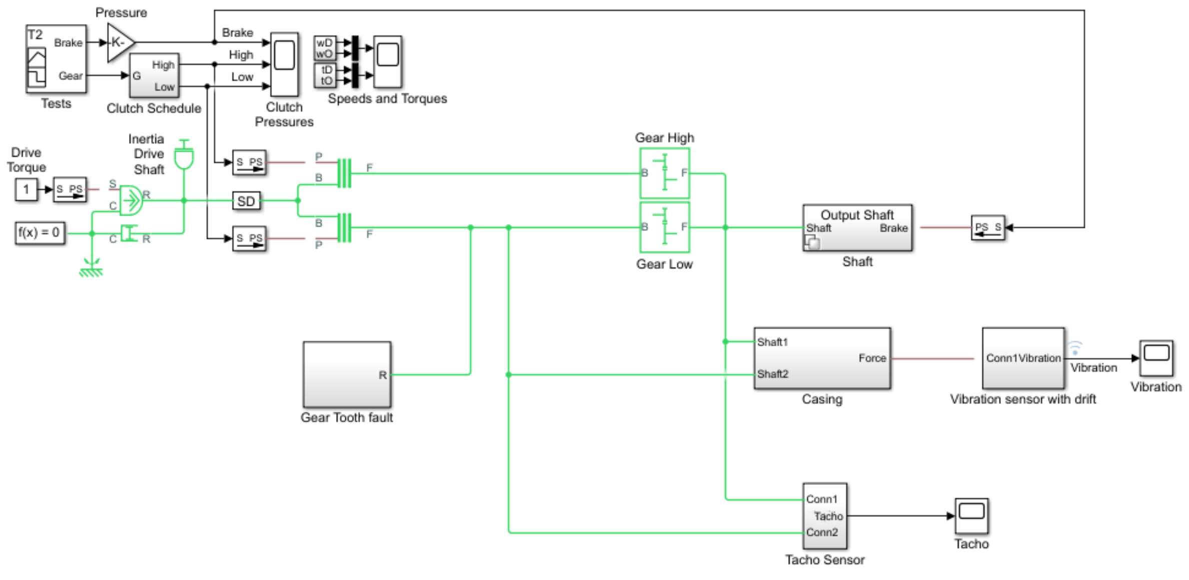
Using Simulink to Generate Fault Data

This example shows how to use a Simulink model to generate fault and healthy data. The fault and healthy data is used to develop a condition monitoring algorithm. The example uses a transmission system and models a gear tooth fault, a sensor drift fault and shaft wear fault.

Transmission System Model

The transmission casing model uses Simscape Driveline blocks to model a simple transmission system. The transmission system consists of a torque drive, drive shaft, clutch, and high and low gears connected to an output shaft.

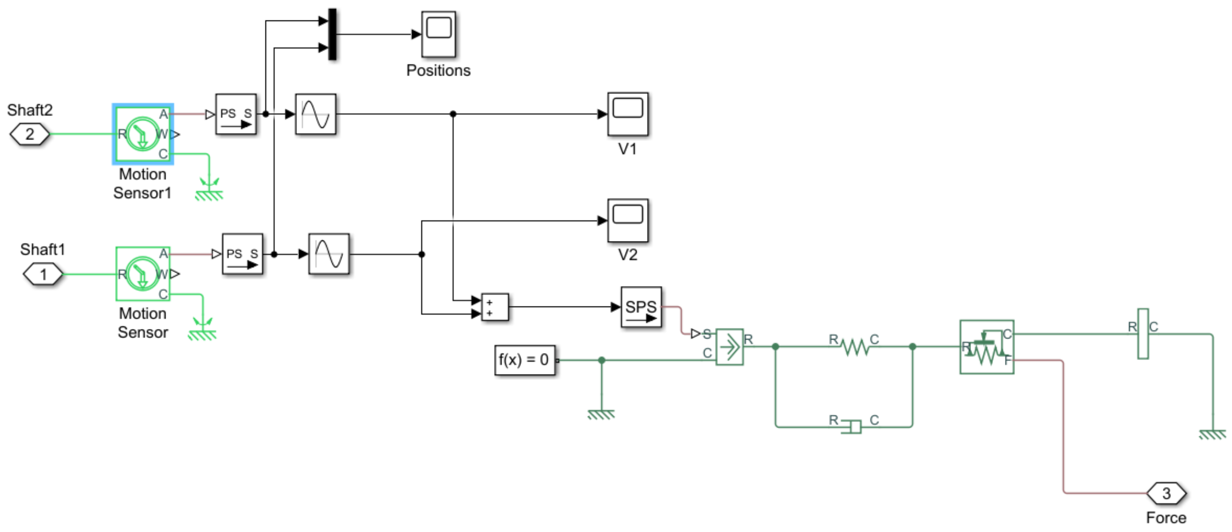
```
mdl = 'pdmTransmissionCasing';
open_system(mdl)
```



Copyright 2017 The MathWorks, Inc.

The transmission system includes a vibration sensor that is monitoring casing vibrations. The casing model translates the shaft angular displacement to a linear displacement on the casing. The casing is modelled as a mass spring damper system and the vibration (casing acceleration) is measured from the casing.

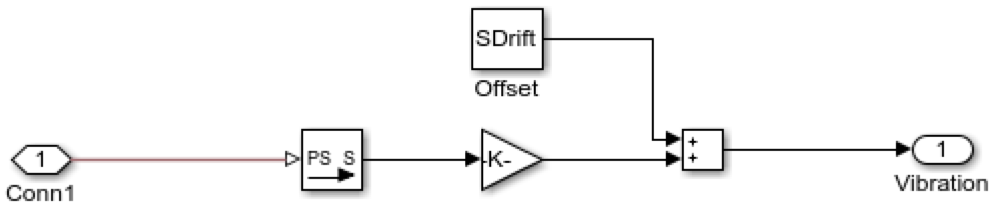
```
open_system([mdl '/Casing'])
```



Fault Modelling

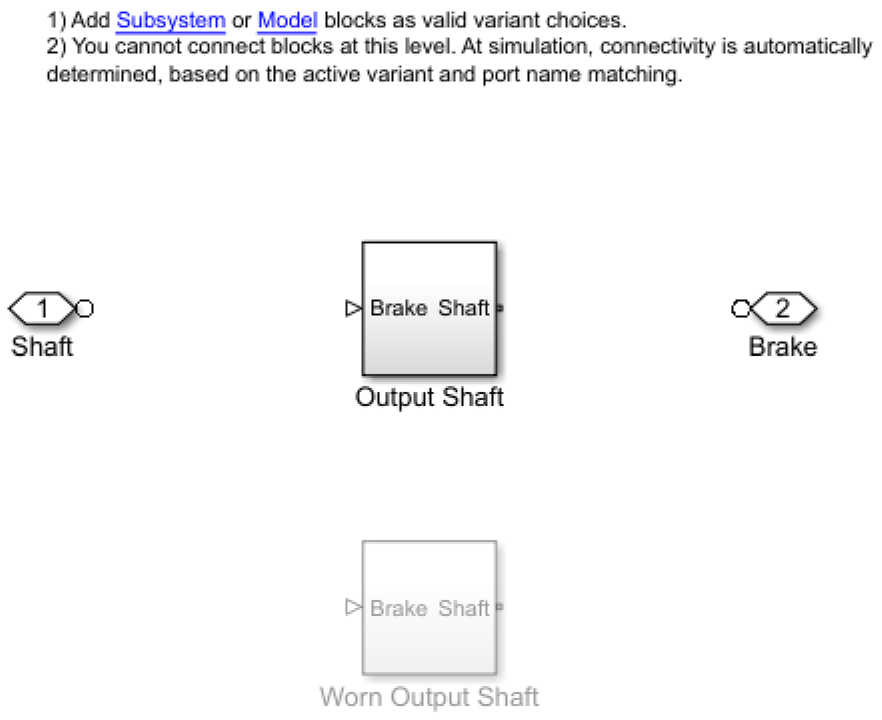
The transmission system includes fault models for vibration sensor drift, gear tooth fault, and shaft wear. The sensor drift fault is easily modeled by introducing an offset in the sensor model. The offset is controlled by the model variable `SDrift`, note that `SDrift=0` implies no sensor fault.

```
open_system([mdl '/Vibration sensor with drift'])
```



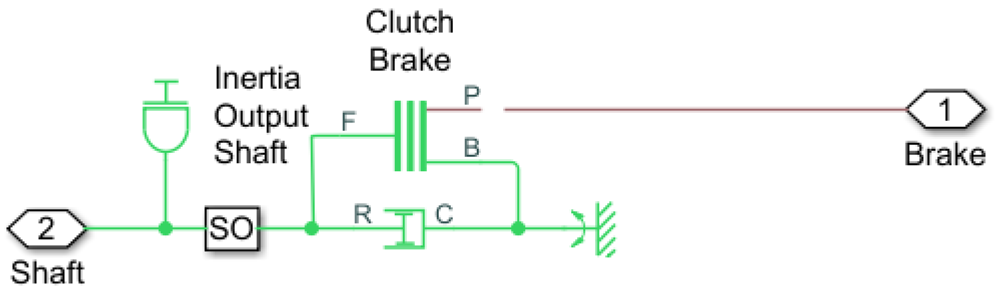
The shaft wear fault is modeled by a variant subsystem. In this case the subsystem variants change the shaft damping but the variant subsystem could be used to completely change the shaft model implementation. The selected variant is controlled by the model variable `ShaftWear`, note that `ShaftWear=0` implies no shaft fault.

```
open_system([mdl '/Shaft'])
```

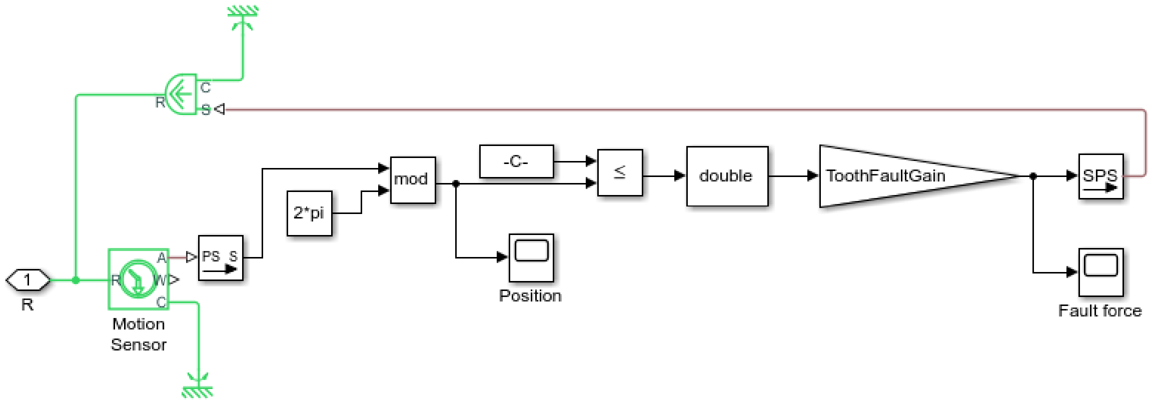


1) Add Subsystem or Model blocks as valid variant choices.
2) You cannot connect blocks at this level. At simulation, connectivity is automatically determined, based on the active variant and port name matching.

```
open_system([mdl,'/Shaft/Output Shaft'])
```



The gear tooth fault is modelled by injecting a disturbance torque at a fixed position in the rotation of the drive shaft. The shaft position is measured in radians and when the shaft position is within a small window around 0 a disturbance force is applied to the shaft. The magnitude of the disturbance is controlled by the model variable `ToothFaultGain`, note that `ToothFaultGain=0` implies no gear tooth fault.



Simulating Fault and Healthy Data

The transmission model is configured with variables that control the presence and severity of the three different fault types, sensor drift, gear tooth, and shaft wear. By varying the model variables, `SDrift`, `ToothFaultGain`, and `ShaftWear`, you can create vibration data for the different fault types. Use an array of `Simulink.SimulationInput` objects to define a number of different simulation scenarios. For example, choose an array of values for each of the model variables and then use the `ndgrid` function to create a `Simulink.SimulationInput` for each combination of model variable values.

```
toothFaultArray = -2:2/10:0; % Tooth fault gain values
sensorDriftArray = -1:0.5:1; % Sensor drift offset values
shaftWearArray = [0 -1];       % Variants available for drive shaft conditions

% Create an n-dimensional array with combinations of all values
[toothFaultValues,sensorDriftValues,shaftWearValues] = ...
    ndgrid(toothFaultArray,sensorDriftArray,shaftWearArray);

for ct = numel(toothFaultValues):-1:1
    % Create a Simulink.SimulationInput for each combination of values
    siminput = Simulink.SimulationInput(mdl);

    % Modify model parameters
    siminput = setVariable(siminput,'ToothFaultGain',toothFaultValues(ct));
    siminput = setVariable(siminput,'SDrift',sensorDriftValues(ct));
    siminput = setVariable(siminput,'ShaftWear',shaftWearValues(ct));

    % Collect the simulation input in an array
    gridSimulationInput(ct) = siminput;
end
```

Similarly create combinations of random values for each model variable. Make sure to include the `0` value so that there are combinations where only subsets of the three fault types are represented.

```
rng('default'); % Reset random seed for reproducibility
toothFaultArray = [0 -rand(1,6)];    % Tooth fault gain values
sensorDriftArray = [0 randn(1,6)/8]; % Sensor drift offset values
shaftWearArray = [0 -1];               % Variants available for drive shaft conditions

%Create an n-dimensional array with combinations of all values
[toothFaultValues,sensorDriftValues,shaftWearValues] = ...
    ndgrid(toothFaultArray,sensorDriftArray,shaftWearArray);

for ct=numel(toothFaultValues):-1:1
    % Create a Simulink.SimulationInput for each combination of values
    siminput = Simulink.SimulationInput(mdl);

    % Modify model parameters
    siminput = setVariable(siminput,'ToothFaultGain',toothFaultValues(ct));
    siminput = setVariable(siminput,'SDrift',sensorDriftValues(ct));
    siminput = setVariable(siminput,'ShaftWear',shaftWearValues(ct));
    % Collect the simulation input in an array
    randomSimulationInput(ct) = siminput;
end
```

With the array of `Simulink.SimulationInput` objects defined use the `generateSimulationEnsemble` function to run the simulations. The `generateSimulationEnsemble` function configures the model to save logged data to file, use the timetable format for signal logging and store the `Simulink.SimulationInput` objects in the saved file. The `generateSimulationEnsemble` function returns a status flag indicating whether the simulation completed successfully.

The code above created 110 simulation inputs from the gridded variable values and 98 simulation inputs from the random variable values giving 208 total simulations. Running these 208 simulations in parallel takes around ten minutes on a standard desktop and generates around 10GB of data, an option to only run the first 10 simulations is provided for convenience.

```
% Run the simulations and create an ensemble to manage the simulation results
if ~exist(fullfile(pwd,'Data'),'dir')
    mkdir(fullfile(pwd,'Data')) % Create directory to store results
end
runAll = true;
if runAll
   [ok,e] = generateSimulationEnsemble([gridSimulationInput, randomSimulationInput], .
        fullfile(pwd,'Data'),'UseParallel', true);
else
    [ok,e] = generateSimulationEnsemble(gridSimulationInput(1:10), fullfile(pwd,'Data')
end
```

```
[28-Feb-2018 13:17:31] Checking for availability of parallel pool...
[28-Feb-2018 13:17:37] Loading Simulink on parallel workers...
Analyzing and transferring files to the workers ...done.
[28-Feb-2018 13:17:56] Configuring simulation cache folder on parallel workers...
[28-Feb-2018 13:17:57] Running SetupFcn on parallel workers...
[28-Feb-2018 13:18:01] Loading model on parallel workers...
[28-Feb-2018 13:18:02] Transferring base workspace variables used in the model to paral
[28-Feb-2018 13:18:07] Running simulations...
[28-Feb-2018 13:18:29] Completed 1 of 208 simulation runs
[28-Feb-2018 13:18:29] Completed 2 of 208 simulation runs
[28-Feb-2018 13:18:30] Completed 3 of 208 simulation runs
...
[28-Feb-2018 13:24:22] Completed 204 of 208 simulation runs
```

```
[28-Feb-2018 13:24:28] Completed 205 of 208 simulation runs
[28-Feb-2018 13:24:28] Completed 206 of 208 simulation runs
[28-Feb-2018 13:24:28] Completed 207 of 208 simulation runs
[28-Feb-2018 13:24:29] Completed 208 of 208 simulation runs
[28-Feb-2018 13:24:33] Cleaning up parallel workers...
```

The `generateSimulationEnsemble` ran and logged the simulation results. Create a simulation ensemble to process and analyze the simulation results using the `simulationEnsembleDatastore` command.

```
ens = simulationEnsembleDatastore(fullfile(pwd,'Data'));
```

Processing the Simulation Results

The `simulationEnsembleDatastore` command created an ensemble object that points to the simulation results. Use the ensemble object to prepare and analyze the data in each member of the ensemble. The ensemble object lists the data variables in the ensemble and by default all the variables are selected for reading.

```
ens
```

```
ens = 
  simulationEnsembleDatastore with properties:

           DataVariables: [6×1 string]
    IndependentVariables: [0×0 string]
      ConditionVariables: [0×0 string]
       SelectedVariables: [6×1 string]
              NumMembers: 208
          LastMemberRead: [0×0 string]
```

```
ens.SelectedVariables
```

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

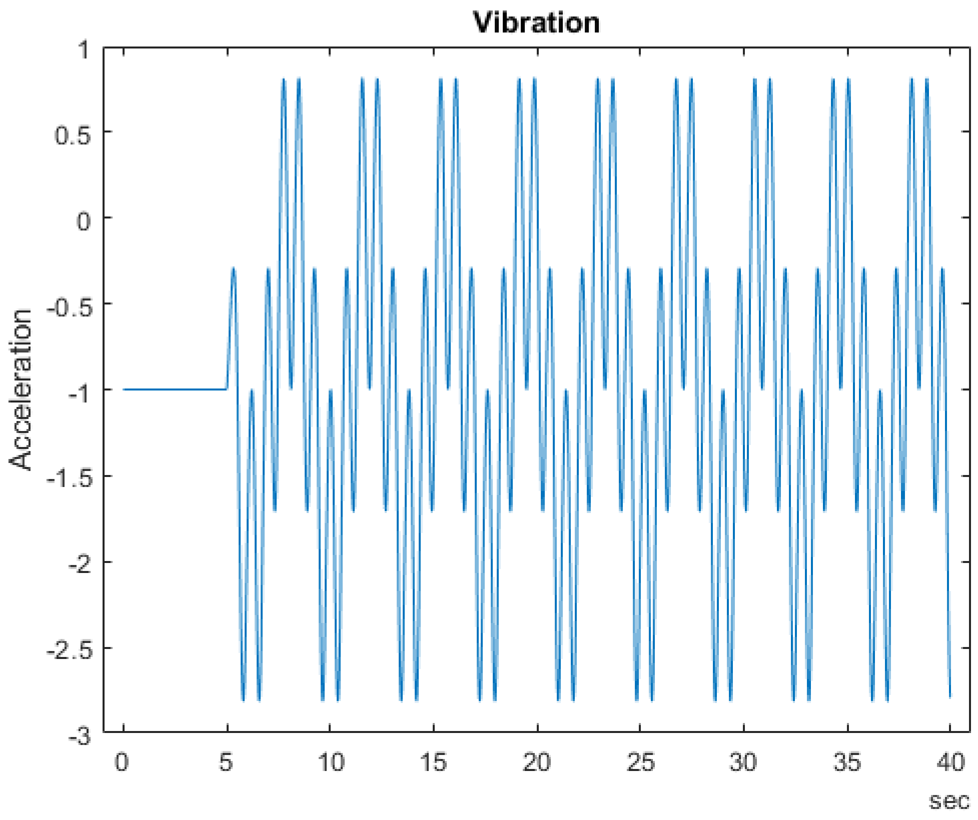
For analysis only read the `Vibration` and `Tacho` signals and the `Simulink.SimulationInput`. The `Simulink.SimulationInput` has the model variable values used for simulation and is used to create fault labels for the ensemble members. Use the ensemble `read` command to get the ensemble member data.

```
ens.SelectedVariables = ["Vibration" "Tacho" "SimulationInput"];
data = read(ens)
```

```
data=1×3 table
         Vibration                  Tacho                     SimulationInput
    ___________________    ___________________    ______________________________

    [40272×1 timetable]    [40272×1 timetable]    [1×1 Simulink.SimulationInput]
```

Extract the vibration signal from the returned data and plot it.

```
vibration = data.Vibration{1};
plot(vibration.Time,vibration.Data)
title('Vibration')
ylabel('Acceleration')
```



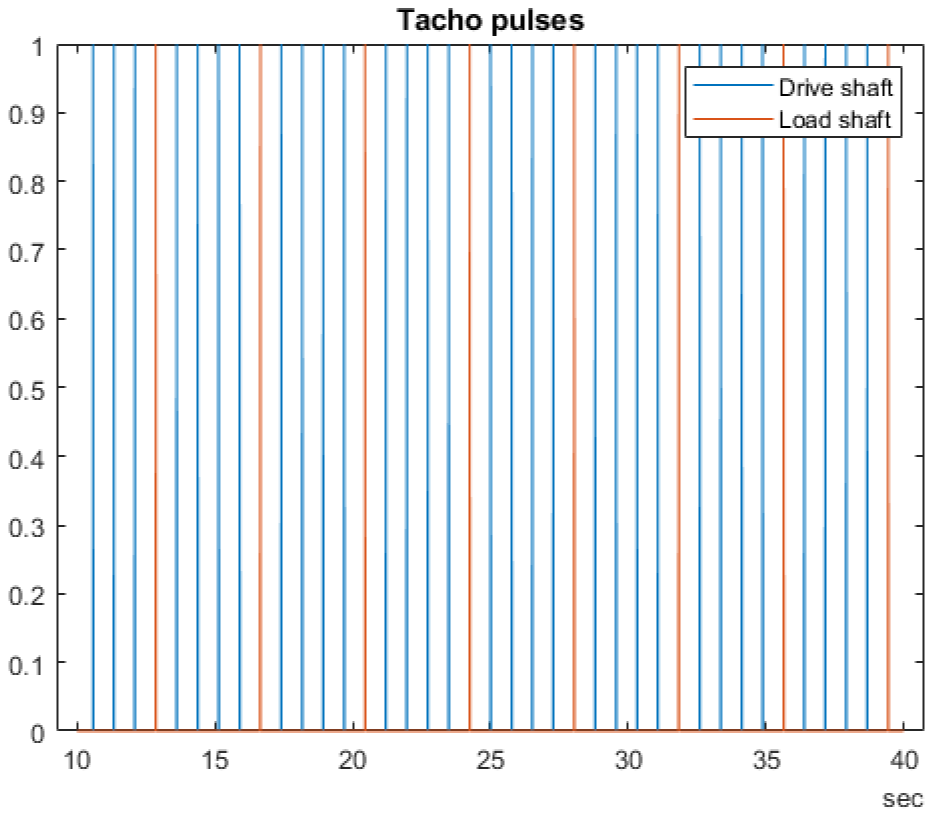
The first 10 seconds of the simulation contains data where the transmission system is starting up; for analysis discard this data.

```
idx = vibration.Time >= seconds(10);
vibration = vibration(idx,:);
vibration.Time = vibration.Time - vibration.Time(1);
```

The `Tacho` signal contains pulses for the rotations of the drive and load shafts but analysis, and specifically time synchronous averaging, requires the times of shaft rotations. The following code discards the first 10 seconds of the `Tacho` data and finds the shaft rotation times in `tachoPulses`.

```
tacho = data.Tacho{1};
idx = tacho.Time >= seconds(10);
```

```
tacho = tacho(idx,:);
plot(tacho.Time,tacho.Data)
title('Tacho pulses')
legend('Drive shaft','Load shaft') % Load shaft rotates more slowly than drive shaft
```



```
idx = diff(tacho.Data(:,2)) > 0.5;
tachoPulses = tacho.Time(find(idx)+1)-tacho.Time(1)

tachoPulses = 8×1 duration array
   2.8543 sec
   6.6508 sec
   10.447 sec
   14.244 sec
    18.04 sec
   21.837 sec
   25.634 sec
    29.43 sec
```

The `Simulink.SimulationInput.Variables` property contains the values of the fault parameters used for the simulation, these values allow us to create fault labels for each ensemble member.

```
vars = data.SimulationInput{1}.Variables;
idx = strcmp({vars.Name},'SDrift');
if any(idx)
    sF = abs(vars(idx).Value) > 0.01; % Small drift values are not faults
else
    sF = false;
end
idx = strcmp({vars.Name},'ShaftWear');
if any(idx)
    sV = vars(idx).Value < 0;
else
    sV = false;
end
if any(idx)
    idx = strcmp({vars.Name},'ToothFaultGain');
    sT = abs(vars(idx).Value) < 0.1; % Small tooth fault values are not faults
else
    sT = false
end
faultCode = sF + 2*sV + 4*sT; % A fault code to capture different fault conditions
```

The processed vibration and tacho signals and the fault labels are added to the ensemble to be used later.

```
sdata = table({vibration},{tachoPulses},sF,sV,sT,faultCode, ...
    'VariableNames',{'Vibration','TachoPulses','SensorDrift','ShaftWear','ToothFault',
```

```
sdata=1×6 table
         Vibration              TachoPulses       SensorDrift    ShaftWear    ToothFault
    ___________________       ______________      ___________    _________    __________

    [30106×1 timetable]       [8×1 duration]         true          false        false
```

```
ens.DataVariables = [ens.DataVariables; "TachoPulses"];
```

The ensemble `ConditionVariables` property can be used to identify the variables in the ensemble that contain condition or fault label data. Set the property to contain the newly created fault labels.

```
ens.ConditionVariables = ["SensorDrift","ShaftWear","ToothFault","FaultCode"];
```

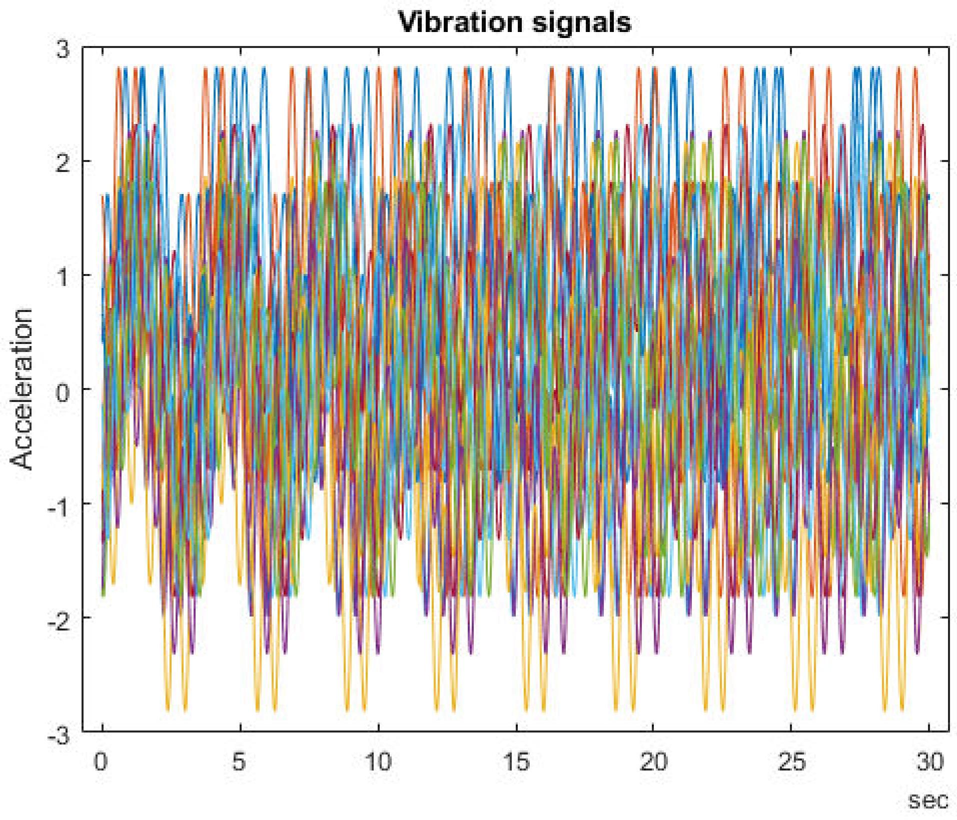
The code above was used to process one member of the ensemble. To process all the ensemble members the code above was converted to the function `prepareData` and using the ensemble `hasdata` command a loop is used to apply `prepareData` to all the ensemble members. The ensemble members can be processed in parallel by partitioning the ensemble and processing the ensemble partitions in parallel.

```
reset(ens)
runLocal = false;
if runLocal
    % Process each member in the ensemble
    while hasdata(ens)
        data = read(ens);
        addData = prepareData(data);
        writeToLastMemberRead(ens,addData)
    end
else
    % Split the ensemble into partitions and process each partition in parallel
    n = numpartitions(ens,gcp);
    parfor ct = 1:n
        subens = partition(ens,n,ct);
        while hasdata(subens)
            data = read(subens);
            addData = prepareData(data);
            writeToLastMemberRead(subens,addData)
        end
    end
end
```

Plot the vibration signal of every 10th member of the ensemble using the `hasdata` and `read` commands to extract the vibration signal.

```
reset(ens)
ens.SelectedVariables = "Vibration";
figure,
ct = 1;
while hasdata(ens)
    data = read(ens);
    if mod(ct,10) == 0
```

```
            vibration = data.Vibration{1};
            plot(vibration.Time,vibration.Data)
            hold on
        end
        ct = ct + 1;
    end
    hold off
    title('Vibration signals')
    ylabel('Acceleration')
```



Analyzing the Simulation Data

Now that the data has been cleaned and pre-processed the data is ready for extracting features to determine the features to use to classify the different faults types. First configure the ensemble so that it only returns the processed data.

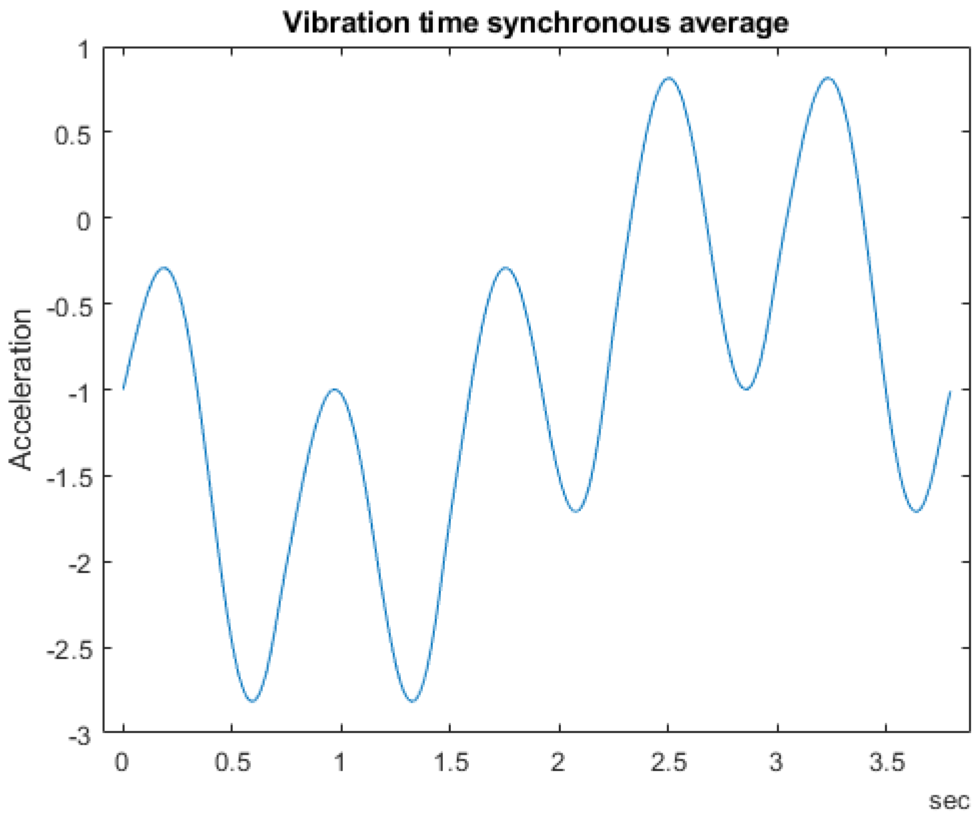
```
ens.SelectedVariables = ["Vibration","TachoPulses"];
```

For each member in the ensemble compute a number of time and spectrum based features. These include signal statistics such as signal mean, variance, peak to peak, non-linear signal characteristics such as approximate entropy and Lyapunov exponent, and spectral features such as the peak frequency of the time synchronous average of the vibration signal, and the power of the time synchronous average envelope signal. The `analyzeData` function contains a full list of the extracted features. By way of example consider computing the spectrum of the time synchronous averaged vibration signal.

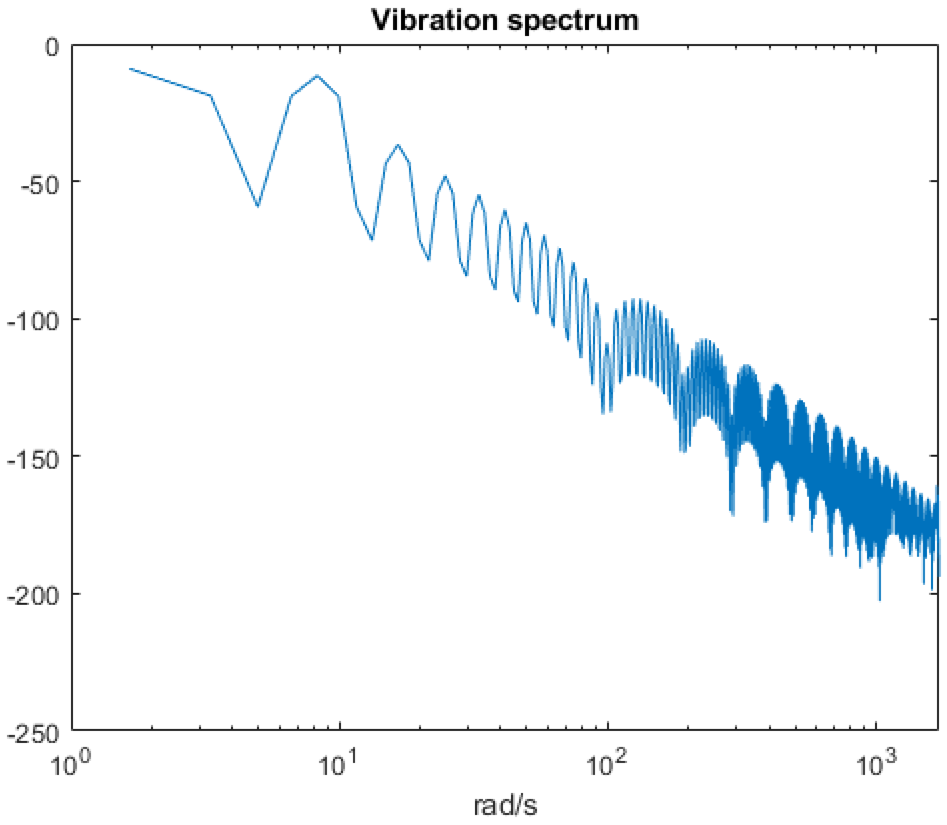
```
reset(ens)
data = read(ens)
```

```
data=1×2 table
         Vibration             TachoPulses  
    ___________________      ______________

    [30106×1 timetable]      [8×1 duration]

```

```
vibration = data.Vibration{1};

% Interpolate the Vibration signal onto periodic time base suitable for fft analysis
np = 2^floor(log(height(vibration))/log(2));
dt = vibration.Time(end)/(np-1);
tv = 0:dt:vibration.Time(end);
y = retime(vibration,tv,'linear');

% Compute the time synchronous average of the vibration signal
tp = seconds(data.TachoPulses{1});
vibrationTSA = tsa(y,tp);
figure
plot(vibrationTSA.ttTime,vibrationTSA.tsa)
title('Vibration time synchronous average')
ylabel('Acceleration')
```



```
% Compute the spectrum of the time synchronous average
np = numel(vibrationTSA);
f = fft(vibrationTSA.tsa.*hamming(np))/np;
frTSA = f(1:floor(np/2)+1);            % TSA frequency response
wTSA = (0:np/2)/np*(2*pi/seconds(dt)); % TSA spectrum frequencies
mTSA = abs(frTSA);                     % TSA spectrum magnitudes
figure
semilogx(wTSA,20*log10(mTSA))
title('Vibration spectrum')
xlabel('rad/s')
```



The frequency that corresponds to the peak magnitude could turn out to be a feature that is useful for classifying the different fault types. The code below computes the features mentioned above for all the ensemble members (running this analysis make take around thirty minutes on a standard desktop, optional code to run the analysis in parallel using the ensemble `partition` command is provided). The names of the features are added to the ensemble data variables property before calling writeToLastMemberRead to add the computed features to each ensemble member.

```
reset(ens)
ens.DataVariables = [ens.DataVariables; ...
    "SigMean";"SigMedian";"SigRMS";"SigVar";"SigPeak";"SigPeak2Peak";"SigSkewness"; ..
    "SigKurtosis";"SigCrestFactor";"SigMAD";"SigRangeCumSum";"SigCorrDimension";"SigApp
    "SigLyapExponent";"PeakFreq";"HighFreqPower";"EnvPower";"PeakSpecKurtosis"];
```

```
if runLocal
    while hasdata(ens)
        data = read(ens);
        addData = analyzeData(data);
        writeToLastMemberRead(ens,addData);
    end
else
    % Split the ensemble into partitions and analyze each partition in parallel
    n = numpartitions(ens,gcp);
    parfor ct = 1:n
        subens = partition(ens,n,ct);
        while hasdata(subens)
            data = read(subens);
            addData = analyzeData(data);
            writeToLastMemberRead(subens,addData)
        end
    end
end
```

Selecting Features for Fault Classification

The features computed above are used to build a classifier to classify the different fault conditions. First configure the ensemble to read only the derived features and the fault labels.

```
featureVariables = analyzeData('GetFeatureNames');
ens.DataVariables = [ens.DataVariables; featureVariables];
ens.SelectedVariables = [featureVariables; ens.ConditionVariables];
reset(ens)
```

Gather the feature data for all the ensemble members into one table.

```
featureData = gather(tall(ens))
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1.8 min
Evaluation completed in 1.8167 min

featureData=208×22 table
    SigMean     SigMedian     SigRMS     SigVar      SigPeak     SigPeak2Peak     SigSkewne
    ________    __________    _______    _______     _______     ____________     _________

    -0.94876      -0.9722      1.3726    0.98387     0.81571        3.6314         -0.0415
    -0.97537     -0.98958      1.3937    0.99105     0.81571        3.6314         -0.0237
      1.0502       1.0267      1.4449    0.98491      2.8157        3.6314          -0.041
      1.0227       1.0045      1.4288    0.99553      2.8157       3.6314        -0.01635
      1.0123       1.0024      1.4202    0.99233      2.8157       3.6314        -0.01470
      1.0275       1.0102      1.4338     1.0001      2.8157       3.6314         -0.0265
      1.0464       1.0275      1.4477     1.0011      2.8157       3.6314        -0.04284
      1.0459       1.0257      1.4402    0.98047      2.8157       3.6314        -0.03540
      1.0263       1.0068      1.4271    0.98341      2.8157       3.6314          -0.016
      1.0103       1.0014      1.4183    0.99091      2.8157       3.6314        -0.01160
      1.0129       1.0023       1.419    0.98764      2.8157       3.6314        -0.01096
      1.0251       1.0104      1.4291     0.9917      2.8157       3.6314        -0.02360
    -0.97301     -0.99243      1.3928    0.99326     0.81571       3.6314        -0.01256
      1.0277       1.0084      1.4315    0.99314      2.8157       3.6314        -0.01354
    0.026994    0.0075709     0.99697    0.99326      1.8157       3.6314        -0.01256
    0.026943    0.0084639     0.99297    0.98531      1.8157       3.6314        -0.01818
        ⋮
```

Consider the sensor drift fault. Use the `fscnca` command with all the features computed above as predictors and the sensor drift fault label (a true false value) as the response. The `fscnca` command returns weights for each feature and features with higher weights have higher importance in predicting the response. For the sensor drift fault the weights indicate that two features are significant predictors (the signal cumulative sum range and the peak frequency of the spectral kurtosis) and the remaining features have little impact.

```
idxResponse = strcmp(featureData.Properties.VariableNames,'SensorDrift');
idxLastFeature = find(idxResponse)-1; % Index of last feature to use as a potential pre
featureAnalysis = fscnca(featureData{:,1:idxLastFeature},featureData.SensorDrift);
featureAnalysis.FeatureWeights
```

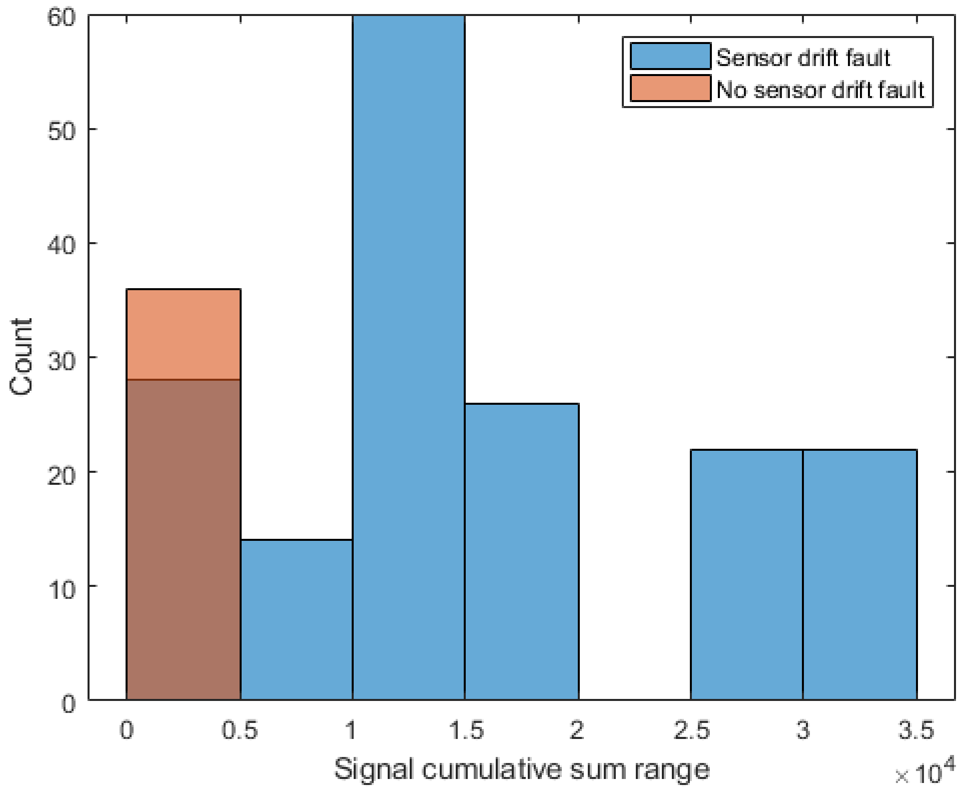
```
ans = 18×1

    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
        ⋮
```

```
idxSelectedFeature = featureAnalysis.FeatureWeights > 0.1;
classifySD = [featureData(:,idxSelectedFeature), featureData(:,idxResponse)]
```

```
classifySD=208×3 table
    SigRangeCumSum    PeakSpecKurtosis    SensorDrift
    ______________    ________________    ___________

         28562             162.13             true   
         29418             226.12             true   
         31710             162.13             true   
         30984             162.13             true   
         30661             230.39             true   
         31102             230.39             true   
         31665             230.39             true   
         31554             230.39             true   
         30951             162.13             true   
         30465             230.39             true   
         30523             230.39             true   
         30896             230.39             true   
         29351             230.39             true   
         30963             226.12             true   
        1083.8             230.39             false  
        1466.6             230.39             false  
      ⋮
```

A grouped histogram of the cumulative sum range gives us insight into why this feature is a significant predictor for the sensor drift fault.

```
figure
histogram(classifySD.SigRangeCumSum(classifySD.SensorDrift),'BinWidth',5e3)
xlabel('Signal cumulative sum range')
ylabel('Count')
hold on
histogram(classifySD.SigRangeCumSum(~classifySD.SensorDrift),'BinWidth',5e3)
hold off
legend('Sensor drift fault','No sensor drift fault')
```



The histogram plot shows that the signal cumulative sum range is a good featured for detecting sensor drift faults though an additional feature is probably needed as there may be false positives when the signal cumulative sum range is below 0.5 if just the signal cumulative sum range is used to classify sensor drift.

Consider the shaft wear fault. In this case the `fscnca` function indicates that there are 3 features that are significant predictors for the fault (the signal Lyapunov exponent, peak frequency, and the peak frequency in the spectral kurtosis), choose these to classify the shaft wear fault.

```
idxResponse = strcmp(featureData.Properties.VariableNames,'ShaftWear');
featureAnalysis = fscnca(featureData{:,1:idxLastFeature},featureData.ShaftWear);
featureAnalysis.FeatureWeights
```

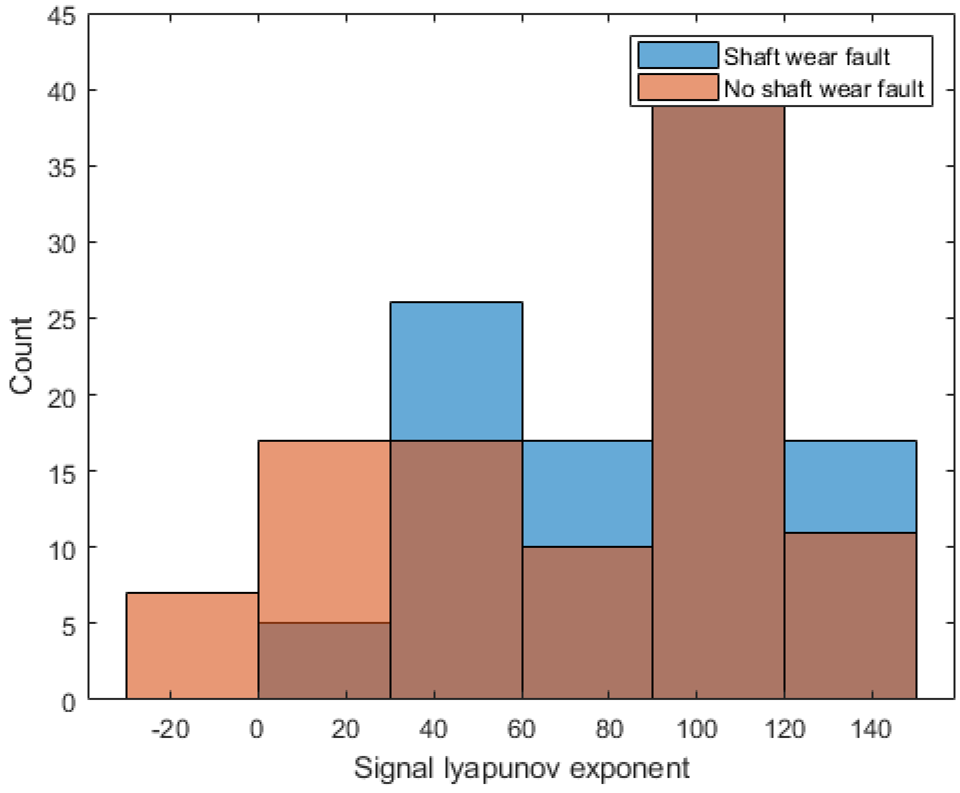
```
ans = 18×1

    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
    0.0000
      ⋮
```

```
idxSelectedFeature = featureAnalysis.FeatureWeights > 0.1;
classifySW = [featureData(:,idxSelectedFeature), featureData(:,idxResponse)]
```

```
classifySW=208×4 table
    SigLyapExponent    PeakFreq    PeakSpecKurtosis    ShaftWear
    _______________    ________    ________________    _________

        79.531              0           162.13           false  
        70.339              0           226.12           false  
        125.18              0           162.13           true   
        112.52              0           162.13           true   
        109.02              0           230.39           true   
        64.499              0           230.39           true   
        98.759              0           230.39           true   
        44.304              0           230.39           true   
        125.28              0           162.13           true   
        17.093              0           230.39           true   
        84.568              0           230.39           true   
        98.463              0           230.39           true   
        42.887              0           230.39           false  
        99.426              0           226.12           true   
        44.448          9.998           230.39           false  
         93.95         1.8618           230.39           false  
      ⋮
```

The grouped histogram for the signal Lyapunov exponent shows why that feature alone is not a good predictor.

```
figure
histogram(classifySW.SigLyapExponent(classifySW.ShaftWear))
xlabel('Signal lyapunov exponent')
ylabel('Count')
hold on
histogram(classifySW.SigLyapExponent(~classifySW.ShaftWear))
hold off
legend('Shaft wear fault','No shaft wear fault')
```



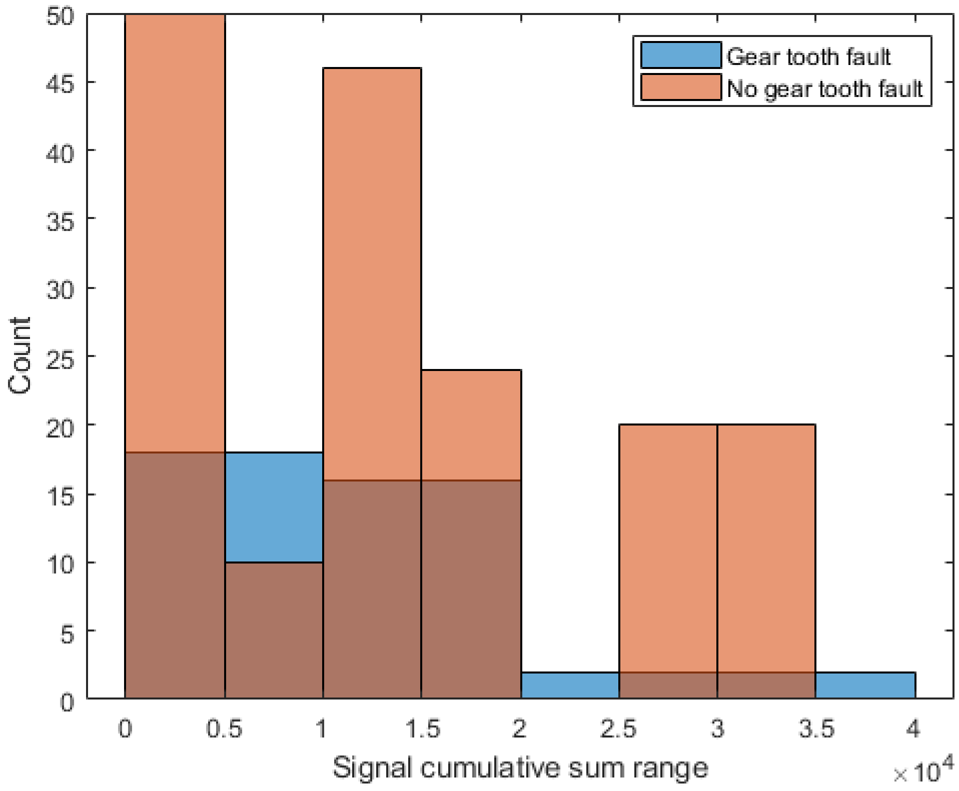
The shaft wear feature selection indicates multiple features are needed to classify the shaft wear fault, the grouped histogram confirms this as the most significant feature (in this case the Lyapunov exponent) has a similar distribution for both faulty and fault free scenarios indicating that more features are needed to correctly classify this fault.

Finally consider the tooth fault, the `fscnca` function indicates that there are 3 features primary that are significant predictors for the fault (the signal cumulative sum range, the signal Lyapunov exponent and the peak frequency in the spectral kurtosis). Choosing those 3 features to classify the tooth fault results in a classifier that has poor performance. Instead, use the 6 most important features.

```
idxResponse = strcmp(featureData.Properties.VariableNames,'ToothFault');
featureAnalysis = fscnca(featureData{:,1:idxLastFeature},featureData.ToothFault);
[~,idxSelectedFeature] = sort(featureAnalysis.FeatureWeights);
classifyTF = [featureData(:,idxSelectedFeature(end-5:end)), featureData(:,idxResponse)
```

```
classifyTF=208×7 table
    PeakFreq    SigPeak    SigCorrDimension    SigLyapExponent    PeakSpecKurtosis    
    ________    _______    ________________    _______________    ________________    _

          0     0.81571         1.1427             79.531              162.13
          0     0.81571         1.1362             70.339              226.12
          0      2.8157         1.1479             125.18              162.13
          0      2.8157         1.1472             112.52              162.13
          0      2.8157          1.147             109.02              230.39
          0      2.8157         1.0975             64.499              230.39
          0      2.8157         1.1417             98.759              230.39
          0      2.8157         1.1345             44.304              230.39
          0      2.8157         1.1515             125.28              162.13
          0      2.8157         1.0619             17.093              230.39
          0      2.8157         1.1371             84.568              230.39
          0      2.8157         1.1261             98.463              230.39
          0     0.81571         1.1277             42.887              230.39
          0      2.8157         1.1486             99.426              226.12
      9.998      1.8157         1.1277             44.448              230.39
     1.8618      1.8157         1.1368              93.95              230.39
      ⋮
```

```
figure
histogram(classifyTF.SigRangeCumSum(classifyTF.ToothFault))
xlabel('Signal cumulative sum range')
ylabel('Count')
hold on
histogram(classifyTF.SigRangeCumSum(~classifyTF.ToothFault))
hold off
legend('Gear tooth fault','No gear tooth fault')
```

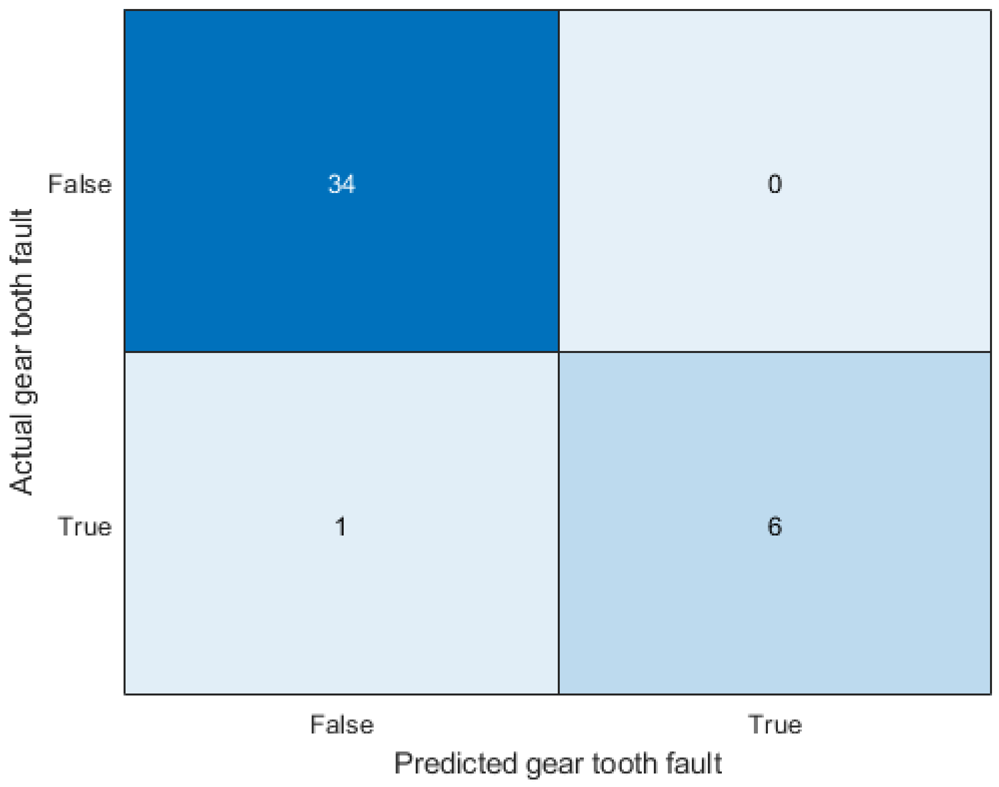


Using the above results a polynomial svm to classify gear tooth faults. Split the feature table into members that are used for training and members for testing and validation. Use the training members to create a svm classifier using the `fitcsvm` command.

```
rng('default') % For reproducibility
cvp = cvpartition(size(classifyTF,1),'KFold',5); % Randomly partition the data for tra
classifierTF = fitcsvm(...
    classifyTF(cvp.training(1),1:end-1), ...
    classifyTF(cvp.training(1),end), ...
    'KernelFunction','polynomial', ...
    'PolynomialOrder',2, ...
    'KernelScale','auto', ...
    'BoxConstraint',1, ...
    'Standardize',true, ...
    'ClassNames',[false; true]);
```

Use the classifier to classify the test points using the `predict` command and check the performance of the predictions using a confusion matrix.

```
% Use the classifier on the test validation data to evaluate performance
actualValue = classifyTF{cvp.test(1),end};
predictedValue = predict(classifierTF, classifyTF(cvp.test(1),1:end-1));

% Check performance by computing and plotting the confusion matrix
confdata = confusionmat(actualValue,predictedValue);
h = heatmap(confdata, ...
    'YLabel', 'Actual gear tooth fault', ...
    'YDisplayLabels', {'False','True'}, ...
    'XLabel', 'Predicted gear tooth fault', ...
    'XDisplayLabels', {'False','True'}, ...
    'ColorbarVisible','off');
```



The confusion matrix indicates that the classifier correctly classifies all non-fault conditions but incorrectly classifies one expected fault condition as not being a fault. Increasing the number of features used in the classifier can help improve the performance further.

Summary

This example walked through the workflow for generating fault data from Simulink, using a simulation ensemble to clean up the simulation data and extract features. The extracted features were then used to build classifiers for the different fault types.

See Also

`simulationEnsembleDatastore`

More About

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

Multi-Class Fault Detection Using Simulated Data

This example shows how to use a Simulink model to generate fault and healthy data. The data is used to develop a multi-class classifier to detect different combinations of faults. The example uses a triplex reciprocating pump model and includes leak, blocking, and bearing faults.

Setup the Model

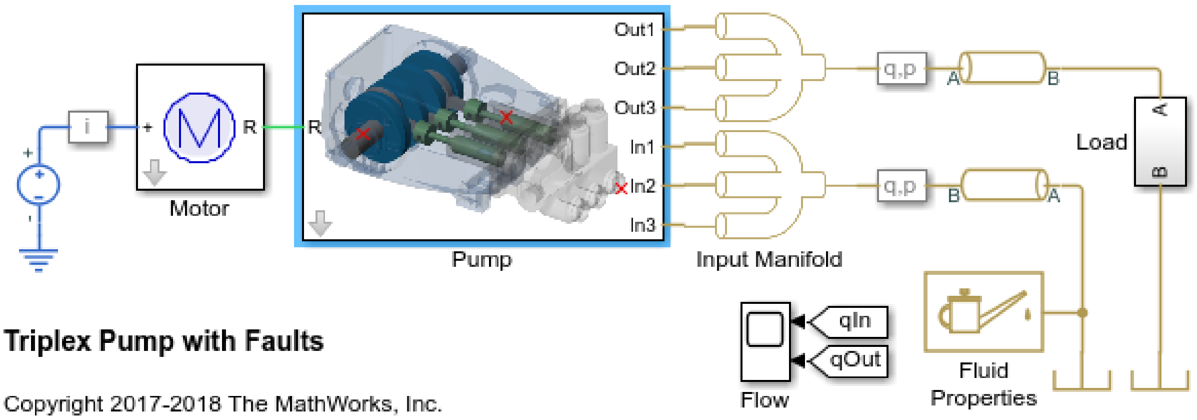
This example uses many supporting files that are stored in a zip file. Unzip the file to get access to the supporting files, load the model parameters, and create the reciprocating pump library.

```
if ~exist('+mech_hydro_forcesPS','dir')
    unzip('pdmRecipPump_supportingfiles.zip')
end

% Load Parameters
pdmRecipPump_Parameters %Pump
CAT_Pump_1051_DataFile_imported %CAD

% Create Simscape library if needed
if exist('mech_hydro_forcesPS_Lib','file')~=4
    ssc_build mech_hydro_forcesPS
end
```

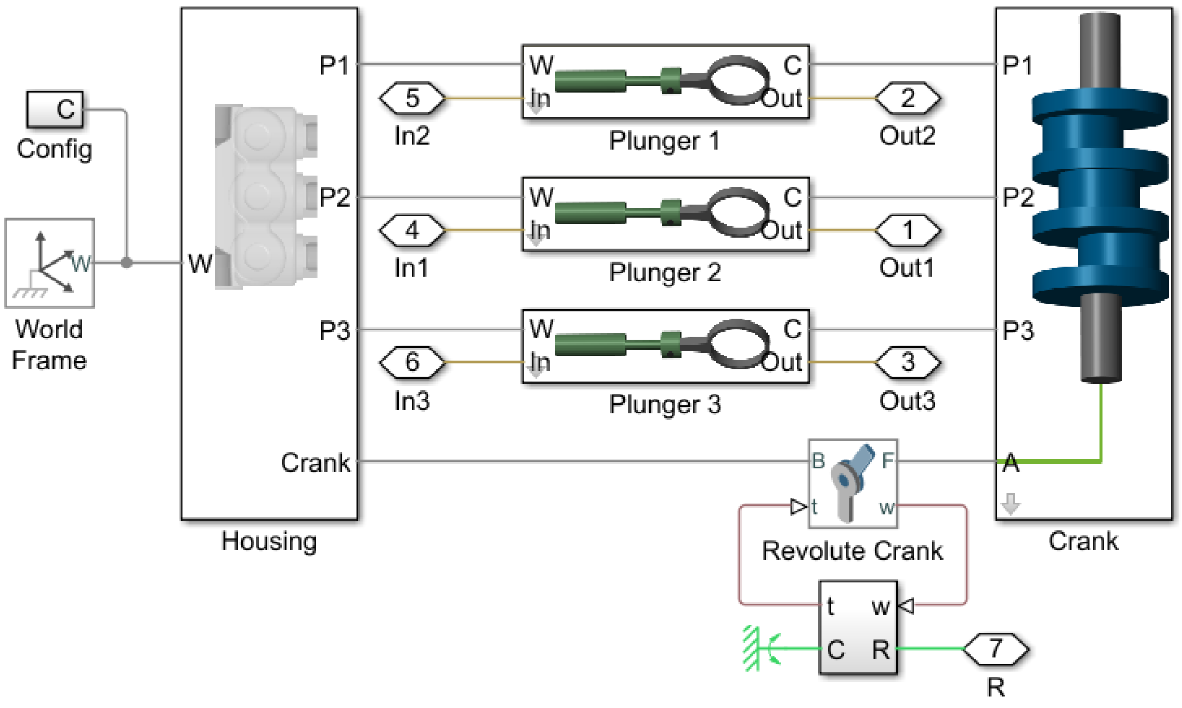
Reciprocating Pump Model

The reciprocating pump consists of an electric motor, the pump housing, pump crank and pump plungers.

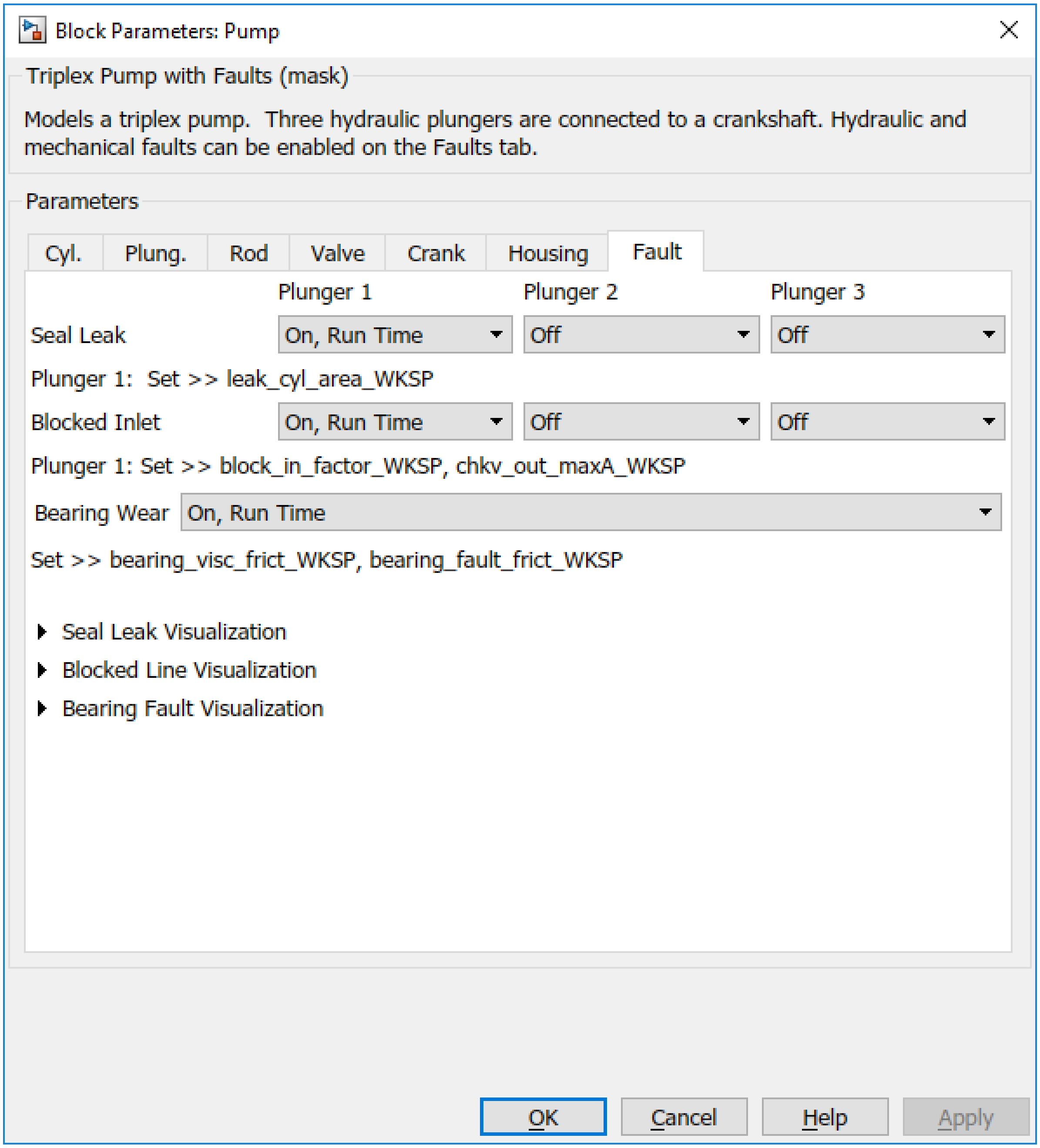
```
mdl = 'pdmRecipPump';
open_system(mdl)
```



```
open_system([mdl,'/Pump'])
```



The pump model is configured to model three types of faults; cylinder leaks, blocked inlet, and increased bearing friction. These faults are parameterized as workspace variables and configured through the pump block dialog.



Triplex Pump with Faults (mask)

Models a triplex pump. Three hydraulic plungers are connected to a crankshaft. Hydraulic and mechanical faults can be enabled on the Faults tab.

Parameters

	Plunger 1	Plunger 2	Plunger 3
Seal Leak	On, Run Time	Off	Off

Plunger 1: Set >> leak_cyl_area_WKSP

	Plunger 1	Plunger 2	Plunger 3
Blocked Inlet	On, Run Time	Off	Off

Plunger 1: Set >> block_in_factor_WKSP, chkv_out_maxA_WKSP

Bearing Wear: On, Run Time

Set >> bearing_visc_frict_WKSP, bearing_fault_frict_WKSP

Simulating Fault and Healthy Data

For each of the three fault types create an array of values that represent the fault severity, ranging from no fault to a significant fault.

```
% Define fault parameter variations
numParValues = 10;
leak_area_set_factor = linspace(0.00,0.036,numParValues);
leak_area_set = leak_area_set_factor*TRP_Par.Check_Valve.In.Max_Area;
leak_area_set = max(leak_area_set,1e-9); % Leakage area cannot be 0
blockinfactor_set = linspace(0.8,0.53,numParValues);
bearingfactor_set = linspace(0,6e-4,numParValues);
```

The pump model is configured to include noise, thus running the model with the same fault parameter values will result in different simulation outputs. This is useful for developing a classifier as it means there can be multiple simulation results for the same fault condition and severity. To configure simulations for such results, create vectors of fault parameter values where the values represent no faults, a single fault, combinations of two faults, and combinations of three faults. For each group (no fault, single fault, etc.) create 125 combinations of fault values from the fault parameter values defined above. This gives a total of 1000 combinations of fault parameter values.

```
nPerGroup = 125; % Number of elements in each fault group

% No fault simulations
leakArea = repmat(leak_area_set(1),nPerGroup,1);
blockingFactor = repmat(blockinfactor_set(1),nPerGroup,1);
bearingFactor = repmat(bearingfactor_set(1),nPerGroup,1);

% Single fault simulations
idx = ceil(10*rand(nPerGroup,1));
leakArea = [leakArea; leak_area_set(idx)'];
blockingFactor = [blockingFactor;repmat(blockinfactor_set(1),nPerGroup,1)];
bearingFactor = [bearingFactor;repmat(bearingfactor_set(1),nPerGroup,1)];
idx = ceil(10*rand(nPerGroup,1));
leakArea = [leakArea; repmat(leak_area_set(1),nPerGroup,1)];
blockingFactor = [blockingFactor;blockinfactor_set(idx)'];
bearingFactor = [bearingFactor;repmat(bearingfactor_set(1),nPerGroup,1)];
idx = ceil(10*rand(nPerGroup,1));
leakArea = [leakArea; repmat(leak_area_set(1),nPerGroup,1)];
blockingFactor = [blockingFactor;repmat(blockinfactor_set(1),nPerGroup,1)];
bearingFactor = [bearingFactor;bearingfactor_set(idx)'];

% Double fault simulations
```

```
idxA = ceil(10*rand(nPerGroup,1));
idxB = ceil(10*rand(nPerGroup,1));
leakArea = [leakArea; leak_area_set(idxA)'];
blockingFactor = [blockingFactor;blockinfactor_set(idxB)'];
bearingFactor = [bearingFactor;repmat(bearingfactor_set(1),nPerGroup,1)];
idxA = ceil(10*rand(nPerGroup,1));
idxB = ceil(10*rand(nPerGroup,1));
leakArea = [leakArea; leak_area_set(idxA)'];
blockingFactor = [blockingFactor;repmat(blockinfactor_set(1),nPerGroup,1)];
bearingFactor = [bearingFactor;bearingfactor_set(idxB)'];
idxA = ceil(10*rand(nPerGroup,1));
idxB = ceil(10*rand(nPerGroup,1));
leakArea = [leakArea; repmat(leak_area_set(1),nPerGroup,1)];
blockingFactor = [blockingFactor;blockinfactor_set(idxA)'];
bearingFactor = [bearingFactor;bearingfactor_set(idxB)'];

% Triple fault simulations
idxA = ceil(10*rand(nPerGroup,1));
idxB = ceil(10*rand(nPerGroup,1));
idxC = ceil(10*rand(nPerGroup,1));
leakArea = [leakArea; leak_area_set(idxA)'];
blockingFactor = [blockingFactor;blockinfactor_set(idxB)'];
bearingFactor = [bearingFactor;bearingfactor_set(idxC)'];
```

Use the fault parameter combinations to create `Simulink.SimulationInput` objects. For each simulation input ensure that the random seed is set differently to generate different results.

```
for ct = numel(leakArea):-1:1
    simInput(ct) = Simulink.SimulationInput(mdl);
    simInput(ct) = setVariable(simInput(ct),'leak_cyl_area_WKSP',leakArea(ct));
    simInput(ct) = setVariable(simInput(ct),'block_in_factor_WKSP',blockingFactor(ct))
    simInput(ct) = setVariable(simInput(ct),'bearing_fault_frict_WKSP',bearingFactor(c
    simInput(ct) = setVariable(simInput(ct),'noise_seed_offset_WKSP',ct-1);
end
```

Use the `generateSimulationEnsemble` function to run the simulations defined by the `Simulink.SimulationInput` objects defined above and store the results in a local sub-folder. Then create a `simulationEnsembleDatastore` from the stored results.

Note that running these 1000 simulations in parallel takes around an hour on a standard desktop and generates around 620MB of data. An option to only run the first 10 simulations is provided for convenience.

```
% Run the simulation and create an ensemble to manage the simulation results
runAll = true;
if runAll
    [ok,e] = generateSimulationEnsemble(simInput,fullfile('.','Data'),'UseParallel',tru
else
    [ok,e] = generateSimulationEnsemble(simInput(1:10),fullfile('.','Data')); %#ok<UNRC
end
```

```
[09-Apr-2018 09:01:38] Checking for availability of parallel pool...
[09-Apr-2018 09:01:38] Loading Simulink on parallel workers...
Analyzing and transferring files to the workers ...done.
[09-Apr-2018 09:01:38] Configuring simulation cache folder on parallel workers...
[09-Apr-2018 09:01:38] Running SetupFcn on parallel workers...
[09-Apr-2018 09:01:39] Loading model on parallel workers...
[09-Apr-2018 09:01:39] Transferring base workspace variables used in the model to paral
[09-Apr-2018 09:01:41] Running simulations...
[09-Apr-2018 09:02:28] Completed 1 of 1000 simulation runs
[09-Apr-2018 09:02:33] Completed 2 of 1000 simulation runs
[09-Apr-2018 09:02:37] Completed 3 of 1000 simulation runs
[09-Apr-2018 09:02:41] Completed 4 of 1000 simulation runs
[09-Apr-2018 09:02:46] Completed 5 of 1000 simulation runs
[09-Apr-2018 09:02:49] Completed 6 of 1000 simulation runs
[09-Apr-2018 09:02:54] Completed 7 of 1000 simulation runs
[09-Apr-2018 09:02:58] Completed 8 of 1000 simulation runs
[09-Apr-2018 09:03:01] Completed 9 of 1000 simulation runs...
```

```
ens = simulationEnsembleDatastore(fullfile('.','Data'));
```

Processing and Extracting Features from the Simulation Results

The model is configured to log the pump output pressure, output flow, motor speed and motor current.

```
ens.DataVariables
```

```
ans = 8×1 string array
    "SimulationInput"
    "SimulationMetadata"
    "iMotor_meas"
    "pIn_meas"
    "pOut_meas"
    "qIn_meas"
    "qOut_meas"
    "wMotor_meas"
```

For each member in the ensemble preprocess the pump output flow and compute condition indicators based on the pump output flow. The condition indicators are later used for fault classification. For preprocessing remove the first 0.8 seconds of the output flow as this contains transients from simulation and pump startup. As part of the preprocessing compute the power spectrum of the output flow, and use the SimulationInput to return the values of the fault variables.

Configure the ensemble so that the read only returns the variables of interest and call the `preprocess` function that is defined at the end of this example.

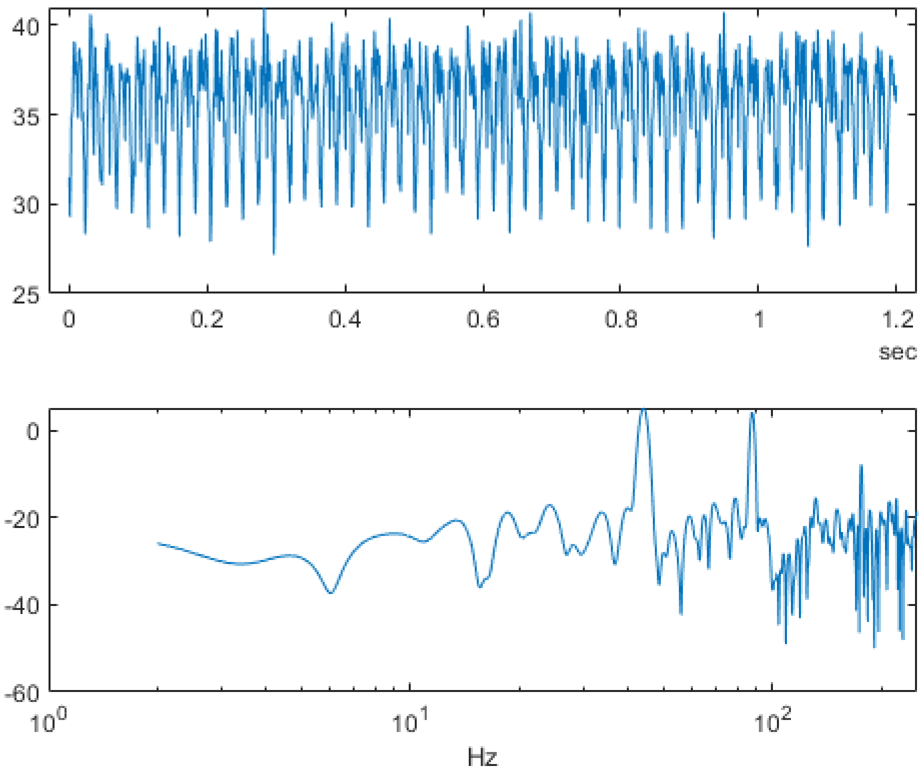
```
ens.SelectedVariables = ["qOut_meas", "SimulationInput"];
reset(ens)
data = read(ens)
```

```
data=1×2 table
        qOut_meas                 SimulationInput
    __________________    ______________________________

    [2001×1 timetable]    [1×1 Simulink.SimulationInput]
```

```
[flow,flowP,flowF,faultValues] = preprocess(data);
```

Plot the flow and flow spectrum. The plotted data is for a fault free condition.

```
% Figure with nominal
subplot(211);
plot(flow.Time,flow.Data);
subplot(212);
semilogx(flowF,pow2db(flowP));
xlabel('Hz')
```



The flow spectrum reveals resonant peaks at expected frequencies. Specifically, the pump motor speed is 950 rpm, or 15.833 Hz, and since the pump has 3 cylinders the flow is expected to have a fundamental at 3*15.833 Hz, or 47.5 Hz, as well as harmonics at multiples of 47.5 Hz. The flow spectrum clearly shows the expected resonant peaks. Faults in one cylinder of the pump will result in resonances at the pump motor speed, 15.833 Hz and its harmonics.

The flow spectrum and slow signal gives some ideas of possible condition indicators. Specifically, common signal statistics such as mean, variance, etc. as well as spectrum characteristics. Spectrum condition indicators relating to the expected harmonics such as the frequency with the peak magnitude, energy around 15.833 Hz, energy around 47.5 Hz, energy above 100 Hz, are computed. The frequency of the spectral kurtosis peak is also computed.

