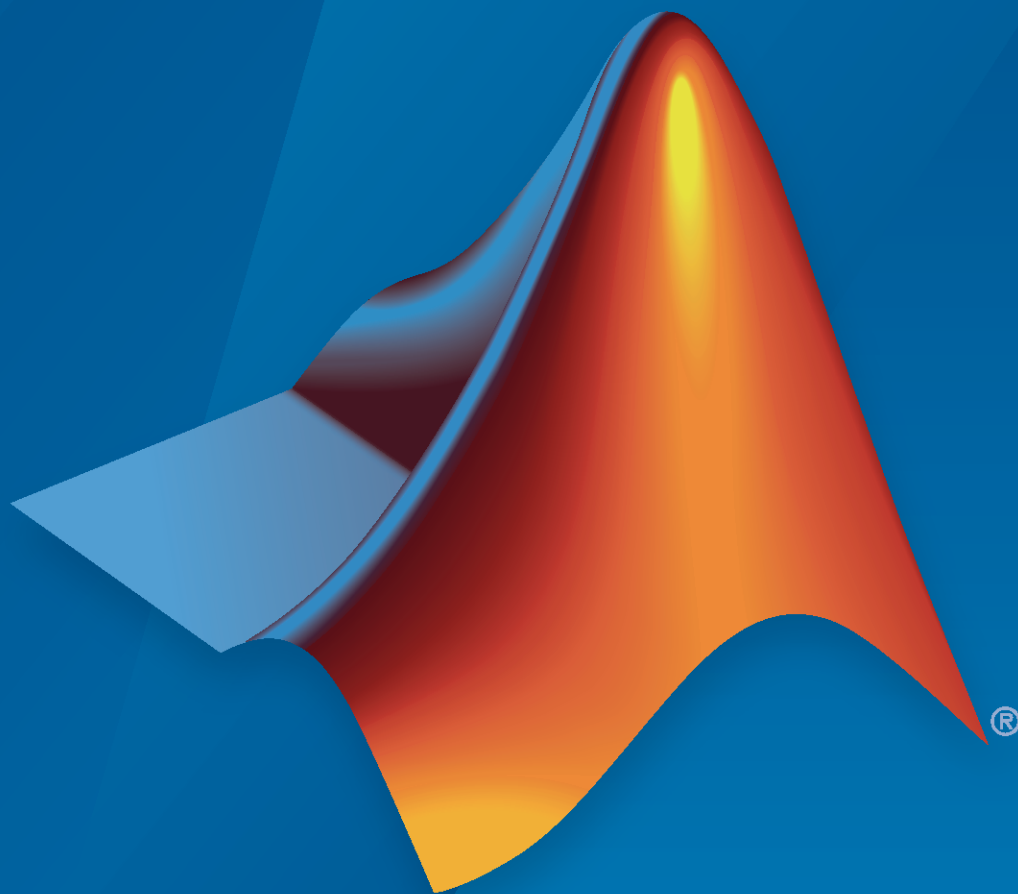# Predictive Maintenance Toolbox™

User's Guide

MATLAB®

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Predictive Maintenance Toolbox™ User's Guide

© COPYRIGHT 2018–2023 by The MathWorks, Inc.

Trademarks

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

Patents

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

Revision History

Contents

Manage System Data

1

Preprocess Data

2

Identify Condition Indicators

3

Detect and Diagnose Faults

4

Predict Remaining Useful Life

5

Deploy Predictive Maintenance Algorithms

6

<div align="right">

Diagnostic Feature Designer

</div>

7

<div style="text-align:right">

Diagnostic Feature Designer Help

</div>

8

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.

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
$\begin{bmatrix} \text{time-series} \\ \text{data} \end{bmatrix}$	$\begin{bmatrix} \text{time-series} \\ \text{data} \end{bmatrix}$	0
$\begin{bmatrix} \text{time-series} \\ \text{data} \end{bmatrix}$	$\begin{bmatrix} \text{time-series} \\ \text{data} \end{bmatrix}$	0.1
$\begin{bmatrix} \text{time-series} \\ \text{data} \end{bmatrix}$	$\begin{bmatrix} \text{time-series} \\ \text{data} \end{bmatrix}$	0.2
$\begin{bmatrix} \text{time-series} \\ \text{data} \end{bmatrix}$	$\begin{bmatrix} \text{time-series} \\ \text{data} \end{bmatrix}$	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.

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.)

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 1-10.

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 1-17
- "File Ensemble Datastore Using Data in Text Files" on page 1-21

Ensembles and MATLAB Datastores

Ensembles in Predictive Maintenance Toolbox are a specialized kind of MATLAB datastore (see "Getting Started with Datastore"). 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 for Out-of-Memory Data").
- `progress` — Determine what percentage of an ensemble datastore has been read.
- `partition` — Partition an ensemble datastore into multiple ensemble datastores for parallel computing. (For ensemble datastores, use the `partition(ds,n,index)` syntax.)
- `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 4-6
- "Using Simulink to Generate Fault Data" on page 1-25

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 | simulationEnsembleDatastore | read | generateSimulationEnsemble

More About
- "Generate and Use Simulated Data Ensemble" on page 1-10
- "File Ensemble Datastore with Measured Data" on page 1-17
- "File Ensemble Datastore Using Data in Text Files" on page 1-21
- "Organize System Data for Diagnostic Feature Designer" on page 7-19

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 1-25. Load the Simulink model.

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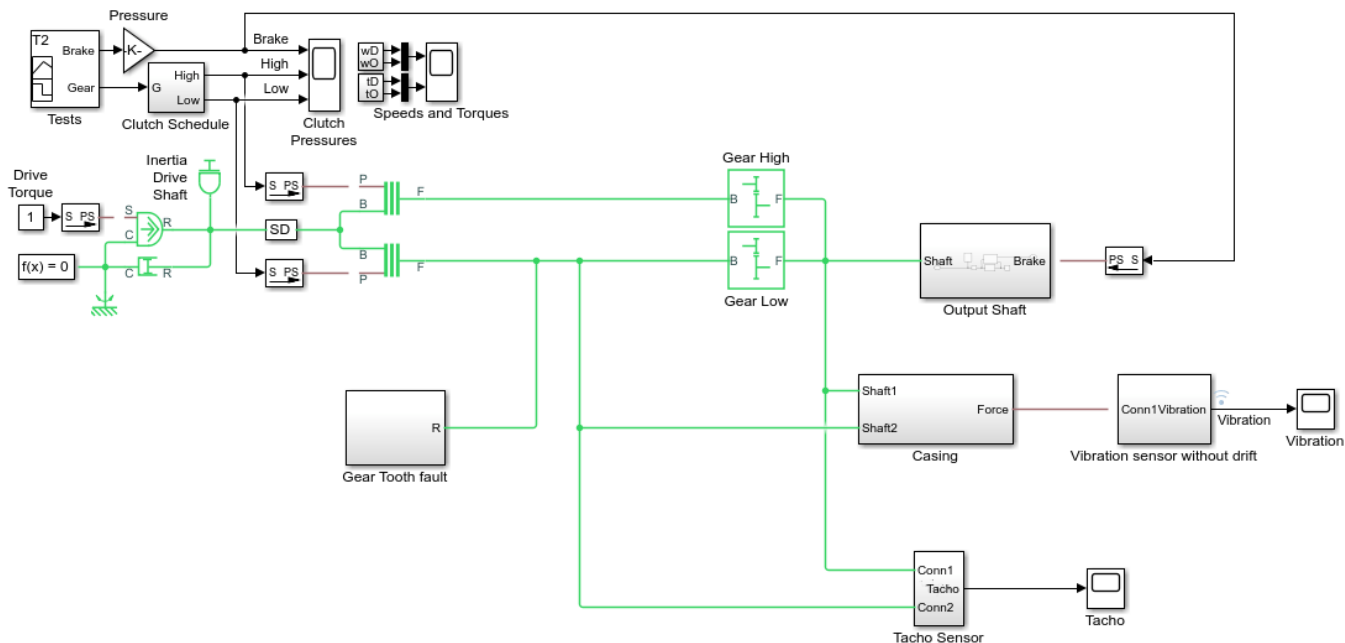

For this example, only one fault mode is modeled. The gear-tooth fault is modeled as a disturbance in the `Gear Tooth fault` subsystem. The magnitude of the disturbance is controlled by the model variable `ToothFaultGain`, where `ToothFaultGain = 0` corresponds to no gear tooth fault (healthy operation).

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 "Save 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);

[03-Mar-2023 15:33:32] Running simulations...
[03-Mar-2023 15:33:44] Completed 1 of 5 simulation runs
[03-Mar-2023 15:33:49] Completed 2 of 5 simulation runs
[03-Mar-2023 15:33:53] Completed 3 of 5 simulation runs
[03-Mar-2023 15:33:56] Completed 4 of 5 simulation runs
[03-Mar-2023 15:34:00] 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
    "SimulationInput"
    "SimulationMetadata"
    "Tacho"
    "Vibration"
```

```
ensemble.SelectedVariables
```

```
ans = 4x1 string
    "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
    _____    _____

    {589x1 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')
```
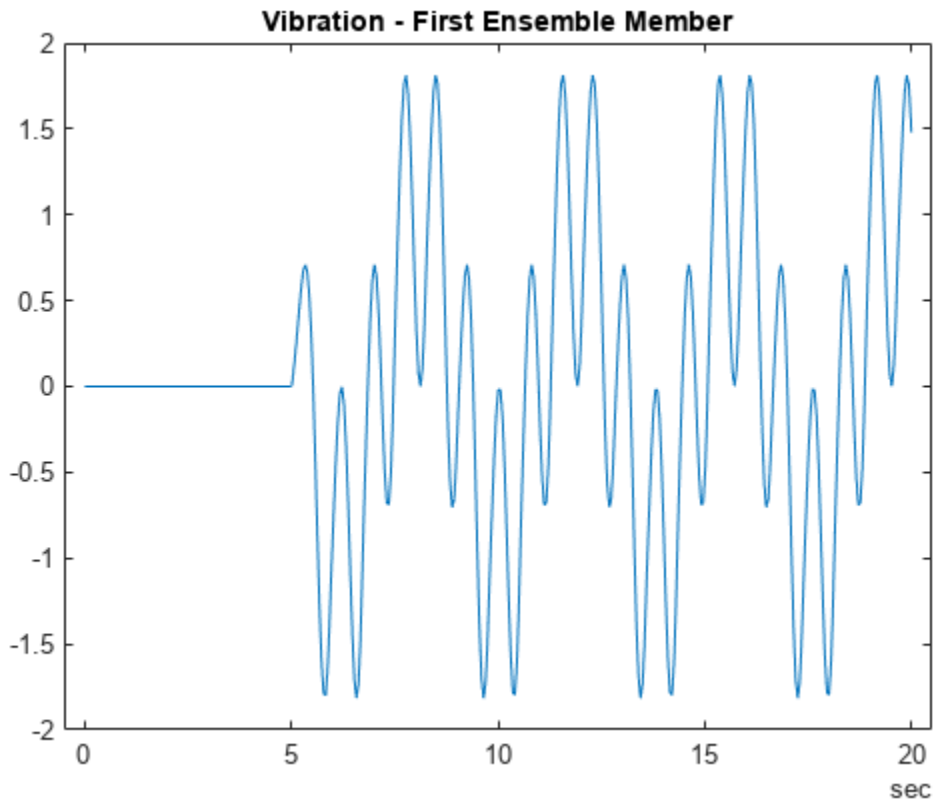
The `LastMemberRead` property of the ensemble contains the file name of the most recently read member. The next time you call `read` on this ensemble, the software advances to the next member of the ensemble. (See "Data Ensembles for Condition Monitoring and Predictive Maintenance" on page 1-2 for more information.) Read the selected variables from the next member of the ensemble.

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

```
data2=1×2 table
       Vibration                 SimulationInput

    _____    _____

    {603x1 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'
```

```
        Context: ''
    Description: ""


SimInput2 = data2.SimulationInput{1};
SimInput2.Variables

ans =
  Variable with properties:

           Name: 'ToothFaultGain'
          Value: -1.5000
      Workspace: 'global-workspace'
        Context: ''
    Description: ""
```

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
    "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 1-25.

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
    _____     _____

    {589x1 timetable}         false
    {603x1 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
    "C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp3cebbc15\predmaint-ex54897023\Data\TransmissionCa
    "C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp3cebbc15\predmaint-ex54897023\Data\TransmissionCa
```

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

`simulationEnsembleDatastore` | `generateSimulationEnsemble` | `read` | `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.

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

```
bearing = struct with fields:
      sr: 97656
      gs: [5000x1 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.

```
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"    {5000x1 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
    "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"      {5000x1 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 4-6. 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}

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 1-21

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.

```
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 timetable}     {12
```

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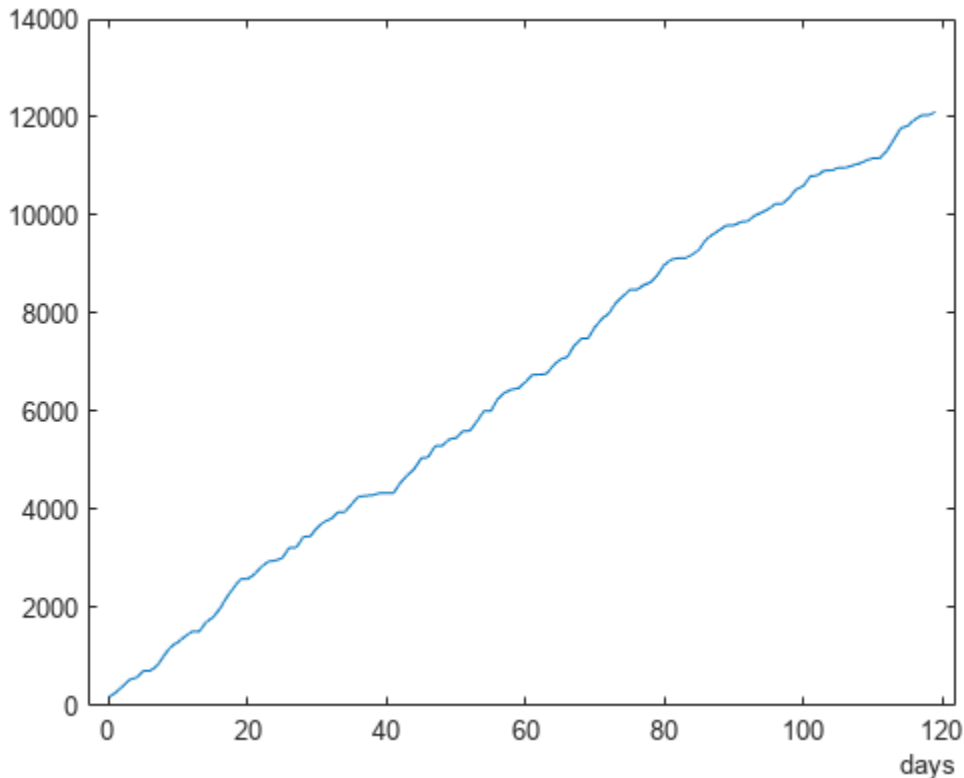
Compute the average gas mileage for this member of the fleet. This value is the odometer reading on the last day, divided by the total fuel consumed.

```
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 = zeros(10,2);     % preallocate array for 10 ensemble members
ct = 1;
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(ct,:) = [ID,mpg];
        ct = ct + 1;
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 1-17
- "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)
```
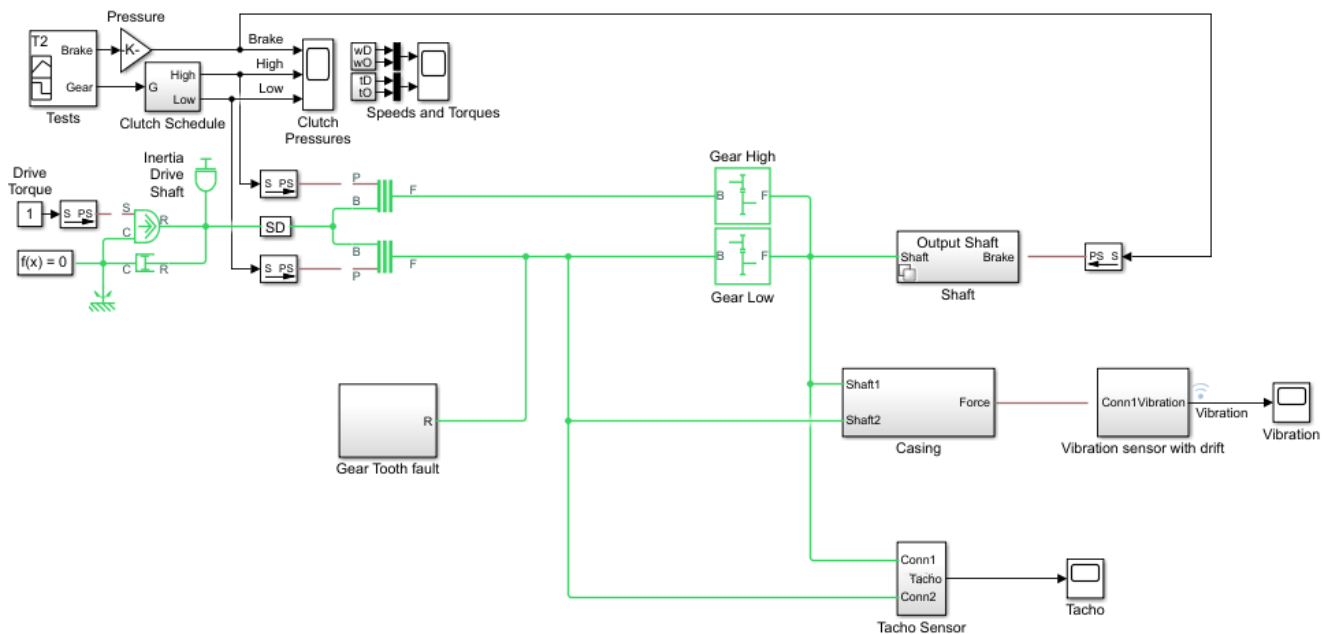

Copyright 2017 The MathWorks, Inc.

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'])
```
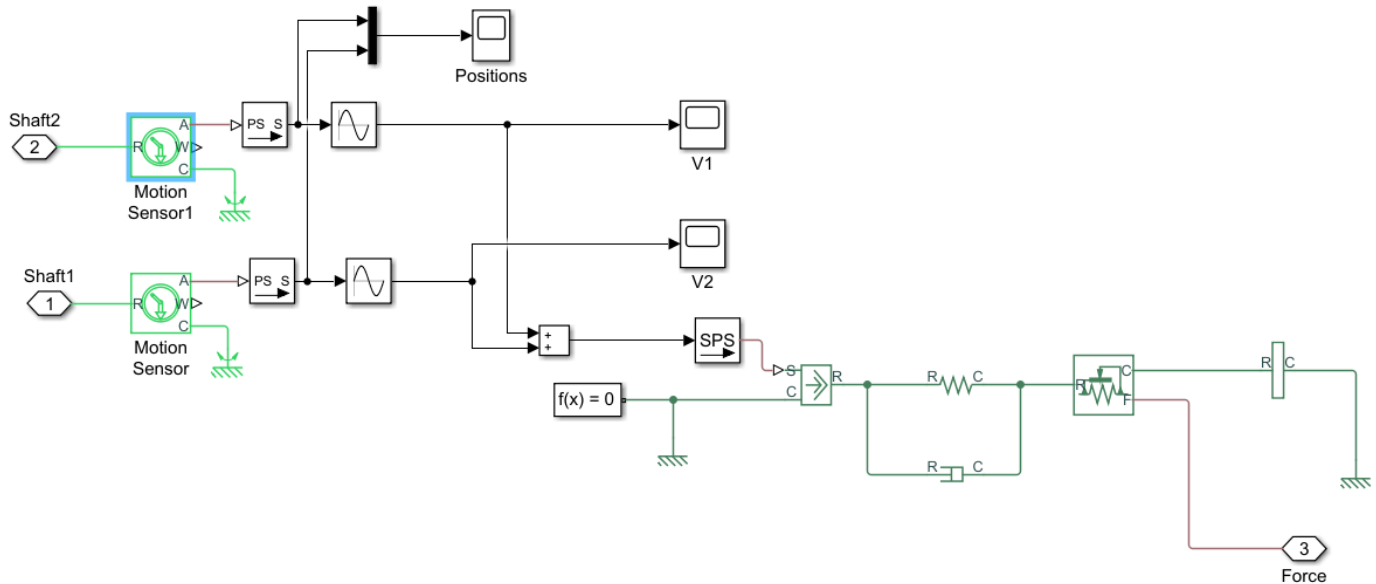
Fault Modelling

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'])
```
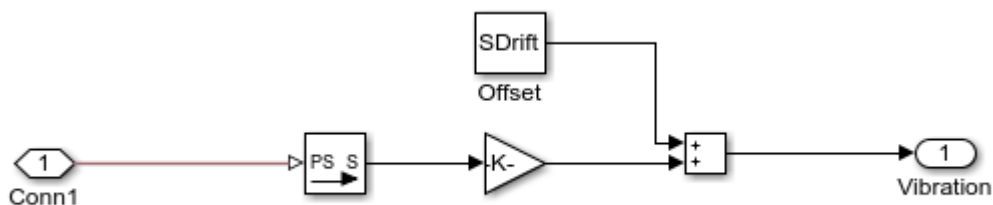

The shaft wear fault is modeled by a variant subsystem. In this case the subsystem variants change the shaft damping but the variant subsystem could be used to completely change the shaft model implementation. The selected variant is controlled by the model variable `ShaftWear`, note that `ShaftWear=0` 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.
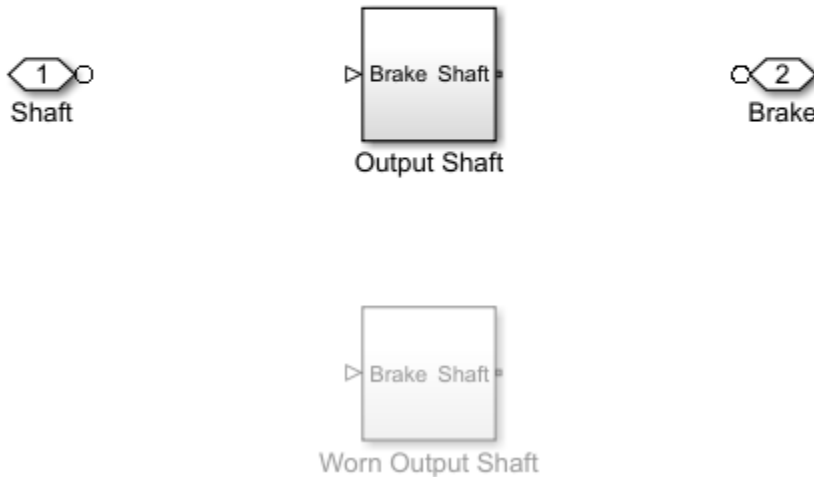

```
open_system([mdl,'/Shaft/Output Shaft'])
```
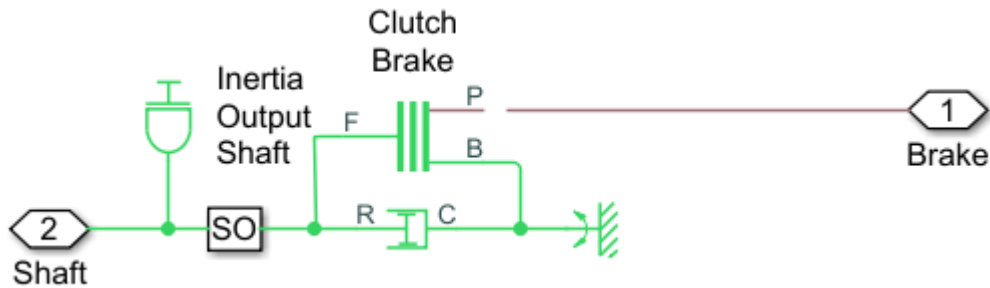

The gear tooth fault is modelled by injecting a disturbance torque at a fixed position in the rotation of the drive shaft. The shaft position is measured in radians and when the shaft position is within a small window around 0 a disturbance force is applied to the shaft. The magnitude of the disturbance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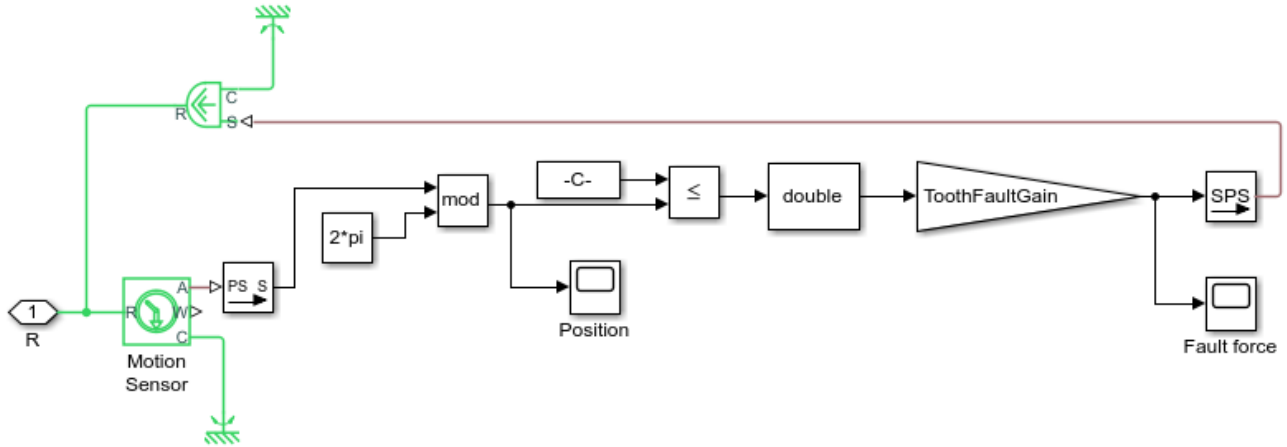
Simulating Fault and Healthy Data

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 can take as much as two hours or more 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')); %#ok<*U
end
```

```
[21-Jul-2022 15:36:27] Checking for availability of parallel pool...
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
[21-Jul-2022 15:37:34] Starting Simulink on parallel workers...
[21-Jul-2022 15:38:00] Configuring simulation cache folder on parallel workers...
[21-Jul-2022 15:38:01] Transferring base workspace variables used in the model to parallel worker
[21-Jul-2022 15:38:01] Loading model on parallel workers...
[21-Jul-2022 15:38:20] Running simulations...
[21-Jul-2022 15:38:55] Completed 1 of 208 simulation runs
[21-Jul-2022 15:38:56] Completed 2 of 208 simulation runs
[21-Jul-2022 15:38:57] Completed 3 of 208 simulation runs
[21-Jul-2022 15:38:58] Completed 4 of 208 simulation runs
[21-Jul-2022 15:38:59] Completed 5 of 208 simulation runs
[21-Jul-2022 15:39:00] Completed 6 of 208 simulation runs
[21-Jul-2022 15:39:09] Completed 7 of 208 simulation runs
[21-Jul-2022 15:39:09] Completed 8 of 208 simulation runs
```

```
[21-Jul-2022 15:39:10] Completed 9 of 208 simulation runs
[21-Jul-2022 15:39:11] Completed 10 of 208 simulation runs
[21-Jul-2022 15:39:12] Completed 11 of 208 simulation runs
[21-Jul-2022 15:39:13] Completed 12 of 208 simulation runs
[21-Jul-2022 15:39:21] Completed 13 of 208 simulation runs
[21-Jul-2022 15:39:22] Completed 14 of 208 simulation runs
[21-Jul-2022 15:39:23] Completed 15 of 208 simulation runs
[21-Jul-2022 15:39:23] Completed 16 of 208 simulation runs
[21-Jul-2022 15:39:24] Completed 17 of 208 simulation runs
[21-Jul-2022 15:39:25] Completed 18 of 208 simulation runs
[21-Jul-2022 15:39:34] Completed 19 of 208 simulation runs
[21-Jul-2022 15:39:35] Completed 20 of 208 simulation runs
[21-Jul-2022 15:39:35] Completed 21 of 208 simulation runs
[21-Jul-2022 15:39:36] Completed 22 of 208 simulation runs
[21-Jul-2022 15:39:37] Completed 23 of 208 simulation runs
[21-Jul-2022 15:39:38] Completed 24 of 208 simulation runs
[21-Jul-2022 15:39:47] Completed 25 of 208 simulation runs
[21-Jul-2022 15:39:47] Completed 26 of 208 simulation runs
[21-Jul-2022 15:39:48] Completed 27 of 208 simulation runs
[21-Jul-2022 15:39:49] Completed 28 of 208 simulation runs
[21-Jul-2022 15:39:50] Completed 29 of 208 simulation runs
[21-Jul-2022 15:39:51] Completed 30 of 208 simulation runs
[21-Jul-2022 15:40:00] Completed 31 of 208 simulation runs
[21-Jul-2022 15:40:00] Completed 32 of 208 simulation runs
[21-Jul-2022 15:40:01] Completed 33 of 208 simulation runs
[21-Jul-2022 15:40:02] Completed 34 of 208 simulation runs
[21-Jul-2022 15:40:03] Completed 35 of 208 simulation runs
[21-Jul-2022 15:40:05] Completed 36 of 208 simulation runs
[21-Jul-2022 15:40:13] Completed 37 of 208 simulation runs
[21-Jul-2022 15:40:14] Completed 38 of 208 simulation runs
[21-Jul-2022 15:40:15] Completed 39 of 208 simulation runs
[21-Jul-2022 15:40:16] Completed 40 of 208 simulation runs
[21-Jul-2022 15:40:17] Completed 41 of 208 simulation runs
[21-Jul-2022 15:40:18] Completed 42 of 208 simulation runs
[21-Jul-2022 15:40:26] Completed 43 of 208 simulation runs
[21-Jul-2022 15:40:27] Completed 44 of 208 simulation runs
[21-Jul-2022 15:40:27] Completed 45 of 208 simulation runs
[21-Jul-2022 15:40:28] Completed 46 of 208 simulation runs
[21-Jul-2022 15:40:29] Completed 47 of 208 simulation runs
[21-Jul-2022 15:40:30] Completed 48 of 208 simulation runs
[21-Jul-2022 15:40:38] Completed 49 of 208 simulation runs
[21-Jul-2022 15:40:39] Completed 50 of 208 simulation runs
[21-Jul-2022 15:40:40] Completed 51 of 208 simulation runs
[21-Jul-2022 15:40:41] Completed 52 of 208 simulation runs
[21-Jul-2022 15:40:42] Completed 53 of 208 simulation runs
[21-Jul-2022 15:40:43] Completed 54 of 208 simulation runs
[21-Jul-2022 15:40:52] Completed 55 of 208 simulation runs
[21-Jul-2022 15:40:55] Completed 56 of 208 simulation runs
[21-Jul-2022 15:40:56] Completed 57 of 208 simulation runs
[21-Jul-2022 15:40:57] Completed 58 of 208 simulation runs
[21-Jul-2022 15:40:58] Completed 59 of 208 simulation runs
[21-Jul-2022 15:40:59] Completed 60 of 208 simulation runs
[21-Jul-2022 15:41:09] Completed 61 of 208 simulation runs
[21-Jul-2022 15:41:10] Completed 62 of 208 simulation runs
[21-Jul-2022 15:41:11] Completed 63 of 208 simulation runs
[21-Jul-2022 15:41:12] Completed 64 of 208 simulation runs
[21-Jul-2022 15:41:13] Completed 65 of 208 simulation runs
[21-Jul-2022 15:41:14] Completed 66 of 208 simulation runs
```

```
[21-Jul-2022 15:41:20] Completed 67 of 208 simulation runs
[21-Jul-2022 15:41:21] Completed 68 of 208 simulation runs
[21-Jul-2022 15:41:22] Completed 69 of 208 simulation runs
[21-Jul-2022 15:41:22] Completed 70 of 208 simulation runs
[21-Jul-2022 15:41:23] Completed 71 of 208 simulation runs
[21-Jul-2022 15:41:24] Completed 72 of 208 simulation runs
[21-Jul-2022 15:41:33] Completed 73 of 208 simulation runs
[21-Jul-2022 15:41:33] Completed 74 of 208 simulation runs
[21-Jul-2022 15:41:34] Completed 75 of 208 simulation runs
[21-Jul-2022 15:41:35] Completed 76 of 208 simulation runs
[21-Jul-2022 15:41:36] Completed 77 of 208 simulation runs
[21-Jul-2022 15:41:37] Completed 78 of 208 simulation runs
[21-Jul-2022 15:41:45] Completed 79 of 208 simulation runs
[21-Jul-2022 15:41:46] Completed 80 of 208 simulation runs
[21-Jul-2022 15:41:47] Completed 81 of 208 simulation runs
[21-Jul-2022 15:41:48] Completed 82 of 208 simulation runs
[21-Jul-2022 15:41:49] Completed 83 of 208 simulation runs
[21-Jul-2022 15:41:49] Completed 84 of 208 simulation runs
[21-Jul-2022 15:41:57] Completed 85 of 208 simulation runs
[21-Jul-2022 15:41:58] Completed 86 of 208 simulation runs
[21-Jul-2022 15:41:59] Completed 87 of 208 simulation runs
[21-Jul-2022 15:42:00] Completed 88 of 208 simulation runs
[21-Jul-2022 15:42:01] Completed 89 of 208 simulation runs
[21-Jul-2022 15:42:02] Completed 90 of 208 simulation runs
[21-Jul-2022 15:42:10] Completed 91 of 208 simulation runs
[21-Jul-2022 15:42:11] Completed 92 of 208 simulation runs
[21-Jul-2022 15:42:12] Completed 93 of 208 simulation runs
[21-Jul-2022 15:42:13] Completed 94 of 208 simulation runs
[21-Jul-2022 15:42:14] Completed 95 of 208 simulation runs
[21-Jul-2022 15:42:15] Completed 96 of 208 simulation runs
[21-Jul-2022 15:42:23] Completed 97 of 208 simulation runs
[21-Jul-2022 15:42:23] Completed 98 of 208 simulation runs
[21-Jul-2022 15:42:24] Completed 99 of 208 simulation runs
[21-Jul-2022 15:42:25] Completed 100 of 208 simulation runs
[21-Jul-2022 15:42:26] Completed 101 of 208 simulation runs
[21-Jul-2022 15:42:27] Completed 102 of 208 simulation runs
[21-Jul-2022 15:42:35] Completed 103 of 208 simulation runs
[21-Jul-2022 15:42:36] Completed 104 of 208 simulation runs
[21-Jul-2022 15:42:37] Completed 105 of 208 simulation runs
[21-Jul-2022 15:42:38] Completed 106 of 208 simulation runs
[21-Jul-2022 15:42:38] Completed 107 of 208 simulation runs
[21-Jul-2022 15:42:39] Completed 108 of 208 simulation runs
[21-Jul-2022 15:42:47] Completed 109 of 208 simulation runs
[21-Jul-2022 15:42:48] Completed 110 of 208 simulation runs
[21-Jul-2022 15:42:50] Completed 111 of 208 simulation runs
[21-Jul-2022 15:42:51] Completed 112 of 208 simulation runs
[21-Jul-2022 15:42:52] Completed 113 of 208 simulation runs
[21-Jul-2022 15:42:53] Completed 114 of 208 simulation runs
[21-Jul-2022 15:43:01] Completed 115 of 208 simulation runs
[21-Jul-2022 15:43:02] Completed 116 of 208 simulation runs
[21-Jul-2022 15:43:03] Completed 117 of 208 simulation runs
[21-Jul-2022 15:43:04] Completed 118 of 208 simulation runs
[21-Jul-2022 15:43:05] Completed 119 of 208 simulation runs
[21-Jul-2022 15:43:06] Completed 120 of 208 simulation runs
[21-Jul-2022 15:43:13] Completed 121 of 208 simulation runs
[21-Jul-2022 15:43:14] Completed 122 of 208 simulation runs
[21-Jul-2022 15:43:14] Completed 123 of 208 simulation runs
[21-Jul-2022 15:43:15] Completed 124 of 208 simulation runs
```

```
[21-Jul-2022 15:43:15] Completed 125 of 208 simulation runs
[21-Jul-2022 15:43:16] Completed 126 of 208 simulation runs
[21-Jul-2022 15:43:28] Completed 127 of 208 simulation runs
[21-Jul-2022 15:43:29] Completed 128 of 208 simulation runs
[21-Jul-2022 15:43:30] Completed 129 of 208 simulation runs
[21-Jul-2022 15:43:31] Completed 130 of 208 simulation runs
[21-Jul-2022 15:43:32] Completed 131 of 208 simulation runs
[21-Jul-2022 15:43:33] Completed 132 of 208 simulation runs
[21-Jul-2022 15:43:38] Completed 133 of 208 simulation runs
[21-Jul-2022 15:43:40] Completed 134 of 208 simulation runs
[21-Jul-2022 15:43:41] Completed 135 of 208 simulation runs
[21-Jul-2022 15:43:42] Completed 136 of 208 simulation runs
[21-Jul-2022 15:43:43] Completed 137 of 208 simulation runs
[21-Jul-2022 15:43:44] Completed 138 of 208 simulation runs
[21-Jul-2022 15:43:51] Completed 139 of 208 simulation runs
[21-Jul-2022 15:43:52] Completed 140 of 208 simulation runs
[21-Jul-2022 15:43:53] Completed 141 of 208 simulation runs
[21-Jul-2022 15:43:54] Completed 142 of 208 simulation runs
[21-Jul-2022 15:43:55] Completed 143 of 208 simulation runs
[21-Jul-2022 15:43:56] Completed 144 of 208 simulation runs
[21-Jul-2022 15:44:03] Completed 145 of 208 simulation runs
[21-Jul-2022 15:44:05] Completed 146 of 208 simulation runs
[21-Jul-2022 15:44:06] Completed 147 of 208 simulation runs
[21-Jul-2022 15:44:07] Completed 148 of 208 simulation runs
[21-Jul-2022 15:44:08] Completed 149 of 208 simulation runs
[21-Jul-2022 15:44:09] Completed 150 of 208 simulation runs
[21-Jul-2022 15:44:16] Completed 151 of 208 simulation runs
[21-Jul-2022 15:44:17] Completed 152 of 208 simulation runs
[21-Jul-2022 15:44:18] Completed 153 of 208 simulation runs
[21-Jul-2022 15:44:18] Completed 154 of 208 simulation runs
[21-Jul-2022 15:44:19] Completed 155 of 208 simulation runs
[21-Jul-2022 15:44:20] Completed 156 of 208 simulation runs
[21-Jul-2022 15:44:28] Completed 157 of 208 simulation runs
[21-Jul-2022 15:44:29] Completed 158 of 208 simulation runs
[21-Jul-2022 15:44:29] Completed 159 of 208 simulation runs
[21-Jul-2022 15:44:31] Completed 160 of 208 simulation runs
[21-Jul-2022 15:44:32] Completed 161 of 208 simulation runs
[21-Jul-2022 15:44:33] Completed 162 of 208 simulation runs
[21-Jul-2022 15:44:42] Completed 163 of 208 simulation runs
[21-Jul-2022 15:44:42] Completed 164 of 208 simulation runs
[21-Jul-2022 15:44:43] Completed 165 of 208 simulation runs
[21-Jul-2022 15:44:44] Completed 166 of 208 simulation runs
[21-Jul-2022 15:44:45] Completed 167 of 208 simulation runs
[21-Jul-2022 15:44:46] Completed 168 of 208 simulation runs
[21-Jul-2022 15:44:54] Completed 169 of 208 simulation runs
[21-Jul-2022 15:44:55] Completed 170 of 208 simulation runs
[21-Jul-2022 15:44:56] Completed 171 of 208 simulation runs
[21-Jul-2022 15:44:57] Completed 172 of 208 simulation runs
[21-Jul-2022 15:44:57] Completed 173 of 208 simulation runs
[21-Jul-2022 15:44:58] Completed 174 of 208 simulation runs
[21-Jul-2022 15:45:07] Completed 175 of 208 simulation runs
[21-Jul-2022 15:45:08] Completed 176 of 208 simulation runs
[21-Jul-2022 15:45:08] Completed 177 of 208 simulation runs
[21-Jul-2022 15:45:09] Completed 178 of 208 simulation runs
[21-Jul-2022 15:45:10] Completed 179 of 208 simulation runs
[21-Jul-2022 15:45:11] Completed 180 of 208 simulation runs
[21-Jul-2022 15:45:20] Completed 181 of 208 simulation runs
[21-Jul-2022 15:45:21] Completed 182 of 208 simulation runs
```

```
[21-Jul-2022 15:45:22] Completed 183 of 208 simulation runs
[21-Jul-2022 15:45:23] Completed 184 of 208 simulation runs
[21-Jul-2022 15:45:24] Completed 185 of 208 simulation runs
[21-Jul-2022 15:45:25] Completed 186 of 208 simulation runs
[21-Jul-2022 15:45:32] Completed 187 of 208 simulation runs
[21-Jul-2022 15:45:33] Completed 188 of 208 simulation runs
[21-Jul-2022 15:45:34] Completed 189 of 208 simulation runs
[21-Jul-2022 15:45:35] Completed 190 of 208 simulation runs
[21-Jul-2022 15:45:36] Completed 191 of 208 simulation runs
[21-Jul-2022 15:45:37] Completed 192 of 208 simulation runs
[21-Jul-2022 15:45:45] Completed 193 of 208 simulation runs
[21-Jul-2022 15:45:45] Completed 194 of 208 simulation runs
[21-Jul-2022 15:45:46] Completed 195 of 208 simulation runs
[21-Jul-2022 15:45:47] Completed 196 of 208 simulation runs
[21-Jul-2022 15:45:48] Completed 197 of 208 simulation runs
[21-Jul-2022 15:45:49] Completed 198 of 208 simulation runs
[21-Jul-2022 15:45:57] Completed 199 of 208 simulation runs
[21-Jul-2022 15:45:58] Completed 200 of 208 simulation runs
[21-Jul-2022 15:45:59] Completed 201 of 208 simulation runs
[21-Jul-2022 15:46:00] Completed 202 of 208 simulation runs
[21-Jul-2022 15:46:01] Completed 203 of 208 simulation runs
[21-Jul-2022 15:46:02] Completed 204 of 208 simulation runs
[21-Jul-2022 15:46:09] Completed 205 of 208 simulation runs
[21-Jul-2022 15:46:10] Completed 206 of 208 simulation runs
[21-Jul-2022 15:46:11] Completed 207 of 208 simulation runs
[21-Jul-2022 15:46:12] Completed 208 of 208 simulation runs
[21-Jul-2022 15:46:12] 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]
                ReadSize: 1
              NumMembers: 208
          LastMemberRead: [0×0 string]
                   Files: [208×1 string]
```

```
ens.SelectedVariables
```

```
ans = 6×1 string
    "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')
```
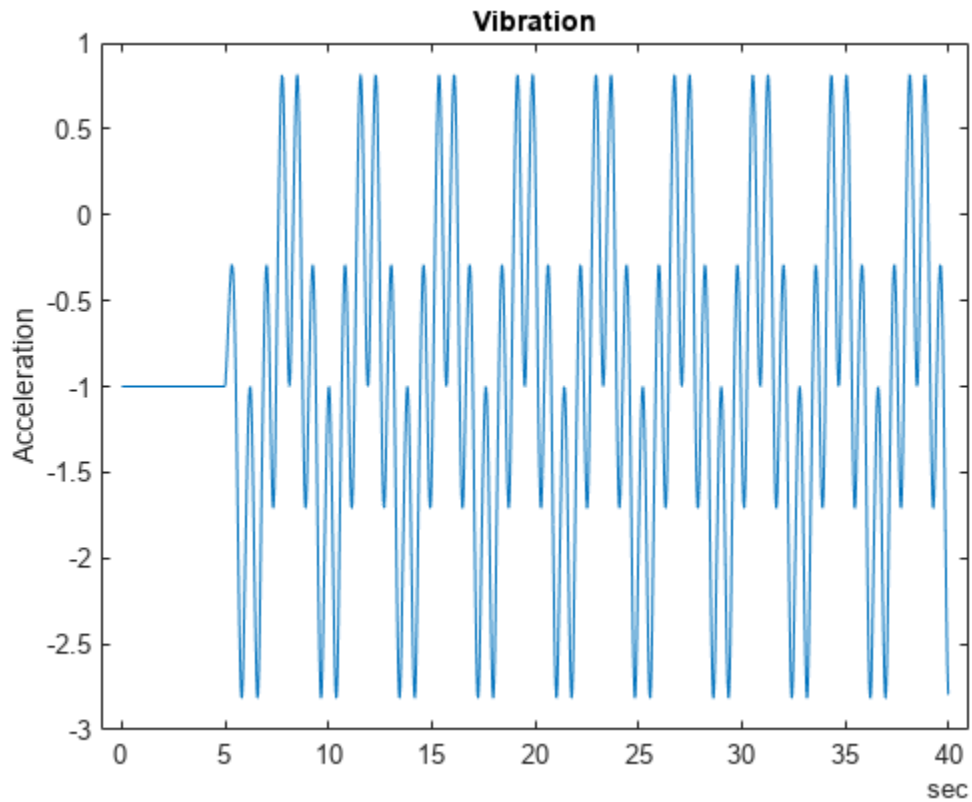
The first 10 seconds of the simulation contains data where the transmission system is starting up; for analysis discard this data.

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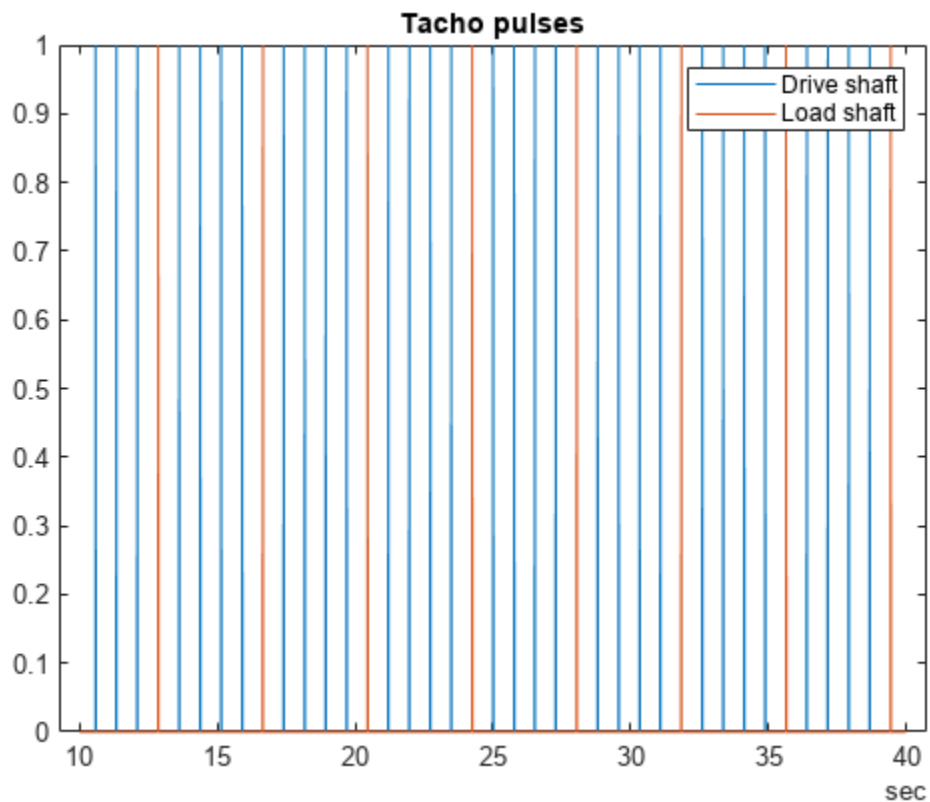
```
idx = diff(tacho.Data(:,2)) > 0.5;
tachoPulses = tacho.Time(find(idx)+1)-tacho.Time(1)
```

tachoPulses = *8×1 duration*
```
    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','FaultCode
```

```
sdata=1×6 table
        Vibration            TachoPulses      SensorDrift    ShaftWear    ToothFault    FaultCode
    _____    _____    _____    _____    _____    _____

    {30106×1 timetable}     {8×1 duration}       true          false         false          1
```

```
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')
```
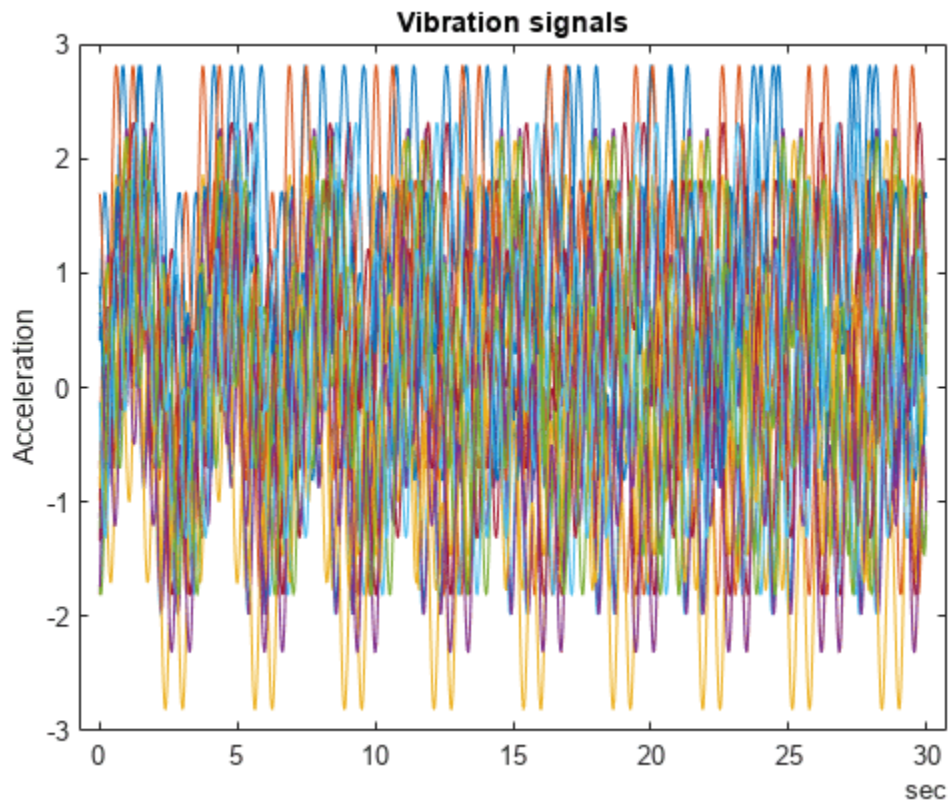

Analyzing the Simulation Data

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.Time,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 can take up to an hour 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";"SigApproxEntropy
    "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
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))
```

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 predictor
featureAnalysis = fscnca(featureData{:,1:idxLastFeature},featureData.SensorDrift);
featureAnalysis.FeatureWeights
idxSelectedFeature = featureAnalysis.FeatureWeights > 0.1;
classifySD = [featureData(:,idxSelectedFeature), featureData(:,idxResponse)]
```

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

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)]
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 training and v
```

```
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
```
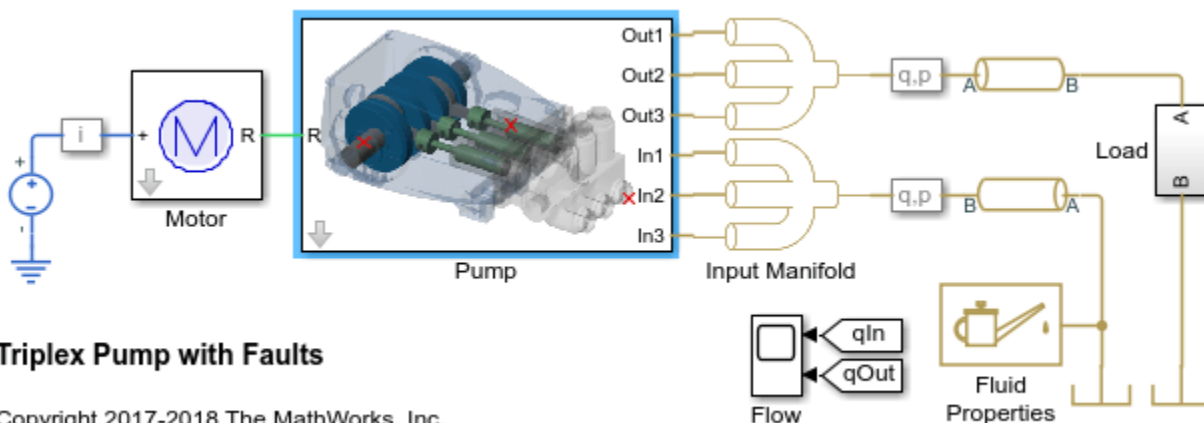
```
Generating Simulink library 'mech_hydro_forcesPS_lib' in the current directory 'L:\misc\ExampleM
```

Reciprocating Pump Model

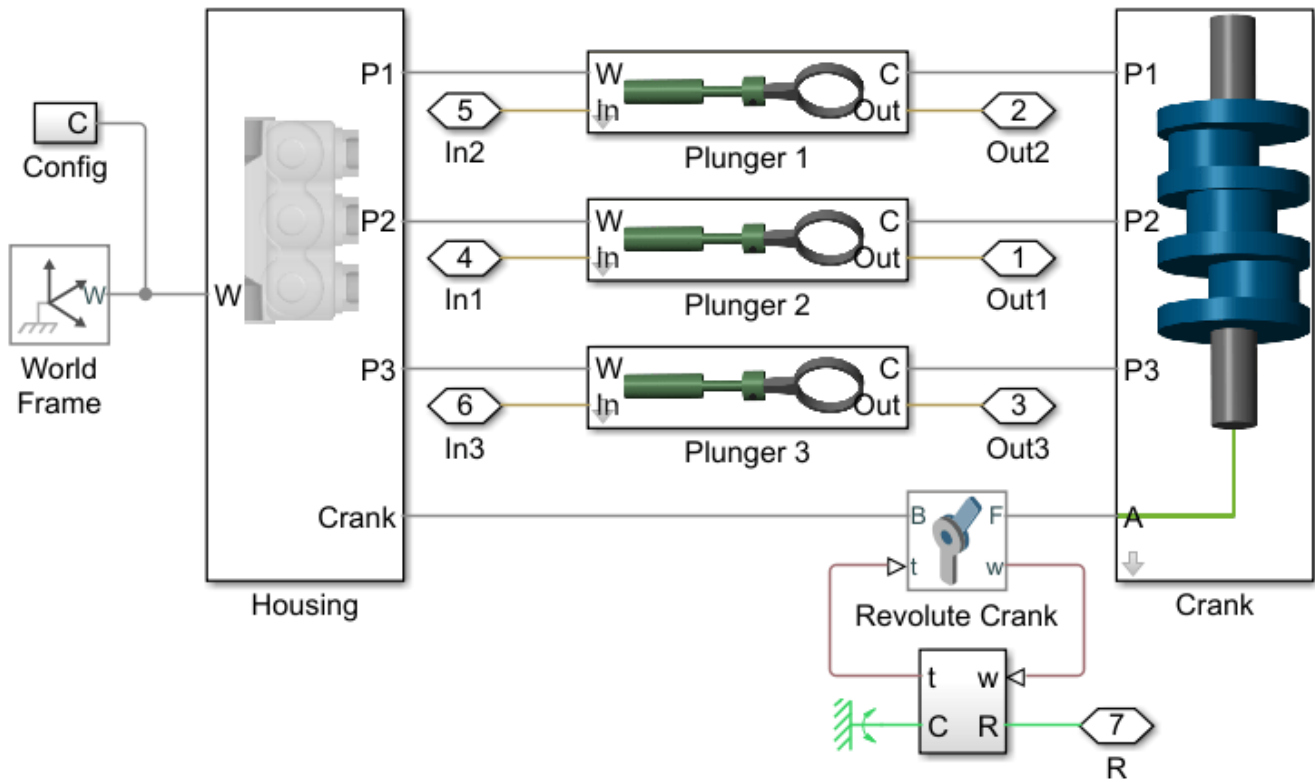The reciprocating pump consists of an electric motor, the pump housing, pump crank and pump plungers.

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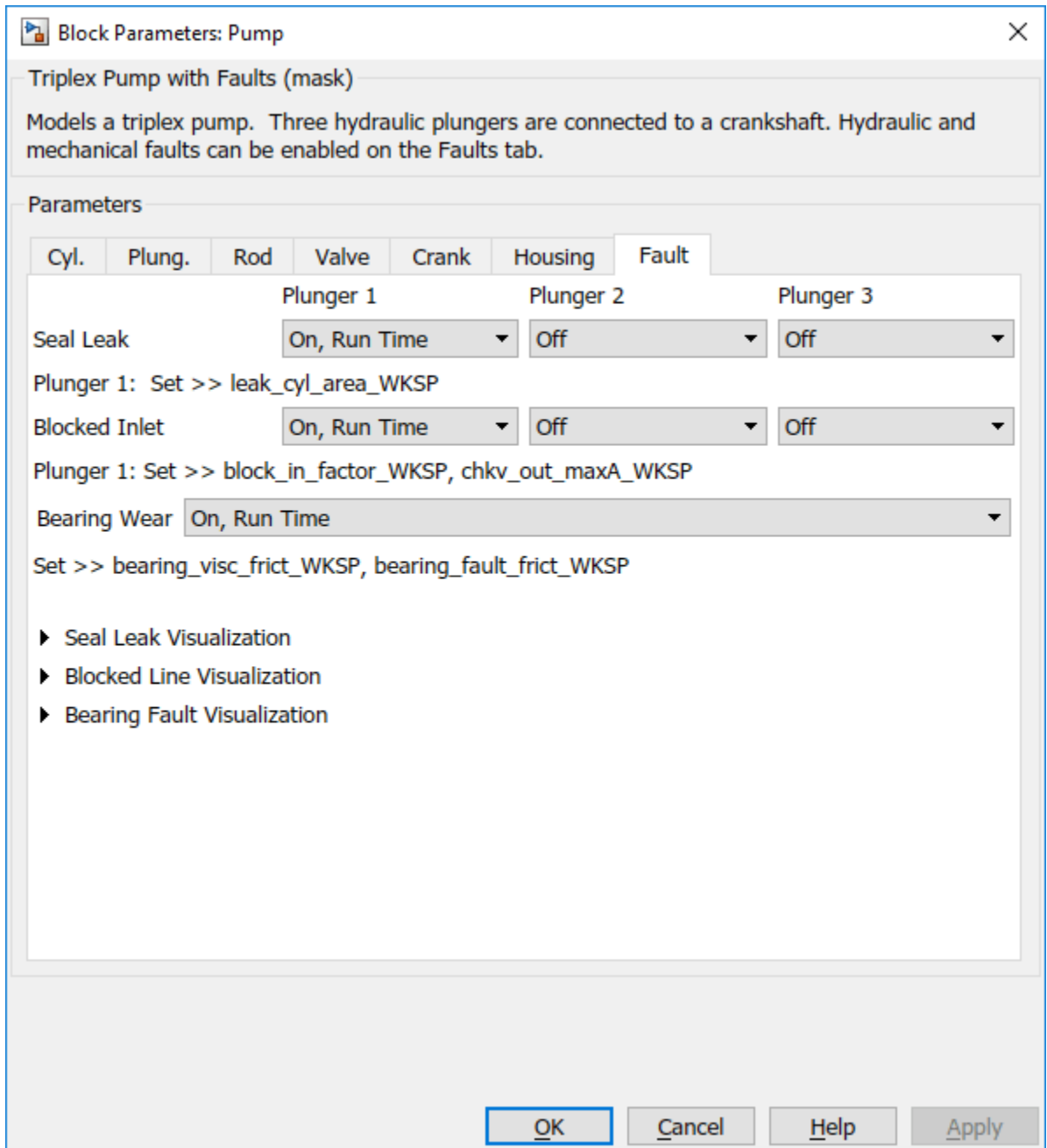


```
open_system([mdl,'/Pump'])
```

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.

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. Note that running these 1000 simulations in parallel takes around an hour on a standard desktop and generates around 620MB of data. To reduce simulation time, reduce the number of fault combinations to 20 by changing `runAll = true` to `runAll = false`. Note that a larger dataset results in a more robust classifier.

```
% Set number of elements in each fault group
runAll = true;
if runAll
    % Create a large dataset to build a robust classifier
    nPerGroup = 100;
else
    % Create a smaller dataset to reduce simulation time
    nPerGroup = 20; %#ok<UNRCH>
end

rng('default');    % Feed default seed to rng (Random number generator)

% 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(ct));
    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.

```
% Run the simulation and create an ensemble to manage the simulation
% results

if isfolder('./Data')
    % Delete existing mat files
    delete('./Data/*.mat')
end

[ok,e] = generateSimulationEnsemble(simInput,fullfile('.','Data'),'UseParallel',true);

[28-Jun-2021 13:38:31] Checking for availability of parallel pool...
Starting parallel pool (parpool) using the 'local' profile ...
Preserving jobs with IDs: 1 because they contain crash dump files.
You can use 'delete(myCluster.Jobs)' to remove all jobs created with profile local. To create 'my
Connected to the parallel pool (number of workers: 6).
[28-Jun-2021 13:39:52] Starting Simulink on parallel workers...
[28-Jun-2021 13:41:17] Configuring simulation cache folder on parallel workers...
[28-Jun-2021 13:41:18] Transferring base workspace variables used in the model to parallel worker
[28-Jun-2021 13:41:19] Loading model on parallel workers...
[28-Jun-2021 13:41:43] Running simulations...
[28-Jun-2021 13:43:07] Completed 1 of 800 simulation runs
[28-Jun-2021 13:43:07] Completed 2 of 800 simulation runs
[28-Jun-2021 13:43:08] Completed 3 of 800 simulation runs
[28-Jun-2021 13:43:08] Completed 4 of 800 simulation runs
[28-Jun-2021 13:43:09] Completed 5 of 800 simulation runs
[28-Jun-2021 13:43:09] Completed 6 of 800 simulation runs
[28-Jun-2021 13:43:32] Completed 7 of 800 simulation runs
[28-Jun-2021 13:43:33] Completed 8 of 800 simulation runs
[28-Jun-2021 13:43:33] Completed 9 of 800 simulation runs
```

```
[28-Jun-2021 13:43:34] Completed 10 of 800 simulation runs
[28-Jun-2021 13:43:34] Completed 11 of 800 simulation runs
[28-Jun-2021 13:43:35] Completed 12 of 800 simulation runs
[28-Jun-2021 13:43:58] Completed 13 of 800 simulation runs
[28-Jun-2021 13:43:59] Completed 14 of 800 simulation runs
[28-Jun-2021 13:43:59] Completed 15 of 800 simulation runs
[28-Jun-2021 13:44:00] Completed 16 of 800 simulation runs
[28-Jun-2021 13:44:00] Completed 17 of 800 simulation runs
[28-Jun-2021 13:44:01] Completed 18 of 800 simulation runs
[28-Jun-2021 13:44:24] Completed 19 of 800 simulation runs
[28-Jun-2021 13:44:24] Completed 20 of 800 simulation runs
[28-Jun-2021 13:44:25] Completed 21 of 800 simulation runs
[28-Jun-2021 13:44:26] Completed 22 of 800 simulation runs
[28-Jun-2021 13:44:26] Completed 23 of 800 simulation runs
[28-Jun-2021 13:44:27] Completed 24 of 800 simulation runs
[28-Jun-2021 13:44:49] Completed 25 of 800 simulation runs
[28-Jun-2021 13:44:50] Completed 26 of 800 simulation runs
[28-Jun-2021 13:44:50] Completed 27 of 800 simulation runs
[28-Jun-2021 13:44:51] Completed 28 of 800 simulation runs
[28-Jun-2021 13:44:51] Completed 29 of 800 simulation runs
[28-Jun-2021 13:44:52] Completed 30 of 800 simulation runs
[28-Jun-2021 13:45:15] Completed 31 of 800 simulation runs
[28-Jun-2021 13:45:16] Completed 32 of 800 simulation runs
[28-Jun-2021 13:45:16] Completed 33 of 800 simulation runs
[28-Jun-2021 13:45:17] Completed 34 of 800 simulation runs
[28-Jun-2021 13:45:17] Completed 35 of 800 simulation runs
[28-Jun-2021 13:45:18] Completed 36 of 800 simulation runs
[28-Jun-2021 13:45:41] Completed 37 of 800 simulation runs
[28-Jun-2021 13:45:42] Completed 38 of 800 simulation runs
[28-Jun-2021 13:45:42] Completed 39 of 800 simulation runs
[28-Jun-2021 13:45:42] Completed 40 of 800 simulation runs
[28-Jun-2021 13:45:43] Completed 41 of 800 simulation runs
[28-Jun-2021 13:45:43] Completed 42 of 800 simulation runs
[28-Jun-2021 13:46:11] Completed 43 of 800 simulation runs
[28-Jun-2021 13:46:11] Completed 44 of 800 simulation runs
[28-Jun-2021 13:46:12] Completed 45 of 800 simulation runs
[28-Jun-2021 13:46:12] Completed 46 of 800 simulation runs
[28-Jun-2021 13:46:13] Completed 47 of 800 simulation runs
[28-Jun-2021 13:46:13] Completed 48 of 800 simulation runs
[28-Jun-2021 13:46:32] Completed 49 of 800 simulation runs
[28-Jun-2021 13:46:36] Completed 50 of 800 simulation runs
[28-Jun-2021 13:46:36] Completed 51 of 800 simulation runs
[28-Jun-2021 13:46:37] Completed 52 of 800 simulation runs
[28-Jun-2021 13:46:37] Completed 53 of 800 simulation runs
[28-Jun-2021 13:46:38] Completed 54 of 800 simulation runs
[28-Jun-2021 13:46:58] Completed 55 of 800 simulation runs
[28-Jun-2021 13:47:00] Completed 56 of 800 simulation runs
[28-Jun-2021 13:47:01] Completed 57 of 800 simulation runs
[28-Jun-2021 13:47:01] Completed 58 of 800 simulation runs
[28-Jun-2021 13:47:02] Completed 59 of 800 simulation runs
[28-Jun-2021 13:47:02] Completed 60 of 800 simulation runs
[28-Jun-2021 13:47:25] Completed 61 of 800 simulation runs
[28-Jun-2021 13:47:26] Completed 62 of 800 simulation runs
[28-Jun-2021 13:47:26] Completed 63 of 800 simulation runs
[28-Jun-2021 13:47:27] Completed 64 of 800 simulation runs
[28-Jun-2021 13:47:27] Completed 65 of 800 simulation runs
[28-Jun-2021 13:47:28] Completed 66 of 800 simulation runs
[28-Jun-2021 13:47:50] Completed 67 of 800 simulation runs
```

```
[28-Jun-2021 13:47:51] Completed 68 of 800 simulation runs
[28-Jun-2021 13:47:51] Completed 69 of 800 simulation runs
[28-Jun-2021 13:47:52] Completed 70 of 800 simulation runs
[28-Jun-2021 13:47:52] Completed 71 of 800 simulation runs
[28-Jun-2021 13:47:53] Completed 72 of 800 simulation runs
[28-Jun-2021 13:48:14] Completed 73 of 800 simulation runs
[28-Jun-2021 13:48:14] Completed 74 of 800 simulation runs
[28-Jun-2021 13:48:15] Completed 75 of 800 simulation runs
[28-Jun-2021 13:48:15] Completed 76 of 800 simulation runs
[28-Jun-2021 13:48:16] Completed 77 of 800 simulation runs
[28-Jun-2021 13:48:17] Completed 78 of 800 simulation runs
[28-Jun-2021 13:48:37] Completed 79 of 800 simulation runs
[28-Jun-2021 13:48:38] Completed 80 of 800 simulation runs
[28-Jun-2021 13:48:38] Completed 81 of 800 simulation runs
[28-Jun-2021 13:48:39] Completed 82 of 800 simulation runs
[28-Jun-2021 13:48:40] Completed 83 of 800 simulation runs
[28-Jun-2021 13:48:40] Completed 84 of 800 simulation runs
[28-Jun-2021 13:49:01] Completed 85 of 800 simulation runs
[28-Jun-2021 13:49:02] Completed 86 of 800 simulation runs
[28-Jun-2021 13:49:02] Completed 87 of 800 simulation runs
[28-Jun-2021 13:49:03] Completed 88 of 800 simulation runs
[28-Jun-2021 13:49:03] Completed 89 of 800 simulation runs
[28-Jun-2021 13:49:04] Completed 90 of 800 simulation runs
[28-Jun-2021 13:49:24] Completed 91 of 800 simulation runs
[28-Jun-2021 13:49:25] Completed 92 of 800 simulation runs
[28-Jun-2021 13:49:25] Completed 93 of 800 simulation runs
[28-Jun-2021 13:49:26] Completed 94 of 800 simulation runs
[28-Jun-2021 13:49:26] Completed 95 of 800 simulation runs
[28-Jun-2021 13:49:27] Completed 96 of 800 simulation runs
[28-Jun-2021 13:49:48] Completed 97 of 800 simulation runs
[28-Jun-2021 13:49:48] Completed 98 of 800 simulation runs
[28-Jun-2021 13:49:49] Completed 99 of 800 simulation runs
[28-Jun-2021 13:49:49] Completed 100 of 800 simulation runs
[28-Jun-2021 13:49:50] Completed 101 of 800 simulation runs
[28-Jun-2021 13:49:50] Completed 102 of 800 simulation runs
[28-Jun-2021 13:50:11] Completed 103 of 800 simulation runs
[28-Jun-2021 13:50:12] Completed 104 of 800 simulation runs
[28-Jun-2021 13:50:12] Completed 105 of 800 simulation runs
[28-Jun-2021 13:50:13] Completed 106 of 800 simulation runs
[28-Jun-2021 13:50:13] Completed 107 of 800 simulation runs
[28-Jun-2021 13:50:14] Completed 108 of 800 simulation runs
[28-Jun-2021 13:50:33] Completed 109 of 800 simulation runs
[28-Jun-2021 13:50:34] Completed 110 of 800 simulation runs
[28-Jun-2021 13:50:34] Completed 111 of 800 simulation runs
[28-Jun-2021 13:50:35] Completed 112 of 800 simulation runs
[28-Jun-2021 13:50:35] Completed 113 of 800 simulation runs
[28-Jun-2021 13:50:36] Completed 114 of 800 simulation runs
[28-Jun-2021 13:50:55] Completed 115 of 800 simulation runs
[28-Jun-2021 13:50:56] Completed 116 of 800 simulation runs
[28-Jun-2021 13:50:57] Completed 117 of 800 simulation runs
[28-Jun-2021 13:50:57] Completed 118 of 800 simulation runs
[28-Jun-2021 13:50:58] Completed 119 of 800 simulation runs
[28-Jun-2021 13:50:59] Completed 120 of 800 simulation runs
[28-Jun-2021 13:51:18] Completed 121 of 800 simulation runs
[28-Jun-2021 13:51:19] Completed 122 of 800 simulation runs
[28-Jun-2021 13:51:19] Completed 123 of 800 simulation runs
[28-Jun-2021 13:51:20] Completed 124 of 800 simulation runs
[28-Jun-2021 13:51:21] Completed 125 of 800 simulation runs
```

```
[28-Jun-2021 13:51:21] Completed 126 of 800 simulation runs
[28-Jun-2021 13:51:42] Completed 127 of 800 simulation runs
[28-Jun-2021 13:51:43] Completed 128 of 800 simulation runs
[28-Jun-2021 13:51:44] Completed 129 of 800 simulation runs
[28-Jun-2021 13:51:44] Completed 130 of 800 simulation runs
[28-Jun-2021 13:51:45] Completed 131 of 800 simulation runs
[28-Jun-2021 13:51:46] Completed 132 of 800 simulation runs
[28-Jun-2021 13:52:06] Completed 133 of 800 simulation runs
[28-Jun-2021 13:52:07] Completed 134 of 800 simulation runs
[28-Jun-2021 13:52:07] Completed 135 of 800 simulation runs
[28-Jun-2021 13:52:08] Completed 136 of 800 simulation runs
[28-Jun-2021 13:52:09] Completed 137 of 800 simulation runs
[28-Jun-2021 13:52:09] Completed 138 of 800 simulation runs
[28-Jun-2021 13:52:29] Completed 139 of 800 simulation runs
[28-Jun-2021 13:52:30] Completed 140 of 800 simulation runs
[28-Jun-2021 13:52:31] Completed 141 of 800 simulation runs
[28-Jun-2021 13:52:31] Completed 142 of 800 simulation runs
[28-Jun-2021 13:52:32] Completed 143 of 800 simulation runs
[28-Jun-2021 13:52:33] Completed 144 of 800 simulation runs
[28-Jun-2021 13:52:52] Completed 145 of 800 simulation runs
[28-Jun-2021 13:52:53] Completed 146 of 800 simulation runs
[28-Jun-2021 13:52:54] Completed 147 of 800 simulation runs
[28-Jun-2021 13:52:54] Completed 148 of 800 simulation runs
[28-Jun-2021 13:52:55] Completed 149 of 800 simulation runs
[28-Jun-2021 13:52:56] Completed 150 of 800 simulation runs
[28-Jun-2021 13:53:15] Completed 151 of 800 simulation runs
[28-Jun-2021 13:53:16] Completed 152 of 800 simulation runs
[28-Jun-2021 13:53:16] Completed 153 of 800 simulation runs
[28-Jun-2021 13:53:17] Completed 154 of 800 simulation runs
[28-Jun-2021 13:53:18] Completed 155 of 800 simulation runs
[28-Jun-2021 13:53:18] Completed 156 of 800 simulation runs
[28-Jun-2021 13:53:37] Completed 157 of 800 simulation runs
[28-Jun-2021 13:53:38] Completed 158 of 800 simulation runs
[28-Jun-2021 13:53:39] Completed 159 of 800 simulation runs
[28-Jun-2021 13:53:39] Completed 160 of 800 simulation runs
[28-Jun-2021 13:53:40] Completed 161 of 800 simulation runs
[28-Jun-2021 13:53:41] Completed 162 of 800 simulation runs
[28-Jun-2021 13:54:00] Completed 163 of 800 simulation runs
[28-Jun-2021 13:54:01] Completed 164 of 800 simulation runs
[28-Jun-2021 13:54:01] Completed 165 of 800 simulation runs
[28-Jun-2021 13:54:02] Completed 166 of 800 simulation runs
[28-Jun-2021 13:54:03] Completed 167 of 800 simulation runs
[28-Jun-2021 13:54:04] Completed 168 of 800 simulation runs
[28-Jun-2021 13:54:23] Completed 169 of 800 simulation runs
[28-Jun-2021 13:54:24] Completed 170 of 800 simulation runs
[28-Jun-2021 13:54:25] Completed 171 of 800 simulation runs
[28-Jun-2021 13:54:26] Completed 172 of 800 simulation runs
[28-Jun-2021 13:54:26] Completed 173 of 800 simulation runs
[28-Jun-2021 13:54:27] Completed 174 of 800 simulation runs
[28-Jun-2021 13:54:46] Completed 175 of 800 simulation runs
[28-Jun-2021 13:54:47] Completed 176 of 800 simulation runs
[28-Jun-2021 13:54:48] Completed 177 of 800 simulation runs
[28-Jun-2021 13:54:48] Completed 178 of 800 simulation runs
[28-Jun-2021 13:54:49] Completed 179 of 800 simulation runs
[28-Jun-2021 13:54:50] Completed 180 of 800 simulation runs
[28-Jun-2021 13:55:09] Completed 181 of 800 simulation runs
[28-Jun-2021 13:55:10] Completed 182 of 800 simulation runs
[28-Jun-2021 13:55:11] Completed 183 of 800 simulation runs
```

```
[28-Jun-2021 13:55:11] Completed 184 of 800 simulation runs
[28-Jun-2021 13:55:12] Completed 185 of 800 simulation runs
[28-Jun-2021 13:55:13] Completed 186 of 800 simulation runs
[28-Jun-2021 13:55:32] Completed 187 of 800 simulation runs
[28-Jun-2021 13:55:33] Completed 188 of 800 simulation runs
[28-Jun-2021 13:55:34] Completed 189 of 800 simulation runs
[28-Jun-2021 13:55:34] Completed 190 of 800 simulation runs
[28-Jun-2021 13:55:35] Completed 191 of 800 simulation runs
[28-Jun-2021 13:55:36] Completed 192 of 800 simulation runs
[28-Jun-2021 13:55:56] Completed 193 of 800 simulation runs
[28-Jun-2021 13:55:56] Completed 194 of 800 simulation runs
[28-Jun-2021 13:55:57] Completed 195 of 800 simulation runs
[28-Jun-2021 13:55:58] Completed 196 of 800 simulation runs
[28-Jun-2021 13:55:59] Completed 197 of 800 simulation runs
[28-Jun-2021 13:55:59] Completed 198 of 800 simulation runs
[28-Jun-2021 13:56:19] Completed 199 of 800 simulation runs
[28-Jun-2021 13:56:20] Completed 200 of 800 simulation runs
[28-Jun-2021 13:56:21] Completed 201 of 800 simulation runs
[28-Jun-2021 13:56:22] Completed 202 of 800 simulation runs
[28-Jun-2021 13:56:23] Completed 203 of 800 simulation runs
[28-Jun-2021 13:56:24] Completed 204 of 800 simulation runs
[28-Jun-2021 13:56:44] Completed 205 of 800 simulation runs
[28-Jun-2021 13:56:46] Completed 206 of 800 simulation runs
[28-Jun-2021 13:56:47] Completed 207 of 800 simulation runs
[28-Jun-2021 13:56:48] Completed 208 of 800 simulation runs
[28-Jun-2021 13:56:49] Completed 209 of 800 simulation runs
[28-Jun-2021 13:56:50] Completed 210 of 800 simulation runs
[28-Jun-2021 13:57:10] Completed 211 of 800 simulation runs
[28-Jun-2021 13:57:11] Completed 212 of 800 simulation runs
[28-Jun-2021 13:57:12] Completed 213 of 800 simulation runs
[28-Jun-2021 13:57:13] Completed 214 of 800 simulation runs
[28-Jun-2021 13:57:14] Completed 215 of 800 simulation runs
[28-Jun-2021 13:57:14] Completed 216 of 800 simulation runs
[28-Jun-2021 13:57:35] Completed 217 of 800 simulation runs
[28-Jun-2021 13:57:36] Completed 218 of 800 simulation runs
[28-Jun-2021 13:57:37] Completed 219 of 800 simulation runs
[28-Jun-2021 13:57:38] Completed 220 of 800 simulation runs
[28-Jun-2021 13:57:39] Completed 221 of 800 simulation runs
[28-Jun-2021 13:57:40] Completed 222 of 800 simulation runs
[28-Jun-2021 13:57:56] Completed 223 of 800 simulation runs
[28-Jun-2021 13:57:59] Completed 224 of 800 simulation runs
[28-Jun-2021 13:58:01] Completed 225 of 800 simulation runs
[28-Jun-2021 13:58:02] Completed 226 of 800 simulation runs
[28-Jun-2021 13:58:03] Completed 227 of 800 simulation runs
[28-Jun-2021 13:58:04] Completed 228 of 800 simulation runs
[28-Jun-2021 13:58:21] Completed 229 of 800 simulation runs
[28-Jun-2021 13:58:26] Completed 230 of 800 simulation runs
[28-Jun-2021 13:58:27] Completed 231 of 800 simulation runs
[28-Jun-2021 13:58:28] Completed 232 of 800 simulation runs
[28-Jun-2021 13:58:29] Completed 233 of 800 simulation runs
[28-Jun-2021 13:58:30] Completed 234 of 800 simulation runs
[28-Jun-2021 13:58:45] Completed 235 of 800 simulation runs
[28-Jun-2021 13:58:46] Completed 236 of 800 simulation runs
[28-Jun-2021 13:58:50] Completed 237 of 800 simulation runs
[28-Jun-2021 13:58:51] Completed 238 of 800 simulation runs
[28-Jun-2021 13:58:52] Completed 239 of 800 simulation runs
[28-Jun-2021 13:58:53] Completed 240 of 800 simulation runs
[28-Jun-2021 13:59:11] Completed 241 of 800 simulation runs
```

```
[28-Jun-2021 13:59:12] Completed 242 of 800 simulation runs
[28-Jun-2021 13:59:14] Completed 243 of 800 simulation runs
[28-Jun-2021 13:59:15] Completed 244 of 800 simulation runs
[28-Jun-2021 13:59:16] Completed 245 of 800 simulation runs
[28-Jun-2021 13:59:17] Completed 246 of 800 simulation runs
[28-Jun-2021 13:59:35] Completed 247 of 800 simulation runs
[28-Jun-2021 13:59:36] Completed 248 of 800 simulation runs
[28-Jun-2021 13:59:37] Completed 249 of 800 simulation runs
[28-Jun-2021 13:59:38] Completed 250 of 800 simulation runs
[28-Jun-2021 13:59:39] Completed 251 of 800 simulation runs
[28-Jun-2021 13:59:40] Completed 252 of 800 simulation runs
[28-Jun-2021 14:00:02] Completed 253 of 800 simulation runs
[28-Jun-2021 14:00:03] Completed 254 of 800 simulation runs
[28-Jun-2021 14:00:04] Completed 255 of 800 simulation runs
[28-Jun-2021 14:00:05] Completed 256 of 800 simulation runs
[28-Jun-2021 14:00:06] Completed 257 of 800 simulation runs
[28-Jun-2021 14:00:07] Completed 258 of 800 simulation runs
[28-Jun-2021 14:00:26] Completed 259 of 800 simulation runs
[28-Jun-2021 14:00:27] Completed 260 of 800 simulation runs
[28-Jun-2021 14:00:28] Completed 261 of 800 simulation runs
[28-Jun-2021 14:00:29] Completed 262 of 800 simulation runs
[28-Jun-2021 14:00:30] Completed 263 of 800 simulation runs
[28-Jun-2021 14:00:31] Completed 264 of 800 simulation runs
[28-Jun-2021 14:00:49] Completed 265 of 800 simulation runs
[28-Jun-2021 14:00:50] Completed 266 of 800 simulation runs
[28-Jun-2021 14:00:53] Completed 267 of 800 simulation runs
[28-Jun-2021 14:00:54] Completed 268 of 800 simulation runs
[28-Jun-2021 14:00:55] Completed 269 of 800 simulation runs
[28-Jun-2021 14:00:56] Completed 270 of 800 simulation runs
[28-Jun-2021 14:01:13] Completed 271 of 800 simulation runs
[28-Jun-2021 14:01:14] Completed 272 of 800 simulation runs
[28-Jun-2021 14:01:18] Completed 273 of 800 simulation runs
[28-Jun-2021 14:01:19] Completed 274 of 800 simulation runs
[28-Jun-2021 14:01:20] Completed 275 of 800 simulation runs
[28-Jun-2021 14:01:21] Completed 276 of 800 simulation runs
[28-Jun-2021 14:01:39] Completed 277 of 800 simulation runs
[28-Jun-2021 14:01:40] Completed 278 of 800 simulation runs
[28-Jun-2021 14:01:41] Completed 279 of 800 simulation runs
[28-Jun-2021 14:01:42] Completed 280 of 800 simulation runs
[28-Jun-2021 14:01:43] Completed 281 of 800 simulation runs
[28-Jun-2021 14:01:44] Completed 282 of 800 simulation runs
[28-Jun-2021 14:02:05] Completed 283 of 800 simulation runs
[28-Jun-2021 14:02:06] Completed 284 of 800 simulation runs
[28-Jun-2021 14:02:07] Completed 285 of 800 simulation runs
[28-Jun-2021 14:02:08] Completed 286 of 800 simulation runs
[28-Jun-2021 14:02:09] Completed 287 of 800 simulation runs
[28-Jun-2021 14:02:10] Completed 288 of 800 simulation runs
[28-Jun-2021 14:02:28] Completed 289 of 800 simulation runs
[28-Jun-2021 14:02:29] Completed 290 of 800 simulation runs
[28-Jun-2021 14:02:30] Completed 291 of 800 simulation runs
[28-Jun-2021 14:02:32] Completed 292 of 800 simulation runs
[28-Jun-2021 14:02:33] Completed 293 of 800 simulation runs
[28-Jun-2021 14:02:34] Completed 294 of 800 simulation runs
[28-Jun-2021 14:02:51] Completed 295 of 800 simulation runs
[28-Jun-2021 14:02:52] Completed 296 of 800 simulation runs
[28-Jun-2021 14:02:53] Completed 297 of 800 simulation runs
[28-Jun-2021 14:02:54] Completed 298 of 800 simulation runs
[28-Jun-2021 14:02:56] Completed 299 of 800 simulation runs
```

```
[28-Jun-2021 14:02:57] Completed 300 of 800 simulation runs
[28-Jun-2021 14:03:07] Completed 301 of 800 simulation runs
[28-Jun-2021 14:03:10] Completed 302 of 800 simulation runs
[28-Jun-2021 14:03:10] Completed 303 of 800 simulation runs
[28-Jun-2021 14:03:11] Completed 304 of 800 simulation runs
[28-Jun-2021 14:03:13] Completed 305 of 800 simulation runs
[28-Jun-2021 14:03:14] Completed 306 of 800 simulation runs
[28-Jun-2021 14:03:21] Completed 307 of 800 simulation runs
[28-Jun-2021 14:03:23] Completed 308 of 800 simulation runs
[28-Jun-2021 14:03:24] Completed 309 of 800 simulation runs
[28-Jun-2021 14:03:25] Completed 310 of 800 simulation runs
[28-Jun-2021 14:03:28] Completed 311 of 800 simulation runs
[28-Jun-2021 14:03:30] Completed 312 of 800 simulation runs
[28-Jun-2021 14:03:38] Completed 313 of 800 simulation runs
[28-Jun-2021 14:03:41] Completed 314 of 800 simulation runs
[28-Jun-2021 14:03:42] Completed 315 of 800 simulation runs
[28-Jun-2021 14:03:45] Completed 316 of 800 simulation runs
[28-Jun-2021 14:03:46] Completed 317 of 800 simulation runs
[28-Jun-2021 14:03:51] Completed 318 of 800 simulation runs
[28-Jun-2021 14:03:57] Completed 319 of 800 simulation runs
[28-Jun-2021 14:03:58] Completed 320 of 800 simulation runs
[28-Jun-2021 14:03:59] Completed 321 of 800 simulation runs
[28-Jun-2021 14:04:00] Completed 322 of 800 simulation runs
[28-Jun-2021 14:04:02] Completed 323 of 800 simulation runs
[28-Jun-2021 14:04:12] Completed 324 of 800 simulation runs
[28-Jun-2021 14:04:13] Completed 325 of 800 simulation runs
[28-Jun-2021 14:04:14] Completed 326 of 800 simulation runs
[28-Jun-2021 14:04:15] Completed 327 of 800 simulation runs
[28-Jun-2021 14:04:19] Completed 328 of 800 simulation runs
[28-Jun-2021 14:04:20] Completed 329 of 800 simulation runs
[28-Jun-2021 14:04:23] Completed 330 of 800 simulation runs
[28-Jun-2021 14:04:26] Completed 331 of 800 simulation runs
[28-Jun-2021 14:04:30] Completed 332 of 800 simulation runs
[28-Jun-2021 14:04:31] Completed 333 of 800 simulation runs
[28-Jun-2021 14:04:33] Completed 334 of 800 simulation runs
[28-Jun-2021 14:04:38] Completed 335 of 800 simulation runs
[28-Jun-2021 14:04:44] Completed 336 of 800 simulation runs
[28-Jun-2021 14:04:45] Completed 337 of 800 simulation runs
[28-Jun-2021 14:04:47] Completed 338 of 800 simulation runs
[28-Jun-2021 14:04:48] Completed 339 of 800 simulation runs
[28-Jun-2021 14:04:49] Completed 340 of 800 simulation runs
[28-Jun-2021 14:05:01] Completed 341 of 800 simulation runs
[28-Jun-2021 14:05:03] Completed 342 of 800 simulation runs
[28-Jun-2021 14:05:04] Completed 343 of 800 simulation runs
[28-Jun-2021 14:05:09] Completed 344 of 800 simulation runs
[28-Jun-2021 14:05:11] Completed 345 of 800 simulation runs
[28-Jun-2021 14:05:12] Completed 346 of 800 simulation runs
[28-Jun-2021 14:05:14] Completed 347 of 800 simulation runs
[28-Jun-2021 14:05:16] Completed 348 of 800 simulation runs
[28-Jun-2021 14:05:19] Completed 349 of 800 simulation runs
[28-Jun-2021 14:05:21] Completed 350 of 800 simulation runs
[28-Jun-2021 14:05:22] Completed 351 of 800 simulation runs
[28-Jun-2021 14:05:25] Completed 352 of 800 simulation runs
[28-Jun-2021 14:05:30] Completed 353 of 800 simulation runs
[28-Jun-2021 14:05:33] Completed 354 of 800 simulation runs
[28-Jun-2021 14:05:35] Completed 355 of 800 simulation runs
[28-Jun-2021 14:05:36] Completed 356 of 800 simulation runs
[28-Jun-2021 14:05:37] Completed 357 of 800 simulation runs
```

```
[28-Jun-2021 14:05:43] Completed 358 of 800 simulation runs
[28-Jun-2021 14:05:47] Completed 359 of 800 simulation runs
[28-Jun-2021 14:05:49] Completed 360 of 800 simulation runs
[28-Jun-2021 14:05:56] Completed 361 of 800 simulation runs
[28-Jun-2021 14:05:57] Completed 362 of 800 simulation runs
[28-Jun-2021 14:05:58] Completed 363 of 800 simulation runs
[28-Jun-2021 14:05:59] Completed 364 of 800 simulation runs
[28-Jun-2021 14:06:08] Completed 365 of 800 simulation runs
[28-Jun-2021 14:06:09] Completed 366 of 800 simulation runs
[28-Jun-2021 14:06:11] Completed 367 of 800 simulation runs
[28-Jun-2021 14:06:12] Completed 368 of 800 simulation runs
[28-Jun-2021 14:06:16] Completed 369 of 800 simulation runs
[28-Jun-2021 14:06:18] Completed 370 of 800 simulation runs
[28-Jun-2021 14:06:23] Completed 371 of 800 simulation runs
[28-Jun-2021 14:06:24] Completed 372 of 800 simulation runs
[28-Jun-2021 14:06:25] Completed 373 of 800 simulation runs
[28-Jun-2021 14:06:27] Completed 374 of 800 simulation runs
[28-Jun-2021 14:06:28] Completed 375 of 800 simulation runs
[28-Jun-2021 14:06:30] Completed 376 of 800 simulation runs
[28-Jun-2021 14:06:33] Completed 377 of 800 simulation runs
[28-Jun-2021 14:06:37] Completed 378 of 800 simulation runs
[28-Jun-2021 14:06:39] Completed 379 of 800 simulation runs
[28-Jun-2021 14:06:43] Completed 380 of 800 simulation runs
[28-Jun-2021 14:06:50] Completed 381 of 800 simulation runs
[28-Jun-2021 14:06:51] Completed 382 of 800 simulation runs
[28-Jun-2021 14:06:53] Completed 383 of 800 simulation runs
[28-Jun-2021 14:06:54] Completed 384 of 800 simulation runs
[28-Jun-2021 14:06:56] Completed 385 of 800 simulation runs
[28-Jun-2021 14:06:59] Completed 386 of 800 simulation runs
[28-Jun-2021 14:07:00] Completed 387 of 800 simulation runs
[28-Jun-2021 14:07:03] Completed 388 of 800 simulation runs
[28-Jun-2021 14:07:04] Completed 389 of 800 simulation runs
[28-Jun-2021 14:07:07] Completed 390 of 800 simulation runs
[28-Jun-2021 14:07:15] Completed 391 of 800 simulation runs
[28-Jun-2021 14:07:18] Completed 392 of 800 simulation runs
[28-Jun-2021 14:07:19] Completed 393 of 800 simulation runs
[28-Jun-2021 14:07:27] Completed 394 of 800 simulation runs
[28-Jun-2021 14:07:29] Completed 395 of 800 simulation runs
[28-Jun-2021 14:07:31] Completed 396 of 800 simulation runs
[28-Jun-2021 14:07:32] Completed 397 of 800 simulation runs
[28-Jun-2021 14:07:36] Completed 398 of 800 simulation runs
[28-Jun-2021 14:07:38] Completed 399 of 800 simulation runs
[28-Jun-2021 14:07:39] Completed 400 of 800 simulation runs
[28-Jun-2021 14:07:53] Completed 401 of 800 simulation runs
[28-Jun-2021 14:07:55] Completed 402 of 800 simulation runs
[28-Jun-2021 14:07:57] Completed 403 of 800 simulation runs
[28-Jun-2021 14:07:59] Completed 404 of 800 simulation runs
[28-Jun-2021 14:08:01] Completed 405 of 800 simulation runs
[28-Jun-2021 14:08:06] Completed 406 of 800 simulation runs
[28-Jun-2021 14:08:18] Completed 407 of 800 simulation runs
[28-Jun-2021 14:08:20] Completed 408 of 800 simulation runs
[28-Jun-2021 14:08:22] Completed 409 of 800 simulation runs
[28-Jun-2021 14:08:24] Completed 410 of 800 simulation runs
[28-Jun-2021 14:08:27] Completed 411 of 800 simulation runs
[28-Jun-2021 14:08:31] Completed 412 of 800 simulation runs
[28-Jun-2021 14:08:42] Completed 413 of 800 simulation runs
[28-Jun-2021 14:08:44] Completed 414 of 800 simulation runs
[28-Jun-2021 14:08:46] Completed 415 of 800 simulation runs
```

```
[28-Jun-2021 14:08:48] Completed 416 of 800 simulation runs
[28-Jun-2021 14:08:52] Completed 417 of 800 simulation runs
[28-Jun-2021 14:08:55] Completed 418 of 800 simulation runs
[28-Jun-2021 14:09:06] Completed 419 of 800 simulation runs
[28-Jun-2021 14:09:09] Completed 420 of 800 simulation runs
[28-Jun-2021 14:09:11] Completed 421 of 800 simulation runs
[28-Jun-2021 14:09:13] Completed 422 of 800 simulation runs
[28-Jun-2021 14:09:15] Completed 423 of 800 simulation runs
[28-Jun-2021 14:09:19] Completed 424 of 800 simulation runs
[28-Jun-2021 14:09:31] Completed 425 of 800 simulation runs
[28-Jun-2021 14:09:33] Completed 426 of 800 simulation runs
[28-Jun-2021 14:09:35] Completed 427 of 800 simulation runs
[28-Jun-2021 14:09:38] Completed 428 of 800 simulation runs
[28-Jun-2021 14:09:40] Completed 429 of 800 simulation runs
[28-Jun-2021 14:09:45] Completed 430 of 800 simulation runs
[28-Jun-2021 14:09:57] Completed 431 of 800 simulation runs
[28-Jun-2021 14:10:00] Completed 432 of 800 simulation runs
[28-Jun-2021 14:10:02] Completed 433 of 800 simulation runs
[28-Jun-2021 14:10:05] Completed 434 of 800 simulation runs
[28-Jun-2021 14:10:07] Completed 435 of 800 simulation runs
[28-Jun-2021 14:10:12] Completed 436 of 800 simulation runs
[28-Jun-2021 14:10:21] Completed 437 of 800 simulation runs
[28-Jun-2021 14:10:25] Completed 438 of 800 simulation runs
[28-Jun-2021 14:10:27] Completed 439 of 800 simulation runs
[28-Jun-2021 14:10:29] Completed 440 of 800 simulation runs
[28-Jun-2021 14:10:31] Completed 441 of 800 simulation runs
[28-Jun-2021 14:10:35] Completed 442 of 800 simulation runs
[28-Jun-2021 14:10:47] Completed 443 of 800 simulation runs
[28-Jun-2021 14:10:51] Completed 444 of 800 simulation runs
[28-Jun-2021 14:10:53] Completed 445 of 800 simulation runs
[28-Jun-2021 14:10:55] Completed 446 of 800 simulation runs
[28-Jun-2021 14:10:57] Completed 447 of 800 simulation runs
[28-Jun-2021 14:10:59] Completed 448 of 800 simulation runs
[28-Jun-2021 14:11:13] Completed 449 of 800 simulation runs
[28-Jun-2021 14:11:15] Completed 450 of 800 simulation runs
[28-Jun-2021 14:11:18] Completed 451 of 800 simulation runs
[28-Jun-2021 14:11:20] Completed 452 of 800 simulation runs
[28-Jun-2021 14:11:22] Completed 453 of 800 simulation runs
[28-Jun-2021 14:11:24] Completed 454 of 800 simulation runs
[28-Jun-2021 14:11:40] Completed 455 of 800 simulation runs
[28-Jun-2021 14:11:42] Completed 456 of 800 simulation runs
[28-Jun-2021 14:11:44] Completed 457 of 800 simulation runs
[28-Jun-2021 14:11:46] Completed 458 of 800 simulation runs
[28-Jun-2021 14:11:48] Completed 459 of 800 simulation runs
[28-Jun-2021 14:11:50] Completed 460 of 800 simulation runs
[28-Jun-2021 14:12:05] Completed 461 of 800 simulation runs
[28-Jun-2021 14:12:06] Completed 462 of 800 simulation runs
[28-Jun-2021 14:12:08] Completed 463 of 800 simulation runs
[28-Jun-2021 14:12:10] Completed 464 of 800 simulation runs
[28-Jun-2021 14:12:12] Completed 465 of 800 simulation runs
[28-Jun-2021 14:12:15] Completed 466 of 800 simulation runs
[28-Jun-2021 14:12:29] Completed 467 of 800 simulation runs
[28-Jun-2021 14:12:31] Completed 468 of 800 simulation runs
[28-Jun-2021 14:12:33] Completed 469 of 800 simulation runs
[28-Jun-2021 14:12:35] Completed 470 of 800 simulation runs
[28-Jun-2021 14:12:37] Completed 471 of 800 simulation runs
[28-Jun-2021 14:12:41] Completed 472 of 800 simulation runs
[28-Jun-2021 14:12:53] Completed 473 of 800 simulation runs
```

```
[28-Jun-2021 14:12:55] Completed 474 of 800 simulation runs
[28-Jun-2021 14:12:58] Completed 475 of 800 simulation runs
[28-Jun-2021 14:13:00] Completed 476 of 800 simulation runs
[28-Jun-2021 14:13:02] Completed 477 of 800 simulation runs
[28-Jun-2021 14:13:07] Completed 478 of 800 simulation runs
[28-Jun-2021 14:13:19] Completed 479 of 800 simulation runs
[28-Jun-2021 14:13:21] Completed 480 of 800 simulation runs
[28-Jun-2021 14:13:23] Completed 481 of 800 simulation runs
[28-Jun-2021 14:13:26] Completed 482 of 800 simulation runs
[28-Jun-2021 14:13:28] Completed 483 of 800 simulation runs
[28-Jun-2021 14:13:35] Completed 484 of 800 simulation runs
[28-Jun-2021 14:13:44] Completed 485 of 800 simulation runs
[28-Jun-2021 14:13:46] Completed 486 of 800 simulation runs
[28-Jun-2021 14:13:48] Completed 487 of 800 simulation runs
[28-Jun-2021 14:13:50] Completed 488 of 800 simulation runs
[28-Jun-2021 14:13:52] Completed 489 of 800 simulation runs
[28-Jun-2021 14:14:01] Completed 490 of 800 simulation runs
[28-Jun-2021 14:14:08] Completed 491 of 800 simulation runs
[28-Jun-2021 14:14:10] Completed 492 of 800 simulation runs
[28-Jun-2021 14:14:12] Completed 493 of 800 simulation runs
[28-Jun-2021 14:14:14] Completed 494 of 800 simulation runs
[28-Jun-2021 14:14:16] Completed 495 of 800 simulation runs
[28-Jun-2021 14:14:27] Completed 496 of 800 simulation runs
[28-Jun-2021 14:14:35] Completed 497 of 800 simulation runs
[28-Jun-2021 14:14:37] Completed 498 of 800 simulation runs
[28-Jun-2021 14:14:39] Completed 499 of 800 simulation runs
[28-Jun-2021 14:14:41] Completed 500 of 800 simulation runs
[28-Jun-2021 14:14:43] Completed 501 of 800 simulation runs
[28-Jun-2021 14:14:45] Completed 502 of 800 simulation runs
[28-Jun-2021 14:14:46] Completed 503 of 800 simulation runs
[28-Jun-2021 14:14:50] Completed 504 of 800 simulation runs
[28-Jun-2021 14:14:51] Completed 505 of 800 simulation runs
[28-Jun-2021 14:14:53] Completed 506 of 800 simulation runs
[28-Jun-2021 14:14:54] Completed 507 of 800 simulation runs
[28-Jun-2021 14:14:55] Completed 508 of 800 simulation runs
[28-Jun-2021 14:15:06] Completed 509 of 800 simulation runs
[28-Jun-2021 14:15:08] Completed 510 of 800 simulation runs
[28-Jun-2021 14:15:10] Completed 511 of 800 simulation runs
[28-Jun-2021 14:15:11] Completed 512 of 800 simulation runs
[28-Jun-2021 14:15:13] Completed 513 of 800 simulation runs
[28-Jun-2021 14:15:15] Completed 514 of 800 simulation runs
[28-Jun-2021 14:15:18] Completed 515 of 800 simulation runs
[28-Jun-2021 14:15:23] Completed 516 of 800 simulation runs
[28-Jun-2021 14:15:25] Completed 517 of 800 simulation runs
[28-Jun-2021 14:15:27] Completed 518 of 800 simulation runs
[28-Jun-2021 14:15:29] Completed 519 of 800 simulation runs
[28-Jun-2021 14:15:30] Completed 520 of 800 simulation runs
[28-Jun-2021 14:15:33] Completed 521 of 800 simulation runs
[28-Jun-2021 14:15:35] Completed 522 of 800 simulation runs
[28-Jun-2021 14:15:40] Completed 523 of 800 simulation runs
[28-Jun-2021 14:15:42] Completed 524 of 800 simulation runs
[28-Jun-2021 14:15:46] Completed 525 of 800 simulation runs
[28-Jun-2021 14:15:48] Completed 526 of 800 simulation runs
[28-Jun-2021 14:15:50] Completed 527 of 800 simulation runs
[28-Jun-2021 14:15:51] Completed 528 of 800 simulation runs
[28-Jun-2021 14:15:53] Completed 529 of 800 simulation runs
[28-Jun-2021 14:15:54] Completed 530 of 800 simulation runs
[28-Jun-2021 14:15:57] Completed 531 of 800 simulation runs
```

```
[28-Jun-2021 14:16:02] Completed 532 of 800 simulation runs
[28-Jun-2021 14:16:04] Completed 533 of 800 simulation runs
[28-Jun-2021 14:16:05] Completed 534 of 800 simulation runs
[28-Jun-2021 14:16:15] Completed 535 of 800 simulation runs
[28-Jun-2021 14:16:17] Completed 536 of 800 simulation runs
[28-Jun-2021 14:16:19] Completed 537 of 800 simulation runs
[28-Jun-2021 14:16:21] Completed 538 of 800 simulation runs
[28-Jun-2021 14:16:23] Completed 539 of 800 simulation runs
[28-Jun-2021 14:16:25] Completed 540 of 800 simulation runs
[28-Jun-2021 14:16:28] Completed 541 of 800 simulation runs
[28-Jun-2021 14:16:30] Completed 542 of 800 simulation runs
[28-Jun-2021 14:16:39] Completed 543 of 800 simulation runs
[28-Jun-2021 14:16:42] Completed 544 of 800 simulation runs
[28-Jun-2021 14:16:44] Completed 545 of 800 simulation runs
[28-Jun-2021 14:16:48] Completed 546 of 800 simulation runs
[28-Jun-2021 14:16:50] Completed 547 of 800 simulation runs
[28-Jun-2021 14:16:52] Completed 548 of 800 simulation runs
[28-Jun-2021 14:16:55] Completed 549 of 800 simulation runs
[28-Jun-2021 14:16:57] Completed 550 of 800 simulation runs
[28-Jun-2021 14:17:00] Completed 551 of 800 simulation runs
[28-Jun-2021 14:17:06] Completed 552 of 800 simulation runs
[28-Jun-2021 14:17:12] Completed 553 of 800 simulation runs
[28-Jun-2021 14:17:14] Completed 554 of 800 simulation runs
[28-Jun-2021 14:17:16] Completed 555 of 800 simulation runs
[28-Jun-2021 14:17:18] Completed 556 of 800 simulation runs
[28-Jun-2021 14:17:20] Completed 557 of 800 simulation runs
[28-Jun-2021 14:17:23] Completed 558 of 800 simulation runs
[28-Jun-2021 14:17:25] Completed 559 of 800 simulation runs
[28-Jun-2021 14:17:28] Completed 560 of 800 simulation runs
[28-Jun-2021 14:17:31] Completed 561 of 800 simulation runs
[28-Jun-2021 14:17:34] Completed 562 of 800 simulation runs
[28-Jun-2021 14:17:36] Completed 563 of 800 simulation runs
[28-Jun-2021 14:17:37] Completed 564 of 800 simulation runs
[28-Jun-2021 14:17:42] Completed 565 of 800 simulation runs
[28-Jun-2021 14:17:46] Completed 566 of 800 simulation runs
[28-Jun-2021 14:17:48] Completed 567 of 800 simulation runs
[28-Jun-2021 14:17:50] Completed 568 of 800 simulation runs
[28-Jun-2021 14:17:52] Completed 569 of 800 simulation runs
[28-Jun-2021 14:17:57] Completed 570 of 800 simulation runs
[28-Jun-2021 14:17:59] Completed 571 of 800 simulation runs
[28-Jun-2021 14:18:07] Completed 572 of 800 simulation runs
[28-Jun-2021 14:18:09] Completed 573 of 800 simulation runs
[28-Jun-2021 14:18:11] Completed 574 of 800 simulation runs
[28-Jun-2021 14:18:13] Completed 575 of 800 simulation runs
[28-Jun-2021 14:18:15] Completed 576 of 800 simulation runs
[28-Jun-2021 14:18:17] Completed 577 of 800 simulation runs
[28-Jun-2021 14:18:30] Completed 578 of 800 simulation runs
[28-Jun-2021 14:18:32] Completed 579 of 800 simulation runs
[28-Jun-2021 14:18:34] Completed 580 of 800 simulation runs
[28-Jun-2021 14:18:35] Completed 581 of 800 simulation runs
[28-Jun-2021 14:18:38] Completed 582 of 800 simulation runs
[28-Jun-2021 14:18:40] Completed 583 of 800 simulation runs
[28-Jun-2021 14:18:42] Completed 584 of 800 simulation runs
[28-Jun-2021 14:18:44] Completed 585 of 800 simulation runs
[28-Jun-2021 14:18:45] Completed 586 of 800 simulation runs
[28-Jun-2021 14:18:47] Completed 587 of 800 simulation runs
[28-Jun-2021 14:18:49] Completed 588 of 800 simulation runs
[28-Jun-2021 14:18:56] Completed 589 of 800 simulation runs
```

```
[28-Jun-2021 14:18:58] Completed 590 of 800 simulation runs
[28-Jun-2021 14:19:00] Completed 591 of 800 simulation runs
[28-Jun-2021 14:19:03] Completed 592 of 800 simulation runs
[28-Jun-2021 14:19:05] Completed 593 of 800 simulation runs
[28-Jun-2021 14:19:07] Completed 594 of 800 simulation runs
[28-Jun-2021 14:19:12] Completed 595 of 800 simulation runs
[28-Jun-2021 14:19:14] Completed 596 of 800 simulation runs
[28-Jun-2021 14:19:16] Completed 597 of 800 simulation runs
[28-Jun-2021 14:19:18] Completed 598 of 800 simulation runs
[28-Jun-2021 14:19:24] Completed 599 of 800 simulation runs
[28-Jun-2021 14:19:27] Completed 600 of 800 simulation runs
[28-Jun-2021 14:19:29] Completed 601 of 800 simulation runs
[28-Jun-2021 14:19:37] Completed 602 of 800 simulation runs
[28-Jun-2021 14:19:39] Completed 603 of 800 simulation runs
[28-Jun-2021 14:19:41] Completed 604 of 800 simulation runs
[28-Jun-2021 14:19:44] Completed 605 of 800 simulation runs
[28-Jun-2021 14:19:46] Completed 606 of 800 simulation runs
[28-Jun-2021 14:19:48] Completed 607 of 800 simulation runs
[28-Jun-2021 14:19:51] Completed 608 of 800 simulation runs
[28-Jun-2021 14:19:53] Completed 609 of 800 simulation runs
[28-Jun-2021 14:19:55] Completed 610 of 800 simulation runs
[28-Jun-2021 14:19:57] Completed 611 of 800 simulation runs
[28-Jun-2021 14:20:01] Completed 612 of 800 simulation runs
[28-Jun-2021 14:20:04] Completed 613 of 800 simulation runs
[28-Jun-2021 14:20:05] Completed 614 of 800 simulation runs
[28-Jun-2021 14:20:07] Completed 615 of 800 simulation runs
[28-Jun-2021 14:20:18] Completed 616 of 800 simulation runs
[28-Jun-2021 14:20:20] Completed 617 of 800 simulation runs
[28-Jun-2021 14:20:23] Completed 618 of 800 simulation runs
[28-Jun-2021 14:20:25] Completed 619 of 800 simulation runs
[28-Jun-2021 14:20:27] Completed 620 of 800 simulation runs
[28-Jun-2021 14:20:30] Completed 621 of 800 simulation runs
[28-Jun-2021 14:20:32] Completed 622 of 800 simulation runs
[28-Jun-2021 14:20:35] Completed 623 of 800 simulation runs
[28-Jun-2021 14:20:42] Completed 624 of 800 simulation runs
[28-Jun-2021 14:20:45] Completed 625 of 800 simulation runs
[28-Jun-2021 14:20:47] Completed 626 of 800 simulation runs
[28-Jun-2021 14:20:50] Completed 627 of 800 simulation runs
[28-Jun-2021 14:20:52] Completed 628 of 800 simulation runs
[28-Jun-2021 14:20:55] Completed 629 of 800 simulation runs
[28-Jun-2021 14:20:57] Completed 630 of 800 simulation runs
[28-Jun-2021 14:21:00] Completed 631 of 800 simulation runs
[28-Jun-2021 14:21:02] Completed 632 of 800 simulation runs
[28-Jun-2021 14:21:05] Completed 633 of 800 simulation runs
[28-Jun-2021 14:21:16] Completed 634 of 800 simulation runs
[28-Jun-2021 14:21:18] Completed 635 of 800 simulation runs
[28-Jun-2021 14:21:21] Completed 636 of 800 simulation runs
[28-Jun-2021 14:21:23] Completed 637 of 800 simulation runs
[28-Jun-2021 14:21:26] Completed 638 of 800 simulation runs
[28-Jun-2021 14:21:29] Completed 639 of 800 simulation runs
[28-Jun-2021 14:21:34] Completed 640 of 800 simulation runs
[28-Jun-2021 14:21:37] Completed 641 of 800 simulation runs
[28-Jun-2021 14:21:39] Completed 642 of 800 simulation runs
[28-Jun-2021 14:21:42] Completed 643 of 800 simulation runs
[28-Jun-2021 14:21:44] Completed 644 of 800 simulation runs
[28-Jun-2021 14:21:46] Completed 645 of 800 simulation runs
[28-Jun-2021 14:21:49] Completed 646 of 800 simulation runs
[28-Jun-2021 14:21:51] Completed 647 of 800 simulation runs
```

```
[28-Jun-2021 14:21:53] Completed 648 of 800 simulation runs
[28-Jun-2021 14:21:56] Completed 649 of 800 simulation runs
[28-Jun-2021 14:21:58] Completed 650 of 800 simulation runs
[28-Jun-2021 14:22:01] Completed 651 of 800 simulation runs
[28-Jun-2021 14:22:03] Completed 652 of 800 simulation runs
[28-Jun-2021 14:22:13] Completed 653 of 800 simulation runs
[28-Jun-2021 14:22:16] Completed 654 of 800 simulation runs
[28-Jun-2021 14:22:18] Completed 655 of 800 simulation runs
[28-Jun-2021 14:22:20] Completed 656 of 800 simulation runs
[28-Jun-2021 14:22:23] Completed 657 of 800 simulation runs
[28-Jun-2021 14:22:26] Completed 658 of 800 simulation runs
[28-Jun-2021 14:22:28] Completed 659 of 800 simulation runs
[28-Jun-2021 14:22:34] Completed 660 of 800 simulation runs
[28-Jun-2021 14:22:37] Completed 661 of 800 simulation runs
[28-Jun-2021 14:22:39] Completed 662 of 800 simulation runs
[28-Jun-2021 14:22:42] Completed 663 of 800 simulation runs
[28-Jun-2021 14:22:45] Completed 664 of 800 simulation runs
[28-Jun-2021 14:22:53] Completed 665 of 800 simulation runs
[28-Jun-2021 14:22:56] Completed 666 of 800 simulation runs
[28-Jun-2021 14:22:59] Completed 667 of 800 simulation runs
[28-Jun-2021 14:23:01] Completed 668 of 800 simulation runs
[28-Jun-2021 14:23:04] Completed 669 of 800 simulation runs
[28-Jun-2021 14:23:07] Completed 670 of 800 simulation runs
[28-Jun-2021 14:23:12] Completed 671 of 800 simulation runs
[28-Jun-2021 14:23:15] Completed 672 of 800 simulation runs
[28-Jun-2021 14:23:18] Completed 673 of 800 simulation runs
[28-Jun-2021 14:23:21] Completed 674 of 800 simulation runs
[28-Jun-2021 14:23:23] Completed 675 of 800 simulation runs
[28-Jun-2021 14:23:25] Completed 676 of 800 simulation runs
[28-Jun-2021 14:23:28] Completed 677 of 800 simulation runs
[28-Jun-2021 14:23:31] Completed 678 of 800 simulation runs
[28-Jun-2021 14:23:34] Completed 679 of 800 simulation runs
[28-Jun-2021 14:23:37] Completed 680 of 800 simulation runs
[28-Jun-2021 14:23:41] Completed 681 of 800 simulation runs
[28-Jun-2021 14:23:44] Completed 682 of 800 simulation runs
[28-Jun-2021 14:23:51] Completed 683 of 800 simulation runs
[28-Jun-2021 14:23:53] Completed 684 of 800 simulation runs
[28-Jun-2021 14:23:56] Completed 685 of 800 simulation runs
[28-Jun-2021 14:23:59] Completed 686 of 800 simulation runs
[28-Jun-2021 14:24:01] Completed 687 of 800 simulation runs
[28-Jun-2021 14:24:04] Completed 688 of 800 simulation runs
[28-Jun-2021 14:24:07] Completed 689 of 800 simulation runs
[28-Jun-2021 14:24:09] Completed 690 of 800 simulation runs
[28-Jun-2021 14:24:11] Completed 691 of 800 simulation runs
[28-Jun-2021 14:24:17] Completed 692 of 800 simulation runs
[28-Jun-2021 14:24:20] Completed 693 of 800 simulation runs
[28-Jun-2021 14:24:28] Completed 694 of 800 simulation runs
[28-Jun-2021 14:24:31] Completed 695 of 800 simulation runs
[28-Jun-2021 14:24:33] Completed 696 of 800 simulation runs
[28-Jun-2021 14:24:36] Completed 697 of 800 simulation runs
[28-Jun-2021 14:24:38] Completed 698 of 800 simulation runs
[28-Jun-2021 14:24:42] Completed 699 of 800 simulation runs
[28-Jun-2021 14:24:45] Completed 700 of 800 simulation runs
[28-Jun-2021 14:24:48] Completed 701 of 800 simulation runs
[28-Jun-2021 14:24:50] Completed 702 of 800 simulation runs
[28-Jun-2021 14:24:57] Completed 703 of 800 simulation runs
[28-Jun-2021 14:24:59] Completed 704 of 800 simulation runs
[28-Jun-2021 14:25:02] Completed 705 of 800 simulation runs
```

```
[28-Jun-2021 14:25:04] Completed 706 of 800 simulation runs
[28-Jun-2021 14:25:07] Completed 707 of 800 simulation runs
[28-Jun-2021 14:25:10] Completed 708 of 800 simulation runs
[28-Jun-2021 14:25:13] Completed 709 of 800 simulation runs
[28-Jun-2021 14:25:15] Completed 710 of 800 simulation runs
[28-Jun-2021 14:25:18] Completed 711 of 800 simulation runs
[28-Jun-2021 14:25:21] Completed 712 of 800 simulation runs
[28-Jun-2021 14:25:24] Completed 713 of 800 simulation runs
[28-Jun-2021 14:25:27] Completed 714 of 800 simulation runs
[28-Jun-2021 14:25:30] Completed 715 of 800 simulation runs
[28-Jun-2021 14:25:33] Completed 716 of 800 simulation runs
[28-Jun-2021 14:25:36] Completed 717 of 800 simulation runs
[28-Jun-2021 14:25:39] Completed 718 of 800 simulation runs
[28-Jun-2021 14:25:41] Completed 719 of 800 simulation runs
[28-Jun-2021 14:25:43] Completed 720 of 800 simulation runs
[28-Jun-2021 14:25:46] Completed 721 of 800 simulation runs
[28-Jun-2021 14:25:51] Completed 722 of 800 simulation runs
[28-Jun-2021 14:25:55] Completed 723 of 800 simulation runs
[28-Jun-2021 14:25:58] Completed 724 of 800 simulation runs
[28-Jun-2021 14:26:01] Completed 725 of 800 simulation runs
[28-Jun-2021 14:26:03] Completed 726 of 800 simulation runs
[28-Jun-2021 14:26:13] Completed 727 of 800 simulation runs
[28-Jun-2021 14:26:16] Completed 728 of 800 simulation runs
[28-Jun-2021 14:26:19] Completed 729 of 800 simulation runs
[28-Jun-2021 14:26:22] Completed 730 of 800 simulation runs
[28-Jun-2021 14:26:25] Completed 731 of 800 simulation runs
[28-Jun-2021 14:26:28] Completed 732 of 800 simulation runs
[28-Jun-2021 14:26:30] Completed 733 of 800 simulation runs
[28-Jun-2021 14:26:33] Completed 734 of 800 simulation runs
[28-Jun-2021 14:26:36] Completed 735 of 800 simulation runs
[28-Jun-2021 14:26:39] Completed 736 of 800 simulation runs
[28-Jun-2021 14:26:41] Completed 737 of 800 simulation runs
[28-Jun-2021 14:26:45] Completed 738 of 800 simulation runs
[28-Jun-2021 14:26:47] Completed 739 of 800 simulation runs
[28-Jun-2021 14:26:52] Completed 740 of 800 simulation runs
[28-Jun-2021 14:26:55] Completed 741 of 800 simulation runs
[28-Jun-2021 14:26:58] Completed 742 of 800 simulation runs
[28-Jun-2021 14:27:01] Completed 743 of 800 simulation runs
[28-Jun-2021 14:27:03] Completed 744 of 800 simulation runs
[28-Jun-2021 14:27:06] Completed 745 of 800 simulation runs
[28-Jun-2021 14:27:09] Completed 746 of 800 simulation runs
[28-Jun-2021 14:27:11] Completed 747 of 800 simulation runs
[28-Jun-2021 14:27:14] Completed 748 of 800 simulation runs
[28-Jun-2021 14:27:17] Completed 749 of 800 simulation runs
[28-Jun-2021 14:27:19] Completed 750 of 800 simulation runs
[28-Jun-2021 14:27:22] Completed 751 of 800 simulation runs
[28-Jun-2021 14:27:25] Completed 752 of 800 simulation runs
[28-Jun-2021 14:27:28] Completed 753 of 800 simulation runs
[28-Jun-2021 14:27:31] Completed 754 of 800 simulation runs
[28-Jun-2021 14:27:34] Completed 755 of 800 simulation runs
[28-Jun-2021 14:27:37] Completed 756 of 800 simulation runs
[28-Jun-2021 14:27:40] Completed 757 of 800 simulation runs
[28-Jun-2021 14:27:42] Completed 758 of 800 simulation runs
[28-Jun-2021 14:27:45] Completed 759 of 800 simulation runs
[28-Jun-2021 14:27:48] Completed 760 of 800 simulation runs
[28-Jun-2021 14:27:50] Completed 761 of 800 simulation runs
[28-Jun-2021 14:27:53] Completed 762 of 800 simulation runs
[28-Jun-2021 14:27:56] Completed 763 of 800 simulation runs
```

```
[28-Jun-2021 14:27:59] Completed 764 of 800 simulation runs
[28-Jun-2021 14:28:02] Completed 765 of 800 simulation runs
[28-Jun-2021 14:28:05] Completed 766 of 800 simulation runs
[28-Jun-2021 14:28:08] Completed 767 of 800 simulation runs
[28-Jun-2021 14:28:12] Completed 768 of 800 simulation runs
[28-Jun-2021 14:28:15] Completed 769 of 800 simulation runs
[28-Jun-2021 14:28:17] Completed 770 of 800 simulation runs
[28-Jun-2021 14:28:21] Completed 771 of 800 simulation runs
[28-Jun-2021 14:28:24] Completed 772 of 800 simulation runs
[28-Jun-2021 14:28:27] Completed 773 of 800 simulation runs
[28-Jun-2021 14:28:30] Completed 774 of 800 simulation runs
[28-Jun-2021 14:28:34] Completed 775 of 800 simulation runs
[28-Jun-2021 14:28:38] Completed 776 of 800 simulation runs
[28-Jun-2021 14:28:41] Completed 777 of 800 simulation runs
[28-Jun-2021 14:28:46] Completed 778 of 800 simulation runs
[28-Jun-2021 14:28:50] Completed 779 of 800 simulation runs
[28-Jun-2021 14:28:56] Completed 780 of 800 simulation runs
[28-Jun-2021 14:29:05] Completed 781 of 800 simulation runs
[28-Jun-2021 14:29:12] Completed 782 of 800 simulation runs
[28-Jun-2021 14:29:17] Completed 783 of 800 simulation runs
[28-Jun-2021 14:29:22] Completed 784 of 800 simulation runs
[28-Jun-2021 14:29:26] Completed 785 of 800 simulation runs
[28-Jun-2021 14:29:31] Completed 786 of 800 simulation runs
[28-Jun-2021 14:29:35] Completed 787 of 800 simulation runs
[28-Jun-2021 14:29:39] Completed 788 of 800 simulation runs
[28-Jun-2021 14:29:42] Completed 789 of 800 simulation runs
[28-Jun-2021 14:29:46] Completed 790 of 800 simulation runs
[28-Jun-2021 14:29:49] Completed 791 of 800 simulation runs
[28-Jun-2021 14:29:52] Completed 792 of 800 simulation runs
[28-Jun-2021 14:29:55] Completed 793 of 800 simulation runs
[28-Jun-2021 14:29:59] Completed 794 of 800 simulation runs
[28-Jun-2021 14:30:02] Completed 795 of 800 simulation runs
[28-Jun-2021 14:30:04] Completed 796 of 800 simulation runs
[28-Jun-2021 14:30:07] Completed 797 of 800 simulation runs
[28-Jun-2021 14:30:10] Completed 798 of 800 simulation runs
[28-Jun-2021 14:30:12] Completed 799 of 800 simulation runs
[28-Jun-2021 14:30:15] Completed 800 of 800 simulation runs
[28-Jun-2021 14:30:23] Cleaning up parallel workers...
```

```
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
    "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')
```
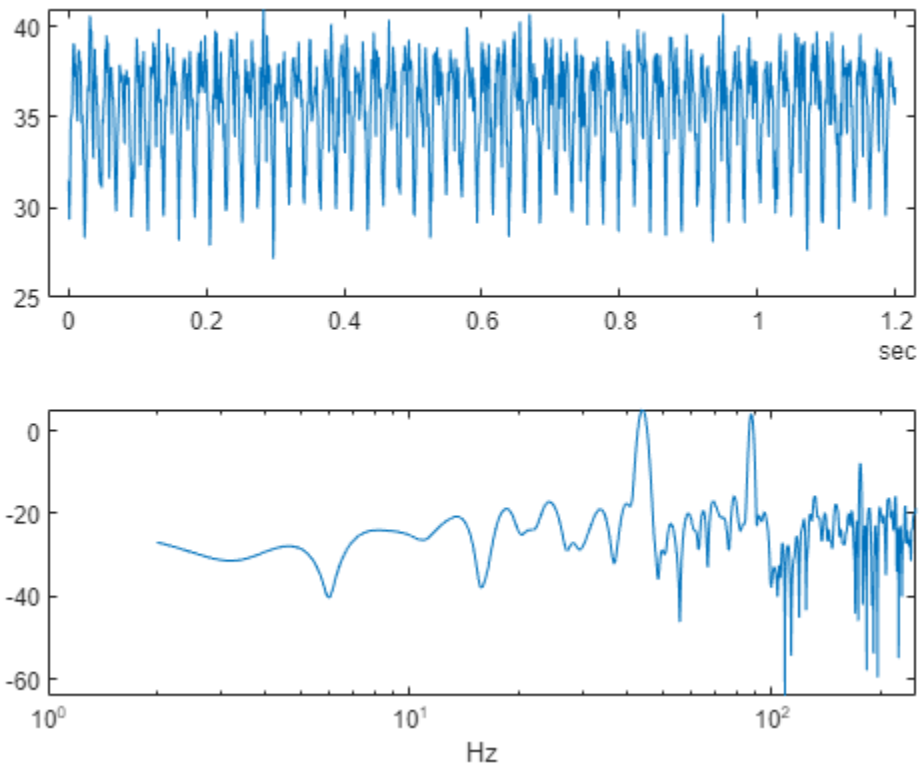
The flow spectrum reveals resonant peaks at expected frequencies. Specifically, the pump motor speed is 950 rpm, or 15.833 Hz, and since the pump has 3 cylinders the flow is expected to have a fundamental at 3*15.833 Hz, or 47.5 Hz, as well as harmonics at multiples of 47.5 Hz. The flow spectrum clearly 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.

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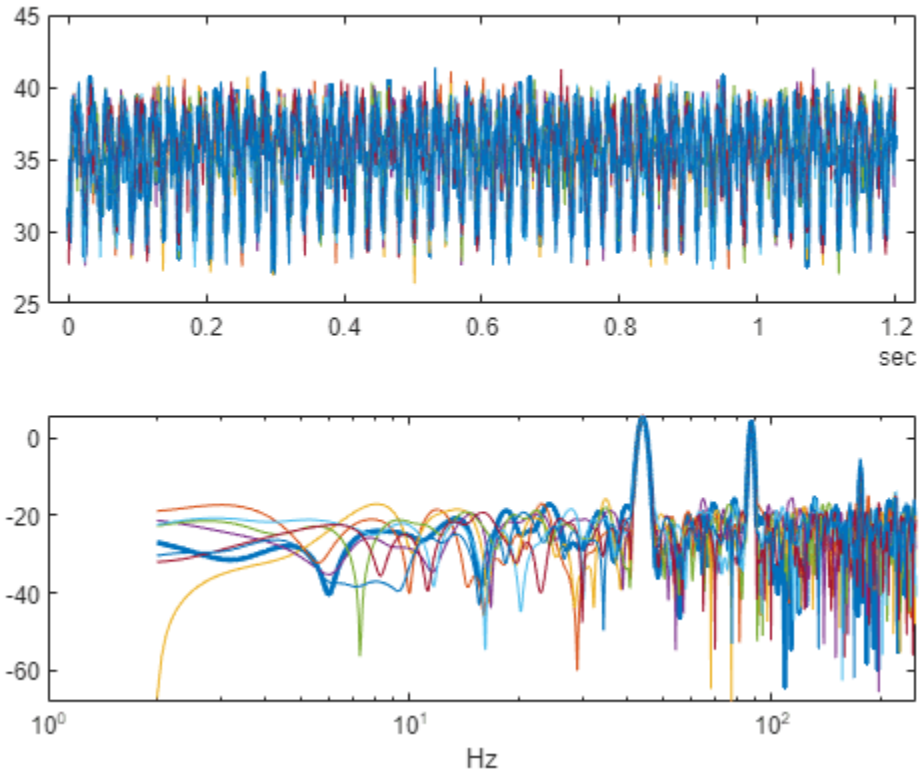
Note that under different fault conditions and severities the spectrum contains harmonics at the expected frequencies.

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));

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

```
Evaluation 0% complete

- Pass 1 of 1: Completed in 37 sec
Evaluation completed in 37 sec
```

```
data(1:10,:)
```

```
ans=10×17 table
    fPeak      pLow      pMid      pHigh     pKurtosis     qMean      qVar     qSkewness    qKurt
    _____    _____    _____    _____    _____    _____    _____    _____    _____

    43.909    0.8468     117.73    18.855      276.49      35.573    7.5235    -0.73064     2.77
    43.909    0.4622     125.98    18.956      12.417      35.577    7.8766    -0.70939      2.6
    43.909    1.1679     138.01     17.54      11.589      35.575    7.4403     -0.7229     2.713
    14.779    235.27     193.49    26.728      197.02      33.223    15.242    -0.24387     2.277
    14.779    287.41     198.79    25.321      487.58      32.955    17.606    -0.20213      2.2
    43.848    4.3805     137.31    19.175      110.93      35.275    7.5471    -0.70987     2.775
    14.839    303.74     176.33    23.665      392.38      32.908    17.638    -0.19533     2.23
    44.151    133.99     159.09    26.973       434.6       33.76    12.137    -0.37195     2.528
    43.848    0.43902    134.89    18.997      12.417      35.562    7.8636     -0.6853     2.666
    43.969    12.292     124.71    22.448      478.48      34.986    8.0954    -0.71245     2.920
```

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.

```matlab
% 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.

```matlab
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');
```


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 66% and the accuracy at predicting that there is a fault 94%.

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

```
ans = 0.6625
```

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

```
ans = 0.9375
```

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=10×3 table
    LeakFault     BlockingFault     BearingFault

    _____     _____     _____

      1e-09           0.71               0
      1e-09           0.77               0
      1e-09           0.71               0
      1e-09           0.77               0
      1e-09           0.71               0
      1e-09           0.77               0
      1e-09           0.71               0
      1e-09           0.74               0
      1e-09           0.77               0
      1e-09           0.77               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: 0% complete
```

```
Evaluation 0% complete

- Pass 1 of 1: Completed in 31 sec
Evaluation completed in 31 sec
```

```matlab
% 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')
```
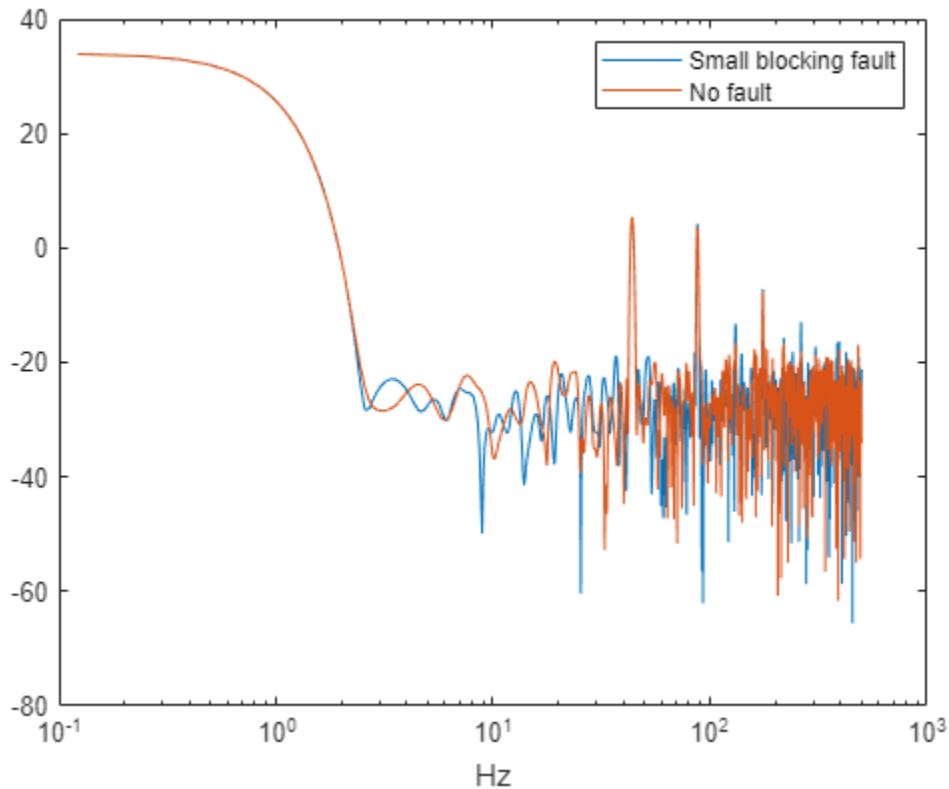
Conclusion

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]);
```

```matlab
% 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

Generate Synthetic Signals Using Conditional GAN

This example shows how to generate synthetic pump signals using a conditional generative adversarial network.

Generative adversarial networks (GANs) can be used to produce synthetic data that resembles real data input to the networks. GANs are useful when simulations are computationally expensive or experiments are costly. Conditional GANs (CGANs) can use data labels during the training process to generate data belonging to specific categories.

This example treats simulated signals obtained by a pump Simulink™ model as the "real" data that plays the role of training data set for a CGAN. The CGAN uses 1-D convolutional networks and is trained using a custom training loop and a deep learning array. In addition, this example uses principal component analysis (PCA) to visually compare the characteristics of generated and real signals.

CGAN for Signal Synthesis

CGANs consist of two networks that train together as adversaries:

1 *Generator* network — Given a label and random array as input, this network generates data with the same structure as the training data observations corresponding to the same label. The objective of the generator is to generate labeled data that the discriminator classifies as "real."

2 *Discriminator* network — Given batches of labeled data containing observations from both training data and generated data from the generator, this network attempts to classify the observations as "real" or "generated." The objective of the discriminator is to not be "fooled" by the generator when given batches of both real and generated labeled data.
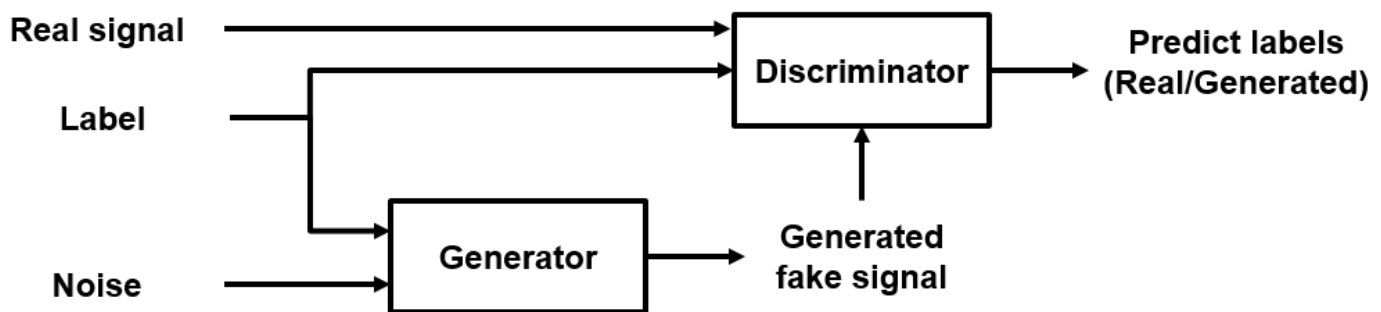
Ideally, these strategies result in a generator that generates convincingly realistic data corresponding to the input labels and a discriminator that has learned strong features characteristic of the training data for each label.

Load Data

The simulated data is generated by the pump Simulink model presented in the "Multi-Class Fault Detection Using Simulated Data" on page 1-43 example. The Simulink model is configured to model three types of faults: cylinder leaks, blocked inlets, and increased bearing friction. The data set contains 1575 pump output flow signals, of which 760 are healthy signals and 815 have a single fault, combinations of two faults, or combinations of three faults. Each signal has 1201 signal samples with a sample rate of 1000 Hz.

Download and unzip the data in your temporary directory, whose location is specified by MATLAB®
`tempdir` command. If you have the data in a folder different from that specified by `tempdir`, change
the directory name in the following code.

```
% Download the data
dataURL = 'https://ssd.mathworks.com/supportfiles/SPT/data/PumpSignalGAN.zip';
saveFolder = fullfile(tempdir,'PumpSignalGAN');
zipFile = fullfile(tempdir,'PumpSignalGAN.zip');
if ~exist(fullfile(saveFolder,'simulatedDataset.mat'),'file')
    websave(zipFile,dataURL);
    % Unzip the data
    unzip(zipFile,saveFolder)
end
```

The zip file contains the training data set and a pretrained CGAN:

- `simulatedDataset` — Simulated signals and their corresponding categorical labels
- `GANModel` — Generator and discriminator trained on the simulated data

Load the training data set and standardize the signals to have zero mean and unit variance.

```
load(fullfile(saveFolder,'simulatedDataset.mat')) % load data set
meanFlow = mean(flow,2);
flowNormalized = flow-meanFlow;
stdFlow = std(flowNormalized(:));
flowNormalized = flowNormalized/stdFlow;
```

Healthy signals are labeled as 1 and faulty signals are labeled as 2.

Define Generator Network

Define the following two-input network, which generates flow signals given 1-by-1-by-100 arrays of
random values and corresponding labels.

The network:

- Projects and reshapes the 1-by-1-by-100 arrays of noise to 4-by-1-by-1024 arrays by a custom
  layer.
- Converts the categorical labels to embedding vectors and reshapes them to a 4-by-1-by-1 arrays.
- Concatenates the results from the two inputs along the channel dimension. The output is a 4-by-1-
  by-1025 array.
- Upsamples the resulting arrays to 1201-by-1-by-1 arrays using a series of 1-D transposed
  convolution layers with batch normalization and ReLU layers.

To project and reshape the noise input, use the custom layer `projectAndReshapeLayer`, attached
to this example as a supporting file. The `projectAndReshapeLayer` object upscales the input using
a fully connected layer and reshapes the output to the specified size.

To input the labels into the network, use an `imageInputLayer` object and specify a size of 1-by-1. To
embed and reshape the label input, use the custom layer `embedAndReshapeLayer`, attached to this
example as a supporting file. The `embedAndReshapeLayer` object converts a categorical label to a
one-channel array of the specified size using an embedding and a fully connected operation. For
categorical inputs, use an embedding dimension of 100.

```
% Generator Network
```

```matlab
numFilters = 64;
numLatentInputs = 100;
projectionSize = [4 1 1024];
numClasses = 2;
embeddingDimension = 100;

layersGenerator = [
    imageInputLayer([1 1 numLatentInputs],'Normalization','none','Name','in')
    projectAndReshapeLayer(projectionSize,numLatentInputs,'proj');
    concatenationLayer(3,2,'Name','cat');
    transposedConv2dLayer([5 1],8*numFilters,'Name','tconv1')
    batchNormalizationLayer('Name','bn1','Epsilon',5e-5)
    reluLayer('Name','relu1')
    transposedConv2dLayer([10 1],4*numFilters,'Stride',4,'Cropping',[1 0],'Name','tconv2')
    batchNormalizationLayer('Name','bn2','Epsilon',5e-5)
    reluLayer('Name','relu2')
    transposedConv2dLayer([12 1],2*numFilters,'Stride',4,'Cropping',[1 0],'Name','tconv3')
    batchNormalizationLayer('Name','bn3','Epsilon',5e-5)
    reluLayer('Name','relu3')
    transposedConv2dLayer([5 1],numFilters,'Stride',4,'Cropping',[1 0],'Name','tconv4')
    batchNormalizationLayer('Name','bn4','Epsilon',5e-5)
    reluLayer('Name','relu4')
    transposedConv2dLayer([7 1],1,'Stride',2,'Cropping',[1 0],'Name','tconv5')
    ];

lgraphGenerator = layerGraph(layersGenerator);

layers = [
    imageInputLayer([1 1],'Name','labels','Normalization','none')
    embedAndReshapeLayer(projectionSize(1:2),embeddingDimension,numClasses,'emb')];

lgraphGenerator = addLayers(lgraphGenerator,layers);
lgraphGenerator = connectLayers(lgraphGenerator,'emb','cat/in2');
```

Plot the network structure for the generator.

```matlab
plot(lgraphGenerator)
```
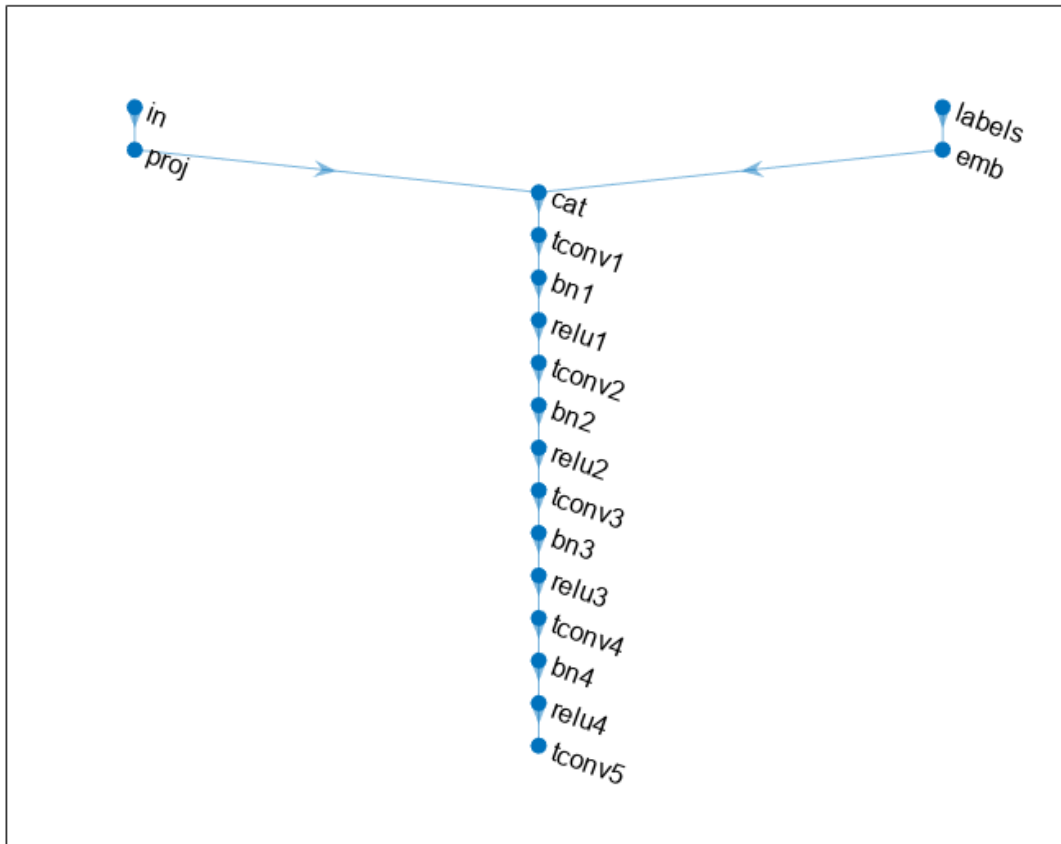
To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetGenerator = dlnetwork(lgraphGenerator);
```

Define Discriminator Network

Define the following two-input network, which classifies real and generated 1201-by-1 signals given a set of signals and their corresponding labels.

This network:

- Takes 1201-by-1-by-1 signals as input.

- Converts categorical labels to embedding vectors and reshapes them to a 1201-by-1-by-1 arrays.

- Concatenates the results from the two inputs along the channel dimension. The output is a 1201-by-1-by-1025 array.

- Downsamples the resulting arrays to scalar prediction scores, which are 1-by-1-by-1 arrays, using a series of 1-D convolution layers with leaky ReLU layers with a scale of 0.2.

```matlab
% Discriminator Network

scale = 0.2;
inputSize = [1201 1 1];

layersDiscriminator = [
    imageInputLayer(inputSize,'Normalization','none','Name','in')
    concatenationLayer(3,2,'Name','cat')
    convolution2dLayer([17 1],8*numFilters,'Stride',2,'Padding',[1 0],'Name','conv1')
    leakyReluLayer(scale,'Name','lrelu1')
    convolution2dLayer([16 1],4*numFilters,'Stride',4,'Padding',[1 0],'Name','conv2')
    leakyReluLayer(scale,'Name','lrelu2')
    convolution2dLayer([16 1],2*numFilters,'Stride',4,'Padding',[1 0],'Name','conv3')
    leakyReluLayer(scale,'Name','lrelu3')
    convolution2dLayer([8 1],numFilters,'Stride',4,'Padding',[1 0],'Name','conv4')
    leakyReluLayer(scale,'Name','lrelu4')
    convolution2dLayer([8 1],1,'Name','conv5')];

lgraphDiscriminator = layerGraph(layersDiscriminator);

layers = [
    imageInputLayer([1 1],'Name','labels','Normalization','none')
    embedAndReshapeLayer(inputSize,embeddingDimension,numClasses,'emb')];

lgraphDiscriminator = addLayers(lgraphDiscriminator,layers);
lgraphDiscriminator = connectLayers(lgraphDiscriminator,'emb','cat/in2');
```

Plot the network structure for the discriminator.

```matlab
plot(lgraphDiscriminator)
```
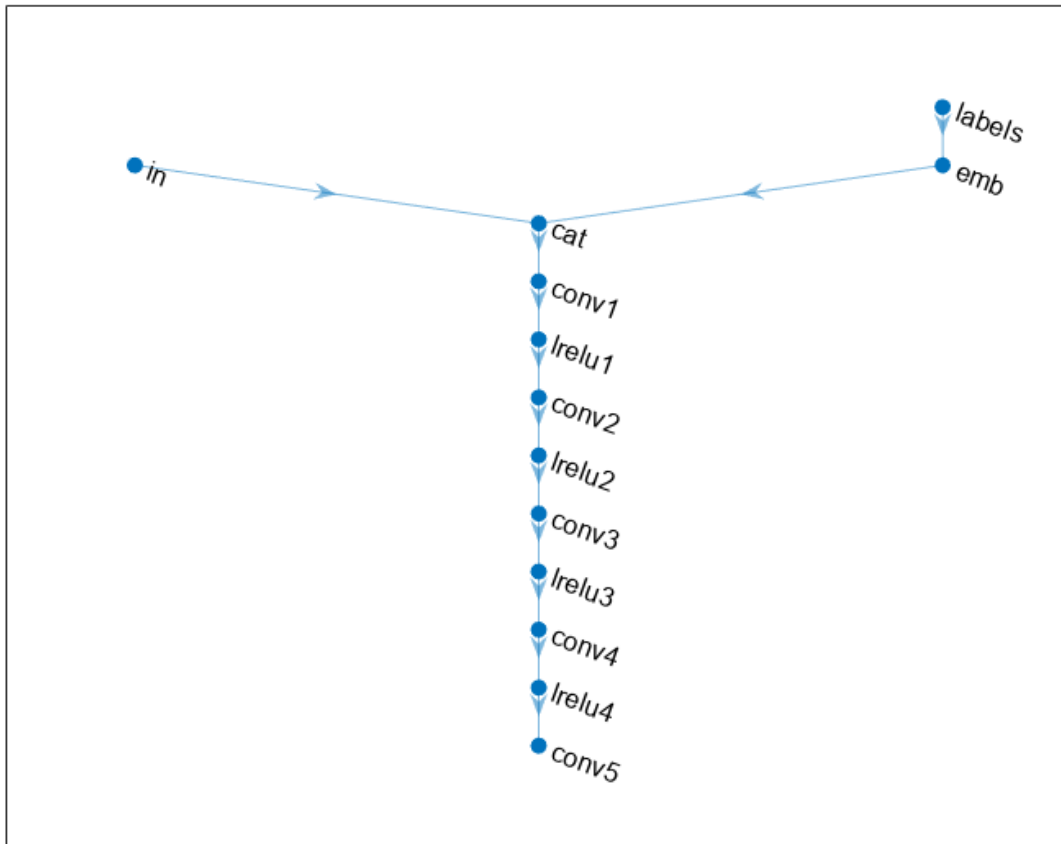
To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetDiscriminator = dlnetwork(lgraphDiscriminator);
```

Train Model

Train the CGAN model using a custom training loop. Loop over the training data and update the network parameters at each iteration. To monitor the training progress, display generated healthy and faulty signals using two fixed arrays of random values to input into the generator as well as a plot of the scores of the two networks.

For each epoch, shuffle the training data and loop over mini-batches of data.

For each mini-batch:

- Generate a `dlarray` (Deep Learning Toolbox) object containing an array of random values for the generator network.
- For GPU training, convert the data to a `gpuArray` (Parallel Computing Toolbox) object.

- Evaluate the model gradients using `dlfeval` (Deep Learning Toolbox) and the helper function `modelGradients`.
- Update the network parameters using the `adamupdate` (Deep Learning Toolbox) function.

The helper function `modelGradients` takes as input the generator and discriminator networks, a mini-batch of input data, and an array of random values, and returns the gradients of the loss with respect to the learnable parameters in the networks and the scores of the two networks. The loss function is defined in the helper function `ganLoss`.

Specify Training Options

Set the training parameters.

```
params.numLatentInputs = numLatentInputs;
params.numClasses = numClasses;
params.sizeData = [inputSize length(labels)];
params.numEpochs = 1000;
params.miniBatchSize = 256;

% Specify the options for Adam optimizer
params.learnRate = 0.0002;
params.gradientDecayFactor = 0.5;
params.squaredGradientDecayFactor = 0.999;
```

Set the execution environment to run the CGANs on the CPU. To run the CGANs on the GPU, set `executionEnvironment` to "`gpu`" or select the "Run on GPU" option in Live Editor. Using a GPU requires Parallel Computing Toolbox™. To see which GPUs are supported, see "GPU Computing Requirements" (Parallel Computing Toolbox).

```
executionEnvironment = [ Run on CPU     ▼ ];
params.executionEnvironment = executionEnvironment;
```

Skip the training process by loading the pretrained network. To train the network on your computer, set `trainNow` to `true` or select the "Train CGAN now" option in Live Editor.

```
trainNow = [ Use trained CGAN   ▼ ];
if trainNow
    % Train the CGAN
    [dlnetGenerator,dlnetDiscriminator] = trainGAN(dlnetGenerator, ...
        dlnetDiscriminator,flowNormalized,labels,params); %#ok
else
    % Use pretrained CGAN (default)
    load(fullfile(tempdir,'PumpSignalGAN','GANModel.mat')) % load data set
end
```

The training plot below shows an example of scores of the generator and discriminator networks. To learn more about how to interpret the network scores, see "Monitor GAN Training Progress and Identify Common Failure Modes" (Deep Learning Toolbox). In this example, the scores of both the generator and discriminator converge close to 0.5, indicating that the training performance is good.
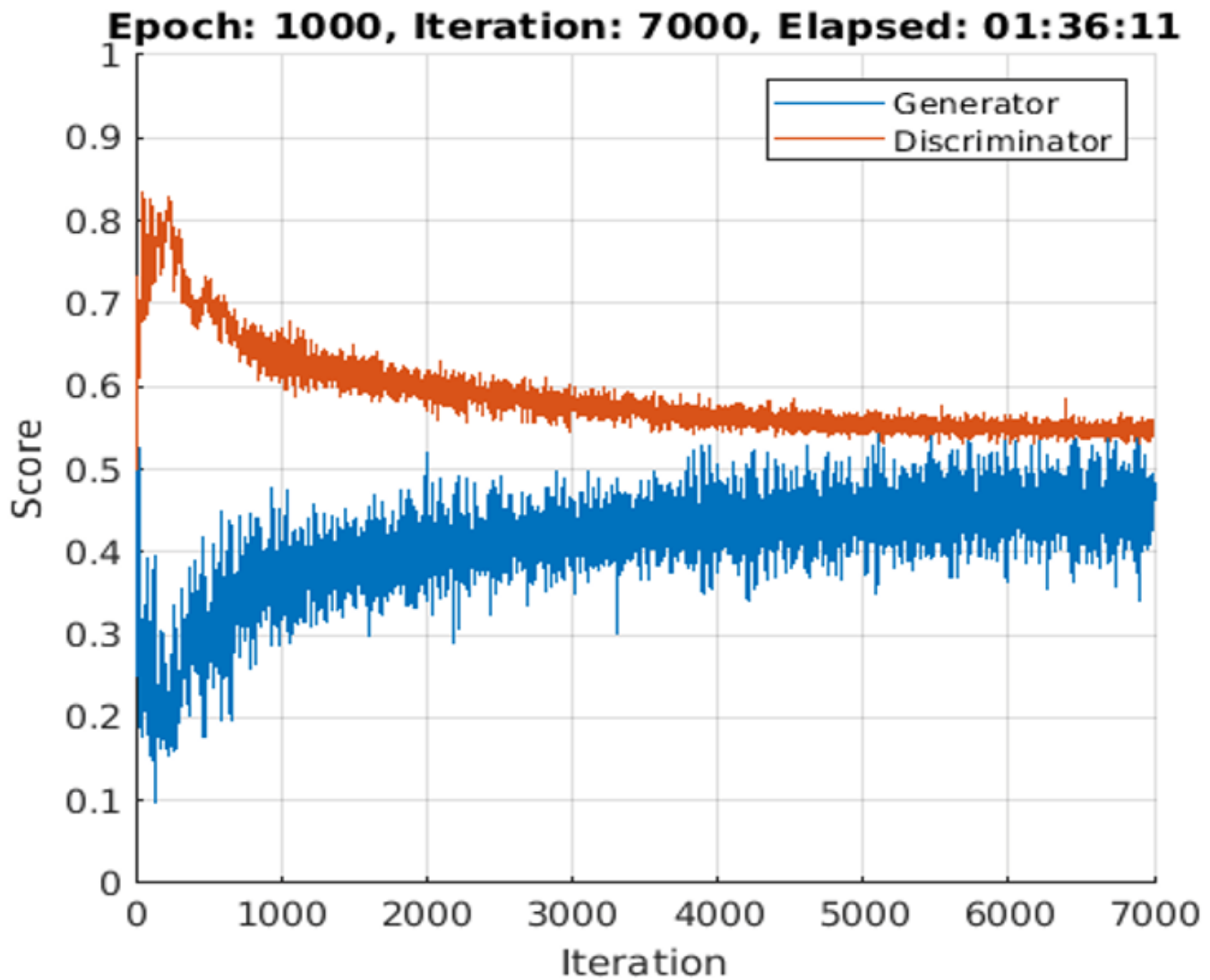
Synthesize Flow Signals

Create a `dlarray` object containing a batch of 2000 1-by-1-by-100 arrays of random values to input into the generator network. Reset the random number generator for reproducible results.

```
rng default
```

```
numTests = 2000;
ZNew = randn(1,1,numLatentInputs,numTests,'single');
dlZNew = dlarray(ZNew,'SSCB');
```

Specify that the first 1000 random arrays are healthy and the rest are faulty.

```
TNew = ones(1,1,1,numTests,'single');
TNew(1,1,1,numTests/2+1:end) = single(2);
dlTNew = dlarray(TNew,'SSCB');
```

To generate signals using the GPU, convert the data to `gpuArray` objects.

```
if executionEnvironment == "gpu"
    dlZNew = gpuArray(dlZNew);
    dlTNew = gpuArray(dlTNew);
end
```

Use the `predict` function on the generator with the batch of 1-by-1-by-100 arrays of random values and labels to generate synthetic signals and revert the standardization step that you performed on the original flow signals.

```
dlXGeneratedNew = predict(dlnetGenerator,dlZNew,dlTNew)*stdFlow+meanFlow;
```

Signal Feature Visualization

Unlike images and audio signals, general signals have characteristics that make them difficult for human perception to tell apart. To compare real and generated signals or healthy and faulty signals, you can apply principal component analysis (PCA) to the statistical features of the real signals and then project the features of the generated signals to the same PCA subspace.

Feature Extraction

Combine the original real signal and the generated signals in one data matrix. Use the helper function `helperExtractFeature` to extract the feature including common signal statistics such as the mean and variance as well as spectral characteristics.

```
idxGenerated = 1:numTests;
idxReal = numTests+1:numTests+size(flow,2);

XGeneratedNew = squeeze(extractdata(gather(dlXGeneratedNew)));
x = [XGeneratedNew single(flow)];

features = zeros(size(x,2),14,'like',x);

for ii = 1:size(x,2)
    features(ii,:) = helperExtractFeature(x(:,ii));
end
```

Each row of `features` corresponds to the features of one signal.

Modify the labels for the generated healthy and faulty signals as well as real healthy and faulty signals.

```
L = [squeeze(TNew)+2;labels.'];
```

The labels now have these definitions:

- 1 — Generated healthy signals
- 2 — Generated faulty signals
- 3 — Real healthy signals
- 4 — Real faulty signals

Principal Component Analysis

Perform PCA on the features of the real signals and project the features of the generated signals to the same PCA subspace. `W` is the coefficient and `Y` is the score.

```
% PCA via svd
featuresReal = features(idxReal,:);
mu = mean(featuresReal,1);
[~,S,W] = svd(featuresReal-mu);
S = diag(S);
Y = (features-mu)*W;
```

From the singular vector S, the first three singular values make up 99% of the energy in S. You can visualize the signal features by taking advantage of the first three principal components.

```
sum(S(1:3))/sum(S)
```

```
ans = single
    0.9923
```

Plot the features of all the signals using the first three principal components. In the PCA subspace, the distribution of the generated signals is similar to the distribution of the real signals.

```
idxHealthyR = L==1;
idxFaultR = L==2;

idxHealthyG = L==3;
idxFaultG = L==4;

pp = Y(:,1:3);

figure
scatter3(pp(idxHealthyR,1),pp(idxHealthyR,2),pp(idxHealthyR,3),'o')
xlabel('1st Principal Component')
ylabel('2nd Principal Component')
zlabel('3rd Principal Component')
hold on
scatter3(pp(idxFaultR,1),pp(idxFaultR,2),pp(idxFaultR,3),'d')
scatter3(pp(idxHealthyG,1),pp(idxHealthyG,2),pp(idxHealthyG,3),'s')
scatter3(pp(idxFaultG,1),pp(idxFaultG,2),pp(idxFaultG,3),'+')
view(-10,20)
legend('Real healthy','Real faulty','Generated healthy','Generated faulty', ...
    'Location','Best')
hold off
```
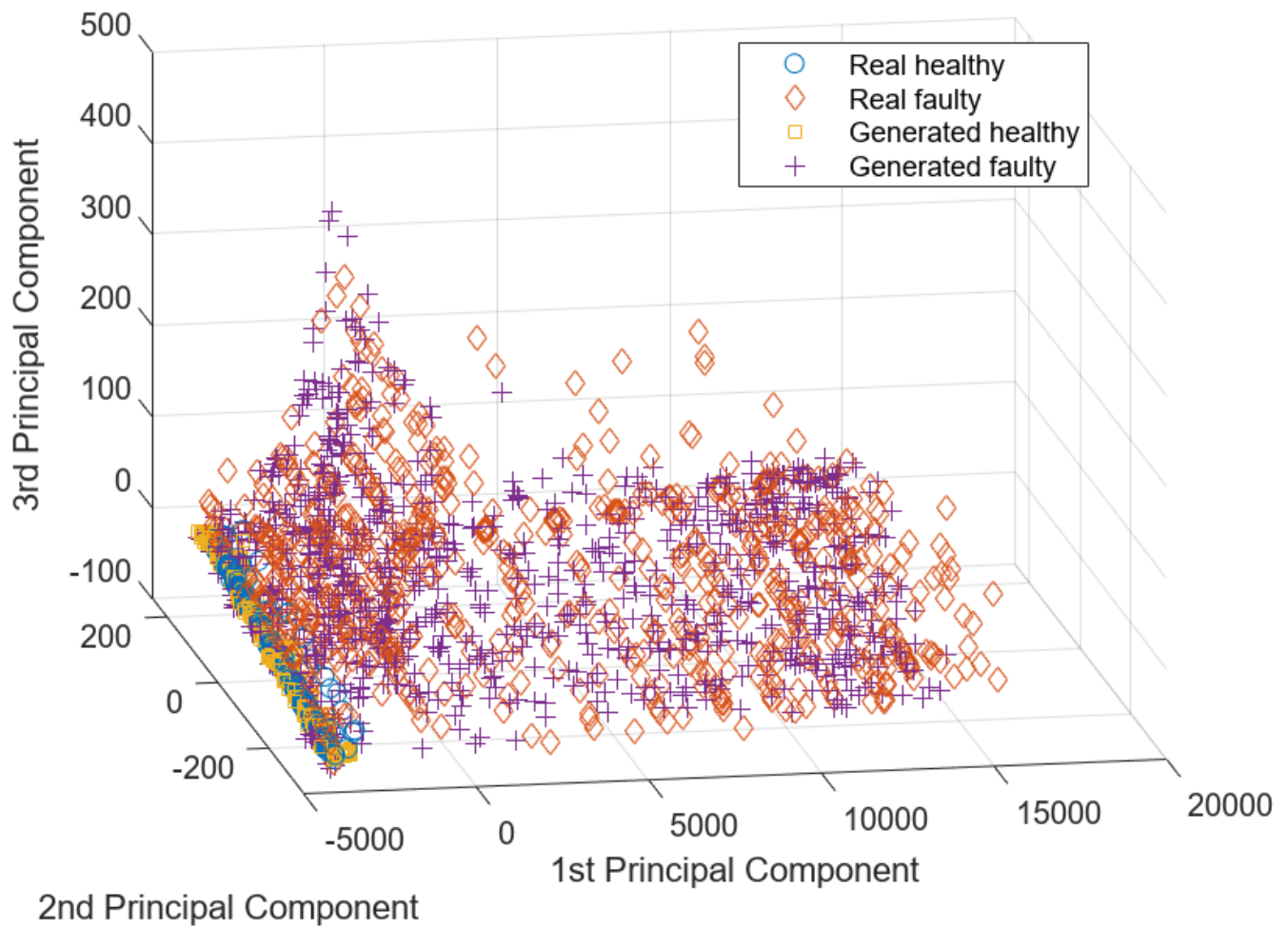
To better capture the difference between the real signals and generated signals, plot the subspace using the first two principal components.

```
view(2)
```
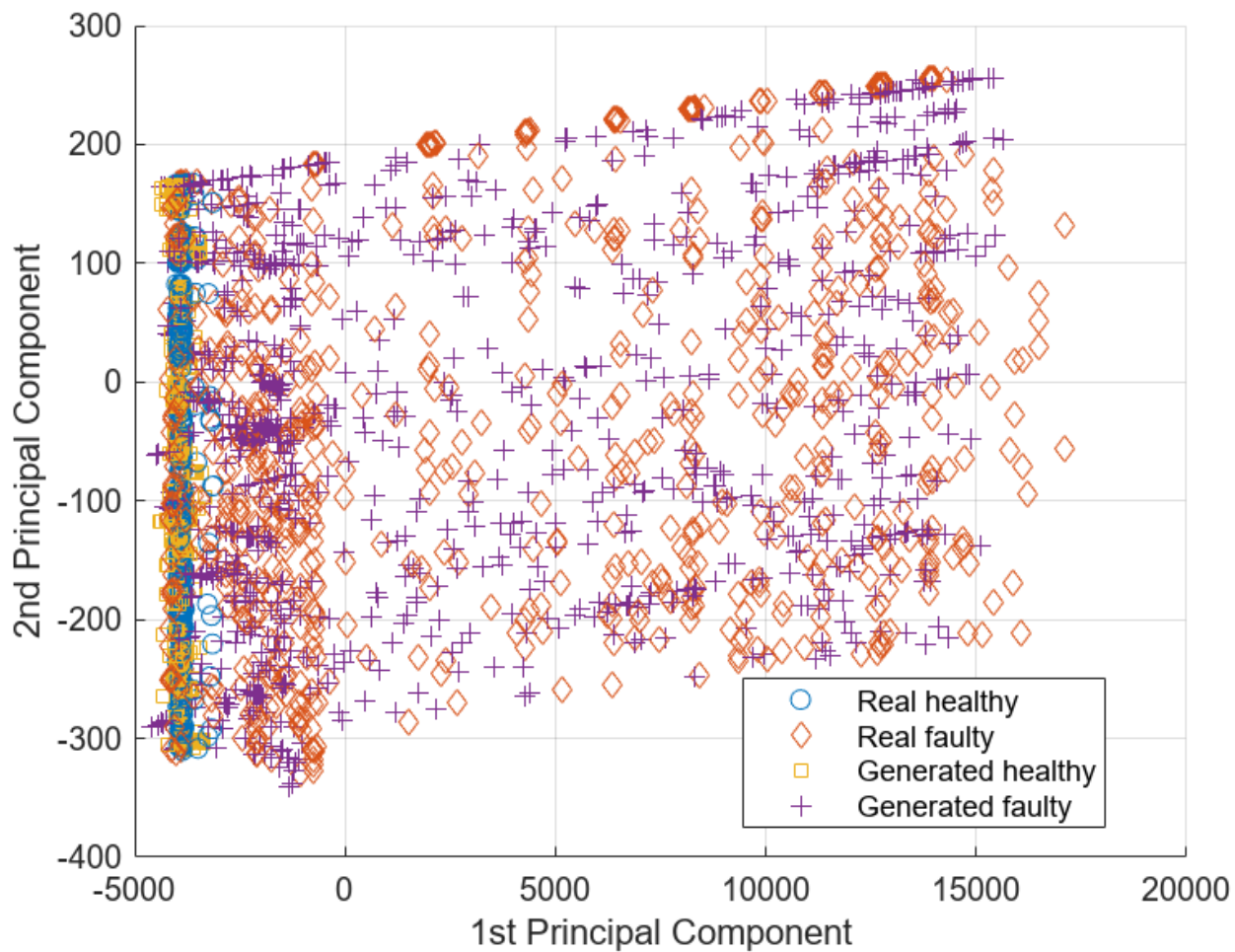
Healthy and faulty signals lie in the same area of the PCA subspace regardless of their being real or generated, demonstrating that the generated signals have features similar to those of the real signals.

Predict Labels of Real Signals

To further illustrate the performance of the CGAN, train an SVM classifier based on the generated signals and then predict whether a real signal is healthy or faulty.

Set the generated signals as the training data set and the real signals as the test data set. Change the numeric labels to character vectors.

```
LABELS = {'Healthy','Faulty'};
strL = LABELS([squeeze(TNew);labels.']).';

dataTrain = features(idxGenerated,:);
dataTest = features(idxReal,:);

labelTrain = strL(idxGenerated);
labelTest = strL(idxReal);
```

```
predictors = dataTrain;
response = labelTrain;
cvp = cvpartition(size(predictors,1),'KFold',5);
```

Train an SVM classifier using the generated signals.

```
SVMClassifier = fitcsvm( ...
    predictors(cvp.training(1),:), ...
    response(cvp.training(1)),'KernelFunction','polynomial', ...
    'PolynomialOrder',2, ...
    'KernelScale','auto', ...
    'BoxConstraint',1, ...
    'ClassNames',LABELS, ...
    'Standardize',true);
```

Use the trained classifier to obtain the predicted labels for the real signals. The classifier achieves a prediction accuracy above 90%.

```
actualValue = labelTest;
predictedValue = predict(SVMClassifier,dataTest);
predictAccuracy = mean(cellfun(@strcmp,actualValue,predictedValue))
```

```
predictAccuracy = 0.9460
```

Use a confusion matrix to view detailed information about prediction performance for each category. The confusion matrix shows that, in each category, the classifier trained based on the generated signals achieves a high degree of accuracy.

```
figure
confusionchart(actualValue,predictedValue)
```
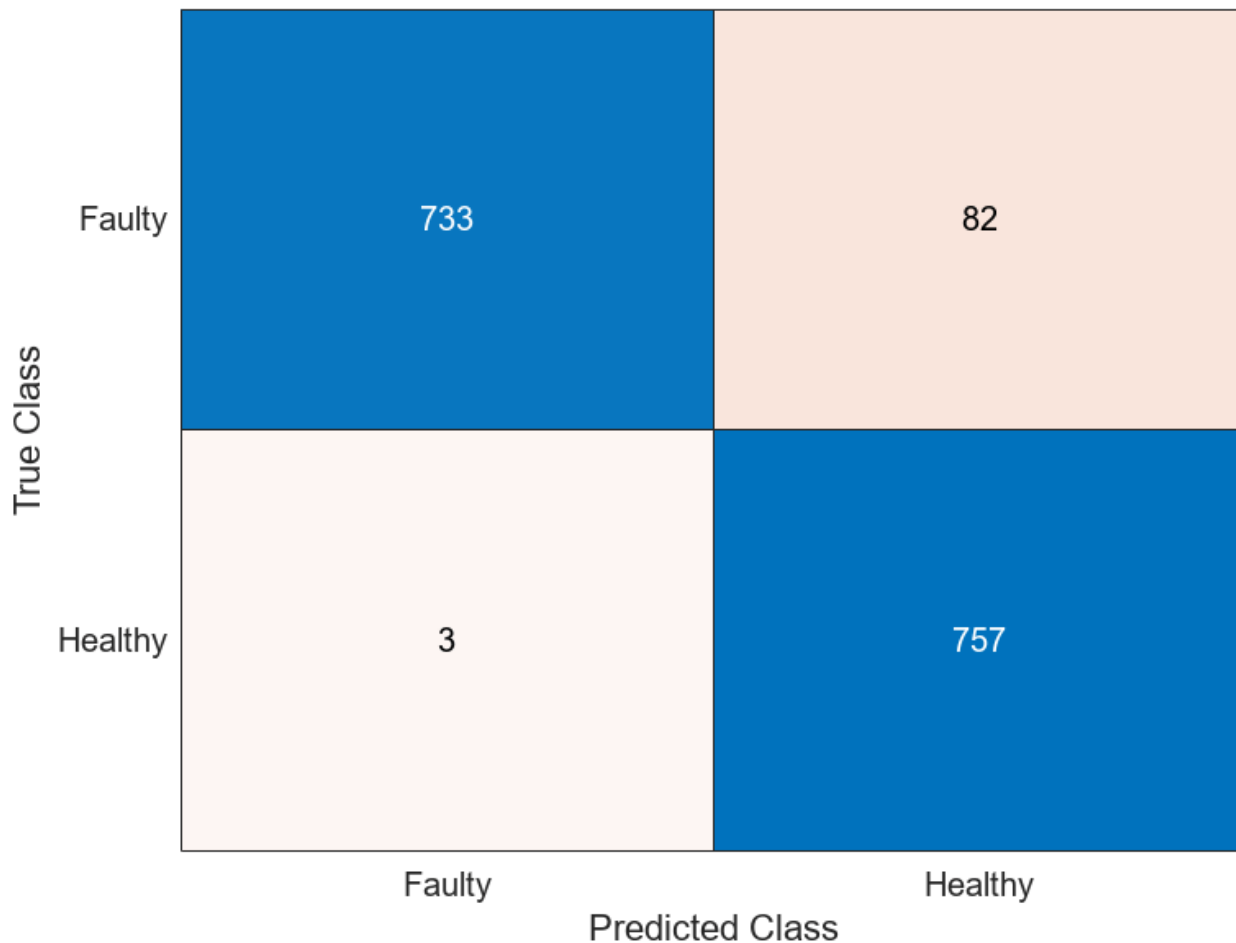
Case Study

Compare the spectral characteristics of real and generated signals. Due to the nondeterministic behavior of GPU training, if you train the CGAN model yourself, your results might differ from the results in this example.

The pump motor speed is 950 rpm, or 15.833 Hz, and since the pump has three cylinders the flow is expected to have a fundamental at 3 times 15.833 Hz, or 47.5 Hz, and harmonics at multiples of 47.5 Hz. Plot the spectrum for one case of the real and generated healthy signals. From the plot, the generated healthy signal has relatively high power values at 47.5 Hz and 2 times 47.5 Hz, which is exactly the same as the real healthy signal.

```
Fs = 1000;
pspectrum([x(:,1) x(:,2006)],Fs)
set(gca,'XScale','log')
legend('Generated healthy','Real healthy')
```
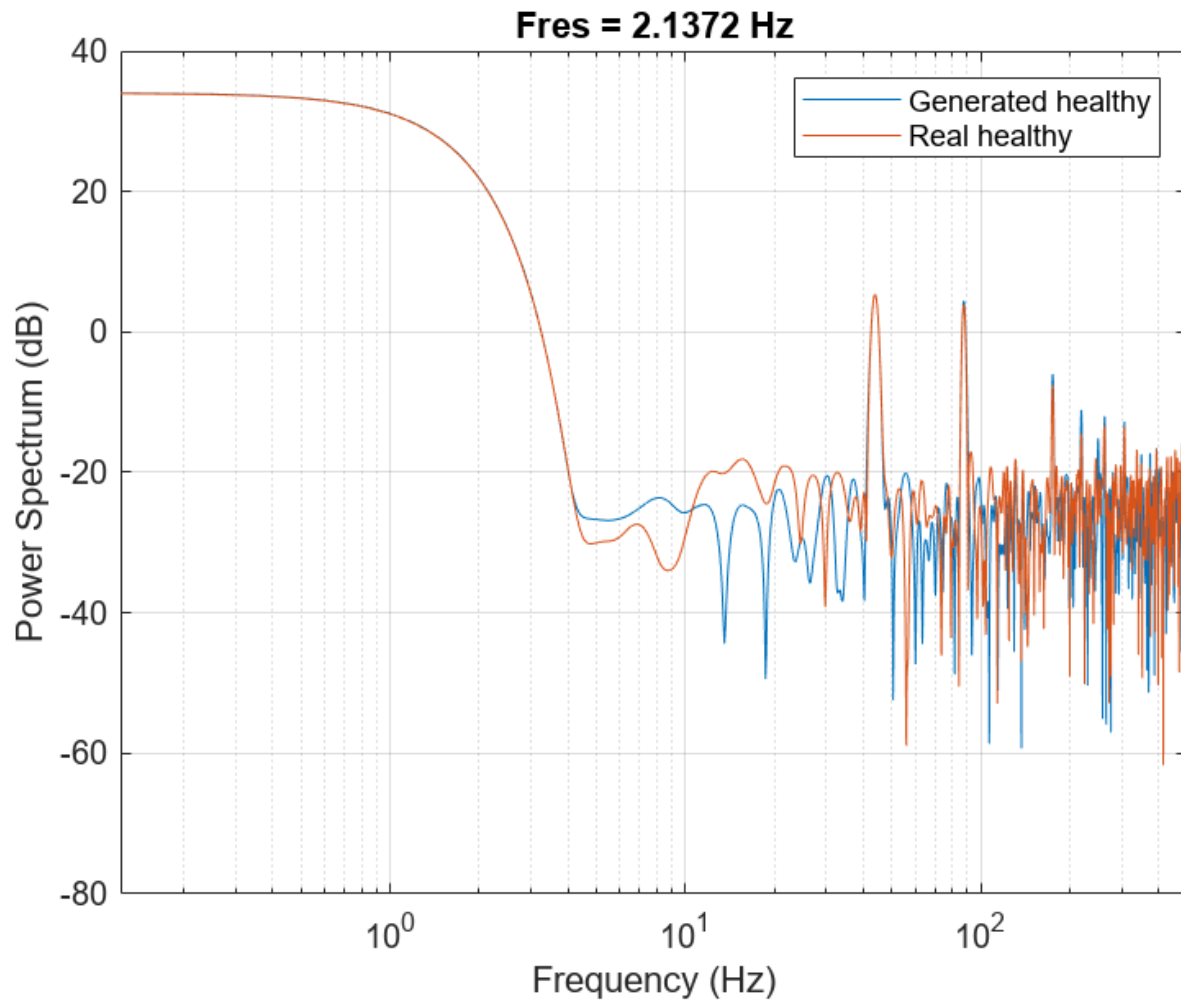
If faults exist, resonances will occur at the pump motor speed, 15.833 Hz, and its harmonics. Plot the spectra for one case of real and generated faulty signals. The generated signal has relatively high power values at around 15.833 Hz and its harmonics, which is similar to the real faulty signal.

```
pspectrum([x(:,1011) x(:,2100)],Fs)
set(gca,'XScale','log')
legend('Generated faulty','Real faulty')
```
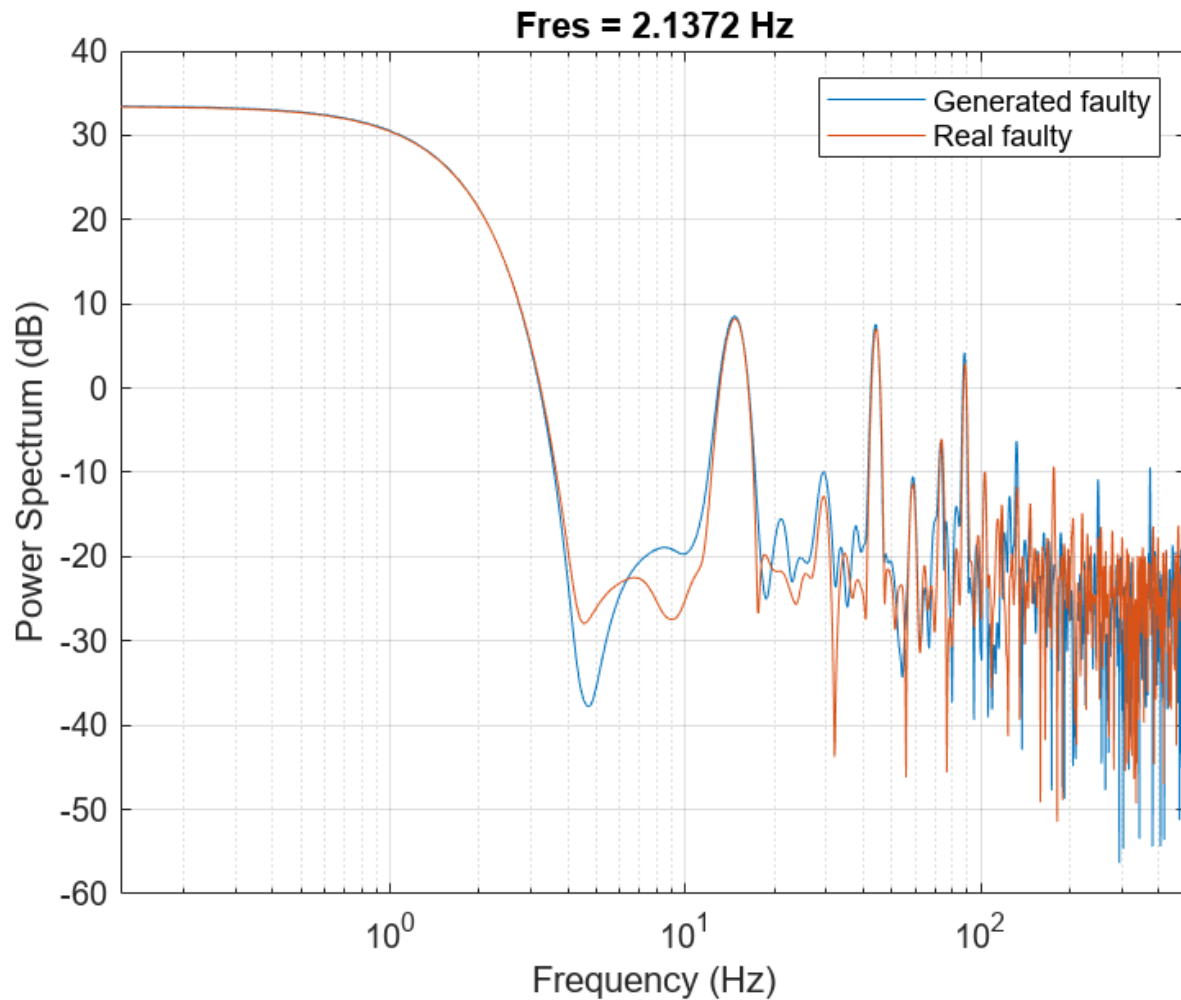
Plot spectra for another case of real and generated faulty signals. The spectral characteristics of the generated faulty signals do not match the theoretical analysis very well and are different from the real faulty signal. The CGAN can still be possibly improved by tuning the network structure or hyperparameters.

```
pspectrum([x(:,1001) x(:,2600)],Fs)
set(gca,'XScale','log')
legend('Generated faulty','Real faulty')
```
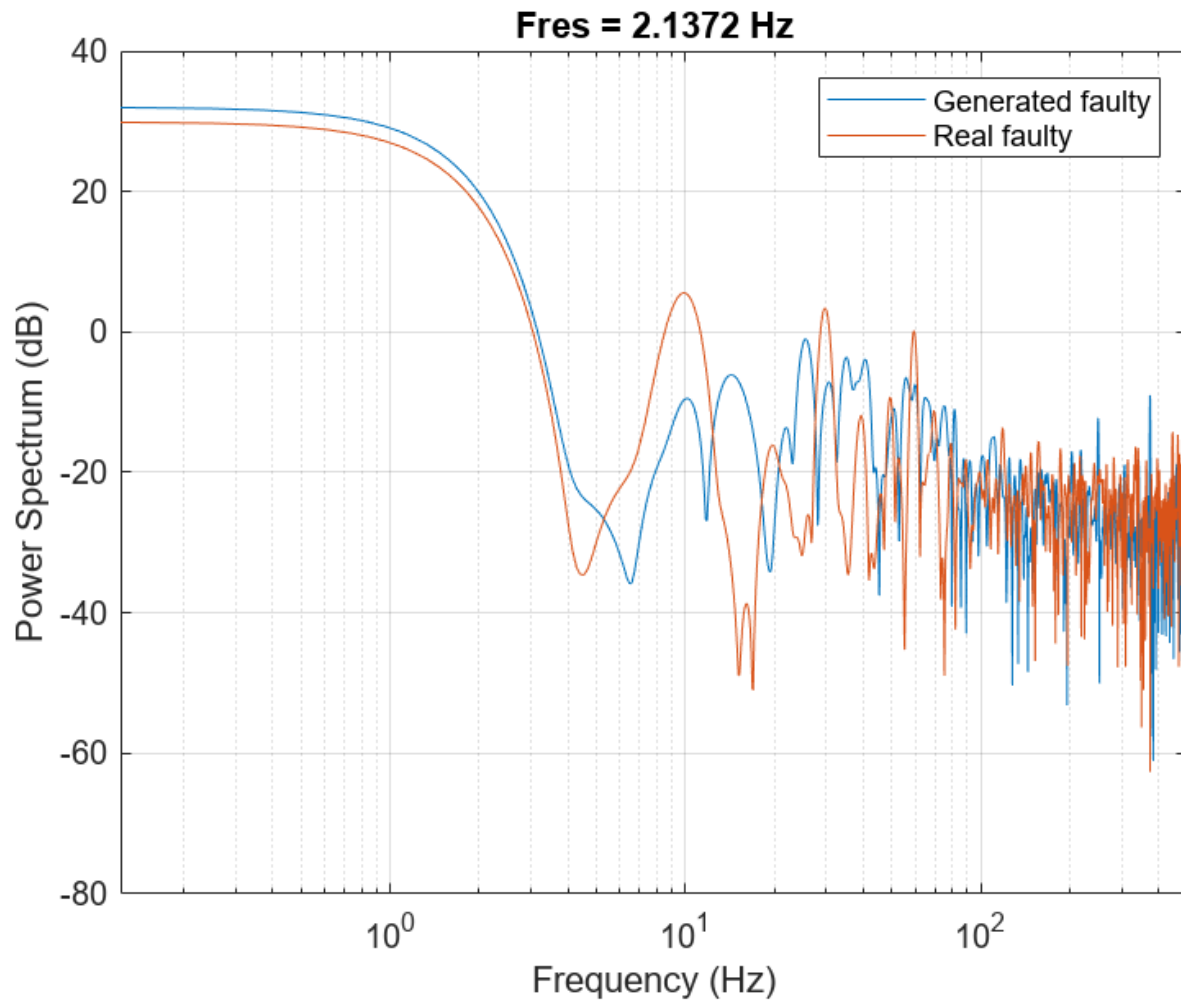
Computation Time

The Simulink simulation takes about 14 hours to generate 2000 pump flow signals. This duration can be reduced to about 1.7 hours with eight parallel workers if you have Parallel Computing Toolbox™.

The CGAN takes 1.5 hours to train and 70 seconds to generate the same amount of synthetic data with an NVIDIA Titan V GPU.

See Also

adamupdate | dlarray | dlfeval

Related Examples

- "Train Conditional Generative Adversarial Network (CGAN)" (Deep Learning Toolbox)
- "Monitor GAN Training Progress and Identify Common Failure Modes" (Deep Learning Toolbox)
- "Multi-Class Fault Detection Using Simulated Data" on page 1-43

Preprocess Data

Data Preprocessing for Condition Monitoring and Predictive Maintenance

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

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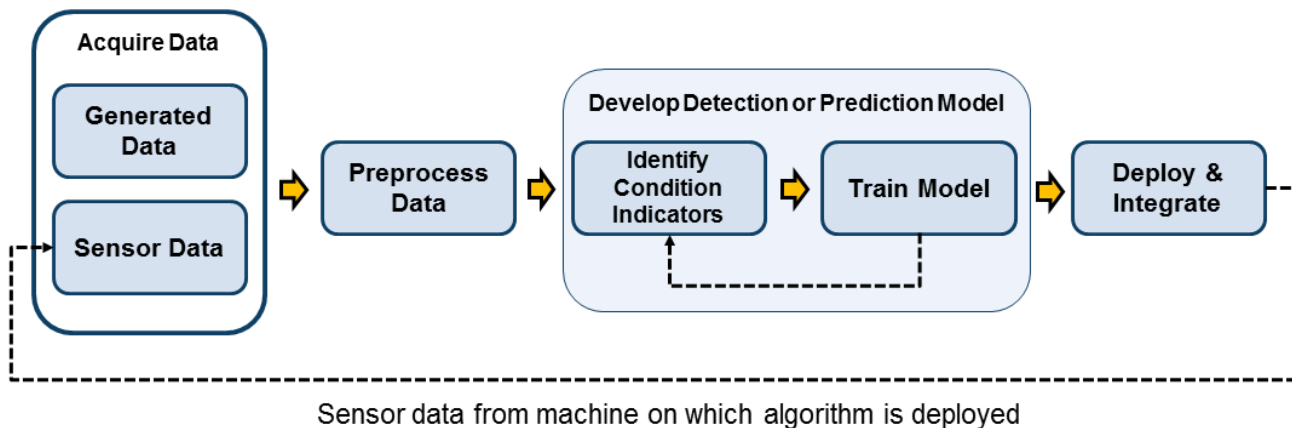
Data preprocessing is often necessary to clean the data and convert it into a form from which you can extract condition indicators. Data preprocessing can include:

- 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 1-25.

For more information on basic preprocessing commands in MATLAB, see "Preprocessing Data".

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".

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 1-25 uses time-synchronous averaging to preprocess vibration data.

- `tsadifference` — Remove the regular signal, the first-order sidebands and other 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 4-6 uses an envelope spectrum for such preprocessing.

For more information on these and related functions, see "Vibration Analysis".

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 1-43 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 4-6 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".

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 5-54 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 4-6 uses spectral kurtosis for preprocessing and extraction of condition indicators.

For more information on these and related functions, see "Time-Frequency Analysis".

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
- "Explore Ensemble Data and Compare Features Using Diagnostic Feature Designer" on page 7-2

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.

Sensor data from machine on which algorithm is deployed
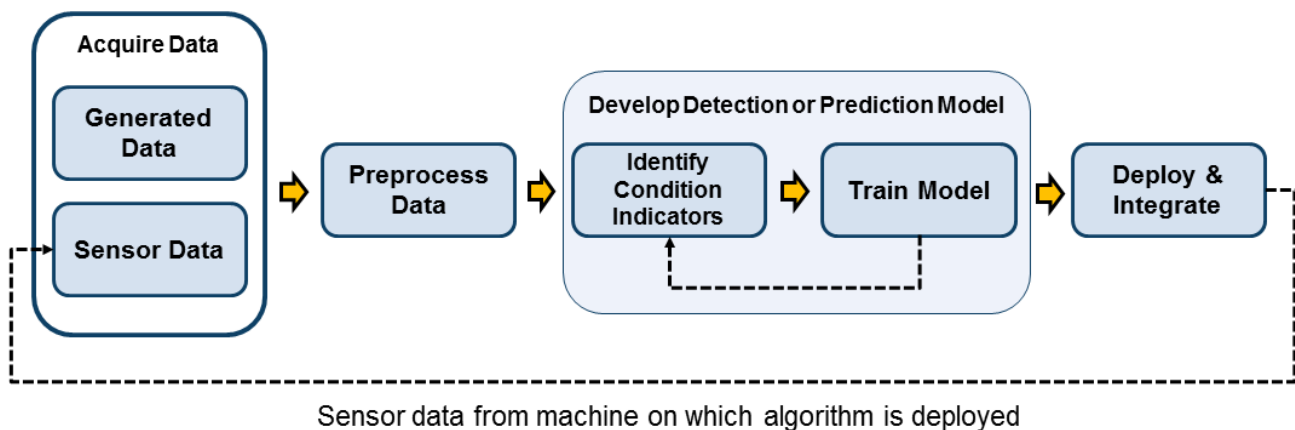
You use condition indicators extracted from system data taken under known conditions to train a model that can then diagnose or predict the condition of a system based on new data taken under unknown conditions. In practice, you might need to explore your data and experiment with different condition 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 4-54 and "Using Simulink to Generate Fault Data" on page 1-25 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 4-6 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 1-43.

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 3-7

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 3-7
- "Designing Algorithms for Condition Monitoring and Predictive Maintenance"
- "Explore Ensemble Data and Compare Features Using Diagnostic Feature Designer" on page 7-2

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 1-25 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 5-54 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 4-6 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 3-7
- "Explore Ensemble Data and Compare Features Using Diagnostic Feature Designer" on page 7-2

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 4-29 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 4-101 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 4-54 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 4-84 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 4-72
- "Nonlinear State Estimation of a Degrading Battery System" on page 5-69

See Also

More About

- "Condition Indicators for Monitoring, Fault Detection, and Prediction" on page 3-2
- "Signal-Based Condition Indicators" on page 3-4
- "Explore Ensemble Data and Compare Features Using Diagnostic Feature Designer" on page 7-2

Condition Indicators for Gear Condition Monitoring

Gear condition monitoring metrics are very important for gearbox development and its time-based preventive maintenance. The indicators enable detection of gear anomalies, and help prevent catastrophic failure before the fault progresses. Condition monitoring systems deal with various types of input data, for instance vibration, acoustic emission, temperature, oil debris analysis. Systems based on vibration analysis, acoustic emission and oil debris are the most common types.

The following figure illustrates the workflow of identifying gear condition metrics and their further evaluation.
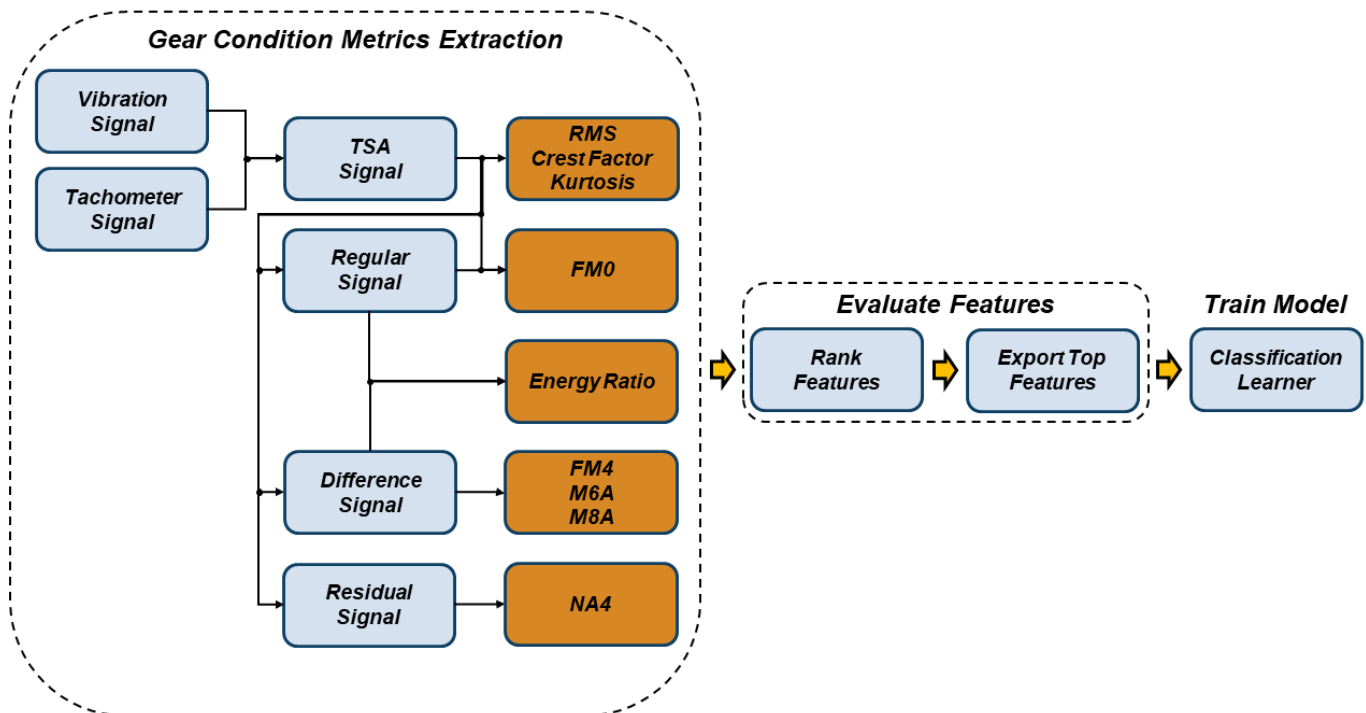
You can use the **Diagnostic Feature Designer** app and command-line functionality in the Predictive Maintenance Toolbox to do the following:

- Import measured or simulated data from individual files, an ensemble file, or an ensemble datastore.
- Derive new variables such as time-synchronous averaged (TSA), regular, residual and difference signals.
- Generate gear condition metrics from your variables.
- Rank your features so that you can determine numerically which ones are likely to best discriminate between nominal and faulty behavior.
- Export your most effective features directly to the **Classification Learner** app for more insight into feature effectiveness and for algorithm training.

Extract Gear Condition Metrics

From your original data, derive the signals and extract the gear condition metrics in the following way:

1 Extract the time-synchronous averaged (TSA).

Diagnostic Feature Designer app	Command-line
Use the **Time-Synchronous Signal Averaging** option from the **Filtering & Averaging** dropdown menu.	Use the `tsa` and `tachorpm` functions.

2 Derive the regular, residual and difference signals.

Diagnostic Feature Designer app	Command-line
Use the **Filter Time-Synchronous Averaged Signals** option from the **Filtering & Averaging** dropdown menu.	Use the following functions: • `tsaregular` • `tsaresidual` • `tsadifference`

3 Compute gear condition monitoring metrics from the set of signals obtained from the previous step.

Diagnostic Feature Designer app	Command-line
Use the **Rotating Machinery Features** option from the **Time-Domain Features** dropdown menu.	Use the `gearConditionMetrics` function.

Gear condition metrics that can identify the precise location of faults, include the following:

Computed from TSA Signal

- `Root-Mean Square (RMS)` — Indicates the general condition of the gearbox in later stages of degradation. RMS is sensitive to gearbox load and speed changes. RMS is usually a good indicator of the overall condition of gearboxes, but not a good indicator of incipient tooth failure. It is also useful to detect unbalanced rotating elements. RMS of a standard normal distribution is 1.

- `Kurtosis` — Fourth order normalized moment of the signal that indicates major peaks in the amplitude distribution. Kurtosis is a measure of how outlier-prone a distribution is. The kurtosis of the standard normal distribution is 3. Distributions that are more outlier-prone have kurtosis values greater than 3; distributions that are less outlier-prone have kurtosis values less than 3. `Kurtosis` values are higher for damaged gear trains due to sharp peaks in the amplitude distribution of the signal.

- `Crest Factor (CF)` — Ratio of signal peak value to RMS value that indicates early signs of damage, especially where vibration signals exhibit impulsive traits.

Computed from Difference Signal

- FM4 — Describes how peaked or flat the difference signal amplitude is. FM4 is normalized by the square of the variance, and detects faults isolated to only a finite number of teeth in a gear mesh. FM4 of a standard normal distribution is 3.

- M6A — Describes how peaked or flat the difference signal amplitude is. M6A is normalized by the cube of the variance, and indicates surface damage on the rotating machine components. M6A of a standard normal distribution is 15.

- M8A — An improved version of the M6A indicator. M8A is normalized by the fourth power of the variance. M8A of a standard normal distribution is 105.

Computed from a Mix of Signals

- FM0 — Compares ratio of peak value of TSA signal to energy of regular signal. FM0 identifies major anomalies, such as tooth breakage or heavy wear, in the meshing pattern of a gear.

- Energy Ratio (ER) — Ratio between energy of the difference signal and the energy of the regular meshing component. Energy Ratio indicates heavy wear, where multiple teeth on the gear are damaged.

Computed from a Set of Residual Signals

- NA4 — An improved version of the FM4 indicator. NA4 indicates the onset of damage and continues to react to the damage as it spreads and increases in magnitude.

Evaluate Features and Train Model

Feature selection techniques help you reduce large data sets by eliminating gear condition metrics 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 gear metrics that are suitable to serve as condition indicators because they change in a detectable, reliable way as a gearbox performance degrades.

To perform a rigorous relative assessment, you can rank your features using specialized statistical methods. Each method scores and ranks features by ability to discriminate between or among data groups, such as between nominal and faulty behavior. The ranking results allow you to eliminate ineffective features and to evaluate the ranking effects of parameter adjustments when computing derived variables or features.

Diagnostic Feature Designer app	Command-line
Histograms allow you to perform an initial assessment of feature effectiveness. To perform a more rigorous relative assessment, you can rank your features, using the **Rank Features** option, by specialized statistical methods. Use the **Export** option to export the selected metrics to the **Classification Learner** app in the Statistics and Machine Learning Toolbox™.	You can choose from the following feature selection functions: - monotonicity - prognosability - trendability - pca - pcares - sequentialfs - fscnca - tsne

Once you have defined your set of candidate gear condition metrics, you can export them to the **Classification Learner** app in the Statistics and Machine Learning Toolbox. **Classification Learner** trains models to classify data, by using automated methods to test different types of models with a

feature set. In doing so, **Classification Learner** determines the best model and the most effective features. For predictive maintenance, the goal of using the **Classification Learner** is to select and train a model that discriminates between data from healthy and from faulty systems. You can incorporate this model into an algorithm for gear train fault detection and prediction.

See Also

Diagnostic Feature Designer | monotonicity | prognosability | trendability | pca | pcares | sequentialfs | fscnca | tsne | gearConditionMetrics | tsa | tachorpm | tsaregular | tsaresidual | tsadifference | **Classification Learner**

More About

- "Explore Ensemble Data and Compare Features Using Diagnostic Feature Designer" on page 7-2
- "Interpret Feature Histograms in Diagnostic Feature Designer" on page 7-13

Motor Current Signature Analysis for Gear Train Fault Detection

This example shows how to identify faults in a gear train using motor current signature analysis (MCSA) of a current signal driving a hobby-grade servo. MCSA is a useful method for the diagnosis of faults that induce torque or speed fluctuations, and has been proven to be ideal for motor fault analysis. Gear fault detection using traditional vibration instruments is challenging, especially in cases where the gear train is not easily accessible for instrumentation with accelerometers or other vibration sensors, like the inner workings of a nuclear power plant. This example illustrates how to apply current signature analysis to extract spectral metrics to detect faults in drive gears of a hobby-grade servo motor. The simplified workflow to obtain the spectral metrics from the current signal is as follows:

1 Compute nominal rpm to detect frequencies of interest.
2 Construct frequency bands where fault signals may be present.
3 Extract power spectral density (PSD) data.
4 Compute spectral metrics at the frequency bands of interest.

Hardware Overview
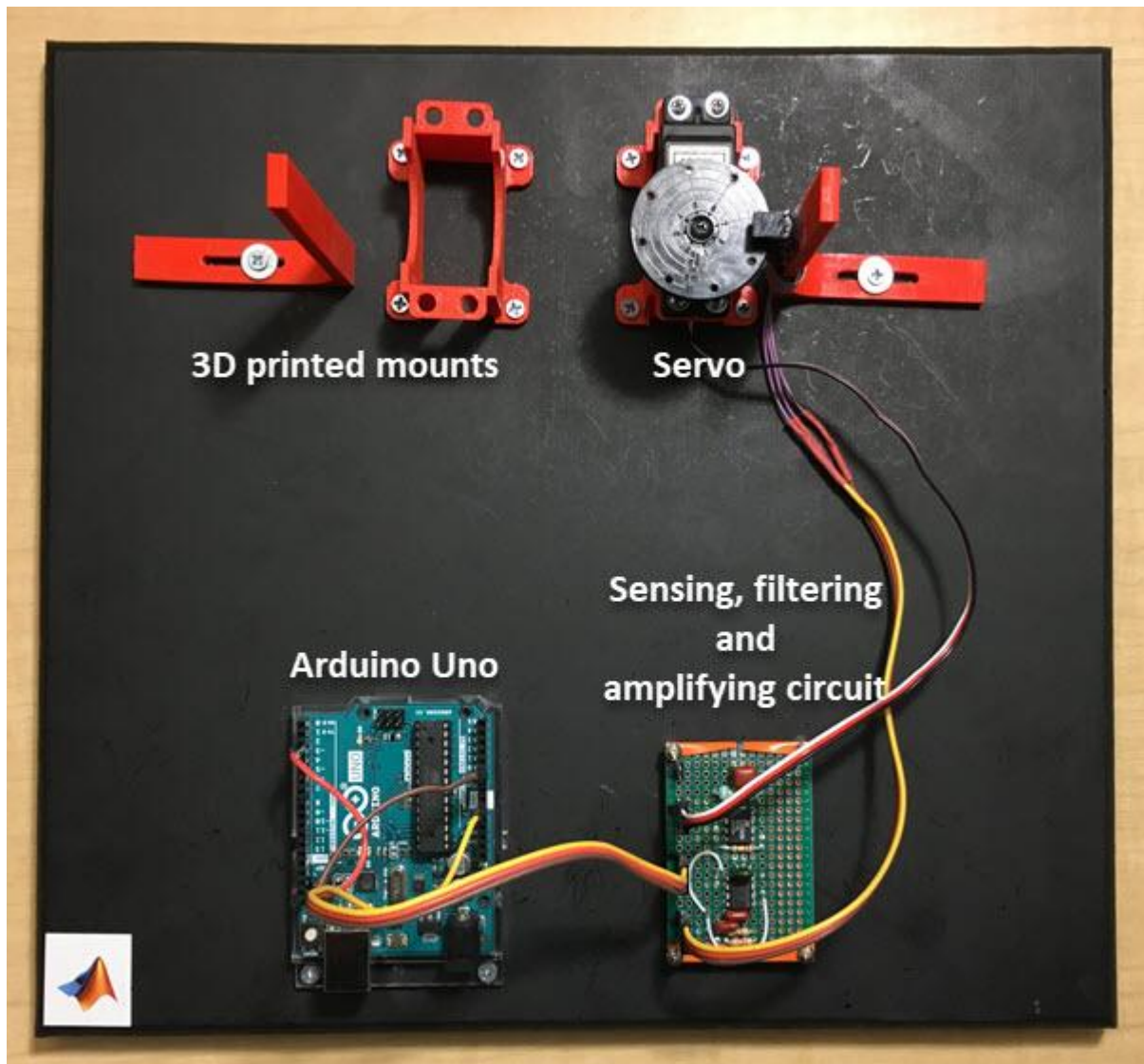
For this example, the electrical current data was collected from a standard Futaba S3003 hobby servo, which was modified for continuous rotation. Servos convert the high speed of the internal DC motor to high torque at the output spline. To achieve this, servos consist of a DC motor, a set of nylon or metal drive gears, and the control circuit. The control circuit was removed to allow the current signal to the DC motor to be directly monitored. The tachometer signal at the output spline of the servo was collected using an infrared photointerrupter along with a 35 mm diameter, eight-holed, standard hobby servo wheel. The eight holes in the wheel were equally spaced and the IR photointerrupter was placed such that there were exactly eight pulses per rotation of the servo wheel horn. The servo and photointerrupter were held in place by custom 3-D printed mounts.
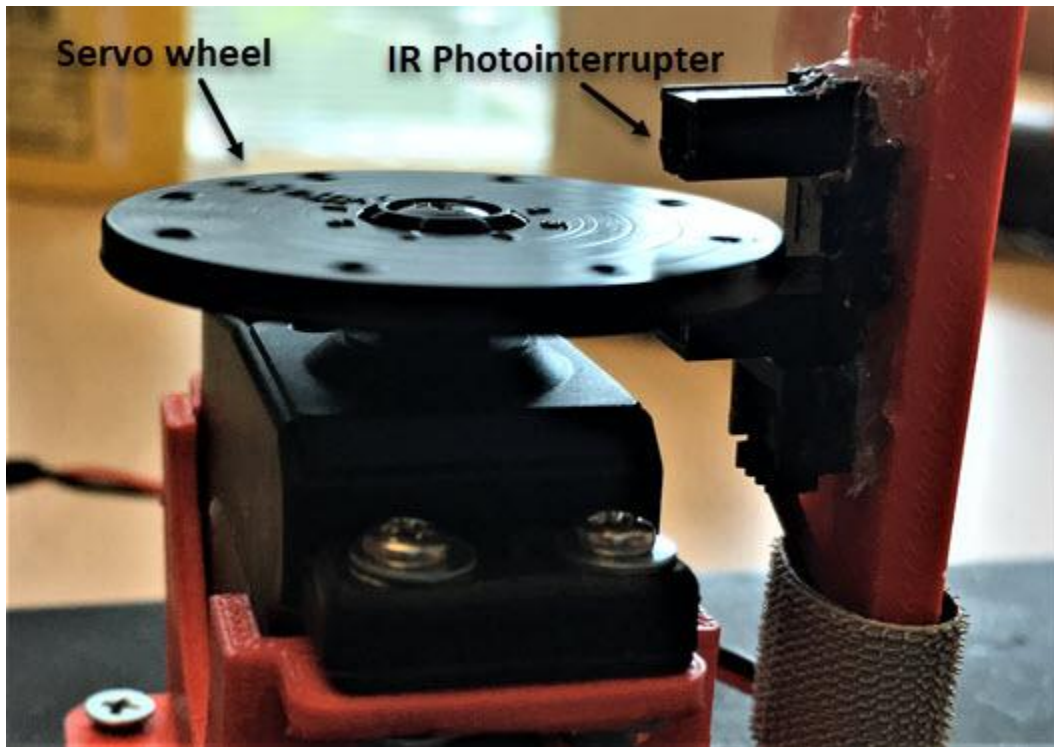
The DC motor was driven at a constant 5 volts, and with four pairs of gears providing 278:1 speed reduction, the shaft speed at the spline was about 50 rpm. The current consumption was calculated using Ohm's law by measuring the voltage drop across a 0.5 Ohm resistor. Since the change in current measurement values was too small for detection, the current signal was amplified using a single-supply sensor interface amplifier. The amplified current signal was then filtered using an anti-aliasing fifth-order elliptic low-pass filter to smooth it and to eliminate noise before sending it to an Arduino Uno through an analog-to-digital converter (ADC).
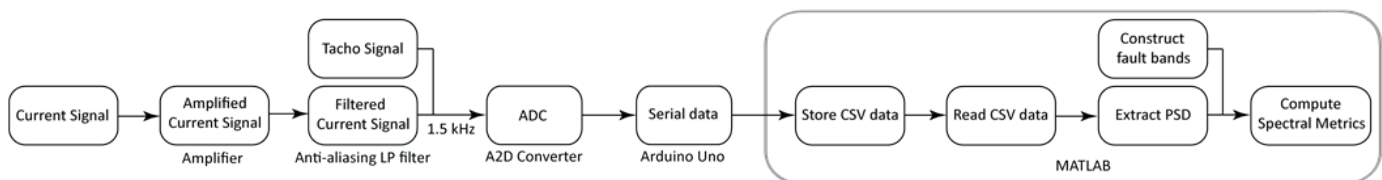
As the flowchart shows, the current signal was first amplified and filtered using the amplifier and the anti-aliasing low-pass filter, respectively. The Arduino Uno sampled the current signal through an ADC at 1.5 kHz and streamed it to the computer along with the tachometer pulses as serial data at a baud rate of 115200. A MATLAB script fetched the serial data from the Arduino Uno and wrote it to a comma-separated values (CSV) file. The CSV files were then read and processed using MATLAB to extract the spectral metrics.

Servo Gear Train

Futaba S3003 standard servo gear train
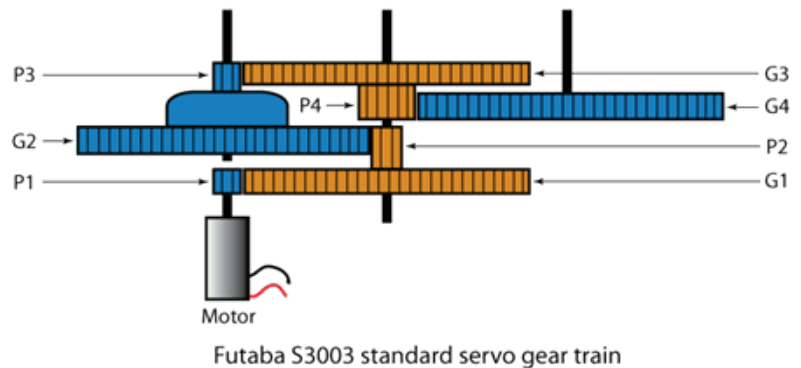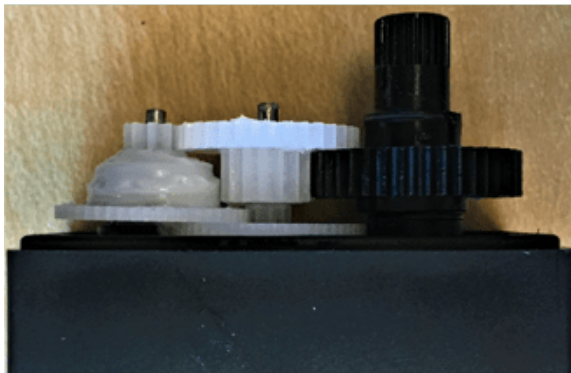
The Futaba S3003 servo consists of four pairs of nylon gears as illustrated in the above figure. The pinion P1 on the DC motor shaft meshes with the stepped gear G1. The pinion P2 is a molded part of the stepped gear G1 and meshes with the stepped gear G2. The pinion P3, which is a molded part of gear G2, meshes with the stepped gear G3. Pinion P4, which is molded with G3, meshes with the final gear G4 that is attached to the output spline. The stepped gear sets G1 and P2, G2 and P3, and G3 and P4 are free spinning gears — that is, they are not attached to their respective shafts. The set of drive gears provides a 278:1 reduction going from a motor speed of 13901 rpm to about 50 rpm at the output spline when the motor is driven at 5 volts. The following table outlines the tooth count and theoretical values of output speed, gear mesh frequencies, and cumulative gear reduction at each gear mesh.

Pinion	Gear	Pinion Teeth	Gear Teeth	Output Speed		Gear Mesh Frequency (Hz)	Cumulative Gear Reduction
				(RPM)	(Hz)		
P1		10		13901.6	231.69		1
P2	G1	10	62	2242.2	37.37	2316.9	6.2
P3	G2	10	50	448.5	7.47	373.7	31
P4	G3	16	35	128.1	2.14	74.7	108.5
	G4		41	50	0.83	34.2	278

Theoretical values calculated from tooth count assuming 50 RPM at the output shaft

A total of 10 healthy data sets were collected before the faults were introduced in the stepped gears G2 and G3. Since the gears were molded out of nylon, simulated cracks were introduced in both the gears by cutting slots in the tooth space with a hobby knife. The tooth space is the gap between two adjacent teeth measured along the pitch circle of the spur gear. The slot depths were about 70 percent of the gear radius. A total of 10 faulty data sets were recorded after introducing the faults in the gears G2 and G3.

Visualize Data

The file `mcsaData.mat` contains `servoData`, a 10-by-2 table of timetables where each timetable corresponds to one data set. The first column of `servoData` contains 10 timetables of healthy data while the second column contains 10 timetables of faulty data. Each data set contains about 11 seconds of data sampled at 1500 Hz.

Load the data.

```
load('mcsaData.mat','servoData')
servoData
```

```
servoData=10×2 table
        healthyData             faultyData

    _____    _____

    {16384x2 timetable}     {16384x2 timetable}
    {16384x2 timetable}     {16384x2 timetable}
    {16384x2 timetable}     {16384x2 timetable}
    {16384x2 timetable}     {16384x2 timetable}
    {16384x2 timetable}     {16384x2 timetable}
    {16384x2 timetable}     {16384x2 timetable}
    {16384x2 timetable}     {16384x2 timetable}
    {16384x2 timetable}     {16384x2 timetable}
    {16384x2 timetable}     {16384x2 timetable}
    {16384x2 timetable}     {16384x2 timetable}
```

```
head(servoData.healthyData{1,1})
```

```
        Time            Pulse     Signal

    _____     _____     _____

    0 sec                 0       66.523
    0.00066667 sec        0       62.798
    0.0013333 sec         0       63.596
    0.002 sec             0       64.128
    0.0026667 sec         0       60.669
```

```
    0.0033333 sec        0       62.798
    0.004 sec            0       65.459
    0.0046667 sec        0       56.678
```

Each timetable in servoData contains one data set with the tachometer signal in the first column and the current signal in the second column.

For this example, consider one set each of healthy and faulty data, and visualize the current signal and tachometer pulses.

```matlab
healthyData = servoData.healthyData{1,1};
faultyData = servoData.faultyData{1,1};
healthyCurrent = healthyData.Signal;
faultyCurrent = faultyData.Signal;
healthyTacho = healthyData.Pulse;
faultyTacho = faultyData.Pulse;
tHealthy = healthyData.Time;
tFaulty = faultyData.Time;

figure
ax1 = subplot(221);
plot(tHealthy,healthyCurrent)
title('Current Signal - Healthy Gears')
ylabel('Current (mA)')
ax2 = subplot(222);
plot(tFaulty,faultyCurrent)
title('Current Signal - Faulty Gears')
ylabel('Current (mA)')
ax3 = subplot(223);
plot(tHealthy,healthyTacho)
title('Tachometer Pulse - Healthy Gears')
ylabel('Pulses, 8/rev')
ax4 = subplot(224);
plot(tFaulty,faultyTacho)
title('Tachometer Pulse - Faulty Gears')
ylabel('Pulses, 8/rev')
linkaxes([ax1,ax2,ax3,ax4],'x');
ax1.XLim = seconds([0 2]);
```
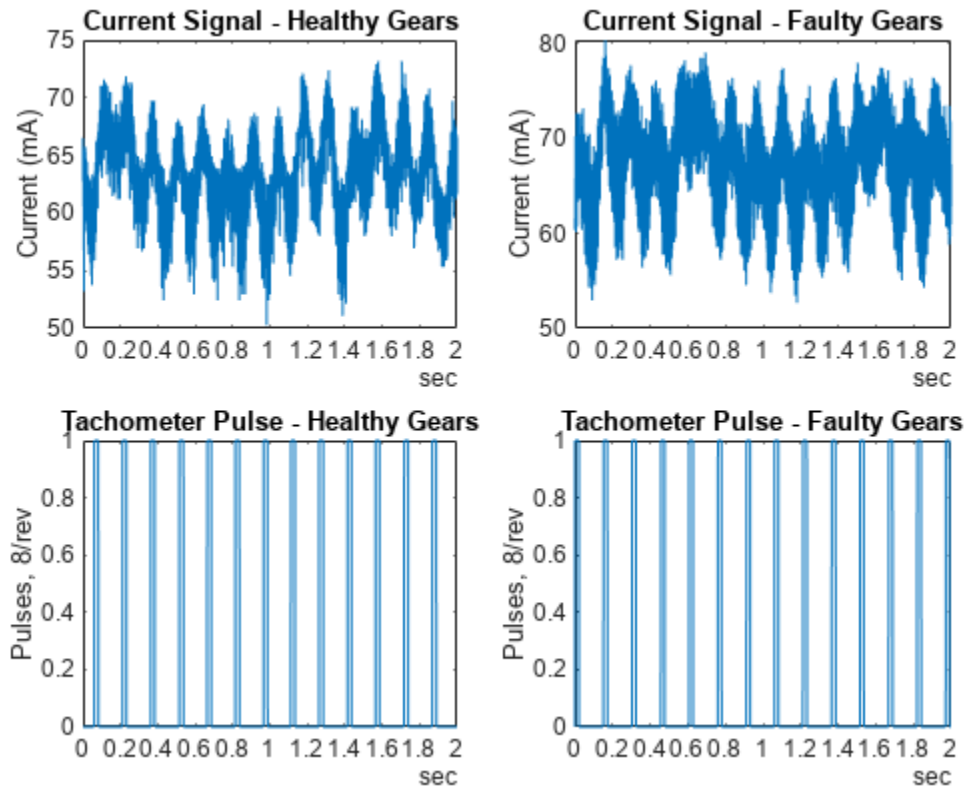
The output spline of the servo has eight pulses per rotation, corresponding to the eight holes on the servo wheel.

Compute Nominal Rpm

Compute the nominal speed to detect frequencies of interest in the gear system and match them correctly with the frequencies on the power spectra. Using the sampling frequency value of 1500 Hz, visualize the tachometer signal and compute the nominal rpm using `tachorpm`.

```
fs = 1500;
figure
tachorpm(healthyTacho,fs,'PulsesPerRev',8)
```
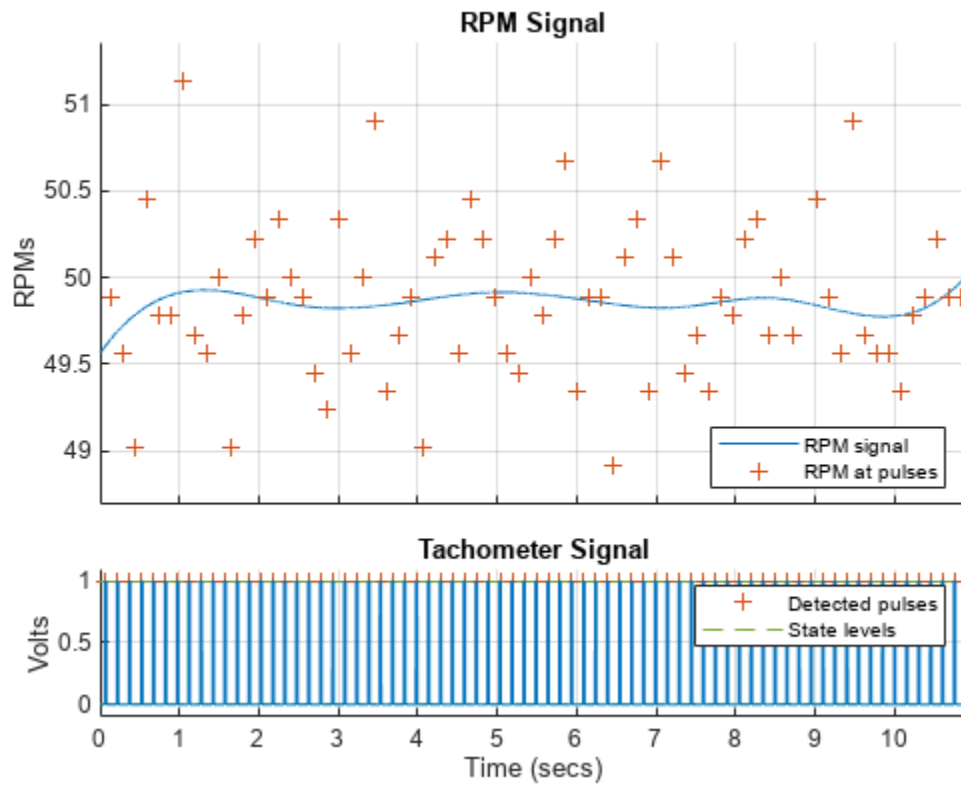
```
figure
tachorpm(faultyTacho,fs,'PulsesPerRev',8)
```
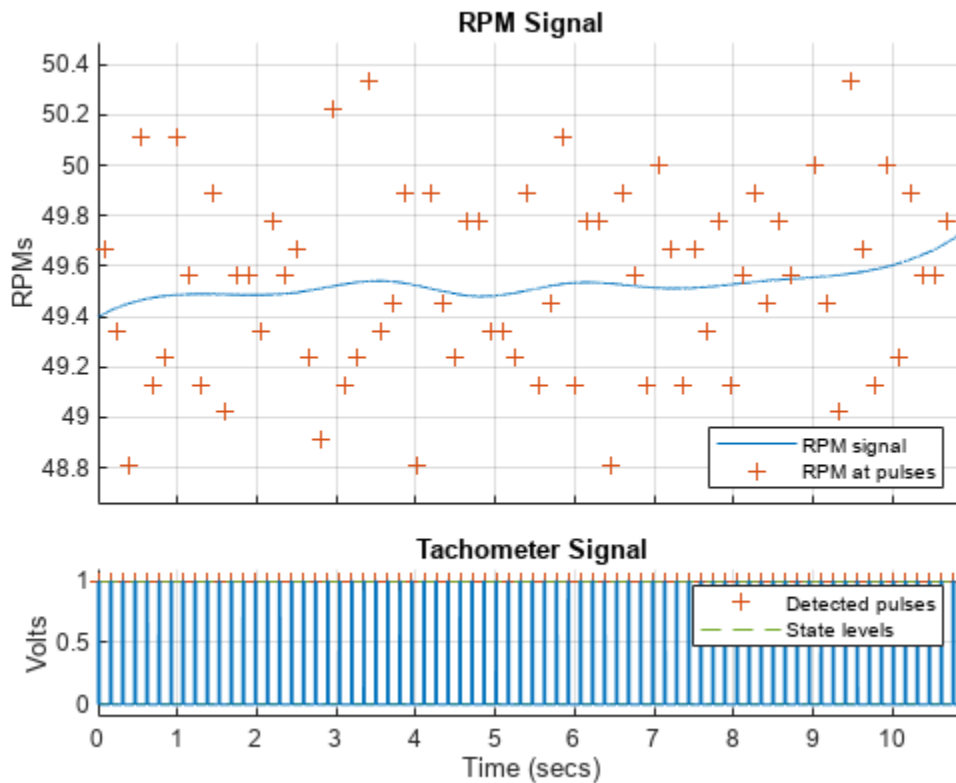
```
rpmHealthy = mean(tachorpm(healthyTacho,fs,'PulsesPerRev',8))
```

rpmHealthy = 49.8550

```
rpmFaulty = mean(tachorpm(faultyTacho,fs,'PulsesPerRev',8))
```

rpmFaulty = 49.5267

Observe that there is a very small difference in the output shaft speed between the healthy and faulty data sets. The actual nominal rpm values are also close to the theoretical value of 50 rpm. Hence, consider the same value of 49.9 rpm for both the healthy and faulty signal analysis.

Construct Frequency Bands

Constructing frequency bands is an important prerequisite for computing the spectral metrics. Using the tooth count of the drive gears in the gear train and the nominal rpm, first compute the frequencies of interest. The frequencies of interest are actual output speed values in hertz whose values are close to the theoretical values listed in the table below.

Pinion	Gear	Pinion Teeth	Gear Teeth	Output Speed		Gear Mesh Frequency (Hz)	Cumulative Gear Reduction
				(RPM)	(Hz)		
P1		10		13901.6	231.69		1
P2	G1	10	62	2242.2	37.37	2316.9	6.2
P3	G2	10	50	448.5	7.47	373.7	31
P4	G3	16	35	128.1	2.14	74.7	108.5
	G4		41	50	0.83	34.2	278

Theoretical values calculated from tooth count assuming 50 RPM at the output shaft

```
G4 = 41; G3 = 35; G2 = 50; G1 = 62;
P4 = 16; P3 = 10; P2 = 10; P1 = 10;
rpm = rpmHealthy;

FS5 = mean(rpm)/60;
FS4 = G4/P4*FS5;
FS3 = G3/P3*FS4;
FS2 =  G2/P2*FS3;
FS1 =  G1/P1*FS2;
FS = [FS1,FS2,FS3,FS4,FS5]
```

```
FS = 1×5

  231.0207   37.2614    7.4523    2.1292    0.8309
```

Next, construct the frequency bands for the all the output speeds which include the following frequencies of interest using `faultBands`.

- FS1 at 231 Hz, its first two harmonics, and 0:1 sidebands of FS2

- FS2 at 37.3 Hz, its first two harmonics, and 0:1 sidebands of FS3

- FS3 at 7.5 Hz and its first two harmonics

- FS4 at 2.1 Hz and its first two harmonics

```
[FB1,info1] = faultBands(FS1,1:2,FS2,0:1);
[FB2,info2] = faultBands(FS2,1:2,FS3,0:1);
[FB3,info3] = faultBands(FS3,1:2);
[FB4,info4] = faultBands(FS4,1:2);
FB = [FB1;FB2;FB3;FB4]
```

```
FB = 16×2

  187.9838   199.5348
  225.2452   236.7962
  262.5066   274.0577
  419.0045   430.5556
  456.2659   467.8170
  493.5273   505.0784
   28.8776    30.7407
```

```
36.3299    38.1929
43.7822    45.6452
66.1390    68.0021
    ⋮
```

The output FB is a 16-by-2 array of frequency ranges for these frequencies of interest.

Extract Power Spectral Density (PSD) Data

Compute and visualize the power spectrum of the healthy and faulty data. Also plot the frequencies of interest on the spectrum plot by using the information in the structure info.

```
figure
pspectrum(healthyCurrent,fs,'FrequencyResolution',0.5)
hold on
pspectrum(faultyCurrent,fs,'FrequencyResolution',0.5)
hold on
info1.Labels = regexprep(info1.Labels,'F0','FS1');
info1.Labels = regexprep(info1.Labels,'F1','FS2');
helperPlotXLines(info1,[0.5 0.5 0.5])
info2.Labels = regexprep(info2.Labels,'F0','FS2');
info2.Labels = regexprep(info2.Labels,'F1','FS3');
helperPlotXLines(info2,[0.5 0.5 0.5])
info3.Labels = regexprep(info3.Labels,'F0','FS3');
helperPlotXLines(info3,[0.5 0.5 0.5])
info4.Labels = regexprep(info4.Labels,'F0','FS4');
helperPlotXLines(info4,[0.5 0.5 0.5])
legend('Healthy Data','Faulty Data','Location','South')
hold off
```
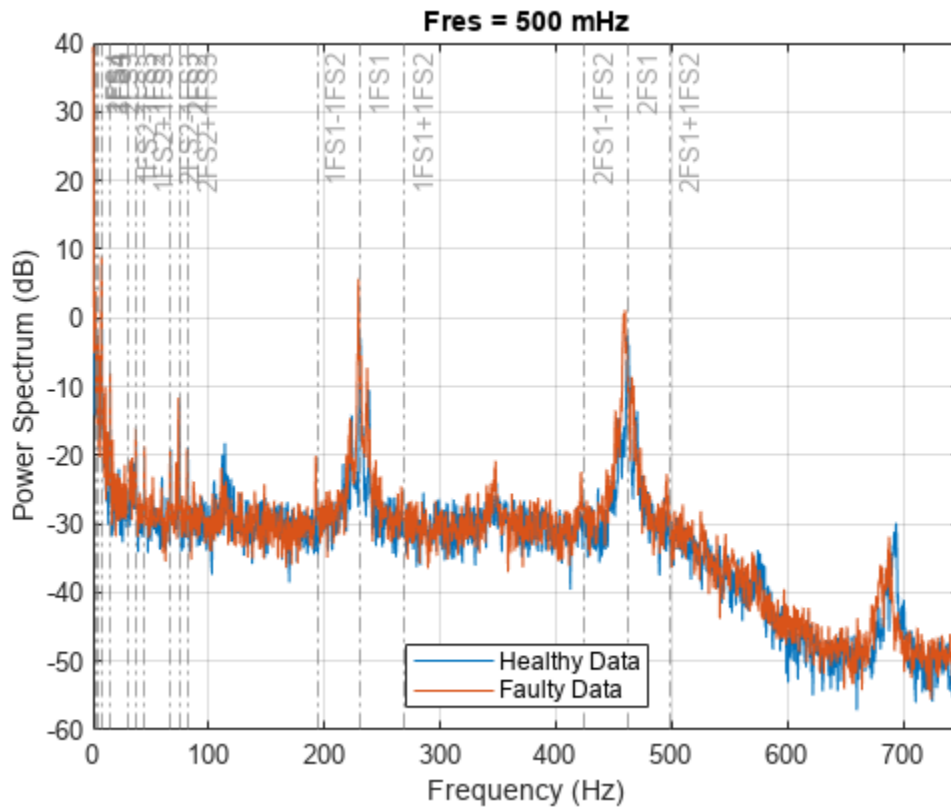
The plot in blue indicates the healthy spectrum while the plot in red indicates the spectrum of the faulty data. From the plot, observe the increase in amplitudes of fault frequencies:

- 1FS1 at 231 Hz, its second harmonic 2FS1 at 462 Hz, and their respective sidebands

Zoom in on the plot to observe the increase in amplitudes of the following remaining frequencies:

- 1FS2 at 37.2 Hz and its sidebands
- 1FS3 at 7.5 Hz and its second harmonic 2FS3 at 15 Hz
- 1FS4 at 2.1 Hz and its second harmonic 2FS4 at 4.2 Hz
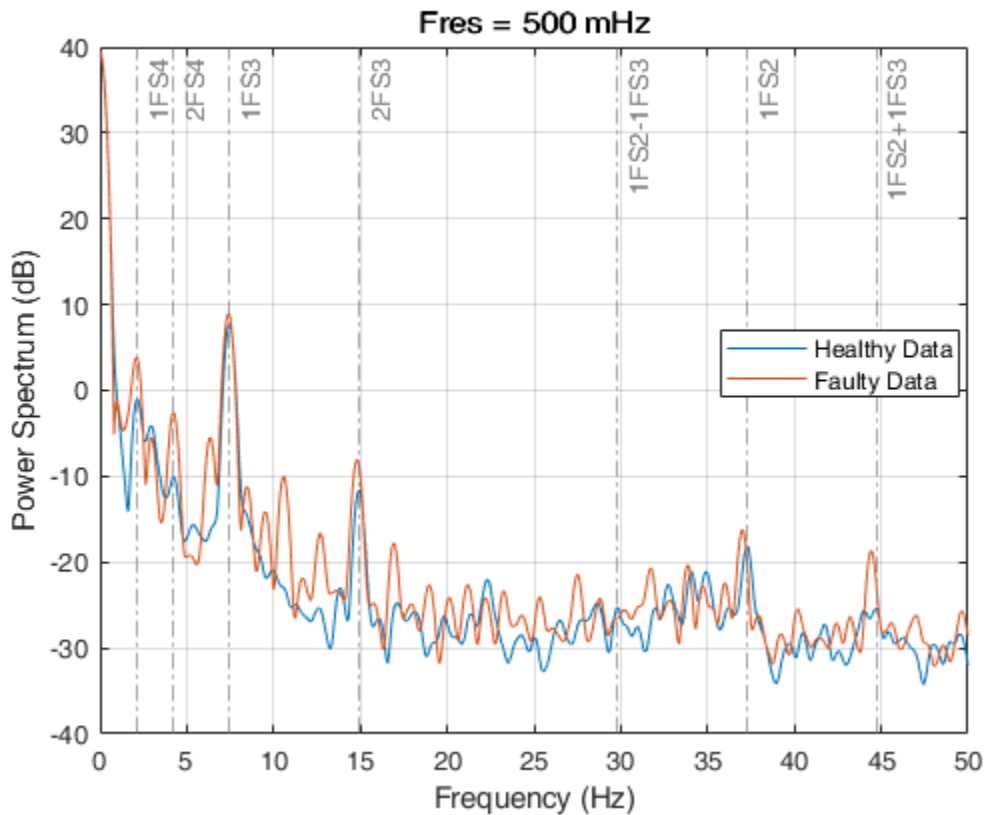
Use pspectrum to compute and store the PSD and the frequency grid data for the healthy and faulty signals respectively.

```
[psdHealthy,fHealthy] = pspectrum(healthyCurrent,fs,'FrequencyResolution',0.5);
[psdFaulty,fFaulty] = pspectrum(faultyCurrent,fs,'FrequencyResolution',0.5);
```

Compute Spectral Metrics

Using the frequency bands and the PSD data, compute the spectral metrics for the healthy and faulty data using faultBandMetrics.

```
spectralMetrics = faultBandMetrics({[psdHealthy,fHealthy],[psdFaulty,fFaulty]},FB)
```

```
spectralMetrics=2×49 table
    PeakAmplitude1    PeakFrequency1    BandPower1    PeakAmplitude2    PeakFrequency2    BandPow
    _____    _____    _____    _____    _____    _____

       0.0020712          193.75         0.010881        0.50813            231.06         0.4665
       0.009804           192.44         0.017916        3.6921             229.44         2.997
```

The output is a 2-by-49 table of metrics for the frequency ranges in FB. The first row contains the metrics for healthy data while the second row contains the faulty data metrics. Observe that the following metrics have considerably higher values for the faulty data than for the healthy data:

- PeakAmplitude2 for the frequency at 231 Hz with a difference of 3.1842 units
- TotalBandPower with a difference of 6.01 units

Hence, use these two metrics to create a scatter plot to group the faulty and healthy data sets separately.

Create Scatter Plot

Create a scatter plot to classify healthy and faulty data using the two spectral metrics `PeakAmplitude2` and `TotalBandPower` for all 20 data sets in the table `servoData`.

```
plotData = zeros(10,4);
for n = 1:max(size(servoData))
    hC = servoData.healthyData{n,1}.Signal;
    fC = servoData.faultyData{n,1}.Signal;

    [psdH,freqH] = pspectrum(hC,fs,'FrequencyResolution',0.5);
    [psdF,freqF] = pspectrum(fC,fs,'FrequencyResolution',0.5);

    sMetrics = faultBandMetrics({[psdH,freqH],[psdF,freqF]},FB);
    plotData(n,:) = [sMetrics{:,4}',sMetrics{:,49}'];
end
figure
scatter(plotData(:,1),plotData(:,3),[],'blue')
hold on;
scatter(plotData(:,2),plotData(:,4),[],'red')
legend('Healthy Data','Faulty Data','Location','best')
xlabel('Peak Amplitude 2')
ylabel('Total Band Power')
hold off
```
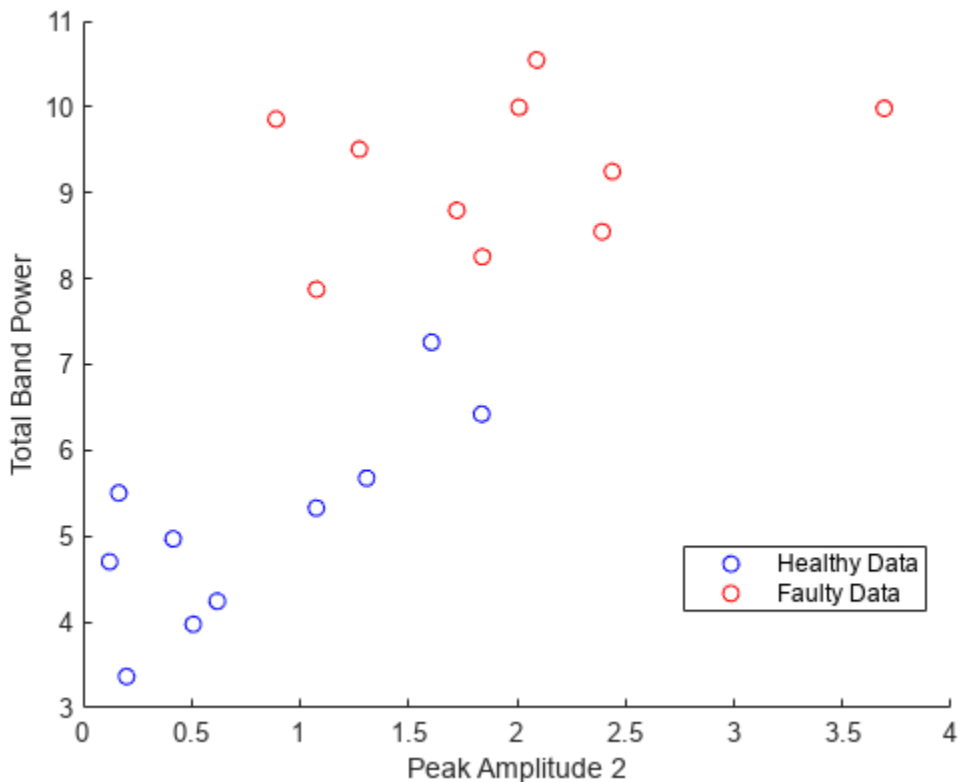
Observe that the healthy data sets and the faulty data sets are grouped in different areas of the scatter plot.

Hence, you can classify faulty and healthy data by analyzing the current signature of the servo motor.

To use all the available spectral metrics for classification, use Classification Learner.

Helper Function

The helper function `helperPlotXLines` uses the information in the structure `info` to plot the frequency band center lines on the power spectrum plot.

```
function helperPlotXLines(I,c)
for k = 1:length(I.Centers)
    xline(I.Centers(k),'Label',I.Labels(k),'LineStyle','-.','Color',c);
end
end
```

References

[1] Moster, P.C. "Gear Fault Detection and Classification Using Learning Machines." *Sound & vibration*. 38. 22-27. 2004

See Also

Classification Learner | `faultBands` | `faultBandMetrics` | `pspectrum`

Reconstruct Phase Space and Estimate Condition Indicators Using Live Editor Tasks

This example shows how use Live Editor tasks to reconstruct phase space of a uniformly sampled signal and then use the reconstructed phase space to estimate the correlation dimension and the Lyapunov exponent.

Live Editor tasks let you interactively iterate on parameters and settings while observing their effects on the result of your computation. The tasks then and automatically generate MATLAB® code that achieves the displayed results. To experiment with the Live Editor tasks in this script, open this example.

For more information about Live Editor Tasks generally, see "Add Interactive Tasks to a Live Script".

Load Data

In this example, assume that you have measurements for a Lorenz Attractor. Your measurements are along the x-direction only, but the attractor is a three-dimensional system. Using this limited data, reconstruct the phase space such that the properties of the original three-dimensional system are recovered.

Load the Lorenz Attractor data, and visualize its `x`, `y` and `z` measurements on a 3-D plot. Since the Lorenz attractor has 3 dimensions, specify `dim` as 3.

```
load('lorenzAttractorExampleData.mat','data','fs')
X = data(:,1);
plot3(data(:,1),data(:,2),data(:,3));
```
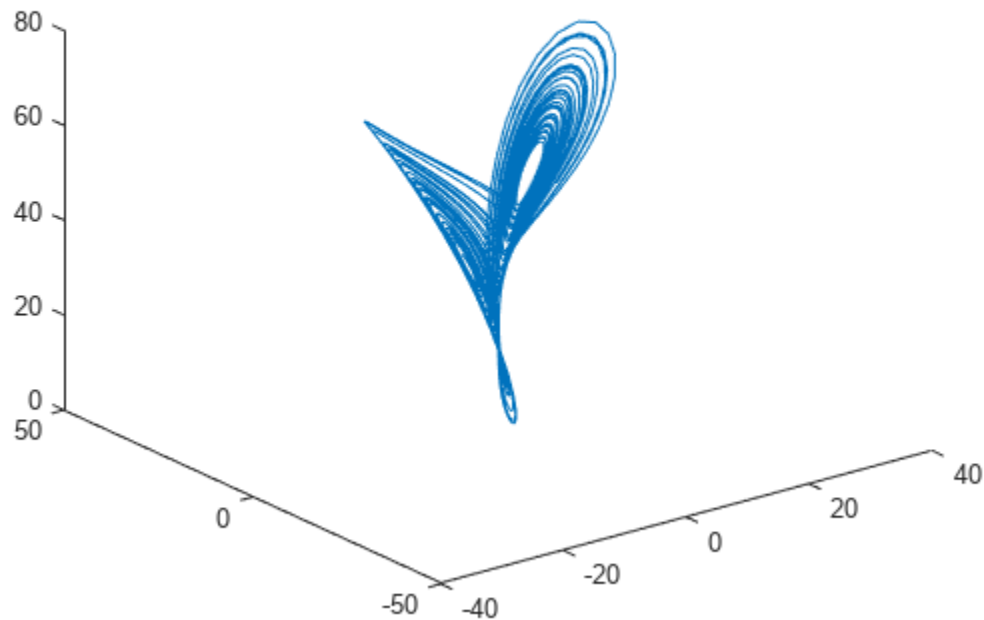
Reconstruct Phase Space

To reconstruct the phase space data, use the Reconstruct Phase Space Live Editor Task. You can insert a task into your script using the **Task** menu in the Live Editor. In this script, **Reconstruct Phase Space** is already inserted. Open the example to experiment with the task.

To perform the phase space reconstruction, in the task, specify the signal you loaded, X and the embedding dimension as 3. In the **Reconstruct Phase Space** task, you can experiment with different lag and embedding dimension values and observe the reconstructed Lorenz attractor displayed in the output plot. For details about the available options and parameters, see the Reconstruct Phase Space task reference page.
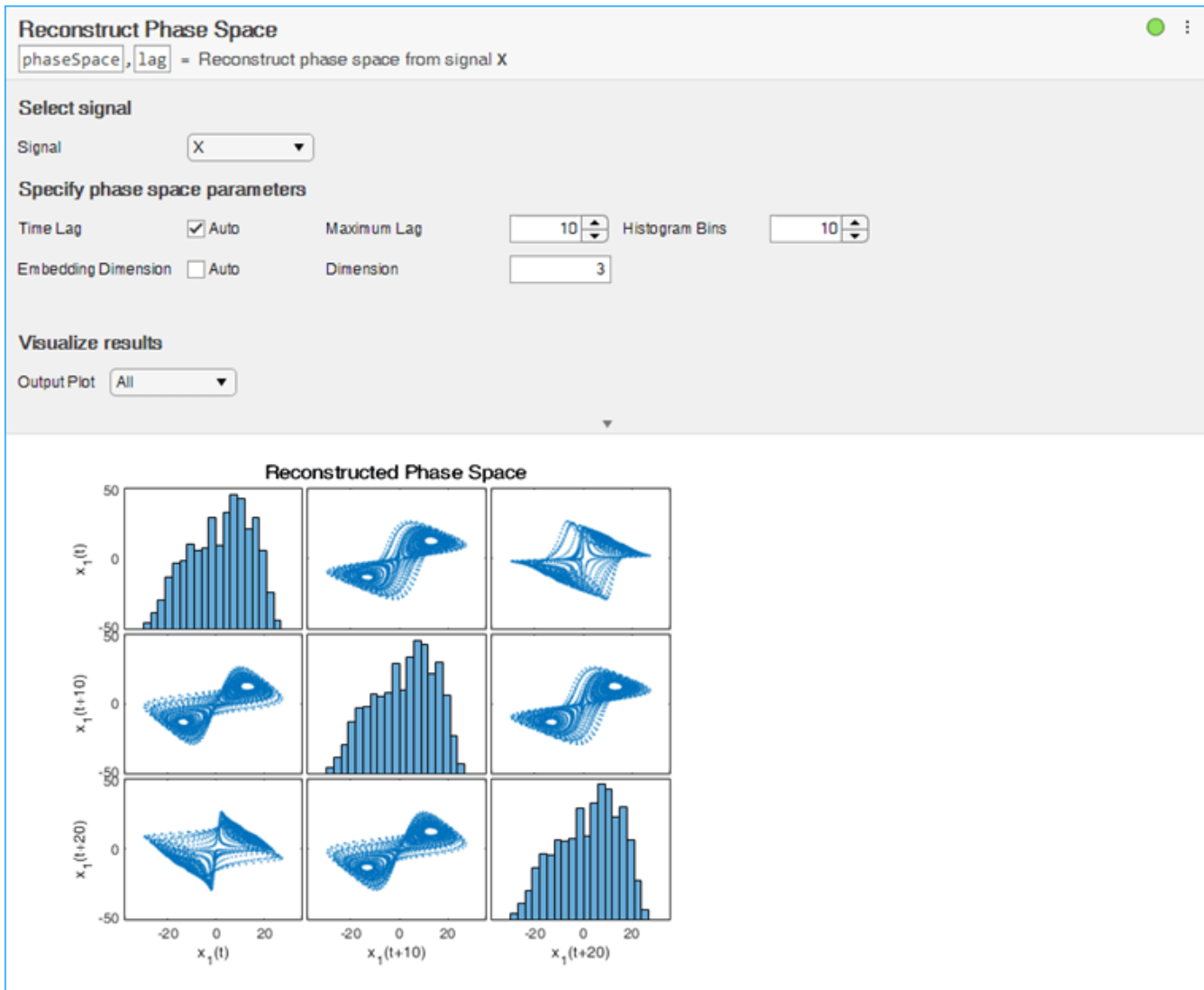
After you finish experimenting with the task, the reconstructed phase space data `phaseSpace` and the estimated time delay `lag` are in the MATLAB® workspace, and you can use them to identify different condition indicators for the Lorenz attractor. For instance, estimate the correlation dimension and the Lyapunov exponent values using `phaseSpace`.

Estimate Correlation Dimension

To estimate the correlation dimension, use the Estimate Correlation Dimension Live Editor Task. In the task, specify the phase space signal, `phaseSpace` from the workspace. Specify signal type as `Phase space`. The task automatically computes the embedding dimension and lag values from the phase space signal. For this example, use 0.05 and 20 for the minimum and maximum similarity radius values and the default value of 10 points. In the **Estimate Correlation Dimension** task, you can experiment with similarity radius values and number of points to align the linear fit line with the original correlation integral data line in the output plot. For details about the available options and parameters, see the Estimate Correlation Dimension task reference page.
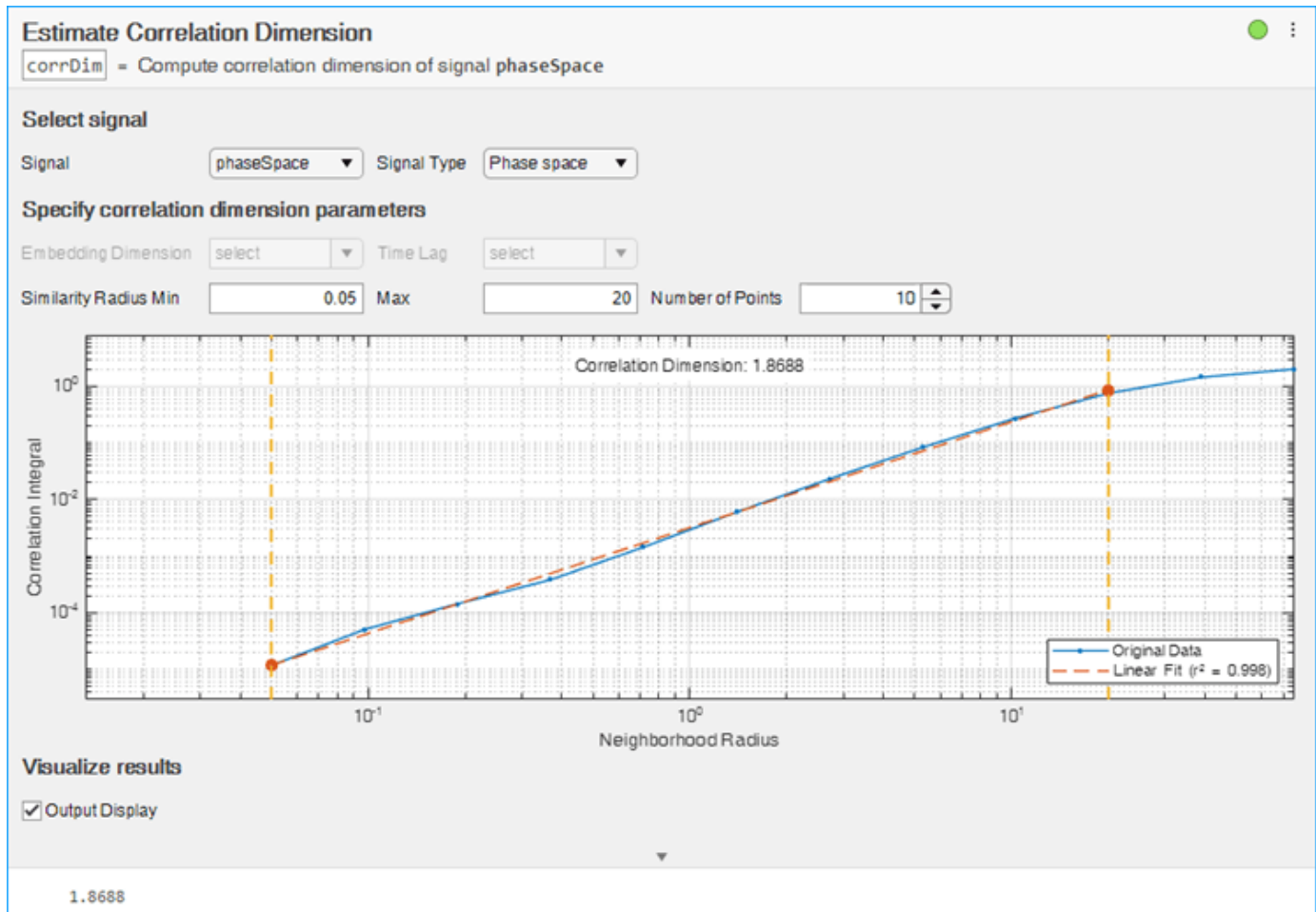
As you vary parameters in the task, it automatically updates the generated code for performing the estimation and creating the plot. (To see the generated code, click ▼ at the bottom of the task.)

Estimate Lyapunov Exponent

To estimate the Lyapunov Exponent, use the Estimate Lyapunov Exponent Live Editor Task. In the task, specify the phase space signal, `phaseSpace` from the workspace. Specify signal type as `Phase space` and the sampling rate as 100 Hz. The task automatically computes the embedding dimension and lag values from the phase space signal. For this example, use 21 and 161 for the minimum and maximum expansion range values and the default value of 110 for the mean period. In the **Estimate Lyapunov Exponent** task, you can experiment with expansion range and mean period values to align the linear fit line with the original log divergence data line in the output plot. For details about the available options and parameters, see the Estimate Lyapunov Exponent task reference page.
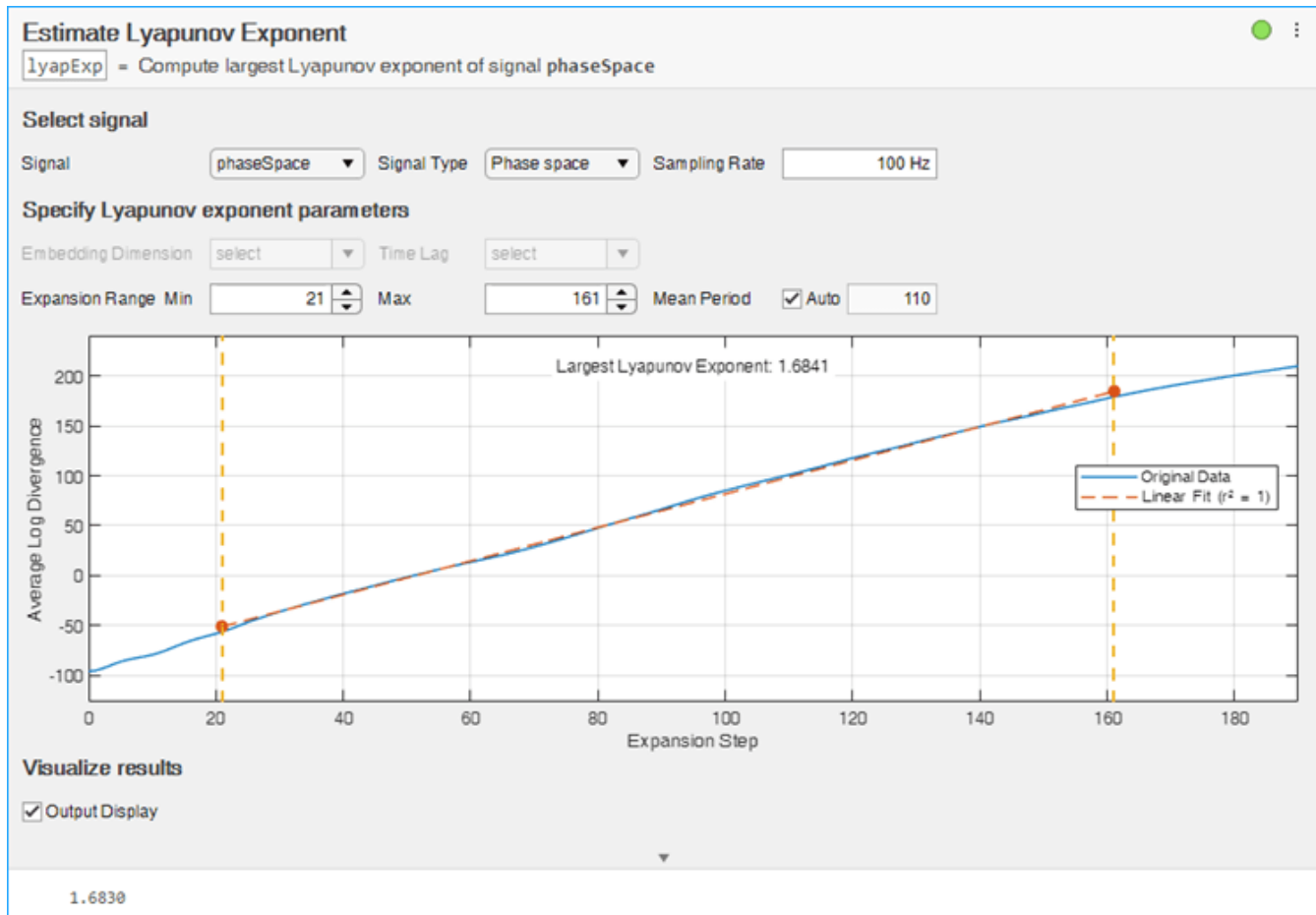
Generate Code

As you vary parameters in each task, it automatically updates the generated code for performing the estimation and creating the plot. To see the generated code, click ▼ at the bottom of the task. You can cut and paste this code to use or modify later in the existing script or a different program. For example:

```
% Compute largest Lyapunov exponent of signal phaseSpace
lyapExp = lyapunovExponent(phaseSpace, 100, ...
    'Dimension', 1, ...
    'Lag', 0, ...
    'ExpansionRange', [21,161]);

% Display the results
disp(lyapExp)
```

Because the underlying code is now part of your live script, you can continue to use the variables created by each task for further processing.

See Also

Reconstruct Phase Space | **Estimate Approximate Entropy** | **Estimate Correlation Dimension** | **Estimate Lyapunov Exponent** | approximateEntropy | correlationDimension | lyapunovExponent | phaseSpaceReconstruction

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

This example shows how to use the Extract Spectral Features Live Editor task to analyze data from a current signal obtained from driving the gear train of a hobby-grade servo. The example also shows how to extract spectral features from the data to aid in fault detection and identification.

Live Editor tasks let you interactively iterate on parameters and settings while observing their effects on the result of your computation. The tasks automatically generate MATLAB® code that achieves the displayed results. For more information about Live Editor tasks generally, see "Add Interactive Tasks to a Live Script".

In particular, this example uses the Extract Spectral Features Live Editor task. This task helps with analyzing and understanding spectral data. Using a comprehensive interface, you can add components to represent various bearings, gear meshes, or other parts of your hardware setup. As you set the physical parameters of these components, the Extract Spectral Features Live Editor task plots fault frequency bands at the charateristic frequencies of the components.

You can overlay power spectrum data on the fault band plot to associate various peaks in the data with the components' characteristic frequencies. This comparison can make fault detection and fault isolation easier, as you can easily trace changes in the power spectrum data back to the physical components causing them.

In addition to a plot of the characteristic frequencies and the power spectrum data, the task can generate spectral metrics of the data within each characteristic frequency band. The output metrics table containing the peak amplitude, peak frequency, and band power of each band aids in characterizing potential mechanical faults.

Hardware Overview
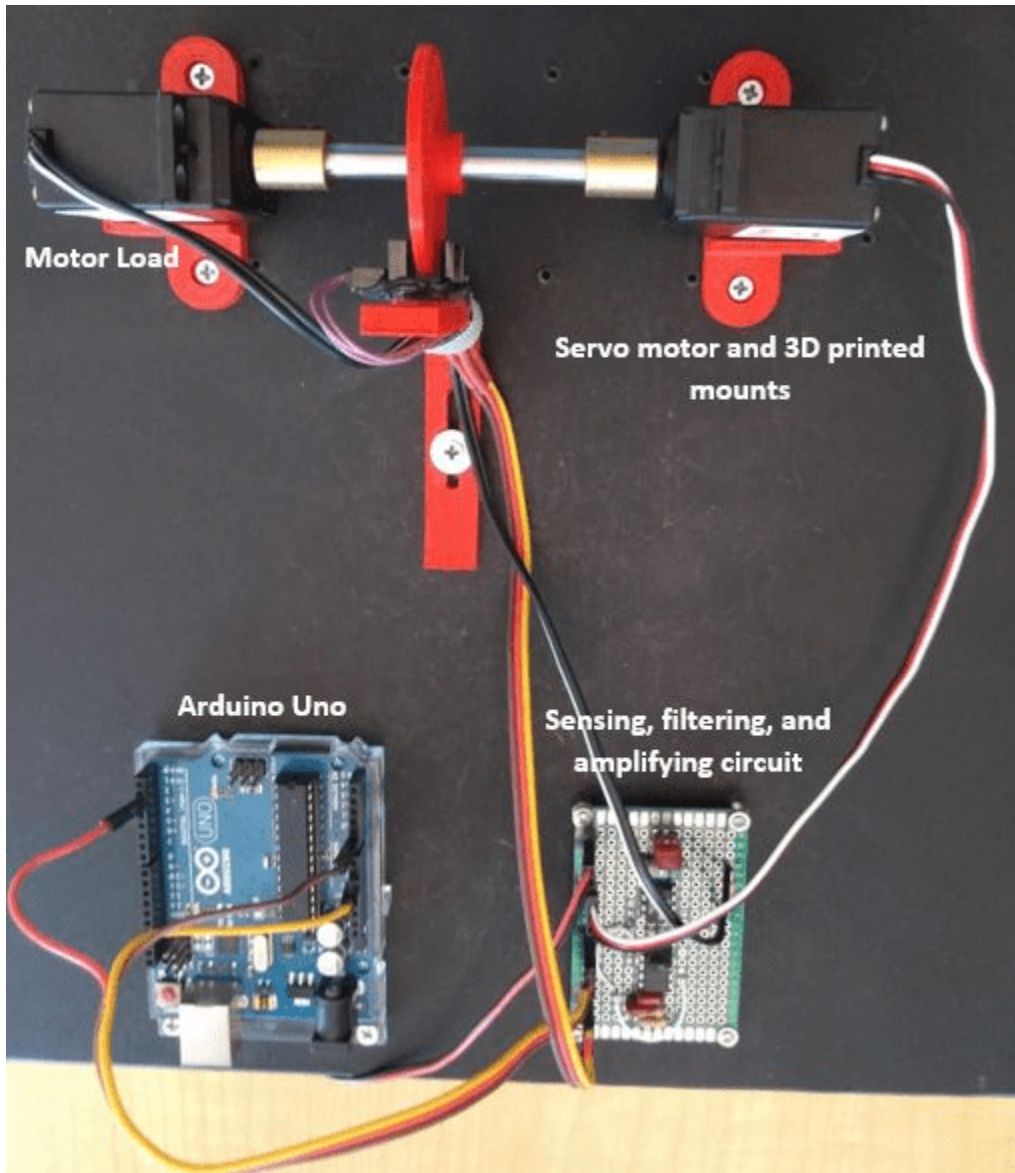
For this example, electrical current data was collected from a standard Futaba S3003 hobby servo, which was modified for continuous rotation. Servos convert the high speed of the internal DC motor to high torque at the output spline. To achieve this, servos consist of a DC motor, a set of nylon or metal drive gears, and the control circuit. The control circuit was removed to allow the current signal to the DC motor to be directly monitored. The tachometer signal at the output spline of the servo was collected using an infrared photointerrupter along with a 35 mm diameter, sixteen-slot wheel. The sixteen slots in the wheel were equally spaced and the IR photointerrupter was placed such that it emitted exactly sixteen pulses per rotation of the slotted wheel. The servo and photointerrupter were held in place by custom 3-D printed mounts.
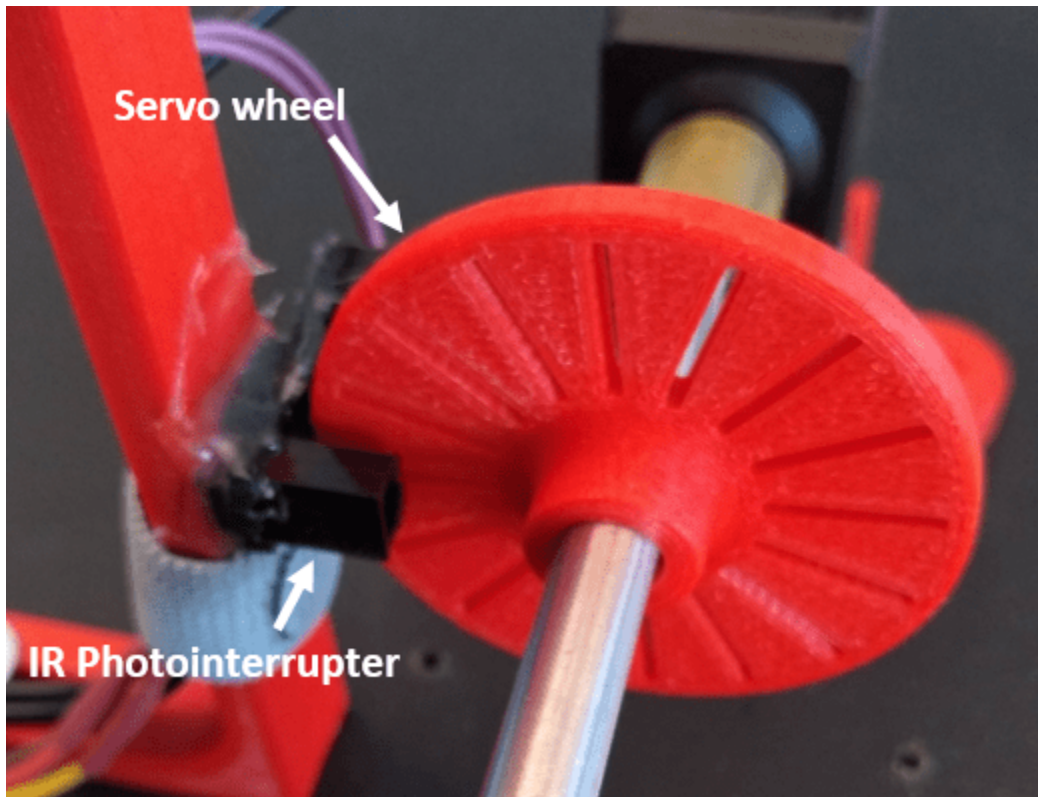
The DC motor was driven at a constant 5 volts, and with four pairs of gears providing 278:1 speed reduction, the shaft speed at the spline was about 19.5 rpm. A second servo motor was shorted out and used as a load for the system. The current consumption was calculated using Ohm's law by measuring the voltage drop across a 0.5 ohm resistor. Since the change in current measurement values was too small for detection, the current signal was amplified using an AD22050 single-supply sensor interface amplifier. The amplified current signal was then filtered using a MAX7408 anti-aliasing fifth-order elliptic low-pass filter to smooth it and to eliminate noise before sending it to an Arduino Uno through an analog-to-digital converter (ADC).
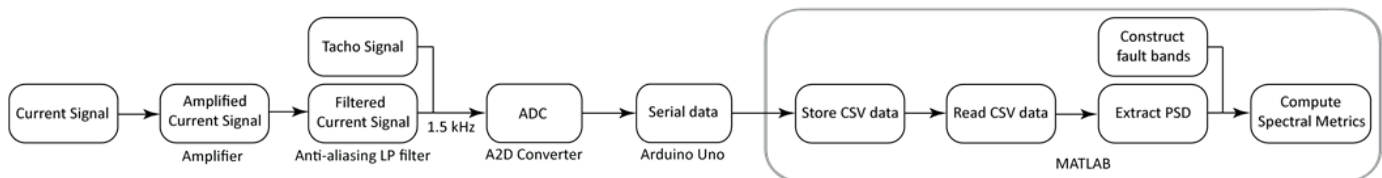
As the flowchart shows, the current signal was first amplified and filtered using the amplifier and the anti-aliasing low-pass filter, respectively. The Arduino Uno sampled the current signal through an ADC at 1.5 kHz and streamed it to the computer along with the tachometer pulses as serial data at a baud rate of 115,200 bps. A MATLAB script fetched the serial data from the Arduino Uno, pre-processed it, and wrote it to a .MAT file. The Extract Spectral Features Live Editor task was then used to extract the spectral metrics.

Servo Gear Train

Futaba S3003 standard servo gear train
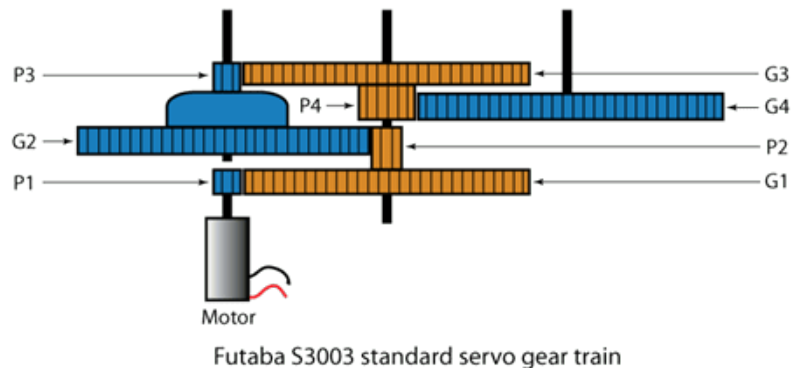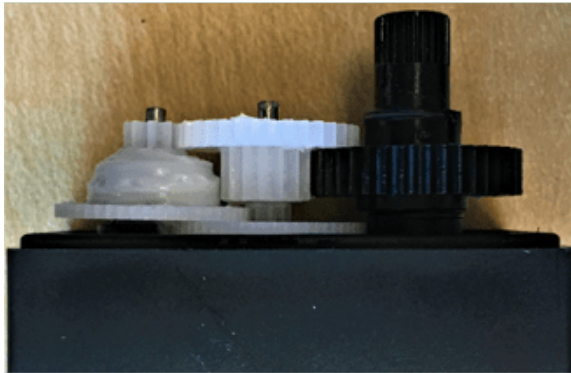
The Futaba S3003 servo consists of four pairs of nylon gears as illustrated in this figure. The pinion P1 on the DC motor shaft meshes with the stepped gear G1. The pinion P2 is a molded part of the stepped gear G1 and meshes with the stepped gear G2. The pinion P3, which is a molded part of gear G2, meshes with the stepped gear G3. Pinion P4, which is molded with G3, meshes with the final gear G4, which is attached to the output spline. The stepped gear sets G1 and P2, G2 and P3, and G3 and P4 are free spinning gears — that is, they are not attached to their respective shafts. The set of drive gears provides a 278:1 reduction, going from a motor speed of 5414.7 rpm to about 19.5 rpm at the output spline when the motor is driven at 5 volts. The following table outlines the tooth count and theoretical values of output speed, gear mesh frequencies, and cumulative gear reduction at each gear mesh.

Pinion	Gear	Pinion Teeth	Gear Teeth	Output Speed (RPM)	Output Speed (Hz)	Gear Mesh Frequency (Hz)	Cumulative Gear Reduction
P1		10		5,414.7	90.24		1
P2	G1	10	62	873.3	14.56	902.4	6.2
P3	G2	10	50	174.7	2.91	145.6	31
P4	G3	16	35	49.9	0.83	29.1	108.5
	G4		41	19.5	0.32	13.3	278

Theoretical values calculated from tooth count assuming 19.5 RPM at the output shaft

Preprocess Data

The file servoData.mat contains two timetables corresponding to data from the servo. One timetable contains healthy data while the second timetable contains faulty data. Each data set contains about 11 seconds of data sampled at 1500 Hz.

Load the data.

```
load('servoData.mat', 'healthyData', 'faultyData')
healthyData
```

```
healthyData=16384×2 timetable
        Time          MotorCurrent     TachoPulse
    _____    _____    _____
```

```
0 sec                307.62              1
0.00066667 sec       301.27              1
0.0013333 sec        309.08              1
0.002 sec            315.92              1
0.0026667 sec         304.2              1
0.0033333 sec        311.04              1
0.004 sec            311.52              1
0.0046667 sec        305.18              1
0.0053333 sec        315.43              0
0.006 sec            310.06              0
0.0066667 sec        305.66              0
0.0073333 sec        310.55              0
0.008 sec            304.69              0
0.0086667 sec        310.55              0
0.0093333 sec        310.06              0
0.01 sec              299.8              0
     ⋮
```

faultyData

faultyData=16384×2 timetable

Time	MotorCurrent	TachoPulse
0 sec	313.48	0
0.00066667 sec	304.2	0
0.0013333 sec	303.22	0
0.002 sec	319.34	0
0.0026667 sec	304.2	0
0.0033333 sec	303.22	0
0.004 sec	319.82	0
0.0046667 sec	303.22	0
0.0053333 sec	306.64	0
0.006 sec	321.29	0
0.0066667 sec	303.71	0
0.0073333 sec	308.11	0
0.008 sec	319.34	0
0.0086667 sec	301.76	0
0.0093333 sec	309.08	0
0.01 sec	319.34	0
⋮		

Each timetable contains one column with the motor current and one column with the tacho pulse from the servo setup. In order to visualize the data in the Extract Spectral Features Live Editor task, compute the power spectrum of the motor current data. Consider the healthy data first.