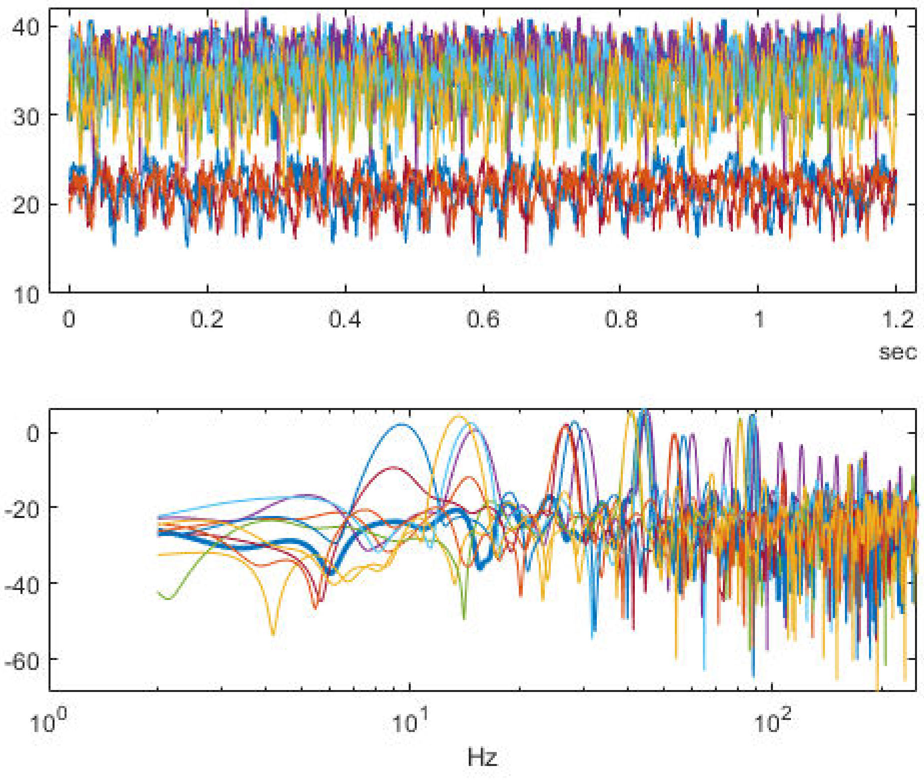
Configure the ensemble with data variables for the condition indicators and condition variables for fault variable values. Then call the `extractCI` function to compute the features, and use the `writeToLastMemberRead` command to add the feature and fault variable values to the ensemble. The `extractCI` function is defined at the end of this example.

```
ens.DataVariables = [ens.DataVariables; ...
    "fPeak"; "pLow"; "pMid"; "pHigh"; "pKurtosis"; ...
    "qMean"; "qVar"; "qSkewness"; "qKurtosis"; ...
    "qPeak2Peak"; "qCrest"; "qRMS"; "qMAD"; "qCSRange"];
ens.ConditionVariables = ["LeakFault","BlockingFault","BearingFault"];

feat = extractCI(flow,flowP,flowF);
dataToWrite = [faultValues, feat];
writeToLastMemberRead(ens,dataToWrite{:})
```

The above code preprocesses and computes the condition indicators for the first member of the simulation ensemble. Repeat this for all the members in the ensemble using the ensemble `hasdata` command. To get an idea of the simulation results under different fault conditions plot every hundredth element of the ensemble.

```
%Figure with nominal and faults
figure,
subplot(211);
lFlow = plot(flow.Time,flow.Data,'LineWidth',2);
subplot(212);
lFlowP = semilogx(flowF,pow2db(flowP),'LineWidth',2);
xlabel('Hz')
ct = 1;
lColors = get(lFlow.Parent,'ColorOrder');
lIdx = 2;

% Loop over all members in the ensemble, preprocess
% and compute the features for each member
while hasdata(ens)

    % Read member data
    data = read(ens);

    % Preprocess and extract features from the member data
    [flow,flowP,flowF,faultValues] = preprocess(data);
    feat = extractCI(flow,flowP,flowF);

    % Add the extracted feature values to the member data
```

```
        dataToWrite = [faultValues, feat];
        writeToLastMemberRead(ens,dataToWrite{:})

        % Plot member signal and spectrum for every 100th member
        if mod(ct,100) == 0
            line('Parent',lFlow.Parent,'XData',flow.Time,'YData',flow.Data, ...
                'Color', lColors(lIdx,:));
            line('Parent',lFlowP.Parent,'XData',flowF,'YData',pow2db(flowP), ...
                'Color', lColors(lIdx,:));
            if lIdx == size(lColors,1)
                lIdx = 1;
            else
                lIdx = lIdx+1;
            end
        end
        ct = ct + 1;
    end
```



Note that under different fault conditions and severities the spectrum contains harmonics at the expected frequencies.

Detect and Classify Pump Faults

The previous section preprocessed and computed condition indicators from the flow signal for all the members of the simulation ensemble, which correspond to the simulation results for different fault combinations and severities. The condition indicators can be used to detect and classify pump faults from a pump flow signal.

Configure the simulation ensemble to read the condition indicators, and use the tall and gather commands to load all the condition indicators and fault variable values into memory

```
% Get data to design a classifier.
reset(ens)
ens.SelectedVariables = [...
    "fPeak","pLow","pMid","pHigh","pKurtosis",...
    "qMean","qVar","qSkewness","qKurtosis",...
    "qPeak2Peak","qCrest","qRMS","qMAD","qCSRange",...
    "LeakFault","BlockingFault","BearingFault"];
idxLastFeature = 14;

% Load the condition indicator data into memory
data = gather(tall(ens));

Starting parallel pool (parpool) using the 'local' profile ...
Preserving jobs with IDs: 1 2 because they contain crash dump files.
You can use 'delete(myCluster.Jobs)' to remove all jobs created with profile local. To
connected to 6 workers.
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 45 sec
Evaluation completed in 45 sec

data(1:10,:)

ans=10×17 table
    fPeak      pLow        pMid       pHigh      pKurtosis     qMean      qVar       qSkewness
    ______    _______     ______     ______     _________     ______     ______     _________

    43.909    0.86472     117.63     18.874       276.49      35.572     7.5242     -0.72832
    43.909    0.44477     125.92     18.899       12.417      35.576      7.869      -0.7094
    43.909     1.1782     137.99     17.526       11.589      35.573     7.4367     -0.72208
    44.151     156.74     173.84     21.073        199.5      33.768     12.466     -0.30256
    43.848    0.71756     110.92     18.579       197.02      35.563     7.5781     -0.72377
    43.909    0.43673     119.56     20.003       11.589       35.57     7.5028     -0.74797
    43.788    0.31617      135.3     19.724       476.82      35.568     7.4406     -0.70964
    43.848    0.72747     121.63     19.733       11.589      35.523      7.791     -0.72736
    43.848    0.62777     128.85     19.244       11.589      35.541     7.5698      -0.6953
    43.848     0.4631     134.83     18.918       12.417      35.561     7.8607     -0.68417
```

The fault variable values for each ensemble member (row in the data table) can be converted to fault flags and the fault flags combined to single flag that captures the different fault status of each member.

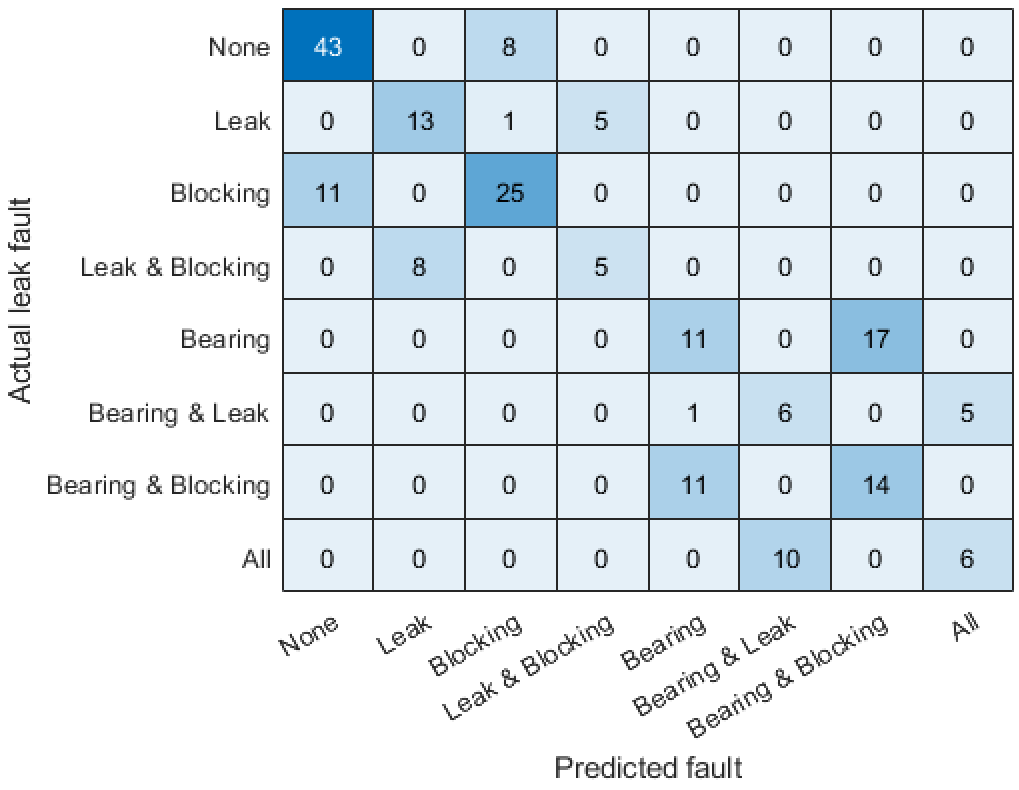
```
% Convert the fault variable values to flags
data.LeakFlag = data.LeakFault > 1e-6;
data.BlockingFlag = data.BlockingFault < 0.8;
data.BearingFlag = data.BearingFault > 0;
data.CombinedFlag = data.LeakFlag+2*data.BlockingFlag+4*data.BearingFlag;
```

Create a classifier that takes as input the condition indicators and returns the combined fault flag. Train a support vector machine that uses a 2nd order polynomial kernel. Use the `cvpartition` command to partition the ensemble members into a set for training and a set for validation.

```
rng('default') % for reproducibility
predictors = data(:,1:idxLastFeature);
response = data.CombinedFlag;
cvp = cvpartition(size(predictors,1),'KFold',5);

% Create and train the classifier
template = templateSVM(...
    'KernelFunction', 'polynomial', ...
    'PolynomialOrder', 2, ...
    'KernelScale', 'auto', ...
    'BoxConstraint', 1, ...
    'Standardize', true);
combinedClassifier = fitcecoc(...
    predictors(cvp.training(1),:), ...
    response(cvp.training(1),:), ...
    'Learners', template, ...
    'Coding', 'onevsone', ...
    'ClassNames', [0; 1; 2; 3; 4; 5; 6; 7]);
```

Check the performance of the trained classifier using the validation data and plot the results on a confusion plot.

```
% Check performance by computing and plotting the confusion matrix
actualValue = response(cvp.test(1),:);
predictedValue = predict(combinedClassifier, predictors(cvp.test(1),:));
confdata = confusionmat(actualValue,predictedValue);
figure,
labels = {'None', 'Leak','Blocking', 'Leak & Blocking', 'Bearing', ...
    'Bearing & Leak', 'Bearing & Blocking', 'All'};
h = heatmap(confdata, ...
    'YLabel', 'Actual leak fault', ...
    'YDisplayLabels', labels, ...
    'XLabel', 'Predicted fault', ...
    'XDisplayLabels', labels, ...
    'ColorbarVisible','off');
```



	None	Leak	Blocking	Leak & Blocking	Bearing	Bearing & Leak	Bearing & Blocking	All
None	43	0	8	0	0	0	0	0
Leak	0	13	1	5	0	0	0	0
Blocking	11	0	25	0	0	0	0	0
Leak & Blocking	0	8	0	5	0	0	0	0
Bearing	0	0	0	0	11	0	17	0
Bearing & Leak	0	0	0	0	1	6	0	5
Bearing & Blocking	0	0	0	0	11	0	14	0
All	0	0	0	0	0	10	0	6

The confusion plot shows for each combination of faults the number of times the fault combination was correctly predicted (the diagonal entries of the plot) and the number of times the fault combination was incorrectly predicted (the off-diagonal entries).

The confusion plot shows that the classifier did not correctly classify some fault conditions (the off diagonal terms). However, the no fault condition was correctly predicted. In a couple of places a no fault condition was predicted when there was a fault (the first column), otherwise a fault was predicted although it may not be exactly the correct fault condition. Overall the validation accuracy was 84% and the accuracy at predicting that there is a fault 98%.

```
% Compute the overall accuracy of the classifier
sum(diag(confdata))/sum(confdata(:))
```

```
ans = 0.6150

% Compute the accuracy of the classifier at predicting
% that there is a fault
1-sum(confdata(2:end,1))/sum(confdata(:)) %#ok<MNEFF>

ans = 0.9450
```

Examine the cases where no fault was predicted but a fault did exist. First find cases in the validation data where the actual fault was a blocking fault but a no fault was predicted.

```
vData = data(cvp.test(1),:);
b1 = (actualValue==2) & (predictedValue==0);
fData = vData(b1,15:17)

fData=11×3 table
    LeakFault    BlockingFault    BearingFault
    _________    _____________    ____________

      1e-09          0.77              0      
      1e-09          0.77              0      
      1e-09          0.71              0      
      1e-09          0.77              0      
      1e-09          0.77              0      
      1e-09          0.62              0      
      1e-09          0.77              0      
      1e-09          0.77              0      
      1e-09          0.71              0      
      8e-07          0.74              0      
      1e-09          0.74              0      
```

Find cases in the validation data where the actual fault was a bearing fault but a no fault was predicted.

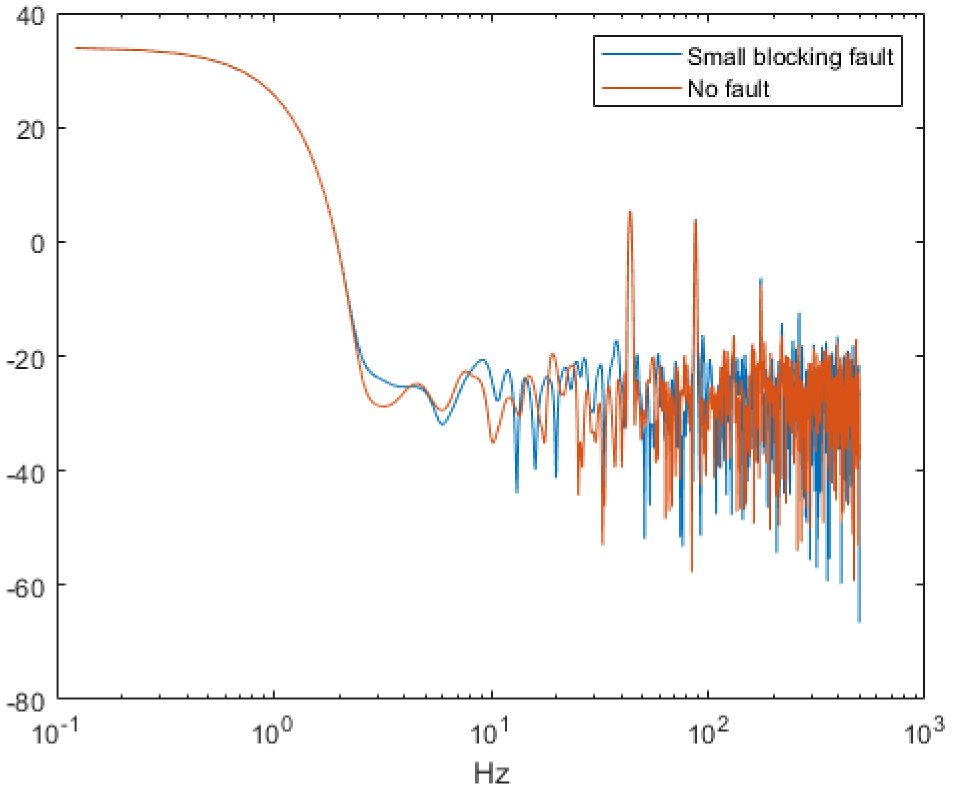
```
b2 = (actualValue==4) & (predictedValue==0);
vData(b2,15:17)

ans =

  0×3 empty table
```

Examining the cases where no fault was predictive but a fault did exist reveals that they occur when the blocking fault value of 0.77 is close to its nominal value of 0.8, or the bearing fault value of 6.6e-5 is close to its nominal value of 0. Plotting the spectrum for the case with a small blocking fault value and comparing with a fault free condition reveals that spectra are very similar making detection difficult. Re-training the classifier but including a blocking value of 0.77 as a non fault condition would significantly improve the performance of the fault detector. Alternatively, using additional pump measurements could provide more information and improve the ability to detect small blocking faults.

```
% Configure the ensemble to only read the flow and fault variable values
ens.SelectedVariables = ["qOut_meas","LeakFault","BlockingFault","BearingFault"];
reset(ens)

% Load the ensemble member data into memory
data = gather(tall(ens));
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 38 sec
Evaluation completed in 38 sec
```

```
% Look for the member that was incorrectly predicted, and
% compute its power spectrum
idx = ...
    data.BlockingFault == fData.BlockingFault(1) & ...
    data.LeakFault == fData.LeakFault(1) & ...
    data.BearingFault == fData.BearingFault(1);
flow1 = data(idx,1);
flow1 = flow1.qOut_meas{1};
[flow1P,flow1F] = pspectrum(flow1);

% Look for a member that does not have any faults
idx = ...
    data.BlockingFault == 0.8 & ...
    data.LeakFault == 1e-9 & ...
    data.BearingFault == 0;
flow2 = data(idx,1);
flow2 = flow2.qOut_meas{1};
[flow2P,flow2F] = pspectrum(flow2);

% Plot the power spectra
semilogx(...
    flow1F,pow2db(flow1P),...
    flow2F,pow2db(flow2P));
xlabel('Hz')
legend('Small blocking fault','No fault')
```



Conclusion

This example showed how to use a Simulink model to model faults in a reciprocating pump, simulate the model under different fault combinations and severities, extract condition indicators from the pump output flow and use the condition indicators to train a classifier to detect pump faults. The example examined the performance of fault detection using the classifier and noted that small blocking faults are very similar to the no fault condition and cannot be reliably detected.

Supporting Functions

```
function [flow,flowSpectrum,flowFrequencies,faultValues] = preprocess(data)
% Helper function to preprocess the logged reciprocating pump data.
% Remove the 1st 0.8 seconds of the flow signal
tMin = seconds(0.8);
flow = data.qOut_meas{1};
flow = flow(flow.Time >= tMin,:);
flow.Time = flow.Time - flow.Time(1);

% Ensure the flow is sampled at a uniform sample rate
flow = retime(flow,'regular','linear','TimeStep',seconds(1e-3));

% Remove the mean from the flow and compute the flow spectrum
fA = flow;
fA.Data = fA.Data - mean(fA.Data);
[flowSpectrum,flowFrequencies] = pspectrum(fA,'FrequencyLimits',[2 250]);

% Find the values of the fault variables from the SimulationInput
simin = data.SimulationInput{1};
vars = {simin.Variables.Name};
idx = strcmp(vars,'leak_cyl_area_WKSP');
LeakFault = simin.Variables(idx).Value;
idx = strcmp(vars,'block_in_factor_WKSP');
BlockingFault = simin.Variables(idx).Value;
idx = strcmp(vars,'bearing_fault_frict_WKSP');
BearingFault = simin.Variables(idx).Value;

% Collect the fault values in a cell array
faultValues = {...
    'LeakFault', LeakFault, ...
    'BlockingFault', BlockingFault, ...
    'BearingFault', BearingFault};
end

function ci = extractCI(flow,flowP,flowF)
% Helper function to extract condition indicators from the flow signal
% and spectrum.

% Find the frequency of the peak magnitude in the power spectrum.
pMax = max(flowP);
fPeak = flowF(flowP==pMax);

% Compute the power in the low frequency range 10-20 Hz.
fRange = flowF >= 10 & flowF <= 20;
pLow = sum(flowP(fRange));

% Compute the power in the mid frequency range 40-60 Hz.
```

```
fRange = flowF >= 40 & flowF <= 60;
pMid = sum(flowP(fRange));

% Compute the power in the high frequency range >100 Hz.
fRange = flowF >= 100;
pHigh = sum(flowP(fRange));

% Find the frequency of the spectral kurtosis peak
[pKur,fKur] = pkurtosis(flow);
pKur = fKur(pKur == max(pKur));

% Compute the flow cumulative sum range.
csFlow = cumsum(flow.Data);
csFlowRange = max(csFlow)-min(csFlow);

% Collect the feature and feature values in a cell array.
% Add flow statistic (mean, variance, etc.) and common signal
% characteristics (rms, peak2peak, etc.) to the cell array.
ci = {...
    'qMean', mean(flow.Data), ...
    'qVar',  var(flow.Data), ...
    'qSkewness', skewness(flow.Data), ...
    'qKurtosis', kurtosis(flow.Data), ...
    'qPeak2Peak', peak2peak(flow.Data), ...
    'qCrest', peak2rms(flow.Data), ...
    'qRMS', rms(flow.Data), ...
    'qMAD', mad(flow.Data), ...
    'qCSRange',csFlowRange, ...
    'fPeak', fPeak, ...
    'pLow', pLow, ...
    'pMid', pMid, ...
    'pHigh', pHigh, ...
    'pKurtosis', pKur(1)};
end
```

See Also

`simulationEnsembleDatastore`

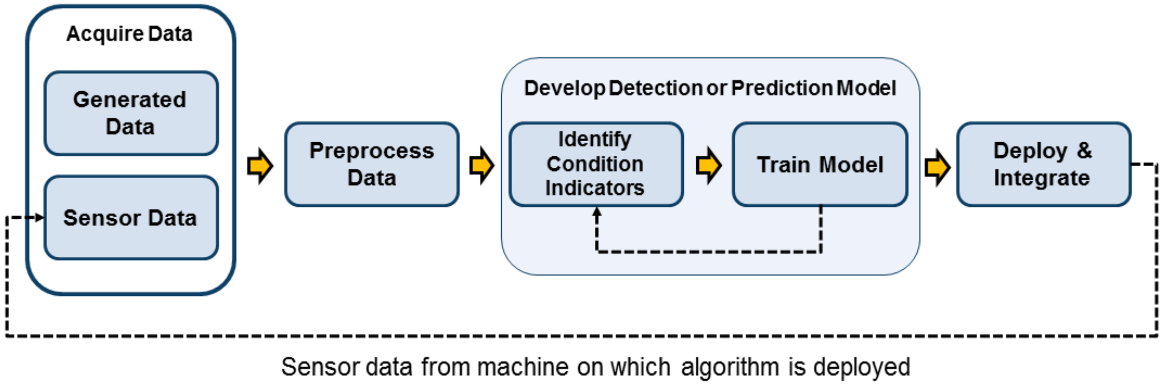
More About

- “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:


Acquire Data
Generated Data
Sensor Data
Preprocess Data
Develop Detection or Prediction Model
Identify Condition Indicators
Train Model
Deploy & Integrate
Sensor data from machine on which algorithm is deployed

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” on page 1-35.

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 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

Predictive Maintenance Toolbox and Signal Processing Toolbox provides functions that let you study and characterize vibrations in mechanical systems in the time domain. Use these functions for preprocessing or 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” on page 1-35 uses time-synchronous averaging to preprocess vibration data.
- `tsadifference` — Remove the residual signal, and specific sidebands with their harmonics from a time-synchronous averaged (TSA) signal.
- `tsaregular` — Isolate the known signal from a TSA signal by removing the residual signal and specific sidebands.
- `tsaresidual` — Isolate the residual signal from a TSA signal by removing the known signal components and their harmonics.
- `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” on page 4-8 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” on page 1-62 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” on page 4-8 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” on page 5-66 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” on page 4-8 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

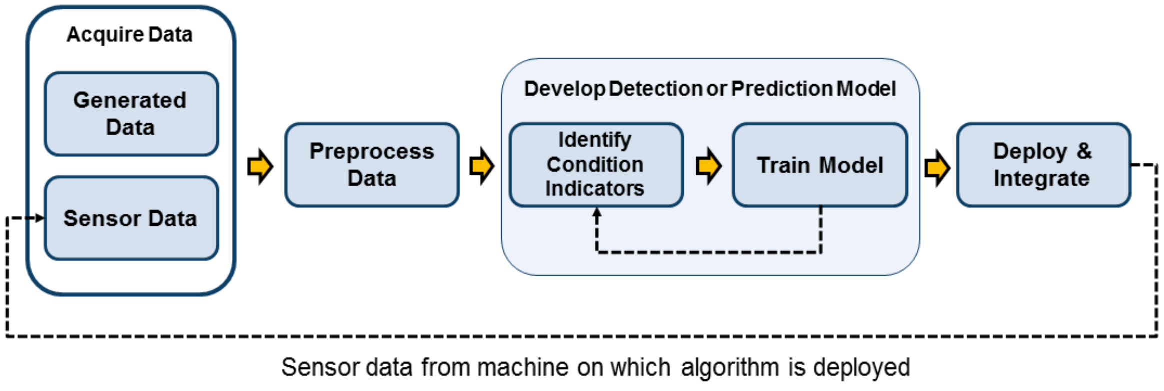
Identify Condition Indicators

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.


Acquire Data
Generated Data
Sensor Data
Preprocess Data
Develop Detection or Prediction Model
Identify Condition Indicators
Train Model
Deploy & Integrate
Sensor data from machine on which algorithm is deployed

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” on page 4-71 and “Using Simulink to Generate Fault Data” on page 1-35 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” on page 4-8 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” on page 1-62.

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” on page 1-35 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” on page 5-66 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” on page 4-8 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 5-4.)

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” on page 4-39 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 autorecursive (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” on page 4-131 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” on page 4-71 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” on page 4-110 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” on page 4-95
- “Nonlinear State Estimation of a Degrading Battery System” on page 5-87

See Also

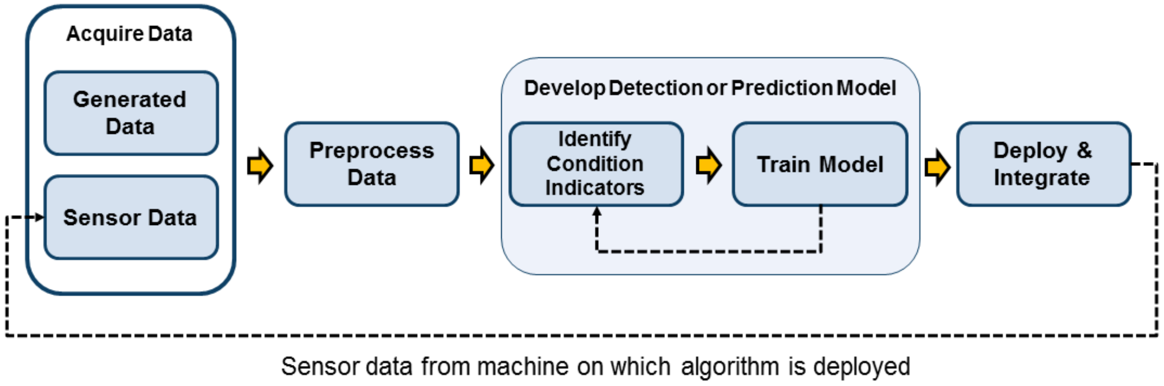
More About

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

Detect and Diagnose Faults

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.


Acquire Data
Generated Data
Sensor Data
Preprocess Data
Develop Detection or Prediction Model
Identify Condition Indicators
Train Model
Deploy & Integrate
Sensor data from machine on which algorithm is deployed

(For information about using condition indicators for fault prediction, see “Models for Predicting Remaining Useful Life” on page 5-4.)

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 extracting 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" on page 5-44 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" on page 1-35 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” on page 4-39 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” on page 1-35 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” on page 4-39 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” on page 1-62 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” on page 4-39 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” on page 4-110 uses this approach. Functions you can use for identifying dynamic models include:

- `ssest`
- `arx`, `armax`, `ar`
- `nlarx`

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" on page 4-110 uses this approach.

See Also

More About

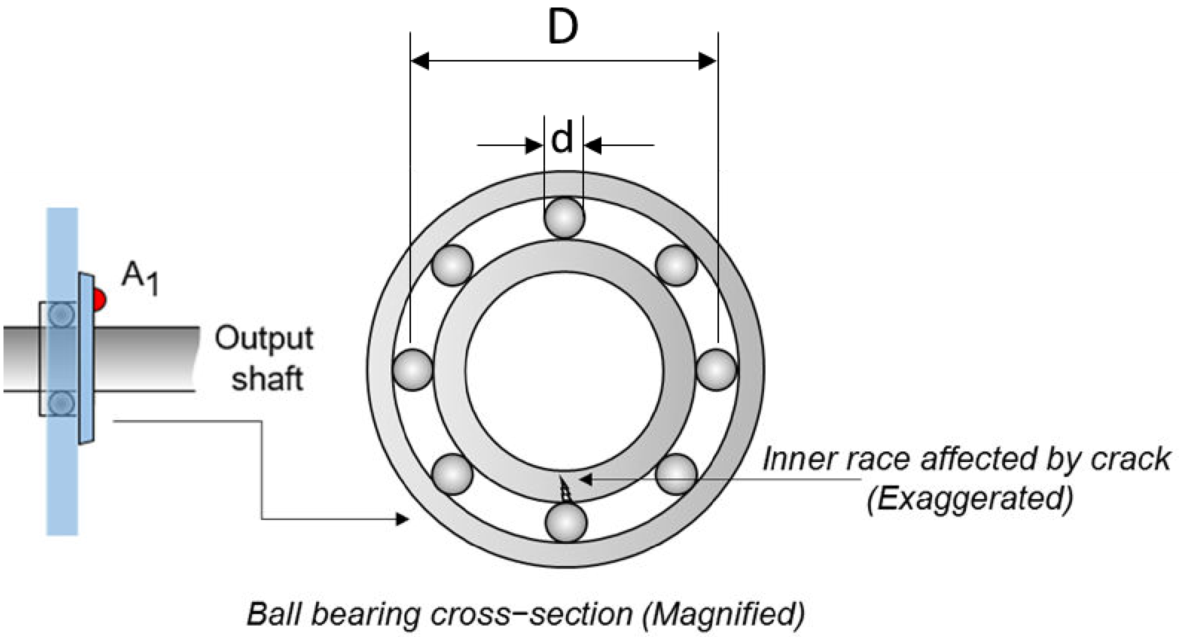
- "Condition Indicators for Monitoring, Fault Detection, and Prediction" on page 3-2
- "Models for Predicting Remaining Useful Life" on page 5-4

Rolling Element Bearing Fault Diagnosis

This example shows how to perform fault diagnosis of a rolling element bearing based on acceleration signals, especially in the presence of strong masking signals from other machine components. The example will demonstrate how to apply envelope spectrum analysis and spectral kurtosis to diagnose bearing faults and it is able to scale up to Big Data applications.

Problem Overview

Localized faults in a rolling element bearing may occur in the outer race, the inner race, the cage, or a rolling element. High frequency resonances between the bearing and the response transducer are excited when the rolling elements strike a local fault on the outer or inner race, or a fault on a rolling element strikes the outer or inner race [1]. The following picture shows a rolling element striking a local fault at the inner race. The problem is how to detect and identify the various types of faults.


D
d
A_1
Output shaft
Inner race affected by crack (Exaggerated)
Ball bearing cross−section (Magnified)

Machinery Failure Prevention Technology (MFPT) Challenge Data

MFPT Challenge data [4] contains 23 data sets collected from machines under various fault conditions. The first 20 data sets are collected from a bearing test rig, with 3 under good conditions, 3 with outer race faults under constant load, 7 with outer race faults under various loads, and 7 with inner race faults under various loads. The remaining 3 data sets are from real-world machines: an oil pump bearing, an intermediate speed bearing, and a planet bearing. The fault locations are unknown. In this example, only the data collected from the test rig with known conditions are used.

Each data set contains an acceleration signal "gs", sampling rate "sr", shaft speed "rate", load weight "load", and four critical frequencies representing different fault locations: ballpass frequency outer race (BPFO), ballpass frequency inner race (BPFI), fundamental train frequency (FTF), and ball spin frequency (BSF). Here are the formulae for those critical frequencies [1].

- Ballpass frequency, outer race (BPFO)

$$BPFO = \frac{nf_r}{2}\left(1 - \frac{d}{D}\cos\phi\right)$$

- Ballpass frequency, inner race (BPFI)

$$BPFI = \frac{nf_r}{2}\left(1 + \frac{d}{D}\cos\phi\right)$$

- Fundamental train frequency (FTF), also known as cage speed

$$FTF = \frac{f_r}{2}\left(1 - \frac{d}{D}\cos\phi\right)$$

- Ball (roller) spin frequency

$$BSF = \frac{D}{2d}\left[1 - \left(\frac{d}{D}\cos\phi\right)^2\right]$$

As shown in the figure, d is the ball diameter, D is the pitch diameter. The variable f_r is the shaft speed, n is the number of rolling elements, ϕ is the bearing contact angle [1].

Envelope Spectrum Analysis for Bearing Diagnosis

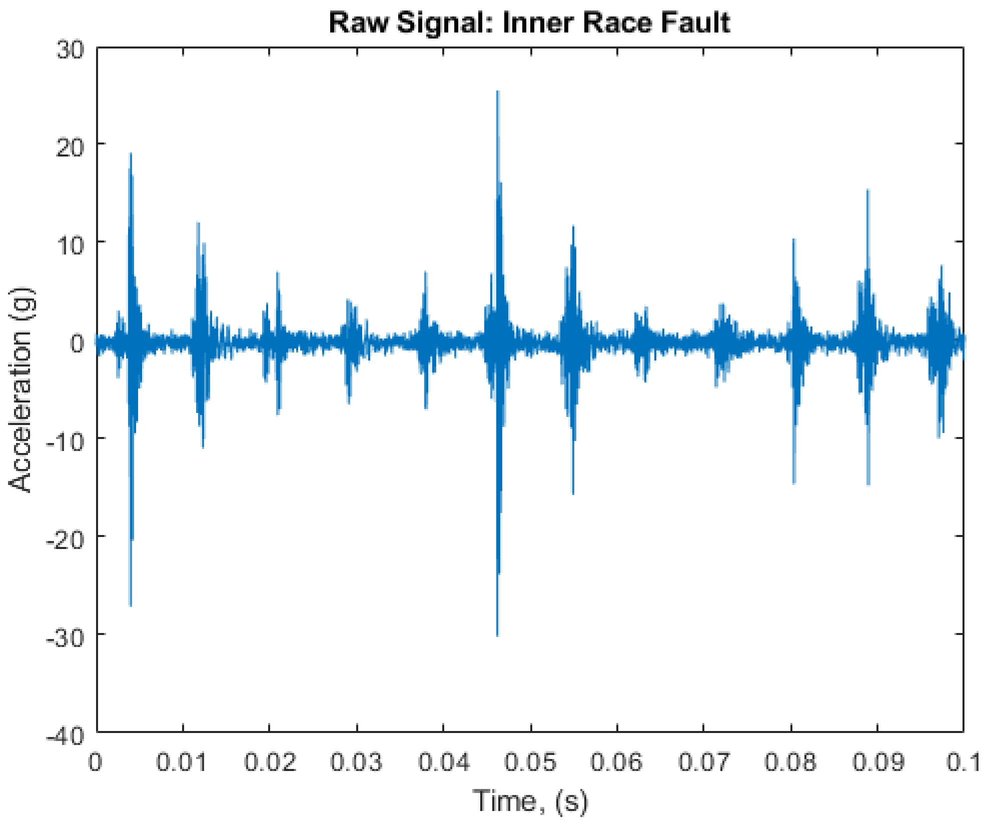
In the MFPT data set, the shaft speed is constant, hence there is no need to perform order tracking as a pre-processing step to remove the effect of shaft speed variations.

When rolling elements hit the local faults at outer or inner races, or when faults on the rolling element hit the outer or inner races, the impact will modulate the corresponding critical frequencies, e.g. BPFO, BPFI, FTF, BSF. Therefore, the envelope signal produced by amplitude demodulation conveys more diagnostic information that is not available from spectrum analysis of the raw signal. Take an inner race fault signal in the MFPT dataset as an example.

```
dataInner = load(fullfile(matlabroot, 'toolbox', 'predmaint', ...
    'predmaintdemos', 'bearingFaultDiagnosis', ...
    'train_data', 'InnerRaceFault_vload_1.mat'));
```

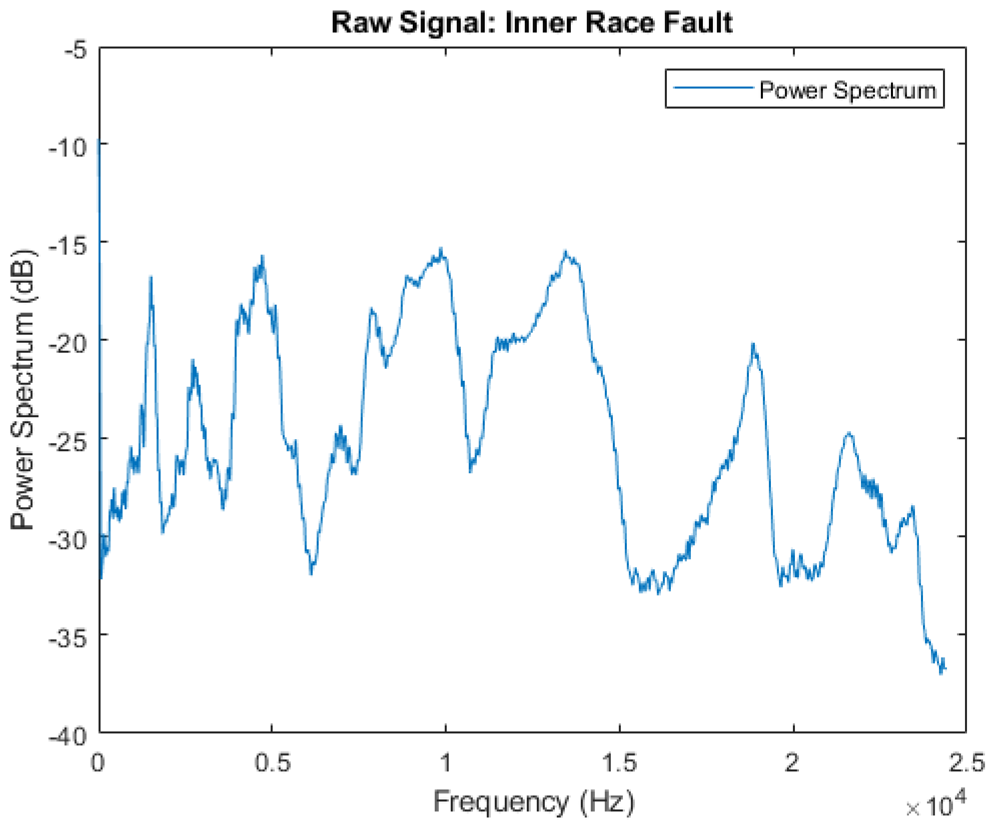
Visualize the raw inner race fault data in time domain.

```
xInner = dataInner.bearing.gs;
fsInner = dataInner.bearing.sr;
tInner = (0:length(xInner)-1)/fsInner;
figure
plot(tInner, xInner)
xlabel('Time, (s)')
ylabel('Acceleration (g)')
title('Raw Signal: Inner Race Fault')
xlim([0 0.1])
```



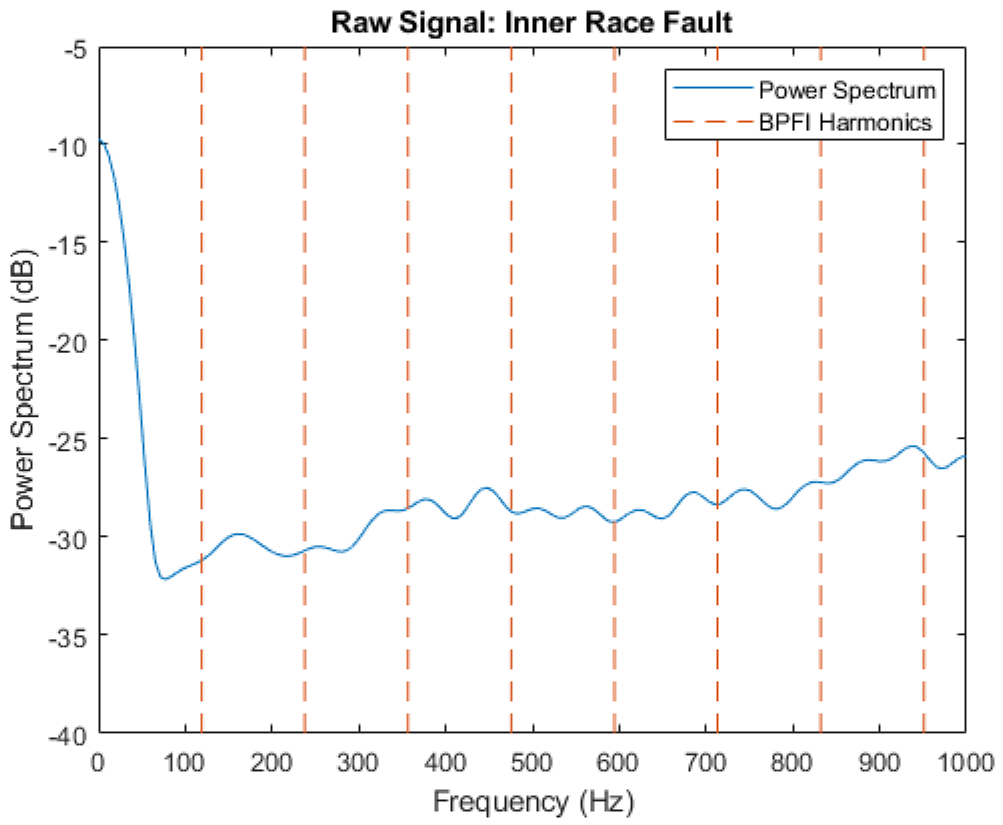
Visualize the raw data in frequency domain.

```
figure
[pInner, fpInner] = pspectrum(xInner, fsInner);
pInner = 10*log10(pInner);
plot(fpInner, pInner)
xlabel('Frequency (Hz)')
ylabel('Power Spectrum (dB)')
title('Raw Signal: Inner Race Fault')
legend('Power Spectrum')
```



Now zoom in the power spectrum of the raw signal in low frequency range to take a closer look at the frequency response at BPFI and its first several harmonics.

```
figure
plot(fpInner, pInner)
ncomb = 10;
helperPlotCombs(ncomb, dataInner.BPFI)
xlabel('Frequency (Hz)')
ylabel('Power Spectrum (dB)')
title('Raw Signal: Inner Race Fault')
legend('Power Spectrum', 'BPFI Harmonics')
xlim([0 1000])
```



No clear pattern is visible at BPFI and its harmonics. Frequency analysis on the raw signal does not provide useful diagnosis information.

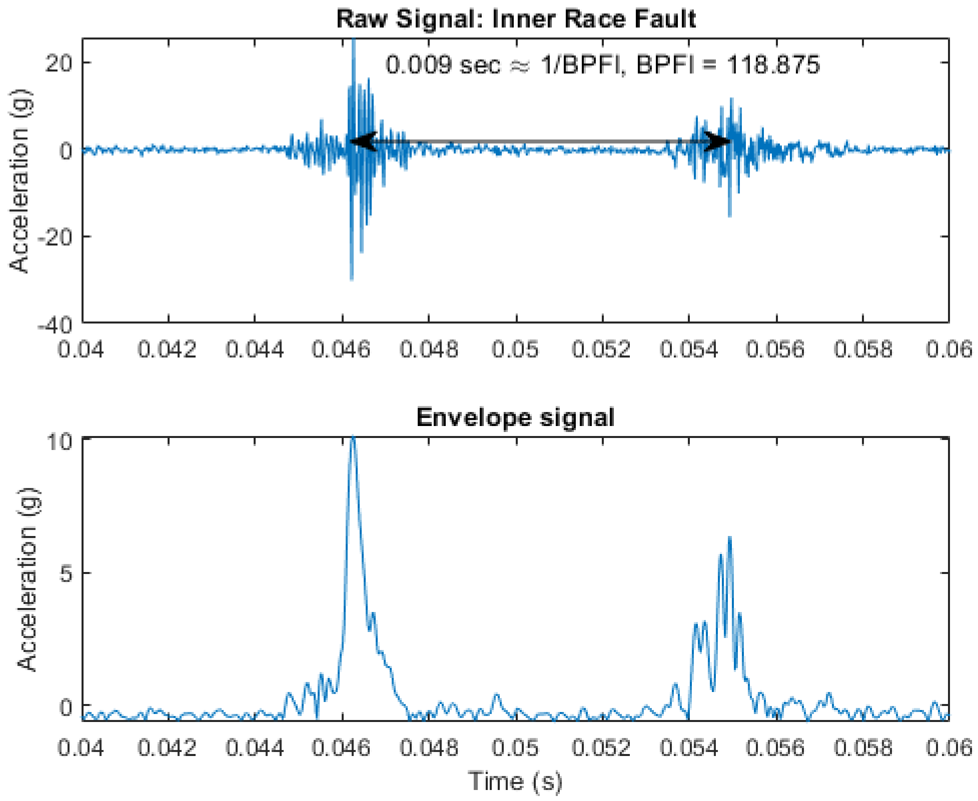
Looking at the time-domain data, it is observed that the amplitude of the raw signal is modulated at a certain frequency, and the main frequency of the modulation is around 1/0.009 Hz $\approx$ 111 Hz. It is known that the frequency the rolling element hitting a local fault at the inner race, that is BPFI, is 118.875 Hz. This indicates that the bearing potentially has an inner race fault.

```
figure
subplot(2, 1, 1)
plot(tInner, xInner)
xlim([0.04 0.06])
```

```
title('Raw Signal: Inner Race Fault')
ylabel('Acceleration (g)')
annotation('doublearrow', [0.37 0.71], [0.8 0.8])
text(0.047, 20, ['0.009 sec \approx 1/BPFI, BPFI = ' num2str(dataInner.BPFI)])
```

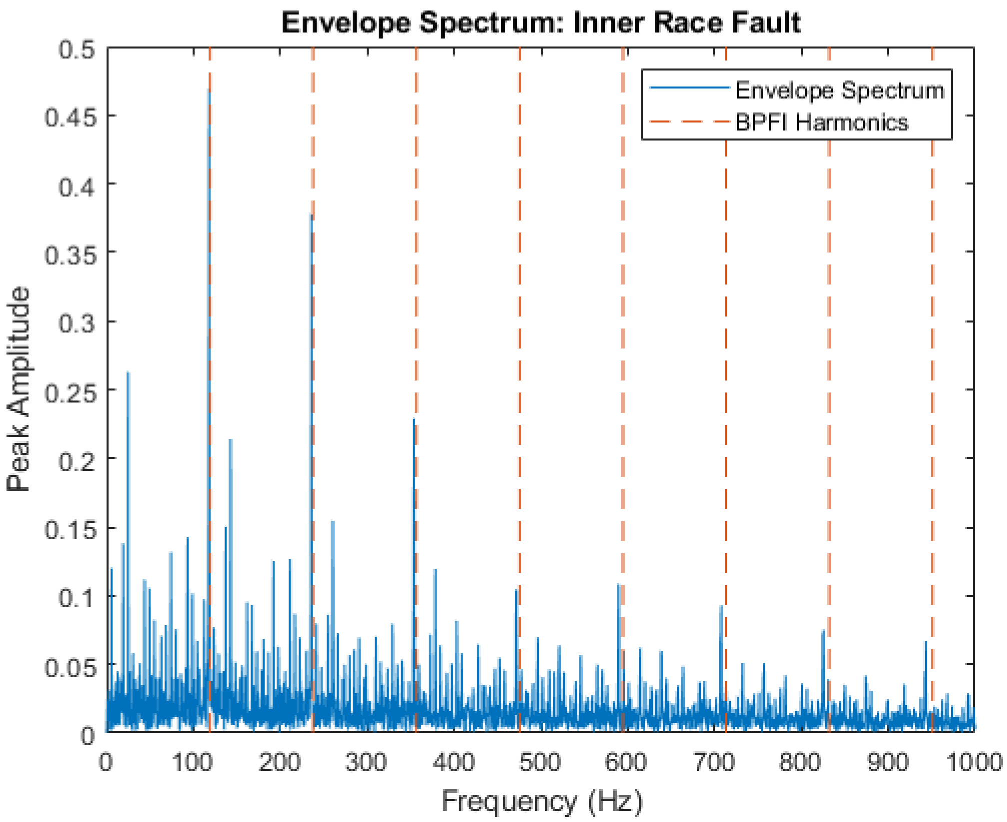
To extract the modulated amplitude, compute the envelope of the raw signal, and visualize it on the bottom subplot.

```
subplot(2, 1, 2)
[pEnvInner, fEnvInner, xEnvInner, tEnvInner] = envspectrum(xInner, fsInner);
plot(tEnvInner, xEnvInner)
xlim([0.04 0.06])
xlabel('Time (s)')
ylabel('Acceleration (g)')
title('Envelope signal')
```



Now compute the power spectrum of the envelope signal and take a look at the frequency response at BPFI and its harmonics.

```
figure
plot(fEnvInner, pEnvInner)
xlim([0 1000])
ncomb = 10;
helperPlotCombs(ncomb, dataInner.BPFI)
xlabel('Frequency (Hz)')
ylabel('Peak Amplitude')
title('Envelope Spectrum: Inner Race Fault')
legend('Envelope Spectrum', 'BPFI Harmonics')
```



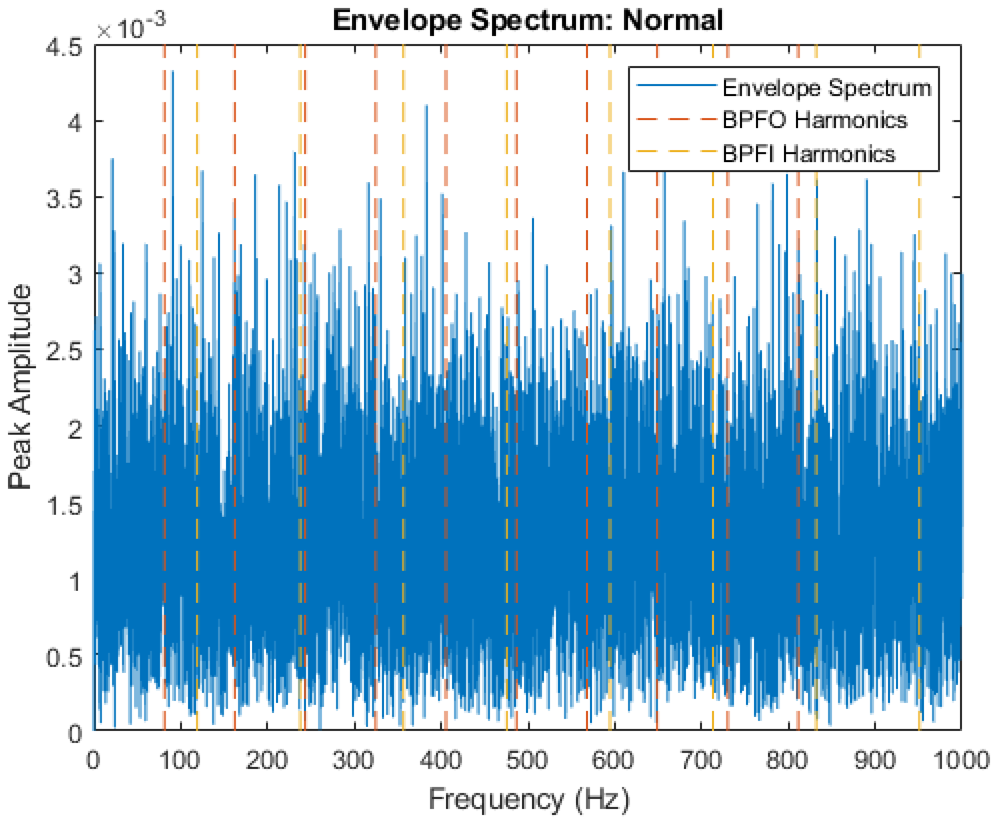
It is shown that most of the energy is focused at BPFI and its harmonics. That indicates an inner race fault of the bearing, which matches the fault type of the data.

Applying Envelope Spectrum Analysis to Other Fault Types

Now repeat the same envelope spectrum analysis on normal data and outer race fault data.

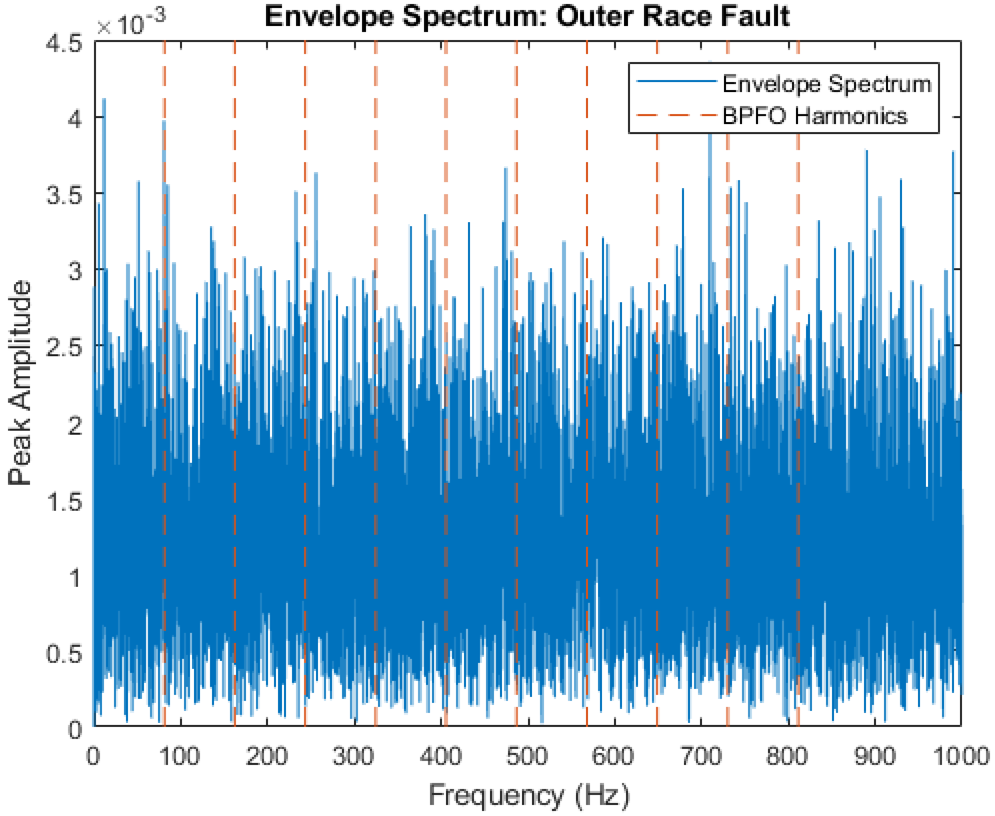
```
dataNormal = load(fullfile(matlabroot, 'toolbox', 'predmaint', ...
    'predmaintdemos', 'bearingFaultDiagnosis', ...
    'train_data', 'baseline_1.mat'));
xNormal = dataNormal.bearing.gs;
fsNormal = dataNormal.bearing.sr;
tNormal = (0:length(xNormal)-1)/fsNormal;
```

```
[pEnvNormal, fEnvNormal] = envspectrum(xNormal, fsNormal);

figure
plot(fEnvNormal, pEnvNormal)
ncomb = 10;
helperPlotCombs(ncomb, [dataNormal.BPFO dataNormal.BPFI])
xlim([0 1000])
xlabel('Frequency (Hz)')
ylabel('Peak Amplitude')
title('Envelope Spectrum: Normal')
legend('Envelope Spectrum', 'BPFO Harmonics', 'BPFI Harmonics')
```



As expected, the envelope spectrum of a normal bearing signal does not show any significant peaks at BPFO or BPFI.

```
dataOuter = load(fullfile(matlabroot, 'toolbox', 'predmaint', ...
    'predmaintdemos', 'bearingFaultDiagnosis', ...
    'train_data', 'OuterRaceFault_2.mat'));
xOuter = dataOuter.bearing.gs;
fsOuter = dataOuter.bearing.sr;
tOuter = (0:length(xOuter)-1)/fsOuter;
[pEnvOuter, fEnvOuter, xEnvOuter, tEnvOuter] = envspectrum(xOuter, fsOuter);

figure
plot(fEnvOuter, pEnvOuter)
ncomb = 10;
helperPlotCombs(ncomb, dataOuter.BPFO)
xlim([0 1000])
xlabel('Frequency (Hz)')
ylabel('Peak Amplitude')
title('Envelope Spectrum: Outer Race Fault')
legend('Envelope Spectrum', 'BPFO Harmonics')
```

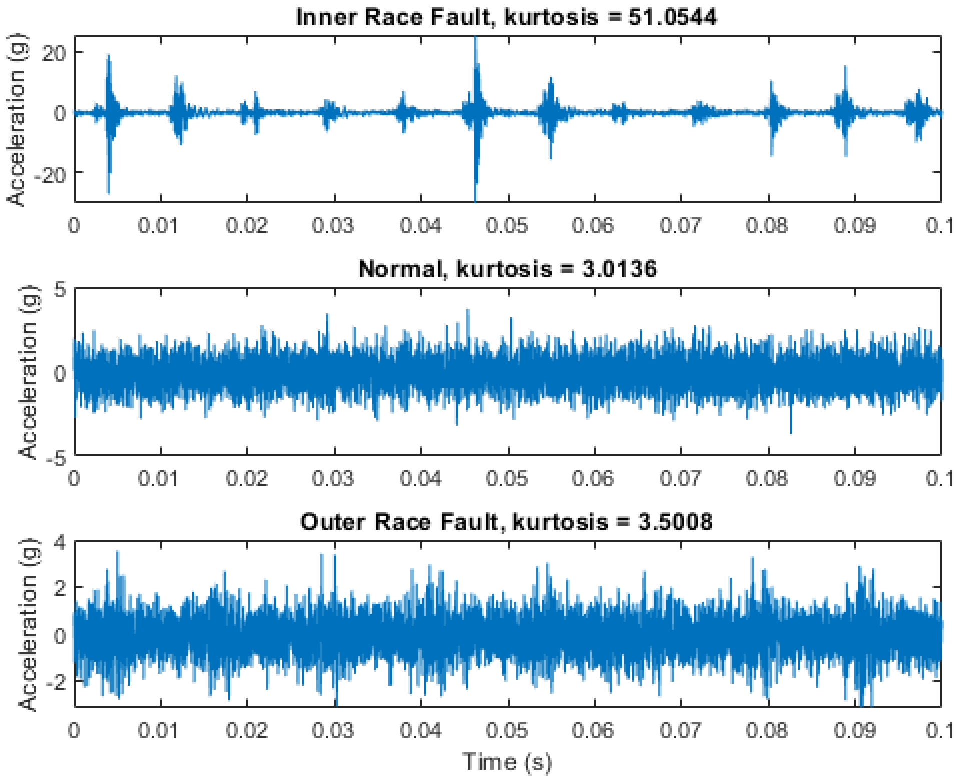


For an outer race fault signal, there are no clear peaks at BPFO harmonics either. Does envelope spectrum analysis fail to differentiate bearing with outer race fault from healthy bearings? Let's take a step back and look at the signals in time domain under different conditions again.

First let's visualize the signals in time domain again and calculate their kurtosis. Kurtosis is the fourth standardized moment of a random variable. It characterizes the impulsiveness of the signal or the heaviness of the random variable's tail.

```
figure
subplot(3, 1, 1)
kurtInner = kurtosis(xInner);
plot(tInner, xInner)
ylabel('Acceleration (g)')
```

```
title(['Inner Race Fault, kurtosis = ' num2str(kurtInner)])
xlim([0 0.1])

subplot(3, 1, 2)
kurtNormal = kurtosis(xNormal);
plot(tNormal, xNormal)
ylabel('Acceleration (g)')
title(['Normal, kurtosis = ' num2str(kurtNormal)])
xlim([0 0.1])

subplot(3, 1, 3)
kurtOuter = kurtosis(xOuter);
plot(tOuter, xOuter)
xlabel('Time (s)')
ylabel('Acceleration (g)')
title(['Outer Race Fault, kurtosis = ' num2str(kurtOuter)])
xlim([0 0.1])
```

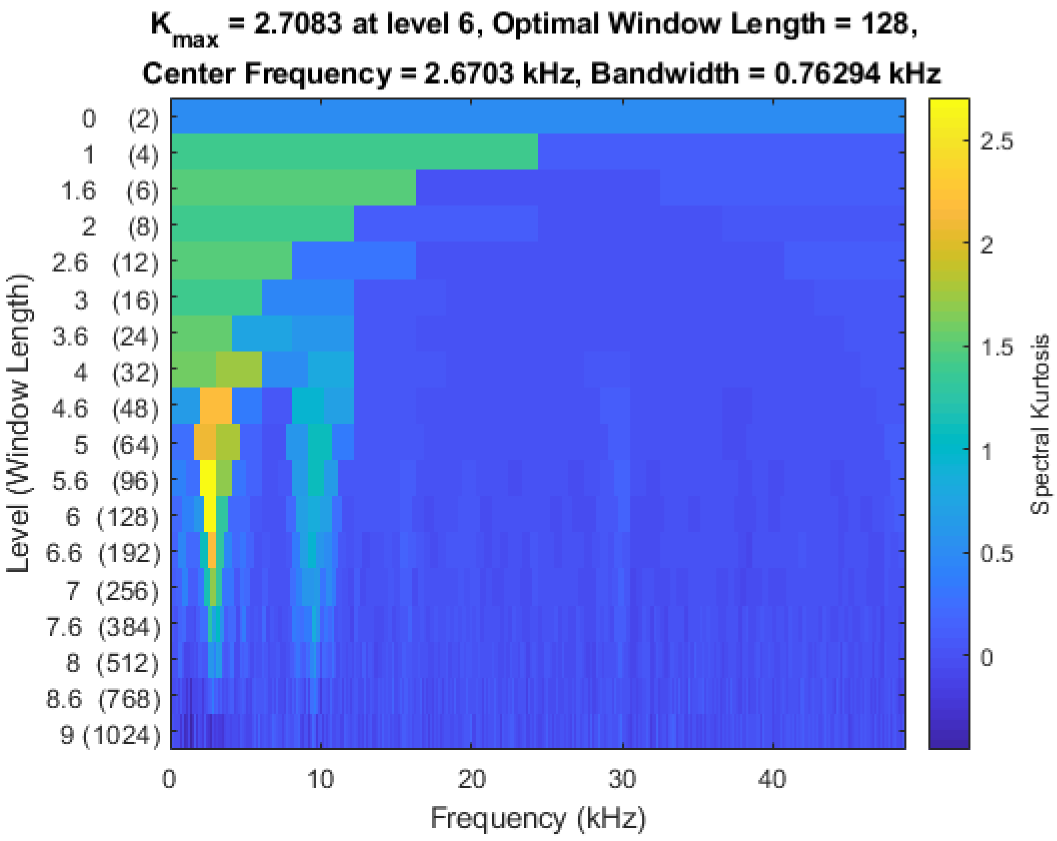


It is shown that inner race fault signal has significantly larger impulsiveness, making envelope spectrum analysis capture the fault signature at BPFI effectively. For an outer race fault signal, the amplitude modulation at BPFO is slightly noticeable, but it is masked by strong noise. The normal signal does not show any amplitude modulation. Extracting the impulsive signal with amplitude modulation at BPFO (or enhancing the signal-to-noise ratio) is a key preprocessing step before envelope spectrum analysis. The next section will introduce kurtogram and spectral kurtosis to extract the signal with highest kurtosis, and perform envelope spectrum analysis on the filtered signal.

Kurtogram and Spectral Kurtosis for Band Selection

Kurtogram and spectral kurtosis compute kurtosis locally within frequency bands. They are powerful tools to locate the frequency band that has the highest kurtosis (or the highest signal-to-noise ratio) [2]. After pinpointing the frequency band with the highest kurtosis, a bandpass filter can be applied to the raw signal to obtain a more impulsive signal for envelope spectrum analysis.

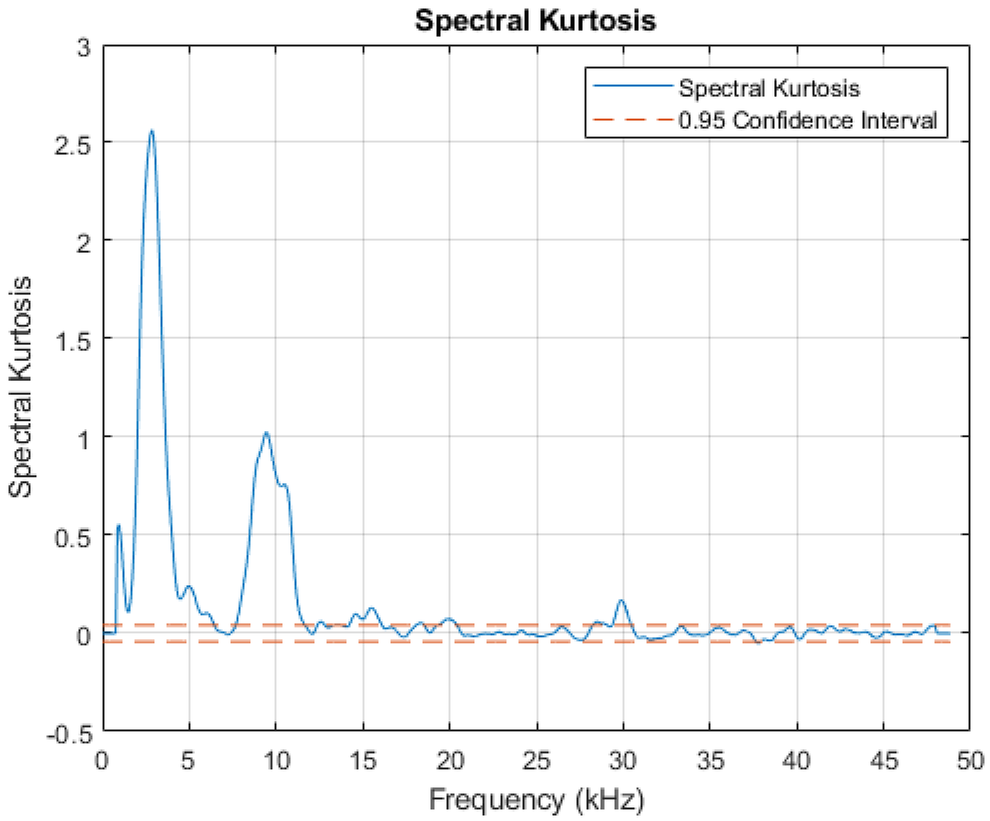
```
level = 9;
figure
kurtogram(xOuter, fsOuter, level)
```



The kurtogram indicates that the frequency band centered at 2.67 kHz with a 0.763 kHz bandwidth has the highest kurtosis of 2.71.

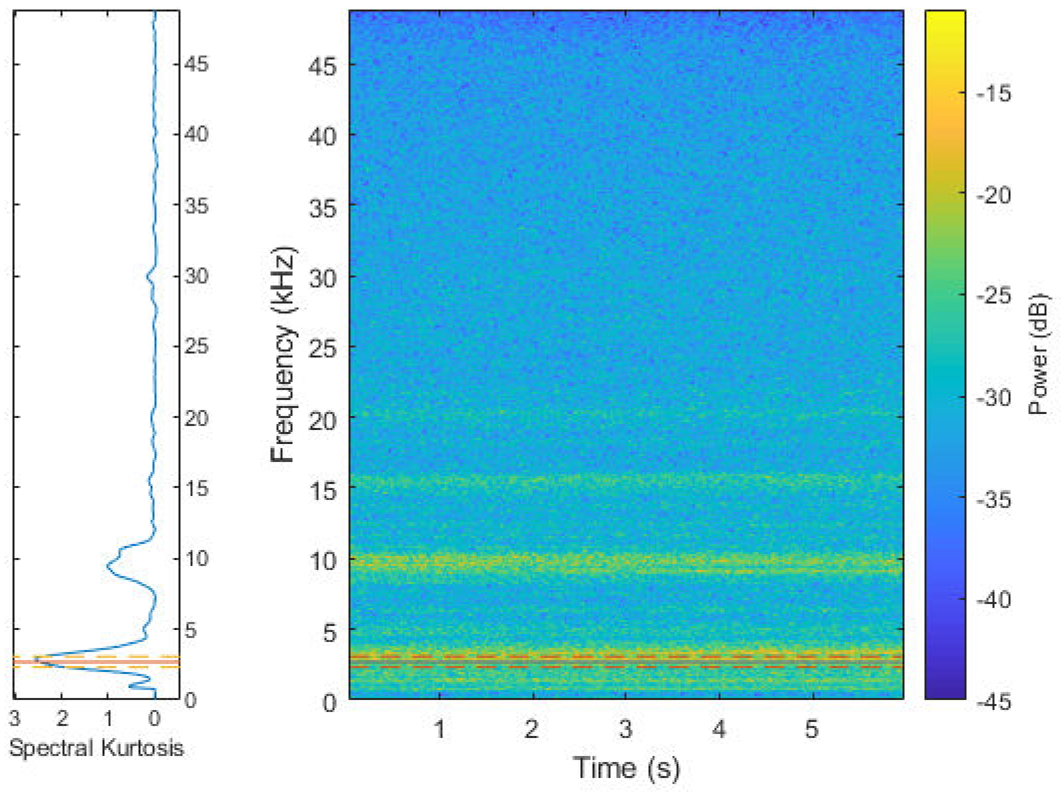
Now use the optimal window length suggested by the kurtogram to compute the spectral kurtosis.

```
figure
wc = 128;
pkurtosis(xOuter, fsOuter, wc)
```



To visualize the frequency band on a spectrogram, compute the spectrogram and place the spectral kurtosis on the side. To interpret the spectral kurtosis in another way, high spectral kurtosis values indicates high variance of power at the corresponding frequency, which makes spectral kurtosis a useful tool to locate nonstationary components of the signal [3].

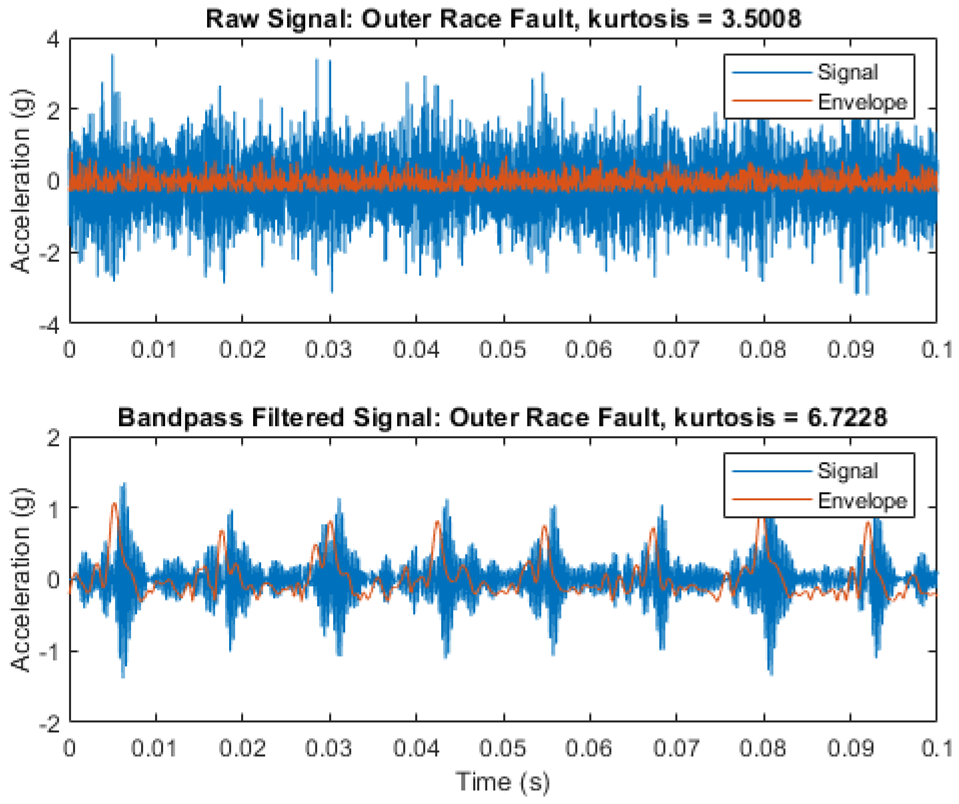
```
helperSpectrogramAndSpectralKurtosis(xOuter, fsOuter, level)
```



By bandpass filtering the signal with the suggested center frequency and bandwidth, the kurtosis can be enhanced and the modulated amplitude of the outer race fault can be retrieved.

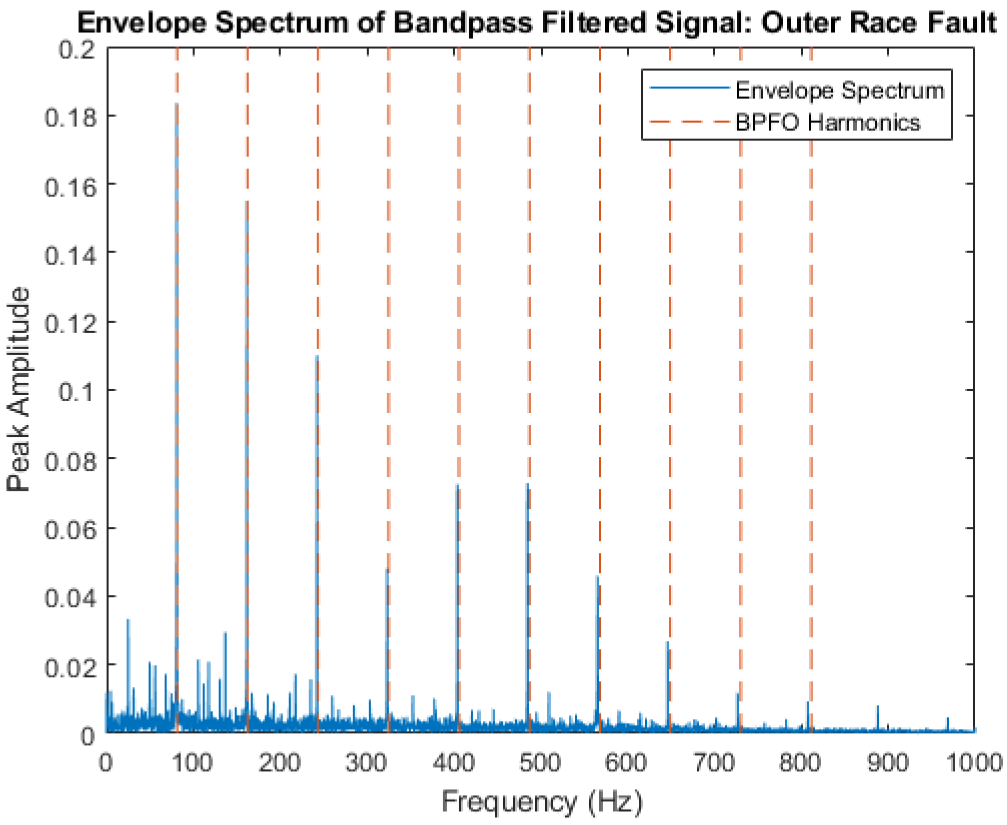
```
[~, ~, ~, fc, ~, BW] = kurtogram(xOuter, fsOuter, level);

bpf = designfilt('bandpassfir', 'FilterOrder', 200, 'CutoffFrequency1', fc-BW/2, ...
    'CutoffFrequency2', fc+BW/2, 'SampleRate', fsOuter);
xOuterBpf = filter(bpf, xOuter);
[pEnvOuterBpf, fEnvOuterBpf, xEnvOuterBpf, tEnvBpfOuter] = envspectrum(xOuter, fsOuter
    'FilterOrder', 200, 'Band', [fc-BW/2 fc+BW/2]);

figure
subplot(2, 1, 1)
```

```
plot(tOuter, xOuter, tEnvOuter, xEnvOuter)
ylabel('Acceleration (g)')
title(['Raw Signal: Outer Race Fault, kurtosis = ', num2str(kurtOuter)])
xlim([0 0.1])
legend('Signal', 'Envelope')

subplot(2, 1, 2)
kurtOuterBpf = kurtosis(xOuterBpf);
plot(tOuter, xOuterBpf, tEnvBpfOuter, xEnvOuterBpf)
ylabel('Acceleration (g)')
xlim([0 0.1])
xlabel('Time (s)')
title(['Bandpass Filtered Signal: Outer Race Fault, kurtosis = ', num2str(kurtOuterBpf
legend('Signal', 'Envelope')
```



It can be seen that the kurtosis value is increased after bandpass filtering. Now visualize the envelope signal in frequency domain.

```
figure
plot(fEnvOuterBpf, pEnvOuterBpf);
ncomb = 10;
helperPlotCombs(ncomb, dataOuter.BPFO)
xlim([0 1000])
xlabel('Frequency (Hz)')
ylabel('Peak Amplitude')
title('Envelope Spectrum of Bandpass Filtered Signal: Outer Race Fault ')
legend('Envelope Spectrum', 'BPFO Harmonics')
```



It is shown that by bandpass filtering the raw signal with the frequency band suggested by kurtogram and spectral kurtosis, the envelope spectrum analysis is able to reveal the fault signature at BPFO and its harmonics.

Batch Process

Now let's apply the algorithm to a batch of training data using a file ensemble datastore.

A limited portion of the dataset is available in the toolbox. Copy the dataset to the current folder and enable the write permission:

```
copyfile(...
    fullfile(matlabroot, 'toolbox', 'predmaint', 'predmaintdemos', ...
    'bearingFaultDiagnosis'), ...
    'RollingElementBearingFaultDiagnosis-Data-master')
fileattrib(fullfile('RollingElementBearingFaultDiagnosis-Data-master', 'train_data', '
fileattrib(fullfile('RollingElementBearingFaultDiagnosis-Data-master', 'test_data', '*
```

For the full dataset, go to this link https://github.com/mathworks/RollingElementBearingFaultDiagnosis-Data to download the entire repository as a zip file and save it in the same directory as the live script. Unzip the file using this command:

```
if exist('RollingElementBearingFaultDiagnosis-Data-master.zip', 'file')
    unzip('RollingElementBearingFaultDiagnosis-Data-master.zip')
end
```

The results in this example are generated from the full dataset. The full dataset contains a training dataset with 14 mat files (2 normal, 4 inner race fault, 7 outer race fault) and a testing dataset with 6 mat files (1 normal, 2 inner race fault, 3 outer race fault).