```
fs = 1500;
[healthyMagnitudes,healthyFrequencies] = pwelch(healthyData{:,1},16384,[],[],fs);
```

Use the Extract Spectral Features Live Editor task to plot the power spectrum data. In the task, specify `healthyFrequencies` for the frequency vector and `healthyMagnitudes` for the power spectrum magnitude.
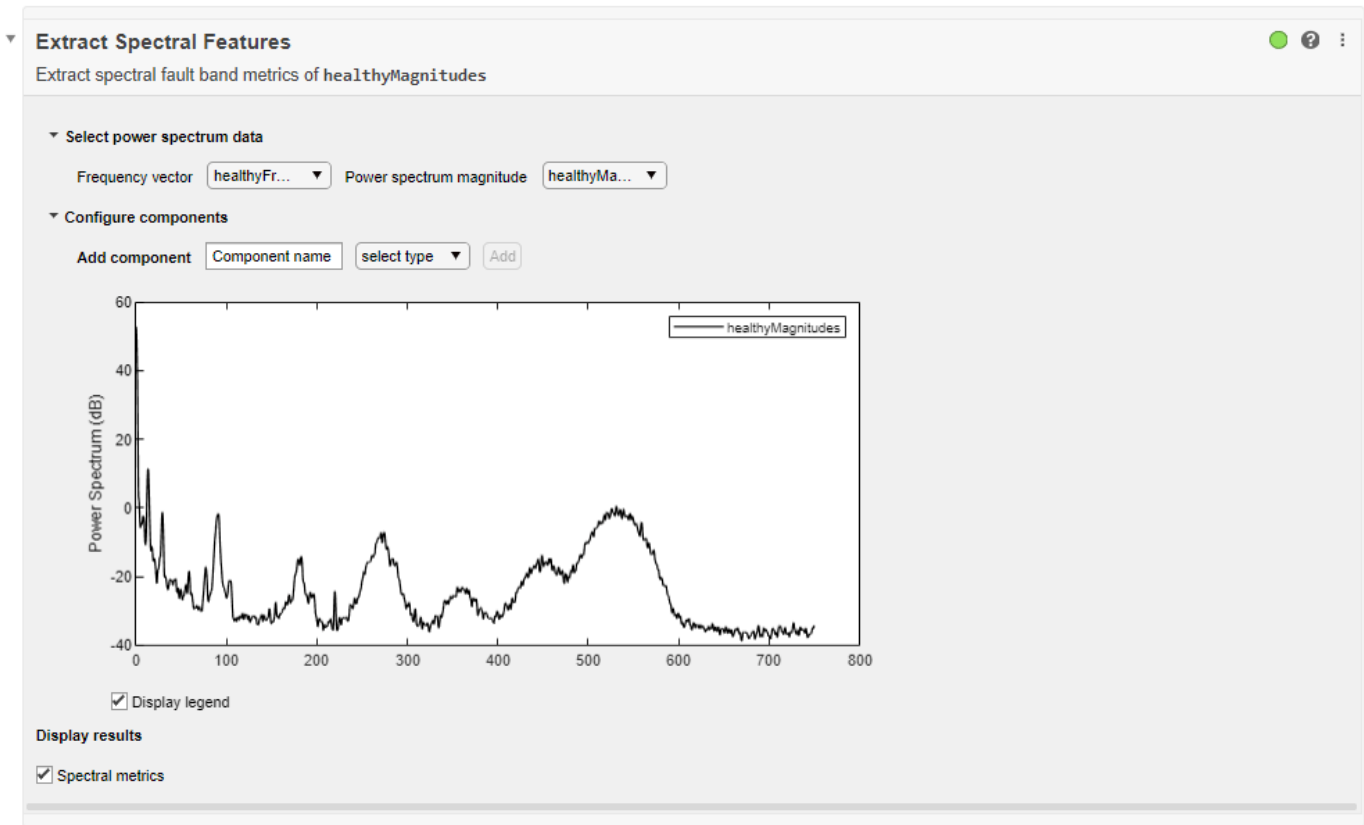
Analyze Power Spectrum Peaks with Harmonic Fault Frequency Bands

The power spectrum plot of the servo's current data contains several noticeable peaks. You can associate these peaks with the rotating shafts in the servo setup. To determine the source of the various peaks, add components to the Extract Spectral Features Live Editor task.

To add a component to represent the first rotating shaft in the servo, enter a name for the component, select its type as **Custom**, and press **Add**. Use the output speeds in the table above to choose the frequency of the shaft component. The output speeds were computed based on the measured speed of the output shaft and the known gear reductions in the setup.

The frequency of the first shaft is 90.24 Hz. After setting the frequency value of the shaft component, note that the fault frequency band overlaps with one of the peaks in the power spectrum data at around 90 Hz. You can therefore associate this peak in large part with the first shaft. Adding in several more harmonics of the first shaft's fundamental frequency creates more fault frequency bands that overlap other peaks in the data. Harmonic frequency bands are centered around integer multiples of the fundamental frequency and can still be associated with the same component. Set the harmonics of the first shaft to be the vector [1 2 3 4 5 6] so that the fault bands spread over most of the frequency range. The first shaft is rotates at the highest frequency, so power spectrum peaks at higher frequencies are due to this shaft's harmonics.

Extract Spectral Features

`FB`, `info`, `spectralMetrics` = Extracted fault frequency bands and spectral metrics of `healthyMagnitudes`

▾ Select power spectrum data

Frequency vector `healthyFr...` ▾   Power spectrum magnitude `healthyMa...` ▾

▾ Configure components

Add component `Component name` `select type` ▾ `Add`

▾ Shaft1 (Custom)

☑ Enable component `Delete`

Frequency `90.24` Harmonics `1:6` ▾ Sidebands `0` ▾

Separation type `Additive` ▾ ☐ Folding Separation `9.024` ☑ Auto

Band width `4.512` ☑ Auto

Fault Frequency Bands

☑ Display legend

Display results

☐ Spectral metrics

To account for smaller power spectrum peaks such as those around 13 Hz or 29 Hz, add in a component for the second rotating shaft. This component is also a custom component, and its fundamental frequency is 14.56 Hz. You do not need to add as many harmonics for the second shaft since most of the higher peak frequencies are largely accounted for by harmonics of the first shaft. Set the harmonics of the second shaft to be the vector [1 2 3 4]. The first, second, and fourth harmonics of this shaft frequency align nicely with peaks in the power spectrum plot. However since the third harmonic is less prominent in the data, you do not need to include this harmonic. Change the harmonics of the second shaft to be the vector [1 2 4].

3-41

Extract Spectral Features

FB , info , spectralMetrics = Extracted fault frequency bands and spectral metrics of `healthyMagnitudes`

▾ Select power spectrum data

Frequency vector `healthyFr...` ▾   Power spectrum magnitude `healthyMa...` ▾

▾ Configure components

Add component [Component name] [select type ▾] [Add]

▸ Shaft1 (Custom)

▾ Shaft2 (Custom)

☑ Enable component [Delete]

Frequency [14.56] Harmonics [[1 2 4] ▾] Sidebands [0 ▾]

Separation type [Additive ▾] ☐ Folding   Separation [1.456] ☑ Auto

Band width [0.728] ☑ Auto

Fault Frequency Bands

☑ Display legend

Display results

☐ Spectral metrics

Similarly to the second shaft, add a component for the third rotating shaft. The fundamental frequency of this component is 2.91 Hz, as seen in the table. Start with the first four harmonics to determine if they align with any prominent peaks in the data. Note that the third harmonic of the third shaft matches a power spectrum spike around 8 Hz. The other harmonics are less prominent and can be removed. It might be the case that the frequency resolution of the power spectrum does not allow distinguishing lower frequencies. Set the harmonics of the third shaft component to be just the third harmonic.
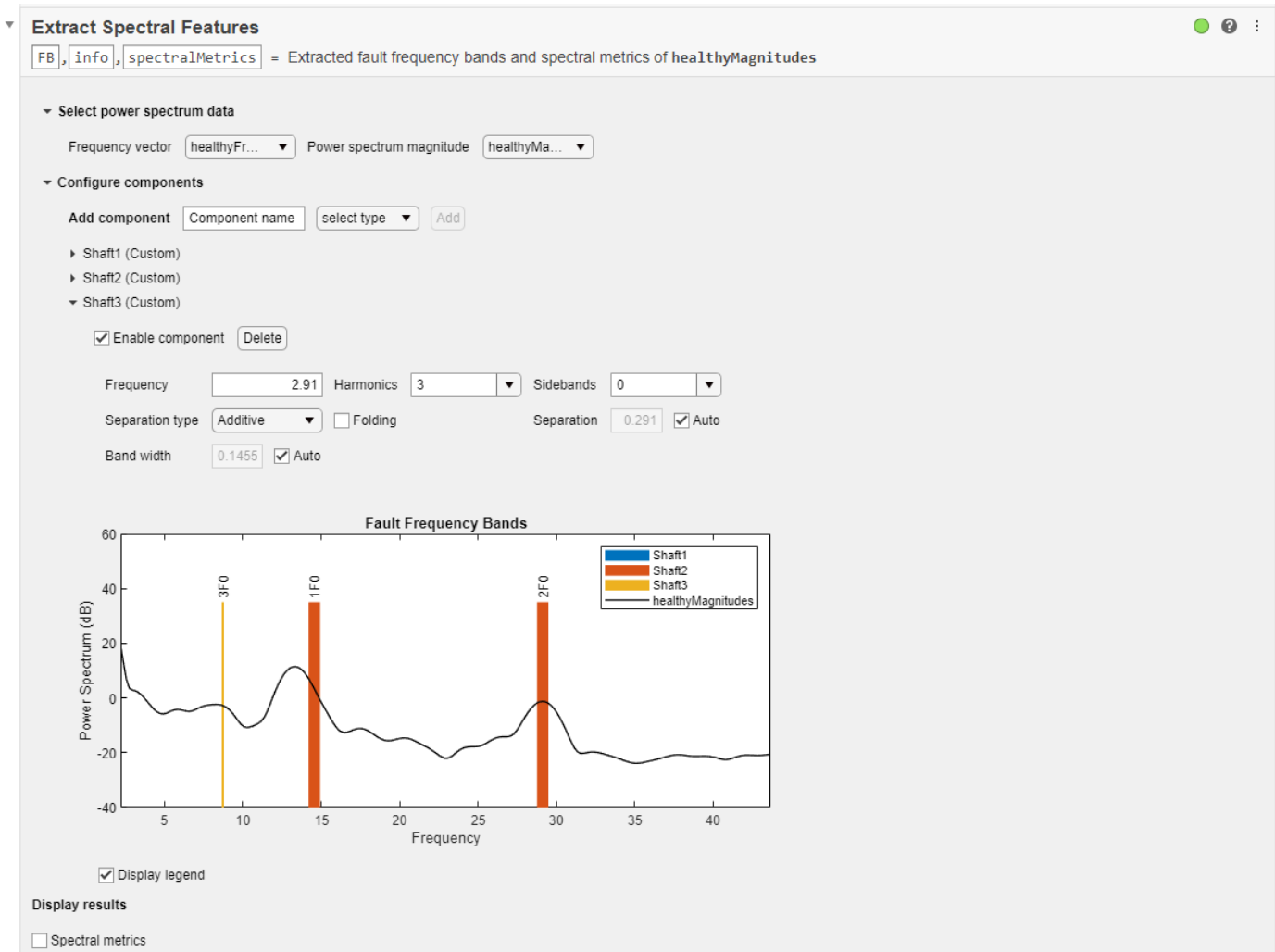
Since the output speeds of the remaining shafts are also low frequencies that might not be distinguishable from the frequency resolution of the power spectrum, adding components for these shafts is not necessary to analyze the major peaks of the motor current data.

Analyze Sideband Peaks

By zooming into the plot in the task, you can see that the power spectrum data contains side peaks next to some of the major peaks. For instance, smaller side peaks around 76 Hz and 104 Hz surround the peak at 90 Hz. These peaks are likely associated with sidebands of the first shaft component. Sidebands are caused by a second related frequency source impacting the primary harmonic frequency source. With the servo setup, this observation leads to the presumption that the sidebands for each shaft are caused by the next shaft in the gear train.
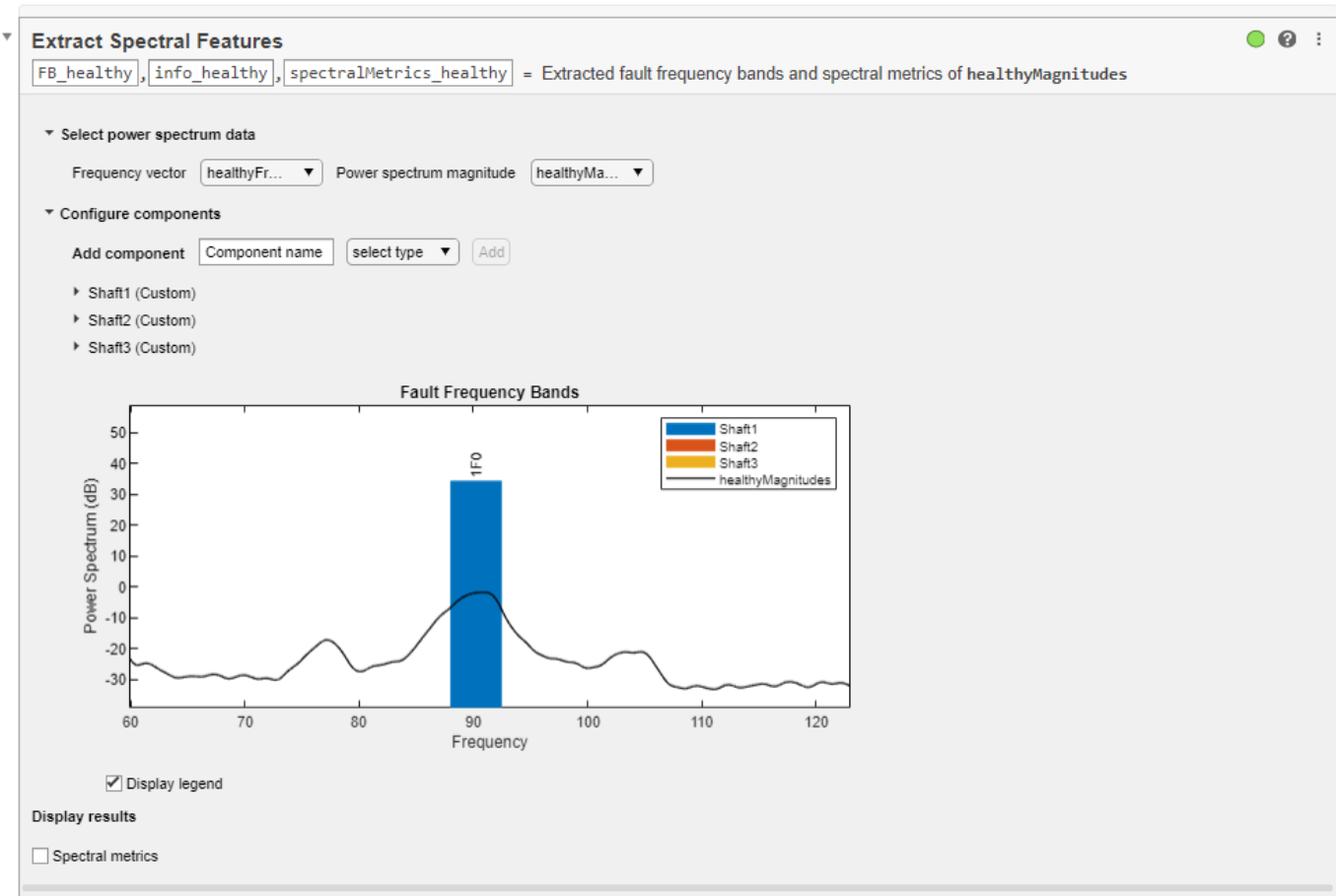
Edit the first two shaft components to include their first sideband. For the first shaft the sideband separation value should be equal to the nominal output frequency of the second shaft, 14.56 Hz.
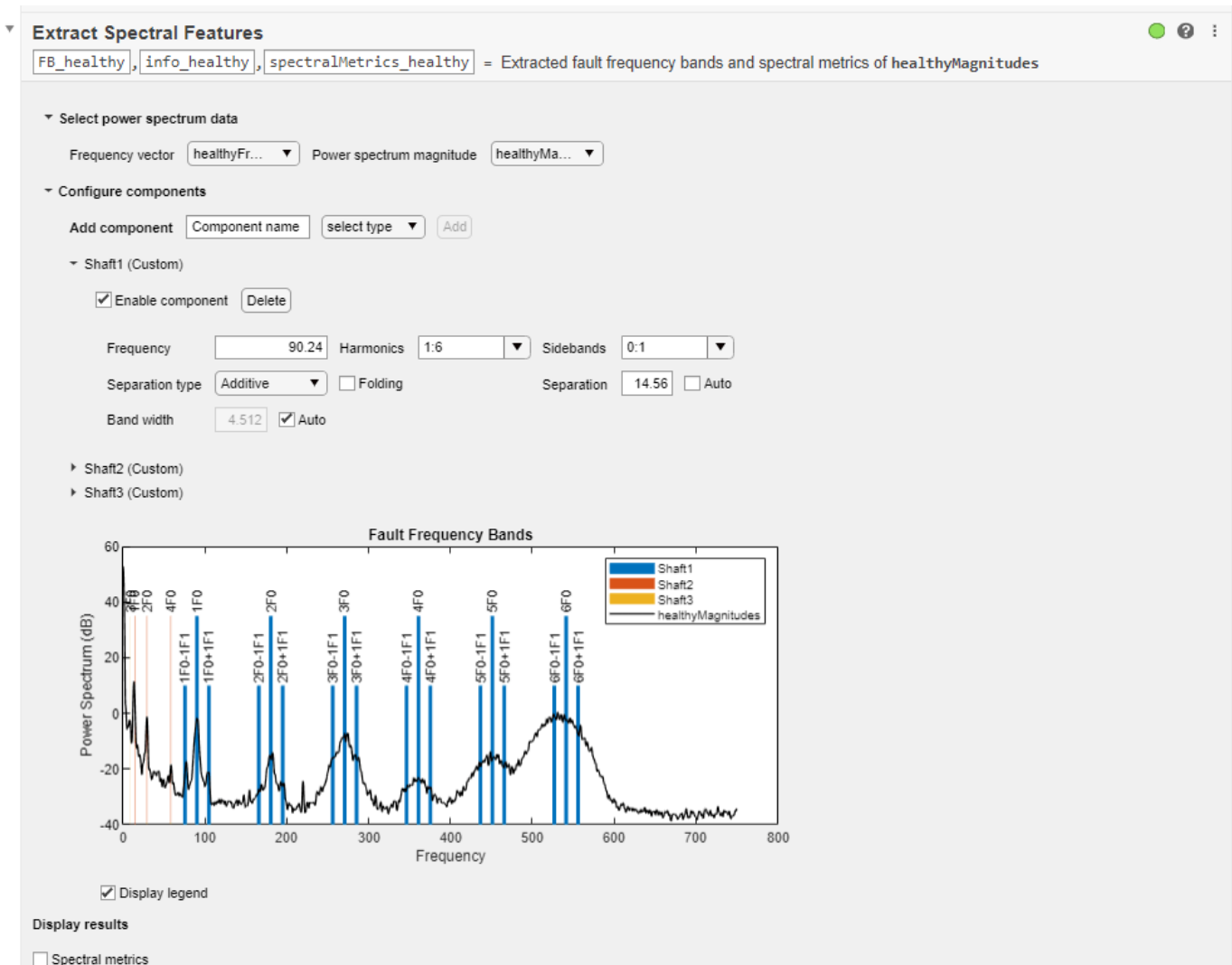
Similarly for the second shaft the sideband separation value should be equal to the nominal frequency of the third shaft, 2.91 Hz. Zooming in again on the plot shows that many of the new sidebands overlap well with side peaks in the data. This is easier to see at low frequencies such as between 0 Hz and 120 Hz.
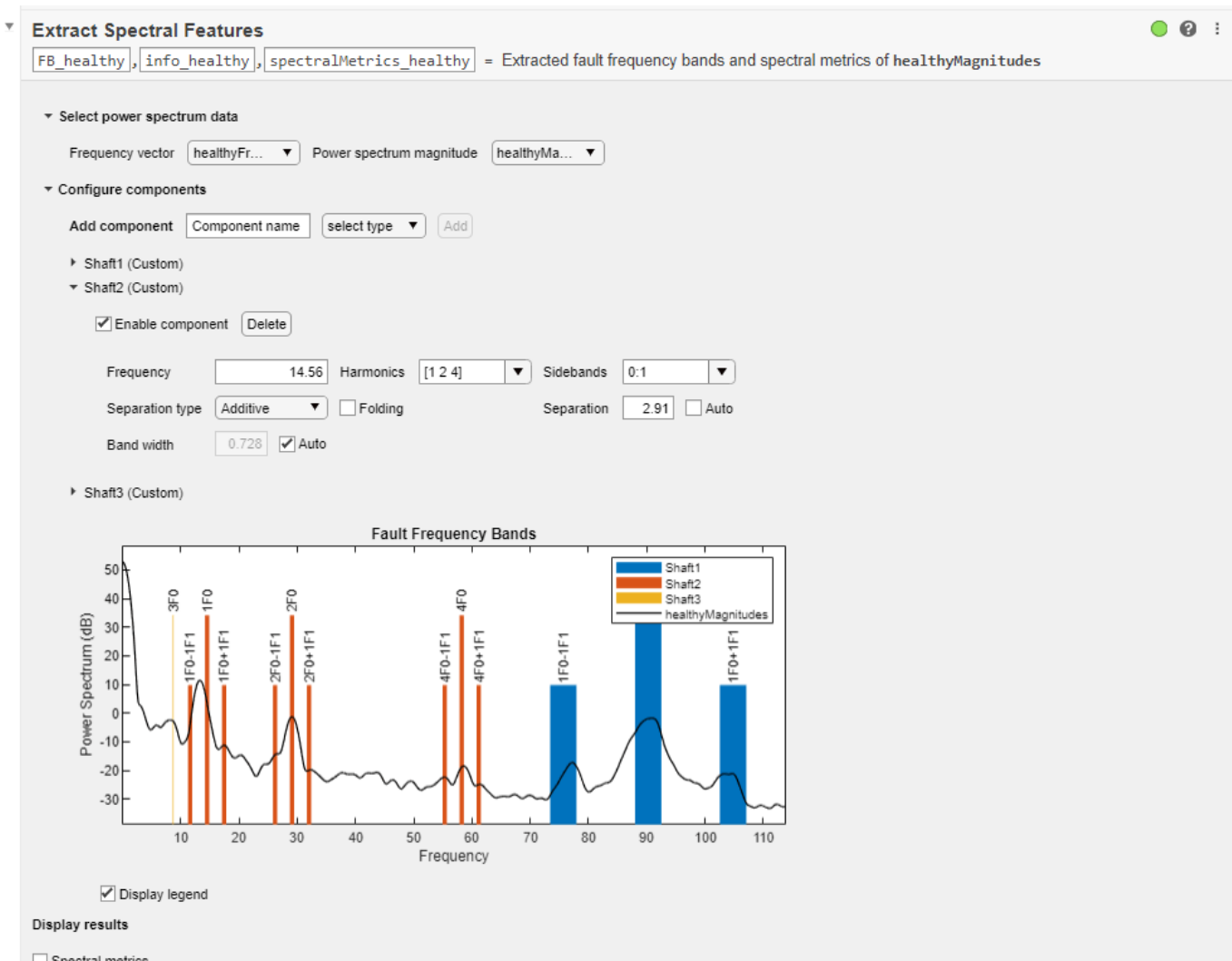
Extract Spectral Metrics to Detect Faults

The Extract Spectral Features Live Editor task generates various spectral metrics of the power spectrum data in fault frequency ranges. For each fault frequency band, compute the peak amplitude, peak frequency, and band power along with the total band power of all fault frequency bands.

```
load('dataSample.mat')
spectralMetrics_healthy
```

```
spectralMetrics_healthy=1×85 table
    PeakAmplitude1    PeakFrequency1    BandPower1    PeakAmplitude2    PeakFrequency2    BandPov
    _____    _____    _____    _____    _____    _____

       0.088268          77.179          0.031107         2.0259            91.37           1.489
```

These metrics may prove useful in detecting faults in the servo setup. Significant changes in the power spectrum data often indicate that some component are changing or failing. If there is a shift in one of the peak frequencies or if a peak amplitude drops significantly over time, then this may be a sign of failure.

To examine this scenario, compute the power spectrum data for the faulty dataset.

```
[faultyMagnitudes,faultyFrequencies] = pwelch(faultyData{:,1},16384,[],[],fs);
```

Plot the spectrum of faulty data in the task, and adjust the fundamental frequencies and sideband separation values of the shaft components based on the measured output speed of the servo setup.

```
Fs = 1500; % 1500 Hz
[outputSpeed,t] = tachorpm(faultyData.TachoPulse,Fs,'PulsesPerRev',16,'FitType','linear');
meanOutputSpeed = mean(outputSpeed)/60 % convert from rpm to Hz
```

```
meanOutputSpeed = 0.3150
```

```
shaft4Speed = meanOutputSpeed * 41 / 16 % 16 pinion teeth, 41 gear teeth
```

```
shaft4Speed = 0.8072
```

```
shaft3Speed = shaft4Speed * 35 / 10 % 10 pinion teeth, 35 gear teeth
```

```
shaft3Speed = 2.8251
```

```
shaft2Speed = shaft3Speed * 50 / 10 % 10 pinion teeth, 50 gear teeth
```

```
shaft2Speed = 14.1254
```

```
shaft1Speed = shaft2Speed * 62 / 10 % 10 pinion teeth, 62 gear teeth
```

```
shaft1Speed = 87.5772
```

For the first shaft component, use `shaft1Speed` as the fundamental frequency and `shaft2Speed` as the sideband separation. For the second shaft component, use `shaft2Speed` as the fundamental frequency and `shaft3Speed` as the sideband separation. For the third shaft component, use `shaft3Speed` as the fundamental frequency.
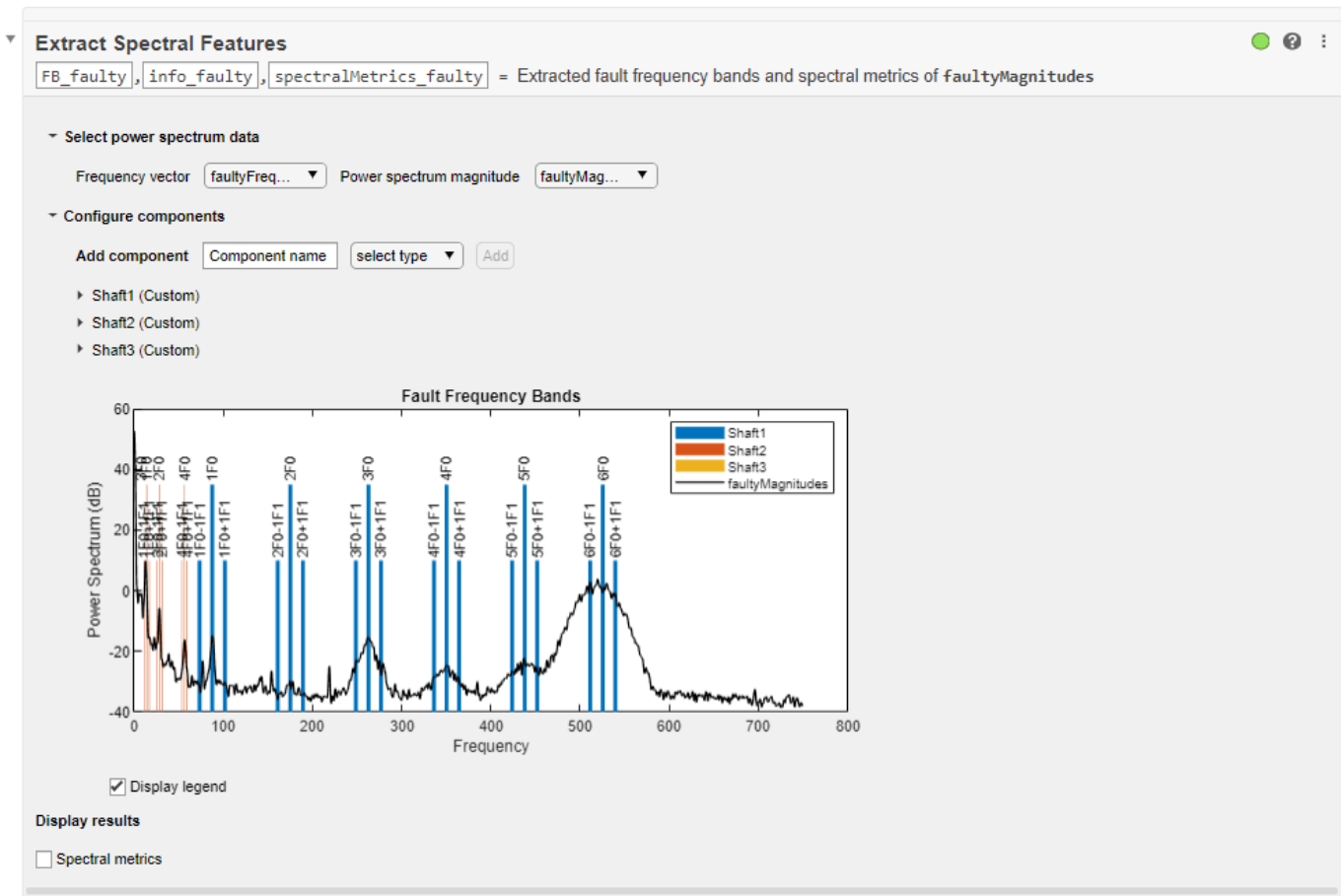
As can be seen in the visualization of the faulty power spectrum data, several of the peaks have diminished in magnitude. For example, the peak aligned in the healthy dataset with the second harmonic of the first shaft around 180 Hz is almost negligible in the faulty dataset. Since it was previously determined that this peak is likely associated with the first shaft, this indicates a potential failure in the first shaft. Further examination of the spectral metrics table can provide more detailed information about the peak frequencies, peak amplitudes, and band powers.

```
spectralMetrics_faulty
```

```
spectralMetrics_faulty=1×85 table
    PeakAmplitude1    PeakFrequency1    BandPower1    PeakAmplitude2    PeakFrequency2    BandPow
    _____    _____    _____    _____    _____    _____

      0.0015834          71.411         0.0017557        0.082928           88.806         0.0708
```

As an alternative to updating the spectral data in the Live Editor task, you can also use the automatically generated MATLAB code to detemine the spectral metrics of the faulty data. The code below was automatically generated when the Live Editor task was used to generate the healhty spectral metrics. Execute the code.

```
% Generate the fault bands and information for each component
[FB_Shaft1, info_Shaft1] = faultBands(90.24, 1:6, 14.56, 0:1);
[FB_Shaft2, info_Shaft2] = faultBands(14.56, [1 2 4], 2.91, 0:1);
```

```
[FB_Shaft3, info_Shaft3] = faultBands(2.91, 3);

% Combine the fault bands of each component
FB_healthy = [FB_Shaft1; ...
        FB_Shaft2; ...
        FB_Shaft3];

% Combine the information regarding the fault bands of each component
info_healthy.Centers = [info_Shaft1.Centers, ...
        info_Shaft2.Centers, ...
        info_Shaft3.Centers];
info_healthy.Labels = [info_Shaft1.Labels, ...
        info_Shaft2.Labels, ...
        info_Shaft3.Labels];
info_healthy.FaultGroups = [info_Shaft1.HarmonicGroups, ...
        info_Shaft2.HarmonicGroups, ...
        info_Shaft3.HarmonicGroups];

% Clear temporary outputs from the workspace
clear FB_Shaft1 info_Shaft1;
clear FB_Shaft2 info_Shaft2;
clear FB_Shaft3 info_Shaft3;

% Compute fault band metrics of the power spectrum healthyMagnitudes
spectralMetrics_healthy = faultBandMetrics(healthyMagnitudes, healthyFrequencies, FB_healthy)
```

spectralMetrics_healthy=*1×85 table*

PeakAmplitude1	PeakFrequency1	BandPower1	PeakAmplitude2	PeakFrequency2	BandPo
0.088268	77.179	0.031107	2.0259	91.37	1.489

This code can easily be adjusted for the new faulty dataset.

```
% Generate the fault bands and information for each component
[FB_Shaft1, info_Shaft1] = faultBands(shaft1Speed, 1:6, shaft2Speed, 0:1);
[FB_Shaft2, info_Shaft2] = faultBands(shaft2Speed, [1 2 4], shaft3Speed, 0:1);
[FB_Shaft3, info_Shaft3] = faultBands(shaft3Speed, 3);

% Combine the fault bands of each component
FB_faulty = [FB_Shaft1; ...
        FB_Shaft2; ...
        FB_Shaft3];

% Combine the information regarding the fault bands of each component
info_faulty.Centers = [info_Shaft1.Centers, ...
        info_Shaft2.Centers, ...
        info_Shaft3.Centers];
info_faulty.Labels = [info_Shaft1.Labels, ...
        info_Shaft2.Labels, ...
        info_Shaft3.Labels];
info_faulty.FaultGroups = [info_Shaft1.HarmonicGroups, ...
        info_Shaft2.HarmonicGroups, ...
        info_Shaft3.HarmonicGroups];

% Clear temporary outputs from the workspace
clear FB_Shaft1 info_Shaft1;
```

```
clear FB_Shaft2 info_Shaft2;
clear FB_Shaft3 info_Shaft3;

% Compute fault band metrics of the power spectrum healthyMagnitudes
spectralMetrics_faulty = faultBandMetrics(faultyMagnitudes, faultyFrequencies, FB_faulty)
```

spectralMetrics_faulty=*1×85 table*

PeakAmplitude1	PeakFrequency1	BandPower1	PeakAmplitude2	PeakFrequency2	BandPov
0.0015834	71.411	0.0017557	0.082928	88.806	0.0708

Combining the healthy and faulty spectral metrics tables improves visibility into changes in peak amplitudes and band powers of the spectral data.

```
spectralMetrics_total = [spectralMetrics_healthy; spectralMetrics_faulty]
```

spectralMetrics_total=*2×85 table*

PeakAmplitude1	PeakFrequency1	BandPower1	PeakAmplitude2	PeakFrequency2	BandPov
0.088268	77.179	0.031107	2.0259	91.37	1.48
0.0015834	71.411	0.0017557	0.082928	88.806	0.0708

For example, if you look at `PeakAmplitude2` in the table, the amplitude of the power spectrum peak drops from 2.0259 to 0.0829. Using the `PeakFrequency2` value, you know that this drop occurs around 90 Hz. Plot the two power spectrums on the same axes to visualize the drop outside of the Live Editor task.

```
plot(healthyFrequencies,10*log10(healthyMagnitudes),'b-'); % plot in decibels
hold on;
plot(faultyFrequencies,10*log10(faultyMagnitudes),'r-'); % plot in decibels
legend('Healthy','Faulty')
xlabel('Frequency (Hz)')
ylabel('Power Spectrum (dB)')
title('Comparison of Power Spectrums')
hold off;
```
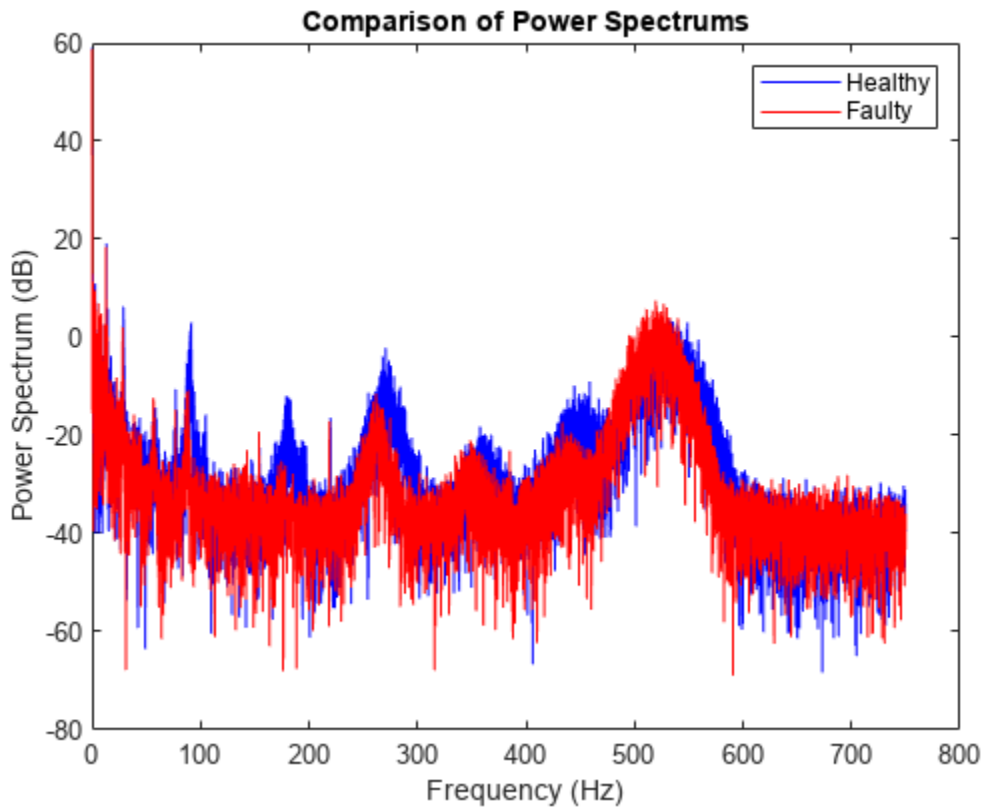
As the metrics table showed, the peak around 90 Hz drops significantly in amplitude. To determine which component frequency caused this, check back in the previous Extract Spectral Feature Live Editor task.

The fault band around 90 Hz is the first harmonic of the first rotating shaft. Thus you know that something is changing in this shaft and it may be trending towards failure.

Plotting the healthy and faulty power spectrums together can be a useful way to highlight changes in peak amplitudes. In addition to the peak around 90 Hz for the first harmonic of the first shaft, other significant decreases in peak amplitudes can be seen such as with the second harmonic of this shaft around 180 Hz. This peak is essentially nonexistent in the faulty dataset.

Peak amplitudes from the healhty and faulty data can also be compared using a bar chart.

```
PeakFrequencies = spectralMetrics_total(:,2:3:end-1).Variables';
PeakAmplitudes = spectralMetrics_total(:,1:3:end-1).Variables';
bar(PeakFrequencies, PeakAmplitudes);
legend('Healthy','Faulty')
xlabel('Frequency (Hz)')
ylabel('Peak Amplitude')
title('Peak Amplitudes of Healthy and Faulty Power Spectrum Data')
```

Zoom in to see the change in peak amplitude at the first harmonic of the first rotating shaft.

```
xlim([87 93])
ylim([0 0.7])
```

Peak Amplitudes of Healthy and Faulty Power Spectrum Data
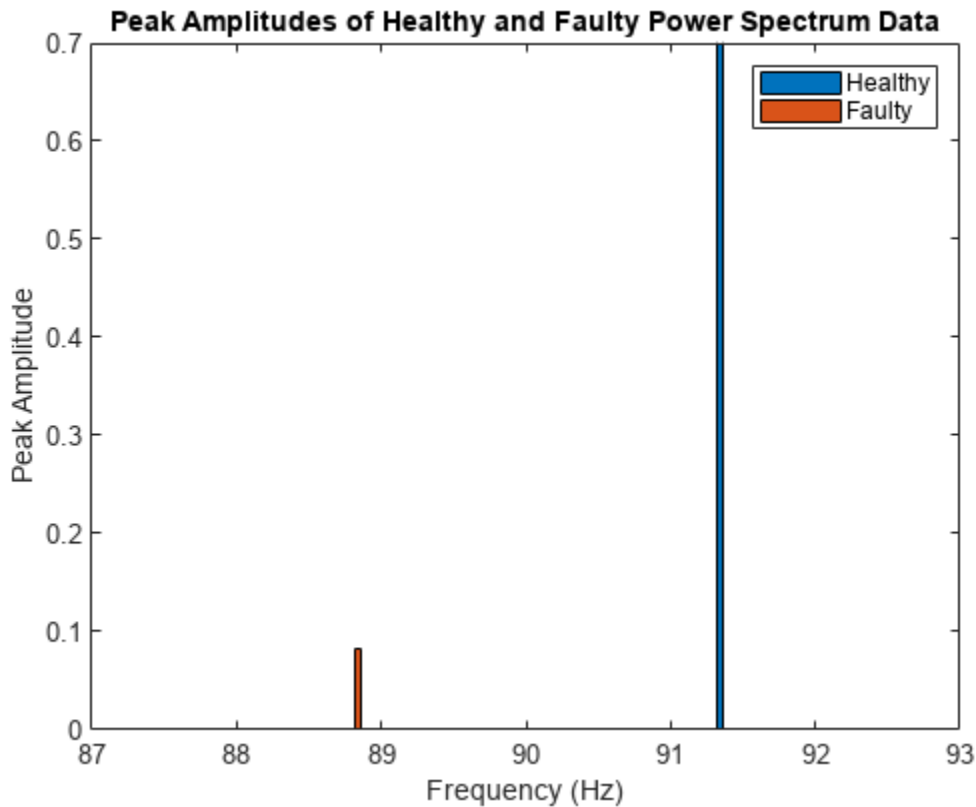
Uses of the Extract Spectral Features Live Editor Task

As shown in this example the Extract Spectral Features Live Editor task can prove useful for several different applications. With the Live Editor task, you can easily match spectral peaks with known machine component frequencies. This helps you better understand your data and the mechanical components causing various features in the data.

Another application of the Extract Spectral Features Live Editor task is to generate metrics which characterize your spectral data in the frequency ranges of interest. The task produces an output table containing the peak amplitude, peak frequency, and band power of each fault frequency band, as well as the total band power of all fault bands. However, these metrics are specific to the power spectrum data input in the task.

To extend this use so that you can track metrics over time as new data sets are gathered, you can either update the power spectrum data in the task or use the automatically generated MATLAB code to produce the metrics table. Copying the generated MATLAB code is an easy way to continue computing fault band metrics for many new data sets.

A third use of the task combines the benefits of the two previously discussed uses. Since the task associates the various mechanical components with peaks in the spectral data, you can quickly determine which components cause significant changes in the spectral data and thus potential failures. For example, one common indicator of machine failure derived from spectral data is a change in the amplitude of spectral peaks. In the spectral metrics table, if you notice a significant

drop in peak amplitude or band power over time you can trace the corresponding peak frequency back to the plot to see which component's fault frequency band aligns with that peak.

See Also

faultBands | bearingFaultBands | gearMeshFaultBands | faultBandMetrics | **Extract Spectral Features**

More About

- "Add Interactive Tasks to a Live Script"

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.

Sensor data from machine on which algorithm is deployed
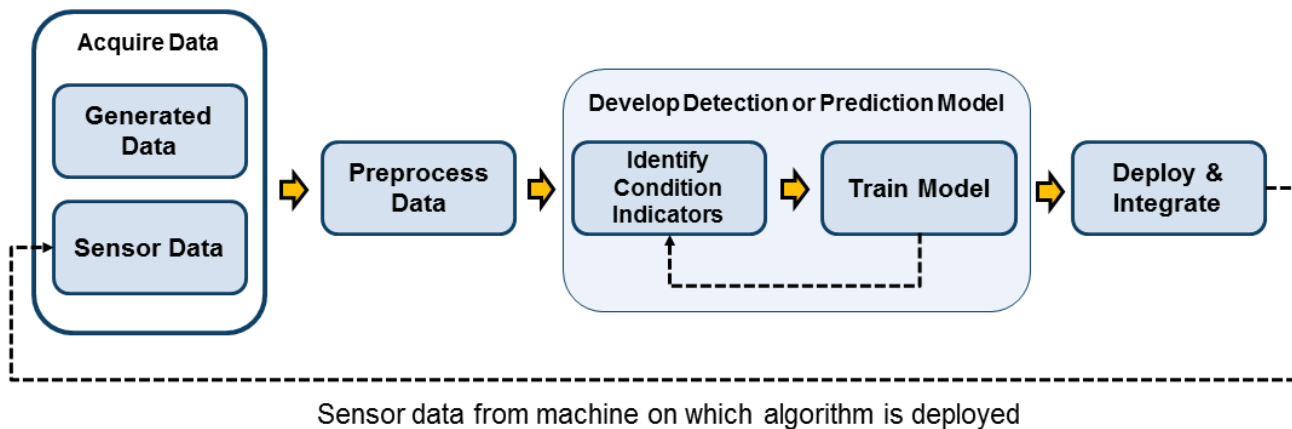
(For information about using condition indicators for fault prediction, see "Models for Predicting Remaining Useful Life" on page 5-4.)

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 5-37 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 1-25 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".

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 4-29 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".

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 1-25 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 4-29 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 1-43 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 4-29 uses a `TreeBagger` classifier.

For more general information about machine-learning techniques for classification, see "Classification".

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 4-84 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".

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 4-84 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.

Ball bearing cross-section (Magnified)
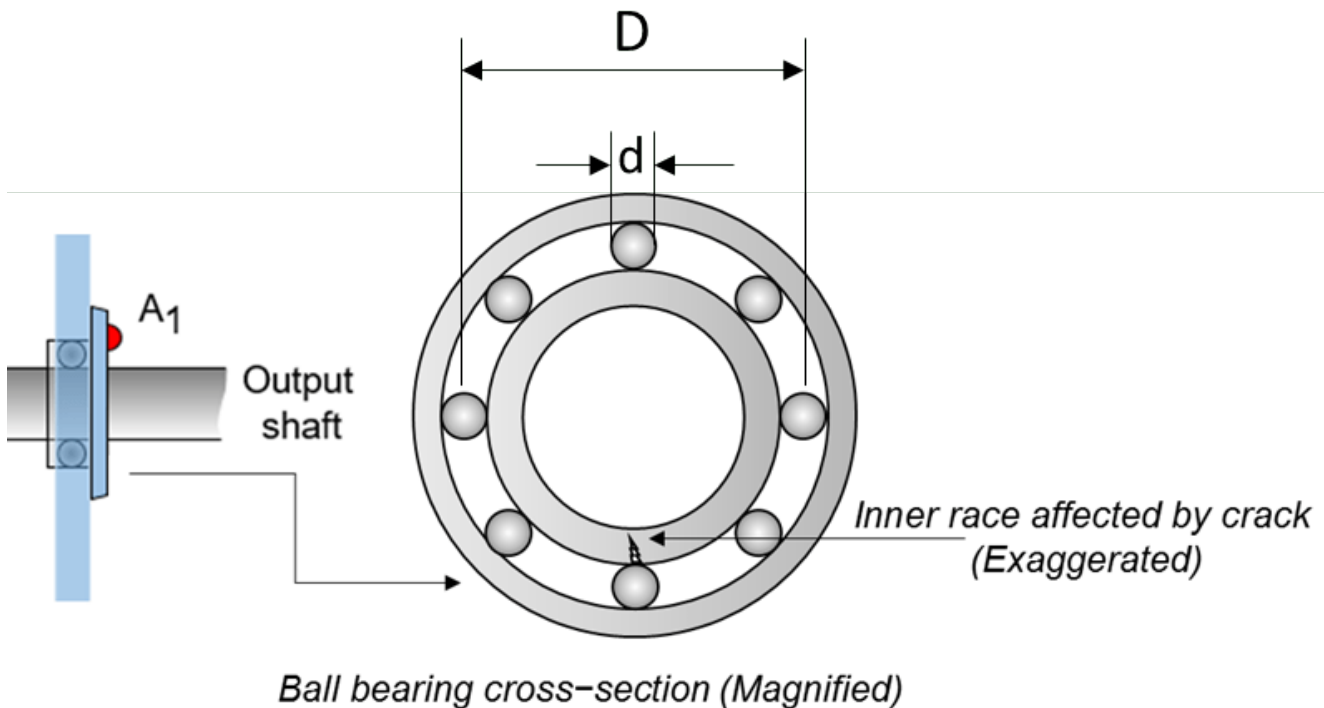
Machinery Failure Prevention Technology (MFPT) Challenge Data

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

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

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 the 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])
```

Raw Signal: Inner Race Fault
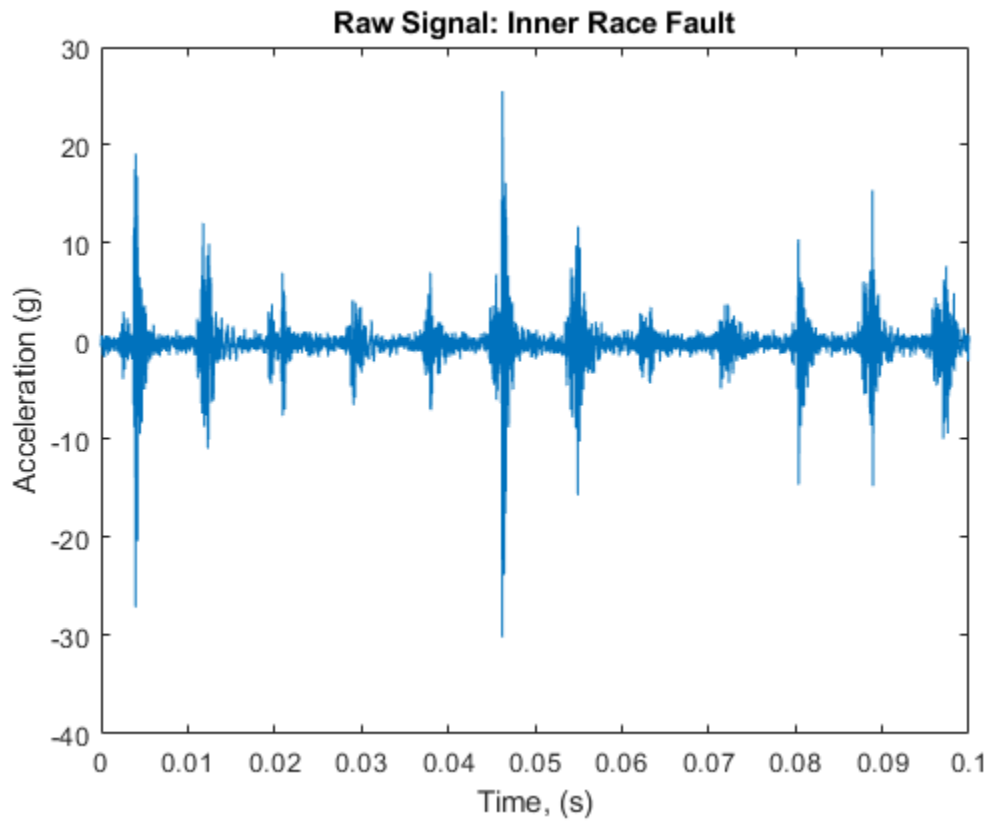
Visualize the raw data in frequency domain.

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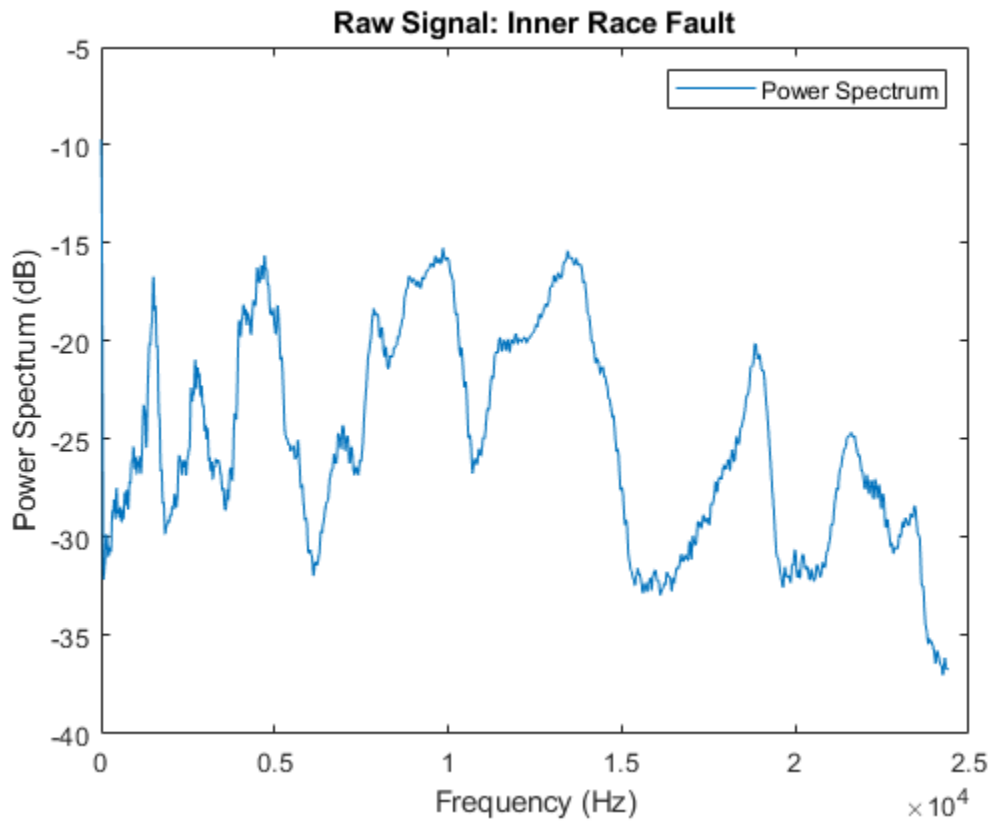
Now zoom in the power spectrum of the raw signal in low frequency range to take a closer look at the frequency response at BPFI and its first several harmonics.

```
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])
```

Raw Signal: Inner Race Fault
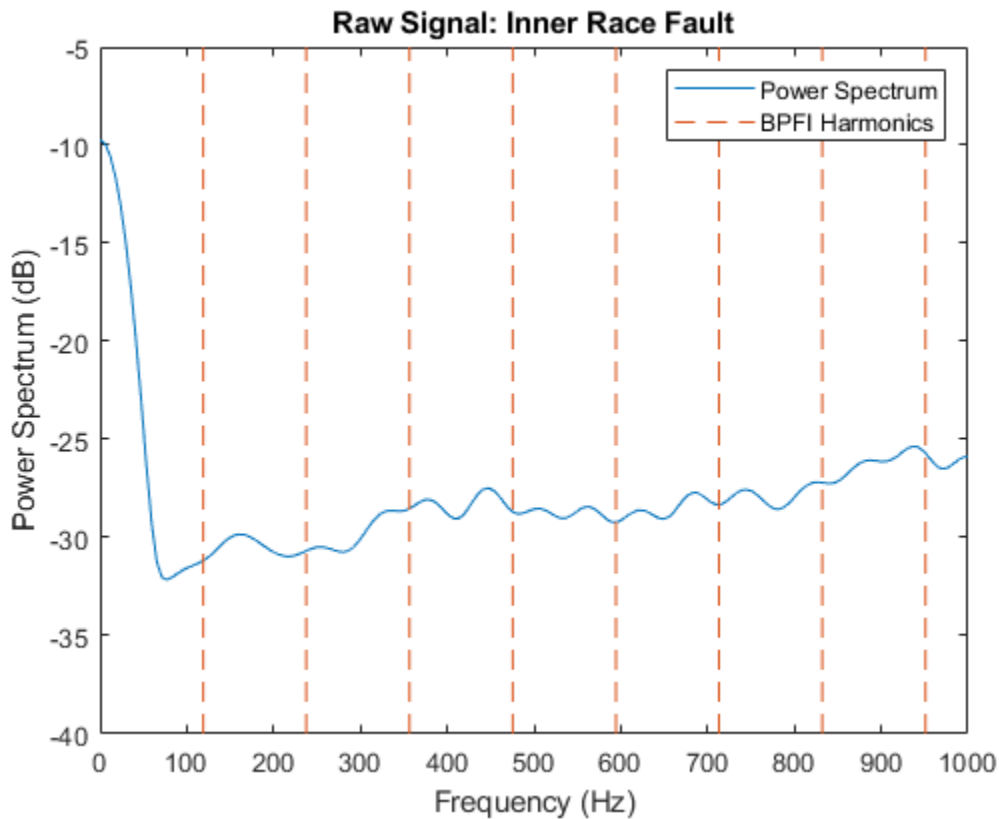
No clear pattern is visible at BPFI and its harmonics. Frequency analysis on the raw signal does not provide useful diagnosis information.

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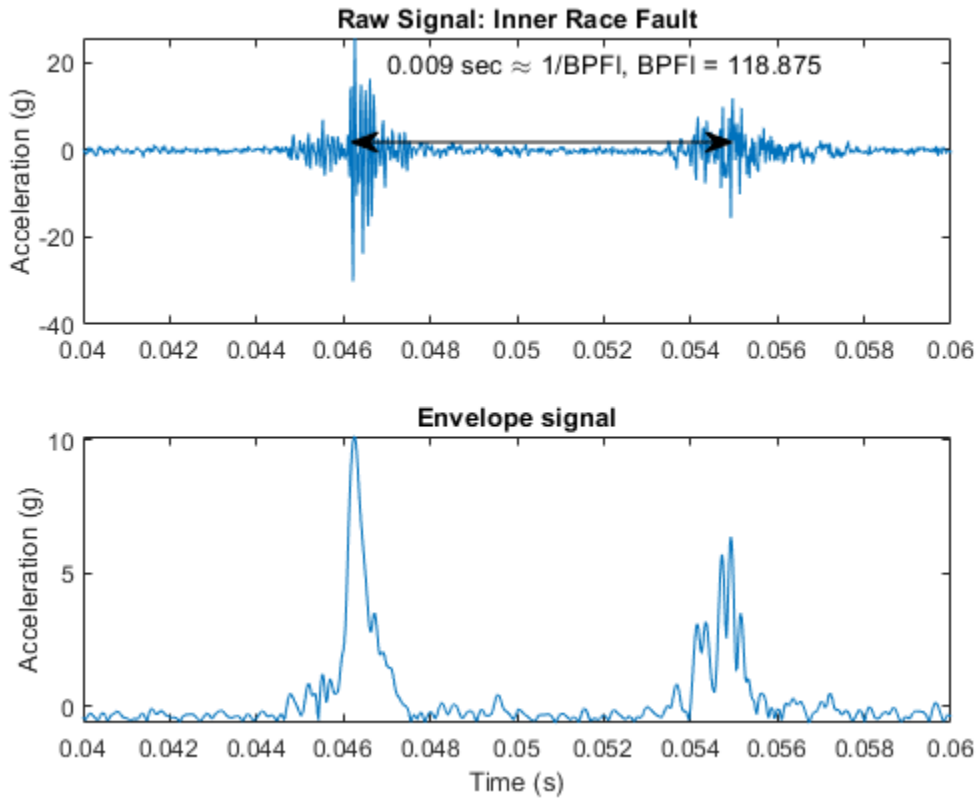
Now compute the power spectrum of the envelope signal and take a look at the frequency response at BPFI and its harmonics.

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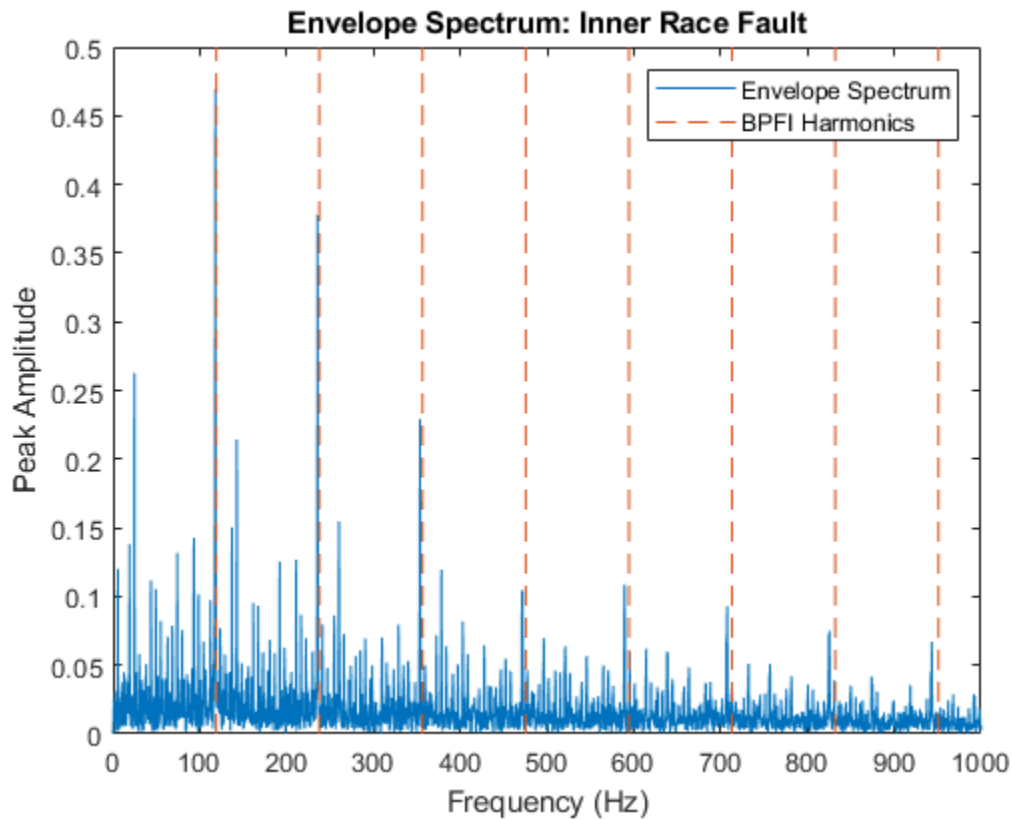
It is shown that most of the energy is focused at BPFI and its harmonics. That indicates an inner race fault of the bearing, which matches the fault type of the data.

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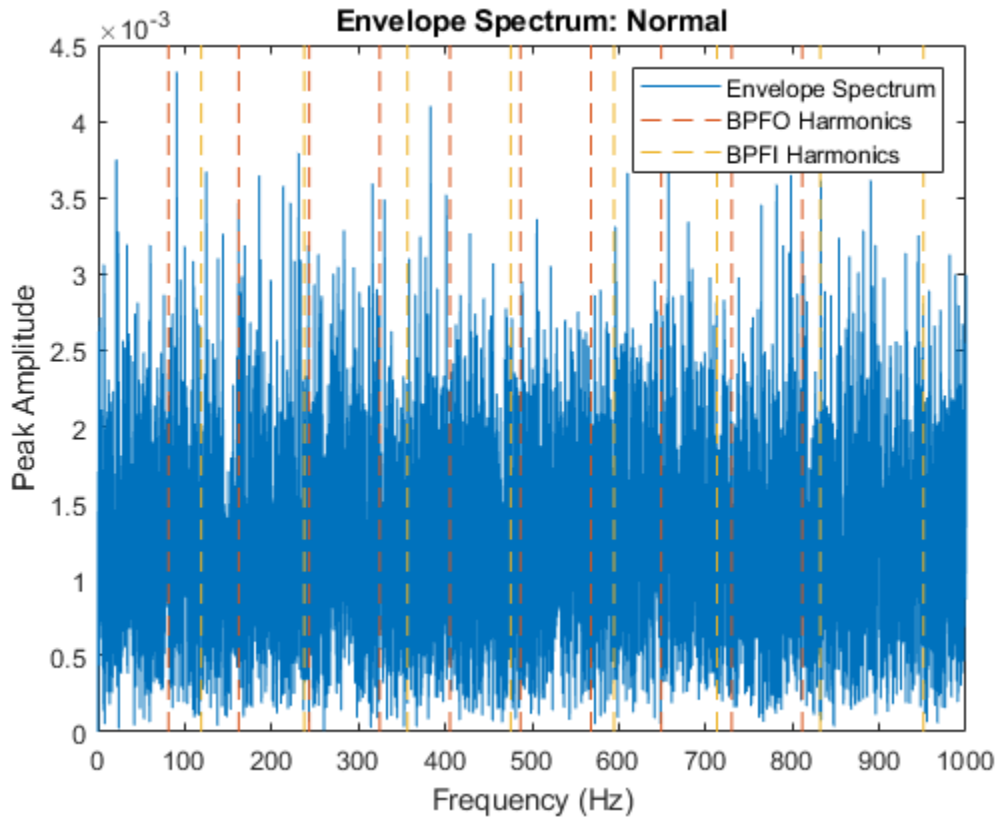
As expected, the envelope spectrum of a normal bearing signal does not show any significant peaks at BPFO or BPFI.

```
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')
```

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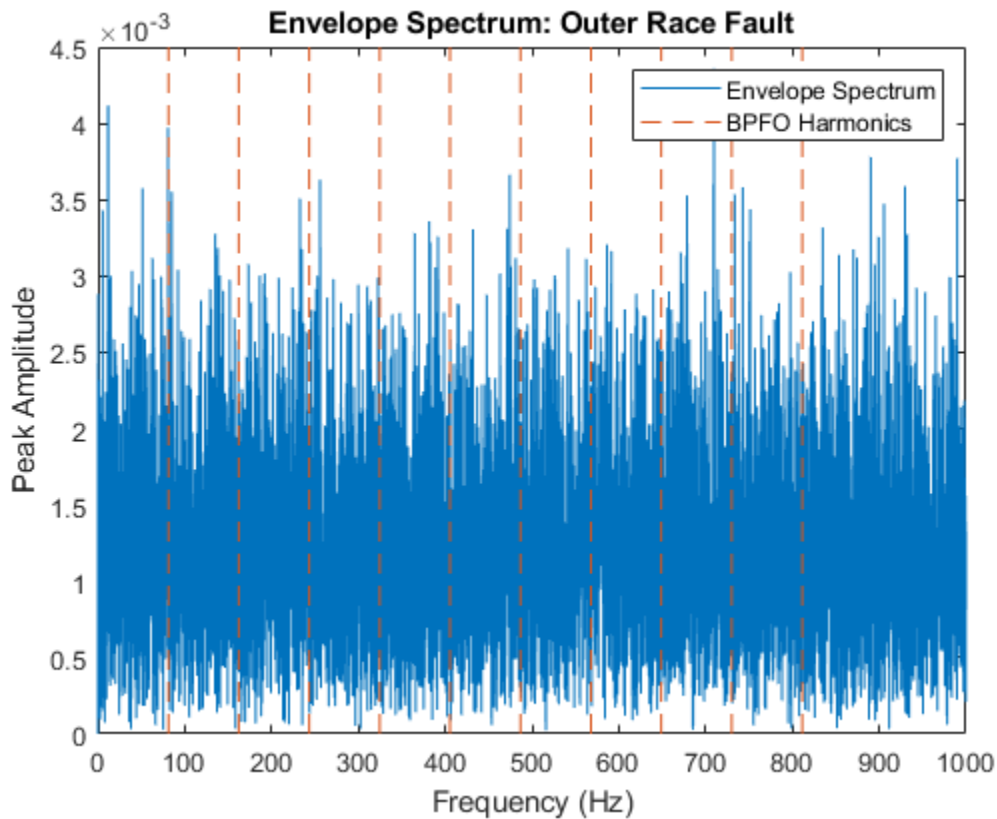
For an outer race fault signal, there are no clear peaks at BPFO harmonics either. Does envelope spectrum analysis fail to differentiate bearing with outer race fault from healthy bearings? Let's take a step back and look at the signals in time domain under different conditions again.

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])
```
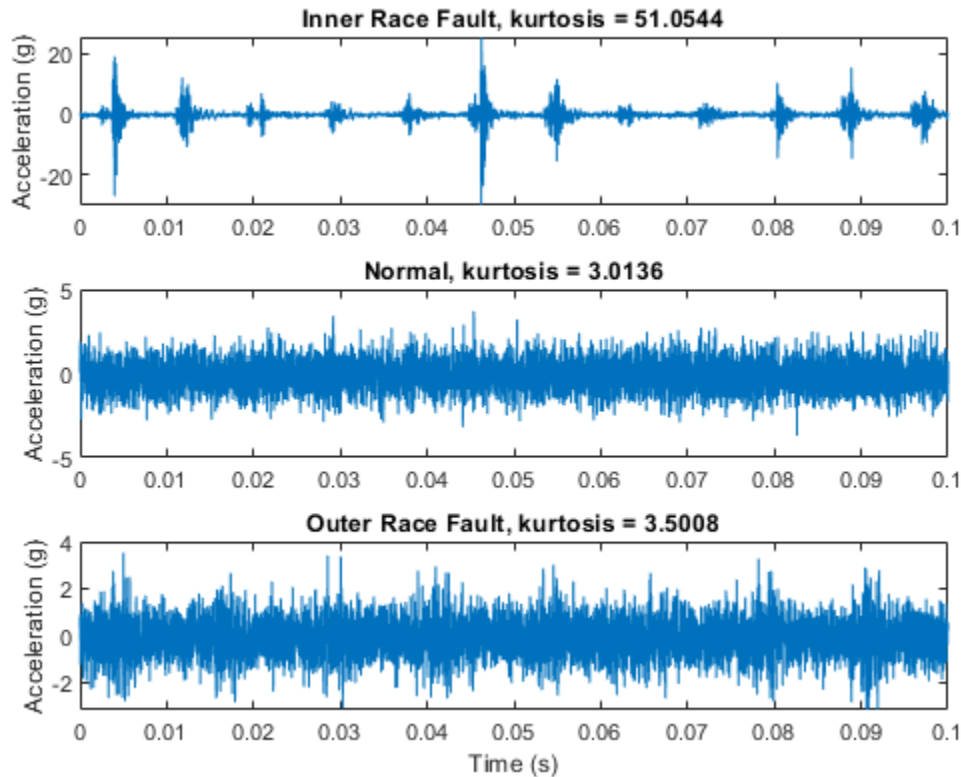

It is shown that inner race fault signal has significantly larger impulsiveness, making envelope spectrum analysis capture the fault signature at BPFI effectively. For an outer race fault signal, the amplitude modulation at BPFO is slightly noticeable, but it is masked by strong noise. The normal signal 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)
```

K$_{max}$ = 2.7083 at level 6, Optimal Window Length = 128,
Center Frequency = 2.6703 kHz, Bandwidth = 0.76294 kHz
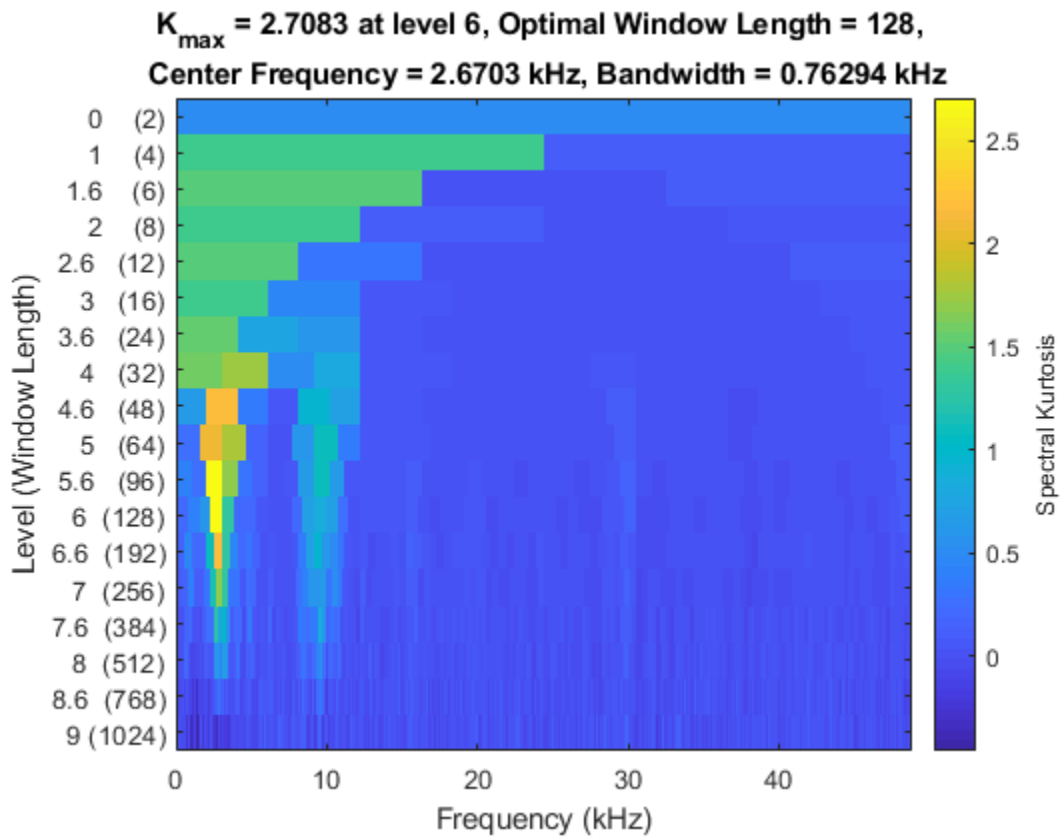
The kurtogram indicates that the frequency band centered at 2.67 kHz with a 0.763 kHz bandwidth has the highest kurtosis of 2.71.

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

```
figure
wc = 128;
pkurtosis(xOuter, fsOuter, wc)
```
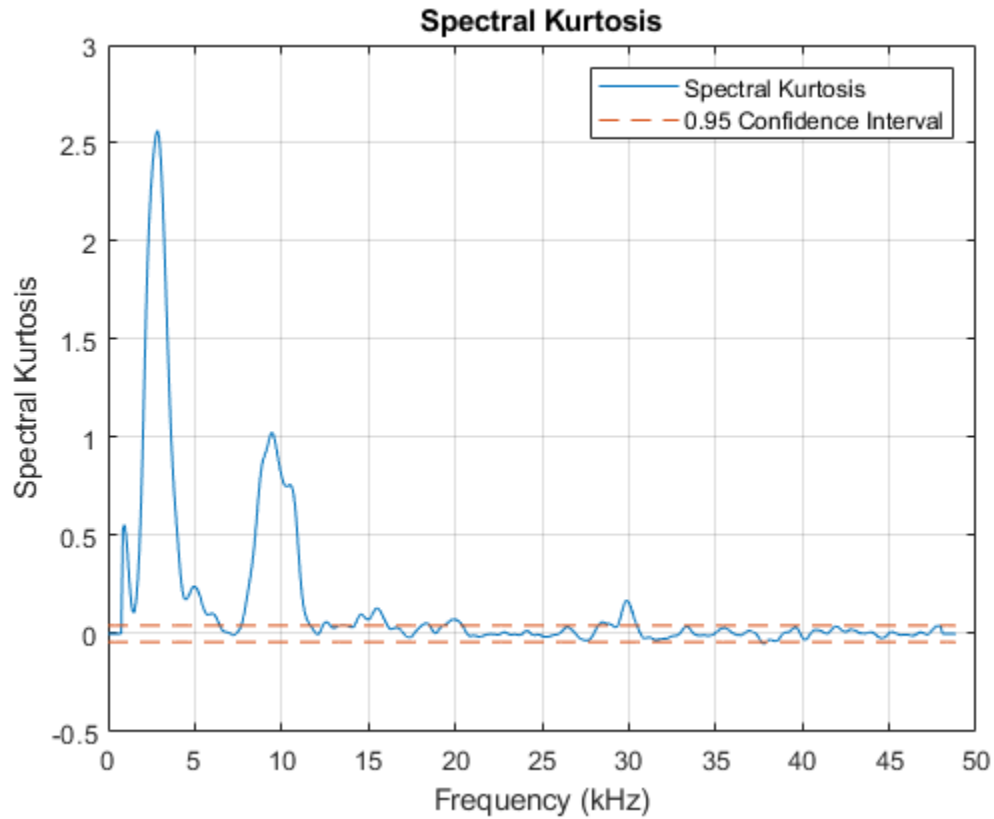
To visualize the frequency band on a spectrogram, compute the spectrogram and place the spectral kurtosis on the side. To interpret the spectral kurtosis in another way, high spectral kurtosis values indicates high variance of power at the corresponding frequency, which makes spectral kurtosis a useful tool to locate nonstationary components of the signal [3].

```
helperSpectrogramAndSpectralKurtosis(xOuter, fsOuter, level)
```
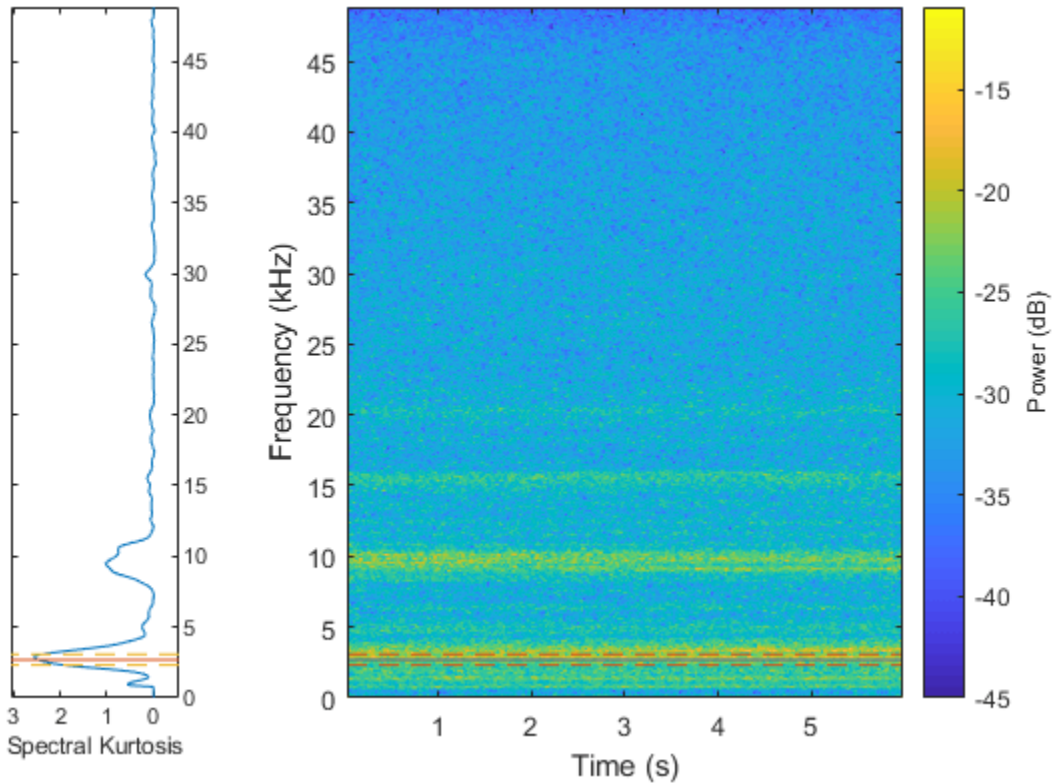
By bandpass filtering the signal with the suggested center frequency and bandwidth, the kurtosis can be enhanced and the modulated amplitude of the outer race fault can be retrieved.

```
[~, ~, ~, 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')
```
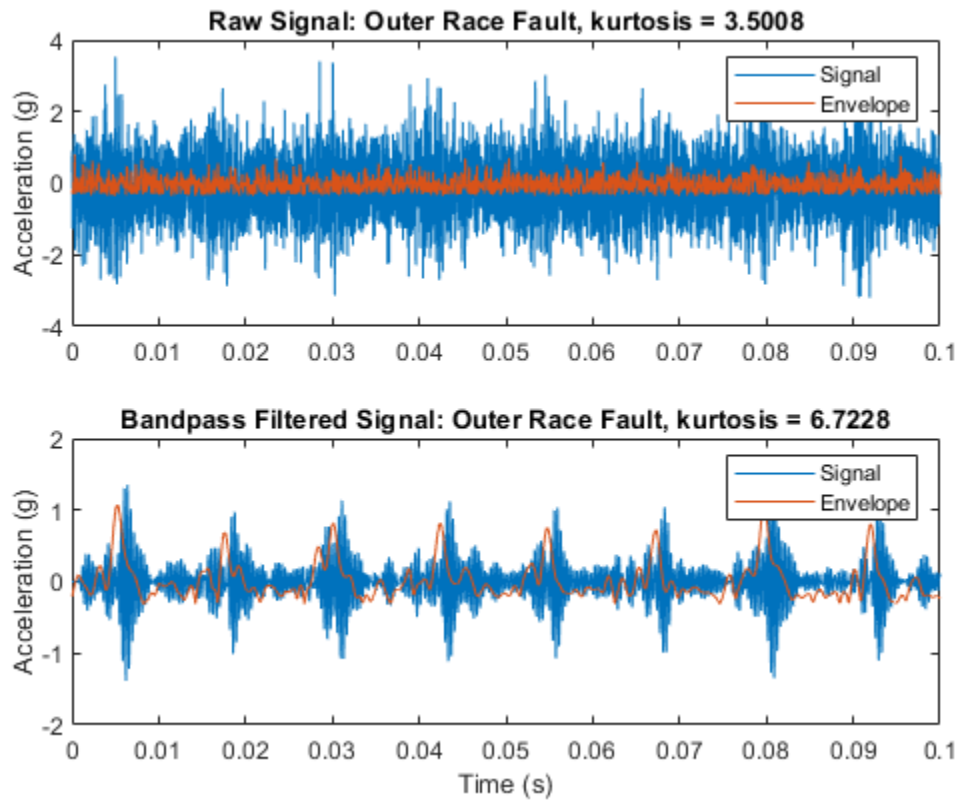
It can be seen that the kurtosis value is increased after bandpass filtering. Now visualize the envelope signal in frequency domain.

```
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')
```

Envelope Spectrum of Bandpass Filtered Signal: Outer Race Fault
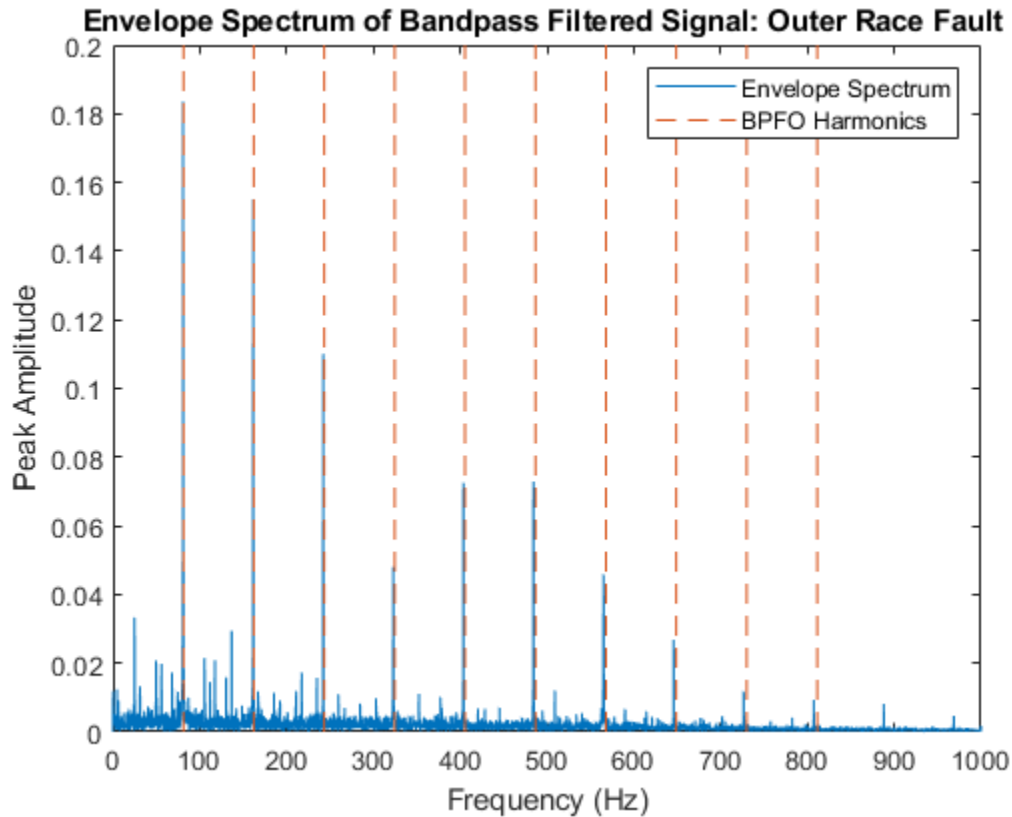
It is shown that by bandpass filtering the raw signal with the frequency band suggested by kurtogram and spectral kurtosis, the envelope spectrum analysis is able to reveal the fault signature at BPFO and its harmonics.

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', '*.mat'), '-
fileattrib(fullfile('RollingElementBearingFaultDiagnosis-Data-master', 'test_data', '*.mat'), '+w
```

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_data');
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", "BSF", "Lal
```

```
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           La
    _____    _____   ____    ____    _____   _____   _____   _____    _____

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

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"; "BPFOAmplitude"];
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
```
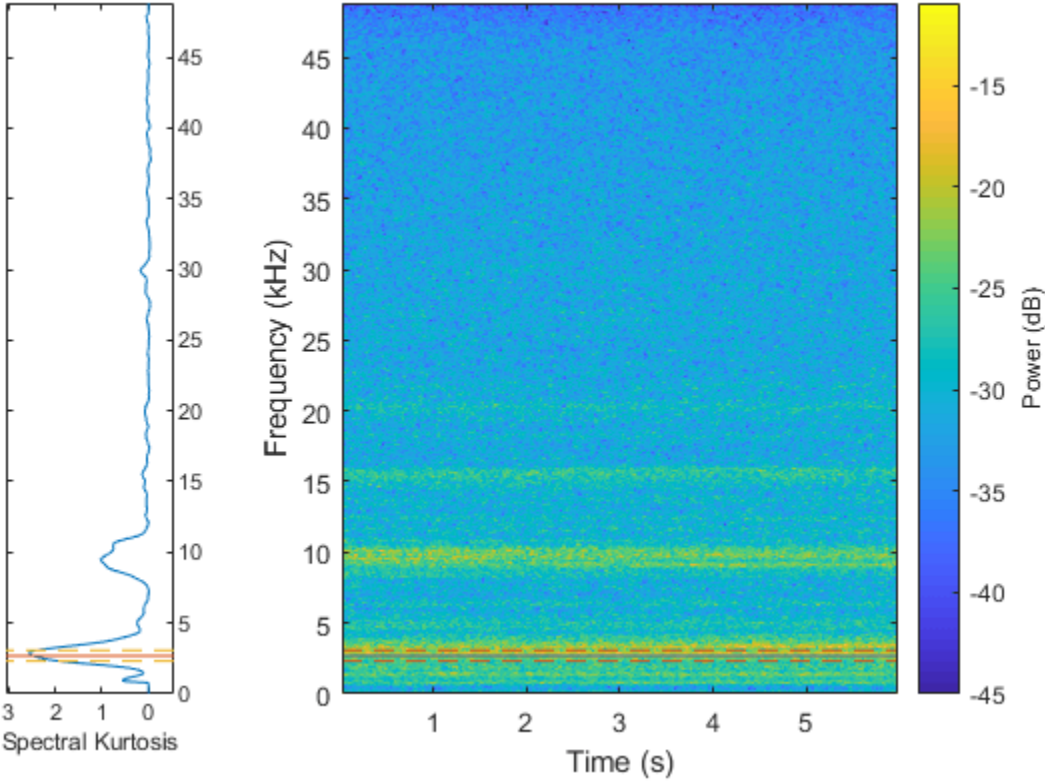
```
- Pass 1 of 1: Completed in 3 sec
Evaluation completed in 3 sec
```

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, featureTableTrain.Labe
xlabel('BPFI Amplitude')
ylabel('BPFO Amplitude')
```
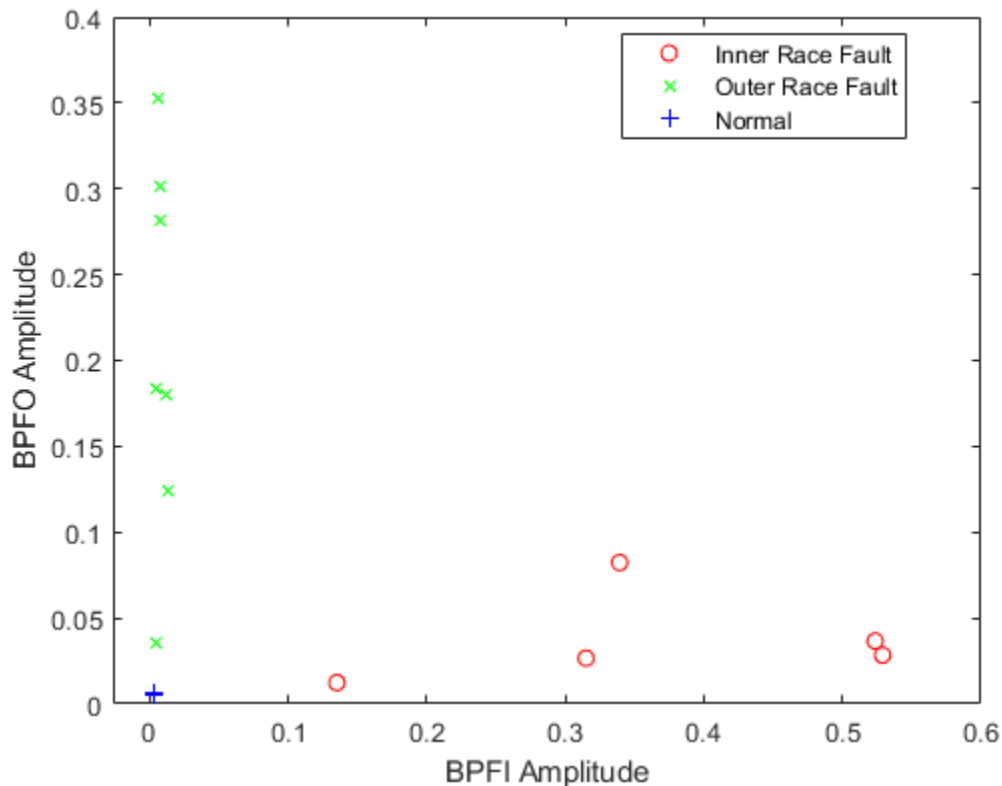

The relative values of BPFI Amplitude and BPFO Amplitude might be an effective indicator of different fault types. Here a new feature is created, which is the log ratio of the two existing features, and is visualized in a histogram grouped by different fault types.

```
featureTableTrain.IOLogRatio = log(featureTableTrain.BPFIAmplitude./featureTableTrain.BPFOAmplitu
figure
hold on
histogram(featureTableTrain.IOLogRatio(featureTableTrain.Label=="Inner Race Fault"), 'BinWidth',
histogram(featureTableTrain.IOLogRatio(featureTableTrain.Label=="Outer Race Fault"), 'BinWidth',
histogram(featureTableTrain.IOLogRatio(featureTableTrain.Label=="Normal"), 'BinWidth', 0.5)
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')
```
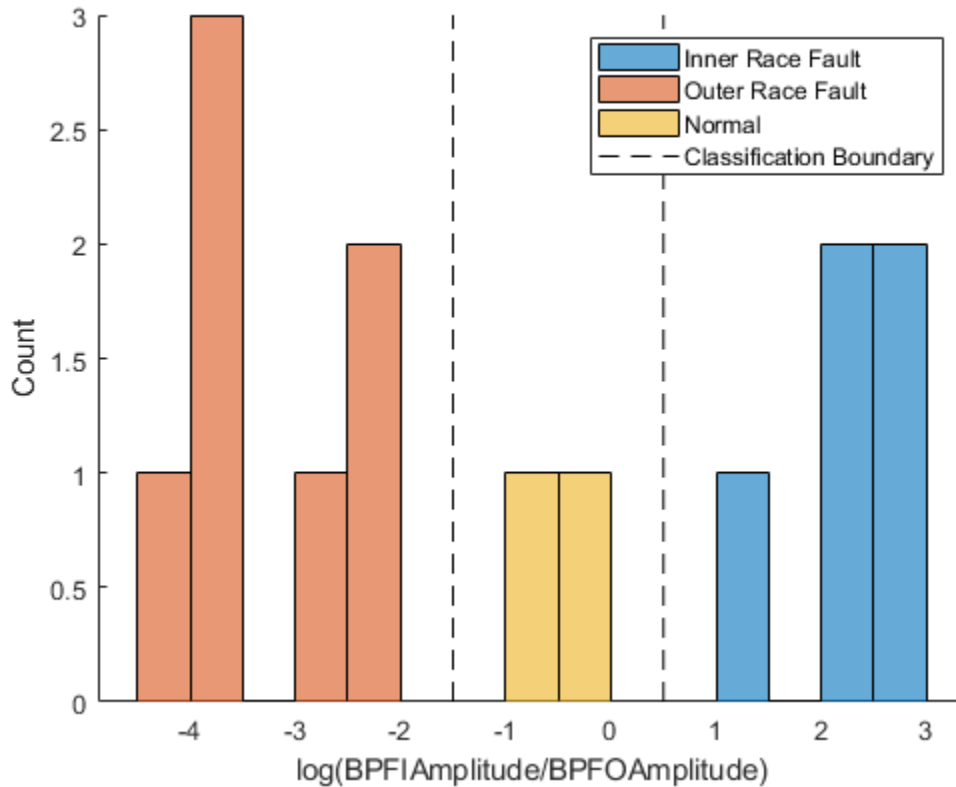
The histogram shows a clear separation among the three different bearing conditions. The log ratio between the BPFI and BPFO amplitudes is a valid feature to classify bearing faults. To simplify the example, a very simple classifier is derived: if $\log\left(\frac{\text{BPFIAmplitude}}{\text{BPFOAmplitude}}\right) \leq -1.5$, 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_data');
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", "BSF", "Labe
```

```
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]
```

```matlab
ensembleTest.DataVariables = [ensembleTest.DataVariables; "BPFIAmplitude"; "BPFOAmplitude"];
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
```

```matlab
featureTableTest.IOLogRatio = log(featureTableTest.BPFIAmplitude./featureTableTest.BPFOAmplitude)

figure
hold on
histogram(featureTableTrain.IOLogRatio(featureTableTrain.Label=="Inner Race Fault"), 'BinWidth',
histogram(featureTableTrain.IOLogRatio(featureTableTrain.Label=="Outer Race Fault"), 'BinWidth',
histogram(featureTableTrain.IOLogRatio(featureTableTrain.Label=="Normal"), 'BinWidth', 0.5)

histogram(featureTableTest.IOLogRatio(featureTableTest.Label=="Inner Race Fault"), 'BinWidth', 0
histogram(featureTableTest.IOLogRatio(featureTableTest.Label=="Outer Race Fault"), 'BinWidth', 0
histogram(featureTableTest.IOLogRatio(featureTableTest.Label=="Normal"), 'BinWidth', 0.1)
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')
```
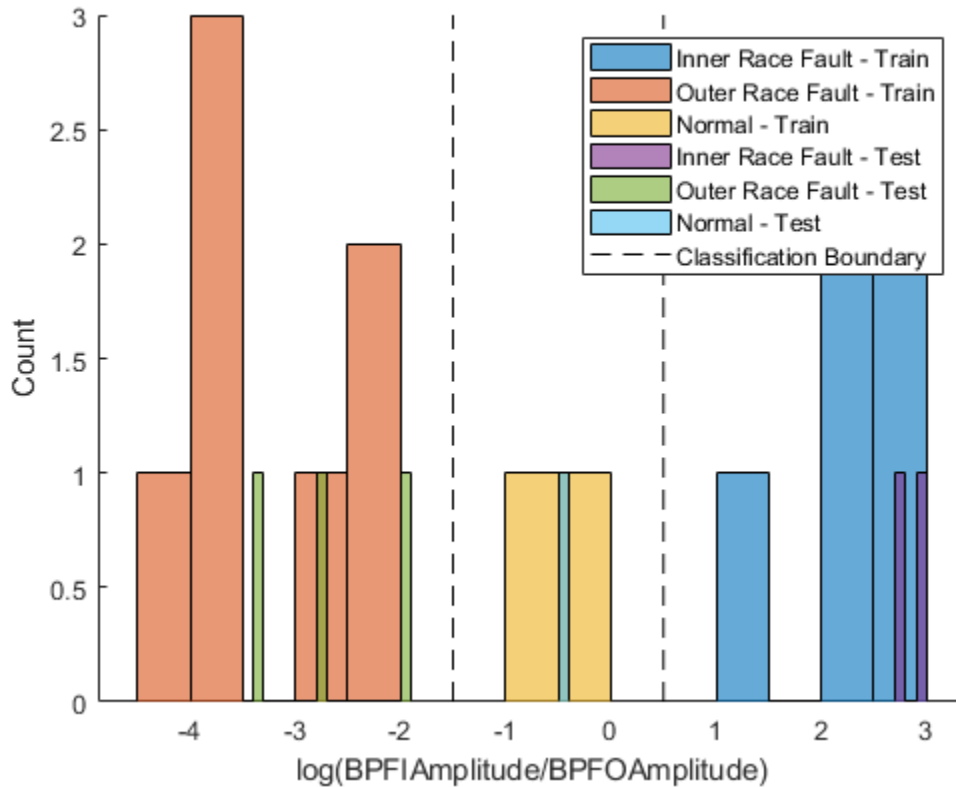
The log ratio of BPFI and BPFO amplitudes from test data sets shows consistent distribution with the log ratio from training data sets. The naive classifier obtained in the last section achieved perfect accuracy on the test data set.

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*. Vol. 25, Number 2, 2011, pp. 485–520.

[2] Antoni, Jérôme. "Fast computation of the kurtogram for the detection of transient faults." *Mechanical Systems and Signal Processing*. Vol. 21, Number 1, 2007, pp. 108–124.

[3] Antoni, Jérôme. "The spectral kurtosis: a useful tool for characterising non-stationary signals." *Mechanical Systems and Signal Processing*. Vol. 20, Number 2, 2006, pp. 282–307.

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

Helper Functions

```matlab
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]) min([fc+BW
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', {'BPFIAmpli
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.
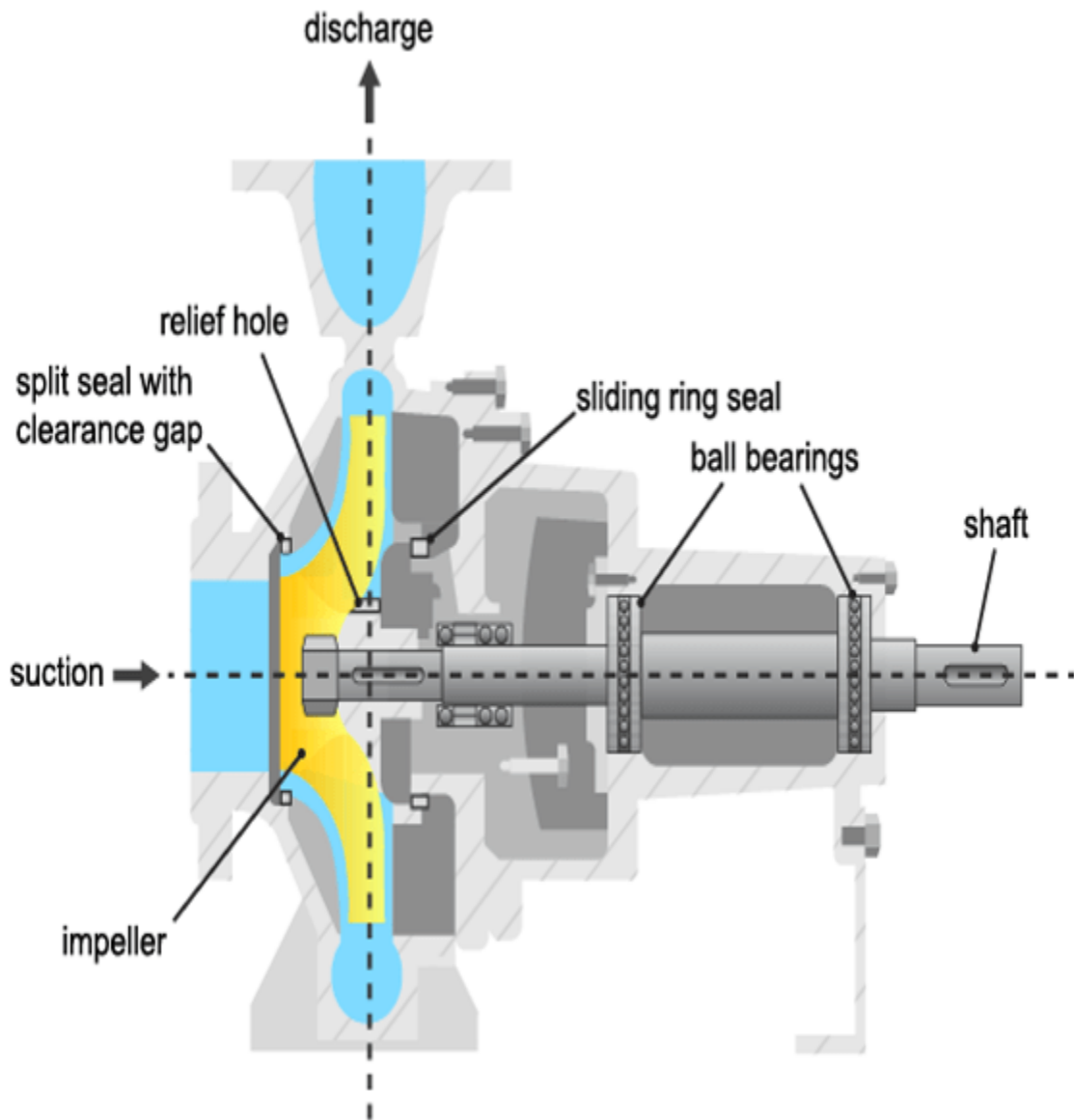
Pumps experience damage to their hydraulic or mechanical components. The most frequently faulting components are the sliding ring seals and the ball bearings, although failures to other components including the driving motor, impeller blades and sliding bearings are also not uncommon. The table below 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 $H_{th} = \dfrac{\Delta p}{\rho g}$ 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, $a_F = \dfrac{l}{gA}$ 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.

```
url = 'https://www.mathworks.com/supportfiles/predmaint/fault-diagnosis-of-centrifugal-pumps-usin
websave('PumpCharacteristicsData.mat',url);
load PumpCharacteristicsData Q0 H0 M0 % manufacturer supplied data for pump's delivery head
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 clearance gap
hold on
plot(Q1, H1);

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

Pump Delivery Head Characteristics at 2900 RPM
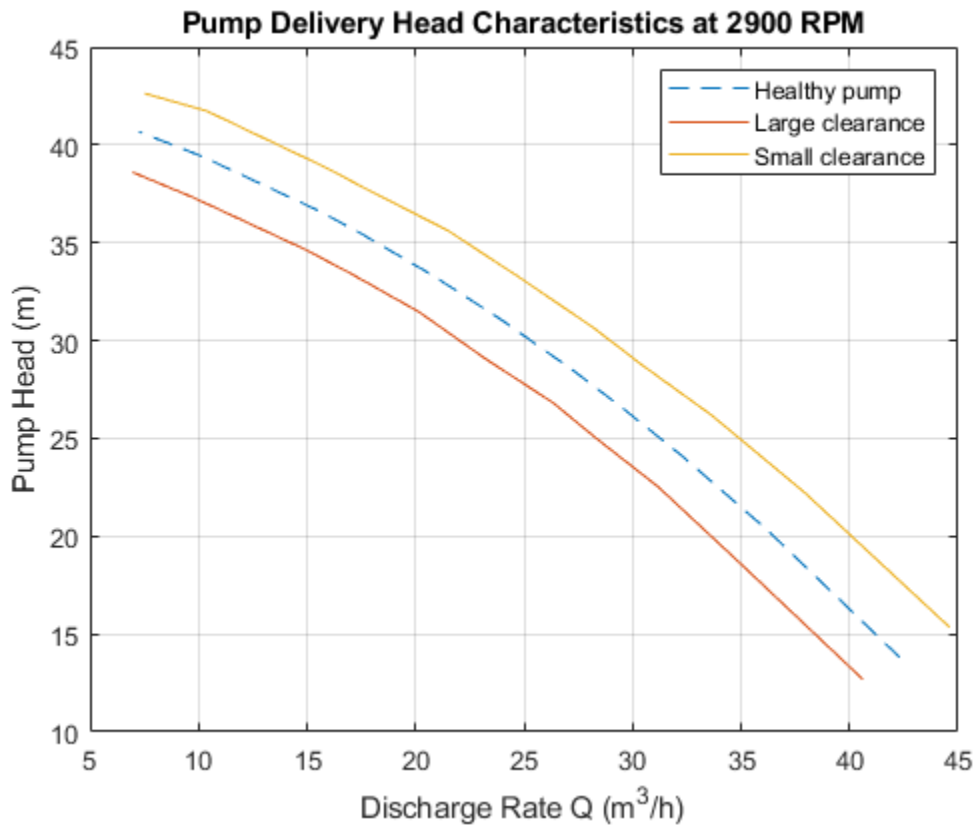
Similar changes can be seen in torque-flow characteristics and for other fault types.

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.

```
url = 'https://www.mathworks.com/supportfiles/predmaint/fault-diagnosis-of-centrifugal-pumps-usin
websave('FaultDiagnosisData.mat',url);
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)')
```

Pump Head Characteristics Ensemble for 5 Test Runs
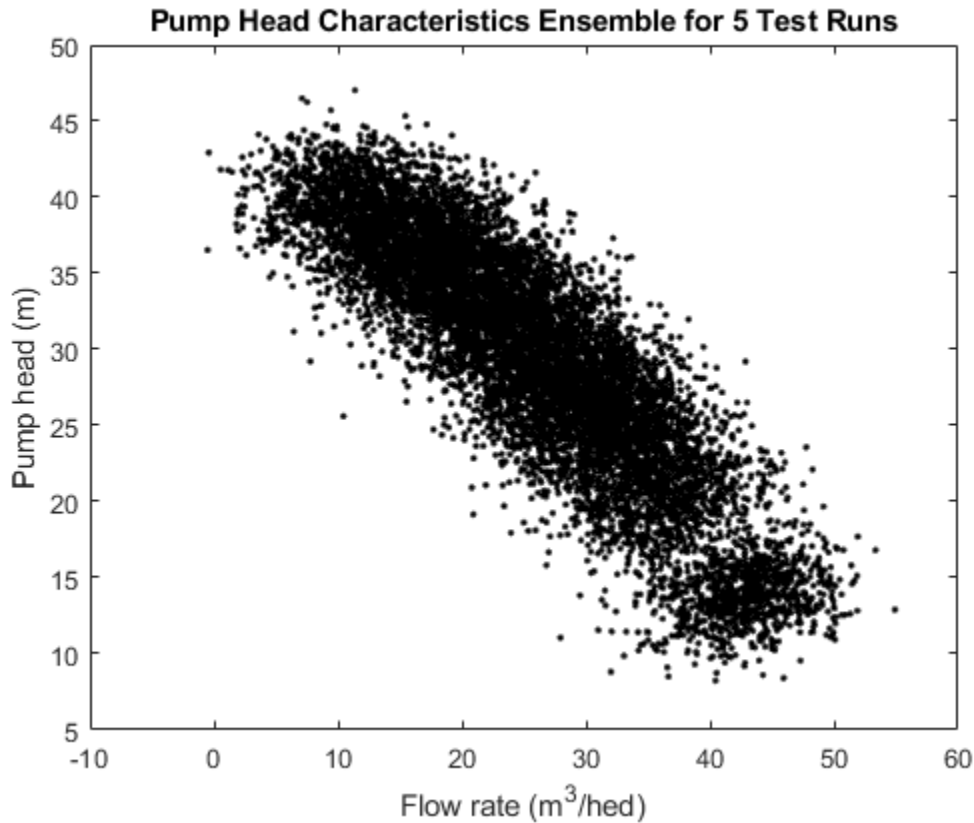
The plot shows variation in characteristics even for a healthy pump under realistic conditions. These variations have to be taken into consideration for making the fault diagnosis reliable. The next sections discuss fault detection and isolation techniques for noisy data.

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 clearance gap 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])
```
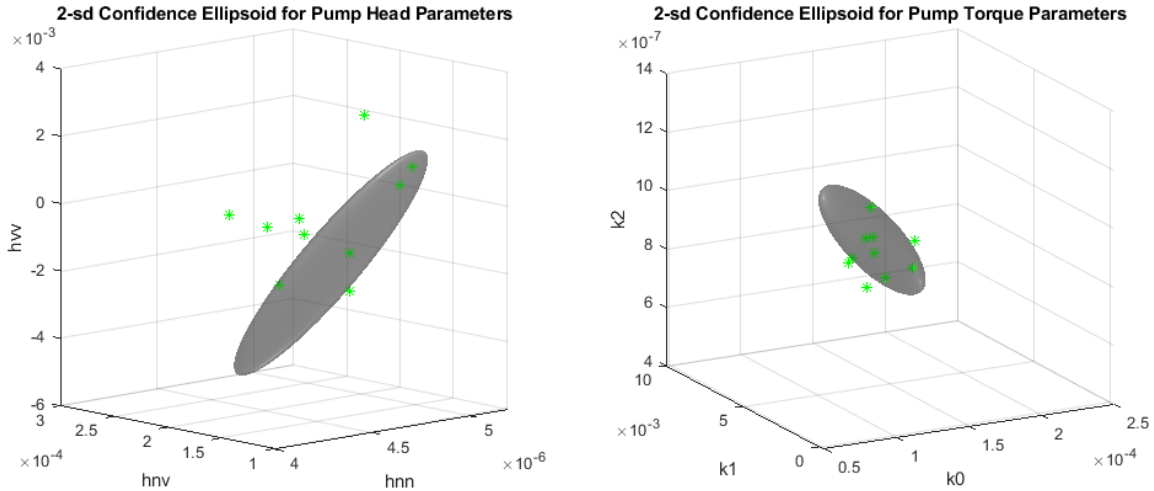
Each green star marker is contributed by one test pump. The markers that are outside the confidence bounds can be treated as outliers while those inside are either from a healthy pump or escaped detection. Note that a marker from a particular pump may be marked as anomalous in the pump head view but 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 $2\verb|^|2 = 4$ 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])
```
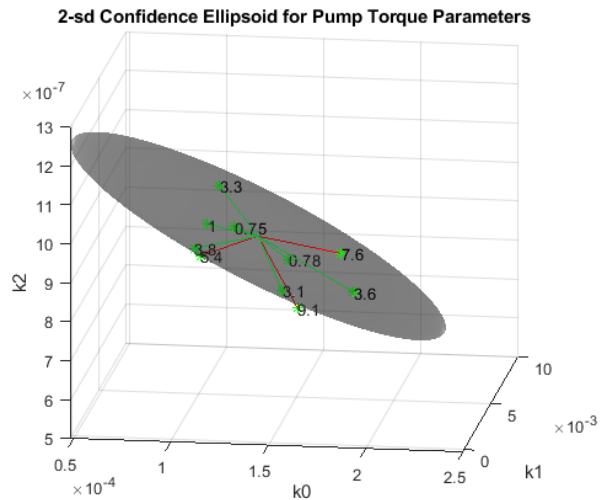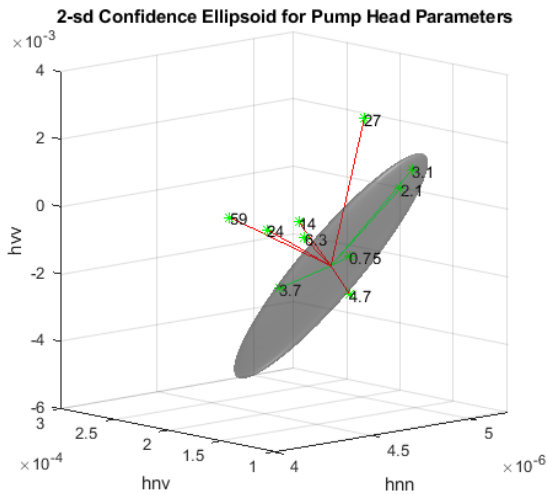

The plots now not only show detection of the anomalous samples but also quantify their severity.

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')
```
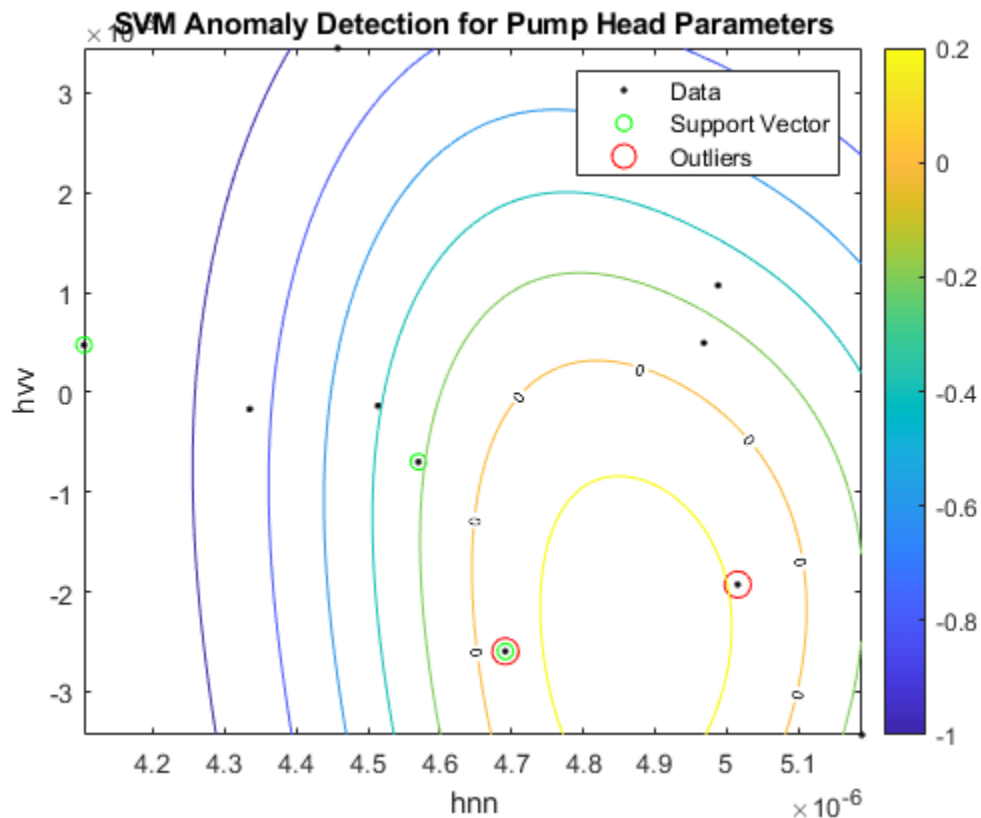

The boundary separating the outliers from the rest of the data occurs where the contour value is 0; this is the level curve marked with "0" in the plot. The outliers are marked with red circles. A similar analysis can be performed for torque parameters.

```
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')
```
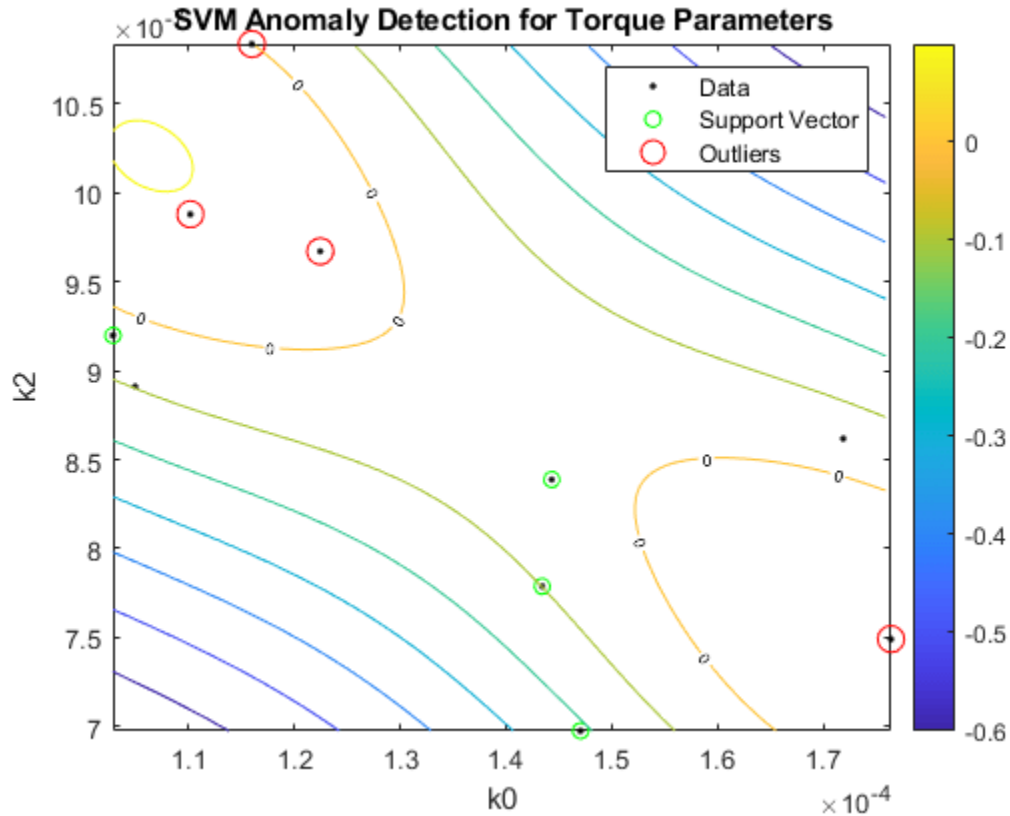

A similar analysis can be carried out for detecting other kinds of faults such as wear or deposits at impeller outlet as discussed next in the context of fault isolation.

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 Clearance Gap Faults by Likelihood Ratio Tests

Changes in clearance gap 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 clearance gap 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 clearance gap (healthy behavior)
- Mode 2: Large clearance gap
- Mode 3: Small clearance gap

```
url = 'https://www.mathworks.com/supportfiles/predmaint/fault-diagnosis-of-centrifugal-pumps-usin
websave('LabeledGapClearanceData.mat',url);
load LabeledGapClearanceData HealthyEnsemble LargeGapEnsemble SmallGapEnsemble
```

4-41

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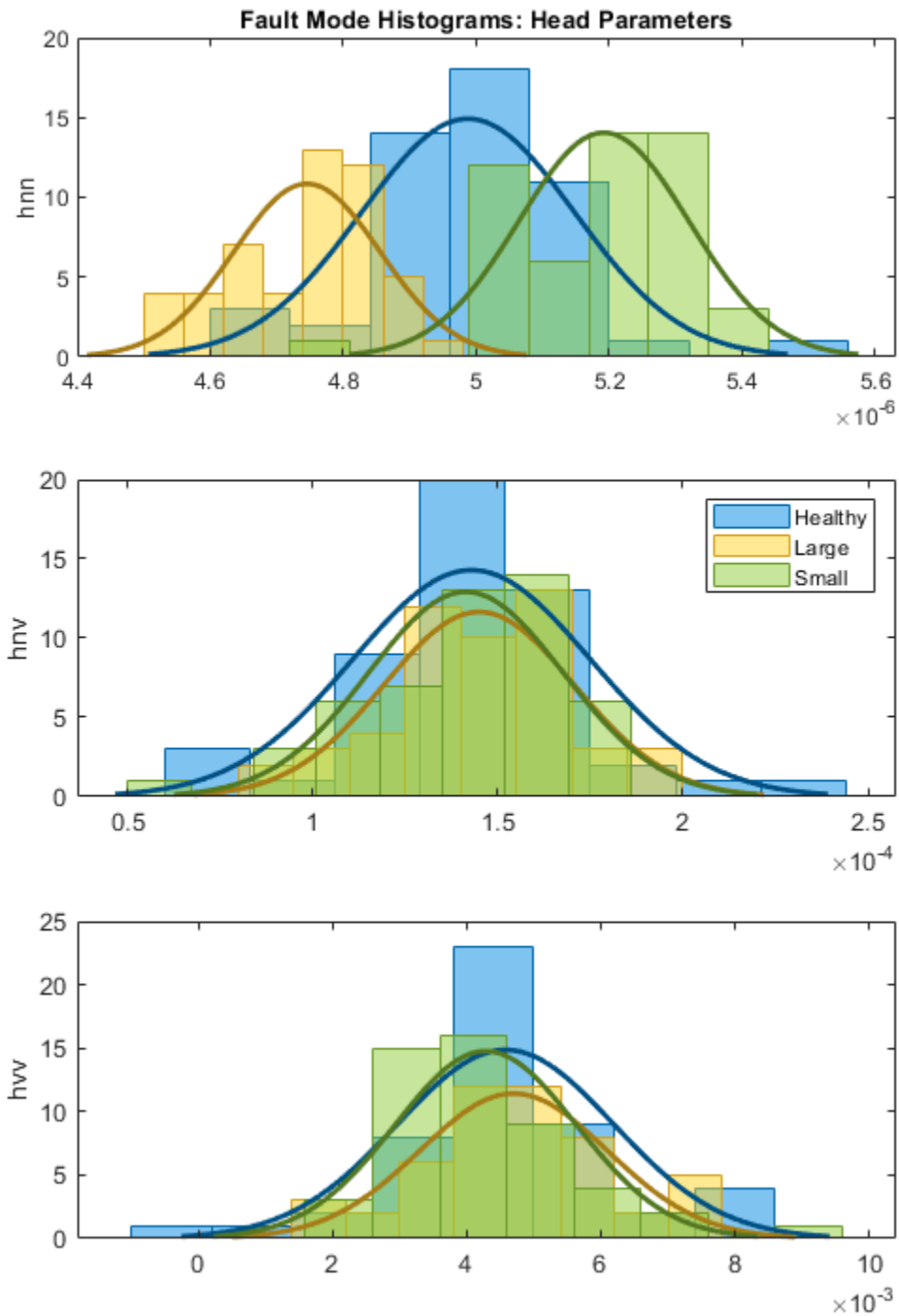
The histogram shows that h_{nn} offers good separability among the three modes but h_{nv}, h_{vv} parameters have overlapping probability distribution functions (PDFs).

Pump torque parameters:

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

Fault Mode Histograms: Torque Parameters
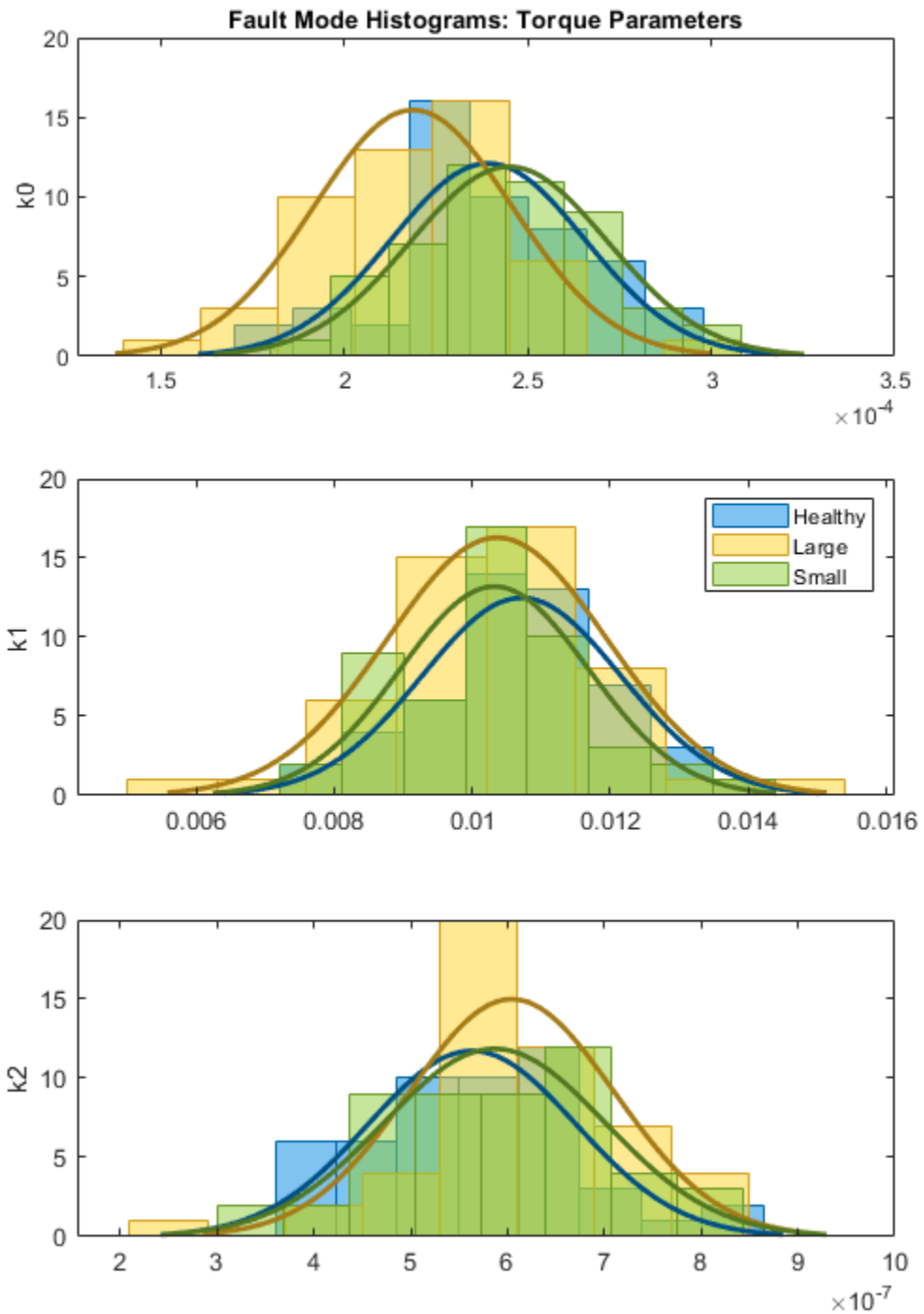
For the Torque parameters, the individual separability is not very good. There is still some variation in mean and variances that can be exploited by a trained 3-mode classifier. If the PDFs show good

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)
```
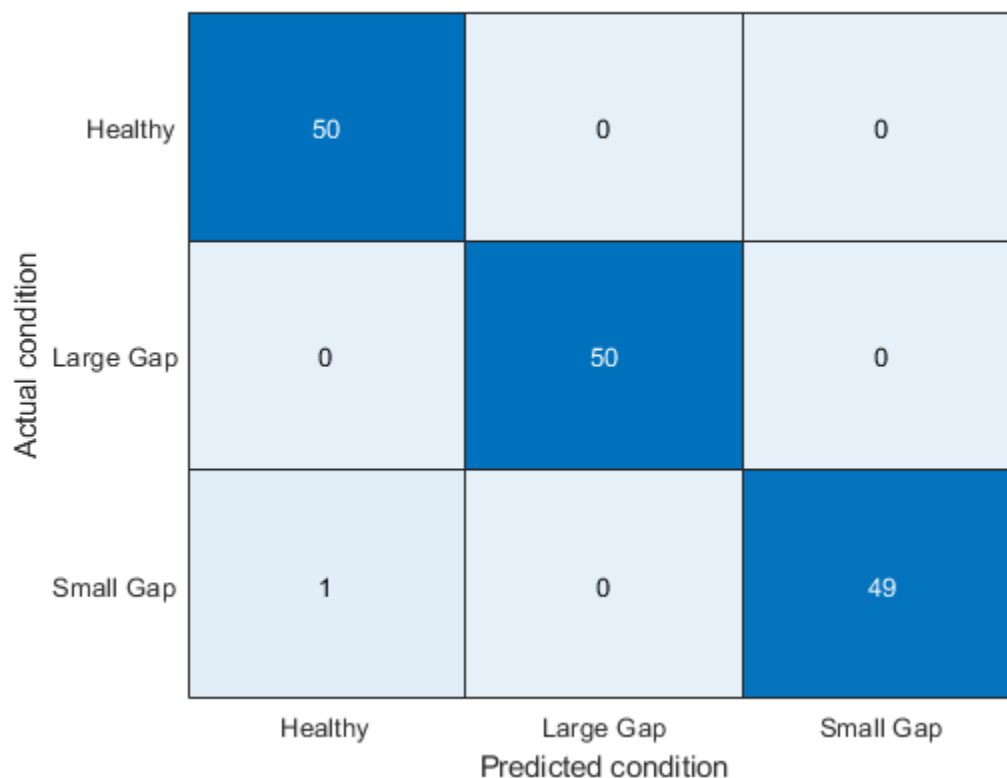

The confusion plot shows perfect separation between the three modes, which is not surprising given the clear separation among the histograms for h_{nn} parameters.

```
% 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.

```matlab
url = 'https://www.mathworks.com/supportfiles/predmaint/fault-diagnosis-of-centrifugal-pumps-usin
websave('MultipleFaultsData.mat',url);
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(1:30,:)];

% 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' 'ClearanceGapWea
```

```
   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.

```matlab
% Compute out of bag error
figure
plot(oobError(Mdl))
xlabel('Number of trees')
ylabel('Misclassification probability')
```
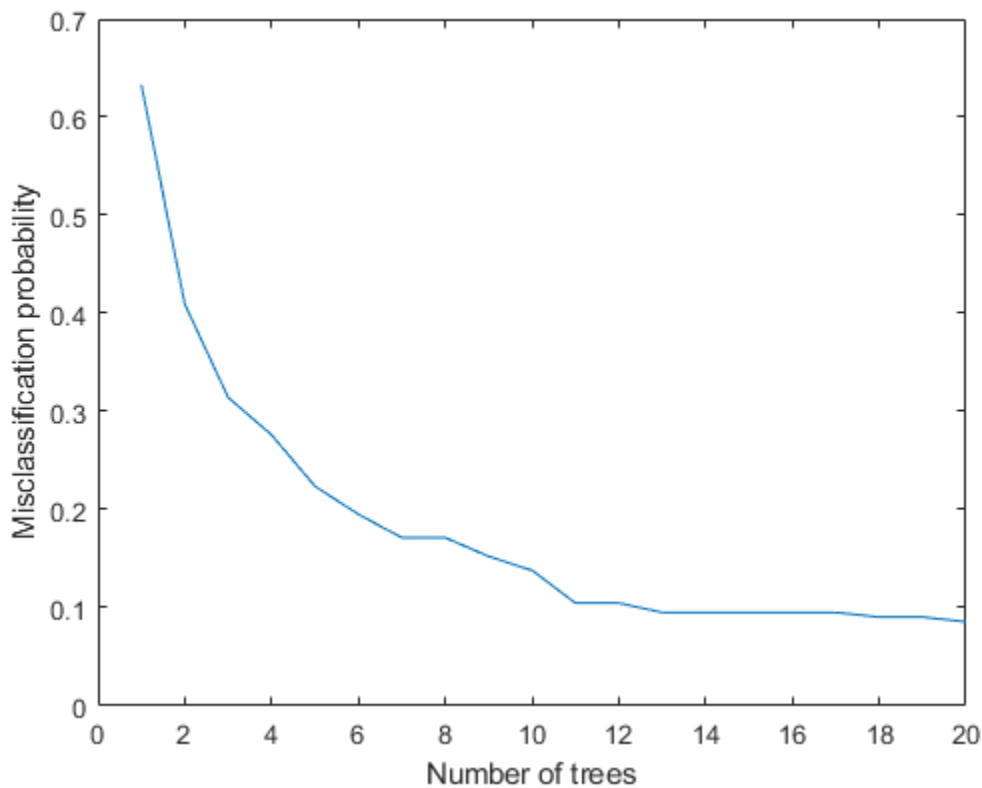

Finally, compute prediction performance of the model on test samples that were never used for growing the decision trees.

```matlab
ValidationData = [T0(31:50,:);T1(31:50,:);T2(31:50,:);T3(31:50,:);T4(31:50,:);T5(31:50,:);T6(31:5
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 (%)')
```
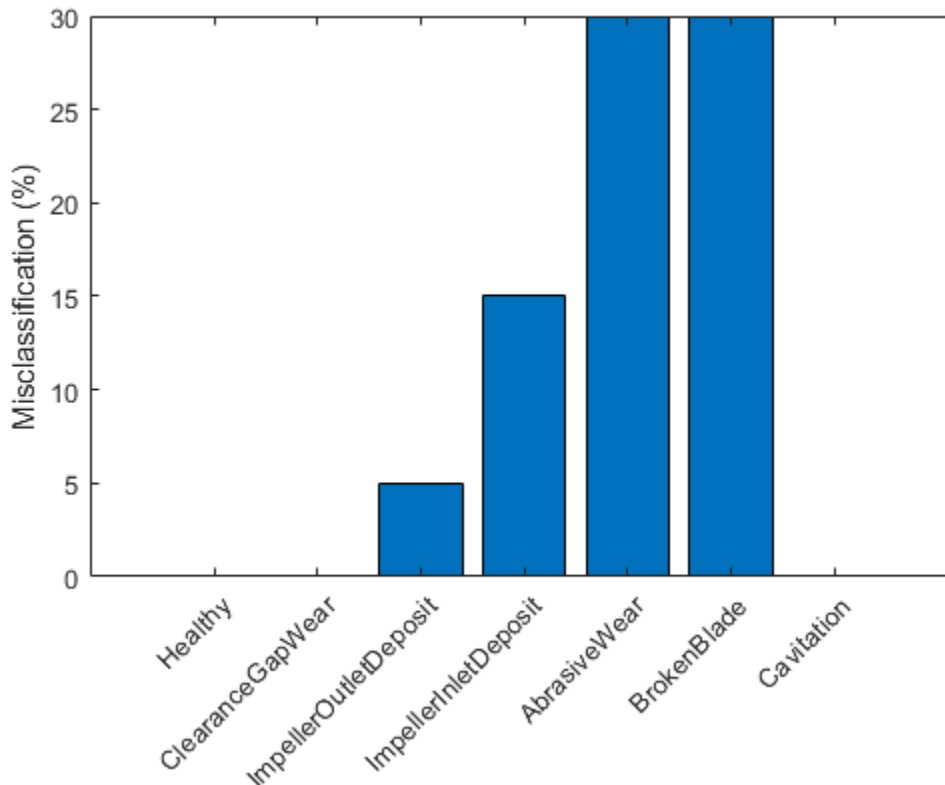

The plot shows that the abrasive wear and broken blade faults are misclassified for 30% of the validation samples. A closer look at the predicted labels shows that in the misclassified cases, the 'AbrasiveWear' and 'BrokenBlade' labels get intermixed between each other only. This suggests that the symptoms 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 4-54 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 only.
% If Form==1, accept an ensemble and return compact parameter vectors. For several experiments (e
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 matrix.

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);
```

```matlab
% 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 4-29 to the situation where the pump operation spans multiple operating conditions.

The example follows the centrifugal pump analysis presented in the Fault Diagnosis Applications book by Rolf Isermann [1]. It uses functionality from System Identification Toolbox™, Statistics and Machine Learning Toolbox™, Control System Toolbox™ and Simulink™ and does not require Predictive Maintenance Toolbox™.