By assigning function handles to `ReadFcn` and `WriteToMemberFcn`, the file ensemble datastore will be able to navigate into the files to retrieve data in the desired format. For example, the MFPT data has a structure `bearing` that stores the vibration signal `gs`, sampling rate `sr`, and so on. Instead of returning the bearing structure itself the `readMFPTBearing` function is written so that file ensemble datastore returns the vibration signal `gs` inside of the `bearing` data structure.

```
fileLocation = fullfile('.', 'RollingElementBearingFaultDiagnosis-Data-master', 'train_
fileExtension = '.mat';
ensembleTrain = fileEnsembleDatastore(fileLocation, fileExtension);
ensembleTrain.ReadFcn = @readMFPTBearing;
ensembleTrain.DataVariables = ["gs", "sr", "rate", "load", "BPFO", "BPFI", "FTF", "BSF
ensembleTrain.ConditionVariables = ["Label", "FileName"];
ensembleTrain.WriteToMemberFcn = @writeMFPTBearing;
ensembleTrain.SelectedVariables = ["gs", "sr", "rate", "load", "BPFO", "BPFI", "FTF", "

ensembleTrain = 
  fileEnsembleDatastore with properties:

                 ReadFcn: @readMFPTBearing
        WriteToMemberFcn: @writeMFPTBearing
           DataVariables: [8×1 string]
    IndependentVariables: [0×0 string]
      ConditionVariables: [2×1 string]
       SelectedVariables: [10×1 string]
                ReadSize: 1
              NumMembers: 14
          LastMemberRead: [0×0 string]
                   Files: [14×1 string]


ensembleTrainTable = tall(ensembleTrain)

Starting parallel pool (parpool) using the 'local' profile ...
connected to 6 workers.

ensembleTrainTable =

  M×10 tall table

            gs               sr      rate    load     BPFO      BPFI      FTF       BSF
    __________________    _____    ____    ____    ______    ______    ______    _____

    [146484×1 double]     48828     25        0    81.125    118.88    14.838    63.91
    [146484×1 double]     48828     25       50    81.125    118.88    14.838    63.91
    [146484×1 double]     48828     25      100    81.125    118.88    14.838    63.91
    [146484×1 double]     48828     25      150    81.125    118.88    14.838    63.91
    [146484×1 double]     48828     25      200    81.125    118.88    14.838    63.91
    [585936×1 double]     97656     25      270    81.125    118.88    14.838    63.91
    [585936×1 double]     97656     25      270    81.125    118.88    14.838    63.91
    [146484×1 double]     48828     25       25    81.125    118.88    14.838    63.91
            :               :        :       :        :         :         :         :
            :               :        :       :        :         :         :         :
```

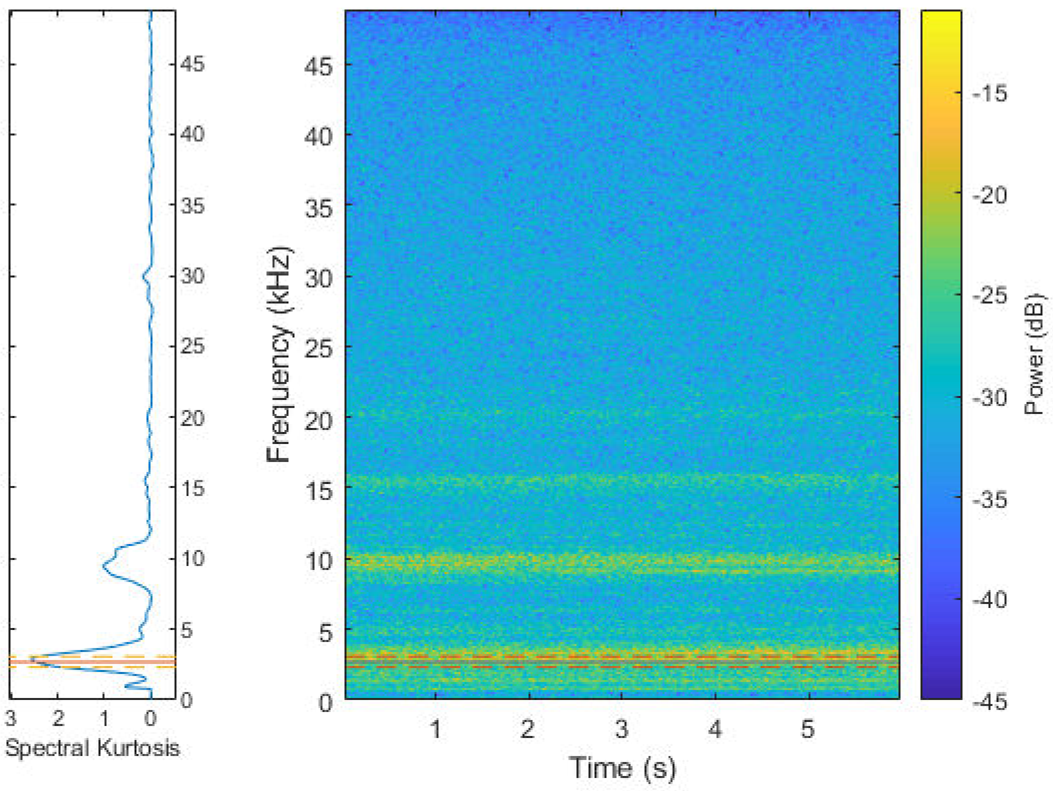
From the last section of analysis, notice that the bandpass filtered envelope spectrum amplitudes at BPFO and BPFI are two condition indicators for bearing fault diagnosis. Therefore, the next step is to extract the two condition indicators from all the training data. To make the algorithm more robust, set a narrow band (bandwidth = $10\Delta f$, where Δf is the frequency resolution of the power spectrum) around BPFO and BPFI, and then find the maximum amplitude inside this narrow band. The algorithm is contained in the `bearingFeatureExtraction` function listed below. Note that the envelope spectrum amplitudes around BPFI and BPFO are referred to as "BPFIAmplitude" and "BPFOAmplitude" in the rest of the example.

```
% To process the data in parallel, use the following code
% ppool = gcp;
% n = numpartitions(ensembleTrain, ppool);
% parfor ct = 1:n
%     subEnsembleTrain = partition(ensembleTrain, n, ct);
%     reset(subEnsembleTrain);
%     while hasdata(subEnsembleTrain)
%         bearingFeatureExtraction(subEnsembleTrain);
%     end
% end
ensembleTrain.DataVariables = [ensembleTrain.DataVariables; "BPFIAmplitude"; "BPFOAmpl
reset(ensembleTrain)
while hasdata(ensembleTrain)
    bearingFeatureExtraction(ensembleTrain)
end
```

Once the new condition indicators are added into the file ensemble datastore, specify `SelectedVariables` to read the relevant data from the file ensemble datastore, and create a feature table containing the extracted condition indicators.

```
ensembleTrain.SelectedVariables = ["BPFIAmplitude", "BPFOAmplitude", "Label"];
featureTableTrain = tall(ensembleTrain);
featureTableTrain = gather(featureTableTrain);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: 0% complete
Evaluation 0% complete
```



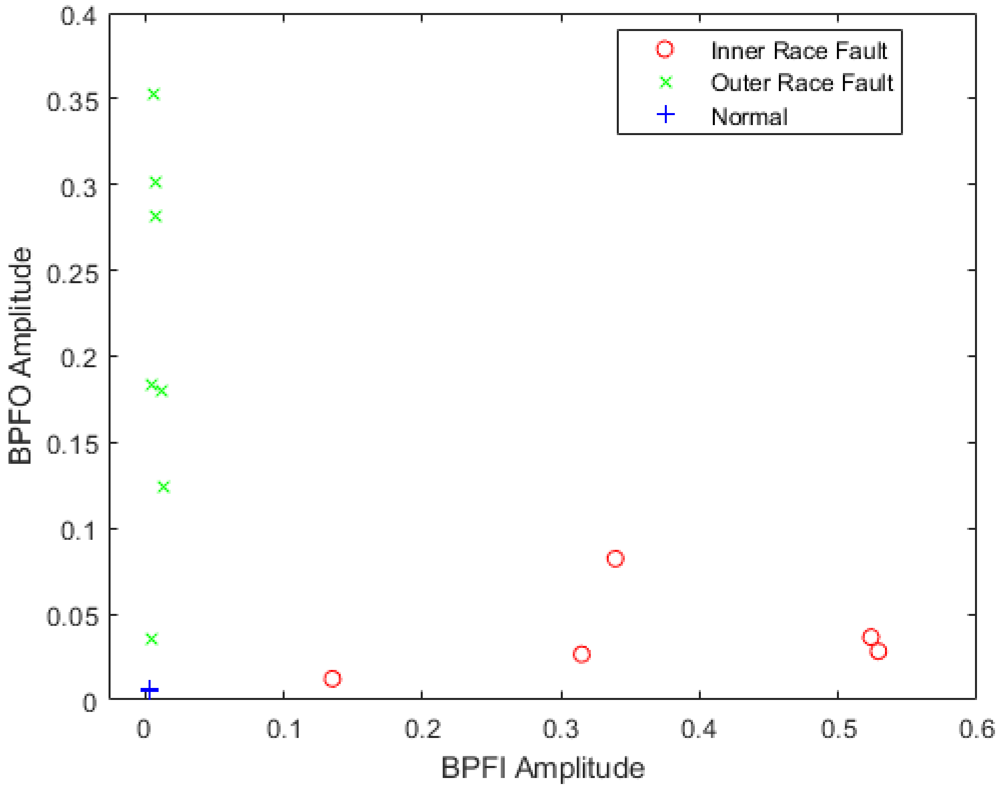
```
- Pass 1 of 1: Completed in 3 sec
Evaluation completed in 3 sec
```

```
featureTableTrain
```

```
featureTableTrain=14×3 table
    BPFIAmplitude    BPFOAmplitude          Label
    _____________    _____________    _________________

       0.33918          0.082296      "Inner Race Fault"
       0.31488          0.026599      "Inner Race Fault"
       0.52356          0.036609      "Inner Race Fault"
       0.52899          0.028381      "Inner Race Fault"
       0.13515          0.012337      "Inner Race Fault"
      0.004024           0.03574      "Outer Race Fault"
     0.0044918          0.1835      "Outer Race Fault"
     0.0074993         0.30166      "Outer Race Fault"
      0.013662         0.12468      "Outer Race Fault"
     0.0070963         0.28215      "Outer Race Fault"
     0.0060772         0.35241      "Outer Race Fault"
      0.011244         0.17975      "Outer Race Fault"
     0.0036798       0.0050208      "Normal"
       0.00359       0.0069449      "Normal"
```

Visualize the feature table that has been created.

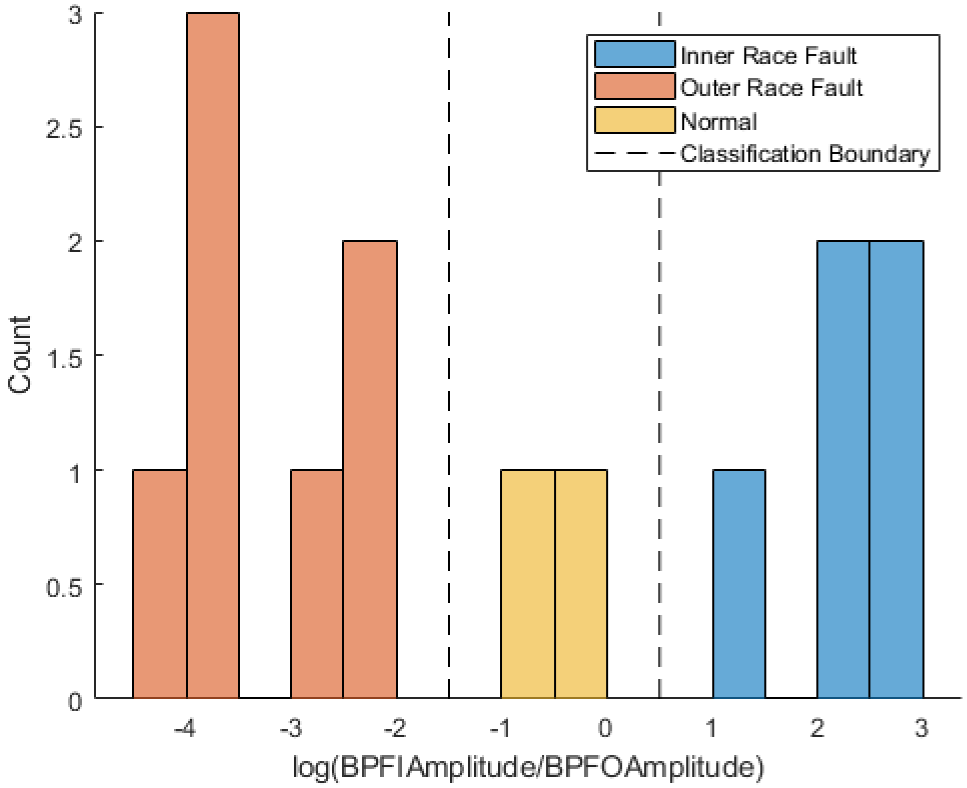
```
figure
gscatter(featureTableTrain.BPFIAmplitude, featureTableTrain.BPFOAmplitude, featureTable
xlabel('BPFI Amplitude')
ylabel('BPFO Amplitude')
```



The relative values of BPFI Amplitude and BPFO Amplitude might be an effective indicator of different fault types. Here a new feature is created, which is the log ratio of the two existing features, and is visualized in a histogram grouped by different fault types.

```
featureTableTrain.IOLogRatio = log(featureTableTrain.BPFIAmplitude./featureTableTrain.E
figure
hold on
histogram(featureTableTrain.IOLogRatio(featureTableTrain.Label=="Inner Race Fault"), 'E
histogram(featureTableTrain.IOLogRatio(featureTableTrain.Label=="Outer Race Fault"), 'E
histogram(featureTableTrain.IOLogRatio(featureTableTrain.Label=="Normal"), 'BinWidth',
plot([-1.5 -1.5 NaN 0.5 0.5], [0 3 NaN 0 3], 'k--')
hold off
ylabel('Count')
```

```
xlabel('log(BPFIAmplitude/BPFOAmplitude)')
legend('Inner Race Fault', 'Outer Race Fault', 'Normal', 'Classification Boundary')
```



The histogram shows a clear separation among the three different bearing conditions. The log ratio between the BPFI and BPFO amplitudes is a valid feature to classify bearing faults. To simplify the example, a very simple classifier is derived: if $\log\left(\frac{\text{BPFIAmplitude}}{\text{BPFOAmplitude}}\right) \le -1.5$, the bearing has an outer race fault; if

$-1.5 < \log\left(\frac{\text{BPFIAmplitude}}{\text{BPFOAmplitude}}\right) \leq 0.5$, the bearing is normal; and if

$\log\left(\frac{\text{BPFIAmplitude}}{\text{BPFOAmplitude}}\right) > 0.5$, the bearing has an inner race fault.

Validation using Test Data Sets

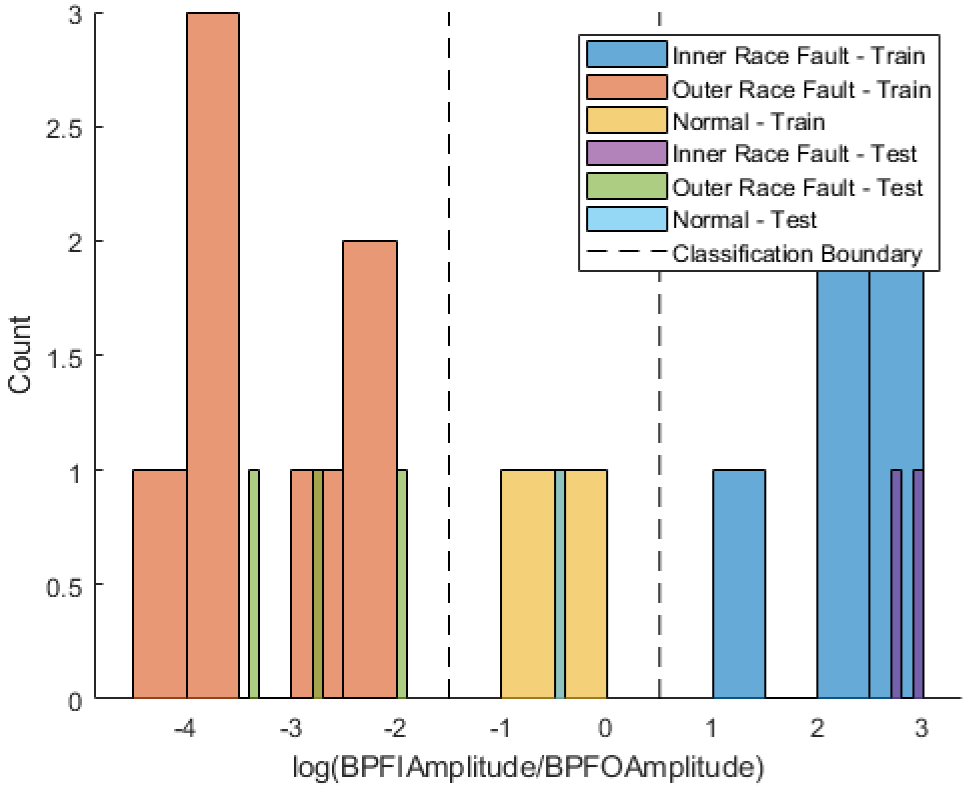
Now, let's apply the workflow to a test data set and validate the classifier obtained in the last section. Here the test data contains 1 normal data set, 2 inner race fault data sets, and 3 outer race fault data sets.

```
fileLocation = fullfile('.', 'RollingElementBearingFaultDiagnosis-Data-master', 'test_c
fileExtension = '.mat';
ensembleTest = fileEnsembleDatastore(fileLocation, fileExtension);
ensembleTest.ReadFcn = @readMFPTBearing;
ensembleTest.DataVariables = ["gs", "sr", "rate", "load", "BPFO", "BPFI", "FTF", "BSF"]
ensembleTest.ConditionVariables = ["Label", "FileName"];
ensembleTest.WriteToMemberFcn = @writeMFPTBearing;
ensembleTest.SelectedVariables = ["gs", "sr", "rate", "load", "BPFO", "BPFI", "FTF", "B
```

```
ensembleTest = 
  fileEnsembleDatastore with properties:

                 ReadFcn: @readMFPTBearing
        WriteToMemberFcn: @writeMFPTBearing
           DataVariables: [8×1 string]
    IndependentVariables: [0×0 string]
      ConditionVariables: [2×1 string]
       SelectedVariables: [10×1 string]
                ReadSize: 1
              NumMembers: 6
          LastMemberRead: [0×0 string]
                   Files: [6×1 string]
```

```
ensembleTest.DataVariables = [ensembleTest.DataVariables; "BPFIAmplitude"; "BPFOAmplitu
reset(ensembleTest)
while hasdata(ensembleTest)
    bearingFeatureExtraction(ensembleTest)
end

ensembleTest.SelectedVariables = ["BPFIAmplitude", "BPFOAmplitude", "Label"];
featureTableTest = tall(ensembleTest);
featureTableTest = gather(featureTableTest);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1 sec
Evaluation completed in 1 sec

featureTableTest.IOLogRatio = log(featureTableTest.BPFIAmplitude./featureTableTest.BPF

figure
hold on
histogram(featureTableTrain.IOLogRatio(featureTableTrain.Label=="Inner Race Fault"), 'B
histogram(featureTableTrain.IOLogRatio(featureTableTrain.Label=="Outer Race Fault"), 'B
histogram(featureTableTrain.IOLogRatio(featureTableTrain.Label=="Normal"), 'BinWidth',

histogram(featureTableTest.IOLogRatio(featureTableTest.Label=="Inner Race Fault"), 'Bin
histogram(featureTableTest.IOLogRatio(featureTableTest.Label=="Outer Race Fault"), 'Bin
histogram(featureTableTest.IOLogRatio(featureTableTest.Label=="Normal"), 'BinWidth', 0
plot([-1.5 -1.5 NaN 0.5 0.5], [0 3 NaN 0 3], 'k--')
hold off
ylabel('Count')
xlabel('log(BPFIAmplitude/BPFOAmplitude)')
legend('Inner Race Fault - Train', 'Outer Race Fault - Train', 'Normal - Train', ...
    'Inner Race Fault - Test', 'Outer Race Fault - Test', 'Normal - Test', ...
    'Classification Boundary')
```



The log ratio of BPFI and BPFO amplitudes from test data sets shows consistent distribution with the log ratio from training data sets. The naive classifier obtained in the last section achieved perfect accuracy on the test data set.

It should be noted that single feature is usually not enough to get a classifier that generalizes well. More sophisticated classifiers could be obtained by dividing the data into multiple pieces (to create more data points), extract multiple diagnosis related features, select a subset of features by their importance ranks, and train various classifiers using the Classification Learner App in Statistics & Machine Learning Toolbox. For more details of this workflow, please refer to the example "Using Simulink to generate fault data".

Summary

This example shows how to use kurtogram, spectral kurtosis and envelope spectrum to identify different types of faults in rolling element bearings. The algorithm is then applied to a batch of data sets in disk, which helped show that the amplitudes of bandpass filtered envelope spectrum at BPFI and BPFO are two important condition indicators for bearing diagnostics.

References

[1] Randall, Robert B., and Jerome Antoni. "Rolling element bearing diagnostics—a tutorial." *Mechanical Systems and Signal Processing* 25.2 (2011): 485-520.

[2] Antoni, Jerome. "Fast computation of the kurtogram for the detection of transient faults." *Mechanical Systems and Signal Processing* 21.1 (2007): 108-124.

[3] Antoni, Jérôme. "The spectral kurtosis: a useful tool for characterising non-stationary signals." *Mechanical Systems and Signal Processing* 20.2 (2006): 282-307.

[4] Bechhoefer, E. "Condition Based Maintenance Fault Database for Testing Diagnostics and Prognostic Algorithms." (2013). https://mfpt.org/fault-data-sets/

Helper Functions

```
function bearingFeatureExtraction(ensemble)
% Extract condition indicators from bearing data
data = read(ensemble);
x = data.gs{1};
fs = data.sr;

% Critical Frequencies
BPFO = data.BPFO;
BPFI = data.BPFI;

level = 9;
[~, ~, ~, fc, ~, BW] = kurtogram(x, fs, level);

% Bandpass filtered Envelope Spectrum
[pEnvpBpf, fEnvBpf] = envspectrum(x, fs, 'FilterOrder', 200, 'Band', [max([fc-BW/2 0])
deltaf = fEnvBpf(2) - fEnvBpf(1);

BPFIAmplitude = max(pEnvpBpf((fEnvBpf > (BPFI-5*deltaf)) & (fEnvBpf < (BPFI+5*deltaf))
BPFOAmplitude = max(pEnvpBpf((fEnvBpf > (BPFO-5*deltaf)) & (fEnvBpf < (BPFO+5*deltaf))
writeToLastMemberRead(ensemble, table(BPFIAmplitude, BPFOAmplitude, 'VariableNames', {
end
```

See Also

`fileEnsembleDatastore`

More About

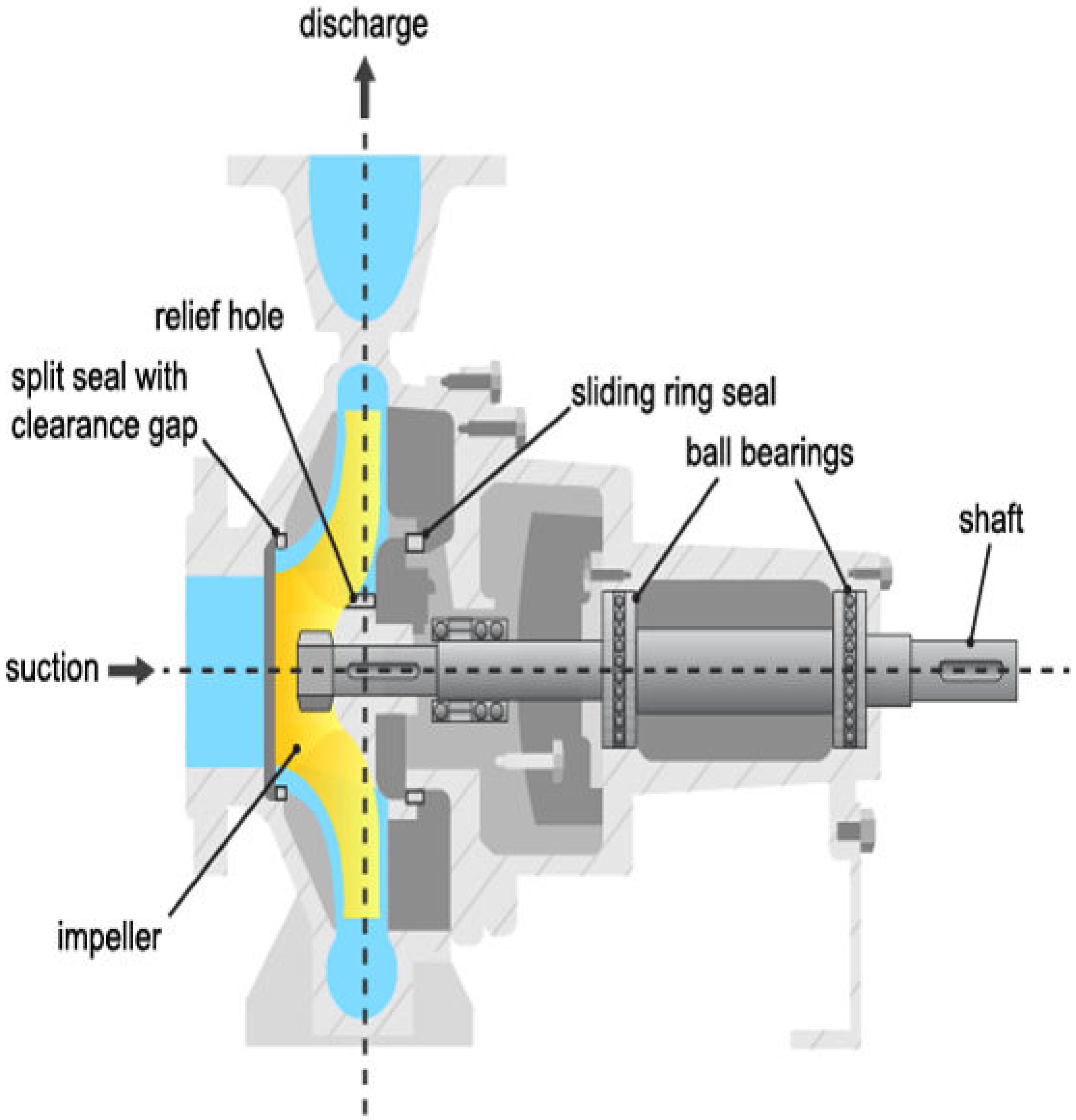
- “Data Ensembles for Condition Monitoring and Predictive Maintenance” on page 1-2
- “Decision Models for Fault Detection and Diagnosis” on page 4-2

Fault Diagnosis of Centrifugal Pumps Using Steady State Experiments

This example shows a model based approach for detection and diagnosis of different types of faults that occur in a pumping system. The example follows the centrifugal pump analysis presented in the Fault Diagnosis Applications book by Rolf Isermann [1].

Pump Supervision and Fault Detection

Pumps are essential equipment in many industries including power and chemical, mineral and mining, manufacturing, heating, air conditioning and cooling. Centrifugal pumps are used to transport fluids by the conversion of rotational kinetic energy to the hydrodynamic energy of the fluid flow. The rotational energy typically comes from a combustion engine or electric motor. The fluid enters the pump impeller along or near to the rotating axis and is accelerated by the impeller, flowing radially outward into a diffuser.


discharge
relief hole
split seal with clearance gap
sliding ring seal
ball bearings
shaft
suction
impeller

Pumps experience damage to their hydraulic or mechanical components. The most frequently faulting components are the sliding ring seals and the ball bearings, although failures to other components including the driving motor, impeller blades and sliding bearings are also not uncommon. The table below lists the most common types of faults.

- **Cavitation:** Development of vapor bubbles inside the fluid if static pressure falls below vapor pressure. Bubbles collapse abruptly leading to damage at the blade wheels.
- **Gas in fluid:** A pressure drop leads to dissolved gas in the fluid. A separation of gas and liquid and lower head results.
- **Dry Run:** Missing fluid leads to lack of cooling and overheating of bearing. Important for starting phase.
- **Erosion:** Mechanical damage to the walls because of hard particles or cavitation
- **Corrosion:** Damage by aggressive fluids
- **Bearing wear:** Mechanical damage through fatigue and metal friction, generation of pitting and tears
- **Plugging of relief bore holes:** Leads to overloading/damage of axial bearings
- **Plugging of sliding ring seals:** Leads to higher friction and smaller efficiency
- **Increase of split seals:** Leads to loss of efficiency
- **Deposits:** Deposits of organic material or through chemical reactions at the rotor entrance or outlet reduce efficiency and increase temperature.
- **Oscillations:** Rotor imbalance through damage or deposits at the rotor. Can cause bearing damage.

Available Sensors

The following signals are typically measured:

- Pressure difference between the inlet and outlet Δp
- Rotational speed ω
- Motor torque M_{mot} and pump torque M_p
- Fluid discharge (flow) rate at the pump outlet Q
- Driving motor current, voltage, temperature (not considered here)

- Fluid temperature, sediments (not considered here)

Mathematical Models of Pump and Pipe System

A torque M applied to the rotor of a radial centrifugal pump leads to a rotational speed ω and transmits a momentum increase of the pump fluid from the rotor inlet of a smaller radius to the rotor outlet of a larger radius. Euler's turbine equations yields the relationship between the pressure differential Δp, speed ω and fluid discharge rate (flow rate) Q:

$$H_{th} = h_1\omega^2 - h_2\omega Q$$

where $H_{th} = \frac{\Delta p}{\rho g}$ is the theoretical (ideal; without losses) pump head measured in meters and h_1, h_2 are proportionality constants. When accounting for a finite number of impeller blades, friction losses and impact losses due to non-tangential flow, the real pump head is given by:

$$H = h_{nn}\omega^2 - h_{nv}\omega Q - h_{vv}Q^2$$

where h_{nn}, h_{nv} and h_{vv} are proportionality constants to be treated as model parameters. The corresponding pump torque is:

$$M_P = \rho g(h_{nn}\omega Q - h_{nv}Q^2 - h_{vv}\frac{Q^3}{\omega})$$

The mechanical parts of the motor and the pump cause the speed to increase when torque is applied according to:

$$J_P\frac{d\omega(t)}{dt} = M_{mot}(t) - M_P(t) - M_f(t)$$

where J_P is the ratio of inertia of the motor and the pump, and M_f is the frictional torque consisting of Coulomb friction M_{f0} and viscous friction $M_{f1}\omega(t)$ according to:

$$M_f(t) = M_{f0}\ sign\ \ \omega(t) + M_{f1}\omega(t)$$

The pump is connected to a piping system that transports the fluid from a lower storage tank to an upper one. The momentum balance equation yields:

$$H(t) = a_F \frac{dQ(t)}{dt} + h_{rr}Q^2(t) + H_{static}$$

where h_{rr} is a resistance coefficient of the pipe, $a_F = \frac{l}{gA}$ with pipe length l and cross-sectional area A, and H_{static} is the height of the storage over the pump. The model parameters h_{nn}, h_{nv}, h_{vv} are either known from physics or can be estimated by fitting the measured sensor signals to the inputs/outputs of the model. The type of the model used may depend upon the operating conditions under which the pump is run. For example, full nonlinear model of the pump-pipe system may not be required if the pump is always run at a constant angular speed.

Fault Detection Techniques

Faults can be detected by examining certain features extracted from the measurements and comparing them to known thresholds of acceptable behavior. The detectability and isolability of different faults depends upon the nature of the experiment and availability of measurements. For example, a constant-speed analysis with pressure measurements only can detect faults causing large pressure changes. Furthermore, it cannot reliably assess the cause of the failure. However, a multi-speed experiment with measurements of pressure differential, motor torque and flow rate can detect and isolate many sources of faults such as those originating from gas enclosures, dry run, large deposits, motor defects etc.

A model based approach employs the following techniques:

1. Parameter estimation: Using the measurements from the healthy (nominal) operation of the machine, the parameters of the model are estimated and their uncertainty is quantified. The test system measurements are then used to re-estimate the parameter values and the resulting estimates are compared against their nominal values. This technique is the main topic of this example.
2. Residue generation: A model is trained as before for a healthy machine. The output of the model is compared against the measured observations from a test system and a

residual signal is computed. This signal is analyzed for its magnitude, variance and other properties to detect faults. A large number of residues may be designed and employed to distinguish different sources of faults. This technique is discussed in the *Fault Diagnosis of Centrifugal Pumps using Residual Analysis* example.

Constant Speed Experimentation: Fault Analysis by Parameter Estimation

A common practice for pump calibration and supervision is to run it at a constant speed and record the pump's static head and fluid discharge rate. By changing the valve position in the piping system, the fluid discharge volume (GPM) is regulated. An increase in discharge rate causes the pump head to decrease. The pump's measured head characteristics can be compared against the manufacturer-supplied values. Any differences would indicate possibility of faults. The measurements for delivery head and flow discharge rate were obtained by simulations of a pump-pipe system model in Simulink.

At a nominal speed of 2900 RPM, the ideal pump head characteristics for a healthy pump supplied by the manufacturer are as shown.

```
load PumpCharacteristicsData Q0 H0 M0 % manufacturer supplied data for pump's delivery
figure
plot(Q0, H0, '--');
xlabel('Discharge Rate Q (m^3/h)')
ylabel('Pump Head (m)')
title('Pump Delivery Head Characteristics at 2900 RPM')
grid on
legend('Healthy pump')
```

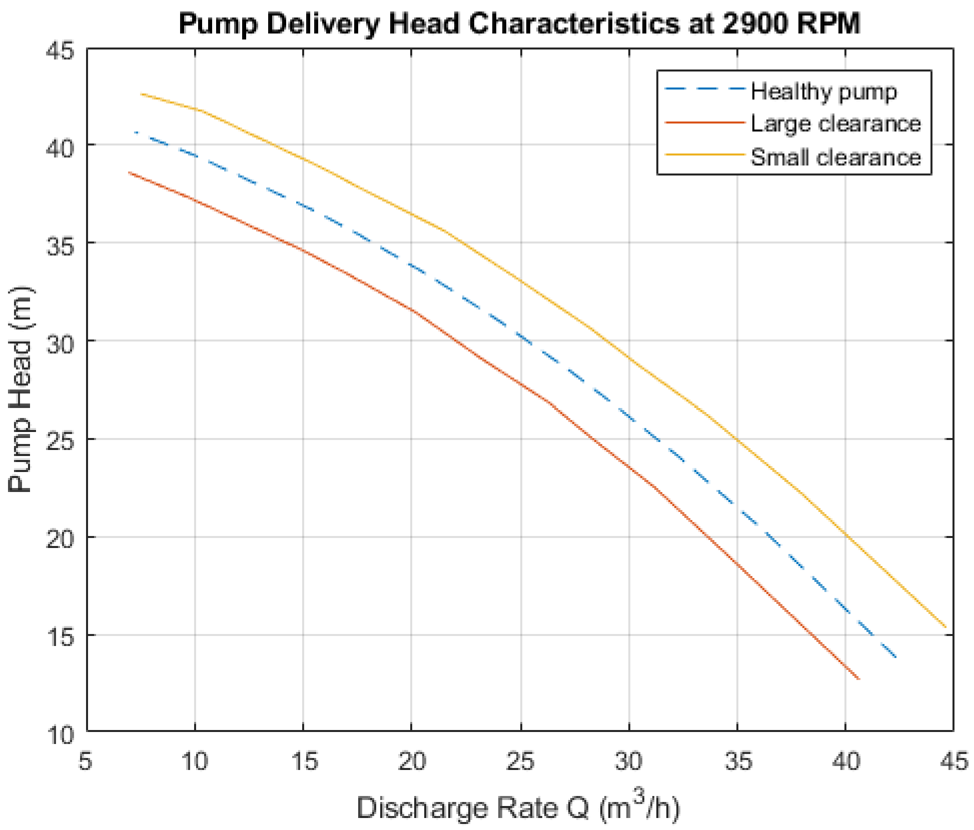
The faults that cause a discernible change in pump characteristics are:

1. Wear at the clearance gap
2. Wear at the impeller outlet
3. Deposits at the impeller outlet

For analyzing the faulty pumps, the speed, torque and flow rate measurements were collected for pumps affected by different faults. For example when the fault introduced is in the clearance ring, the measured head characteristics for pumps show a clear shift in the characteristic curve.

```
load PumpCharacteristicsData Q1 H1 M1  % signals measured for a pump with a large clea
hold on
plot(Q1, H1);
```

```
load PumpCharacteristicsData Q2 H2 M2  % signals measured for a pump with a small clea
plot(Q2, H2);
legend('Healthy pump','Large clearance','Small clearance')
hold off
```



Similar changes can be seen in torque-flow characteristics and for other fault types.

For automation of fault diagnosis, you turn the observed changes to quantitative information. A reliable way to do so is to fit a parameterized curve to the head-flow characteristic data plotted above. Using the pump-pipe dynamics governing equations, and using a simplified torque relationship, the following equations are obtained:

$$H \approx h_{nn}\omega^2 - h_{nv}\omega Q - h_{vv}Q^2$$

$$M_p \approx k_0\omega Q - k_1 Q^2 + k_2\omega^2$$

$h_{nn}, h_{nv}, h_{vv}, k_0, k_1, k_2$ are the parameters to be estimated. If you measure ω, Q, H and M_P, the parameter can be estimated by linear least squares. These parameters are the *features* that can be used to develop a fault detection and diagnosis algorithm.

Preliminary Analysis: Comparing parameter values

Compute and plot the parameter values estimated for the above 3 curves. Use the measured values of Q, H and M_P as data and $\omega = 2900\ \ RPM$ as the nominal pump speed.

```
w = 2900; % RPM
% Healthy pump
[hnn_0, hnv_0, hvv_0, k0_0, k1_0, k2_0] = linearFit(0, {w, Q0, H0, M0});
% Pump with large clearance
[hnn_1, hnv_1, hvv_1, k0_1, k1_1, k2_1] = linearFit(0, {w, Q1, H1, M1});
% Pump with small clearance
[hnn_2, hnv_2, hvv_2, k0_2, k1_2, k2_2] = linearFit(0, {w, Q2, H2, M2});
X = [hnn_0 hnn_1 hnn_2; hnv_0  hnv_1  hnv_2; hvv_0  hvv_1  hvv_2]';
disp(array2table(X,'VariableNames',{'hnn','hnv','hvv'},...
    'RowNames',{'Healthy','Large Clearance', 'Small Clearance'}))
```

```
                           hnn           hnv           hvv
                        __________    __________    _________

    Healthy             5.1164e-06    8.6148e-05     0.010421
    Large Clearance      4.849e-06     8.362e-05     0.011082
    Small Clearance     5.3677e-06    8.4764e-05    0.0094656
```

```
Y = [k0_0 k0_1 k0_2; k1_0  k1_1  k1_2; k2_0  k2_1  k2_2]';
disp(array2table(Y,'VariableNames',{'k0','k1','k2'},...
    'RowNames',{'Healthy','Large Clearance', 'Small Clearance'}))
```

```
                            k0            k1           k2
                        __________    ________    __________

    Healthy             0.00033347    0.016535    2.8212e-07
    Large Clearance     0.00031571    0.016471    3.0285e-07
    Small Clearance     0.00034604    0.015886    2.6669e-07
```

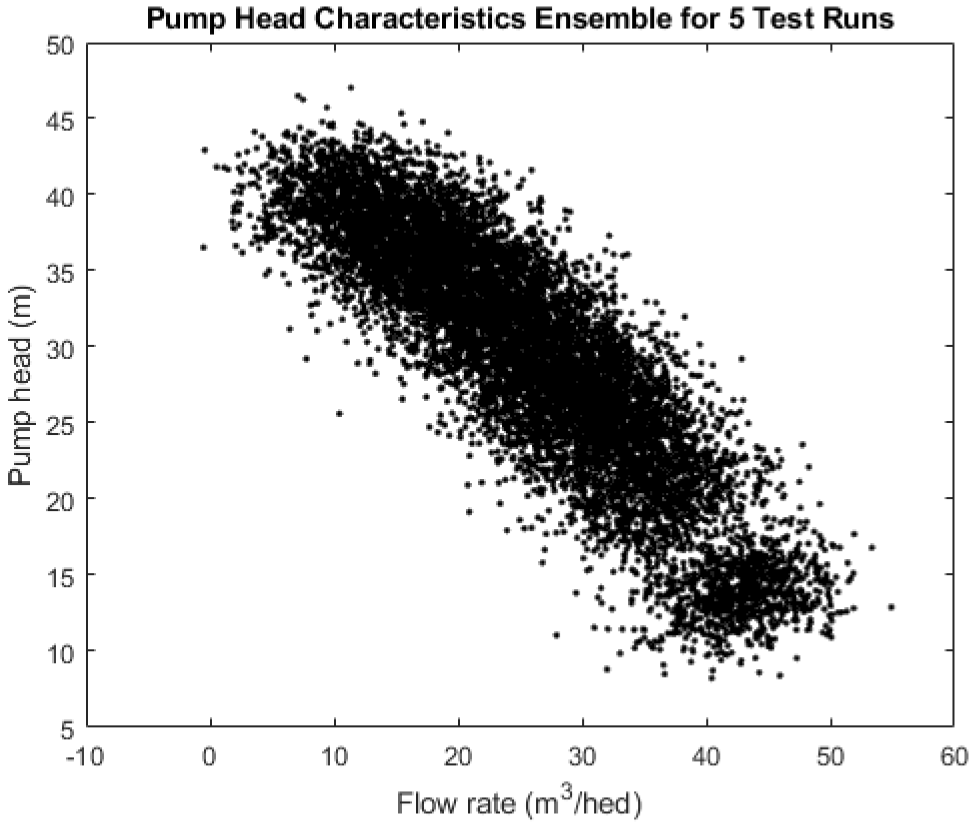
The tables show that h_{nn} and k_0 values reduce when clearance gap is large while they are larger than nominal values for small clearance. On the other hand, h_{vv} and k_2 values increase for large clearance gap and decrease for small gap. The dependence of h_{nv} and k_1 on clearance gap is less clear.

Incorporating Uncertainty

The preliminary analysis showed how parameter changes can indicate fault. However, even for healthy pumps there are variations in measurements owing to the measurement noise, fluid contamination and viscosity changes and the slip-torque characteristics of the motor running the pump. These measurement variations introduce uncertainty in the parameter estimates.

Collect 5 sets of measurements from a pump operating under no-fault condition by running it at 2900 RPM for 10 discharge throttle valve positions.

```
load FaultDiagnosisData HealthyEnsemble
H = cellfun(@(x)x.Head,HealthyEnsemble,'uni',0);
Q = cellfun(@(x)x.Discharge,HealthyEnsemble,'uni',0);
plot(cat(2,Q{:}),cat(2,H{:}),'k.')
title('Pump Head Characteristics Ensemble for 5 Test Runs')
xlabel('Flow rate (m^3/hed)')
ylabel('Pump head (m)')
```



The plot shows variation in characteristics even for a healthy pump under realistic conditions. These variations have to be taken into consideration for making the fault diagnosis reliable. The next sections discuss fault detection and isolation techniques for noisy data.

Anomaly Detection

In many situations, the measurements of only the healthy machines are available. In that case, a statistical description of the healthy state, encapsulated by mean value and covariance of the parameter vector, can be created using available measurements. The measurements of the test pump can be compared against the nominal statistics to test whether it is plausible that the test pump is a healthy pump. A faulty pump is expected to detected as an anomaly in a view of detection features.

Estimate mean and covariance of pump head and torque parameters.

```
load FaultDiagnosisData HealthyEnsemble
[HealthyTheta1, HealthyTheta2] = linearFit(1, HealthyEnsemble);
meanTheta1 = mean(HealthyTheta1,1);
meanTheta2 = mean(HealthyTheta2,1);
covTheta1  = cov(HealthyTheta1);
covTheta2  = cov(HealthyTheta2);
```

Visualize the parameter uncertainty as 74% confidence regions, which corresponds to 2 standard deviations ($\sqrt{chi2inv(0.74,3)} \approx 2$). See helper function `helperPlotConfidenceEllipsoid` for details.

```
% Confidence ellipsoid for pump head parameters
f = figure;
f.Position(3) = f.Position(3)*2;
subplot(121)
helperPlotConfidenceEllipsoid(meanTheta1,covTheta1,2,0.6);
xlabel('hnn')
ylabel('hnv')
zlabel('hvv')
title('2-sd Confidence Ellipsoid for Pump Head Parameters')
hold on

% Confidence ellipsoid for pump torque parameters
subplot(122)
helperPlotConfidenceEllipsoid(meanTheta2,covTheta2,2,0.6);
xlabel('k0')
ylabel('k1')
zlabel('k2')
title('2-sd Confidence Ellipsoid for Pump Torque Parameters')
hold on
```

The grey ellipsoids show confidence regions of healthy pump parameters. Load unlabeled test data for comparison against the healthy region.

```
load FaultDiagnosisData TestEnsemble
```

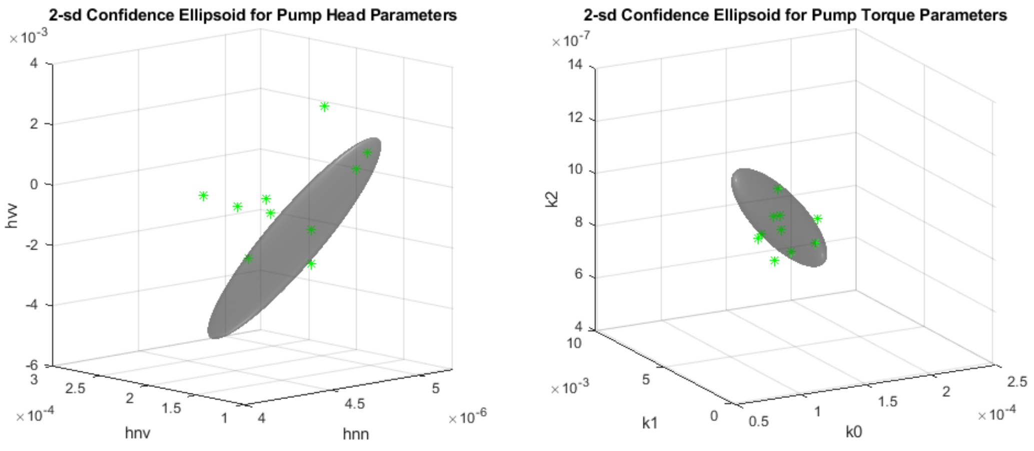
TestEnsemble contains a set of pump speed, torque, head and flow rate measurements at various valve positions. All measurements contain gap clearance fault of different magnitudes.

```
% Test data preview
disp(TestEnsemble{1}(1:5,:)) % first 5 measurement rows from the first ensemble member
```

```
    Time        Run    ValvePosition    Speed      Head     Discharge    Torque 
  __________    ___    _____________    ______    ______    _________    _______

  180 sec        1          10          3034.6    12.367     35.339      0.35288
  180.1 sec      1          10          2922.1    9.6762     36.556       4.6953
  180.2 sec      1          10          2636.1    11.168     36.835       9.8898
  180.3 sec      1          10          2717.4    10.562      40.22      -12.598
  180.4 sec      1          10          3183.7     10.55     40.553       14.672
```

Compute test parameters. See helper function `linearFit`.

```
% TestTheta1: pump head parameters
% TestTheta2: pump torque parameters
[TestTheta1,TestTheta2] = linearFit(1, TestEnsemble);
subplot(121)
plot3(TestTheta1(:,1),TestTheta1(:,2),TestTheta1(:,3),'g*')
view([-42.7 10])
subplot(122)
plot3(TestTheta2(:,1),TestTheta2(:,2),TestTheta2(:,3),'g*')
view([-28.3 18])
```



Each green star marker is contributed by one test pump. The markers that are outside the confidence bounds can be treated as outliers while those inside are either from a healthy pump or escaped detection. Note that a marker from a particular pump may be marked as anomalous in the pump head view but not in the pump torque view. This could be owing to different sources of faults being detected by these views, or the underlying reliability of pressure and torque measurements.

Quantifying Anomaly Detection Using Confidence Regions

In this section a way of utilizing the confidence region information for detection and assessing severity of faults is discussed. The technique is to compute the "distance" of a test sample from the mean or median of the healthy region distribution. The distance must be relative to the normal "spread" of the healthy parameter data represented by its covariance. The function MAHAL computes the Mahalanobis distance of test samples from the distribution of a reference sample set (the healthy pump parameter set here):

```
ParDist1 = mahal(TestTheta1, HealthyTheta1);  % for pump head parameters
```

If you assume 74% confidence bound (2 standard deviations) as acceptable variation for healthy data, any values in ParDist1 that are greater than 2^2 = 4 should be tagged as anomalous and hence indicative of faulty behavior.

Add the distance values to the plot. The red lines mark the anomalous test samples. See helper function `helperAddDistanceLines`.

```
Threshold = 2;
disp(table((1:length(ParDist1))',ParDist1, ParDist1>Threshold^2,...
    'VariableNames',{'PumpNumber','SampleDistance','Anomalous'}))

    PumpNumber    SampleDistance    Anomalous
    __________    ______________    _________

         1            58.874          true   
         2            24.051          true   
         3             6.281          true   
         4            3.7179          false  
         5             13.58          true   
         6            3.0723          false  
         7            2.0958          false  
         8            4.7127          true   
         9            26.829          true   
        10           0.74682          false  
```

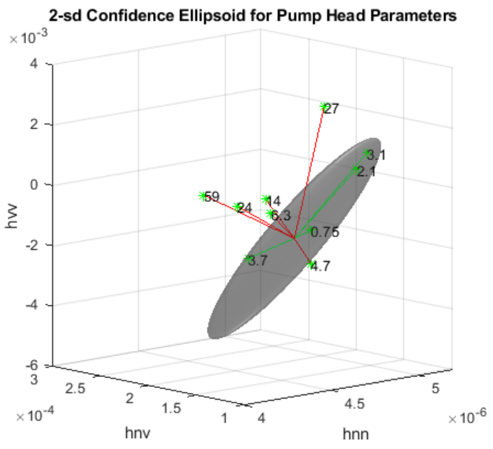
```
helperAddDistanceLines(1, ParDist1, meanTheta1, TestTheta1, Threshold);
```

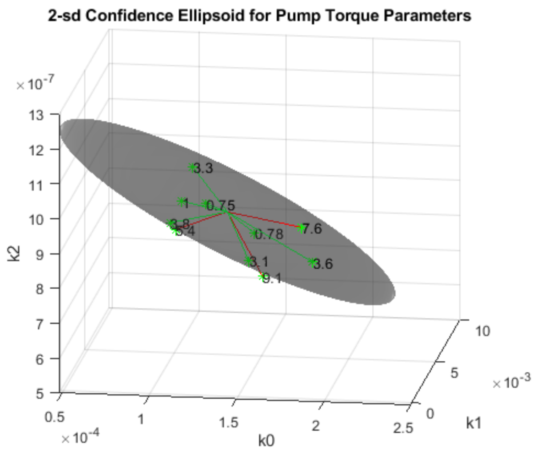
Similarly for pump torque:

```
ParDist2 = mahal(TestTheta2, HealthyTheta2);  % for pump torque parameters
disp(table((1:length(ParDist2))',ParDist2, ParDist2>Threshold^2,...
    'VariableNames',{'PumpNumber','SampleDistance','Anomalous'}))

    PumpNumber    SampleDistance    Anomalous
    __________    ______________    _________
```

```
         1            9.1381          true 
         2            5.4249          true 
         3            3.0565          false
         4             3.775          false
         5           0.77961          false
         6            7.5508          true 
         7            3.3368          false
         8           0.74834          false
         9            3.6478          false
        10            1.0241          false
```

```
helperAddDistanceLines(2, ParDist2, meanTheta2, TestTheta2, Threshold);
view([8.1 17.2])
```





The plots now not only show detection of the anomalous samples but also quantify their severity.

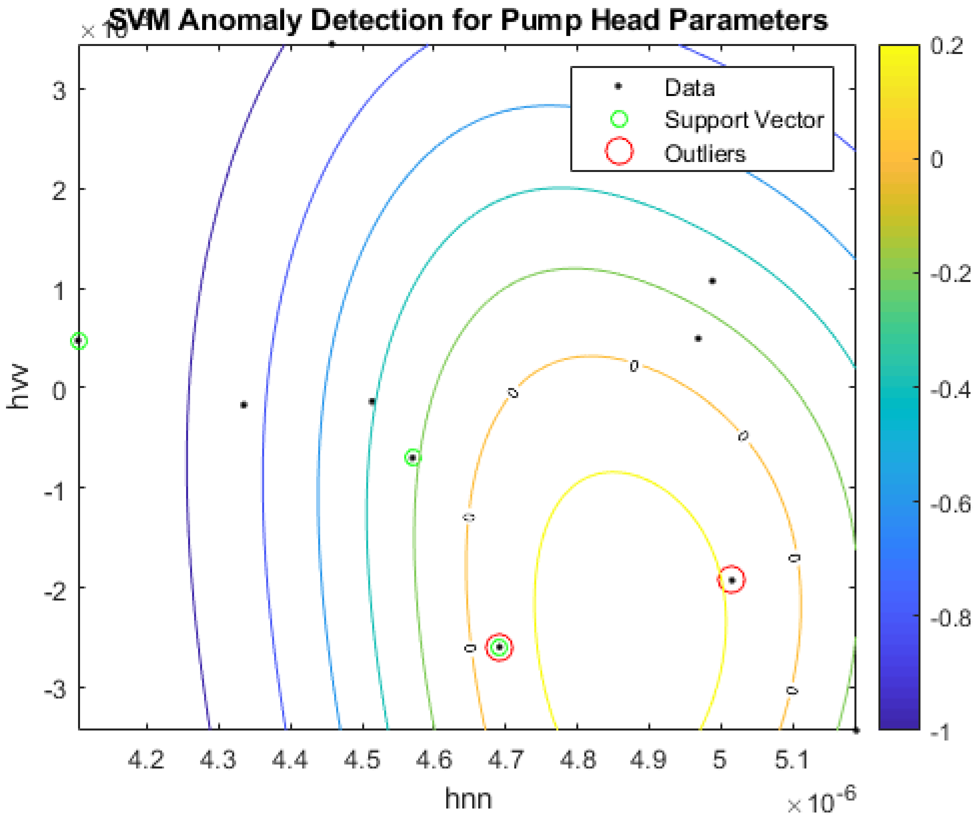
Quantifying Anomaly Detection Using One-Class Classifier

Another effective technique to flag anomalies is to build a one-class classifier for the healthy parameter dataset. Train an SVM classifier using the healthy pump parameter data. Since there are no fault labels utilized, treat all samples as coming from the same (healthy) class. Since changes in parameters h_{nn} and h_{vv} are most indicative of potential faults, use only these parameters for training the SVM classifier.

```
nc = size(HealthyTheta1,1);
rng(2)  % for reproducibility
SVMOneClass1 = fitcsvm(HealthyTheta1(:,[1 3]),ones(nc,1),...
    'KernelScale','auto',...
    'Standardize',true,...
    'OutlierFraction',0.0455);
```

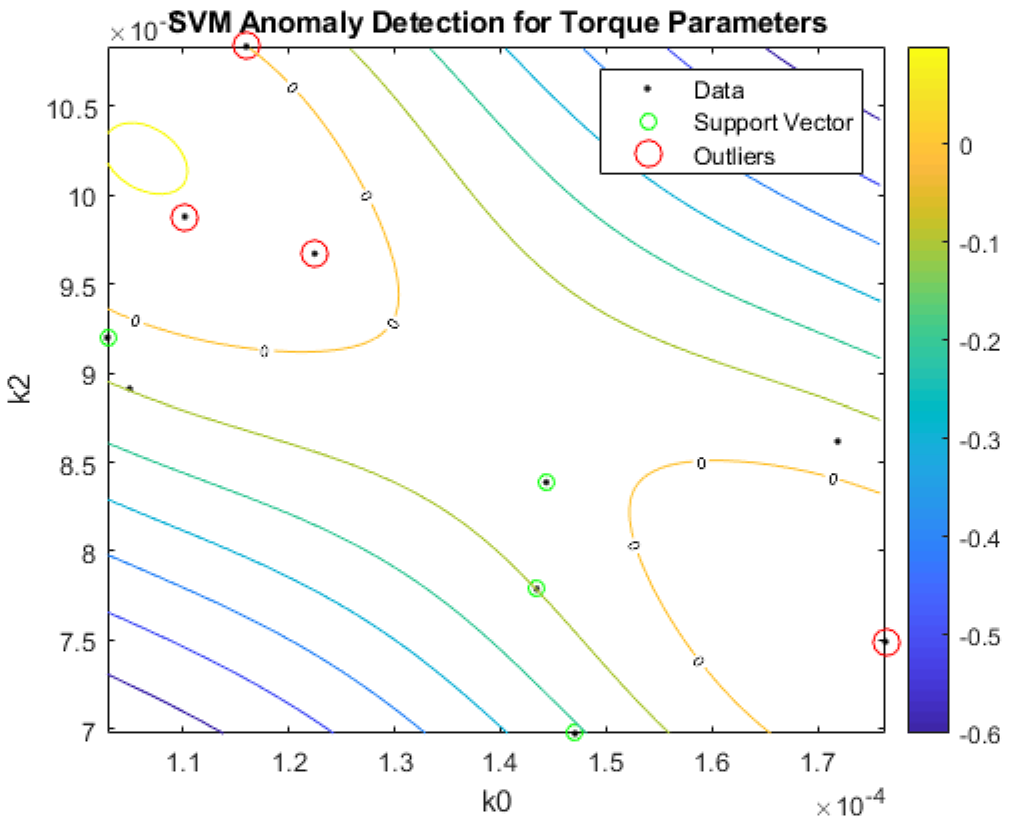
Plot the test observations and the decision boundary. Flag the support vectors and potential outliers. See helper function `helperPlotSVM`.

```
figure
helperPlotSVM(SVMOneClass1,TestTheta1(:,[1 3]))
title('SVM Anomaly Detection for Pump Head Parameters')
xlabel('hnn')
ylabel('hvv')
```



The boundary separating the outliers from the rest of the data occurs where the contour value is 0; this is the level curve marked with "0" in the plot. The outliers are marked with red circles. A similar analysis can be performed for torque parameters.

```
SVMOneClass2 = fitcsvm(HealthyTheta2(:,[1 3]),ones(nc,1),...
    'KernelScale','auto',...
    'Standardize',true,...
    'OutlierFraction',0.0455);
figure
helperPlotSVM(SVMOneClass2,TestTheta2(:,[1 3]))
title('SVM Anomaly Detection for Torque Parameters')
xlabel('k0')
ylabel('k2')
```



A similar analysis can be carried out for detecting other kinds of faults such as wear or deposits at impeller outlet as discussed next in the context of fault isolation.

Fault Isolation Using Steady-State Parameters as Features

If the information on the type of fault(s) in the test system is available, it can be used to create algorithms that not only detect faults but also indicate their type.

A. Distinguishing Gap Clearance Faults by Likelihood Ratio Tests

Changes in gap clearance can be split into two types - smaller than expected gap (yellow line in the pump head characteristic plot) and larger than expected gap (red line). Load pump test datasets containing gap clearance faults where the nature of fault (large or small) is known in advance. Use these fault labels to perform 3-way classification among the following modes:

- Mode 1: Normal gap clearance (healthy behavior)
- Mode 2: Large gap clearance
- Mode 3: Small gap clearance

```
load LabeledGapClearanceData HealthyEnsemble LargeGapEnsemble SmallGapEnsemble
```

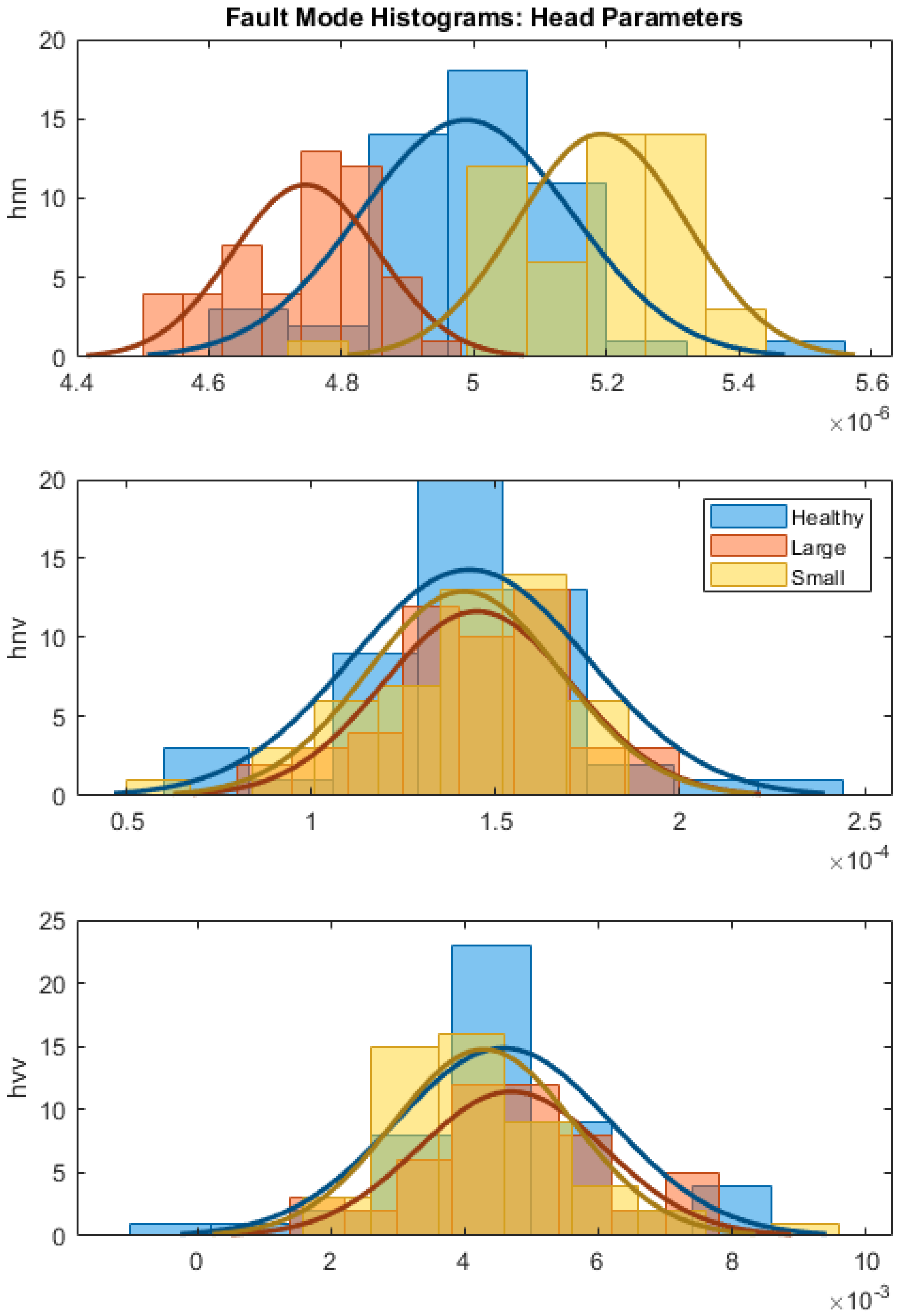
The ensembles contain data from 50 independent experiments. Fit steady-state linear models as before to parameterize the pump head and torque data.

```
[HealthyTheta1, HealthyTheta2] = linearFit(1,HealthyEnsemble);
[LargeTheta1, LargeTheta2]     = linearFit(1,LargeGapEnsemble);
[SmallTheta1, SmallTheta2]     = linearFit(1,SmallGapEnsemble);
```

Plot the parameter histograms to check if there is separability among the 3 modes. The function `histfit` is used to plot the histograms and the corresponding fitted normal distribution curves. See helper function `helperPlotHistogram`.

Pump head parameters:

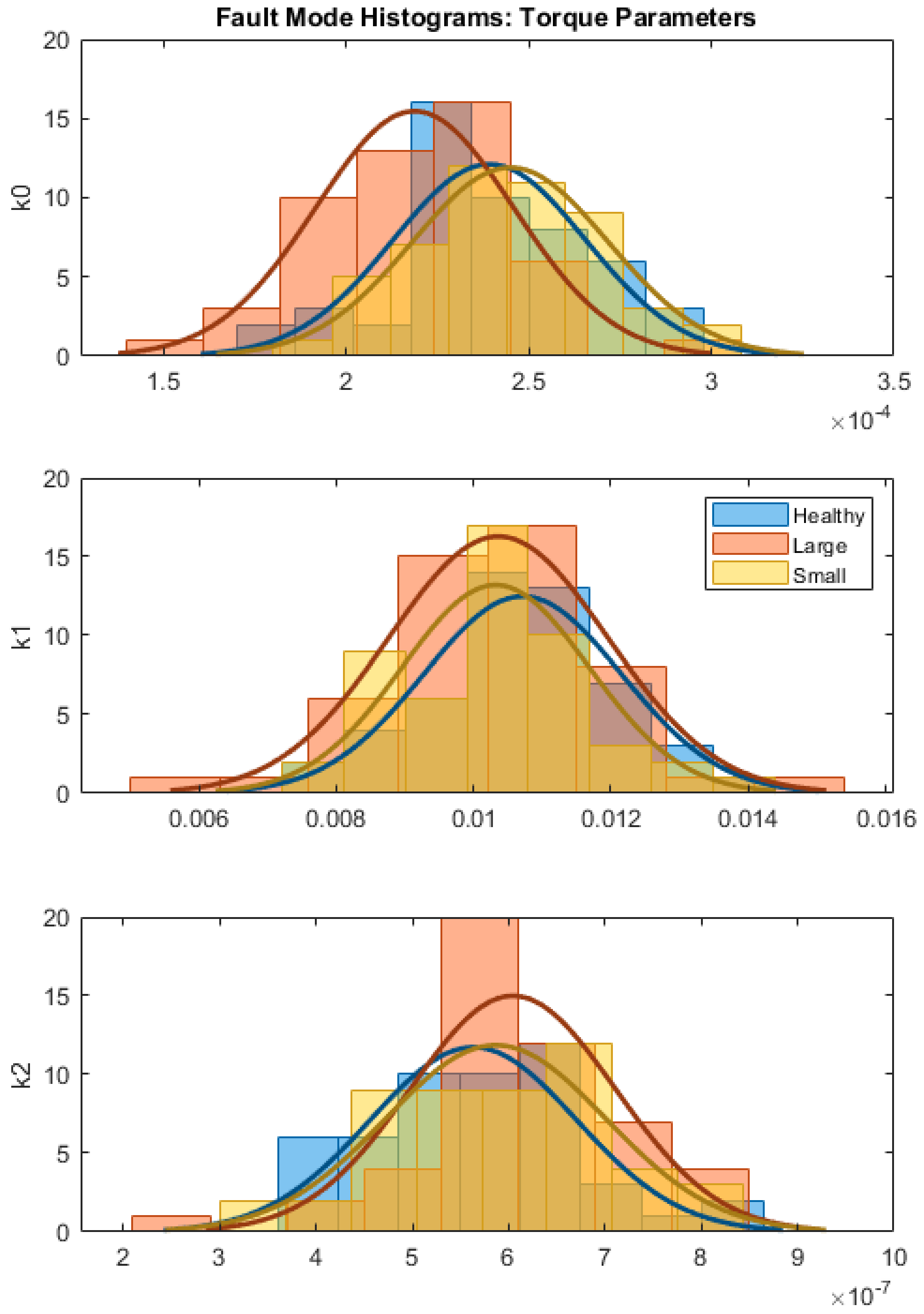
```
helperPlotHistogram(HealthyTheta1, LargeTheta1, SmallTheta1, {'hnn','hnv','hvv'})
```


Fault Mode Histograms: Head Parameters
hnn
×10^-6
hnv
×10^-4
Healthy
Large
Small
hvv
×10^-3

The histogram shows that h_{nn} offers good separability among the three modes but h_{nv}, h_{vv} parameters have overlapping probability distribution functions (PDFs).

Pump torque parameters:

```
helperPlotHistogram(HealthyTheta2, LargeTheta2, SmallTheta2, {'k0','k1','k2'})
```


Fault Mode Histograms: Torque Parameters
k0
×10^-4
k1
Healthy
Large
Small
k2
×10^-7

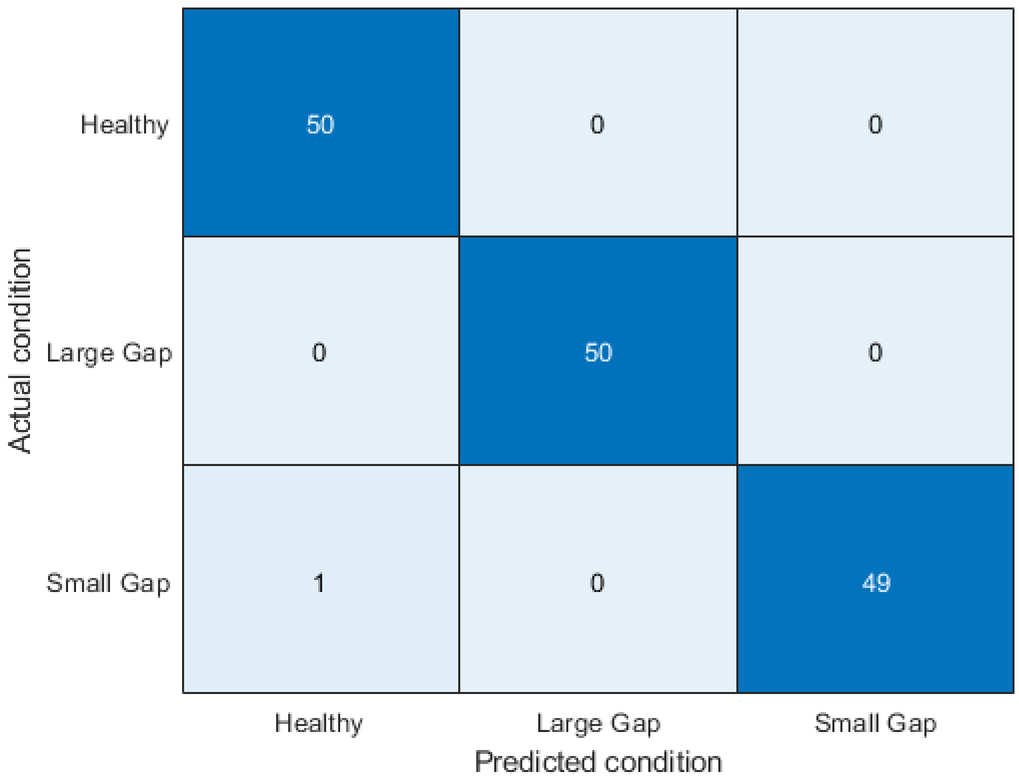
For the Torque parameters, the individual separability is not very good. There is still some variation in mean and variances that can be exploited by a trained 3-mode classifier. If the PDFs show good separation in mean or variance, you can design likelihood ratio tests to quickly assign a test dataset to the most likely mode. This is shown next for the pump head parameters.

Let:

- H_0: hypothesis that head parameters belong to the healthy pump mode
- H_1: hypothesis that head parameters belong to the pump with large clearance gap
- H_2: hypothesis that head parameters belong to the pump with small clearance gap

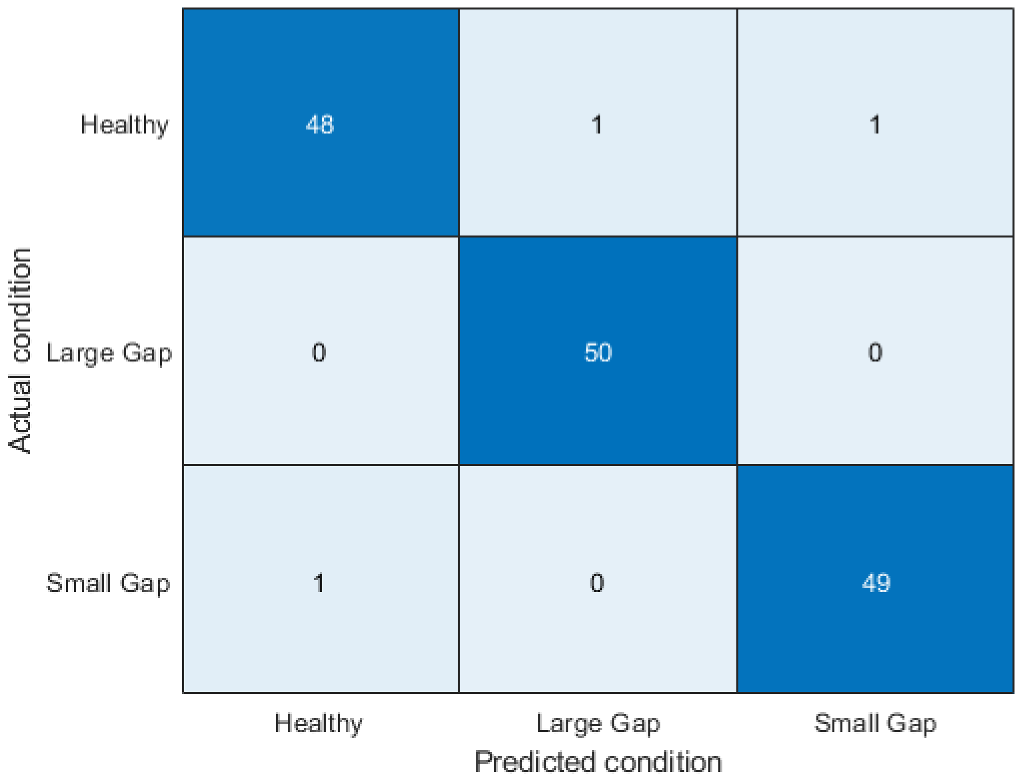
Consider the available parameter sets as test samples for mode prediction. Assign the predicted mode as belonging to one for which the joint PDF has the highest value (Hypothesis H_0 is selected over H_1 if $p(H_0) > p(H_1)$). Then plot the results comparing the true and predicted modes in a confusion matrix. Function `mvnpdf` is used for computing the PDF value and the functions `confusionmatrix` and `heatmap` are used for confusion matrix visualization. See helper function `pumpModeLikelihoodTest`.

```
% Pump head confusion matrix
figure
pumpModeLikelihoodTest(HealthyTheta1, LargeTheta1, SmallTheta1)
```



The confusion plot shows perfect separation between the three modes, which is not surprising given the clear separation among the histograms for h_{nn} parameters.

```
% Pump torque confusion matrix
pumpModeLikelihoodTest(HealthyTheta2, LargeTheta2, SmallTheta2)
```



The results are slightly worse for the torque parameters. Still, the success rate is quite high (97%) even though the PDFs of the three modes overlapped significantly. This is because the PDF value calculation is affected both by the location (mean) as well as the amplitude (variance).

B. Multi-Class Classification of Fault Modes Using Tree Bagging

In this section, another classification technique is discussed that is more suitable when classification among a larger number of modes is required. Consider the following fault modes of operation of the pump:

1. Healthy operation
2. Wear at clearance gap

3. Small deposits at impeller outlet
4. Deposits at impeller inlet
5. Abrasive wear at impeller outlet
6. Broken blade
7. Cavitation

The classification problem is harder because there are only three parameters computed and you need to distinguish among 7 modes of operation. Thus, not only you have to compare the estimated parameters for each fault mode to the healthy mode, but also to each other - both the *direction* (increase or reduction in value) and *magnitude* (10% change vs. 70% change) of the parameter change must be taken into consideration.

Here the use of TreeBagger classifier for this problem is shown. Tree Bagger is an ensemble learning technique that uses bootstrap aggregation (bagging) of features to create decision trees that classify labeled data. 50 labeled datasets are collected for the 7 modes of operation. Estimate pump head parameters for each dataset and train the classifier using a subset of the parameter estimates from each mode.