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.
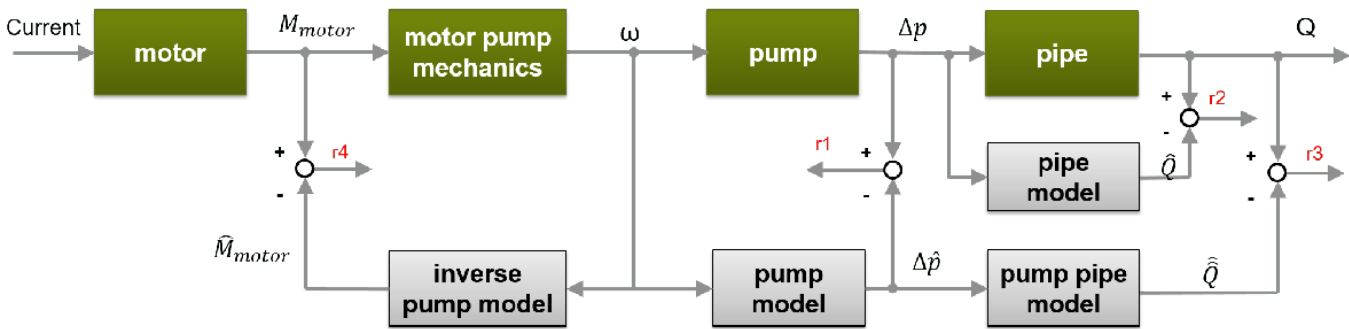
The model has of the following components:

- 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: $\widehat{\widehat{Q}}(t) = \theta_3 + \theta_4\sqrt{\widehat{\Delta p(t)}} + \theta_5\widehat{\widehat{Q}}(t-1)$

- 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 $\omega \leq 900 \ RPM$

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 (these are large datasets, ~20MB, that are downloaded from the MathWorks support files site).

```
url = 'https://www.mathworks.com/supportfiles/predmaint/fault-diagnosis-of-centrifugal-pumps-usi
websave('DynamicOperationData.mat',url);
load DynamicOperationData
figure
plot(t, RefSpeed, t, w)
xlabel('Time (s)')
ylabel('Pump Speed (RPM)')
legend('Reference','Actual')
```
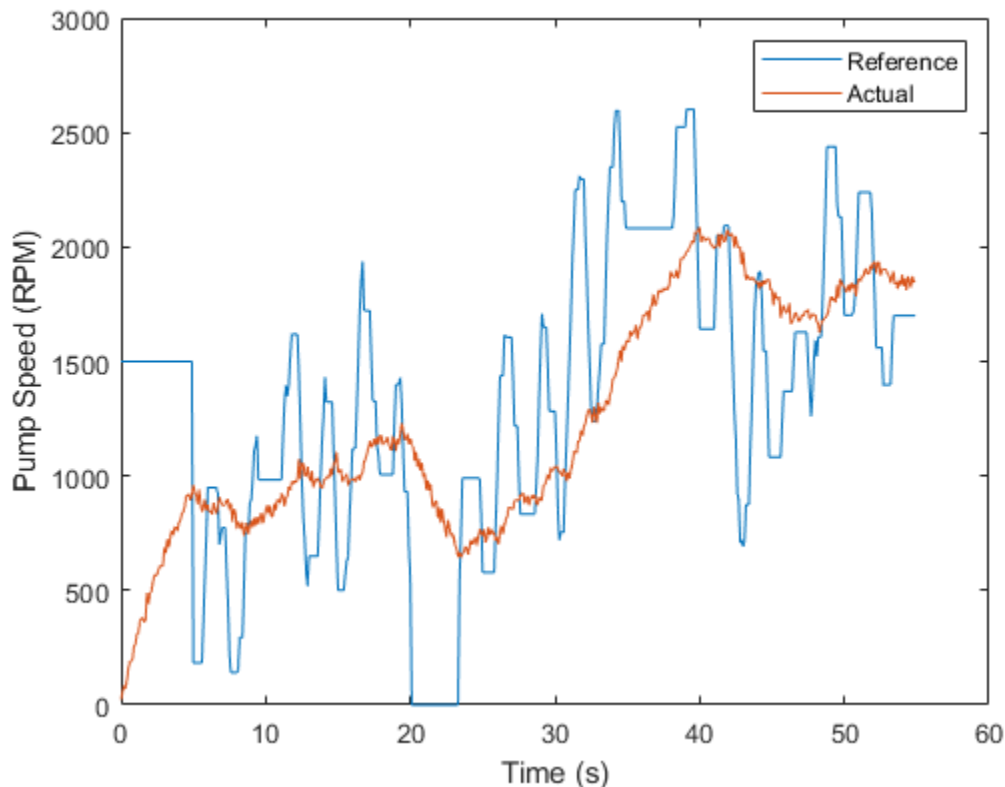

Define operating regimes based on pump speed ranges.

```
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 for regime 1
[th1(2), th2(2), dpest(I2)] = staticPumpEst(w, dp, I2);  % Theta1, Theta2 estimates for regime 2
[th1(3), th2(3), dpest(I3)] = staticPumpEst(w, dp, I3);  % Theta1, Theta2 estimates for regime 3
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')
```
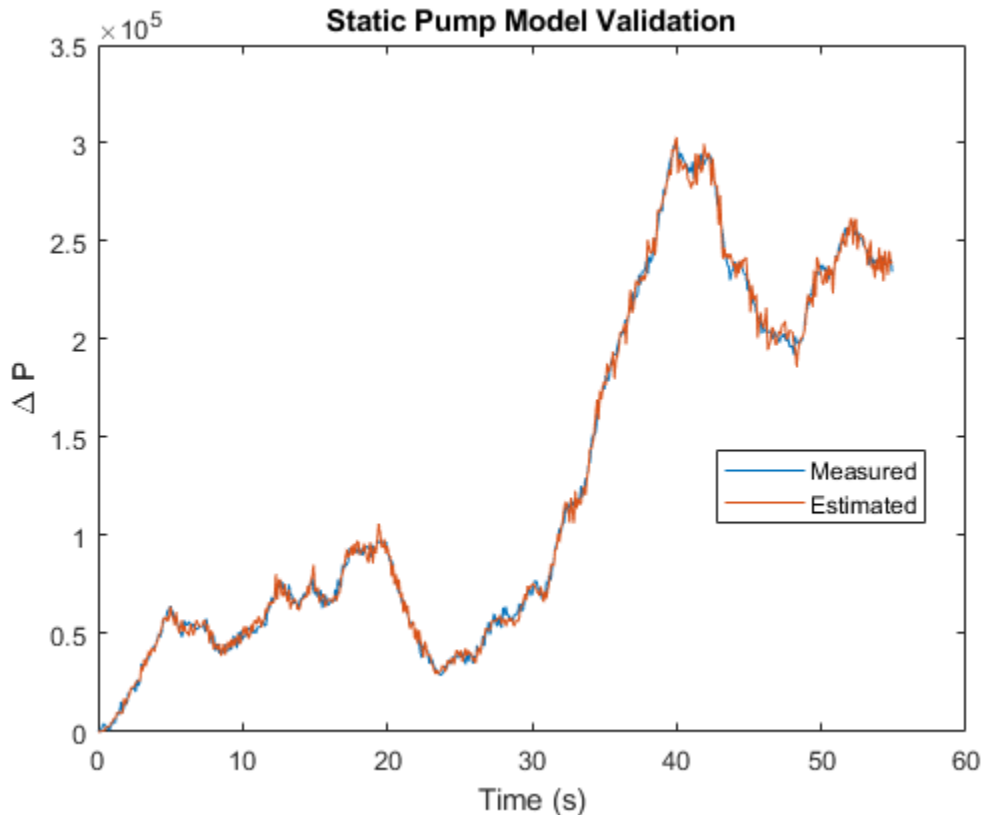

B. Dynamic Pipe Model Identification

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 estimates for reg
```

```
[th3(2), th4(2), th5(2)] = dynamicPipeEst(dp, Q, I2); % Theta3, Theta4, Theta5 estimates for reg:
[th3(3), th4(3), th5(3)] = dynamicPipeEst(dp, Q, I3); % Theta3, Theta4, Theta5 estimates for reg:
```

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.
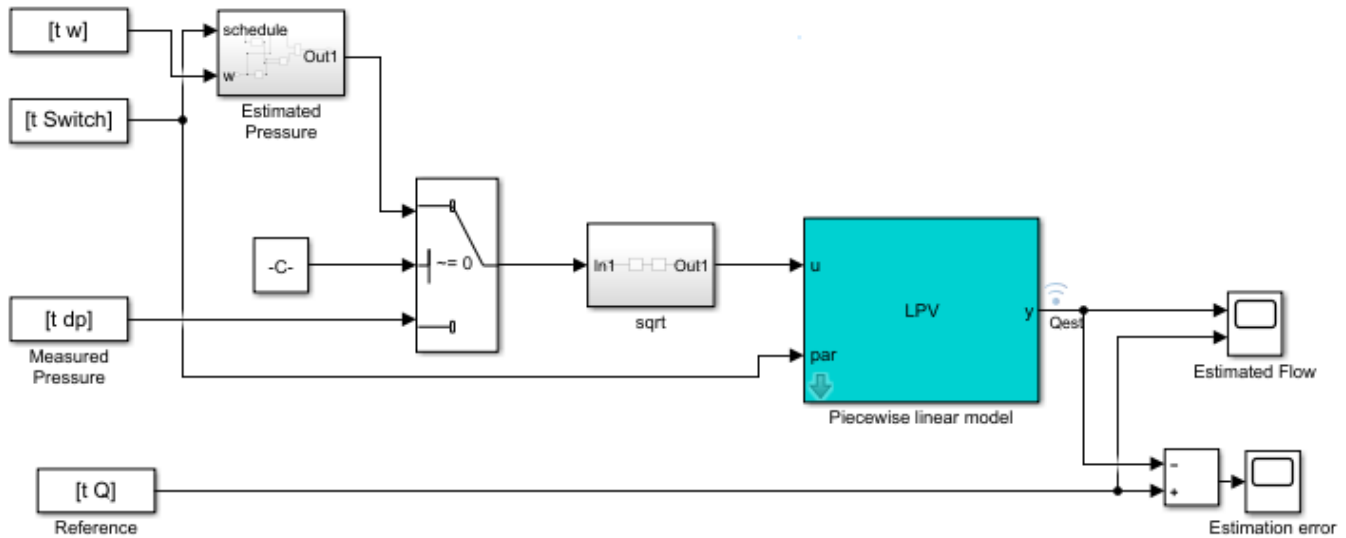

```
% Check Control System Toolbox availability
ControlsToolboxAvailable = ~isempty(ver('control')) && license('test', 'Control_Toolbox');
if ControlsToolboxAvailable
    % Simulate the dynamic pipe model. Use measured value of pressure as input
    Ts = t(2)-t(1);
    Switch = ones(size(w));
    Switch(I2) = 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')
```
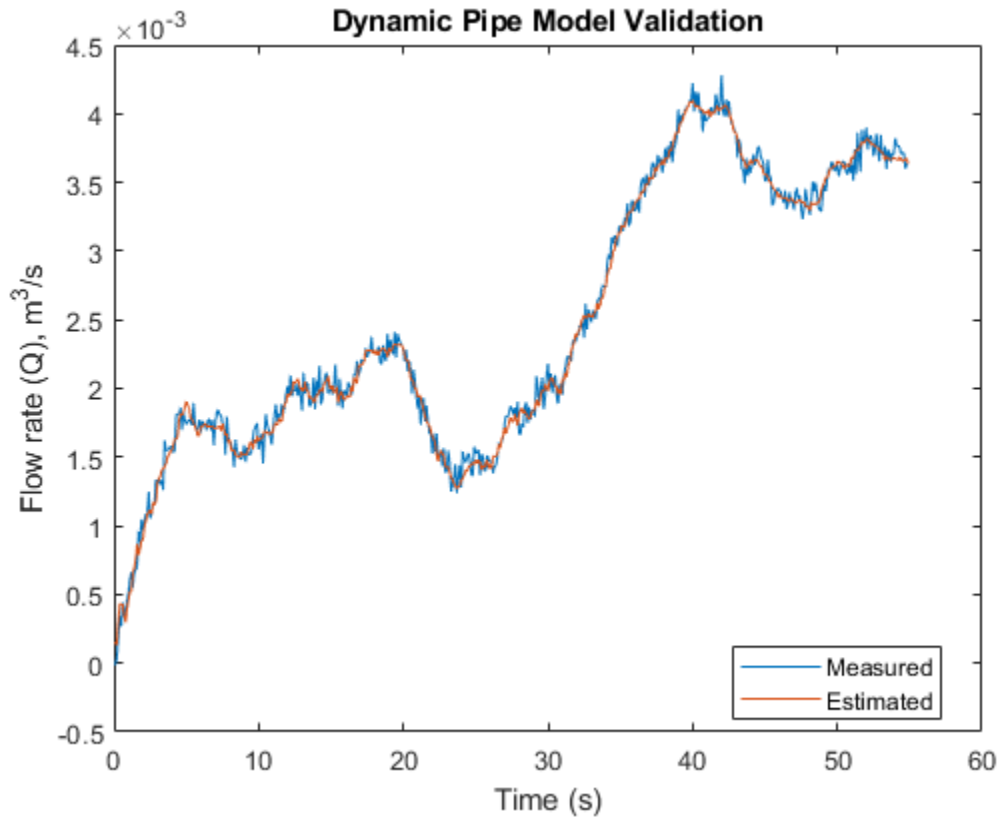
C. Dynamic Pump Pipe Model Identification

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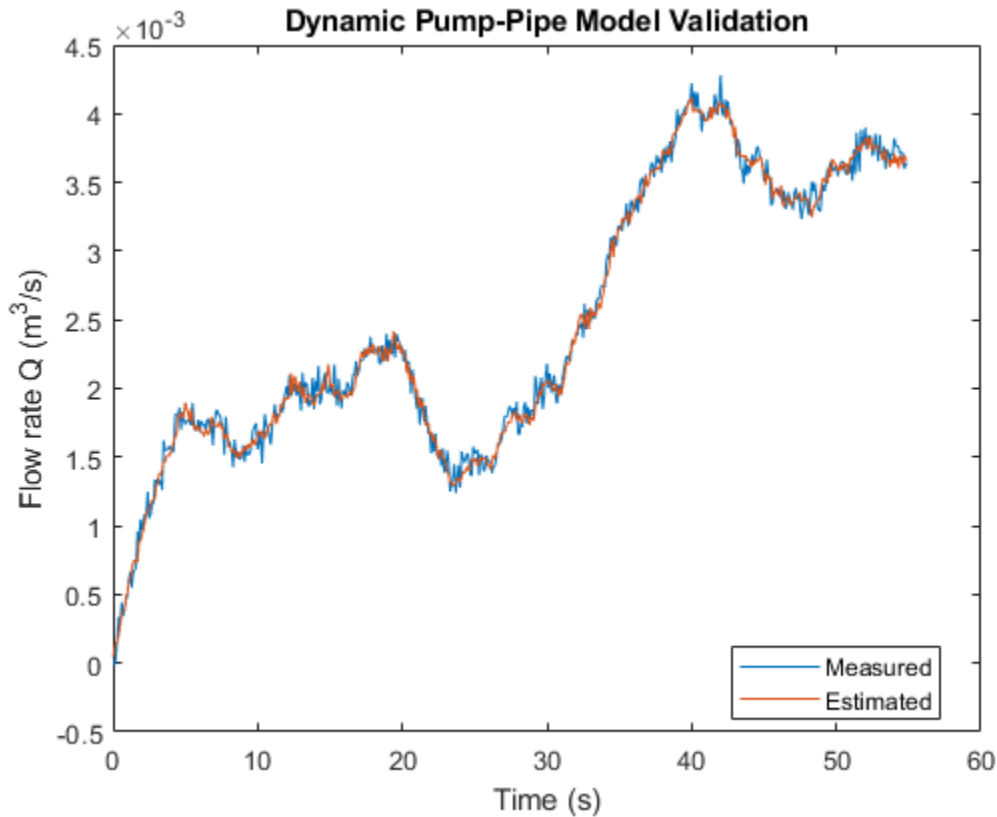
The fit is virtually identical to the one obtained using measured pressure values.

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

Regressors:
  1. Linear regressors in variables y1, u1, u2
  2. Custom regressor: u1(t-2)^2
  3. Custom regressor: u1(t)*u2(t-2)
  4. Custom regressor: u2(t)^2
  List of all regressors

Output function: Linear Function
```

4-59

```
Sample time: 0.1 seconds

Status:
Estimated using NLARX on time domain data.
Fit to estimation data: 49.2% (simulation focus)
FPE: 1798, MSE: 3392
```

```
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')
```
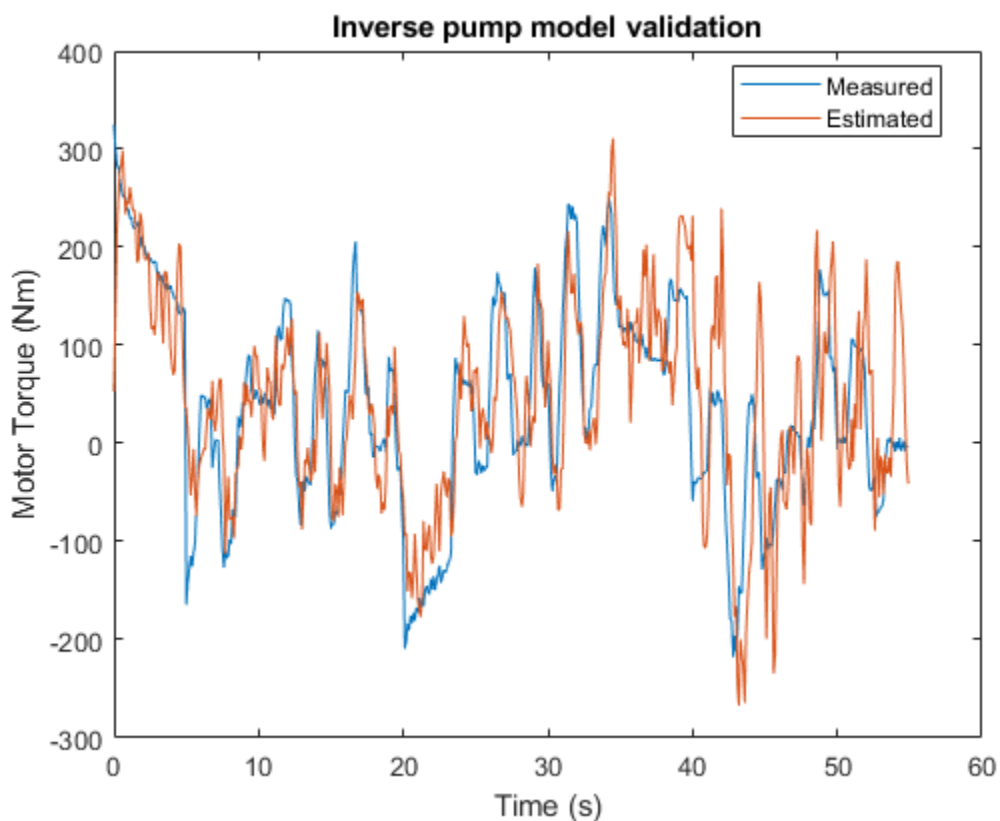

Residue Generation

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)')
```
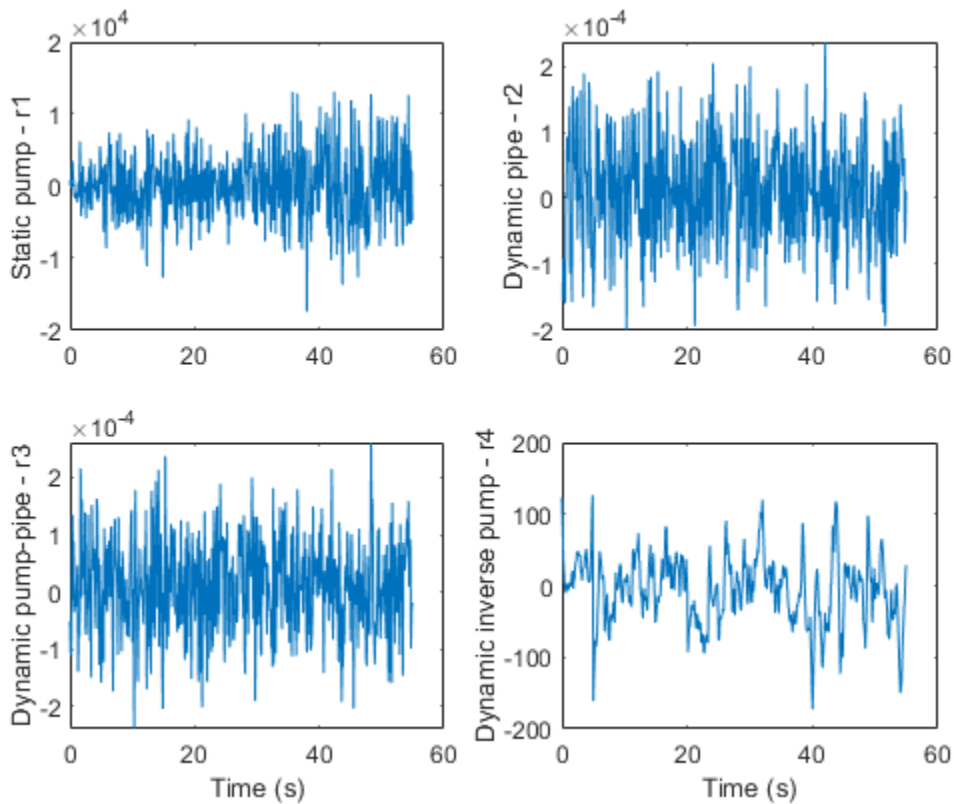

Residue Feature Extraction

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

```
url = 'https://www.mathworks.com/supportfiles/predmaint/fault-diagnosis-of-centrifugal-pumps-usir
websave('MultiSpeedOperationData.mat',url);
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); % Healtl

% Load pre-saved data from "helperComputeEnsembleResidues" run
url = 'https://www.mathworks.com/supportfiles/predmaint/fault-diagnosis-of-centrifugal-pumps-usir
websave('Residuals.mat',url);
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. All the features are scaled using the range of values in the healthy ensemble.

```
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, MinMax] = helperGenerateFeatureTable(HealthyR, CandidateFeatures, FeatureNames)
Fault1Feature  = helperGenerateFeatureTable(Fault1R,  CandidateFeatures, FeatureNames, MinMax);
Fault2Feature  = helperGenerateFeatureTable(Fault2R,  CandidateFeatures, FeatureNames, MinMax);
Fault3Feature  = helperGenerateFeatureTable(Fault3R,  CandidateFeatures, FeatureNames, MinMax);
Fault4Feature  = helperGenerateFeatureTable(Fault4R,  CandidateFeatures, FeatureNames, MinMax);
Fault5Feature  = helperGenerateFeatureTable(Fault5R,  CandidateFeatures, FeatureNames, MinMax);
Fault6Feature  = helperGenerateFeatureTable(Fault6R,  CandidateFeatures, FeatureNames, MinMax);
Fault7Feature  = helperGenerateFeatureTable(Fault7R,  CandidateFeatures, FeatureNames, MinMax);
Fault8Feature  = helperGenerateFeatureTable(Fault8R,  CandidateFeatures, FeatureNames, MinMax);
Fault9Feature  = helperGenerateFeatureTable(Fault9R,  CandidateFeatures, FeatureNames, MinMax);
```

There are 20 features in each feature table (5 features for each residue signal). 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	Max4
0.32049	0.6358	0.6358	0.74287	0.39187	0.53219	0.53219	0.041795
0.21975	0.19422	0.19422	0.83704	0	0.12761	0.12761	0.27644
0.1847	0.25136	0.25136	0.87975	0.32545	0.13929	0.1393	0.084162
1	1	1	0	0.78284	0.94535	0.94535	0.9287
-2.6545	0.90555	0.90555	0.86324	1.3037	0.7492	0.7492	0.97823
-0.89435	0.43384	0.43385	1.0744	0.82197	0.30254	0.30254	-0.022325
-1.2149	0.53579	0.53579	1.0899	0.87439	0.47339	0.47339	0.29547
-1.0949	0.44616	0.44616	1.143	0.56693	0.19719	0.19719	-0.012055
-0.1844	0.16651	0.16651	0.87747	0.25597	0.080265	0.080265	0.49715
-3.0312	0.9786	0.9786	0.75241	1.473	0.97839	0.97839	1.0343

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;
```
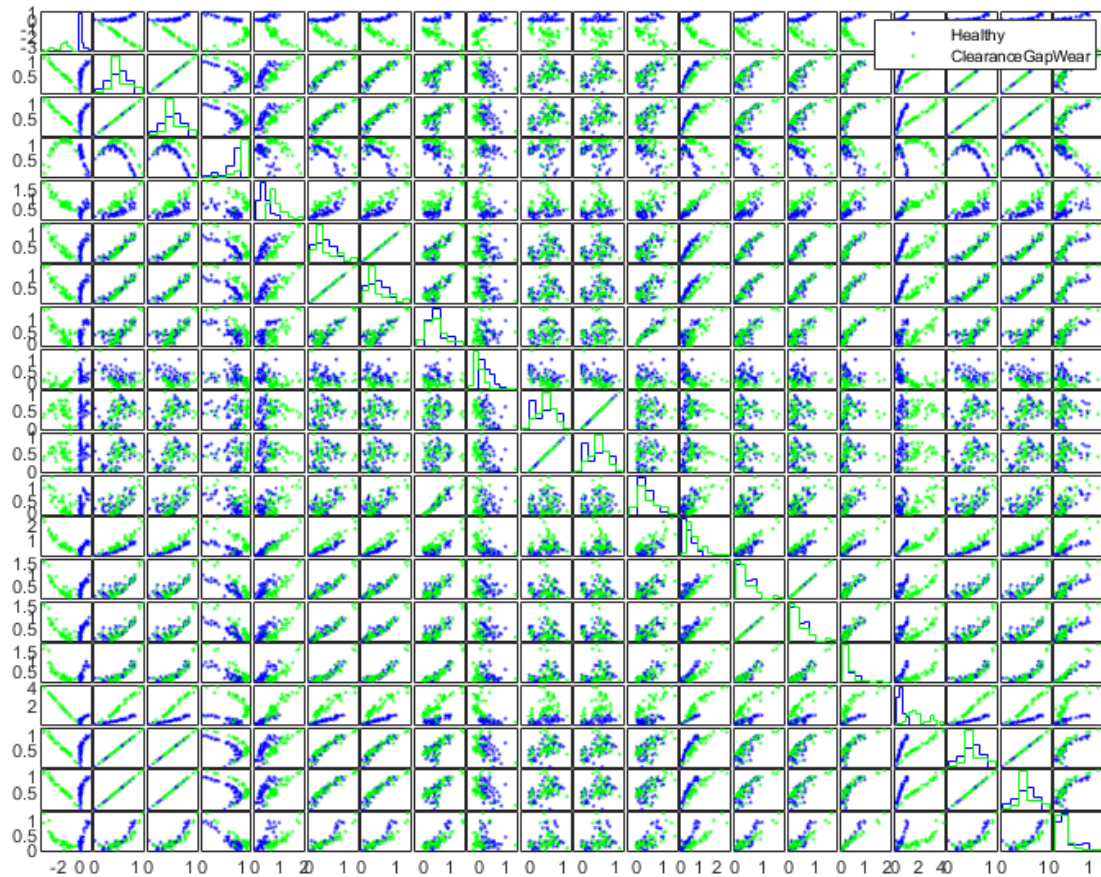
Although not clearly visible, features in columns 1 and 17 provide the most separation. Analyze these features more closely.

```
f = figure;
Names = FeatureTable.Properties.VariableNames;
J = [1 17];
fprintf('Selected features for clearance gap fault: %s\n',strjoin(Names(J),', '))
```

Selected features for clearance gap fault: Mean1, OneNorm1

```
gplotmatrix(P(I,[1 17]),[],R(I))
```
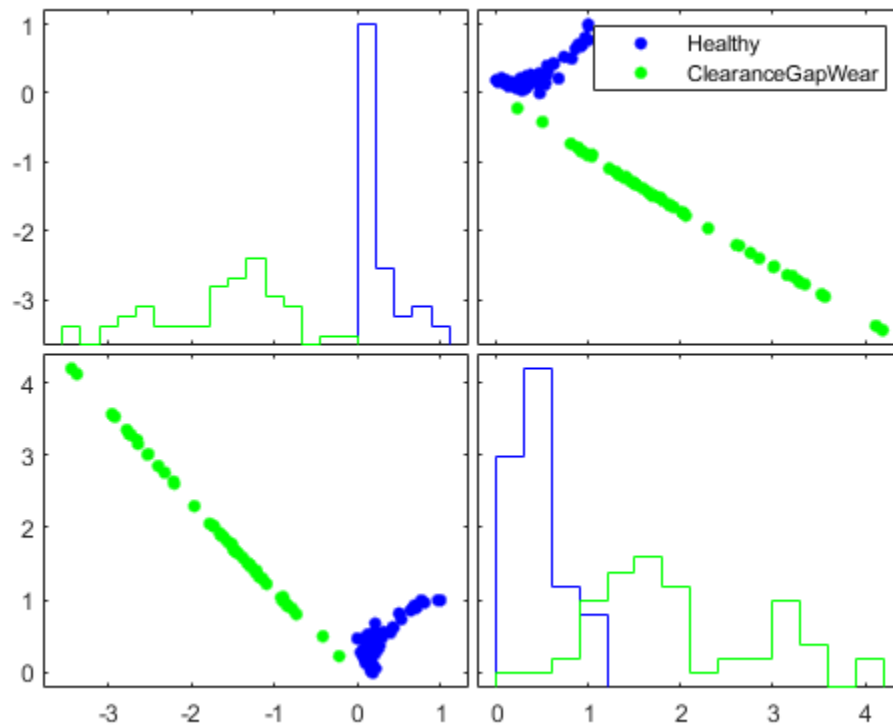
The plot now clearly shows that features Mean1 and OneNorm1 can be used to separate healthy mode from clearance gap fault mode. A similar analysis can be performed for each fault mode. In all cases, it is possible to find a set of features that distinguish the fault modes. Hence *detection* of a faulty behavior is always possible. However, fault *isolation* is more difficult since the same features are affected by multiple fault types. For example, the features Mean1 (Mean of r1) and OneNorm1 (1-norm of r1) 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') | strcmp(R
J = [1 4 6 16 20]; % selected feature indices
fprintf('Selected features for sensors'' bias: %s\n',strjoin(Names(J),', '))
```

```
Selected features for sensors' bias: Mean1, Mean4, Max2, Variance4, OneNorm4
```

```
gplotmatrix(P(I,J),[],R(I))
```
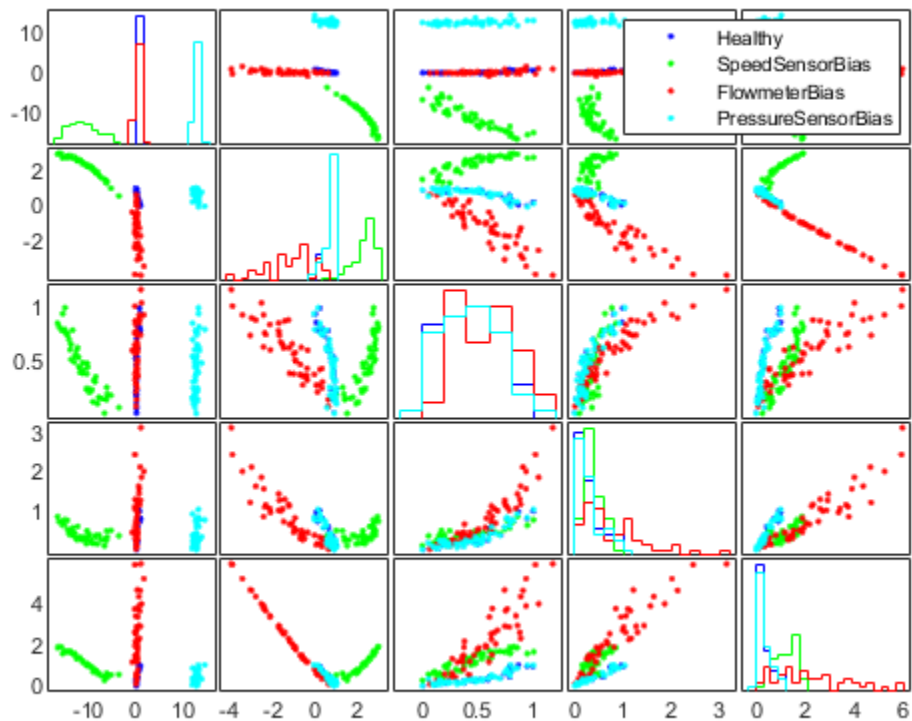
The scatter plot of selected features shows that the 4 modes can be distinguished on one or more pairs of features. Train a 20 member Tree Bagger classifier for the reduced set of faults (sensor biases only) using a reduced set of features.

```
rng default % for reproducibility
Mdl = TreeBagger(20, FeatureTable(I,[J 21]), 'Condition',...
    'OOBPrediction','on',...
    'OOBPredictorImportance','on');
figure
plot(oobError(Mdl))
xlabel('Number of trees')
ylabel('Misclassification probability')
```
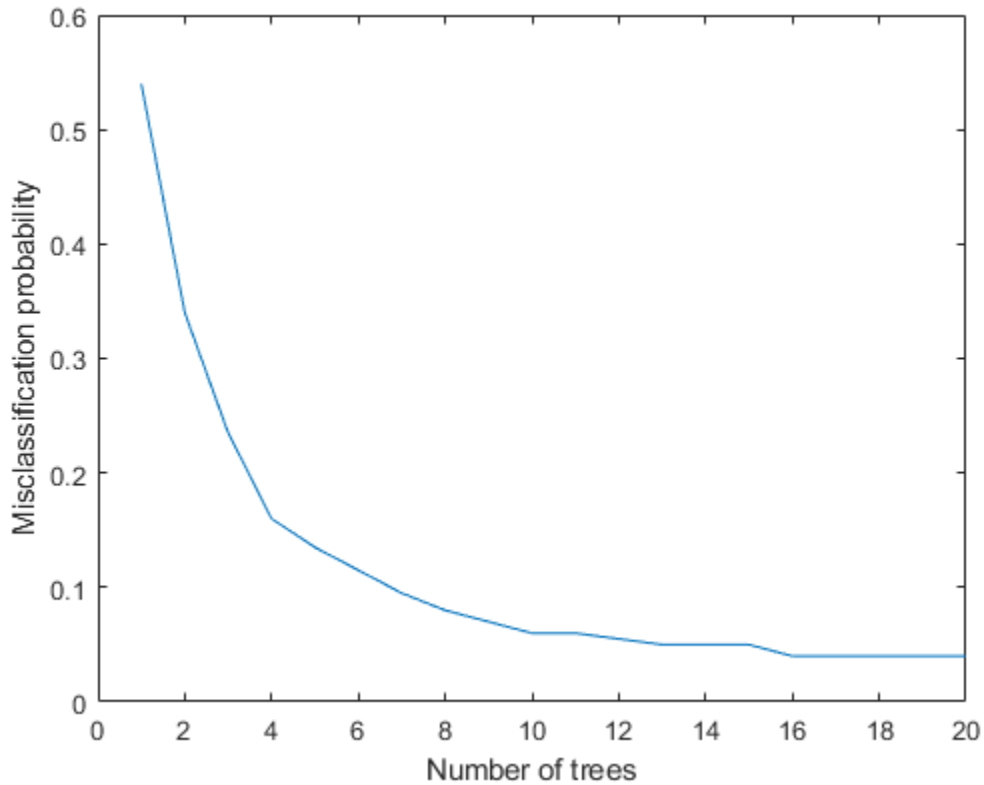
The misclassification error is less than 3%. Thus it is possible to pick and work with a smaller set of features for classifying a certain subset of faults.

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.
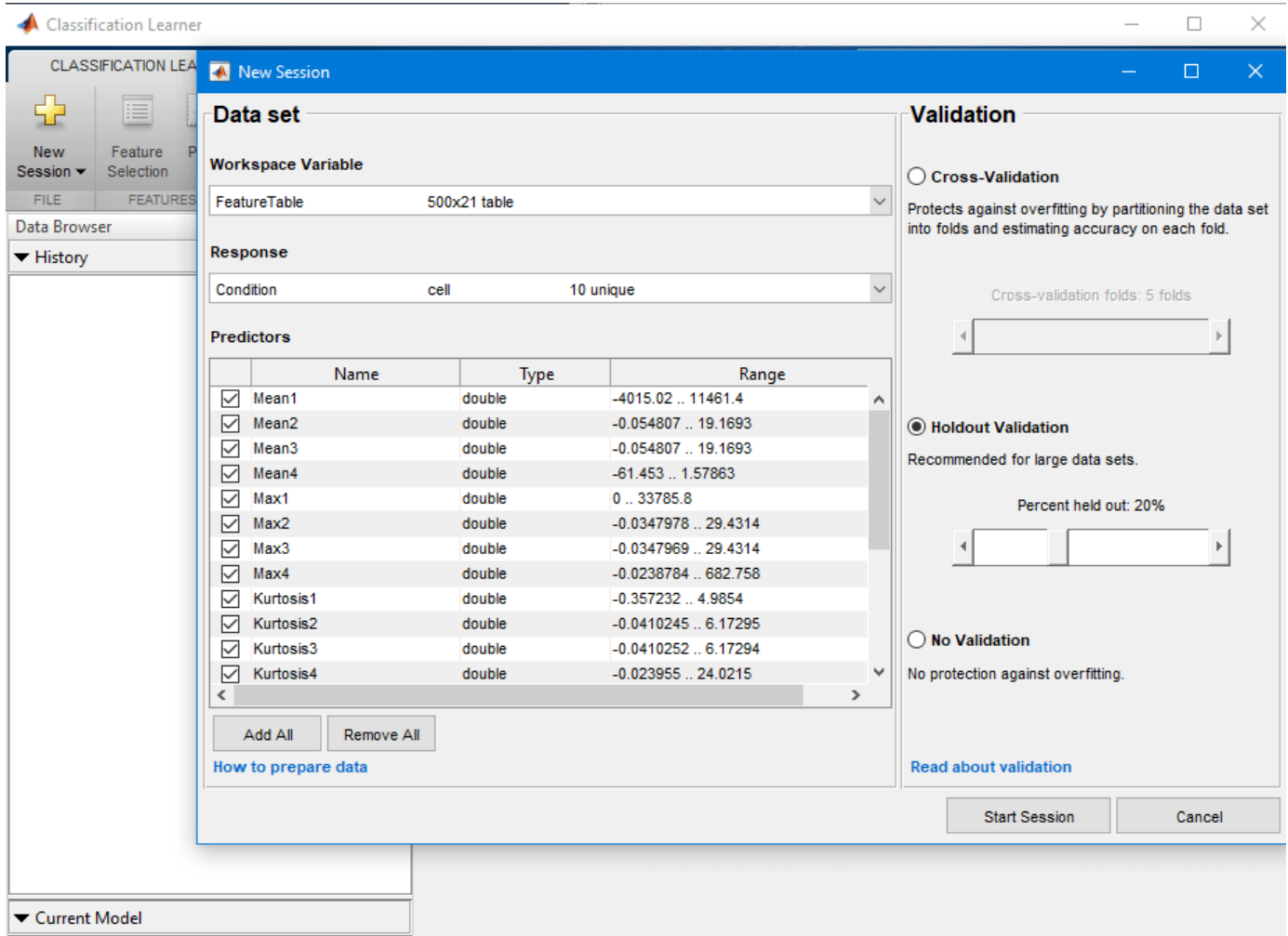
2. Select **All** under *Model Type* section of the main tab. Then press the **Train** button.
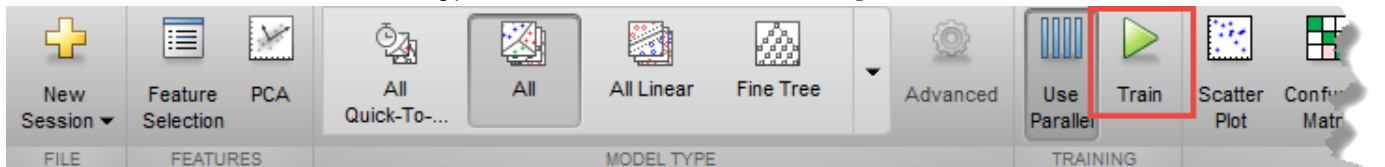
3. In a short time, about 20 classifiers are trained. Their accuracies are displayed next to their names under the history panel. A linear SVM classifier performs the best, producing 86% accuracy on the hold out samples. This classifier has some difficulty in identifying "ClearanceGapWear" which it classifies as "ImpellerOutletDeposit" 40% of the time.

4. To get a graphical view of the performance, open a Confusion Matrix plot from the PLOTS section of the main tab. The plot shows the performance of the selected classifier (the Linear SVM

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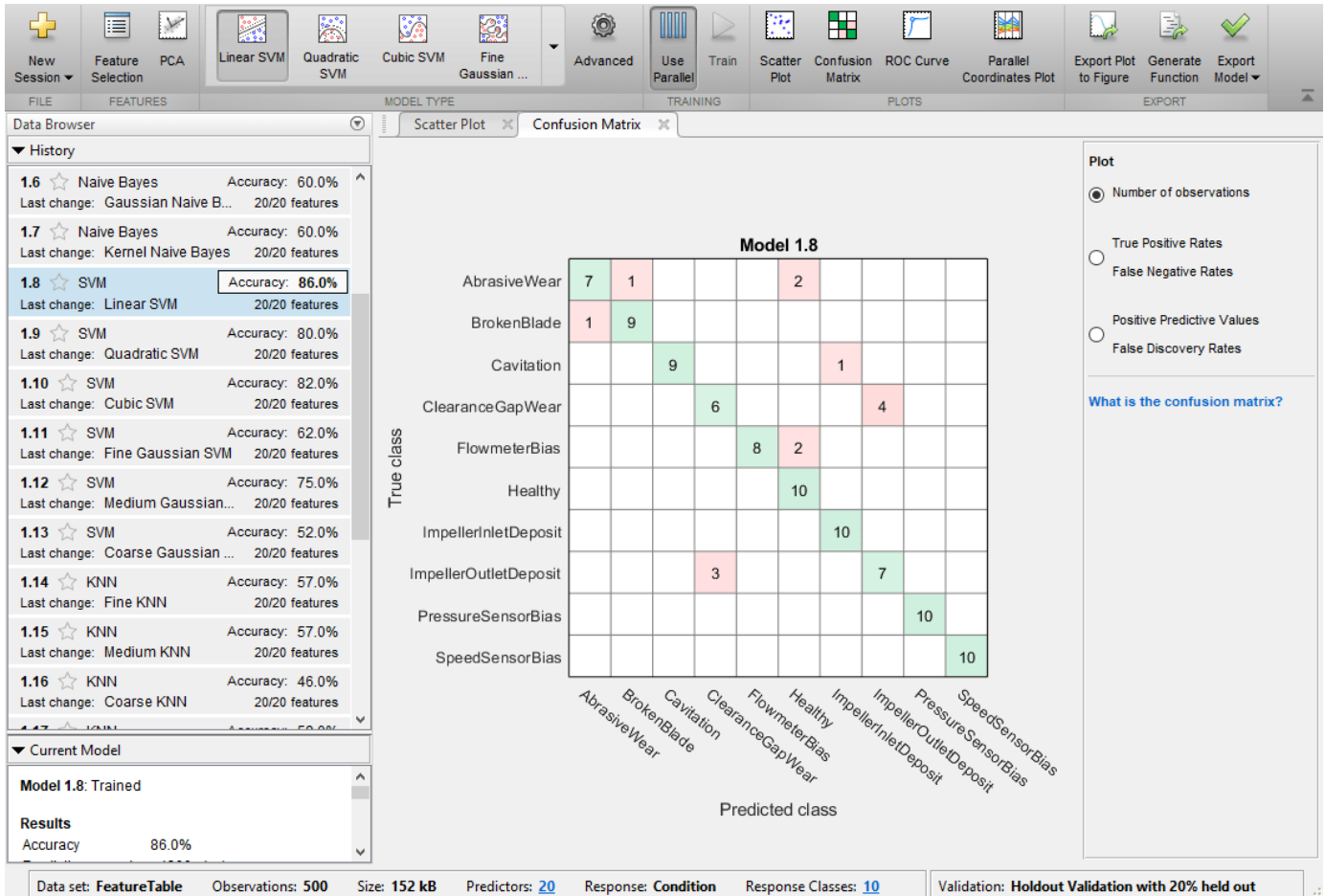
Export the best performing classifier to the workspace and use it for prediction on new measurements.

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.

```matlab
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.

```matlab
function syse = identifyNonlinearARXModel(Mmot,w,Q,Ts,N)
%identifyNonlinearARXModel Identify a nonlinear ARX model for 2-input (w, Q), 1-output (Mmot) da
% 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 $\dot{x} = \dfrac{x_{n+1} - x_n}{T_s}$, 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
```

```
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
```

type pdmMotorModelMeasurementFcn

```
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
```

```
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
```

```
type pdmMotorModelMeasJacobianFcn
```

```
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 measurement
    xSigEst(:,ct) = x_corr;
    xstd(:,:,ct) = chol(ekf.StateCovariance);
    predict(ekf,u(ct),J,Ts);              % Predict next state given the current state and input.
```

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)-3*fSTD(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','SouthWest')
subplot(212), plot(t,u);
title('Motor Input - Torque')
```

Motor Outputs

Motor Input - Torque
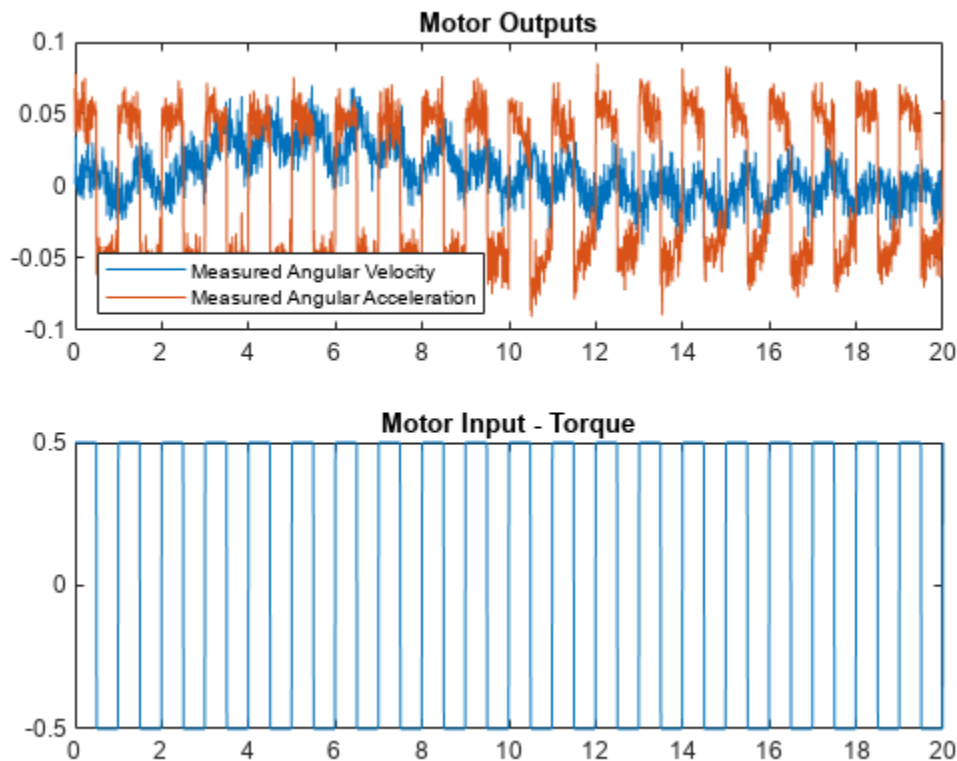
The model input-output responses indicate that it is difficult to detect the friction change directly from the measured signals. The extended Kalman filter enables us to estimate the states, in particular the friction state. Compare the true model states and estimated states. The estimated states are 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(1,1,:))'])
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');
```
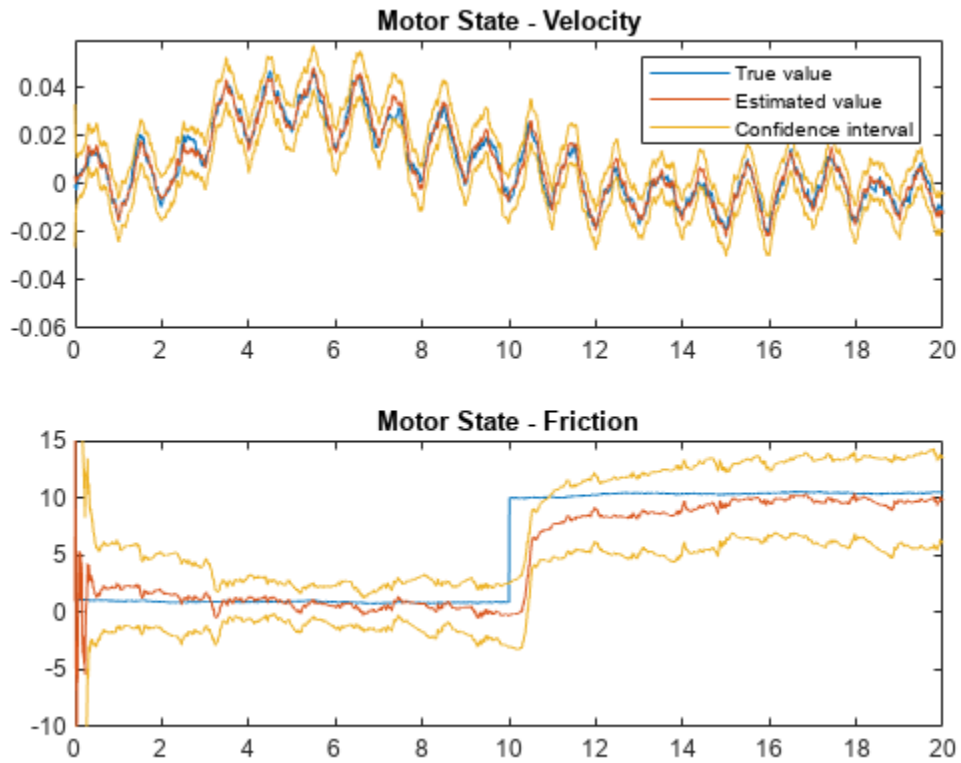
Note that the filter estimate tracks the true values, and that the confidence intervals remain bounded. Examining the estimation errors provide more insight into the filter behavior.

```
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')
```
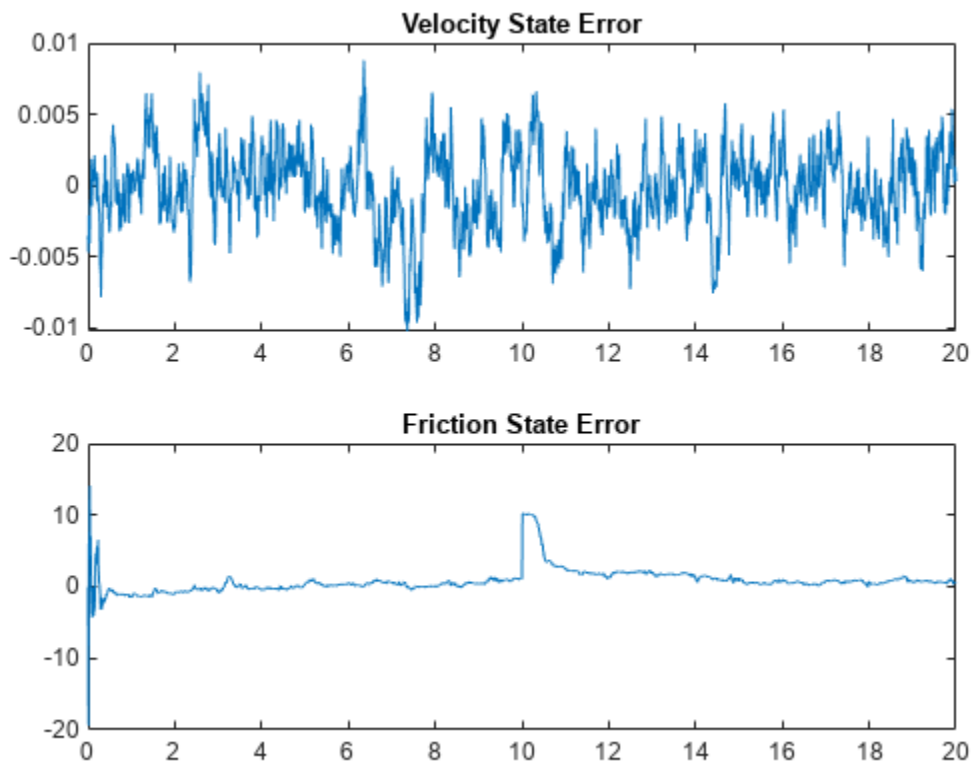
The error plots show that the filter adapts after the friction change at 10 seconds and reduces the estimation errors to zero. However, the error plots cannot be used for fault detection as they rely on knowing the true states. Comparing the measured state value to the estimated state values for acceleration 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')
```
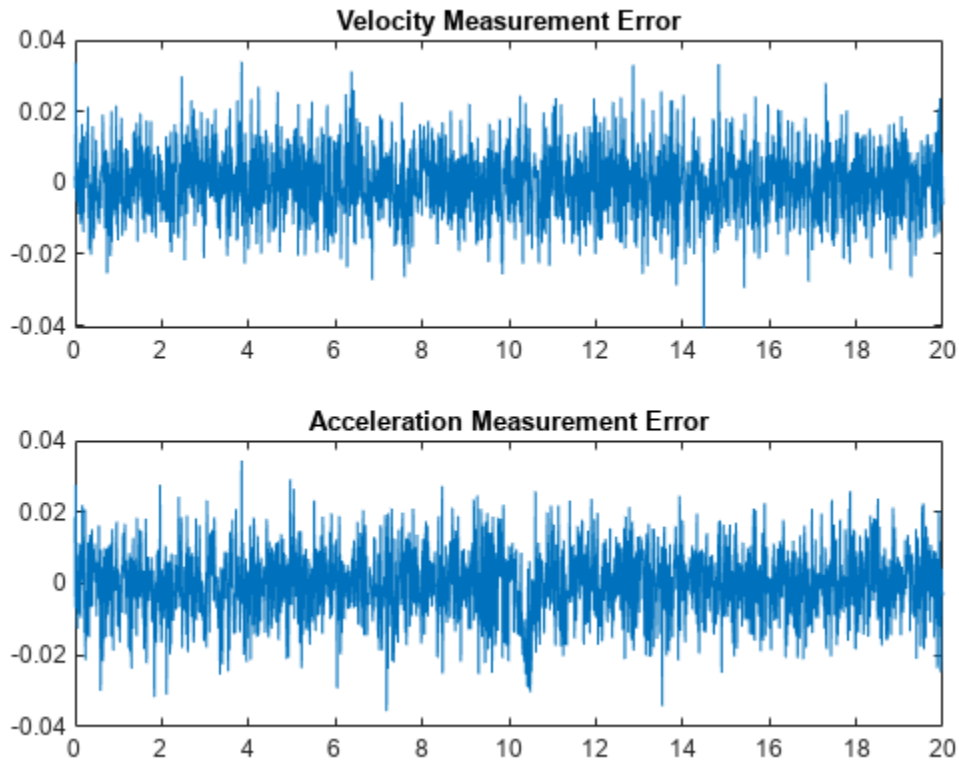
The acceleration error plot shows a minor difference in mean error around 10 seconds when the fault is introduced. View the error statistics to see if the fault can be detected from the computed errors. The acceleration and velocity errors are expected to be normally distributed (the noise models are 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')
```

Velocity Error Kurtosis

Acceleration Error Kurtosis
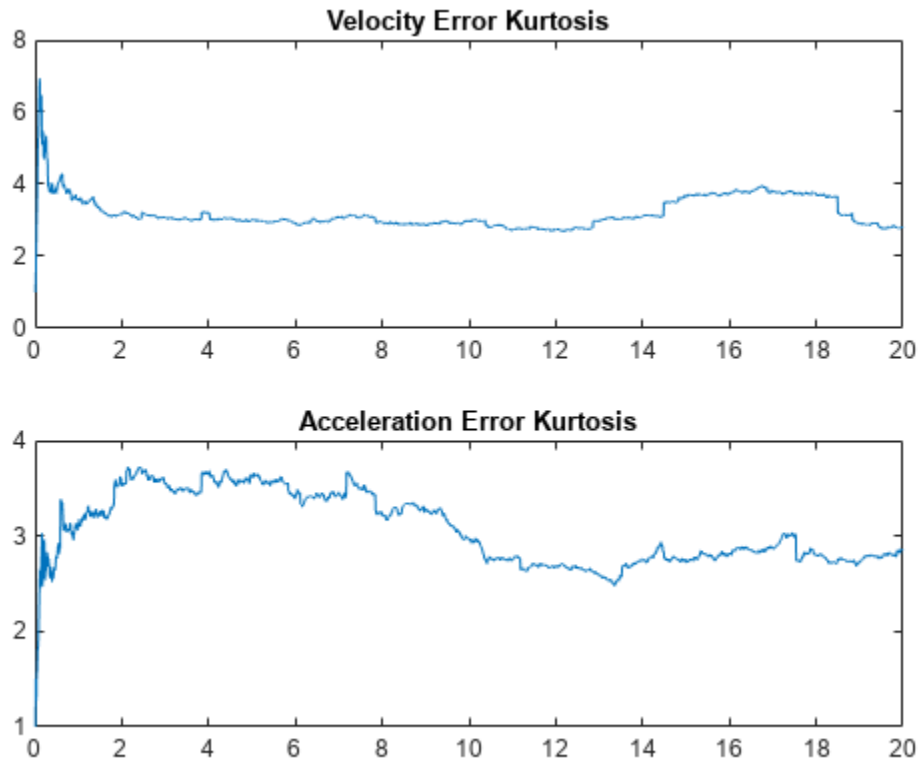
Ignoring the first 4 seconds when the estimator is still converging and data is being collected, the kurtosis of the errors is relatively constant with minor variations around 3 (the expected kurtosis value for a Gaussian distribution). Thus, the error statistics cannot be used to automatically detect 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
```
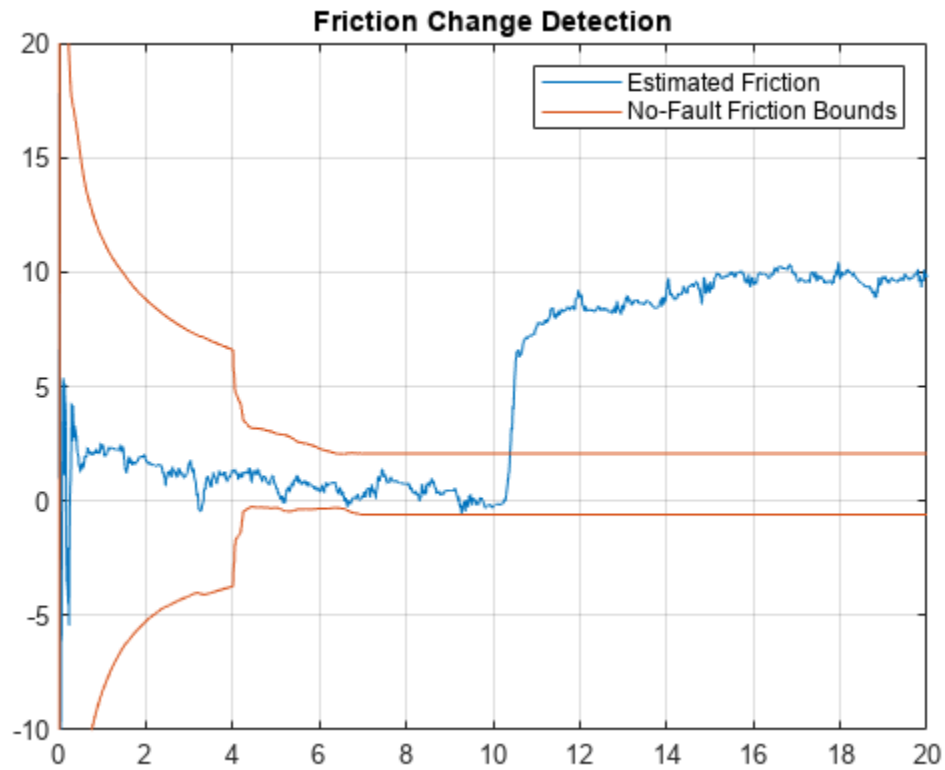
Summary

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 3-7

Fault Detection Using Data Based Models

This example shows how to use a data-based modeling approach for fault detection.

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

Normal Operation

Pulse

Mechanical System
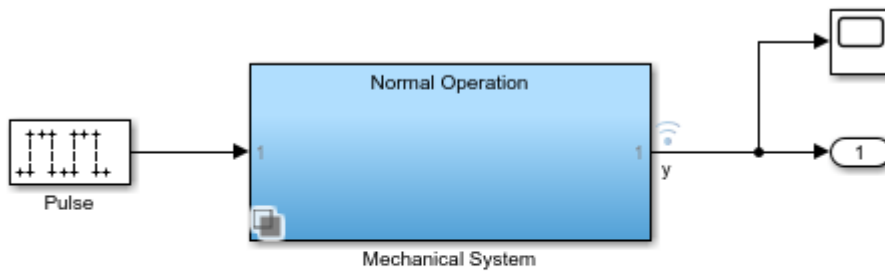
Copyright 2015-2018 The MathWorks, Inc.
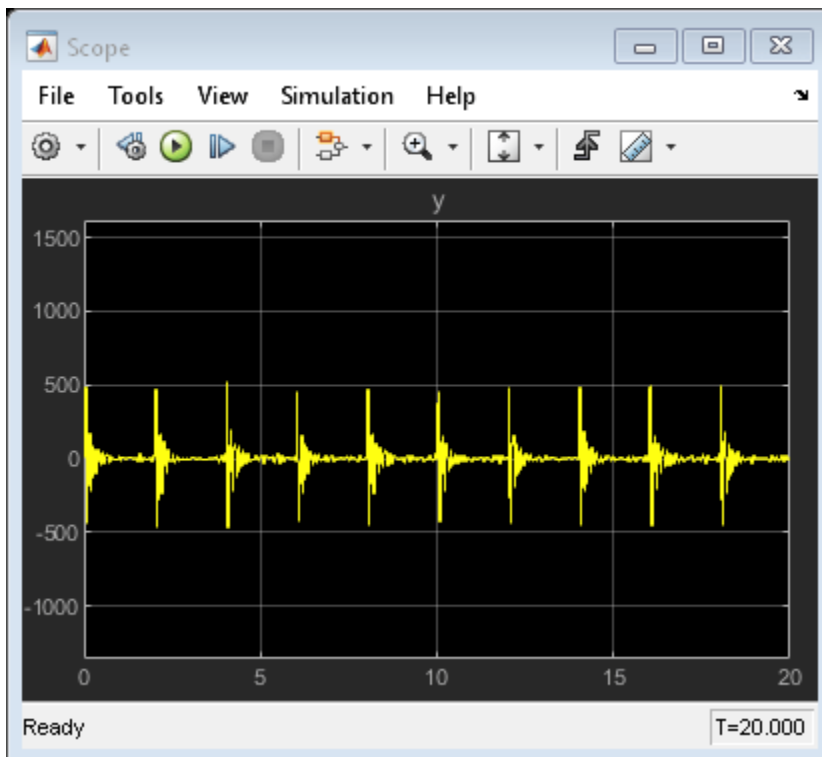
The input signal was not measured; all we have recorded is the response `ynormal`. Hence we build a dynamic model of the system using "blind identification" techniques. In particular, we build an ARMA model of the recorded signal as a representation of the system. This approach works when the input signal 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 Segments')
```

Measured Response Segments

y1
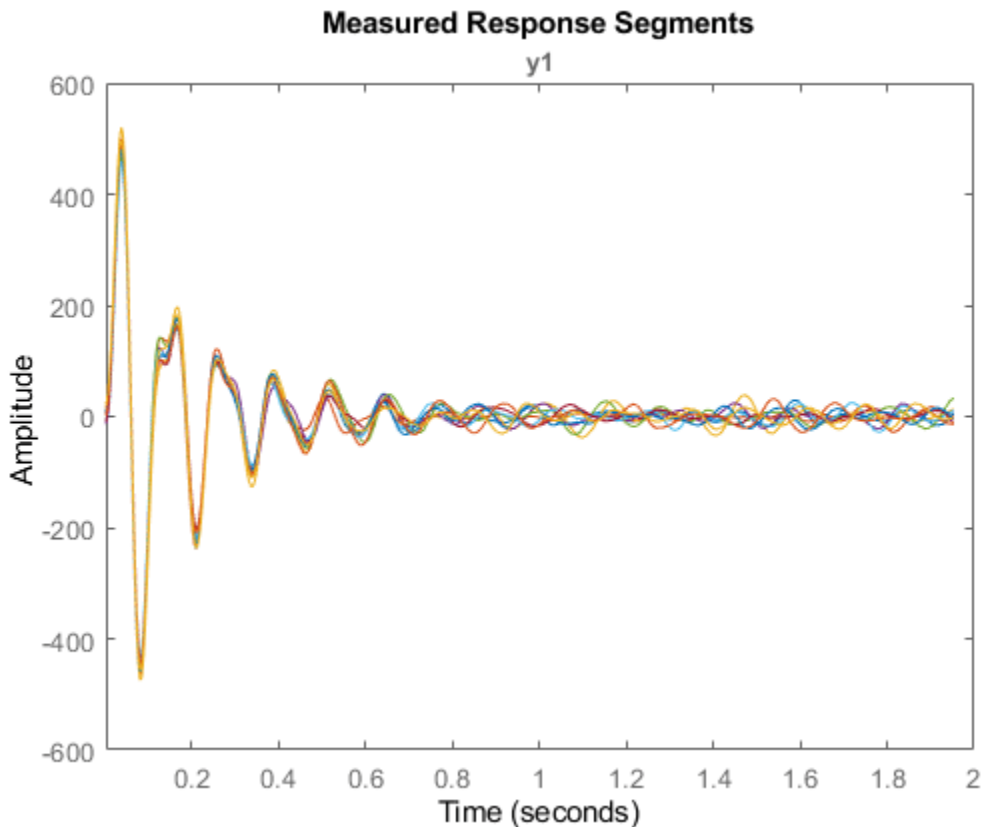
Split the data into estimation and validation pieces.

```
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: 0.0039062 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)
```
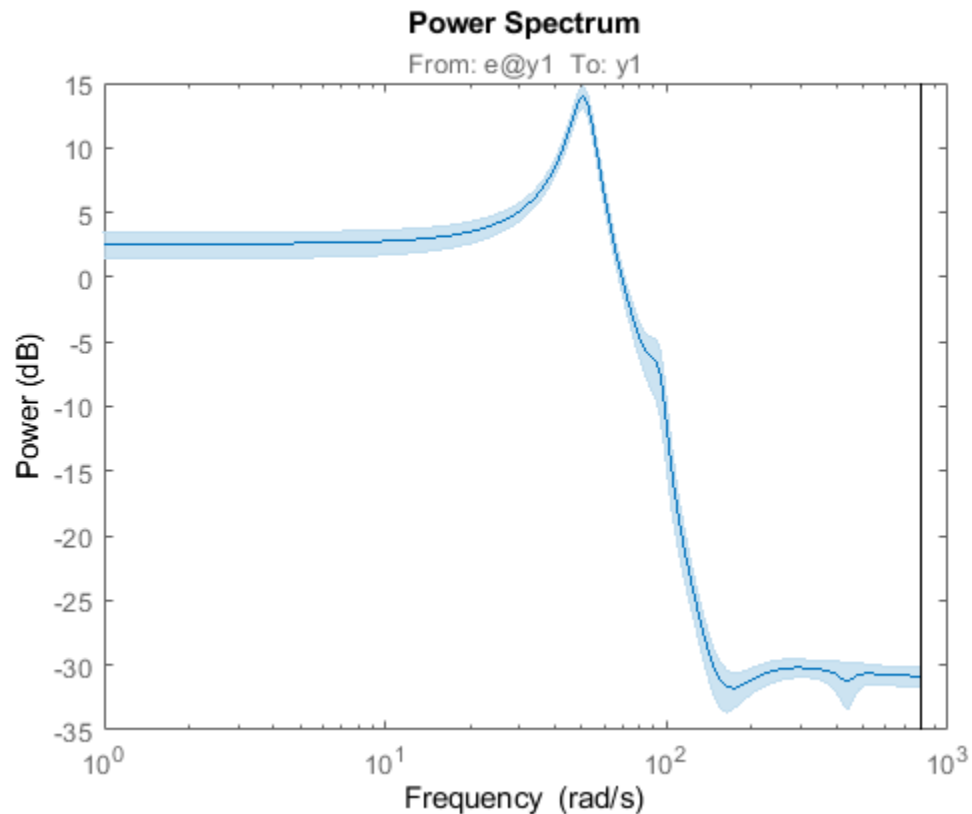

The confidence region is small, although there is about 30% uncertainty in the response at lower frequencies. The next step in validation is to see how well the model predicts the responses in the validation dataset `zv`. We use a 25-step ahead prediction horizon.

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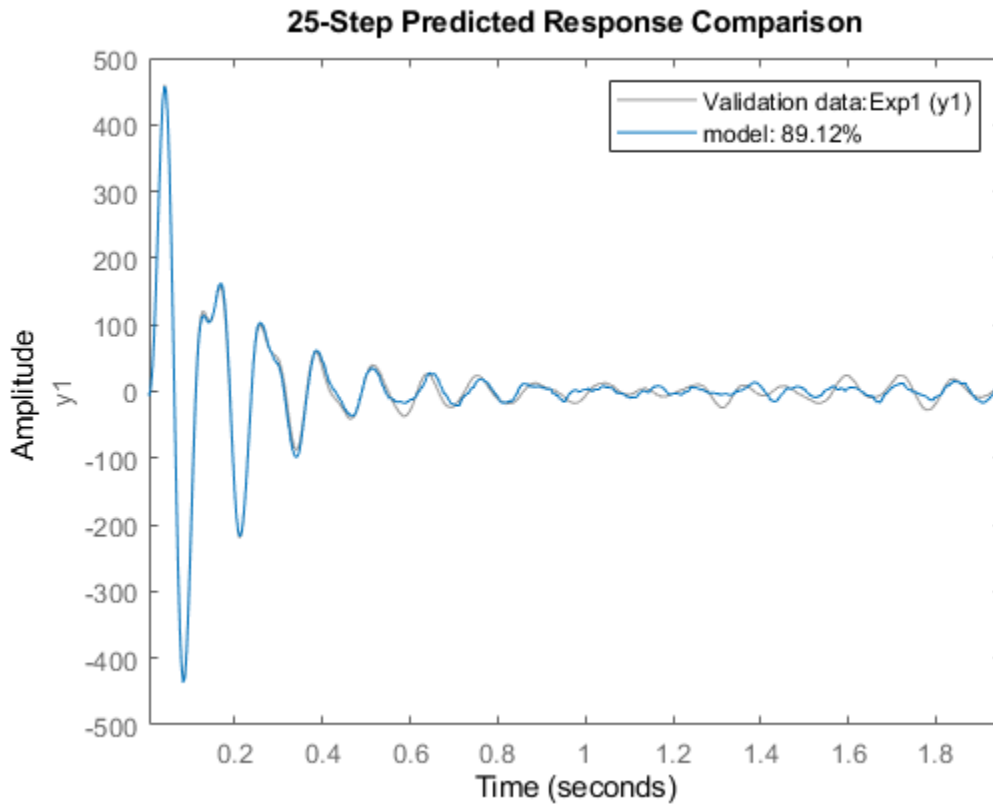
The plot shows that the model is able to predict the response in the first experiment of the validation dataset 25 time steps (= 0.1 sec) in future with > 85% accuracy. To view the fit to other experiments in the dataset, use the right-click context menu of the plot axes.

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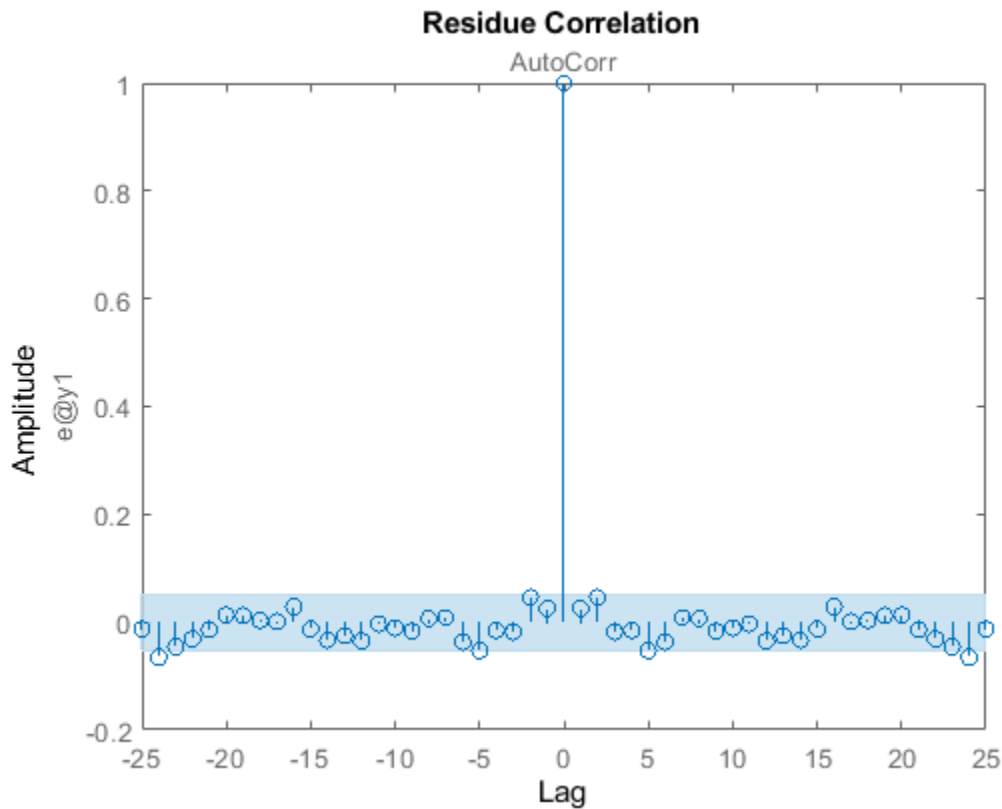
The residuals are mostly uncorrelated at nonzero lags. Having derived a model of the normal behavior we move on to investigate how the model can be used to detect faults.

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'],'LabelModeActiveChoice','DamagedSystem');
set_param([sysA,'/Pulse'],'Period','5120') % to force only one bump
sim(sysA)
y = logsout.getElement('y').Values;
```
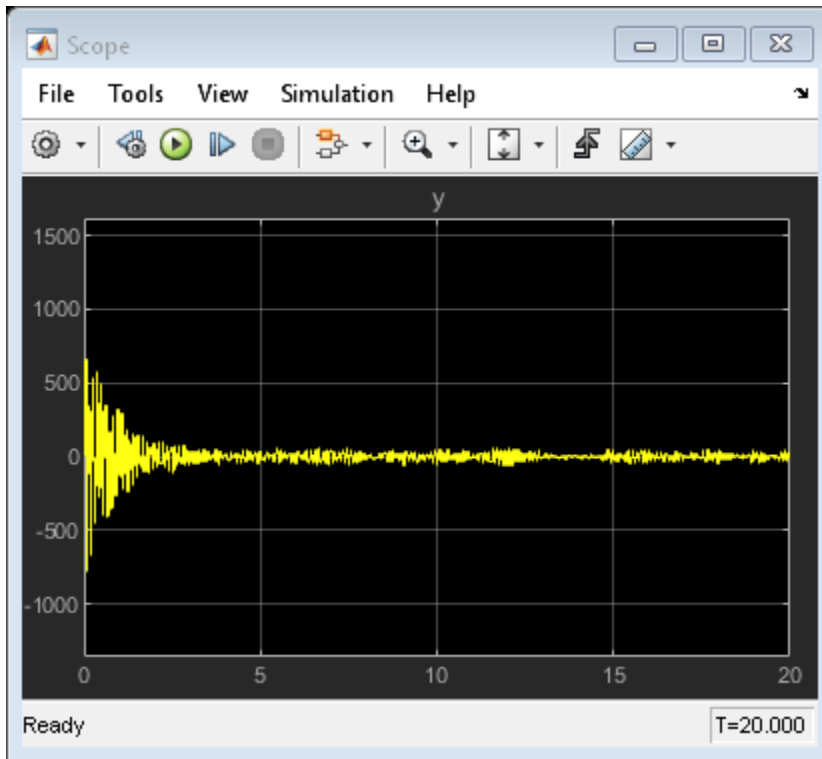
```
resid(model, y.Data)
set_param([sysA,'/Pulse'],'Period','512') % restore original
```
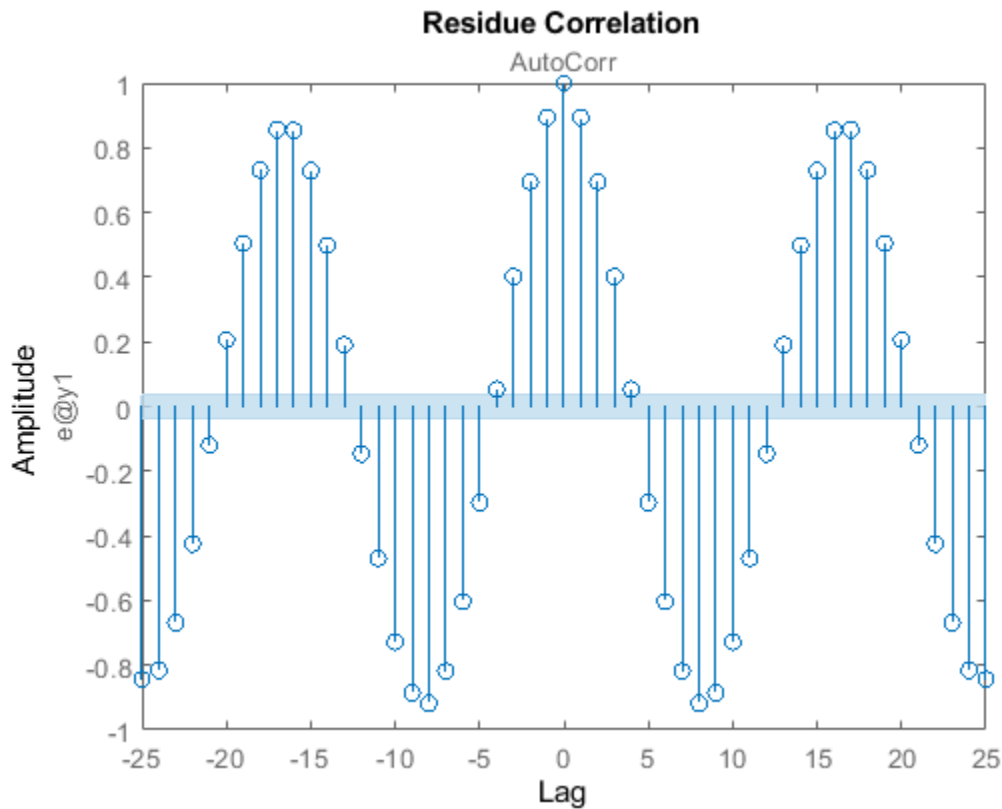
The residuals are now larger and show correlations at non-zero lags. This is the basic idea behind detection of faults - creating a residual metric and observing how it changes with each new set of measurements. What is used here is a simple residual based on 1-step prediction error. In practice, more 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 `pdmMechanicalSystem` 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'],'LabelModeActiveChoice','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
```
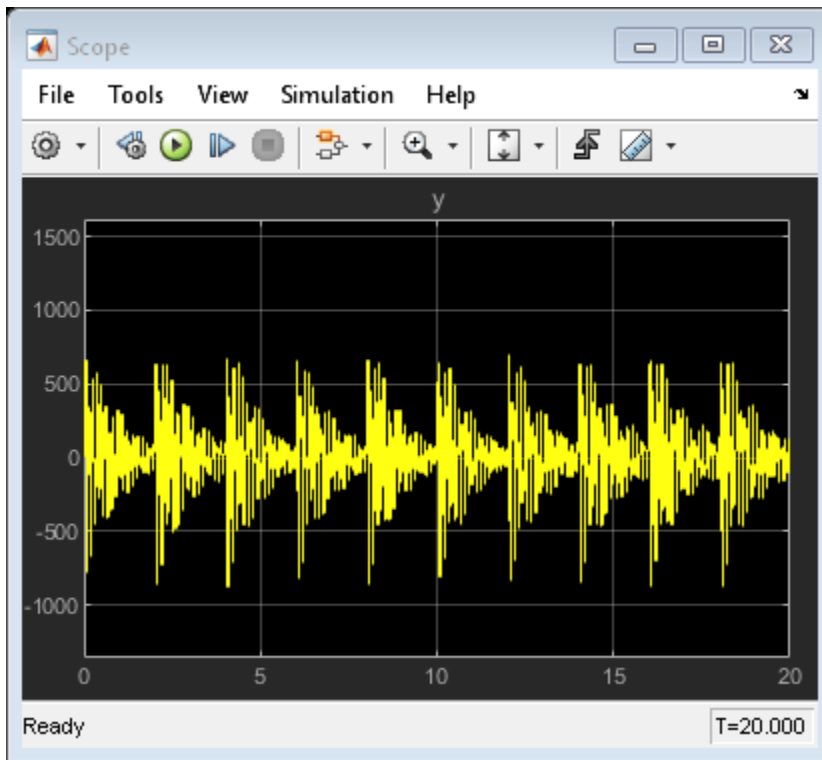

We now create a set of models, one for each data segment. As before we build 7th order time series models in state-space form. Turn off covariance computation for speed.

```
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)
```

25-Step Predicted Response Comparison
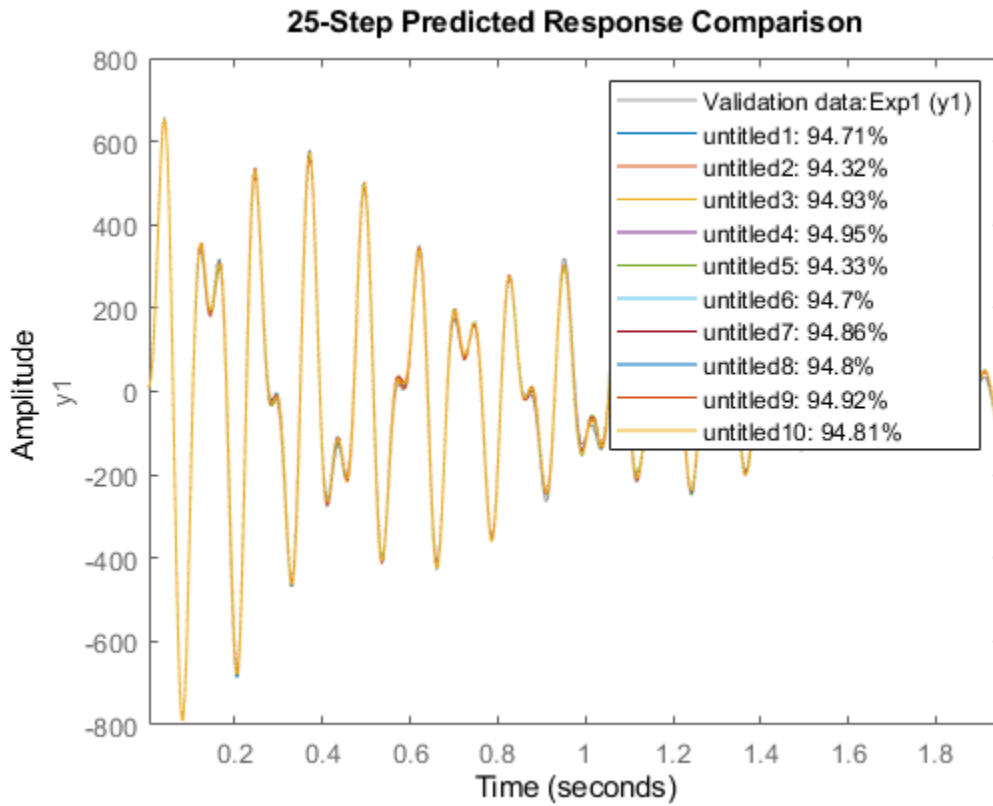
Normal and faulty estimated spectra are plotted below.

```
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)')
```

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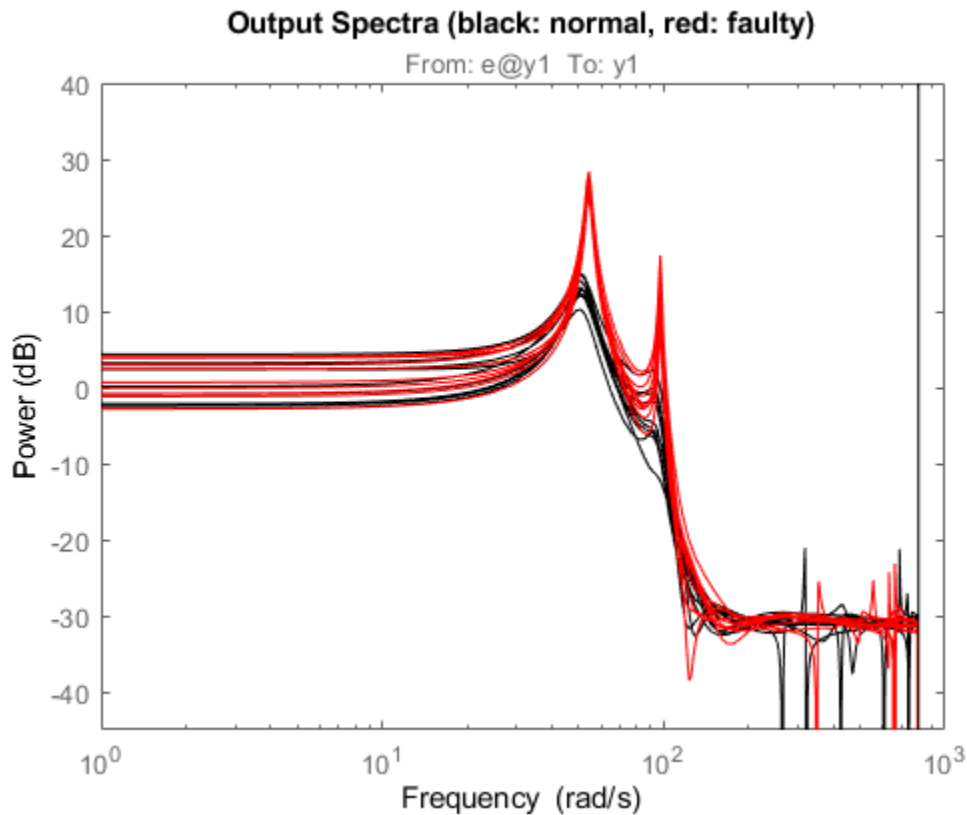
The spectrum plot shows the difference: the damaged mode has its primary resonances amplified but the spectra are otherwise overlapping. Next, we create a way to quantitatively distinguish between the normal and the faulty state. We can use data clustering and classification approaches such as:

- 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 resonances
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'],'LabelModeActiveChoice','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')
```
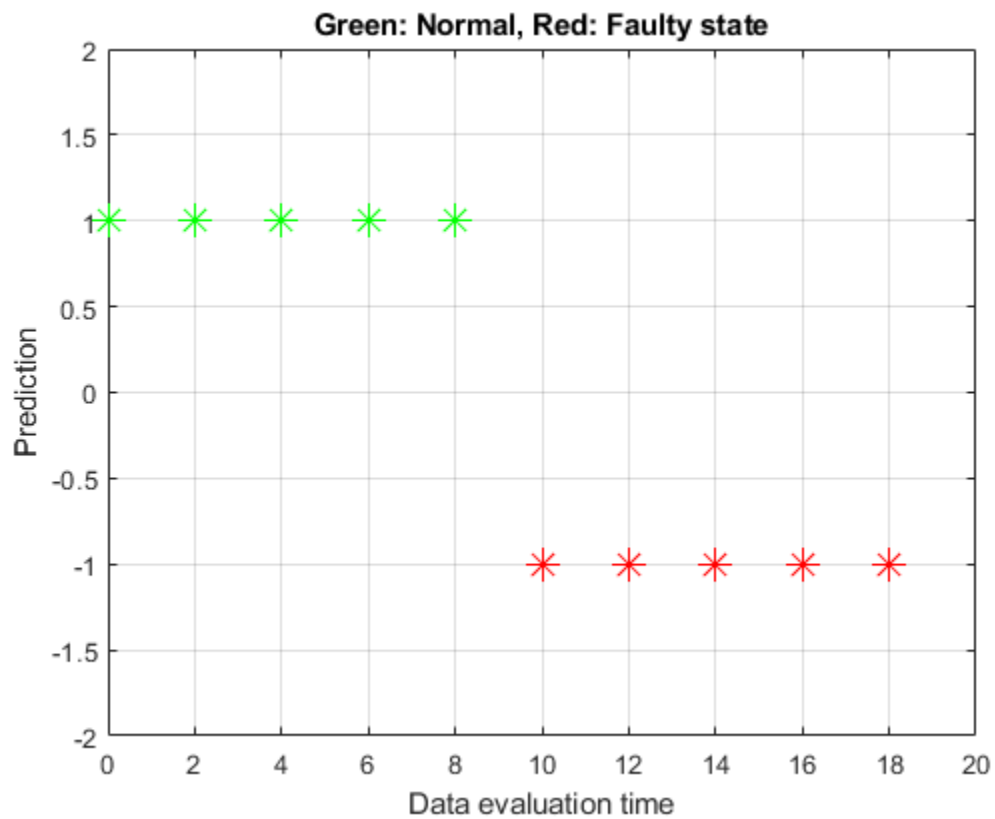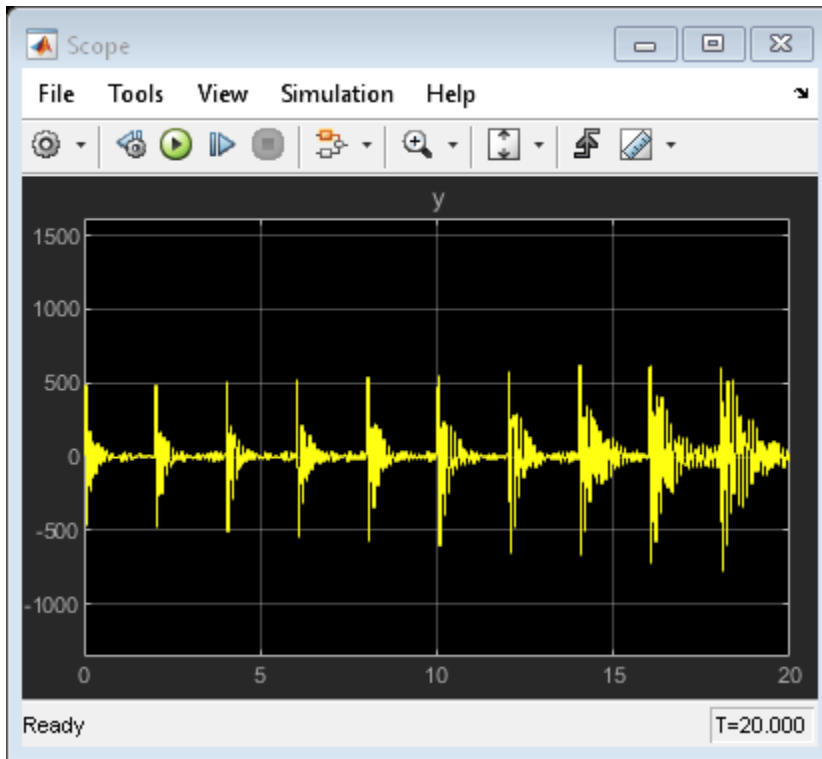
The plot shows that the classifier predicts the behavior up to about the mid-point to be normal and in a state of fault thereafter.

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 `pdmDeterioratingSystemEstimation` which is same as the 'DeterioratingSystem' mode of `pdmMechanicalSystem` 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);
```
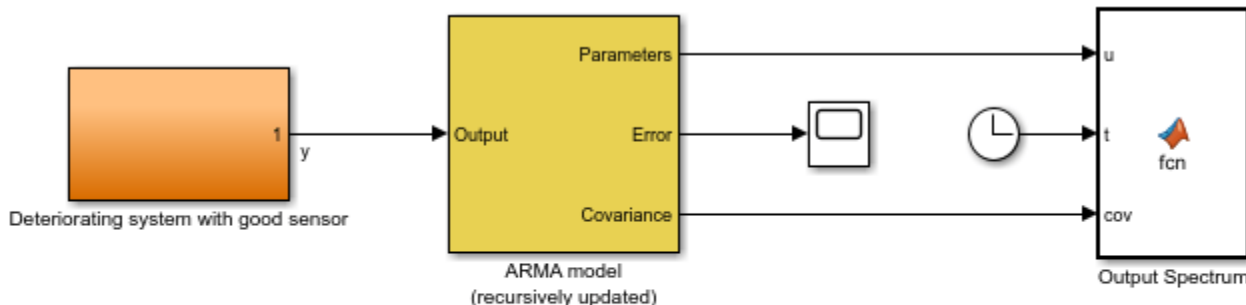

Copyright 2015-2018 The MathWorks, Inc.

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)
```
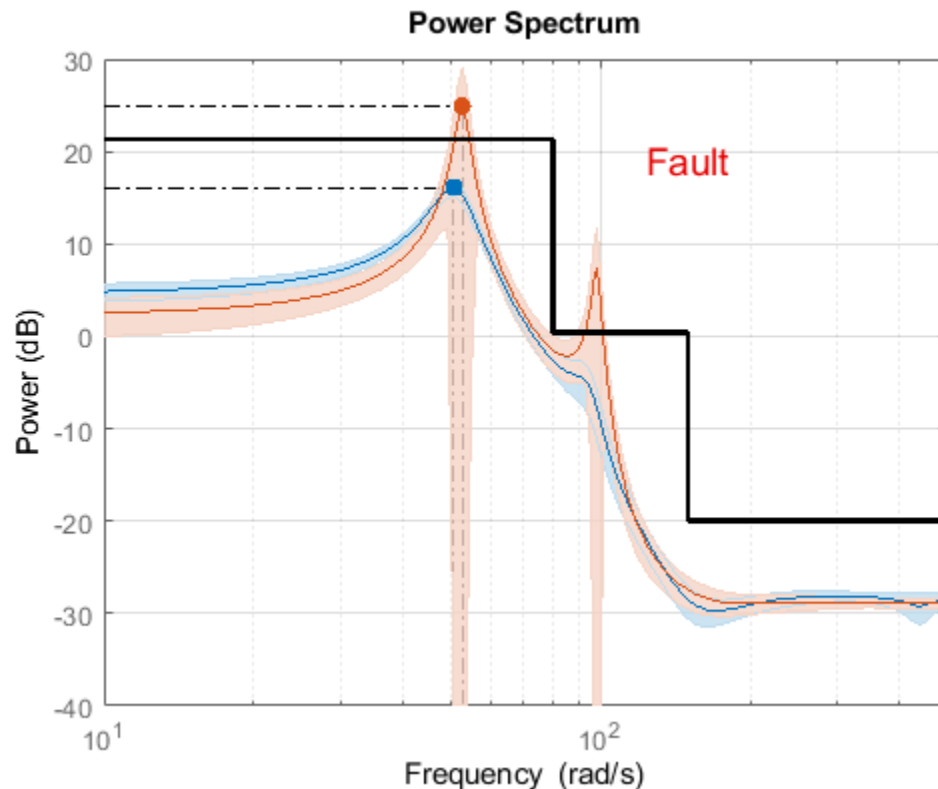

The running estimates of model parameters are also used to compute the system pole locations which are then fed into the SVM classifier to predict if the system is in the "good" or "fault" state. This decision is also displayed on the plot. When the normalized score of prediction is less than .3, the 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 and classification 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 classification 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 3-7

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
```
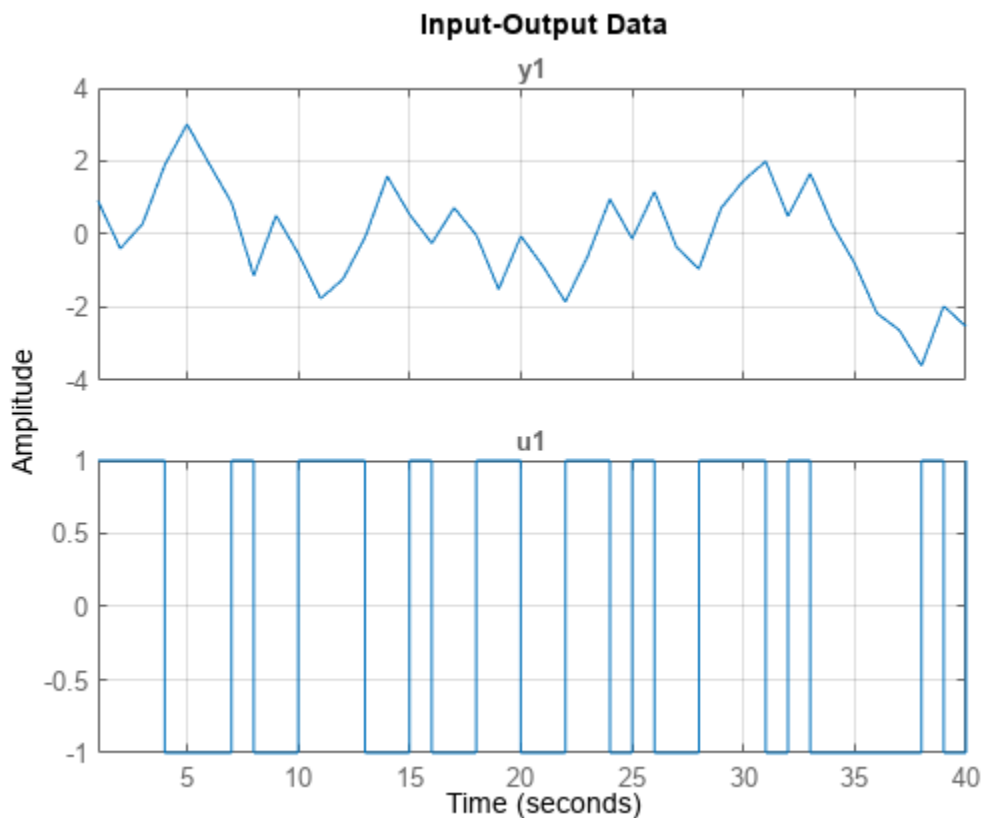

The transport delay change takes place around 20 seconds, but is not easy to see in the plot.

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)')
```

```matlab
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
```
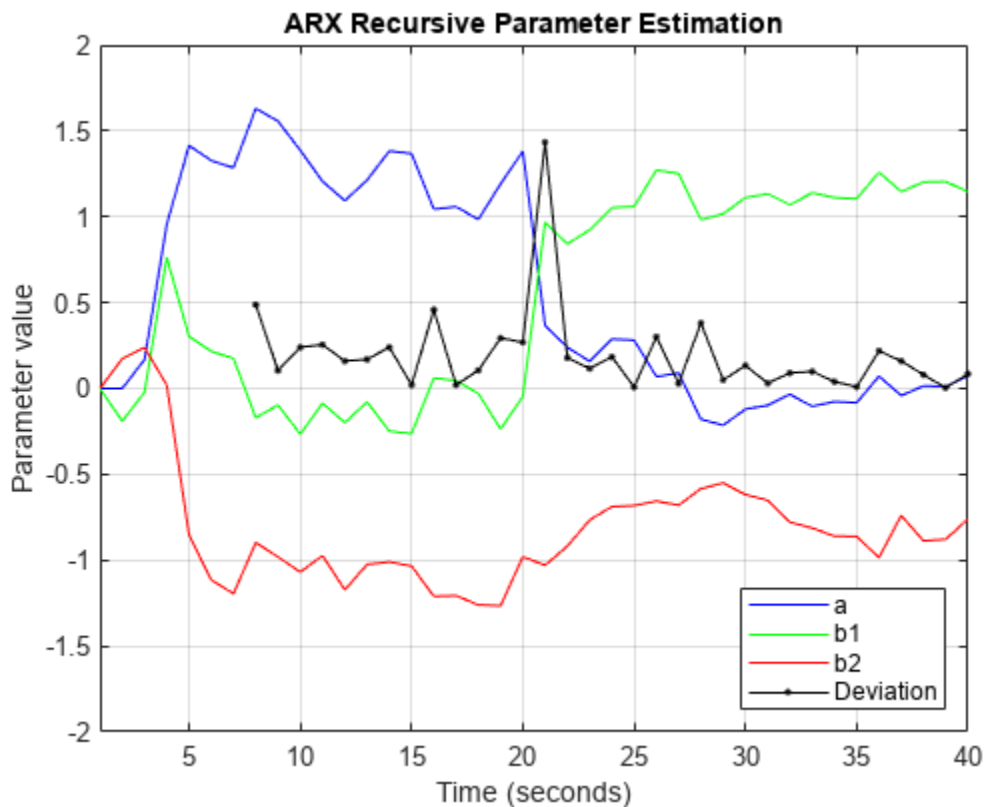

The first `n0` = 6 samples of the data are not used for computing the change-detector, L. During this interval the parameter changes are large owing to the unknown initial conditions.

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)
```
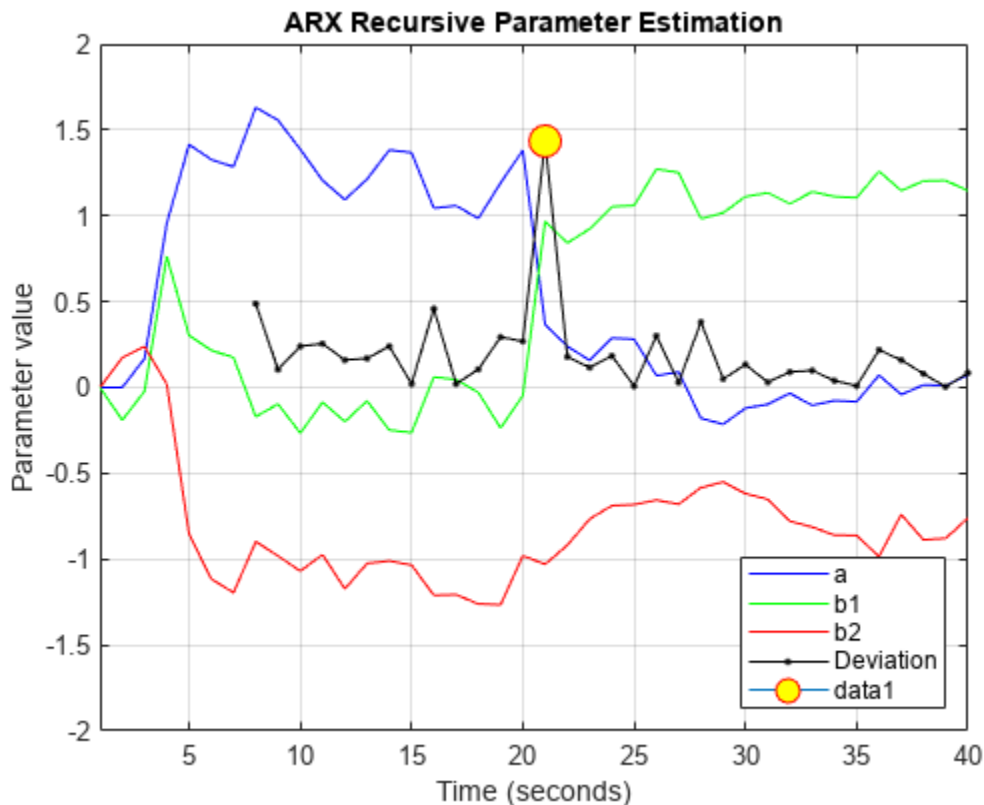


```
fprintf('Change in system delay detected at sample number %d.\n',Loc(I));
```

Change in system delay detected at sample number 21.

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')
```
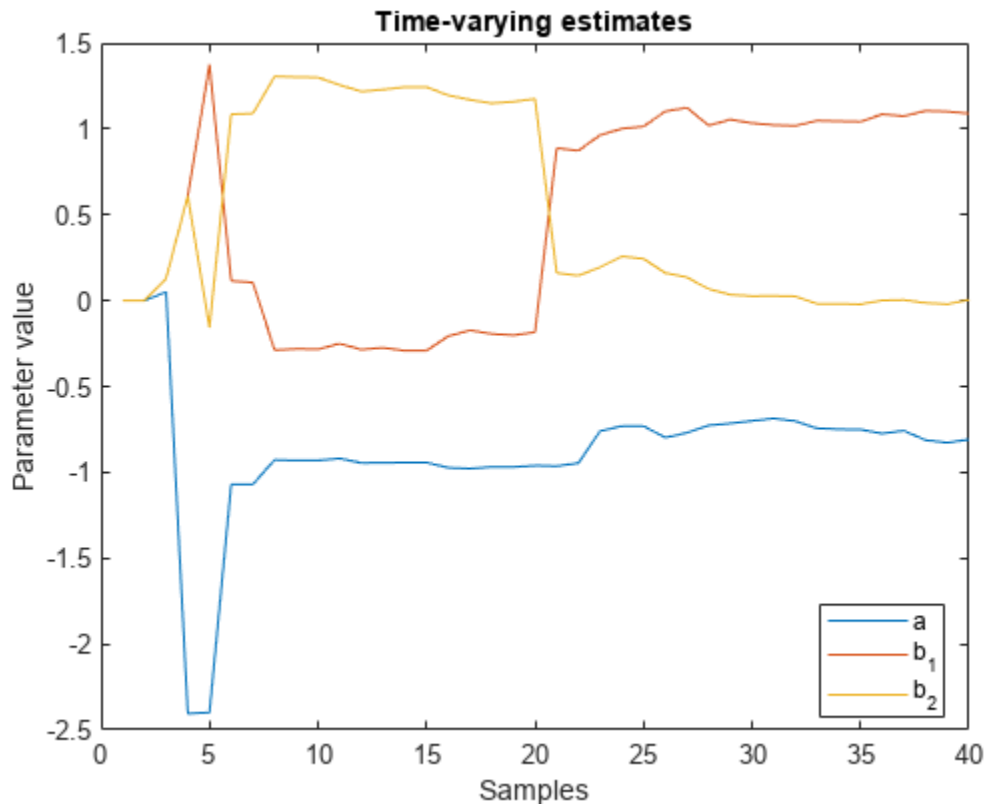

Note the similarity between these parameter trajectories and those estimated using `recursiveARX`.

`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')
```
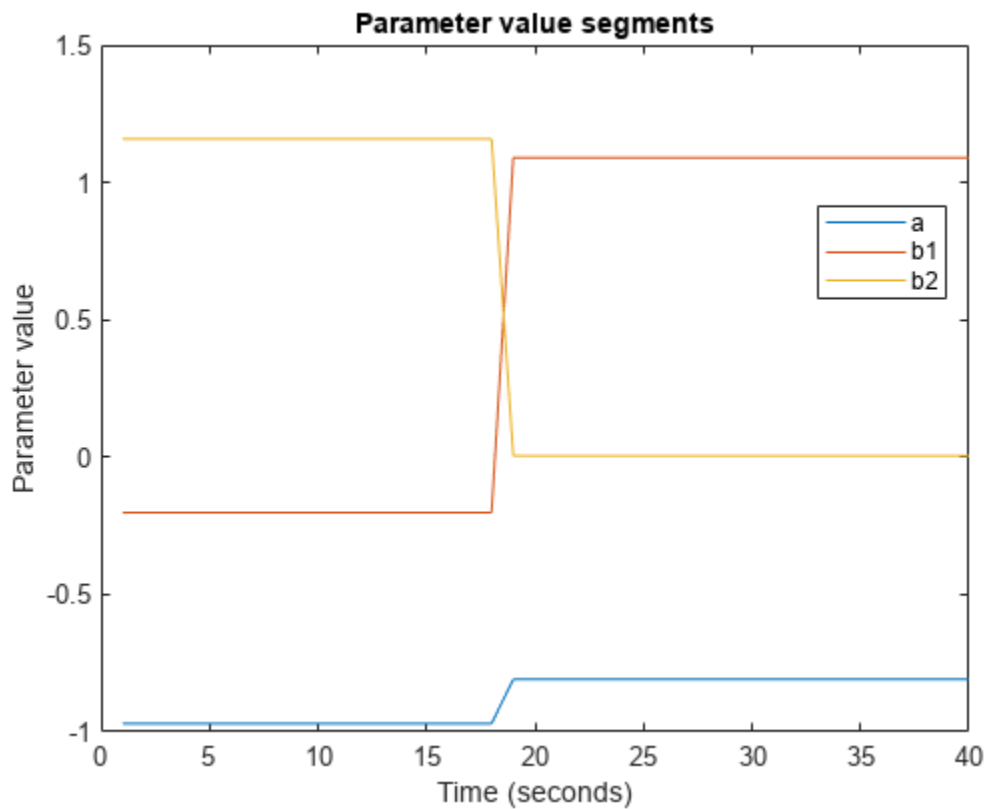

A change is seen in the parameter values around sample number 19. The value of `b1` changes from a small (close to zero) to large (close to 1) value. The value of `b2` shows the opposite pattern. This change in the values of the `B` parameters indicates a change in the transport delay.

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 assess 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 3-7

Chemical Process Fault Detection Using Deep Learning

This example shows how to use simulation data to train a neural network that can detect faults in a chemical process. The network detects the faults in the simulated process with high accuracy. The typical workflow is as follows:

1 Preprocess the data

2 Design the layer architecture

3 Train the network

4 Perform validation

5 Test the network

Download Data Set

This example uses MATLAB-formatted files converted by MathWorks® from the Tennessee Eastman Process (TEP) simulation data [1] on page 4-116. These files are available at the MathWorks support files site. See the disclaimer.

The data set consists of four components — fault-free training, fault-free testing, faulty training, and faulty testing. Download each file separately.

```
url = 'https://www.mathworks.com/supportfiles/predmaint/chemical-process-fault-detection-data/fau
websave('faultytesting.mat',url);
url = 'https://www.mathworks.com/supportfiles/predmaint/chemical-process-fault-detection-data/fau
websave('faultytraining.mat',url);
url = 'https://www.mathworks.com/supportfiles/predmaint/chemical-process-fault-detection-data/fau
websave('faultfreetesting.mat',url);
url = 'https://www.mathworks.com/supportfiles/predmaint/chemical-process-fault-detection-data/fau
websave('faultfreetraining.mat',url);
```

Load the downloaded files into the MATLAB® workspace.

```
load('faultfreetesting.mat');
load('faultfreetraining.mat');
load('faultytesting.mat');
load('faultytraining.mat');
```

Each component contains data from simulations that were run for every permutation of two parameters:

• Fault Number — For faulty data sets, an integer value from 1 to 20 that represents a different simulated fault. For fault-free data sets, a value of 0.

• Simulation run — For all data sets, integer values from 1 to 500, where each value represents a unique random generator state for the simulation.

The length of each simulation was dependent on the data set. All simulations were sampled every three minutes.

• Training data sets contain 500 time samples from 25 hours of simulation.

• Testing data sets contain 960 time samples from 48 hours of simulation.

Each data frame has the following variables in its columns:

- Column 1 (`faultNumber`) indicates the fault type, which varies from 0 through 20. A fault number 0 means fault-free while fault numbers 1 to 20 represent different fault types in the TEP.
- Column 2 (`simulationRun`) indicates the number of times the TEP simulation ran to obtain complete data. In the training and test data sets, the number of runs varies from 1 to 500 for all fault numbers. Every `simulationRun` value represents a different random generator state for the simulation.
- Column 3 (`sample`) indicates the number of times TEP variables were recorded per simulation. The number varies from 1 to 500 for the training data sets and from 1 to 960 for the testing data sets. The TEP variables (columns 4 to 55) were sampled every 3 minutes for a duration of 25 hours and 48 hours for the training and testing data sets respectively.
- Columns 4–44 (`xmeas_1` through `xmeas_41`) contain the measured variables of the TEP.
- Columns 45–55 (`xmv_1` through `xmv_11`) contain the manipulated variables of the TEP.

Examine subsections of two of the files.

```
head(faultfreetraining,4)
```

```
ans=4×55 table
    faultNumber     simulationRun     sample     xmeas_1     xmeas_2     xmeas_3     xmeas_4     xmeas_5
    _____     _____     _____     _____     _____     _____     _____     _____

         0               1              1        0.25038      3674        4529        9.232       26.889
         0               1              2        0.25109      3659.4      4556.6      9.4264       26.721
         0               1              3        0.25038      3660.3      4477.8      9.4426       26.875
         0               1              4        0.24977      3661.3      4512.1      9.4776       26.758
```

```
head(faultytraining,4)
```

```
ans=4×55 table
    faultNumber     simulationRun     sample     xmeas_1     xmeas_2     xmeas_3     xmeas_4     xmeas_5
    _____     _____     _____     _____     _____     _____     _____     _____

         1               1              1        0.25038      3674        4529        9.232       26.889
         1               1              2        0.25109      3659.4      4556.6      9.4264       26.721
         1               1              3        0.25038      3660.3      4477.8      9.4426       26.875
         1               1              4        0.24977      3661.3      4512.1      9.4776       26.758
```

Clean Data

Remove data entries with the fault numbers 3, 9, and 15 in both the training and testing data sets. These fault numbers are not recognizable, and the associated simulation results are erroneous.

```
faultytesting(faultytesting.faultNumber == 3,:) = [];
faultytesting(faultytesting.faultNumber == 9,:) = [];
faultytesting(faultytesting.faultNumber == 15,:) = [];
```

```
faultytraining(faultytraining.faultNumber == 3,:) = [];
faultytraining(faultytraining.faultNumber == 9,:) = [];
faultytraining(faultytraining.faultNumber == 15,:) = [];
```

Divide Data

Divide the training data into training and validation data by reserving 20 percent of the training data for validation. Using a validation data set enables you to evaluate the model fit on the training data

set while you tune the model hyperparameters. Data splitting is commonly used to prevent the network from overfitting and underfitting.

Get the total number of rows in both faulty and fault-free training data sets.

```
H1 = height(faultfreetraining);
H2 = height(faultytraining);
```

The simulation run is the number of times the TEP process was repeated with a particular fault type. Get the maximum simulation run from the training data set as well as from the testing data set.

```
msTrain = max(faultfreetraining.simulationRun);
msTest = max(faultytesting.simulationRun);
```

Calculate the maximum simulation run for the validation data.

```
rTrain = 0.80;
msVal = ceil(msTrain*(1 - rTrain));
msTrain = msTrain*rTrain;
```

Get the maximum number of samples or time steps (that is, the maximum number of times that data was recorded during a TEP simulation).

```
sampleTrain = max(faultfreetraining.sample);
sampleTest = max(faultfreetesting.sample);
```

Get the division point (row number) in the fault-free and faulty training data sets to create validation data sets from the training data sets.

```
rowLim1 = ceil(rTrain*H1);
rowLim2 = ceil(rTrain*H2);

trainingData = [faultfreetraining{1:rowLim1,:}; faultytraining{1:rowLim2,:}];
validationData = [faultfreetraining{rowLim1 + 1:end,:}; faultytraining{rowLim2 + 1:end,:}];
testingData = [faultfreetesting{:,:}; faultytesting{:,:}];
```

Network Design and Preprocessing

The final data set (consisting of training, validation, and testing data) contains 52 signals with 500 uniform time steps. Hence, the signal, or sequence, needs to be classified to its correct fault number which makes it a problem of sequence classification.

- Long short-term memory (LSTM) networks are suited to the classification of sequence data.
- LSTM networks are good for time-series data as they tend to remember the uniqueness of past signals in order to classify new signals
- An LSTM network enables you to input sequence data into a network and make predictions based on the individual time steps of the sequence data. For more information on LSTM networks, see "Long Short-Term Memory Neural Networks" (Deep Learning Toolbox).
- To train the network to classify sequences using the `trainNetwork` (Deep Learning Toolbox) function, you must first preprocess the data. The data must be in cell arrays, where each element of the cell array is a matrix representing a set of 52 signals in a single simulation. Each matrix in the cell array is the set of signals for a particular simulation of TEP and can either be faulty or fault-free. Each set of signals points to a specific fault class ranging from 0 through 20.

As was described previously in the Data Set section, the data contains 52 variables whose values are recorded over a certain amount of time in a simulation. The `sample` variable represents the number

of times these 52 variables are recorded in one simulation run. The maximum value of the `sample` variable is 500 in the training data set and 960 in the testing data set. Thus, for each simulation, there is a set of 52 signals of length 500 or 960. Each set of signals belongs to a particular simulation run of the TEP and points to a particular fault type in the range 0 – 20.

The training and test datasets both contain 500 simulations for each fault type. Twenty percent (from training) is kept for validation which leaves the training data set with 400 simulations per fault type and validation data with 100 simulations per fault type. Use the helper function `helperPreprocess` to create sets of signals, where each set is a double matrix in a single element of the cell array that represents a single TEP simulation. Hence, the sizes of the final training, validation, and testing data sets are as follows:

- Size of `Xtrain`: (Total number of simulations) X (Total number of fault types) = 400 X 18 = 7200
- Size of `XVal`: (Total number of simulations) X (Total number of fault types) = 100 X 18 = 1800
- Size of `Xtest`: (Total number of simulations) X (Total number of fault types) = 500 X 18 = 9000

In the data set, the first 500 simulations are of `0` fault type (fault-free) and the order of the subsequent faulty simulations is known. This knowledge enables the creation of true responses for the training, validation, and testing data sets.

```
Xtrain = helperPreprocess(trainingData,sampleTrain);
Ytrain = categorical([zeros(msTrain,1);repmat([1,2,4:8,10:14,16:20],1,msTrain)']);

XVal = helperPreprocess(validationData,sampleTrain);
YVal = categorical([zeros(msVal,1);repmat([1,2,4:8,10:14,16:20],1,msVal)']);

Xtest = helperPreprocess(testingData,sampleTest);
Ytest = categorical([zeros(msTest,1);repmat([1,2,4:8,10:14,16:20],1,msTest)']);
```

Normalize Data Sets

Normalization is a technique that scales the numeric values in a data set to a common scale without distorting differences in the range of values. This technique ensures that a variable with a larger value does not dominate other variables in the training. It also converts the numeric values in a higher range to a smaller range (usually –1 to 1) without losing any important information required for training.

Compute the mean and the standard deviation for 52 signals using data from all simulations in the training data set.

```
tMean = mean(trainingData(:,4:end))';
tSigma = std(trainingData(:,4:end))';
```

Use the helper function `helperNormalize` to apply normalization to each cell in the three data sets based on the mean and standard deviation of the training data.

```
Xtrain = helperNormalize(Xtrain, tMean, tSigma);
XVal = helperNormalize(XVal, tMean, tSigma);
Xtest = helperNormalize(Xtest, tMean, tSigma);
```

Visualize Data

The `Xtrain` data set contains 400 fault-free simulations followed by 6800 faulty simulations.Visualize the fault-free and faulty data. First, create a plot of the fault-free data. For the purposes of this example, plot and label only 10 signals in the `Xtrain` data set to create an easy-to-read figure.

```
figure;
splot = 10;
plot(Xtrain{1}(1:10,:)');
xlabel("Time Step");
title("Training Observation for Non-Faulty Data");
legend("Signal " + string(1:splot),'Location','northeastoutside');
```
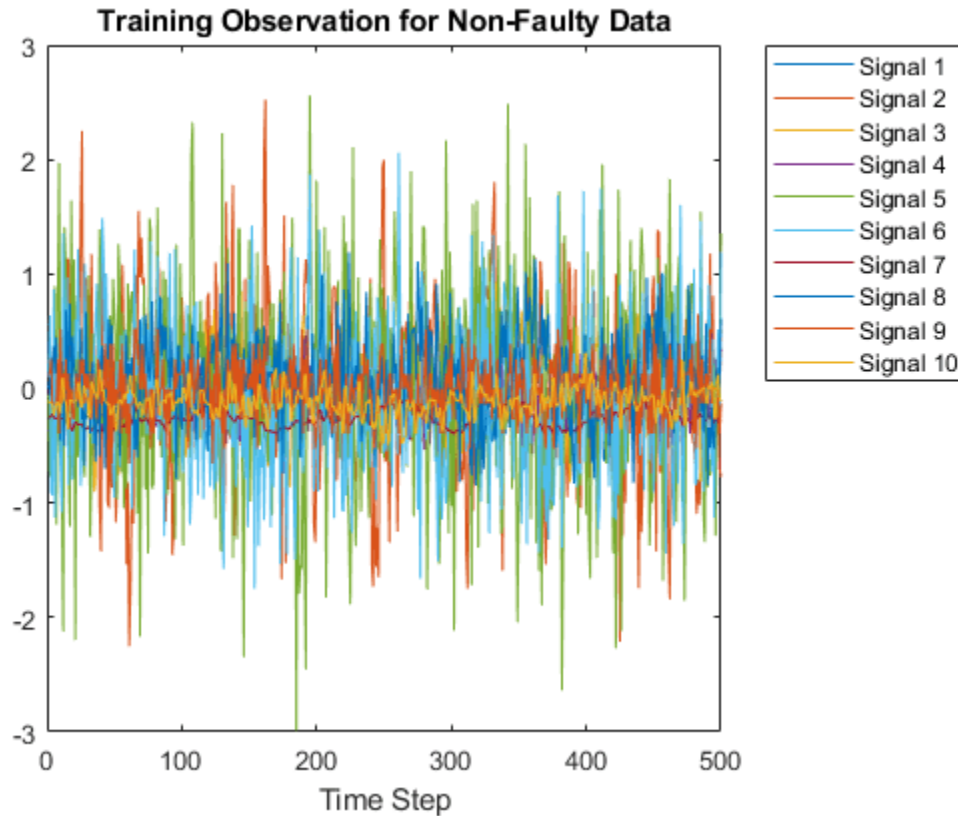

Now, compare the fault-free plot to a faulty plot by plotting any of the cell array elements after 400.

```
figure;
plot(Xtrain{1000}(1:10,:)');
xlabel("Time Step");
title("Training Observation for Faulty Data");
legend("Signal " + string(1:splot),'Location','northeastoutside');
```
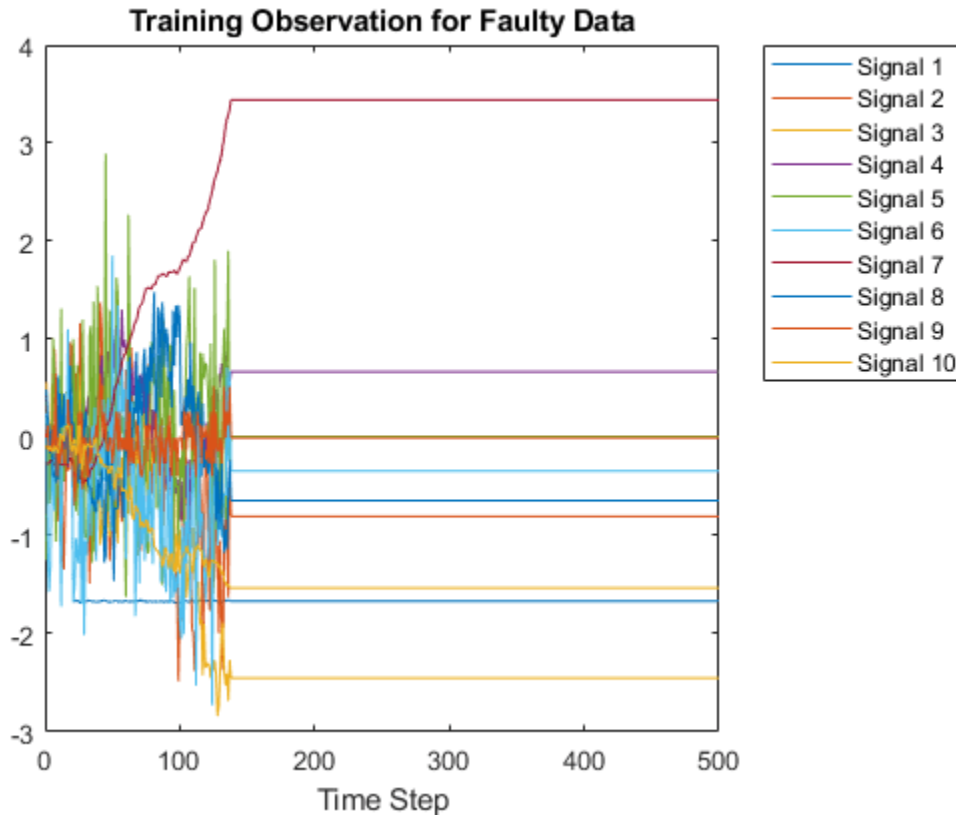
Layer Architecture and Training Options

LSTM layers are a good choice for sequence classification as LSTM layers tend to remember only the important aspects of the input sequence.

- Specify the input layer `sequenceInputLayer` to be of the same size as the number of input signals (52).
- Specify 3 LSTM hidden layers with 52, 40, and 25 units. This specification is inspired by the experiment performed in [2] on page 4-116. For more information on using LSTM networks for sequence classification, see "Sequence Classification Using Deep Learning" (Deep Learning Toolbox).
- Add 3 dropout layers in between the LSTM layers to prevent over-fitting. A dropout layer randomly sets input elements of the next layer to zero with a given probability so that the network does not become sensitive to a small set of neurons in the layer
- Finally, for classification, include a fully connected layer of the same size as the number of output classes (18). After the fully connected layer, include a softmax layer that assigns decimal probabilities (prediction possibility) to each class in a multi-class problem and a classification layer to output the final fault type based on output from the softmax layer.

```
numSignals = 52;
numHiddenUnits2 = 52;
numHiddenUnits3 = 40;
numHiddenUnits4 = 25;
numClasses = 18;
```

```
layers = [ ...
    sequenceInputLayer(numSignals)
    lstmLayer(numHiddenUnits2,'OutputMode','sequence')
    dropoutLayer(0.2)
    lstmLayer(numHiddenUnits3,'OutputMode','sequence')
    dropoutLayer(0.2)
    lstmLayer(numHiddenUnits4,'OutputMode','last')
    dropoutLayer(0.2)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Set the training options that `trainNetwork` uses.

Maintain the default value of name-value pair `'ExecutionEnvironment'` as `'auto'`. With this setting, the software chooses the execution environment automatically. By default, `trainNetwork` uses a GPU if one is available, otherwise, it uses a CPU. Training on a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see "GPU Computing Requirements" (Parallel Computing Toolbox). Because this example uses a large amount of data, using GPU speeds up training time considerably.

Setting the name-value argument pair `'Shuffle'` to `'every-epoch'` avoids discarding the same data every epoch.

For more information on training options for deep learning, see `trainingOptions` (Deep Learning Toolbox).

```
maxEpochs = 30;
miniBatchSize = 50;

options = trainingOptions('adam', ...
    'ExecutionEnvironment','auto', ...
    'GradientThreshold',1, ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize', miniBatchSize,...
    'Shuffle','every-epoch', ...
    'Verbose',0, ...
    'Plots','training-progress',...
    'ValidationData',{XVal,YVal});
```

Train Network

Train the LSTM network using `trainNetwork`.

```
net = trainNetwork(Xtrain,Ytrain,layers,options);
```
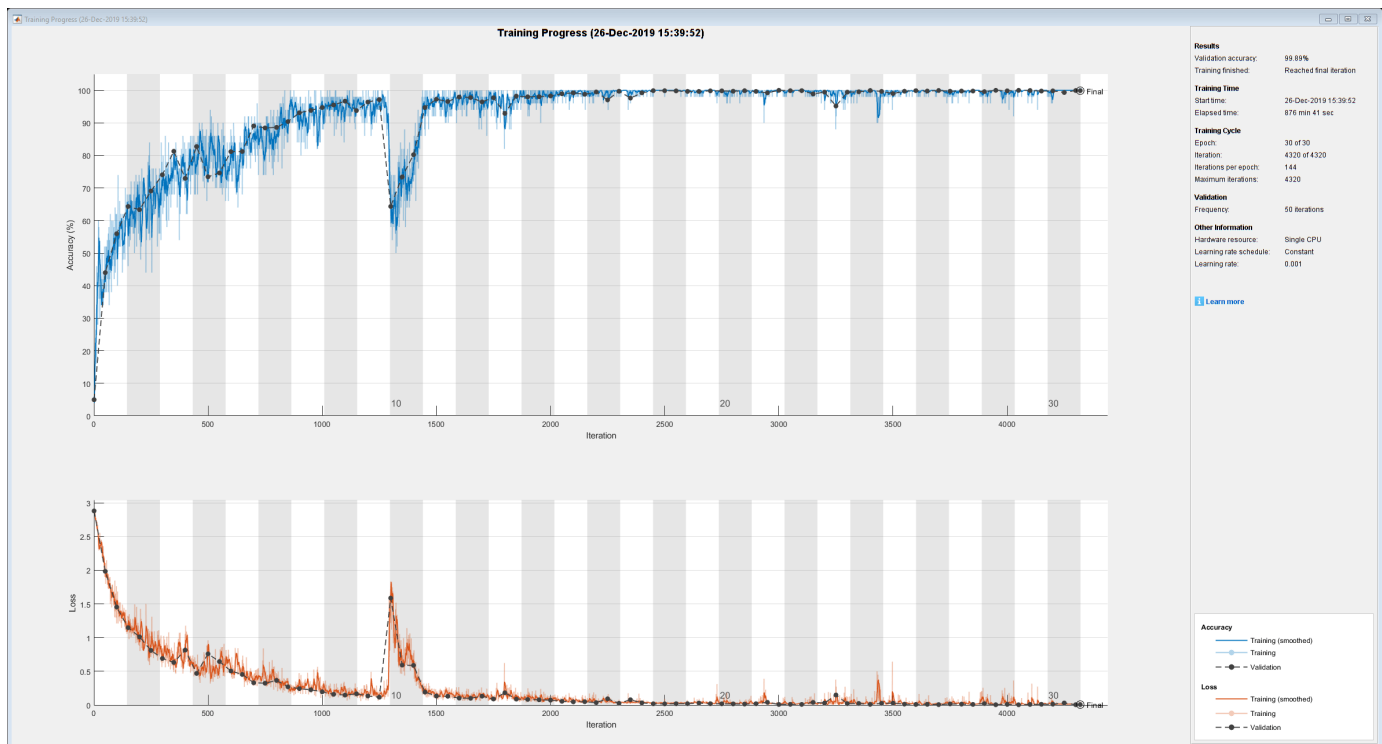
The training progress figure displays a plot of the network accuracy. To the right of the figure, view information on the training time and settings.

Testing Network

Run the trained network on the test set and predict the fault type in the signals.

```
Ypred = classify(net,Xtest,...
    'MiniBatchSize', miniBatchSize,...
    'ExecutionEnvironment','auto');
```

Calculate the accuracy. The accuracy is the number of true labels in the test data that match the classifications from `classify` divided by the number of images in the test data.

```
acc = sum(Ypred == Ytest)./numel(Ypred)
```

```
acc = 0.9992
```

High accuracy indicates that the neural network is successfully able to identify the fault type of unseen signals with minimal errors. Hence, the higher the accuracy, the better the network.

Plot a confusion matrix using true class labels of the test signals to determine how well the network identifies each fault.

```
confusionchart(Ytest,Ypred);
```
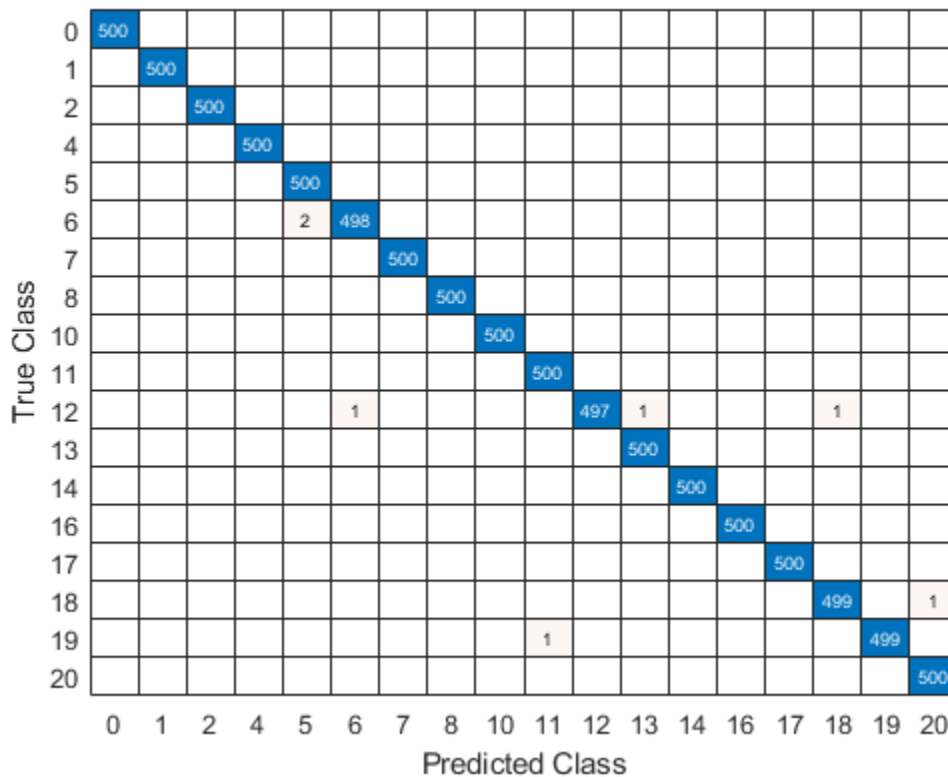
Using a confusion matrix, you can assess the effectiveness of a classification network. the confusion matrix has numerical values in the main diagonal and zeros elsewhere. The trained network in this example is effective and classifies more than 99% of signals correctly.

References

[1] Rieth, C. A., B. D. Amsel, R. Tran., and B. Maia. "Additional Tennessee Eastman Process Simulation Data for Anomaly Detection Evaluation." Harvard Dataverse, Version 1, 2017. https://doi.org/10.7910/DVN/6C3JR1.

[2] Heo, S., and J. H. Lee. "Fault Detection and Classification Using Artificial Neural Networks." Department of Chemical and Biomolecular Engineering, Korea Advanced Institute of Science and Technology.

Helper Functions

helperPreprocess

The helper function `helperPreprocess` uses the maximum sample number to preprocess the data. The sample number indicates the signal length, which is consistent across the data set. A for-loop goes over the data set with a signal length filter to form sets of 52 signals. Each set is an element of a cell array. Each cell array represents a single simulation.

```matlab
function processed = helperPreprocess(mydata,limit)
    H = size(mydata);
    processed = {};
    for ind = 1:limit:H
        x = mydata(ind:(ind+(limit-1)),4:end);
        processed = [processed; x'];
    end
end
```

helperNormalize

The helper function `helperNormalize` uses the data, mean, and standard deviation to normalize the data.

```matlab
function data = helperNormalize(data,m,s)
    for ind = 1:size(data)
        data{ind} = (data{ind} - m)./s;
    end
end
```

See Also
trainNetwork | trainingOptions

More About
- "Long Short-Term Memory Neural Networks" (Deep Learning Toolbox)
- "Sequence Classification Using Deep Learning" (Deep Learning Toolbox)

Remaining Useful Life Estimation Using Convolutional Neural Network

This example shows how to predict the remaining useful life (RUL) of engines by using deep convolutional neural networks (CNNs) [1] on page 4-128. The advantage of a deep learning approach is that you do not need manual feature extraction or feature selection for your model to predict RUL. Furthermore,you do not need prior knowledge of machine health prognostics or signal processing to develop a deep learning based RUL prediction model.

Download Data set

This example uses the Turbofan Engine Degradation Simulation data set [1] on page 4-128. The data set is in ZIP file format, and contains run-to-failure time-series data for four different sets (namely FD001, FD002, FD003, and FD004) simulated under different combinations of operational conditions and fault modes.

This example uses only the FD001 data set, which is further divided into training and test subsets. The training subset contains simulated time series data for 100 engines. Each engine has several sensors whose values are recorded at a given instance in a continuous process. Hence, the sequence of recorded data varies in length and corresponds to a full run-to-failure (RTF) instance. The test subset contains 100 partial sequences and corresponding values of the remaining useful life at the end of each sequence.

Create a directory to store the Turbofan Engine Degradation Simulation data set.

```
dataFolder = "data";
if ~exist(dataFolder,'dir')
    mkdir(dataFolder);
end
```

Download and extract the Turbofan Engine Degradation Simulation data set.

```
filename = matlab.internal.examples.downloadSupportFile("nnet","data/TurbofanEngineDegradationSir
unzip(filename,dataFolder)
```

The data folder now contains text files with 26 columns of numbers, separated by spaces. Each row is a snapshot of data taken during a single operational cycle, and each column represents a different variable:

- Column 1 — Unit number
- Column 2 — Timestamp
- Columns 3–5 — Operational settings
- Columns 6–26 — Sensor measurements 1–21

Preprocess Training Data

Load the data using the function `localLoadData`. The function extracts the data from a data file and returns a table which contains the training predictors and corresponding response (i.e., RUL) sequences. Each row represents a different engine.

```
filenameTrainPredictors = fullfile(dataFolder,"train_FD001.txt");
rawTrain = localLoadData(filenameTrainPredictors);
```

Examine the run-to-failure data for one of the engines.

```
head(rawTrain.X{1},8)
```

ans=8×26 table

id	timeStamp	op_setting_1	op_setting_2	op_setting_3	sensor_1	sensor_2	se
1	1	-0.0007	-0.0004	100	518.67	641.82	
1	2	0.0019	-0.0003	100	518.67	642.15	
1	3	-0.0043	0.0003	100	518.67	642.35	
1	4	0.0007	0	100	518.67	642.35	
1	5	-0.0019	-0.0002	100	518.67	642.37	
1	6	-0.0043	-0.0001	100	518.67	642.1	
1	7	0.001	0.0001	100	518.67	642.48	
1	8	-0.0034	0.0003	100	518.67	642.56	

Examine the response data for one of the engines.

```
rawTrain.Y{1}(1:8)
```

ans = 8×1

```
   191
   190
   189
   188
   187
   186
   185
   184
```

Visualize the time-series data for some of the predictors.

```
stackedplot(rawTrain.X{1},[3,5,6,7,8,15,16,24],XVariable='timeStamp')
```

4-119
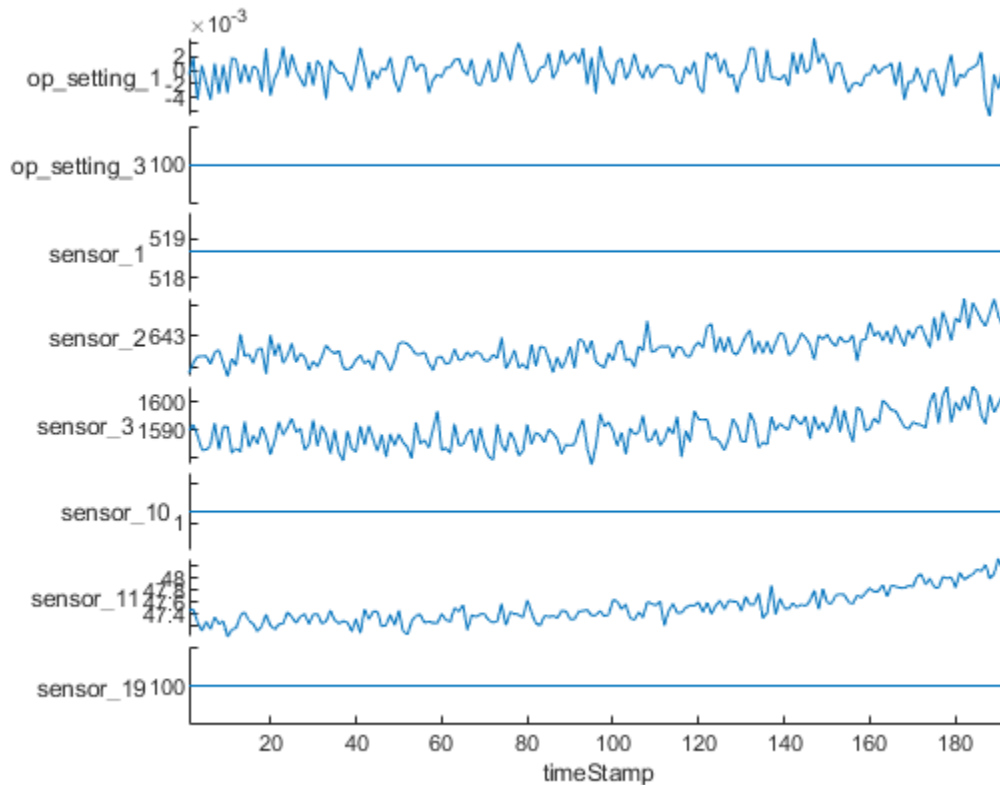
Remove Features with Less Variability

Features that remain constant for all time steps can negatively impact the training. Use the `prognosability` function to measure the variability of features at failure.

```
prog = prognosability(rawTrain.X,"timeStamp");
```

For some features, prognosability is equal to zero or NaN. Discard these features.

```
idxToRemove = prog.Variables==0 | isnan(prog.Variables);
featToRetain = prog.Properties.VariableNames(~idxToRemove);
for i = 1:height(rawTrain)
    rawTrain.X{i} = rawTrain.X{i}{:,featToRetain};
end
```

Normalize Training Predictors

Normalize the training predictors to have zero mean and unit variance.

```
[~,Xmu,Xsigma] = zscore(vertcat(rawTrain.X{:}));
preTrain = table();
for i = 1:numel(rawTrain.X)
    preTrain.X{i} = (rawTrain.X{i} - Xmu) ./ Xsigma;
end
```

Clip Responses

The response data represents the RUL value over life for each engine and is based individual engine lifetime. The sequence assumes a linear degradation from the time of the initial measurement to the time of engine failure.