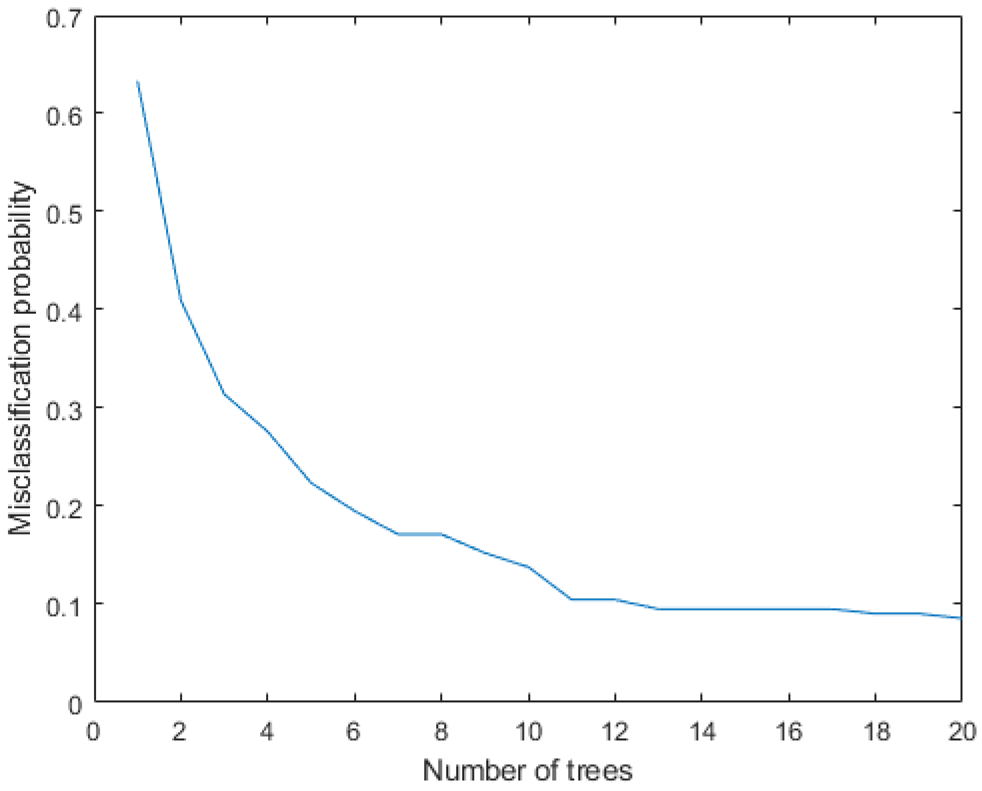
```
load MultipleFaultsData
% Compute pump head parameters
HealthyTheta = linearFit(1, HealthyEnsemble);
Fault1Theta  = linearFit(1, Fault1Ensemble);
Fault2Theta  = linearFit(1, Fault2Ensemble);
Fault3Theta  = linearFit(1, Fault3Ensemble);
Fault4Theta  = linearFit(1, Fault4Ensemble);
Fault5Theta  = linearFit(1, Fault5Ensemble);
Fault6Theta  = linearFit(1, Fault6Ensemble);

% Generate labels for each mode of operation
Label = {'Healthy','ClearanceGapWear','ImpellerOutletDeposit',...
    'ImpellerInletDeposit','AbrasiveWear','BrokenBlade','Cavitation'};
VarNames = {'hnn','hnv','hvv','Condition'};
% Assemble results in a table with parameters and corresponding labels
N = 50;
T0 = [array2table(HealthyTheta),repmat(Label(1),[N,1])];
T0.Properties.VariableNames = VarNames;
T1 = [array2table(Fault1Theta), repmat(Label(2),[N,1])];
T1.Properties.VariableNames = VarNames;
T2 = [array2table(Fault2Theta), repmat(Label(3),[N,1])];
T2.Properties.VariableNames = VarNames;
T3 = [array2table(Fault3Theta), repmat(Label(4),[N,1])];
T3.Properties.VariableNames = VarNames;
```

```
T4 = [array2table(Fault4Theta), repmat(Label(5),[N,1])];
T4.Properties.VariableNames = VarNames;
T5 = [array2table(Fault5Theta), repmat(Label(6),[N,1])];
T5.Properties.VariableNames = VarNames;
T6 = [array2table(Fault6Theta), repmat(Label(7),[N,1])];
T6.Properties.VariableNames = VarNames;

% Stack all data
% Use 30 out of 50 datasets for model creation
TrainingData = [T0(1:30,:);T1(1:30,:);T2(1:30,:);T3(1:30,:);T4(1:30,:);T5(1:30,:);T6(1

% Create an ensemble Mdl of 20 decision trees for predicting the
% labels using the parameter values
rng(3) % for reproducibility
Mdl = TreeBagger(20, TrainingData, 'Condition',...
   'OOBPrediction','on',...
   'OOBPredictorImportance','on')

Mdl = 
  TreeBagger
Ensemble with 20 bagged decision trees:
                    Training X:              [210x3]
                    Training Y:              [210x1]
                        Method:       classification
                 NumPredictors:                    3
         NumPredictorsToSample:                    2
                   MinLeafSize:                    1
                 InBagFraction:                    1
         SampleWithReplacement:                    1
          ComputeOOBPrediction:                    1
 ComputeOOBPredictorImportance:                    1
                     Proximity:                   []
                    ClassNames:  'AbrasiveWear'   'BrokenBlade'    'Cavitation' 'Cleara

  Properties, Methods
```

The performance of the TreeBagger model can be computed by studying its misclassification probability for out-of-bag observations as a function of number of decision trees.

```
% Compute out of bag error
figure
plot(oobError(Mdl))
```

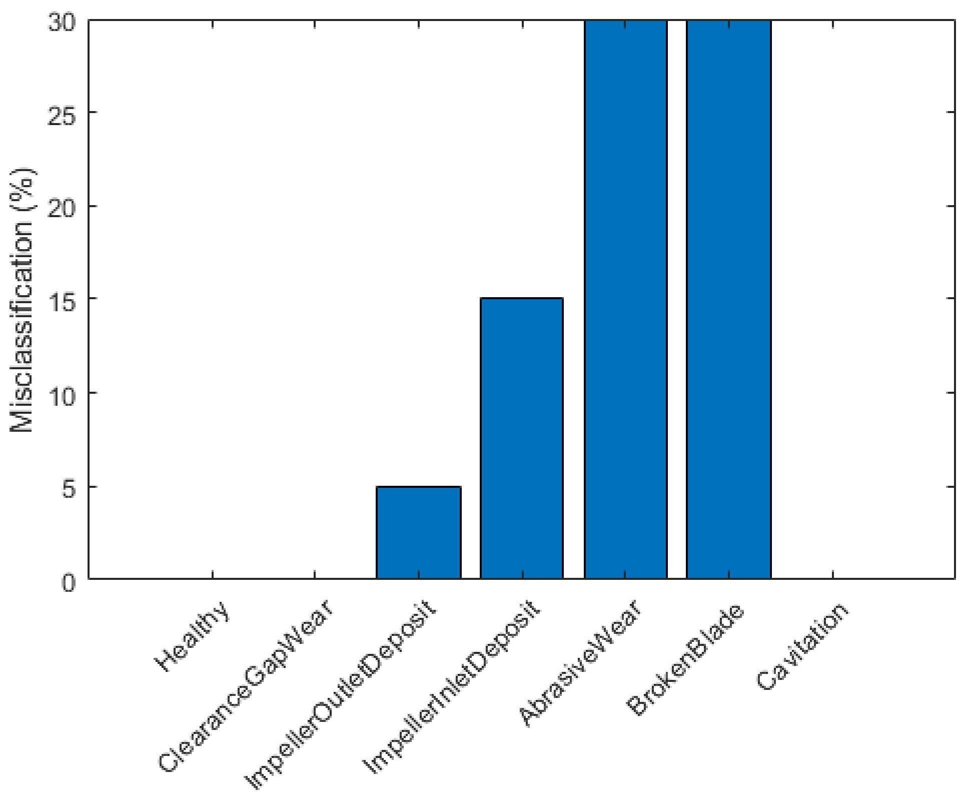
```
xlabel('Number of trees')
ylabel('Misclassification probability')
```



Finally, compute prediction performance of the model on test samples that were never used for growing the decision trees.

```
ValidationData = [T0(31:50,:);T1(31:50,:);T2(31:50,:);T3(31:50,:);T4(31:50,:);T5(31:50
PredictedClass = predict(Mdl,ValidationData);
E = zeros(1,7);
% Healthy data misclassification
E(1) = sum(~strcmp(PredictedClass(1:20), Label{1}));
% Clearance gap fault misclassification
E(2) = sum(~strcmp(PredictedClass(21:40), Label{2}));
% Impeller outlet deposit fault misclassification
```

```
E(3) = sum(~strcmp(PredictedClass(41:60), Label{3}));
% Impeller inlet deposit fault misclassification
E(4) = sum(~strcmp(PredictedClass(61:80), Label{4}));
% Abrasive wear fault misclassification
E(5) = sum(~strcmp(PredictedClass(81:100), Label{5}));
% Broken blade fault misclassification
E(6) = sum(~strcmp(PredictedClass(101:120), Label{6}));
% Cavitation fault misclassification
E(7) = sum(~strcmp(PredictedClass(121:140), Label{7}));
figure
bar(E/20*100)
xticklabels(Label)
set(gca,'XTickLabelRotation',45)
ylabel('Misclassification (%)')
```



The plot shows that the abrasive wear and broken blade faults are misclassified for 30% of the validation samples. A closer look at the predicted labels shows that in the misclassified cases, the 'AbrasiveWear' and 'BrokenBlade' labels get intermixed between each other only. This suggests that the symptoms for these fault categories are not sufficiently distinguishable for this classifier.

Summary

A well designed fault diagnosis strategy can save operating costs by minimizing service downtime and component replacement costs. The strategy benefits from a good knowledge about the operating machine's dynamics which is used in combination with sensor measurements to detect and isolate different kinds of faults.

This example discussed a parametric approach for fault detection and isolation based on steady-state experiments. This approach requires careful modeling of the system dynamics and using the parameters (or transformations thereof) as features for designing fault diagnosis algorithms. The parameters were used for training anomaly detectors, performing likelihood ratio tests and for training of multi-class classifiers.

How to make use of classification techniques in real-life testing of pumps

A summary of the fault diagnosis workflow follows.

1. Run the test pump at its nominal speed. Turn the discharge valve to various settings to control the flow rate. For each valve position, note down the pump speed, flow rate, pressure differentials and torque.
2. Estimate parameters for the pump head and pump torque characteristic (steady state) equations.
3. If the uncertainty/noise is low and the parameter estimates are reliable, the estimated parameters can be directly compared to their nominal values. Their relative magnitudes would indicate the nature of the fault.
4. In a general noisy situation, use the anomaly detection techniques to first check if there is a fault present in the system at all. This can be done very quickly by comparing the estimated parameter values against the mean and covariance values obtained from a historical database of healthy pumps.
5. If a fault is indicated, use the fault classification techniques (such as likelihood ratio tests or output of a classifier) to isolate the most probable cause(s). The choice of classification technique would depend upon sensor data available, their reliability, the severity of the fault and availability of historical information regarding the fault modes.

For a fault diagnosis approach based on residual analysis, see the “Fault Diagnosis of Centrifugal Pumps Using Residual Analysis” on page 4-71 example.

References

1 Isermann, Rolf, *Fault-Diagnosis Applications. Model-Based Condition Monitoring: Actuators, Drives, Machinery, Plants, Sensors, and Fault-tolerant System*, Edition 1, Springer-Verlag Berlin Heidelberg, 2011.

Supporting Functions

Linear fit to pump parameters.

```
function varargout = linearFit(Form, Data)
%linearFit Linear least squares solution for Pump Head and Torque parameters.
%
% If Form==0, accept separate inputs and return separate outputs. For one experiment or
% If Form==1, accept an ensemble and return compact parameter vectors. For several expe
if Form==0
   w = Data{1};
   Q = Data{2};
   H = Data{3};
   M = Data{4};
   n = length(Q);
   if isscalar(w), w = w*ones(n,1); end
   Q = Q(:); H = H(:); M = M(:);
   Predictor = [w.^2, w.*Q, Q.^2];
   Theta1 = Predictor\H;
   hnn =  Theta1(1);
   hnv = -Theta1(2);
   hvv = -Theta1(3);
   Theta2 = Predictor\M;
   k0 =  Theta2(2);
   k1 = -Theta2(3);
   k2 =  Theta2(1);
   varargout = {hnn, hnv, hvv, k0, k1, k2};
else
   H = cellfun(@(x)x.Head,Data,'uni',0);
   Q = cellfun(@(x)x.Discharge,Data,'uni',0);
   M = cellfun(@(x)x.Torque,Data,'uni',0);
   W = cellfun(@(x)x.Speed,Data,'uni',0);
   N = numel(H);

   Theta1 = zeros(3,N);
   Theta2 = zeros(3,N);
   for kexp = 1:N
      Predictor = [W{kexp}.^2, W{kexp}.*Q{kexp}, Q{kexp}.^2];
      X1 = Predictor\H{kexp};
      hnn =  X1(1);
      hnv = -X1(2);
      hvv = -X1(3);
      X2 = Predictor\M{kexp};
      k0 =  X2(2);
      k1 = -X2(3);
      k2 =  X2(1);

      Theta1(:,kexp) = [hnn; hnv; hvv];
      Theta2(:,kexp) = [k0; k1; k2];
   end
   varargout = {Theta1', Theta2'};
end
end
```

Membership likelihood test.

```
function pumpModeLikelihoodTest(HealthyTheta, LargeTheta, SmallTheta)
%pumpModeLikelihoodTest Generate predictions based on PDF values and plot confusion mat

m1 = mean(HealthyTheta);
c1 = cov(HealthyTheta);
m2 = mean(LargeTheta);
c2 = cov(LargeTheta);
m3 = mean(SmallTheta);
c3 = cov(SmallTheta);

N = size(HealthyTheta,1);

% True classes
% 1: Healthy: group label is 1.
X1t = ones(N,1);
% 2: Large gap: group label is 2.
X2t = 2*ones(N,1);
% 3: Small gap: group label is 3.
X3t = 3*ones(N,1);

% Compute predicted classes as those for which the joint PDF has the maximum value.
X1 = zeros(N,3);
X2 = zeros(N,3);
X3 = zeros(N,3);
```

```
for ct = 1:N
   % Membership probability density for healthy parameter sample
   HealthySample  = HealthyTheta(ct,:);
   x1 = mvnpdf(HealthySample, m1, c1);
   x2 = mvnpdf(HealthySample, m2, c2);
   x3 = mvnpdf(HealthySample, m3, c3);
   X1(ct,:) = [x1 x2 x3];

   % Membership probability density for large gap pump parameter
   LargeSample  = LargeTheta(ct,:);
   x1 = mvnpdf(LargeSample, m1, c1);
   x2 = mvnpdf(LargeSample, m2, c2);
   x3 = mvnpdf(LargeSample, m3, c3);
   X2(ct,:) = [x1 x2 x3];

   % Membership probability density for small gap pump parameter
   SmallSample  = SmallTheta(ct,:);
   x1 = mvnpdf(SmallSample, m1, c1);
   x2 = mvnpdf(SmallSample, m2, c2);
   x3 = mvnpdf(SmallSample, m3, c3);
   X3(ct,:) = [x1 x2 x3];
end

[~,PredictedGroup] = max([X1;X2;X3],[],2);
TrueGroup = [X1t; X2t; X3t];
C = confusionmat(TrueGroup,PredictedGroup);
heatmap(C, ...
    'YLabel', 'Actual condition', ...
    'YDisplayLabels', {'Healthy','Large Gap','Small Gap'}, ...
    'XLabel', 'Predicted condition', ...
    'XDisplayLabels', {'Healthy','Large Gap','Small Gap'}, ...
    'ColorbarVisible','off');
end
```

See Also

More About

- “Decision Models for Fault Detection and Diagnosis” on page 4-2

Fault Diagnosis of Centrifugal Pumps Using Residual Analysis

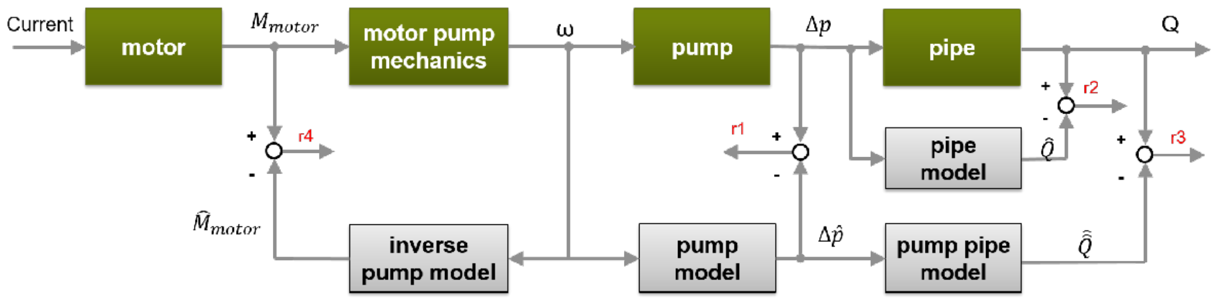
This example shows a model parity-equations based approach for detection and diagnosis of different types of faults that occur in a pumping system. This example extends the techniques presented in the “Fault Diagnosis of Centrifugal Pumps Using Steady State Experiments” on page 4-39 to the situation where data spans multiple operating conditions.

The example follows the centrifugal pump analysis presented in the Fault Diagnosis Applications book by Rolf Isermann [1].

Multi-Speed Pump Runs - Diagnosis by Residual Analysis

The steady-state pump head and torque equations do not produce accurate results if the pump is run at rapidly varying or a wider range of speeds. Friction and other losses could become significant and the model's parameters exhibit dependence on speed. A widely applicable approach in such cases is to create a black box model of the behavior. The parameters of such models need not be physically meaningful. The model is used as a device for simulation of known behaviors. The model's outputs are subtracted from the corresponding measured signals to compute *residuals*. The properties of residuals, such as their mean, variance and power are used to distinguish between normal and faulty operations.

Using the static pump head equation, and the dynamic pump-pipe equations, the 4 residuals as shown in the figure can be computed.



The model has of the following components:

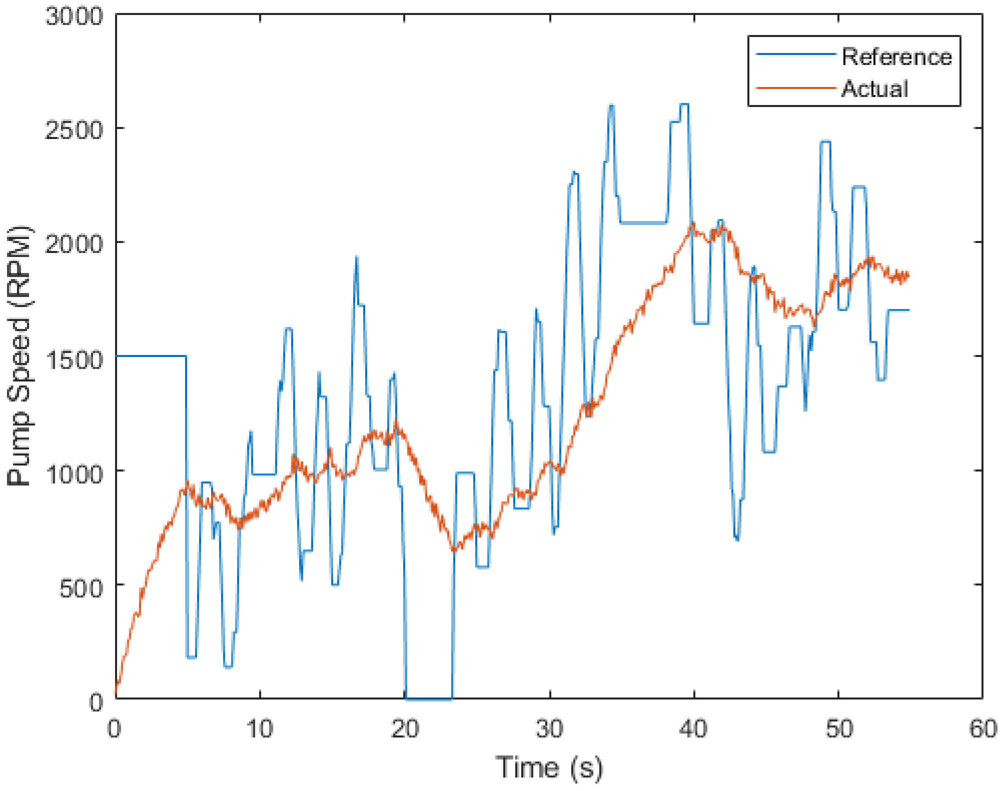
- Static pump model: $\Delta\widehat{p}(t) = \theta_1\omega^2(t) + \theta_2\omega(t)$
- Dynamic pipe model: $\widehat{Q}(t) = \theta_3 + \theta_4\sqrt{\Delta p(t)} + \theta_5\widehat{Q}(t-1)$
- Dynamic pump-pipe model: $\widehat{\widehat{Q}}(t) = \theta_3 + \theta_4\sqrt{\Delta\widehat{p}(t)} + \theta_5\widehat{\widehat{Q}}(t-1)$
- Dynamic inverse pump model:
  $\widehat{M}_{motor}(t) = \theta_6 + \theta_7\omega(t) + \theta_8\omega(t-1) + \theta_9\omega^2(t) + \theta_{10}\widehat{M}_{motor}(t-1)$

The model parameters $\theta_1, \ldots, \theta_{10}$ show dependence on pump speed. In this example, a piecewise linear approximation for the parameters is computed. Divide the operating region into 3 regimes:

1. $\omega \le 900\ \ RPM$
2. $900 < \omega \le 1500\ \ RPM$
3. $\omega > 1500\ \ RPM$

A healthy pump was run over a reference speed range of 0 - 3000 RPM in closed loop with a closed-loop controller. The reference input is a modified PRBS signal. The measurements for motor torque, pump torque, speed and pressure were collected at 10 Hz sampling frequency. Load the measured signals and plot the reference and actual pump speeds.

```
load DynamicOperationData
figure
plot(t, RefSpeed, t, w)
xlabel('Time (s)')
ylabel('Pump Speed (RPM)')
legend('Reference','Actual')
```



Define operating regimes based on pump speed ranges.

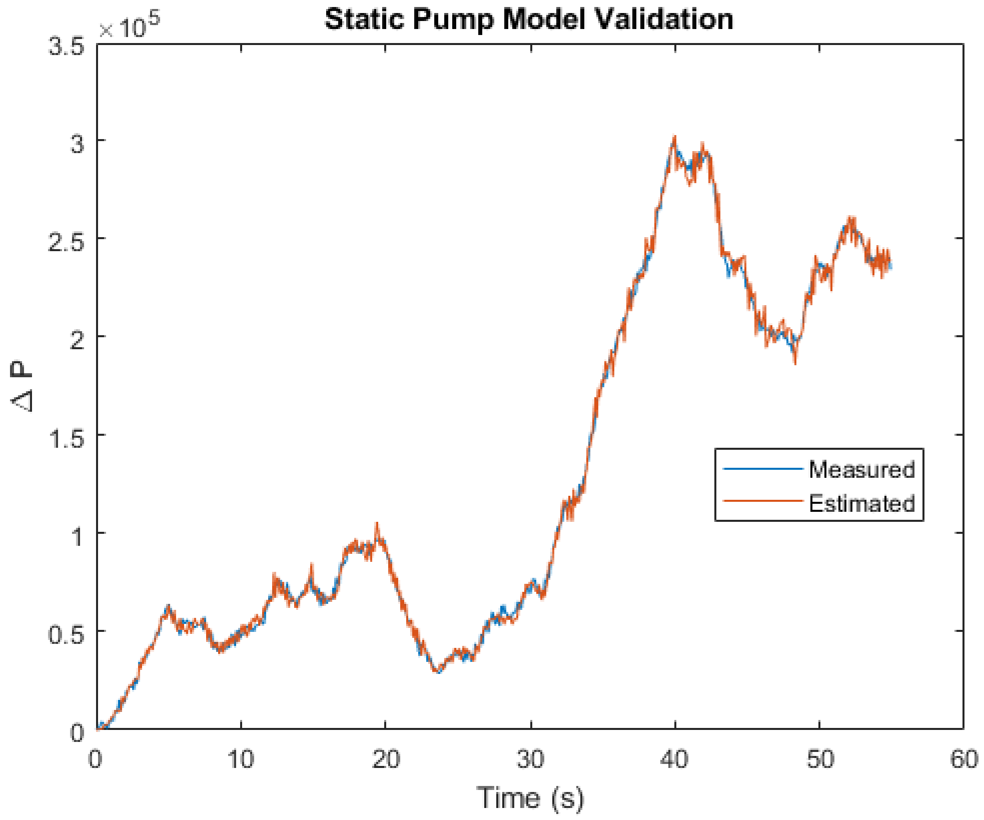
```
I1 = w<=900;            % first operating regime
I2 = w>900 & w<=1500;   % second operating regime
I3 = w>1500;            % third operating regime
```

Model Identification

A. Static Pump Model Identification

Estimate the parameters θ_1 and θ_2 in the static pump equation using the measured values of pump speed $\omega(t)$ and pressure differential $\Delta p(t)$ as input-output data. See the helper function `staticPumpEst` that performs this estimation.

```
th1 = zeros(3,1);
th2 = zeros(3,1);
dpest = nan(size(dp));  % estimated pressure difference
[th1(1), th2(1), dpest(I1)] = staticPumpEst(w, dp, I1);  % Theta1, Theta2 estimates fo
[th1(2), th2(2), dpest(I2)] = staticPumpEst(w, dp, I2);  % Theta1, Theta2 estimates fo
[th1(3), th2(3), dpest(I3)] = staticPumpEst(w, dp, I3);  % Theta1, Theta2 estimates fo
plot(t, dp, t, dpest) % compare measured and predicted pressure differential
xlabel('Time (s)')
ylabel('\Delta P')
legend('Measured','Estimated','Location','best')
title('Static Pump Model Validation')
```

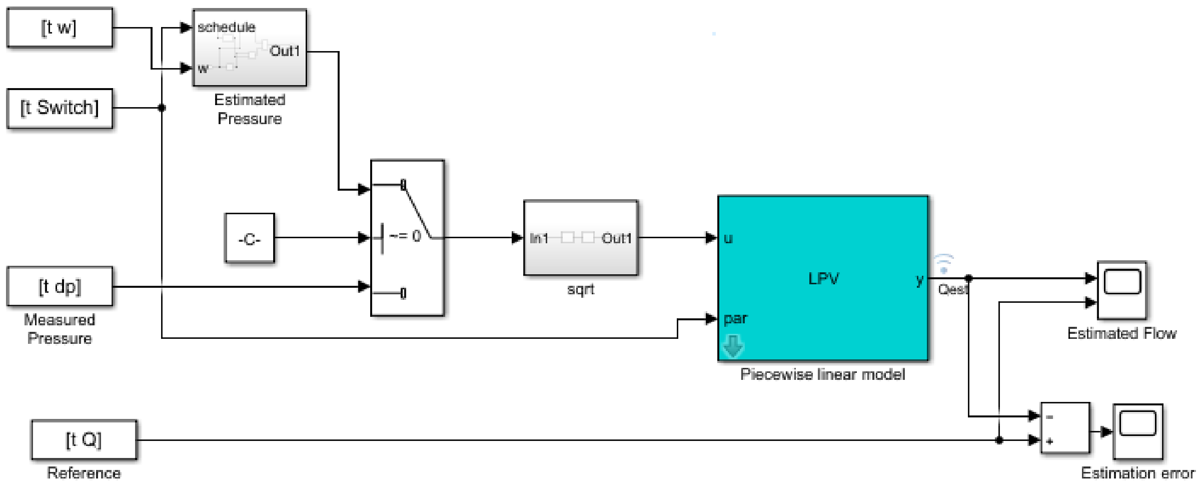


B. Dynamic Pipe Model Identification

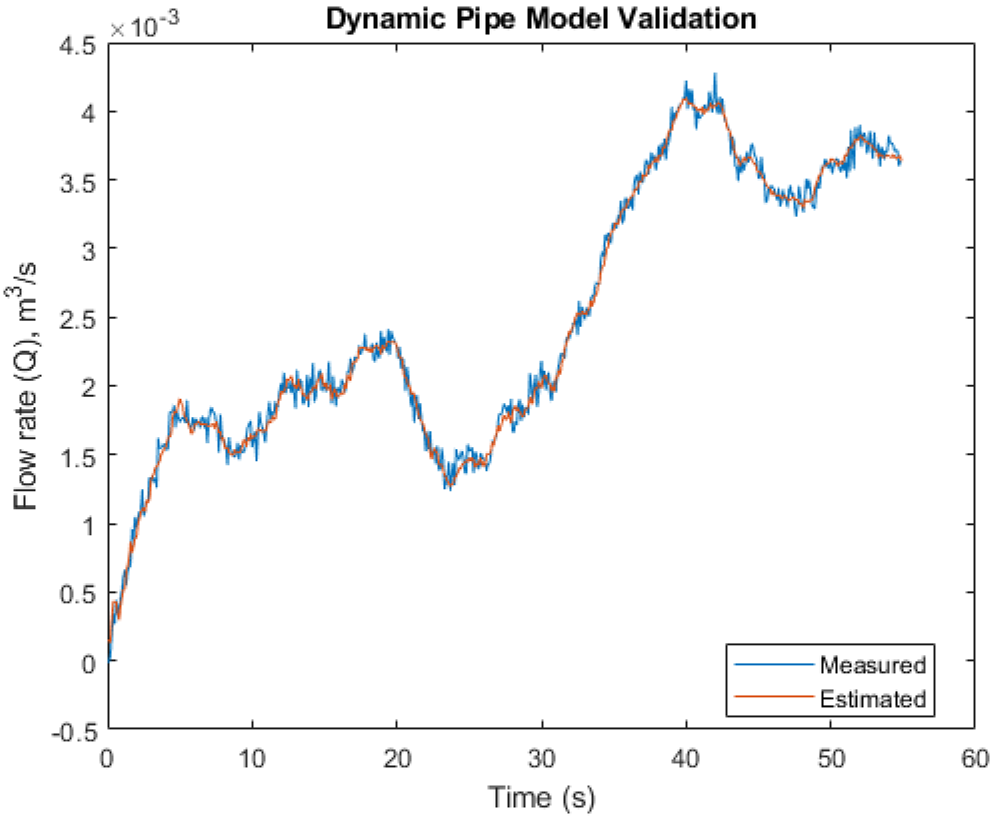
Estimate the parameters θ_3, θ_4 and θ_5 in the pipe discharge flow equation $\widehat{Q}(t) = \theta_3 + \theta_4\sqrt{\Delta p(t)} + \theta_5\widehat{Q}(t-1)$, using the measured values of flow rate $Q(t)$ and pressure differential $\Delta p(t)$ as input-output data. See the helper function `dynamicPipeEst` that performs this estimation.

```
th3 = zeros(3,1);
th4 = zeros(3,1);
th5 = zeros(3,1);
[th3(1), th4(1), th5(1)] = dynamicPipeEst(dp, Q, I1); % Theta3, Theta4, Theta5 estimate
[th3(2), th4(2), th5(2)] = dynamicPipeEst(dp, Q, I2); % Theta3, Theta4, Theta5 estimate
[th3(3), th4(3), th5(3)] = dynamicPipeEst(dp, Q, I3); % Theta3, Theta4, Theta5 estimate
```

Unlike the static pump model case, the dynamic pipe model shows dynamic dependence on flow rate values. To simulate the model under varying speed regimes, a piecewise-linear model is created in Simulink using the "LPV System" block of Control System Toolbox. See the Simulink model `LPV_pump_pipe` and the helper function `simulatePumpPipeModel` that performs the simulation.



```
% Check Control System Toolbox availability
ControlsToolboxAvailable = ~isempty(ver('control')) && license('test', 'Control_Toolbox
if ControlsToolboxAvailable
    % Simulate the dynamic pipe model. Use measured value of pressure as input
    Ts = t(2)-t(1);
    Switch = ones(size(w));
    Switch(I2) = 2;
    Switch(I3) = 3;
    UseEstimatedP = 0;
    Qest_pipe = simulatePumpPipeModel(Ts,th3,th4,th5);
    plot(t,Q,t,Qest_pipe) % compare measured and predicted flow rates
else
    % Load pre-saved simulation results from the piecewise linear Simulink model
    load DynamicOperationData Qest_pipe
    Ts = t(2)-t(1);
    plot(t,Q,t,Qest_pipe)
end
xlabel('Time (s)')
ylabel('Flow rate (Q), m^3/s')
legend('Measured','Estimated','Location','best')
title('Dynamic Pipe Model Validation')
```

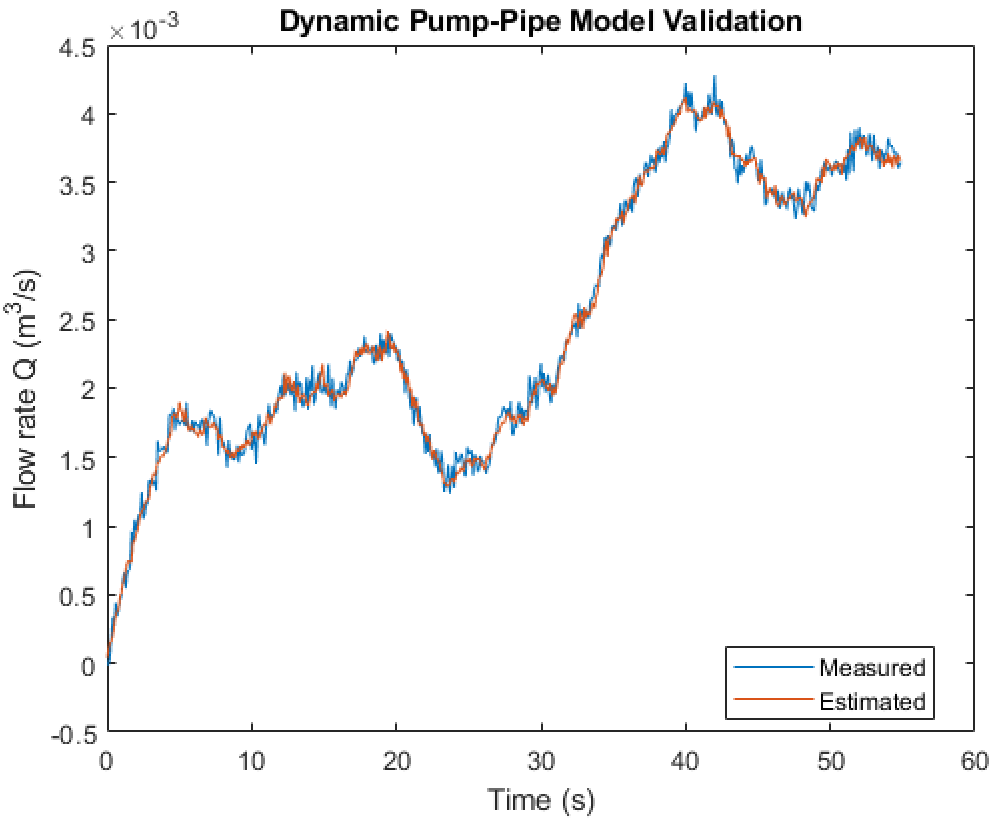


C. Dynamic Pump Pipe Model Identification

The Dynamic pump-pipe model uses the same parameters identified above ($\theta_3, \theta_4, \theta_5$) except that the model simulation requires the use of estimated pressure difference rather than the measured one. Hence no new identification is required. Check that the estimated values of $\theta_3, \theta_4, \theta_5$ give a good reproduction of the pump-pipe dynamics.

```
if ControlsToolboxAvailable
    UseEstimatedP = 1;
    Qest_pump_pipe = simulatePumpPipeModel(Ts,th3,th4,th5);
    plot(t,Q,t,Qest_pump_pipe) % compare measured and predicted flow rates
else
```

```
    load DynamicOperationData Qest_pump_pipe
    plot(t,Q,t,Qest_pump_pipe)
end

xlabel('Time (s)')
ylabel('Flow rate Q (m^3/s)')
legend('Measured','Estimated','location','best')
title('Dynamic Pump-Pipe Model Validation')
```



The fit is virtually identical to the one obtained using measured pressure values.

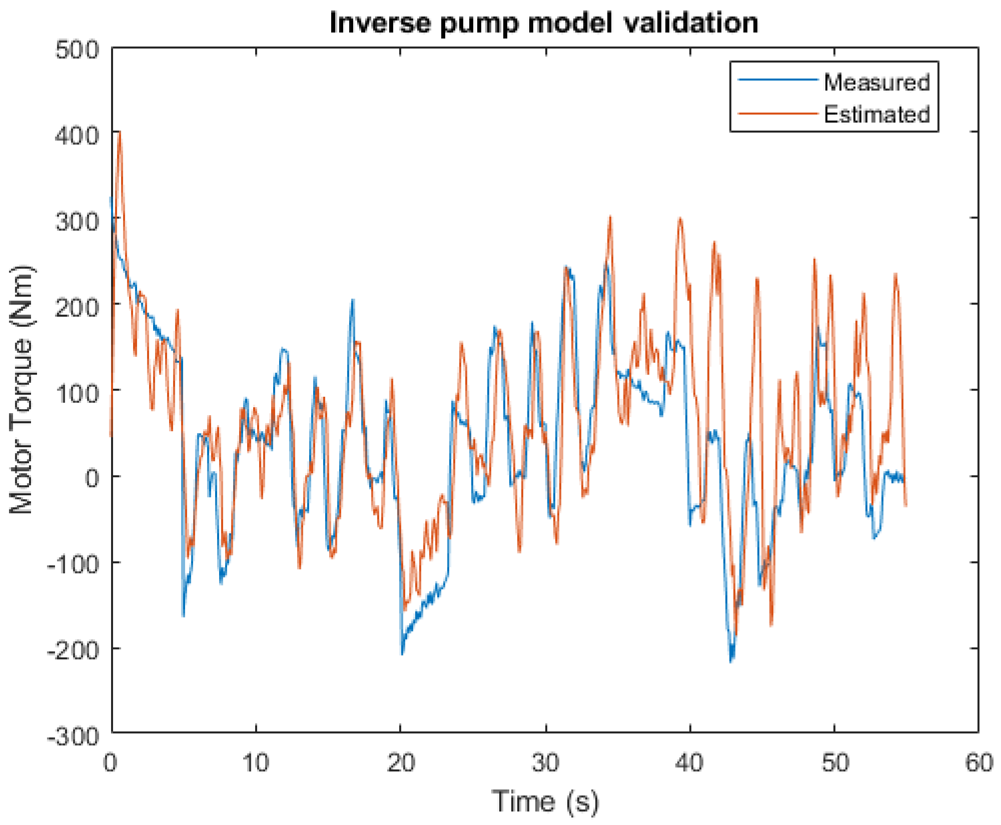
D. Dynamic Inverse Pump Model Identification

The parameters $\theta_6, \ldots, \theta_{10}$ can be identified in a similar manner, by regressing the measured torque values on the previous torque and speed measurements. However, a complete multi-speed simulation of the resulting piecewise linear model does not provide a good fit to the data. Hence a different black box modeling approach is tried which involves identifying a Nonlinear ARX model with rational regressors to fit the data.

```
% Use first 300 samples out of 550 for identification
N = 350;
sys3 = identifyNonlinearARXModel(Mmot,w,Q,Ts,N)
```

```
sys3 =
Nonlinear ARX model with 1 output and 2 inputs
 Inputs: u1, u2
 Outputs: y1
 Standard regressors corresponding to the orders:
   na = [2]
   nb = [2 1]
   nk = [0 1]
 Custom regressors:
   u1(t-2)^2
   u1(t)*u2(t-2)
   u2(t)^2
 Nonlinear regressors:
  none
 Model output is linear in regressors.
Sample time: 0.1 seconds

Status:
Estimated using NLARX on time domain data.
Fit to estimation data: 50.55% (simulation focus)
FPE: 1759, MSE: 3214
```

```
Mmot_est = sim(sys3,[w Q]);
plot(t,Mmot,t,Mmot_est) % compare measured and predicted motor torque
xlabel('Time (s)')
ylabel('Motor Torque (Nm)')
legend('Measured','Estimated','location','best')
title('Inverse pump model validation')
```



Residue Generation

Define *residue* of a model as the difference between a measured signal and the corresponding model-produced output. Compute the four residuals corresponding to the four model components.

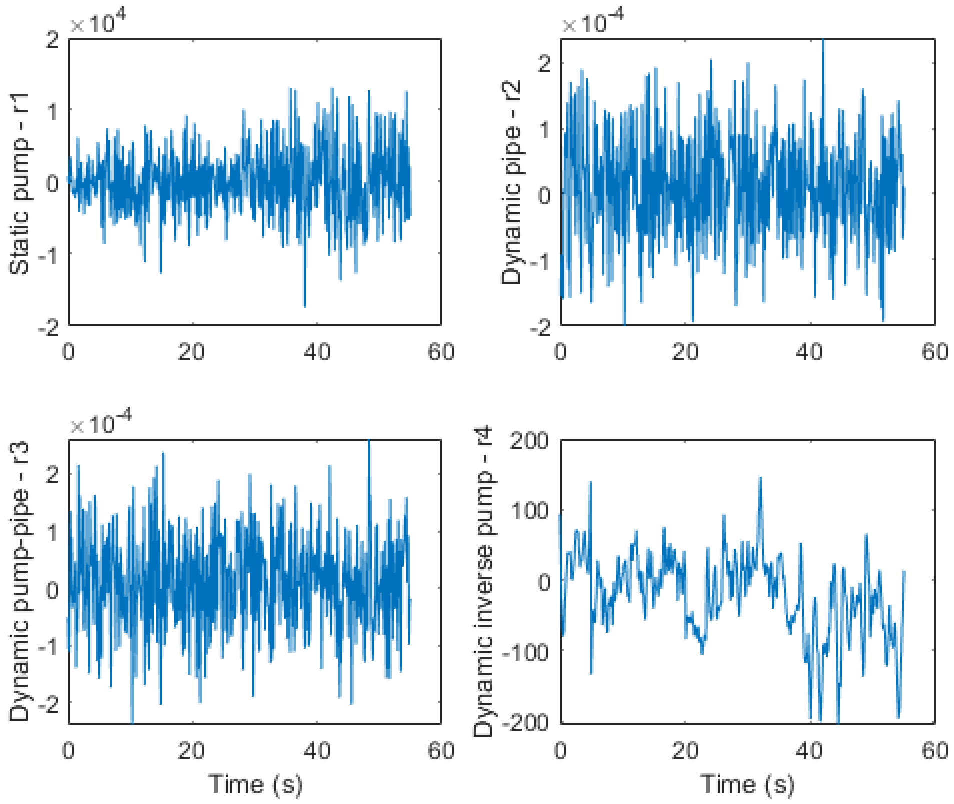
```
r1 = dp - dpest;
r2 = Q - Qest_pipe;
r3 = Q - Qest_pump_pipe;
```

For computing the inverse pump model residue, apply a smoothing operation on the model output using a moving average filter since the original residues show large variance.

```
r4 = Mmot - movmean(Mmot_est,[1 5]);
```

A view of training residues:

```
figure
subplot(221)
plot(t,r1)
ylabel('Static pump - r1')
subplot(222)
plot(t,r2)
ylabel('Dynamic pipe - r2')
subplot(223)
plot(t,r3)
ylabel('Dynamic pump-pipe - r3')
xlabel('Time (s)')
subplot(224)
plot(t,r4)
ylabel('Dynamic inverse pump - r4')
xlabel('Time (s)')
```



Residue Feature Extraction

Residues are signals from which suitable features are extracted for fault isolation. Since no parametric information is available, consider features that are derived purely from signal properties such as maximum amplitude or variance of the signal.

Consider a set of 20 experiments on the pump-pipe system using PRBS input realizations. The experiment set is repeated for each of the following modes:

1. Healthy pump
2. Fault 1: Wear at clearance gap
3. Fault 2: Small deposits at impeller outlet

4 Fault 3: Deposits at impeller inlet

5 Fault 4: Abrasive wear at impeller outlet

6 Fault 5: Broken blade

7 Fault 6: Cavitation

8 Fault 7: Speed sensor bias

9 Fault 8: Flowmeter bias

10 Fault 9: Pressure sensor bias

Load the experimental data

```
load MultiSpeedOperationData
% Generate operation mode labels
Labels = {'Healthy','ClearanceGapWear','ImpellerOutletDeposit',...
    'ImpellerInletDeposit','AbrasiveWear','BrokenBlade','Cavitation','SpeedSensorBias'
    'FlowmeterBias','PressureSensorBias'};
```

Compute residues for each ensemble and each mode of operation. This takes several minutes. Hence the residual data is saved in a data file. Run the `helperComputeEnsembleResidues` to generate the residuals, as in:

```
% HealthyR = helperComputeEnsembleResidues(HealthyEnsemble,Ts,sys3,th1,th2,th3,th4,th5

% Load pre-saved data from "helperComputeEnsembleResidues" run
load Residuals
```

The feature of the residues that would have the most mode-discrimination power is not known a-priori. So generate several candidate features: mean, maximum amplitude, variance, kurtosis and 1-norm for each residual.

```
CandidateFeatures = {@mean, @(x)max(abs(x)), @kurtosis, @var, @(x)sum(abs(x))};
FeatureNames = {'Mean','Max','Kurtosis','Variance','OneNorm'};
% generate feature table from gathered residuals of each fault mode
HealthyFeature = helperGenerateFeatureTable(HealthyR, CandidateFeatures, FeatureNames)
Fault1Feature  = helperGenerateFeatureTable(Fault1R,  CandidateFeatures, FeatureNames)
Fault2Feature  = helperGenerateFeatureTable(Fault2R,  CandidateFeatures, FeatureNames)
Fault3Feature  = helperGenerateFeatureTable(Fault3R,  CandidateFeatures, FeatureNames)
Fault4Feature  = helperGenerateFeatureTable(Fault4R,  CandidateFeatures, FeatureNames)
Fault5Feature  = helperGenerateFeatureTable(Fault5R,  CandidateFeatures, FeatureNames)
Fault6Feature  = helperGenerateFeatureTable(Fault6R,  CandidateFeatures, FeatureNames)
Fault7Feature  = helperGenerateFeatureTable(Fault7R,  CandidateFeatures, FeatureNames)
Fault8Feature  = helperGenerateFeatureTable(Fault8R,  CandidateFeatures, FeatureNames)
Fault9Feature  = helperGenerateFeatureTable(Fault9R,  CandidateFeatures, FeatureNames)
```

There are 20 features in each feature table. Each table contains 50 observations (rows), one from each experiment.

```
N = 50; % number of experiments in each mode
FeatureTable = [...
   [HealthyFeature(1:N,:), repmat(Labels(1),[N,1])];...
   [Fault1Feature(1:N,:),  repmat(Labels(2),[N,1])];...
   [Fault2Feature(1:N,:),  repmat(Labels(3),[N,1])];...
   [Fault3Feature(1:N,:),  repmat(Labels(4),[N,1])];...
   [Fault4Feature(1:N,:),  repmat(Labels(5),[N,1])];...
   [Fault5Feature(1:N,:),  repmat(Labels(6),[N,1])];...
   [Fault6Feature(1:N,:),  repmat(Labels(7),[N,1])];...
   [Fault7Feature(1:N,:),  repmat(Labels(8),[N,1])];...
   [Fault8Feature(1:N,:),  repmat(Labels(9),[N,1])];...
   [Fault9Feature(1:N,:),  repmat(Labels(10),[N,1])]];
FeatureTable.Properties.VariableNames{end} = 'Condition';

% Preview some samples of training data
disp(FeatureTable([2 13 37 49 61 62 73 85 102 120],:))
```

```
      Mean1        Mean2      Mean3       Mean4        Max1       Max2       Max3       Ma
    _________    _______    _______    ________    _______    _______    _______    ___

    -0.033922    0.24455    0.24455    -0.17203    0.18878    0.48601    0.48601    0.6
      0.24157    0.94012    0.94012    -0.33526    0.60909     1.0537     1.0537    0.8
     0.033674    0.27526    0.27526    -0.17174    0.23597    0.60549    0.60549    0.5
      0.31984    0.86747    0.86747    -0.36405    0.90086     1.2882     1.2881      1
     -0.55312    0.86563    0.86564     0.27727    0.62675     1.0929     1.0929    0.4
     -0.37678    0.74538    0.74538     0.17598    0.45508    0.96567    0.96568    0.3
     -0.20143    0.51949    0.51949    -0.15955    0.35449    0.78562    0.78562    0.7
     -0.31755    0.90996    0.90996    0.078696    0.38023     1.0958     1.0958    0.4
    -0.049827    0.20009    0.20009    -0.63497    0.16226    0.48398    0.48398    0.4
     -0.21855    0.53551    0.53551     0.12487    0.28397    0.63171    0.63171     0
```

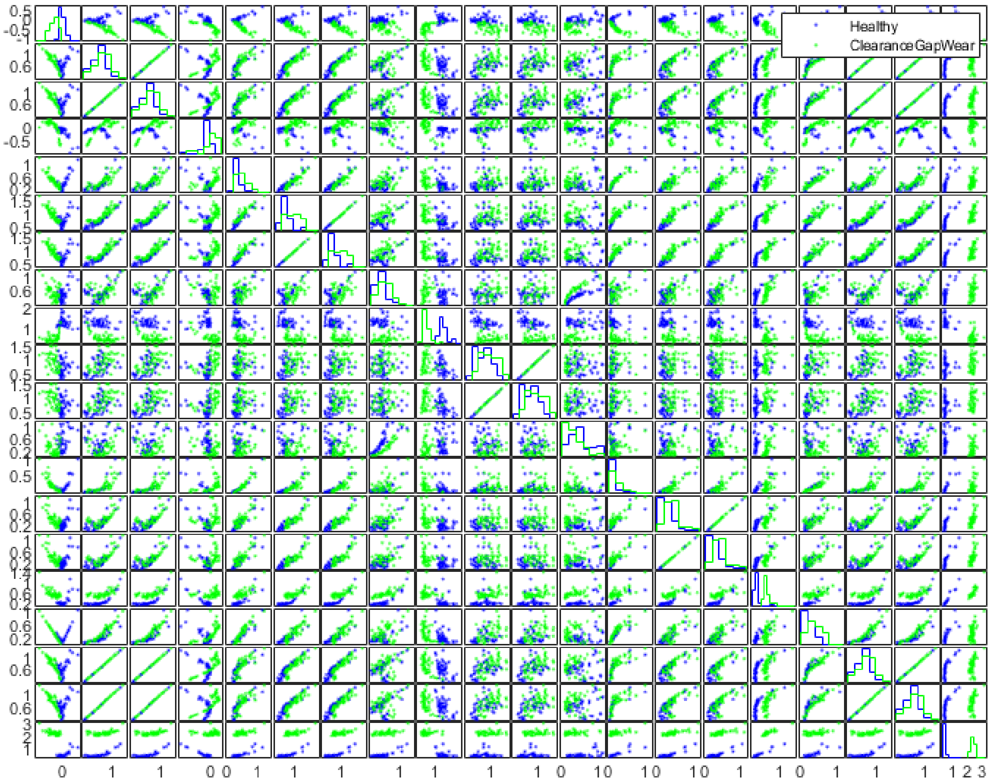
Classifier Design

A. Visualizing mode separability using scatter plot

Begin the analysis by visual inspection of the features. For this, consider Fault 1: Wear at clearance gap. To view which features are most suitable to detect this fault, generate a scatter plot of features with labels 'Healthy' and 'ClearanceGapWear'.

```
T = FeatureTable(:,1:20);
P = T.Variables;
R = FeatureTable.Condition;
```

```
I = strcmp(R,'Healthy') | strcmp(R,'ClearanceGapWear');
f = figure;
gplotmatrix(P(I,:),[],R(I))
f.Position(3:4) = f.Position(3:4)*1.5;
```

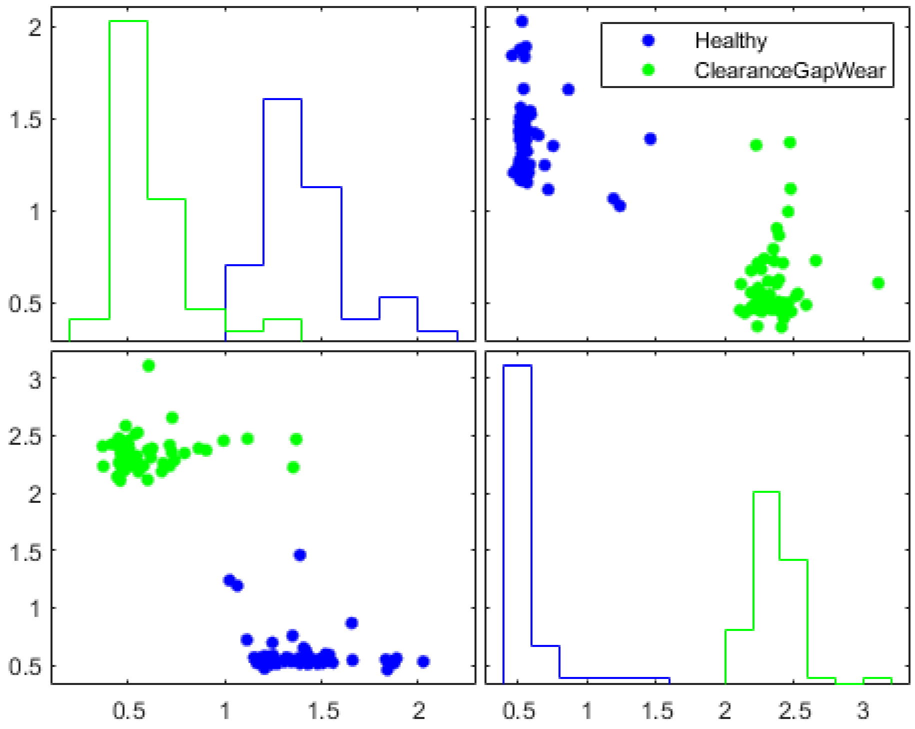


Although not clearly visible, features in columns 9 and 20 provide the most separation. Analyze these features more closely.

```
f = figure;
Names = FeatureTable.Properties.VariableNames;
fprintf('Selected features for clearance gap fault: %s, %s\n',Names{9},Names{20})
```

```
Selected features for clearance gap fault: Kurtosis1, OneNorm4
```

```
gplotmatrix(P(I,[9 20]),[],R(I))
```



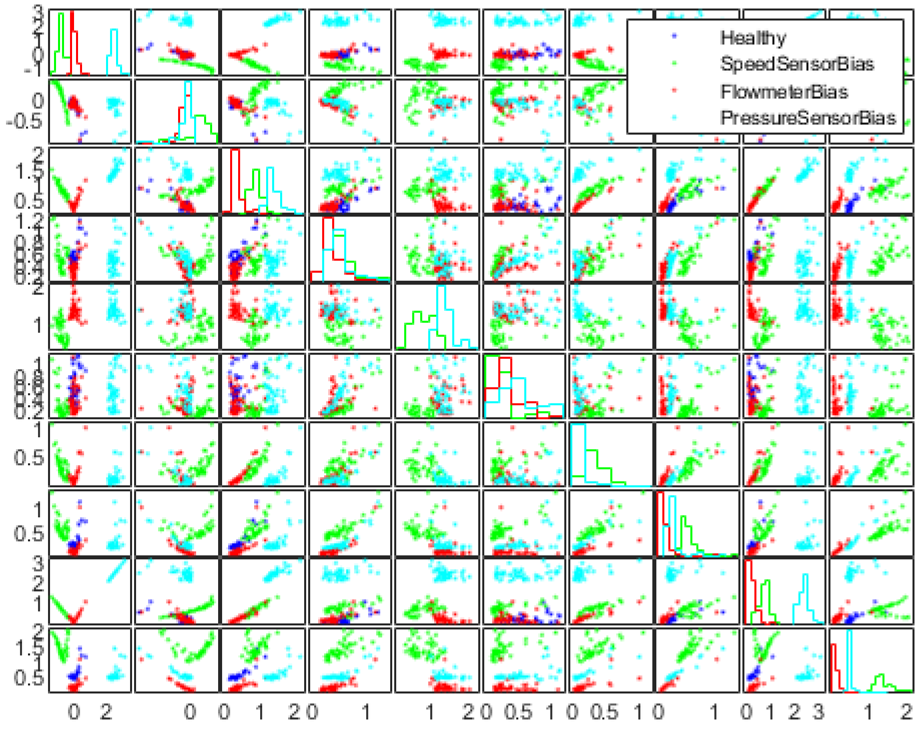
The plot now clearly shows that features Kurtosis1 and OneNorm4 can be used to separate healthy mode from gap clearance fault mode. A similar analysis can be performed for each fault mode. In all cases, it is possible to find a set of features that distinguish the fault modes. Hence detection of a faulty behavior is always possible. However, fault isolation is more difficult since the same features are affected by multiple fault types. For example, the features Kurtosis1 (Kurtosis of r1) and OneNorm4 (1-norm of r4) show a change for many fault types. Still some faults such as sensor biases are more easily isolable where the fault is separable in many features.

For the three sensor bias faults, pick features from a manual inspection of the scatter plot.

```
figure;
I = strcmp(R,'Healthy') | strcmp(R,'PressureSensorBias') | strcmp(R,'SpeedSensorBias')
J = [1 4 5 8 9 12 13 16 17 20]; % selected feature indices
fprintf('Selected features for sensors'' bias: %s\n',strjoin(FeatureTable.Properties.Va
```

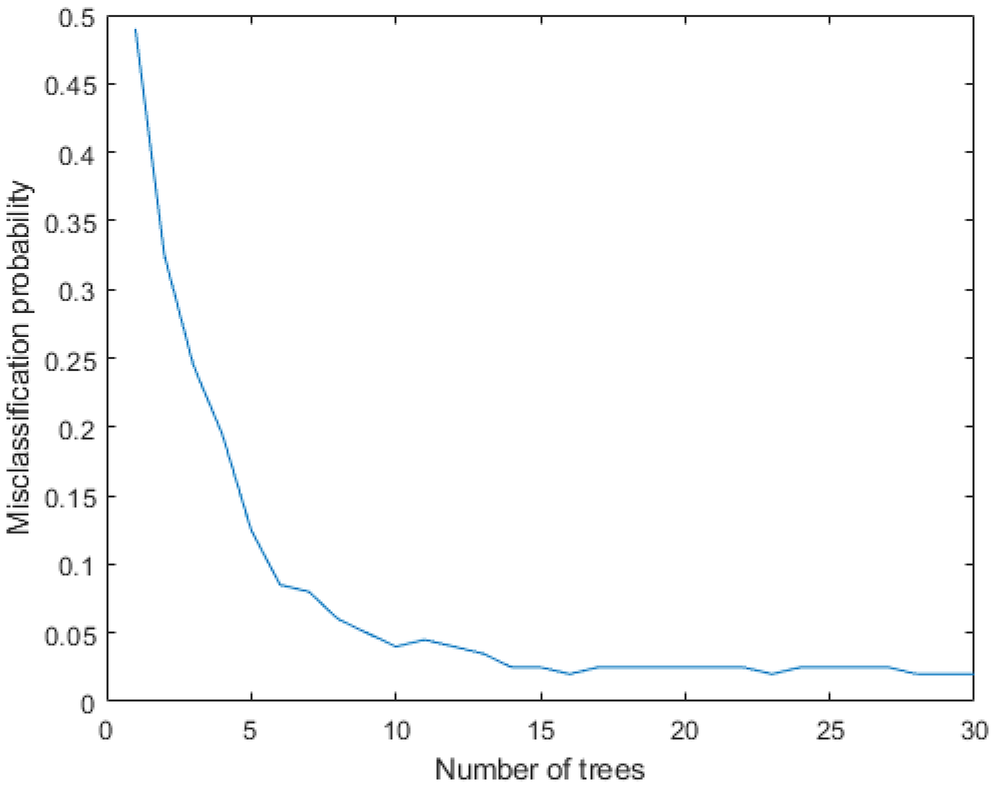
```
Selected features for sensors' bias: Mean1, Mean4, Max1, Max4, Kurtosis1, Kurtosis4, Va
```

```
gplotmatrix(P(I,J),[],R(I))
```



Train a 30 member Tree Bagger classifier for the reduced set of faults (sensor biases only) using a reduced set of features.

```
rng(9) % for reproducibility
Mdl = TreeBagger(30, FeatureTable(I,[J 21]), 'Condition',...
   'OOBPrediction','on',...
   'OOBPredictorImportance','on');
figure
plot(oobError(Mdl))
xlabel('Number of trees')
ylabel('Misclassification probability')
```

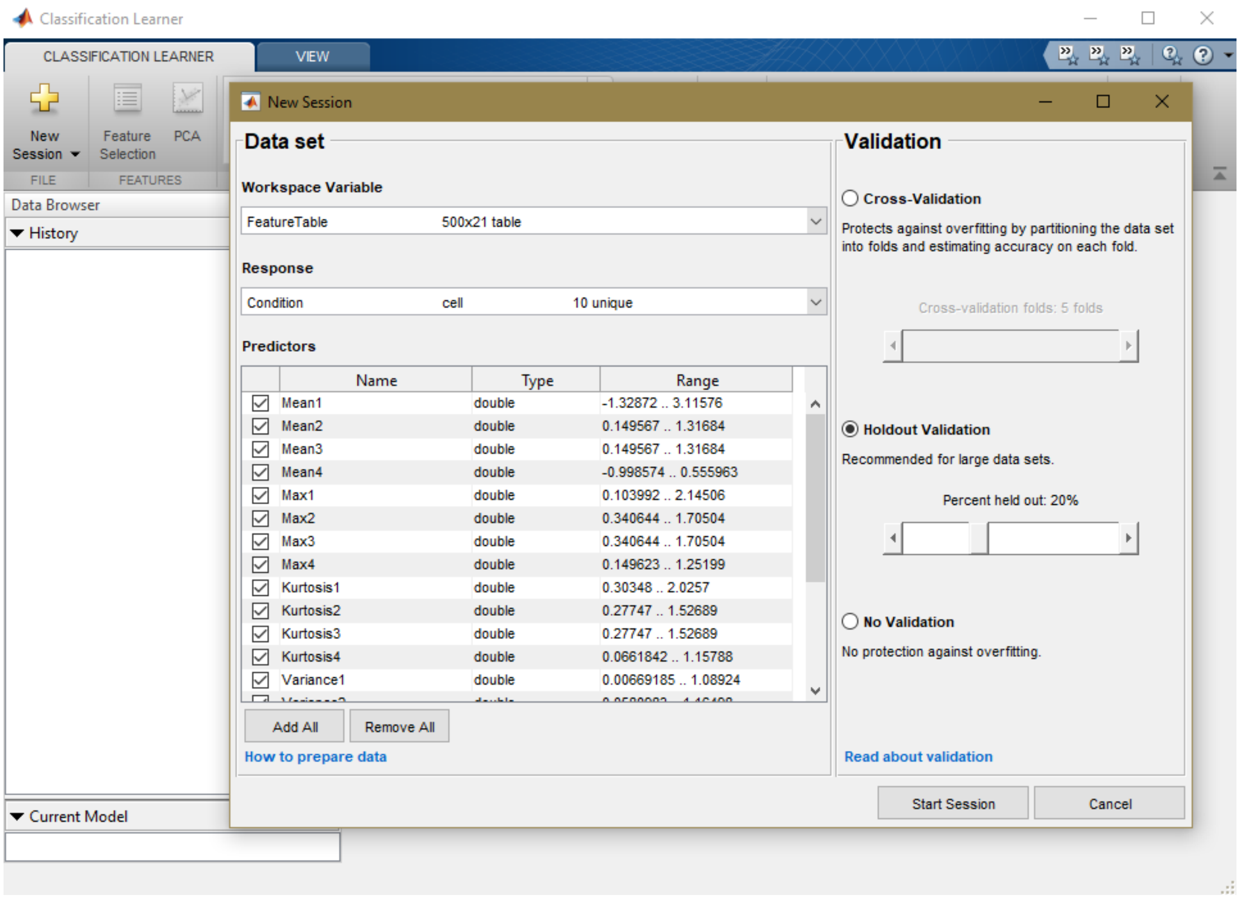


The misclassification error is less than 5%. Thus it is possible to pick a smaller set of features for certain subset of faults.

B. Multi-class Classification using Classification Learner App

The previous section focused on manual inspection of scatter plots to reduce the feature set for particular fault types. This approach can get tedious and may not cover all fault types. Can you design a classifier that can handle all fault modes in a more automated fashion? There are many classifiers available in Statistics and Machine Learning Toolbox. A quick way to try many of them and compare their performances is to use the Classification Learner App.

1 Launch the Classification Learner App and select FeatureTable from workspace as working data for a new session. Set aside 20% of data (10 samples of each mode) for holdout validation.



Data set

Workspace Variable

FeatureTable 500x21 table

Response

Condition cell 10 unique

Predictors

	Name	Type	Range
✓	Mean1	double	-1.32872 .. 3.11576
✓	Mean2	double	0.149567 .. 1.31684
✓	Mean3	double	0.149567 .. 1.31684
✓	Mean4	double	-0.998574 .. 0.555963
✓	Max1	double	0.103992 .. 2.14506
✓	Max2	double	0.340644 .. 1.70504
✓	Max3	double	0.340644 .. 1.70504
✓	Max4	double	0.149623 .. 1.25199
✓	Kurtosis1	double	0.30348 .. 2.0257
✓	Kurtosis2	double	0.27747 .. 1.52689
✓	Kurtosis3	double	0.27747 .. 1.52689
✓	Kurtosis4	double	0.0661842 .. 1.15788
✓	Variance1	double	0.00669185 .. 1.08924

Add All Remove All

How to prepare data

Validation

Cross-Validation

Protects against overfitting by partitioning the data set into folds and estimating accuracy on each fold.

Cross-validation folds: 5 folds

[✓ Holdout Validation]

Recommended for large data sets.

Percent held out: 20%

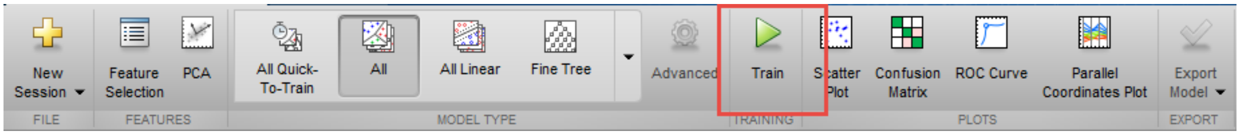
No Validation

No protection against overfitting.

Read about validation

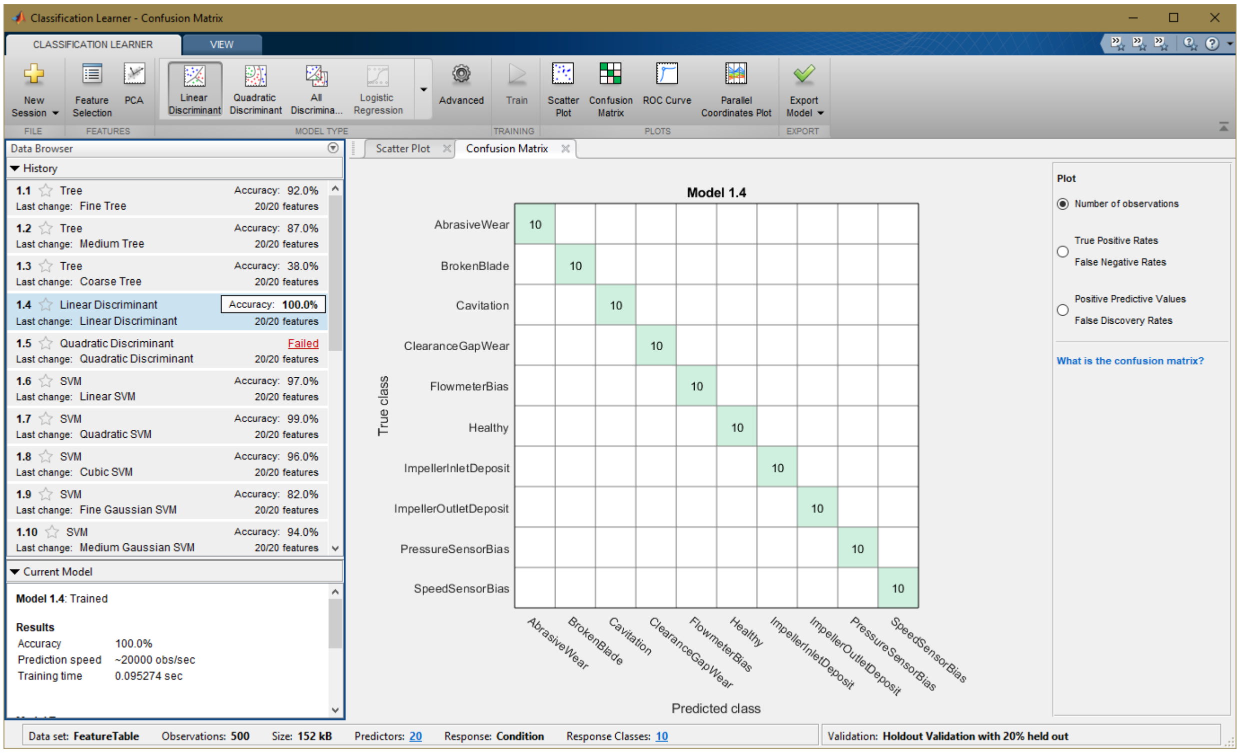
Start Session Cancel

2 Select **All** under *Model Type* section of the main tab. Then press the **Train** button.



3 In a short time, about 20 classifiers are trained. Their accuracy are displayed next to their names under the history panel. A Linear Discriminant classifier performs the best, producing 100% accuracy on the hold out samples.

4 To get a graphical view of the performance, open a Confusion Matrix plot from the *Plots* section of the main tab. The plot show the performance of the selected classifier (the Linear Discriminant classifier here).



1.1 Tree — Accuracy: 92.0%
Last change: Fine Tree — 20/20 features

1.2 Tree — Accuracy: 87.0%
Last change: Medium Tree — 20/20 features

1.3 Tree — Accuracy: 38.0%
Last change: Coarse Tree — 20/20 features

1.4 Linear Discriminant — Accuracy: 100.0%
Last change: Linear Discriminant — 20/20 features

1.5 Quadratic Discriminant — Failed
Last change: Quadratic Discriminant — 20/20 features

1.6 SVM — Accuracy: 97.0%
Last change: Linear SVM — 20/20 features

1.7 SVM — Accuracy: 99.0%
Last change: Quadratic SVM — 20/20 features

1.8 SVM — Accuracy: 96.0%
Last change: Cubic SVM — 20/20 features

1.9 SVM — Accuracy: 82.0%
Last change: Fine Gaussian SVM — 20/20 features

1.10 SVM — Accuracy: 94.0%
Last change: Medium Gaussian SVM — 20/20 features

Model 1.4: Trained

Results

Accuracy	100.0%
Prediction speed	~20000 obs/sec
Training time	0.095274 sec

	AbrasiveWear	BrokenBlade	Cavitation	ClearanceGapWear	FlowmeterBias	Healthy	ImpellerInletDeposit	ImpellerOutletDeposit	PressureSensorBias	SpeedSensorBias
AbrasiveWear	10									
BrokenBlade		10								
Cavitation			10							
ClearanceGapWear				10						
FlowmeterBias					10					
Healthy						10				
ImpellerInletDeposit							10			
ImpellerOutletDeposit								10		
PressureSensorBias									10	
SpeedSensorBias										10

Data set: FeatureTable Observations: 500 Size: 152 kB Predictors: 20 Response: Condition Response Classes: 10 Validation: Holdout Validation with 20% held out

Export the best performing classifier to the workspace and use it for prediction on new measurements.

Summary

A well designed fault diagnosis strategy can save operating costs by minimizing service downtime and component replacement costs. The strategy benefits from a good knowledge about the operating machine's dynamics which is used in combination with sensor measurements to detect and isolate different kinds of faults. This example described a residual based approach for fault diagnosis of centrifugal pumps. This approach is a good alternative to parameter estimation and tracking based approaches when the modeling task is complex and model parameters show dependence on operating conditions.

A residual based fault diagnosis approach involves the following steps:

1. Model the dynamics between the measurable inputs and outputs of the system using physical considerations or black box system identification techniques.
2. Compute residues as difference between measured and model produced signals. The residues may need to be further filtered to improve fault isolability.
3. Extract features such as peak amplitude, power, kurtosis etc from each residual signal.
4. Use features for fault detection and classification using anomaly detection and classification techniques.
5. Not all residues and derived features are sensitive to every fault. A view of feature histograms and scatter plots can reveal which features are suitable for detecting a certain fault type. This process of picking features and assessing their performance for fault isolation can be an iterative procedure.

References

1. Isermann, Rolf, *Fault-Diagnosis Applications. Model-Based Condition Monitoring: Actuators, Drives, Machinery, Plants, Sensors, and Fault-tolerant System*, Edition 1, Springer-Verlag Berlin Heidelberg, 2011.

Supporting Functions

Static pump equation parameter estimation

```
function [x1, x2, dpest] = staticPumpEst(w, dp, I)
%staticPumpEst Static pump parameter estimation in a varying speed setting
% I: sample indices for the selected operating region.

w1 = [0; w(I)];
```

```
dp1 = [0; dp(I)];
R1 = [w1.^2 w1];
x = pinv(R1)*dp1;
x1 = x(1);
x2 = x(2);

dpest = R1(2:end,:)*x;
end
```

Dynamic pipe parameter estimation

```
function [x3, x4, x5, Qest] = dynamicPipeEst(dp, Q, I)
%dynamicPipeEst Dynamic pipe parameter estimation in a varying speed setting
% I: sample indices for the selected operating region.

Q = Q(I);
dp = dp(I);
R1 = [0; Q(1:end-1)];
R2 = dp; R2(R2<0) = 0; R2 = sqrt(R2);
R = [ones(size(R2)), R2, R1];

% Remove out-of-regime samples
ii = find(I);
j = find(diff(ii)~=1);
R = R(2:end,:); R(j,:) = [];
y = Q(2:end); y(j) = [];
x = R\y;

x3 = x(1);
x4 = x(2);
x5 = x(3);

Qest = R*x;
end
```

Dynamic, multi-operating mode simulation of pump-pipe model using LPV System block.

```
function Qest = simulatePumpPipeModel(Ts,th3,th4,th5)
%simulatePumpPipeModel Piecewise linear modeling of dynamic pipe system.
% Ts: sample time
% w: Pump rotational speed
% th1, th2, th3 are estimated model parameters for the 3 regimes.
% This function requires Control System Toolbox.

ss1 = ss(th5(1),th4(1),th5(1),th4(1),Ts);
```

```
ss2 = ss(th5(2),th4(2),th5(2),th4(2),Ts);
ss3 = ss(th5(3),th4(3),th5(3),th4(3),Ts);
offset = permute([th3(1),th3(2),th3(3)]',[3 2 1]);
OP = struct('Region',[1 2 3]');
sys = cat(3,ss1,ss2,ss3);
sys.SamplingGrid = OP;

assignin('base','sys',sys)
assignin('base','offset',offset)
mdl = 'LPV_pump_pipe';
sim(mdl);
Qest = logsout.get('Qest');
Qest = Qest.Values;
Qest = Qest.Data;
end
```

Identify a dynamic model for inverse pump dynamics.

```
function syse = identifyNonlinearARXModel(Mmot,w,Q,Ts,N)
%identifyNonlinearARXModel Identify a nonlinear ARX model for 2-input (w, Q), 1-output
% Inputs:
%  w: rotational speed
%  Q: Flow rate
%  Mmot: motor torque
%  N: number of data samples to use
% Outputs:
%  syse: Identified model
%
% This function uses NLARX estimator from System Identification Toolbox.

sys = idnlarx([2 2 1 0 1],'','CustomRegressors',{'u1(t-2)^2','u1(t)*u2(t-2)','u2(t)^2'
data = iddata(Mmot,[w Q],Ts);
opt = nlarxOptions;
opt.Focus = 'simulation';
opt.SearchOptions.MaxIterations = 500;
```

```
syse = nlarx(data(1:N),sys,opt);
end
```

See Also

More About

- “Decision Models for Fault Detection and Diagnosis” on page 4-2

Fault Detection Using an Extended Kalman Filter

This example shows how to use an extended Kalman filter for fault detection. The example uses an extended Kalman filter for online estimation of the friction of a simple DC motor. Significant changes in the estimated friction are detected and indicate a fault. This example uses functionality from System Identification Toolbox™, and does not require Predictive Maintenance Toolbox™.

Motor Model

The motor is modelled as an inertia J with damping coefficient c, driven by a torque u. The motor angular velocity w and acceleration $\dot{w}$, are the measured outputs.

$$\dot{w} = (u - cw)/J$$

To estimate the damping coefficient c using an extended Kalman filter, introduce an auxiliary state for the damping coefficient and set its derivative to zero.

$$\dot{c} = 0$$

Thus, the model state, x = [w;c], and measurement, y, equations are:

$$\begin{bmatrix} \dot{w} \\ \dot{c} \end{bmatrix} = \begin{bmatrix} (u - cw)/J \\ 0 \end{bmatrix}$$

$$y = \begin{bmatrix} w \\ (u - cw)/J \end{bmatrix}$$

The continuous-time equations are transformed to discrete time using the approximation $\dot{x} = \frac{x_{n+1} - x_n}{T_s}$, where Ts is the discrete sampling period. This gives the discrete-time model equations which are implemented in the `pdmMotorModelStateFcn.m` and `pdmMotorModelMeasurementFcn.m` functions.

$$\begin{bmatrix} w_{n+1} \\ c_{n+1} \end{bmatrix} = \begin{bmatrix} w_n + (u_n - c_n w_n) T_s/J \\ c_n \end{bmatrix}$$

$$y_n = \begin{bmatrix} w_n \\ (u_n - c_n w_n)/J \end{bmatrix}$$

Specify motor parameters.

```
J  = 10;    % Inertia
Ts = 0.01;  % Sample time
```

Specify initial states.

```
x0 = [...
    0; ...  % Angular velocity
    1];     % Friction

type pdmMotorModelStateFcn
type pdmMotorModelMeasurementFcn
```

```
function x1 = pdmMotorModelStateFcn(x0,varargin)
%PDMMOTORMODELSTATEFCN
%
% State update equations for a motor with friction as a state
%
%  x1 = pdmMotorModelStateFcn(x0,u,J,Ts)
%
%  Inputs:
%    x0 - initial state with elements [angular velocity; friction]
%    u  - motor torque input
%    J  - motor inertia
%    Ts - sampling time
%
%  Outputs:
%    x1 - updated states
%

%  Copyright 2016-2017 The MathWorks, Inc.

% Extract data from inputs
u  = varargin{1};   % Input
J  = varargin{2};   % System innertia
Ts = varargin{3};   % Sample time

% State update equation
x1 = [...
    x0(1)+Ts/J*(u-x0(1)*x0(2)); ...
    x0(2)];
end
```

```
function y = pdmMotorModelMeasurementFcn(x,varargin)
%PDMMOTORMODELMEASUREMENTFCN
%
% Measurement equations for a motor with friction as a state
%
%  y = pdmMotorModelMeasurementFcn(x0,u,J,Ts)
%
%  Inputs:
%    x  - motor state with elements [angular velocity; friction]
%    u  - motor torque input
%    J  - motor inertia
%    Ts - sampling time
%
%  Outputs:
%    y - motor measurements with elements [angular velocity; angular acceleration]
%

%  Copyright 2016-2017 The MathWorks, Inc.

% Extract data from inputs
u  = varargin{1};   % Input
J  = varargin{2};   % System innertia

% Output equation
y = [...
    x(1); ...
    (u-x(1)*x(2))/J];
end
```

The motor experiences state (process) noise disturbances, q, and measurement noise disturbances, r. The noise terms are additive.

$$\begin{bmatrix} w_{n+1} \\ c_{n+1} \end{bmatrix} = \begin{bmatrix} w_n + (u_n - c_n w_n) T_s / J \\ c_n \end{bmatrix} + q$$

$$y_n = \begin{bmatrix} w_n \\ (u_n - c_n w_n) / J \end{bmatrix} + r$$

The process and measurement noise have zero mean, `E[q]=E[r]=0`, and covariances `Q = E[qq']` and `R = E[rr']`. The friction state has a high process noise disturbance. This reflects the fact that we expect the friction to vary during normal operation of the motor and want the filter to track this variation. The acceleration and velocity state noise is low but the velocity and acceleration measurements are relatively noisy.

Specify the process noise covariance.

```
Q = [...
    1e-6 0; ...   % Angular velocity
    0 1e-2];      % Friction
```

Specify the measurement noise covariance.

```
R = [...
    1e-4 0; ...  % Velocity measurement
    0 1e-4];     % Acceleration measurement
```

Creating an Extended Kalman Filter

Create an extended Kalman Filter to estimate the states of the model. We are particularly interested in the damping state because dramatic changes in this state value indicate a fault event.

Create an `extendedKalmanFilter` object, and specify the Jacobians of the state transition and measurement functions.

```
ekf = extendedKalmanFilter(...
    @pdmMotorModelStateFcn, ...
    @pdmMotorModelMeasurementFcn, ...
    x0,...
    'StateCovariance',            [1 0; 0 1000], ...[1 0 0; 0 1 0; 0 0 100], ...
    'ProcessNoise',               Q, ...
    'MeasurementNoise',           R, ...
    'StateTransitionJacobianFcn', @pdmMotorModelStateJacobianFcn, ...
    'MeasurementJacobianFcn',     @pdmMotorModelMeasJacobianFcn);
```

The extended Kalman filter has as input arguments the state transition and measurement functions defined previously. The initial state value `x0`, initial state covariance, and process and measurement noise covariances are also inputs to the extended Kalman filter. In this example, the exact Jacobian functions can be derived from the state transition function `f`, and measurement function `h`:

$$\frac{\partial}{\partial x} f = \begin{bmatrix} 1 - T_s c_n / J & -T_s w_n / J \\ 0 & 1 \end{bmatrix}$$

$$\frac{\partial}{\partial x} h = \begin{bmatrix} 1 & 0 \\ -c_n / J & -w_n / J \end{bmatrix}$$

The state Jacobian is defined in the `pdmMotorModelStateJacobianFcn.m` function and the measurement Jacobian is defined in the `pdmMotorModelMeasJacobianFcn.m` function.

```
type pdmMotorModelStateJacobianFcn
type pdmMotorModelMeasJacobianFcn
```

```
function Jac = pdmMotorModelStateJacobianFcn(x,varargin)
%PDMMOTORMODELSTATEJACOBIANFCN
%
% Jacobian of motor model state equations. See pdmMotorModelStateFcn for
% the model equations.
%
%  Jac = pdmMotorModelJacobianFcn(x,u,J,Ts)
%
%  Inputs:
%    x  - state with elements [angular velocity; friction]
%    u  - motor torque input
%    J  - motor inertia
%    Ts - sampling time
%
%  Outputs:
%    Jac - state Jacobian computed at x
%

%  Copyright 2016-2017 The MathWorks, Inc.

% Model properties
J  = varargin{2};
Ts = varargin{3};

% Jacobian
Jac = [...
    1-Ts/J*x(2) -Ts/J*x(1); ...
    0 1];
end

function J = pdmMotorModelMeasJacobianFcn(x,varargin)
%PDMMOTORMODELMEASJACOBIANFCN
%
% Jacobian of motor model measurement equations. See
% pdmMotorModelMeasurementFcn for the model equations.
%
%  Jac = pdmMotorModelMeasJacobianFcn(x,u,J,Ts)
```

```
%
%  Inputs:
%    x  - state with elements [angular velocity; friction]
%    u  - motor torque input
%    J  - motor inertia
%    Ts - sampling time
%
%  Outputs:
%    Jac - measurement Jacobian computed at x
%

%  Copyright 2016-2017 The MathWorks, Inc.

% System parameters
J  = varargin{2};   % System innertia

% Jacobian
J = [ ...
    1 0;
    -x(2)/J -x(1)/J];
end
```

Simulation

To simulate the plant, create a loop and introduce a fault in the motor (a dramatic change in the motor fiction). Within the simulation loop, use the extended Kalman filter to estimate the motor states and to specifically track the friction state to detect when there is a statistically significant change in friction.

The motor is simulated with a pulse train that repeatedly accelerates and decelerates the motor. This type of motor operation is typical for a picker robot in a production line.

```
t  = 0:Ts:20;                  % Time, 20s with Ts sampling period
u  = double(mod(t,1)<0.5)-0.5; % Pulse train, period 1, 50% duty cycle
nt = numel(t);                 % Number of time points
nx = size(x0,1);               % Number of states
ySig = zeros([2, nt]);         % Measured motor outputs
xSigTrue = zeros([nx, nt]);    % Unmeasured motor states
xSigEst = zeros([nx, nt]);     % Estimated motor states
xstd = zeros([nx nx nt]);      % Standard deviation of the estimated states
ySigEst = zeros([2, nt]);      % Estimated model outputs
fMean = zeros(1,nt);           % Mean estimated friction
fSTD = zeros(1,nt);            % Standard deviation of estimated friction
fKur = zeros(2,nt);            % Kurtosis of estimated friction
fChanged = false(1,nt);        % Flag indicating friction change detection
```

When simulating the motor, add process and measurement noise similar to the Q and R noise covariance values used when constructing the extended Kalman filter. For the friction, use a much smaller noise value because the friction is mostly constant except when the fault occurs. Artificially induce the fault during the simulation.

```
rng('default');
Qv = chol(Q);   % Standard deviation for process noise
Qv(end) = 1e-2; % Smaller friction noise
Rv = chol(R);   % Standard deviation for measurement noise
```

Simulate the model using the state update equation, and add process noise to the model states. Ten seconds into the simulation, force a change in the motor friction. Use the model measurement function to simulate the motor sensors, and add measurement noise to the model outputs.

```
for ct = 1:numel(t)

   % Model output update
   y = pdmMotorModelMeasurementFcn(x0,u(ct),J,Ts);
   y = y+Rv*randn(2,1);   % Add measurement noise
   ySig(:,ct) = y;

   % Model state update
   xSigTrue(:,ct) = x0;
   x1 = pdmMotorModelStateFcn(x0,u(ct),J,Ts);
   % Induce change in friction
   if t(ct) == 10
       x1(2) = 10;  % Step change
   end
   x1n = x1+Qv*randn(nx,1);  % Add process noise
   x1n(2) = max(x1n(2),0.1); % Lower limit on friction
   x0 = x1n; % Store state for next simulation iteration
```

To estimate the motor states from the motor measurements, use the `predict` and `correct` commands of the extended Kalman Filter.

```
   % State estimation using the Extended Kalman Filter
   x_corr = correct(ekf,y,u(ct),J,Ts); % Correct the state estimate based on current me
   xSigEst(:,ct) = x_corr;
   xstd(:,:,ct) = chol(ekf.StateCovariance);
   predict(ekf,u(ct),J,Ts);            % Predict next state given the current state anc
```

To detect changes in friction, compute the estimated friction mean and standard deviation using a 4 second moving window. After an initial 7-second period, lock the computed mean and standard deviation. This initially computed mean is the expected no-fault mean value for the friction. After 7 seconds, if the estimated friction is greater than 3 standard deviations away from the expected no-fault mean value, it signifies a significant change in the friction. To reduce the effect of noise and variability in the estimated friction, use the mean of the estimated friction when comparing to the 3-standard-deviations bound.

```
   if t(ct) < 7
       % Compute mean and standard deviation of estimated fiction.
       idx = max(1,ct-400):max(1,ct-1); % Ts = 0.01 seconds
       fMean(ct) = mean( xSigEst(2, idx) );
       fSTD(ct)  = std( xSigEst(2, idx) );
   else
       % Store the computed mean and standard deviation without
       % recomputing.
       fMean(ct) = fMean(ct-1);
       fSTD(ct)  = fSTD(ct-1);
       % Use the expected friction mean and standard deviation to detect
       % friction changes.
       estFriction = mean(xSigEst(2,max(1,ct-10):ct));
       fChanged(ct) = (estFriction > fMean(ct)+3*fSTD(ct)) || (estFriction < fMean(ct)
   end
   if fChanged(ct) && ~fChanged(ct-1)
       % Detect a rising edge in the friction change signal |fChanged|.
       fprintf('Significant friction change at %f\n',t(ct));
   end
```

```
Significant friction change at 10.450000
```

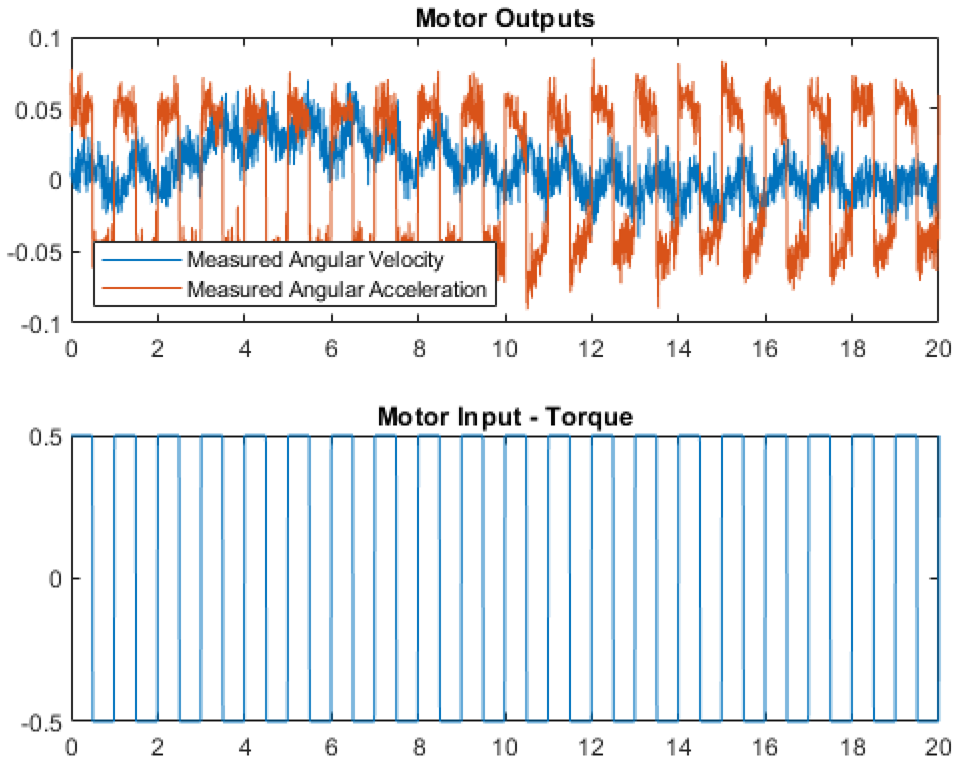
Use the estimated state to compute the estimated output. Compute the error between the measured and estimated outputs, and calculate the error statistics. The error statistics can be used for detecting the friction change. This is discussed in more detail later.

```
   ySigEst(:,ct) = pdmMotorModelMeasurementFcn(x_corr,u(ct),J,Ts);
   idx = max(1,ct-400):ct;
   fKur(:,ct) = [...
       kurtosis(ySigEst(1,idx)-ySig(1,idx)); ...
       kurtosis(ySigEst(2,idx)-ySig(2,idx))];

end
```

Extended Kalman Filter Performance

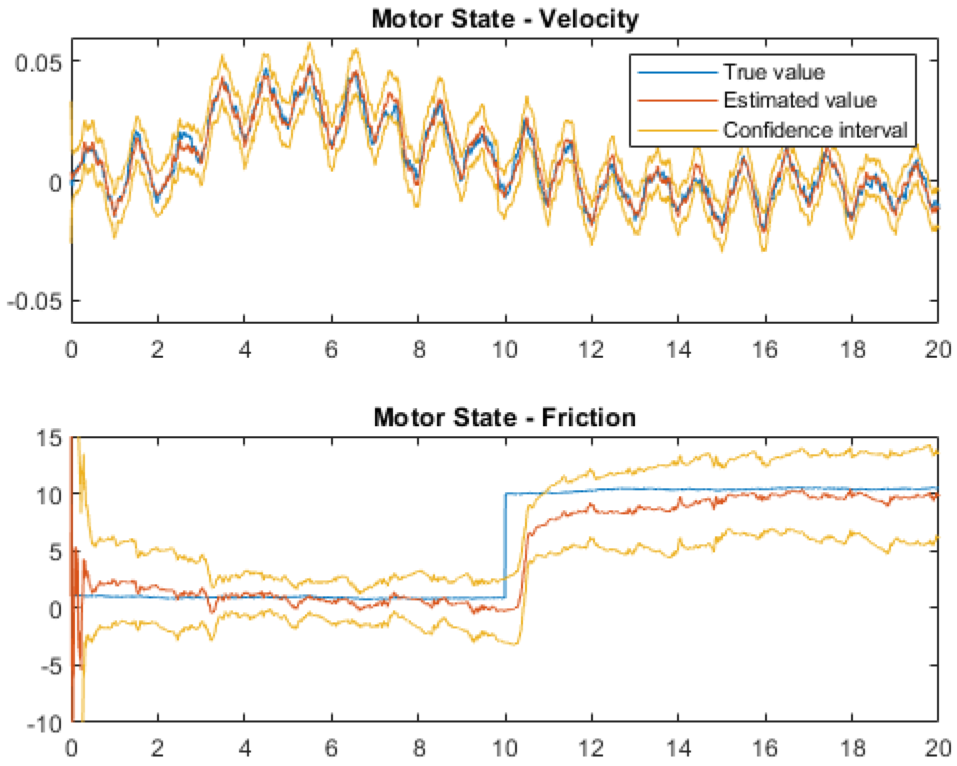
Note that a friction change was detected at 10.45 seconds. We now describe how this fault-detection rule was derived. First examine the simulation results and filter performance.

```
figure,
subplot(211), plot(t,ySig(1,:),t,ySig(2,:));
title('Motor Outputs')
legend('Measured Angular Velocity','Measured Angular Acceleration', 'Location','SouthWe
subplot(212), plot(t,u);
title('Motor Input - Torque')
```



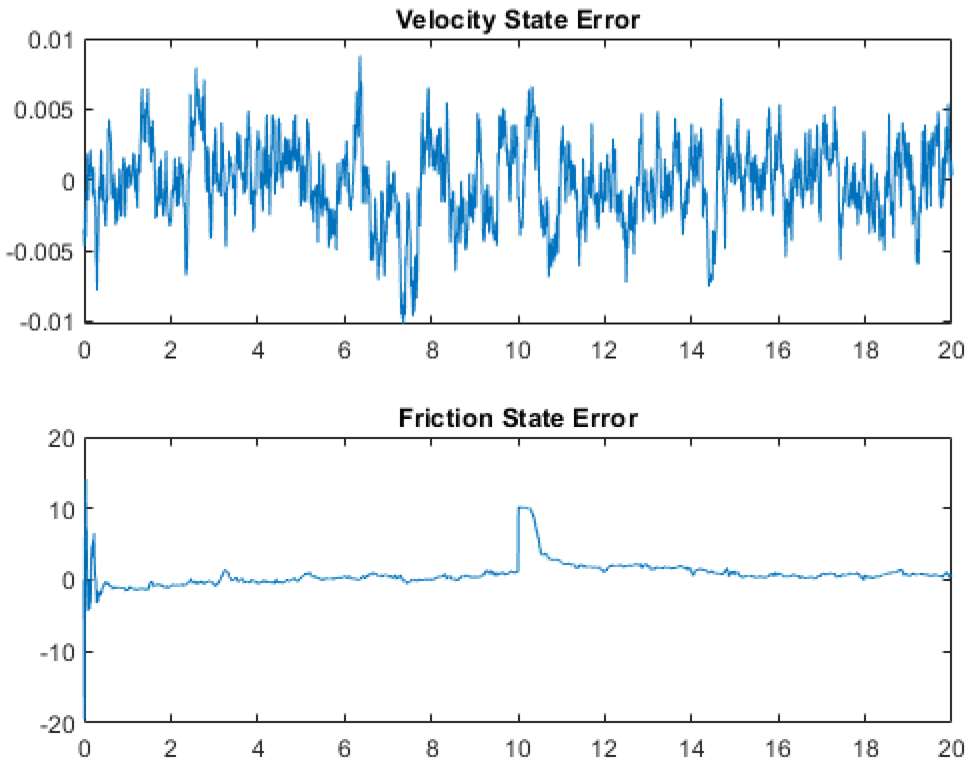
The model input-output responses indicate that it is difficult to detect the friction change directly from the measured signals. The extended Kalman filter enables us to estimate the states, in particular the friction state. Compare the true model states and estimated states. The estimated states are shown with confidence intervals corresponding to 3 standard deviations.

```
figure,
subplot(211),plot(t,xSigTrue(1,:), t,xSigEst(1,:), ...
    [t nan t],[xSigEst(1,:)+3*squeeze(xstd(1,1,:))', nan, xSigEst(1,:)-3*squeeze(xstd(1
axis([0 20 -0.06 0.06]),
legend('True value','Estimated value','Confidence interval')
title('Motor State - Velocity')
subplot(212),plot(t,xSigTrue(2,:), t,xSigEst(2,:),  ...
    [t nan t],[xSigEst(2,:)+3*squeeze(xstd(2,2,:))' nan xSigEst(2,:)-3*squeeze(xstd(2,2
axis([0 20 -10 15])
title('Motor State - Friction');
```



Note that the filter estimate tracks the true values, and that the confidence intervals remain bounded. Examining the estimation errors provide more insight into the filter behavior.

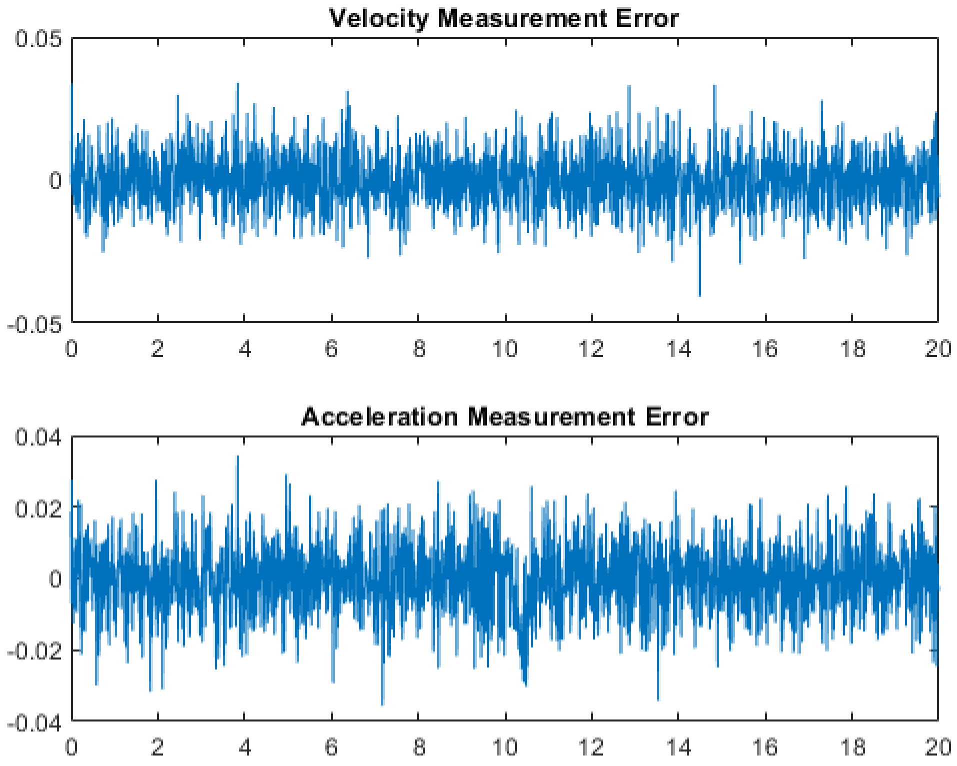
```
figure,
subplot(211),plot(t,xSigTrue(1,:)-xSigEst(1,:))
title('Velocity State Error')
subplot(212),plot(t,xSigTrue(2,:)-xSigEst(2,:))
title('Friction State Error')
```



The error plots show that the filter adapts after the friction change at 10 seconds and reduces the estimation errors to zero. However, the error plots cannot be used for fault detection as they rely on knowing the true states. Comparing the measured state value to the estimated state values for acceleration and velocity could provide a detection mechanism.

```
figure
subplot(211), plot(t,ySig(1,:)-ySigEst(1,:))
```

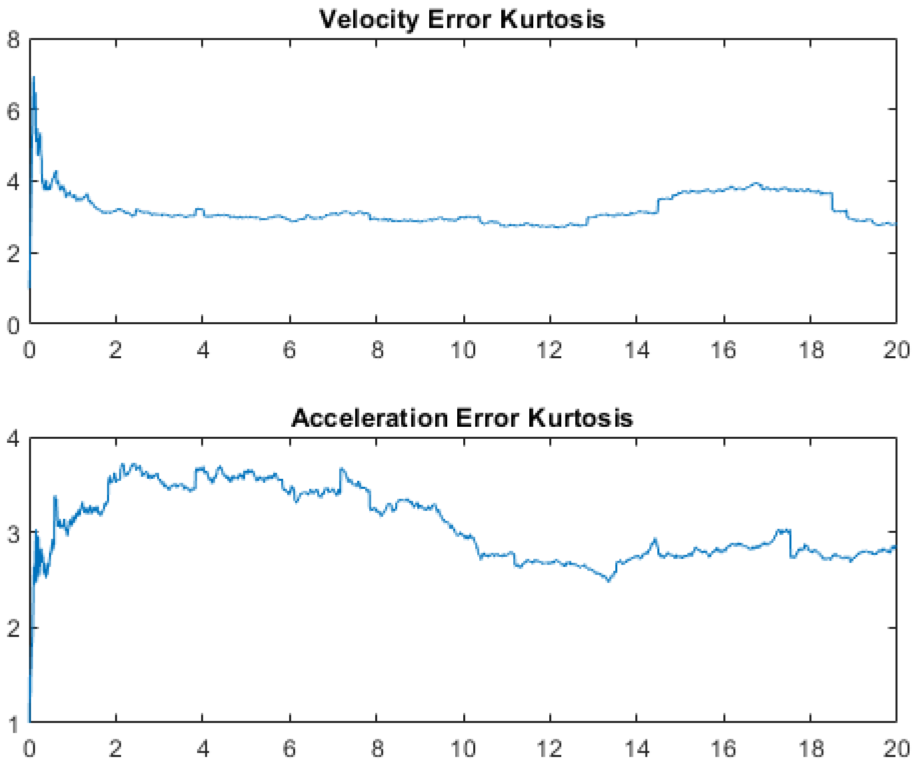
```
title('Velocity Measurement Error')
subplot(212),plot(t,ySig(2,:)-ySigEst(2,:))
title('Acceleration Measurement Error')
```



The acceleration error plot shows a minor difference in mean error around 10 seconds when the fault is introduced. View the error statistics to see if the fault can be detected from the computed errors. The acceleration and velocity errors are expected to be normally distributed (the noise models are all Gaussian). Therefore, the kurtosis of the acceleration error may help identify when the error distribution change from symmetrical to asymmetrical due to the friction change and resulting change in error distribution.

```
figure,
subplot(211),plot(t,fKur(1,:))
title('Velocity Error Kurtosis')
```

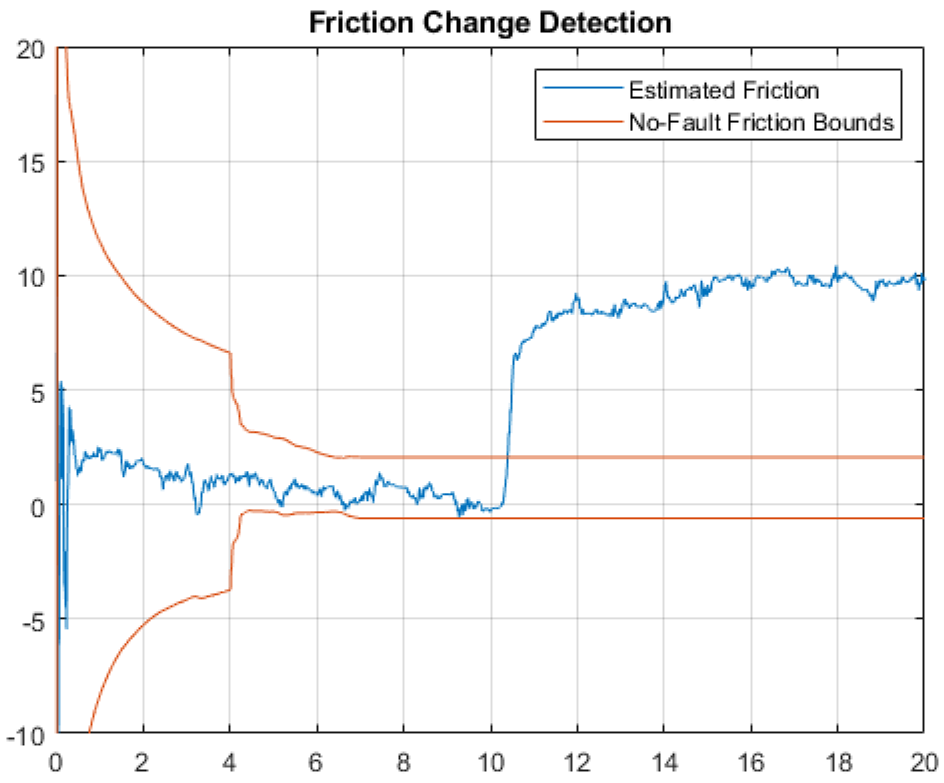
```
subplot(212),plot(t,fKur(2,:))
title('Acceleration Error Kurtosis')
```



Ignoring the first 4 seconds when the estimator is still converging and data is being collected, the kurtosis of the errors is relatively constant with minor variations around 3 (the expected kurtosis value for a Gaussian distribution). Thus, the error statistics cannot be used to automatically detect friction changes in this application. Using the kurtosis of the errors is also difficult in this application as the filter is adapting and continually driving the errors to zero, only giving a short time window where the error distributions differ from zero.

Thus in this application, using the changes in estimated friction provide the best way to automatically detect faults in the motor. The friction estimates (mean and standard deviation) from known no-fault data provide expected bounds for the friction and it is easy to detect when these bounds are violated. The following plot highlights this fault-detection approach.

```
figure
plot(t,xSigEst(2,:),[t nan t],[fMean+3*fSTD,nan,fMean-3*fSTD])
title('Friction Change Detection')
legend('Estimated Friction','No-Fault Friction Bounds')
axis([0 20 -10 20])
grid on
```



Summary

This example has shown how to use an extended Kalman filter to estimate the friction in a simple DC motor and use the friction estimate for fault detection.

See Also

More About

- "Model-Based Condition Indicators" on page 3-8

Fault Detection Using Data Based Models

This example shows how to use a data-based modeling approach for fault detection. This example uses functionality from Signal Processing Toolbox™ and System Identification Toolbox™, and does not require Predictive Maintenance Toolbox™.

Introduction

Early detection and isolation of anomalies in a machine's operation can help to reduce accidents, reduce downtime and thus save operational costs. The approach involves processing live measurements from a system's operation to flag any unexpected behavior that would point towards a newly developed fault.

This example explores the following fault diagnosis aspects:

1. Detection of abnormal system behavior by residual analysis
2. Detection of deterioration by building models of a damaged system
3. Tracking system changes using online adaptation of model parameters

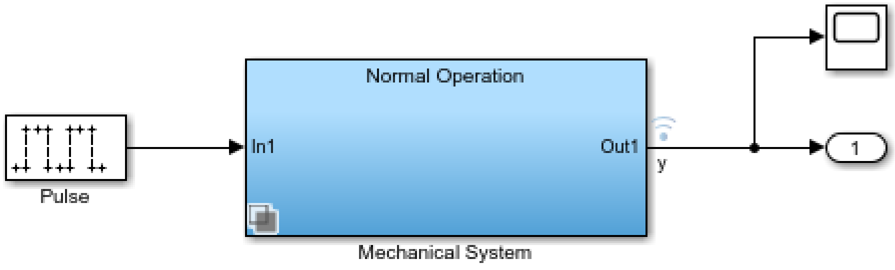
Identifying a Dynamic Model of System Behavior

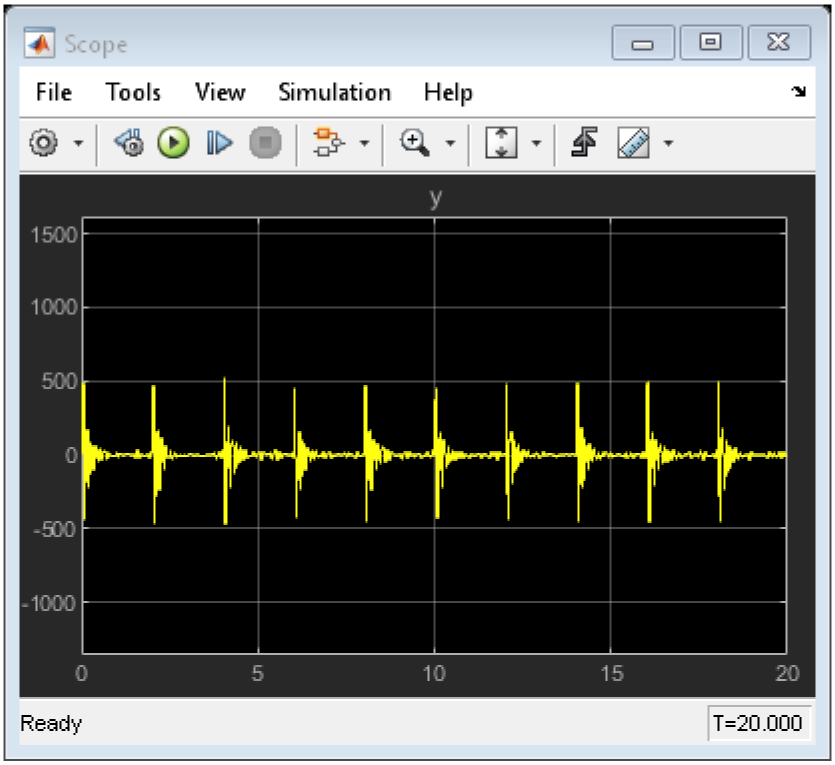
In a model based approach to detection, a dynamic model of the concerned system is first built using measured input and output data. A good model is able to accurately predict the response of the system for a certain future time horizon. When the prediction is not good, the residuals may be large and could contain correlations. These aspects are exploited to detect the incidence of failure.

Consider a building subject to impacts and vibrations. The source of vibrations can be different types of stimuli depending upon the system such as wind gusts, contact with running engines and turbines, or ground vibrations. The impacts are a result of impulsive bump tests on the system that are added to excite the system sufficiently. Simulink model `idMechanicalSystem.slx` is a simple example of such a structure. The excitation comes from periodic bumps as well as ground vibrations modeled by filtered white noise. The output of the system is collected by a sensor that is subject to measurement noise. The model is able to simulate various scenarios involving the structure in a healthy or a damaged state.

```
sysA = 'pdmMechanicalSystem';
open_system(sysA)
% Set the model in the healthy mode of operation
set_param([sysA,'/Mechanical System'],'OverrideUsingVariant','Normal')
```

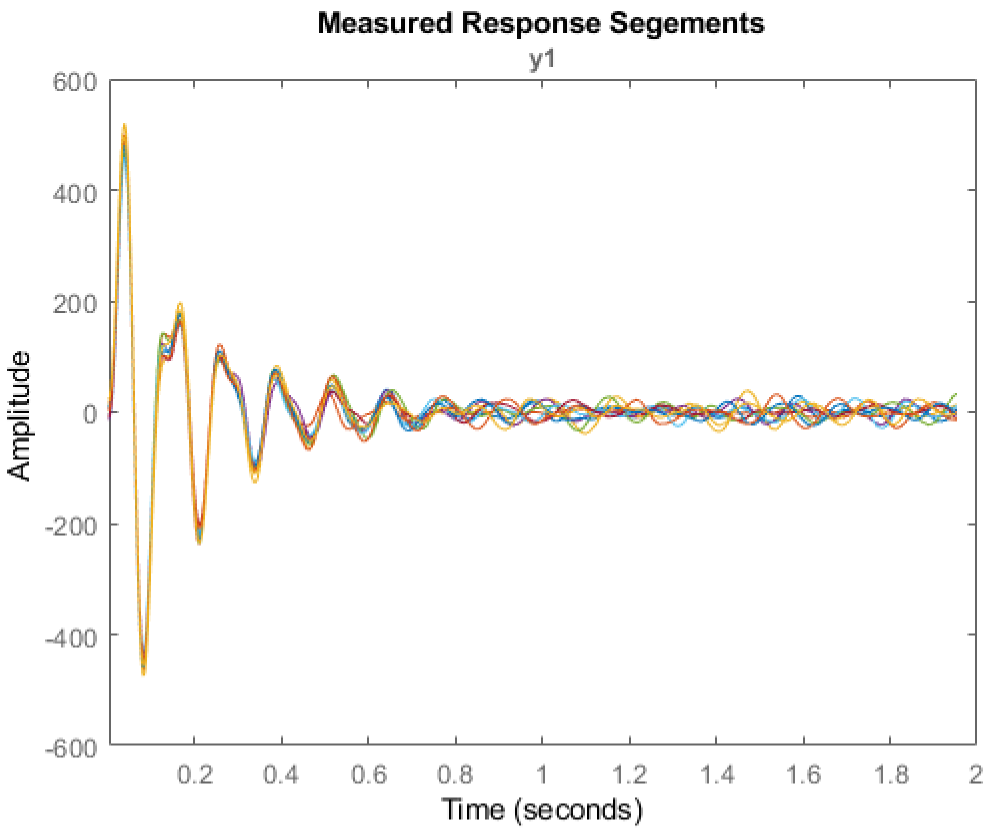
```
% Simulate the system and log the response data
sim(sysA)
ynormal = logsout.getElement('y').Values;
```





The input signal was not measured; all we have recorded is the response `ynormal`. Hence we build a dynamic model of the system using "blind identification" techniques. In particular, we build an ARMA model of the recorded signal as a representation of the system. This approach works when the input signal is assumed to be (filtered) white noise. Since the data is subject to periodic bumps, we split the data into several pieces each starting at the incidence of a bump. This way, each data segment contains the response to one bump plus random excitations - a situation that can be captured using a time series model, where the effect of the bump is attributed to suitable initial conditions.

```
Ts = 1/256;  % data sample time
nr = 10;     % number of bumps in the signal
N = 512;     % length of data between bumps
znormal = cell(nr,1);
for ct = 1:nr
   ysegment = ynormal.Data((ct-1)*N+(1:500));
   z = iddata(ysegment,[],Ts);
   znormal{ct} = z;  % each segment has only one bump
end
plot(znormal{:}) % plot a sampling of the recorded segments
title('Measured Response Segements')
```



Split the data into estimation and validation pieces.

```
ze = merge(znormal{1:5});
zv = merge(znormal{6:10});
```

Estimate a 7th order time-series model in state-space form using the `ssest()` command. The model order was chosen by cross validation (checking the fit to validation data) and residual analysis (checking that residuals are uncorrelated).

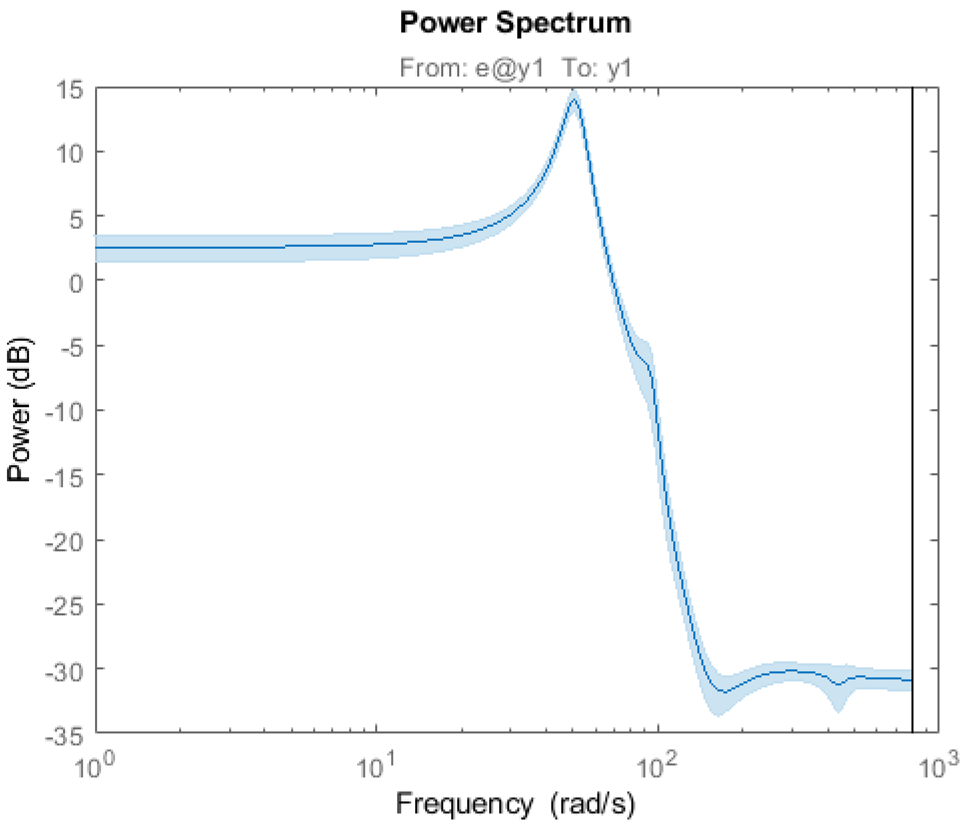
```
nx = 7;
model = ssest(ze, nx, 'form', 'canonical', 'Ts', Ts);
present(model)  % view model equations with parameter uncertainty
```

```
model =
Discrete-time identified state-space model:
  x(t+Ts) = A x(t) + K e(t)
     y(t) = C x(t) + e(t)

A =
                     x1                   x2                   x3
 x1                   0                    1                    0
 x2                   0                    0                    1
 x3                   0                    0                    0
 x4                   0                    0                    0
 x5                   0                    0                    0
 x6                   0                    0                    0
 x7  0.5548 +/- 0.04606   -2.713 +/- 0.2198    5.885 +/- 0.4495

                     x4                   x5                   x6
 x1                   0                    0                    0
 x2                   0                    0                    0
 x3                   1                    0                    0
 x4                   0                    1                    0
 x5                   0                    0                    1
 x6                   0                    0                    0
 x7    -8.27 +/- 0.5121    9.234 +/- 0.3513   -7.956 +/- 0.1408

                     x7
 x1                   0
 x2                   0
 x3                   0
 x4                   0
 x5                   0
 x6                   1
 x7   4.263 +/- 0.02599

C =
     x1  x2  x3  x4  x5  x6  x7
 y1   1   0   0   0   0   0   0

K =
                    y1
 x1  1.025 +/- 0.01401
 x2  1.444 +/-  0.0131
 x3  1.907 +/- 0.01271
 x4  2.385 +/- 0.01203
 x5  2.857 +/- 0.01456
 x6   3.26 +/-  0.0222
```

```
   x7  3.552 +/-  0.0336

Sample time: 0.0039063 seconds

Parameterization:
   CANONICAL form with indices: 7.
   Disturbance component: estimate
   Number of free coefficients: 14
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Termination condition: Near (local) minimum, (norm(g) < tol)..
Number of iterations: 7, Number of function evaluations: 15

Estimated using SSEST on time domain data "ze".
Fit to estimation data: [99.07 99.04 99.15 99.05 99.04]% (prediction focus)
FPE: 0.6242, MSE: [0.5974 0.6531 0.5991 0.5871 0.6496]
More information in model's "Report" property.
```

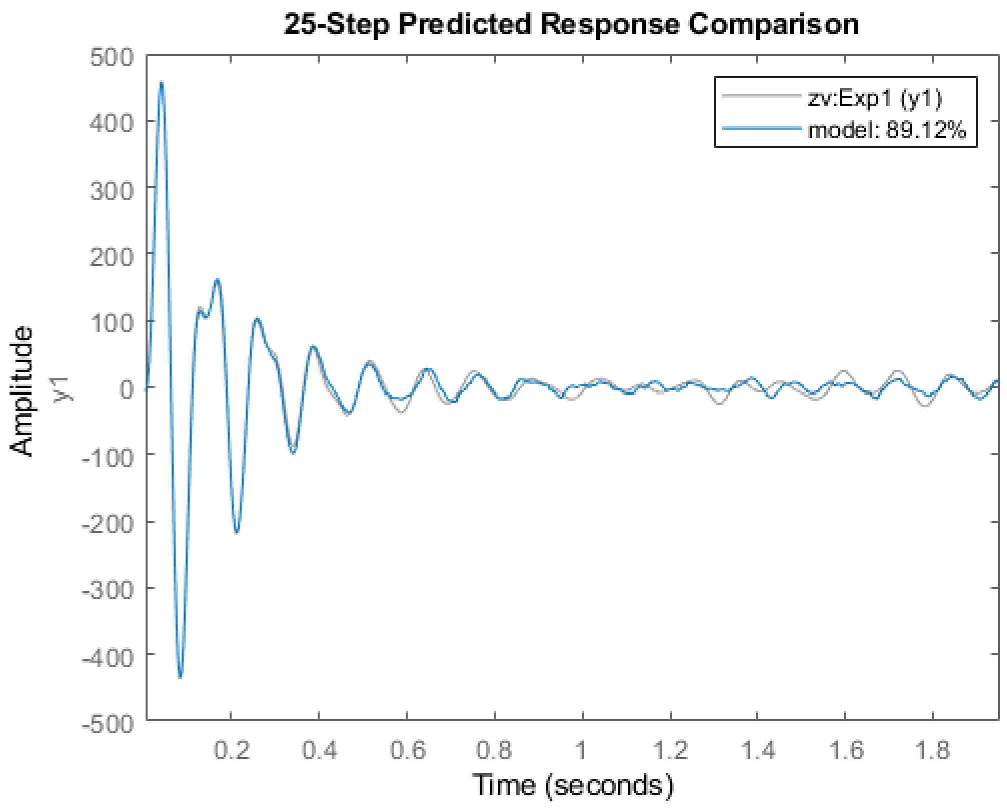
The model display shows relatively small uncertainty in parameter estimates. We can confirm the reliability by computing the 1-sd (99.73%) confidence bound on the estimated spectrum of the measured signal.

```
h = spectrumplot(model);
showConfidence(h, 3)
```



The confidence region is small, although there is about 30% uncertainty in the response at lower frequencies. The next step in validation is to see how well the model predicts the responses in the validation dataset `zv`. We use a 25-step ahead prediction horizon.

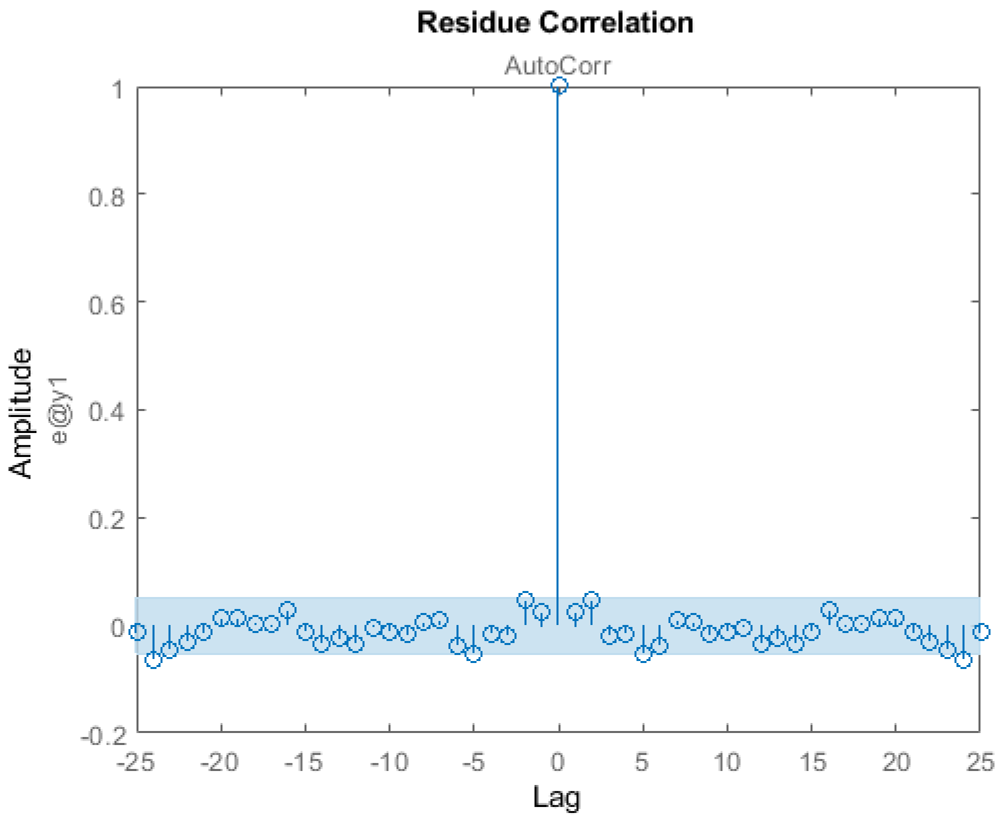
```
compare(zv, model, 25) % Validation against one dataset
```



The plot shows that the model is able to predict the response in the first experiment of the validation dataset 25 time steps (= 0.1 sec) in future with > 85% accuracy. To view the fit to other experiments in the dataset, use the right-click context menu of the plot axes.

The final step in validating the model is to analyze the residuals generated by it. For a good model, these residuals should be white, i.e., show statistically insignificant correlations for non-zero lags:

```
resid(model, zv)
```



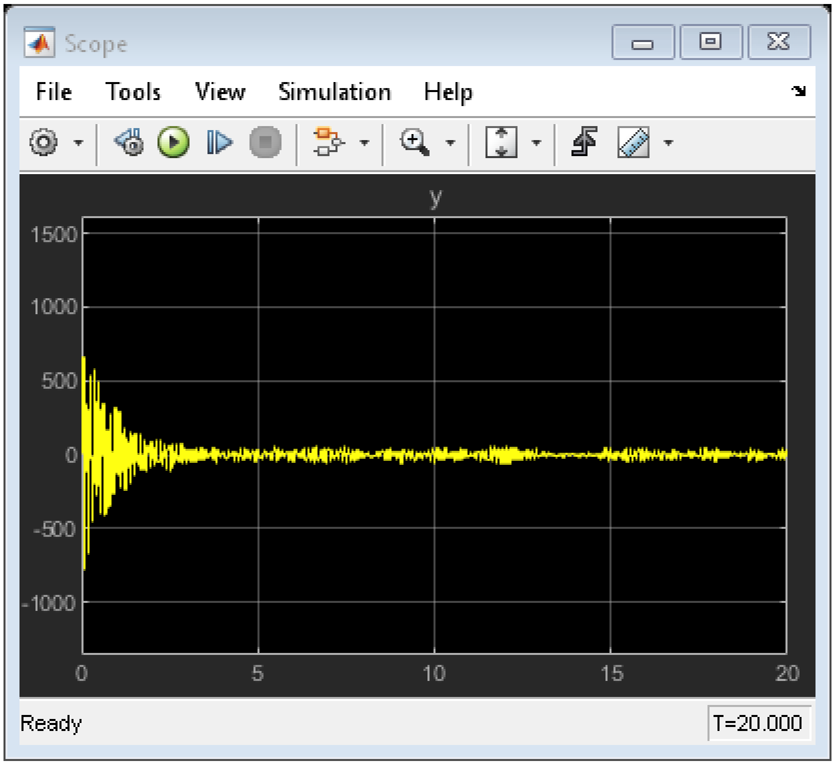
The residuals are mostly uncorrelated at nonzero lags. Having derived a model of the normal behavior we move on to investigate how the model can be used to detect faults.

Fault Detection by Residual Analysis Using Model of Healthy State

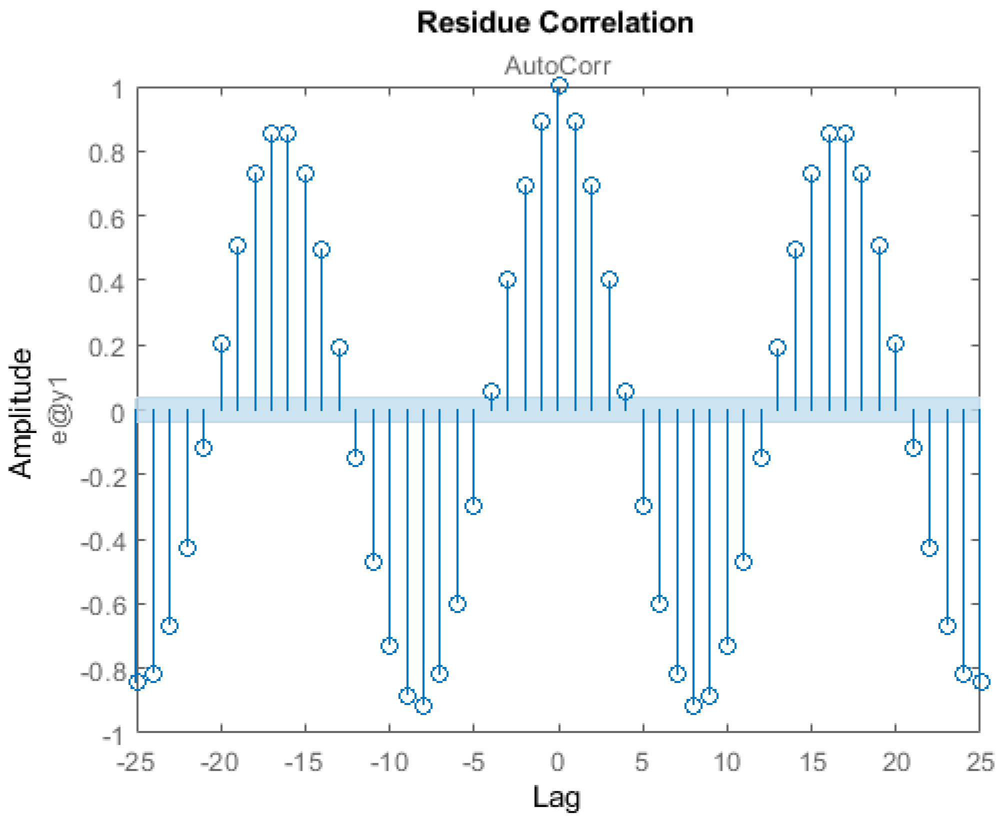
Fault detection is tagging of unwanted or unexpected changes in observations of the system. A fault causes changes in the system dynamics owing either to gradual wear and tear or sudden changes caused by sensor failure or broken parts. When a fault appears, the model obtained under normal working conditions is unable to predict the observed responses. This causes the difference between the measured and predicted response (the residuals) to increase. Such deviations are usually flagged by a large squared-sum-of-residuals or by presence of correlations.

Put the Simulink model in the damaged-system variant and simulate. We use a single bump as input since the residual test needs white input with possibly a transient owing to initial conditions.

```
set_param([sysA,'/Mechanical System'],'OverrideUsingVariant','DamagedSystem');
set_param([sysA,'/Pulse'],'Period','5120') % to force only one bump
sim(sysA)
y = logsout.getElement('y').Values;
```



```
resid(model, y.Data)
set_param([sysA,'/Pulse'],'Period','512') % restore original
```



The residuals are now larger and show correlations at non-zero lags. This is the basic idea behind detection of faults - creating a residual metric and observing how it changes with each new set of measurements. What is used here is a simple residual based on 1-step prediction error. In practice, more advanced residuals are generated that are tailor-made to the application needs.

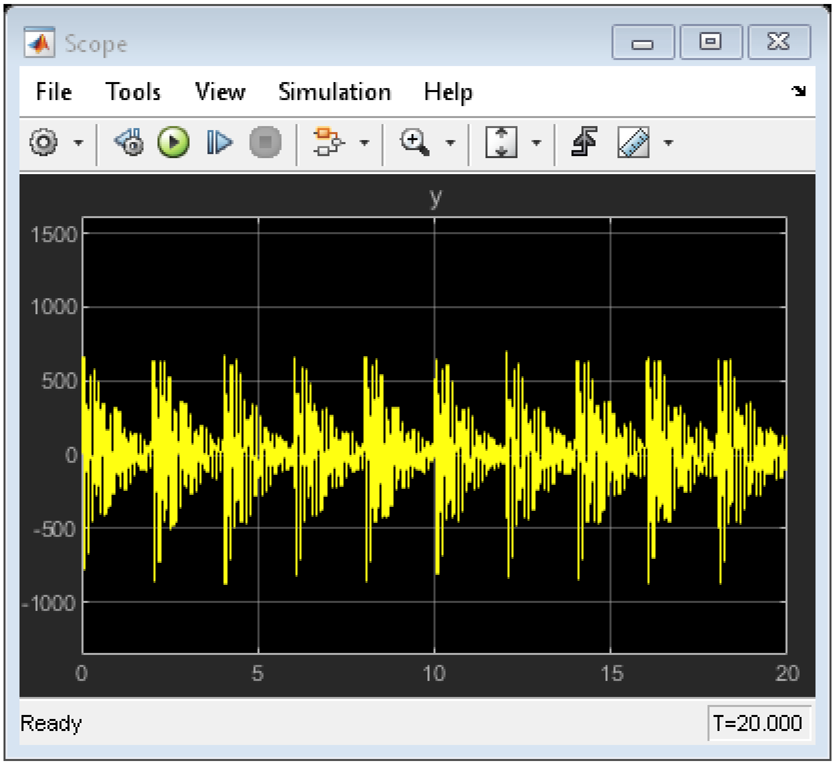
Fault Detection Using Models of Normal and Deteriorated State

A more detailed approach to fault detection is to also identify a model of the faulty (damaged) state of the system. We can then analyze which model is more likely to explain the live measurements from the system. This arrangement can be generalized to models for various types of faults and thus used for not just detecting the fault but also identifying which one ("isolation"). In this example, we take the following approach:

1. We collect data with system operating in the normal (healthy) and a known wear-and-tear induced end-of-life state.
2. We identify a dynamic model representing the behavior in each state.
3. We use a data clustering approach to draw a clear distinction between these states.
4. For fault detection, we collect data from the running machine and identify a model of its behavior. We then predict which state (normal or damaged) is most likely to explain the observed behavior.

We have already simulated the system in its normal operation mode. We now simulate the model `idMechanicalSystem` in the "end of life" mode. This is the scenario where the system has already deteriorated to its final state of permissible operation.

```
set_param([sysA,'/Mechanical System'],'OverrideUsingVariant','DamagedSystem');
sim(sysA)
y = logsout.getElement('y').Values;
zfault = cell(nr,1);
for ct = 1:nr
   z = iddata(y.Data((ct-1)*N+(1:500)),[],Ts);
   zfault{ct} = z;
end
```

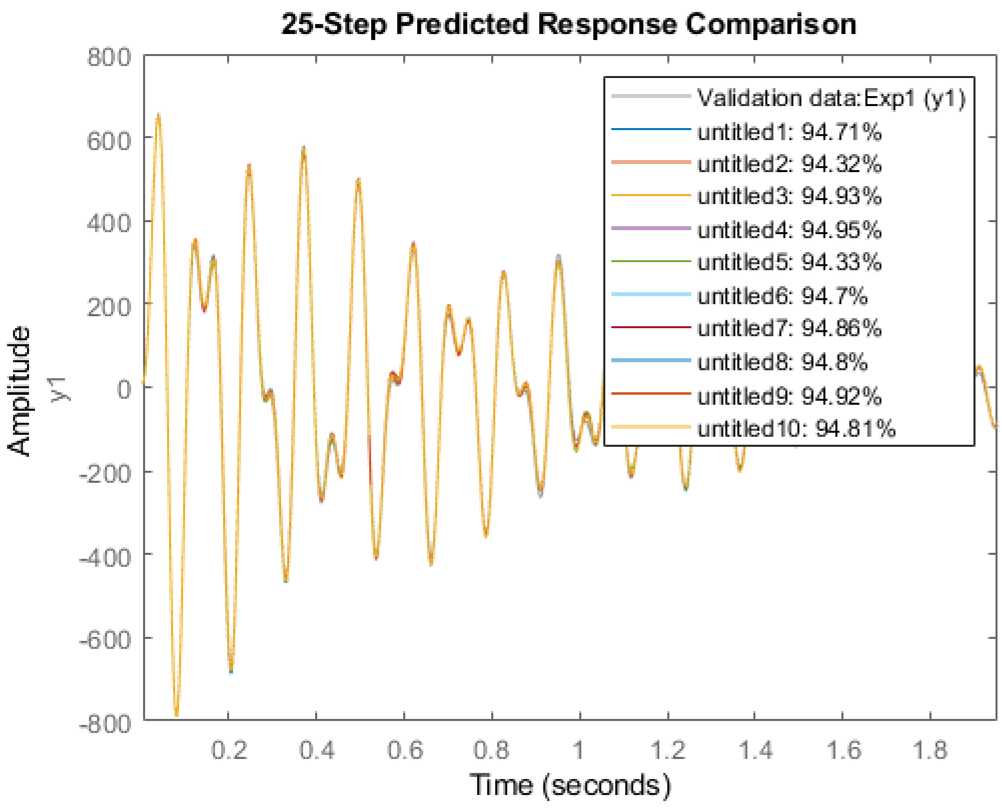


We now create a set of models, one for each data segment. As before we build 7th order time series models in state-space form. Turn off covariance computation for speed.

```
mNormal =  cell(nr,1);
mFault = cell(nr, 1);
nx = order(model);
opt = ssestOptions('EstimateCovariance',0);
for ct = 1:nr
   mNormal{ct} = ssest(znormal{ct}, nx, 'form', 'canonical', 'Ts', Ts, opt);
   mFault{ct} = ssest(zfault{ct}, nx, 'form', 'canonical', 'Ts', Ts, opt);
end
```

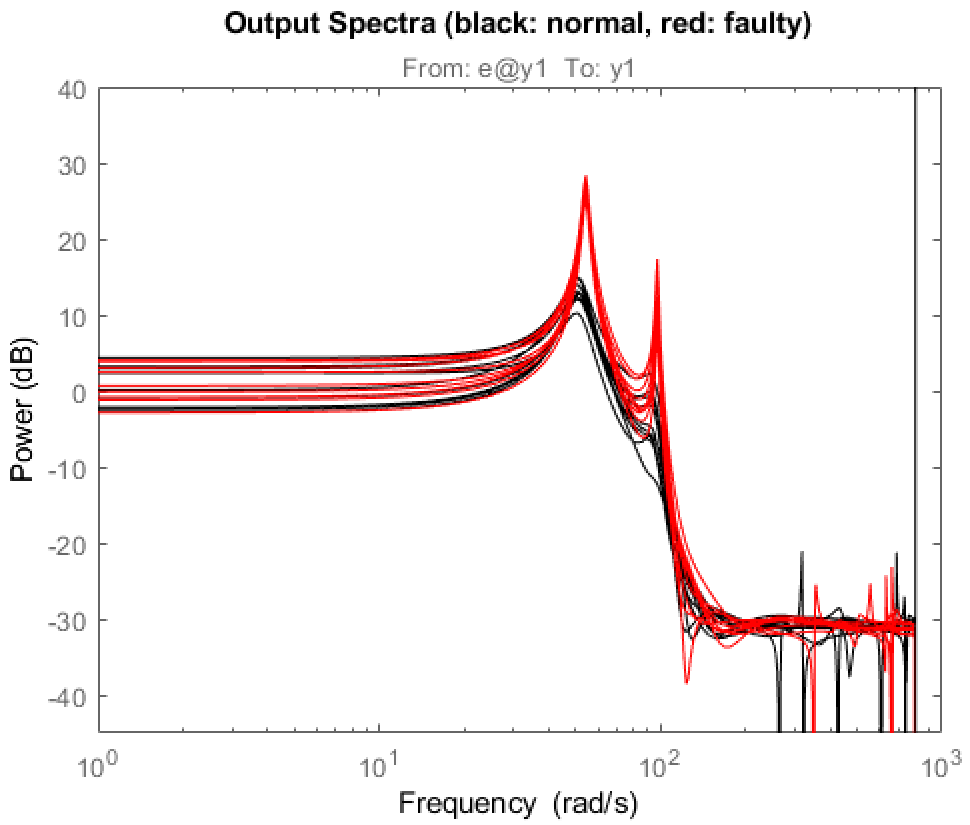
Verify that the models `mFault` are a good representation of the faulty mode of operation:

```
compare(merge(zfault{:}), mFault{:}, 25)
```



Normal and faulty estimated spectra are plotted below.

```
Color1 = 'k'; Color2 = 'r';
ModelSet1 = cat(2,mNormal,repmat({Color1},[nr, 1]))';
ModelSet2 = cat(2,mFault,repmat({Color2},[nr, 1]))';

spectrum(ModelSet1{:},ModelSet2{:})
axis([1  1000  -45  40])
title('Output Spectra (black: normal, red: faulty)')
```



The spectrum plot shows the difference: the damaged mode has its primary resonances amplified but the spectra are otherwise overlapping. Next, we create a way to quantitatively distinguish between the normal and the faulty state. We can use data clustering approaches such as:

- Fuzzy C-Means Clustering. See `fcm()` in Fuzzy Logic Toolbox.
- Support Vector Machine Classifier. See `fitcsvm ()` in Statistics and Machine Learning Toolbox.
- Self-organizing Maps. See `selforgmap()` in Deep Learning Toolbox.

In this example, we use the Support Vector Machine classification technique. The clustering of information from the two types of models (`mNormal` and `mFault`) can be based on different kinds of information that these models can provide such as the locations of their poles and zeroes, their locations of peak resonances or their list of parameters. Here, we classify the modes by the pole locations corresponding to the two resonances. For clustering, we tag the poles of the healthy state models with 'good' and the poles of the faulty state models with 'faulty'.

```
ModelTags = cell(nr*2,1);  % nr is number of data segments
ModelTags(1:nr) = {'good'};
ModelTags(nr+1:end) = {'faulty'};
ParData = zeros(nr*2,4);
plist = @(p)[real(p(1)),imag(p(1)),real(p(3)),imag(p(3))]; % poles of dominant resonanc
for ct = 1:nr
   ParData(ct,:) =  plist(esort(pole(mNormal{ct})));
   ParData(nr+ct,:) = plist(esort(pole(mFault{ct})));
end
cl = fitcsvm(ParData,ModelTags,'KernelFunction','rbf', ...
   'BoxConstraint',Inf,'ClassNames',{'good', 'faulty'});
cl.ConvergenceInfo.Converged
```

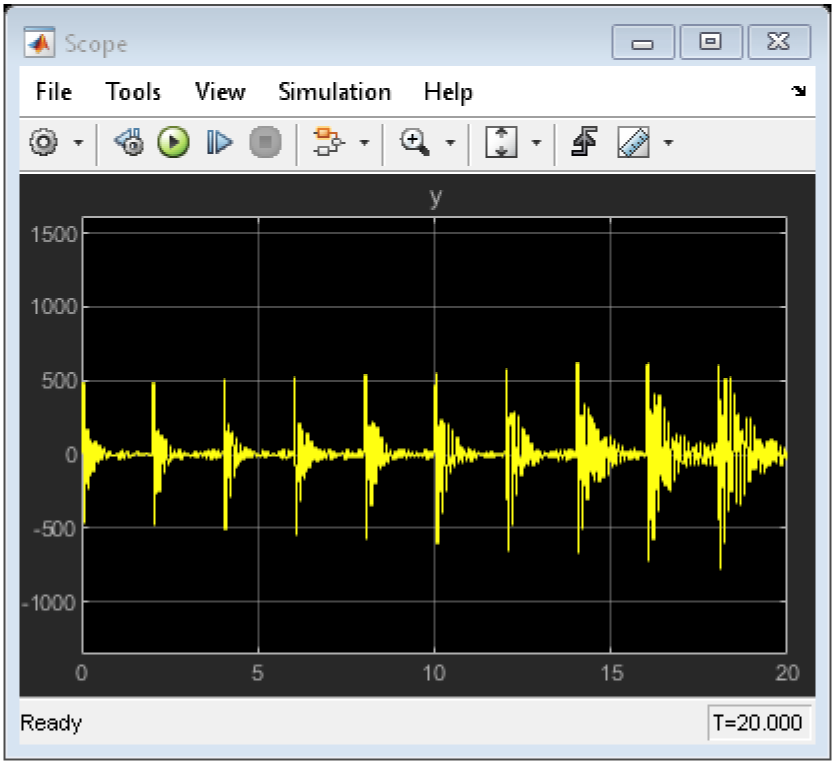
```
ans =

  logical

   1
```

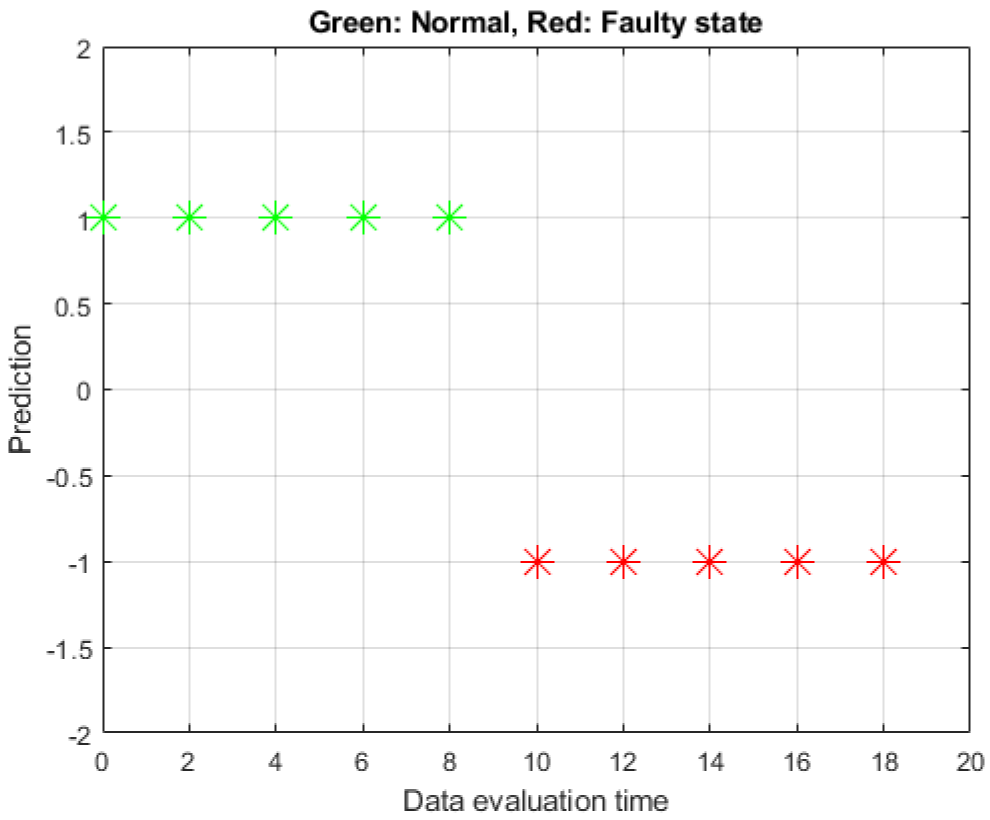
`cl` is an SVM classifier that separates the training data `ParData` into good and faulty regions. Using the `predict` method of this classifier one can assign an input nx-by-1 vector to one of the two regions.

Now we can test the classifier for its prediction (normal vs damaged) collect data batches from a system whose parameters are changing in a manner that it goes from being healthy (mode = 'Normal') to being fully damaged (mode = 'DamagedSystem') in a continuous manner. To simulate this scenario, we put the model in 'DeterioratingSystem' mode.

```
set_param([sysA,'/Mechanical System'],'OverrideUsingVariant','DeterioratingSystem');
sim(sysA)
ytv = logsout.getElement('y').Values; ytv = squeeze(ytv.Data);
PredictedMode = cell(nr,1);
for ct = 1:nr
   zSegment = iddata(ytv((ct-1)*512+(1:500)),[],Ts);
   mSegment = ssest(zSegment, nx, 'form', 'canonical', 'Ts', Ts);
   PredictedMode(ct) = predict(cl, plist(esort(pole(mSegment))));
```

```
end

I = strcmp(PredictedMode,'good');
Tags = ones(nr,1);
Tags(~I) = -1;
t = (0:5120)'*Ts;  % simulation time
Time = t(1:512:end-1);
plot(Time(I),Tags(I),'g*',Time(~I),Tags(~I),'r*','MarkerSize',12)
grid on
axis([0 20 -2 2])
title('Green: Normal, Red: Faulty state')
xlabel('Data evaluation time')
ylabel('Prediction')
```





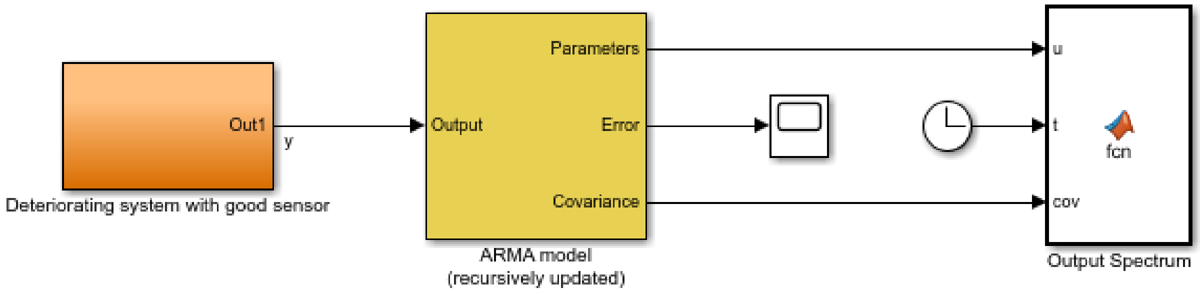
The plot shows that the classifier predicts the behavior up to about the mid-point to be normal and in a state of fault thereafter.

Fault Detection by Online Adaptation of Model Parameters

The preceding analysis used batches of data collected at different times during the operation of the system. An alternative, often more convenient, way of monitoring the health of the system is to create an adaptive model of its behavior. The new measurements are processed continuously and are used to update the parameters of a model in a recursive fashion. The effect of wear and tear or a fault is indicated by a change in the model parameter values.

Consider the wear-and-tear scenario again. As the system ages, there is a greater "rattling" which manifests itself as excitation of several resonant modes as well as a rise in the system's peak response. This scenario is described in model `idDeterioratingSystemEstimation` which is same as the 'DeterioratingSystem' mode of `idMechanicalSystem` except that the impulsive bumps that were added for offline identification are not present. The response of the system is passed to a "Recursive Polynomial Model Estimator" block which has been configured to estimate the parameters of an ARMA model structure. The actual system starts in a healthy state but deteriorates to end-of-life conditions over a time span of 200 seconds.

```
initial_model = translatecov(@(x)idpoly(x),model);
sysB = 'pdmDeterioratingSystemEstimation';
open_system(sysB);
```

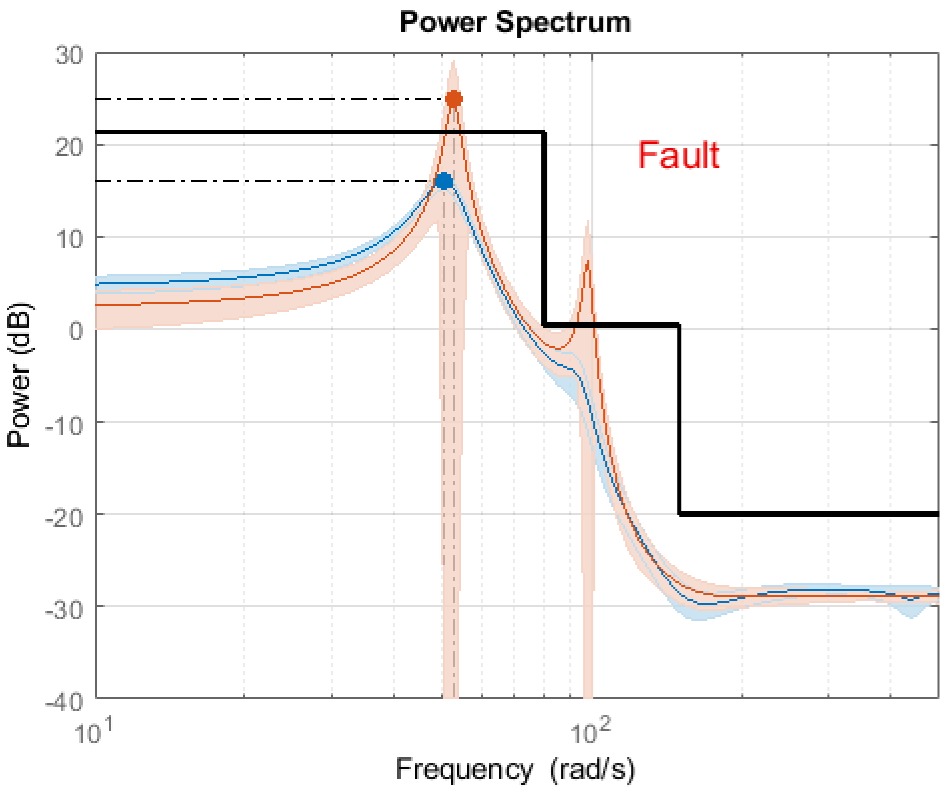


The "ARMA model" block has been initialized using the parameters and covariance data from the estimated model of normal behavior derived in the previous section after conversion to polynomial (ARMA) format. The `translatecov()` function is used so that the parameter covariance data is also converted. The block uses a "Forgetting factor" algorithm with the forgetting factor set to slightly less than 1 to update the parameters at each sampling instant. The choice of forgetting factor influences how rapidly the system updates. A small value means that the updates will have high variance while a large value will make it harder for the estimator to adapt to fast changes.

The model parameters estimate is used to update the output spectrum and its 3-sd confidence region. The system will have clearly changed when the spectrum's confidence region does not overlap that of the healthy system at frequencies of interest. A fault detection threshold is shown using a black line in the plot marking the maximum allowed gains at certain frequencies. As changes in the system accumulate, the spectrum drifts across this line. This serves as a visual indicator of a fault which can be used to call for repairs in real-life systems.

Run the simulation and watch the spectrum plot as it updates.

```
sim(sysB)
```



The running estimates of model parameters are also used to compute the system pole locations which are then fed into the SVM classifier to predict if the system is in the "good" or "fault" state. This decision is also displayed on the plot. When the normalized score of prediction is less than .3, the decision is considered tentative (close to the boundary of distinction). See the script `pdmARMASpectrumPlot.m` for details on how the running estimate of spectrum and classifier prediction is computed.

It is possible to implement the adaptive estimation and plotting procedure outside Simulink using the `recursiveARMA()` function. Both the "Recursive Polynomial Model

Estimator" block as well as the `recursiveARMA()` function support code generation for deployment purposes.

The classification scheme can be generalized to the case where there are several known modes of failure. For this we will need multi-group classifiers where a mode refers to a certain type of failure. These aspects are not explored in this example.

Conclusions

This example showed how system identification schemes combined with data clustering approaches can assist in detection and isolation of faults. Both sequential batch analysis as well as online adaptation schemes were discussed. A model of ARMA structure of the measured output signal was identified. A similar approach can be adopted in situations where one has access to both input and output signals, and would like to employ other types of model structures such as the State-space or Box-Jenkins polynomial models.

In this example, we found that:

1. Correlations in residuals based on a model of normal operation can indicate onset of failure.
2. Gradually worsening faults can be detected by employing a continuously adapting model of the system behavior. Preset thresholds on a model's characteristics such as bounds on its output spectrum can help visualize the onset and progression of failures.
3. When the source of a fault needs to be isolated, a viable approach is to create separate models of concerned failure modes beforehand. Then a clustering approach can be used to assign the predicted state of the system to one of these modes.

See Also

More About

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

Detect Abrupt System Changes Using Identification Techniques

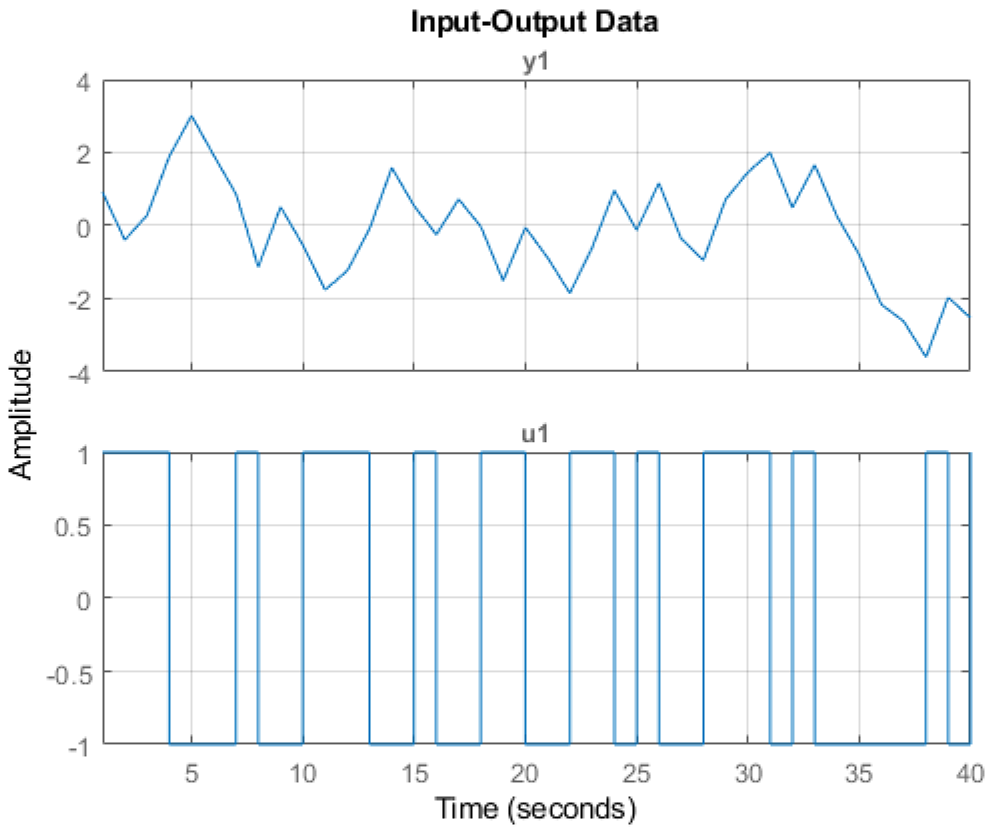
This example shows how to detect abrupt changes in the behavior of a system using online estimation and automatic data segmentation techniques. This example uses functionality from System Identification Toolbox™, and does not require Predictive Maintenance Toolbox™.

Problem Description

Consider a linear system whose transport delay changes from two to one second. Transport delay is the time taken for the input to affect the measured output. In this example, you detect the change in transport delay using online estimation and data segmentation techniques. Input-output data measured from the system is available in the data file `pdmAbruptChangesData.mat`.

Load and plot the data.

```
load pdmAbruptChangesData.mat
z = iddata(z(:,1),z(:,2));
plot(z)
grid on
```



The transport delay change takes place around 20 seconds, but is not easy to see in the plot.

Model the system using an ARX structure with one `A` polynomial coefficient, two `B` polynomial coefficients, and one delay.

$$y(t) + ay(t-1) = b1u(t-1) + b2u(t-2)$$

Here, `A = [1 a]` and `B = [0 b1 b2]`.