In order for network to focus on the part of the data where engines are more likely to fail (end of the engine's life), clip the responses at the threshold of 150. Clipping the responses causes the network to treat instances with higher RUL values as equal.

```
rulThreshold = 150;
for i = 1:numel(rawTrain.Y)
    preTrain.Y{i} = min(rawTrain.Y{i},rulThreshold);
end
```

This figure shows the first observation and the corresponding response (RUL), which is clipped at the threshold. The green overlay defines the clipping region on both sensor amd RUL plots.
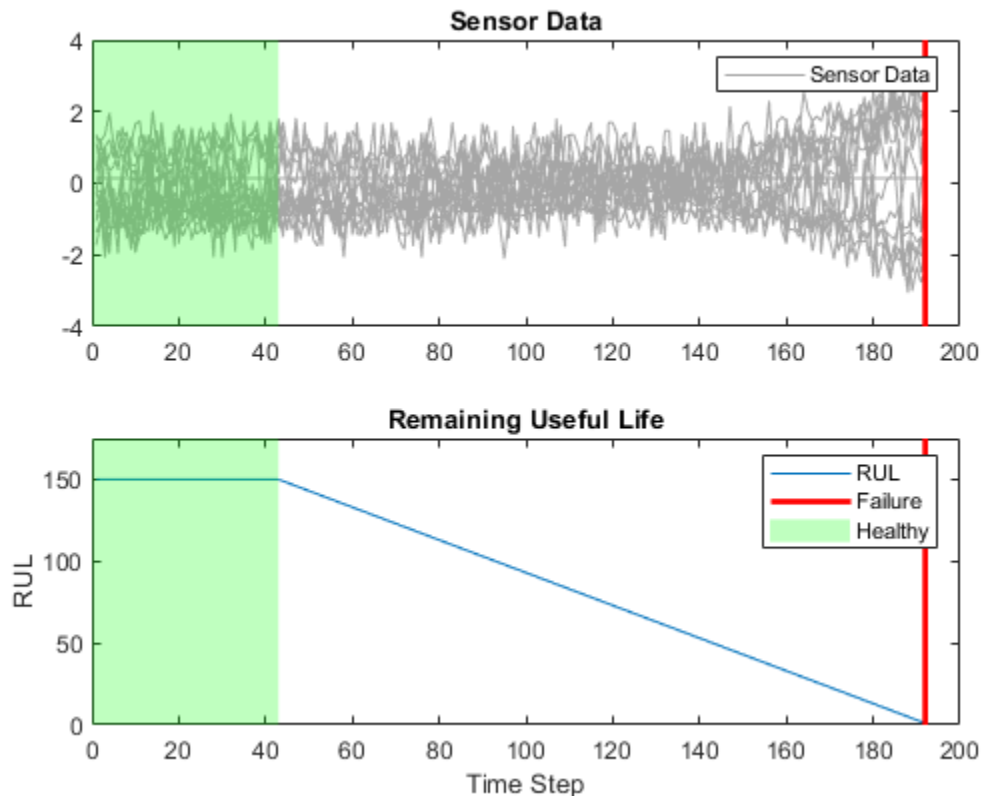
Prepare Data for Padding

This network supports input data with varying sequence lengths. When passing data through the network, the software pads, truncates, or splits sequences so that all the sequences in each mini-batch have the specified length.

To minimize the amount of padding added to the mini-batches, sort the training data by sequence length. Then, choose a mini-batch size which divides the training data evenly and reduces the amount of padding in the mini-batches.

Sort the training data by sequence length.

```
for i = 1:size(preTrain,1)
    preTrain.X{i} = preTrain.X{i}';    %Transpose training data to have features in the first dir
    preTrain.Y{i} = preTrain.Y{i}';    %Transpose responses corresponding to the training data
    sequence = preTrain.X{i};
    sequenceLengths(i) = size(sequence,2);
end

[sequenceLengths,idx] = sort(sequenceLengths,'descend');
XTrain = preTrain.X(idx);
YTrain = preTrain.Y(idx);
```

Network Architecture

The deep convolutional neural network architecture used for RUL estimation is described in [1] on page 4-128.

Here, you process and sort the data in a sequence format, with the first dimension representing the number of selected features and the second dimension representing the length of the time sequence. You bundle convolutional layers with batch normalization layer followed by an activation layer (relu in this case) and then stack the layers together for feature extraction. The fully connected layers and regression layer are used at the end to get the final RUL value as output .

The selected network architecture applies a 1-D convolution along the time sequence direction only. Therefore, the order of features do not impact the training and only trends in one feature at a time are considered.

Define the network architecture. Create a CNN that consists of five consecutive sets of a convolution 1-d, batch normalization and, a relu layer, with increasing `filterSize` and `numFilters` as the first two input arguments to `convolution1dLayer`, followed by a fully connected layer of size `numHiddenUnits` and a dropout layer with a dropout probability of 0.5. Since the network predicts the remaining useful life (RUL) of the turbofan engine, set `numResponses` to 1 in the second fully connected layer and a regression layer as the last layer of the network.

To compensate for the varying time-sequences in the training data, use `Padding="causal"` as the Name-value pair input argument in `convolution1dLayer`.

```
numFeatures = size(XTrain{1},1);
numHiddenUnits = 100;
numResponses = 1;

layers = [
    sequenceInputLayer(numFeatures)
    convolution1dLayer(5,32,Padding="causal")
    batchNormalizationLayer()
    reluLayer()
    convolution1dLayer(7,64,Padding="causal")
    batchNormalizationLayer
    reluLayer()
    convolution1dLayer(11,128,Padding="causal")
    batchNormalizationLayer
    reluLayer()
    convolution1dLayer(13,256,Padding="causal")
    batchNormalizationLayer
    reluLayer()
    convolution1dLayer(15,512,Padding="causal")
    batchNormalizationLayer
    reluLayer()
```

```
fullyConnectedLayer(numHiddenUnits)
reluLayer()
dropoutLayer(0.5)
fullyConnectedLayer(numResponses)
regressionLayer()];
```

Train Network

Specify `trainingOptions` (Deep Learning Toolbox). Train for 40 epochs with minibatches of size 16 using the Adam optimizer. Set `LearnRateSchedule` to `piecewise`. Specify the learning rate as 0.01. To prevent the gradients from exploding, set the gradient threshold to 1. To keep the sequences sorted by length, set `'Shuffle'` to `'never'`. Turn on the training progress plot, and turn off the command window output (`Verbose`).

```
maxEpochs = 40;
miniBatchSize = 16;

options = trainingOptions('adam',...
    LearnRateSchedule='piecewise',...
    MaxEpochs=maxEpochs,...
    MiniBatchSize=miniBatchSize,...
    InitialLearnRate=0.01,...
    GradientThreshold=1,...
    Shuffle='never',...
    Plots='training-progress',...
    Verbose=0);
```

Train the network using `trainNetwork`. It should take about 1-2 minutes.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```
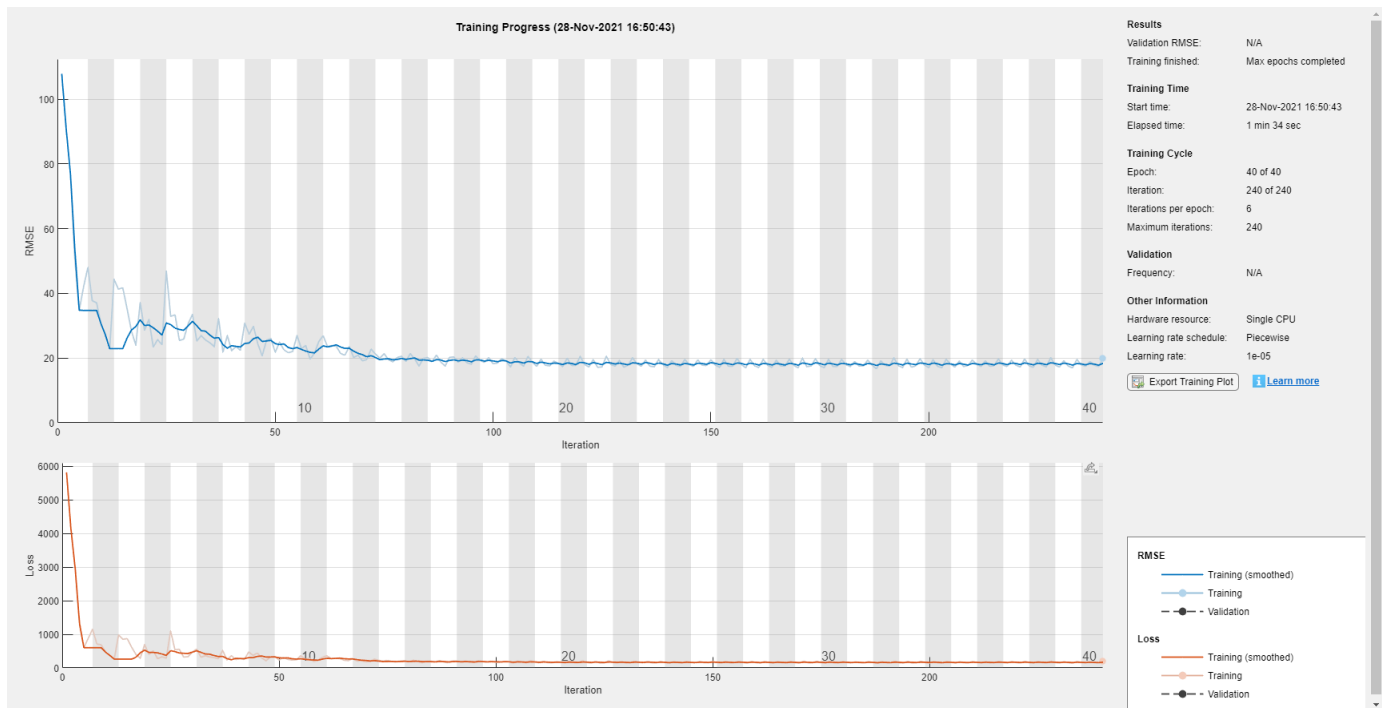

Plot the layer graph of the network to visualize the underlying network architecture.

```
figure;
lgraph = layerGraph(net.Layers);
plot(lgraph)
```
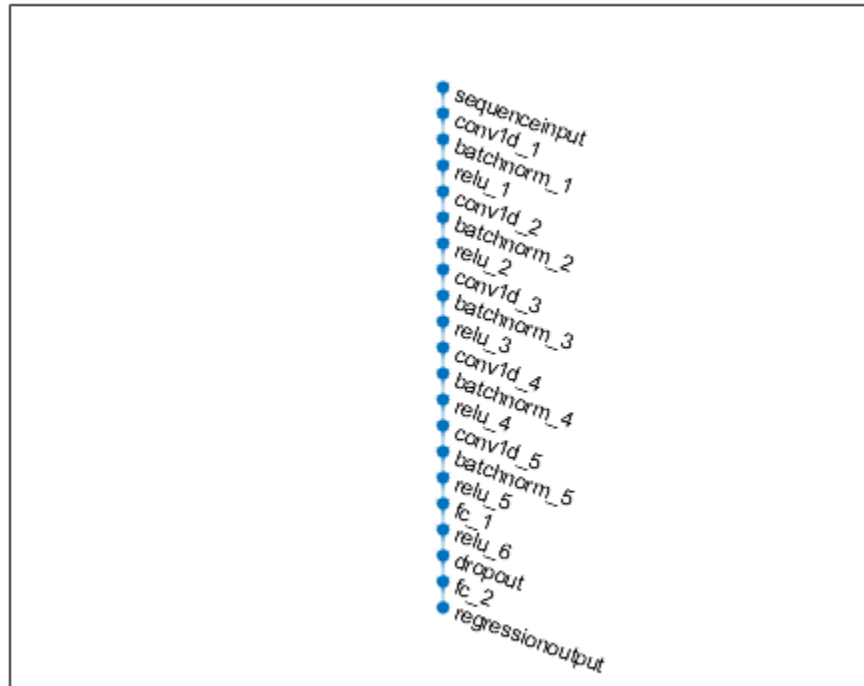

Test Network

The test data contains 100 partial sequences and corresponding values of the remaining useful life at the end of each sequence.

```
filenameTestPredictors = fullfile(dataFolder,'test_FD001.txt');
filenameTestResponses = fullfile(dataFolder,'RUL_FD001.txt');
dataTest = localLoadData(filenameTestPredictors,filenameTestResponses);
```

Prepare the test data set for predictions by performing the same preprocessing steps you use to prepare the training data set.

```
for i = 1:numel(dataTest.X)
    dataTest.X{i} = dataTest.X{i}{:,featToRetain};
    dataTest.X{i} = (dataTest.X{i} - Xmu) ./ Xsigma;
    dataTest.Y{i} = min(dataTest.Y{i},rulThreshold);
end
```

Create a table for storing the predicted response (`YPred`) along with the true response (`Y`). Make predictions on the test data using `predict`. To prevent the function from adding padding to the test data, specify the mini-batch size 1.

```
predictions = table(Size=[height(dataTest) 2],VariableTypes=["cell","cell"],VariableNames=["Y","
```

```
for i=1:height(dataTest)
    unit = dataTest.X{i}';
    predictions.Y{i} = dataTest.Y{i}';
    predictions.YPred{i} = predict(net,unit,MiniBatchSize=1);
end
```

Performance Metrics

Compute the root mean squared error (RMSE) across all time cycles of the test sequences to analyze how well the network performs on the test data.

```
for i = 1:size(predictions,1)
    predictions.RMSE(i) = sqrt(mean((predictions.Y{i} - predictions.YPred{i}).^2));
end
```

Create a histogram to visualize the distribution of RMSE values across all test engines.

```
figure;
histogram(predictions.RMSE,NumBins=10);
title("RMSE ( Mean: " + round(mean(predictions.RMSE),2) + " , StDev: " + round(std(predictions.RM
ylabel('Frequency');
xlabel('RMSE');
```
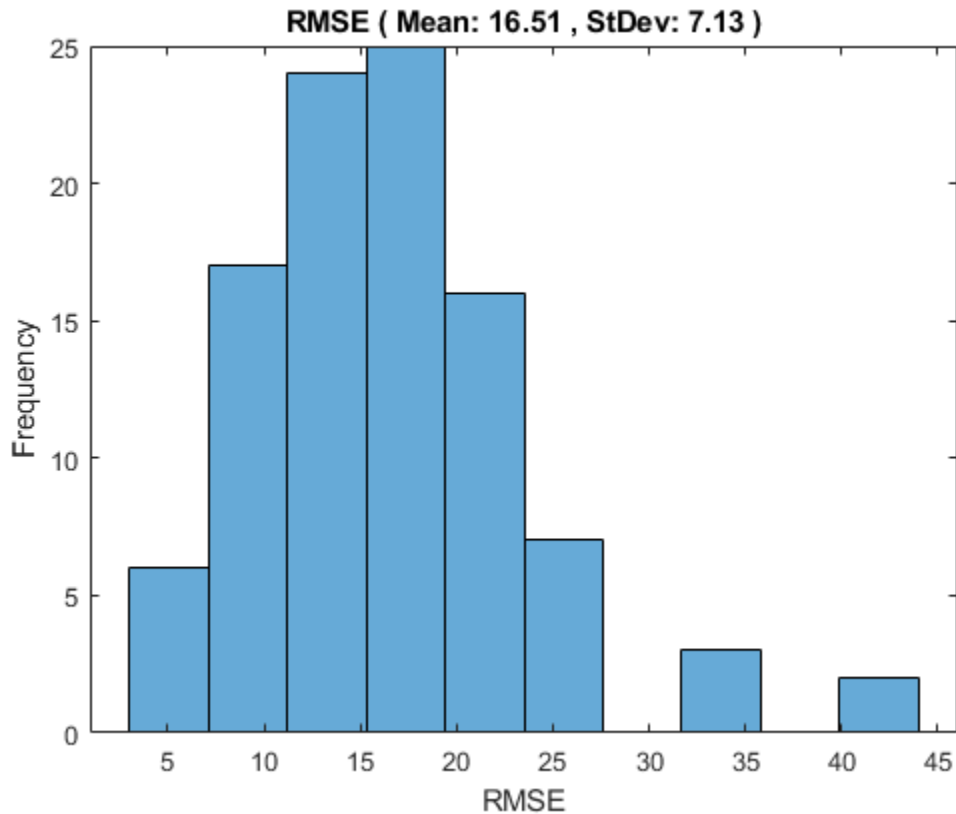

Additionally, to see how the network predictor performs throughout the given sequence of data in the test engines, use the `localLambdaPlot` function to plot the predicted RUL against the true RUL of a random test engine.

```
figure;
localLambdaPlot(predictions,"random");
```
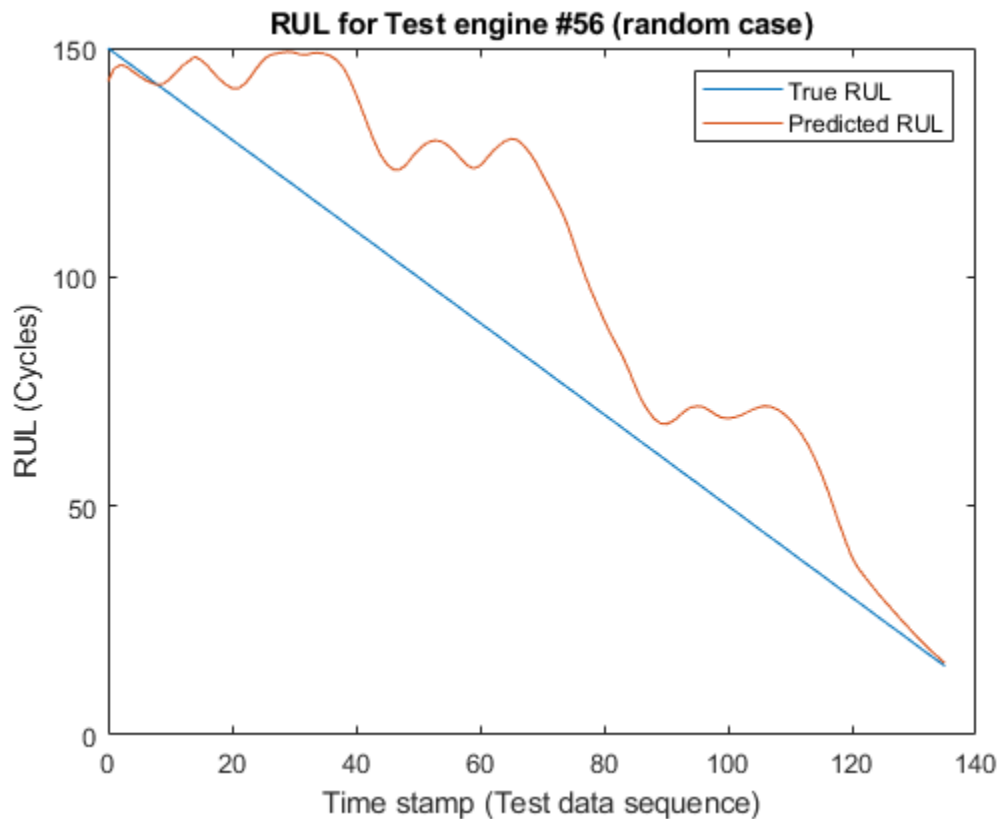

The result shows that the CNN deep learning architecture for estimating RUL of the turbo engine data is a viable approach to predict RUL. The RMSE values at all timestamps indicates that the network can perform well towards the end of the given test sequence data. Therefore, having a brief history of the sensor data is important when trying to predict RUL.

Helper Functions

Load Data Function

This function loads run-to-failure data from the provided text file and groups time-series data and its corresponding RUL values in a table as predictors and responses.

```
function data = localLoadData(filenamePredictors,varargin)

if isempty(varargin)
    filenameResponses = [];
else
    filenameResponses = varargin{:};
end

%% Load the text file as a table
rawData = readtable(filenamePredictors);

% Add variable names to the table
```

```matlab
VarNames = {...
    'id', 'timeStamp', 'op_setting_1', 'op_setting_2', 'op_setting_3', ...
    'sensor_1', 'sensor_2', 'sensor_3', 'sensor_4', 'sensor_5', ...
    'sensor_6', 'sensor_7', 'sensor_8', 'sensor_9', 'sensor_10', ...
    'sensor_11', 'sensor_12', 'sensor_13', 'sensor_14', 'sensor_15', ...
    'sensor_16', 'sensor_17', 'sensor_18', 'sensor_19', 'sensor_20', ...
    'sensor_21'};
rawData.Properties.VariableNames = VarNames;

if ~isempty(filenameResponses)
    RULTest = readmatrix(filenameResponses);
end

% Split the signals for each unit ID
IDs = rawData{:,1};
nID = unique(IDs);
numObservations = numel(nID);

% Initialize a table for storing data
data = table(Size=[numObservations 2],...
    VariableTypes={'cell','cell'},...
    VariableNames={'X','Y'});

for i=1:numObservations
    idx = IDs == nID(i);
    data.X{i} = rawData(idx,:);
    if isempty(filenameResponses)
        % Calculate RUL from time column for train data
        data.Y{i} = flipud(rawData.timeStamp(idx))-1;
    else
        % Use RUL values from filenameResponses for test data
        sequenceLength = sum(idx);
        endRUL = RULTest(i);
        data.Y{i} = [endRUL+sequenceLength-1:-1:endRUL]'; %#ok<NBRAK>
    end
end
end
```

Lambda Plot function

This helper function accepts the `predictions` table and a `lambdaCase` argument, and plots the predicted RUL against the true RUL throughout its sequence (at every timestamp) for a visualization of how the prediction changes with every timestamp. The second argument, `lambdaCase`, can be the test engine number or one of a set of valid strings to find an engine number : "random", "best", "worst", or "average".

```matlab
function localLambdaPlot(predictions,lambdaCase)

if isnumeric(lambdaCase)
    idx = lambdaCase;
else
    switch lambdaCase
        case {"Random","random","r"}
            idx = randperm(height(predictions),1); % Randomly choose a test case to plot
        case {"Best","best","b"}
            idx = find(predictions.RMSE == min(predictions.RMSE)); % Best case
        case {"Worst","worst","w"}
            idx = find(predictions.RMSE == max(predictions.RMSE)); % Worst case
```

```
        case {"Average","average","a"}
            err = abs(predictions.RMSE-mean(predictions.RMSE));
            idx = find(err==min(err),1);
    end
end
y = predictions.Y{idx};
yPred = predictions.YPred{idx};
x = 0:numel(y)-1;
plot(x,y,x,yPred)
legend("True RUL","Predicted RUL")
xlabel("Time stamp (Test data sequence)")
ylabel("RUL (Cycles)")

title("RUL for Test engine #"+idx+ " ("+lambdaCase+" case)")
end
```

References

1   Li, Xiang, Qian Ding, and Jian-Qiao Sun. "Remaining Useful Life Estimation in Prognostics Using Deep Convolution Neural Networks." Reliability Engineering & System Safety 172 (April 2018): 1–11. https://doi.org/10.1016/j.ress.2017.11.021.

See Also
imageInputLayer | prognosability | trainingOptions

Related Examples

- "Learn About Convolutional Neural Networks" (Deep Learning Toolbox)
- "Sequence-to-Sequence Regression Using Deep Learning" (Deep Learning Toolbox)
- "Similarity-Based Remaining Useful Life Estimation" on page 5-15
- "Battery Cycle Life Prediction Using Deep Learning" on page 5-123

External Websites

- Predictive Maintenance, Part 3: Remaining Useful Life Estimation
- Convolutional Neural Network - 3 things you need to know

Rolling Element Bearing Fault Diagnosis Using Deep Learning

This example shows how to perform fault diagnosis of a rolling element bearing using a deep learning approach. The example demonstrates how to classify bearing faults by converting 1-D bearing vibration signals to 2-D images of scalograms and applying transfer learning using a pretrained network. Transfer learning significantly reduces the time spent on feature extraction and feature selection in conventional bearing diagnostic approaches, and provides good accuracy for the small MFPT data set used in this example.

To run this example, go to https://github.com/mathworks/RollingElementBearingFaultDiagnosis-Data, download the entire repository as a ZIP file, and save it in the same directory as the live script.

Rolling Element Bearing Faults

Localized faults in a rolling element bearing can 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] on page 4-137. The following figure shows a rolling element striking a local fault at the inner race. A common problem is detecting and identifying these faults.
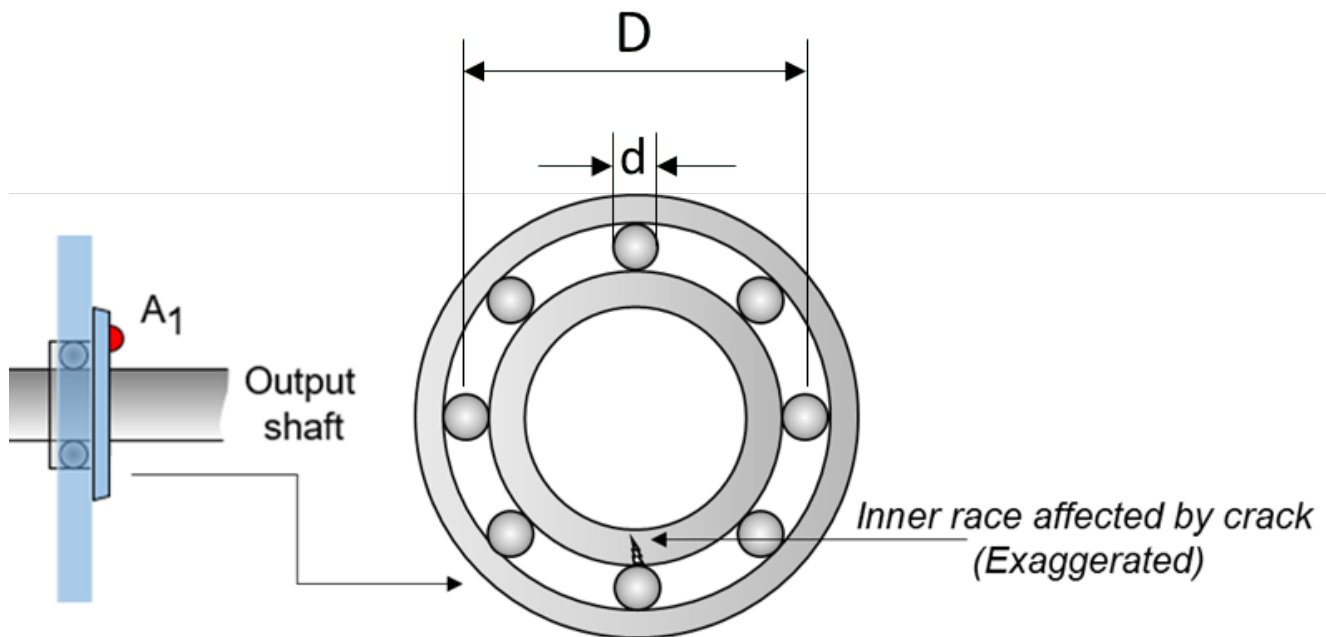
Ball bearing cross-section (Magnified)

Machinery Failure Prevention Technology (MFPT) Challenge Data

MFPT Challenge data [2] on page 4-137 contains 23 data sets collected from machines under various fault conditions. The first 20 data sets are collected from a bearing test rig, with three under good conditions, three with outer race faults under constant load, seven with outer race faults under various loads, and seven with inner race faults under various loads. The remaining three 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, you use only the data collected from the test rig with known conditions.

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: ball pass frequency outer race (BPFO), ball pass frequency inner race (BPFI), fundamental train frequency (FTF), and ball spin frequency (BSF). The formulas for BPFO and BPFI are as follows [1] on page 4-137.

- BPFO:

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

- BPFI:

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

As shown in the figure, d is the ball diameter and D is the pitch diameter. The variable f_r is the shaft speed, n is the number of rolling elements, and ϕ is the bearing contact angle [1] on page 4-137.

Scalogram of Bearing Data

To benefit from pretrained CNN deep networks, use the `plotBearingSignalAndScalogram` helper function to convert 1-D vibration signals in the MFPT dataset to 2-D scalograms. A scalogram is a time-frequency domain representation of the original time-domain signal [3] on page 4-137. The two dimensions in a scalogram image represent time and frequency. To visualize the relationship between a scalogram and its original vibration signal, plot the vibration signal with an inner race fault against its scalogram.

```
% Import data with inner race fault
data_inner = load(fullfile(matlabroot, 'toolbox', 'predmaint', ...
    'predmaintdemos', 'bearingFaultDiagnosis', ...
    'train_data', 'InnerRaceFault_vload_1.mat'));
% Plot bearing signal and scalogram
plotBearingSignalAndScalogram(data_inner)
```
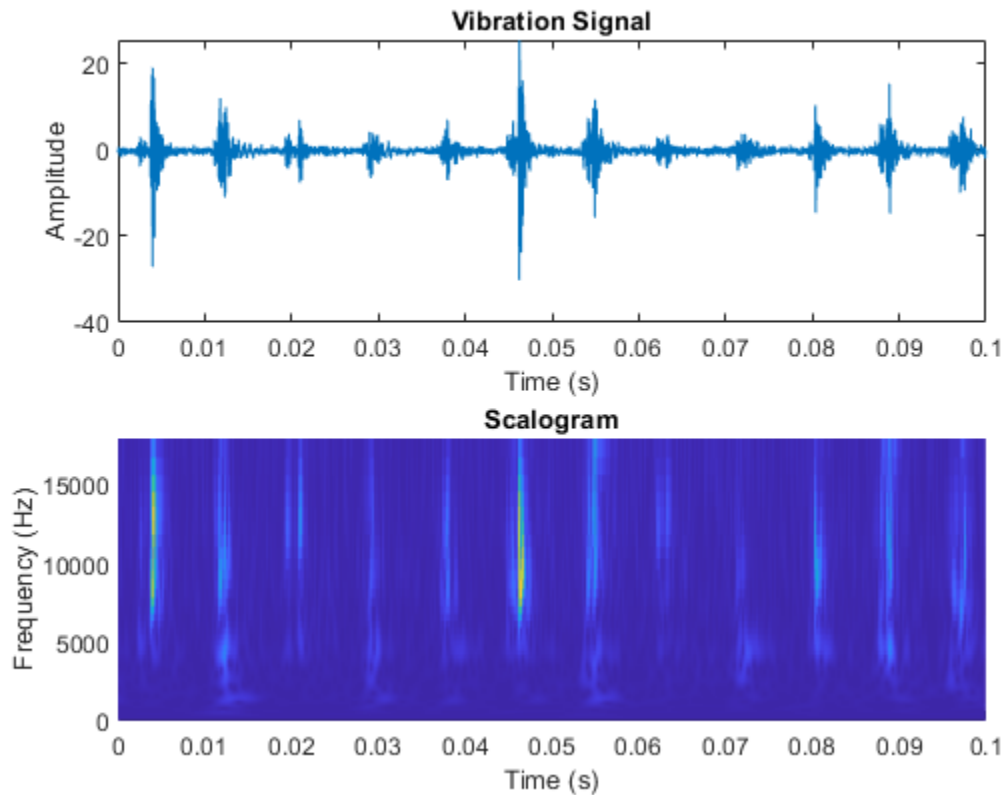
During the 0.1 seconds shown in the plot, the vibration signal contains 12 impulses because the tested bearing's BPFI is 118.875 Hz. Accordingly, the scalogram shows 12 distinct peaks that align with the impulses in the vibration signal. Next, visualize scalograms for the outer race fault.

```
% Import data with outer race fault
data_outer = load(fullfile(matlabroot, 'toolbox', 'predmaint', ...
    'predmaintdemos', 'bearingFaultDiagnosis', ...
    'test_data', 'OuterRaceFault_3.mat'));
% Plot original signal and its scalogram
plotBearingSignalAndScalogram(data_outer)
```
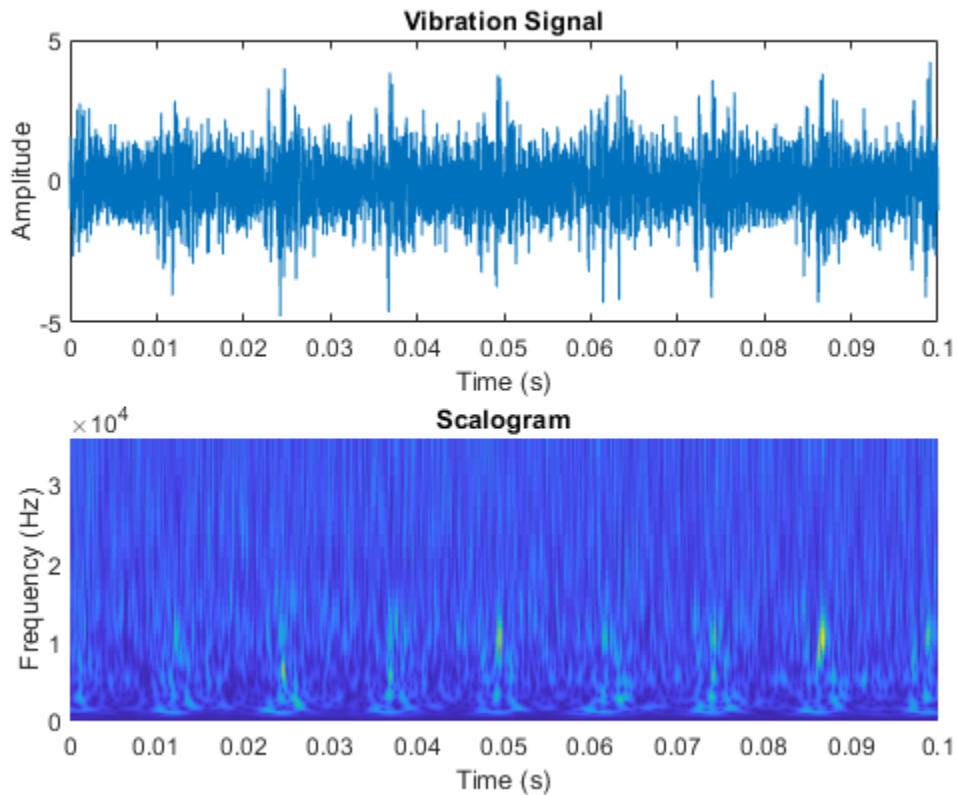
The scalogram of the outer race fault shows 8 distinct peaks during the first 0.1 seconds, which is consistent with the ballpass frequencies. Because the impulses in the time-domain signal is not as dominant as in the inner race fault case, the distinct peaks in the scalogram show less contrast with the background. The scalogram of the normal condition does not show dominant distinct peaks.

```
% Import normal bearing data
data_normal = load(fullfile(matlabroot, 'toolbox', 'predmaint', ...
    'predmaintdemos', 'bearingFaultDiagnosis', ...
    'train_data', 'baseline_1.mat'));
% Plot original signal and its scalogram
plotBearingSignalAndScalogram(data_normal)
```
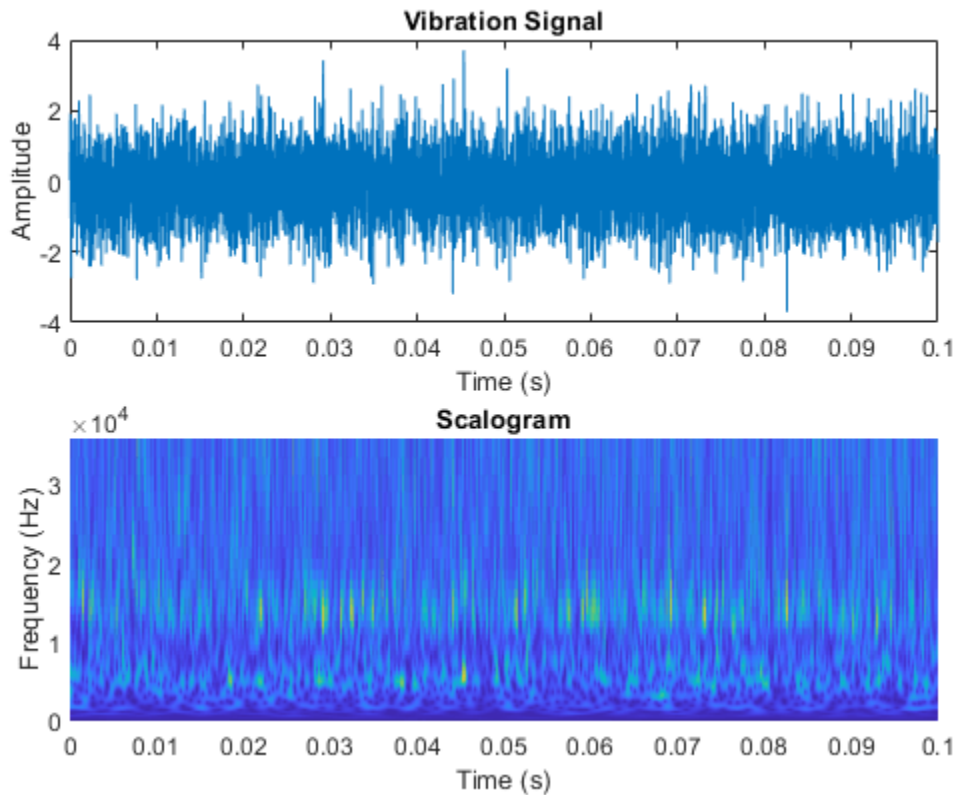
The number of distinct peaks is a good feature to differentiate between inner race faults, outer race faults, and normal conditions. Therefore, a scalogram can be a good candidate for classifying bearing faults. In this example, all bearing signal measurements come from tests using the same shaft speed. To apply this example to bearing signals under different shaft speeds, the data needs to be normalized by shaft speed. Otherwise, the number of "pillars" in the scalogram will be wrong.

Prepare Training Data

Unzip the downloaded file.

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

The downloaded dataset contains a training dataset with 14 MAT-files (2 normal, 5 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`, the file ensemble datastore can 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 the file ensemble datastore returns the vibration signal `gs` inside of the `bearing` data structure.

```
fileLocation = fullfile('.', 'RollingElementBearingFaultDiagnosis-Data-master', 'train_data');
fileExtension = '.mat';
ensembleTrain = fileEnsembleDatastore(fileLocation, fileExtension);
```

```
ensembleTrain.ReadFcn = @readMFPTBearing;
ensembleTrain.DataVariables = ["gs", "sr", "rate", "load", "BPFO", "BPFI", "FTF", "BSF"];
ensembleTrain.ConditionVariables = ["Label", "FileName"];
ensembleTrain.SelectedVariables = ["gs", "sr", "rate", "load", "BPFO", "BPFI", "FTF", "BSF", "Lab
```

```
ensembleTrain =
  fileEnsembleDatastore with properties:

                 ReadFcn: @readMFPTBearing
         WriteToMemberFcn: []
           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]
```

Now, convert the 1-D vibration signals to scalograms and save the images for training. The size of each scalogram is 227-by-227-by-3, which is the same input size required by SqueezeNet. To improve accuracy, the helper function `convertSignalToScalogram` envelops the raw signal and divides it into multiple segments. After running the following commands, a folder named "train_image" appears in the current folder. All scalogram images of the bearing signals in the "RollingElementBearingFaultDiagnosis-Data-master/train_data" folder are saved in the "train_image" folder.

```
reset(ensembleTrain)
while hasdata(ensembleTrain)
  folderName = 'train_image';
  convertSignalToScalogram(ensembleTrain,folderName);
end
```

Create an image datastore and split the training data into training and validation data sets, using 80% of the images from the "train_image" folder for training and 20% for validation.

```
% Create image datastore to store all training images
path = fullfile('.', folderName);
imds = imageDatastore(path, ...
  'IncludeSubfolders',true,'LabelSource','foldernames');
% Use 20% training data as validation set
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.8,'randomize');
```

Train Network with Transfer Learning

Next, fine-tune the pretrained SqueezeNet convolutional neural network to perform classification on the scalograms. SqueezeNet has been trained on over a million images and has learned rich feature representations. Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point for a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features using a smaller number of training images. Load and view the SqueezeNet network:

```
net = squeezenet
```

```
net =
  DAGNetwork with properties:
```

```
      Layers: [68×1 nnet.cnn.layer.Layer]
  Connections: [75×2 table]
   InputNames: {'data'}
  OutputNames: {'ClassificationLayer_predictions'}
```

```
analyzeNetwork(net)
```

SqueezeNet uses the convolutional layer 'conv10' to extract image features and the classification layer 'ClassificationLayer_predictions' to classify the input image. These two layers contain information to combine the features that the network extracts into class probabilities, a loss value, and predicted labels. To retrain SqueezeNet for classifying new images, the convolutional layers 'conv10' and the classification layer 'ClassificationLayer_predictions' need to be replaced with new layers adapted to the bearing images.

Extract the layer graph from the trained network.

```
lgraph = layerGraph(net);
```

In most networks, the last layer with learnable weights is a fully connected layer. In some networks, such as SqueezeNet, the last learnable layer is a 1-by-1 convolutional layer instead. In this case, replace the convolutional layer with a new convolutional layer with a number of filters equal to the number of classes.

```
numClasses = numel(categories(imdsTrain.Labels));
```

```
newConvLayer = convolution2dLayer([1, 1],numClasses,'WeightLearnRateFactor',10,'BiasLearnRateFac
lgraph = replaceLayer(lgraph,'conv10',newConvLayer);
```

The classification layer specifies the output classes of the network. Replace the classification layer with a new one without class labels. trainNetwork automatically sets the output classes of the layer at training time.

```
newClassificationLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'ClassificationLayer_predictions',newClassificationLayer);
```

Specify the training options. To slow down learning in the transferred layers, set the initial learning rate to a small value. When you create the convolutional layer, you include larger learning rate factors to speed up learning in the new final layers. This combination of learning rate settings results in fast learning only in the new layers and slower learning in the other layers. When performing transfer learning, you do not need to train for as many epochs. An epoch is a full training cycle on the entire training data set. The software validates the network every ValidationFrequency iterations during training.

```
options = trainingOptions('sgdm', ...
  'InitialLearnRate',0.0001, ...
  'MaxEpochs',4, ...
  'Shuffle','every-epoch', ...
  'ValidationData',imdsValidation, ...
  'ValidationFrequency',30, ...
  'Verbose',false, ...
  'MiniBatchSize',20, ...
  'Plots','training-progress');
```

Train the network that consists of the transferred and new layers. By default, trainNetwork uses a GPU if you have Parallel Computing Toolbox™ and a supported GPU device. For information on

supported devices, see "GPU Computing Requirements" (Parallel Computing Toolbox). Otherwise, `trainNetwork` uses a CPU. You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value argument of `trainingOptions`.

```
net = trainNetwork(imdsTrain,lgraph,options);
```

Validate Using Test Data Sets

Use bearing signals in the "RollingElementBearingFaultDiagnosis-Data-master/test_data" folder to validate the accuracy of the trained network. The test data needs to be processed in the same way as the training data.

Create a file ensemble datastore to store the bearing vibration signals in the test folder.

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

Convert 1-D signals to 2-D scalograms.

```
reset(ensembleTest)
while hasdata(ensembleTest)
  folderName = 'test_image';
  convertSignalToScalogram(ensembleTest,folderName);
end
```

Create an image datastore to store the test images.

```
path = fullfile('.','test_image');
imdsTest = imageDatastore(path, ...
  'IncludeSubfolders',true,'LabelSource','foldernames');
```

Classify the test image datastore with the trained network.

```
YPred = classify(net,imdsTest,'MiniBatchSize',20);
```

Compute the accuracy of the prediction.

```
YTest = imdsTest.Labels;
accuracy = sum(YPred == YTest)/numel(YTest)
```

```
accuracy = 0.9957
```

Plot a confusion matrix.

```
figure
confusionchart(YTest,YPred)
```

When you train the network multiple times, you might see some variation in accuracy between trainings, but the average accuracy should be around 98%. Even though the training set is quite small, this example benefits from transfer learning and achieves good accuracy.

Conclusion

This example demonstrates that deep learning can be an effective tool to identify different types of faults in rolling element bearing, even when the data size is relatively small. A deep learning approach reduces the time that conventional approach requires for feature engineering. For comparison, see the example "Rolling Element Bearing Fault Diagnosis" on page 4-6.

References

[1] Randall, Robert B., and Jérôme Antoni. "Rolling Element Bearing Diagnostics—A Tutorial." Mechanical Systems and Signal Processing 25, no. 2 (February 2011): 485–520. https://doi.org/10.1016/j.ymssp.2010.07.017.

[2] Bechhoefer, Eric. "Condition Based Maintenance Fault Database for Testing Diagnostics and Prognostic Algorithms." 2013. https://www.mfpt.org/fault-data-sets/.

[3] Verstraete, David, Andrés Ferrada, Enrique López Droguett, Viviana Meruane, and Mohammad Modarres. "Deep Learning Enabled Fault Diagnosis Using Time-Frequency Image Analysis of Rolling Element Bearings." Shock and Vibration 2017 (2017): 1–17. https://doi.org/10.1155/2017/5067651.

Helper Functions

```matlab
function plotBearingSignalAndScalogram(data)
% Convert 1-D bearing signals to scalograms through wavelet transform
fs = data.bearing.sr;
t_total = 0.1; % seconds
n = round(t_total*fs);
bearing = data.bearing.gs(1:n);
[cfs,frq] = cwt(bearing,'amor', fs);

% Plot the original signal and its scalogram
figure
subplot(2,1,1)
plot(0:1/fs:(n-1)/fs,bearing)
xlim([0,0.1])
title('Vibration Signal')
xlabel('Time (s)')
ylabel('Amplitude')
subplot(2,1,2)
surface(0:1/fs:(n-1)/fs,frq,abs(cfs))
shading flat
xlim([0,0.1])
ylim([0,max(frq)])
title('Scalogram')
xlabel('Time (s)')
ylabel('Frequency (Hz)')
end

function convertSignalToScalogram(ensemble,folderName)
% Convert 1-D signals to scalograms and save scalograms as images
data = read(ensemble);
fs = data.sr;
x = data.gs{:};
label = char(data.Label);
fname = char(data.FileName);
ratio = 5000/97656;
interval = ratio*fs;
N = floor(numel(x)/interval);

% Create folder to save images
path = fullfile('.',folderName,label);
if ~exist(path,'dir')
  mkdir(path);
end

for idx = 1:N
  sig = envelope(x(interval*(idx-1)+1:interval*idx));
  cfs = cwt(sig,'amor', seconds(1/fs));
  cfs = abs(cfs);
  img = ind2rgb(round(rescale(flip(cfs),0,255)),jet(320));
  outfname = fullfile('.',path,[fname '-' num2str(idx) '.jpg']);
  imwrite(imresize(img,[227,227]),outfname);
```

```
end
end
```

See Also

trainingOptions | trainNetwork | squeezenet | analyzeNetwork | layerGraph | convolution2dLayer | classificationLayer | replaceLayer | classify | confusionchart

Related Examples

- "Rolling Element Bearing Fault Diagnosis" on page 4-6

External Websites

- Convolutional Neural Network - 3 things you need to know

Anomaly Detection in Industrial Machinery Using Three-Axis Vibration Data

This example shows how to detect anomalies in vibration data using machine learning and deep learning. The example uses vibration data from an industrial machine. First, you extract features from the raw measurements corresponding to normal operation using the Diagnostic Feature Designer App. You use the selected features to train three different models (one-class SVM, isolation forest, and LSTM autoencoder) for anomaly detection. Then, you use each trained model to identify whether the machine is operating normally.

Data Set

The data set contains three-axis vibration measurements from an industrial machine. The data is collected both immediately before and after a scheduled maintenance. The data collected after scheduled maintenance is assumed to represent normal operating conditions of the machine. The data from before maintenance can represent either normal or anomalous conditions. Data for each axis is stored in a separate column. Save and unzip the data set and then, load the training data.

```
url = 'https://ssd.mathworks.com/supportfiles/predmaint/anomalyDetection3axisVibration/v1/vibrat:
websave('vibrationData.zip',url);
unzip('vibrationData.zip');
load("MachineData.mat")
trainData
```

```
trainData=40×4 table
        ch1                  ch2                  ch3              label

   _____      _____      _____      _____

   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
   {70000×1 double}     {70000×1 double}     {70000×1 double}     Before
      ⋮
```

To better understand the data, visualize it before and after maintenance. Plot vibration data for the fourth member of the ensemble and note that the data for the two conditions looks different.

```
ensMember = 4;
helperPlotVibrationData(trainData, ensMember)
```

Extract Features with Diagnostic Feature Designer App

Because raw data can be correlated and noisy, using raw data for training machine learning models is not very efficient. The Diagnostic Feature Designer app lets you interactively explore and preprocess your data, extract time and frequency domain features, and then rank the features to determine which are most effective for diagnosing faulty or otherwise anomalous systems. You can then export a function to extract the selected features from your data set programmatically. Open **Diagnostic Feature Designer** by typing `diagnosticFeatureDesigner` at the command prompt. For a tutorial on using **Diagnostic Feature Designer**, see "Identify Condition Indicators for Predictive Maintenance Algorithm Design".

Click the **New Session** button, select `trainData` as the source, and then set `label` as **Condition Variable**. The `label` variable identifies the condition of the machine for the corresponding data.

You can use **Diagnostic Feature Designer** to iterate on the features and rank them. The app creates a histogram view for all generated features to visualize the distribution for each label. For example, the following histograms show distributions of various features extracted from ch1. These histograms are derived from a much larger data set than the data set that you use in this example, in order to better illustrate the label-group separation. Because you are using a smaller data set, your results will look different.
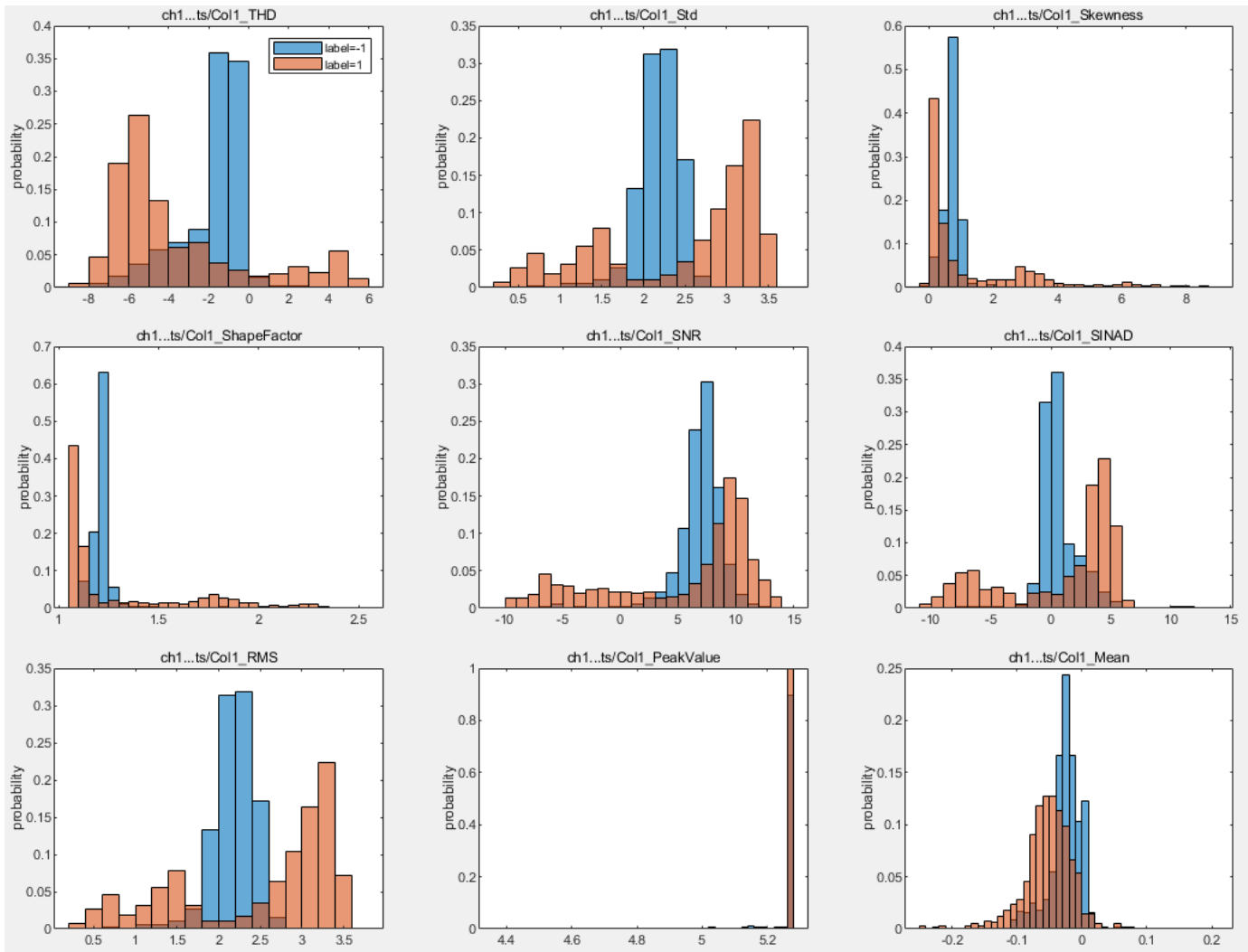
Use the top four ranked features for each channel.

- ch1 : Crest Factor, Kurtosis, RMS, Std
- ch2 : Mean, RMS, Skewness, Std
- ch3 : Crest Factor, SINAD, SNR, THD

Export a function to generate the features from the Diagnostic Feature designer app and save it with the name `generateFeatures`. This function extracts the top 4 relevant features from each channel in the entire data set from the command line.

```
trainFeatures = generateFeatures(trainData);
head(trainFeatures)
```

label	ch1_stats/Col1_CrestFactor	ch1_stats/Col1_Kurtosis	ch1_stats/Col1_RMS	ch1_
Before	2.2811	1.8087	2.3074	
Before	2.3276	1.8379	2.2613	
Before	2.3276	1.8626	2.2613	

```
Before              2.8781                  2.1986                  1.8288
Before              2.8911                   2.06                   1.8205
Before              2.8979                  2.1204                  1.8163
Before              2.9494                   1.92                   1.7846
Before              2.5106                  1.6774                  1.7513
```

Prepare Full Data Sets for Training

The data set you use to this point is only a small subset of a much larger data set to illustrate the process of feature extraction and selection. Training your algorithm on all available data yields the best performance. To this end, load the same 12 features as previously extracted from the larger data set of 17,642 signals.

```
load("FeatureEntire.mat")
head(featureAll)
```

label	ch1_stats/Col1_CrestFactor	ch1_stats/Col1_Kurtosis	ch1_stats/Col1_RMS	ch1
Before	2.3683	1.927	2.2225	
Before	2.402	1.9206	2.1807	
Before	2.4157	1.9523	2.1789	
Before	2.4595	1.8205	2.14	
Before	2.2502	1.8609	2.3391	
Before	2.4211	2.2479	2.1286	
Before	3.3111	4.0304	1.5896	
Before	2.2655	2.0656	2.3233	

Use `cvpartition` to partition data into a training set and an independent test set. Use the `helperExtractLabeledData` helper function to find all features corresponding to the label `'After'` in the `featureTrain` variable.

```
rng(0) % set for reproducibility
idx = cvpartition(featureAll.label, 'holdout', 0.1);
featureTrain = featureAll(idx.training, :);
featureTest = featureAll(idx.test, :);
```

For each model, train on only the after maintenance data, which is assumed to be normal. Extract only this data from `featureTrain`.

```
trueAnomaliesTest = featureTest.label;
featureNormal = featureTrain(featureTrain.label=='After', :);
```

Detect Anomalies with One-Class SVM

Support Vector Machines are powerful classifiers, and the variant that trains on only the normal data is used here.. This model works well for identifying abnormalities that are "far" from the normal data. Train a one-class SVM model using the `fitcsvm` function and the data for normal conditions.

```
mdlSVM = fitcsvm(featureNormal, 'label', 'Standardize', true, 'OutlierFraction', 0);
```

Validate the trained SVM model by using test data, which contains both normal and anomalous data.

```
featureTestNoLabels = featureTest(:, 2:end);
[~,scoreSVM] = predict(mdlSVM,featureTestNoLabels);
isanomalySVM = scoreSVM<0;
predSVM = categorical(isanomalySVM, [1, 0], ["Anomaly", "Normal"]);
trueAnomaliesTest = renamecats(trueAnomaliesTest,["After","Before"], ["Normal","Anomaly"]);
```

```
figure;
confusionchart(trueAnomaliesTest, predSVM, Title="Anomaly Detection with One-class SVM", Normaliz
```
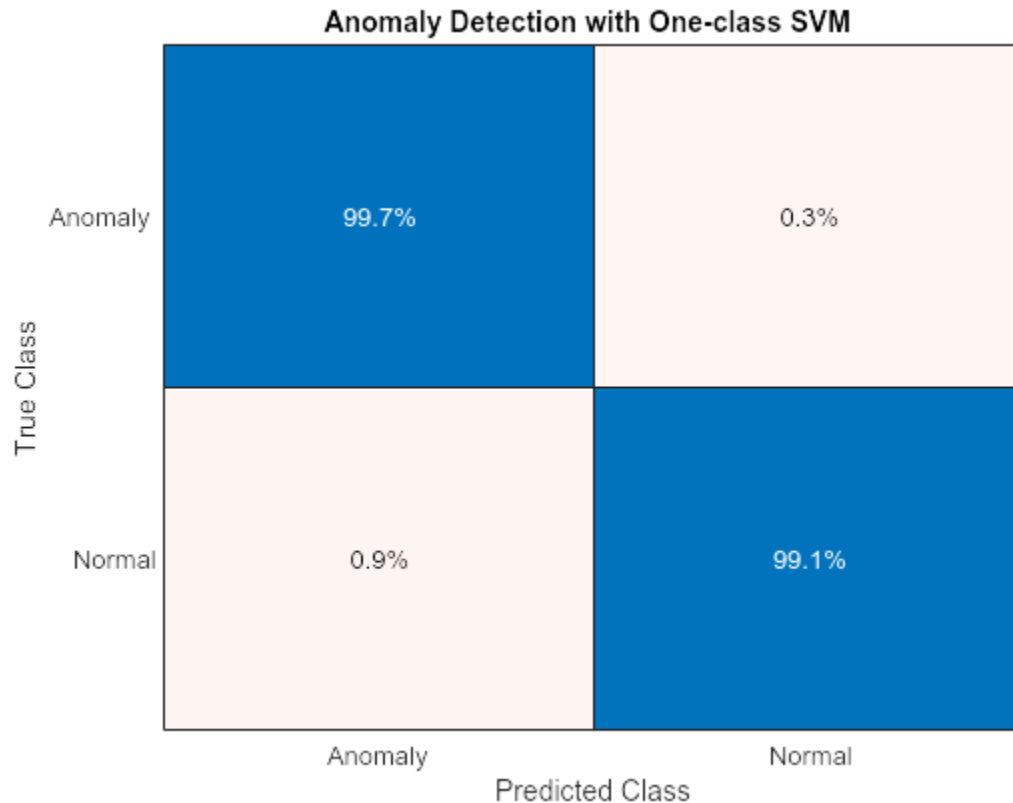

From the confusion matrix, you can see that the one-class SVM performs well. Only 0.3% of anomalous samples are misclassified as normal and about 0.9% of normal data is misclassified as anomalous.

Detect Anomalies with Isolation Forest

The decision trees of an isolation forest isolate each observation in a leaf. How many decisions a sample passes through to get to its leaf is a measure of how difficult isolating it from the others is. The average depth of trees for a specific sample is used as their anomaly score and returned by `iforest`.

Train the isolation forest model on normal data only.

```
[mdlIF,~,scoreTrainIF] = iforest(featureNormal{:,2:13},'ContaminationFraction',0.09);
```

Validate the trained isolation forest model by using the test data. Visualize the performance of this model by using a confusion chart.

```
[isanomalyIF,scoreTestIF] = isanomaly(mdlIF,featureTestNoLabels.Variables);
predIF = categorical(isanomalyIF, [1, 0], ["Anomaly", "Normal"]);
figure;
confusionchart(trueAnomaliesTest,predIF,Title="Anomaly Detection with Isolation Forest",Normaliza
```

Anomaly Detection with Isolation Forest
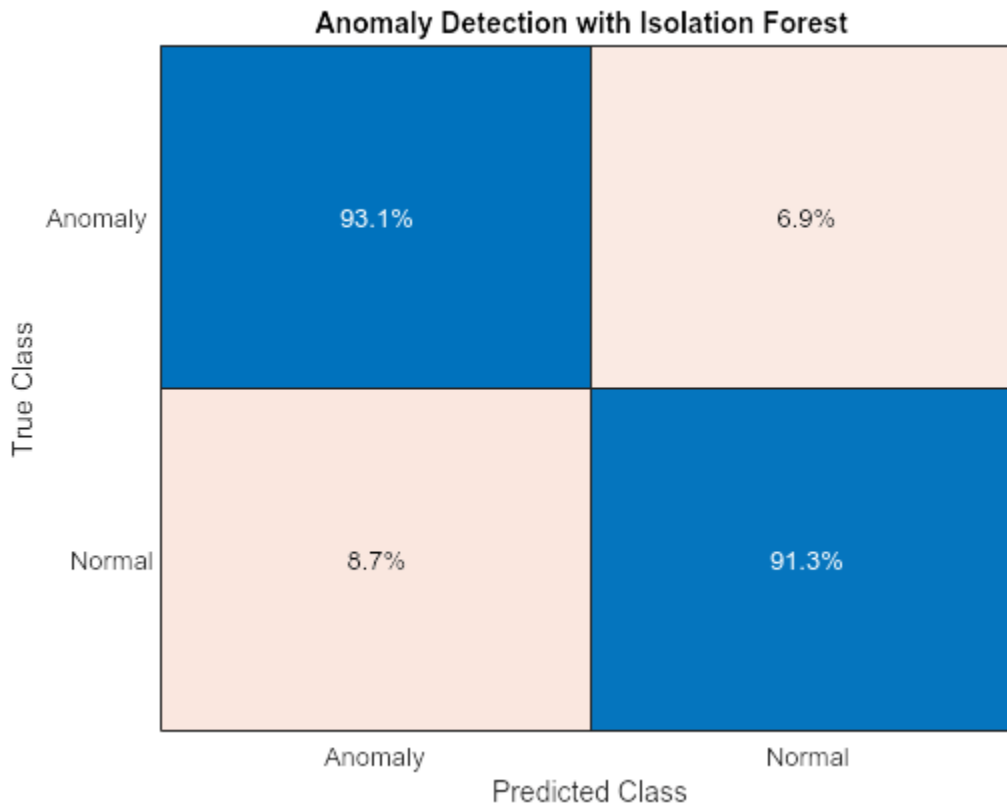
On this data, the isolation forest doesn't do as well as the one-class SVM. The reason for this poorer performance is that the training data contains only normal data while the test data contains about 30% anomalous data. Therefore, the isolation forest model is a better choice when the proportion of anomalous data to normal data is similar for both training and test data.

Detect Anomalies with LSTM Autoencoder Network

Autoencoders are a type of neural network that learn a compressed representation of unlabeled data. LSTM autoencoders are a variant of this network that can learn a compressed representation of sequence data. Here, you train an LSTM autoencoder with only normal data and use this trained network to identify when a signal does not look normal.

Start by extracting features from the after maintenance data.

```
featuresAfter = helperExtractLabeledData(featureTrain, ...
    "After");
```

Construct the LSTM autoencoder network and set the training options.

```
featureDimension = 1;

% Define biLSTM network layers
layers = [ sequenceInputLayer(featureDimension, 'Name', 'in')
    bilstmLayer(16, 'Name', 'bilstm1')
    reluLayer('Name', 'relu1')
    bilstmLayer(32, 'Name', 'bilstm2')
    reluLayer('Name', 'relu2')
```

```
    bilstmLayer(16, 'Name', 'bilstm3')
    reluLayer('Name', 'relu3')
    fullyConnectedLayer(featureDimension, 'Name', 'fc')
    regressionLayer('Name', 'out') ];

% Set Training Options
options = trainingOptions('adam', ...
    'Plots', 'training-progress', ...
    'MiniBatchSize', 500,...
    'MaxEpochs',200);
```

The `MaxEpochs` training options parameter is set to 200. For higher validation accuracy, you can set this parameter to a larger number; However, the network might overfit.

Train the model.

```
net = trainNetwork(featuresAfter, featuresAfter, layers, options);
```

Training on single GPU.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch RMSE	Mini-batch Loss	Base Learning Rate
1	1	00:00:14	5.81	16.9	0.0010
3	50	00:00:18	5.43	14.8	0.0010
5	100	00:00:20	3.99	7.9	0.0010
8	150	00:00:23	4.27	9.1	0.0010
10	200	00:00:25	3.47	6.0	0.0010
13	250	00:00:28	3.97	7.9	0.0010
15	300	00:00:30	3.17	5.0	0.0010
18	350	00:00:33	3.72	6.9	0.0010
20	400	00:00:35	2.89	4.2	0.0010
23	450	00:00:37	3.49	6.1	0.0010
25	500	00:00:40	2.67	3.6	0.0010
28	550	00:00:42	3.31	5.5	0.0010
30	600	00:00:45	2.49	3.1	0.0010
33	650	00:00:47	3.14	4.9	0.0010
35	700	00:00:50	2.29	2.6	0.0010
38	750	00:00:52	2.96	4.4	0.0010
40	800	00:00:55	2.11	2.2	0.0010
43	850	00:00:57	2.82	4.0	0.0010
45	900	00:01:00	1.98	2.0	0.0010
48	950	00:01:02	2.71	3.7	0.0010
50	1000	00:01:05	1.89	1.8	0.0010
53	1050	00:01:07	2.63	3.5	0.0010
55	1100	00:01:10	1.81	1.6	0.0010
58	1150	00:01:12	2.55	3.3	0.0010
60	1200	00:01:15	1.74	1.5	0.0010
63	1250	00:01:17	2.48	3.1	0.0010
65	1300	00:01:20	1.67	1.4	0.0010
68	1350	00:01:22	2.40	2.9	0.0010
70	1400	00:01:25	1.54	1.2	0.0010
73	1450	00:01:27	2.30	2.6	0.0010
75	1500	00:01:29	1.45	1.1	0.0010
78	1550	00:01:32	2.23	2.5	0.0010
80	1600	00:01:34	1.37	0.9	0.0010
83	1650	00:01:37	2.16	2.3	0.0010
85	1700	00:01:39	1.30	0.8	0.0010

```
|        88 |       1750 |   00:01:42 |        2.10 |         2.2 |         0.0010 |
|        90 |       1800 |   00:01:44 |        1.23 |         0.8 |         0.0010 |
|        93 |       1850 |   00:01:47 |        2.04 |         2.1 |         0.0010 |
|        95 |       1900 |   00:01:49 |        1.17 |         0.7 |         0.0010 |
|        98 |       1950 |   00:01:52 |        1.99 |         2.0 |         0.0010 |
|       100 |       2000 |   00:01:54 |        1.11 |         0.6 |         0.0010 |
|       103 |       2050 |   00:01:57 |        1.94 |         1.9 |         0.0010 |
|       105 |       2100 |   00:01:59 |        1.06 |         0.6 |         0.0010 |
|       108 |       2150 |   00:02:02 |        1.90 |         1.8 |         0.0010 |
|       110 |       2200 |   00:02:04 |        1.01 |         0.5 |         0.0010 |
|       113 |       2250 |   00:02:06 |        1.86 |         1.7 |         0.0010 |
|       115 |       2300 |   00:02:09 |        0.97 |         0.5 |         0.0010 |
|       118 |       2350 |   00:02:11 |        1.82 |         1.7 |         0.0010 |
|       120 |       2400 |   00:02:14 |        0.93 |         0.4 |         0.0010 |
|       123 |       2450 |   00:02:16 |        1.79 |         1.6 |         0.0010 |
|       125 |       2500 |   00:02:18 |        0.90 |         0.4 |         0.0010 |
|       128 |       2550 |   00:02:21 |        1.76 |         1.6 |         0.0010 |
|       130 |       2600 |   00:02:23 |        0.87 |         0.4 |         0.0010 |
|       133 |       2650 |   00:02:25 |        1.73 |         1.5 |         0.0010 |
|       135 |       2700 |   00:02:28 |        0.84 |         0.4 |         0.0010 |
|       138 |       2750 |   00:02:30 |        1.71 |         1.5 |         0.0010 |
|       140 |       2800 |   00:02:32 |        0.81 |         0.3 |         0.0010 |
|       143 |       2850 |   00:02:35 |        1.68 |         1.4 |         0.0010 |
|       145 |       2900 |   00:02:37 |        0.78 |         0.3 |         0.0010 |
|       148 |       2950 |   00:02:40 |        1.66 |         1.4 |         0.0010 |
|       150 |       3000 |   00:02:42 |        0.76 |         0.3 |         0.0010 |
|       153 |       3050 |   00:02:44 |        1.63 |         1.3 |         0.0010 |
|       155 |       3100 |   00:02:47 |        0.74 |         0.3 |         0.0010 |
|       158 |       3150 |   00:02:49 |        1.61 |         1.3 |         0.0010 |
|       160 |       3200 |   00:02:52 |        0.72 |         0.3 |         0.0010 |
|       163 |       3250 |   00:02:54 |        1.59 |         1.3 |         0.0010 |
|       165 |       3300 |   00:02:56 |        0.69 |         0.2 |         0.0010 |
|       168 |       3350 |   00:02:59 |        1.57 |         1.2 |         0.0010 |
|       170 |       3400 |   00:03:01 |        0.68 |         0.2 |         0.0010 |
|       173 |       3450 |   00:03:03 |        1.55 |         1.2 |         0.0010 |
|       175 |       3500 |   00:03:06 |        0.66 |         0.2 |         0.0010 |
|       178 |       3550 |   00:03:08 |        1.53 |         1.2 |         0.0010 |
|       180 |       3600 |   00:03:10 |        0.64 |         0.2 |         0.0010 |
|       183 |       3650 |   00:03:13 |        1.51 |         1.1 |         0.0010 |
|       185 |       3700 |   00:03:15 |        0.62 |         0.2 |         0.0010 |
|       188 |       3750 |   00:03:18 |        1.49 |         1.1 |         0.0010 |
|       190 |       3800 |   00:03:20 |        0.61 |         0.2 |         0.0010 |
|       193 |       3850 |   00:03:22 |        1.48 |         1.1 |         0.0010 |
|       195 |       3900 |   00:03:25 |        0.59 |         0.2 |         0.0010 |
|       198 |       3950 |   00:03:27 |        1.46 |         1.1 |         0.0010 |
|       200 |       4000 |   00:03:29 |        0.58 |         0.2 |         0.0010 |
|========================================================================================|
Training finished: Max epochs completed.
```
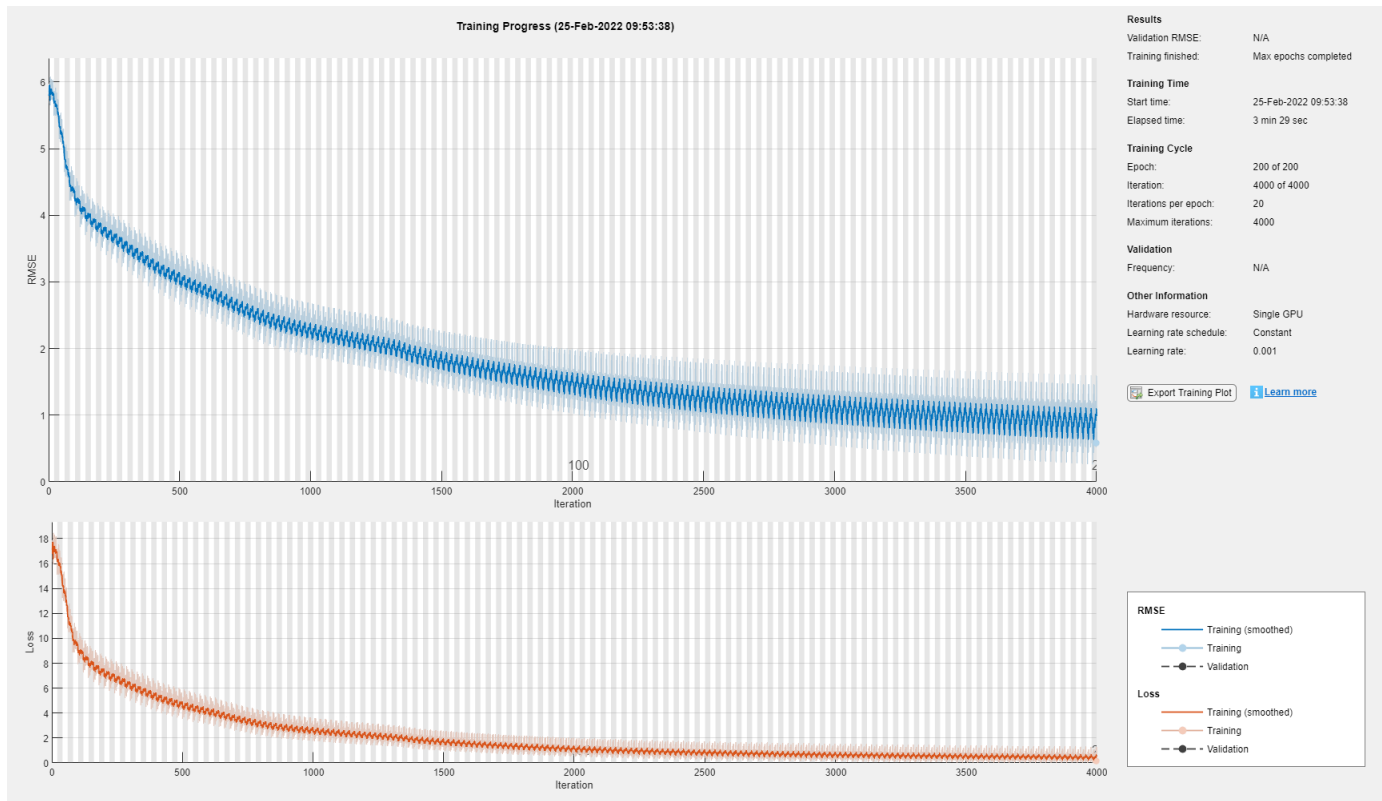
Visualize Model Behavior and Error on Validation Data

Extract and visualize a sample each from Anomalous and Normal condition. The following plots show the reconstruction errors of the autoencoder model for each of the 12 features (indicated on the X-axis). The reconstructed feature value is referred to as "Decoded" signal in the plot. In this sample, features 10, 11, and 12 do not reconstruct well for the anomalous input and thus have high errors. We can use reconstructon errors to identify an anomaly.

```
testNormal = {featureTest(1200, 2:end).Variables};
testAnomaly = {featureTest(200, 2:end).Variables};

% Predict decoded signal for both
decodedNormal = predict(net,testNormal);
decodedAnomaly = predict(net,testAnomaly);

% Visualize
helperVisualizeModelBehavior(testNormal, testAnomaly, decodedNormal, decodedAnomaly)
```
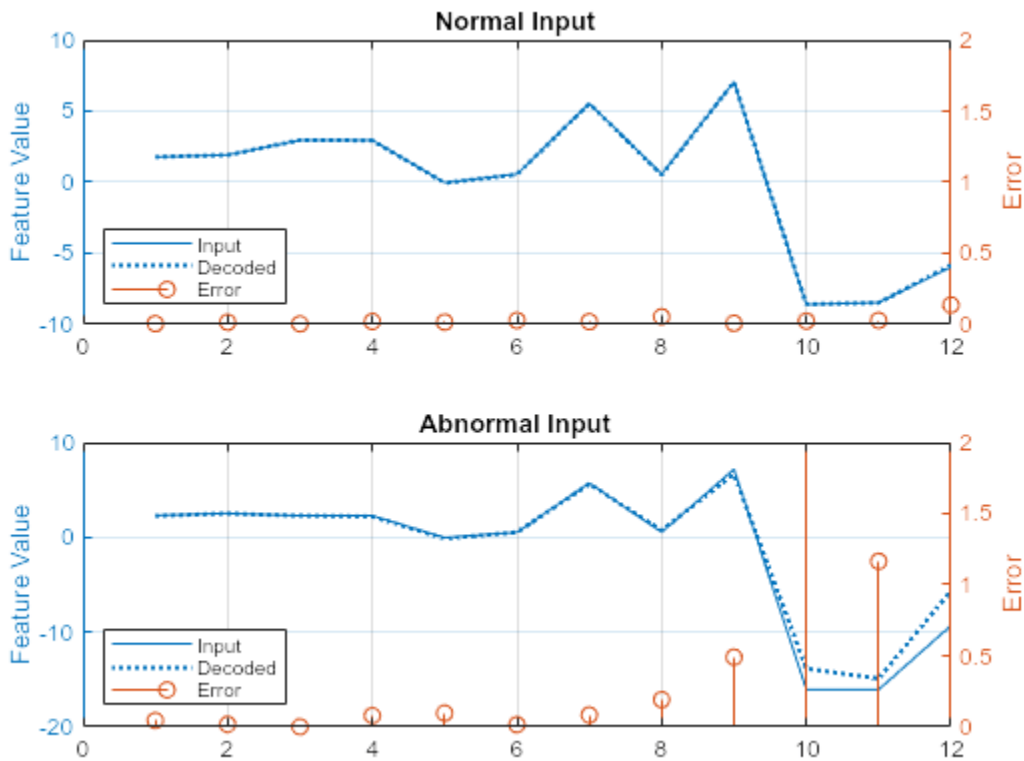
Extract features for all the normal and anomalous data. Use the trained autoencoder model to predict the selected 12 features for both before and after maintenance data. The following plots show the root mean square reconstruction error across the twelve features. The figure shows that the reconstruction error for the anomalous data is much higher than the normal data. This result is expected, since the autoencoder is trained on the normal data, so it better reconstructs similar signals.

```
% Extract data Before maintenance
XTestBefore = helperExtractLabeledData(featureTest, "Before");

% Predict output before maintenance and calculate error
yHatBefore = predict(net, XTestBefore);
errorBefore = helperCalculateError(XTestBefore, yHatBefore);

% Extract data after maintenance
XTestAfter = helperExtractLabeledData(featureTest, "After");

% Predict output after maintenance and calculate error
yHatAfter = predict(net, XTestAfter);
errorAfter = helperCalculateError(XTestAfter, yHatAfter);

helperVisualizeError(errorBefore, errorAfter);
```

Before Maintenance
Mean Error: 4.19

After Maintenance
Mean Error: 0.52
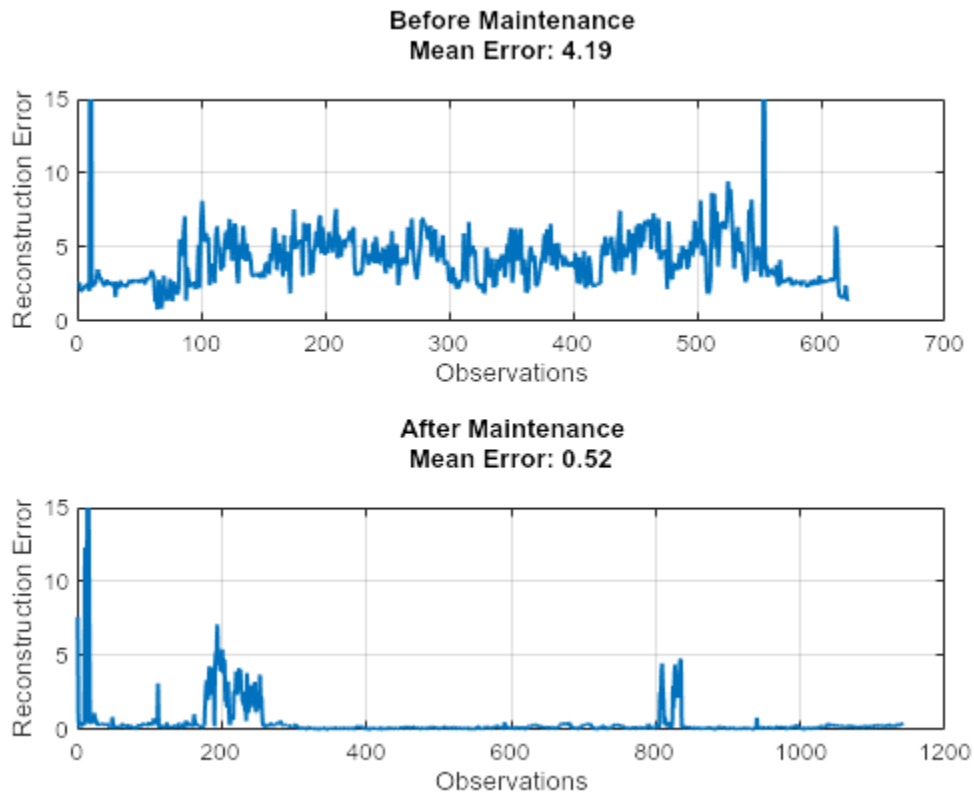
Identify Anomalies

Calculate the reconstruction error on the full validation data.

```
XTestAll = helperExtractLabeledData(featureTest, "All");

yHatAll = predict(net, XTestAll);
errorAll = helperCalculateError(XTestAll, yHatAll);
```

Define an anomaly as a point that has reconstruction error 0.5 times the mean across all observations. This threshold was determined through previous experimentation and can be changed as required.

```
thresh = 0.5;
anomalies = errorAll > thresh*mean(errorAll);

helperVisualizeAnomalies(anomalies, errorAll, featureTest);
```
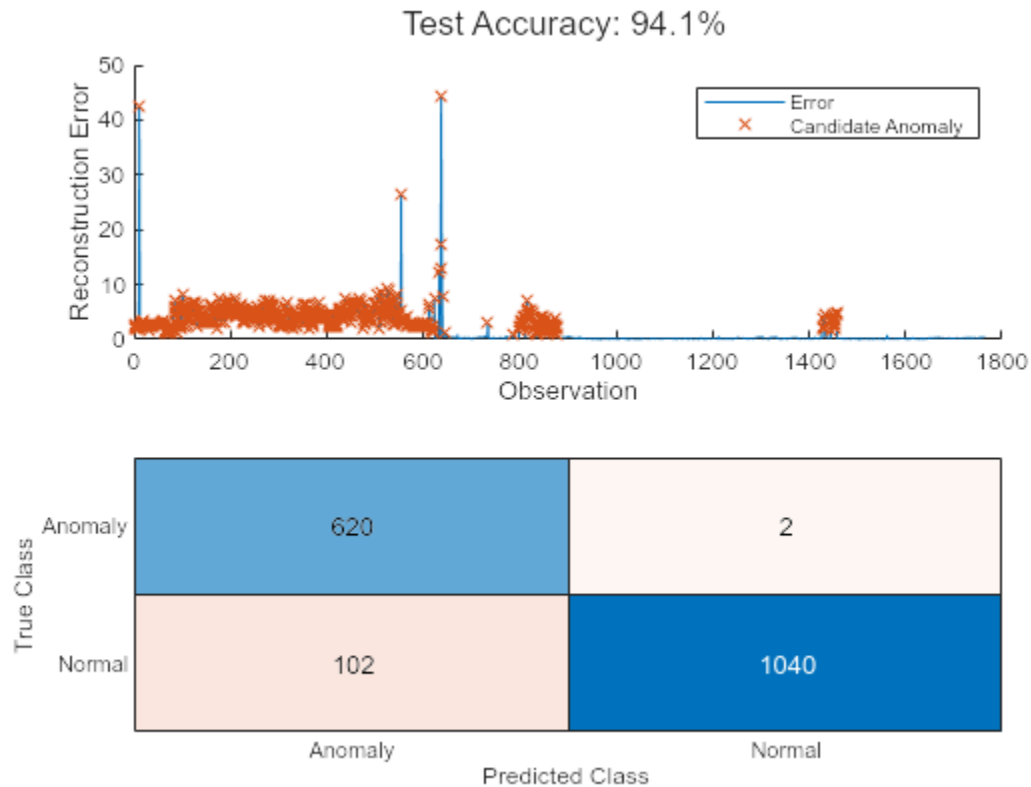
In this example, three different models are used to detect anomalies. The one-class SVM had the best performance at 99.7% for detecting anomalies in the test data, while the other two models are around 93% accurate. The relative performance of the models can change if a different set of features are selected or if different hyper-parameters are used for each model. Use the Diagnostic Feature Designer MATLAB App to further experiment with feature selection.

Supporting Functions

```
function E = helperCalculateError(X, Y)
% HELPERCALCULATEERROR calculates the rms error value between the
% inputs X, Y
E = zeros(length(X),1);
for i = 1:length(X)
    E(i,:) = sqrt(sum((Y{i} - X{i}).^2));
end

end


function helperVisualizeError(errorBefore, errorAfter)
% HELPERVISUALIZEERROR creates a plot to visualize the errors on detecting
% before and after conditions
figure("Color", "W")
tiledlayout("flow")

nexttile
plot(1:length(errorBefore), errorBefore, 'LineWidth',1.5), grid on
title(["Before Maintenance", ...
```

```matlab
    sprintf("Mean Error: %.2f\n", mean(errorBefore))])
xlabel("Observations")
ylabel("Reconstruction Error")
ylim([0 15])

nexttile
plot(1:length(errorAfter), errorAfter, 'LineWidth',1.5), grid on,
title(["After Maintenance", ...
    sprintf("Mean Error: %.2f\n", mean(errorAfter))])
xlabel("Observations")
ylabel("Reconstruction Error")
ylim([0 15])

end

function helperVisualizeAnomalies(anomalies, errorAll, featureTest)
% HELPERVISUALIZEANOMALIES creates a plot of the detected anomalies
anomalyIdx = find(anomalies);
anomalyErr = errorAll(anomalies);

predAE = categorical(anomalies, [1, 0], ["Anomaly", "Normal"]);
trueAE = renamecats(featureTest.label,["Before","After"],["Anomaly","Normal"]);

acc = numel(find(trueAE == predAE))/numel(predAE)*100;
figure;
t = tiledlayout("flow");
title(t, "Test Accuracy: " + round(mean(acc),2) + "%");
nexttile
hold on
plot(errorAll)
plot(anomalyIdx, anomalyErr, 'x')
hold off
ylabel("Reconstruction Error")
xlabel("Observation")
legend("Error", "Candidate Anomaly")

nexttile
confusionchart(trueAE,predAE)

end

function helperVisualizeModelBehavior(normalData, abnormalData, decodedNorm, decodedAbNorm)
%HELPERVISUALIZEMODELBEHAVIOR Visualize model behavior on sample validation data

figure("Color", "W")
tiledlayout("flow")

nexttile()
hold on
colororder('default')
yyaxis left
plot(normalData{:})
plot(decodedNorm{:},":","LineWidth",1.5)
hold off
title("Normal Input")
grid on
ylabel("Feature Value")
yyaxis right
```

```matlab
stem(abs(normalData{:} - decodedNorm{:}))
ylim([0 2])
ylabel("Error")
legend(["Input", "Decoded","Error"],"Location","southwest")

nexttile()
hold on
yyaxis left
plot(abnormalData{:})
plot(decodedAbNorm{:},":","LineWidth",1.5)
hold off
title("Abnormal Input")
grid on
ylabel("Feature Value")
yyaxis right
stem(abs(abnormalData{:} - decodedAbNorm{:}))
ylim([0 2])
ylabel("Error")
legend(["Input", "Decoded","Error"],"Location","southwest")

end

function X = helperExtractLabeledData(featureTable, label)
%HELPEREXTRACTLABELEDDATA Extract data from before or after operating
%conditions and re-format to support input to autoencoder network

% Select data with label After
if label == "All"
    Xtemp = featureTable(:, 2:end).Variables;
else
    tF = featureTable.label == label;
    Xtemp = featureTable(tF, 2:end).Variables;
end

% Arrange data into cells
X = cell(length(Xtemp),1);
for i = 1:length(Xtemp)
    X{i,:} = Xtemp(i,:);
end

end
```

See Also

cvpartition | fitcsvm | iforest | **Diagnostic Feature Designer**

Related Examples

- "Identify Condition Indicators for Predictive Maintenance Algorithm Design"

Broken Rotor Fault Detection in AC Induction Motors Using Vibration and Electrical Signals

This example shows how to detect failures in an AC induction motor. Typically, AC motors are driven by a three-phase power converter or an industrial power supply. Failures in the rotor bars of the motor can present themselves in various electrical and vibration signals collected from the system, including voltages and currents of each of the three phases and the acceleration and velocity signals in radial, axial, and tangential directions. Here, you use some of these signals in the **Diagnostic Feature Designer** app to extract features, which you then use to identify the number of broken rotor bars in the motor.

Data Set

The experimental workbench consists of a three-phase induction motor coupled with a direct-current machine, which works as a generator simulating the load torque, connected by a shaft containing a rotating torque meter. For more details about the workbench setup, see IEEE Broken Rotor Bar Database.

The data set contains electrical and mechanical signals from experiments on three-phase induction motors, using the experimental setup shown in the figure.
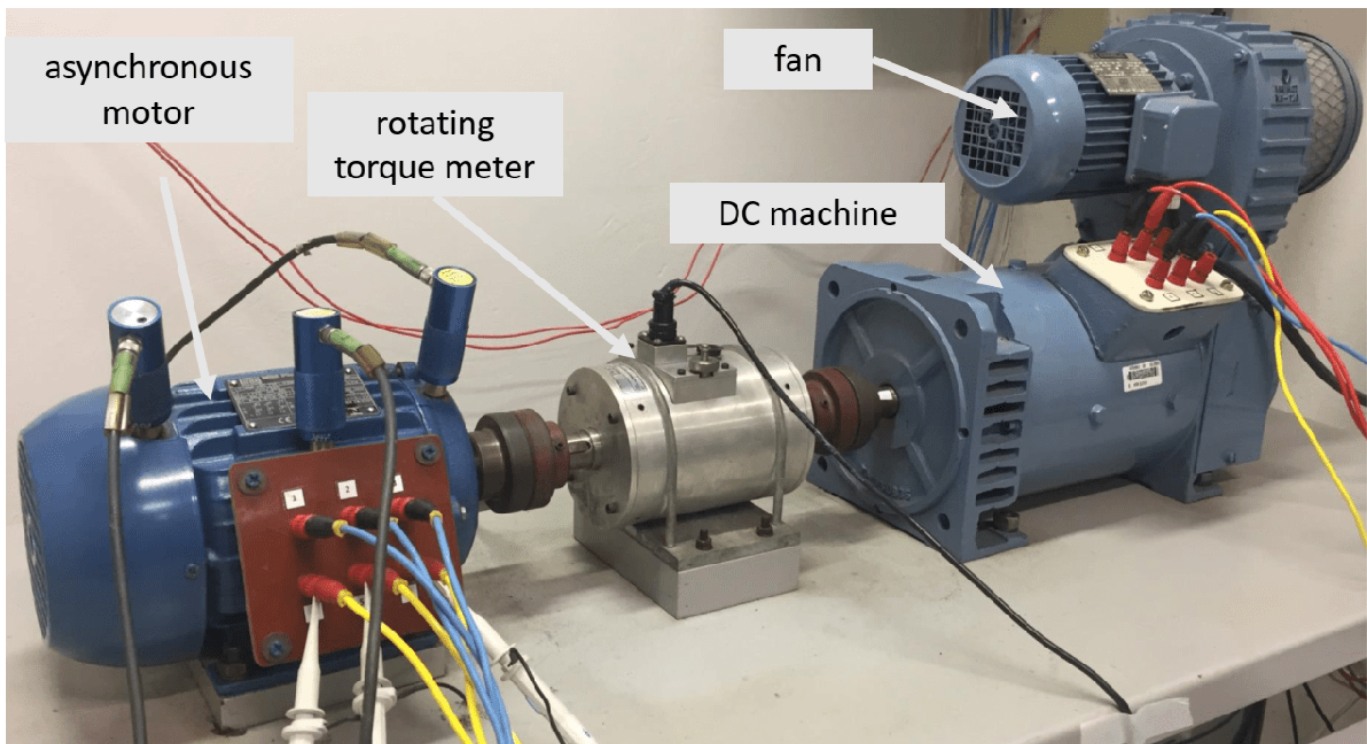
Electrical and vibration signals were collected from the system under the following health and loading conditions:

- 0, 1, 2, 3, and 4 broken rotor bars as the health condition
- 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, and 4.0 Nm torque as the load condition

Ten experiments were performed for each combination of health and loading condition. The following signals were acquired during each experiment:

- Voltage of each phase (V_a, V_b, V_c)

- Current draw of each phase (I_a, I_b, I_c)

- Mechanical vibration speeds tangential to the housing (Vib $_{carc}$) and base (Vib $_{base}$)

- Mechanical vibration speed axial on the driven side (Vib $_{axial}$)

- Mechanical vibration speeds radial on the driven side (Vib $_{acpi}$) and nondriven side (Vib $_{acpe}$).

For the full data set, visit IEEE Broken Rotor Bar Database, download the entire repository as a ZIP file, and save it in the same directory as this Live Script.

```
% Remove this comment when using the original data set.
% filename = "banco de dados experimental.zip";
```

Alternatively, you can use a reduced data set from the MathWorks repository. This data set contains two experiments per operating condition.

```
% Comment out this section when using the original data set.
filename = "experimental_database_short.zip";
if ~exist(filename,'file')
  url = "https://ssd.mathworks.com/supportfiles/predmaint/broken-rotor-bar-fault-data/" + filenam
  websave(filename,url);
end
```

Extract the compressed data files into the current folder.

```
unzip(filename)
```

Extract Ensemble Member Data

Convert the original motor data files into individual ensemble member data files for all combinations of health condition, load torque, and experiment index. The files are saved to a target folder and used by the `fileEnsembleDatastore` object in the **Diagnostic Feature Designer** app for data processing and feature extraction.

You can change the amount of experiments per operating condition to use in the ensemble data. If you use the reduced data set, use two experiments; otherwise, you can use up to 10.

```
numExperiments = 2; % Up to a maximum of 10 if using the original data set
```

The unzipped files of the original data set are associated with different numbers of broken rotor bars. Create bookkeeping variables to associate the original data files with the corresponding number of broken rotor bars.

```
% Names of the original data files restored from the zip archive.
files = [ ...
  "struct_rs_R1.mat", ...
  "struct_r1b_R1.mat", ...
  "struct_r2b_R1.mat", ...
  "struct_r3b_R1.mat", ...
  "struct_r4b_R1.mat", ...
  ];

% Rotor conditions (that is, number of broken bars) corresponding to original data files.
```

```matlab
health = [
  "healthy", ...
  "broken_bar_1", ...
  "broken_bar_2", ...
  "broken_bar_3", ...
  "broken_bar_4", ...
  ];

Fs_vib = 7600; % Sampling frequency of vibration signals in Hz.
Fs_elec = 50000; % Sampling frequency of electrical signals in Hz.
```

Create a target folder to store the ensemble member data files that you generate next.

```matlab
folder = 'data_files';
if ~exist(folder, 'dir')
  mkdir(folder);
end
```

Iterate over the original data files, where each file corresponds to a given number of broken rotor bars, and construct individual ensemble member data files for each combination of load and number of broken rotor bars.

```matlab
% Iterate over the number of broken rotor bars.
for i = 1:numel(health)
  fprintf('Processing data file %s\n', files(i))

  % Load the original data set stored as a struct.
  S = load(files(i));
  fields = fieldnames(S);
  dataset = S.(fields{1});

  loadLevels = fieldnames(dataset);
  % Iterate over load (torque) levels in each data set.
  for j = 1:numel(loadLevels)
    experiments = dataset.(loadLevels{j});
    data = struct;

    % Iterate over the given number of experiments for each load level.
    for k = 1:numExperiments
      signalNames = fieldnames(experiments(k));
      % Iterate over the signals in each experimental data set.
      for l = 1:numel(signalNames)
        % Experimental (electrical and vibration) data
        data.(signalNames{l}) = experiments(k).(signalNames{l});
      end

      % Operating conditions
      data.Health = health(i);
      data.Load = string(loadLevels{j});

      % Constant parameters
      data.Fs_vib = Fs_vib;
      data.Fs_elec = Fs_elec;

      % Save memberwise data.
      name = sprintf('rotor%db_%s_experiment%02d',  i-1, loadLevels{j}, k);
      fprintf('\tCreating the member data file %s.mat\n', name)
      filename = fullfile(pwd, folder, name);
```

4-157

```matlab
        save(filename, '-v7.3', '-struct', 'data'); % Save fields as individual variables.
      end
    end
end
```

```
Processing data file struct_rs_R1.mat

    Creating the member data file rotor0b_torque05_experiment01.mat
    Creating the member data file rotor0b_torque05_experiment02.mat
    Creating the member data file rotor0b_torque10_experiment01.mat
    Creating the member data file rotor0b_torque10_experiment02.mat
    Creating the member data file rotor0b_torque15_experiment01.mat
    Creating the member data file rotor0b_torque15_experiment02.mat
    Creating the member data file rotor0b_torque20_experiment01.mat
    Creating the member data file rotor0b_torque20_experiment02.mat
    Creating the member data file rotor0b_torque25_experiment01.mat
    Creating the member data file rotor0b_torque25_experiment02.mat
    Creating the member data file rotor0b_torque30_experiment01.mat
    Creating the member data file rotor0b_torque30_experiment02.mat
    Creating the member data file rotor0b_torque35_experiment01.mat
    Creating the member data file rotor0b_torque35_experiment02.mat
    Creating the member data file rotor0b_torque40_experiment01.mat
    Creating the member data file rotor0b_torque40_experiment02.mat

Processing data file struct_r1b_R1.mat

    Creating the member data file rotor1b_torque05_experiment01.mat
    Creating the member data file rotor1b_torque05_experiment02.mat
    Creating the member data file rotor1b_torque10_experiment01.mat
    Creating the member data file rotor1b_torque10_experiment02.mat
    Creating the member data file rotor1b_torque15_experiment01.mat
    Creating the member data file rotor1b_torque15_experiment02.mat
    Creating the member data file rotor1b_torque20_experiment01.mat
    Creating the member data file rotor1b_torque20_experiment02.mat
    Creating the member data file rotor1b_torque25_experiment01.mat
    Creating the member data file rotor1b_torque25_experiment02.mat
    Creating the member data file rotor1b_torque30_experiment01.mat
    Creating the member data file rotor1b_torque30_experiment02.mat
    Creating the member data file rotor1b_torque35_experiment01.mat
    Creating the member data file rotor1b_torque35_experiment02.mat
    Creating the member data file rotor1b_torque40_experiment01.mat
    Creating the member data file rotor1b_torque40_experiment02.mat

Processing data file struct_r2b_R1.mat

    Creating the member data file rotor2b_torque05_experiment01.mat
    Creating the member data file rotor2b_torque05_experiment02.mat
    Creating the member data file rotor2b_torque10_experiment01.mat
    Creating the member data file rotor2b_torque10_experiment02.mat
    Creating the member data file rotor2b_torque15_experiment01.mat
    Creating the member data file rotor2b_torque15_experiment02.mat
    Creating the member data file rotor2b_torque20_experiment01.mat
    Creating the member data file rotor2b_torque20_experiment02.mat
    Creating the member data file rotor2b_torque25_experiment01.mat
    Creating the member data file rotor2b_torque25_experiment02.mat
    Creating the member data file rotor2b_torque30_experiment01.mat
    Creating the member data file rotor2b_torque30_experiment02.mat
    Creating the member data file rotor2b_torque35_experiment01.mat
    Creating the member data file rotor2b_torque35_experiment02.mat
```

```
    Creating the member data file rotor2b_torque40_experiment01.mat
    Creating the member data file rotor2b_torque40_experiment02.mat

Processing data file struct_r3b_R1.mat

    Creating the member data file rotor3b_torque05_experiment01.mat
    Creating the member data file rotor3b_torque05_experiment02.mat
    Creating the member data file rotor3b_torque10_experiment01.mat
    Creating the member data file rotor3b_torque10_experiment02.mat
    Creating the member data file rotor3b_torque15_experiment01.mat
    Creating the member data file rotor3b_torque15_experiment02.mat
    Creating the member data file rotor3b_torque20_experiment01.mat
    Creating the member data file rotor3b_torque20_experiment02.mat
    Creating the member data file rotor3b_torque25_experiment01.mat
    Creating the member data file rotor3b_torque25_experiment02.mat
    Creating the member data file rotor3b_torque30_experiment01.mat
    Creating the member data file rotor3b_torque30_experiment02.mat
    Creating the member data file rotor3b_torque35_experiment01.mat
    Creating the member data file rotor3b_torque35_experiment02.mat
    Creating the member data file rotor3b_torque40_experiment01.mat
    Creating the member data file rotor3b_torque40_experiment02.mat

Processing data file struct_r4b_R1.mat

    Creating the member data file rotor4b_torque05_experiment01.mat
    Creating the member data file rotor4b_torque05_experiment02.mat
    Creating the member data file rotor4b_torque10_experiment01.mat
    Creating the member data file rotor4b_torque10_experiment02.mat
    Creating the member data file rotor4b_torque15_experiment01.mat
    Creating the member data file rotor4b_torque15_experiment02.mat
    Creating the member data file rotor4b_torque20_experiment01.mat
    Creating the member data file rotor4b_torque20_experiment02.mat
    Creating the member data file rotor4b_torque25_experiment01.mat
    Creating the member data file rotor4b_torque25_experiment02.mat
    Creating the member data file rotor4b_torque30_experiment01.mat
    Creating the member data file rotor4b_torque30_experiment02.mat
    Creating the member data file rotor4b_torque35_experiment01.mat
    Creating the member data file rotor4b_torque35_experiment02.mat
    Creating the member data file rotor4b_torque40_experiment01.mat
    Creating the member data file rotor4b_torque40_experiment02.mat
```

Construct File Ensemble Datastore

Create a file ensemble datastore for the data stored in the MAT-files, and configure it with functions that interact with the software to read from and write to the datastore.

```
location = fullfile(pwd, folder);
ens = fileEnsembleDatastore(location,'.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 supplied functions.

- `readMemberData` — Extract requested variables stored in the file. The function returns a table row containing one table variable for each requested variable.

- `writeMemberData` — Take a structure and write its variables to a data file as individual stored variables.

```
ens.ReadFcn = @readMemberData;
ens.WriteToMemberFcn = @writeMemberData;
```

Finally, set properties of the ensemble to identify data variables, condition variables, and the variables selected to read. These variables include ones that are not in the original data set but rather specified and constructed by `readMemberData`: `Vib_acpi_env`, the band-pass filtered radial vibration signal, and `Ia_env_ps`, the band-pass filtered envelope spectrum of the first-phase current signal. **Diagnostic Feature Designer** can read synthetic signals such as these as well.

```
ens.DataVariables = [...
  "Va"; "Vb"; "Vc"; "Ia"; "Ib"; "Ic"; ...
  "Vib_acpi"; "Vib_carc"; "Vib_acpe"; "Vib_axial"; "Vib_base"; "Trigger"];
ens.ConditionVariables = ["Health"; "Load"];
ens.SelectedVariables = ["Ia"; "Vib_acpi"; "Health"; "Load"];

% Add synthetic signals and spectra generated directly in the readMemberData
% function.
ens.DataVariables = [ens.DataVariables; "Vib_acpi_env"; "Ia_env_ps"];
ens.SelectedVariables = [ens.SelectedVariables; "Vib_acpi_env"; "Ia_env_ps"];
```

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

```
ens
```

```
ens =
  fileEnsembleDatastore with properties:

                 ReadFcn: @readMemberData
         WriteToMemberFcn: @writeMemberData
            DataVariables: [14×1 string]
     IndependentVariables: [0×0 string]
       ConditionVariables: [2×1 string]
        SelectedVariables: [6×1 string]
                ReadSize: 1
              NumMembers: 80
          LastMemberRead: [0×0 string]
                   Files: [80×1 string]
```

Examine a member of the ensemble and confirm that the desired variables are read or generated.

```
T = read(ens)
```

```
T=1×6 table
        Ia                  Vib_acpi             Vib_acpi_env            Ia_env_ps             Heal
    _____    _____    _____    _____    _____

    {50001×1 timetable}  {7601×1 timetable}   {7601×1 timetable}   {25001×2 double}     "heal
```

From the power spectrum of one of the vibration signals, `Vib_acpi`, observe that there are frequency components of interest in the [900 1300] Hz region. These components are a primary reason why you use `readMemberData` to compute the band-pass filtered variables - `Vib_acpi_env` and `Ia_env_ps`.

```
% Power spectrum of vibration signal, Vib_acpi
vib = T.Vib_acpi{1};
```

```
pspectrum(vib.Data, Fs_vib);
annotation("textarrow", [0.45 0.38], [0.65 0.54], "String", "Fault frequency region of interest")
```
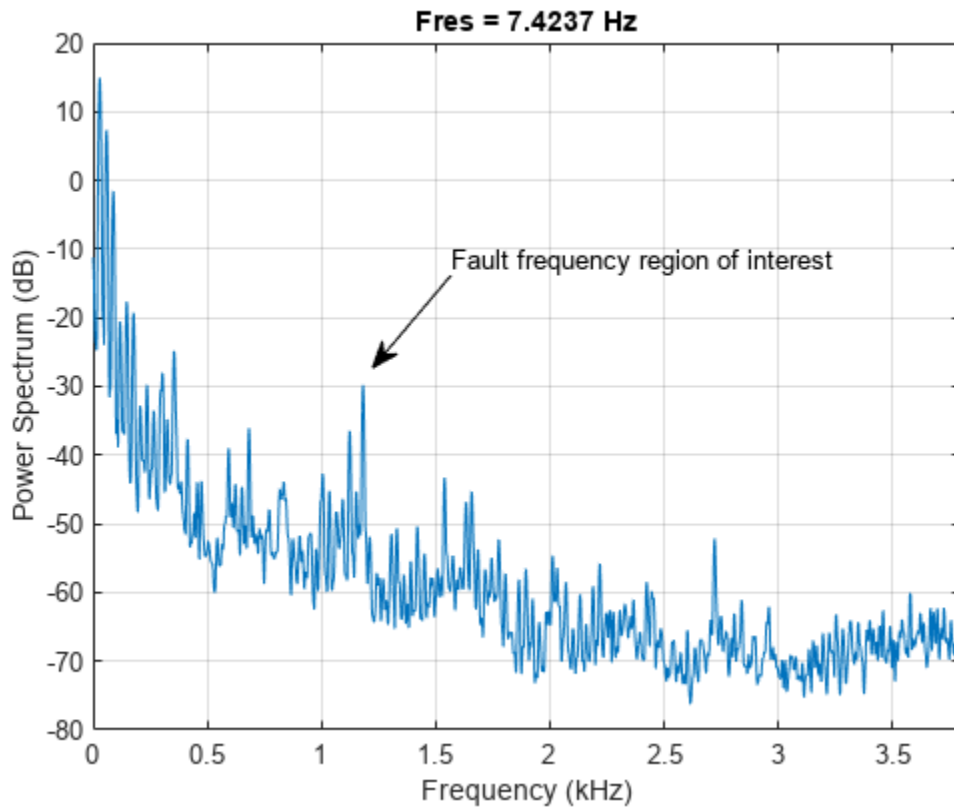

The envelope of signals after band-pass filtering them can help reveal demodulated signals containing behavior related to the health of the system.

```
% Envelop of vibration signal
y = bandpass(vib.Data, [900 1300], Fs_vib);
envelope(y)
axis([0 800 -0.5 0.5])
```
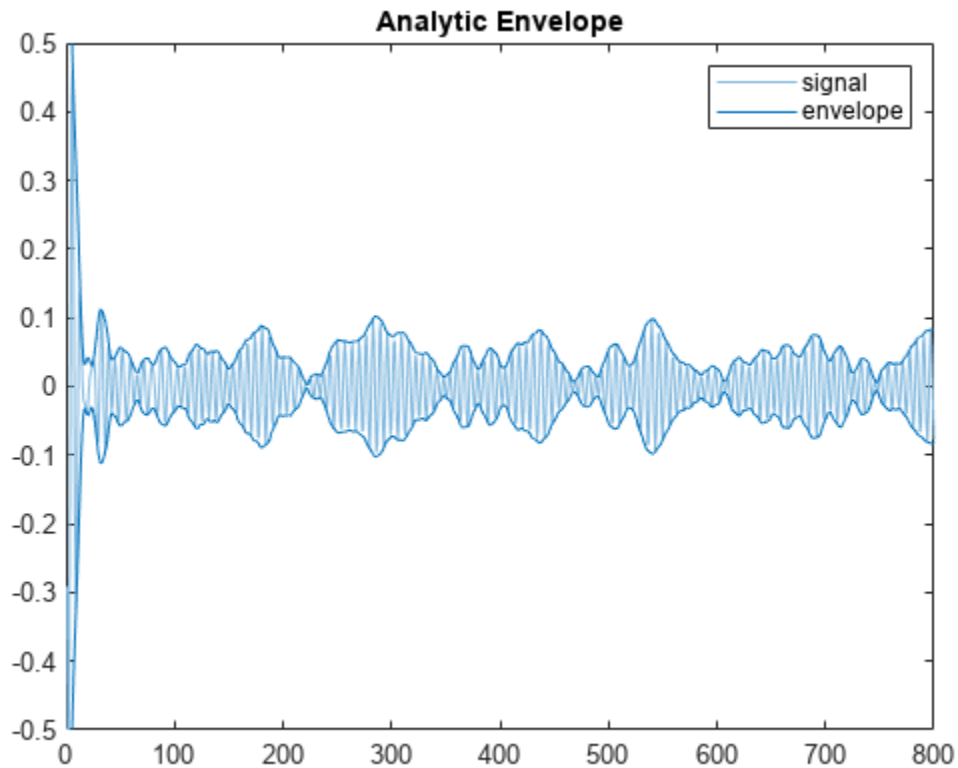
Import Data into Diagnostic Feature Designer App

Open the **Diagnostic Feature Designer** app using the following command.

```
app = diagnosticFeatureDesigner;
```

You can then import the file ensemble datastore into the app by clicking the **New Session** button and selecting the `ens` variable from the list.

Once preliminary data is read by the app, make sure to uncheck the **Append data to file ensemble** option if you do not want to write new features or data back to the ensemble member files. When you are working with the smaller data set in this example, generating new features directly in memory is faster, so uncheck the option.

Alternatively, if the whole data set is small enough to fit into your computer's memory, then you can read it into a table object using the following command then import the table into the app directly.

```
% Working with in-memory data in the app. This may take a couple of minutes to
% run.
T = readall(ens)
```

T=*80×6 table*

Ia	Vib_acpi	Vib_acpi_env	Ia_env_ps	Heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal
{50001×1 timetable}	{7601×1 timetable}	{7601×1 timetable}	{25001×2 double}	"heal

⋮

Envelope Signal Data

Select the time-domain signals to import in the tree view of the **New Session** dialog. Here, you only use the synthetic signals, so uncheck Ia and Vib_acpi. Keep the envelope of the radial vibration speed on the driven side Vib_acpi_env checked; you do not need to modify it otherwise.

Envelope Spectrum Data

Select the frequency-domain data to import. Keep the envelope spectrum of Phase A current `Ia_env_ps` selected and change its **Variable type** to `Spectrum`.

Specify `Col1` as an independent (frequency) variable with a corresponding 50 kHz sampling frequency.

Condition Variables

Finally, the condition variable of interest for fault classification is the `Health` variable, which corresponds to the number of broken rotor bars in the experiments. Ensure that its **Variable type** is `Condition Variable`.
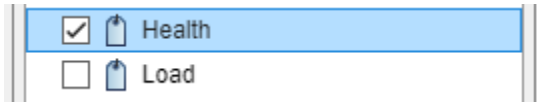
Confirm the final ensemble specification and click **Import**.
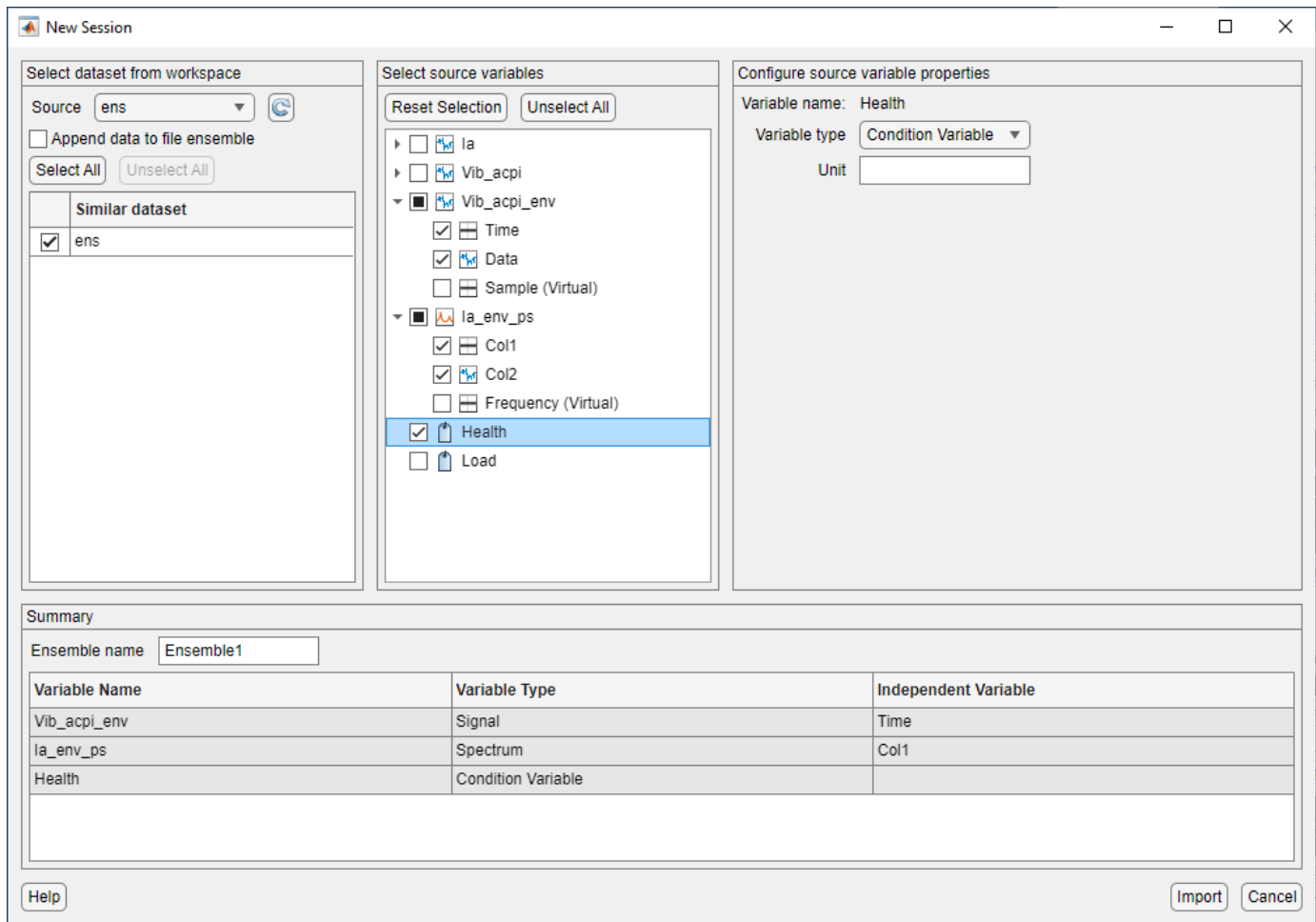
Once the file ensemble is imported, you can visualize the data by creating **Signal Trace** plots and grouping the plot colors based on the **Health** status of the experimental data. Note that reading all members of the file ensemble to produce plots and process the data might be time-consuming depending on the size and number of the data files that need to be read.
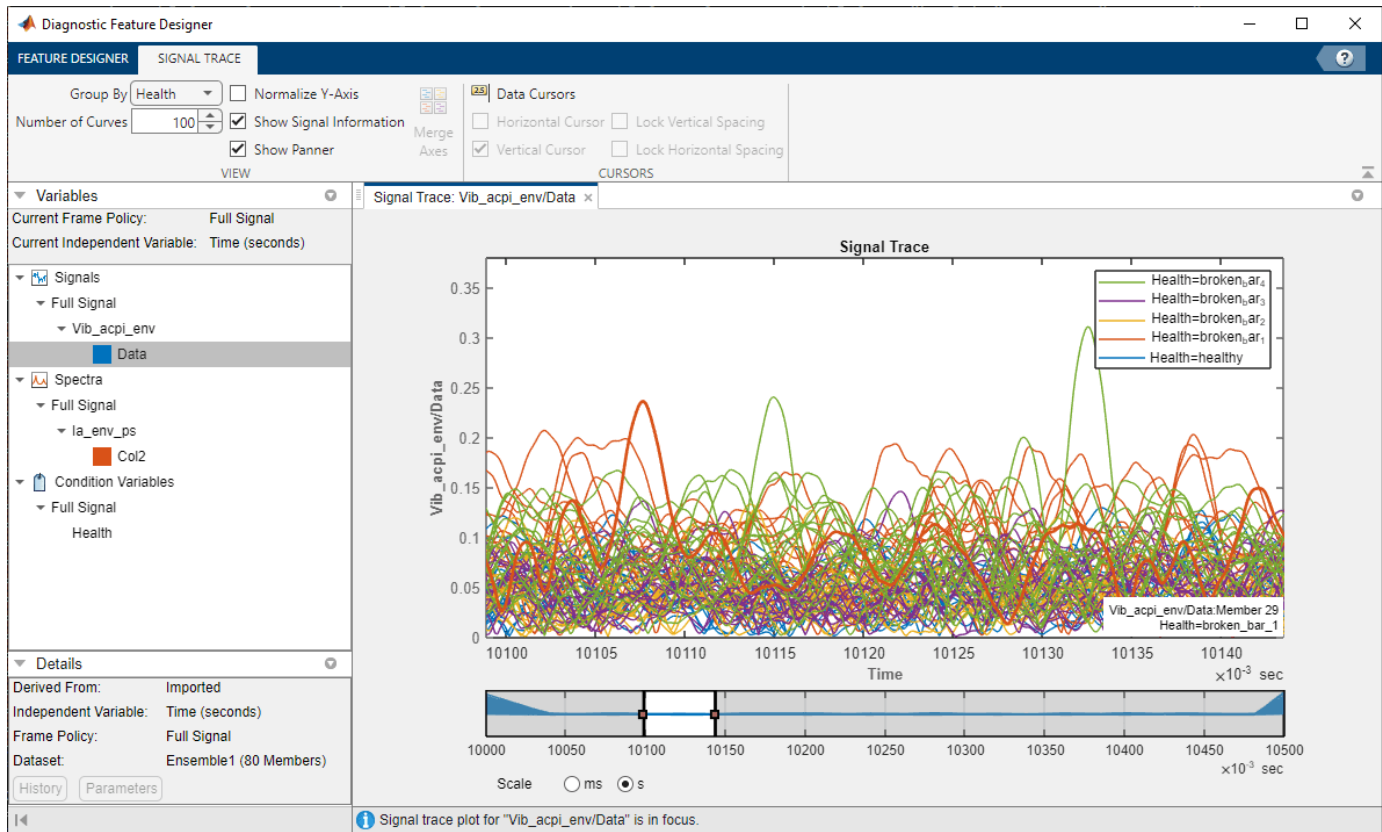
Note that the `getSignal` function used in `readMemberData` reads a portion of the signals from each member file after the initial transients due to motor startup have died down. In this case, the function loads only the data between 10.0 and 11.0 seconds

Create Predictive Features

The **Auto Features** option is a fast way of creating a large number of features from available signals and ranking them according to their power in separating various health classes. For the imported time-domain signal `Vib_acpi_env`, apply the **Auto Features** option by selecting the signal in the Data Browser tree and clicking **Auto Features** on the **Feature Designer** tab.
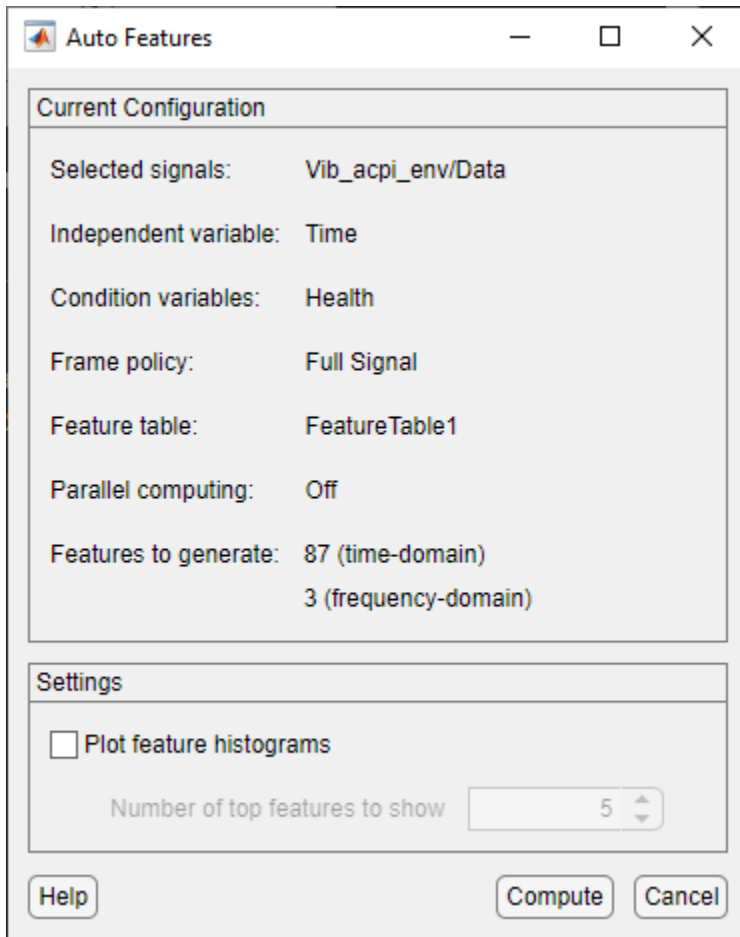
Clicking **Compute** generates a large number of features automatically and ranks them according their importance with respect to the **One-way ANOVA** criterion. This computation might take a few minutes to run.
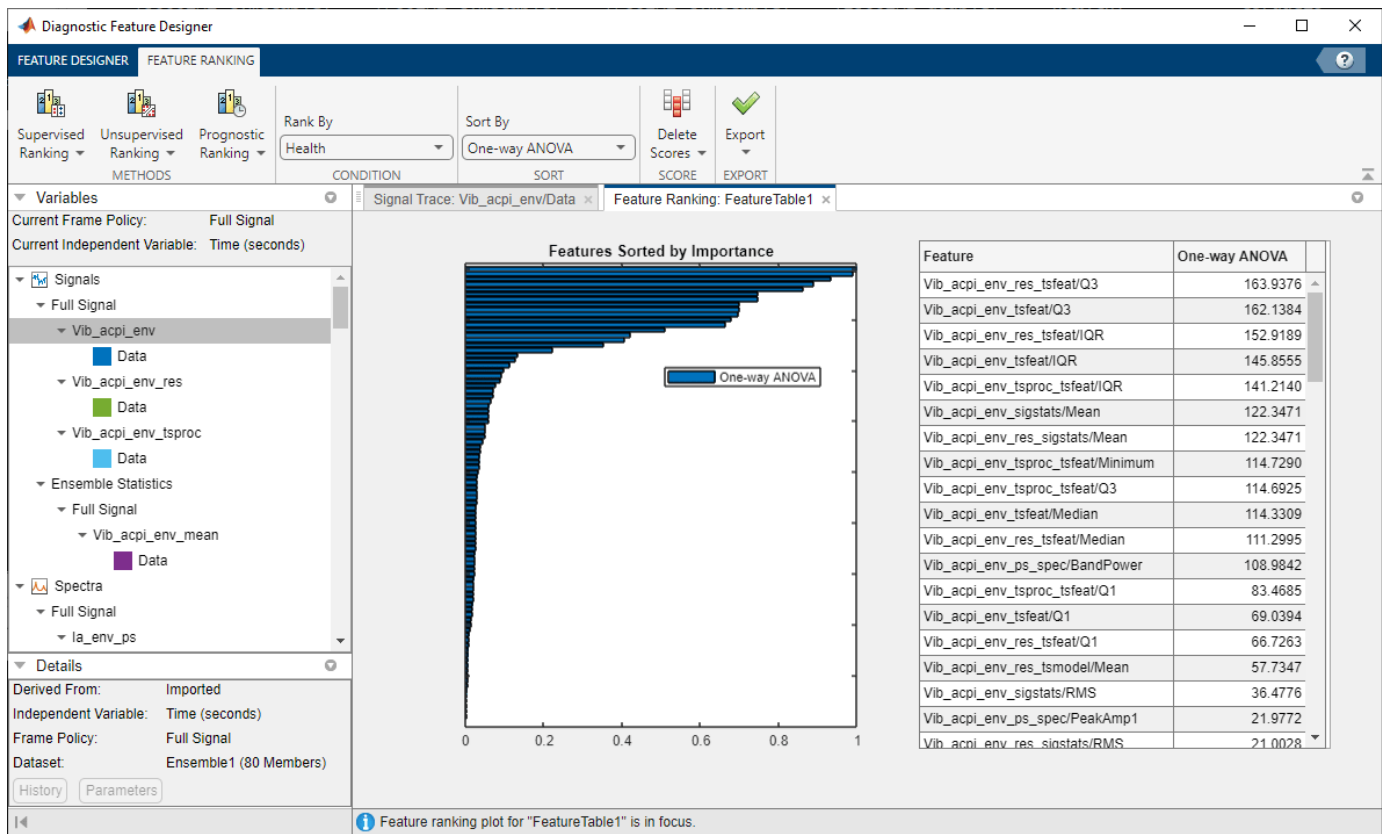
For generating features from envelope spectrum data, a more manual approach can yield more precise results. Select the `Ia_env_ps` signal in the Data Browser, and select **Custom Faults Features** under the **Frequency-Domain Features** menu. Select fault bands around a fundamental frequency of 60 Hz and its first 6 harmonics along with a sideband of 30 Hz to cover most of the spectral peaks in the envelope spectra. Use a fault band width of 10 Hz. Click **Apply** to generate features.
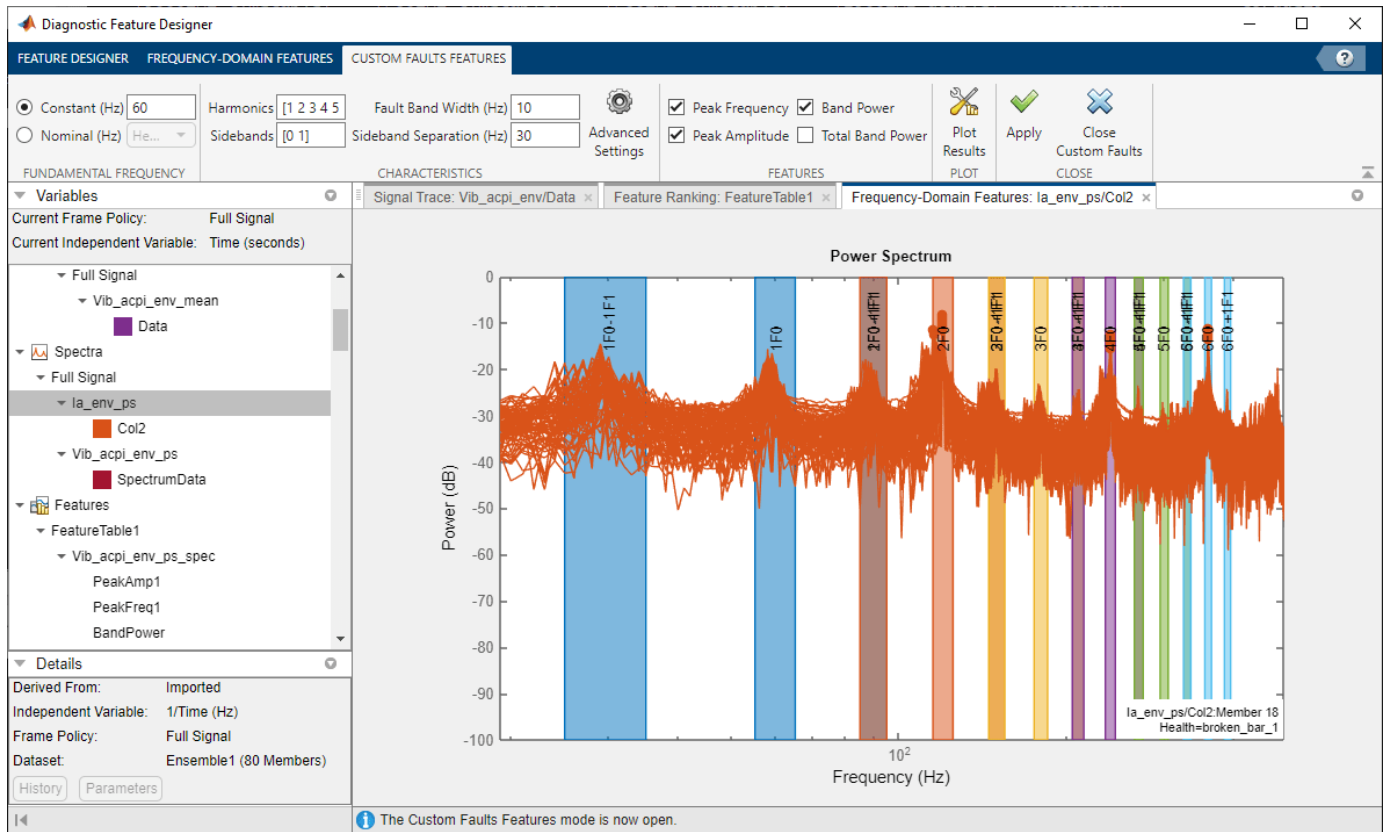
You have now generated a large number of features from the various electrical and vibration signals of the AC motor system.

Rank Features

This process allows you to quickly create about 150 features both manually and automatically. The final ranking table indicates that the first 10 to 15 features, based on **One-way ANOVA** ranking, help classify the motors according to the number of broken rotor bars in them.
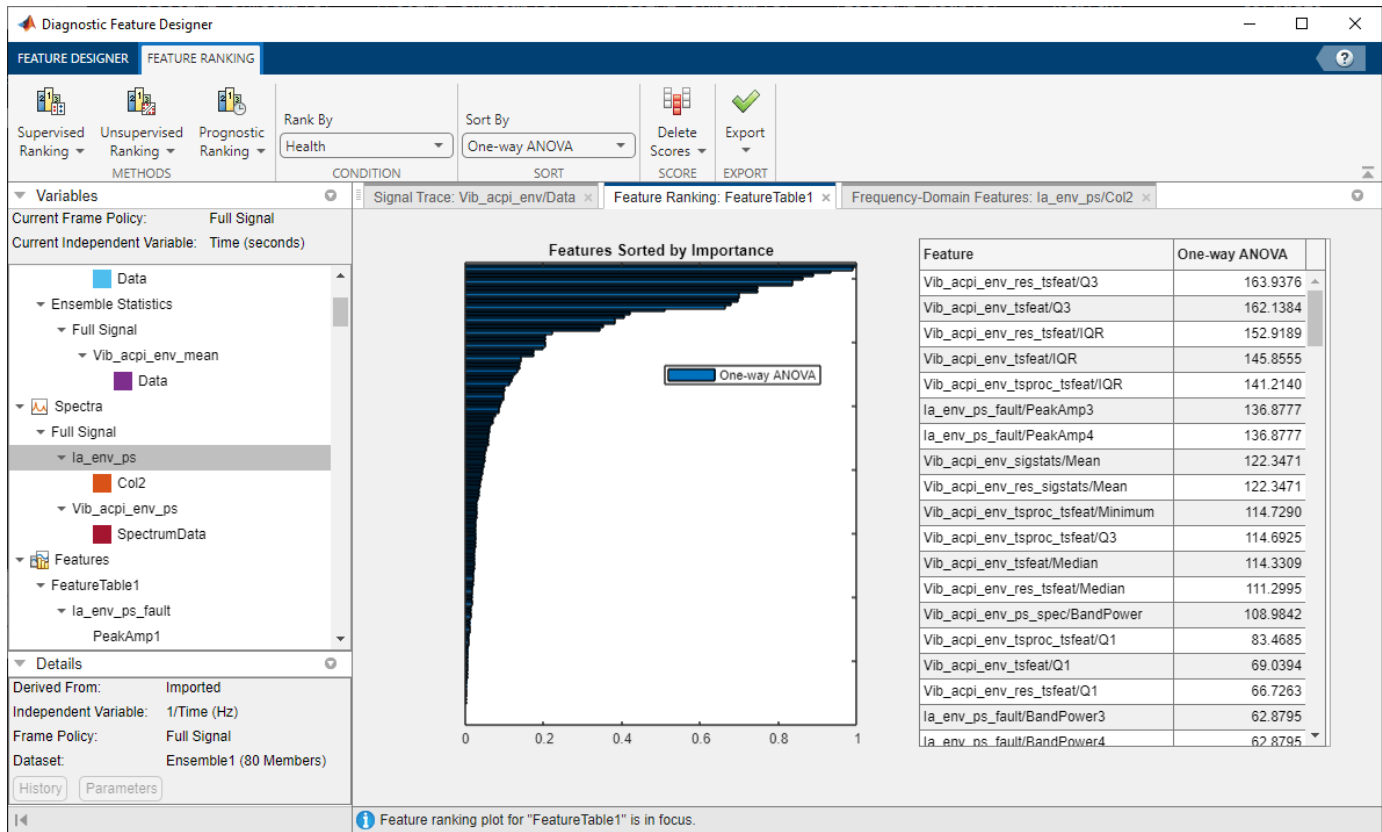
When fault bands and their corresponding names are difficult to identify, use the Peak Frequency feature to identify the corresponding fault band frequency/region.

Export Features to Classification Learner App and Build a Tree Model

Export the top features from the **Diagnostic Feature Designer** app using **Export > Export Features To Classification Learner** on the **Feature Ranking** tab. Change **Select top features** to 10.
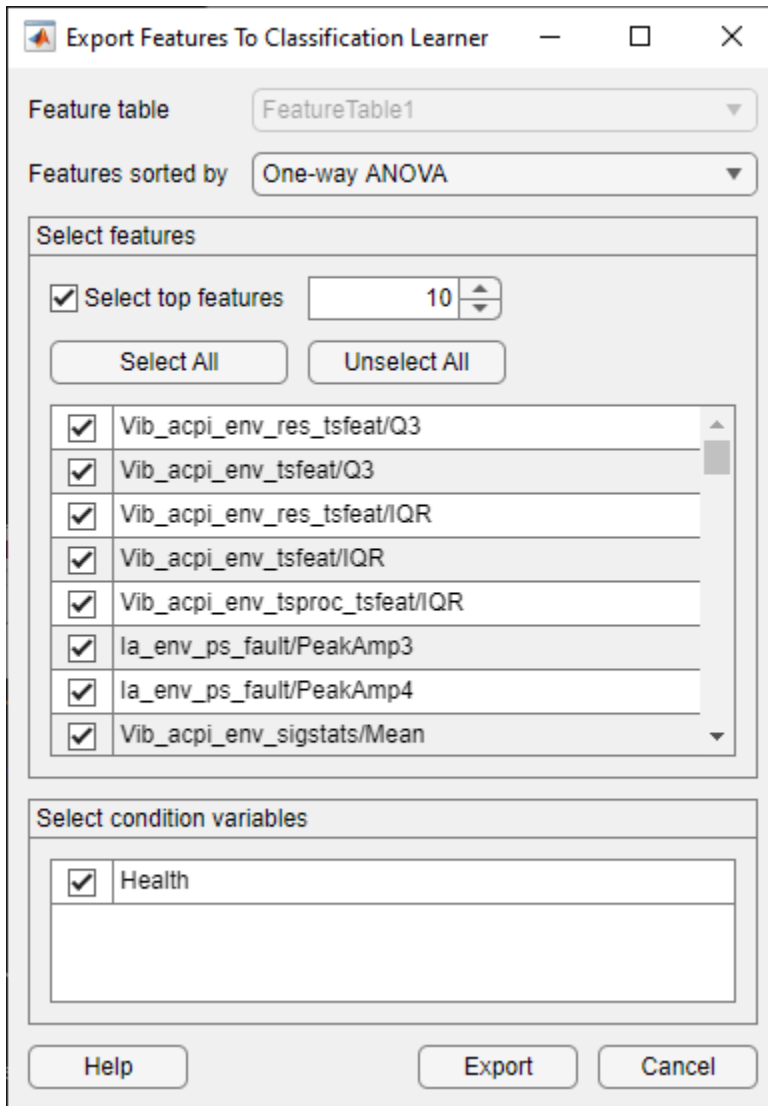
Click **Export** to start the **Classification Learner** app. The **New Session from File** dialog is populated correctly, so click **Start Session** to load the feature data into the app.
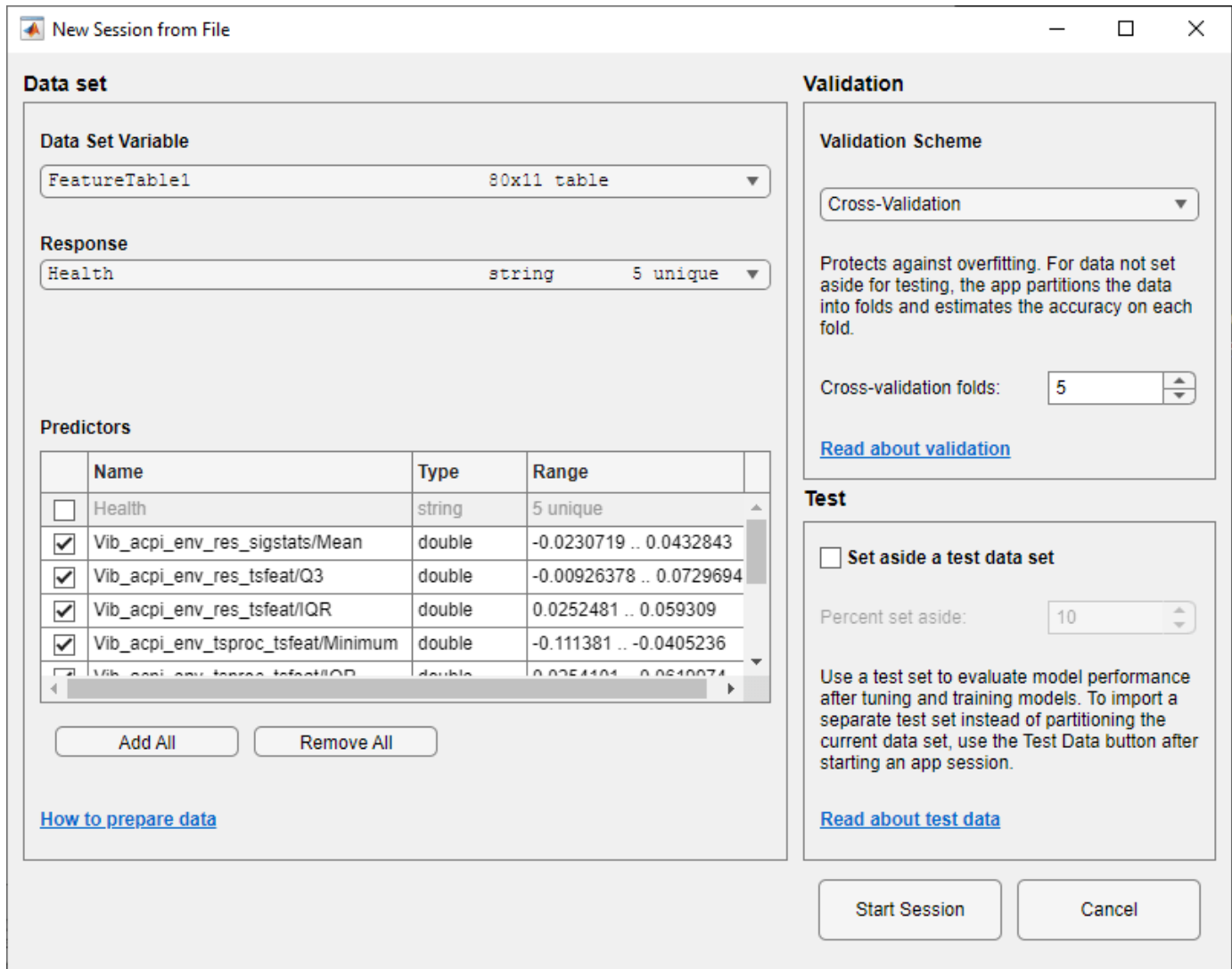
Click **Train All** on the **Classification Learner** tab to obtain a Tree model that achieves about 94% accuracy in classifying the motor faults (that is, the number of broken rotor bars) using the 10 features imported into the app.
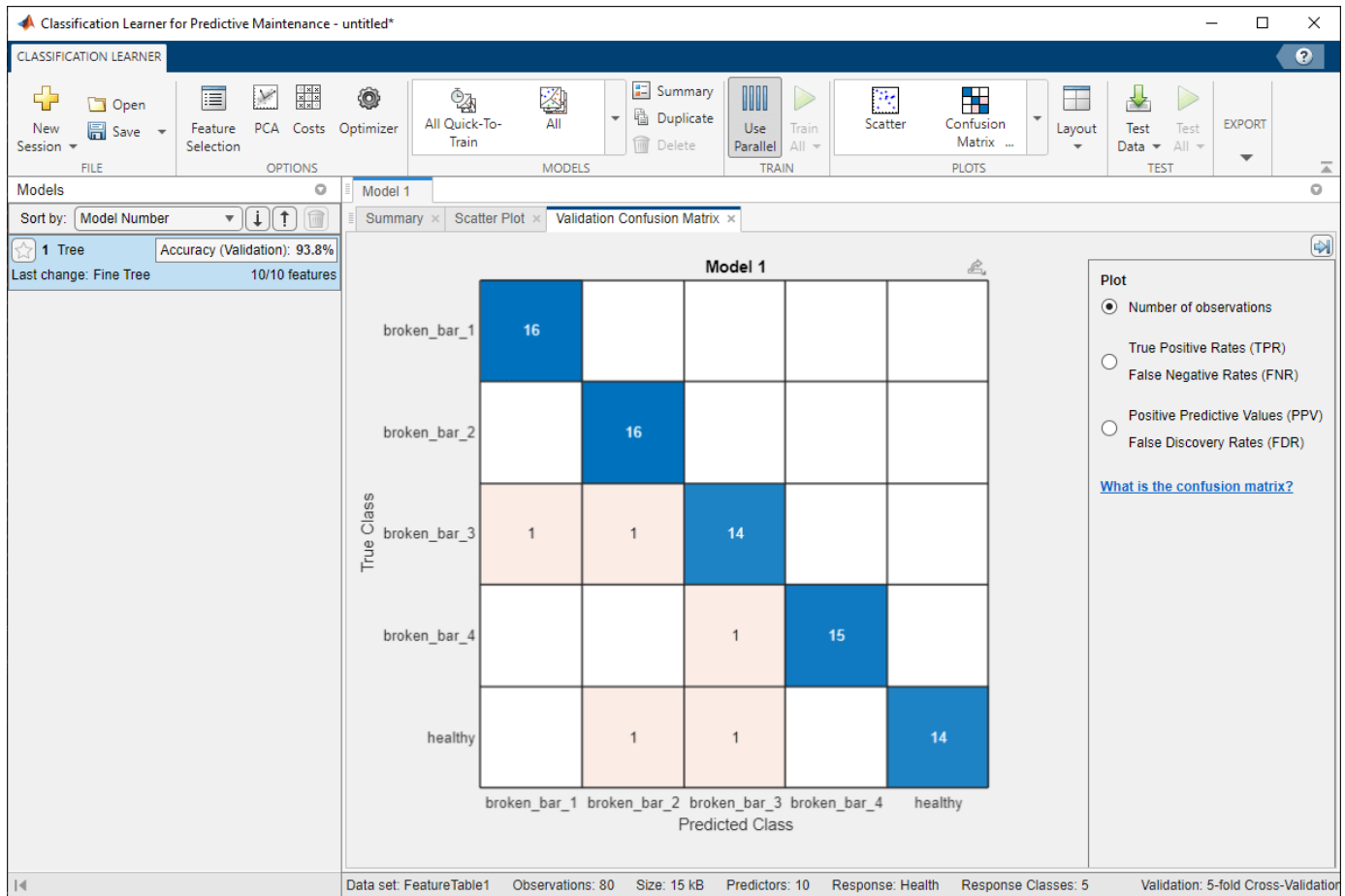
To generate a selection of fast-fitting models, **All Quick-To-Train** from the **Models** gallery, click **Train All**. The weighted KNN model number 2.9 in the figure is able to classify broken rotor faults with about 98% accuracy.
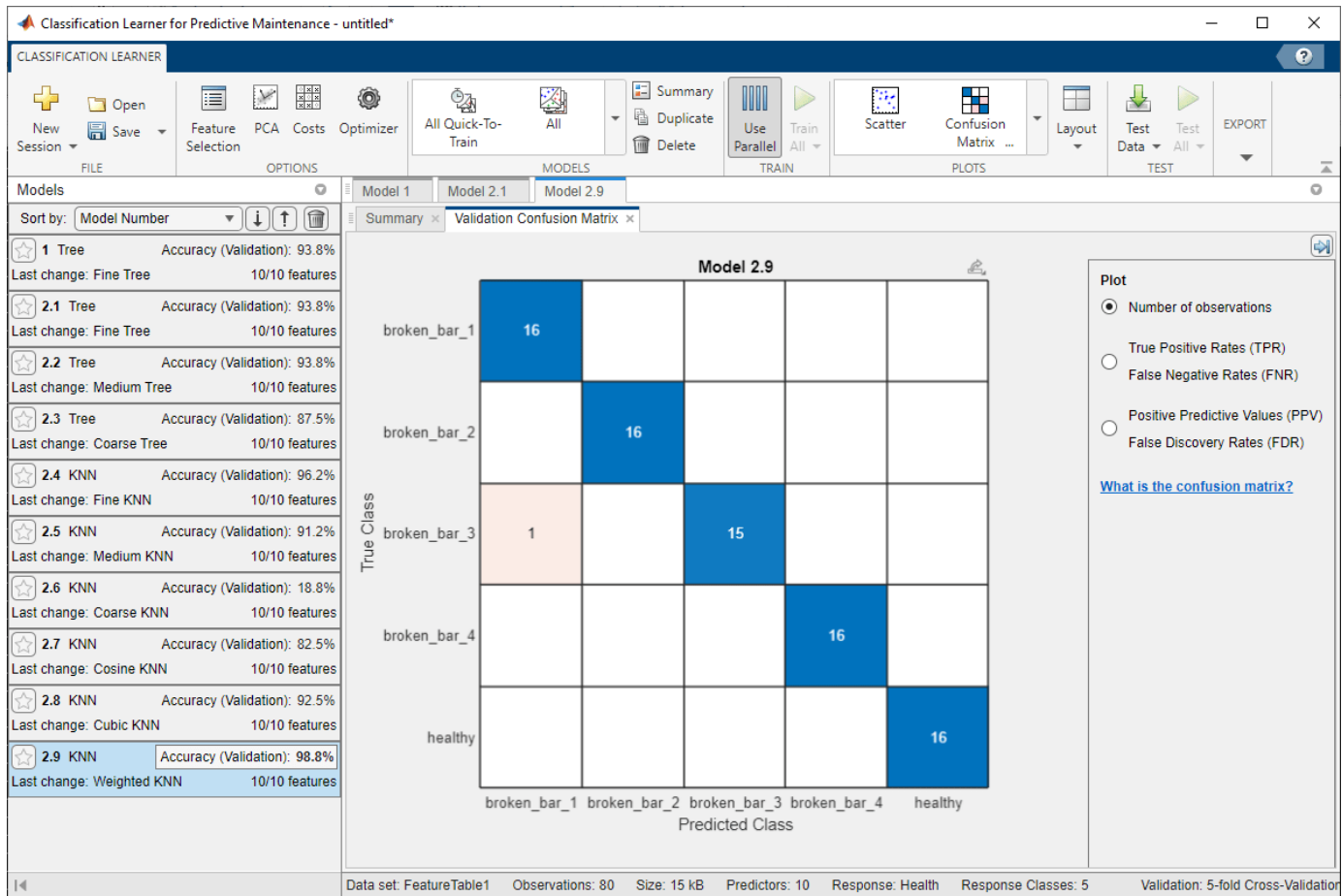
Supporting Functions

The `WriteToMemberFcn` property of the `fileEnsembleDatastore` object stores the handle of the function responsible for adding new data to each member of the ensemble. The **Diagnostic Feature Designer** app uses this function to save new processed data and extracted features.