The leading coefficient of the `B` polynomial is zero because the model has no feedthrough. As the system dynamics change, the values of the three coefficients `a`, `b1`, and `b2` change. When `b1` is close to zero, the effective transport delay will be 2 samples because the `B` polynomial has 2 leading zeros. When b1 is larger, the effective transport delay will be 1 sample.

Thus, to detect changes in transport delay you can monitor changes in the B polynomial coefficients.

Use Online Estimation for Change Detection

Online estimation algorithms update model parameters and state estimates in a recursive manner, as new data becomes available. You can perform online estimation using Simulink blocks from the System Identification Toolbox library or at the command line using recursive identification routines such as recursiveARX. Online estimation can be used to model time varying dynamics such as aging machinery and changing weather patterns, or to detect faults in electromechanical systems.

As the estimator updates the model parameters, a change in system dynamics (delay) will be indicated by a larger than usual change in the values of the parameters b1 and b2. Changes in the B polynomial coefficients will be tracked by computing:

$$L(t) = abs(B(t) - B(t-1))$$

Use the recursiveARX object for online parameter estimation of the ARX model.

```
na = 1;
nb = 2;
nk = 1;
Estimator = recursiveARX([na nb nk]);
```

Specify the recursive estimation algorithm as NormalizedGradient and the adaptation gain as 0.9.

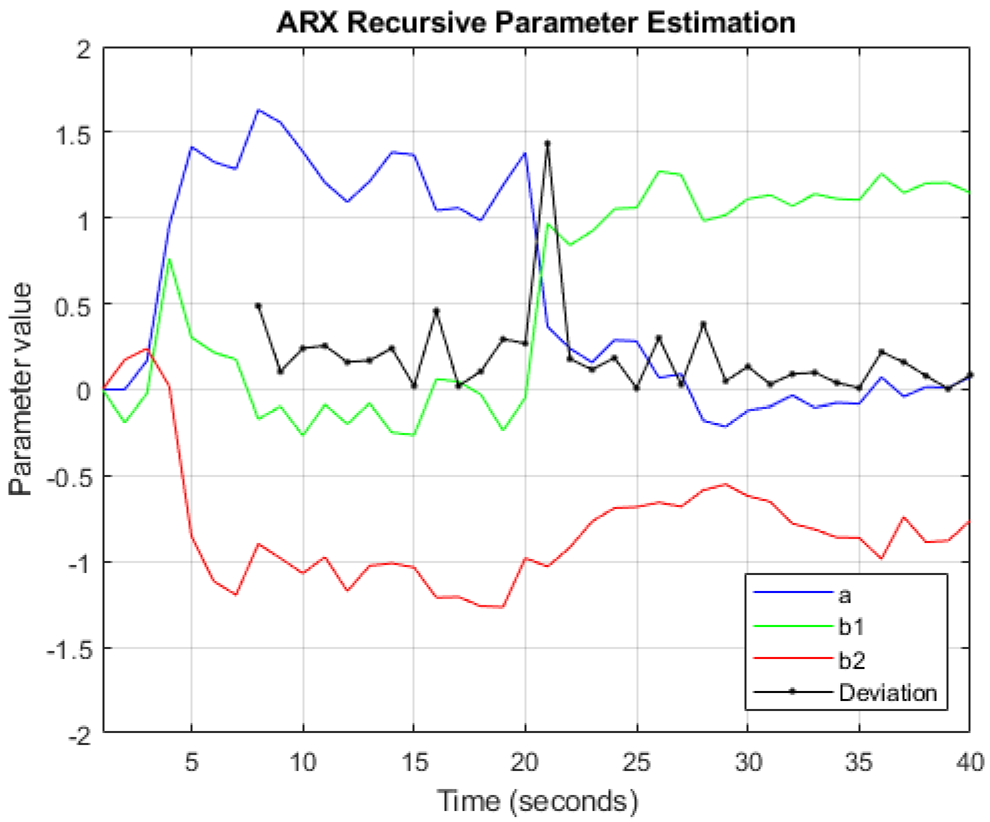
```
Estimator.EstimationMethod = 'NormalizedGradient';
Estimator.AdaptationGain = .9;
```

Extract the raw data from the iddata object, z.

```
Output = z.OutputData;
Input = z.InputData;
t = z.SamplingInstants;
N = length(t);
```

Use animated lines to plot the estimated parameter values and L. Initialize these animated lines prior to estimation. To simulate streaming data, feed the data to the estimator one sample at a time. Initialize the model parameters before estimation, and then perform online estimation.

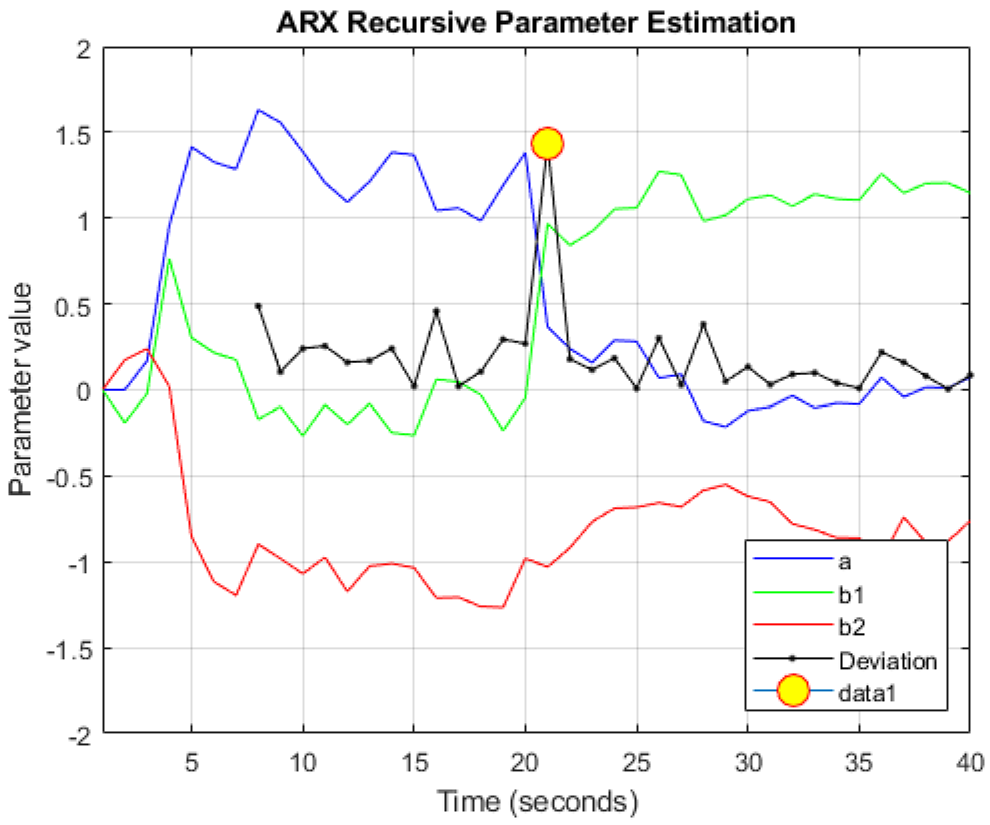
```
%% Initialize plot
Colors = {'r','g','b'};
ax = gca;
cla(ax)
for k = 3:-1:1
   h(k) = animatedline('Color',Colors{k}); % lines for a, b1 and b2 parameters
end
h(4) =  animatedline('Marker','.','Color',[0 0 0]); % line for L
legend({'a','b1','b2','Deviation'},'location','southeast')
title('ARX Recursive Parameter Estimation')
xlabel('Time (seconds)')
ylabel('Parameter value')
ax.XLim = [t(1),t(end)];
ax.YLim = [-2, 2];
grid on
box on

%% Now perform recursive estimation and show results
n0 = 6;
L = NaN(N,nk);
B_old = NaN(1,3);
for ct = 1:N
   [A,B] = step(Estimator,Output(ct),Input(ct));
   if ct>n0
      L(ct) = norm(B-B_old);
      B_old = B;
   end
   addpoints(h(1),t(ct),A(2))
   addpoints(h(2),t(ct),B(2))
   addpoints(h(3),t(ct),B(3))
   addpoints(h(4),t(ct),L(ct))
   pause(0.1)
end
```



The first `n0` = 6 samples of the data are not used for computing the change-detector, `L`. During this interval the parameter changes are large owing to the unknown initial conditions.

Find the location of all peaks in `L` by using the `findpeaks` command from Signal Processing Toolbox.

```
[v,Loc] = findpeaks(L);
[~,I] = max(v);
line(t(Loc(I)),L(Loc(I)),'parent',ax,'Marker','o','MarkerEdgeColor','r',...
   'MarkerFaceColor','y','MarkerSize',12)
```



```
fprintf('Change in system delay detected at %g:th sample.\n',t(Loc(I)-1));
```

```
Change in system delay detected at 20:th sample.
```

The location of the largest peak corresponds to the largest change in the B polynomial coefficients, and is thus the location of a change in transport delay.

While online estimation techniques provide more options for choosing estimation methods and model structure, the data segmentation method can help automate detection of abrupt and isolated changes.

Use Data Segmentation for Change Detection

A data segmentation algorithm automatically segments the data into regions of different dynamic behavior. This is useful for capturing abrupt changes arising from a failure or change of operating conditions. The `segment` command facilitates this operation for single-output data. `segment` is an alternative to online estimation techniques when you do not need to capture the time-varying behavior during system operation.

Applications of data segmentation include segmentation of speech signals (each segment corresponds to a phoneme), failure detection (the segments correspond to operation with and without failures), and estimation of different working modes of a system.

Inputs to the `segment` command include the measured data, the model orders, and a guess for the variance, `r2`, of the noise that affects the system. If the variance is entirely unknown, it can be estimated automatically. Perform data segmentation using an ARX model of the same orders as used for online estimation. Set the variance to 0.1.

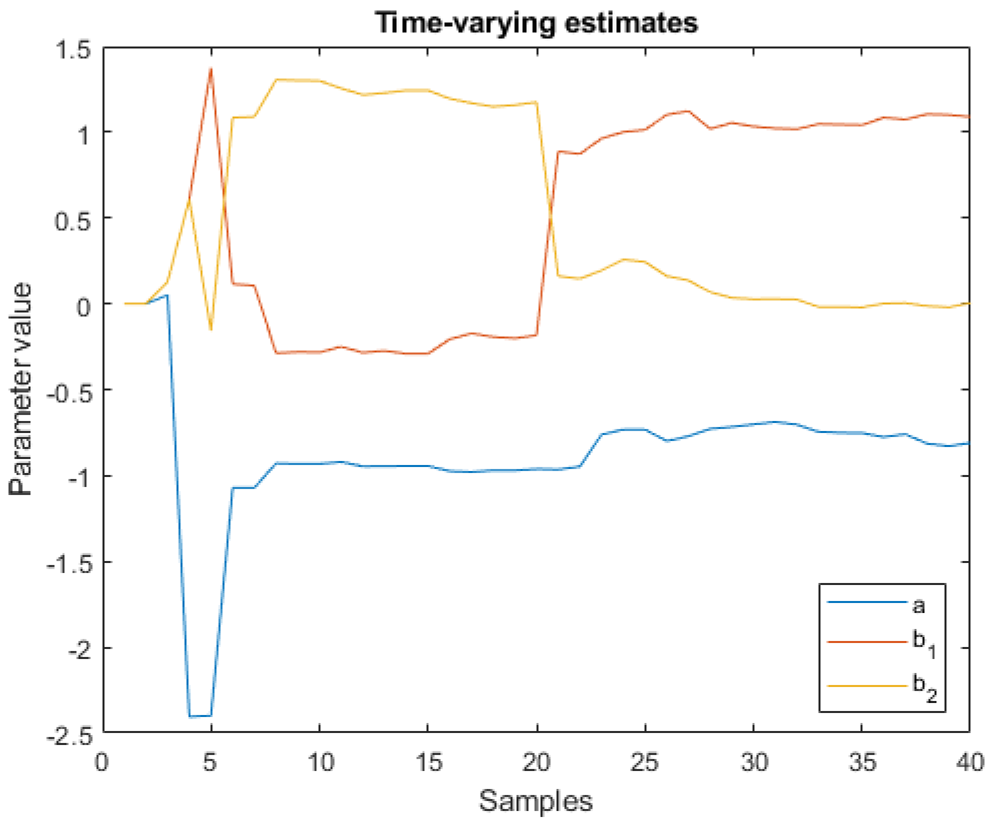
```
[seg,V,tvmod] = segment(z,[na nb nk],0.1);
```

The method for segmentation is based on AFMM (adaptive forgetting through multiple models). For details about the method, see Andersson, Int. J. Control Nov 1985.

A multi-model approach is used to track the time-varying system. The resulting tracking model is an average of the multiple models and is returned as the third output argument of `segment`, `tvmod`.

Plot the parameters of the tracking model.

```
plot(tvmod)
legend({'a','b_1','b_2'},'Location','best')
xlabel('Samples'), ylabel('Parameter value')
title('Time-varying estimates')
```



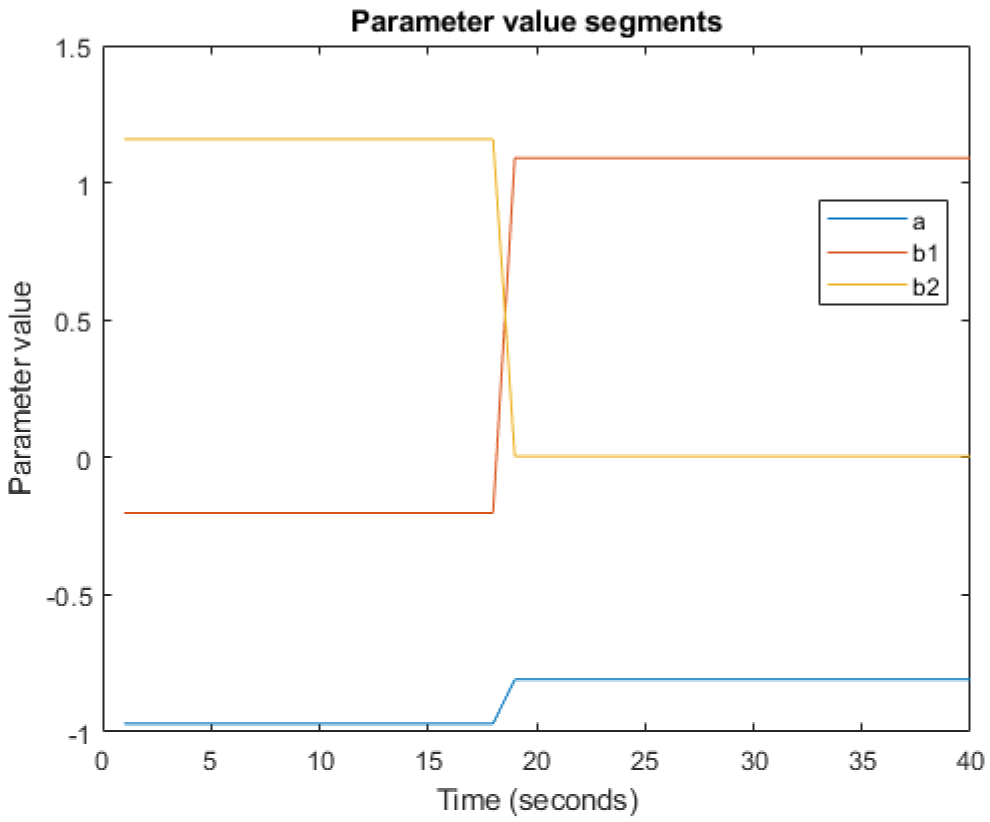
Note the similarity between these parameter trajectories and those estimated using `recursiveARX`.

`segment` determines the time points when changes have occurred using `tvmod` and `q`, the probability that a model exhibits abrupt changes. These time points are used to construct the segmented model by employing a smoothing procedure over the tracking model.

The parameter values of the segmented model are returned in `seg`, the first output argument of `segment`. The values in each successive row are the parameter values of the underlying segmented model at the corresponding time instants. These values remain constant over successive rows and change only when the system dynamics are determined to have changed. Thus, values in `seg` are piecewise constant.

Plot the estimated values for parameters a, b1, and b2.

```
plot(seg)
title('Parameter value segments')
legend({'a','b1','b2'},'Location','best')
xlabel('Time (seconds)')
ylabel('Parameter value')
```



A change is seen in the parameter values around sample number 19. The value of b1 changes from a small (close to zero) to large (close to 1) value. The value of b2 shows the opposite pattern. This change in the values of the B parameters indicates a change in the transport delay.

The second output argument of `segment`, `V`, is the loss function for the segmented model (i.e. the estimated prediction error variance for the segmented model). You can use `V` to asses the quality of the segmented model.

Note that the two most important inputs for the segmentation algorithm are `r2` and `q`, the fourth input argument to `segment`. In this example, `q` was not specified because the default value, 0.01, was adequate. A smaller value of `r2` and a larger value of `q` will result in more segmentation points. To find appropriate values, you can vary `r2` and `q` and use the ones that work the best. Typically, the segmentation algorithm is more sensitive to `r2` than `q`.

Conclusions

The use of online estimation and data segmentation techniques for detecting abrupt changes in system dynamics was evaluated. Online estimation techniques offer more flexibility and more control over the estimation process. However, for changes that are infrequent or abrupt, `segment` facilitates an automatic detection technique based on smoothing of time-varying parameter estimates.

See Also

More About

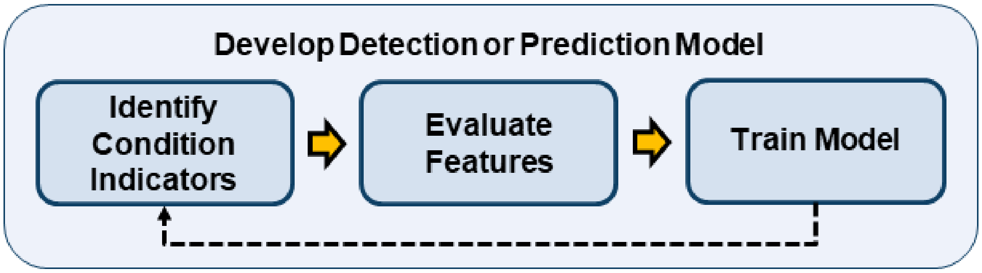
- “Model-Based Condition Indicators” on page 3-8

Predict Remaining Useful Life

Feature Selection for Remaining Useful Life Prediction

For reliable remaining useful life (RUL) estimations, you want a condition indicator whose change over time is observable and connected with the system degradation process in a reliable, measurable way. The remaining useful life 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.

After you identify condition indicators (see “Condition Indicators for Monitoring, Fault Detection, and Prediction” on page 3-2), selecting useful condition indicators out of all available features is the next step in building a reliable RUL prediction model.


Develop Detection or Prediction Model
Identify Condition Indicators
Evaluate Features
Train Model

Predictive Maintenance Toolbox offers three feature selection metrics for accurate RUL prediction: monotonicity, trendability, and prognosability. These metrics rank the identified condition indicators on a scale ranging from 0 through 1. A higher ranked feature tracks the degradation process more reliably and hence, is more desirable to train the RUL prediction model.

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

Using the selected features to train an appropriate RUL estimation model is the next step in the algorithm-design process. For information, see "Models for Predicting Remaining Useful Life" on page 5-4.

See Also

`monotonicity` | `prognosability` | `trendability`

More About

- "Wind Turbine High-Speed Bearing Prognosis" on page 5-44
- "Condition Indicators for Monitoring, Fault Detection, and Prediction" on page 3-2
- "Models for Predicting Remaining Useful Life" on page 5-4

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 lifetime 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 lifetime 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.

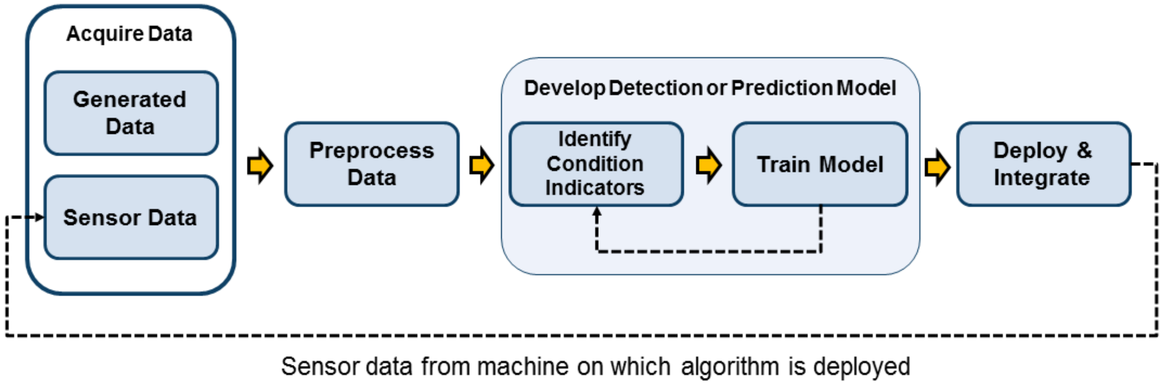
Typically, you estimate the RUL of a system by developing a model that can perform the estimation based on the time evolution or statistical properties of condition indicator values, such as:

- 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. For more information about such models, see “RUL Estimation Using Identified Models or State Estimators” on page 5-6.
- Specialized Predictive Maintenance Toolbox models designed for computing RUL from different types of measured system data. For more information about these models, see “RUL Estimation Using RUL Estimator Models” on page 5-8.

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. For more information, see “Feature Selection for Remaining Useful Life Prediction” on page 5-2.


Acquire Data
Generated Data
Sensor Data
Preprocess Data
Develop Detection or Prediction Model
Identify Condition Indicators
Train Model
Deploy & Integrate
Sensor data from machine on which algorithm is deployed

See Also

More About

- “RUL Estimation Using Identified Models or State Estimators” on page 5-6
- “RUL Estimation Using RUL Estimator Models” on page 5-8
- “Feature Selection for Remaining Useful Life Prediction” on page 5-2

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” on page 5-66 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.

See Also

More About

- “Nonlinear State Estimation of a Degrading Battery System” on page 5-87
- “Condition Monitoring and Prognostics Using Vibration Signals” on page 5-66
- “Models for Predicting Remaining Useful Life” on page 5-4
- “Feature Selection for Remaining Useful Life Prediction” on page 5-2

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 (lifetime)

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 lifetime 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 lifetime.

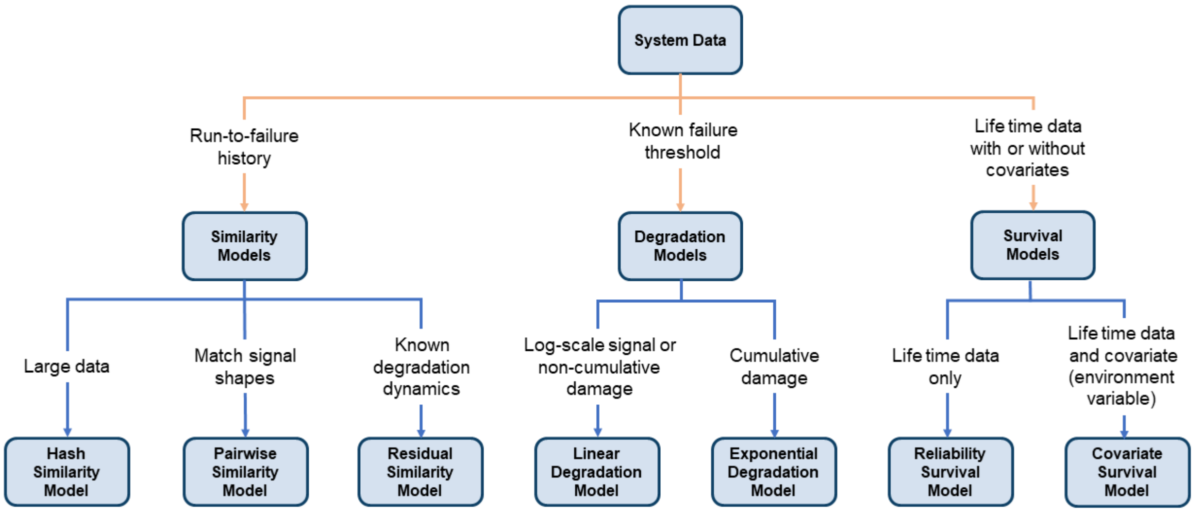
The general workflow for using RUL estimation models is:

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` command.

For a basic example illustrating these steps, see “Update RUL Prediction as Data Arrives” on page 5-13.

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.


System Data
Run-to-failure history
Known failure threshold
Life time data with or without covariates
Similarity Models
Degradation Models
Survival Models
Large data
Match signal shapes
Known degradation dynamics
Log-scale signal or non-cumulative damage
Cumulative damage
Life time data only
Life time data and covariate (environment variable)
Hash Similarity Model
Pairwise Similarity Model
Residual Similarity Model
Linear Degradation Model
Exponential Degradation Model
Reliability Survival Model
Covariate Survival Model

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 lifetime 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” on page 5-18.

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” on page 5-44.

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

- “Models for Predicting Remaining Useful Life” on page 5-4
- “Feature Selection for Remaining Useful Life Prediction” on page 5-2
- “Update RUL Prediction as Data Arrives” on page 5-13
- “Similarity-Based Remaining Useful Life Estimation” on page 5-18
- “Wind Turbine High-Speed Bearing Prognosis” on page 5-44

Update RUL Prediction as Data Arrives

This example shows how to update an RUL prediction as new data arrives from a machine under test. In the example, you use an ensemble of training data to train an RUL model. You then loop through a sequence of test data from a single machine, updating the RUL prediction for each new data point. The example shows the evolution of the RUL prediction as new data becomes available.

This example uses `exponentialDegradationModel`. For degradation RUL models, when a new data point becomes available, you must first update the degradation model before predicting a new RUL value. For other RUL model types, skip this update step.

Data for Training and Prediction

Load the data for this example, which consists of two variables, `TestData` and `TrainingData`.

```
load UpdateRULExampleData
```

`TestData` is a table containing the value of some condition indicator, `Condition`, recorded every hour, as the first few entries show.

```
head(TestData,5)
```

```
ans=5×2 table
    Time    Condition
    ____    _________

     1        1.0552 
     2        1.2013 
     3       0.79781 
     4          1.09 
     5        1.0324 
```

`TrainingData` is a cell array of tables having the same variables as `TestData`. Each cell in `TrainingData` represents the evolution to failure of the condition indicator `Condition` over the lifetime of one machine in the ensemble.

Train Prediction Model

Use `TrainingData` to train an `exponentialDegradationModel` model for RUL prediction. The `fit` command estimates a prior for the model's parameters based on the historical records in TrainingData. The Prior property of the trained model contains the model parameters Theta, Beta, and Rho. (For details of these model parameters, see exponentialDegradationModel.)

```
mdl = exponentialDegradationModel('LifeTimeUnit',"hours");
fit(mdl,TrainingData,"Time","Condition")
mdl.Prior
```

```
ans = struct with fields:
            Theta: 0.6389
    ThetaVariance: 0.0661
             Beta: 0.0583
     BetaVariance: 1.8383e-04
              Rho: -0.3129
```

Degradation models are most reliable for degradation tracking after an initial slope emerges in the condition-indicator measurements. Set the slope detection level at 0.1 to tell the model not to make RUL predictions until that slope is reached. (When you know in advance that the measurements are for a component whose degradation has already started, you can disable slope detection by setting mdl.SlopeDetectionLevel = [].)

```
mdl.SlopeDetectionLevel = 0.1;
```

Predict RUL at Each Time Step

Define a threshold condition indicator value that indicates the end of life of a machine. The RUL is the predicted time left before the condition indicator for the test machine reaches this threshold value.

```
threshold = 400;
```

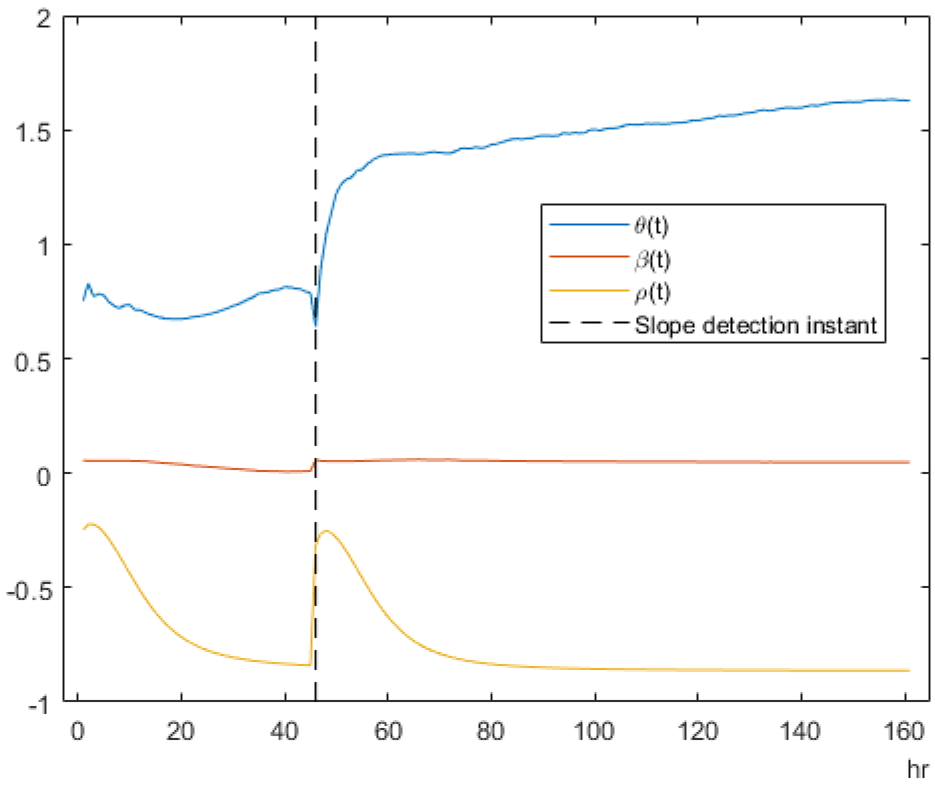
For RUL prediction, assume that TestData begins at time t = 1 hour, and a new data sample becomes available every hour. In general, you can predict a new RUL value with each new data point. For the degradation model of this example, loop through TestData and update the model with each new data point using the update command. Then, check whether the model detects a sufficient change in slope for reliable RUL prediction. If it does, predict a new RUL value using the predictRUL command. To observe the evolution of the estimation, store the estimated RUL values and the associated confidence intervals in the vectors EstRUL and CI, respectively. Similarly, store the model parameters in the array ModelParameters.

```
N = height(TestData);
EstRUL = hours(zeros(N,1));
```

```
CI = hours(zeros(N,2));
ModelParameters = zeros(N,3);
for t = 1:N
   CurrentDataSample = TestData(t,:);
   update(mdl,CurrentDataSample)
   ModelParameters(t,:) = [mdl.Theta mdl.Beta mdl.Rho];
   % Compute RUL only if the data indicates a change in slope.
   if ~isempty(mdl.SlopeDetectionInstant)
      [EstRUL(t),CI(t,:)] = predictRUL(mdl,CurrentDataSample,threshold);
   end
end
```

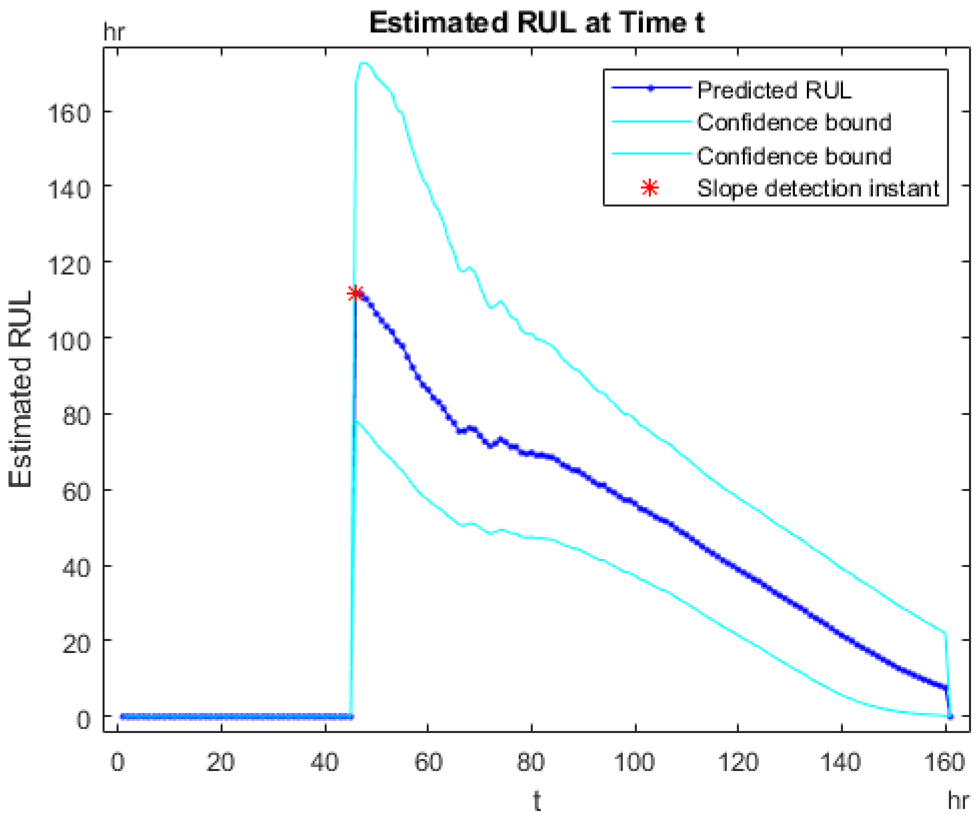
Plot the trajectories of the estimated model-parameter values. The values change rapidly after a slope is detected in the degradation data. They tend to converge as more data points become available.

```
Time = hours(1:N)';
tStart = mdl.SlopeDetectionInstant; % slope detection time instant
plot(Time,ModelParameters);
hold on
plot([tStart, tStart],[-1,2],'k--')
legend({'\theta(t)','\beta(t)','\rho(t)','Slope detection instant'},'Location','best')
hold off
```



Plot the predicted RUL to observe its evolution as more degradation data becomes available. There is no new estimated RUL value until a slope is detected in the degradation data. After that, the predicted RUL decreases over time, as expected. `predictRUL` computes a statistical distribution of RUL values. The confidence bounds on the predicted RUL become narrower over time.

```
plot(Time,EstRUL,'b.-',Time,CI,'c',tStart,EstRUL(hours(tStart)),'r*')
title('Estimated RUL at Time t')
xlabel('t')
ylabel('Estimated RUL')
legend({'Predicted RUL','Confidence bound','Confidence bound','Slope detection instant
```



See Also

`exponentialDegradationModel` | `predictRUL` | `update`

More About

- “RUL Estimation Using RUL Estimator Models” on page 5-8
- “Wind Turbine High-Speed Bearing Prognosis” on page 5-44

Similarity-Based Remaining Useful Life Estimation

This example shows how to build a complete Remaining Useful Life (RUL) estimation workflow including the steps for preprocessing, selecting trendable features, constructing a health indicator by sensor fusion, training similarity RUL estimators, and validating prognostics performance. The example uses the training data from the PHM2008 challenge dataset from https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/ [1].

Data Preparation

Since the dataset is small it is feasible to load the whole degradation data into memory. Download and unzip the the data set from https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/ to the current directory. Use the `helperLoadData` helper function to load and convert the training text file to a cell array of timetables. The training data contains 218 run-to-failure simulations. This group of measurements is called an "ensemble".

```
degradationData = helperLoadData('train.txt');
degradationData(1:5)
```

```
ans = 5×1 cell array
    {223×26 table}
    {164×26 table}
    {150×26 table}
    {159×26 table}
    {357×26 table}
```

Each ensemble member is a table with 26 columns. The columns contain data for the machine ID, time stamp, 3 operating conditions and 21 sensor measurements.

```
head(degradationData{1})
```

```
ans=8×26 table
    id    time    op_setting_1    op_setting_2    op_setting_3    sensor_1    sensor_2
    __    ____    ____________    ____________    ____________    ________    ________

    1      1         10.005          0.2501            20          489.05      604.13
    1      2         0.0015          0.0003           100          518.67      642.13
    1      3         34.999          0.8401            60          449.44      555.42
    1      4         20.003          0.7005             0          491.19      607.03
    1      5         42.004          0.8405            40             445      549.52
    1      6          20.003          0.7017              0          491.19      607.37
    1      7              42            0.84             40             445      549.57
    1      8          0.0011               0            100          518.67      642.08
```

Split the degradation data into a training data set and a validation data set for later performance evaluation.

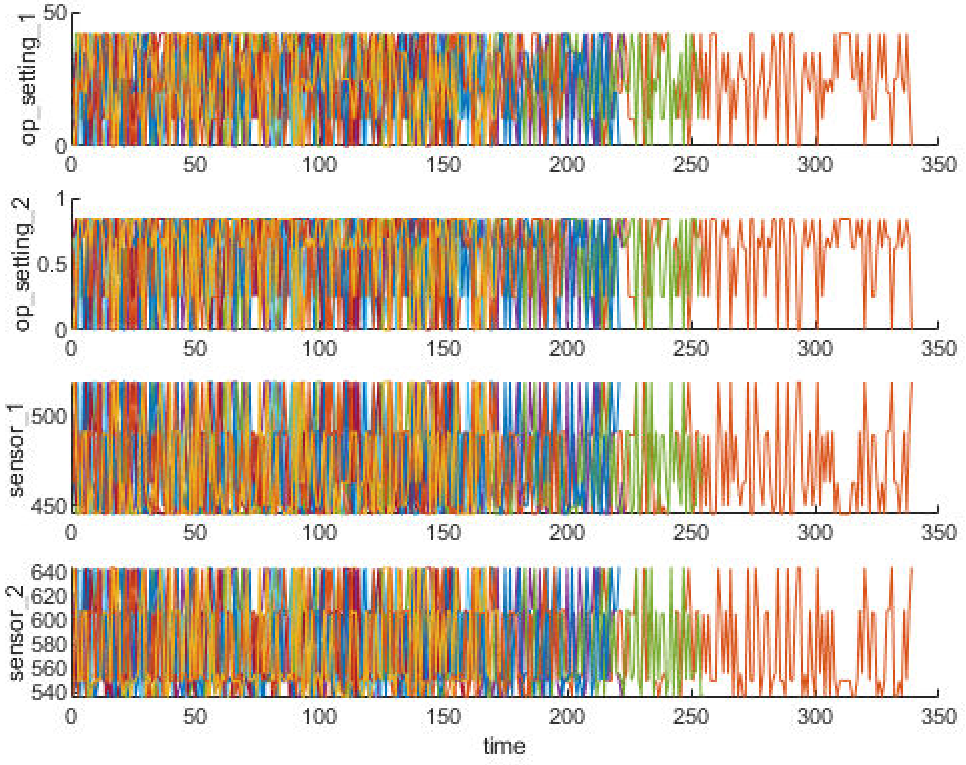
```
rng('default')  % To make sure the results are repeatable
numEnsemble = length(degradationData);
numFold = 5;
cv = cvpartition(numEnsemble, 'KFold', numFold);
trainData = degradationData(training(cv, 1));
validationData = degradationData(test(cv, 1));
```

Specify groups of variables of interest.

```
varNames = string(degradationData{1}.Properties.VariableNames);
timeVariable = varNames{2};
conditionVariables = varNames(3:5);
dataVariables = varNames(6:26);
```

Visualize a sample of the ensemble data.

```
nsample = 10;
figure
helperPlotEnsemble(trainData, timeVariable, ...
    [conditionVariables(1:2) dataVariables(1:2)], nsample)
```



Working Regime Clustering

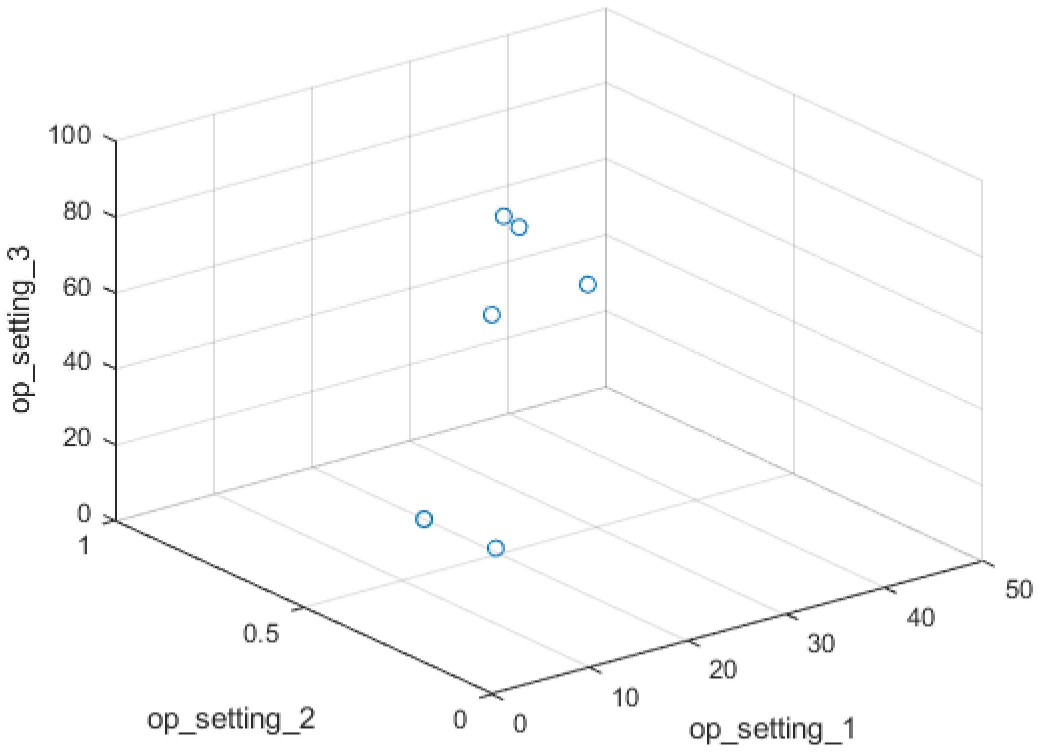
As shown in the previous section, there is no clear trend showing the degradation process in each run-to-failure measurement. In this and the next section, the operating conditions will be used to extract clearer degradation trends from sensor signals.

Notice that each ensemble member contains 3 operating conditions: "op_setting_1", "op_setting_2", and "op_setting_3". First, let's extract the table from each cell and concatenate them into a single table.

```
trainDataUnwrap = vertcat(trainData{:});
opConditionUnwrap = trainDataUnwrap(:, cellstr(conditionVariables));
```

Visualize all operating points on a 3D scatter plot. It clearly shows 6 regimes and the points in each regime are in very close proximity.

```
figure
helperPlotClusters(opConditionUnwrap)
```



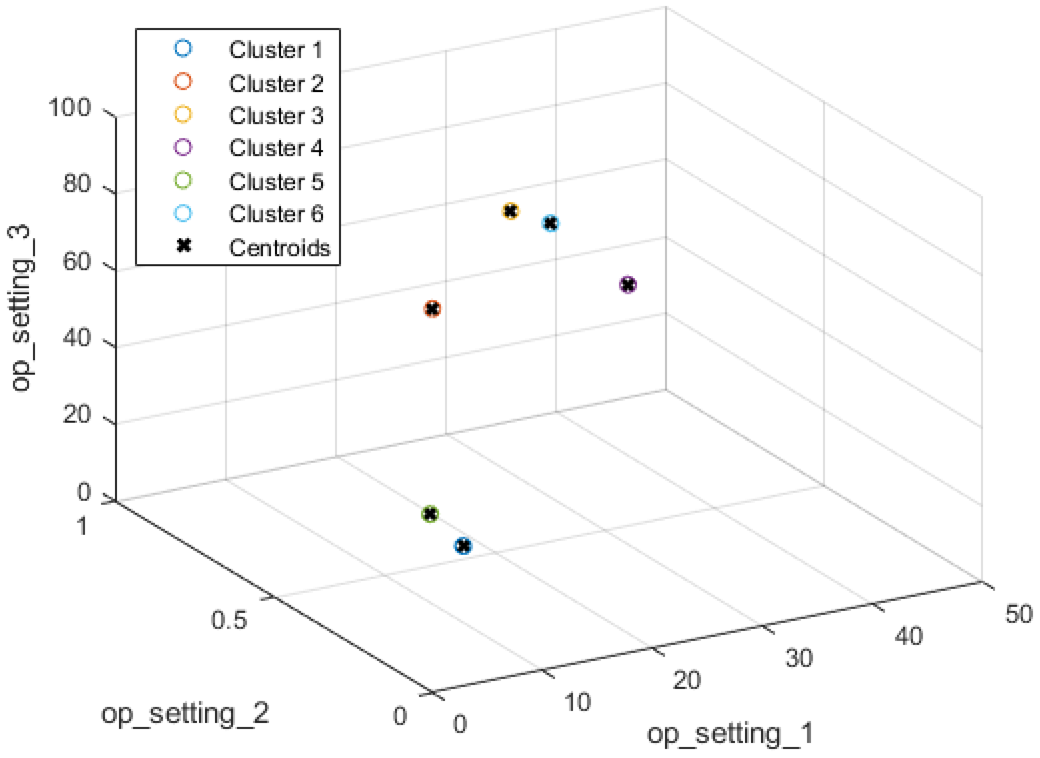
Let's use clustering techniques to locate the 6 clusters automatically. Here, the K-means algorithm is used. K-means is one of the most popular clustering algorithms, but it can result in local optima. It is a good practice to repeat the K-means clustering algorithm several times with different initial conditions and pick the results with the lowest cost. In this case, the algorithm runs 5 times and the results are identical.

```
opts = statset('Display', 'final');
[clusterIndex, centers] = kmeans(table2array(opConditionUnwrap), 6, ...
    'Distance', 'sqeuclidean', 'Replicates', 5, 'Options', opts);
```

```
Replicate 1, 1 iterations, total sum of distances = 0.279547.
Replicate 2, 1 iterations, total sum of distances = 0.279547.
Replicate 3, 1 iterations, total sum of distances = 0.279547.
Replicate 4, 1 iterations, total sum of distances = 0.279547.
Replicate 5, 1 iterations, total sum of distances = 0.279547.
Best total sum of distances = 0.279547
```

Visualize the clustering results and the identified cluster centroids.

```
figure
helperPlotClusters(opConditionUnwrap, clusterIndex, centers)
```



As the plot illustrates, the clustering algorithm successfully finds the 6 working regimes.

Working Regime Normalization

Let's perform a normalization on measurements grouped by different working regimes. First, compute the mean and standard deviation of each sensor measurement grouped by the working regimes identified in the last section.

```
centerstats = struct('Mean', table(), 'SD', table());
for v = dataVariables
    centerstats.Mean.(char(v)) = splitapply(@mean, trainDataUnwrap.(char(v)), clusterI
    centerstats.SD.(char(v))   = splitapply(@std,  trainDataUnwrap.(char(v)), clusterI
end
centerstats.Mean
```

```
ans=6×21 table
    sensor_1    sensor_2    sensor_3    sensor_4    sensor_5    sensor_6    sensor_7
    ________    ________    ________    ________    ________    ________    ________

     489.05      604.92      1502.1      1311.4      10.52       15.493      394.32
     518.67      642.71      1590.7      1409.4      14.62        21.61       553.3
     462.54      536.87      1262.8      1050.6       7.05       9.0275       175.4
        445      549.72      1354.7      1128.2       3.91       5.7158      138.62
     491.19      607.59        1486      1253.6       9.35       13.657      334.46
     449.44      555.82      1366.9      1131.9       5.48       8.0003      194.43
```

```
centerstats.SD
```

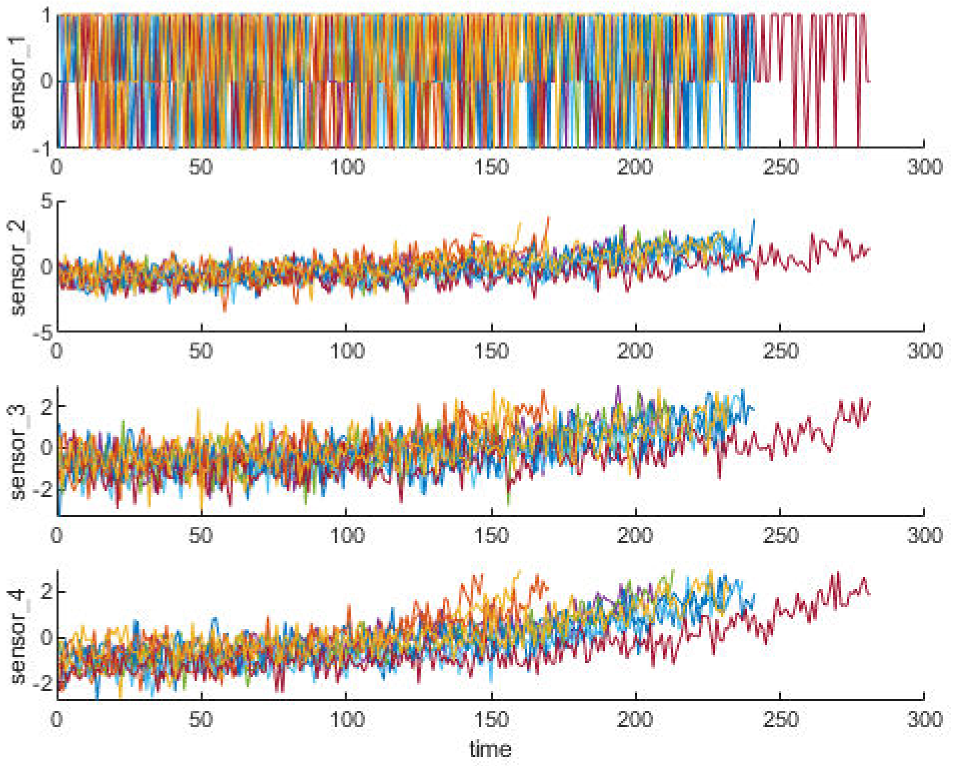
```
ans=6×21 table
      sensor_1      sensor_2    sensor_3    sensor_4     sensor_5      sensor_6     senso
    __________     ________    ________    ________    __________    _________    _____

    1.4553e-11     0.47617      5.8555      8.3464     1.1618e-12    0.0047272    0.655
     4.059e-11     0.48566      5.9258      8.8223     3.7307e-14    0.0011406    0.862
    2.7685e-11     0.35468      5.2678      6.9664      2.043e-14    0.0043301    0.450
             0     0.44169      5.6853      7.5741     1.9763e-13    0.0049401    0.441
    4.4456e-11     0.46992      5.7664      7.8679     8.9892e-13    0.0046633    0.599
    4.3489e-11     0.44341      5.7224      7.4842     3.7485e-13    0.0017642     0.47
```

The statistics in each regime can be used to normalize the training data. For each ensemble member, extract the operating points of each row, compute its distance to each cluster centers and find the nearest cluster center. Then, for each sensor measurement, subtract the mean and divide it by the standard deviation of that cluster. If the standard deviation is close to 0, set the normalized sensor measurement to 0 because a nearly constant sensor measurement is not useful for remaining useful life estimation. Refer to the last section, "Helper Functions", for more details on regimeNormalization function.

```
trainDataNormalized = cellfun(@(data) regimeNormalization(data, centers, centerstats), ...
    trainData, 'UniformOutput', false);
```

Visualize the data normalized by working regime. Degradation trends for some sensor measurements are now revealed after normalization.

```
figure
helperPlotEnsemble(trainDataNormalized, timeVariable, dataVariables(1:4), nsample)
```



Trendability Analysis

Now select the most trendable sensor measurements to construct a health indicator for prediction. For each sensor measurement, a linear degradation model is estimated and the slopes of the signals are ranked.

```
numSensors = length(dataVariables);
signalSlope = zeros(numSensors, 1);
warn = warning('off');
for ct = 1:numSensors
    tmp = cellfun(@(tbl) tbl(:, cellstr(dataVariables(ct))), trainDataNormalized, 'Uni
    mdl = linearDegradationModel(); % create model
    fit(mdl, tmp); % train mode
    signalSlope(ct) = mdl.Theta;
end
warning(warn);
```

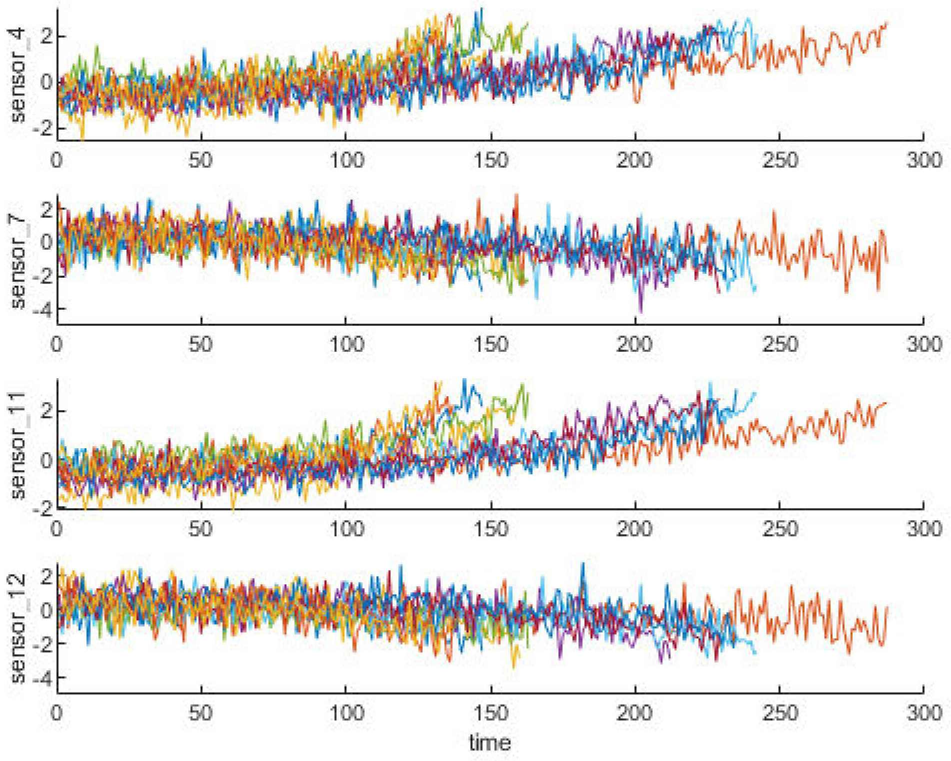
Sort the signal slopes and select 8 sensors with the largest slopes.

```
[~, idx] = sort(abs(signalSlope), 'descend');
sensorTrended = sort(idx(1:8))
```

```
sensorTrended = 8×1

     2
     3
     4
     7
    11
    12
    15
    17
```

Visualize the selected trendable sensor measurements.

```
figure
helperPlotEnsemble(trainDataNormalized, timeVariable, dataVariables(sensorTrended(3:6)
```



Notice that some of the most trendable signals show positive trends, while others show negative trends.

Construct Health Indicator

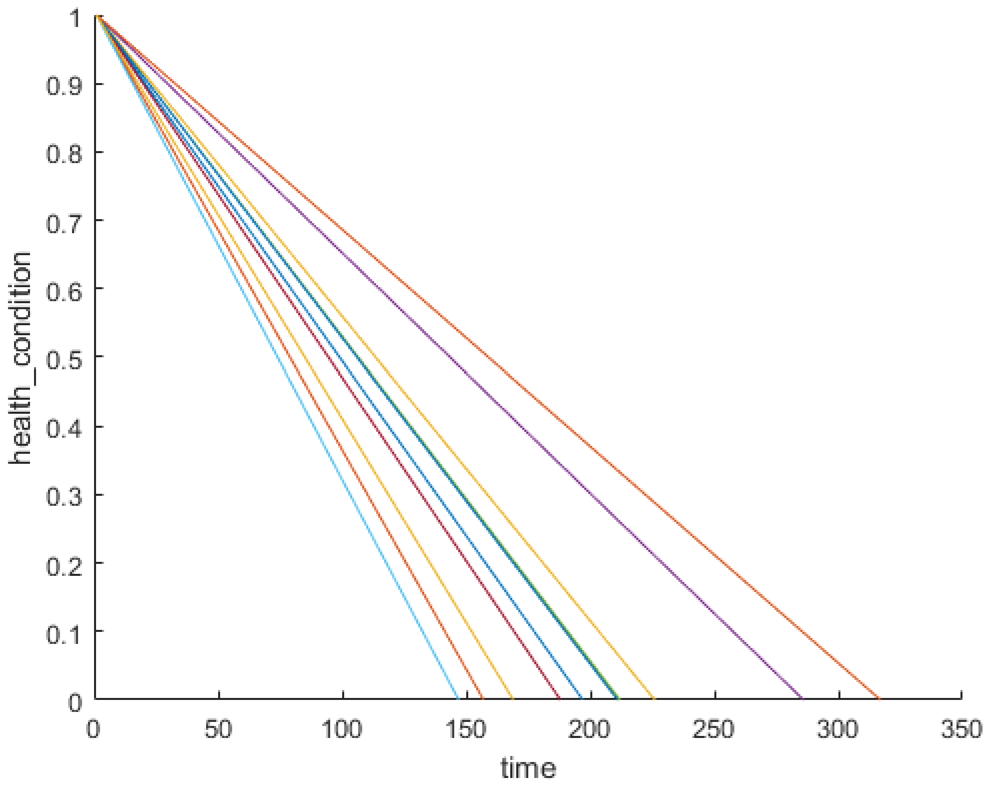
This section focuses on fusing the sensor measurements into a single health indicator, with which a similarity-based model is trained.

All the run-to-failure data is assumed to start with a healthy condition. The health condition at the beginning is assigned a value of 1 and the health condition at failure is assigned a value of 0. The health condition is assumed to be linearly degrading from 1 to 0 over time. This linear degaradtion is used to help fuse the sensor values, more sophisticated sensor fusion techniques are described in the literature [2-5].

```
for j=1:numel(trainDataNormalized)
    data = trainDataNormalized{j};
    rul = max(data.time)-data.time;
    data.health_condition = rul / max(rul);
    trainDataNormalized{j} = data;
end
```

Visualize the health condition.

```
figure
helperPlotEnsemble(trainDataNormalized, timeVariable, "health_condition", nsample)
```



The health condition of all ensemble members change from 1 to 0 with varying degrading speeds.

Now fit a linear regression model of Health Condition with the most trended sensor measurements as regressors:

Health Condition ~ 1 + Sensor2 + Sensor3 + Sensor4 + Sensor7 + Sensor11 + Sensor12 + Sensor15 + Sensor17

```
trainDataNormalizedUnwrap = vertcat(trainDataNormalized{:});

sensorToFuse = dataVariables(sensorTrended);
X = trainDataNormalizedUnwrap{:, cellstr(sensorToFuse)};
y = trainDataNormalizedUnwrap.health_condition;
regModel = fitlm(X,y);
bias = regModel.Coefficients.Estimate(1)
```

```
bias = 0.5000
```

```
weights = regModel.Coefficients.Estimate(2:end)
```

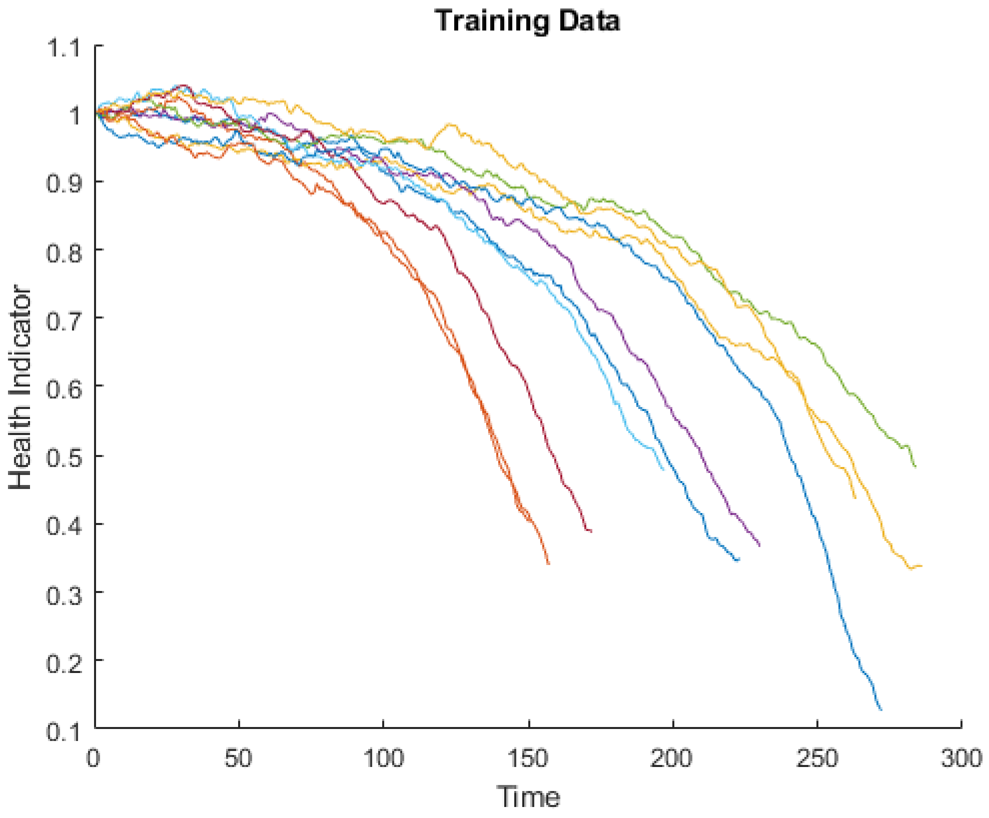
```
weights = 8×1

   -0.0308
   -0.0308
   -0.0536
    0.0033
   -0.0639
    0.0051
   -0.0408
   -0.0382
```

Construct a single health indicator by multiplying the sensor measurements with their associated weights .

```
trainDataFused = cellfun(@(data) degradationSensorFusion(data, sensorToFuse, weights),
    'UniformOutput', false);
```

Visualize the fused health indicator for training data.

```
figure
helperPlotEnsemble(trainDataFused, [], 1, nsample)
xlabel('Time')
ylabel('Health Indicator')
title('Training Data')
```



The data from multiple sensors are fused into a single health indicator. The health indicator is smoothed by a moving average filter. See helper function "degradationSensorFusion" in the last section for more details.

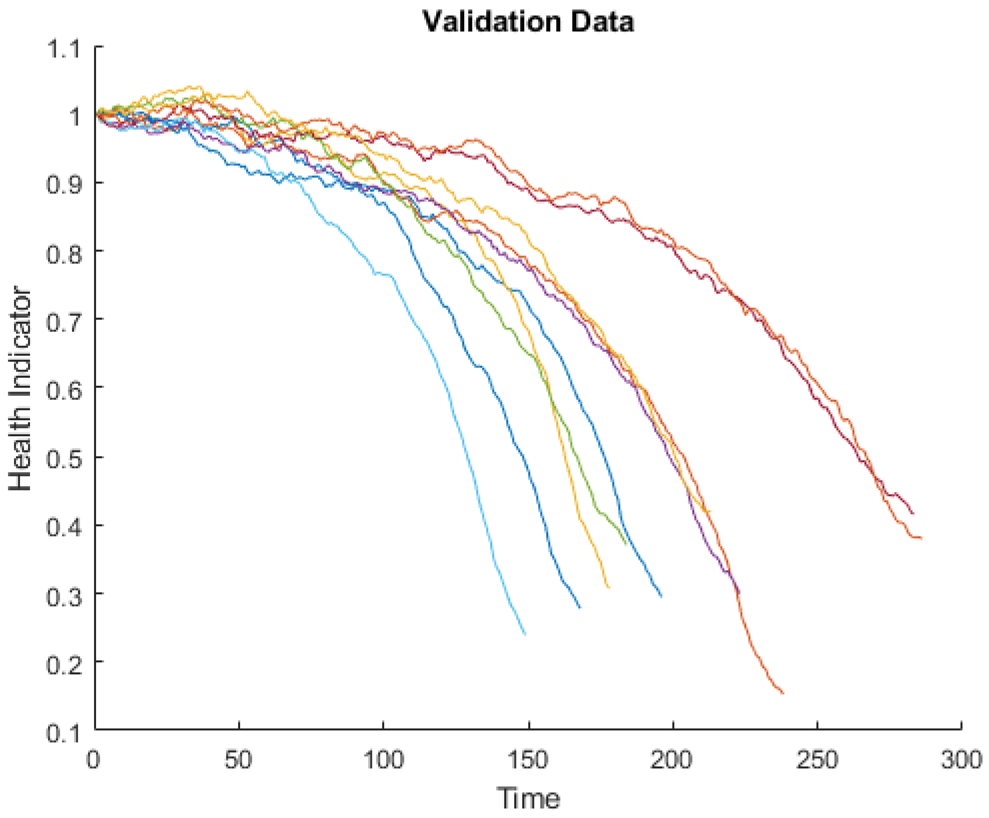
Apply same operation to validation data

Repeat the regime normalization and sensor fusion process with the validation data set.

```
validationDataNormalized = cellfun(@(data) regimeNormalization(data, centers, centersta
    validationData, 'UniformOutput', false);
validationDataFused = cellfun(@(data) degradationSensorFusion(data, sensorToFuse, weigh
    validationDataNormalized, 'UniformOutput', false);
```

Visualize the health indicator for validation data.

```
figure
helperPlotEnsemble(validationDataFused, [], 1, nsample)
xlabel('Time')
ylabel('Health Indicator')
title('Validation Data')
```



Build Similarity RUL Model

Now build a residual-based similarity RUL model using the training data. In this setting, the model tries to fit each fused data with a 2nd order polynomial.

The distance between data i and data j is computed by the 1-norm of the residual

$$d(i, j) = ||y_j - \hat{y}_{j,i}||_1$$

where y_j is the health indicator of machine j, $\hat{y}_{j,i}$ is the estimated health indicator of machine j using the 2nd order polynomial model identified in machine i.

The similarity score is computed by the following formula

$$score(i, j) = exp(-|d(i, j)|)$$

Given one ensemble member in the validation data set, the model will find the nearest 50 ensemble members in the training data set, fit a probability distribution based on the 50 ensemble members, and use the median of the distribution as an estimate of RUL.

```
mdl = residualSimilarityModel(...
    'Method', 'poly2',...
    'Distance', 'absolute',...
    'NumNearestNeighbors', 50,...
    'Standardize', 1);

fit(mdl, trainDataFused);
```

Performance Evaluation

To evaluate the similarity RUL model, use 50%, 70% and 90% of a sample validation data to predict its RUL.

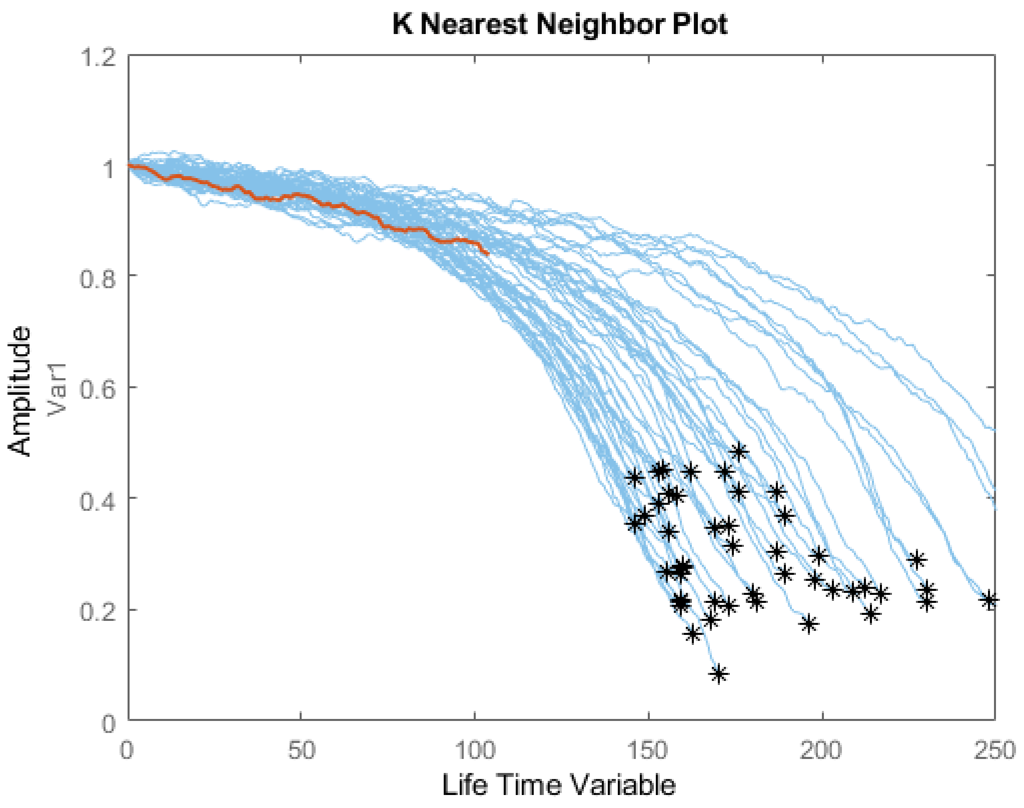
```
breakpoint = [0.5, 0.7, 0.9];
validationDataTmp = validationDataFused{3}; % use one validation data for illustration
```

Use the validation data before the first breakpoint, which is 50% of the lifetime.

```
bpidx = 1;
validationDataTmp50 = validationDataTmp(1:ceil(end*breakpoint(bpidx)),:);
trueRUL = length(validationDataTmp) - length(validationDataTmp50);
[estRUL, ciRUL, pdfRUL] = predictRUL(mdl, validationDataTmp50);
```

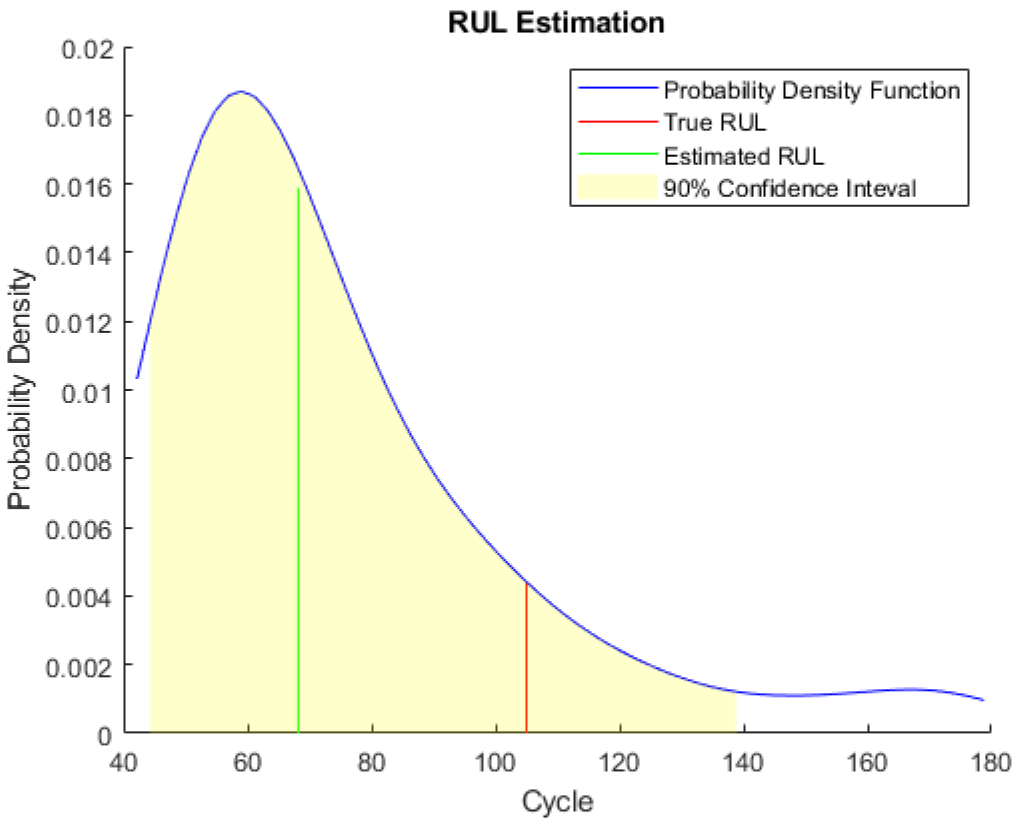
Visualize the validation data truncated at 50% and its nearest neighbors.

```
figure
compare(mdl, validationDataTmp50);
```



Visualize the estimated RUL compared to the true RUL and the probability distribution of the estimated RUL.

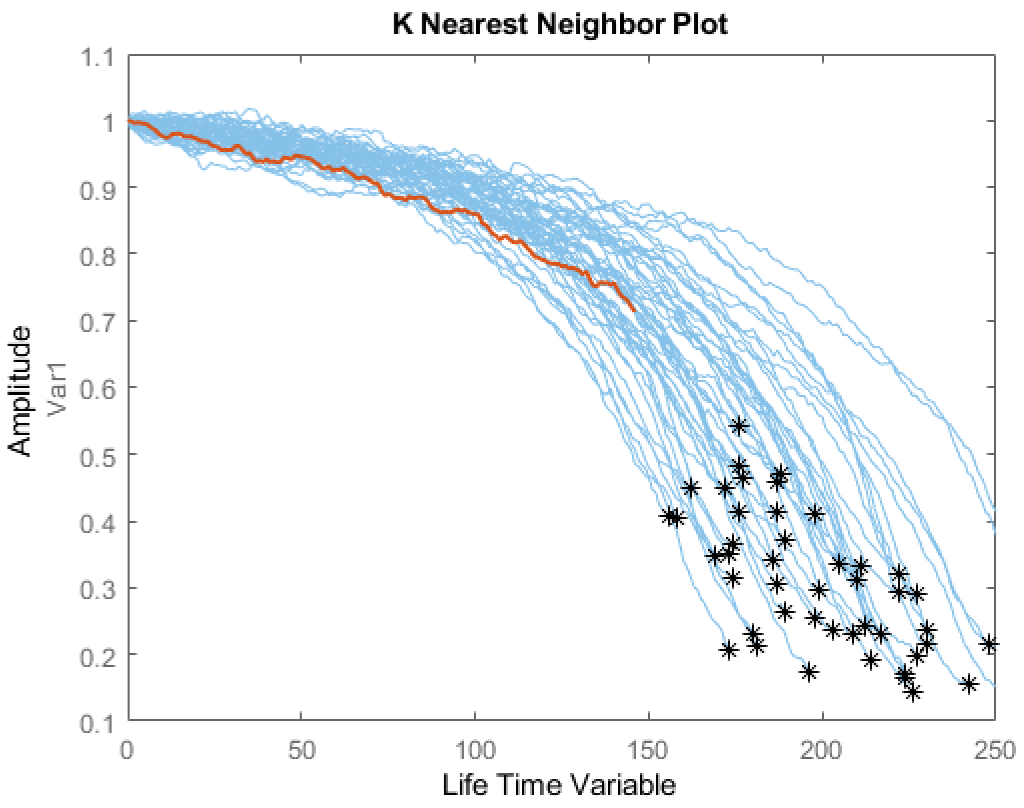
```
figure
helperPlotRULDistribution(trueRUL, estRUL, pdfRUL, ciRUL)
```



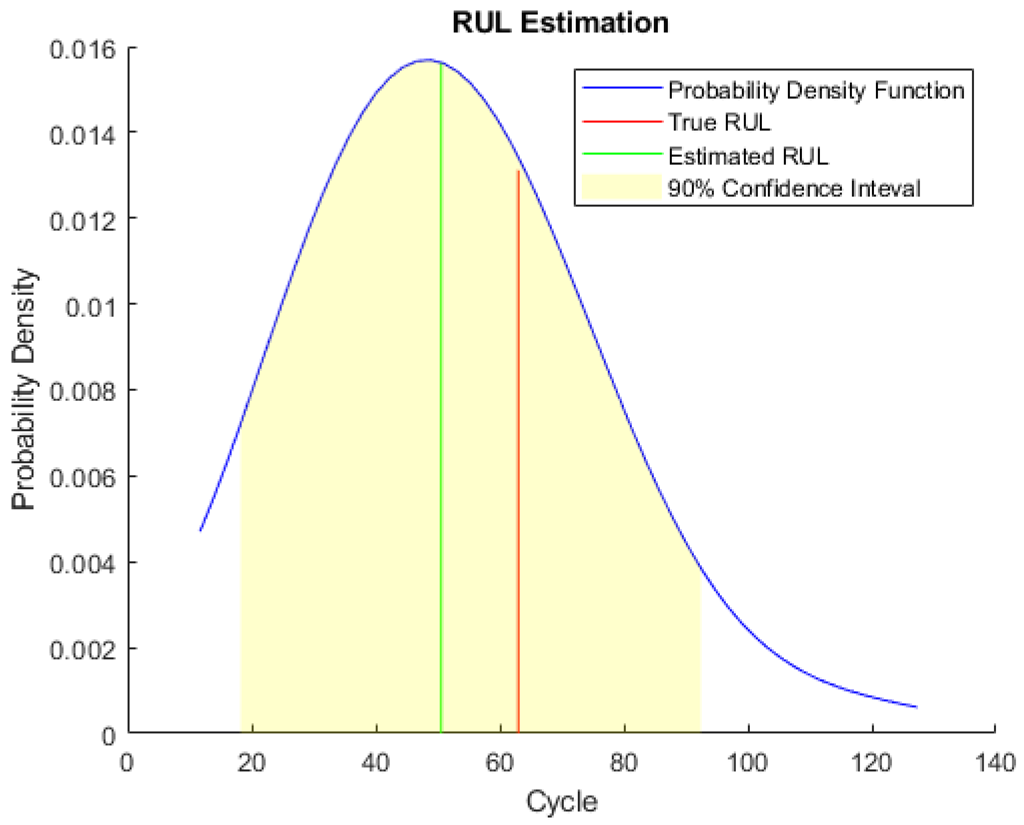
There is a relatively large error between the estimated RUL and the true RUL when the machine is in an intermediate health stage. In this example, the most similar 10 curves are close at the beginning, but bifurcate when they approach the failure state, resulting in roughly two modes in the RUL distribution.

Use the validation data before the second breakpoint, which is 70% of the lifetime.

```
bpidx = 2;
validationDataTmp70 = validationDataTmp(1:ceil(end*breakpoint(bpidx)), :);
trueRUL = length(validationDataTmp) - length(validationDataTmp70);
[estRUL,ciRUL,pdfRUL] = predictRUL(mdl, validationDataTmp70);

figure
compare(mdl, validationDataTmp70);
```



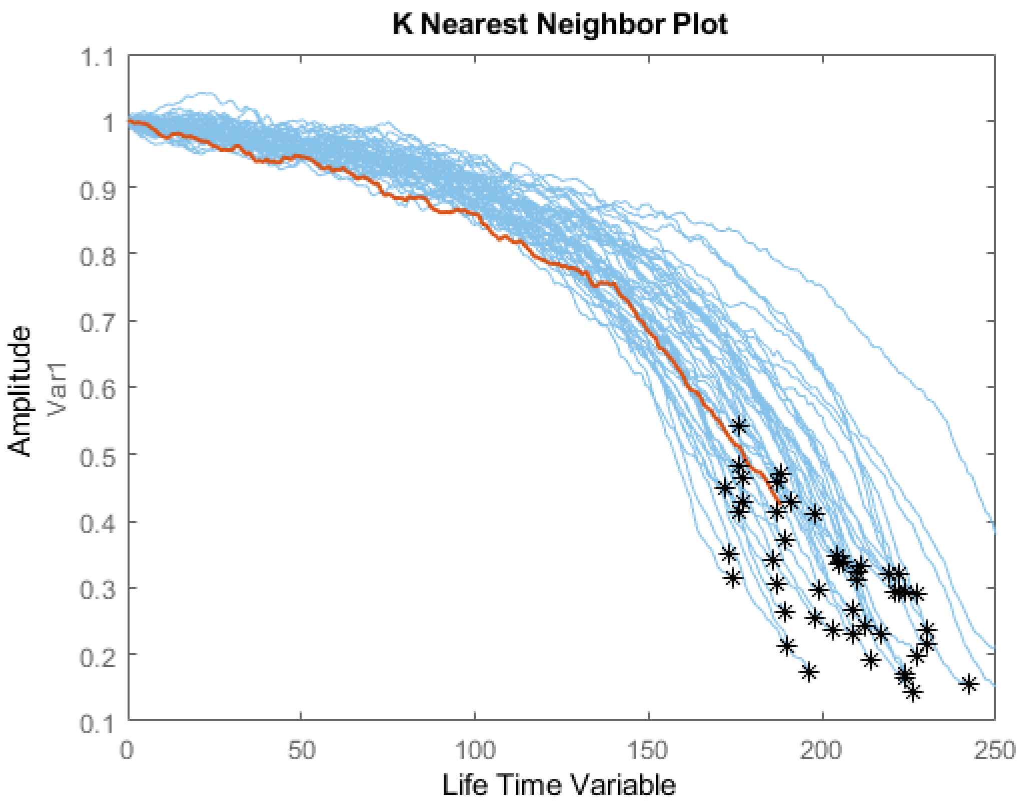
```
figure
helperPlotRULDistribution(trueRUL, estRUL, pdfRUL, ciRUL)
```



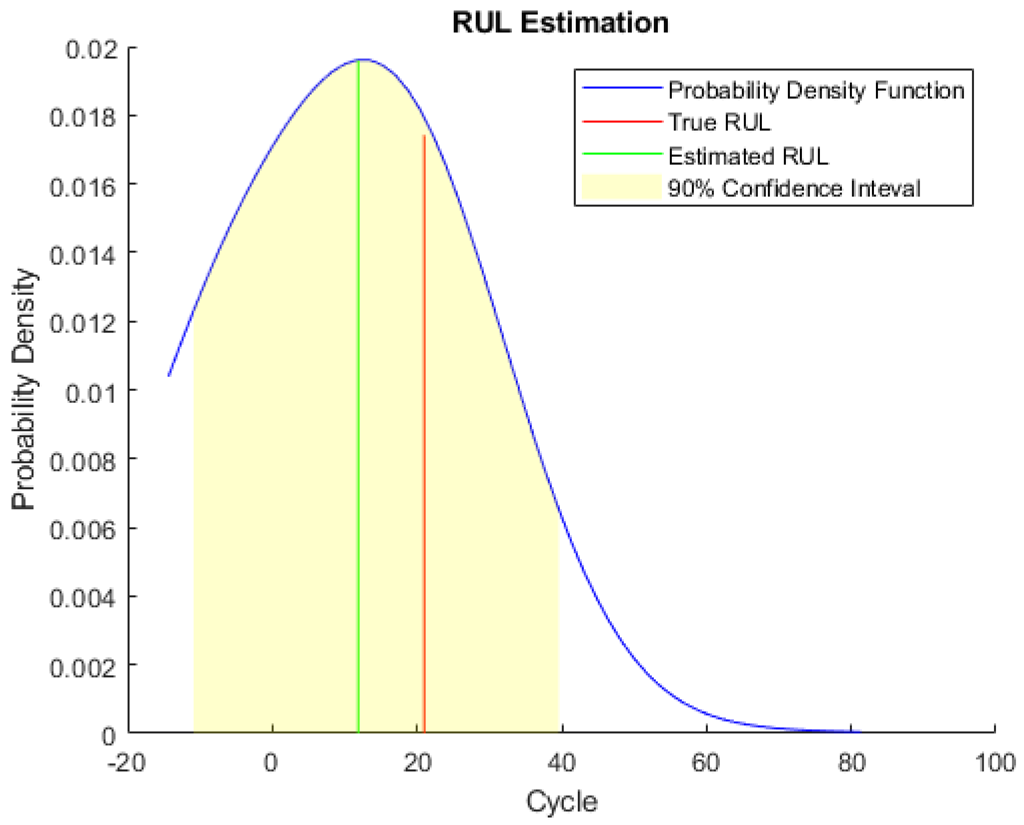
When more data observed, the RUL estimation is enhanced.

Use the validation data before the third breakpoint, which is 90% of the lifetime.

```
bpidx = 3;
validationDataTmp90 = validationDataTmp(1:ceil(end*breakpoint(bpidx)), :);
trueRUL = length(validationDataTmp) - length(validationDataTmp90);
[estRUL,ciRUL,pdfRUL] = predictRUL(mdl, validationDataTmp90);

figure
compare(mdl, validationDataTmp90);
```



```
figure
helperPlotRULDistribution(trueRUL, estRUL, pdfRUL, ciRUL)
```



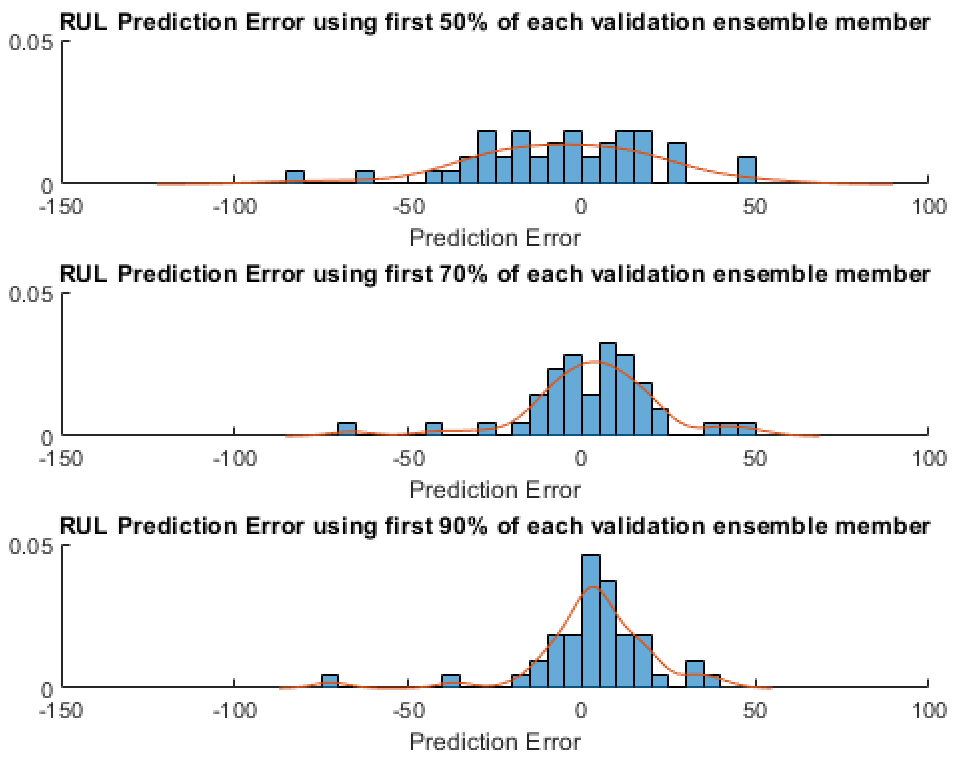
When the machine is close to failure, the RUL estimation is even more enhanced in this example.

Now repeat the same evaluation procedure for the whole validation data set and compute the error between estimated RUL and true RUL for each breakpoint.

```
numValidation = length(validationDataFused);
numBreakpoint = length(breakpoint);
error = zeros(numValidation, numBreakpoint);

for dataIdx = 1:numValidation
    tmpData = validationDataFused{dataIdx};
    for bpidx = 1:numBreakpoint
        tmpDataTest = tmpData(1:ceil(end*breakpoint(bpidx)), :);
        trueRUL = length(tmpData) - length(tmpDataTest);
        [estRUL, ~, ~] = predictRUL(mdl, tmpDataTest);
        error(dataIdx, bpidx) = estRUL - trueRUL;
    end
end
```

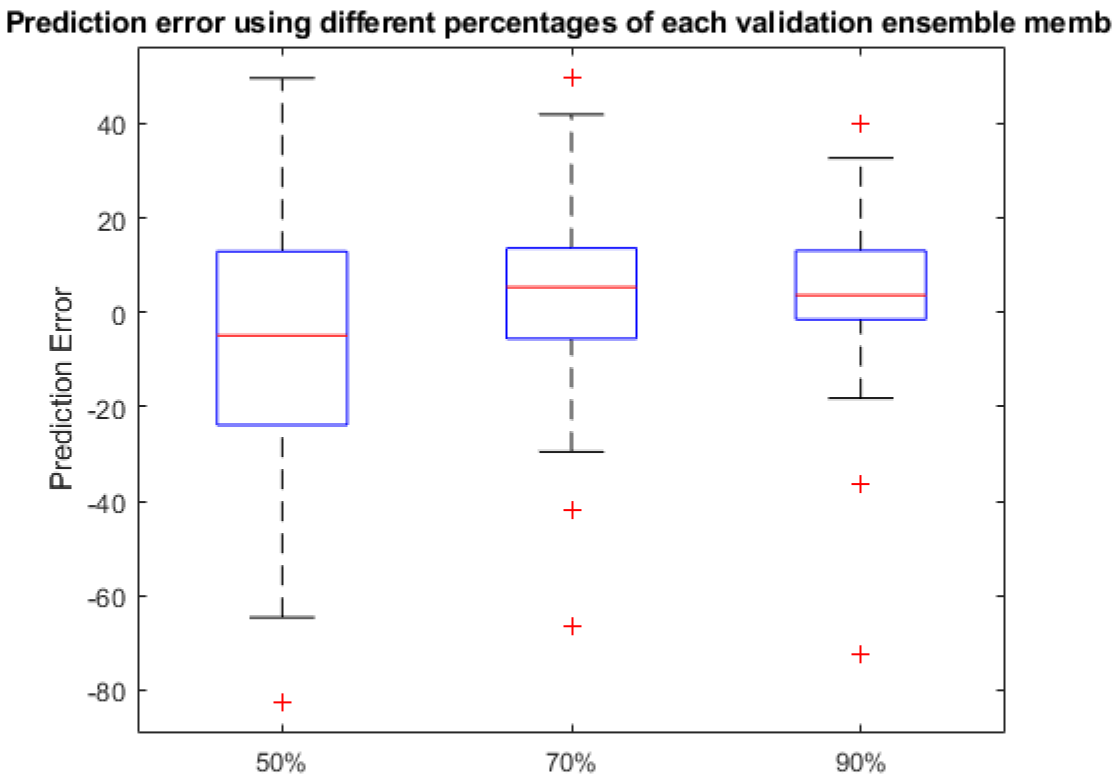
Visualize the histogram of the error for each breakpoint together with its probability distribution.

```
[pdf50, x50] = ksdensity(error(:, 1));
[pdf70, x70] = ksdensity(error(:, 2));
[pdf90, x90] = ksdensity(error(:, 3));

figure
ax(1) = subplot(3,1,1);
hold on
histogram(error(:, 1), 'BinWidth', 5, 'Normalization', 'pdf')
plot(x50, pdf50)
hold off
xlabel('Prediction Error')
title('RUL Prediction Error using first 50% of each validation ensemble member')

ax(2) = subplot(3,1,2);
hold on
histogram(error(:, 2), 'BinWidth', 5, 'Normalization', 'pdf')
plot(x70, pdf70)
hold off
xlabel('Prediction Error')
title('RUL Prediction Error using first 70% of each validation ensemble member')

ax(3) = subplot(3,1,3);
hold on
histogram(error(:, 3), 'BinWidth', 5, 'Normalization', 'pdf')
plot(x90, pdf90)
hold off
xlabel('Prediction Error')
title('RUL Prediction Error using first 90% of each validation ensemble member')
linkaxes(ax)
```



Plot the prediction error in box plot to visualize the median, 25-75 quantile and outliers.

```
figure
boxplot(error, 'Labels', {'50%', '70%', '90%'})
ylabel('Prediction Error')
title('Prediction error using different percentages of each validation ensemble member
```



Compute and visualize the mean and standard deviation of the prediction error.

```
errorMean = mean(error)
```

```
errorMean = 1×3

   -5.8944    3.1359    3.3555
```

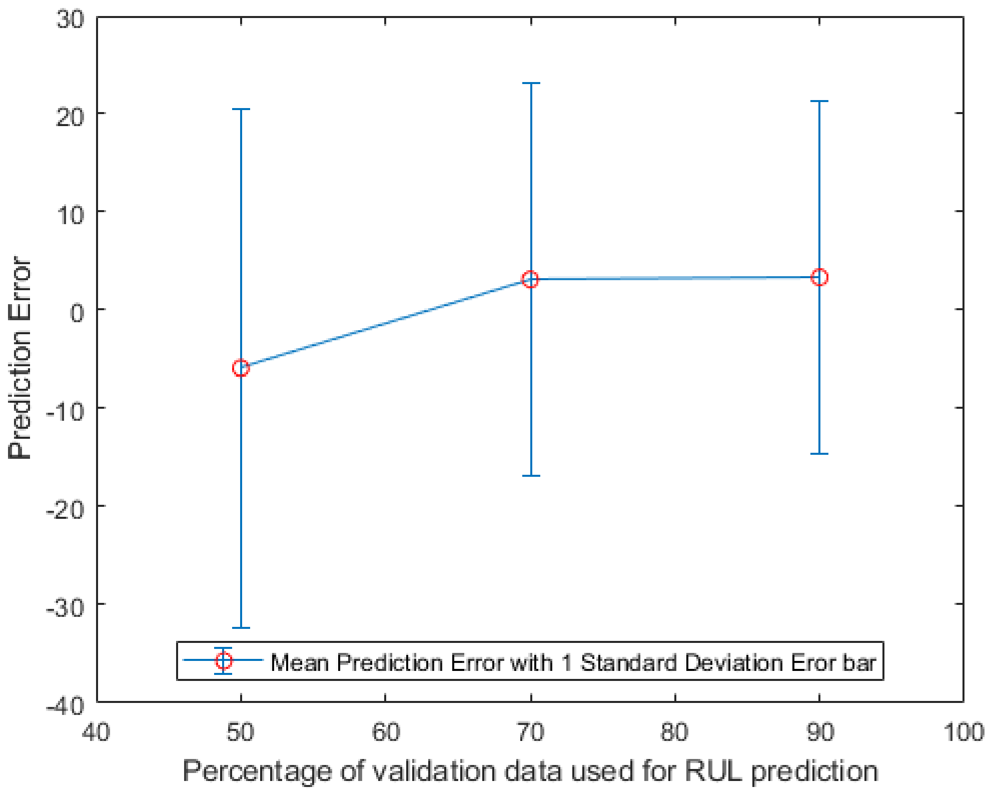
```
errorMedian = median(error)
```

```
errorMedian = 1×3

   -4.8538    5.3763    3.6580
```

```
errorSD = std(error)
```

```
errorSD = 1×3

   26.4916   20.0720   18.0313
```

```
figure
errorbar([50 70 90], errorMean, errorSD, '-o', 'MarkerEdgeColor','r')
xlim([40, 100])
xlabel('Percentage of validation data used for RUL prediction')
ylabel('Prediction Error')
legend('Mean Prediction Error with 1 Standard Deviation Eror bar', 'Location', 'south')
```



It is shown that the error becomes more concentrated around 0 (less outliers) as more data is observed.

References

[1] A. Saxena and K. Goebel (2008). "PHM08 Challenge Data Set", NASA Ames Prognostics Data Repository (http://ti.arc.nasa.gov/project/prognostic-data-repository), NASA Ames Research Center, Moffett Field, CA

[2] Roemer, Michael J., Gregory J. Kacprzynski, and Michael H. Schoeller. "Improved diagnostic and prognostic assessments using health management information fusion." *AUTOTESTCON Proceedings, 2001. IEEE Systems Readiness Technology Conference*. IEEE, 2001.

[3] Goebel, Kai, and Piero Bonissone. "Prognostic information fusion for constant load systems." *Information Fusion, 2005 8th International Conference on*. Vol. 2. IEEE, 2005.

[4] Wang, Peng, and David W. Coit. "Reliability prediction based on degradation modeling for systems with multiple degradation measures." *Reliability and Maintainability, 2004 Annual Symposium-RAMS*. IEEE, 2004.

[5] Jardine, Andrew KS, Daming Lin, and Dragan Banjevic. "A review on machinery diagnostics and prognostics implementing condition-based maintenance." *Mechanical systems and signal processing* 20.7 (2006): 1483-1510.

Helper Functions

```
function data = regimeNormalization(data, centers, centerstats)
% Perform normalization for each observation (row) of the data
% according to the cluster the observation belongs to.
conditionIdx = 3:5;
dataIdx = 6:26;

% Perform row-wise operation
data{:, dataIdx} = table2array(...
    rowfun(@(row) localNormalize(row, conditionIdx, dataIdx, centers, centerstats), ..
    data, 'SeparateInputs', false));
end

function rowNormalized = localNormalize(row, conditionIdx, dataIdx, centers, centerstat
% Normalization operation for each row.

% Get the operating points and sensor measurements
```

```
ops = row(1, conditionIdx);
sensor = row(1, dataIdx);

% Find which cluster center is the closest
dist = sum((centers - ops).^2, 2);
[~, idx] = min(dist);

% Normalize the sensor measurements by the mean and standard deviation of the cluster.
% Reassign NaN and Inf to 0.
rowNormalized = (sensor - centerstats.Mean{idx, :}) ./ centerstats.SD{idx, :};
rowNormalized(isnan(rowNormalized) | isinf(rowNormalized)) = 0;
end

function dataFused = degradationSensorFusion(data, sensorToFuse, weights)
% Combine measurements from different sensors according
% to the weights, smooth the fused data and offset the data
% so that all the data start from 1

% Fuse the data according to weights
dataToFuse = data{:, cellstr(sensorToFuse)};
dataFusedRaw = dataToFuse*weights;

% Smooth the fused data with moving mean
stepBackward = 10;
stepForward = 10;
dataFused = movmean(dataFusedRaw, [stepBackward stepForward]);

% Offset the data to 1
dataFused = dataFused + 1 - dataFused(1);
end
```

See Also

`residualSimilarityModel`

More About

- “Feature Selection for Remaining Useful Life Prediction” on page 5-2
- “RUL Estimation Using RUL Estimator Models” on page 5-8
- “Wind Turbine High-Speed Bearing Prognosis” on page 5-44

Wind Turbine High-Speed Bearing Prognosis

This example shows how to build an exponential degradation model to predict the Remaining Useful Life (RUL) of a wind turbine bearing in real time. The exponential degradation model predicts the RUL based on its parameter priors and the latest measurements (historical run-to-failure data can help estimate the model parameters priors, but they are not required). The model is able to detect the significant degradation trend in real time and updates its parameter priors when a new observation becomes available. The example follows a typical prognosis workflow: data import and exploration, feature extraction and postprocessing, feature importance ranking and fusion, model fitting and prediction, and performance analysis.

Dataset

The dataset is collected from a 2MW wind turbine high-speed shaft driven by a 20-tooth pinion gear [1]. A vibration signal of 6 seconds was acquired each day for 50 consecutive days (there are 2 measurements on March 17, which are treated as two days in this example). An inner race fault developed and caused the failure of the bearing across the 50-day period.

A compact version of the dataset is available in the toolbox. To use the compact dataset, copy the dataset to the current folder and enable its write permission.

```
copyfile(...
    fullfile(matlabroot, 'toolbox', 'predmaint', ...
    'predmaintdemos', 'windTurbineHighSpeedBearingPrognosis'), ...
    'WindTurbineHighSpeedBearingPrognosis-Data-master')
fileattrib(fullfile('WindTurbineHighSpeedBearingPrognosis-Data-master', '*.mat'), '+w'
```

The measurement time step for the compact dataset is 5 days.

```
timeUnit = '\times 5 day';
```

For the full dataset, go to this link https://github.com/mathworks/WindTurbineHighSpeedBearingPrognosis-Data, download the entire repository as a zip file and save it in the same directory as this live script. Unzip the file using this command. The measurement time step for the full dataset is 1 day.

```
if exist('WindTurbineHighSpeedBearingPrognosis-Data-master.zip', 'file')
    unzip('WindTurbineHighSpeedBearingPrognosis-Data-master.zip')
    timeUnit = 'day';
end
```

The results in this example are generated from the full dataset. It is highly recommended to download the full dataset to run this example. Results generated from the compact dataset might not be meaningful.

Data Import

Create a `fileEnsembleDatastore` of the wind turbine data. The data contains a vibration signal and a tachometer signal. The `fileEnsembleDatastore` will parse the file name and extract the date information as `IndependentVariables`. See the Helper Functions section for more details.

```
hsbearing = fileEnsembleDatastore(...
    fullfile('.', 'WindTurbineHighSpeedBearingPrognosis-Data-master'), ...
    '.mat');
hsbearing.DataVariables = ["vibration", "tach"];
hsbearing.IndependentVariables = "Date";
hsbearing.SelectedVariables = ["Date", "vibration", "tach"];
hsbearing.ReadFcn = @helperReadData;
hsbearing.WriteToMemberFcn = @helperWriteToHSBearing;
tall(hsbearing)
```

```
ans =

  M×3 tall table

            Date                vibration              tach
    ____________________    _________________    _______________

    07-Mar-2013 01:57:46    [585936×1 double]    [2446×1 double]
    08-Mar-2013 02:34:21    [585936×1 double]    [2411×1 double]
    09-Mar-2013 02:33:43    [585936×1 double]    [2465×1 double]
    10-Mar-2013 03:01:02    [585936×1 double]    [2461×1 double]
    11-Mar-2013 03:00:24    [585936×1 double]    [2506×1 double]
    12-Mar-2013 06:17:10    [585936×1 double]    [2447×1 double]
    13-Mar-2013 06:34:04    [585936×1 double]    [2438×1 double]
    14-Mar-2013 06:50:41    [585936×1 double]    [2390×1 double]
             :                      :                    :
             :                      :                    :
```

Sample rate of vibration signal is 97656 Hz.

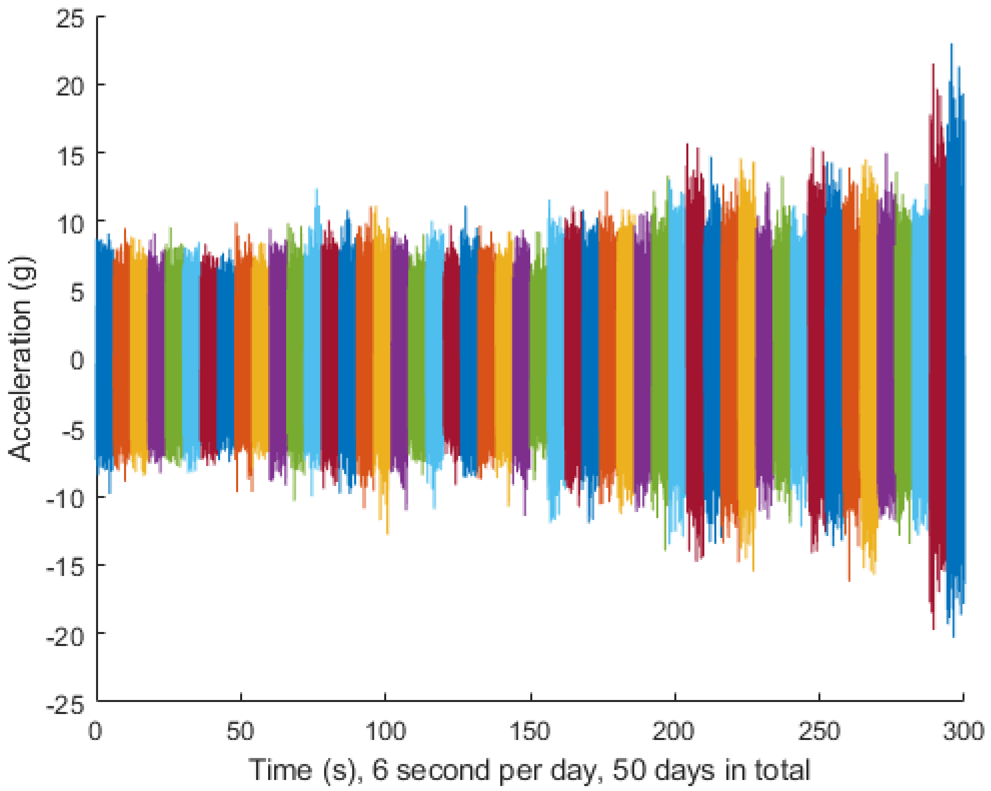
```
fs = 97656; % Hz
```

Data Exploration

This section explores the data in both time domain and frequency domain and seeks inspiration of what features to extract for prognosis purposes.

First visualize the vibration signals in the time domain. In this dataset, there are 50 vibration signals of 6 seconds measured in 50 consecutive days. Now plot the 50 vibration signals one after each other.

```
reset(hsbearing)
tstart = 0;
figure
hold on
while hasdata(hsbearing)
    data = read(hsbearing);
    v = data.vibration{1};
    t = tstart + (1:length(v))/fs;
    % Downsample the signal to reduce memory usage
    plot(t(1:10:end), v(1:10:end))
    tstart = t(end);
end
hold off
xlabel('Time (s), 6 second per day, 50 days in total')
ylabel('Acceleration (g)')
```

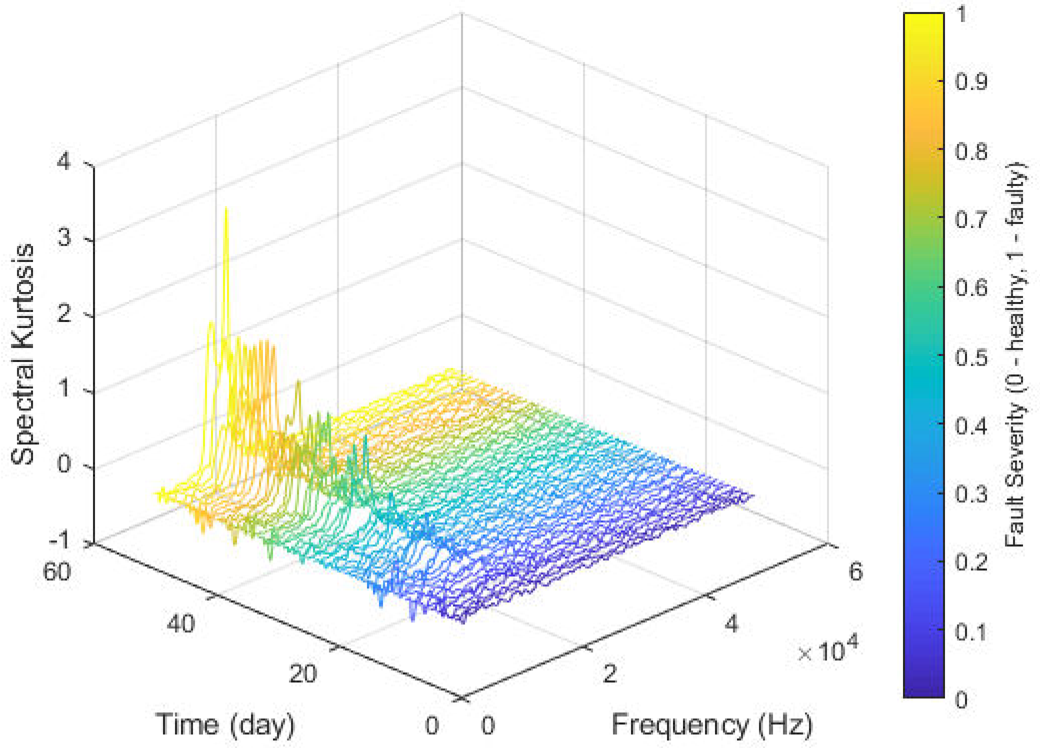


The vibration signals in time domain reveals an increasing trend of the signal impulsiveness. Indicators quantifying the impulsiveness of the signal, such as kurtosis, peak-to-peak value, crest factors *etc.*, are potential prognostic features for this wind turbine bearing dataset [2].

On the other hand, spectral kurtosis is considered powerful tool for wind turbine prognosis in frequency domain [3]. To visualize the spectral kurtosis changes along time, plot the spectral kurtosis values as a function of frequency and the measurement day.

```
hsbearing.DataVariables = ["vibration", "tach", "SpectralKurtosis"];
colors = parula(50);
figure
hold on
reset(hsbearing)
```

```
day = 1;
while hasdata(hsbearing)
    data = read(hsbearing);
    data2add = table;

    % Get vibration signal and measurement date
    v = data.vibration{1};

    % Compute spectral kurtosis with window size = 128
    wc = 128;
    [SK, F] = pkurtosis(v, fs, wc);
    data2add.SpectralKurtosis = {table(F, SK)};

    % Plot the spectral kurtosis
    plot3(F, day*ones(size(F)), SK, 'Color', colors(day, :))

    % Write spectral kurtosis values
    writeToLastMemberRead(hsbearing, data2add);

    % Increment the number of days
    day = day + 1;
end
hold off
xlabel('Frequency (Hz)')
ylabel('Time (day)')
zlabel('Spectral Kurtosis')
grid on
view(-45, 30)
cbar = colorbar;
ylabel(cbar, 'Fault Severity (0 - healthy, 1 - faulty)')
```



Fault Severity indicated in colorbar is the measurement date normalized into 0 to 1 scale. It is observed that the spectral kurtosis value around 10 kHz gradually increases as the machine condition degrades. Statistical features of the spectral kurtosis, such as mean, standard deviation *etc.*, will be potential indicators of the bearing degradation [3].

Feature Extraction

Based on the analysis in the previous section, a collection of statistical features derived from time-domain signal and spectral kurtosis are going to be extracted. More mathematical details about the features are provided in [2-3].

First, pre-assign the feature names in DataVariables before writing them into the fileEnsembleDatastore.

```
hsbearing.DataVariables = [hsbearing.DataVariables; ...
    "Mean"; "Std"; "Skewness"; "Kurtosis"; "Peak2Peak"; ...
    "RMS"; "CrestFactor"; "ShapeFactor"; "ImpulseFactor"; "MarginFactor"; "Energy"; ..
    "SKMean"; "SKStd"; "SKSkewness"; "SKKurtosis"];
```

Compute feature values for each ensemble member.

```
hsbearing.SelectedVariables = ["vibration", "SpectralKurtosis"];
reset(hsbearing)
while hasdata(hsbearing)
    data = read(hsbearing);
    v = data.vibration{1};
    SK = data.SpectralKurtosis{1}.SK;
    features = table;

    % Time Domain Features
    features.Mean = mean(v);
    features.Std = std(v);
    features.Skewness = skewness(v);
    features.Kurtosis = kurtosis(v);
    features.Peak2Peak = peak2peak(v);
    features.RMS = rms(v);
    features.CrestFactor = max(v)/features.RMS;
    features.ShapeFactor = features.RMS/mean(abs(v));
    features.ImpulseFactor = max(v)/mean(abs(v));
    features.MarginFactor = max(v)/mean(abs(v))^2;
    features.Energy = sum(v.^2);

    % Spectral Kurtosis related features
    features.SKMean = mean(SK);
    features.SKStd = std(SK);
    features.SKSkewness = skewness(SK);
    features.SKKurtosis = kurtosis(SK);

    % write the derived features to the corresponding file
    writeToLastMemberRead(hsbearing, features);
end
```

Select the independent variable `Date` and all the extracted features to construct the feature table.

```
hsbearing.SelectedVariables = ["Date", "Mean", "Std", "Skewness", "Kurtosis", "Peak2Pea
    "RMS", "CrestFactor", "ShapeFactor", "ImpulseFactor", "MarginFactor", "Energy", ..
    "SKMean", "SKStd", "SKSkewness", "SKKurtosis"];
```

Since the feature table is small enough to fit in memory (50 by 15), gather the table before processing. For big data, it is recommended to perform operations in tall format until you are confident that the output is small enough to fit in memory.

```
featureTable = gather(tall(hsbearing));
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1 sec
Evaluation completed in 1 sec
```

Convert the table to timetable so that the time information is always associated with the feature values.

```
featureTable = table2timetable(featureTable)
```

```
featureTable=50×15 timetable
            Date                Mean       Std        Skewness       Kurtosis    Peak2Peak
    ____________________    _______    ______    ___________    ________    _________

    07-Mar-2013 01:57:46    0.34605    2.2705      0.0038699     2.9956       21.621
    08-Mar-2013 02:34:21    0.24409    2.0621      0.0030103     3.0195        19.31
    09-Mar-2013 02:33:43    0.21873    2.1036     -0.0010289     3.0224       21.474
    10-Mar-2013 03:01:02    0.21372    2.0081       0.001477     3.0415        19.52
    11-Mar-2013 03:00:24    0.21518    2.0606      0.0010116     3.0445       21.217
    12-Mar-2013 06:17:10    0.29335    2.0791      -0.008428      3.018        20.05
    13-Mar-2013 06:34:04    0.21293     1.972     -0.0014294     3.0174       18.837
    14-Mar-2013 06:50:41    0.24401    1.8114      0.0022161     3.0057       17.862
    15-Mar-2013 06:50:03    0.20984    1.9973       0.001559     3.0711        21.12
    16-Mar-2013 06:56:43    0.23318    1.9842     -0.0019594     3.0072       18.832
    17-Mar-2013 06:56:04    0.21657     2.113     -0.0013711     3.1247       21.858
    17-Mar-2013 18:47:56    0.19381    2.1335      -0.012744     3.0934       21.589
    18-Mar-2013 18:47:15    0.21919    2.1284     -0.0002039     3.1647       24.051
    20-Mar-2013 00:33:54    0.35865    2.2536      -0.002308     3.0817       22.633
    21-Mar-2013 00:33:14     0.1908    2.1782    -0.00019286     3.1548       25.515
    22-Mar-2013 00:39:50    0.20569    2.1861      0.0020375     3.2691       26.439
      ⋮
```

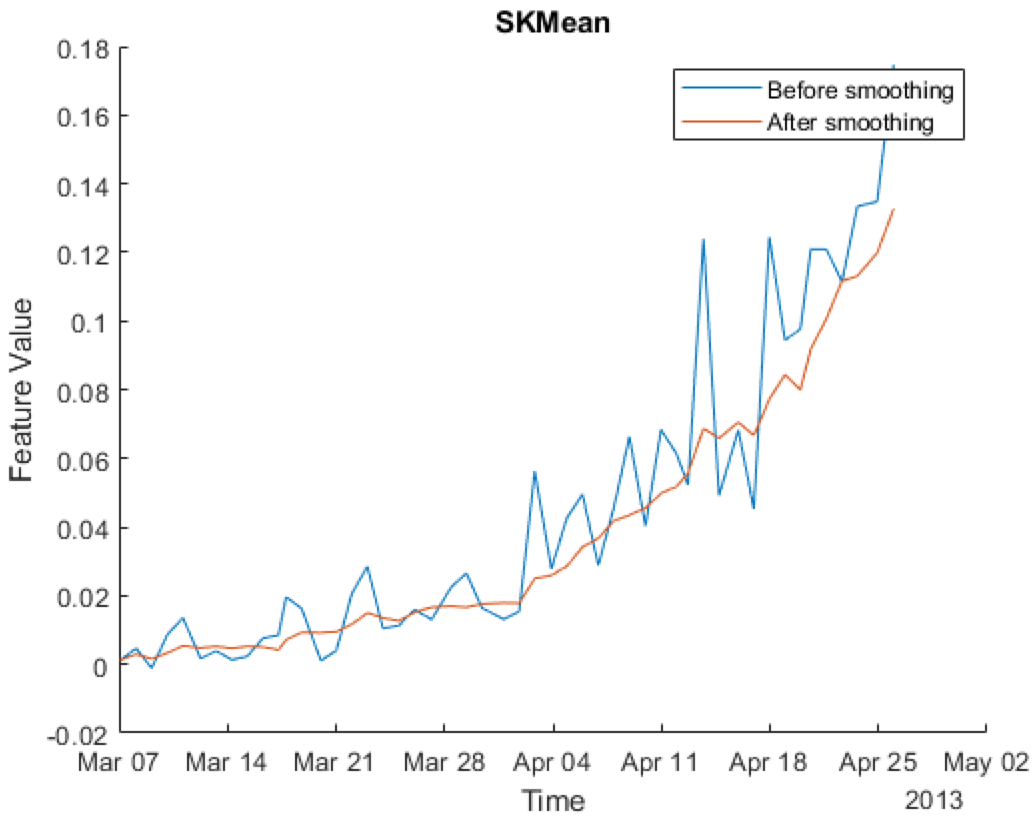
Feature Postprocessing

Extracted features are usually associated with noise. The noise with opposite trend can sometimes be harmful to the RUL prediction. In addition, one of the feature performance metrics, monotonicity, to be introduced next is not robust to noise. Therefore, a causal moving mean filter with a lag window of 5 steps is applied to the extracted features, where "causal" means no future value is used in the moving mean filtering.

```
variableNames = featureTable.Properties.VariableNames;
featureTableSmooth = varfun(@(x) movmean(x, [5 0]), featureTable);
featureTableSmooth.Properties.VariableNames = variableNames;
```

Here is an example showing the feature before and after smoothing.

```
figure
hold on
plot(featureTable.Date, featureTable.SKMean)
plot(featureTableSmooth.Date, featureTableSmooth.SKMean)
hold off
xlabel('Time')
ylabel('Feature Value')
legend('Before smoothing', 'After smoothing')
title('SKMean')
```



Moving mean smoothing introduces a time delay of the signal, but the delay effect can be mitigated by selecting proper threshold in the RUL prediction.

Training Data

In practice, the data of the whole life cycle is not available when developing the prognostic algorithm, but it is reasonable to assume that some data in the early stage of the life cycle has been collected. Hence data collected in the first 20 days (40% of the life cycle) is treated as training data. The following feature importance ranking and fusion is only based on the training data.

```
breaktime = datetime(2013, 3, 27);
breakpoint = find(featureTableSmooth.Date < breaktime, 1, 'last');
trainData = featureTableSmooth(1:breakpoint, :);
```

Feature Importance Ranking

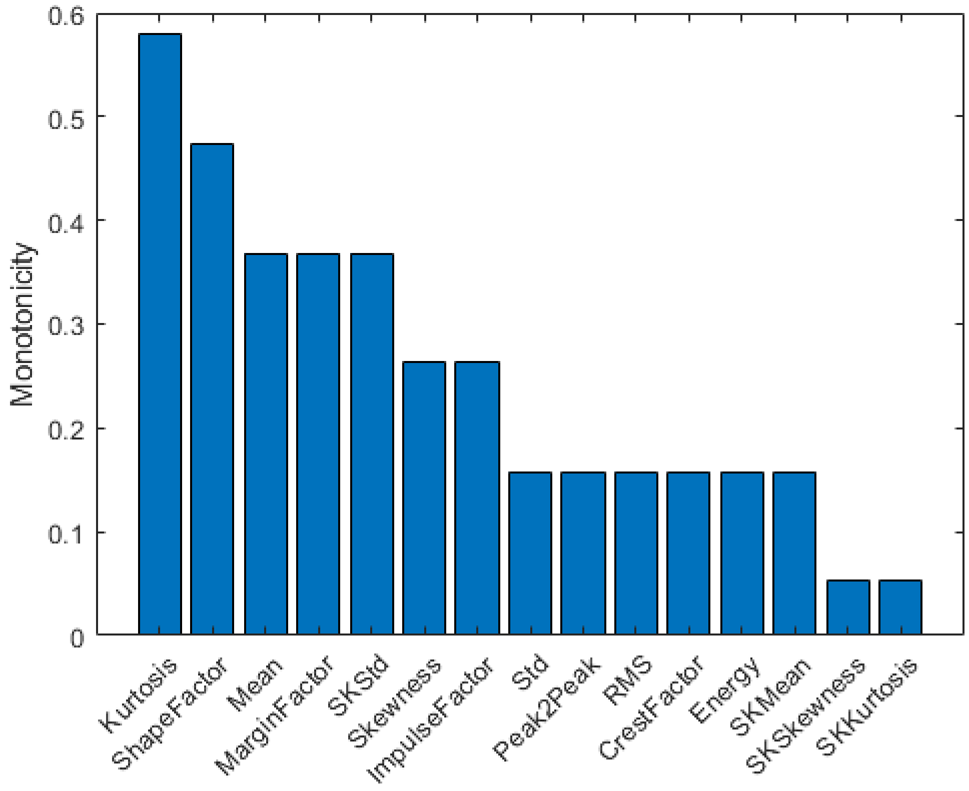
In this example, monotonicity proposed by [3] is used to quantify the merit of the features for prognosis purpose.

Monotonicity of ith feature x_i is computed as

$$\text{Monotonicity}(x_i) = \frac{1}{m}\sum_{j=1}^{m}\frac{\left|\text{number of positive diff}(x_i^j) - \text{number of negative diff}(x_i^j)\right|}{n-1}$$

where n is the number of measurement points, in this case $n = 50$. m is the number of machines monitored, in this case $m = 1$. x_i^j is the ith feature measured on jth machine. $\text{diff}(x_i^j) = x_i^j(t) - x_i^j(t-1)$, i.e. the difference of the signal x_i^j.

```
% Since moving window smoothing is already done, set 'WindowSize' to 0 to
% turn off the smoothing within the function
featureImportance = monotonicity(trainData, 'WindowSize', 0);
helperSortedBarPlot(featureImportance, 'Monotonicity');
```



Kurtosis of the signal is the top feature based on the monotonicity.

Features with feature importance score larger than 0.3 are selected for feature fusion in the next section.

```
trainDataSelected = trainData(:, featureImportance{:,:}>0.3);
featureSelected = featureTableSmooth(:, featureImportance{:,:}>0.3)
```

```
featureSelected=50×5 timetable
             Date               Mean       Kurtosis     ShapeFactor     MarginFactor      SKS
    ____________________     _______     ________     ___________     ____________     ____

    07-Mar-2013 01:57:46     0.34605      2.9956         1.2535           3.3625        0.025
    08-Mar-2013 02:34:21     0.29507      3.0075          1.254           3.5428        0.023
    09-Mar-2013 02:33:43    0.26962     3.0125        1.254          3.6541        0.023
    10-Mar-2013 03:01:02    0.25565     3.0197       1.2544          3.7722        0.025
    11-Mar-2013 03:00:24    0.24756     3.0247       1.2546          3.7793        0.027
    12-Mar-2013 06:17:10    0.25519     3.0236       1.2544          3.7479        0.027
    13-Mar-2013 06:34:04      0.233     3.0272       1.2545          3.8282        0.027
    14-Mar-2013 06:50:41    0.23299     3.0249       1.2544          3.9047         0.02
    15-Mar-2013 06:50:03     0.2315      3.033       1.2548          3.9706        0.032
    16-Mar-2013 06:56:43    0.23475     3.0273       1.2546          3.9451        0.031
    17-Mar-2013 06:56:04    0.23498     3.0407       1.2551          3.9924        0.032
    17-Mar-2013 18:47:56    0.21839     3.0533       1.2557          3.9792        0.037
    18-Mar-2013 18:47:15    0.21943     3.0778       1.2567          4.0538        0.043
    20-Mar-2013 00:33:54    0.23854     3.0905       1.2573          3.9658        0.047
    21-Mar-2013 00:33:14    0.23537     3.1044       1.2578          3.9862        0.051
    22-Mar-2013 00:39:50    0.23079     3.1481       1.2593           4.072        0.058
      ⋮
```

Dimension Reduction and Feature Fusion

Principal Component Analysis (PCA) is used for dimension reduction and feature fusion in this example. Before performing PCA, it is a good practice to normalize the features into the same scale. Note that PCA coefficients and the mean and standard deviation used in normalization are obtained from training data, and applied to the entire dataset.

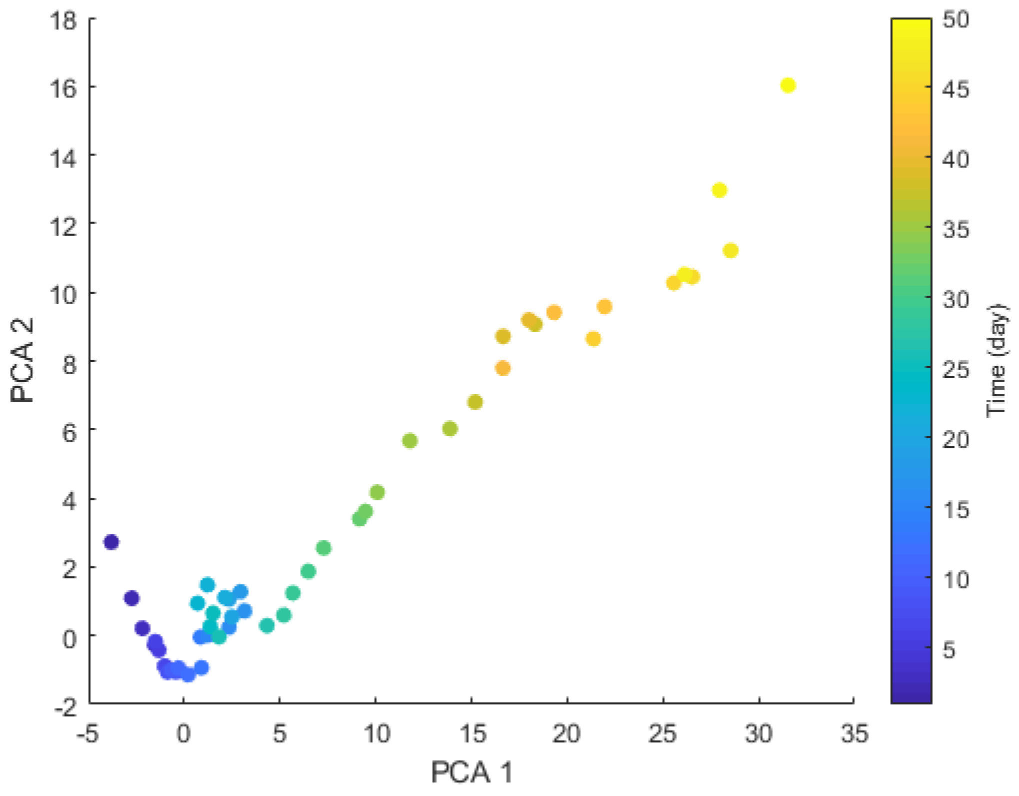
```
meanTrain = mean(trainDataSelected{:,:});
sdTrain = std(trainDataSelected{:,:});
trainDataNormalized = (trainDataSelected{:,:} - meanTrain)./sdTrain;
coef = pca(trainDataNormalized);
```

The mean, standard deviation and PCA coefficients are used to process the entire data set.

```
PCA1 = (featureSelected{:,:} - meanTrain) ./ sdTrain * coef(:, 1);
PCA2 = (featureSelected{:,:} - meanTrain) ./ sdTrain * coef(:, 2);
```

Visualize the data in the space of the first two principal components.

```
figure
numData = size(featureTable, 1);
scatter(PCA1, PCA2, [], 1:numData, 'filled')
xlabel('PCA 1')
ylabel('PCA 2')
cbar = colorbar;
ylabel(cbar, ['Time (' timeUnit ')'])
```

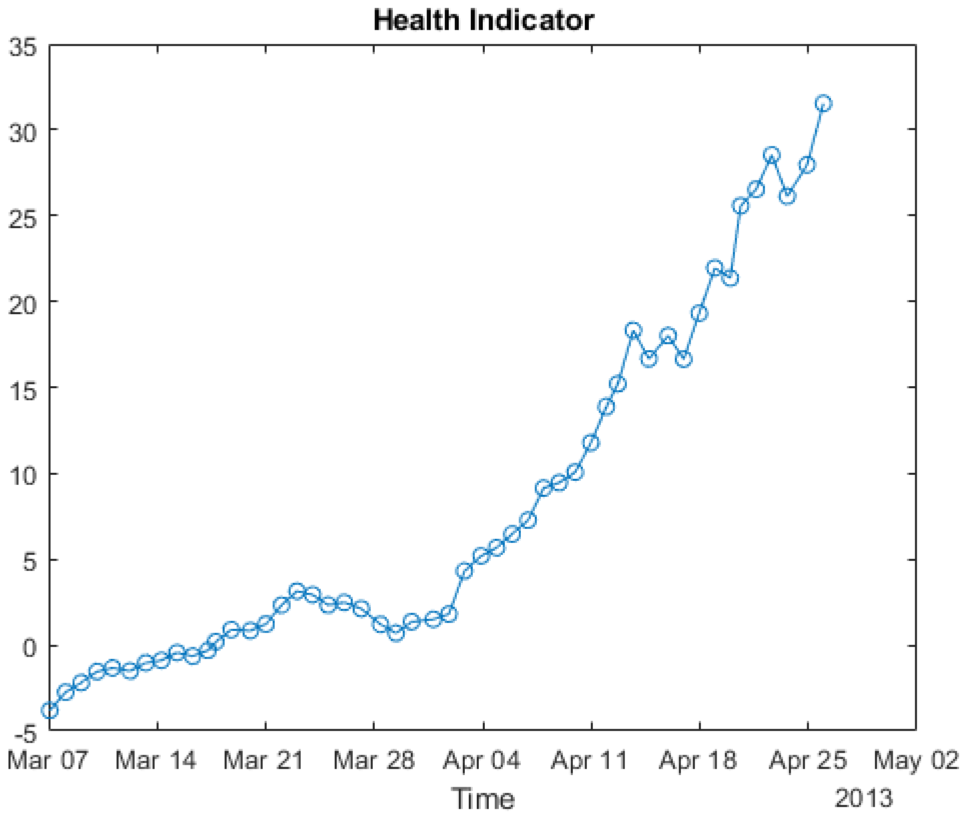


The plot indicates that the first principal component is increasing as the machine approaches to failure. Therefore, the first principal component is a promising fused health indicator.

```
healthIndicator = PCA1;
```

Visualize the health indicator.

```
figure
plot(featureSelected.Date, healthIndicator, '-o')
xlabel('Time')
title('Health Indicator')
```



Fit Exponential Degradation Models for Remaining Useful Life (RUL) Estimation

Exponential degradation model is defined as

$$h(t) = \phi + \theta \exp\left(\beta t + \epsilon - \frac{\sigma^2}{2}\right)$$

where $h(t)$ is the health indicator as a function of time. ϕ is the intercept term considered as a constant. θ and β are random parameters determining the slope of the model, where θ is lognormal-distributed and β is Gaussian-distributed. At each time step t, the distribution of θ and β is updated to the posterior based on the latest observation of $h(t)$. ϵ is a Gaussian white noise yielding to $N(0, \sigma^2)$. The $-\frac{\sigma^2}{2}$ term in the exponential is to make the expectation of $h(t)$ satisfy

$E[h(t)|\theta, \beta] = \phi + \theta \exp(\beta t)$.

Here an Exponential Degradation Model is fit to the health indicator extracted in the last section, and the performances is evaluated in the next section.

First shift the health indicator so that it starts from 0.

```
healthIndicator = healthIndicator - healthIndicator(1);
```

The selection of threshold is usually based on the historical records of the machine or some domain-specific knowledge. Since no historical data is available in this dataset, the last value of the health indicator is chosen as the threshold. It is recommended to choose the threshold based on the smoothed (historical) data so that the delay effect of smoothing will be partially mitigated.

```
threshold = healthIndicator(end);
```

If historical data is available, use `fit` method provided by `exponentialDegradationModel` to estimate the priors and intercept. However, historical data is not available for this wind turbine bearing dataset. The prior of the slope parameters are chosen arbitrarily with large variances ($E(\theta) = 1,\ \mathrm{Var}(\theta) = 10^6, E(\beta) = 1, \mathrm{Var}(\beta) = 10^6$) so that the model is mostly relying on the observed data. Based on $E[h(0)] = \phi + E(\theta)$, intercept ϕ is set to -1 so that the model will start from 0 as well.

The relationship between the variation of health indicator and the variation of noise can be derived as

$$\Delta h(t) \approx (h(t) - \phi)\Delta\epsilon(t)$$

Here the standard deviation of the noise is assumed to cause 10% of variation of the health indicator when it is near the threshold. Therefore, the standard deviation of the noise can be represented as $\frac{10\% \cdot \text{threshold}}{\text{threshold} - \phi}$.

The exponential degradation model also provides a functionality to evaluate the significance of the slope. Once a significant slope of the health indicator is detected, the model will forget the previous observations and restart the estimation based on the original priors. The sensitivity of the detection algorithm can be tuned by specifying `SlopeDetectionLevel`. If p value is less than `SlopeDetectionLevel`, the slope is declared to be detected. Here `SlopeDetectionLevel` is set to 0.05.

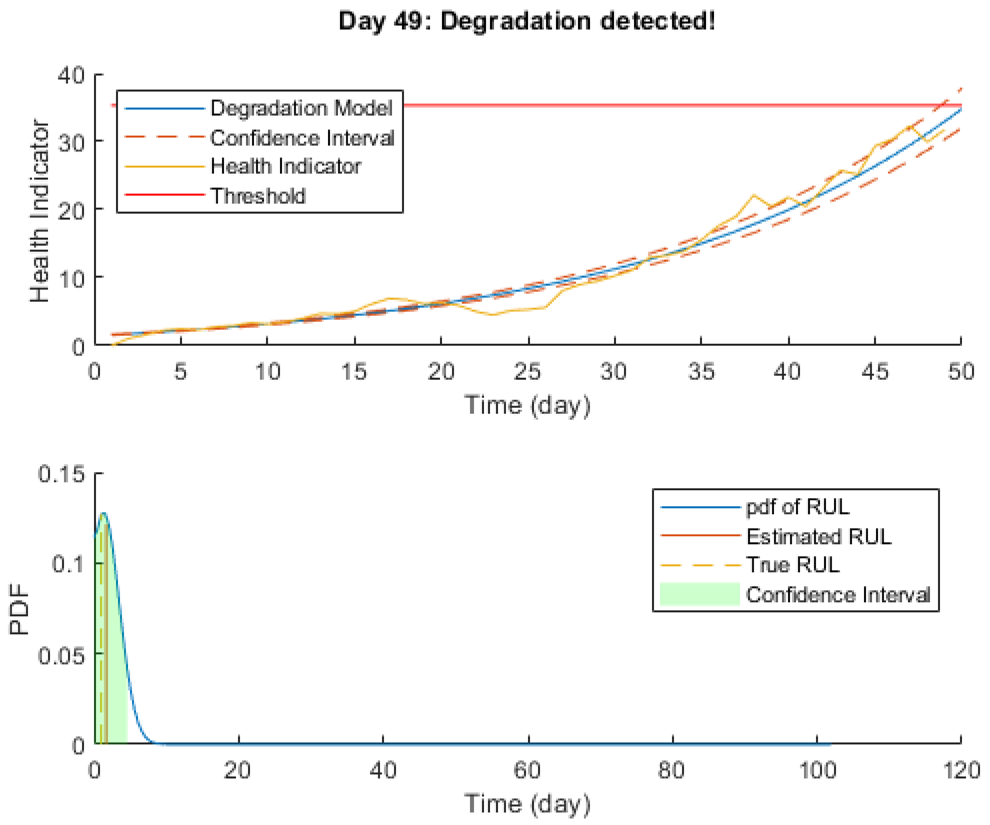
Now create an exponential degradation model with the parameters discussed above.

```
mdl = exponentialDegradationModel(...
    'Theta', 1, ...
    'ThetaVariance', 1e6, ...
    'Beta', 1, ...
    'BetaVariance', 1e6, ...
    'Phi', -1, ...
    'NoiseVariance', (0.1*threshold/(threshold + 1))^2, ...
    'SlopeDetectionLevel', 0.05);
```

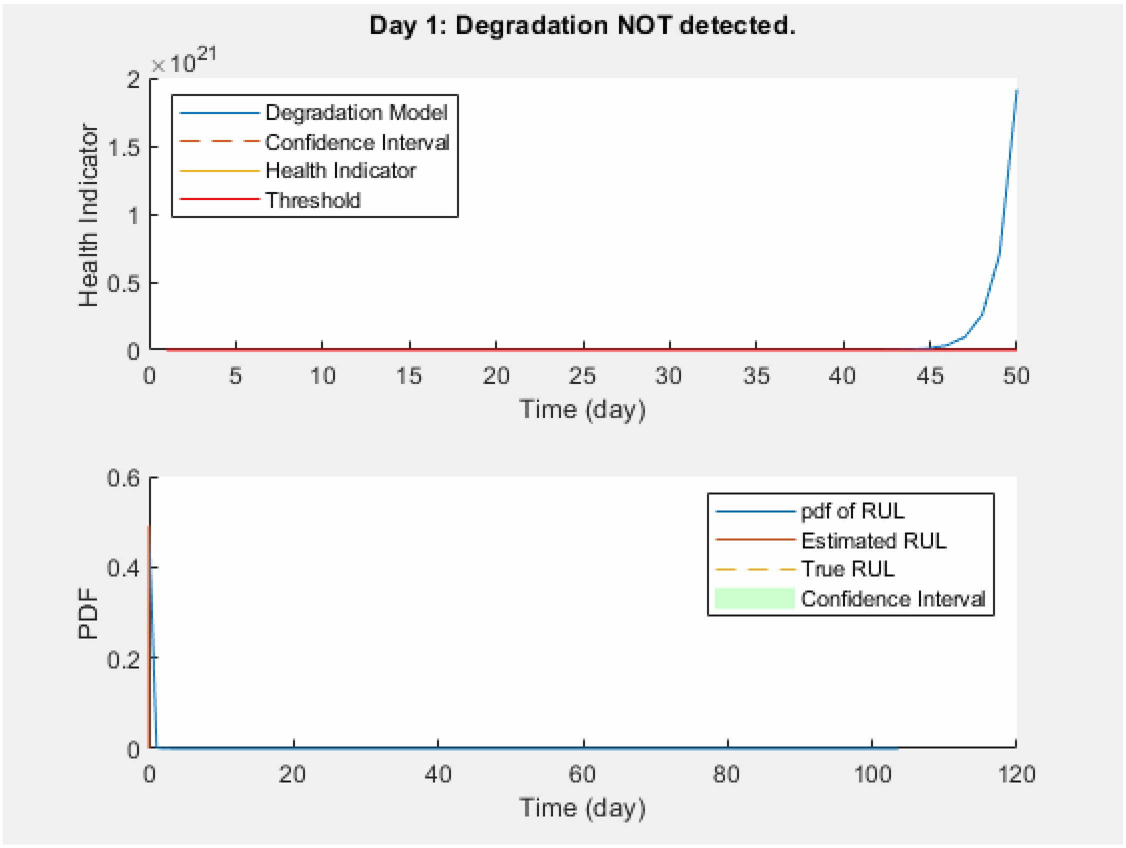
Use `predictRUL` and `update` methods to predict the RUL and update the parameter distribution in real time.

```
% Keep records at each iteration
totalDay = length(healthIndicator) - 1;
estRULs = zeros(totalDay, 1);
trueRULs = zeros(totalDay, 1);
CIRULs = zeros(totalDay, 2);
pdfRULs = cell(totalDay, 1);

% Create figures and axes for plot updating
figure
ax1 = subplot(2, 1, 1);
ax2 = subplot(2, 1, 2);

for currentDay = 1:totalDay

    % Update model parameter posterior distribution
    update(mdl, [currentDay healthIndicator(currentDay)])

    % Predict Remaining Useful Life
    [estRUL, CIRUL, pdfRUL] = predictRUL(mdl, ...
                                         [currentDay healthIndicator(currentDay)], ...
                                         threshold);
    trueRUL = totalDay - currentDay + 1;
```

```
        % Updating RUL distribution plot
        helperPlotTrend(ax1, currentDay, healthIndicator, mdl, threshold, timeUnit);
        helperPlotRUL(ax2, trueRUL, estRUL, CIRUL, pdfRUL, timeUnit)

        % Keep prediction results
        estRULs(currentDay) = estRUL;
        trueRULs(currentDay) = trueRUL;
        CIRULs(currentDay, :) = CIRUL;
        pdfRULs{currentDay} = pdfRUL;

        % Pause 0.1 seconds to make the animation visible
        pause(0.1)
    end
```



Here is the animation of the real-time RUL estimation.



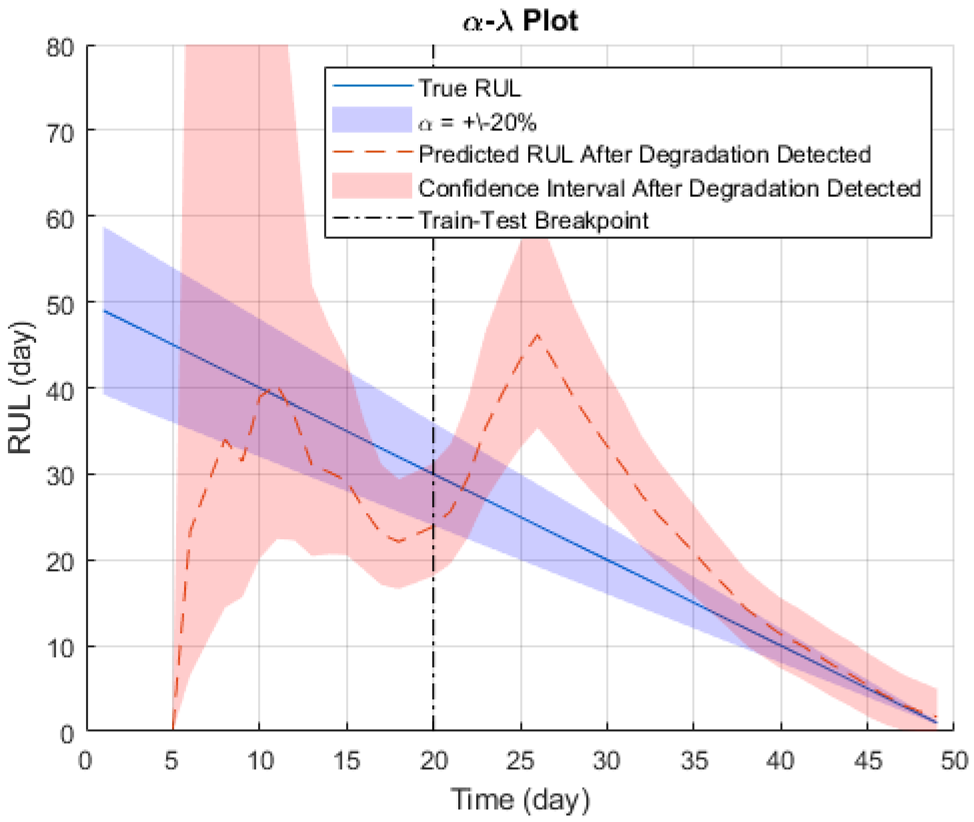
Performance Analysis

α-λ plot is used for prognostic performance analysis [5], where α bound is set to 20%. The probability that the estimated RUL is between the α bound of the true RUL is calculated as a performance metric of the model:

$$\Pr(r^*(t) - \alpha r^*(t) < r(t) < r^*(t) + \alpha r^*(t) \,|\, \Theta(t))$$

where $r(t)$ is the estimated RUL at time t, $r^*(t)$ is the true RUL at time t, $\Theta(t)$ is the estimated model parameters at time t.

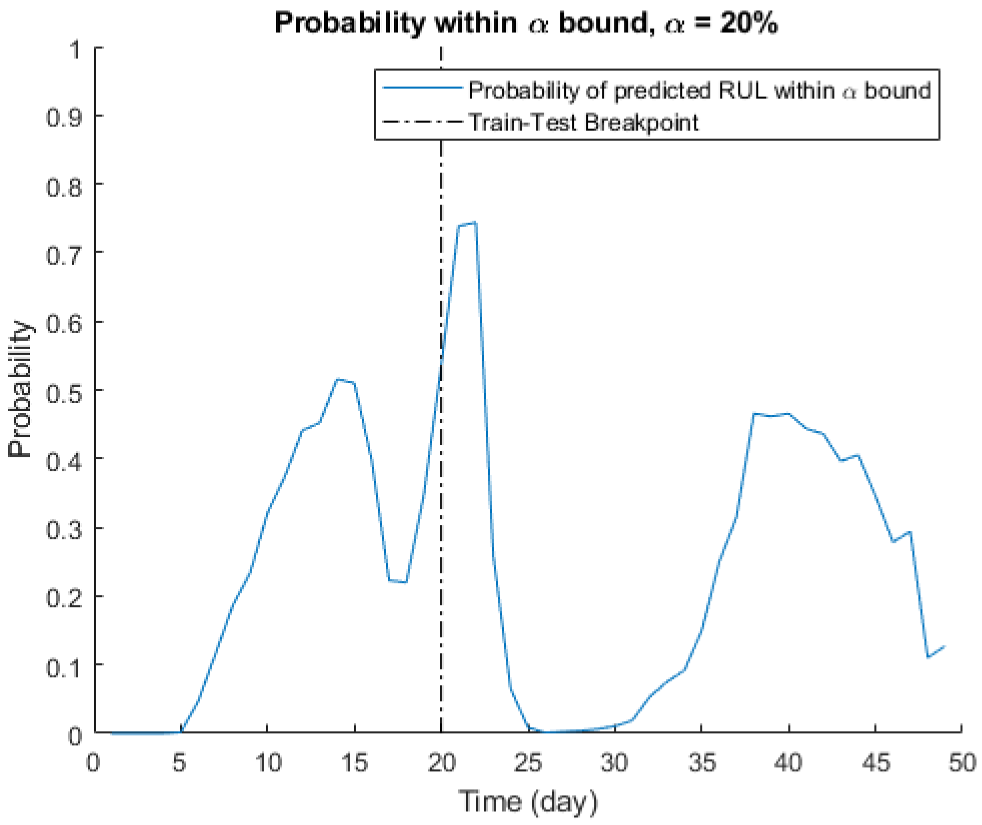
```
alpha = 0.2;
detectTime = mdl.SlopeDetectionInstant;
prob = helperAlphaLambdaPlot(alpha, trueRULs, estRULs, CIRULs, ...
    pdfRULs, detectTime, breakpoint, timeUnit);
title('\alpha-\lambda Plot')
```



Since the preset prior does not reflect the true prior, the model usually need a few time steps to adjust to a proper parameter distribution. The prediction becomes more accurate as more data points are available.

Visualize the probability of the predicted RUL within the α bound.

```
figure
t = 1:totalDay;
hold on
plot(t, prob)
plot([breakpoint breakpoint], [0 1], 'k-.')
hold off
xlabel(['Time (' timeUnit ')'])
ylabel('Probability')
legend('Probability of predicted RUL within \alpha bound', 'Train-Test Breakpoint')
title(['Probability within \alpha bound, \alpha = ' num2str(alpha*100) '%'])
```



References

[1] <http://data-acoustics.com/measurements/bearing-faults/bearing-3/> Bechhoefer, Eric, Brandon Van Hecke, and David He. "Processing for improved spectral analysis." *Annual Conference of the Prognostics and Health Management Society, New Orleans, LA, Oct*. 2013.

[2] Ali, Jaouher Ben, et al. "Online automatic diagnosis of wind turbine bearings progressive degradations under real experimental conditions based on unsupervised machine learning." *Applied Acoustics* 132 (2018): 167-181.

[3] Saidi, Lotfi, et al. "Wind turbine high-speed shaft bearings health prognosis through a spectral Kurtosis-derived indices and SVR." *Applied Acoustics* 120 (2017): 1-8.

[4] Coble, Jamie Baalis. "Merging data sources to predict remaining useful life–an automated method to identify prognostic parameters." (2010).

[5] Saxena, Abhinav, et al. "Metrics for offline evaluation of prognostic performance." *International Journal of Prognostics and Health Management* 1.1 (2010): 4-23.

See Also

`exponentialDegradationModel`

More About

- “Feature Selection for Remaining Useful Life Prediction” on page 5-2
- “RUL Estimation Using RUL Estimator Models” on page 5-8
- “Similarity-Based Remaining Useful Life Estimation” on page 5-18

Condition Monitoring and Prognostics Using Vibration Signals

This example shows how to extract features from vibration signals from a ball bearing, conduct health monitoring, and perform prognostics. This example uses functionality from Signal Processing Toolbox™ and System Identification Toolbox™, and does not require Predictive Maintenance Toolbox™.

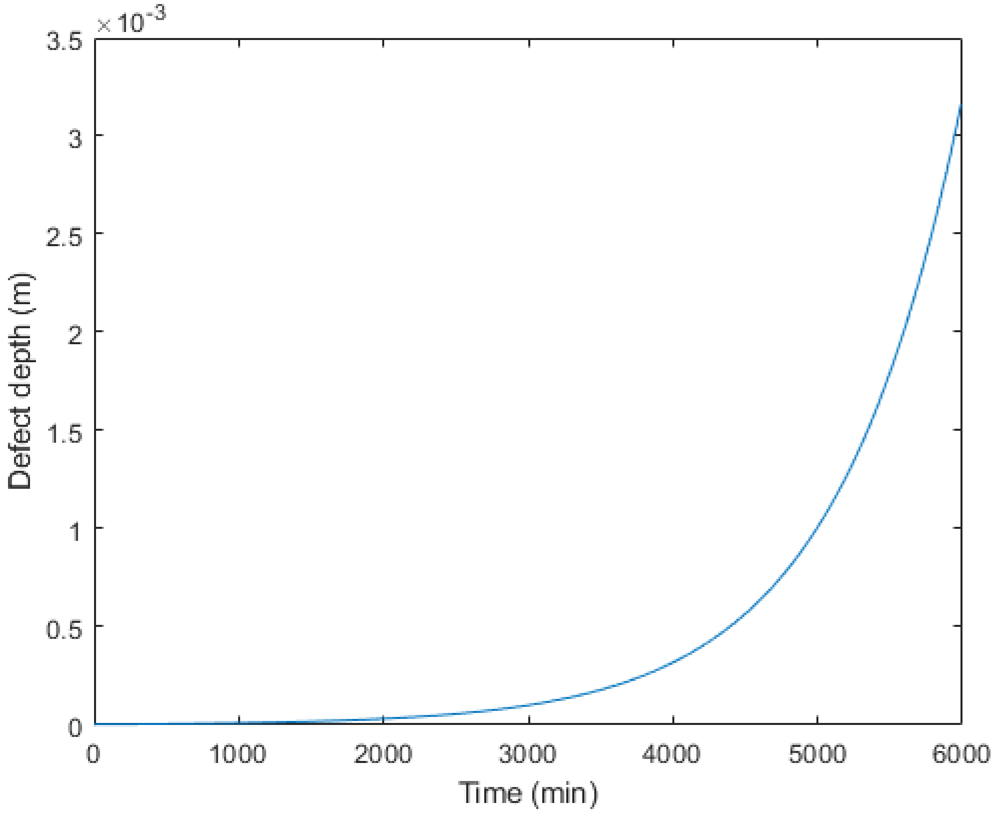
Data Description

Load vibration data stored in `pdmBearingConditionMonitoringData.mat`. The data is stored in a cell array, which was generated using a ball bearing signal simulator with single point defect on the outer race of the bearing. It contains multiple segments of vibration signals for bearings simulated in different health conditions (defect depth increases from 3um to above 3mm). Each segment stores signals collected for 1 second at a sampling rate of 20 kHz. In pdmBearingConditionMonitoring`Data.mat`, the `defectDepthVec` stores how defect depth changes versus time while the `expTime` stores the corresponding time in minutes.

```
load pdmBearingConditionMonitoringData.mat

% Define the number of data points to be processed.
numSamples = length(data);

% Define sampling frequency.
fs = 20E3;          % unit: Hz
```

Plot how defect depth changes in different segments of data.

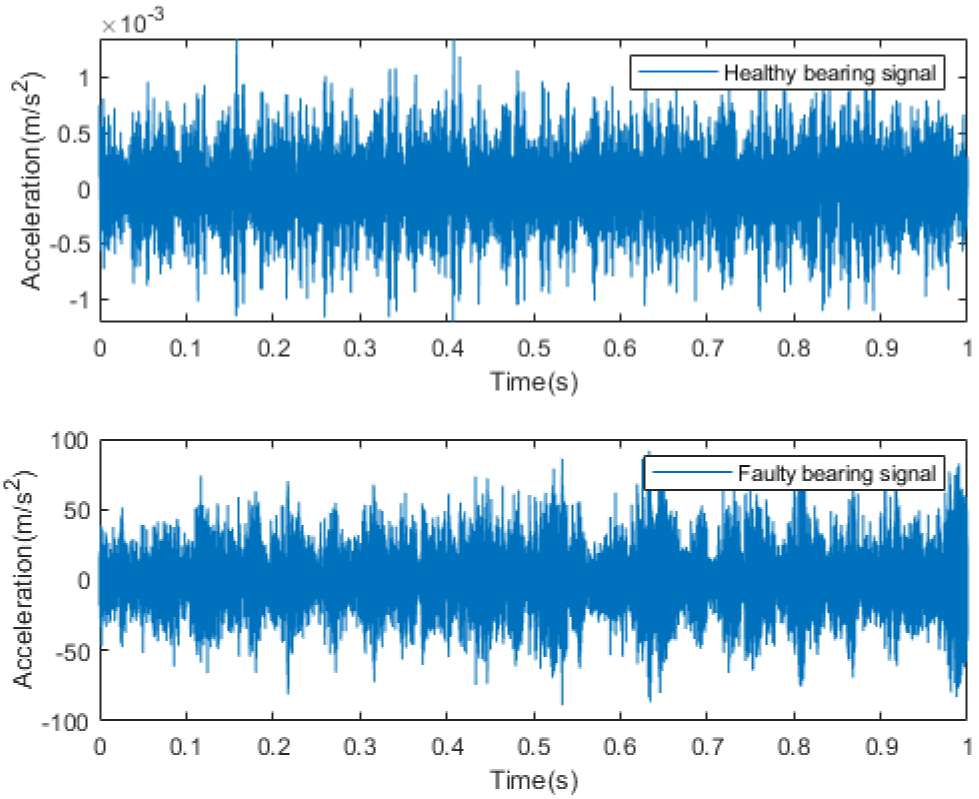
```
plot(expTime,defectDepthVec);
xlabel('Time (min)');
ylabel('Defect depth (m)');
```



Plot the healthy and faulty data.

```
time = linspace(0,1,fs)';

% healthy bearing signal
subplot(2,1,1);
plot(time,data{1});
xlabel('Time(s)');
ylabel('Acceleration(m/s^2)');
legend('Healthy bearing signal');

% faulty bearing signal
subplot(2,1,2);
plot(time,data{end});
```

```
xlabel('Time(s)');
ylabel('Acceleration(m/s^2)');
legend('Faulty bearing signal');
```



Feature Extraction

In this section, representative features are extracted from each segment of data. These features will be used for health monitoring and prognostics. Typical features for bearing diagnostics and prognostics include time-domain features (root mean square, peak value, signal kurtosis, etc.) or frequency-domain features (peak frequency, mean frequency, etc.).

Before selecting which features to use, plot the vibration signals spectrogram. Visualizing signals in time-domain or frequency-domain or time-frequency domain can help discover signal patterns that indicate degradation or failure.

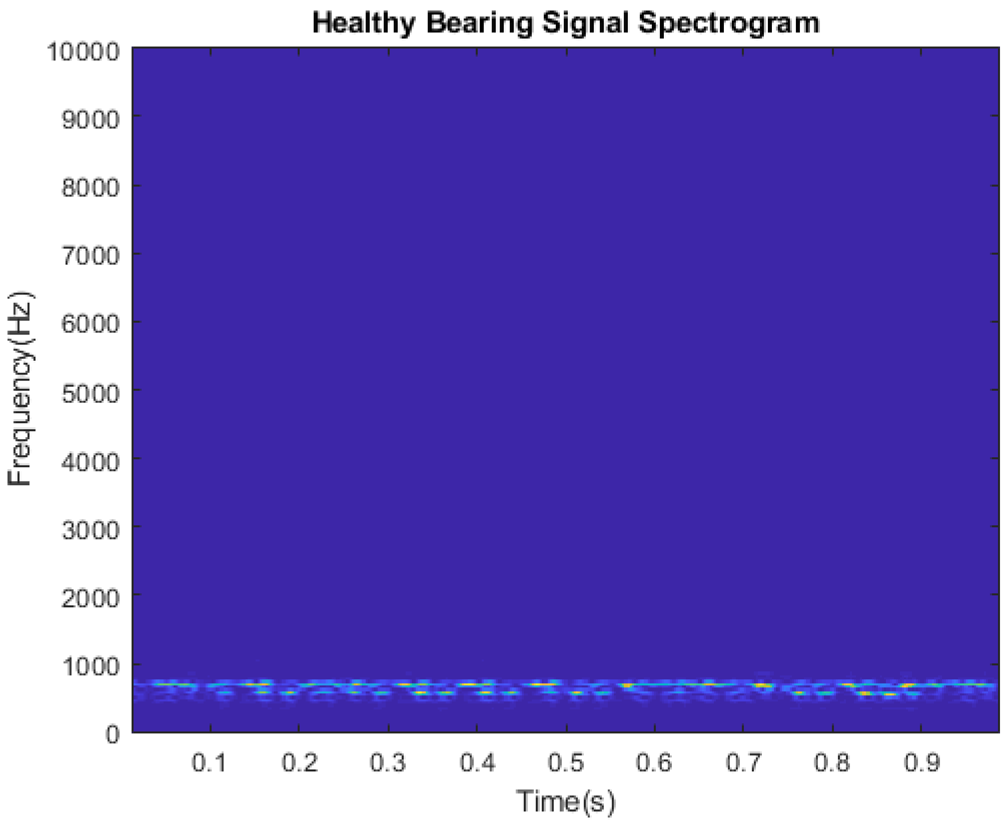
First calculate the spectrogram of the healthy bearing data. Use a window size of 500 data points and an overlap ratio of 90% (equivalent to 450 data points). Set the number of points for the FFT to be 512. `fs` represents the sampling frequency defined previously.

```
[~,fvec,tvec,P0] = spectrogram(data{1},500,450,512,fs);
```

`P0` is the spectrogram, `fvec` is the frequency vector and `tvec` is the time vector.