```
function writeMemberData(filename, data)
% Write data into the fileEnsembleDatastore.
%
% Inputs:
%   filename - a string for the file name to write
%   data    - a data structure to write to the file

% Save fields as individual variables.
save(filename, '-v7.3', '-append', '-struct', 'data');
end
```

The function stored in the `ReadFcn` property of the `fileEnsembleDatastore` object reads member data from ensemble member files. The **Diagnostic Feature Designer** app uses this function to read signals, spectra, and features. You can also use this function to create synthetic signals on the fly that can be used in the app.

```
function T = readMemberData(filename, variables)
% Read variables from a fileEnsembleDatastore
```

```matlab
%
% Inputs:
%  filename  - file to read, specified as a string
%  variables - variable names to read, specified as a string array
%              Variables must be a subset of SelectedVariables specified in
%              the fileEnsembleDatastore.
% Output:
%  T         - a table with a single row

mfile = matfile(filename); % Allows partial loading

% Read condition variables directly from the top-level structure fields
T = table();
for i = 1:numel(variables)
  var = variables(i);

  switch var
    case {'Health', 'Load'}
      % Condition variables
      val = mfile.(var);
    case {'Va', 'Vb', 'Vc', 'Ia', 'Ib', 'Ic'}
      % Electrical signals
      val = getSignal(mfile, var, mfile.Fs_elec);
    case {'Vib_acpi', 'Vib_carc', 'Vib_acpe', 'Vib_axial', 'Vib_base', 'Trigger'}
      % Vibration signals
      val = getSignal(mfile, var, mfile.Fs_vib);
    case {'Vib_acpi_env'}
      % Synthetic envelope signals for vibration data
      sig = regexprep(var, '_env', '');
      TT = getSignal(mfile, sig, mfile.Fs_vib);
      % Envelope of band-pass filtered signal
      y = bandpass(TT.Data, [900 1300], mfile.Fs_vib);
      yUpper = envelope(y);
      val = timetable(yUpper, 'VariableNames', "Data", 'RowTimes', TT.Time);
    case {'Ia_env'}
      % Synthetic envelope signals for electrical data
      sig = regexprep(var, '_env', '');
      TT = getSignal(mfile, sig, mfile.Fs_elec);
      % Envelope of band-pass filtered signal
      y = bandpass(TT.Data, [900 1300], mfile.Fs_elec);
      yUpper = envelope(y);
      val = timetable(yUpper, 'VariableNames', "Data", 'RowTimes', TT.Time);
    case {'Ia_env_ps'}
      % Synthetic envelope spectra for electrical data
      sig = regexprep(var, '_env_ps', '');
      TT = getSignal(mfile, sig, mfile.Fs_elec);
      % Envelope spectrum of band-pass filtered signal
      [ES,F] = envspectrum(TT, 'Method', 'hilbert', 'Band', [900 1300]);
      val = [F,ES];
    otherwise
      % Other features and signals.
      val = mfile.(var);
  end

  if numel(val) > 1
    val = {val};
  end
end
```

```
  % Add the data to the output table, using the variable name.
  T.(var) = val;
end
end
```

When the original signal is longer than what is needed to extract useful features from it, you can use the following helper function to extract a shorter portion of the full signal from the member data.

```
function TT = getSignal(mfile, signame, Fs)
% Extract a 1.0 second portion of the signal after 10 seconds of measurements.
signame = char(signame);
n = size(mfile, signame, 1);
t = (0:n-1)' / Fs;
I = find((t >= 10.0) & (t <= 11.0)); % 1.0 sec of data
TT = timetable(mfile.(signame)(I,1), 'VariableNames', "Data", 'RowTimes', seconds(t(I)));
end
```

References

1 Treml, Aline Elly, Rogério Andrade Flauzino, Marcelo Suetake, and Narco Afonso Ravazzoli Maciejewski. "Experimental database for detecting and diagnosing rotor broken bar in a threephase induction motor." IEEE Dataport, updated September 24, 2020, https://dx.doi.org/10.21227/fmnm-bn95.

See Also

fileEnsembleDatastore | **Diagnostic Feature Designer** | **Classification Learner**

Related Examples

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.
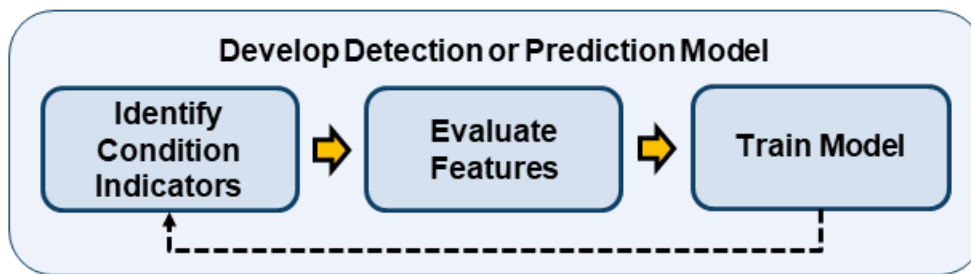
Predictive Maintenance Toolbox offers three feature selection metrics for accurate RUL prediction: monotonicity, trendability, and prognosability. These metrics rank the identified condition indicators on a scale ranging from 0 through 1. A higher ranked feature tracks the degradation process more reliably 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`.

In addition to using these functions at the command line, you can apply these feature-selection metrics in **Diagnostic Feature Designer** by selecting the prognostic ranking options.

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 5-37

- "Condition Indicators for Monitoring, Fault Detection, and Prediction" on page 3-2
- "Perform Prognostic Feature Ranking for a Degrading System Using Diagnostic Feature Designer" on page 7-65
- "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 5-7.

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.

Sensor data from machine on which algorithm is deployed
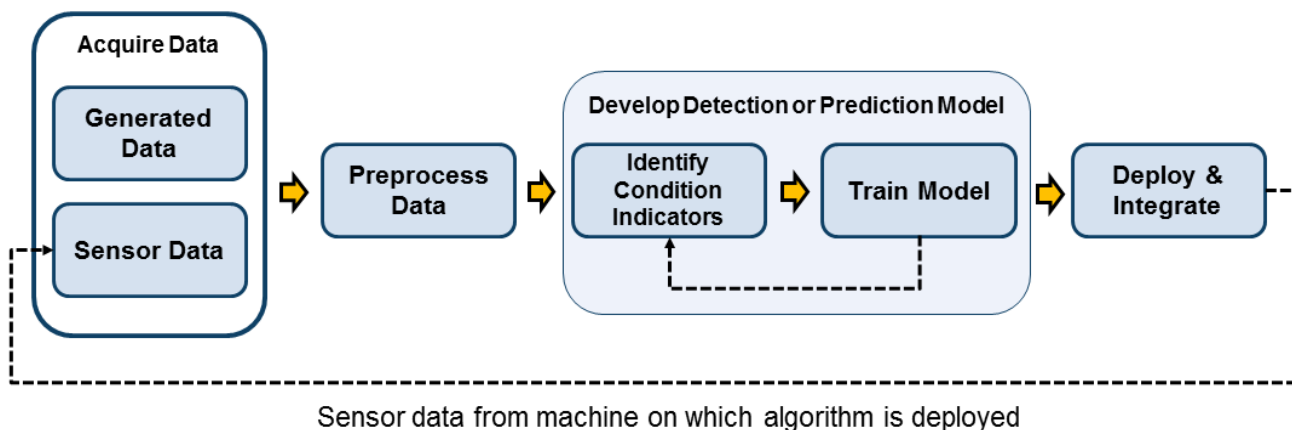
See Also

More About

- "RUL Estimation Using Identified Models or State Estimators" on page 5-6
- "RUL Estimation Using RUL Estimator Models" on page 5-7
- "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 5-54 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 5-69
- "Condition Monitoring and Prognostics Using Vibration Signals" on page 5-54
- "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 5-11.

Choose an RUL Estimator

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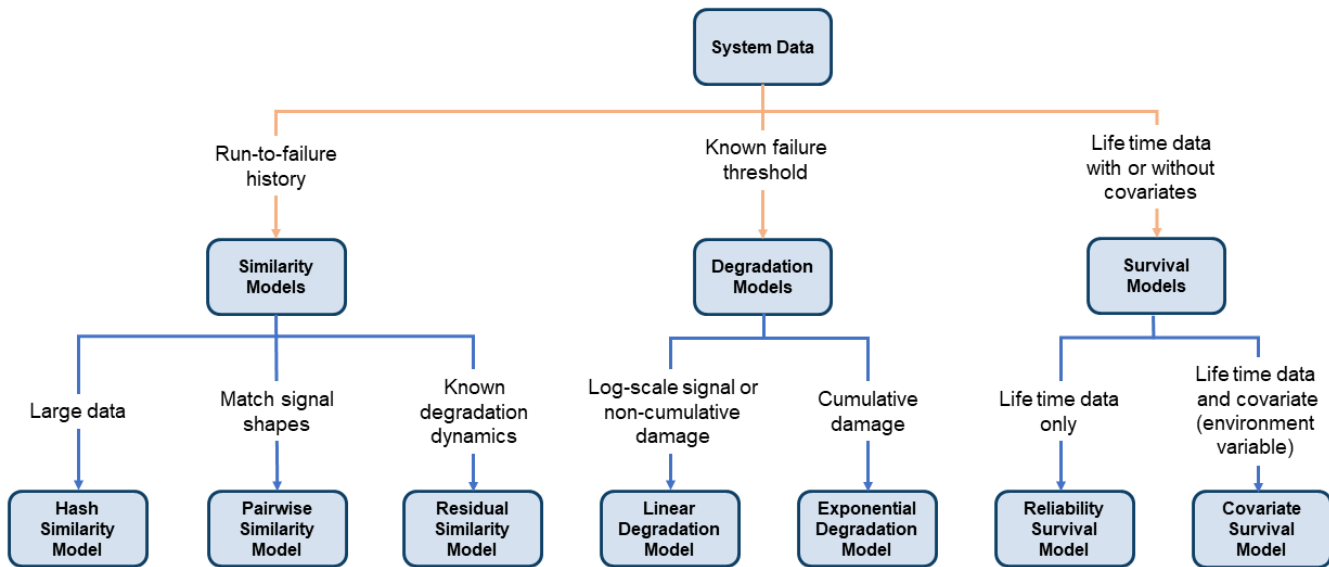
Similarity Models

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

Similarity models are useful when:

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

Thus you can use similarity models when you can obtain degradation profiles from your data ensemble. The degradation profiles represent the evolution of one or more condition indicators for each machine in the ensemble (each component), as the machine transitions from a healthy state to a faulty state.

Predictive Maintenance Toolbox includes three types of similarity models. All three types estimate RUL by determining the similarity between the degradation history of a test data set and the degradation history of data sets in the ensemble. For similarity models, `predictRUL` estimates the RUL of the test component as the median life span of most similar components minus the current 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 5-15.

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 5-37.

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

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 5-11
- "Similarity-Based Remaining Useful Life Estimation" on page 5-15
- "Wind Turbine High-Speed Bearing Prognosis" on page 5-37

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)
```

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: 0.6762
    ThetaVariance: 0.0727
             Beta: 0.0583
     BetaVariance: 1.8383e-04
              Rho: -0.2811
```

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
```
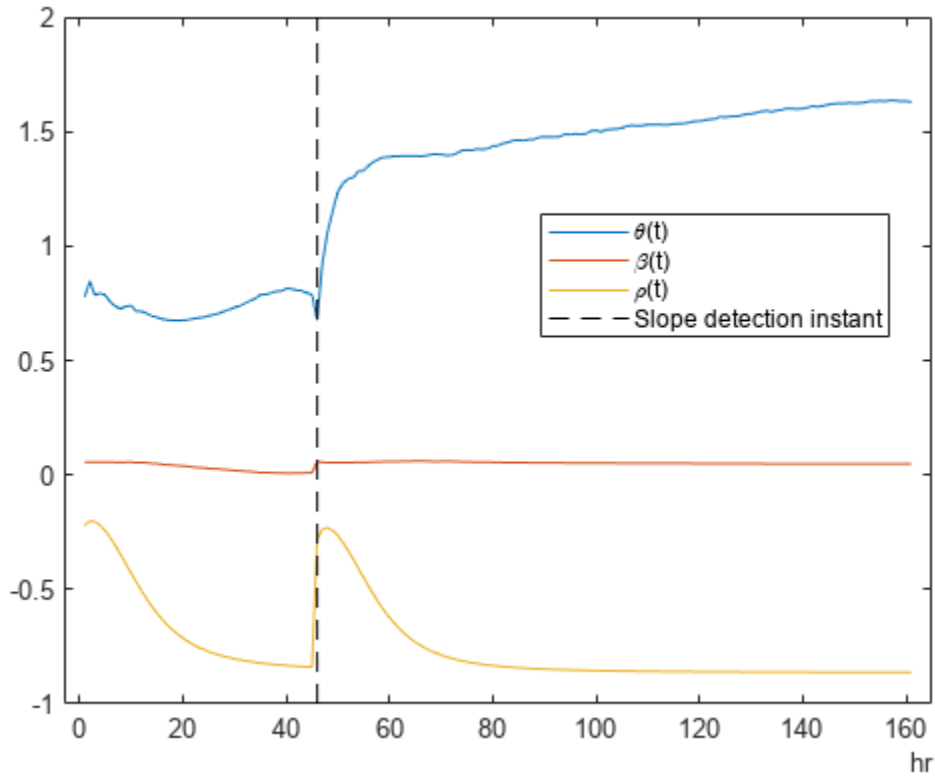
Plot the predicted RUL to observe its evolution as more degradation data becomes available. There is no new estimated RUL value until a slope is detected in the degradation data. After that, the predicted RUL decreases over time, as expected. `predictRUL` computes a statistical distribution of RUL values. 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'})
```
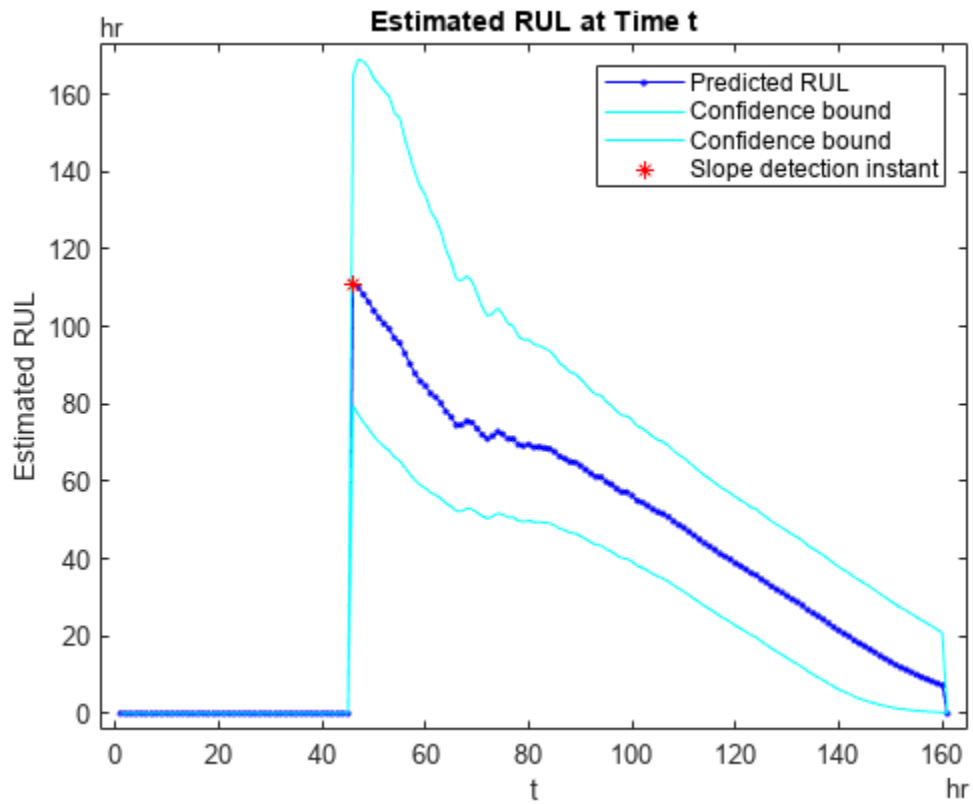
See Also
`exponentialDegradationModel` | `predictRUL` | `update`

More About

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 [1].

Data Preparation

Since the dataset is small it is feasible to load the whole degradation data into memory. Download and unzip the data set 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 260 run-to-failure simulations. This group of measurements is called an "ensemble".

```
url = 'https://ssd.mathworks.com/supportfiles/nnet/data/TurbofanEngineDegradationSimulationData.z
websave('TurbofanEngineDegradationSimulationData.zip',url);
unzip('TurbofanEngineDegradationSimulationData.zip')
degradationData = helperLoadData('train_FD002.txt');
degradationData(1:5)
```

```
ans=5×1 cell array
    {149×26 table}
    {269×26 table}
    {206×26 table}
    {235×26 table}
    {154×26 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})
```

id	time	op_setting_1	op_setting_2	op_setting_3	sensor_1	sensor_2	sensor_
1	1	34.998	0.84	100	449.44	555.32	1358.(
1	2	41.998	0.8408	100	445	549.9	1353.2
1	3	24.999	0.6218	60	462.54	537.31	1256.8
1	4	42.008	0.8416	100	445	549.51	1354
1	5	25	0.6203	60	462.54	537.07	1257.7
1	6	25.005	0.6205	60	462.54	537.02	1266.4
1	7	42.004	0.8409	100	445	549.74	1347.5
1	8	20.002	0.7002	100	491.19	607.44	1481.7

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)
```
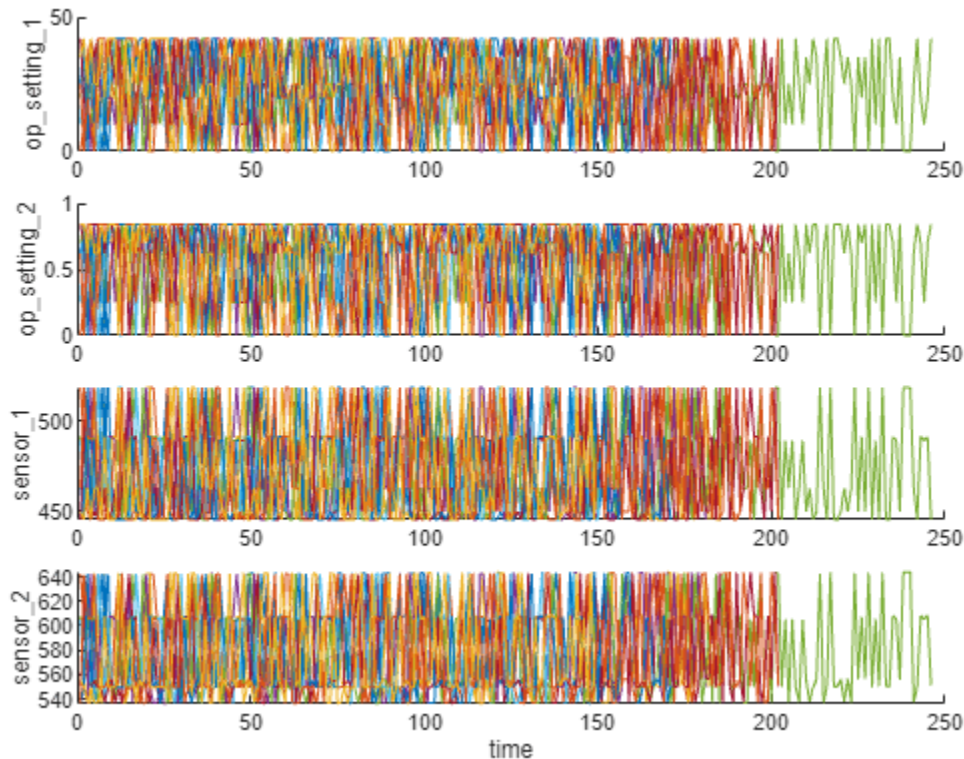

Working Regime Clustering

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)
```
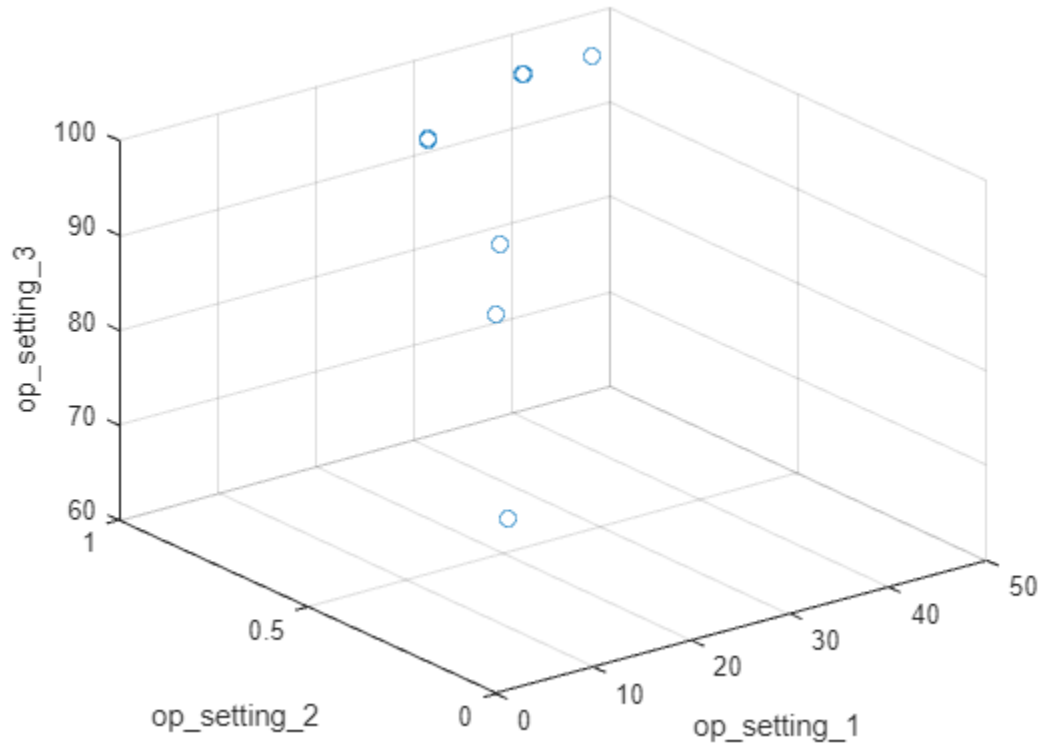

Let's use clustering techniques to locate the 6 clusters automatically. Here, the K-means algorithm is used. K-means is one of the most popular clustering algorithms, but it can result in local optima. It is a good practice to repeat the K-means clustering algorithm several times with different initial 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 = 0.331378.
Replicate 2, 1 iterations, total sum of distances = 0.331378.
Replicate 3, 1 iterations, total sum of distances = 0.331378.
Replicate 4, 1 iterations, total sum of distances = 0.331378.
Replicate 5, 1 iterations, total sum of distances = 0.331378.
Best total sum of distances = 0.331378
```

Visualize the clustering results and the identified cluster centroids.

```
figure
helperPlotClusters(opConditionUnwrap, clusterIndex, centers)
```
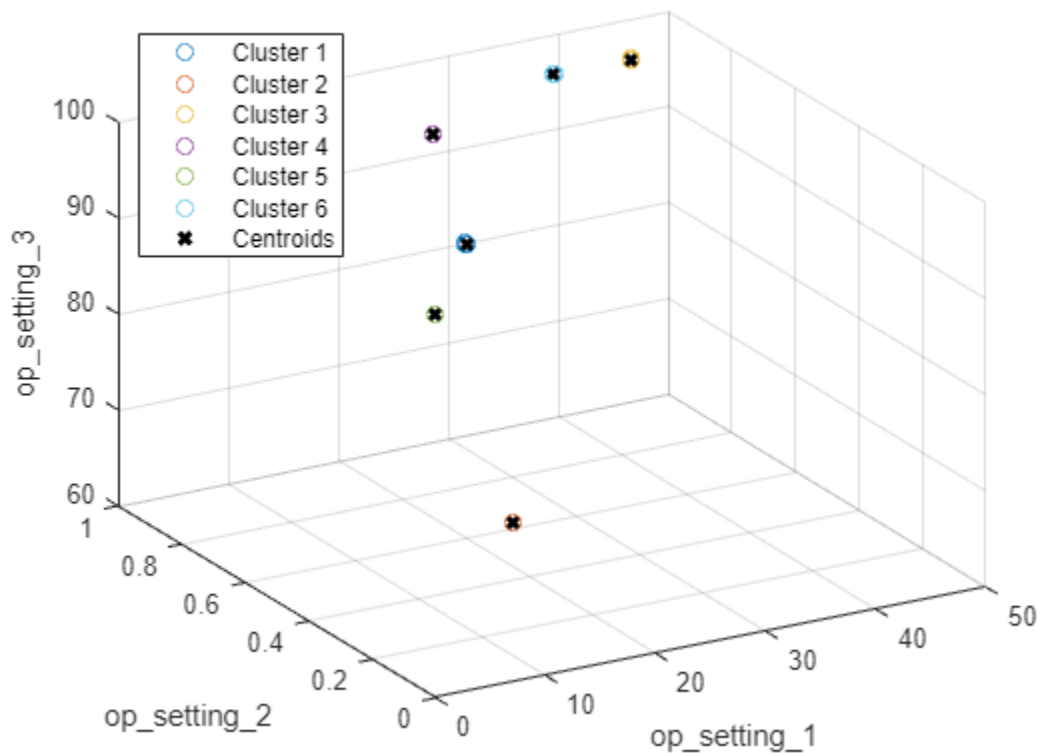
As the plot illustrates, the clustering algorithm successfully finds the 6 working regimes.

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)), clusterIndex);
    centerstats.SD.(char(v))  = splitapply(@std,  trainDataUnwrap.(char(v)), clusterIndex);
end
centerstats.Mean
```

```
ans=6×21 table
    sensor_1    sensor_2    sensor_3    sensor_4    sensor_5    sensor_6    sensor_7    sensor_8
    _____    _____    _____    _____    _____    _____    _____    _____

     489.05      604.91       1502       1311.2       10.52      15.493      394.33       2319
     462.54      536.86      1262.7      1050.2        7.05      9.0277      175.43      1915.4
        445       549.7      1354.4      1127.7        3.91      5.7158      138.63       2212
     491.19      607.56      1485.5        1253        9.35      13.657       334.5       2324
     518.67      642.67      1590.3      1408.7       14.62       21.61      553.37      2388.1
     449.44       555.8      1366.7      1131.5        5.48      8.0003      194.45       2223
```

```
centerstats.SD
```

```
ans=6×21 table
    sensor_1      sensor_2    sensor_3    sensor_4     sensor_5      sensor_6     sensor_7    sens
    _____    _____    _____    _____    _____    _____    _____    ____

    1.0346e-11    0.47582     5.8057      8.5359      3.2687e-13    0.0047002    0.67361     0.09
    4.6615e-12    0.36083     5.285       6.8365       1.137e-13    0.0042209    0.44862      0.2
             0    0.43282     5.6422      7.6564      1.1014e-13    0.0049312    0.43972      0.3
    1.8362e-11    0.46339     5.7888      7.7768      2.5049e-13    0.0047449     0.6046      0.1
    1.2279e-11     0.4967     6.0349      8.9734       6.573e-14    0.0013408    0.86985      0.0
    1.4098e-11     0.4401     5.6013      7.4461      2.2828e-13    0.0017508    0.47564      0.2
```

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)
```
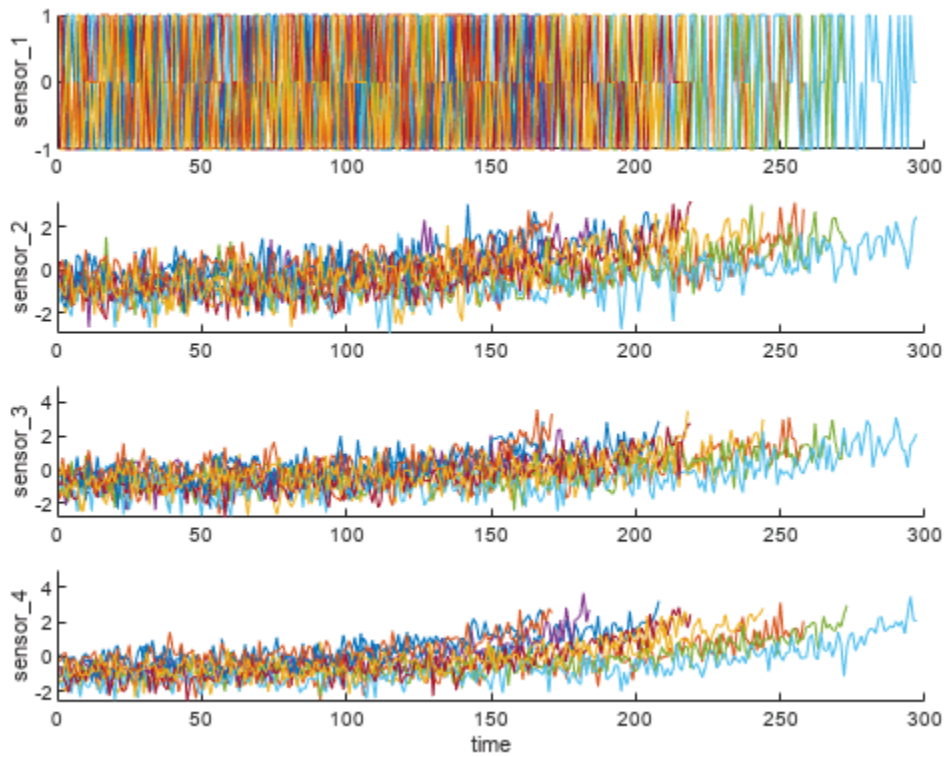
Trendability Analysis

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, 'UniformOutput
    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)), nsample
```
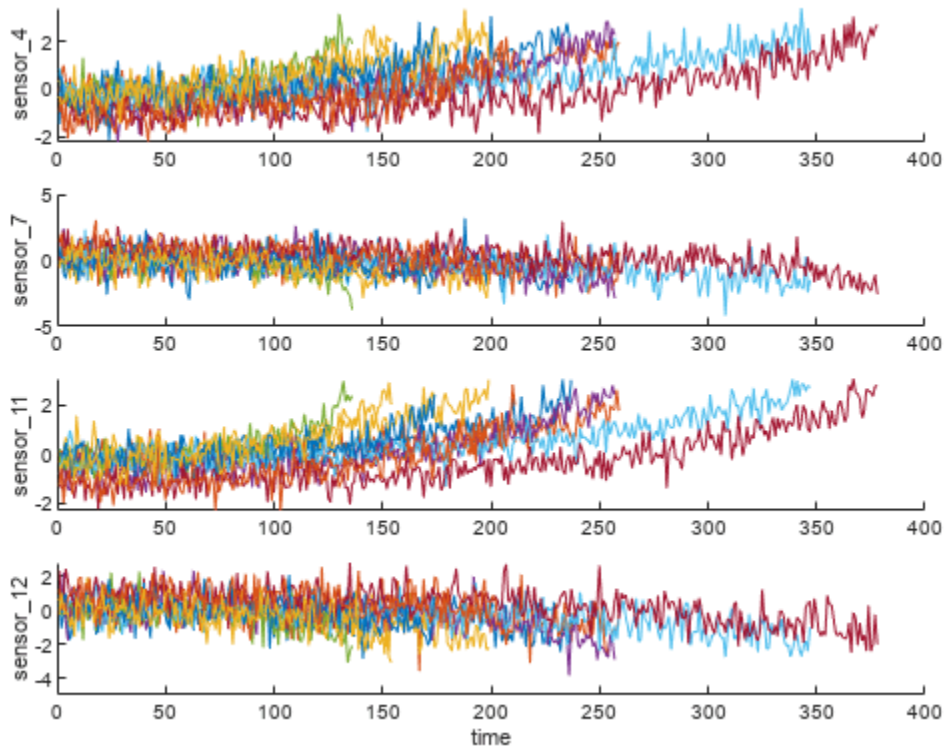

Notice that some of the most trendable signals show positive trends, while others show negative trends.

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 degradation 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)
```
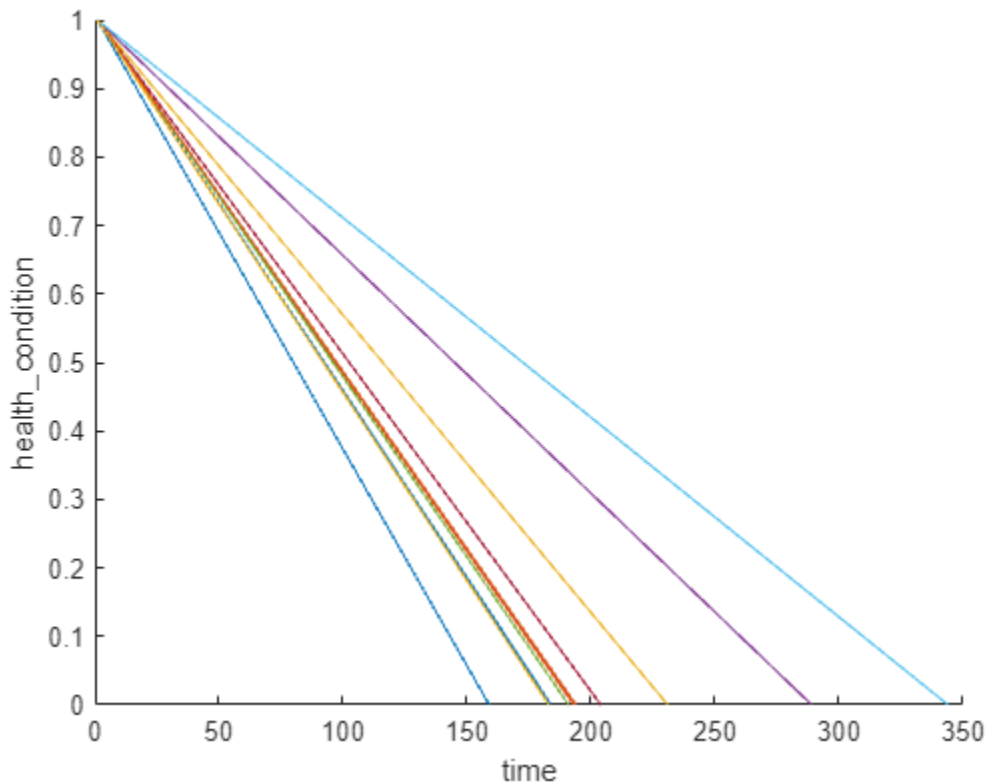

The health condition of all ensemble members change from 1 to 0 with varying degrading speeds.

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*

```
   -0.0323
   -0.0300
   -0.0527
    0.0057
   -0.0646
    0.0054
   -0.0431
   -0.0379
```

Construct a single health indicator by multiplying the sensor measurements with their associated weights .

```
trainDataFused = cellfun(@(data) degradationSensorFusion(data, sensorToFuse, weights), trainDataI
    'UniformOutput', false);
```

Visualize the fused health indicator for training data.

```
figure
helperPlotEnsemble(trainDataFused, [], 1, nsample)
xlabel('Time')
ylabel('Health Indicator')
title('Training Data')
```
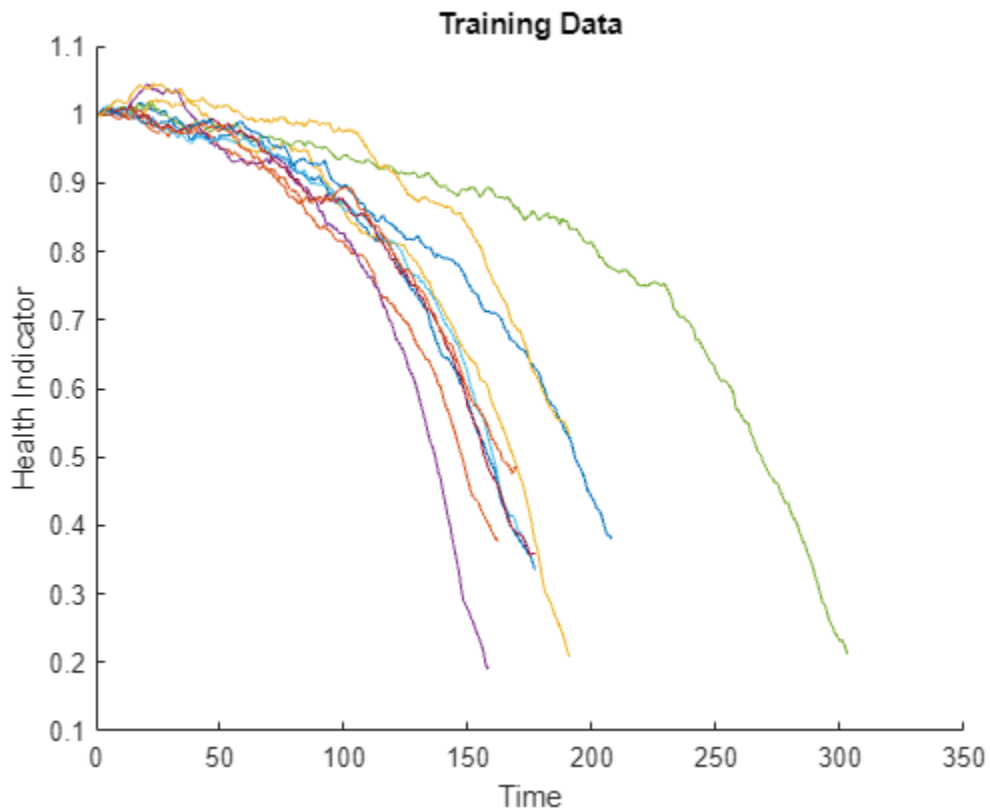
The data from multiple sensors are fused into a single health indicator. The health indicator is smoothed by a moving average filter. See helper function "degradationSensorFusion" in the last section for more details.

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, centerstats), ...
    validationData, 'UniformOutput', false);
validationDataFused = cellfun(@(data) degradationSensorFusion(data, sensorToFuse, weights), ...
    validationDataNormalized, 'UniformOutput', false);
```

Visualize the health indicator for validation data.

```
figure
helperPlotEnsemble(validationDataFused, [], 1, nsample)
xlabel('Time')
ylabel('Health Indicator')
title('Validation Data')
```
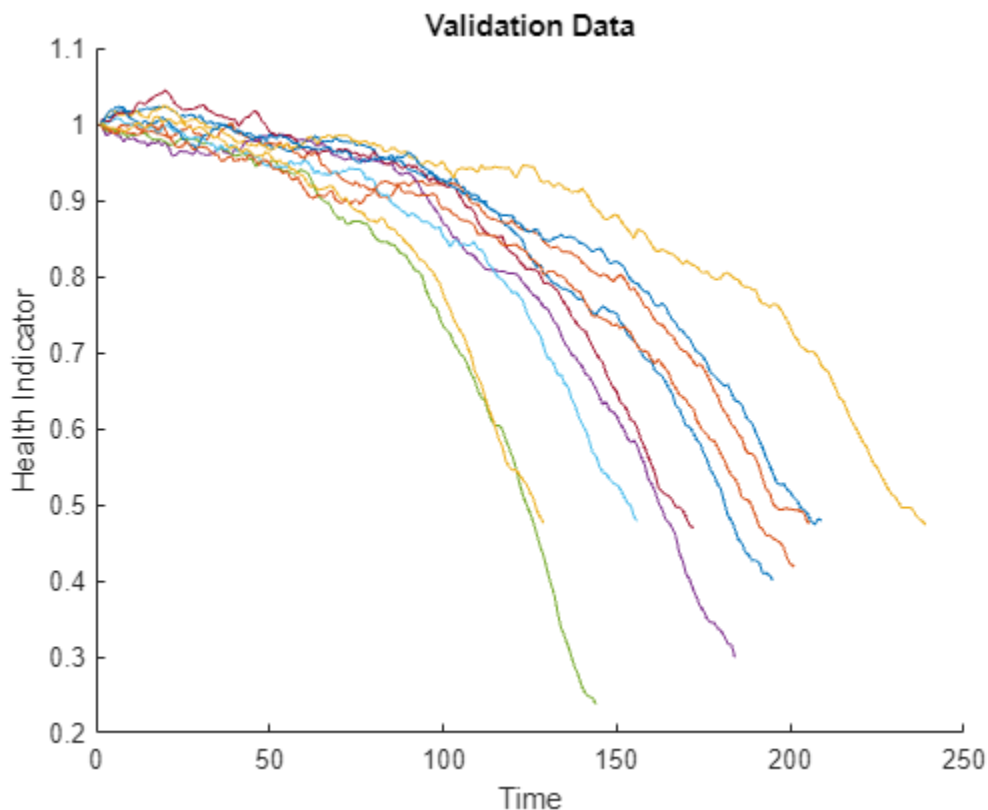

Build Similarity RUL Model

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

$$score(i, j) = exp(-d(i, j)^2)$$

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);
```
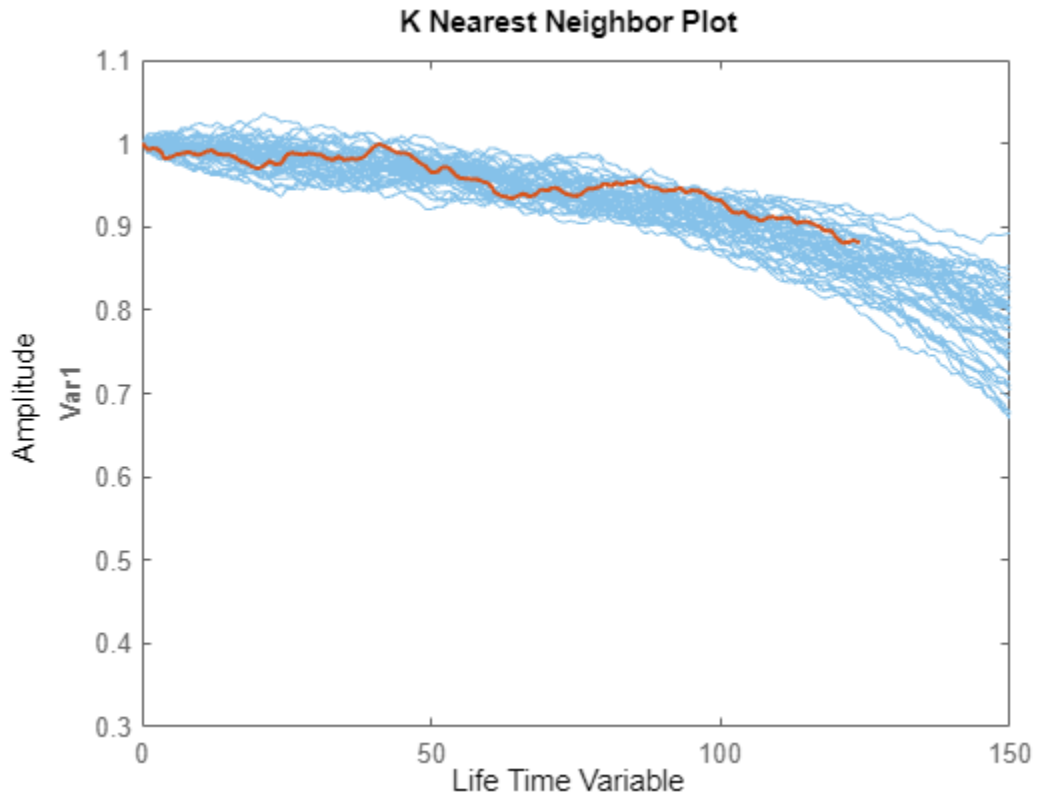
Visualize the estimated RUL compared to the true RUL and the probability distribution of the estimated RUL.

```
figure
helperPlotRULDistribution(trueRUL, estRUL, pdfRUL, ciRUL)
```
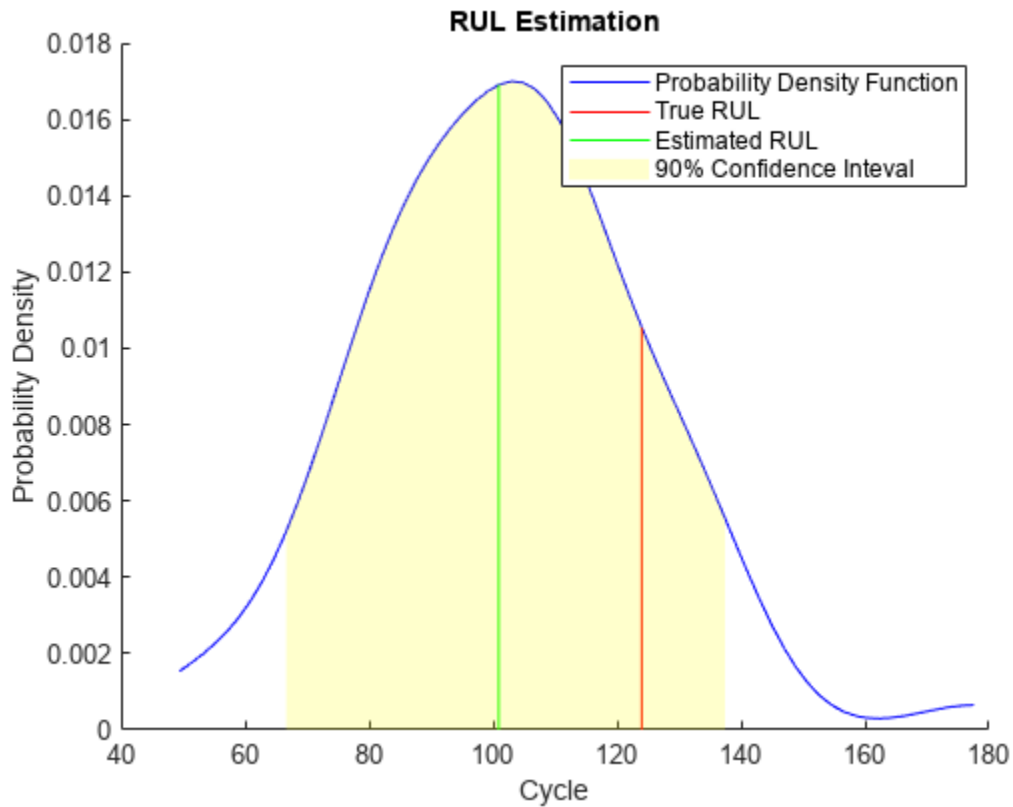
There is a relatively large error between the estimated RUL and the true RUL when the machine is in an intermediate health stage. In this example, the most similar 10 curves are close at the beginning, but bifurcate when they approach the failure state, resulting in roughly two modes in the RUL distribution.

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);
```

K Nearest Neighbor Plot
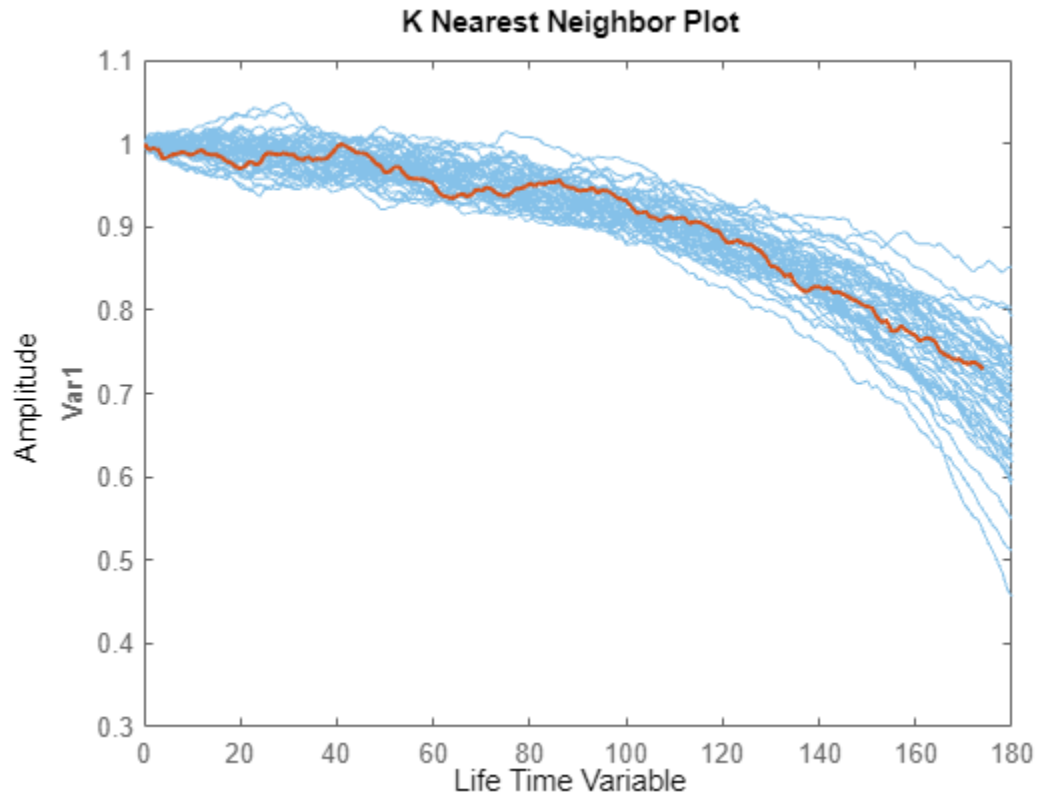
```
figure
helperPlotRULDistribution(trueRUL, estRUL, pdfRUL, ciRUL)
```
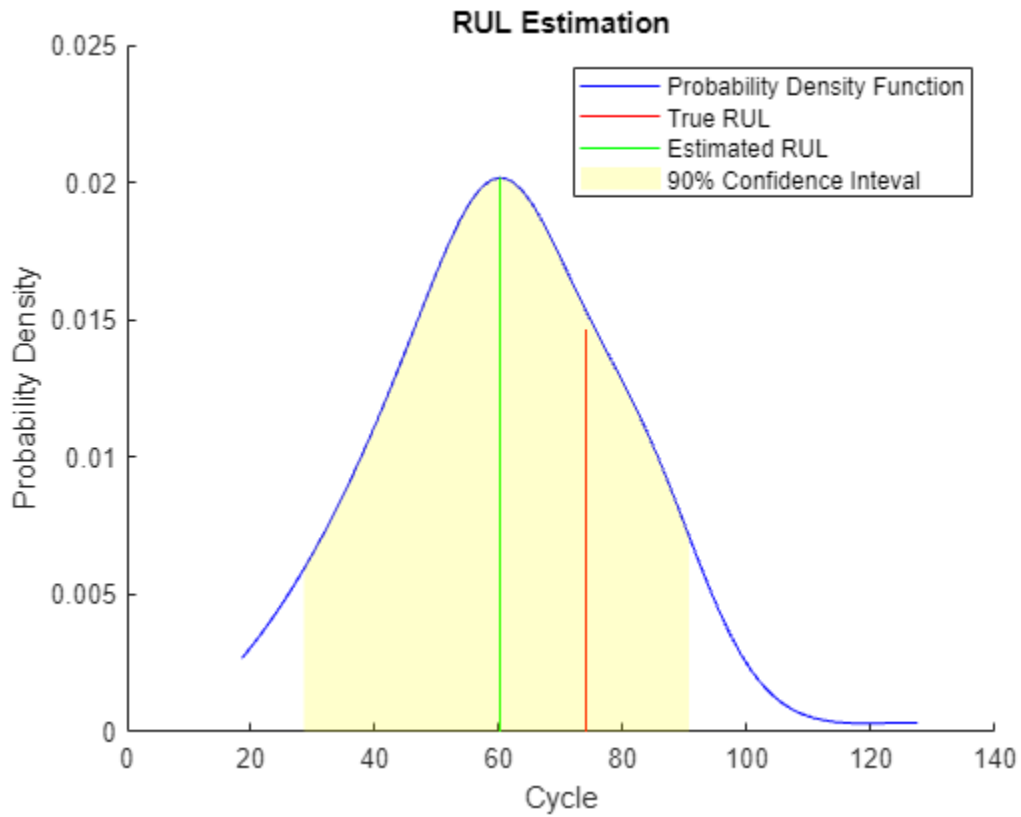
When more data is observed, the RUL estimation is enhanced.

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);
```

K Nearest Neighbor Plot
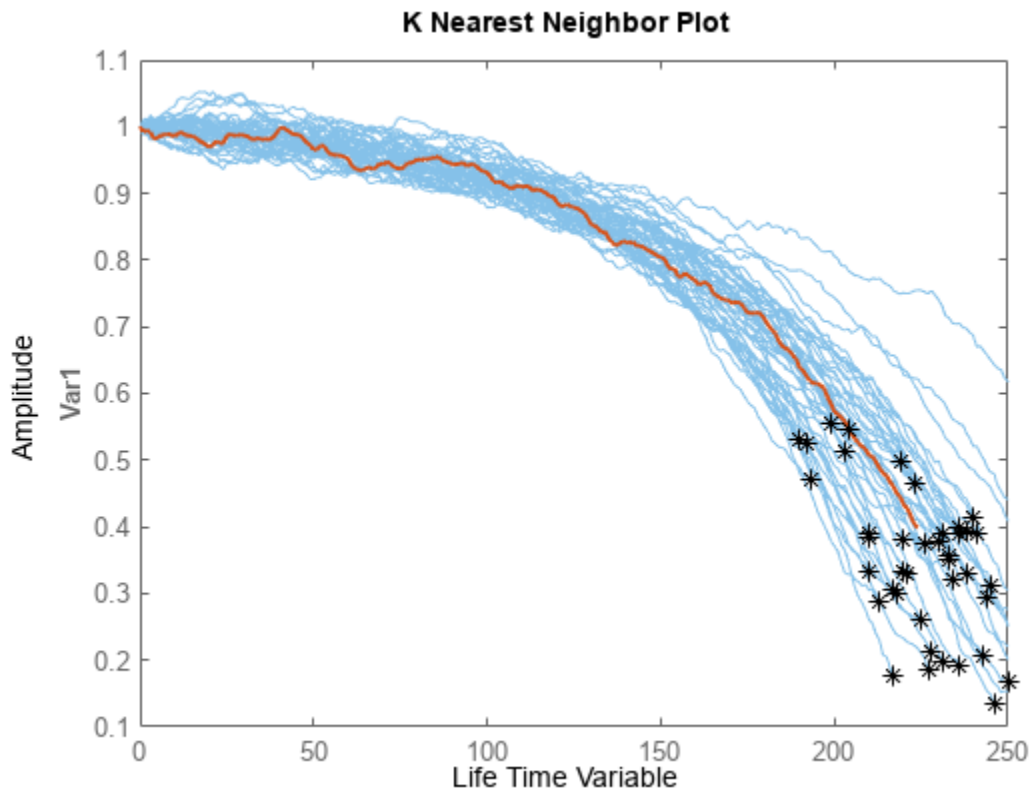
```
figure
helperPlotRULDistribution(trueRUL, estRUL, pdfRUL, ciRUL)
```
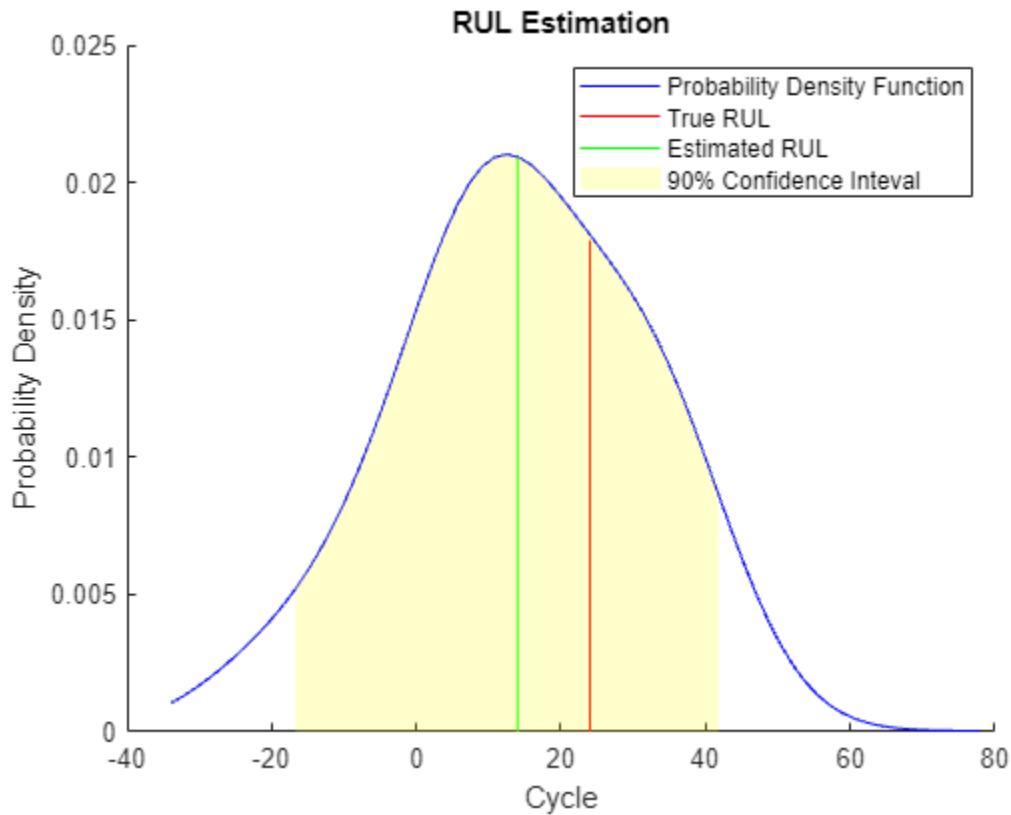
When the machine is close to failure, the RUL estimation is even more enhanced in this example.

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)
```
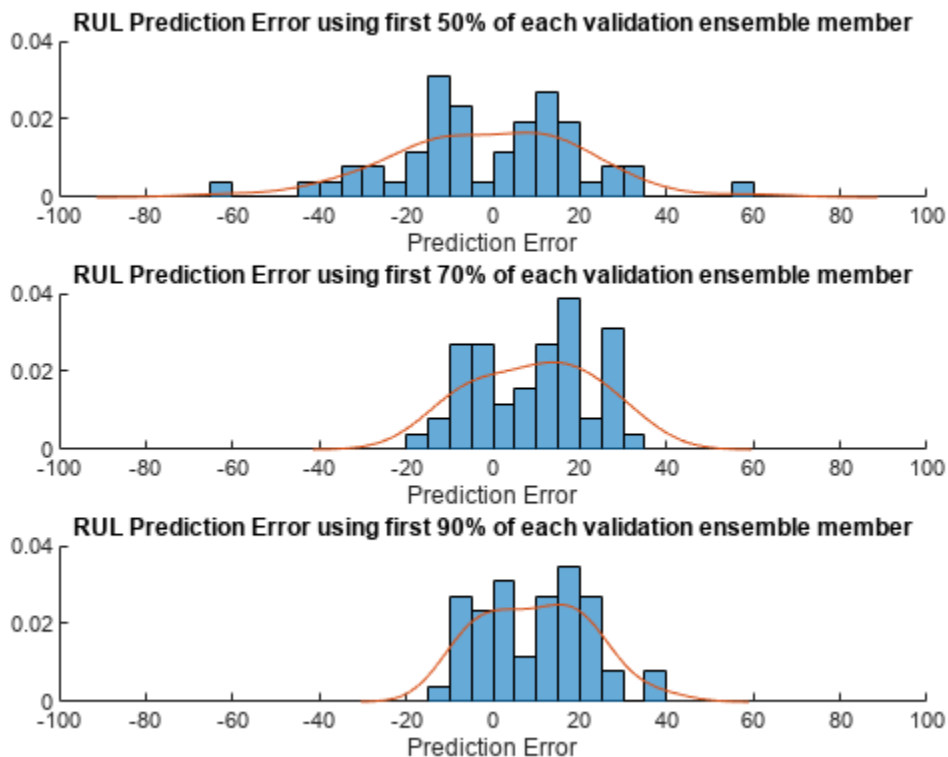

Plot the prediction error in a box plot to visualize the median, 25-75 quantile and outliers.

```
figure
boxplot(error, 'Labels', {'50%', '70%', '90%'})
```

```
ylabel('Prediction Error')
title('Prediction error using different percentages of each validation ensemble member')
```
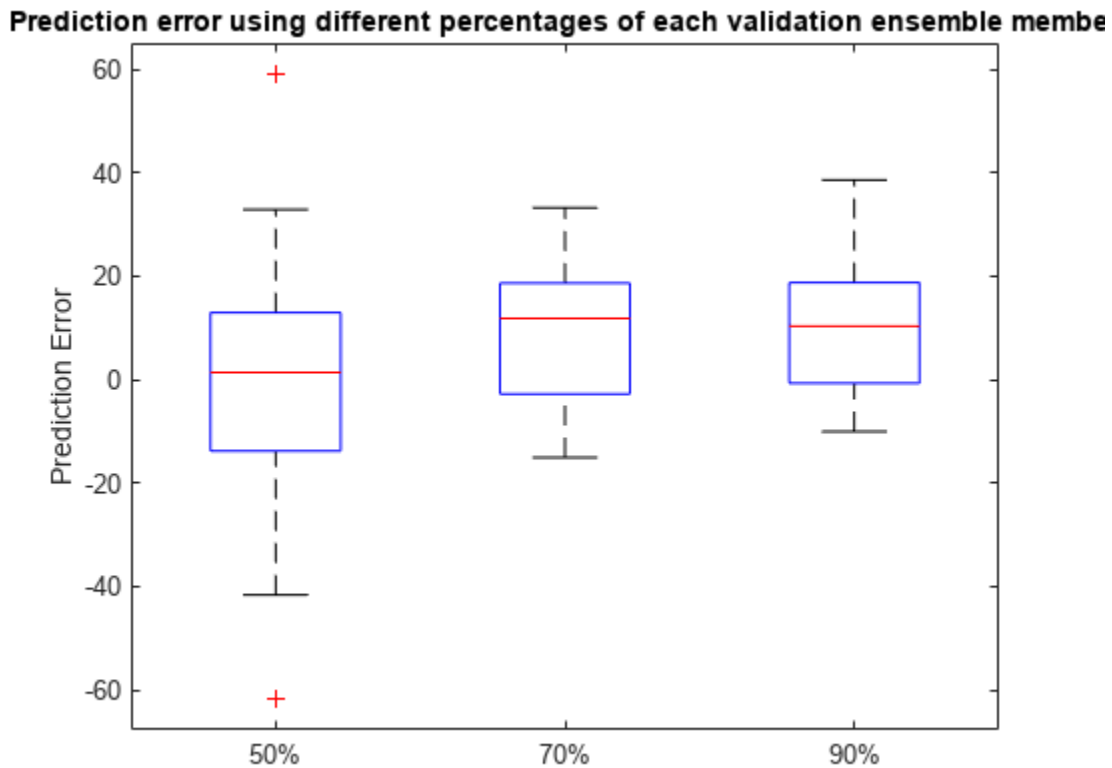

Compute and visualize the mean and standard deviation of the prediction error.

```
errorMean = mean(error)
```

errorMean = *1×3*

```
   -1.1217    9.5186    9.6540
```

```
errorMedian = median(error)
```

errorMedian = *1×3*

```
    1.3798   11.8172   10.3457
```

```
errorSD = std(error)
```

errorSD = *1×3*

```
   21.7315   13.5194   12.3083
```

```
figure
errorbar([50 70 90], errorMean, errorSD, '-o', 'MarkerEdgeColor','r')
xlim([40, 100])
```

5-33

```
xlabel('Percentage of validation data used for RUL prediction')
ylabel('Prediction Error')
legend('Mean Prediction Error with 1 Standard Deviation Eror bar', 'Location', 'south')
```
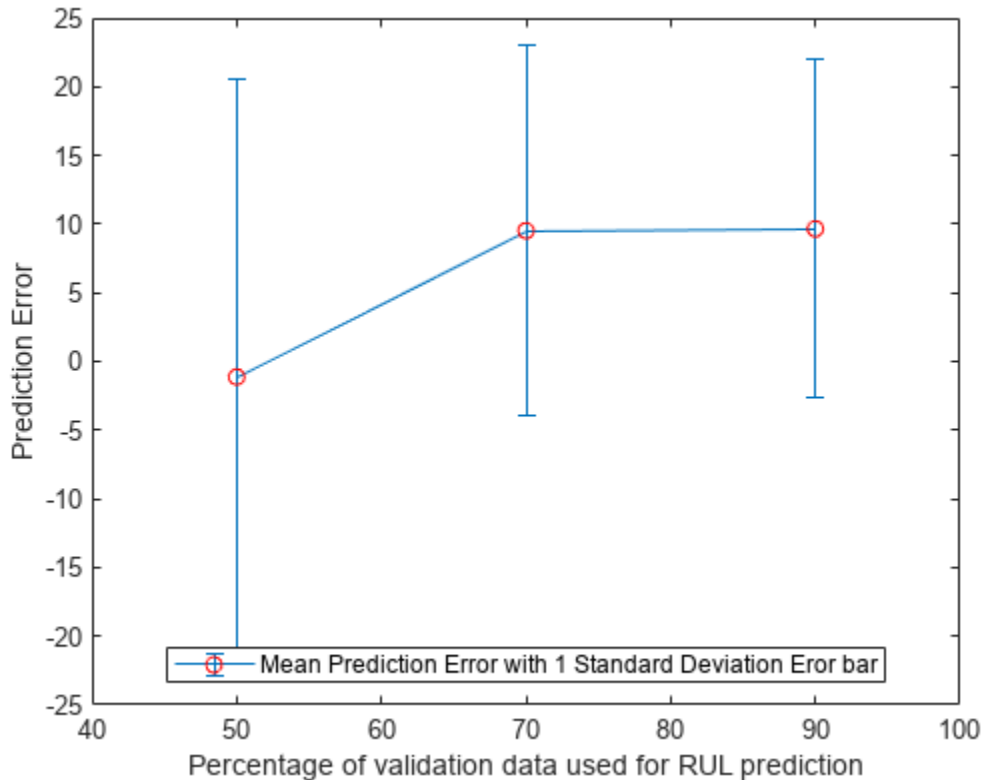

It is shown that the error becomes more concentrated around 0 (less outliers) as more data is observed.

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

```matlab
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, centerstats)
% 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 5-7

- "Wind Turbine High-Speed Bearing Prognosis" on page 5-37

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 in the supporting files associated with this example 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)')
```
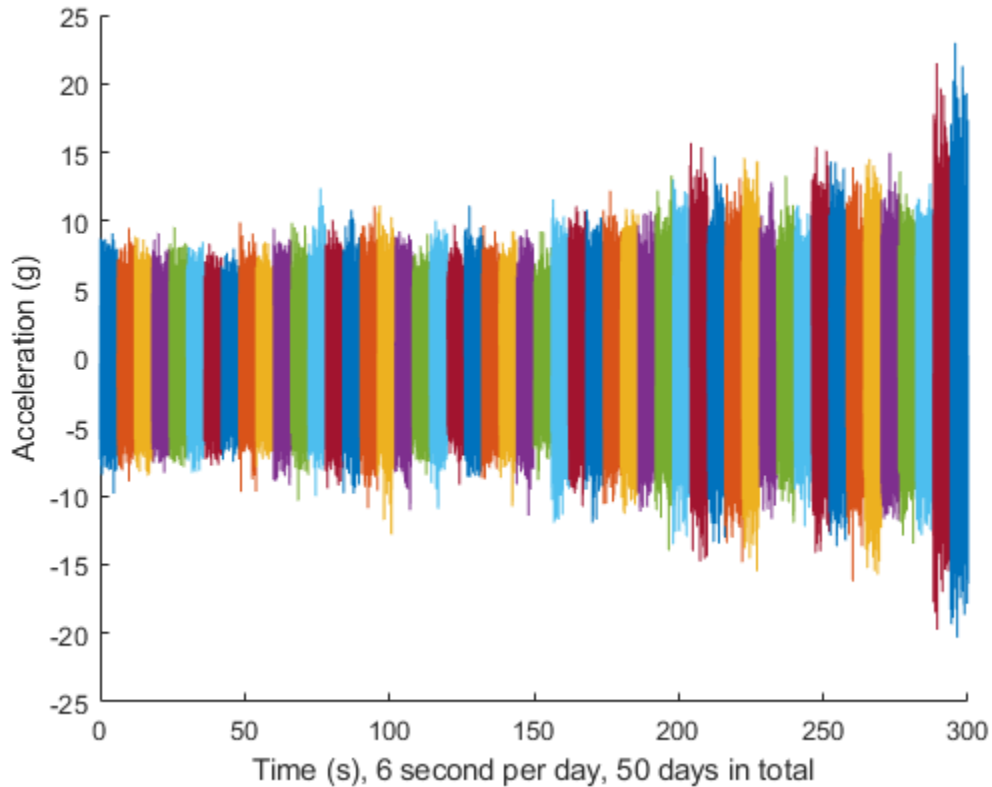
The vibration signals in time domain reveals an increasing trend of the signal impulsiveness. Indicators quantifying the impulsiveness of the signal, such as kurtosis, peak-to-peak value, crest factors *etc.*, are potential prognostic features for this wind turbine bearing dataset [2].

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)')
```
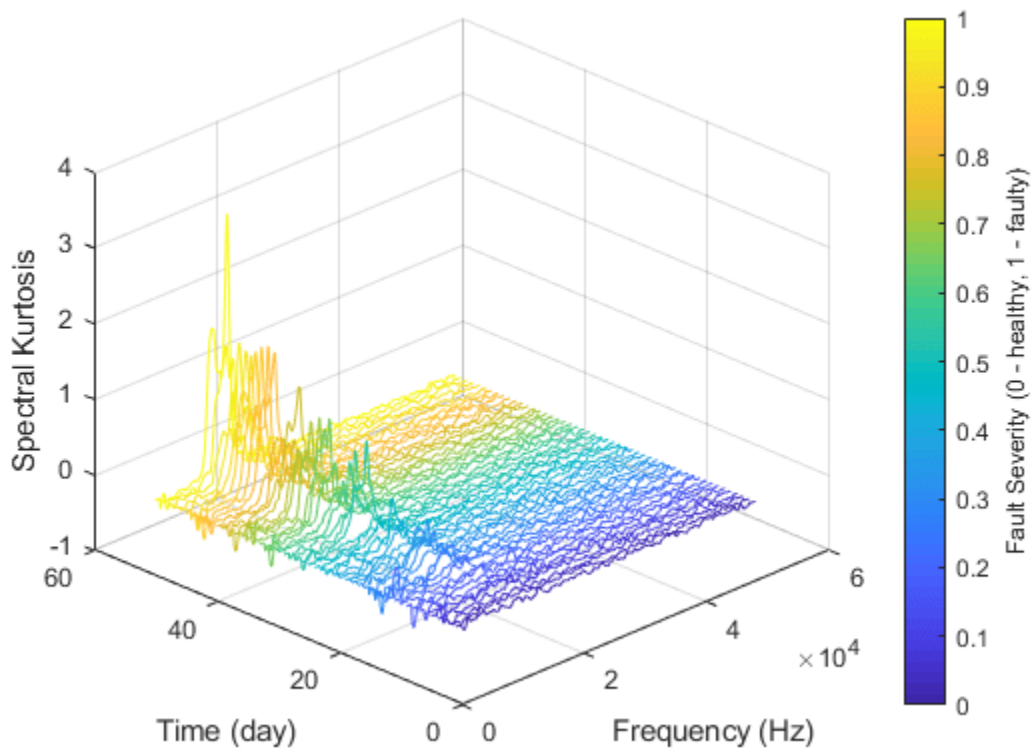

Fault Severity indicated in colorbar is the measurement date normalized into 0 to 1 scale. It is observed that the spectral kurtosis value around 10 kHz gradually increases as the machine condition degrades. Statistical features of the spectral kurtosis, such as mean, standard deviation *etc.,* will 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.

```matlab
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.

```matlab
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.

```matlab
hsbearing.SelectedVariables = ["Date", "Mean", "Std", "Skewness", "Kurtosis", "Peak2Peak", ...
    "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.

```matlab
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      RMS

    _____     _____    _____   _____   _____    _____    _____

    07-Mar-2013 01:57:46  0.34605    2.2705     0.0038699     2.9956      21.621       2.2967
    08-Mar-2013 02:34:21  0.24409    2.0621     0.0030103     3.0195      19.31        2.0765
    09-Mar-2013 02:33:43  0.21873    2.1036    -0.0010289     3.0224      21.474       2.1149
    10-Mar-2013 03:01:02  0.21372    2.0081     0.001477      3.0415      19.52        2.0194
    11-Mar-2013 03:00:24  0.21518    2.0606     0.0010116     3.0445      21.217       2.0718
    12-Mar-2013 06:17:10  0.29335    2.0791    -0.008428      3.018       20.05        2.0997
    13-Mar-2013 06:34:04  0.21293    1.972     -0.0014294     3.0174      18.837       1.9834
    14-Mar-2013 06:50:41  0.24401    1.8114     0.0022161     3.0057      17.862       1.8278
    15-Mar-2013 06:50:03  0.20984    1.9973     0.001559      3.0711      21.12        2.0083
    16-Mar-2013 06:56:43  0.23318    1.9842    -0.0019594     3.0072      18.832       1.9979
    17-Mar-2013 06:56:04  0.21657    2.113     -0.0013711     3.1247      21.858       2.1241
    17-Mar-2013 18:47:56  0.19381    2.1335    -0.012744      3.0934      21.589       2.1423
    18-Mar-2013 18:47:15  0.21919    2.1284    -0.0002039     3.1647      24.051       2.1396
    20-Mar-2013 00:33:54  0.35865    2.2536    -0.002308      3.0817      22.633       2.2819
    21-Mar-2013 00:33:14  0.1908     2.1782    -0.00019286    3.1548      25.515       2.1865
    22-Mar-2013 00:39:50  0.20569    2.1861     0.0020375     3.2691      26.439       2.1958
      ⋮
```

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')
```
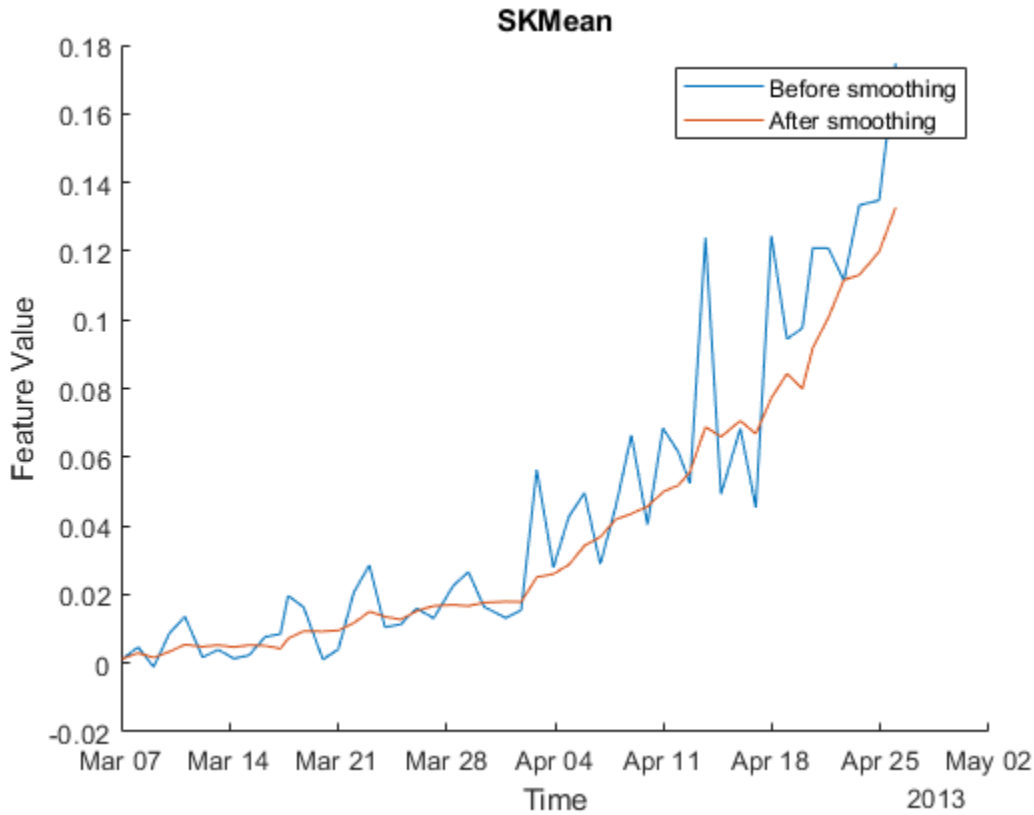
Moving mean smoothing introduces a time delay of the signal, but the delay effect can be mitigated by selecting proper threshold in the RUL prediction.

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

$$\text{Monotonicity}(x_i) = \frac{1}{m} \sum_{j=1}^{m} \frac{\left| \text{number of positive diff}\left(x_i^j\right) - \text{number of negative diff}\left(x_i^j\right) \right|}{n - 1}$$

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. $\mathrm{diff}\left(x_i^j\right) = x_i^j(t) - x_i^j(t-1)$, 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');
```
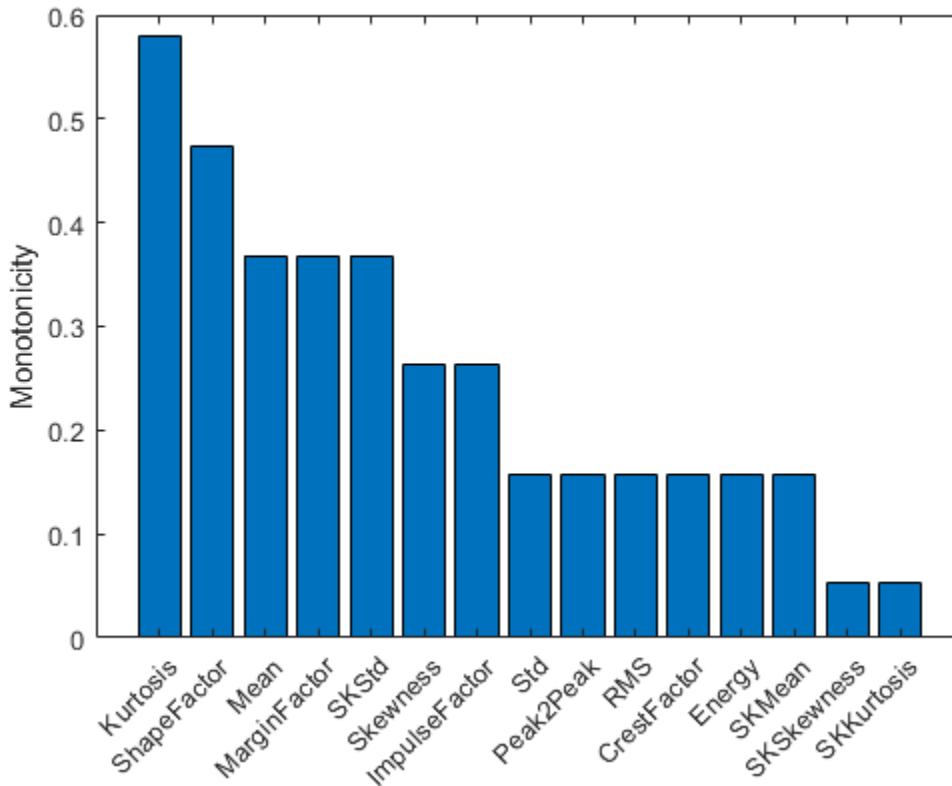

Kurtosis of the signal is the top feature based on the monotonicity.

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	SKStd
07-Mar-2013 01:57:46	0.34605	2.9956	1.2535	3.3625	0.025674
08-Mar-2013 02:34:21	0.29507	3.0075	1.254	3.5428	0.023281
09-Mar-2013 02:33:43	0.26962	3.0125	1.254	3.6541	0.023089
10-Mar-2013 03:01:02	0.25565	3.0197	1.2544	3.7722	0.025931
11-Mar-2013 03:00:24	0.24756	3.0247	1.2546	3.7793	0.027183

```
12-Mar-2013 06:17:10    0.25519    3.0236    1.2544    3.7479    0.027374
13-Mar-2013 06:34:04     0.233     3.0272    1.2545    3.8282    0.027977
14-Mar-2013 06:50:41    0.23299    3.0249    1.2544    3.9047     0.02824
15-Mar-2013 06:50:03     0.2315     3.033    1.2548    3.9706    0.032417
16-Mar-2013 06:56:43    0.23475    3.0273    1.2546    3.9451    0.031744
17-Mar-2013 06:56:04    0.23498    3.0407    1.2551    3.9924    0.032691
17-Mar-2013 18:47:56    0.21839    3.0533    1.2557    3.9792    0.037226
18-Mar-2013 18:47:15    0.21943    3.0778    1.2567    4.0538    0.043097
20-Mar-2013 00:33:54    0.23854    3.0905    1.2573    3.9658    0.047942
21-Mar-2013 00:33:14    0.23537    3.1044    1.2578    3.9862    0.051023
22-Mar-2013 00:39:50    0.23079    3.1481    1.2593     4.072    0.058908
        ⋮
```

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 ')'])
```
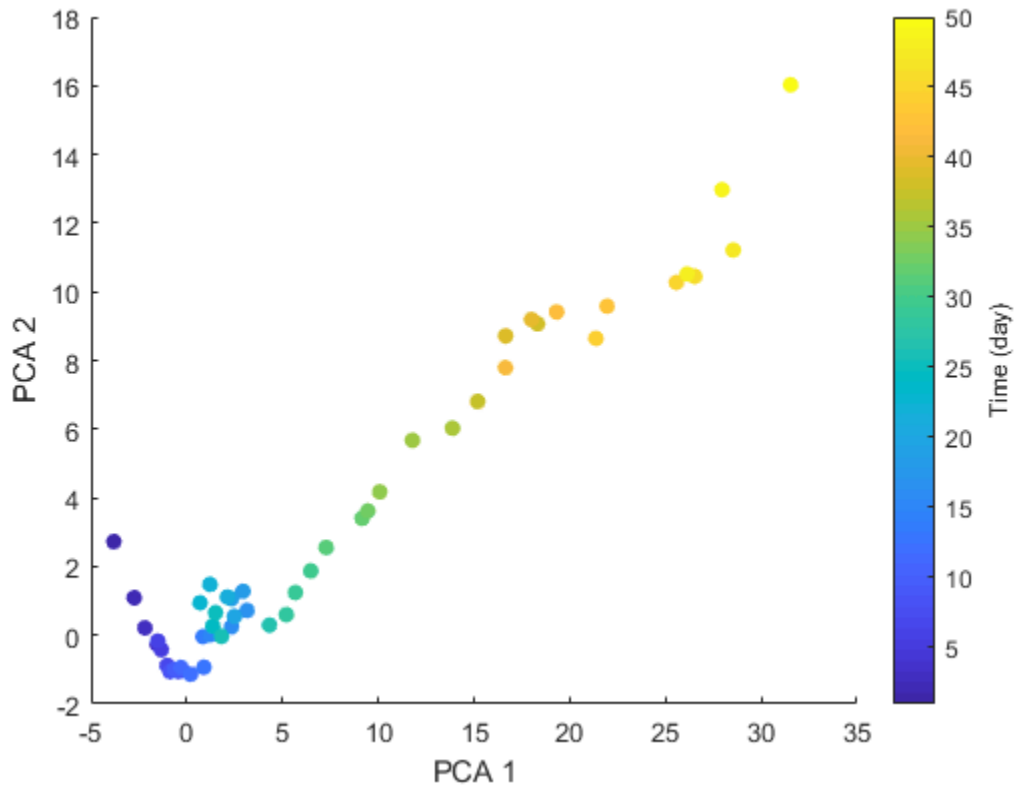
The plot indicates that the first principal component is increasing as the machine approaches to failure. Therefore, the first principal component is a promising fused health indicator.

```
healthIndicator = PCA1;
```

Visualize the health indicator.

```
figure
plot(featureSelected.Date, healthIndicator, '-o')
xlabel('Time')
title('Health Indicator')
```
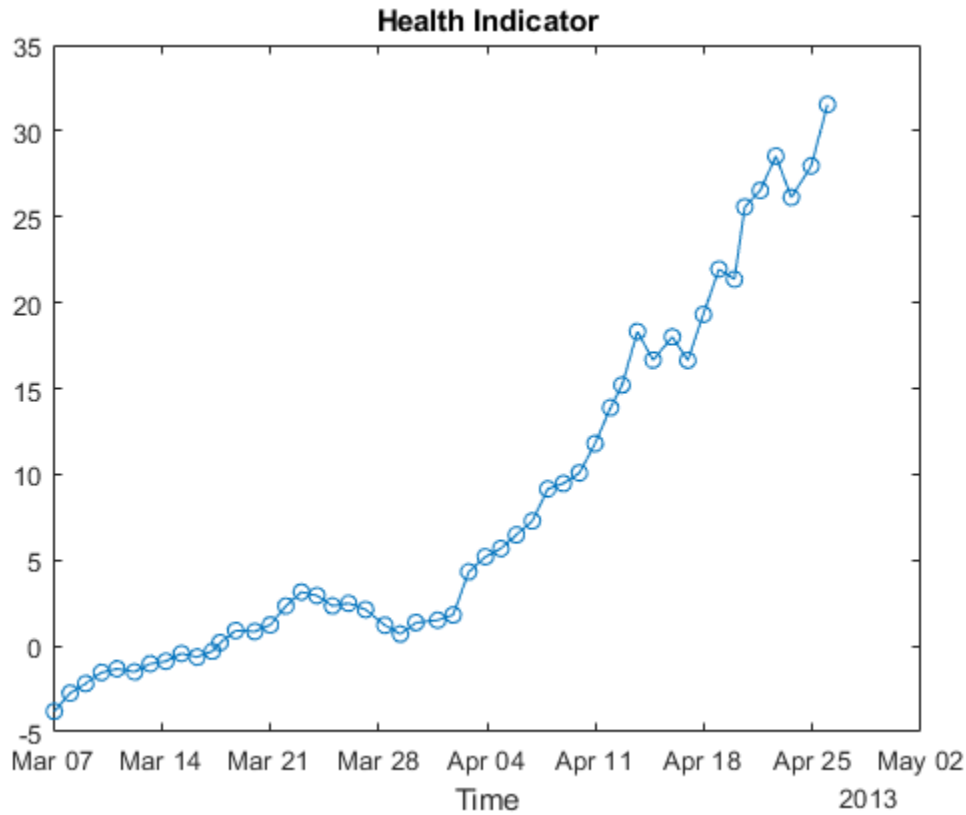
Fit Exponential Degradation Models for Remaining Useful Life (RUL) Estimation

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 $N\big(0, \sigma^2\big)$.

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
```


Day 49: Degradation detected!
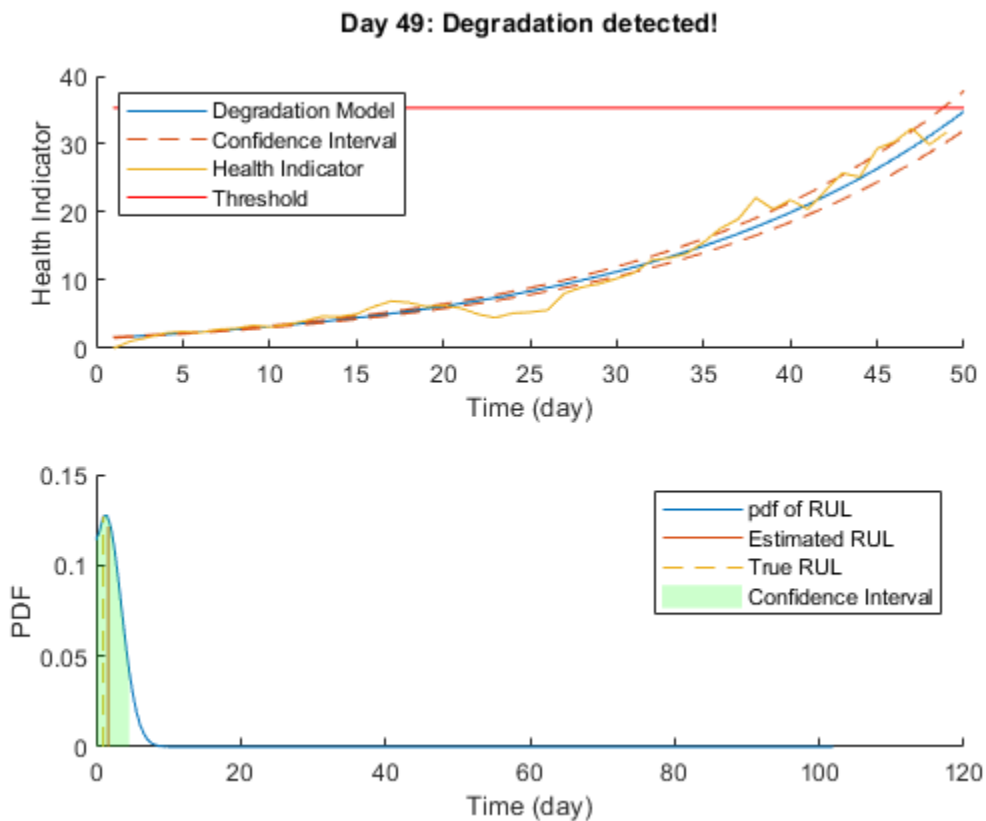
Here is the animation of the real-time RUL estimation.
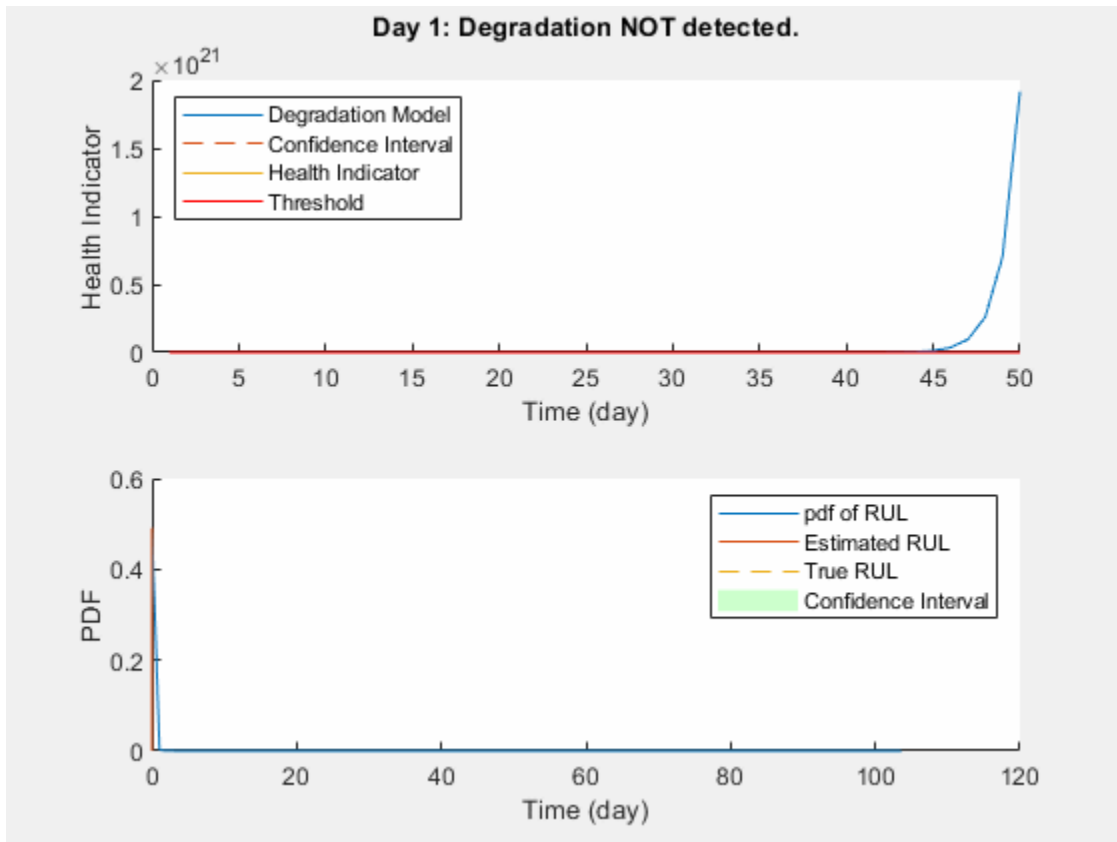
Performance Analysis

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')
```
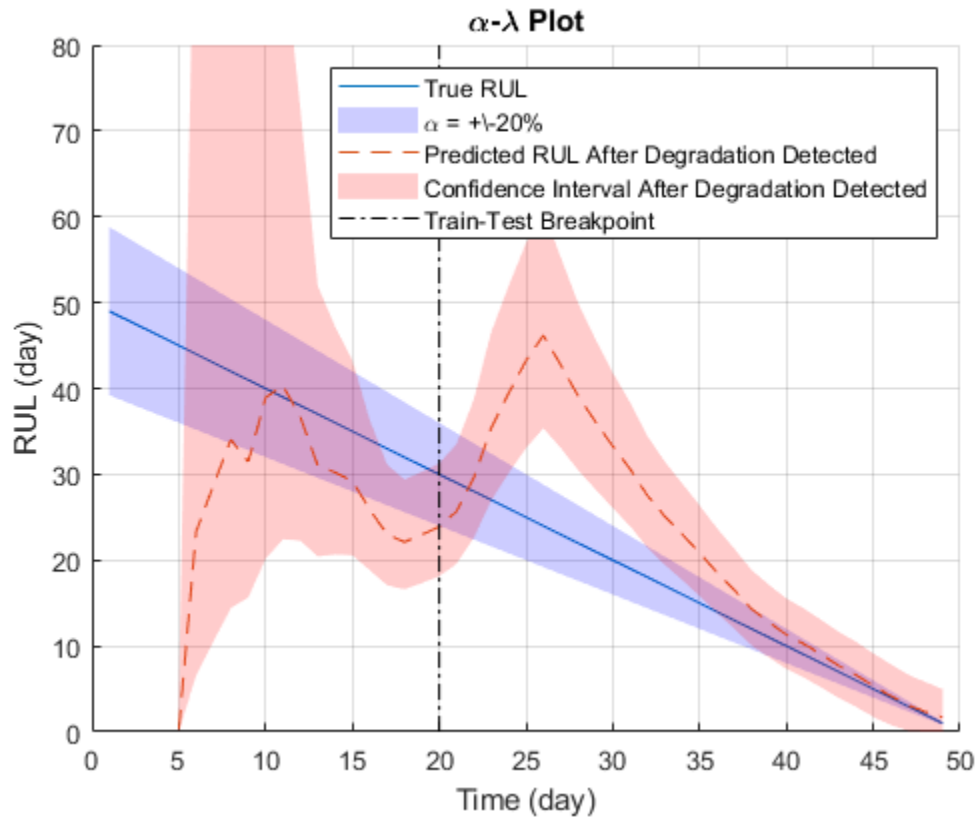
Since the preset prior does not reflect the true prior, the model usually need a few time steps to adjust to a proper parameter distribution. The prediction becomes more accurate as more data points are available.

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) '%'])
```

Probability within α bound, α = 20%
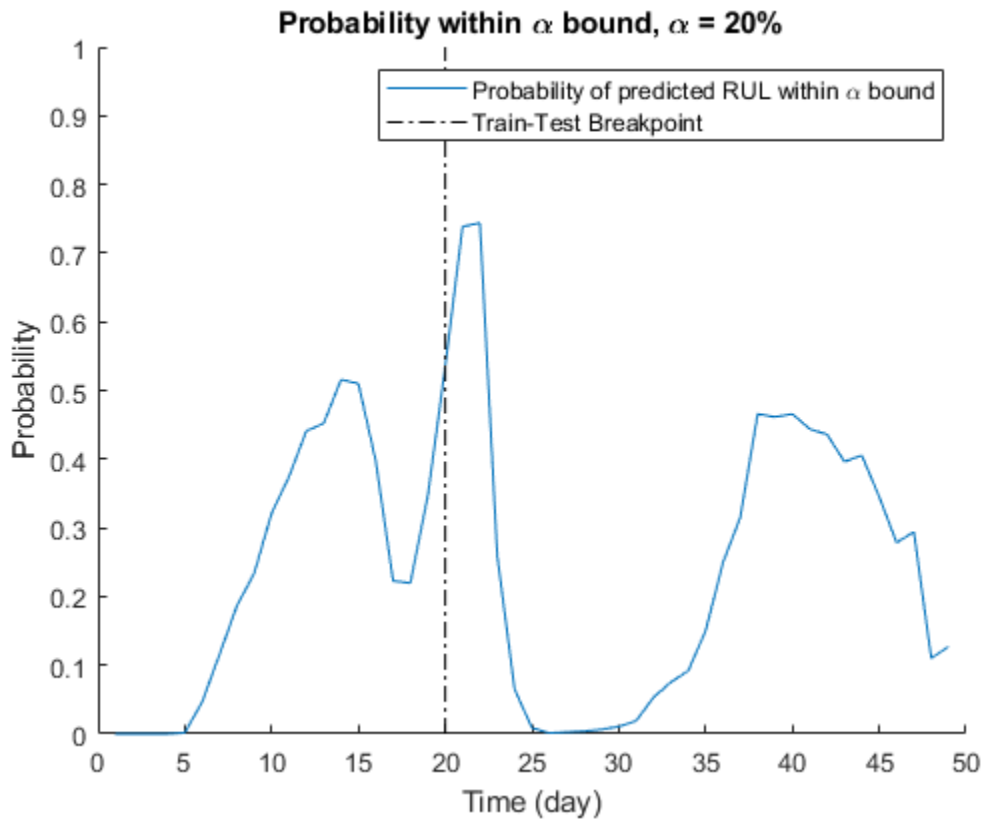
References

[1] 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

- "RUL Estimation Using RUL Estimator Models" on page 5-7
- "Similarity-Based Remaining Useful Life Estimation" on page 5-15

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` (this is a large dataset ~88MB that is downloaded from the MathWorks support files site). 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.

```matlab
url = 'https://www.mathworks.com/supportfiles/predmaint/condition-monitoring-and-prognostics-usir
websave('pdmBearingConditionMonitoringData.mat',url);
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.

```matlab
plot(expTime,defectDepthVec);
xlabel('Time (min)');
ylabel('Defect depth (m)');
```
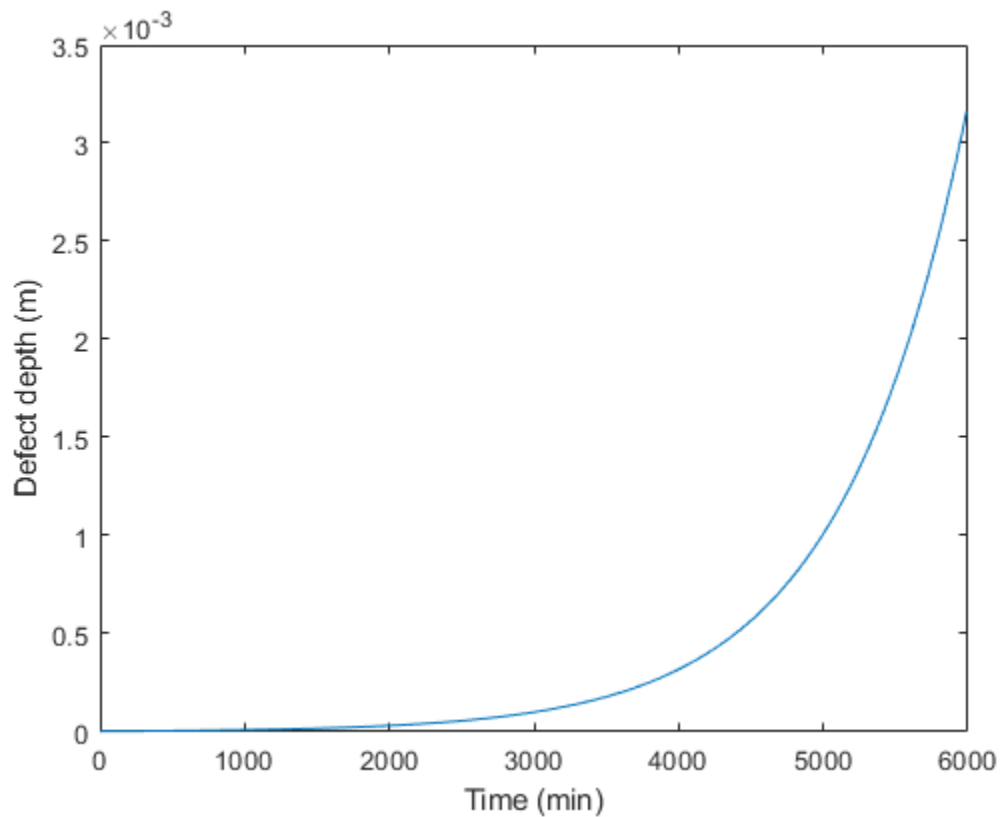
Plot the healthy and faulty data.

```
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');
```
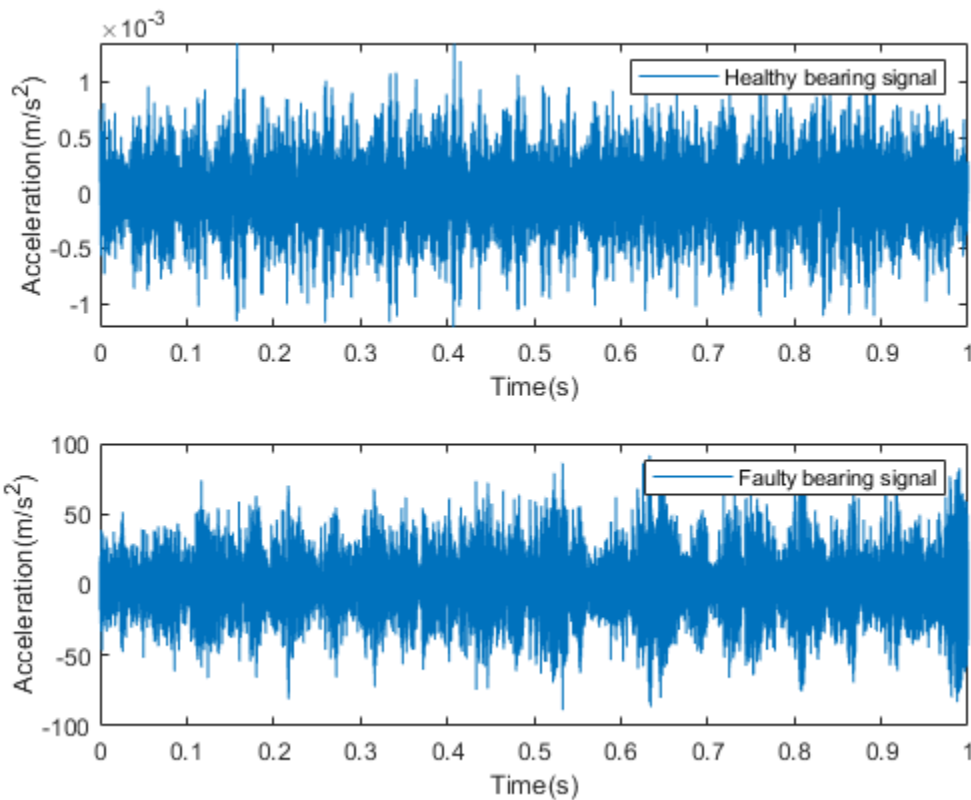
Feature Extraction

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
```

Healthy Bearing Signal Spectrogram
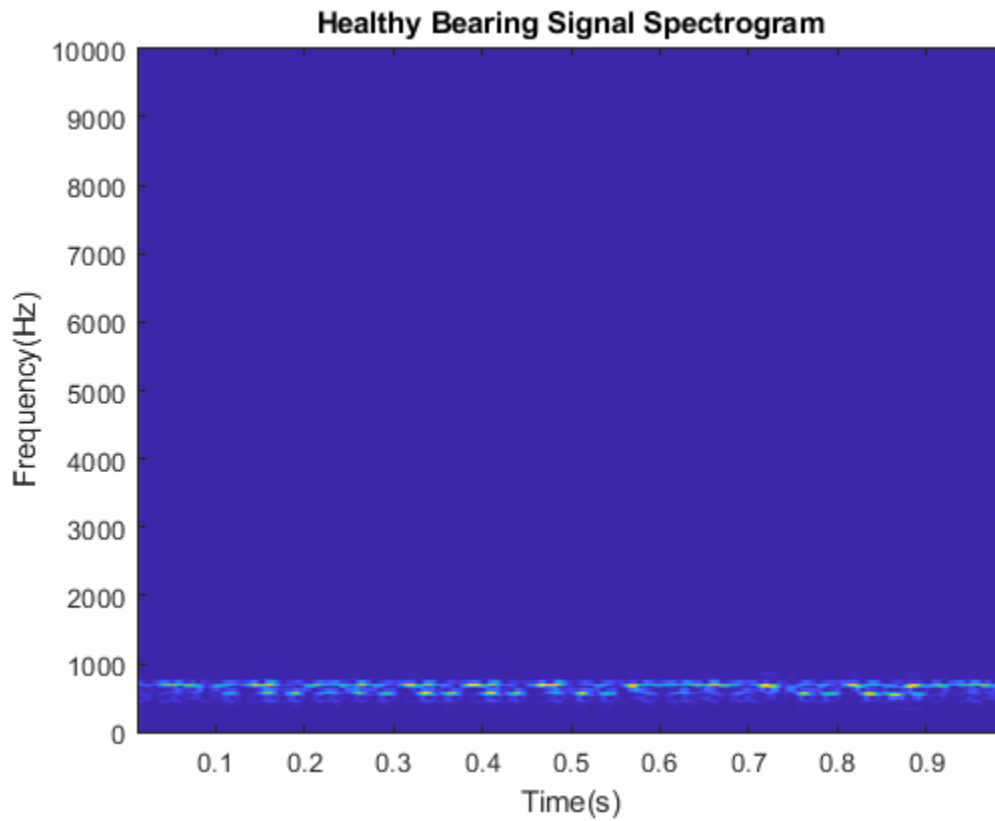
Now plot the spectrogram of vibration signals that have developed faulty patterns. You can see that signal energies are concentrated at higher frequencies.

```
[~,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
```
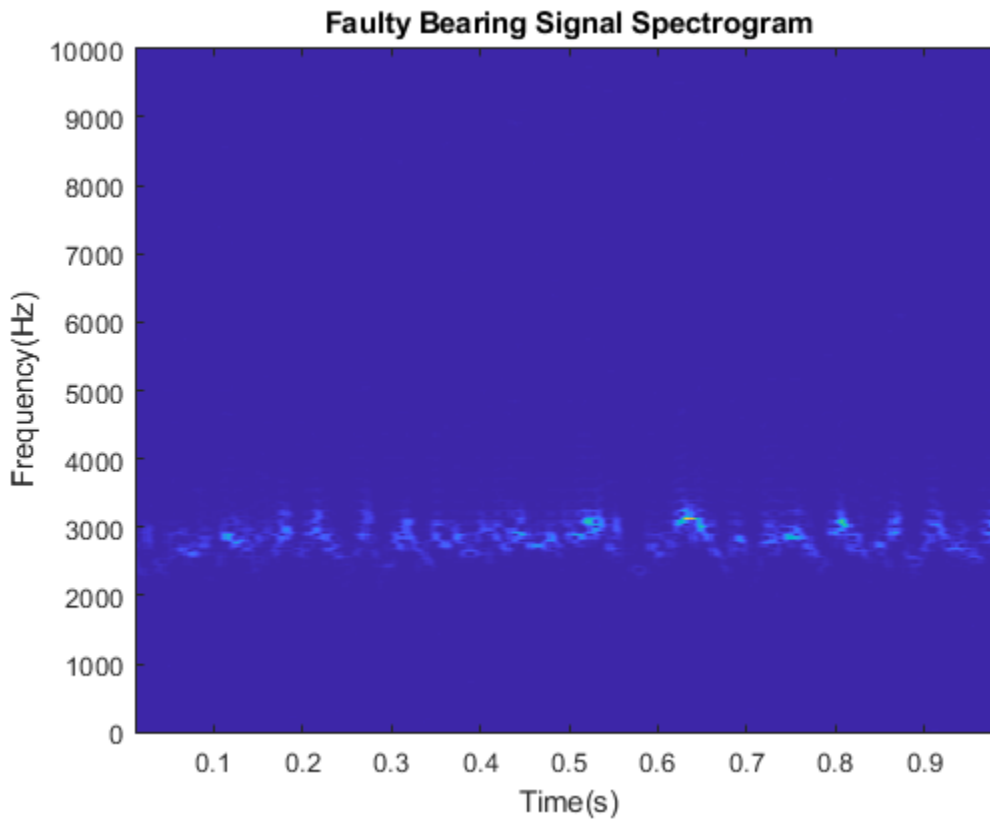
Since the spectrograms for data from healthy and faulty bearings are different, representative features can be extracted from spectrograms and used for condition monitoring and prognostics. In this example, extract mean peak frequencies from spectrograms as health indicators. Denote the spectrogram as $P(t, \omega)$. 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
```
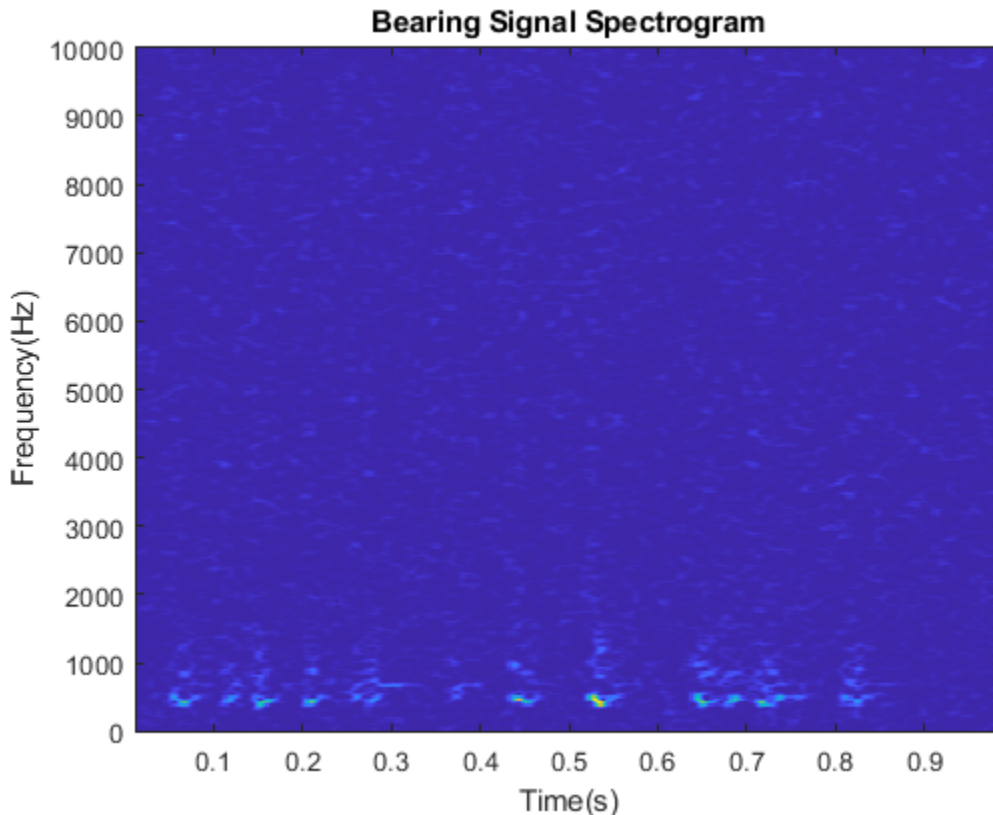

The high frequency noise components are spread all over the spectrogram. Such phenomena are mixed effects of both original vibrations and the vibrations induced by small defects. To accurately calculate mean peak frequency, filter the data to remove those high frequency components.

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
```

Filtered Bearing Signal Spectrogram
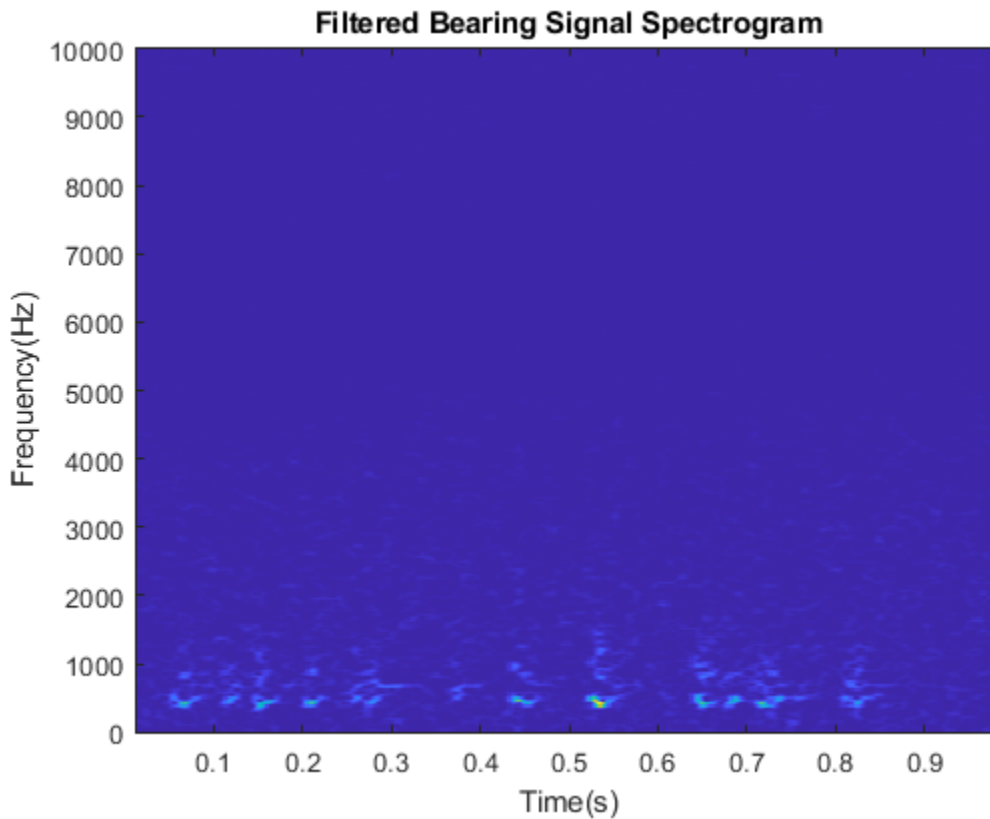
Since the mean peak frequency successfully distinguishes healthy ball bearings from faulty ball bearings, extract mean peak frequency from each segment of data.

```matlab
% 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.

```matlab
plot(expTime,meanPeakFreq);
xlabel('Time(min)');
ylabel('Mean peak frequency (Hz)');
grid on;
```
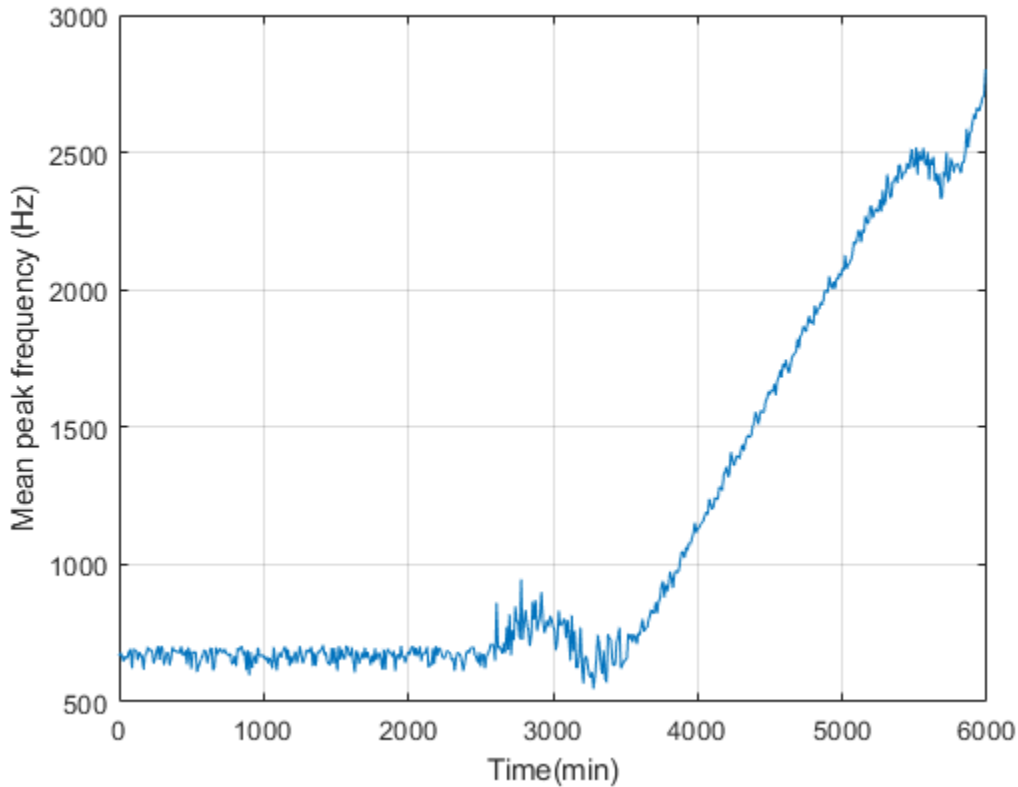
Condition Monitoring and Prognostics

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.

```
url = 'https://www.mathworks.com/supportfiles/predmaint/condition-monitoring-and-prognostics-usin
websave('pdmBearingConditionMonitoringStatistics.mat',url);
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');
```
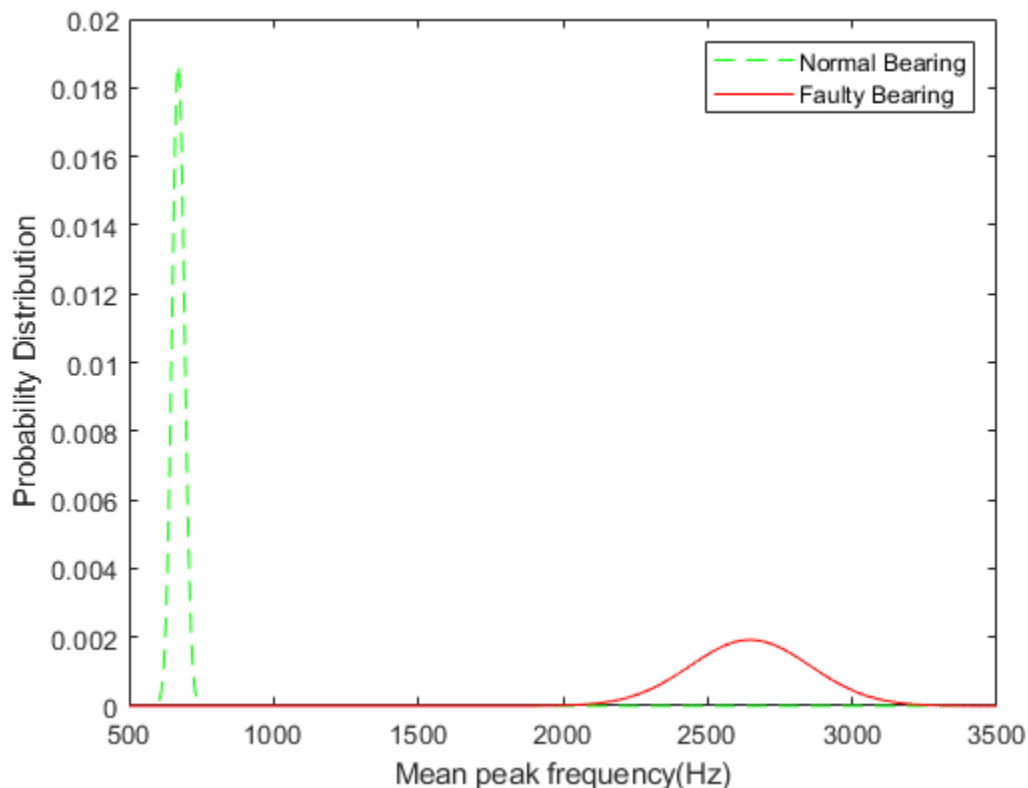

Based on this plot, set the threshold for the mean peak frequency to 2000Hz to distinguish normal bearings from faulty bearings as well as maximize the use of the bearings.

```
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)
```
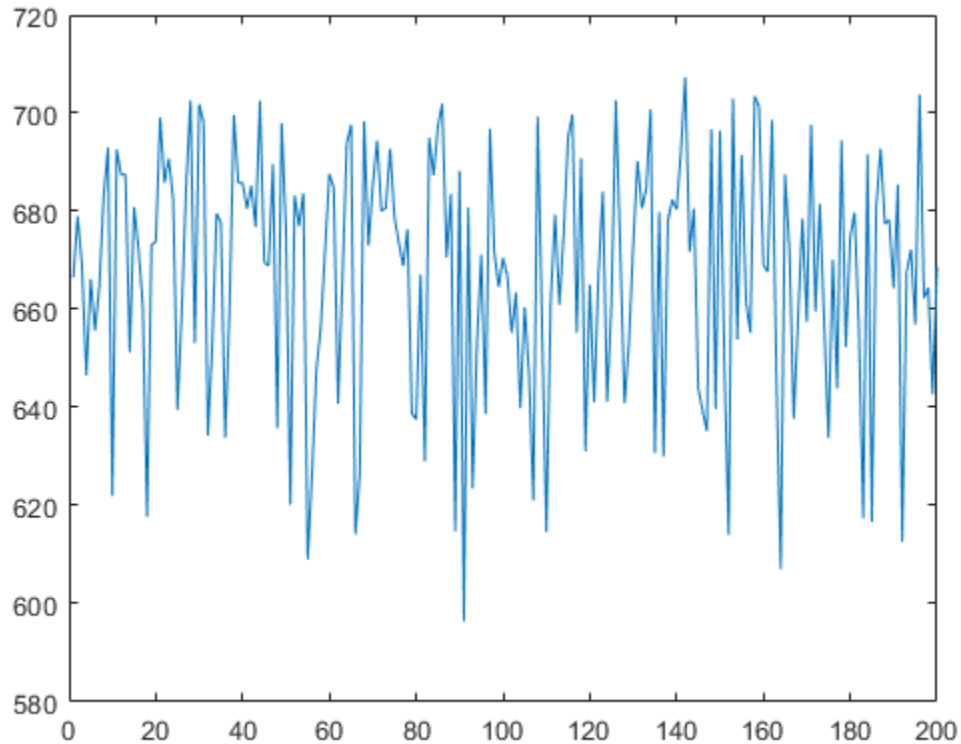
The plot shows that the initial data is a combination of constant level and noise. This is expected as initially the bearing is healthy and the mean peak frequency is not expected to change significantly

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)`
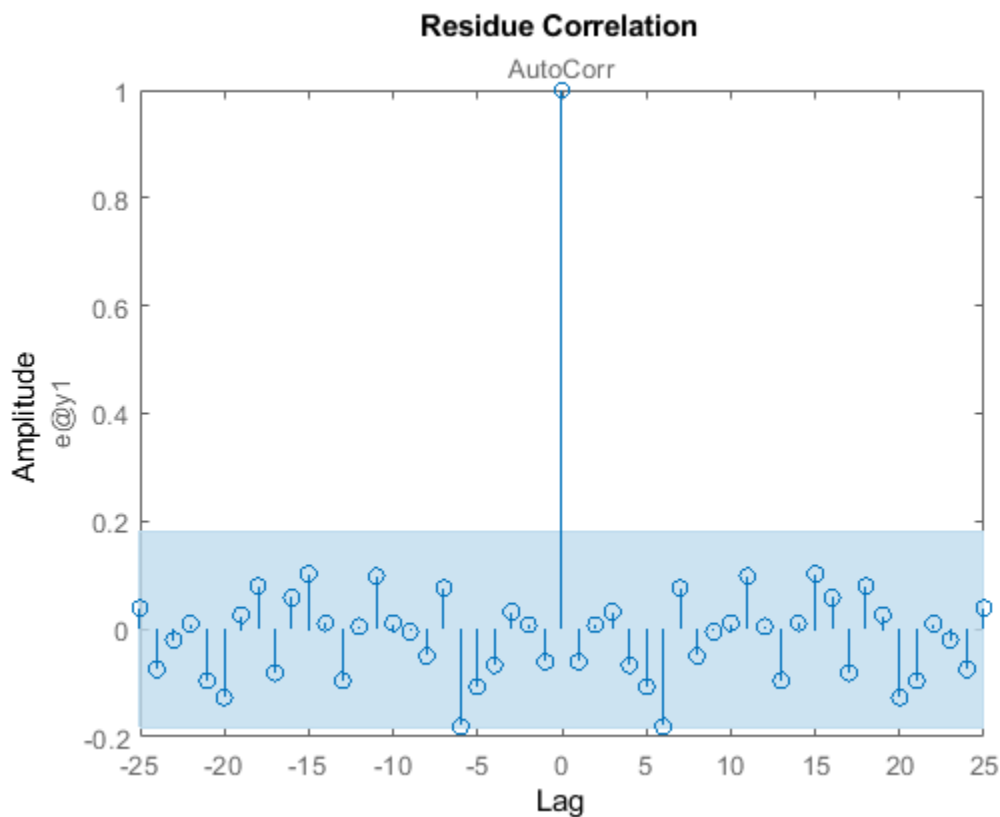
As seen, the residuals are uncorrelated and the generated model is valid.

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;
```
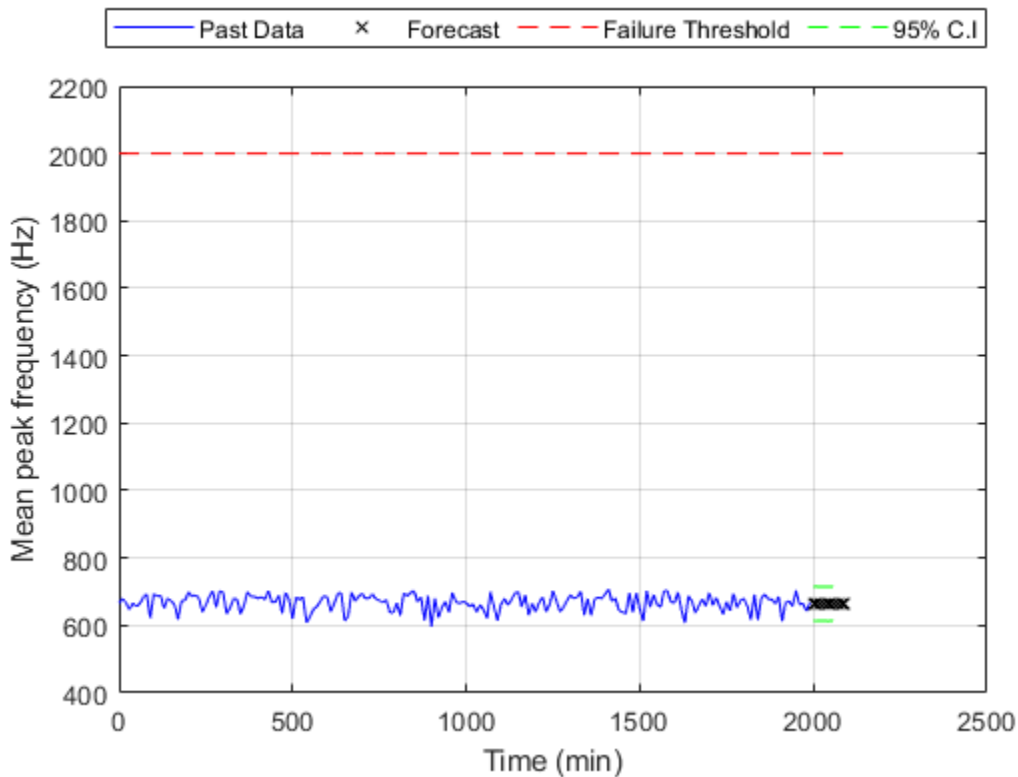

The plot shows that the forecasted values of the mean peak frequency are well below the threshold.

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.

```matlab
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), '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;

    % 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)) ' minutes f
        break;
    end
end
```
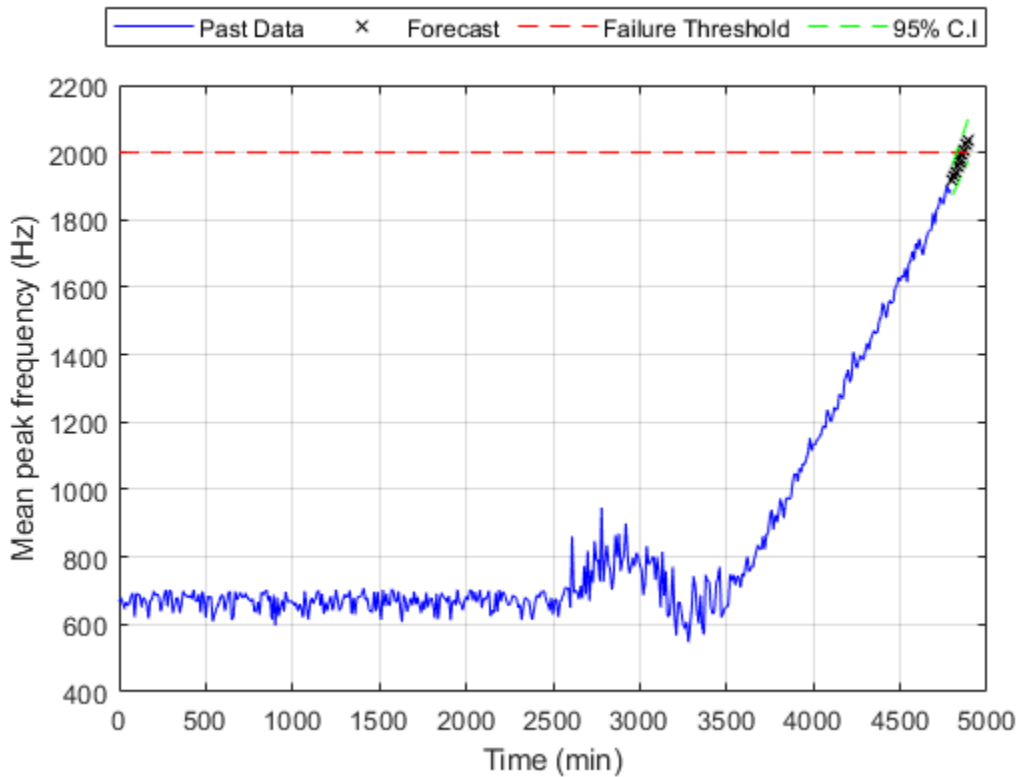
Estimated to hit failure threshold in 80 minutes from now.

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. The example runs with either Control System Toolbox™ or System Identification Toolbox™. The example does not require Predictive Maintenance Toolbox™.

Overview

Consider a battery model with the following equivalent circuit [1]
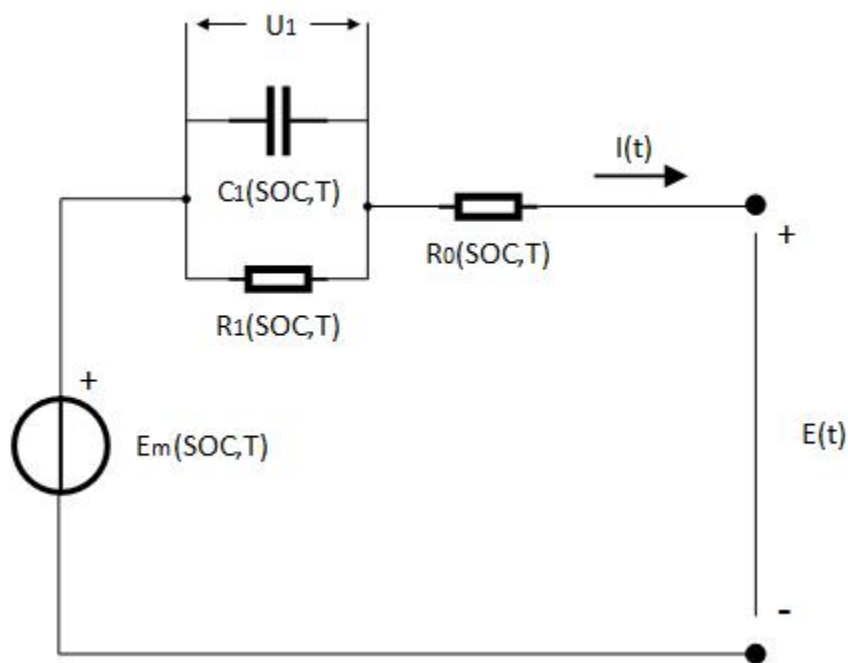
The model consists of a voltage source E_m, a series resistor R_0 and a single RC block R_1 and C_1. The battery alternates between charging and discharging cycles. In this example, you estimate the state of charge (SOC) of a battery model using measured currents, voltages and temperatures of the battery. 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/')
```
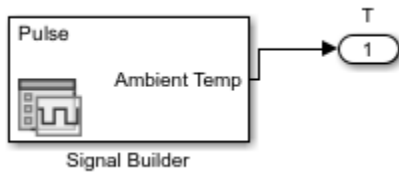
Battery Model

The battery model with thermal effects is implemented using the Simscape™ language.
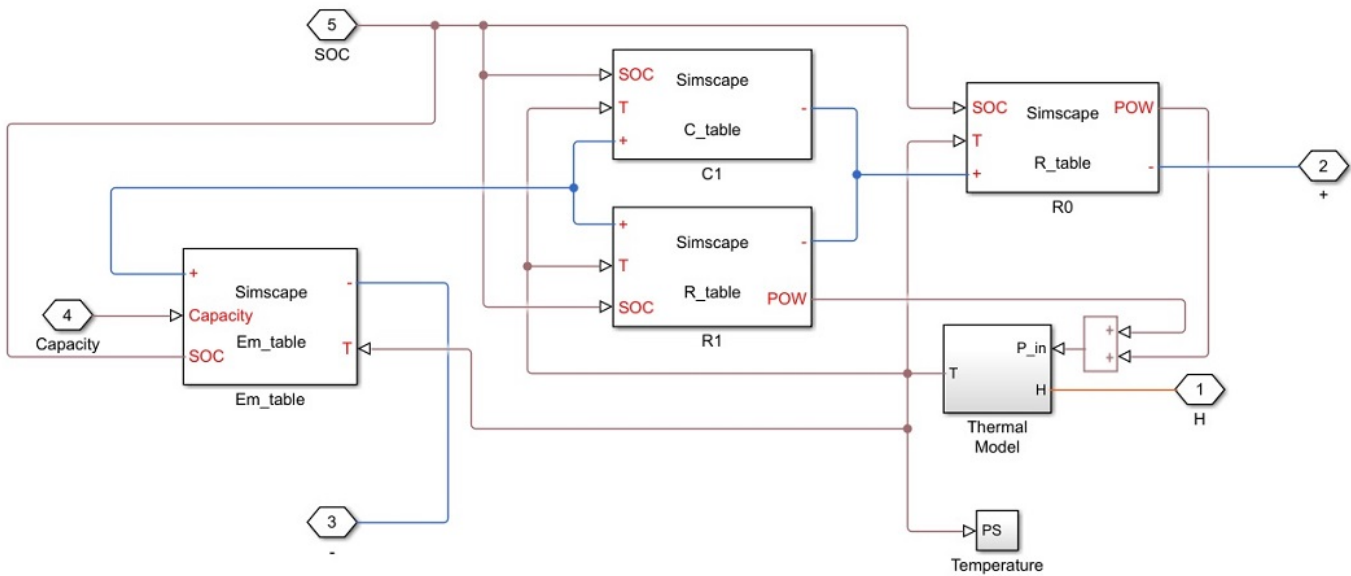
The state transition equations for the battery model are given by:

$$\frac{d}{dt}\begin{pmatrix} SOC \\ U_1 \end{pmatrix} = \begin{pmatrix} 0 \\ -\frac{1}{R_1(SOC,T_b)*C_1(SOC,T_b)}U_1 \end{pmatrix} + \begin{pmatrix} -\frac{1}{3600*C_q} \\ \frac{1}{C_1(SOC,T_b)} \end{pmatrix} I + W$$

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.
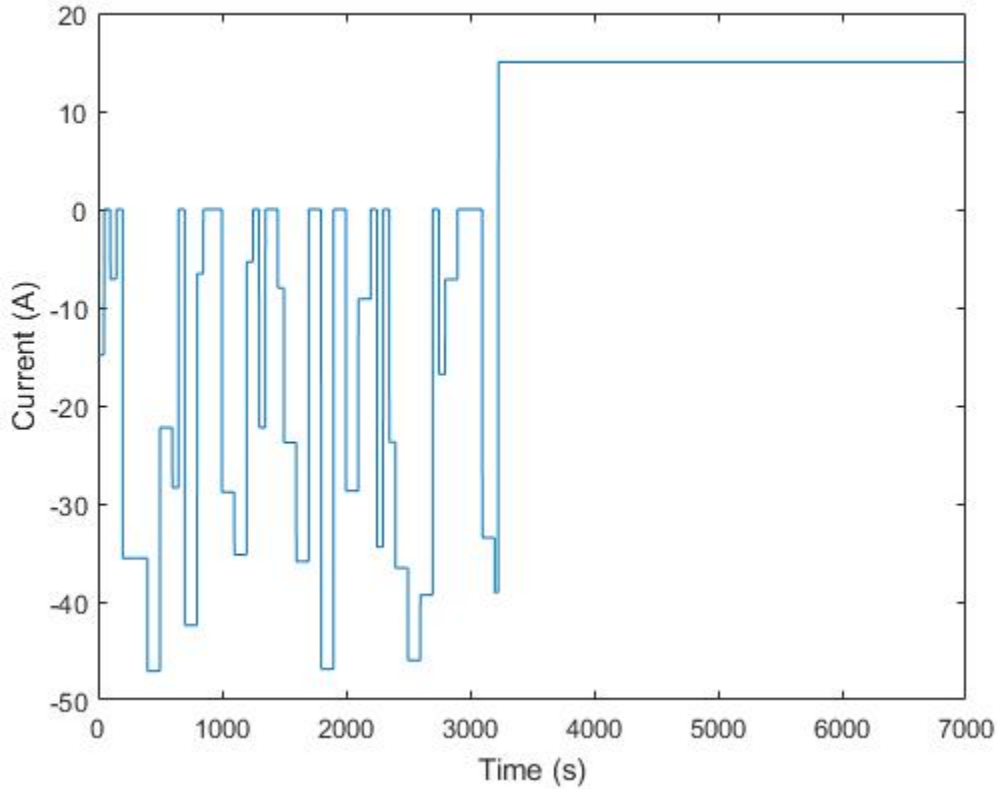
The measurement equation is given by:

$$E = E_m(SOC, T_b) - U_1 - IR_0(SOC, T_b) + V$$

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.
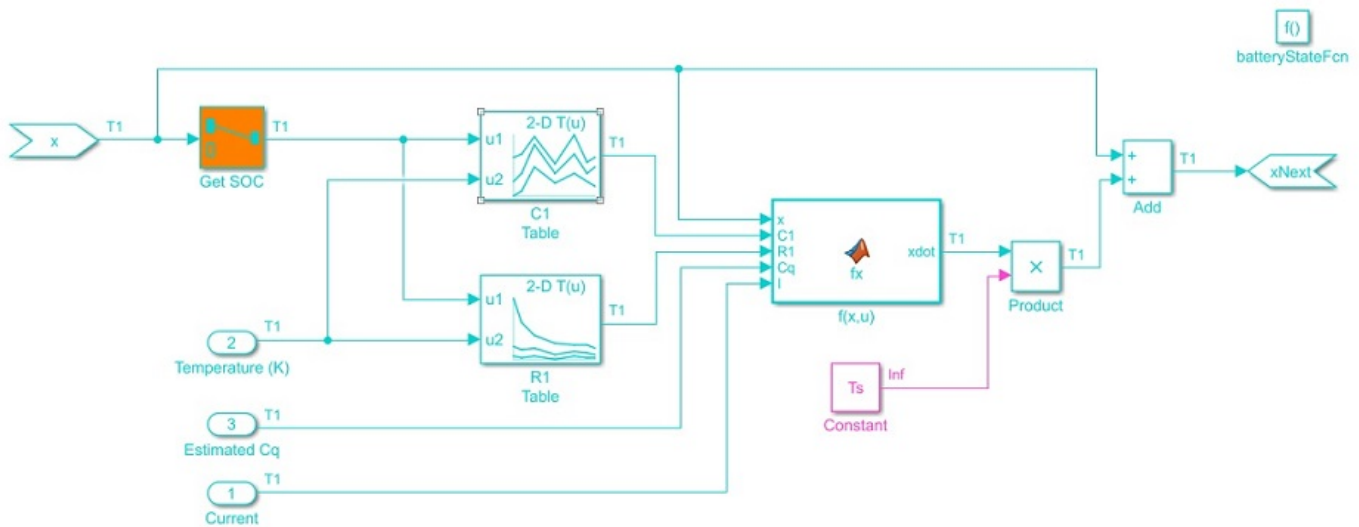
The measurement function is also implemented as a Simulink function named "batteryMeasurementFcn" as shown below.
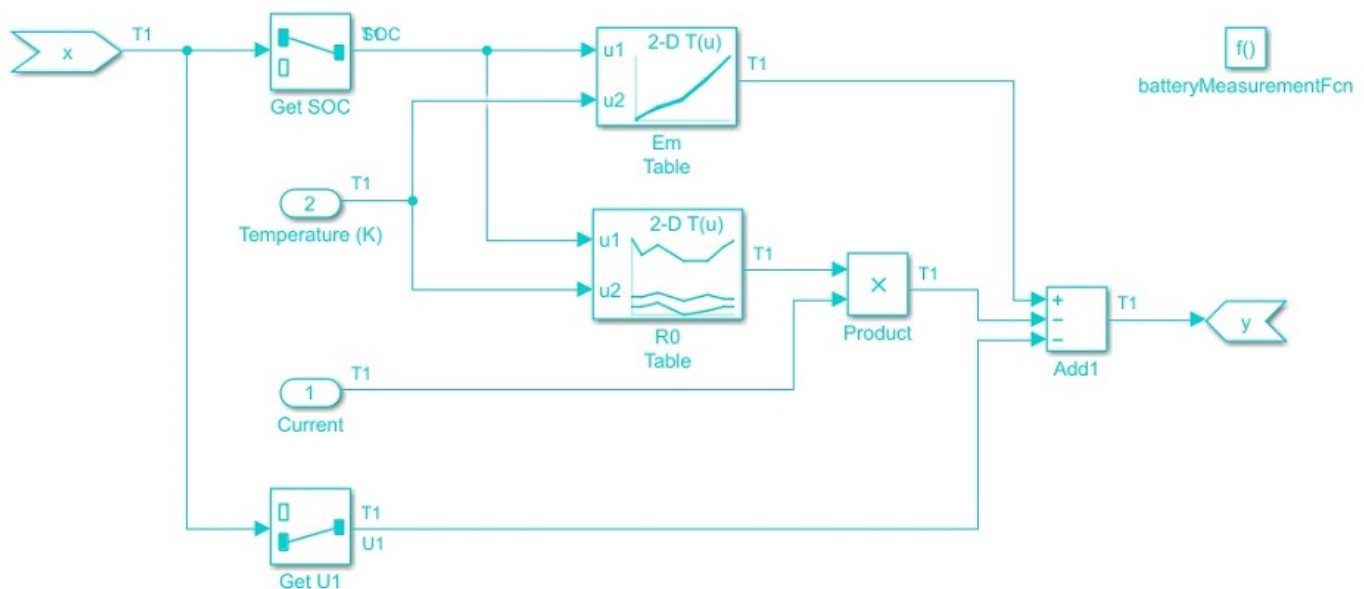
Configure the block parameters as follows:

In the **System Model** tab, specify the block parameters as shown:

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 larger spre
  - Beta: 2. Used to incorporate prior knowledge of the distribution. The nominal value for Be
  - 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 the battery state of charge is at 30% and switch to discharging when state of charge is at 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:
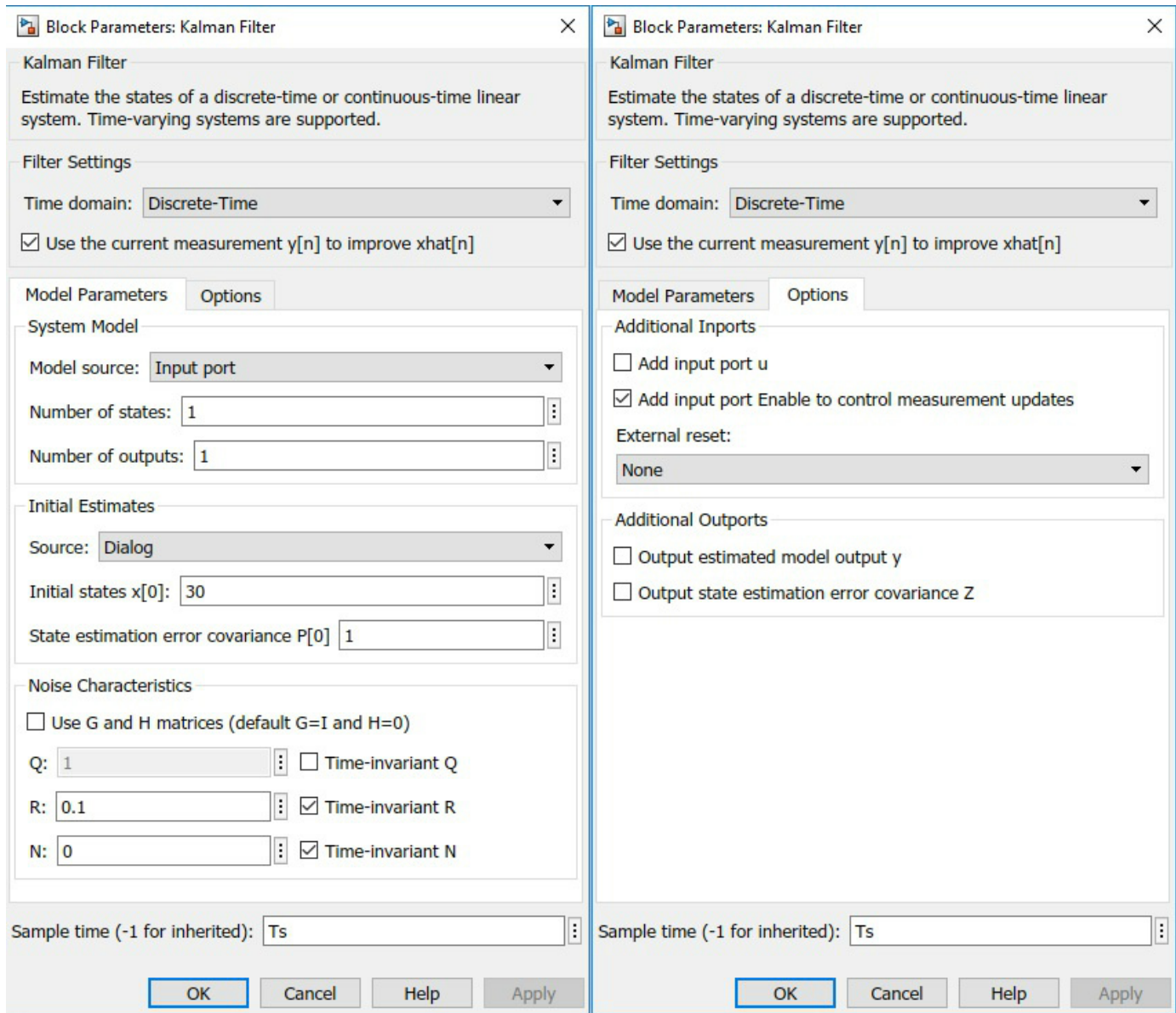
Click **Model Parameters** to specify the plant model and noise characteristics:

- **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 $C_{q_k}^{Measured} = C_{q_k} + V_{C_q} = \frac{\int_{t_{k-1}}^{t_k} I dt}{0.6}$.

- **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 Estimated SOC','Location','Best')
axis tight
```
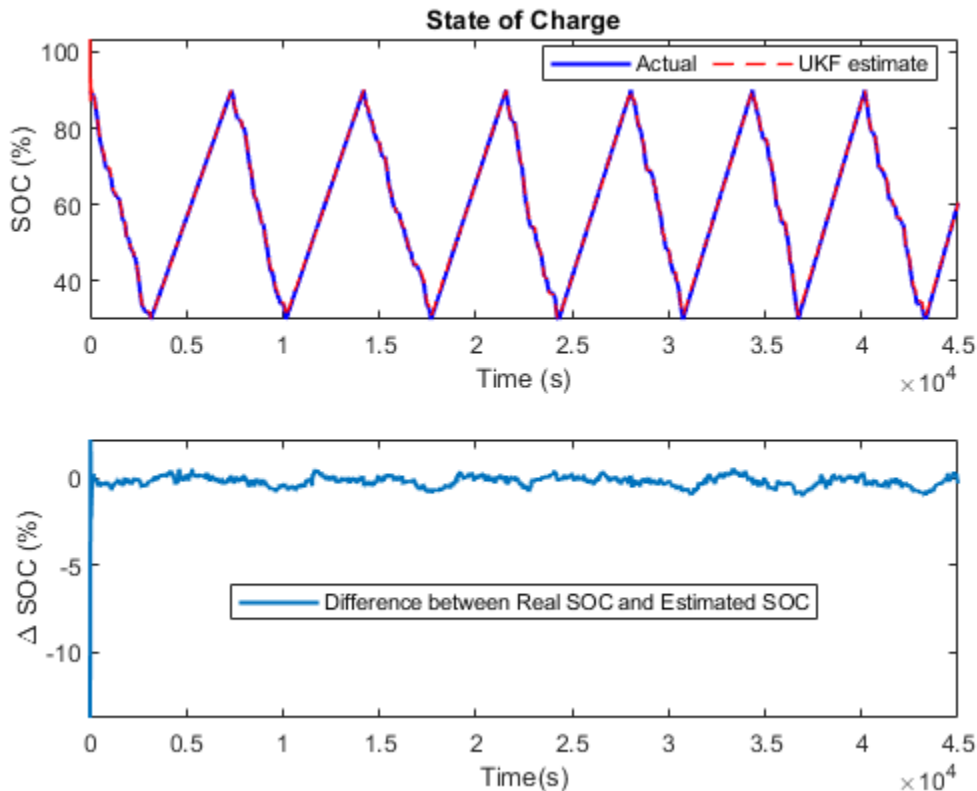

After an initial estimation error, the SOC converges quickly to the real SOC. The final estimation error is within 0.5% error. The unscented Kalman filter gives an accurate estimation of SOC.

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');
```
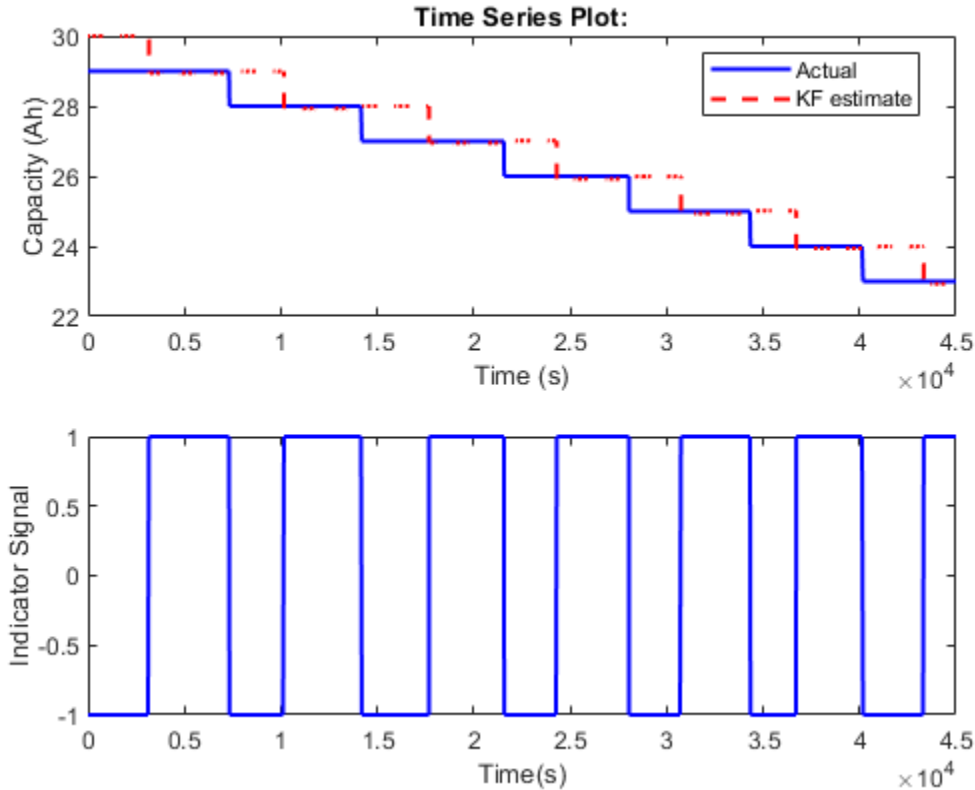

In general, the Kalman filter is able to track the real capacity. There is a half cycle delay between estimated capacity and real capacity. This delay is due to the timing of the measurements. The battery capacity degradation calculation occurs when one full discharge-charge cycle ends. The coulomb counting gives a capacity measurement based on the previous discharge or charge cycle.

Summary

This example shows how to use the 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. IEEE, 2000.

See Also

More About

- "RUL Estimation Using Identified Models or State Estimators" on page 5-6

Battery Cycle Life Prediction From Initial Operation Data

This example shows how to predict the remaining cycle life of a fast charging Li-ion battery using linear regression, a supervised machine learning algorithm. Lithium-ion battery cycle life prediction using a physics-based modelling approach is very complex due to varying operating conditions and significant device variability even with batteries from the same manufacturer. For this scenario, machine learning based approaches provide promising results when sufficient test data is available. Accurate battery cycle life prediction at the early stages of battery life would allow for rapid validation of new manufacturing processes. It also allows end-users to identify deteriorated performance with sufficient lead-time to replace faulty batteries. To this end, only the first 100 cycles based features are considered for predicting remaining cycle life and the prediction error is shown to be about 10% [1].

Dataset

The dataset contains measurements from 124 lithium-ion cells with nominal capacity of 1.1 Ah and a nominal voltage of 3.3 V under various charge and discharge profiles. The full dataset can be accessed here [2] with detailed description here [1]. For this example, the data is curated to only contain a subset of measurements relevant to the features being extracted. This reduces the size of the download without changing the performance of the machine learning model being considered. Training data contains measurements from 41 cells, validation data contains measurements from 43 cells and the test data contains measurements from 40 cells. Data for each cell is stored in a structure, which includes the following information:

- Descriptive data: Cell barcode, charging policy, cycle life
- Per-cycle summary data: Cycle number, discharge capacity, internal resistance, charge time
- Data collected within a cycle: Time, temperature, linearly interpolated discharge capacity, linearly interpolated voltage

Load the data from the MathWorks supportfiles site (this is a large dataset, ~1.7GB).

```matlab
url = 'https://ssd.mathworks.com/supportfiles/predmaint/batterycyclelifeprediction/v1/batteryDis
websave('batteryDischargeData.zip',url);
unzip('batteryDischargeData.zip')
load('batteryDischargeData');
```

Feature Extraction

Discharge capacity is a key feature that can reflect the health of a battery, and its value is indicative of driving range in an electric vehicle. Plot discharge capacity for the first 1,000 cycles of cells in training data to visualize its change across the life of a cell.

```matlab
figure, hold on;
for i = 1:size(trainData,2)
    if numel(trainData(i).summary.cycle) == numel(unique(trainData(i).summary.cycle))
      plot(trainData(i).summary.cycle, trainData(i).summary.QDischarge);
    end
end
ylim([0.85,1.1]),xlim([0,1000]);
ylabel('Discharge capacity (Ah)');
xlabel('cycle');
```
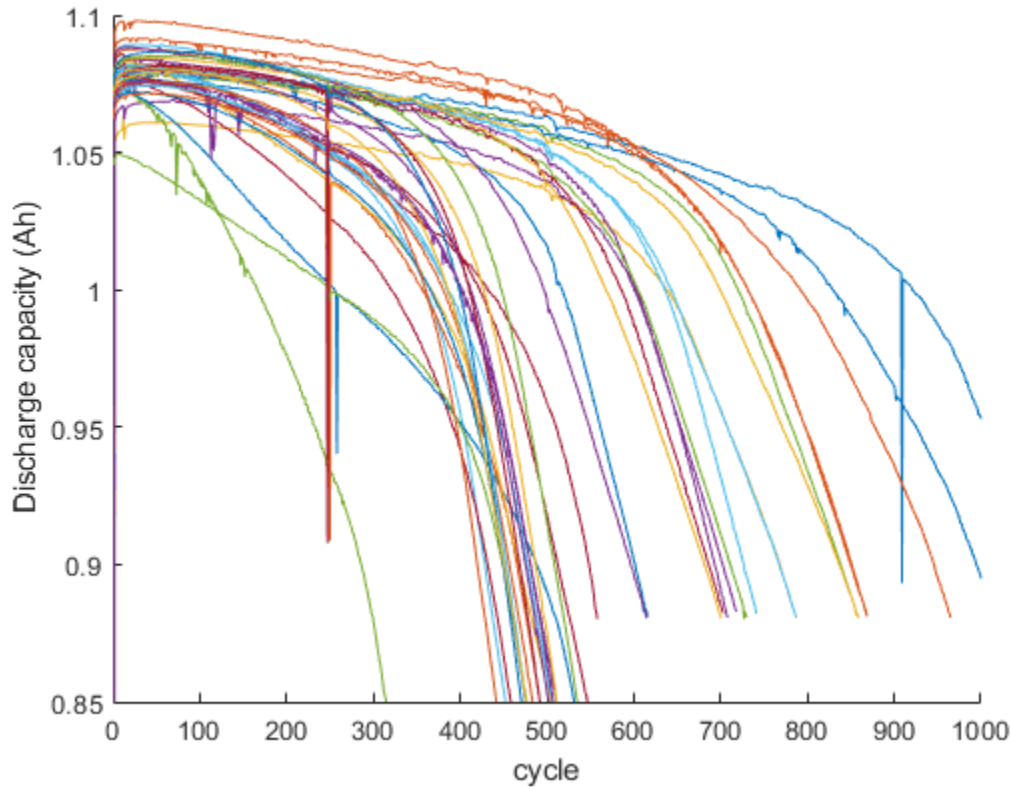
As seen in the plot, capacity fade accelerates near the end of life. However, capacity fade is negligible in the first 100 cycles and by itself is not a good feature for battery cycle life prediction. Therefore, a data-driven approach that considers voltage curves of each cycle, along with additional measurements, including cell internal resistance and temperature, is considered for predicting remaining cycle life. The cycle-to-cycle evolution of Q(V), the discharge voltage curve as a function of voltage for a given cycle, is an important feature space [1]. In particular, the change in discharge voltage curves between cycles, $\Delta Q(V)$, is very effective in degradation diagnosis. Therefore, select the difference in discharge capacity as a function of voltage between 100th and 10th cycle, $\Delta Q_{100-10}(V)$.

```
figure, hold on;
for i = 1:size(testData,2)
    plot((testData(i).cycles(100).Qdlin - testData(i).cycles(10).Qdlin), testData(i).Vdlin)
end
ylabel('Voltage (V)'); xlabel('Q_{100} - Q_{10} (Ah)');
```
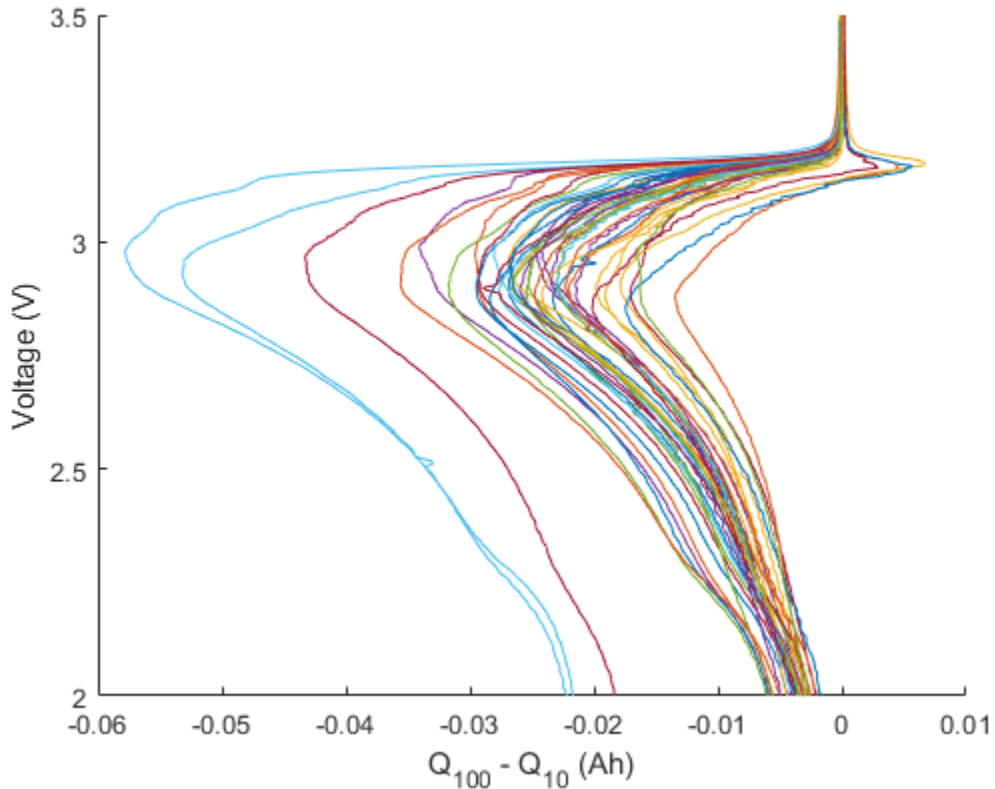
Compute summary statistics using $\Delta Q_{100-10}(V)$ curves of each cell as the condition indicator. It is important to bear in mind that these summary statistics do not need to have a clear physical meaning, they just need to exhibit predictive capabilities. For example, the variance of $\Delta Q_{100-10}(V)$ and cycle life are highly correlated as seen in this log–log plot, thereby confirming this approach.

```
figure;
trainDataSize = size(trainData,2);
cycleLife = zeros(trainDataSize,1);
DeltaQ_var = zeros(trainDataSize,1);
for i = 1:trainDataSize
    cycleLife(i) = size(trainData(i).cycles,2) + 1;
    DeltaQ_var(i) = var(trainData(i).cycles(100).Qdlin - trainData(i).cycles(10).Qdlin);
end
loglog(DeltaQ_var,cycleLife, 'o')
ylabel('Cycle life'); xlabel('Var(Q_{100} - Q_{10}) (Ah^2)');
```
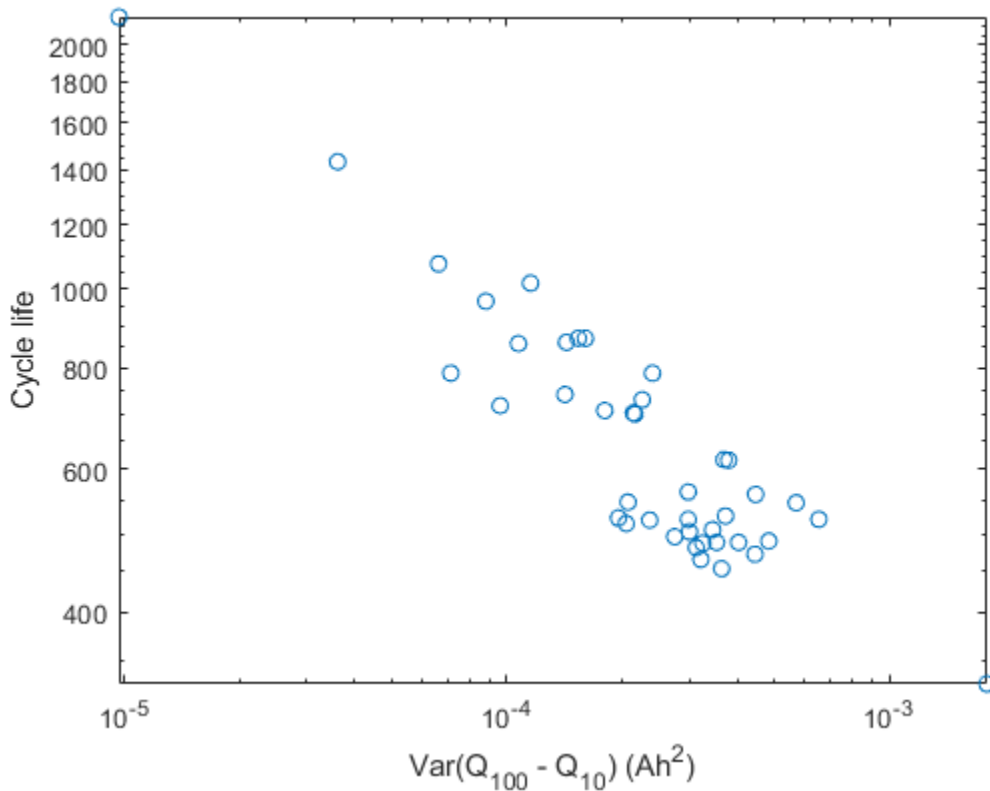
Next, extract the following features from the raw measurement data (the variable names are in brackets) [1]:

- $\Delta Q_{100-10}(V)$ log variance [DeltaQ_var]
- $\Delta Q_{100-10}(V)$ log minimum [DeltaQ_min]
- Slope of linear fit to capacity fade curve, cycles 2 to 100 [CapFadeCycle2Slope]
- Intercept of linear fit to capacity fade curve, cycles 2 to 100 [CapFadeCycle2Intercept]
- Discharge capacity at cycle 2 [Qd2]
- Average charge time over first 5 cycles [AvgChargeTime]
- Minimum Internal Resistance, cycles 2 to 100 [MinIR]
- Internal Resistance difference between cycles 2 and 100 [IRDiff2And100]

The `helperGetFeatures` function accepts the raw measurement data and computes the above listed features. It returns `XTrain` containing the features in a table and `yTrain` containing the expected remaining cycle life.

```
[XTrain,yTrain] = helperGetFeatures(trainData);
head(XTrain)
```

```
ans=8×8 table
    DeltaQ_var    DeltaQ_min    CapFadeCycle2Slope    CapFadeCycle2Intercept    Qd2       AvgChar
    _____    _____    _____    _____    _____    _____

     -5.0839       -1.9638          6.4708e-06               1.0809            1.0753       13.4
```

-4.3754	-1.6928	1.6313e-05	1.0841	1.0797	12.0
-4.1464	-1.5889	8.1708e-06	1.08	1.0761	10.9
-3.8068	-1.4216	-8.491e-06	1.0974	1.0939	10.0
-4.1181	-1.6089	2.2859e-05	1.0589	1.0538	11.0
-4.0225	-1.5407	2.5969e-05	1.0664	1.0611	10.7
-3.9697	-1.5077	1.7886e-05	1.0762	1.0721	10.1
-3.6195	-1.3383	-1.0356e-05	1.0889	1.0851	9.92

Model Development: Linear Regression with Elastic Net Regularization

Use a regularized linear model to predict the remaining cycle life of a cell. Linear models have low computational cost but provide high interpretability. The linear model is of the form:

$$y_i = \mathbf{w}^T\mathbf{x}_i + \beta$$

where y_i is the number of cycles for cell i, $\mathbf{x}_i$ is a p-dimensional feature vector for cell i and $\mathbf{w}$ is p-dimensional model coefficient vector and β is a scalar intercept. The linear model is regularized using elastic net to address high correlations between the features. For α strictly between 0 and 1, and nonnegative λ, the regularization coefficient, elastic net solves the problem:

$$\widehat{\mathbf{w}} = \min_w\|\mathbf{y} - \mathbf{X}\mathbf{w} - \beta\|_2^2 + \lambda P_\alpha(\mathbf{w})$$

where $P_\alpha(\mathbf{w}) = \dfrac{(1-\alpha)}{2}\|\mathbf{w}\|_2^2 + \alpha\|\mathbf{w}\|_1$. Elastic net approaches ridge regression when α is close to 0 and it is the same as lasso regularization when $\alpha = 1$. Training data is used to choose the hyperparameters α and λ, and determine values of the coefficients, $\mathbf{w}$. The `lasso` function returns fitted least-squares regression coefficients and information about the fit of the model as output for each α and λ value. The `lasso` function accepts a vector of values for λ parameter. Therefore, for each value of α, the model coefficients $\mathbf{w}$ and λ yielding the lowest RMSE value is computed. As proposed in [1], use four-fold cross validation with 1 Monte Carlo repetition for cross-validation.

```
rng("default")

alphaVec = 0.01:0.1:1;
lambdaVec = 0:0.01:1;
MCReps = 1;
cvFold = 4;

rmseList = zeros(length(alphaVec),1);
minLambdaMSE = zeros(length(alphaVec),1);
wModelList = cell(1,length(alphaVec));
betaVec = cell(1,length(alphaVec));

for i=1:length(alphaVec)
    [coefficients,fitInfo] = ...
        lasso(XTrain.Variables,yTrain,'Alpha',alphaVec(i),'CV',cvFold,'MCReps',MCReps,'Lambda',la
    wModelList{i} = coefficients(:,fitInfo.IndexMinMSE);
    betaVec{i} = fitInfo.Intercept(fitInfo.IndexMinMSE);
    indexMinMSE = fitInfo.IndexMinMSE;
    rmseList(i) = sqrt(fitInfo.MSE(indexMinMSE));
    minLambdaMSE(i) = fitInfo.LambdaMinMSE;
end
```

To make the fitted model robust, use validation data to select final values of α and λ hyperparameters. To this end, pick the coefficients corresponding to the four lowest RMSE values computed using the training data.

```
numVal = 4;
[out,idx] = sort(rmseList);
val = out(1:numVal);
index = idx(1:numVal);

alpha = alphaVec(index);
lambda = minLambdaMSE(index);
wModel = wModelList(index);
beta = betaVec(index);
```

For each set of coefficients, compute the predicted values and RMSE on validation data.

```
[XVal,yVal] = helperGetFeatures(valData);
rmseValModel = zeros(numVal);
for valList = 1:numVal
    yPredVal = (XVal.Variables*wModel{valList} + beta{valList});
    rmseValModel(valList) = sqrt(mean((yPredVal-yVal).^2));
end
```

Select the model with the least RMSE when using the validation data. The model related to the lowest RMSE value for training data is not the model corresponding to the lowest RMSE value for validation data. Using the validation data thereby increases model robustness.

```
[rmseMinVal,idx] = min(rmseValModel);
wModelFinal = wModel{idx};
betaFinal = beta{idx};
```

Performance evaluation of the trained model

The test data contains measurements corresponding to 40 cells. Extract the features and the corresponding responses from `testData`. Use the trained model to predict the remaining cycle life of each cell in `testData`.

```
[XTest,yTest] = helperGetFeatures(testData);
yPredTest = (XTest.Variables*wModelFinal + betaFinal);
```

Visualize the predicted vs. actual cycle life plot for the test data.

```
figure;
scatter(yTest,yPredTest)
hold on;
refline(1, 0);
title('Predicted vs Actual Cycle Life')
ylabel('Predicted cycle life');
xlabel('Actual cycle life');
```
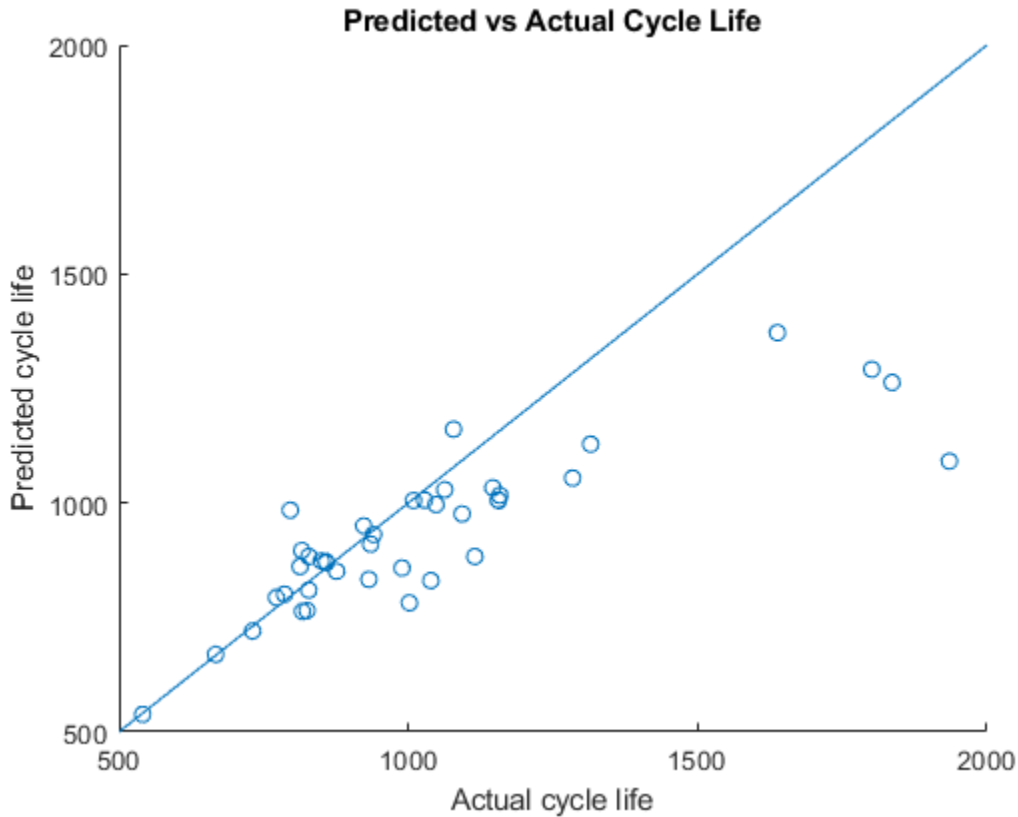
Ideally, all the points in the above plot should be close to the diagonal line. From the above plot, it is seen that the trained model is performing well when the remaining cycle life is between 500 and 1200 cycles. Beyond 1200 cycles, the model performance deteriorates. The model consistently underestimates the remaining cycle life in this region. This is primarily because test and validation data sets contain significantly more cells with total life around 1000 cycles.

Compute the RMSE of the predicted remaining cycle life.

```
errTest = (yPredTest-yTest);
rmseTestModel = sqrt(mean(errTest.^2))
```

rmseTestModel = 211.6148

Another metric considered for performance evaluation is average percentage error defined [1] as

$$\% \, err = \frac{1}{n} \Sigma_{i=1}^{n} \frac{|y_i - \hat{y}_i|}{y_i} \times 100$$

```
n = numel(yTest);
nr = abs(yTest - yPredTest);
errVal = (1/n)*sum(nr./yTest)*100
```

errVal = 9.9817

Conclusion

This example showed how to use a linear regression model with elastic net regularization for battery cycle life prediction based on measurements from only the first 100 cycles. Custom features were

extracted from the raw measurement data and a linear regression model with elastic net regularization was fitted using training data. The hyperparameters were then selected using the validation dataset. This model was used on test data for performance evaluation. Using measurements for just the first 100 cycles, RMSE of remaining cycle life prediction of cells in the test data set is 211.6 and an average percentage error of 9.98% is obtained.

References

[1] Severson, K.A., Attia, P.M., Jin, N. *et al.* "Data-driven prediction of battery cycle life before capacity degradation." *Nat Energy* **4,** 383–391 (2019). https://doi.org/10.1038/s41560-019-0356-8

[2] https://data.matr.io/1/

Supporting Functions

```
function [xTable, y] = helperGetFeatures(batch)
% HELPERGETFEATURES function accepts the raw measurement data and computes
% the following feature:
%
% Q_{100-10}(V) variance [DeltaQ_var]
% Q_{100-10}(V) minimum [DeltaQ_min]
% Slope of linear fit to the capacity fade curve cycles 2 to 100 [CapFadeCycle2Slope]
% Intercept of linear fit to capacity fade curve, cycles 2 to 100 [CapFadeCycle2Intercept]
% Discharge capacity at cycle 2 [Qd2]
% Average charge time over first 5 cycles [AvgChargeTime]
% Minimum Internal Resistance, cycles 2 to 100 [MinIR]
% Internal Resistance difference between cycles 2 and 100 [IRDiff2And100]

N = size(batch,2); % Number of batteries

% Preallocation of variables
y = zeros(N, 1);
DeltaQ_var = zeros(N,1);
DeltaQ_min = zeros(N,1);
CapFadeCycle2Slope = zeros(N,1);
CapFadeCycle2Intercept = zeros(N,1);
Qd2 = zeros(N,1);
AvgChargeTime = zeros(N,1);
IntegralTemp = zeros(N,1);
MinIR = zeros(N,1);
IRDiff2And100 = zeros(N,1);

for i = 1:N
    % cycle life
    y(i,1) = size(batch(i).cycles,2) + 1;

    % Identify cycles with time gaps
    numCycles = size(batch(i).cycles, 2);
    timeGapCycleIdx = [];
    jBadCycle = 0;
    for jCycle = 2: numCycles
        dt = diff(batch(i).cycles(jCycle).t);
        if max(dt) > 5*mean(dt)
            jBadCycle = jBadCycle + 1;
            timeGapCycleIdx(jBadCycle) = jCycle;
        end
    end
```

```matlab
    % Remove cycles with time gaps
    batch(i).cycles(timeGapCycleIdx) = [];
    batch(i).summary.QDischarge(timeGapCycleIdx) = [];
    batch(i).summary.IR(timeGapCycleIdx) = [];
    batch(i).summary.chargetime(timeGapCycleIdx) = [];

    % compute Q_100_10 stats
    DeltaQ = batch(i).cycles(100).Qdlin - batch(i).cycles(10).Qdlin;
    DeltaQ_var(i) = log10(abs(var(DeltaQ)));
    DeltaQ_min(i) = log10(abs(min(DeltaQ)));

    % Slope and intercept of linear fit for capacity fade curve from cycle
    % 2 to cycle 100
    coeff2 = polyfit(batch(i).summary.cycle(2:100), batch(i).summary.QDischarge(2:100),1);
    CapFadeCycle2Slope(i) = coeff2(1);
    CapFadeCycle2Intercept(i) = coeff2(2);

    %  Discharge capacity at cycle 2
    Qd2(i) = batch(i).summary.QDischarge(2);

    % Avg charge time, first 5 cycles (2 to 6)
    AvgChargeTime(i) = mean(batch(i).summary.chargetime(2:6));

    % Integral of temperature from cycles 2 to 100
    tempIntT = 0;
    for jCycle = 2:100
        tempIntT = tempIntT + trapz(batch(i).cycles(jCycle).t, batch(i).cycles(jCycle).T);
    end
    IntegralTemp(i) = tempIntT;

    % Minimum internal resistance, cycles 2 to 100
    temp = batch(i).summary.IR(2:100);
    MinIR(i) = min(temp(temp~=0));
    IRDiff2And100(i) = batch(i).summary.IR(100) - batch(i).summary.IR(2);
end

xTable = table(DeltaQ_var, DeltaQ_min, ...
    CapFadeCycle2Slope, CapFadeCycle2Intercept,...
    Qd2, AvgChargeTime, MinIR, IRDiff2And100);

end
```

See Also

```
lasso
```

Related Examples

- "Nonlinear State Estimation of a Degrading Battery System" on page 5-69
- "Remaining Useful Life Estimation Using Convolutional Neural Network" on page 4-118
- "Battery Cycle Life Prediction Using Deep Learning" on page 5-123

Live RUL Estimation of a Servo Gear Train Using ThingSpeak

This example shows how to estimate the Remaining Useful Life (RUL) of a servo motor gear train through real-time streaming of servo motor data from an Arduino-based data acquisition system to ThingSpeak™ and from ThingSpeak to an RUL estimation engine running in MATLAB®.

Motor current signature analysis (MCSA) of the current signal driving a hobby servo motor is used to extract frequency-domain (spectral) features from several frequency regions of interest indicative of motor and gear train faults. A combination of features are used construct a Health Indicator (HI) for subsequent RUL estimation.

MCSA is a useful method for the diagnosis of faults that induce torque or speed fluctuations in the servo gear train, which in turn result in correlated motor current changes. MCSA has been proven to be ideal for motor fault analysis as only the motor current signal is required for analysis, which eliminates the need for additional and expensive sensing hardware. Gear fault detection using traditional vibration sensors is challenging, especially in cases where the gear train is not easily accessible for instrumentation with accelerometers or other vibration sensors.

This example illustrates how to build a real-time data streaming, feature extraction, and RUL estimation system using simple, off-the-shelf components suitable for educational lab exercises or for prototyping of industrial applications. To run this example without hardware, use the `sendSyntheticFeaturesToThingSpeak` function to generate synthetic data.

The simplified workflow to construct the data streaming system and the RUL estimation engine includes the following steps:

1 Develop the hardware and the data acquisition system using off-the-shelf components.

2 Stream real-time data from a microcontroller like an Arduino® UNO to MATLAB.

3 Process the real-time data in MATLAB to extract features and stream the features to the cloud using ThingSpeak.

4 Stream the cloud data to an RUL estimation and visualization engine in MATLAB.

RUL Estimation Flowchart
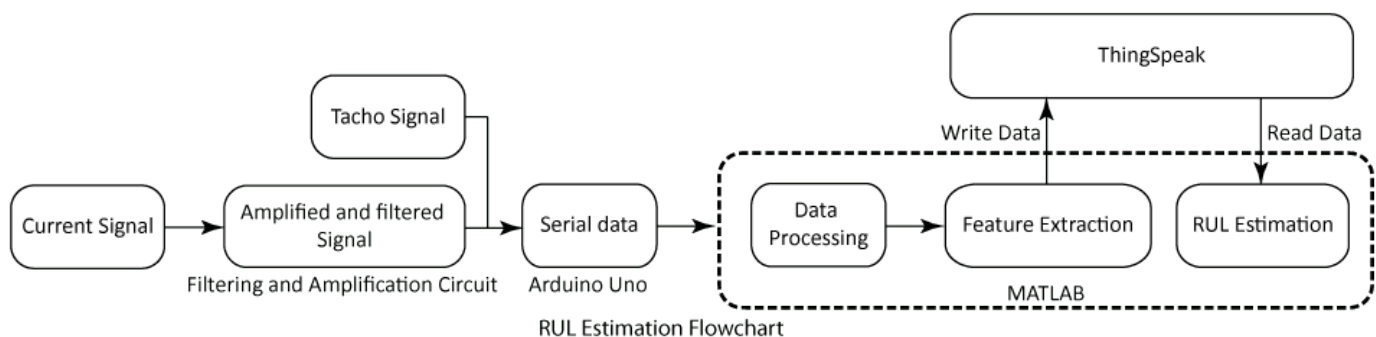
Hardware Setup and Data Acquisition

Hardware Overview

For this example, the electrical current data was collected from a standard Futaba S3003 hobby servo, which was modified for continuous rotation. Servos convert the high speed of an internal DC motor to high torque at their output shaft. To achieve this, servos consist of a DC motor, a set of nylon or metal gear pairs, and a control circuit. The control circuit was removed to be able to apply a

constant 5V voltage to the DC motor leads directly and allow the current through the DC motor to be monitored through a current sensing resistor.
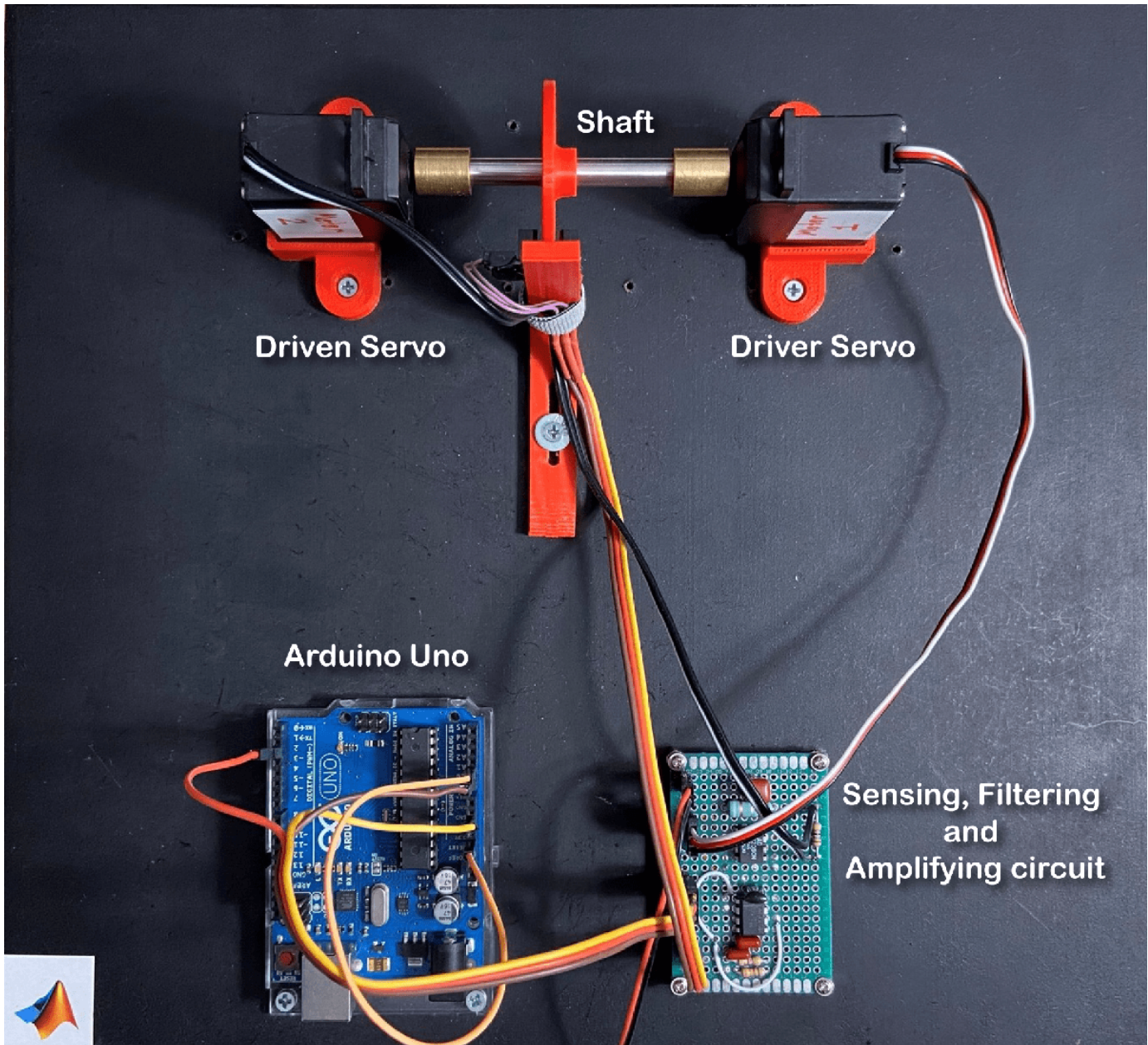
A second, opposing servo with its DC motor terminals shunted together through a low-value resistor was used to generate a high-torque opposing load for the driver servo to speed up the degradation of the gear train and induce larger variations in the motor current signal for MCSA analysis.

Servo Motor and Gear Train

The Futaba S3003 servo consists of four pairs of meshing nylon gears as illustrated in the figure below. The pinion P1 on the DC motor shaft meshes with the stepped gear G1. The pinion P2 is a molded part of the stepped gear G1 and meshes with the stepped gear G2. The pinion P3, which is a

molded part of gear G2, meshes with the stepped gear G3. Pinion P4, which is molded with G3, meshes with the final gear G4 that is attached to the output shaft. The stepped gear sets G1 and P2, G2 and P3, and G3 and P4 are free spinning gears — that is, they are not fixed to their respective shafts. The set of gears provides a 278:1 reduction going from a motor speed of about 6117 rpm to about 22 rpm at the output shaft when the DC motor is driven at 5 volts.

Futaba S3003 standard servo gear train

The following table outlines the tooth count and theoretical values of output speed, gear mesh frequencies, and cumulative gear reduction at each gear mesh.

Pinion	Gear	Pinion Teeth	Gear Teeth	Output Speed		Gear Mesh Frequency (Hz)	Cumulative Gear Reduction
				(RPM)	(Hz)		
P1	-	10	-	6116.7	101.94	-	1
P2	G1	10	62	986.6	16.44	1019.4	6.2
P3	G2	10	50	197.3	3.29	164.4	31
P4	G3	16	35	56.4	0.94	32.9	108.5
-	G4	-	41	22	0.37	15	278

Theoretical values calculated from tooth count assuming 22 RPM at the output shaft

The rotation speed of the output shaft of the servo was measured using an infrared photointerrupter along with a 40 mm diameter, 3-D printed tacho wheel with 16 slots. The sixteen slots on the wheel are equally spaced and the IR photointerrupter was placed such that there were exactly sixteen pulses per rotation of the wheel. The servos and the photointerrupter were held in place by 3-D printed mounts.
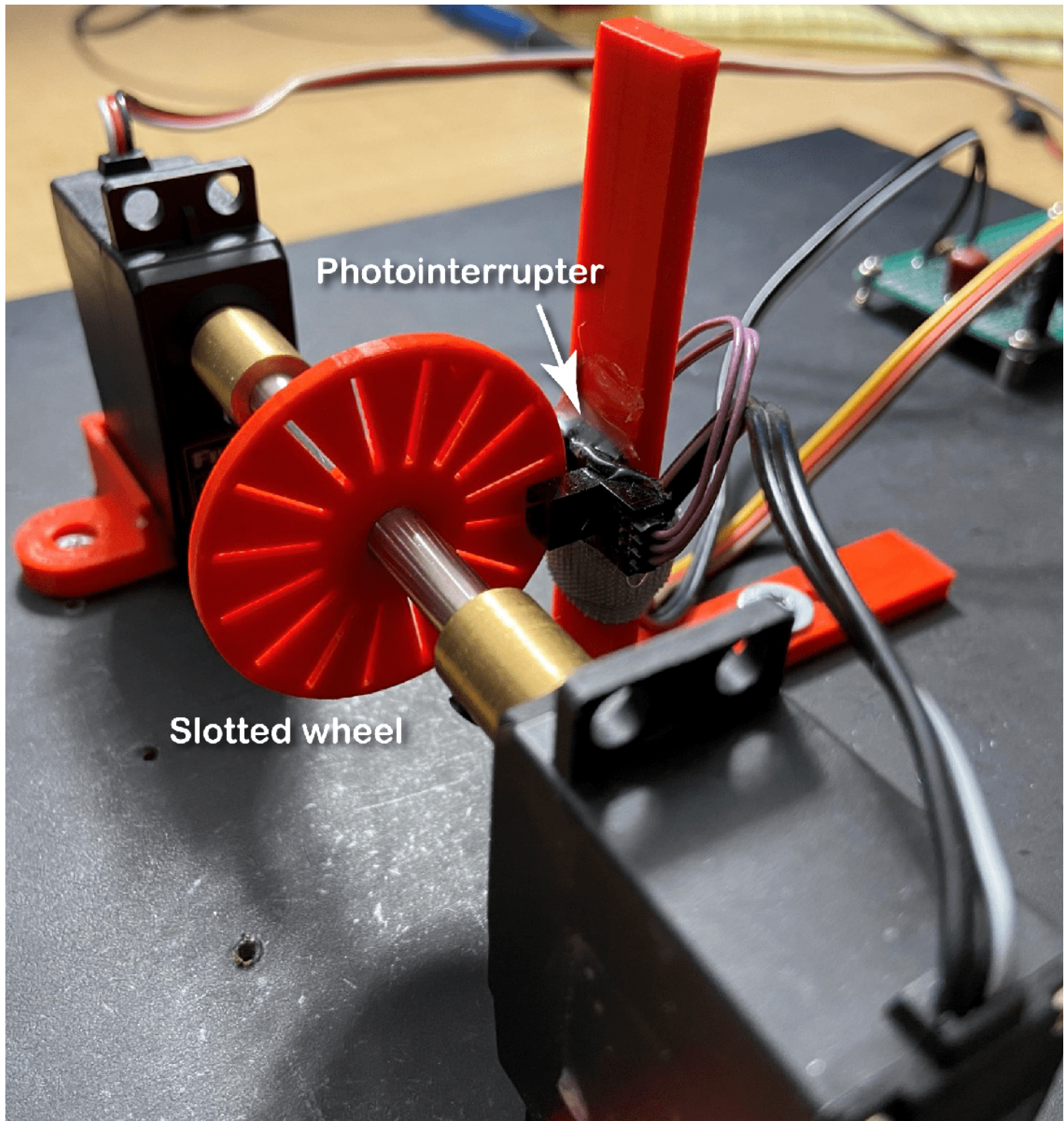
The driving DC motor was driven at a constant 5 volts, and with four pairs of gears providing 278:1 speed reduction, the output shaft speed had an average value of 22 rpm. The fluctuations of output speed and the corresponding changes in the motor current due to gear meshing and gear teeth degradation over time were captured by sensors for further analysis in MATLAB.

Filtering and Amplification Circuit

The current consumption through the DC motor windings was calculated using Ohm's law by measuring the voltage drop across a 0.5-ohm, 0.5 W resistor. Since the change in current measurement values was too small for high-quality detection, the current signal was amplified 20 times using a single-supply sensor interface amplifier (Analog Devices AD22050N). The amplified current signal was then filtered using an anti-aliasing fifth-order elliptic low-pass filter (Maxim MAX7408) to smooth it and to eliminate noise before sending it to an analog-to-digital converter (ADC) pin on the Arduino UNO. The corner frequency of the anti-aliasing filter was placed at around 470 Hz to provide ample attenuation at the Nyquist frequency (750 Hz) of the analog-to-digital converter.

The electronic circuit was realized on a prototype printed circuit board (PCB) using a small number of additional passive components as shown in the picture below.
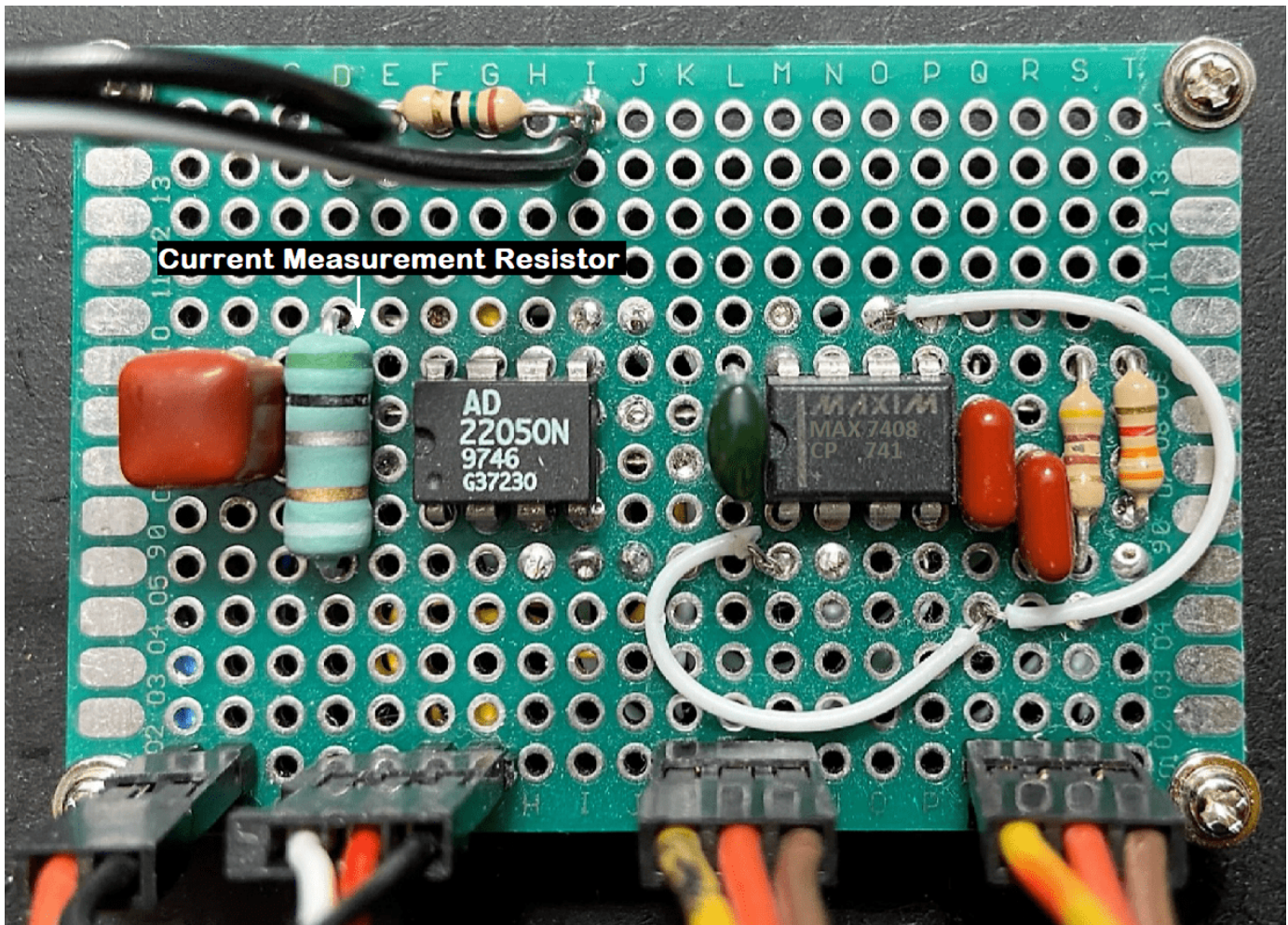
As the flowchart shows below, Arduino UNO sampled the current signal through an ADC input at a 1500 Hz sampling frequency and streamed it to the computer, along with the tachometer pulses, over a serial communication channel running at a baud rate of 250000 bps.

Signal Flow
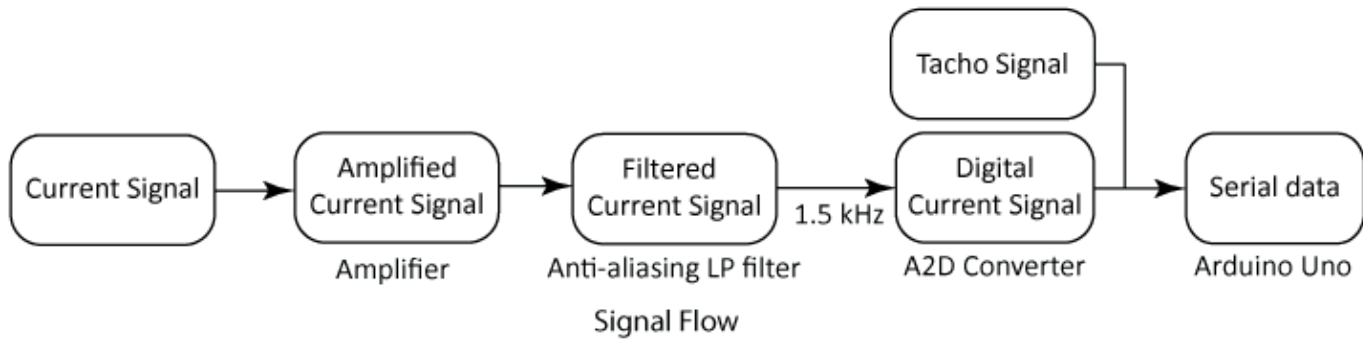
For the data in this example, the serial signal was encoded on the Arduino UNO as three 8-bit binary values per data sample (current + tacho) to improve the serial communication efficiency with the computer.

Real-Time Data Streaming

Streaming Data from Arduino UNO to MATLAB

The Arduino UNO encoded the motor current and the tacho pulse values into a binary buffer for efficiency and sent batches of data to the computer consisting of 16384 (2^14) samples every 5 minutes. The data was sent from the Arduino UNO at a rate of 1500 Hz using a simple data protocol consisting of start and end markers around the data bytes for easy extraction of data batches in MATLAB code.

See the attached Arduino sketch file `servo_data_producer.ino` for more information and to stream data to MATLAB from your hardware setup.

Receiving Live Data Stream in MATLAB

You can use a MATLAB script to fetch batches of serial data from the Arduino UNO with the appropriate baud rates and serial COM port numbers.

First, open the serial port for communication with your Arduino or equivalent hardware. You can use the `serialportlist` command to find the serial COM port. For this example, the Arduino UNO is on `COM4`. Use a baud rate of 250000 bps.

```
port = "COM4";
baudRate = 250000;
```

Set an appropriate timeout and use the correct data terminator from the Arduino code contained in the file `servo_data_producer.ino`. For this example, use a timeout of 6 minutes. Then, clear the serial port buffer using `flush`.

```
s = serialport(port,baudRate,'Timeout',360);
configureTerminator(s,"CR/LF");
flush(s);
```

Now, read and process the encoded data stream from the Arduino with an appropriate number of read cycles. For this example, use 10 read cycles. For more details, see the supporting function `readServoDataFromArduino` provided in the **Supporting Functions** section.

```
count = 0;
countMax = 10;
while (count < countMax)
```

```
  try
    count = count + 1;
    fprintf("\nWaiting for dataset #%d...\n", count);
    readServoDataFromArduino(s);
  catch e
    % Report any errors.
    clear s;
    throw(e);
  end
end
```

Now, use the `clear` command to close the serial port.

```
clear s;
```

Data Processing and Feature Extraction in MATLAB

Processing Live Stream Data

Once the data is streamed into MATLAB, you can use the `readServoDataFromArduino` script to construct a data matrix to store the latest batch of servo motor data. For more details, see the supporting function `readServoDataFromArduino` provided in the **Supporting Functions** section.

```
readServoDataFromArduino(s)
```

The resultant timetable contains the time stamp, motor current, and tacho pulse values for the latest dataset streamed from the Arduino UNO.

Use the `processServoData` function to output a timetable with relevant physical units.

```
T = processServoData(X)
```

The resulting timetable provides a convenient container for extracting predictive features from the motor data.

If you do not have the necessary hardware, you can run this example using the `sendSyntheticFeaturesToThingSpeak` function to generate synthetic data.

Feature Extraction

From each new dataset, compute the nominal speed to detect frequencies of interest in the gear train and match them correctly with the frequencies on their power spectra.

```
F = extractFeaturesFromData(T);
```

Using the sampling frequency value of 1500 Hz in the `tachorpm` command, compute the nominal speed of the output shaft. For this instance, the RPM value is constant around 22 rpm.

```
tP = T.TachoPulse;
rpm = tachorpm(tP,Fs,'PulsesPerRev',16,'FitType','linear');
rpm = mean(rpm);
```

The motor speed along with the physical parameters of the gear sets, enable the construction of fault frequency bands, which is an important prerequisite for computing the spectral metrics. Using the tooth count of the drive gears in the gear train and the nominal rpm, first compute the frequencies of interest. The frequencies of interest are the actual output speed values in Hertz whose values were close to the theoretical values listed in the table below.

Pinion	Gear	Pinion Teeth	Gear Teeth	Output Speed		Gear Mesh Frequency (Hz)	Cumulative Gear Reduction
				(RPM)	(Hz)		
P1	-	10	-	6116.7	101.94	-	1
P2	G1	10	62	986.6	16.44	1019.4	6.2
P3	G2	10	50	197.3	3.29	164.4	31
P4	G3	16	35	56.4	0.94	32.9	108.5
-	G4	-	41	22	0.37	15	278

Theoretical values calculated from tooth count assuming 22 RPM at the output shaft

Next, construct the frequency bands for the all the output speeds which include the following frequencies of interest using the `faultBands` command. The selected fundamental frequencies, harmonics, and sidebands are discussed in more details in the "Analyze Gear Train Data and Extract Spectral Features Using Live Editor Tasks" on page 3-35 example. In particular, the important fault frequencies to monitor in the servo system were identified as follows:

- First six harmonics of FS1 with 0:1 sidebands of FS2
- 1st, 2nd, and 4th harmonics of FS2 with 0:1 sidebands of FS3
- 3rd harmonic of FS3

```
% Number of gear and pinion teeth.
G4 = 41; G3 = 35; G2 = 50; G1 = 62;
P4 = 16; P3 = 10; P2 = 10; P1 = 10;

% Shaft speeds in Hz.
FS5 = rpm / 60;
FS4 = G4 / P4 * FS5;
FS3 = G3 / P3 * FS4;
FS2 = G2 / P2 * FS3;
FS1 = G1 / P1 * FS2;

% Generate the fault bands of interest.
FB_S1 = faultBands(FS1, 1:6, FS2, 0:1);
FB_S2 = faultBands(FS2, [1 2 4], FS3, 0:1);
FB_S3 = faultBands(FS3, 3);
FB = [FB_S1; FB_S2; FB_S3];
```

Use the `faultBandMetrics` command along with the power spectral density (`pwelch` command) of the motor current signal to compute a total of 85 spectral metrics.

Compute the power spectrum of the motor current data.

```
mC = T.MotorCurrent;
[ps,f] = pwelch(mC,[],[],length(mC),Fs);
```

Compute spectral metrics for all fault bands.

```
metrics = faultBandMetrics(ps,f,FB);
metrics = metrics{:, [4 13 85]}; % Select significant metrics.
```

Pick features to track out of the metrics computed. Select some of the significant ones such as the peak amplitude of the spectrum at the first and second harmonics of the shaft 1 speed (FS1).

```
features = [metrics,mean(mC),rpm];
```

Note that the average motor current and the average speed in rpm of the motor are also recorded as additional features.

Live Data Streaming and RUL Estimation

Store Feature Data in Cloud

Once various features are computed from real-time data in batches every 5 minutes, send them to ThingSpeak as part of the same MATLAB script for cloud storage and remaining useful life analysis.

Send feature values to a ThingSpeak channel as separate fields using the `sendFeaturesToThingSpeak` function.

```
sendFeaturesToThingSpeak(features)
```

Here is an example of a ThingSpeak channel that is configured with a number of fields to store the feature values. The channel serves as a convenient data repository and visualization platform.
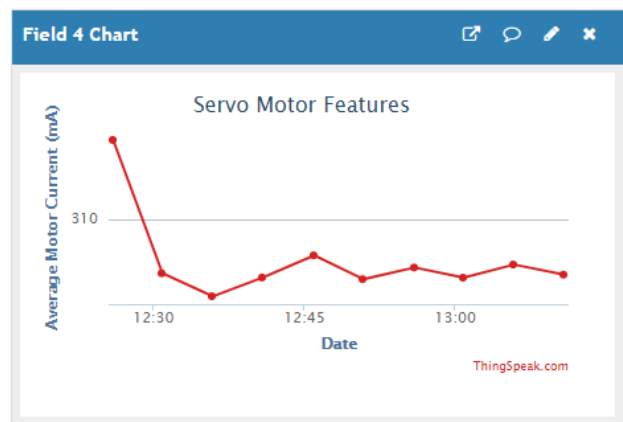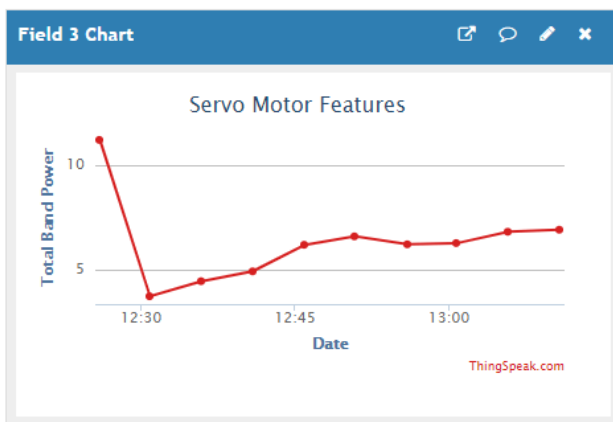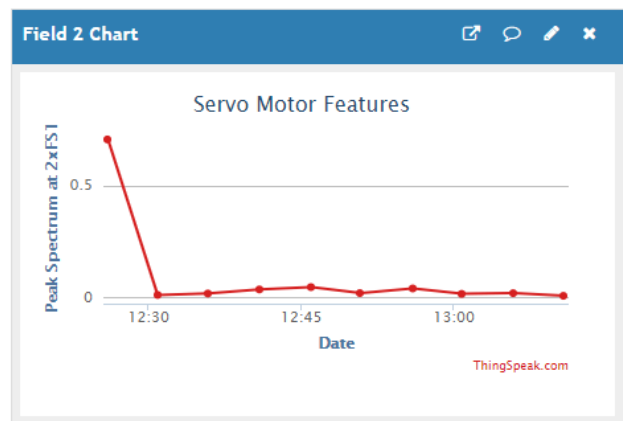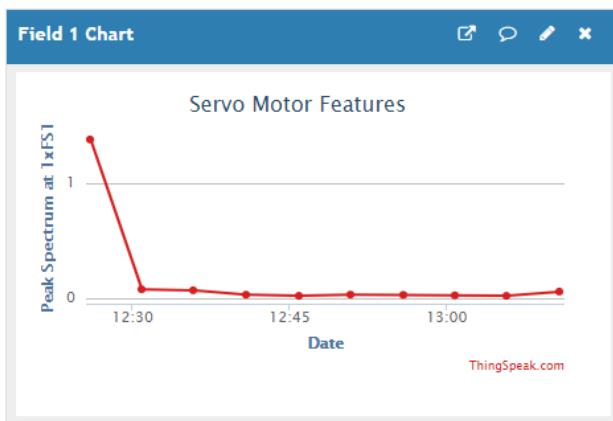
The ThingSpeak channel provides a real-time feature monitoring dashboard for tracking the changes in feature values. It also acts as the data source for feature processing such as remaining useful life estimations.

If you don't have the hardware setup to generate features from the servo motor, you can generate synthetic feature values to illustrate the computation of subsequent RUL estimates using the following code example. You can have separate instances of MATLAB running on different machines to read and write data to ThingSpeak.

```matlab
hasArduino = false; % Set to true when using Arduino UNO.
if ~hasArduino
  % Set timer to send new feature data every minute.
  tmr1 = timer('ExecutionMode','fixedSpacing','Period',60);
  tmr1.TimerFcn = @(~,~) sendSyntheticFeaturesToThingSpeak();
  tmr1.start();
  return;
end
```

Read Feature Data from Cloud

A separate MATLAB script, which can also be deployed as a compiled MATLAB app or as a deployed Web App, can be used to read the servo motor features from ThingSpeak and compute real-time estimates of the RUL metric. For this instance, construct an initial exponential degradation model and update the model periodically as new feature values become available.

```matlab
% Set timer to check for new data every minute and update the RUL model, mdl.
tmr2 = timer('ExecutionMode','fixedSpacing','Period',60);
tmr2.TimerFcn = @(~,~) readFeaturesFromThingSpeak(mdl,channelID,readKey);
tmr2.start();
```

Each new set of features are used to update the exponential degradation model and compute real-time estimates of the RUL.

Live Remaining Useful Life Estimation

For this example, assume that the training data is not historical data, but rather real-time observations of the component condition as captured by the features computed in the previous steps. An exponential degradation model (`exponentialDegradationModel`) is suitable for this type of real-time RUL estimation and visualization. The RUL model uses a health index as a threshold to estimate the RUL of the system. For this example, the threshold is on the `Total Band Power` feature of the servo motor as computed in the previous steps. The total band power captures the changes in the spectral energy within the fault band frequencies associated with important frequency regions of the servo motor gear train.

Construct the exponential degradation model with an arbitrary prior distribution data and a specified noise variance.

Specify the lifetime and data variable names for the observation data from a subset of the data table read from ThingSpeak. Then, use each feature values to predict the RUL of the component using the current life time value stored in the model.

```matlab
channelID = 1313749; % Replace with the channel ID to write data to.
readKey  = 'KYIDUZ1ENDT3TGG0'; % Replace with the read API key of your channel.

% Read most recent degradation data from ThingSpeak.
T = thingSpeakRead(channelID,'ReadKey',readKey,'OutputFormat','timetable');
```

```
% Construct the exponential degradation model using Total Band Power as the
% health index.
mdl = exponentialDegradationModel( ...
   'Theta', 1, 'ThetaVariance', 100, ...
   'Beta', 1, 'BetaVariance', 100, ...
   'NoiseVariance', 0.01, ...
   'LifeTimeVariable', T.Properties.DimensionNames{1}, ...
   'DataVariables', T.Properties.VariableNames{3}, ...
   'LifeTimeUnit', "hours");

% Set timer to check for new data every minute.
tmr2 = timer('ExecutionMode','fixedSpacing','Period',60);
tmr2.TimerFcn = @(~,~) readFeaturesFromThingSpeak(mdl,channelID,readKey);
tmr2.start();
```
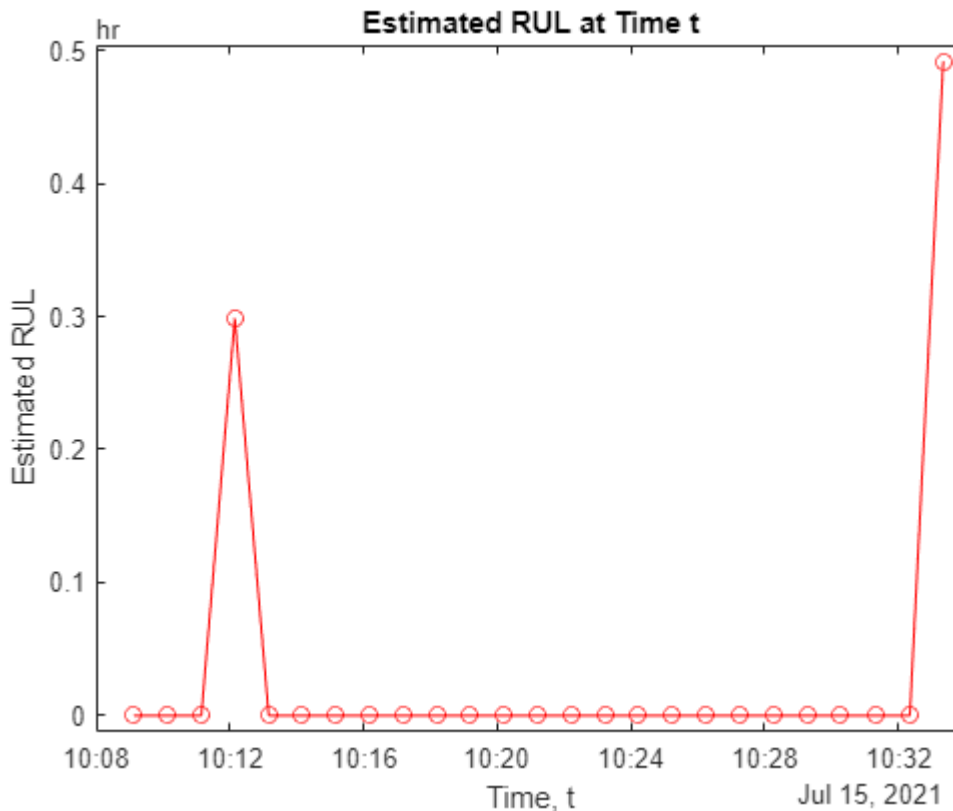

The MATLAB script tracks the feature data available on the ThingSpeak channel and updates the RUL model as new feature values become available, thereby providing a real-time RUL estimation and visualization capability.

Supporting Functions

The supporting function `readServoDataFromArduino` reads and processes the encoded data streams from the Arduino UNO.

```
function readServoDataFromArduino(s)
% Reads and processes encoded data streams from Arduino.
```

```matlab
% Find start of encoded data stream.
str = readline(s);
N = sscanf(str, "BEGIN:%d"); % Expected number of data points (should be 16384).

% Acquire new data until the end of data stream.
if ~isempty(N) && N > 0
  X = zeros(N,2);

  % Read and store encoded binary data.
  disp("Reading servo motor data...");
  for k = 1:N
    % Read three bytes each time containing motor current and tacho pulse values.
    x = [read(s,1,"uint16"), read(s,1,"uint8")];
    X(k,:) = double(x);
  end

  % Find end of encoded data stream.
  readline(s); % Remove CR/LF from data stream.
  str = readline(s);
  if str == "END"
    disp("Processing servo data...");
    T = processServoData(X);

    disp('Extracting features...');
    F = extractFeaturesFromData(T);

    disp('Sending features to ThingSpeak...');
    sendFeaturesToThingSpeak(F);
  else
    error('Error reading end-of-file marker from serial port.');
  end
end
end
```

The supporting function `processServoData` converts raw data matrices to a timetable with appropriate variable names and physical units.

```matlab
function T = processServoData(X)
% Converts raw data matrix to a timetable with appropriate variable names and
% physical units.
Fs = 1500; % Sampling rate.
Rsens = 0.5; % Current sense resistor value in ohm.
Kamp = 20; % Sensor amplifier gain.
count2volt = 5/1024; % Scaling factor between digital reading to analog voltage.

T = timetable('SampleRate', Fs);
T.MotorCurrent = X(:,1) * count2volt / Kamp / Rsens * 1000; % Convert to mA.
T.TachoPulse   = X(:,2); % On/off tacho pulse values.
T.Properties.VariableUnits = {'mA', 'level'};
end
```

The supporting function `extractFeaturesFromData` extracts features from time-domain servo motor data in table format.

```matlab
function features = extractFeaturesFromData(T)
% Extracts up to 8 features from time-domain servo data provided in table
% format.
```

```
Fs = 1500; % Sampling rate.

% Average motor speed.
tP = T.TachoPulse;
rpm = tachorpm(tP, Fs, 'PulsesPerRev', 16, 'FitType', 'linear');
rpm = mean(rpm);

% Number of gear and pinion teeth.
G4 = 41; G3 = 35; G2 = 50; G1 = 62;
P4 = 16; P3 = 10; P2 = 10; P1 = 10;

% Shaft speeds in Hz.
FS5 = rpm / 60;
FS4 = G4 / P4 * FS5;
FS3 = G3 / P3 * FS4;
FS2 = G2 / P2 * FS3;
FS1 = G1 / P1 * FS2;

% Generate the fault bands of interest.
FB_S1 = faultBands(FS1, 1:6, FS2, 0:1);
FB_S2 = faultBands(FS2, [1 2 4], FS3, 0:1);
FB_S3 = faultBands(FS3, 3);
FB = [FB_S1; FB_S2; FB_S3];

% Compute the power spectrum of the motor current data.
mC = T.MotorCurrent;
[ps,f] = pwelch(mC, [], [], length(mC), Fs);

% Compute spectral metrics for all fault bands.
metrics = faultBandMetrics(ps, f, FB);
metrics = metrics{:, [4 13 85]}; % Select significant metrics.

% Pick features to track.
features = [metrics, mean(mC), rpm];
end
```

The supporting function `sendFeaturesToThingSpeak` sends feature values to a ThingSpeak channel as separate fields.

```
function sendFeaturesToThingSpeak(features)
% Sends feature values to a ThingSpeak channel as separate fields.

% Configure the Channel ID and the Write API Key values.
channelID = 1313749; % Replace with your channel ID to write data to.
writeKey = 'X96MRW1TTZC1XNV3'; % Replace with the write API key of your channel.

% Write the features to the channel specified by the 'channelID' variable.
fields = 1:numel(features); % Up to 8 features
thingSpeakWrite(channelID, features, 'Fields', fields, 'WriteKey', writeKey);
end
```

The supporting function `readFeaturesFromThingSpeak` reads feature values from a ThingSpeak channel as new data becomes available by tracking the time stamps of the latest data readings. The function also updates the RUL model using the new feature values and plots the predicted RUL values.

```
function readFeaturesFromThingSpeak(mdl, channelID, readKey)
% Read feature values from ThingSpeak to update the RUL estimation.
```

```matlab
persistent timestamp
persistent hplot

if isempty(timestamp)
  % Start with last ten days of data.
  timestamp = dateshift(datetime, "start", "day", -10);
end

% Read recent data from all fields.
disp('Reading features from ThingSpeak...');
T = thingSpeakRead(channelID, 'ReadKey', readKey, ...
  'OutputFormat', 'timetable', 'DateRange', [timestamp, datetime]);
if ~isempty(T)
  T = T(T.Timestamps > timestamp, :); % Only keep most recent data.
end

if ~isempty(T)
  T = T(T.Timestamps > timestamp, :); % Only keep most recent data.
  timestamp = T.Timestamps(end); % Update time stamp for most recent data.

  % Set RUL degradation threshold.
  threshold = 10;

  % The training data is not historical data, but rather real-time observations
  % of the servo motor features.
  N = height(T);
  estRUL = hours(zeros(1,N));
  for i = 1:N
    update(mdl, T(i,:))
    estRUL(i) = predictRUL(mdl, threshold);
  end

  % Visualize RUL over time.
  if isempty(hplot) || ~isvalid(hplot)
    hplot = plot(T.Timestamps(:)', estRUL, 'r-o');
    title('Estimated RUL at Time t')
    xlabel('Time, t')
    ylabel('Estimated RUL')
  else
    hplot.XData = [hplot.XData, T.Timestamps(:)'];
    hplot.YData = [hplot.YData, estRUL];
  end
end
end
```

When a servo motor and an Arduino UNO are not available to generate feature values, use the supporting function sendSyntheticFeaturesToThingSpeak to provide synthetic feature data to illustrate streaming of features values to ThingSpeak.

```matlab
function sendSyntheticFeaturesToThingSpeak()
% Generate synthetic features and send them to ThinkSpeak.
persistent bandPower
if isempty(bandPower)
  bandPower = 5;
end

% Simulate motor features and fault band metrics.
motorCurrent = 308+12*rand(1); % between [308 320] mA
```

```
rpm = 20+3*rand(1); % between [20 23] rpm
bandPower = bandPower + rand(1)/2; % bandPower degradation
metrics = [rand(1), rand(1), bandPower];
features = [metrics, motorCurrent, rpm];

disp('Sending features to ThingSpeak...');
sendFeaturesToThingSpeak(features)
end
```

See Also
faultBands | faultBandMetrics | exponentialDegradationModel | pwelch | tachorpm

Related Examples
- "Analyze Gear Train Data and Extract Spectral Features Using Live Editor Tasks" on page 3-35

ThingSpeak Dashboard for Live RUL Estimation of a Servo Motor Gear Train

This example shows how to setup a ThingSpeak™ dashboard to estimate and visualize the Remaining Useful Life (RUL) of a servo motor gear train. A ThingSpeak dashboard consists of a ThingSpeak channel set up with relevant data streams and MATLAB analysis scripts. The MATLAB scripts run real-time in ThingSpeak.

For this example, a set of synthetic motor current signal data is generated that is used to visualize and predict the RUL of the gear train inside the servo. Motor current signature analysis (MCSA) of the current signal driving a hobby servo motor is used to extract frequency-domain (spectral) features from several frequency regions of interest indicative of motor and gear train faults. A single feature or a combination of features are used construct a Health Indicator (HI) for subsequent RUL estimation. MCSA is a useful method for the diagnosis of faults that induce torque or speed fluctuations in the servo gear train, which in turn result in correlated motor current changes. MCSA has been proven to be ideal for motor fault analysis as only the motor current signal is required for analysis, which eliminates the need for additional and expensive sensing hardware. Gear fault detection using traditional vibration sensors is challenging, especially in cases where the gear train is not easily accessible for instrumentation with accelerometers or other vibration sensors.

For more information about the data stream and hardware setup, see "Motor Current Signature Analysis for Gear Train Fault Detection" on page 3-14 and "Live RUL Estimation of a Servo Gear Train Using ThingSpeak" on page 5-90.

The simplified workflow to estimate and display the RUL of the servo motor gear train includes the following steps:

1 Set up ThingSpeak dashboard
2 Periodically generate servo gear train features
3 Estimate the RUL as the features are generated
4 Setup RUL alert in ThingSpeak

Dashboard Setup

The first step of creating a dashboard is to setup a ThingSpeak channel with the appropriate channel IDs to stream, analyze and display the data. You can then create relevant MATLAB Analysis scripts as described in the successive sections in this example. For more information on setting up a ThingSpeak channel and to stream your data, see "Configure Accounts and Channels" (ThingSpeak) and "Live RUL Estimation of a Servo Gear Train Using ThingSpeak" on page 5-90.

For this example, channel 1558487 was created to run, analyze and display the RUL of the synthetic servo motor data. Run the following command to access this dashboard.

```
web('https://thingspeak.com/channels/1558487')
```

Remaining Useful Life Estimation for Servo Motor

Channel ID: **1558487**
Author: beryilma
Access: Public

🏷 matlab, rul, predictive, maintenance, estimation, servo, motor

| Private View | Public View | Channel Settings | Sharing | API Keys | Data Import / Export |

➕ Add Visualizations   ➕ Add Widgets   🔗 Export recent data   **MATLAB Analysis**   **MATLAB Visualization**

🔗 More Information

Channel Stats

Created: about a month ago
Last entry: less than a minute ago
Entries: 21360

Current RUL Estimate for Servo Motor

13.10357290554929
hour

Field 6 Chart

Channel Status Updates

HEALTHY: More than 13 hours to failure as of 09-Dec-2021 10:30:05
less than a minute ago

Field 3 Chart

Field 1 Chart

Peak Spectrum at 1xFS1

Field 2 Chart

Peak Spectrum at 2xFS1
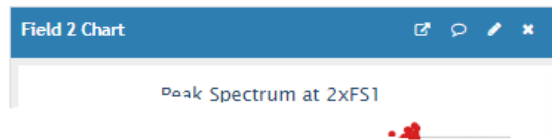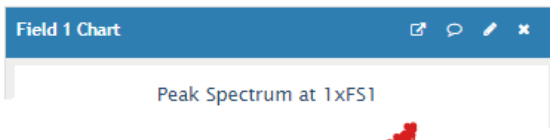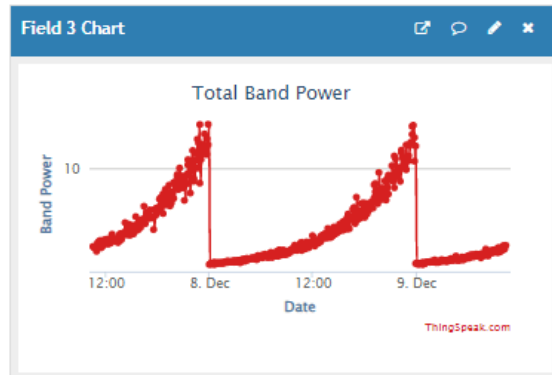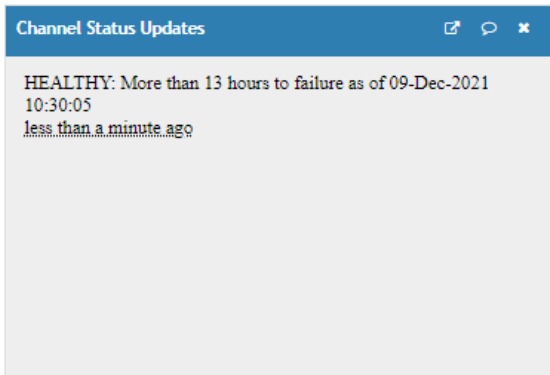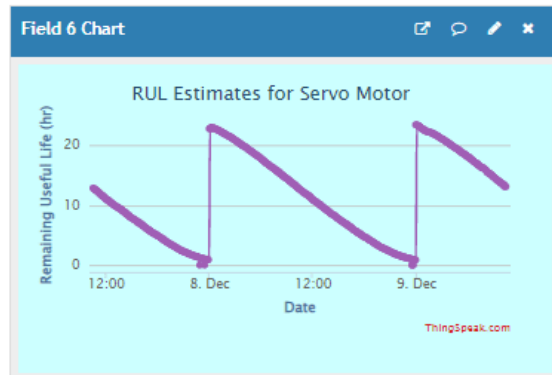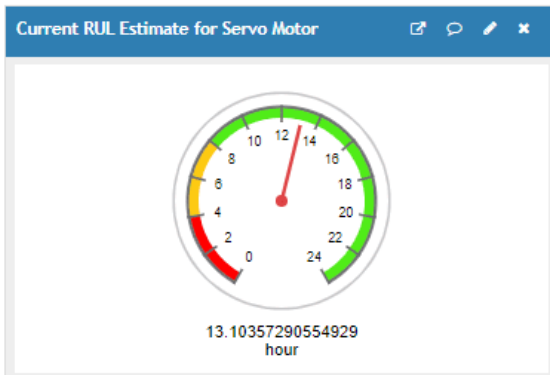
ThingSpeak Dashboard - Remaining Useful Life Estimation

Since this example requires the model state to be stored between evaluations, change the `Metadata` setting of the channel based on the RUL model for this example. You normally do not need to change the `Metadata` setting when setting up your own ThingSpeak channel.

For this example, assume that an `ExponentialDegradationModel` was trained with historical data using the `fit` command. The `fit` command estimates a prior for the model's parameters based on the historical records in training data. The `Prior` property of the trained model contains the estimated model parameters `Theta`, `Beta`, `Rho`, etc. For details of these model parameters, see `exponentialDegradationModel`.

Extract the model's prior from the fitted exponential degradation model and add any fixed parameters (e.g., Phi) as part of the state structure. See the example "Update RUL Prediction as Data Arrives" on page 5-11 for using the fit method to initialize model parameters from training (historical) data.

```
mdl = exponentialDegradationModel;
state = mdl.Prior;
state.Phi = 0.6931;
```

For this example, assume that the model parameter priors are set as follows:

```
state = struct( ...
  'Theta', 0.2790, ...
  'ThetaVariance', 0.0102, ...
  'Beta', 0.1686, ...
  'BetaVariance', 6.5073e-04, ...
  'Rho', -0.9863, ...
  'Phi', 0.6931);
```

Convert the state structure to a string representation so that you can manually copy it to the channel's `Metadata` setting.

```
state = jsonencode(state);
```

Manually save the resulting model state string into the `Metadata` setting of the channel:

Metadata	{"Theta":0.279,"ThetaVariance":0.0102,"Beta":0.1686,"BetaVariance":0.00065073,"Rho":-0.9863,"Phi":0.6931}

Generate Synthetic Servo Motor Features

For this example, the next step is to generate synthetic servo motor features and save the values to the ThingSpeak channel as separate fields. For your use case, you can use feature values from a Cloud source or compute them from real-time hardware streaming data.

The synthetic motor signal data being generated in this example utilizes the servo motor and gear train specifications listed below. The servo consists of four pairs of meshing nylon gears as illustrated in the figure below. The pinion P1 on the DC motor shaft meshes with the stepped gear G1. The pinion P2 is a molded part of the stepped gear G1 and meshes with the stepped gear G2. The pinion P3, which is a molded part of gear G2, meshes with the stepped gear G3. Pinion P4, which is molded with G3, meshes with the final gear G4 that is attached to the output shaft. The stepped gear sets G1 and P2, G2 and P3, and G3 and P4 are free spinning gears — that is, they are not fixed to their respective shafts. The set of gears provides a 278:1 reduction going from a motor speed of about 6117 rpm to about 22 rpm at the output shaft when the DC motor is driven at 5 volts.

For more information about the hardware model, see "Live RUL Estimation of a Servo Gear Train Using ThingSpeak" on page 5-90.

Futaba S3003 standard servo gear train

The following table outlines the tooth count and theoretical values of output speed, gear mesh frequencies, and cumulative gear reduction at each gear mesh.

Pinion	Gear	Pinion Teeth	Gear Teeth	Output Speed (RPM)	Output Speed (Hz)	Gear Mesh Frequency (Hz)	Cumulative Gear Reduction
P1	-	10	-	6116.7	101.94	-	1
P2	G1	10	62	986.6	16.44	1019.4	6.2
P3	G2	10	50	197.3	3.29	164.4	31
P4	G3	16	35	56.4	0.94	32.9	108.5
-	G4	-	41	22	0.37	15	278

Theoretical values calculated from tooth count assuming 22 RPM at the output shaft

The ThingSpeak dashboard for this example uses the following three MATLAB Analysis scripts:

- **Generate Synthetic Servo Motor Features**
- **Calculate Remaining Useful Life Estimate**
- **Send RUL Alert**

You can use the MATLAB Analysis App to explore and analyze the data stored in your ThingSpeak channel. For more information, see "MATLAB Analysis App" (ThingSpeak).

You can find the above MATLAB scripts associated with channel 1558487 in the current directory of this example.

Apps / MATLAB Analysis

Click **New** and choose a template to get started. Templates contain sample MATLAB® code for analyzing data.

New

Name	Created
Generate Synthetic Servo Motor Features	2021-11-03
Calculate Remaining Useful Life Estimate	2021-11-03
Send RUL Alert	2021-11-04

The following "MATLAB Analysis App" (ThingSpeak) code is associated with the **Generate Synthetic Servo Motor Features** script. Use "TimeControl App" (ThingSpeak) to run the script every 5 minutes to trigger the computation of new servo motor features from available or synthetic data. The **Generate Synthetic Servo Motor Features** script is associated with a **TimeControl** configured as follows:

Apps / TimeControl

New TimeControl

Recurring TimeControls

Name	Recurrence	Last Ran	Run At
☑ Generate Synthetic Servo Motor Features View Edit	Every 5 minutes	2021-11-09 1:52 pm	2021-11-09 1:57 pm

The contents of the **Generate Synthetic Servo Motor Features** script are as follows:

Simulate time-domain motor features

For the servo motor used in the example, the motor winding current fluctuates between 308 mA and 320 mA, and the speed fluctuates between 20 rpm and 23 rpm.

```
motorCurrent = 308 + 12*rand(1);
rpm = 20 + 3*rand(1);
```

Simulate frequency domain features of the motor current

The power spectrum amplitudes at motor shaft frequencies (1xFS1 and 2xFS1) and the total band power are simulated as follows:

Input the parameters to simulate complete degradation of the servo motor over the course of a full day.

```
dStart = dateshift(datetime, "start", "day");
dNow = datetime;
dayScale = seconds(dNow - dStart) / (24*3600);
```

Peak power spectrum amplitude at the first harmonic of the motor shaft frequency (1xFS1) changes linearly between -30 dB and 10 dB as degradation increases. Add white noise to simulate more realistic sensor data. For more information about the frequencies, see "Live RUL Estimation of a Servo Gear Train Using ThingSpeak" on page 5-90.

```
peak1FS1 = -30 + (10+30)*dayScale + randn(1);
```

Peak power spectrum amplitude at the second harmonic of the motor shaft frequency (2xFS1) changes linearly between -20 dB and 0 dB as degradation increases. Add white noise to simulate more realistic sensor data.

```
peak2FS1 = -20 + (0+20)*dayScale + randn(1);
```

Total band power of the motor current changes exponentially between 1 and 13 as degradation increases and failure is assumed to occur at the threshold value 13. Add white noise to simulate more realistic sensor data. Use the band power as the health index for RUL estimations.

```
phi = 0.5;
theta = 0.5;
threshold = 13;
beta = log((threshold-phi) / theta);
bandPower = phi + theta * exp(beta*dayScale + randn(1)/10);
```

Configure the Channel ID and the Write API Key values

Replace the [] with channel ID to write data to your ThingSpeak dashboard:

```
writeChannelID = [];
```

Enter the `Write API Key` between the quotes:

```
writeAPIKey = '';
```

Write the features to the channel specified by the `writeChannelID` variable.

```
features = [peak1FS1, peak2FS1, bandPower, motorCurrent, rpm];
fields = 1:numel(features);
thingSpeakWrite(writeChannelID, features, 'Fields', fields, 'WriteKey', writeAPIKey);
```

The features will be displayed in the fields 1-5 of your ThingSpeak channel:
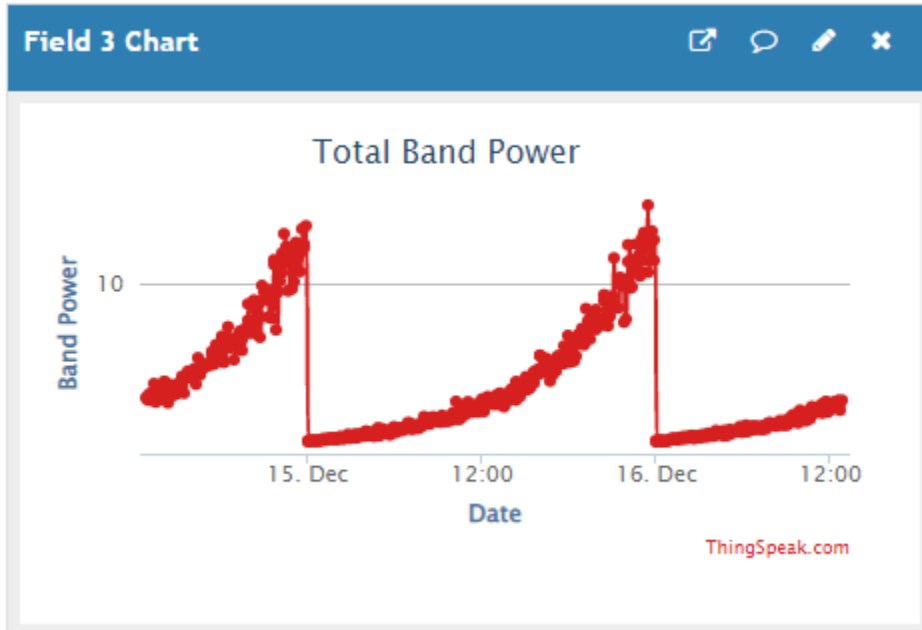
Total Band Power Chart

Calculate Remaining Useful Life Estimate

Next, setup a "MATLAB Analysis App" (ThingSpeak) script that calculates the remaining useful life of the servo motor using the total band power of the motor current as the health indicator. The script also saves the RUL estimate to the ThingSpeak channel in the field **Remaining Useful Life (hr)**.

The following "MATLAB Analysis App" (ThingSpeak) code is associated with the **Calculate Remaining Useful Life Estimate** script. Use the "React App" (ThingSpeak) control that runs whenever a new, valid feature value is added to the **Total Band Power** field of your channel.

Apps / React

New React

Name	Created	Last Ran
☑ Calculate Remaining Useful Life Estimate View Edit	2021-11-03	2021-11-09 1:52 pm
☑ Send RUL Alert View Edit	2021-11-04	2021-11-09 12:31 am

Apps / React / Calculate Remaining Useful Life Estimate / Edit

React Name

Calculate Remaining Useful Life Estimate

Condition Type

Numeric

Test Frequency

On Data Insertion

Condition

If channel

Remaining Useful Life Estimation for Servo Motor (1558487

field

3 (Total Band Power)

is greater than

0

Action

MATLAB Analysis

Code to execute

Calculate Remaining Useful Life Estimate

Options

○ Run action only the first time the condition is met

● Run action each time condition is met

Save React

Configure the channel ID and then read the API key values

Replace the [] with channel ID to read data from your ThingSpeak channel.

```
readChannelID = [];
```

If your channel is private, then enter the **Read API Key** between the quotes.

```
readAPIKey = '';
```

For this example, the **Total Band Power** feature is stored in field 3 of the channel.

```
fieldID = 3;
```

Read the data since the last servo motor replacement

For the example, we assume that the servo motor fails and is replaced every 24 hours at 12:00 AM.

```
dStart = dateshift(datetime, "start", "day");
dNow = datetime;
range = [dStart, dNow];
```

Read feature data and remove any invalid or empty values.

```
data = thingSpeakRead(readChannelID, 'Fields', fieldID, ...
  'ReadKey', readAPIKey, 'DateRange', range, 'OutputFormat', 'timetable');
data = rmmissing(data);
```

Return if there is no valid data available, for example, right after the servo motor was replaced.

```
if isempty(data)
  return;
end
```

Construct the exponential degradation model

Use the **Total Band Power** as the health index and compute the current RUL estimate of the servo motor. Reload from channel **Metadata** setting the initial state of the model parameters obtained from fitting historical data to the degradation model. Use the prior parameters to initialize the degradation model before estimating a new RUL value from available data.

```
state = loadModelState();
mdl = exponentialDegradationModel( ...
  'Theta', state.Theta, ...
  'ThetaVariance', state.ThetaVariance, ...
  'Beta', state.Beta, ...
  'BetaVariance', state.BetaVariance, ...
  'Phi', state.Phi, ...
  'Rho', state.Rho, ...
  'NoiseVariance', 0.1, ...
  'SlopeDetectionLevel', [], ...
  'LifeTimeVariable', data.Properties.DimensionNames{1}, ...
  'LifeTimeUnit', "hours", ...
  'DataVariables', data.Properties.VariableNames{1});
```

Set RUL degradation threshold for the health index (Total Band Power).

```
threshold = 13;
```

The data here is not historical data, but rather recent real-time observations of the servo motor features recorded in the fields of the ThingSpeak channel.

Update the degradation model using available feature values since the servo motor was last replaced. Then, compute the current RUL estimate based on the failure threshold.

```
update(mdl, data)
rul = hours(predictRUL(mdl, threshold));
```

Configure the Channel ID and the Write API Key values

Replace the [] with channel ID to write data to your ThingSpeak channel.

```
writeChannelID = [];
```

Enter the **Write API Key** between the quotes.

```
writeAPIKey = '';
```

Write the updated RUL estimate to the channel specified by the `writeChannelID` variable. For this example, the RUL estimate value is stored in field 6 of the ThingSpeak channel.

```
fieldID = 6;
thingSpeakWrite(writeChannelID, rul, 'Fields', fieldID, 'WriteKey', writeAPIKey);
```

Field 6 of the channel tracks the RUL estimates for the servo motor:
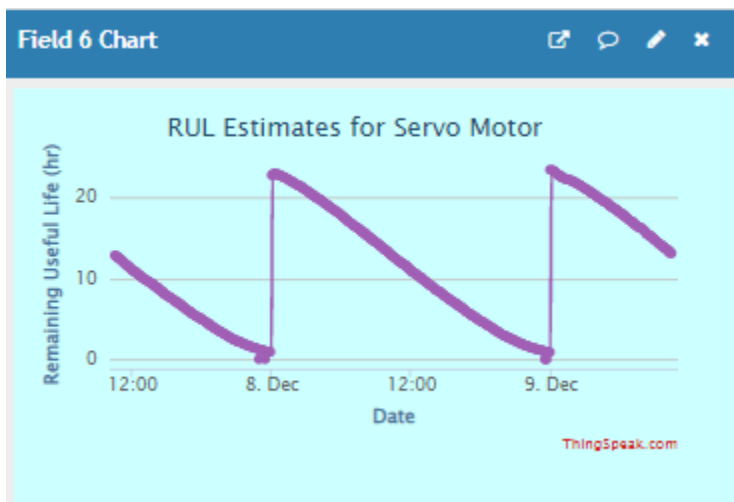
RUL Estimate Chart

Add a **Field Value Gauge** widget to display the current RUL estimate. For more information, see "Channel Display Widgets" (ThingSpeak).
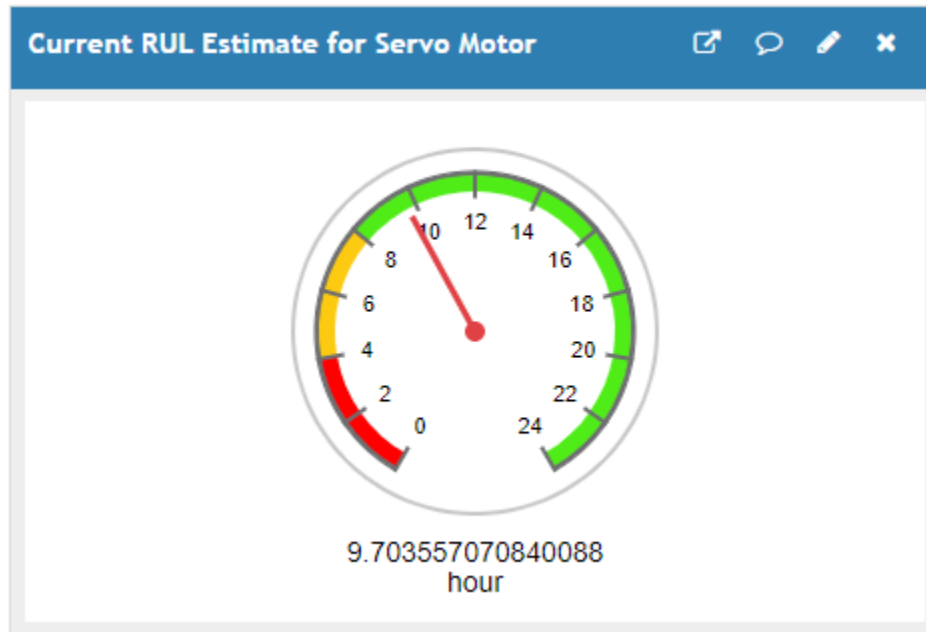
RUL Estimate Gauge

Send RUL Alert

You can also setup a "MATLAB Analysis App" (ThingSpeak) script to send an RUL status alert to the channel about the impending servo failure.

The following "MATLAB Analysis App" (ThingSpeak) code is associated with the **Send RUL Alert** script. Use "React App" (ThingSpeak) control to send a text message alert (as a status update in the channel) every 10 minutes while the current RUL estimate for an impending failure is determined to be less than 4 hours. If the RUL estimate is less than 8 hours, a warning message is displayed.

Apps / React

New React

Name	Created	Last Ran
☑ Calculate Remaining Useful Life Estimate View Edit	2021-11-03	2021-11-09 1:52 pm
☑ Send RUL Alert View Edit	2021-11-04	2021-11-09 12:31 am

Configure the Channel ID value

Replace the [] with channel ID to read data from your ThingSpeak channel.

```
readChannelID = 1558487;
```

In this example, the RUL estimate is stored in field 6 of the channel.

```
fieldID = 6;
```

Read recent RUL estimates and remove any invalid or empty values.

```
data = thingSpeakRead(readChannelID, 'Fields', fieldID, 'NumPoints', 10, ...
  'OutputFormat', 'timetable');
data = rmmissing(data);
```

Configure the Write API Key value

Enter the **Write API Key** value between the quotes:

```
writeAPIKey = '';
```

Post status update message to your ThingSpeak channel.

```
url = sprintf('https://api.thingspeak.com/update.json');
if data.RemainingUsefulLifehr(end) <= 4
  msg = sprintf("ALERT: %3.1f hours to failure detected at %s", ...
    data.RemainingUsefulLifehr(end), string(datetime));
elseif data.RemainingUsefulLifehr(end) <= 8
  msg = sprintf("WARNING: %3.1f hours to failure detected at %s", ...
    data.RemainingUsefulLifehr(end), string(datetime));
else
  msg = sprintf("HEALTHY: More than %3.0f hours to failure as of %s", ...
    data.RemainingUsefulLifehr(end), string(datetime));
end

try
  webwrite(url, 'api_key', writeAPIKey, 'status', msg);
catch E
  fprintf("Failed to send alert:\n %s\n", E.message);
end
```

Any alerts for a pending failure are shown in the **Channel Status Updates** view:
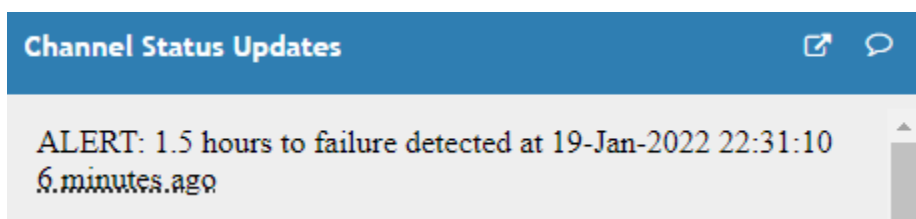
Channel Status Update Window

This functionality can be replaced with code to send text messages, emails, or Twitter updates instead. For more information, see "ThingTweet App" (ThingSpeak) and "ThingHTTP App" (ThingSpeak).

Alternatively, use the following code to send an email alert. For more information and an example, see "Alerts API" (ThingSpeak) and "Analyze Channel Data to Send Email Notification" (ThingSpeak)

```
sendAlertEmail(data.RemainingUsefulLifehr(end));
```

Email Alert Message

Supporting Functions

The `loadModelState` function loads the model state (i.e., the prior for the model parameters) from the **Metadata** setting of the channel and converts it to a MATLAB variable (struct). You can access the following MATLAB scripts in the working directory of this example.

```
function state = loadModelState()
% Configure the Channel ID.
% Replace the [] with channel ID to read data from:
readChannelID = [];

% Read the stored RUL model initialization state from the Metadata field of the
% channel specified by the 'readChannelID' variable.
url = sprintf('https://api.thingspeak.com/channels/%d/feeds.json', readChannelID);
response = webread(url, 'metadata', 'true', 'status', 'true', 'results', 30);

state = jsondecode(response.channel.metadata);
end
```

The `sendALertEmail` function sends an alert email indicating the estimated RUL value.

```
function sendAlertEmail(rul)
alertMessage = sprintf("ALERT: %3.1f hours to failure detected.", rul);
```

```
alertSubject = sprintf("Pending servo motor failure");

% Configure the Alert API Key value.
% Enter the Alert API Key between the quotes below:
alertAPIKey = '';
options = weboptions("HeaderFields", ["ThingSpeak-Alerts-API-Key", alertAPIKey ]);

alertURL = "https://api.thingspeak.com/alerts/send";
try
  webwrite(alertURL, "body", alertMessage, "subject", alertSubject, options);
catch E
  fprintf("Failed to send alert:\n %s\n", E.message);
end
end
```

See Also

faultBands | faultBandMetrics | exponentialDegradationModel | pwelch | tachorpm | **MATLAB Analysis App** | **TimeControl App** | **React App** | "ThingTweet App" (ThingSpeak) | **ThingHTTP App**

Related Examples

- "Live RUL Estimation of a Servo Gear Train Using ThingSpeak" on page 5-90
- "Analyze Gear Train Data and Extract Spectral Features Using Live Editor Tasks" on page 3-35

Battery Cycle Life Prediction Using Deep Learning

Lithium-ion battery cycle life prediction using a physics-based modelling approach is very complex due to varying operating conditions and significant device variability even with batteries from the same manufacturer. Further, every battery ages differently depending on usage and conditions during manufacturing. In this example, we illustrate the use of deep learning technique for estimating the remaining cycles of a fast charging lithium-ion battery. Data representing the full lifecycle of the batteries is used to train a 2D Convolution Neural Network based architecture and this trained network is used to estimate the remaining cycle life of new batteries.

Dataset

The dataset contains measurements from 40 lithium-ion cells with nominal capacity of 1.1 Ah and a nominal voltage of 3.3 V under various charge and discharge profiles. Each battery is charged and discharged, according to one of many predetermined policies, until the battery reaches 80% of its original capacity. The number of cycles until this state is reached is called the battery cycle life. This number varies broadly between 150 and 2300 cycles as seen in the histogram of the data used for this example.
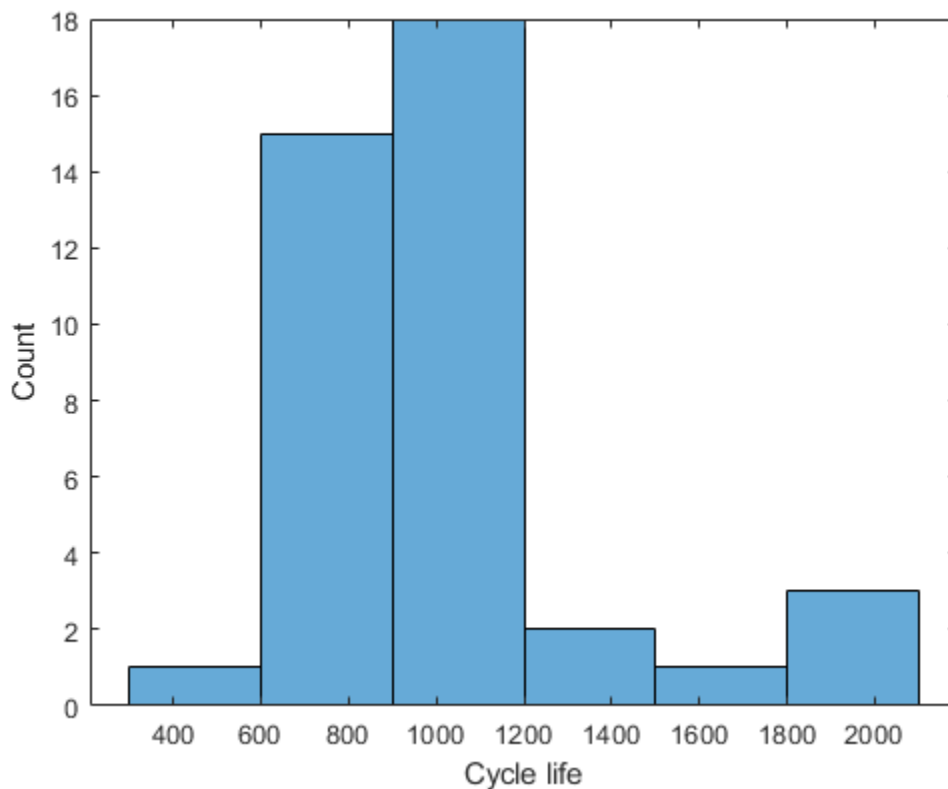
The full dataset containing measurements from 124 cells can be accessed here [2] with detailed description here [1]. This example uses a reduced dataset containing measurements from 40 cells only to make it easier to download and to run this example. Data for each battery is stored in a structure, which includes the following information:

- Data collected within a cycle: Current, voltage, temperature, capacity, differential discharge capacity

Load the data from the MathWorks supportfiles site (this is a large dataset, ~1.2GB).

```
url = 'https://ssd.mathworks.com/supportfiles/predmaint/batterycyclelifeprediction/v2/batteryDisc
websave('batteryDischargeData.zip',url);
unzip('batteryDischargeData.zip')
load('batteryDischargeData');
```

Visualize the data characteristics by creating a plot of current, voltage, and temperature measurements for one full cycle of the first battery in the data.

```
battIndx = 1; cycleIndx = 1;
batteryMeasurements = table(batteryDischargeData(battIndx).cycles(cycleIndx).I,batteryDischargeDa
    batteryDischargeData(battIndx).cycles(cycleIndx).T, batteryDischargeData(battIndx).cycles(cyc
stackedplot(batteryMeasurements, "Title","Measurements over one cycle",...
    "DisplayLabels", ["Current (A)","Voltage(V)","Temperature(C)","DischargeCapacity(Ah)"], ...
    "Xlabel", "Sample Index");
```
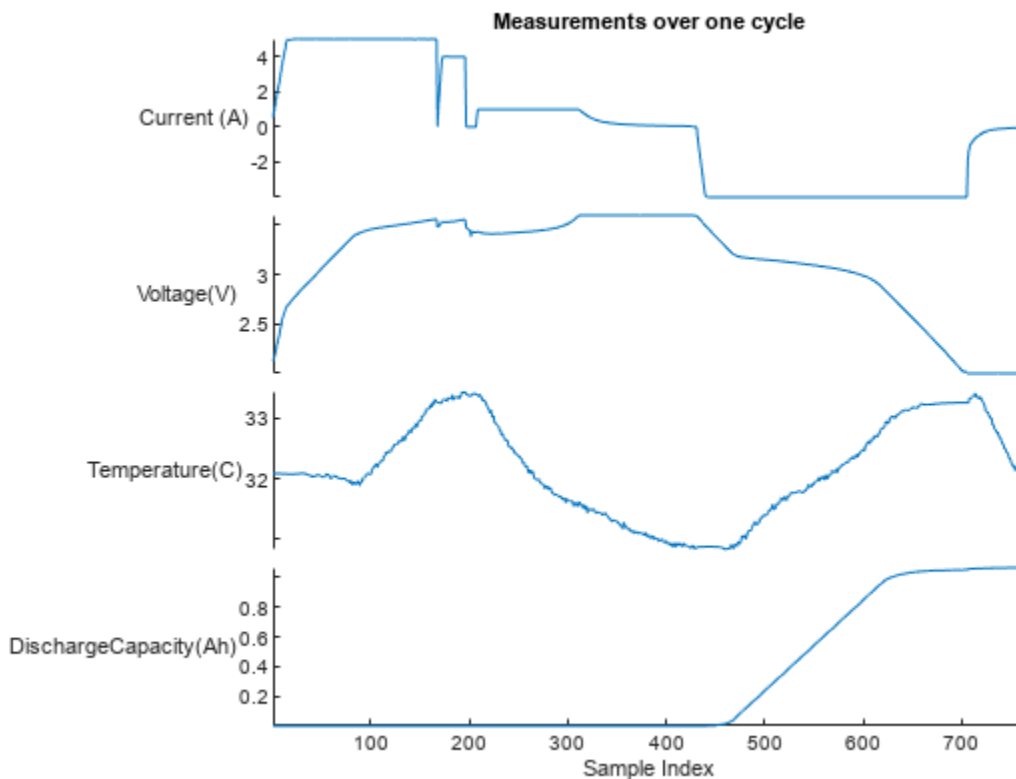

In the preceding plot, positive current indicates a charging process while negative current indicates a discharge operation. The battery is fully charged when it reaches 3.6V and fully discharged when it reaches 2V. Further, the batteries are subjected to different fast charging policies in this dataset to understand their degradation profile across time and load.