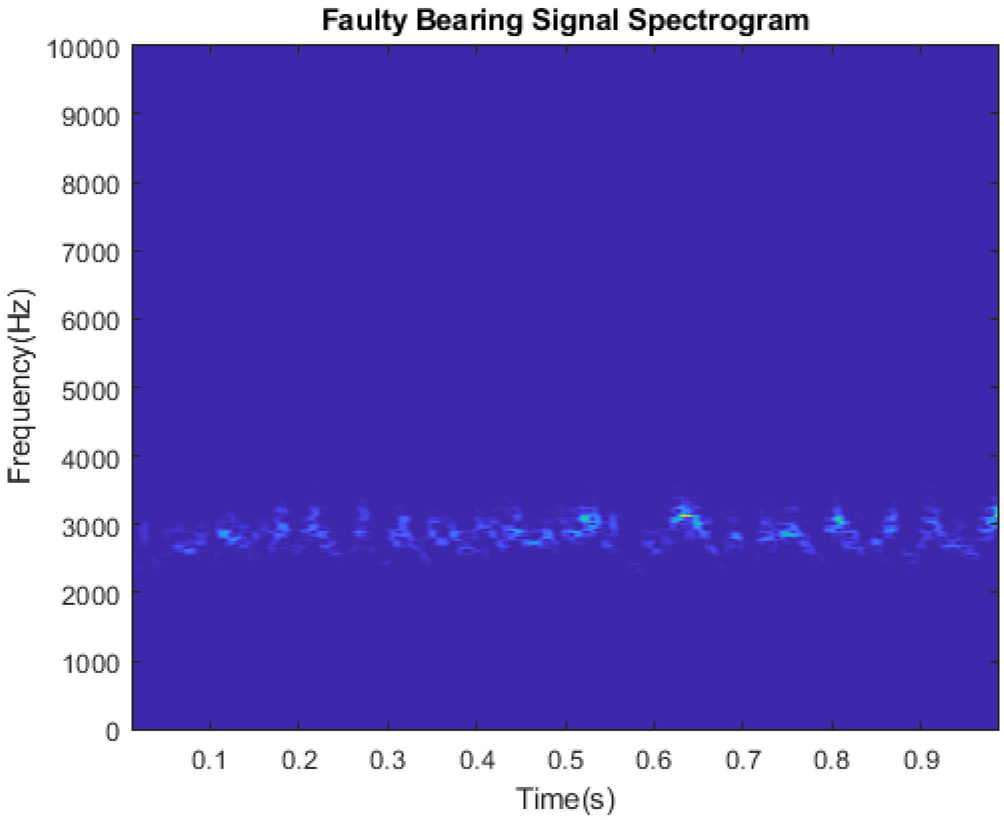
Plot the spectrogram of the healthy bearing signal.

```
clf;
imagesc(tvec,fvec,P0)
xlabel('Time(s)');
ylabel('Frequency(Hz)');
title('Healthy Bearing Signal Spectrogram');
axis xy
```



Now plot the spectrogram of vibration signals that have developed faulty patterns. You can see that signal energies are concentrated at higher frequencies.

```
[~,fvec,tvec,Pfinal] = spectrogram(data{end},500,450,512,fs);
imagesc(tvec,fvec,Pfinal)
xlabel('Time(s)');
ylabel('Frequency(Hz)');
title('Faulty Bearing Signal Spectrogram');
axis xy
```



Since the spectrograms for data from healthy and faulty bearings are different, representative features can be extracted from spectrograms and used for condition monitoring and prognostics. In this example, extract mean peak frequencies from spectrograms as health indicators. Denote the spectrogram as $P(t, \omega)$. Peak frequency at each time instance is defined as:

$$PeakFreq(t) = argmax_{\omega} P(t, \omega)$$

The mean peak frequency is the average of peak frequencies defined above.

$$meanPeakFreq = \frac{1}{T}\int_0^T PeakFreq(t)dt$$

Calculate the mean peak frequency for healthy ball bearing signals.

```
[~,I0] = max(P0);                  % Find out where the peak frequencies are located.
meanPeakFreq0 = mean(fvec(I0))  % Calculate mean peak frequency.
```

```
meanPeakFreq0 = 666.4602
```

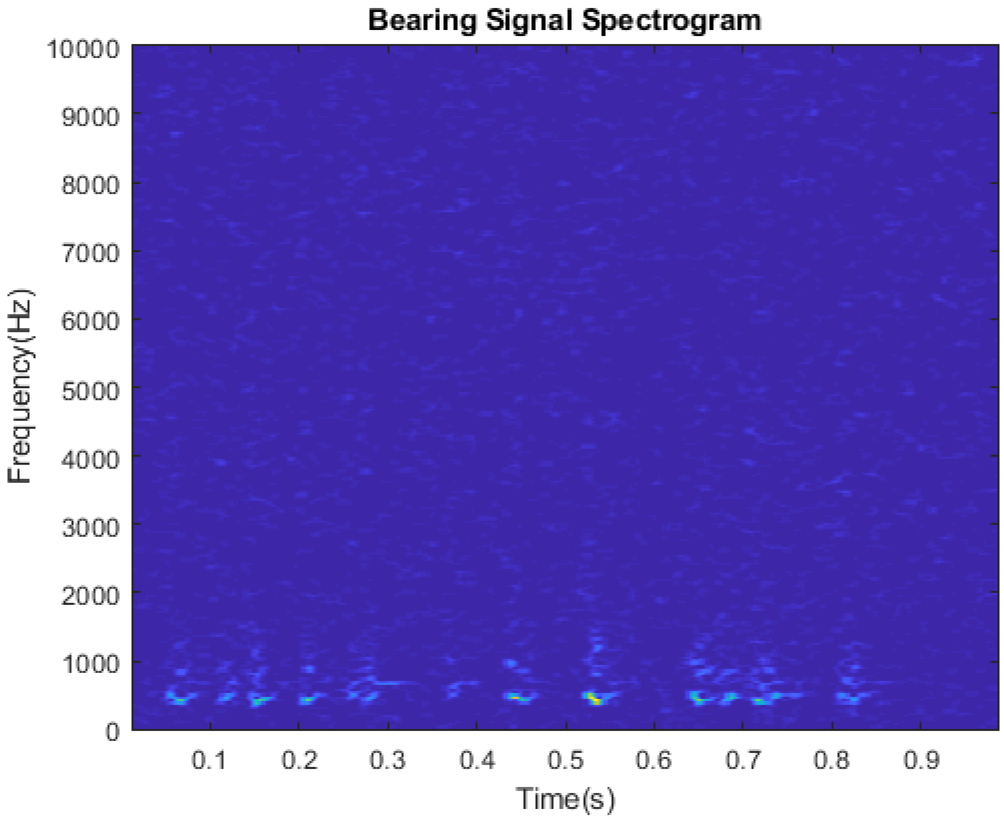
The healthy bearing vibration signals have mean peak frequency at around 650 Hz. Now calculate the mean peak frequency for faulty bearing signals. The mean peak frequency shifts to above 2500 Hz.

```
[~,Ifinal] = max(Pfinal);
meanPeakFreqFinal = mean(fvec(Ifinal))
```

```
meanPeakFreqFinal = 2.8068e+03
```

Examine the data at middle stage, when the defect depth is not very large but starting to affect the vibration signals.

```
[~,fvec,tvec,Pmiddle] = spectrogram(data{end/2},500,450,512,fs);
imagesc(tvec,fvec,Pmiddle)
xlabel('Time(s)');
ylabel('Frequency(Hz)');
title('Bearing Signal Spectrogram');
axis xy
```



The high frequency noise components are spread all over the spectrogram. Such phenomena are mixed effects of both original vibrations and the vibrations induced by small defects. To accurately calculate mean peak frequency, filter the data to remove those high frequency components.

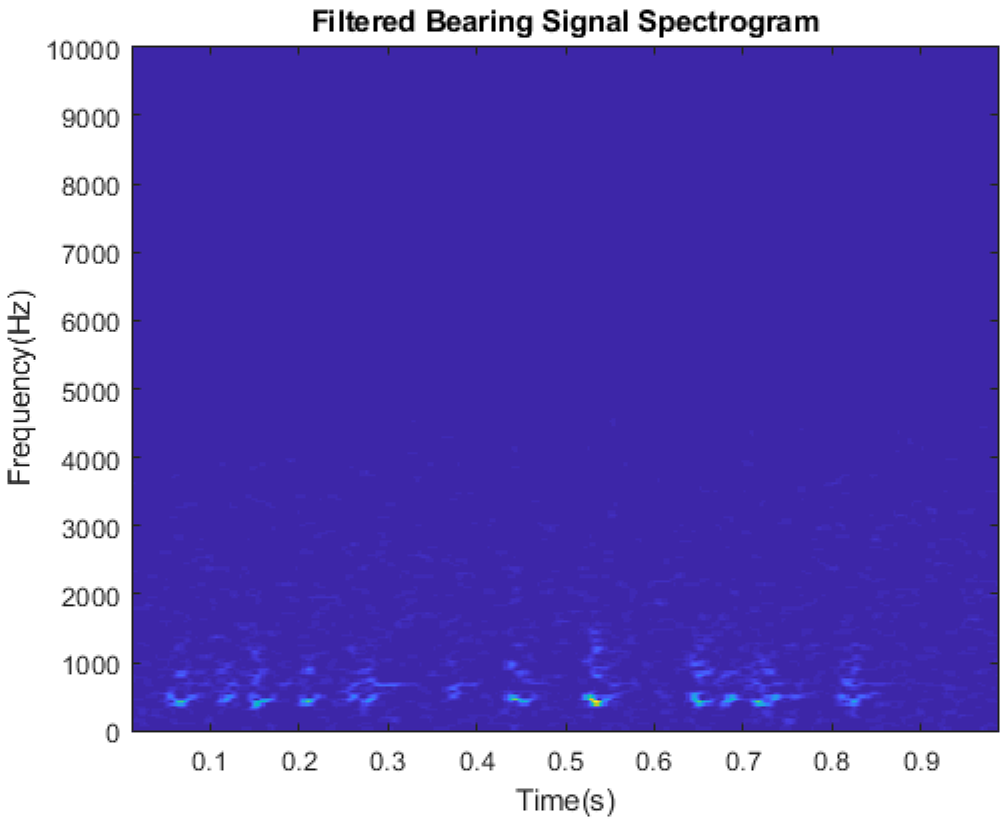
Apply a median filter to the vibration signals to remove high frequency noise components as well as to preserve useful information in the high frequencies.

```
dataMiddleFilt = medfilt1(data{end/2},3);
```

Plot spectrogram after median filtering. The high frequency components are suppressed.

```
[~,fvec,tvec,Pmiddle] = spectrogram(dataMiddleFilt,500,450,512,fs);
imagesc(tvec,fvec,Pmiddle)
```

```
xlabel('Time(s)');
ylabel('Frequency(Hz)');
title('Filtered Bearing Signal Spectrogram');
axis xy
```

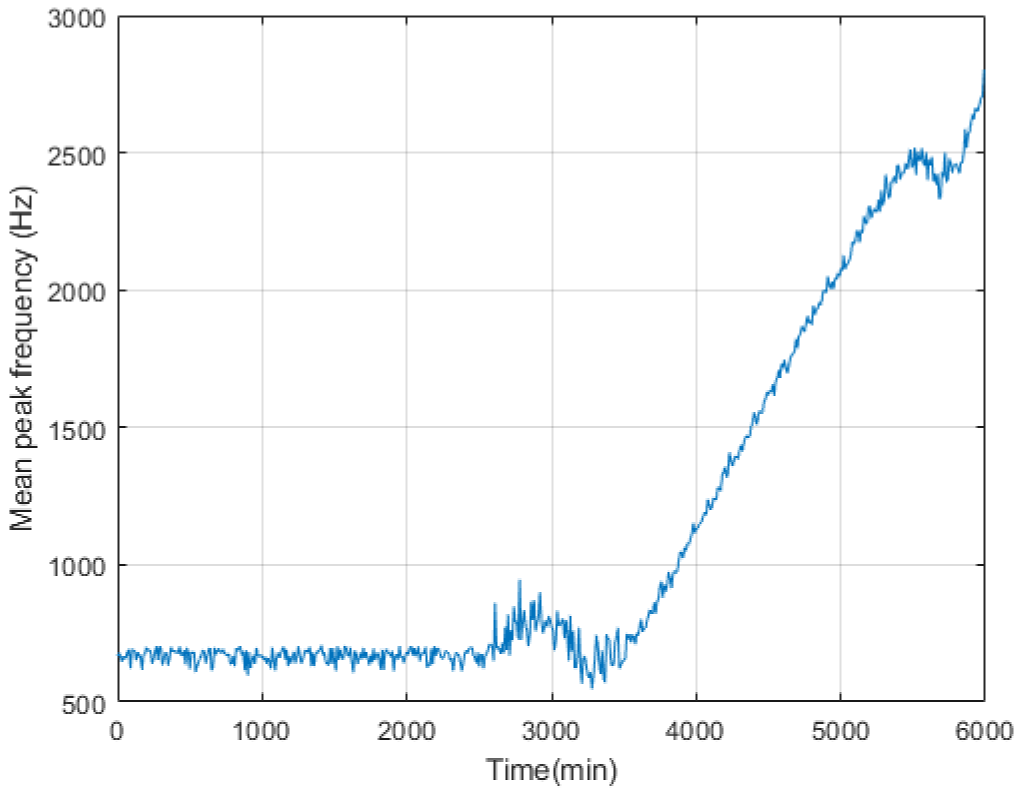


Since the mean peak frequency successfully distinguishes healthy ball bearings from faulty ball bearings, extract mean peak frequency from each segment of data.

```
% Define a progress bar.
h = waitbar(0,'Start to extract features');
% Initialize a vector to store the extracted mean peak frequencies.
meanPeakFreq = zeros(numSamples,1);
for k = 1:numSamples
    % Get most up-to-date data.
    curData = data{k};
    % Apply median filter.
    curDataFilt = medfilt1(curData,3);
    % Calculate spectrogram.
    [~,fvec,tvec,P_k] = spectrogram(curDataFilt,500,450,512,fs);
    % Calculate peak frequency at each time instance.
    [~,I] = max(P_k);
    meanPeakFreq(k) = mean(fvec(I));
    % Show progress bar indicating how many samples have been processed.
    waitbar(k/numSamples,h,'Extracting features');
end
close(h);
```

Plot the extracted mean peak frequencies versus time.

```
plot(expTime,meanPeakFreq);
xlabel('Time(min)');
ylabel('Mean peak frequency (Hz)');
grid on;
```



Condition Monitoring and Prognostics

In this section, condition monitoring and prognostics are performed using a pre-defined threshold and dynamic models. For condition monitoring, create an alarm that triggers if the mean peak frequency exceeds the predefined threshold. For prognostics, identify a dynamic model to forecast the values of mean peak frequencies in the next few hours. Create an alarm that triggers if the forecast mean peak frequency exceeds the predefined threshold.

Forecasting helps us better prepare for a potential fault or even stop the machine before failure. Consider the mean peak frequency as a time series. We can estimate a time series model for the mean peak frequency and use the model to forecast the future values. Use the first 200 mean peak frequency values to create an initial time series model, then once

10 new values are available, use the last 100 values to update the time series model. This batch mode of updating the time series model captures instantaneous trends. The updated time series model is used to compute a 10 step ahead forecast.

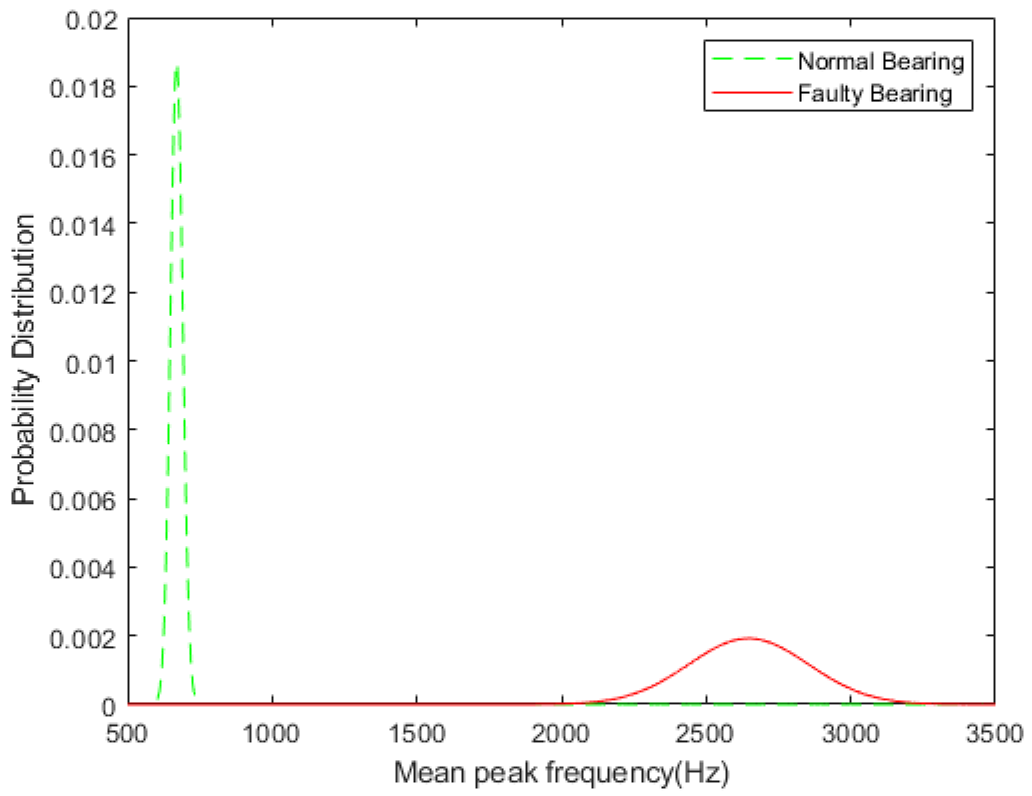
```
tStart = 200;                % Start Time
timeSeg = 100;               % Length of data for building dynamic model
forecastLen = 10;            % Define forecast time horizon
batchSize = 10;              % Define batch size for updating the dynamic model
```

For prognostics and condition monitoring, you need to set a threshold to decide when to stop the machine. In this example, use the statistical data of healthy and faulty bearing generated from simulation to determine the threshold. The `pdmBearingConditionMonitoringStatistics.mat` stores the probability distributions of mean peak frequencies for healthy and faulty bearings. The probability distributions are computed by perturbing the defect depth of healthy and faulty bearings.

```
load pdmBearingConditionMonitoringStatistics.mat
```

Plot the probability distributions of mean peak frequency for healthy and faulty bearings.

```
plot(pFreq,pNormal,'g--',pFreq,pFaulty,'r');
xlabel('Mean peak frequency(Hz)');
ylabel('Probability Distribution');
legend('Normal Bearing','Faulty Bearing');
```



Based on this plot, set the threshold for the mean peak frequency to 2000Hz to distinguish normal bearings from faulty bearings as well as maximize the use of the bearings.

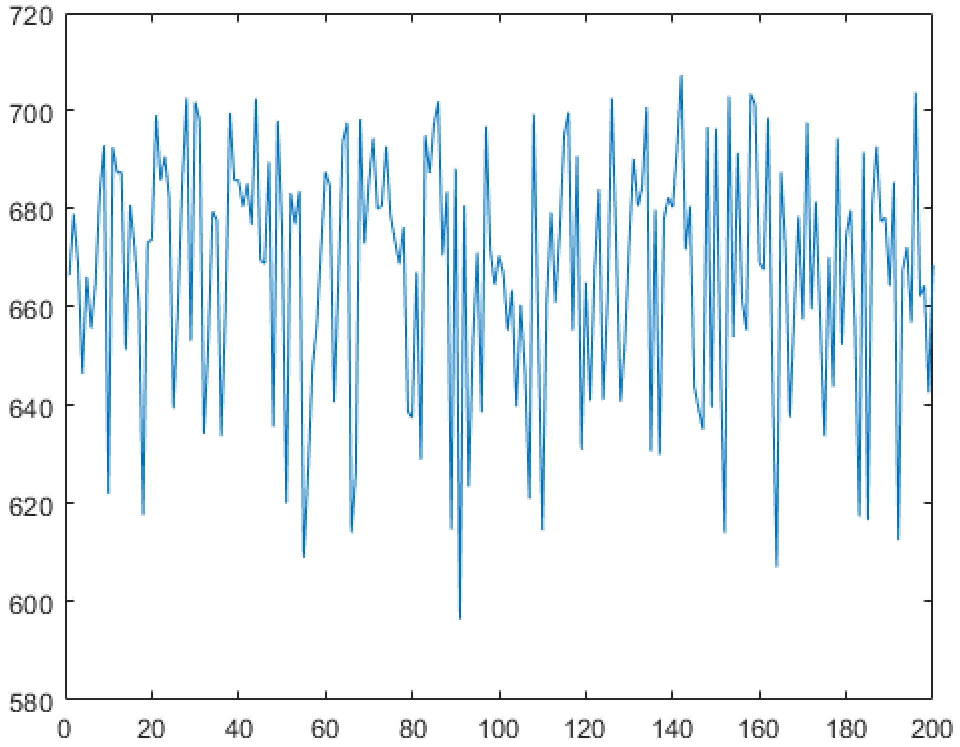
```
threshold = 2000;
```

Calculate sampling time and convert its unit to seconds.

```
samplingTime = 60*(expTime(2)-expTime(1));                  % unit: seconds
tsFeature = iddata(meanPeakFreq(1:tStart),[],samplingTime);
```

Plot the initial 200 mean peak frequency data.

```
plot(tsFeature.y)
```



The plot shows that the initial data is a combination of constant level and noise. This is expected as initially the bearing is healthy and the mean peak frequency is not expected to change significantly

Identify a second-order state-space model using the first 200 data points. Obtain the model in canonical form and specify the sampling time.

```
past_sys = ssest(tsFeature,2,'Ts',samplingTime,'Form','canonical')
```

```
past_sys =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + K e(t)
       y(t) = C x(t) + e(t)
  A = 
             x1       x2
   x1         0        1
   x2    0.9605  0.03942
 
  C = 
        x1  x2
   y1    1   0
 
  K = 
               y1
   x1  -0.003899
   x2   0.003656
 
Sample time: 600 seconds
  
Parameterization:
   CANONICAL form with indices: 2.
   Disturbance component: estimate
   Number of free coefficients: 4
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:                                                   
Estimated using SSEST on time domain data "tsFeature".
Fit to estimation data: 0.2763% (prediction focus)
FPE: 640, MSE: 602.7
```

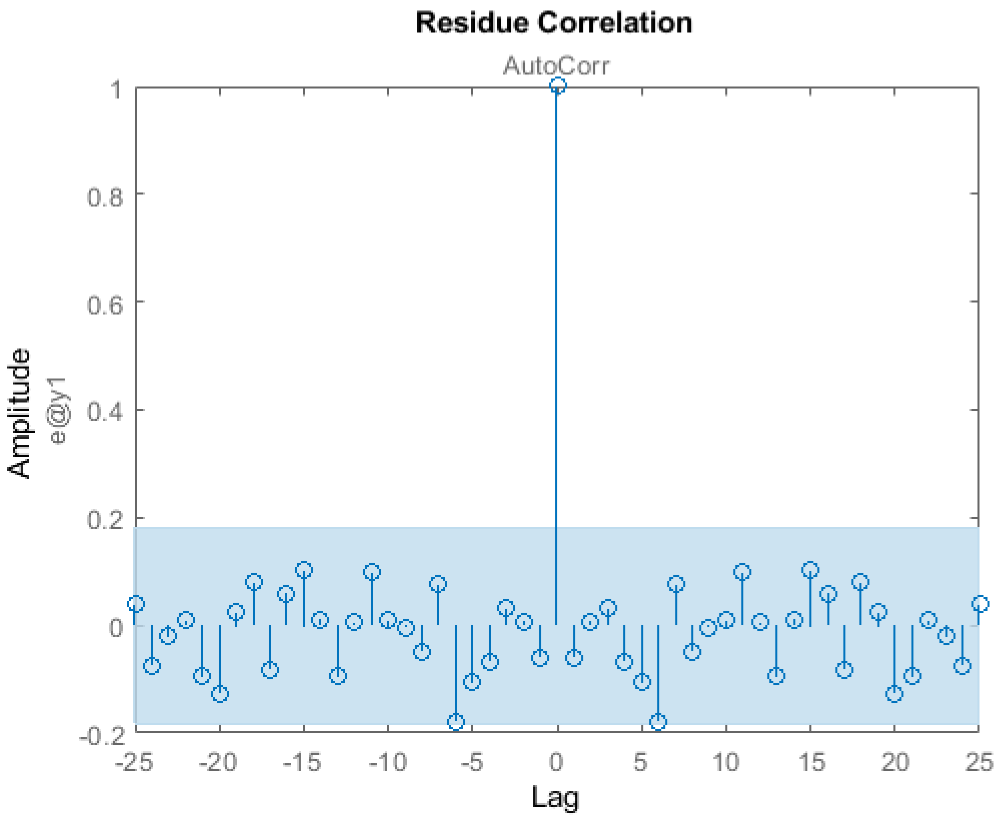
The initial estimated dynamic model has a low goodness of fit. The goodness of fit metric is the normalized root mean square error (NRMSE), calculated as

$$NRMSE = 1 - \frac{\|x_{true} - x_{pred}\|}{\|x_{true} - mean(x_{true})\|}$$

where x_{true} is the true value, x_{pred} is the predicted value.

When the estimation data is a combination of constant level and noise, $x_{pred} \approx mean(x_{true})$, giving a NRMSE close to 0. To validate the model, plot the autocorrelation of residuals.

```
resid(tsFeature,past_sys)
```



As seen, the residuals are uncorrelated and the generated model is valid.

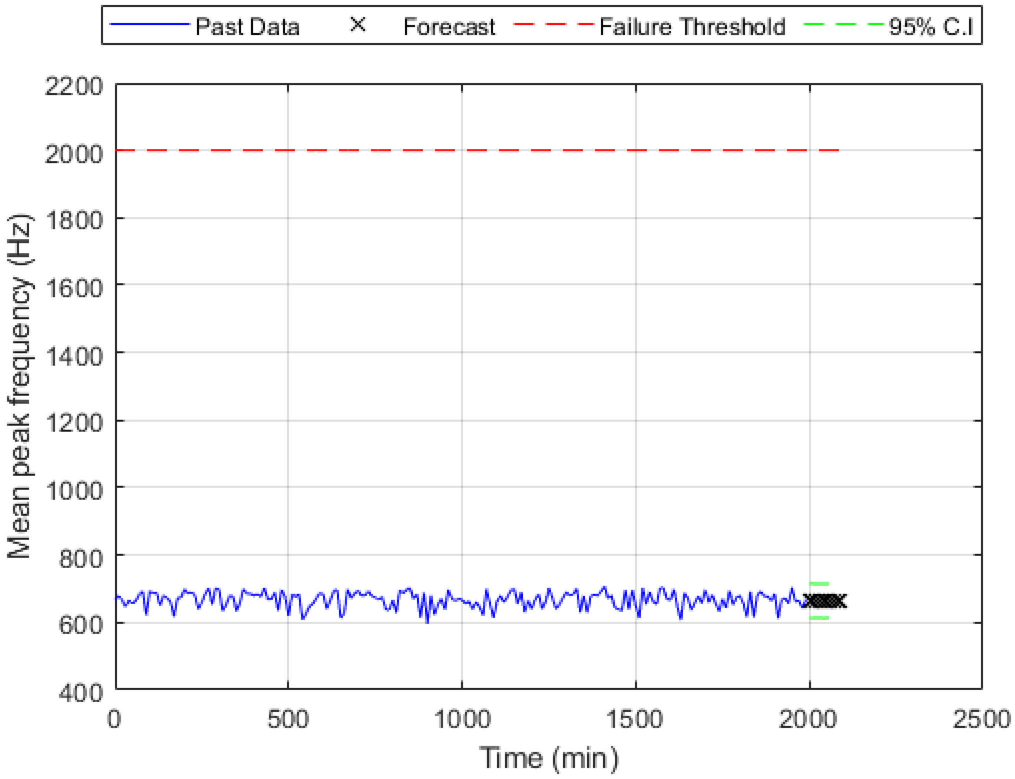
Use the identified model `past_sys` to forecast mean peak frequency values and compute the standard deviation of the forecasted values.

```
[yF,~,~,yFSD] = forecast(past_sys,tsFeature,forecastLen);
```

Plot the forecasted values and confidence intervals.

```
tHistory = expTime(1:tStart);
forecastTimeIdx = (tStart+1):(tStart+forecastLen);
tForecast = expTime(forecastTimeIdx);

% Plot historical data, forecast value and 95% confidence interval.
```

```
plot(tHistory,meanPeakFreq(1:tStart),'b',...
     tForecast,yF.OutputData,'kx',...
     [tHistory; tForecast], threshold*ones(1,length(tHistory)+forecastLen), 'r--',...
     tForecast,yF.OutputData+1.96*yFSD,'g--',...
     tForecast,yF.OutputData-1.96*yFSD,'g--');

ylim([400, 1.1*threshold]);
ylabel('Mean peak frequency (Hz)');
xlabel('Time (min)');
legend({'Past Data', 'Forecast', 'Failure Threshold', '95% C.I'},...
    'Location','northoutside','Orientation','horizontal');
grid on;
```

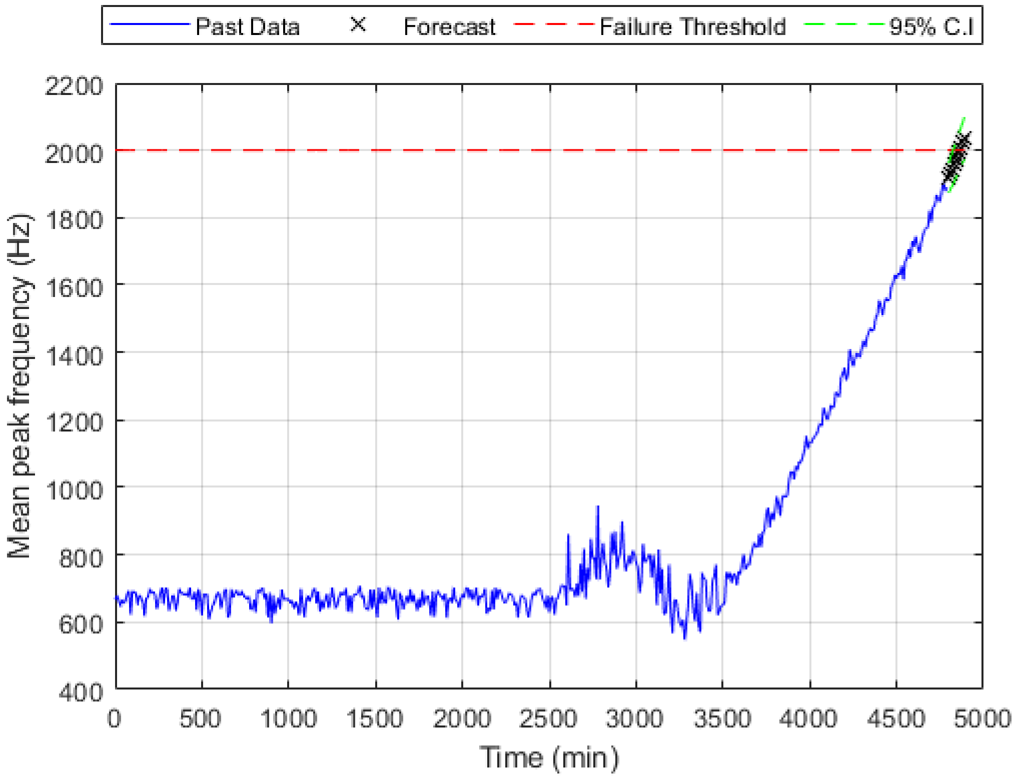


The plot shows that the forecasted values of the mean peak frequency are well below the threshold.

Now update model parameters as new data comes in, and re-estimate the forecasted values. Also create an alarm to check if the signals or the forecasted value exceed the failure threshold.

```
for tCur = tStart:batchSize:numSamples
    %  latest features into iddata object.
    tsFeature = iddata(meanPeakFreq((tCur-timeSeg+1):tCur),[],samplingTime);

    % Update system parameters when new data comes in. Use previous model
    % parameters as initial guesses.
    sys = ssest(tsFeature,past_sys);
    past_sys = sys;

    % Forecast the output of the updated state-space model. Also compute
    % the standard deviation of the forecasted output.
    [yF,~,~,yFSD]  = forecast(sys, tsFeature, forecastLen);

    % Find the time corresponding to historical data and forecasted values.
    tHistory = expTime(1:tCur);
    forecastTimeIdx = (tCur+1):(tCur+forecastLen);
    tForecast = expTime(forecastTimeIdx);

    % Plot historical data, forecasted mean peak frequency value and 95%
    % confidence interval.
    plot(tHistory,meanPeakFreq(1:tCur),'b',...
              tForecast,yF.OutputData,'kx',...
              [tHistory; tForecast], threshold*ones(1,length(tHistory)+forecastLen), '
              tForecast,yF.OutputData+1.96*yFSD,'g--',...
              tForecast,yF.OutputData-1.96*yFSD,'g--');

    ylim([400, 1.1*threshold]);
    ylabel('Mean peak frequency (Hz)');
    xlabel('Time (min)');
    legend({'Past Data', 'Forecast', 'Failure Threshold', '95% C.I'},...
           'Location','northoutside','Orientation','horizontal');
    grid on;

    % Display an alarm when actual monitored variables or forecasted values exceed
    % failure threshold.
    if(any(meanPeakFreq(tCur-batchSize+1:tCur)>threshold))
        disp('Monitored variable exceeds failure threshold');
```

```
            break;
        elseif(any(yF.y>threshold))
            % Estimate the time when the system will reach failure threshold.
            tAlarm = tForecast(find(yF.y>threshold,1));
            disp(['Estimated to hit failure threshold in ' num2str(tAlarm-tHistory(end)) '
            break;
        end
    end
```



```
Estimated to hit failure threshold in 80 minutes from now.
```

Examine the most recent time series model.

```
sys
```

```
sys =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + K e(t)
       y(t) = C x(t) + e(t)

  A =
           x1      x2
   x1       0       1
   x2  0.2624   0.746

  C =
       x1  x2
   y1   1   0

  K =
           y1
   x1  0.3902
   x2  0.3002

Sample time: 600 seconds

Parameterization:
   CANONICAL form with indices: 2.
   Disturbance component: estimate
   Number of free coefficients: 4
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "tsFeature".
Fit to estimation data: 92.53% (prediction focus)
FPE: 499.3, MSE: 442.7
```

The goodness of fit increases to above 90%, and the trend is correctly captured.

Conclusions

This example shows how to extract features from measured data to perform condition monitoring and prognostics. Based on extracted features, dynamic models are generated, validated and used to forecast time of failures so that actions can be taken before actual failures happen.

See Also

More About

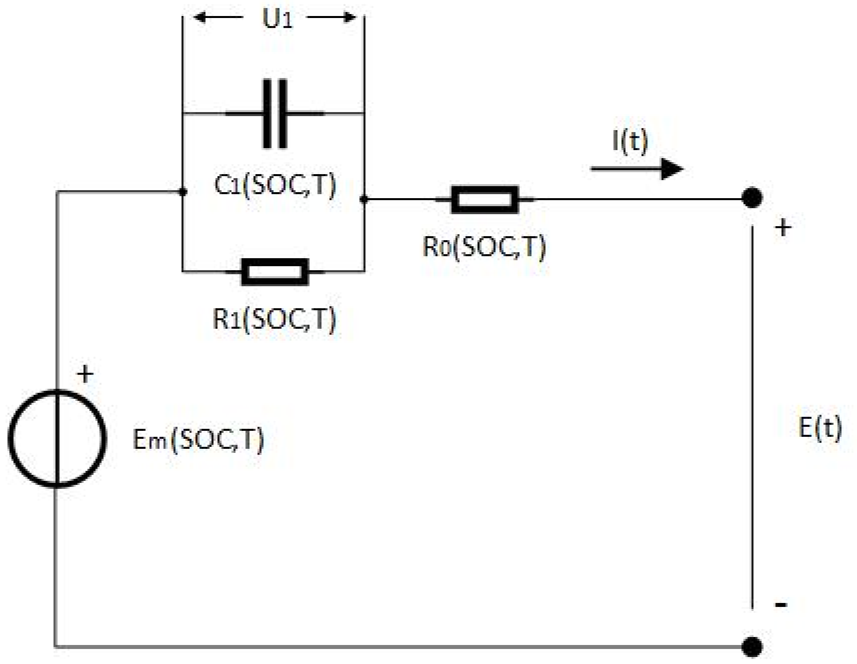
- “RUL Estimation Using Identified Models or State Estimators” on page 5-6

Nonlinear State Estimation of a Degrading Battery System

This example shows how to estimate the states of a nonlinear system using an unscented Kalman filter in Simulink™. The example also illustrates how to develop an event-based Kalman filter to update system parameters for more accurate state estimation. This example uses functionality from System Identification Toolbox™, and does not require Predictive Maintenance Toolbox™.

Overview

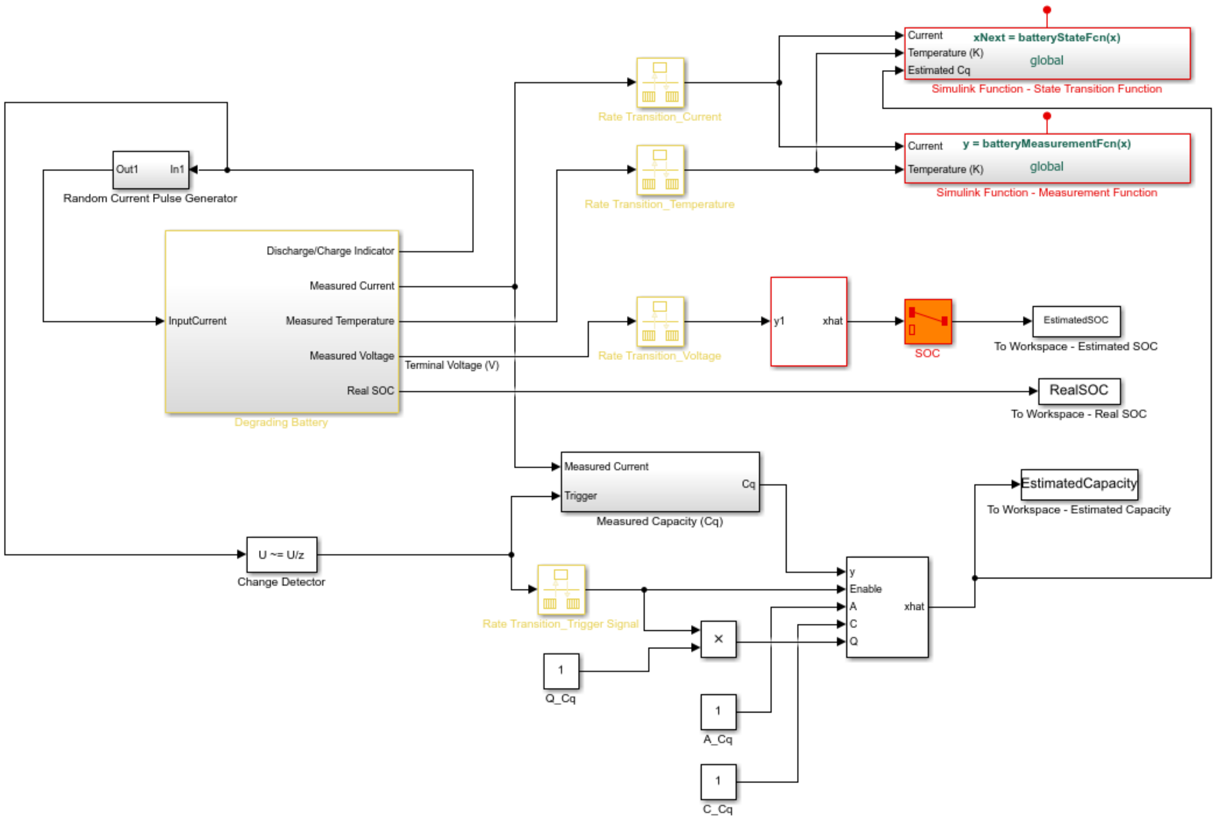
Consider a battery model with the following equivalent circuit [1]



The model consists of a voltage source E_m, a series resistor R_0 and a single RC block R_1 and C_1. The battery alternates between charging and discharging cycles. In this example, you estimate the state of charge (SOC) of a battery model using measured currents, voltages and temperatures of the battery. You assume the battery is a nonlinear system, estimate the SOC using an unscented Kalman filter. The capacity of the battery degrades with every discharge-charge cycle, giving an inaccurate SOC estimation. Use an event-based linear Kalman filter to estimate the battery capacity when the battery transitions between charging and discharging. The estimated capacity in turn can be used to indicate the health condition of the battery.

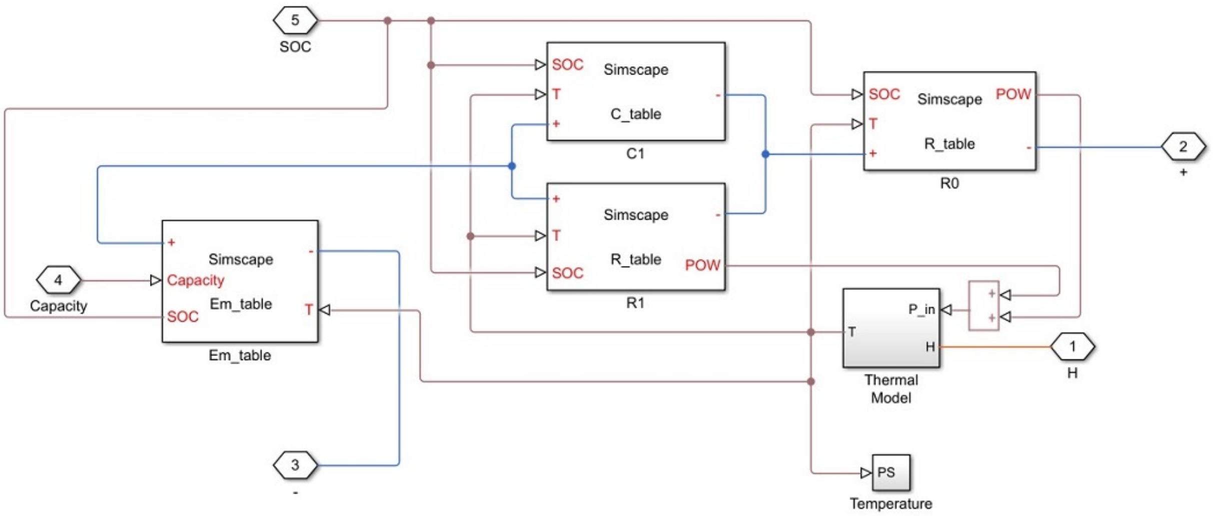
The Simulink model contains three major components: a battery model, an unscented Kalman filter block and an event-based Kalman filter block. Further explanations are given in the following sections.

```
open_system('BatteryExampleUKF/')
```



Battery Model

The battery model with thermal effect is implemented using the Simscape language.

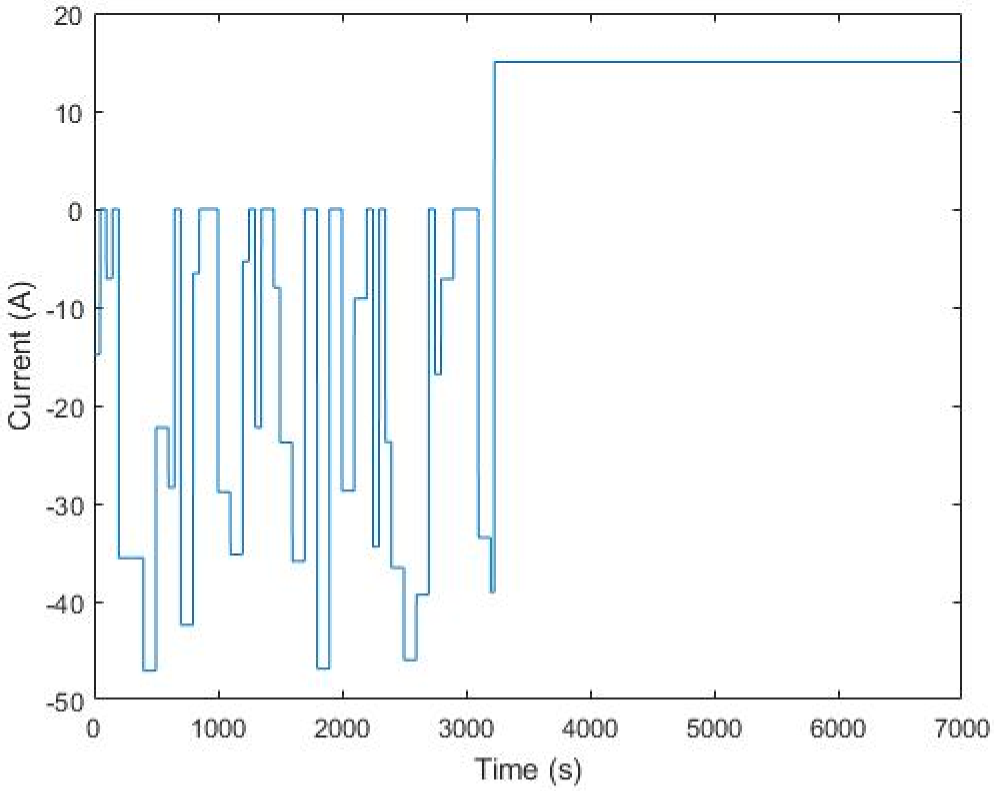


The state transition equations for the battery model are given by:

$$\frac{d}{dt}\begin{pmatrix} SOC \\ U_1 \end{pmatrix} = \begin{pmatrix} 0 \\ -\frac{1}{R_1(SOC,T_b)*C_1(SOC,T_b)}U_1 \end{pmatrix} + \begin{pmatrix} -\frac{1}{3600*C_q} \\ \frac{1}{C_1(SOC,T_b)} \end{pmatrix} I + W$$

where $R_1(SOC,T_b)$ and $C_1(SOC,T_b)$ are the thermal and SOC dependent resistor and capacitor in the RC block, U_1 is the voltage across capacitor C_1, I is the input current, T_b is the battery temperature, C_q is the battery capacity (unit: Ah), and W is the process noise.

The input currents are randomly generated pulses when the battery is discharging and constant when the battery is charging, as shown in the following figure.



The measurement equation is given by:

$$E = E_m(SOC, T_b) - U_1 - IR_0(SOC, T_b) + V$$

where E is the measured voltage output, $R_0(SOC, T_b)$ is the serial resistor, $E_m = E_m(SOC, T_b)$ is the electromotive force from voltage source, and V is the measurement noise.

In the model, R_0, R_1, C_1 and E_m are 2D look-up tables that are dependent on SOC and battery temperature. The parameters in the look-up tables are identified using experimental data [1].

Estimating State of Charge (SOC)

To use the unscented Kalman filter block, either Matlab or Simulink functions for the state and measurement equations need to be defined. This example demonstrates the use of Simulink functions .Since unscented Kalman filters are discrete-time filters, first discretize the state equations. In this example, Euler discretization is employed. Let the sampling time be Ts. For a general nonlinear system $\dot{x} = f(x, u)$, the system can be discretized as $x_{T+1} = x_T + f(x_T, u_T) * Ts$

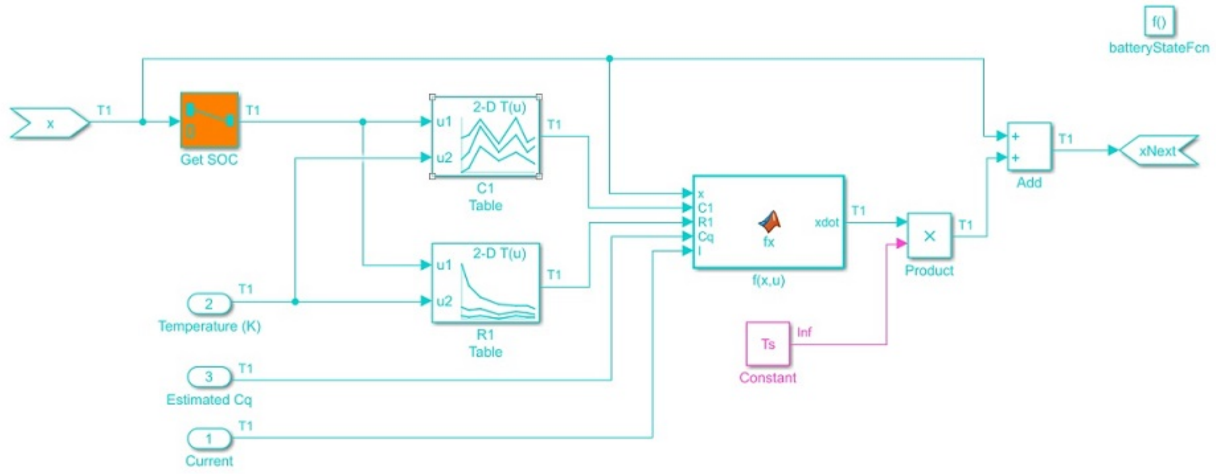
The state vectors of the nonlinear battery system are

$$x_T = \begin{pmatrix} SOC_T \\ U_{1_T} \end{pmatrix}.$$

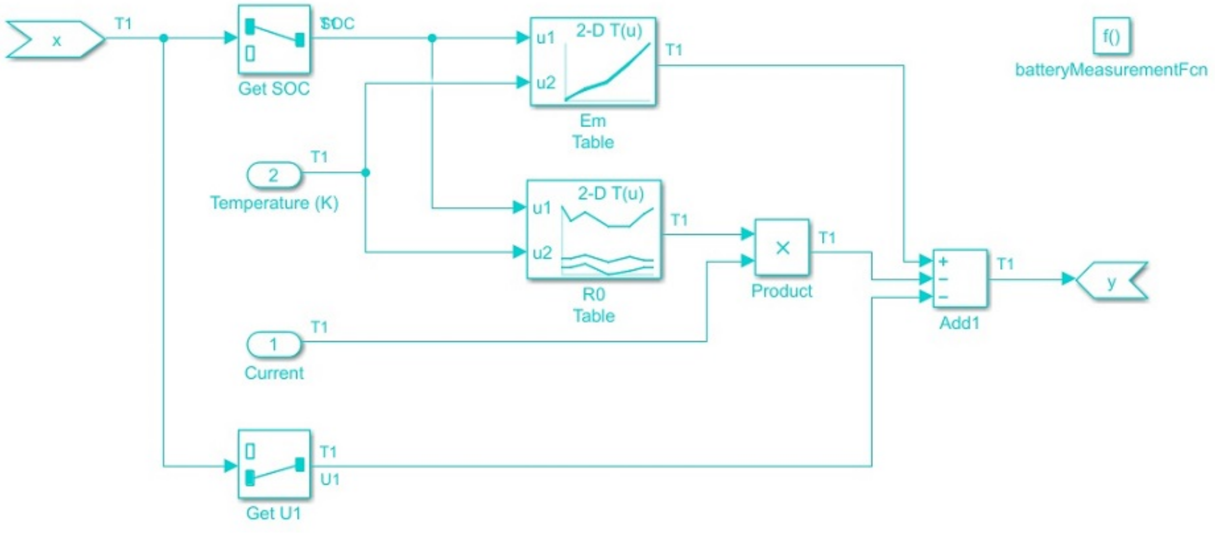
Applying Euler discretization gives the following equations:

$$\begin{pmatrix} SOC_{T+1} \\ U_{1_{T+1}} \end{pmatrix} = \begin{pmatrix} SOC_T \\ U_{1_T} \end{pmatrix} + \begin{pmatrix} -\frac{1}{3600*C_q} I \\ -\frac{1}{R_1(SOC_T,T_b)*C_1(SOC_T,T_b)} U_1 + \frac{1}{C_1(SOC_T,T_b)} I \end{pmatrix} Ts + W_T$$

The discretized state transition equation is implemented as a Simulink function named "batteryStateFcn" shown below. The function input `x` is the state vector while the function output `xNext` is the state vector at next step, calculated using discretized state transition equations. You need to specify the signal dimensions and data type of `x` and `xNext`. In this example, the signal dimension for `x` and `xNext` is 2 and the data type is double. Additional inputs are the temperature, estimated capacity, and current. Note that the additional inputs are inputs to the state transition equations and not required by the UKF block.

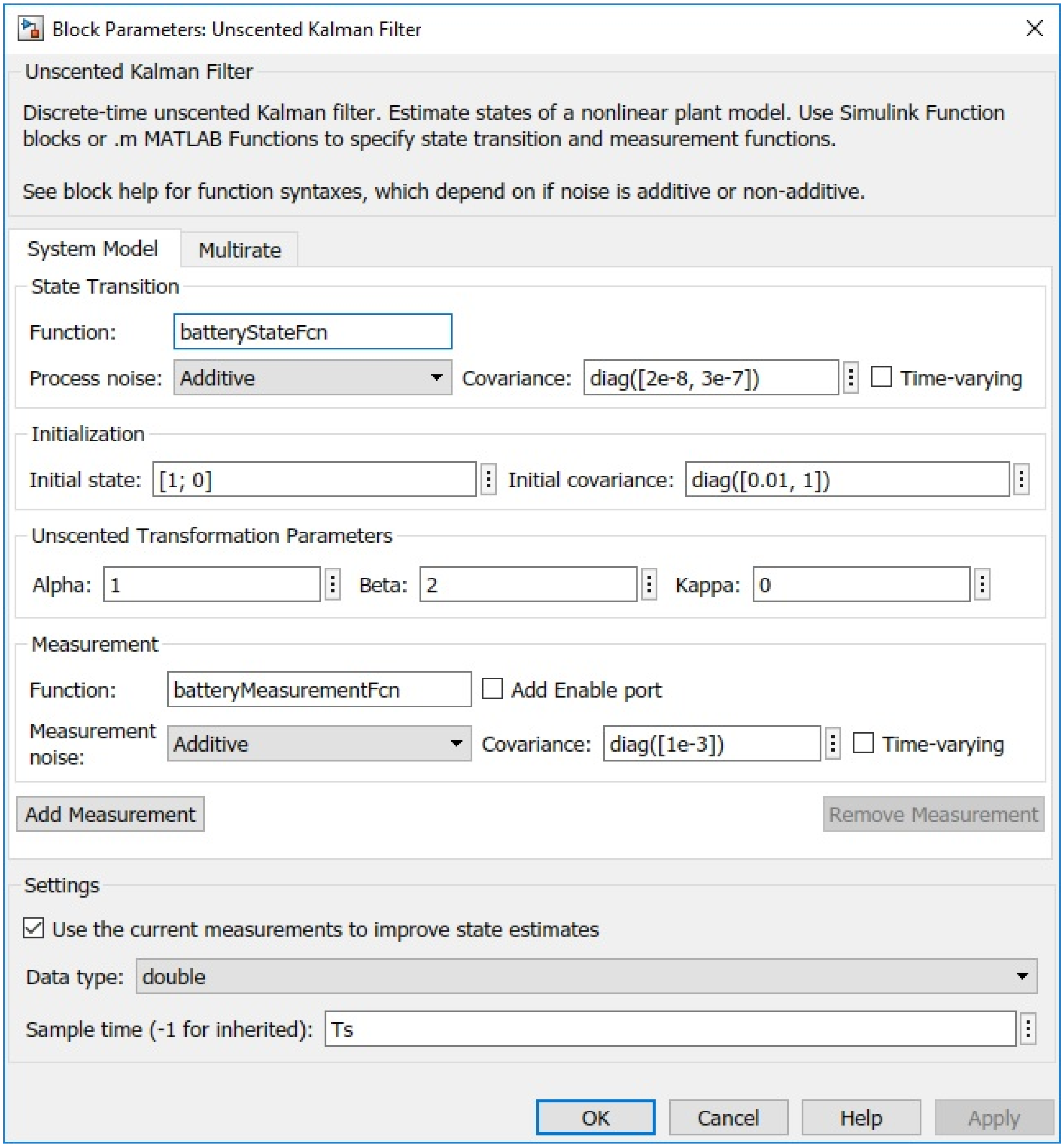


The measurement function is also implemented as a Simulink function named "batteryMeasurementFcn" as shown below.



Configure the block parameters as follows:

In the **System Model** tab, specify the block parameters as shown:



Block Parameters: Unscented Kalman Filter

Unscented Kalman Filter

Discrete-time unscented Kalman filter. Estimate states of a nonlinear plant model. Use Simulink Function blocks or .m MATLAB Functions to specify state transition and measurement functions.

See block help for function syntaxes, which depend on if noise is additive or non-additive.

System Model | Multirate

State Transition

Function: batteryStateFcn

Process noise: Additive Covariance: diag([2e-8, 3e-7]) ☐ Time-varying

Initialization

Initial state: [1; 0] Initial covariance: diag([0.01, 1])

Unscented Transformation Parameters

Alpha: 1 Beta: 2 Kappa: 0

Measurement

Function: batteryMeasurementFcn ☐ Add Enable port

Measurement noise: Additive Covariance: diag([1e-3]) ☐ Time-varying

Add Measurement | Remove Measurement

Settings

☑ Use the current measurements to improve state estimates

Data type: double

Sample time (-1 for inherited): Ts

You specify the following parameters:

- **Function in State Transition:** `batteryStateFcn`.

The name of the simulink function defined previously that implements the discretized state transition equation.

- **Process noise:** `Additive`, with time-varying covariance $\begin{bmatrix} 2e-8 & 0 \\ 0 & 3e-7 \end{bmatrix}$. The `Additive` means the noise term is added to the final signals directly.

The process noise for SOC and U_1 are estimated based on the dynamic characteristics of the battery system. The battery has nominal capacity of 30 Ah and undergoes discharge/charge cycles at an average current amplitude of 15A. Therefore, one discharging or charging process would take around 2 hours (7200 seconds). The maximum change is 100% for SOC and around 4 volts for U_1.

The maximum changes per step in SOC and U_1 are $max(|dSOC|) \approx \frac{100\%}{3600*2} * Ts$ and $max(|dU_1|) \approx \frac{4}{3600*2} * Ts$, where T_s is the sampling time of the filter. In this example, T_s is set to be 1 second.

The process noise W is:

$$W = \begin{bmatrix} (max(|dSOC|))^2 & 0 \\ 0 & (max(|dU_1|))^2 \end{bmatrix} \approx \begin{bmatrix} 2e-8 & 0 \\ 0 & 3e-7 \end{bmatrix}.$$

- **Initial State**: $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$.

The initial value for SOC is assumed to be 100% (fully charged battery) while initial value for U_1 is set to be 0, as we do not have any prior information of U_1.

- **Initial Covariance**:

Initial covariance indicates how accurate and reliable the initial guesses are. Assume the maximum initial guess error is 10% for SOC and 1V for U_1. The initial covariance matrix is set to be $\begin{bmatrix} 0.01 & 0 \\ 0 & 1 \end{bmatrix}$.

- **Unscented Transformation Parameters**: The setting is based on [2]

```
- Alpha: 1. Determine the spread of sigma points around x. Set Alpha to be 1 for l
- Beta: 2. Used to incorporate prior knowledge of the distribution. The nominal va
- Kappa: 0. Secondary scaling parameter. The nominal value for Kappa is 0.
```

- **Function in Measurement:** `batteryMeasurementFcn`.

the name of the simulink function defined previously that implements the measurement function.

- **Measurement noise:** `Additive`, with time-invariant covariance 1e-3.

The measurement noise is estimated based on measurement equipment accuracy. A voltage meter for battery voltage measurement has approximately 1% accuracy. The battery voltage is around 4V. Equivalently, we have $max(dE_m) \approx 4 * 1\% = 0.04$. Therefore, set $V = (max(dE_m))^2 \approx 1e-3$.

- **Sample Time**: Ts.

Estimating Battery Degradation

The battery degradation is modeled by decreasing capacity C_q. In this example, the battery capacity is set to decrease 1 Ah per discharge-charge cycle to illustrate the effect of degradation. Since the degradation rate of capacity is not known in advance, set the state equation of C_q to be a random walk:

$$C_{q_{k+1}} = C_{q_k} + W_{C_q}$$

where k is the number of discharge-charge cycles and W_{C_q} is the process noise.

The battery is configured to automatically charge when capacity is at 30% and switch to discharging capacity is 90%. Use this information to measure the battery capacity by integrating the current I over a charge or discharge cycle (coulomb counting).

The measurement equation for C_q is:

$$C_{q_k}^{Measured} = C_{q_k} + V_{C_q} = \frac{\int_{t_{k-1}}^{t_k} I dt}{(\Delta SOC)_{nominal}} = \frac{\int_{t_{k-1}}^{t_k} I dt}{|0.9 - 0.3|} = \frac{\int_{t_{k-1}}^{t_k} I dt}{0.6}$$

where V_{C_q} is the measurement noise.

The state and measurement equation of battery degradation can be put into the following state space form:

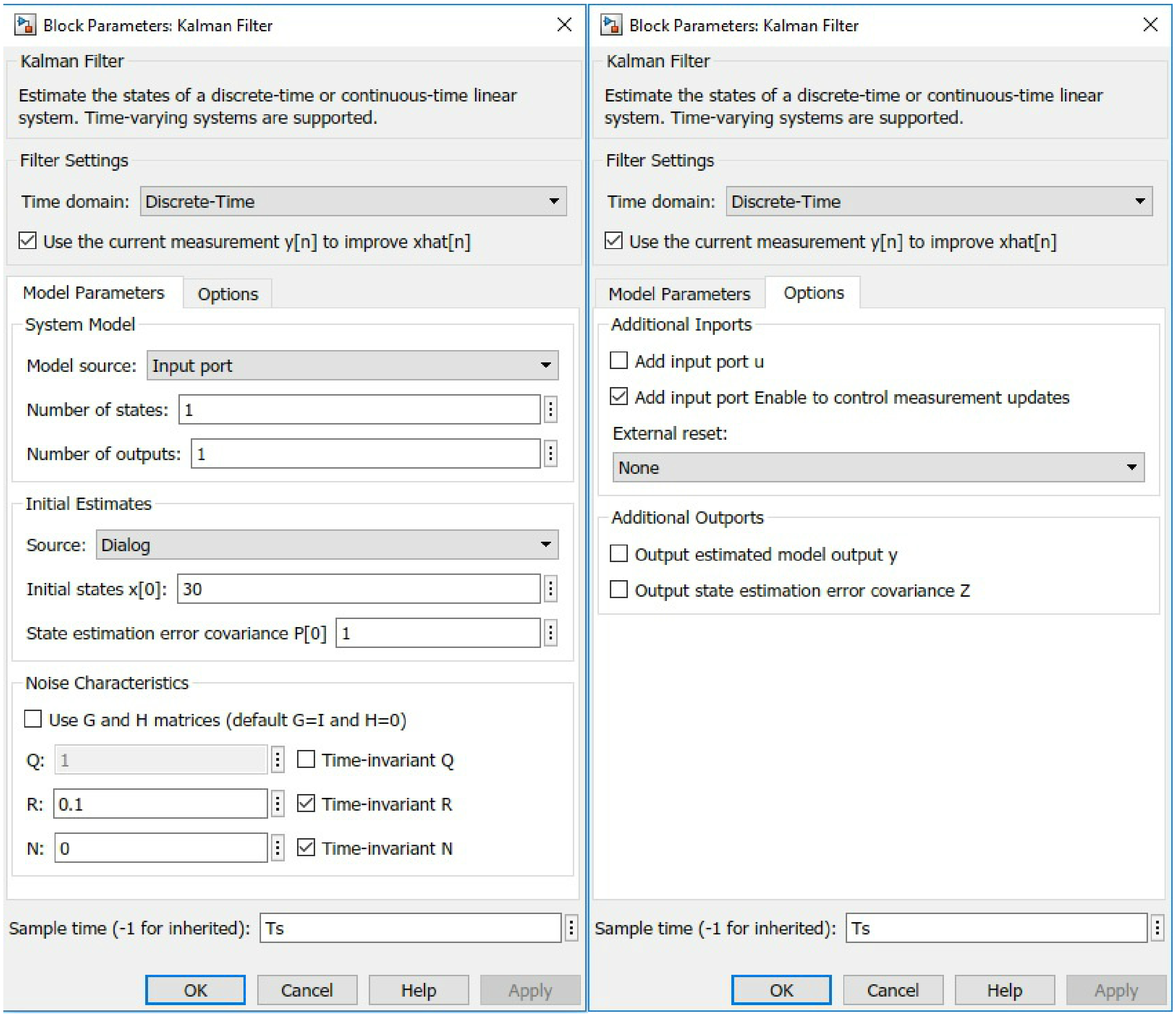
$$C_{q_{k+1}} = A_{C_q} C_{q_k} + W_{C_q}$$

$$C_{q_k}^{Measured} = C_{C_q} C_{q_k} + V_{C_q}$$

where A_{C_q} and C_{C_q} are equal to 1.

For the above linear system, use a Kalman filter to estimate battery capacity. The estimated C_q from the linear Kalman filter is used to improve SOC estimation. In the example, an event-based linear Kalman filter is used to estimate C_q. Since C_q is measured once over a charge or discharge cycle, the linear Kalman filter is enabled only when charging or discharging ends.

Configure the block parameters and options as follows:



Block Parameters: Kalman Filter

Kalman Filter

Estimate the states of a discrete-time or continuous-time linear system. Time-varying systems are supported.

Filter Settings

Time domain: Discrete-Time

☑ Use the current measurement y[n] to improve xhat[n]

Model Parameters | Options

System Model

Model source: Input port

Number of states: 1

Number of outputs: 1

Initial Estimates

Source: Dialog

Initial states x[0]: 30

State estimation error covariance P[0] 1

Noise Characteristics

☐ Use G and H matrices (default G=I and H=0)

Q: 1 ☐ Time-invariant Q

R: 0.1 ☑ Time-invariant R

N: 0 ☑ Time-invariant N

Sample time (-1 for inherited): Ts

Block Parameters: Kalman Filter

Kalman Filter

Estimate the states of a discrete-time or continuous-time linear system. Time-varying systems are supported.

Filter Settings

Time domain: Discrete-Time

☑ Use the current measurement y[n] to improve xhat[n]

Model Parameters | Options

Additional Inports

☐ Add input port u

☑ Add input port Enable to control measurement updates

External reset: None

Additional Outports

☐ Output estimated model output y

☐ Output state estimation error covariance Z

Sample time (-1 for inherited): Ts

Click **Model Parameters** to specify the plant model and noise characteristics:

- **Model source:** `Input Port`.

To implement an event-based Kalman filter, the state equation is enabled only when the event happens. In other word, the state equation is event-based as well. For a linear system $x_{t+1} = Ax_t + Bu_t + w_t$, set the state equation to be

$$x_{t+1} = \begin{cases} Ax_t + Bu_t + w_t, t = t_{enabled} \\ x_t, t \neq t_{enabled} \end{cases}.$$

- **A**: $\begin{cases} A_{C_q}, t = t_{enabled} \\ 1, t \neq t_{enabled} \end{cases}$. In this example, $A_{C_q} = 1$. As a result, A equals 1 all the time.
- **C**: 1, from $C_{q_k}^{Measured} = C_{q_k} + V_{C_q} = \frac{\int_{t_{k-1}}^{t_k} Idt}{0.6}$.
- **Initial Estimate Source**: `Dialog`. You specify initial states in `Initial state x[0]`
- **Initial states x[0]**: 30. It is the nominal capacity of battery (30Ah).
- **Q**: $\begin{cases} 1, t = t_{enabled} \\ 0, t \neq t_{enabled} \end{cases}$

This is the covariance of the process noise W_{C_q}. Since degradation rate in the capacity is around 1 Ah per discharge-charge cycle, set the process noise to be 1.

- **R**: 0.1. This is the covariance of the measurement noise V_{C_q}. Assume the capacity measurement error is less than 1%. With battery capacity of 30 Ah, the measurement noise $V_{C_q} \approx (0.3)^2 \approx 0.1$.
- **Sample Time**: Ts.

Click **Options** to add input port `Enable` to control measurement updates. The enable port is used to update the battery capacity estimate on charge/discharge events as opposed to continually updating.

Note that setting `Enable` to 0 does not disable predictions using state equations. It is the reason why the state equation is configured to be event-based as well. By setting an event-based A and Q for the Kalman filter block, predictions using state equations are disabled when `Enable` is set to be 0.

Results

To simulate the system, load the battery parameters. The file contains the battery parameters including $E_m(SOC,T)$, $R_0(SOC,T)$, $R_1(SOC,T)$ and etc.

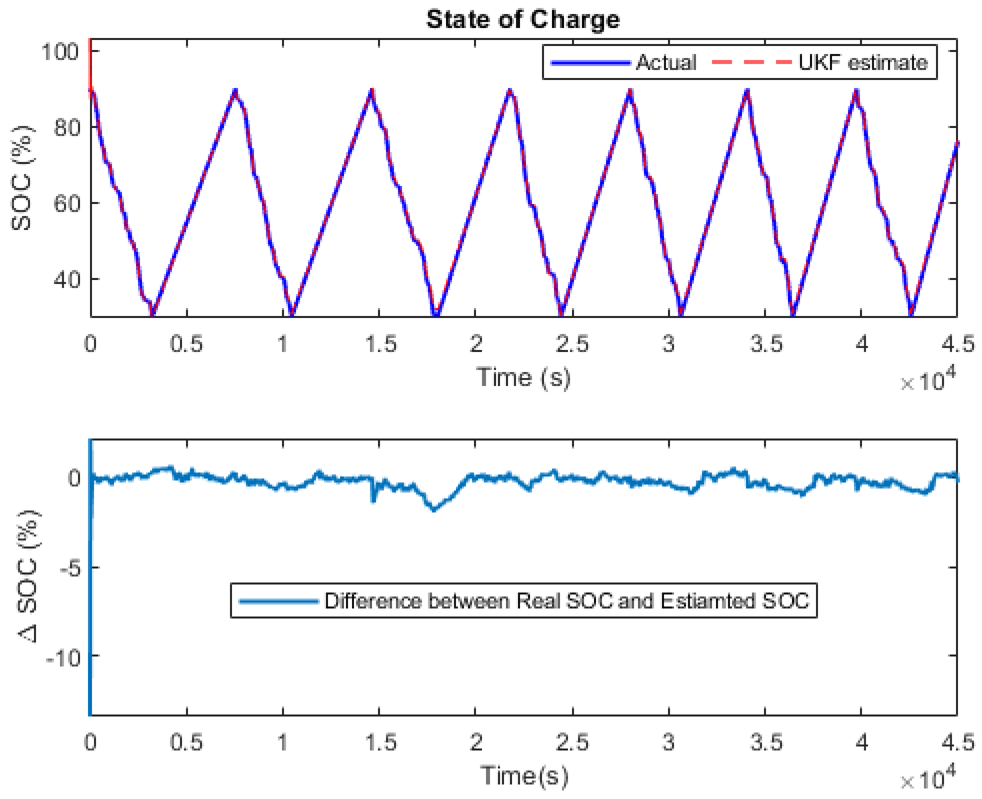
```
load BatteryParameters.mat
```

Simulate the system.

```
sim('BatteryExampleUKF')
```

At every time step, the unscented Kalman filter provides an estimation for SOC based on voltage measurements E_m. Plot the real SOC, the estimated SOC, and the difference between them.

```
% Synchronize two time series
[RealSOC, EstimatedSOC] = synchronize(RealSOC, EstimatedSOC, 'intersection');

figure;
subplot(2,1,1)
plot(100*RealSOC,'b','LineWidth',1.5);
hold on
plot(100*EstimatedSOC,'r--','LineWidth',1);
title('State of Charge');
xlabel('Time (s)');
ylabel('SOC (%)');
legend('Actual','UKF estimate','Location','Best','Orientation','horizontal');
axis tight

subplot(2,1,2)
DiffSOC = 100*(RealSOC - EstimatedSOC);
plot(DiffSOC.Time, DiffSOC.Data, 'LineWidth', 1.5);
xlabel('Time(s)');
ylabel('\Delta SOC (%)','Interpreter','Tex');
legend('Difference between Real SOC and Estiamted SOC','Location','Best')
axis tight
```

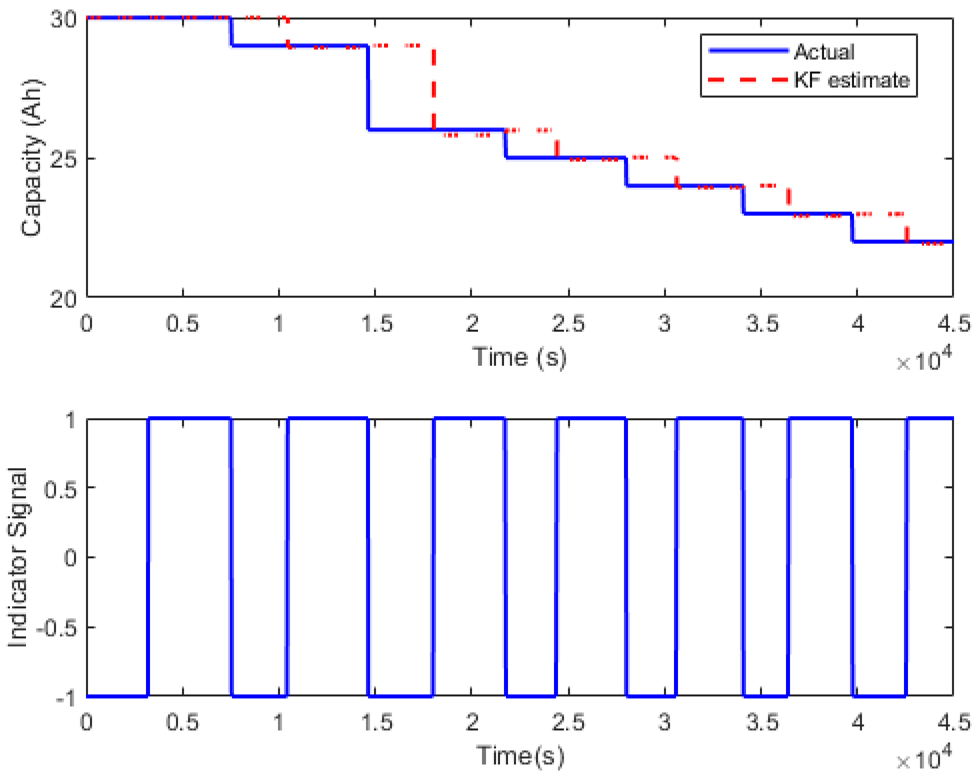


After an initial estimation error, the SOC converges quickly to the real SOC. The final estimation error is within 0.5% error. The unscented Kalman filter gives an accurate estimation of SOC.

At every discharge-charge transitions, the battery capacity is estimated to improve the SOC estimation. The battery system outputs indicator signals to inform what process the battery is in. Discharging process is represented by -1 in the indicator signals while charging process is represented by 1. In this example, changes in the indicator signals are used to determine when to enable or disable Kalman filter for capacity estimation. We plot the real and estimated capacity as well as the charge-discharge indicator signals.

```
figure;
subplot(2,1,1);
```

```
plot(RealCapacity,'b','LineWidth',1.5);
hold on
plot(EstimatedCapacity,'r--','LineWidth',1.5);
xlabel('Time (s)');
ylabel('Capacity (Ah)');
legend('Actual','KF estimate','Location','Best');

subplot(2,1,2);
plot(DischargeChargeIndicator.Time,DischargeChargeIndicator.Data,'b','LineWidth',1.5);
xlabel('Time(s)');
ylabel('Indicator Signal');
```



In general, the Kalman filter is able to track the real capacity. There is half cycle delay between estimated capacity and real capacity. It is because the battery capacity degradation happens when one full discharge-charge cycle ends. While the coulomb counting gives a capacity measurement of the last discharge or charge cycle.

Summary

This example shows how to use Simulink unscented Kalman filter block to perform nonlinear state estimation for a lithium battery. In addition, steps to develop an event-based Kalman filter for battery capacity estimation are illustrated. The newly estimated capacity is used to improve SOC estimation in the unscented Kalman filter.

Reference

[1] Huria, Tarun, et al. "High fidelity electrical model with thermal dependence for characterization and simulation of high power lithium battery cells." Electric Vehicle Conference (IEVC), 2012 IEEE International. IEEE, 2012.

[2] Wan, Eric A., and Rudolph Van Der Merwe. "The unscented Kalman filter for nonlinear estimation." Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000. Ieee, 2000.

See Also

More About

- “RUL Estimation Using Identified Models or State Estimators” on page 5-6

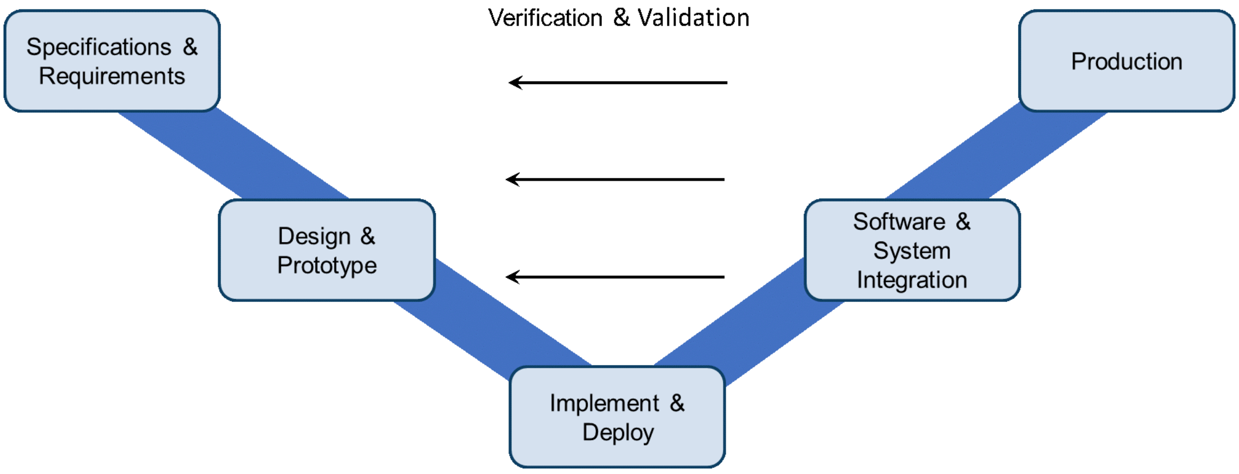
6

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:


Verification & Validation
Specifications & Requirements
Production
Design & Prototype
Software & System Integration
Implement & Deploy

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"