Extract Battery Discharge Measurements

Since all batteries have different charging policies but identical discharge voltage range, you use only the discharge portions of the signals in this example. Extract the measurements corresponding to the discharge portion of the cycle using the `hExtractDischargeData` helper function. Plot the discharge data for the first cycle of the first battery.

```
dischargeData = hExtractDischargeData(batteryDischargeData);

batteryMeasurements = table(dischargeData{battIndx}.Vd{cycleIndx},dischargeData{battIndx}.Td{cyc
    dischargeData{battIndx}.QdClipped{cycleIndx});
stackedplot(batteryMeasurements, "Title","Measurements over one cycle",...
    "DisplayLabels", ["Voltage(V)","Temperature(C)","DischargeCapacity(Ah)"],...
    "Xlabel", "Sample Index");
```
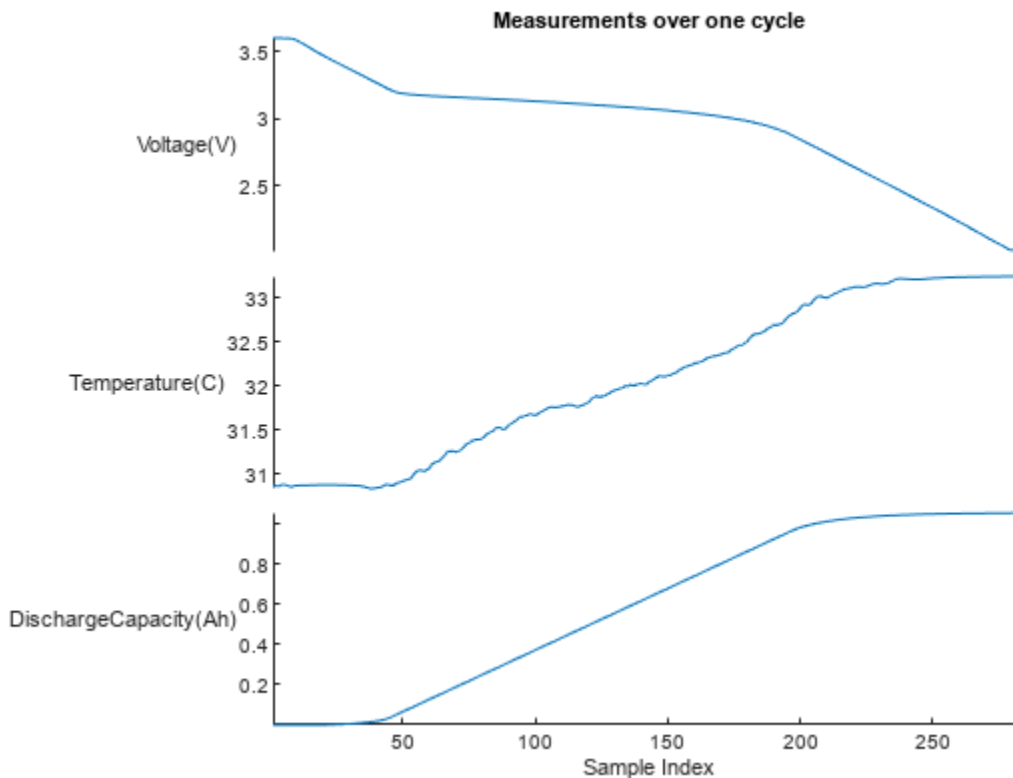

Since the batteries in this data set are tested with different charging policies, some cycles are completed sooner than others. Therefore, cycle time cannot be used to compare charge and temperature across batteries. The voltage range is used as the reference instead of time because the discharge time varies based on the connected load and the health of the batteries. The charge and temperature measurements are then interpolated over this voltage range. Use `hLinearInterpolation` function to interpolate voltage, temperature and discharge capacity measurements onto a uniformly sampled 900 point voltage range between 3.6V and 2V. The interpolated data is returned as a 30x30 array for each measurement to form a 2D representation for each battery discharge cycle. Note that reshaping the `900x1` vector to a `30x30` matrix leads to the convolutional network searching for spatial relation between each column of the matrix. This example assumes that such a relationship might exist across the various cycles and attempts to leverage it if

present. The image below shows the temperature and voltage data for a single cycle of a cell being interpolated to 900 points and then being reshaped to a 30x30. The 2D representation the measurement of each cycle converts the sensor measurements into an image format for the CNN layers.
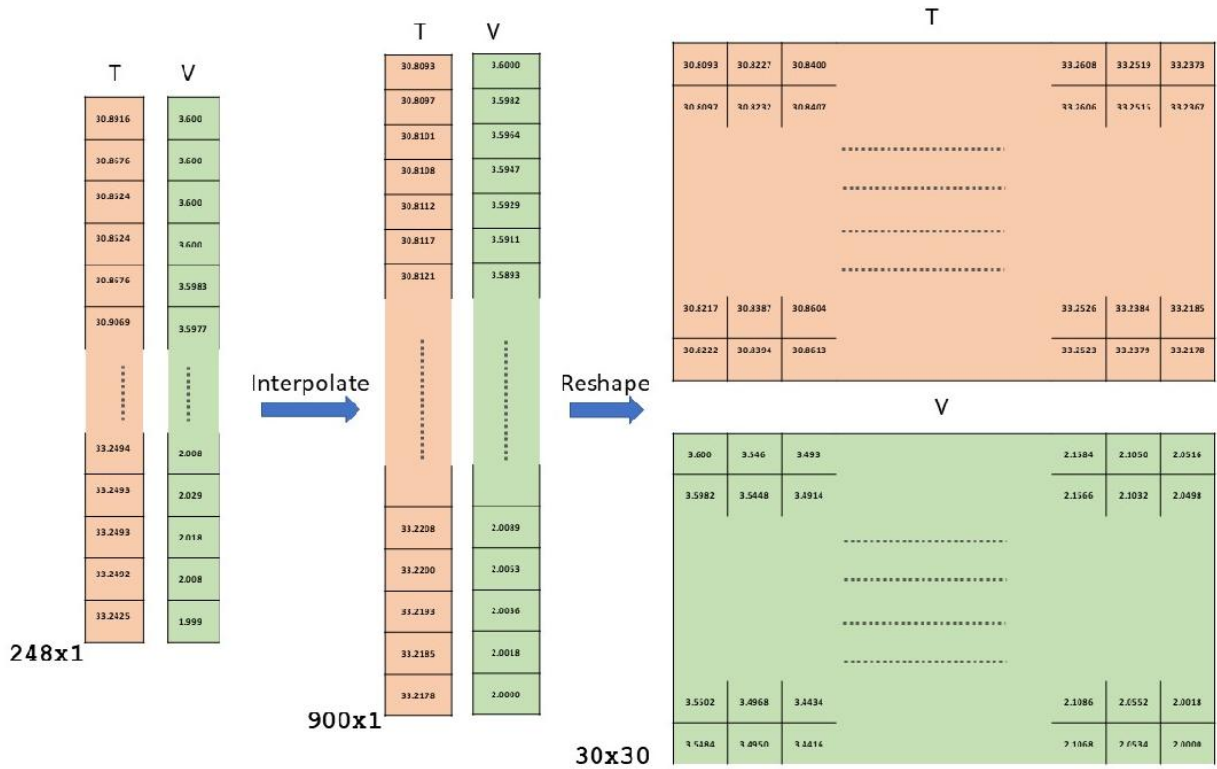

```
[VInterpol,TInterpol,QdInterpol] = hLinearInterpolation(dischargeData);
```

Plot the interpolated temperature and discharge capacity as a function of voltage.

```
figure
yyaxis left
plot(reshape(VInterpol{1}{1}, 900, 1),reshape(TInterpol{1}{1},900,1))
title('Measurements as a function of Voltage')
ylabel('Temperature')
xlabel('Voltage')
yyaxis right
plot(reshape(VInterpol{1}{1},900,1),reshape(QdInterpol{1}{1},900,1))
ylabel('Discharge Capacity')
```
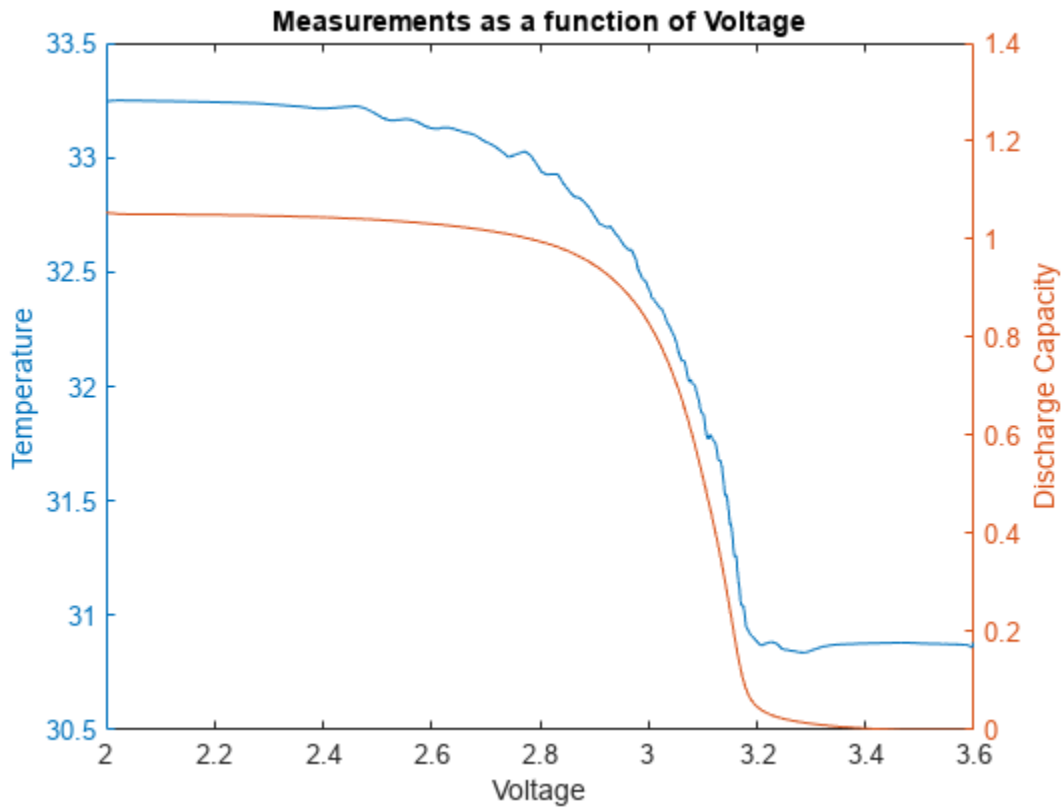
For the 2D Convolution Neural Network layers in the deep network, the 30x30 matrix of interpolated voltage, discharge capacity and temperature are reshaped to a form a 30x30x3 matrix for each cycle. This is like the RGB channels of an image. To minimize the range of the estimated remaining cycles, the expected output signal is normalized by dividing by 2000 (the maximum life of the battery in the data). The data from 30 batteries is used for training, 5 batteries for validation and 5 batteries for testing of the deep neural network. Use the `hreshapeData` helper function to create the 30x30x3 dataset for each cycle. This function outputs the measurement data (`trainData`) and the RUL data (`trainRulData`) to use as labels for each case.

```
testBatteryIndex = 2:8:40;
valBatteryIndex = 1:8:40;
trainBatteryIndex = setdiff(1:40,[2:8:40 1:8:40]);

[trainData,trainRulData] = hreshapeData(VInterpol(trainBatteryIndex), ...
    TInterpol(trainBatteryIndex),QdInterpol(trainBatteryIndex));
[valData,ValRulData] = hreshapeData(VInterpol(valBatteryIndex), ...
    TInterpol(valBatteryIndex),QdInterpol(valBatteryIndex));
[testData,testRulData] = hreshapeData(VInterpol(testBatteryIndex), ...
    TInterpol(testBatteryIndex),QdInterpol(testBatteryIndex));

fprintf('Size of reshaped matrix of interpolated measurement data:%dx%dx%dx%d\n', ...
    size(trainData))
```

```
Size of reshaped matrix of interpolated measurement data:30x30x3x30706
```

Define Network Architecture

Defining the architecture of convolutional neural network includes selecting the types of layers, selecting the number of layers, and tuning the hyperparameters until satisfactory performance is achieved against test data. In this section, the types of layers and number of layers are specified. To create your deep neural network structure, define a set of successive network layers. Use a network structure with the following layers:

- Image input layer — Treat the voltage, discharge capacity, temperature data as the three color channels of an input image and normalize the measurements to the range [0,1].
- 2D convolutional layers — Each of these layers applies sliding convolutional filters to the image input. This example uses four hidden convolutional layers. This number of layers, which was selected through trial and error, gives the best result while keeping a reasonable training time.
- Batch normalization layers — Each convolutional layer is followed by a batch normalization layer, which speeds up the training of the network and reduces the sensitivity to network initialization.
- ReLU layers — Each batch normalization layer is followed by a nonlinear activation function, which performs a threshold operation to each element of the input.
- Pooling layers — The first two batch ReLU layers are followed by pooling layers, which reduce the size of the feature map and remove redundant information, which reduces the number of parameters to be learned in subsequent layers.
- Drop-out layer — The final ReLU layer is followed by a dropout layer, which helps reduce overfitting in the network.
- Fully connected layer — The dropout layer is followed by a fully connected layer, which combines all of the learned features into a single input to the regression layer.
- Regression layer — Since the estimation of remaining useful life is a regression problem, the final output layer of the network is a regression layer.

```
layers = [
    imageInputLayer([30 30 3],"Normalization","rescale-zero-one")
    convolution2dLayer(3,8,"Padding","same")
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,"Stride",2)
    convolution2dLayer(3,16,"Padding","same")
    batchNormalizationLayer
    reluLayer
    averagePooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    dropoutLayer(0.5)
    fullyConnectedLayer(1)
    regressionLayer];
figure
plot(layerGraph(layers))
```
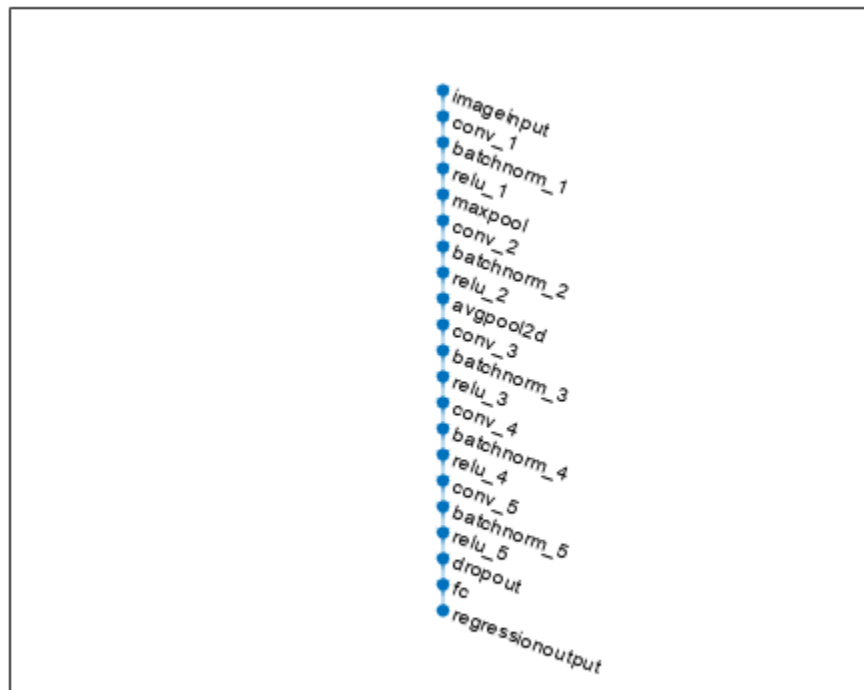
Define Network Hyperparameters and Train Network

In this section, define the hyperparameters for the network specified in the previous section. Selecting hyperparameters, such as the learning rate or batch size, is generally through trial and error with the goal of finding the best set for the selected network and the data set to achieve satisfactory performance from the network.

For this example, use the Adam (adaptive moment estimation) optimizer, which has a fast computation time and few parameters to tune. Configure the solver to:

- Use a mini-batch size of 256 observations.
- Train on the entire data set 50 times, which the number of training epochs.
- Shuffle the dataset before each epoch to improve convergence.
- Use a learning rate of 0.001, which achieves a good balance between convergence and overshooting.
- Validate the network periodically to identify when the network is overfitting the training data.

For more information on training options for the Adam solver, see `TrainingOptionsADAM` (Deep Learning Toolbox). The training hyperparameters used in this example were selected based using trial-and-error experimentation. You can adjust the parameters to further improve the training.

```
miniBatchSize = 256;
validationFrequency = 10*floor(numel(trainRulData)/miniBatchSize);
options = trainingOptions("adam", ...
    "MaxEpochs",100, ...
```

```
        "MiniBatchSize",miniBatchSize, ...
        "Plots","training-progress", ...
        "Verbose",false, ...
        "Shuffle","every-epoch", ...
        "InitialLearnRate",0.001, ...
        "OutputNetwork","best-validation-loss", ...
        "ValidationData",{valData, ValRulData}, ...
        "ValidationFrequency",validationFrequency, ...
        "ValidationPatience",10, ...
        "ResetInputNormalization",false);

rng("default")

batteryNet = trainNetwork(trainData, trainRulData,layers,options);
```
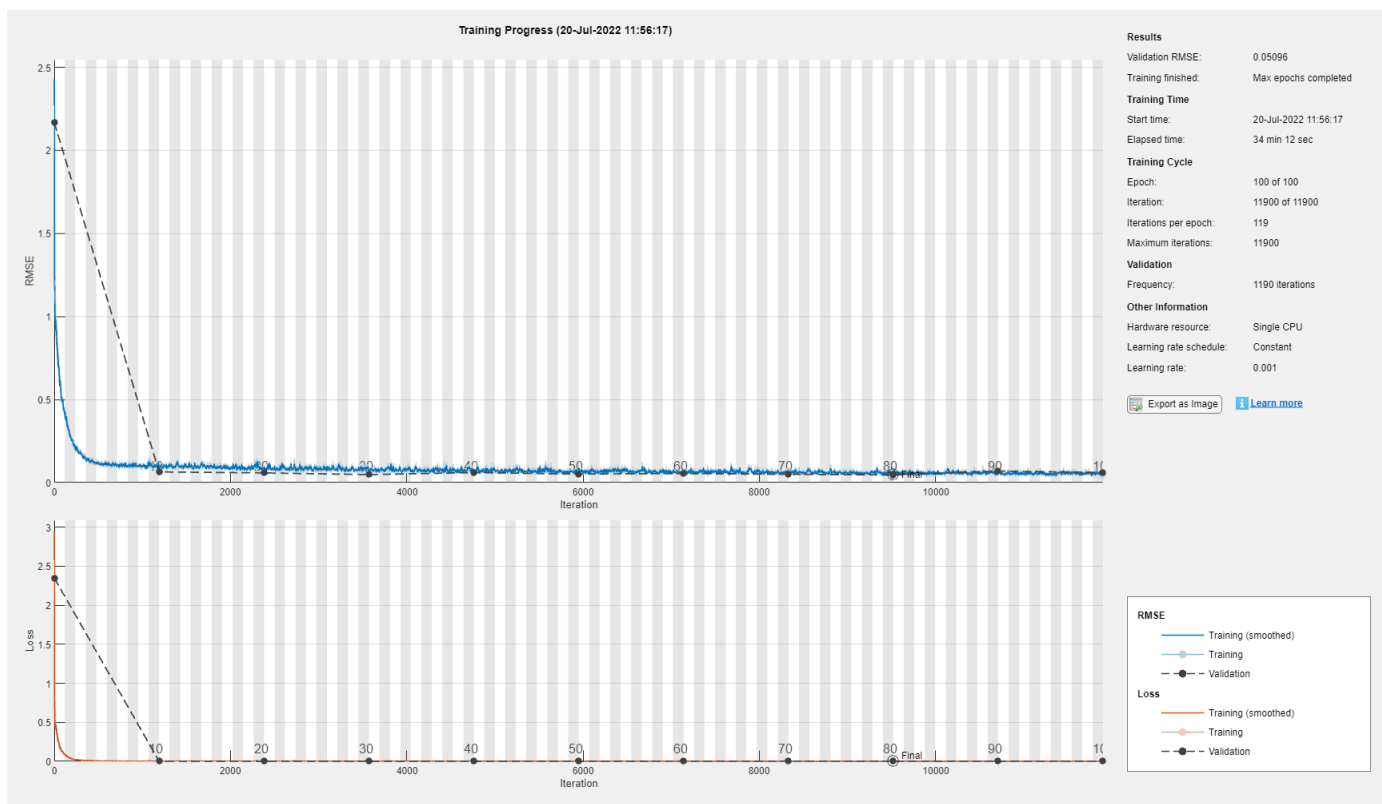

Evaluate Performance of Trained Model

Use the trained model to predict the remaining cycle life for `testData`. The values must be rescaled back to the original RUL range to make it easier to visualize the performance.

```
yPredTest = predict(batteryNet,testData)*2000;
testRulScaled = testRulData*2000;
```

Compare the actual cycle life with the predicted cycle life using a scatter plot.

```
figure;
scatter(testRulScaled,yPredTest)
hold on;
refline(1,0);
```

```
title("Predicted vs Actual Cycle Life")
ylabel("Predicted cycle life");
xlabel("Actual cycle life");
```

Predicted vs Actual Cycle Life
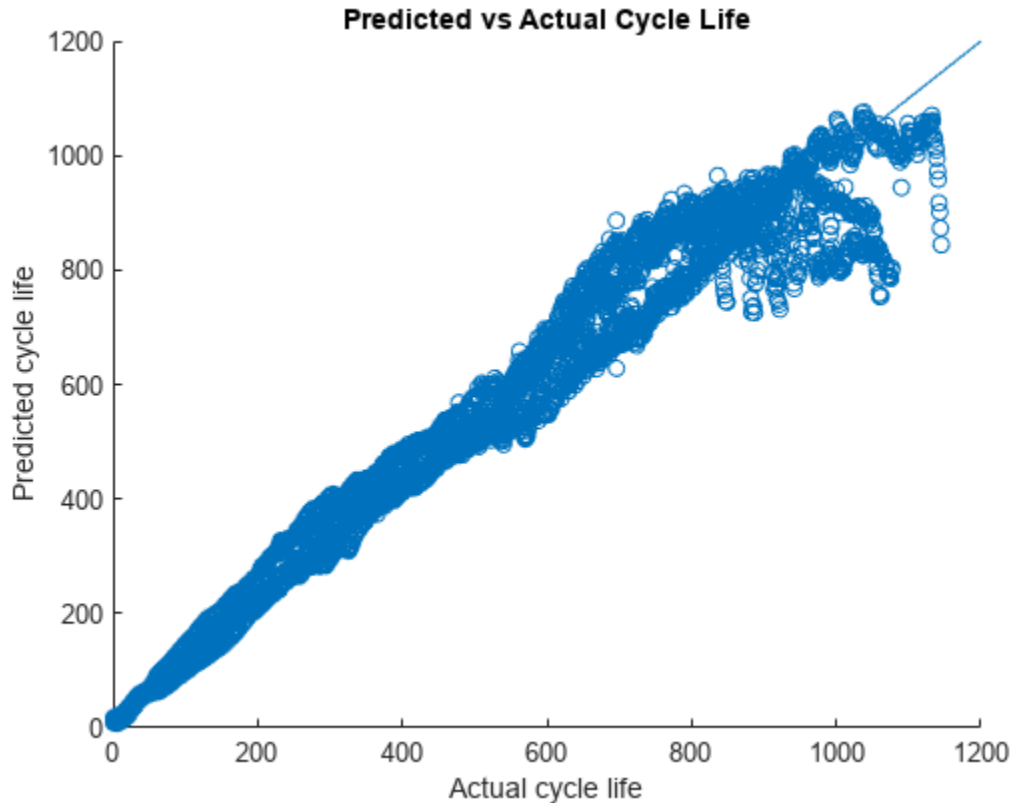
Ideally, the scatter plot should have all data points along the diagonal with a narrow confidence band. However, in this example, there is a broader spread and different behaviors for different range of values in the scatter plot. In the scatter plot, there five distinct trends, one for each battery in the test data.

Across the five batteries, when the actual cycle life is small, the model is good at predicting the remaining useful life. This result implies that, as a battery gets closer to the end of its life, the model is good at predicting the remaining cycle life.

However, during the early part of a battery's life when the actual cycle life is larger, the model has greater uncertainty. The model also seems to generally overestimate the remaining cycle life during the initial period of a battery's life. To address these model characteristics, you can train the network using richer and larger data sets and experiment with the deep neural network architecture and its hyperparameters.

Compute the root mean squared error (RMSE) and the average percentage error of the predicted remaining cycle life.

```
errTest = (yPredTest-testRulScaled);
rmseTestModel = sqrt(mean(errTest.^2))
```

```
rmseTestModel = single
    65.2403
```

```
n = numel(testRulScaled);
nr = abs(testRulScaled - yPredTest);
errVal = (1/n)*sum(nr./testRulScaled)*100
```

```
errVal = single
    15.4083
```

These performance metrics are relatively close to their equivalent values for when a linear regression model with regularization is used with custom features for estimating remaining cycle life, as shown in "Battery Cycle Life Prediction From Initial Operation Data" on page 5-81. Note that in the machine learning based example only the first 100 cycles of data is used for estimating the remaining cycle life while in this example data from any cycle can be used. This result indicates that depending on the application and system requirements, either a machine learning or a deep learning approach can be used to estimate the remaining cycle life of batteries.

After tuning the model to a desired performance level, you can operationalize it to estimate remaining cycle life of batteries in use. To deploy a trained network to embedded hardware, generate C/C++, GPU, or HDL code. For more information, see "Code Generation" (Deep Learning Toolbox). To deploy the trained network to the cloud, choose the appropriate packaging option.

Conclusion

This example shows how to use deep learning techniques for battery cycle life prediction based on measurements from 40 batteries. Raw sensor signals are directly used as inputs to train a deep neural network without any manual extraction of features. This model is used on test data for performance evaluation. Using measurements for the test data, the average percentage error is ~15%.

Helper Functions

```
function [dischargeData] = hExtractDischargeData(data)
% HEXTRACTDISCHARGEDATA Extract measurements corresponding to discharge
% portion of cycle
dischargeData = cell(1, size(data, 2));
% For each battery in the data (which has many charge discharge cycles)
for iBattery = 1:size(data,2)
    timeSeriesTable = struct2table(data(iBattery).cycles);
    % Keep only the data related to discharge [ between 3.6V and 2 V)
    clipIdxFun1 = @(x) {find(x{1,1}>=3.6,1,"last")};
    clipIdxFun2 = @(x) {find(x{1,1}<=2.00,1,"first")};

    clipIdx1 = rowfun(clipIdxFun1,timeSeriesTable,"InputVariables","V",...
        "OutputVariableNames","clipIdx1");
    clipIdx2 = rowfun(clipIdxFun2,timeSeriesTable,"InputVariables","V",...
        "OutputVariableNames","clipIdx2");
    timeSeriesTable = [timeSeriesTable clipIdx1 clipIdx2];

    clipSignals = @(x,y,z) {smoothdata(x(y:z),"movmean",3)};
    % Extract Voltage
    Vd = rowfun(clipSignals,timeSeriesTable,"InputVariables",...
        ["V","clipIdx1","clipIdx2"],"OutputVariableNames","Vd",...
        "ExtractCellContents",true);
    % Extract Temperature
    Td = rowfun(clipSignals,timeSeriesTable,"InputVariables",...
        ["T","clipIdx1","clipIdx2"],"OutputVariableNames","Td",...
        "ExtractCellContents",true);
    % Extract Discharge Capacity
```

```matlab
        QdClipped = rowfun(clipSignals,timeSeriesTable,"InputVariables",...
            ["Qd","clipIdx1","clipIdx2"],"OutputVariableNames","QdClipped",...
            "ExtractCellContents",true);

        dischargeData{iBattery} = [Vd Td QdClipped];
    end
end

function [Vdlin, Tdlin, Qdlin] = hLinearInterpolation(dischargeData)
% HLINEARINTERPOLATION Interpolate on the voltage range of 2V to 3.6V
% linear interpolation onto 900 points between the two voltages and the
% data is then reshaped into a 30x30 matrix
Vdlin = cellfun(@(x)rowfun(@hLinInterp,x,"InputVariables",["Vd","Vd"],...
    "OutputVariableNames","Vdlin","OutputFormat","cell"), dischargeData, ...
    'UniformOutput', false);

Tdlin = cellfun(@(x)rowfun(@hLinInterp,x,"InputVariables",["Vd","Td"],...
    "OutputVariableNames","Tdlin","OutputFormat","cell"), dischargeData, ...
    'UniformOutput', false);

Qdlin = cellfun(@(x)rowfun(@hLinInterp,x,"InputVariables",["Vd","QdClipped"],...
    "OutputVariableNames","Qdlin","OutputFormat","cell"), dischargeData, ...
    'UniformOutput', false);
end


function xInterpolated = hLinInterp(volt,x)
% HLININTERP Function to linearly interpolate data for battery voltage discharge range

volt = volt{1,1};
x = x{1,1};

% Set seed for consistent results
rng("default");

% Linearly interpolate voltage range 3.6 to 2.
voltRange = linspace(3.6,2,900);
[~, ia, ~] = unique(volt,'sorted');
f = griddedInterpolant(volt(ia),x(ia));

xInterpolated= reshape(f(voltRange)',[30,30]);
end

function [signalData, rul] = hreshapeData(VInterpol, TInterpol, QdInterpol)
%    HRESHAPEDATA Arrange the data as 30x30x3 - where each 30x30 is the 900 point
%    interpolated version for a single discharge and 3 is for V, Q, T
for i =1:numel(VInterpol)
    VData = VInterpol{i};
    TData = TInterpol{i};
    QdData = QdInterpol{i};
    predictor = zeros(30,30,3,size(VData,1));
    for j = 1: size(VData,1)
        temp(:,:,1) = VData{j,1};
        temp(:,:,2) = QdData{j,1};
        temp(:,:,3) = TData{j,1};
        predictor(:,:,:,j) = temp;
    end
```

```
maxBatteryLife = 2000; % Used for scaling output
numCycles = size(VData,1);
cycle = (1:numCycles)';
rulBattery = (numCycles+1 - cycle)/maxBatteryLife;

if i == 1
    signalData = predictor;
    rul = rulBattery;
else
    signalData = cat(4,signalData,predictor);
    rul = [rul; rulBattery];
end
end
end
```

References

[1] Severson, K.A., Attia, P.M., Jin, N. *et al.* "Data-driven prediction of battery cycle life before capacity degradation." *Nat Energy* **4,** 383–391 (2019). https://doi.org/10.1038/s41560-019-0356-8

[2] https://data.matr.io/1/

See Also

Related Examples

- "Battery Cycle Life Prediction From Initial Operation Data" on page 5-81
- "Remaining Useful Life Estimation Using Convolutional Neural Network" on page 4-118
- "Nonlinear State Estimation of a Degrading Battery System" on page 5-69

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:
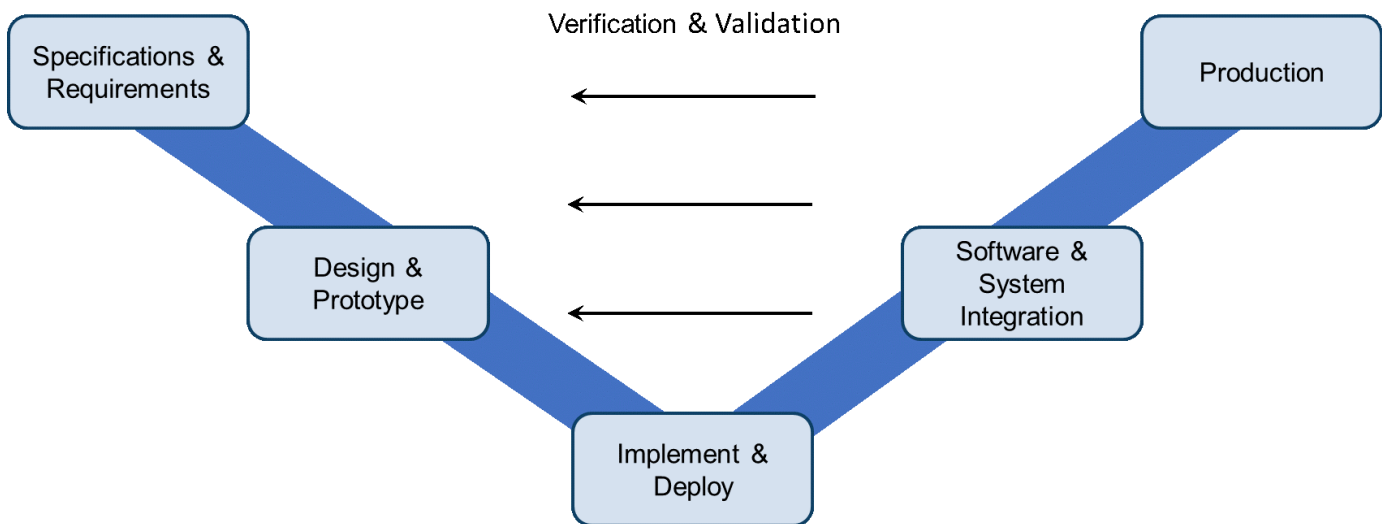
Specifications and Requirements

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 Coder and Simulink Coder

In some cases, you can use MATLAB Coder™ and Simulink Coder to generate C/C++ code from MATLAB or Simulink. For example:

- You can generate code for estimating remaining useful life with all Predictive Maintenance Toolbox RUL model types using MATLAB Coder. For examples, see "Generate Code for Predicting Remaining Useful Life" on page 6-6 and "Generate Code That Preserves RUL Model State for System Restart" on page 6-11.
- Many System Identification Toolbox functions support code generation. For example, you can generate code from algorithms that use state estimation (such as `extendedKalmanFilter`) and recursive parameter estimation (such as `recursiveAR`).
- Many Signal Processing Toolbox and Statistics and Machine Learning Toolbox functions and objects also support MATLAB Coder.

For a more comprehensive list of MathWorks functionality with code-generation support, see "Functions and Objects Supported for C/C++ Code Generation" (MATLAB Coder).

MATLAB Compiler

Use MATLAB Compiler™ to 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 "Get Started with MATLAB Compiler" (MATLAB Compiler).

MATLAB Production Server

Use MATLAB Production Server™ to 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

The ThingSpeak™ 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.

Where to Deploy

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

See Also

More About

- "Designing Algorithms for Condition Monitoring and Predictive Maintenance"

Generate Code for Predicting Remaining Useful Life

This example shows how to deploy an algorithm for predicting remaining useful life (RUL) using MATLAB® Coder™. Such code generation is useful when you have trained an RUL prediction model in MATLAB and are ready to deploy the prediction algorithm to another environment. This example uses MATLAB Coder to generate a MEX file that is executable from MATLAB. You can use a similar procedure to generate code for any target that MATLAB Coder supports.

This example shows how to generate C code for predicting RUL using a `linearDegradationModel`. You can use the same procedure to generate code for a prediction algorithm with any Predictive Maintenance Toolbox™ RUL model, including degradation models, similarity-based models such as `residualSimilarityModel`, and survival-based models such as `covariateSurvivalModel`.

The workflow for generating code for predicting RUL is illustrated in the following diagram. The first step is to fit an RUL model using historical data from your system. You must also write a MATLAB function that receives new data from your system and uses it with the fitted model to predict a new RUL. This function is called the *entry-point function*. You then generate C/C++ code from the entry-point function.
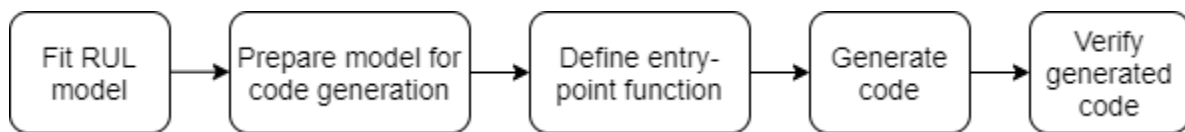
Fit RUL Model

Before generating code for RUL prediction, you must fit an RUL model using historical data. For this example, load the data in `linTrainTables.mat`. This file contains measurements of some condition indicator taken over time, organized into tables with column labels `"Time"` and `"Condition"`. Use this data to train a linear degradation model. (For more information about configuring and training RUL models, see `linearDegradationModel` or the reference pages for the other RUL model types.)

```
load('linTrainTables.mat')
```

```
mdl = linearDegradationModel;
fit(mdl,linTrainTables,"Time","Condition")
```

Prepare Model for Code Generation

Once you have a trained RUL model, save the model using `saveRULModelForCoder`. This function saves the RUL model to a MAT file. Later, in the entry-point function, use `loadRULModelForCoder` to load and reconstruct the RUL model from that file.

```
saveMATfilename = 'savedModel.mat';
saveRULModelForCoder(mdl,saveMATfilename);
```

Define Entry-Point Function

The *entry-point function* is the function for which you want to generate code. When predicting RUL, your entry-point function might take input data, process it in some way to extract the condition indicator, and then use `predictRUL` to obtain a new RUL estimate from the model. For an example that shows the complete workflow for identifying condition indicators, processing data, and predicting RUL, see "Wind Turbine High-Speed Bearing Prognosis" on page 5-37. For this example, create the simple entry-point function `degradationRULPredict.m`, as shown here.

```
type degradationRULPredict.m

function [estRUL,ci,pdfRUL] = degradationRULPredict(data)
%#codegen

threshold = 60;

% Load prepared model
mdl = loadRULModelForCoder('savedModel.mat');

% Use input data for new prediction
[estRUL,ci,pdfRUL] = predictRUL(mdl,data,threshold);

end
```

This function takes as input a data point consisting of a time and a condition-indicator value. The function uses `loadRULModelForCoder` to load the version of the trained model previously saved with `saveRULModelForCoder`. The function also includes the required `%#codegen` directive, which instructs the Code Analyzer to help you diagnose and fix violations that might result in errors during code generation or at runtime. (For more information about the Code Analyzer, see "Check Code with the Code Analyzer" (MATLAB Coder).)

Capabilities and Limitations of the Entry-Point Function

The simple entry-point function of this example loads the model and obtains the new RUL prediction. Your entry-point function can do other operations, such as further processing on input data to extract a condition indicator for use in the prediction. However, all functions and operations within the entry-point function must support code generation. For degradation-based RUL models, your function can also use the `update` command to update the prediction model based on new data. When you do so, you can include additional code to preserve the updated model parameters when you shut down and restart the deployed system. For more information, see "Generate Code That Preserves RUL Model State for System Restart" on page 6-11.

For information about limitations on code generation for RUL models, see the reference pages for `predictRUL` and for the individual RUL model types, in the **Extended Capabilities** section.

Generate Code

To generate code, you must provide sample data having the data type and format expected by the entry-point function. For this example, load some test data in the same format as the data you used to train the RUL model, a table of times and condition-indicator values. Because your entry-point function takes as its input one time and one value, extract one row from the table of test data. For code generation, the specific values of the sample data do not matter, only the data types.

```
load('linTestData.mat','linTestData1')
sampleData = linTestData1(1,:);
sampleData

sampleData=1×2 table
    Time     Condition
    ____     _____

     1        2.1316
```

You can now generate code in one of two ways: using the MATLAB Coder app, or at the MATLAB command line.

Generate Code with MATLAB Coder App

In the MATLAB desktop, on the **Apps** tab, under **Code Generation**, click **MATLAB Coder**. The MATLAB Coder app opens on the Select Source Files page. In the **Generate code for function** box, enter the name of the entry-point function, `degradationRULPredict`. Then, click **Next**.
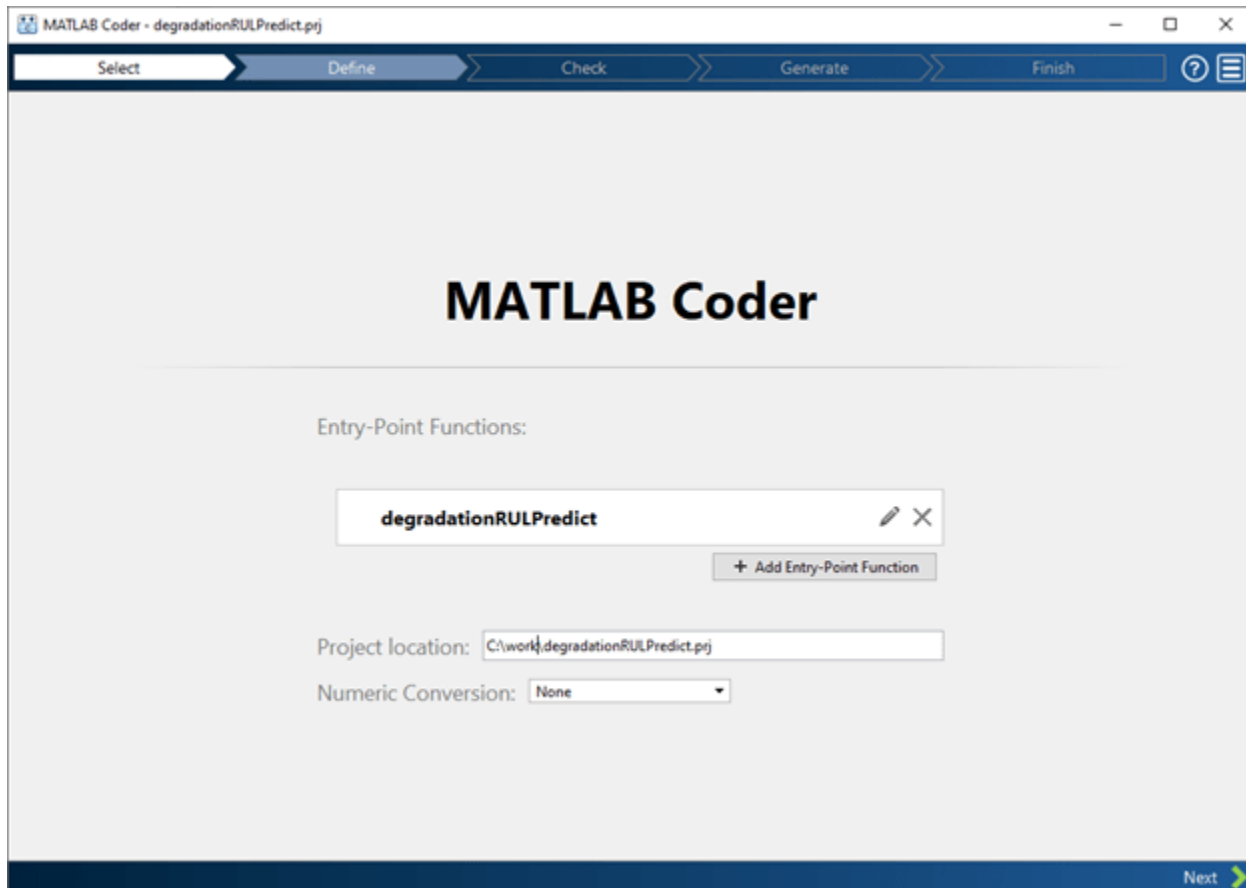
To specify the input data types for the entry-point function, on the Define Input Types page, use `sampleData` in a call to `degradationRULPredict`. When you enter the call, MATLAB Coder displays the detected input types and number of outputs. Click **Next** to confirm.
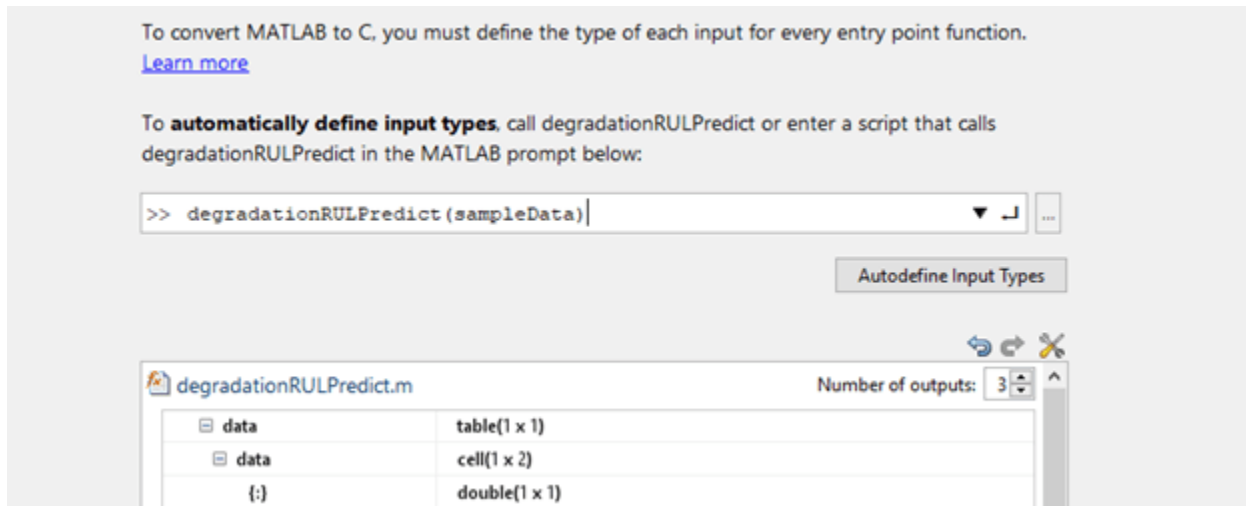
Optionally, check the entry-point function for issues arising at run time. To do so, click **Check for Issues**. When you are ready, click **Next** to advance to the Generate Code page. In this page, you specify the target for code generation. You can generate RUL prediction code for any of the targets that MATLAB Coder supports, including standalone C/C++ code, C/C++ code compiled to a library, or C/C++ code compiled to an executable. For this example, in the **Build type** list, select MEX. A MEX file is an executable that you can call from within MATLAB.
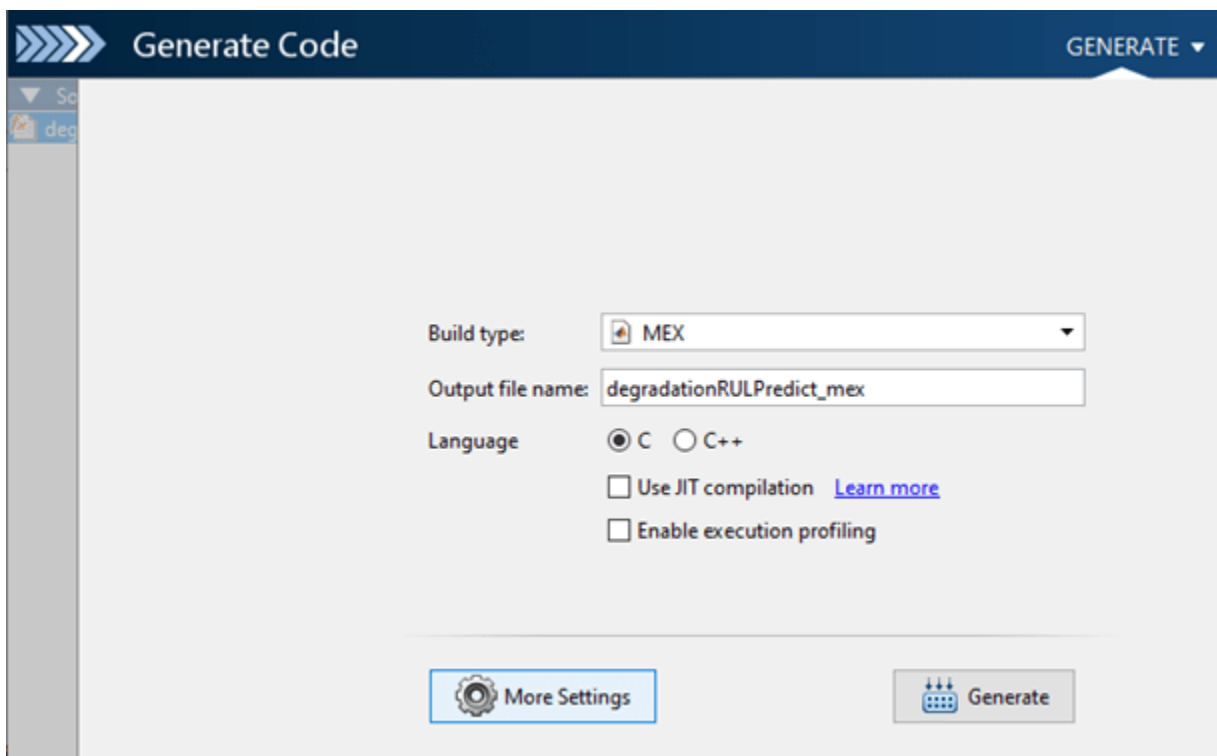
Click **Generate** to generate the MEX file, `degradationRULPredict_mex`. For additional information about MATLAB Coder capabilities and the files it generates, see "Generate C Code by Using the MATLAB Coder App" (MATLAB Coder).

Generate Code with codegen Command

As an alternative to using the MATLAB Coder app, you can generate code using the following codegen (MATLAB Coder) command.

```
codegen degradationRULPredict -args {sampleData} -nargout 3
```

```
Code generation successful.
```

Validate Generated Code

To validate the generated code, at the MATLAB command prompt, run the entry-point MATLAB function on the test data. Then, run the generated MEX file on the same data and confirm that the results are the same.

```
[estRUL,ci,pdfRUL] = degradationRULPredict(sampleData);
```

```
[estRUL_mex,ci_mex,pdfRUL_mex] = degradationRULPredict_mex(sampleData);
```

For example, compare the estimated RUL obtained with the MATLAB function and the generated MEX file.

```
estRUL
```

```
estRUL = 114.2927
```

```
estRUL_mex
```

```
estRUL_mex = 114.2927
```

You can now use the generated code as part of your deployed system for predicting remaining useful life.

See Also
saveRULModelForCoder | loadRULModelForCoder | predictRUL | linearDegradationModel | exponentialDegradationModel | covariateSurvivalModel | reliabilitySurvivalModel | pairwiseSimilarityModel | hashSimilarityModel | residualSimilarityModel

Related Examples
- "Generate Code That Preserves RUL Model State for System Restart" on page 6-11

Generate Code That Preserves RUL Model State for System Restart

This example shows how to generate code for predicting remaining useful life (RUL) that preserves the state of the RUL model when the prediction algorithm is stopped and restarted. This example builds on the workflow shown in "Generate Code for Predicting Remaining Useful Life" on page 6-6.

The workflow of this example is useful for degradation-based RUL models, `linearDegradationModel` and `exponentialDegradationModel`. With these models, you can use the `update` command to update the RUL model with incoming data during system operation, as described in "Update RUL Prediction as Data Arrives" on page 5-11. When you perform such an update in a deployed prediction algorithm, on system shutdown, you risk losing any updates you made to the model during operation. This example shows how to write an entry-point function to preserve updates made to the model at run time. To do so, incorporate the `readState` and `restoreState` commands into the entry-point function.

Fit RUL Model

Before generating code for RUL prediction, you must fit an RUL model using historical data. For this example, load the data in `linTrainTables.mat`. This file contains measurements of some condition indicator taken over time, organized into tables with column labels `"Time"` and `"Condition"`. Use this data to train a linear degradation model. (For more information about configuring and training this type of RUL model, see `linearDegradationModel`.)

```
load('linTrainTables.mat')

mdl = linearDegradationModel;
fit(mdl,linTrainTables,"Time","Condition")
```

Prepare Model for Code Generation

As described in "Generate Code for Predicting Remaining Useful Life" on page 6-6, to prepare the model for code generation, use `saveRULModelForCoder` to store the RUL model as a data structure in a MAT file. Later, in the entry-point function, you use `loadRULModelForCoder` to load and reconstruct the `linearDegradationModel` object.

```
saveMATfilename = 'savedModel.mat';
saveRULModelForCoder(mdl,saveMATfilename);
```

Optionally, you can update the model parameters using additional data collected after the training data. For this example, load `linTestData1`, which is a 121-row table of times and condition-indicator values. Use the first two entries in this table to update the model.

```
load('linTestData.mat','linTestData1')
updateData = linTestData1(1:2,:);
update(mdl,updateData);
```

Next, read the model state using `readState`. This command converts the RUL model object into a structure that you can pass to the entry-point function as an input argument.

```
savedState = readState(mdl);
```

Define Entry-Point Function

The entry-point function is the function for which you want to generate code. For this example, create the entry-point function `degradationRULPreserveState.m`, as shown here.

```
type degradationRULPreserveState.m

function [estRULOut,ciOut,newState] = degradationRULPreserveState(data,restoreStateFlag,savedSta
%#codegen

persistent mdl

% Load the model the first time function is called
if isempty(mdl)
    mdl = loadRULModelForCoder('savedModel');
end

% Restore the saved model parameters if needed
if restoreStateFlag
    restoreState(mdl,savedState);
end

% Update model and prediction with new data
threshold = 60;
update(mdl,data);
[estRULOut,ciOut] = predictRUL(mdl,threshold);

% Read the updated model parameters
newState = readState(mdl);

end
```

This function updates the RUL prediction model using the `update` command each time it is called with new input data. Declaring `mdl` as a `persistent` variable preserves the updated model parameters between calls while the function remains in memory. This function writes the updated model parameters to the output argument `newState`. Save this value outside the entry-point function to preserve the updated model state when the function is cleared from memory. Thus, for example, when restarting the prediction algorithm after a system shutdown, you can set `restoreStateFlag` to `true` and pass in the most recently saved state value as `savedState`, ensuring that the system resumes prediction using the most recently updated model parameters.

Generate Code

To generate code, you must provide sample data having the data type and format of each input argument expected by the entry-point function. For this example, use the next row in `linTestData1`. Also, set `restoreStateFlag` to the logical value `true`.

```
sampleData = linTestData1(3,:);
restoreStateFlag = true;
```

Now you can generate code using the following `codegen` (MATLAB Coder) command. The list of variables `{sampleData,restoreStateFlag,savedState}` tells the `codegen` command that the function takes as arguments a table row consisting of a time and a numerical value, a logical value, and a structure of the form `savedState`, as returned by `readState`.

```
codegen degradationRULPreserveState -args {sampleData,restoreStateFlag,savedState} -nargout 3
```

```
Code generation successful.
```

This command generates a MEX file, which is an executable that you can call from within MATLAB®. You can use procedures similar to this example to generate code for any other target that `codegen` supports. Alternatively, you can use the MATLAB Coder™ app, as shown in "Generate Code for Predicting Remaining Useful Life" on page 6-6.

Validate the Generated Code

To validate the generated code, at the MATLAB command prompt, run the entry-point MATLAB function on the sample data. Then, run the generated MEX file on the same data and confirm that the results are the same.

```
[estRUL,ci,newState] = degradationRULPreserveState(sampleData,restoreStateFlag,savedState);
[estRUL_mex,ci_mex,newState_mex] = degradationRULPreserveState_mex(sampleData,restoreStateFlag,sa
```

For example, compare the estimated RUL obtained with the MATLAB function and the generated MEX file.

```
estRUL,ci
```

```
estRUL = 113.8920
```

```
ci = 1×2
```

```
   83.0901   172.5393
```

```
estRUL_mex,ci_mex
```

```
estRUL_mex = 113.8920
```

```
ci_mex = 1×2
```

```
   83.0901   172.5393
```

You can now use the generated code as part of your deployed system for predicting remaining useful life. Store the value of `newState` to use when you need to restart the system. For example, take the next data point in `linTestData1` and use it to update the prediction, starting with `newState_mex`, the model state generated in the previous call to the MEX file.

```
nextData = linTestData1(4,:);
[estRUL2_mex,ci2_mex,newState2_mex] = degradationRULPreserveState_mex(nextData,restoreStateFlag,n
```

```
estRUL2_mex,ci2_mex
```

```
estRUL2_mex = 104.4336
```

```
ci2_mex = 1×2
```

```
   77.8216   154.1263
```

Note that the MATLAB function and the MEX file each manage their own persistent variables. Therefore, the MEX file value of `mdl` now incorporates the update from `nextData`, which the function value of `mdl` does not. To reset the values of these persistent variables, explicitly clear the function and the MEX file from memory.

```
clear degradationRULPreserveState
clear degradationRULPreserveState_mex
```

See Also

loadRULModelForCoder | saveRULModelForCoder | readState | restoreState | linearDegradationModel | exponentialDegradationModel | covariateSurvivalModel | reliabilitySurvivalModel | pairwiseSimilarityModel | hashSimilarityModel | residualSimilarityModel

Related Examples

*   "Generate Code for Predicting Remaining Useful Life" on page 6-6

7

Diagnostic Feature Designer

Explore Ensemble Data and Compare Features Using Diagnostic Feature Designer

The **Diagnostic Feature Designer** app allows you to accomplish the feature design portion of the predictive maintenance workflow using a multifunction graphical interface. You can design and compare features interactively and then determine which features are best at discriminating between data from different groups, such as nominal systems and faulty systems. If you have run-to-failure data, you can also evaluate which features are best for determining remaining useful life (RUL). The most effective features can ultimately become your condition indicators for fault diagnosis and prognostics.

The following figure illustrates the relationship between the predictive maintenance workflow and **Diagnostic Feature Designer** functions.
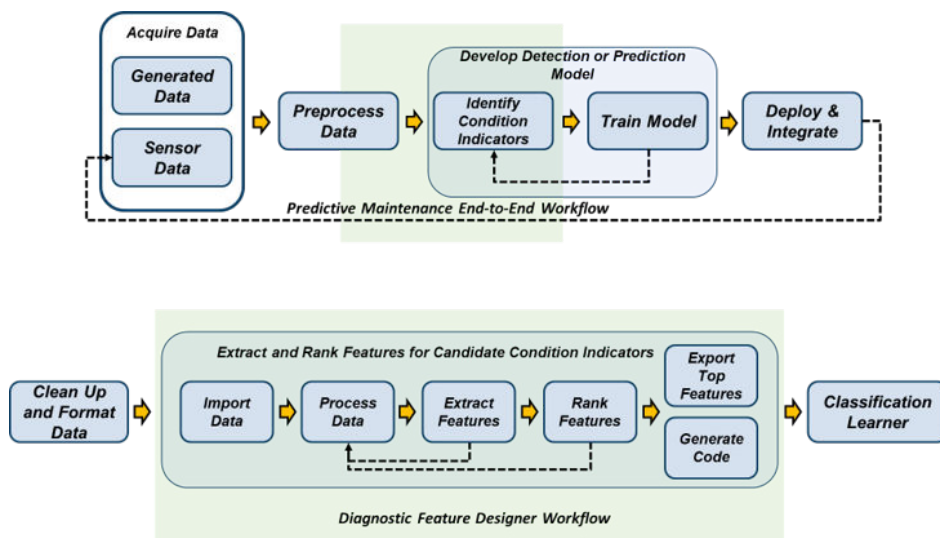
The app operates on ensemble data. Ensemble data contains data measurements from multiple members, such as from multiple similar machines or from a single machine whose data is segmented by time intervals such as days or years. The data can also include condition variables, which describe the fault condition or operating condition of the ensemble member. Often condition variables have defined values known as labels. For more information on data ensembles, see "Data Ensembles for Condition Monitoring and Predictive Maintenance" on page 1-2.

The in-app workflow starts at the point of data import with data that is already:

- Preprocessed with cleanup functions
- Organized into either individual data files or a single ensemble data file that contains or references all ensemble members

Within **Diagnostic Feature Designer**, the workflow includes the steps required to further process your data, extract features from your data, and rank those features by effectiveness. The workflow concludes with selecting the most effective features and exporting those features to the **Classification Learner** app for model training.

The workflow includes an optional MATLAB code generation step. When you generate code that captures the calculations for the features you choose, you can automate those calculations for a

larger set of measurement data that includes more members, such as similar machines from different factories. The resulting feature set provides additional training inputs for **Classification Learner**.

Perform Predictive Maintenance Tasks with Diagnostic Feature Designer

The following image illustrates the basic functionalities of **Diagnostic Feature Designer**. Interact with your data and your results by using controls in tabs such as the **Feature Designer** tab, which is shown in the figure. View your imported and derived variables, features, and datasets in the **Data Browser**. Visualize your results in the plotting area.
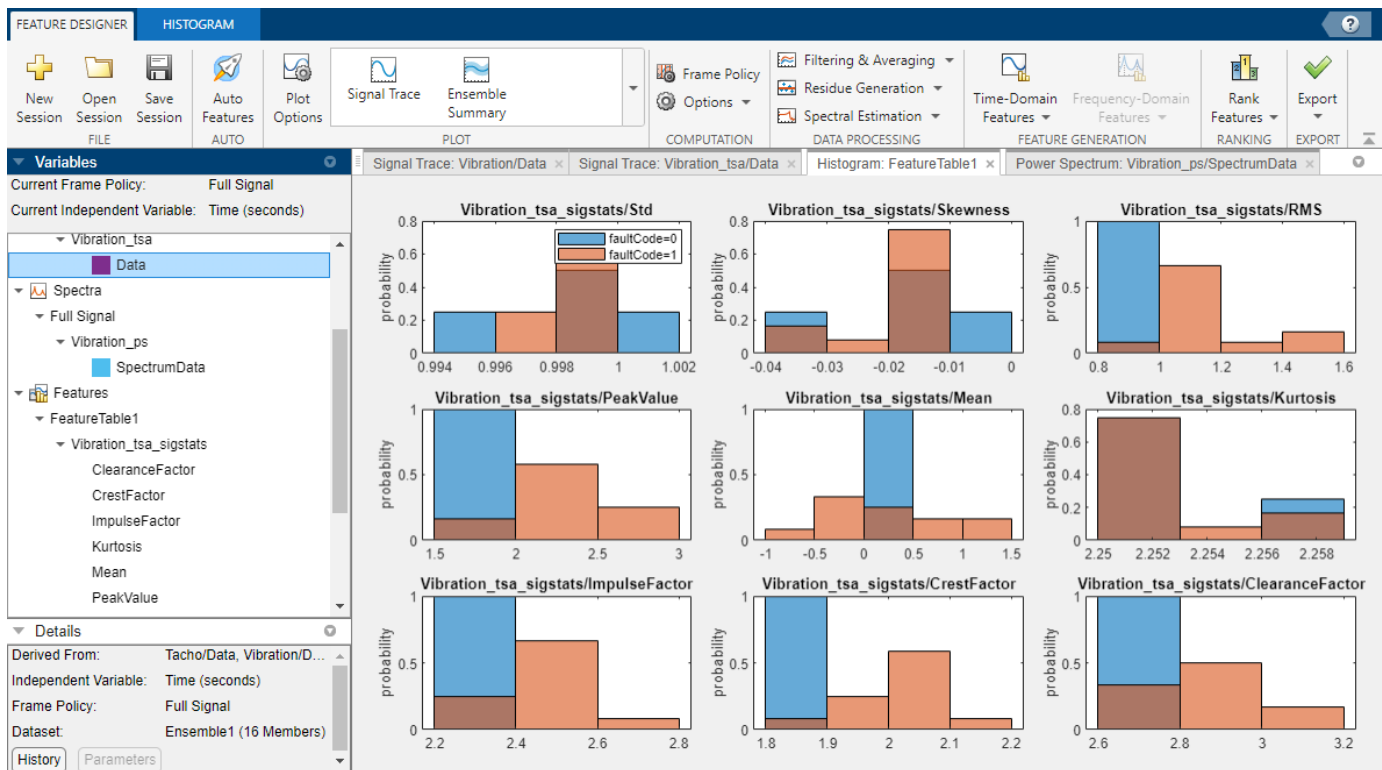
Convert Imported Data into Unified Ensemble Dataset

The first step in using the app is to create a new session and import your data. You can import data from tables, timetables, cell arrays, or matrices. You can also import an ensemble datastore that contains information that allows the app to interact with external data files. Your files can contain actual or simulated time-domain measurement data, spectral models or data, variable names, condition and operational variables, and features you generated previously. **Diagnostic Feature Designer** combines all your member data into a single ensemble dataset. In this dataset, each variable is a collective signal or model that contains all the individual member values.

To use the same data in multiple sessions, you can save your initial session in a session file. The session data includes both imported variables and any additional variables and features you have computed. In a subsequent session, you can then open the session file and continue working with your imported and derived data.

For information on preparing and importing your data, see:

- "Import Data into Diagnostic Feature Designer" on page 7-127
- "Organize System Data for Diagnostic Feature Designer" on page 7-19
- "Prepare Matrix Data for Diagnostic Feature Designer" on page 7-10
- "Generate and Use Simulated Data Ensemble" on page 1-10
- "File Ensemble Datastore with Measured Data" on page 1-17

For information on the import process itself, see "Import and Visualize Ensemble Data in Diagnostic Feature Designer".

Visualize Data

To plot the signals or spectra that you import or that you generate with the processing tools, select a plot type from the plot gallery. The figure here illustrates a typical signal trace. Interactive plotting tools allow you to pan, zoom, display peak locations and distances between peaks, and show statistical variation within the ensemble. Grouping data by condition label in plots allows you to clearly see whether member data comes from, for example, nominal or faulty systems.
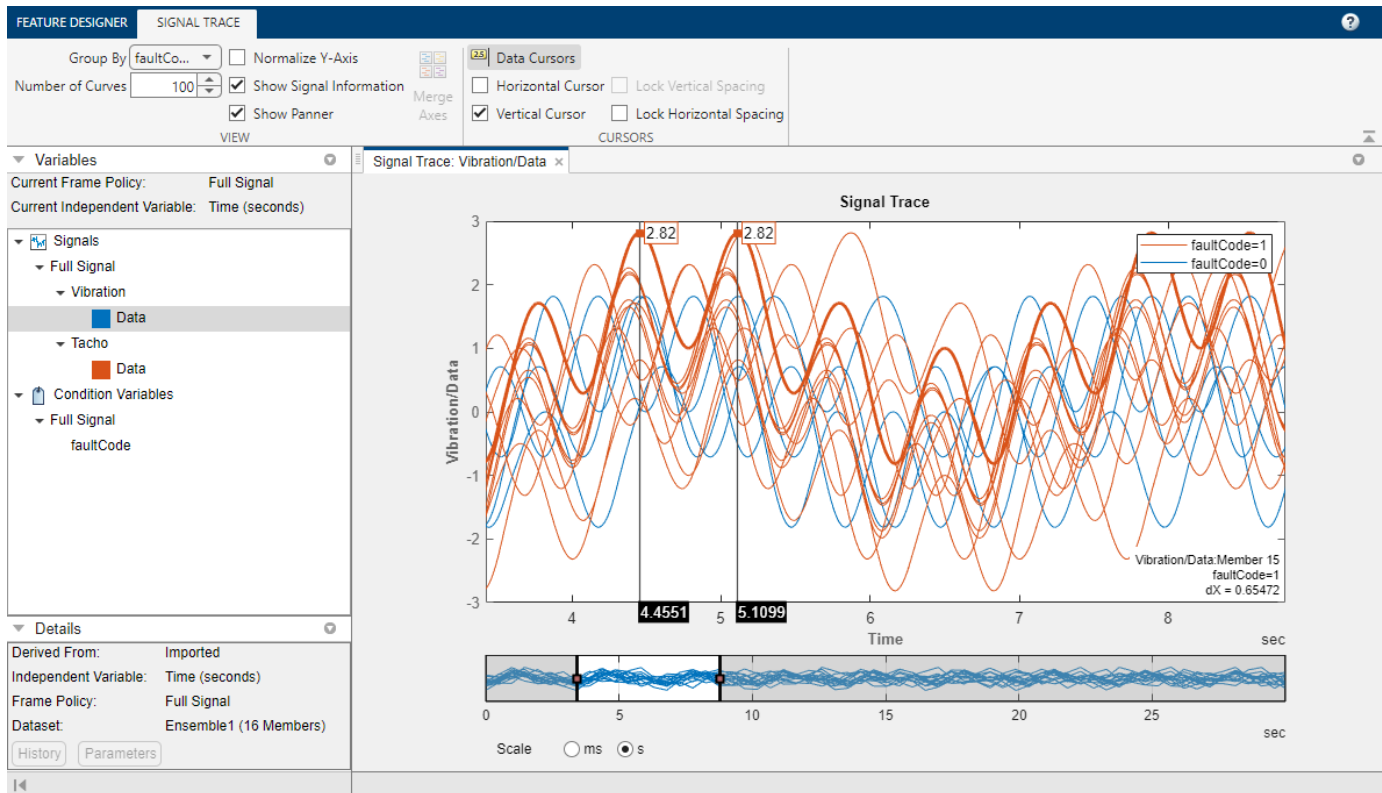
For information on plotting in the app, see "Import and Visualize Ensemble Data in Diagnostic Feature Designer".

Compute New Variables

To explore your data and to prepare your data for feature extraction, use the data processing tools. Every time you apply a processing tool, the app creates a new derived variable with a name that contains both the source variable and the most recent processing step that you used. For example:

- If you apply time-synchronous signal averaging (TSA) processing to the variable `Vibration/Data`, the new derived variable name is `Vibration_tsa/Data`.
- If you then compute a power spectrum from `Vibration_tsa/Data`, the new variable name is `Vibration_ps/SpectrumData`. This new name reflects both the most recent processing `ps` and the fact that the variable is a spectrum rather than a signal.

To obtain more information about a variable, use the **Details** pane. In this area, you can find information such as the direct source signal and the independent variable. You can also plot the processing history for the variable and view the parameters for the most recent operation.

Data processing options for all signals include ensemble-level statistics, signal residues, filtering, and power and order spectra. You can also interpolate your data to a uniform grid if your member samples do not occur at the same independent variable intervals.

If your data comes from rotating machinery, you can perform TSA processing based on your tachometer outputs or your nominal rpm. From the TSA signal, you can generate additional signals such as TSA residual and difference signals. These TSA-derived signals isolate physical components within your system by retaining or discarding harmonics and sidebands, and they are the basis for many of the gear condition features.

Many of the processing options can be used independently. Some options can or must be performed as a sequence. In addition to the rotating machinery and TSA signals previously discussed, another example is residue generation for any signal. You can:

1 Use **Ensemble Statistics** to generate single-member statistical variables such as mean and max that characterize the entire ensemble.
2 Use **Subtract Reference** to generate residue signals for each member by subtracting the ensemble-level values. These residues represent the variation among signals, and can more clearly reveal signals that deviate from the rest of the ensemble.
3 Use these residual signals as the source for additional processing options or for feature generation.

For information on data processing options in the app, see "Process Data and Explore Features in Diagnostic Feature Designer".

Frame-Based and Parallel Processing

The app provides options for frame-based (segmented) and parallel processing.

By default, the app processes your entire signal in one operation. You can also segment the signals and process the individual frames. Frame-based processing is particularly useful if the members in your ensemble exhibit nonstationary, time-varying, or periodic behavior. Frame-based processing also supports prognostic ranking, since it provides a time history of feature values.

If you have Parallel Computing Toolbox™, you can use parallel processing. Because the app often performs the same processing independently on all members, parallel processing can significantly improve computation time.

Generate Features

From your original and derived signals and spectra, you can compute features and assess their effectiveness. You might already know which features are likely to work best, or you might want to experiment with all the applicable features. Available features range from general signal statistics, to

specialized gear condition metrics that can identify the precise location of faults, to nonlinear features that highlight chaotic behavior.

Any time you compute a set of features, the app adds them to the feature table and generates a histogram of the distribution of values across the members. The figure here illustrates histograms for two features. The histograms illustrate how well each feature differentiates labeled data. For example, suppose that your condition variable is `faultCode` with states `0` (color blue) for nominal-system data and `1` (color orange) for faulty-system data, as shown in the following figure. You can see in the histogram whether the nominal and faulty groupings result in distinct or intermixed histogram bins by the presence of color mixing within the bins. You can view all the feature histograms at once or select which features the app includes in the histogram plot set. In the figure, the `CrestFactor` bins are either primarily pure blue or pure orange, which indicates good differentiation. The `Kurtosis` histogram bins are primarily a dark orange that is a mix between blue and orange, indicating poor differentiation.
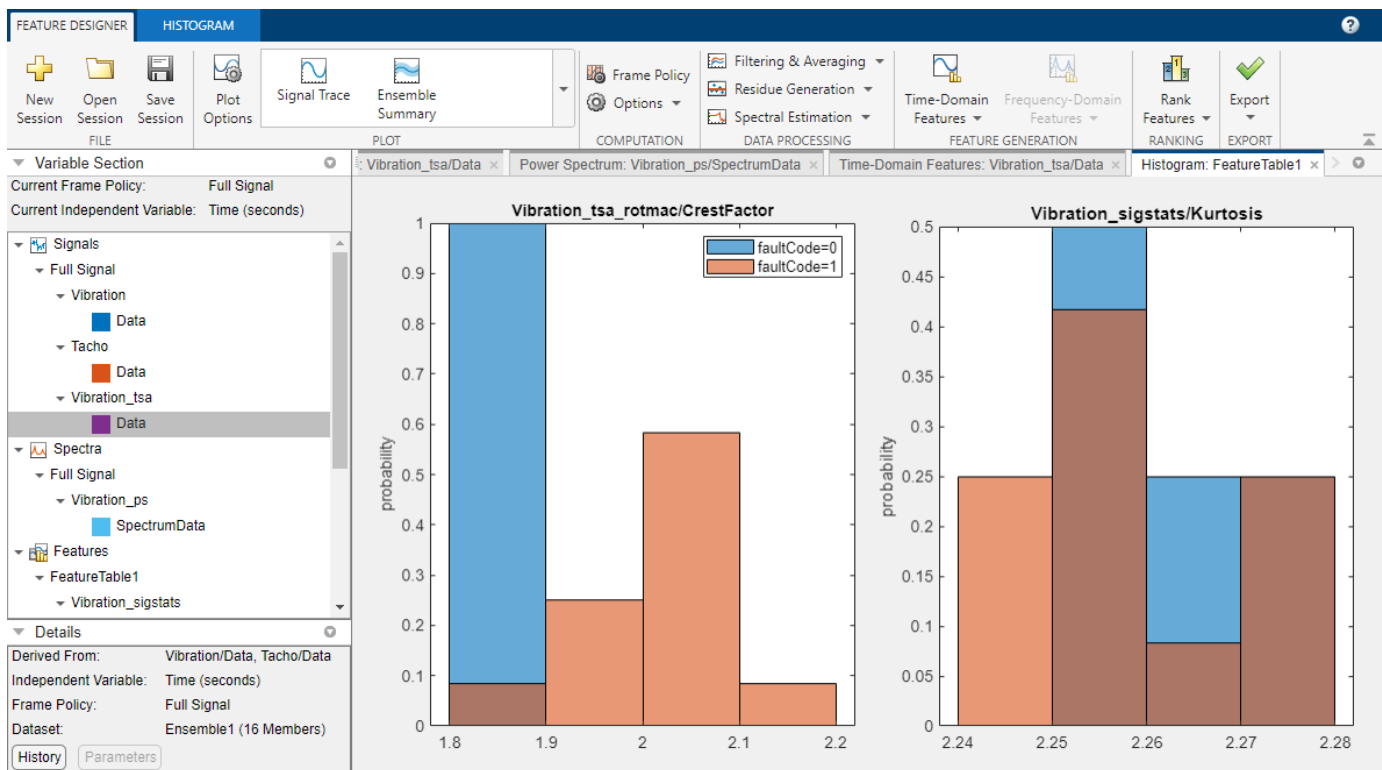
To compare the values of all your features together, use the feature table view and the feature trace plot. The feature table view displays a table of all the feature values of all the ensemble members. The feature trace plots these values. This plot visualizes the divergence of feature values within your ensemble and allows you to identify the specific member that a feature value represents.

For information on feature generation and histogram interpretation in the app, see:

- "Process Data and Explore Features in Diagnostic Feature Designer"
- "Interpret Feature Histograms in Diagnostic Feature Designer" on page 7-13

Rank Features

Histograms allow you to perform an initial assessment of feature effectiveness. To perform a more rigorous relative assessment, you can rank your features using specialized statistical methods. The app provides three types of ranking: supervised ranking, unsupervised ranking, and prognostic ranking.

- Supervised ranking, which includes methods for classification ranking, scores and ranks features by the ability to discriminate between or among data groups, such as between nominal and faulty behavior. Supervised ranking requires condition variables that contain the labels that characterize the data groups.

- Unsupervised ranking does not require data labels. This type of ranking scores and ranks features based on their tendency to cluster with other features.

- Prognostic ranking methods score and rank features based on the ability to track degradation in order to enable prediction of remaining useful life (RUL). Prognostic ranking requires real or simulated run-to-failure or fault-progression data and does not use condition variables.

The following figure illustrates supervised classification ranking results. You can try multiple ranking methods and view the results from each method together. The ranking results allow you to eliminate ineffective features and to evaluate the ranking effects of parameter adjustments when computing derived variables or features.
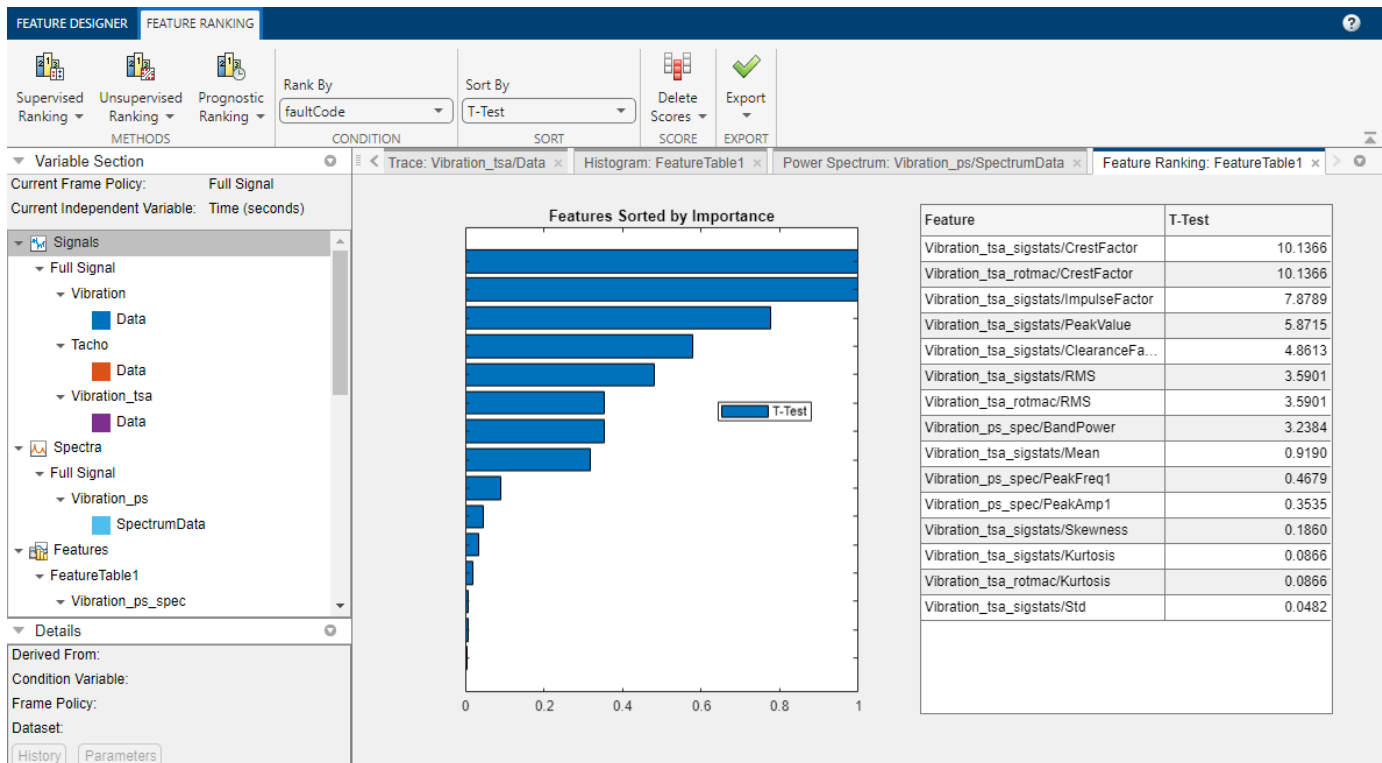
For information on feature ranking, see:

- "Rank and Export Features in Diagnostic Feature Designer"
- **Diagnostic Feature Designer** sections for **Feature Ranking Tab** and **Ranking Technique**

- "Perform Prognostic Feature Ranking for a Degrading System Using Diagnostic Feature Designer" on page 7-65

Export Features to Classification Learner

After you have defined your set of candidate features, you can export them to the **Classification Learner** app in the Statistics and Machine Learning Toolbox. **Classification Learner** trains models to classify data by using automated methods to test different types of models with a feature set. In doing so, **Classification Learner** determines the best model and the most effective features. For predictive maintenance, the goal of using the **Classification Learner** is to select and train a model that discriminates between data from healthy and from faulty systems. You can incorporate this model into an algorithm for fault detection and prediction. For an example of exporting from the app into **Classification Learner**, see "Analyze and Select Features for Pump Diagnostics" on page 7-24.

You can also export your features and datasets to the MATLAB workspace. Doing so allows you to visualize and process your original and derived ensemble data using command-line functions or other apps. At the command line, you can also save features and variables that you select into files, including files referenced in an ensemble datastore.

For information on export, see "Rank and Export Features in Diagnostic Feature Designer".

Generate MATLAB Code for Your Features

In addition to exporting the features themselves, you can generate a MATLAB function that reproduces the computations that created those features. Generating code allows you to automate the feature computations with different data sets. For instance, suppose you have a large input data set with many members, but for faster app response, you want to use a subset of that data when you first explore possible features interactively. After you identify your most effective features using the app, you can generate code and then use that code to apply the same feature computations to the all-member data set. The larger member set lets you provide more samples as training inputs to **Classification Learner**.

The following figure illustrates the **Code Generation** tab, which allows you to perform a detailed query to select features based on criteria such as feature input and computation method.


```
function [featureTable,outputTable] = diagnosticFeatures(inputData)
%DIAGNOSTICFEATURES recreates results in Diagnostic Feature Designer.
%
```

You can also generate code that is formatted for streaming data as well as a Simulink block that uses that code.

For more information, see:

- "Automatic Feature Extraction Using Generated MATLAB Code" on page 7-86

- "Generate a MATLAB Function in Diagnostic Feature Designer" on page 7-94
- "Export Feature Extraction Function and Simulink Model for Streaming Data" on page 7-179
- "Apply Generated MATLAB Function to Expanded Data Set" on page 7-101
- "Anatomy of App-Generated MATLAB Code" on page 7-114

See Also
Diagnostic Feature Designer

See Also

Related Examples
- "Isolate a Shaft Fault Using Diagnostic Feature Designer" on page 7-46

More About
- "Identify Condition Indicators for Predictive Maintenance Algorithm Design"
- "Organize System Data for Diagnostic Feature Designer" on page 7-19
- "Import Data into Diagnostic Feature Designer" on page 7-127
- "Data Preprocessing for Condition Monitoring and Predictive Maintenance" on page 2-2
- "Condition Indicators for Monitoring, Fault Detection, and Prediction" on page 3-2

Prepare Matrix Data for Diagnostic Feature Designer

This example shows how to prepare matrix data for import into **Diagnostic Feature Designer**. You step first through conversion of a single-member matrix and its associated fault code into a `table`. Then, you combine multiple tables into a single ensemble table that you can import as a multimember ensemble.

Diagnostic Feature Designer accepts member data that is contained in individual numeric matrices. However, member tables provide more flexibility and ease of use.

- With a table, you can embed your measurements as signal variables, each containing both an independent variable and one or more data variables. With a matrix, you can specify only one independent variable that applies to all the columns of data. The app can more easily interpret table data.

- With a table, you can specify informative variable names. With a matrix, the app uses column numbers to identify data vectors.

- With a table, you can include scalar information for each member, such as condition variables or features. With a matrix, you cannot combine scalar information with signal information. This limitation means that you cannot group data by label in the app, and cannot assess feature effectiveness in separating, say, `healthy` data from `unhealthy` data.

An ensemble table is even easier to use, because it combines all member tables into one dataset. To import an ensemble table, you import just one item, instead of needing to select multiple items.
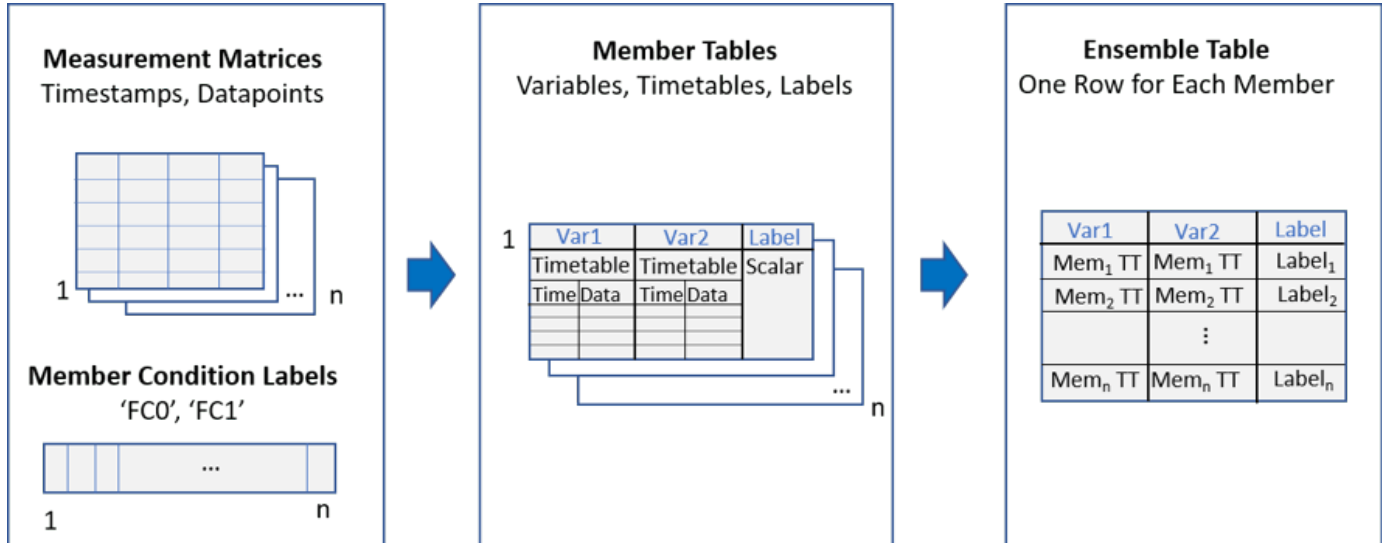
Load Member Matrices and Fault Codes

Load the member data and fault code data. The member data `dmatall` consists of four sets of timestamped vibration and tachometer measurements taken over an interval of 30 seconds. These member matrices are stacked together in a 3-D matrix. An independent fault code vector `fc` indicates whether each member is from a healthy (`0`) or unhealthy (`1`) system.

Initialize `tv_ensemble table`, which ultimately includes both the time-tagged data and the fault code for each member.

```
load tvmatrixmembers dmatall fc
```

Convert a Matrix to a Table

Start by converting a single member matrix to a table that contains timetables for the measurement signals and the member fault code. Extract the first member matrix from `dmatall`.

```
memmat = dmatall(:,:,1);
```

The first two columns of `memmat` contain the measured vibration signal. The third and fourth contain the measured tacho signal. Each signal consists of an independent variable (time) and a data variable (vibration or tacho measurement). Extract these signals into independent matrices `vibmat` and `tachmat` .

```
vibmat = memmat(:,[1 2]);
tachmat = memmat(:,[3 4]);
```

Convert each signal into a `timetable`. First, separate each signal into its time and data components. Use the function `seconds` to convert the timestamps into duration variables for the timetable. Then input the signal components into `array2timetable` to convert the signals into `timetables vibtt` and `tachtt`. Assign the variable name `Data` to the data column. The timetable automatically assigns the name `Time` to the time column.

```
vibtime = seconds(vibmat(:,1));
vibdata = vibmat(:,2);
tachtime = seconds(tachmat(:,1));
tachdata = tachmat(:,2);
vibtt = array2timetable(vibdata,'RowTimes',vibtime,'VariableNames',{'Data'});
tachtt = array2timetable(tachdata,'RowTimes',tachtime,'VariableNames',{'Data'});
```

Extract the fault code `faultcode` from the fault code vector `fc`.

```
faultcode = fc(1);
```

Assemble the member table that contains the two timetables, the fault code scalar, and descriptive variable names.

```
memtable = table({vibtt},{tachtt},faultcode,'VariableNames',{'Vibration','Tacho','FaultCode'});
```

You now have a member table that you can insert into an ensemble table that contains multiple member tables. Initialize the ensemble table and insert the first member.

```
tv_ensemble_table = table();
tv_ensemble_table(1,:) = memtable
```

```
tv_ensemble_table=1×3 table
        Vibration                Tacho            FaultCode
    _____    _____    _____

    {30001x1 timetable}    {30001x1 timetable}        1
```

Convert Multiple Member Matrices into Ensemble Table

You can repeat the same processing steps on all member matrices to create the full ensemble table. You can also automate the processing steps for each matrix. To do so, first initialize an ensemble table. Then loop through the member matrices to convert the members to tables and insert them into the ensemble table.

Initialize `tv_ensemble_table`.

`tv_ensemble_table = table();`

Loop through the conversion and insertion sequence

```
for idx = 1:size(dmatall,3)
    vibmat = dmatall(:,[1 2],idx);
    tachmat = dmatall(:,[3 4],idx);
vibtt = array2timetable(vibmat(:,2),'RowTimes',seconds(vibmat(:,1)),'VariableNames',{'Data'});
tachtt = array2timetable(tachmat(:,2),'RowTimes',seconds(tachmat(:,1)),'VariableNames',{'Data'})
tv_member = table({vibtt},{tachtt},fc(idx),'VariableNames',{'Vibration','Tacho','FaultCode'});
tv_ensemble_table(idx,:) = tv_member;
end
```

You have created a single ensemble table. Each row represents one member, and each member consists of two timetables representing the vibration and tachometer signals, and the scalar fault code.

`tv_ensemble_table`

```
tv_ensemble_table=4×3 table
        Vibration               Tacho           FaultCode
    _____    _____    _____

    {30001x1 timetable}    {30001x1 timetable}       1
    {30001x1 timetable}    {30001x1 timetable}       0
    {30001x1 timetable}    {30001x1 timetable}       1
    {30001x1 timetable}    {30001x1 timetable}       0
```

You can import this ensemble table into **Diagnostic Feature Designer** by clicking **New Session** and selecting `DataTable` in the **Select more variables** pane.

See Also
Diagnostic Feature Designer | table | timetable | array2timetable

More About
- "Organize System Data for Diagnostic Feature Designer" on page 7-19
- "Data Ensembles for Condition Monitoring and Predictive Maintenance" on page 1-2

Interpret Feature Histograms in Diagnostic Feature Designer

A feature is effective when it clearly separates data groups with different condition variable labels. **Diagnostic Feature Designer** provides various feature options, but the most effective features depend on your data and the systems and conditions your data represents.

To perform a preliminary assessment of how effective a feature is, you can evaluate the feature histogram. The histogram plot visualizes the separation between labeled groups. To do so, the histogram bins the data distribution and uses color to identify the label groups within each bin. You can customize the histogram to enhance the visualization and highlight information in features of interest. You can also view numerical information about the separation between group distributions.

Histograms allow you to get an early sense of feature effectiveness. To perform a more rigorous quantitative assessment using specialized statistical methods, use ranking, as described in Rank Features in "Explore Ensemble Data and Compare Features Using Diagnostic Feature Designer" on page 7-2. The feature-ranking computations are independent of the visualization choices you make during histogram analysis.

The following figure shows separation visualization. These examples have a relatively small sample size, which exaggerates differences.
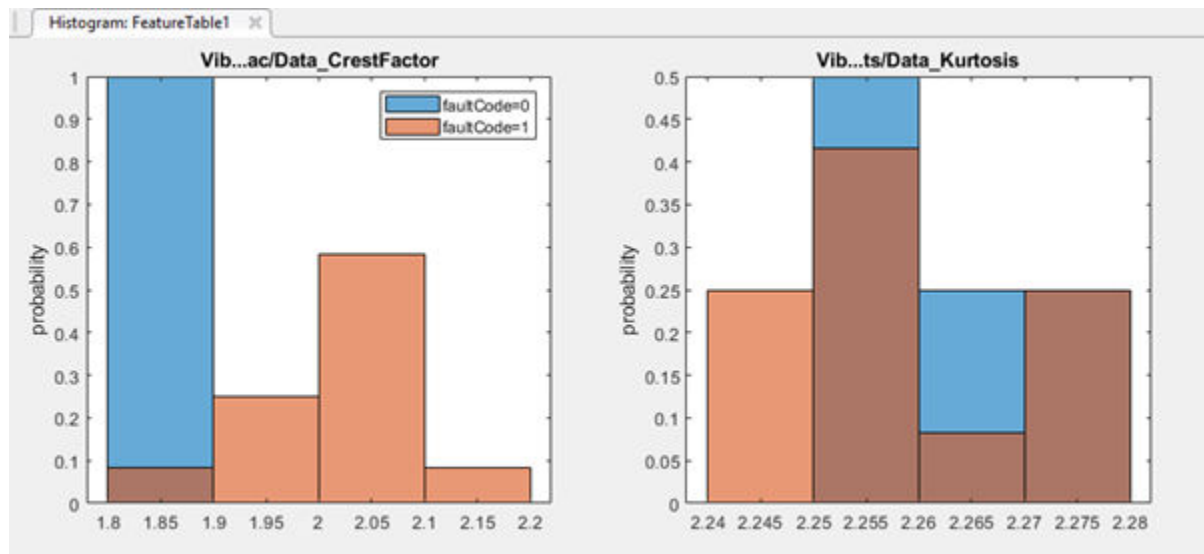
In both plots, the two-state condition code is `faultCode`. A value of 0 (blue) indicates a healthy system and a value of 1 (orange) indicates a faulty system. The histograms represent the crest factor and the kurtosis of the `Vibration` signal.

The crest factor histogram shows that:

* All the healthy system feature values fall within the range of the first bin.
* Most of the faulty system values fall into the remaining three bins.
* The first bin also contains some data from the faulty system, but the amount is small relative to the healthy system data.

For this case, the histogram indicates that the crest factor feature distinguishes between healthy and faulty behavior well, but not completely.

By contrast, the kurtosis histogram shows that:

• Data with values in the range of the first bin is always faulty.
• Data within the range of the other bins come from both healthy and faulty groups. The fault state is ambiguous in these regions.

From these two histograms, you can infer that the crest factor feature is more effective than the kurtosis feature.

The app provides interactive tools for customizing the histogram. For example, you can increase the histogram resolution by changing bin width, changing the condition variable that specifies the groups, or modifying the normalization that the histogram applies. For more information on customizing histograms in the app, see "Generate and Customize Feature Histograms" on page 7-16.

Interpret Feature Histograms for Multiclass Condition Variables

If your condition variable has more than two states, or classes, the resulting histograms might be harder to interpret on their own because of the additional color combinations. For example, suppose that your fault code can represent two independent fault states in addition to the healthy state, `fault1` and `fault2`. The following figure shows histograms similar to the previous histograms but corresponding to such a three-class condition variable.
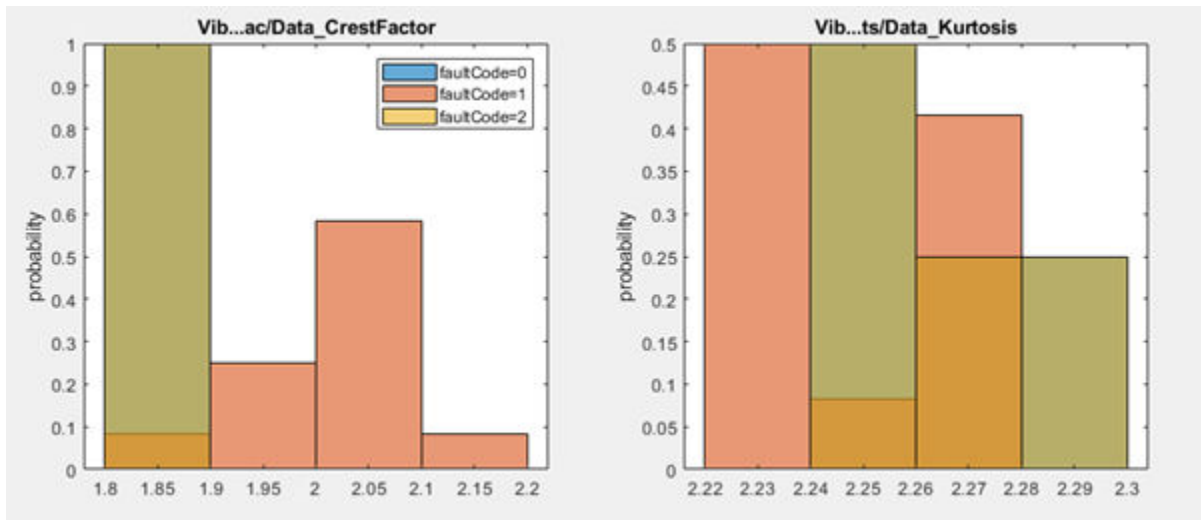
Get additional information on feature effectiveness by viewing numerical group distances. The **Show Group Distance** option provides a value, the KS statistic, for each combination of condition variable classes. Using the two-sample Kolmogorov-Smirnov test, the KS statistic indicates how well separated the cumulative distribution functions of the distributions of the two classes are.

The following table shows the group distances corresponding to the previous histograms.

The KS statistic indicates the separation between every pairing of the `faultCode` values. The statistic value ranges from 0 to 1, where 0 is no separation between the distributions, and 1 is complete separation.

For the crest factor feature as with the two-class `faultCode`, differentiation between healthy `fault0` and faulty `fault1` data is strong, with a KS statistic of 1. Differentiation is also strong between `fault1` and `fault2` data. However, differentiation between `fault0` and `fault2` data is relatively poor.

For the kurtosis feature, differentiation between pairs in all pairings is relatively poor.

For more information on the KS statistic, see `kstest2`.

Generate and Customize Feature Histograms

To generate a set of feature histograms from a feature table:

- Select the feature table in the **Feature Tables** section of the data browser.
- Click the **Histogram** icon in the plot gallery.

To optimize separation visualization, customize the histograms. The **Histogram** tab provides parameters that allow you to modify the histogram to enhance interpretation.
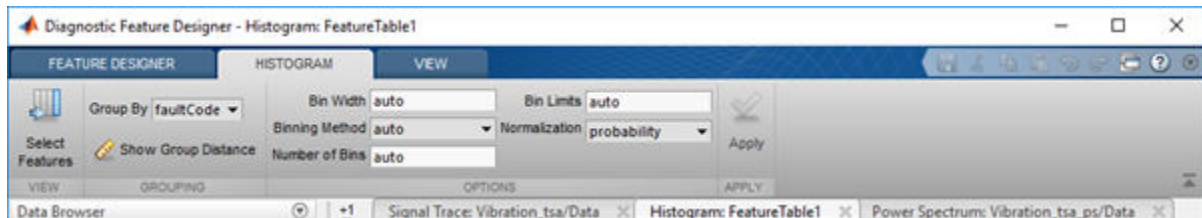
Select Features

By default, the app plots histograms for all your features, and displays them in reverse-alphabetical order. If you want to focus on a smaller set of features, click **Select Features**.

Group Data by a Condition Variable

You can group data in the histogram set for any condition variable you have imported. This condition variable might indicate system health. The variable might also be an operational condition such as temperature or machine mode. To select a condition variable to group by color code, select a variable from **Group By**.

Display the Group Separation Distance

To display the group separation distance, or KS Statistic, that was discussed in "Interpret Feature Histograms for Multiclass Condition Variables" on page 7-14, click **Show Group Distance**. This option brings up a table providing the group separation value for each pairing of condition variable values. In the window, choose which feature you want to examine.

Modify the Bin Settings

By default, the app determines the bin size automatically. Override the automation by typing a different value for bin width or selecting an alternate binning method. The bin settings apply to all the histograms for the feature table.
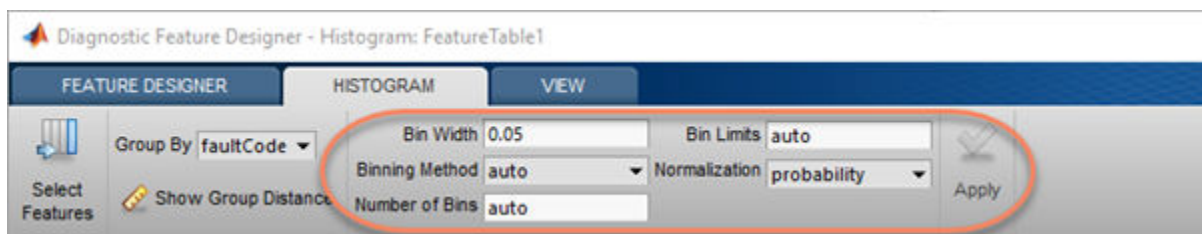
The bin settings for bin width, binning method, and number of bins are not independent. The algorithm uses an order of precedence to determine what to use:

- The **Binning Method** is the default driver for the bin width.
- A **Bin Width** specification overrides the Binning Method.
- The bin width and the independent **Bin Limits** drive the number of bins. A **Number of Bins** specification has an effect only when there is no data grouping.

Modify the Binning Algorithm

By default, **Diagnostic Feature Designer** uses an automatic binning algorithm that returns bins with a uniform bin width. The algorithm chooses the bin settings to cover the data range and reveal the underlying shape of the distribution. To change the binning algorithm, choose from the **Binning Method** menu.

For information on the binning algorithms, see the 'BinMethod' description in histogram.

Increase Resolution by Specifying Bin Width

Increase the resolution of your data by specifying a width that is narrower than what the 'auto' setting provides for the feature you are examining. For example, the following figure repeats the earlier histograms showing separation of data for two fault code values and two features. For the crest factor, the first bin has intermixed healthy and degraded data.
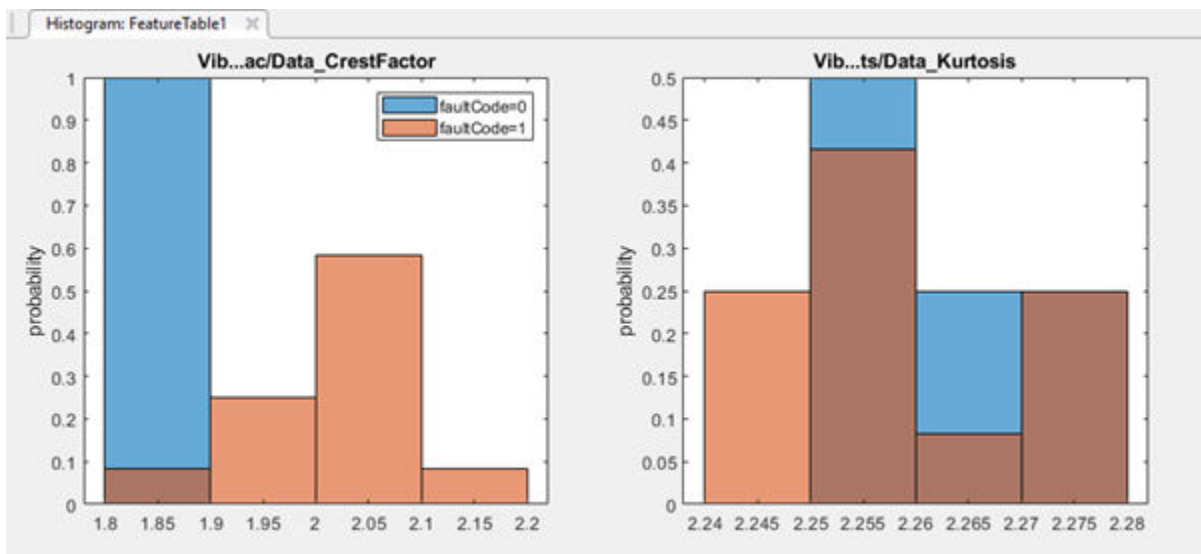
The bin width for the Crest Factor feature is 0.1. If you decrease the bin width to 0.05, the histogram changes as shown here.
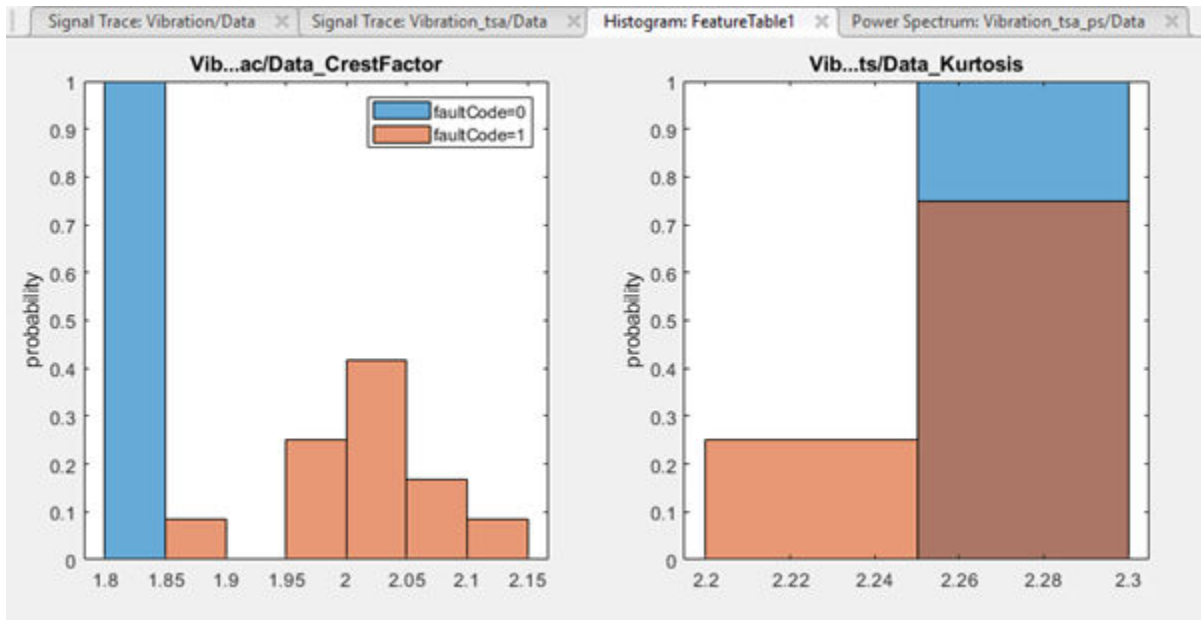
Now the healthy crest factor data is isolated to the first bin, and the remaining bins contain only unhealthy data. However, you have lost resolution on the kurtosis histogram, because a specified bin width applies to all features.

Exclude Outlying Data by Changing Bin Limits

If you are interested in only a portion of the feature distribution, use **Bin Limits** to exclude data outside of the area of interest. Enter the desired limits in the form [lower upper]. This selection does not affect the KS statistic calculation in the group distance table.

Change the Normalization Method

The default histograms use probability for the y axis, with a corresponding range from 0 to 1 for all features. Viewing multiple histograms on the same scale makes it easier to visually compare them. Choose other axis settings from the **Normalization** menu. These methods include raw counts and statistical metrics such as CDF.

See Also
Diagnostic Feature Designer | histogram | kstest2

More About

- "Process Data and Explore Features in Diagnostic Feature Designer"
- "Isolate a Shaft Fault Using Diagnostic Feature Designer" on page 7-46
- "Analyze and Select Features for Pump Diagnostics" on page 7-24

Organize System Data for Diagnostic Feature Designer

The **Diagnostic Feature Designer** app allows you to interactively analyze data and develop features that can distinguish between data from healthy systems and degraded systems. The app operates on a collection of measurement data and information from set of similar systems such as machines. To use the app, you must first organize your data into a form that the app can import. One way to organize your data is with numerical matrices, which can capture all your measurement data. However, you can also use more flexible formats such as tables, which allow you to incorporate additional information such as health condition and operating conditions. With this information, you can explore features within the app and assess feature ability to distinguish between different specific conditions.

Data Ensembles

Data analysis is the heart of any condition monitoring and predictive maintenance activity.

The data can come from measurements on systems using sensors such as accelerometers, pressure gauges, thermometers, altimeters, voltmeters, and tachometers. 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 a number of separate engines all manufactured to the same specifications). And you might have data representing both healthy operation and fault conditions. Evaluating effective features for predictive maintenance requires organizing and analyzing this data while keeping track of the systems and conditions the data represents.

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.

Ensemble Variables

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

- Data variables (DV) — 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 (IV) — 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 (CV) — 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 gear-box data, `sensor health` might be a condition variable whose state is known for each engine. Condition variables can also be derived values, such as a single scalar value that encodes multiple fault and operating conditions.

Data variables and independent variables typically have many elements. Condition variables are often scalars. In the app, condition variables must be scalars.

Representing Ensemble Data for the App

You can use one of three general approaches to combine your ensemble data and import it into the app. All these approaches require that your ensemble members all contain the same variables.
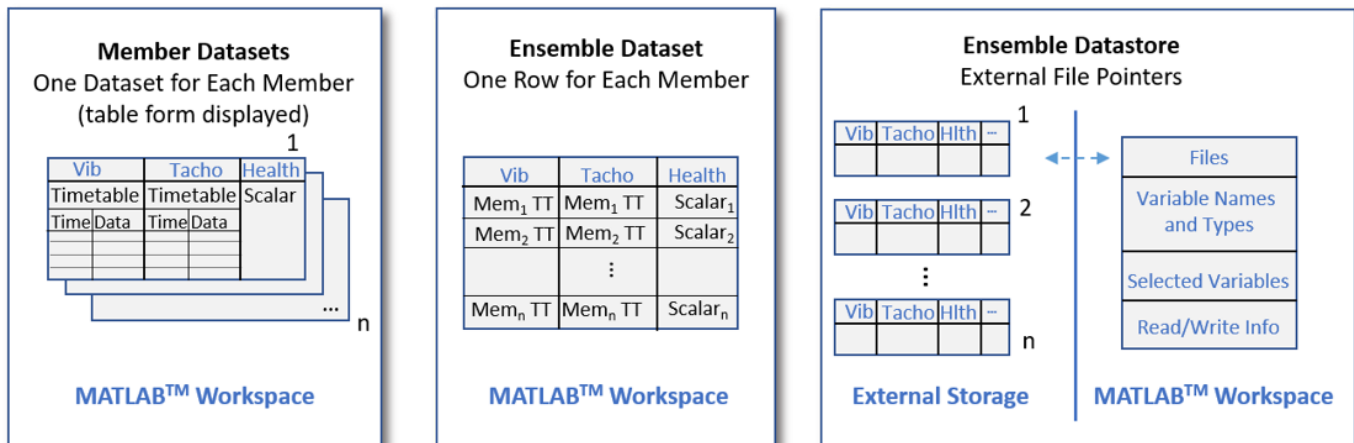
Create Individual Member Datasets

Import your data in the form of individual datasets — one for each member — and let the app combine these datasets into an ensemble. When you select one of the member datasets in during the import process, the app displays all the datasets in the workspace that have the same variables, and are therefore compatible.

This approach requires the least setup before importing the data. If you want to update the ensemble with new members, you must import all members again.

Create an Ensemble Dataset

Import a single ensemble dataset that you create from your member datasets. Each row of your ensemble dataset represents one of your members.

This approach requires more setup before importing the data. It can be more practical than the individual approach when you have larger member sets. if you want to update the ensemble with new members, you can do so outside of the app by adding to your existing table. Then import the updated table.

For an example on creating an ensemble dataset from individual member matrices, see "Prepare Matrix Data for Diagnostic Feature Designer" on page 7-10

Create an Ensemble Datastore Object

Import an ensemble datastore object that contains only the names and paths of member files rather than importing the data itself. This object also includes the information needed for the app to interact with the external files.

This approach is best when you have large amounts of data and variables. 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, or Hadoop Distributed File System (HDFS).

Typically, when you begin exploring your data in the app, you want to import a relatively small number of members and variables. However, later, you might want to test your conclusions on feature effectiveness by bringing in a larger sample size. The ensemble datastore is one method for handling the larger amount of data, especially if the data size exceeds memory limitations for MATLAB.

For more information on ensemble datastore objects, see "Data Ensembles for Condition Monitoring and Predictive Maintenance" on page 1-2.

Data Types and Constraints for Dataset Import

The app accepts various data types, including numerical matrices and tables that contain condition-variable scalars and embedded measurement timetables.

Before importing your data, it must already be clean, with preprocessing such as outlier and missing-value removal. For more information, see "Data Preprocessing for Condition Monitoring and Predictive Maintenance" on page 2-2.

For detailed information on data types and constraints, and on the actual data import, see "Import Data into Diagnostic Feature Designer" on page 7-127.

See Also
table | timetable | fileEnsembleDatastore | simulationEnsembleDatastore

More About
- "Prepare Matrix Data for Diagnostic Feature Designer" on page 7-10
- "Import Data into Diagnostic Feature Designer" on page 7-127
- "Data Ensembles for Condition Monitoring and Predictive Maintenance" on page 1-2

- "File Ensemble Datastore with Measured Data" on page 1-17
- "Generate and Use Simulated Data Ensemble" on page 1-10

Analyze and Select Features for Pump Diagnostics

This example shows how to use the **Diagnostic Feature Designer** app to analyze and select features to diagnose faults in a triplex reciprocating pump.

The example uses simulated pump fault data generated by the "Multi-Class Fault Detection Using Simulated Data" on page 1-43 example. The data has been preprocessed to remove the pump startup transients.

Open Diagnostic Feature Designer

Load the triplex pump fault data. The pump data contains 240 flow and pressure measurements for different fault conditions. There are three fault types (leaking pump cylinder, blocked pump inlet, increased pump bearing friction). The measurements cover conditions where none, one, or multiple faults are present. The data is collected in a table where each row is a different measurement.

```
load('savedPumpData')
pumpData
```

```
pumpData=240×3 table
        flow                 pressure           faultCode

    _____     _____    _____

    {1201x1 timetable}    {1201x1 timetable}        0
    {1201x1 timetable}    {1201x1 timetable}        0
    {1201x1 timetable}    {1201x1 timetable}        100
    {1201x1 timetable}    {1201x1 timetable}        100
    {1201x1 timetable}    {1201x1 timetable}        100
    {1201x1 timetable}    {1201x1 timetable}        100
    {1201x1 timetable}    {1201x1 timetable}        100
    {1201x1 timetable}    {1201x1 timetable}        100
    {1201x1 timetable}    {1201x1 timetable}        100
    {1201x1 timetable}    {1201x1 timetable}        100
    {1201x1 timetable}    {1201x1 timetable}        100
    {1201x1 timetable}    {1201x1 timetable}        100
    {1201x1 timetable}    {1201x1 timetable}        0
    {1201x1 timetable}    {1201x1 timetable}        100
    {1201x1 timetable}    {1201x1 timetable}        100
    {1201x1 timetable}    {1201x1 timetable}        100
       ⋮
```

Open **Diagnostic Feature Designer** by using the `diagnosticFeatureDesigner` command. Initiate a new session by clicking **New Session**, which opens a dialog box for importing data.
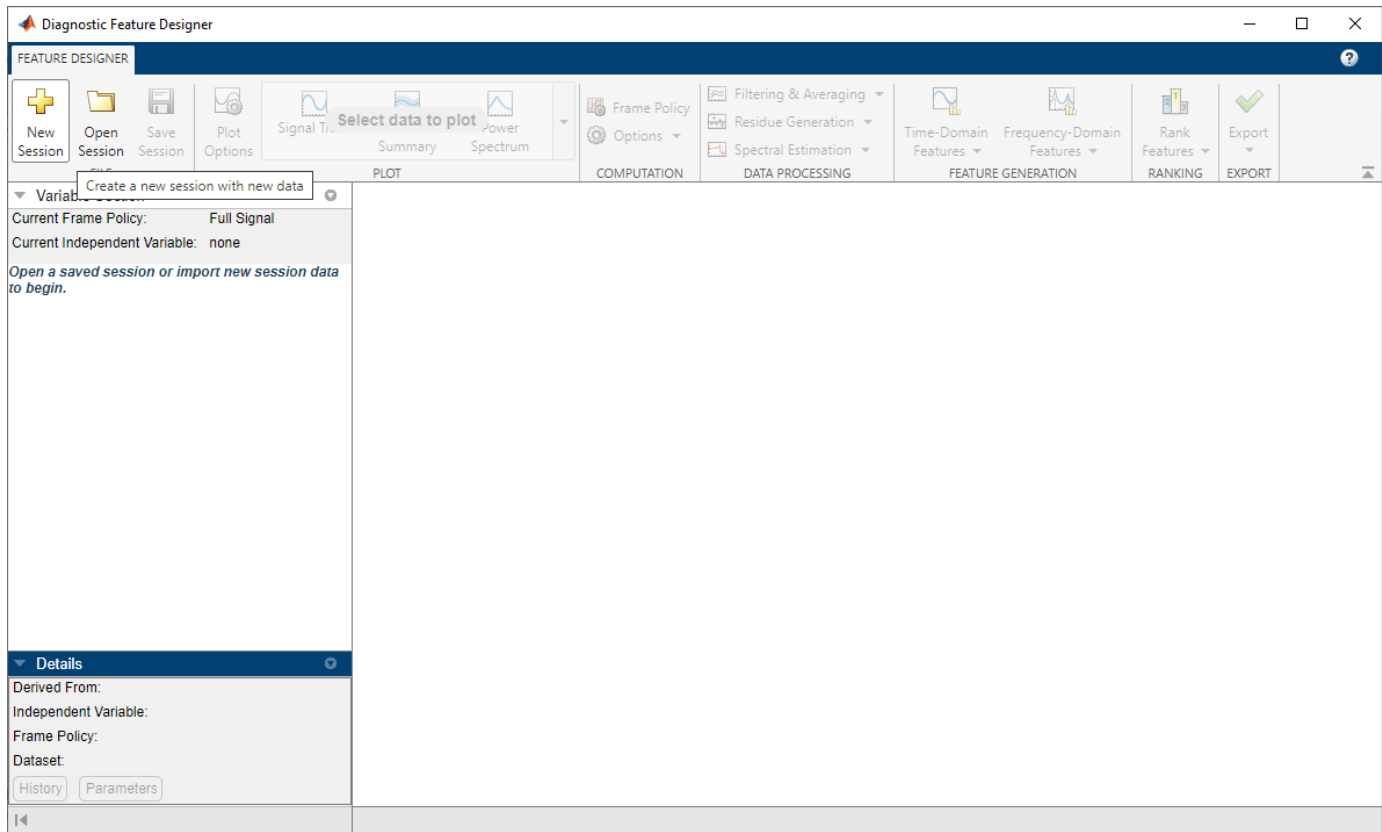
In the **Select dataset from workspace** pane, select `pumpData` as your data source. In the Select source variables pane, confirm that the variable names match those that you viewed at the command line. `flow` and `pressure` are both signals. `faultCode` is a *condition variable*. Condition variables denote the presence or absence of a fault and are used by the app for grouping and classification. When you first open the **New Session** dialog box, the app displays the variable properties for the first variable.
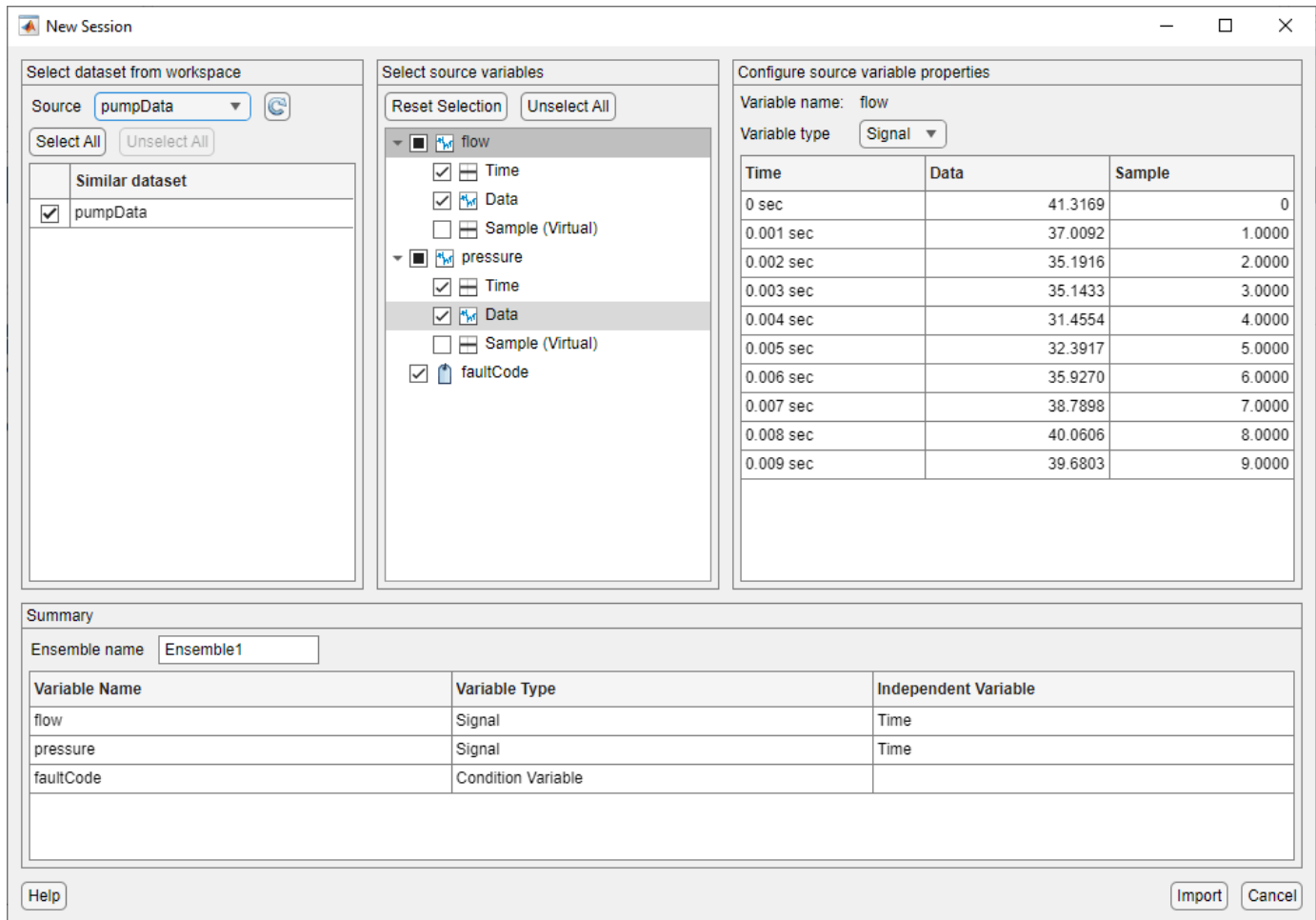
Click **Import** to import the pump data into the app.

Plot Data and Group by Fault Code

Plot the flow signal by selecting `flow` from the **Variables** section of the data browser and clicking **Signal Trace** in the plot gallery. Plot the `pressure` signal the same way.
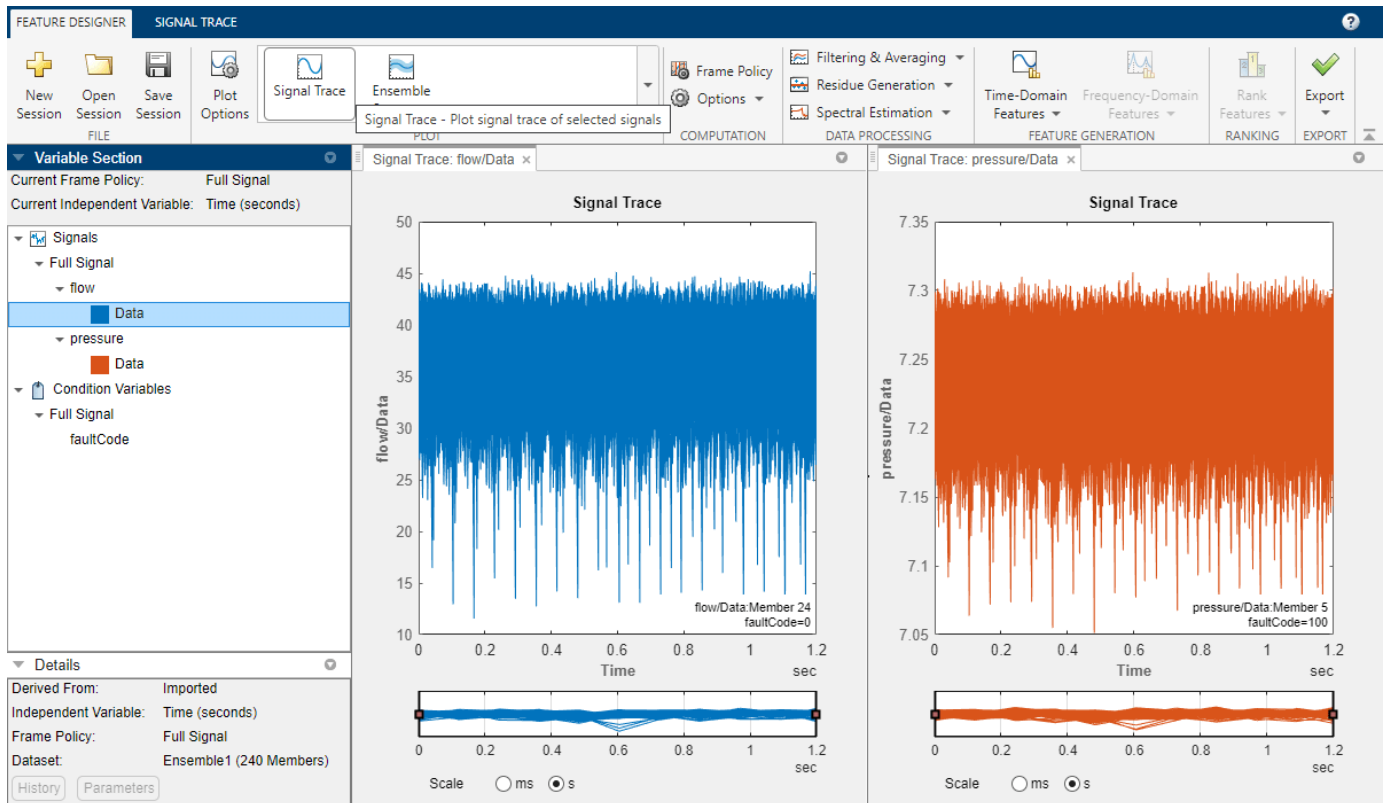
These plots show the pressure and flow signals for all 240 members in the dataset. You can click the **Signal Trace** tab and select **Group by `faultCode`** to display signals with the same fault code in the same color. Grouping signals in this way can help you to quickly determine if there are any clear differences between signals of different fault types. In this case, the measured signals do not show any clear differences for different fault codes.
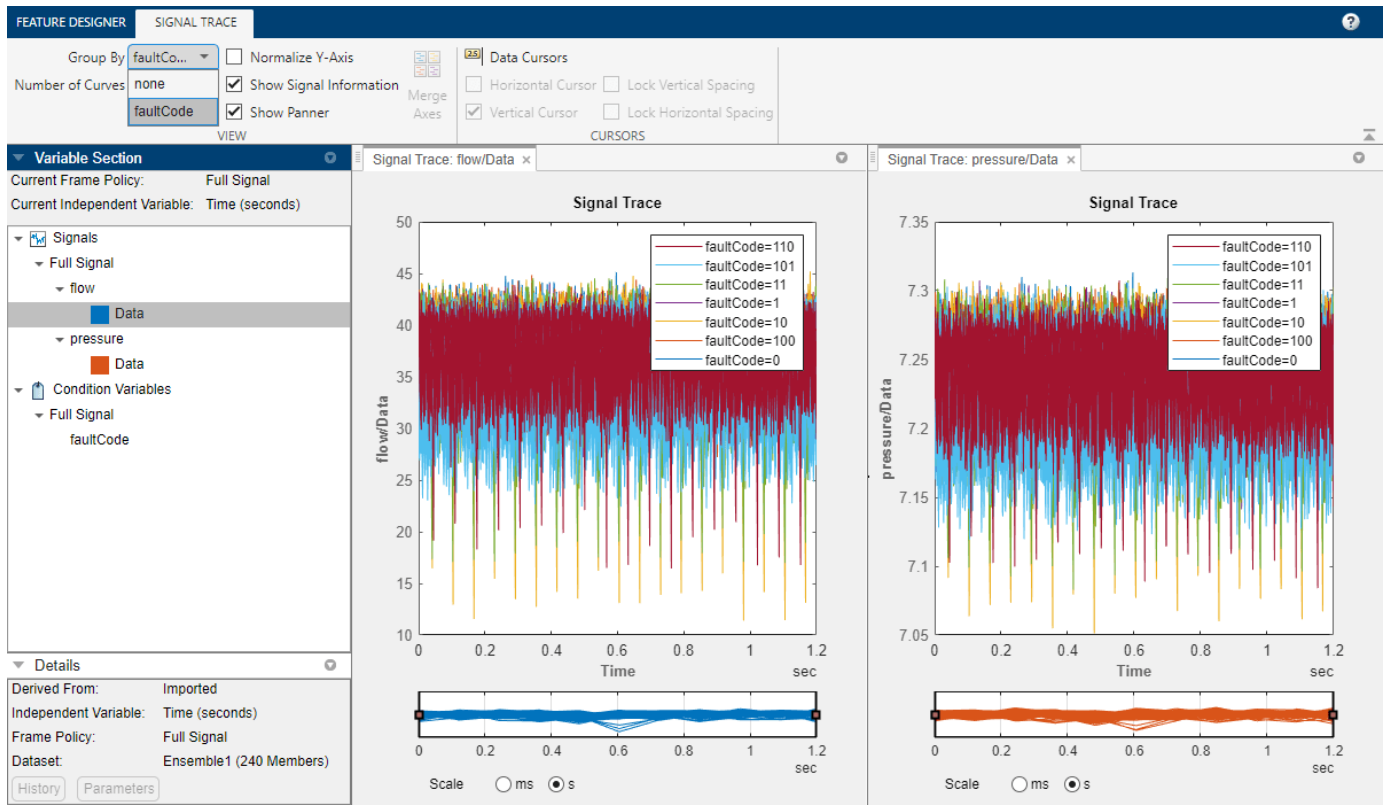
To group all future plots by `faultCode`, use **Plot Options**. Clicking **Plot Options** opens a dialog box that lets you set preferences for the session.
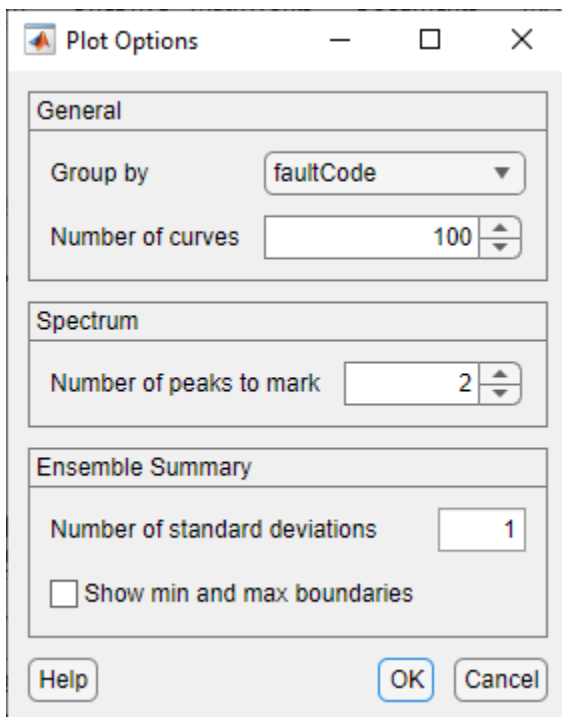
Extract Time Domain Features

As the measured signals do not show any differences for different fault conditions, the next step is to extract time-domain features such as signal mean and standard deviation from the signal. First, select `flow/Data` in the data browser. Then, select **Time-Domain Features** and then **Signal Features**.
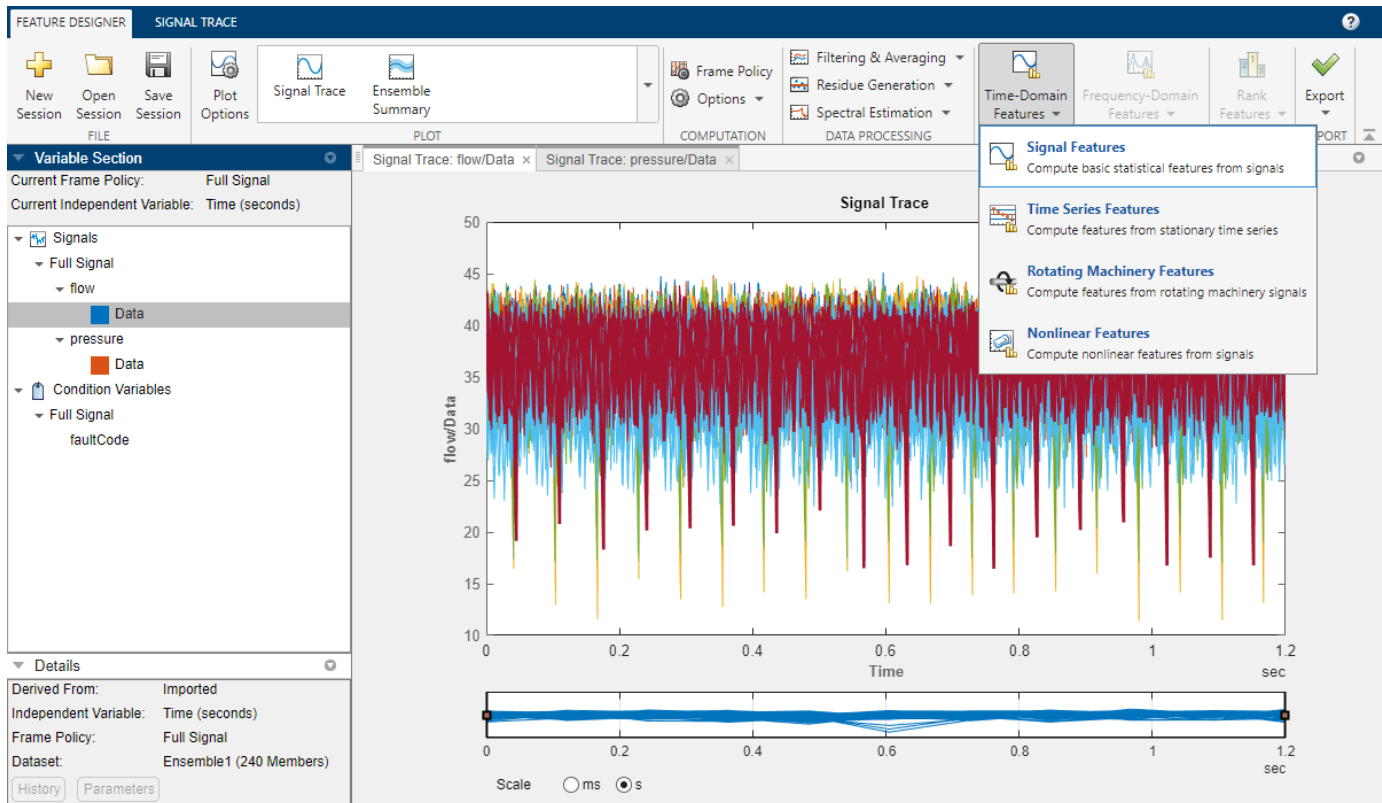
Two new tabs open, **Signal Features** and **Time-Domain Features**. In **Signal Features**, select the features you would like to extract and click **Apply**. For now, clear the **Plot results** check box. You will plot results later to see if the features help distinguish different fault conditions. Repeat this process for the pressure signal.
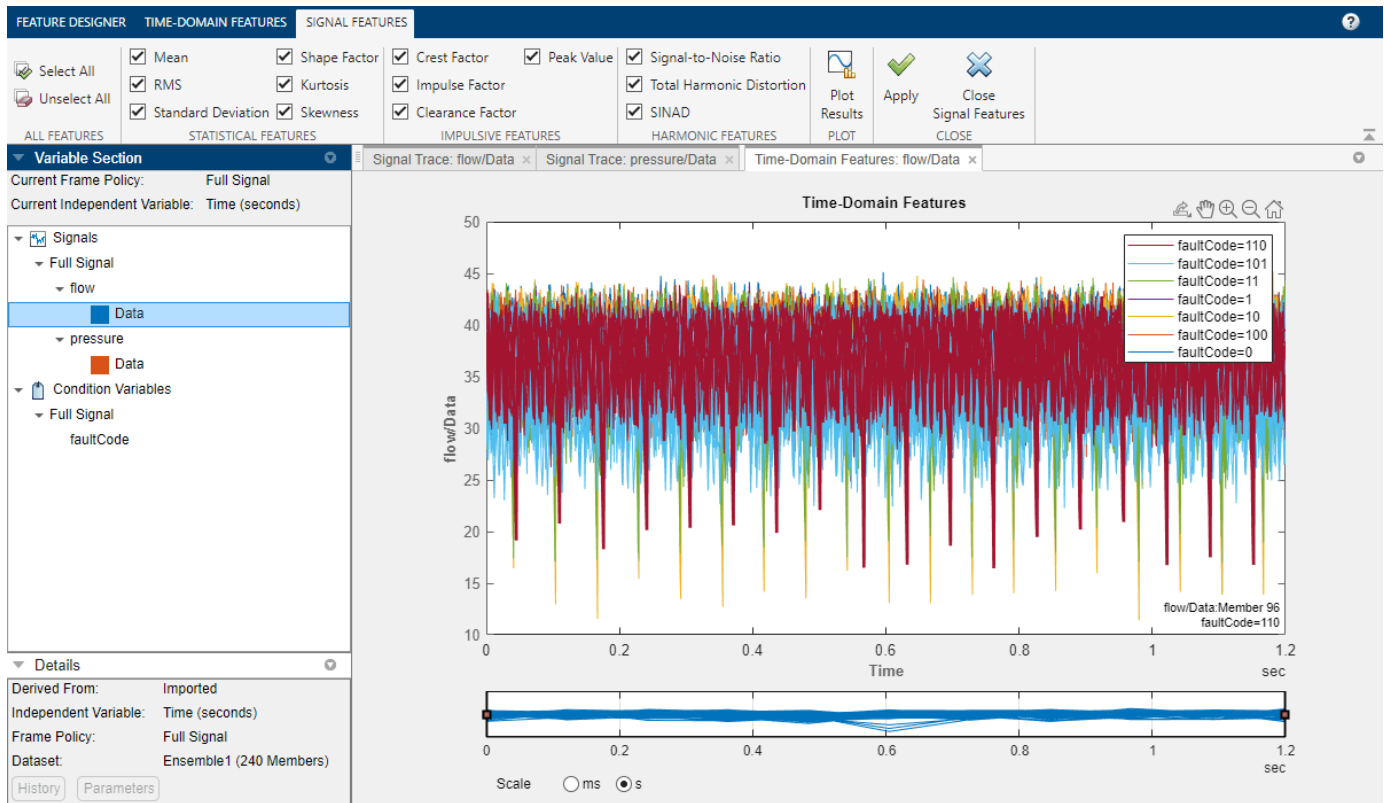
Extract Frequency Domain Features

A reciprocating pump uses a drive shaft and cylinders to pump fluid. Because of the mechanical construction of the pump, there are likely to be cyclic fluctuations in the pump flow and pressure. For example, zoom into a section of the flow signals using the signal panner below the signal trace plot.
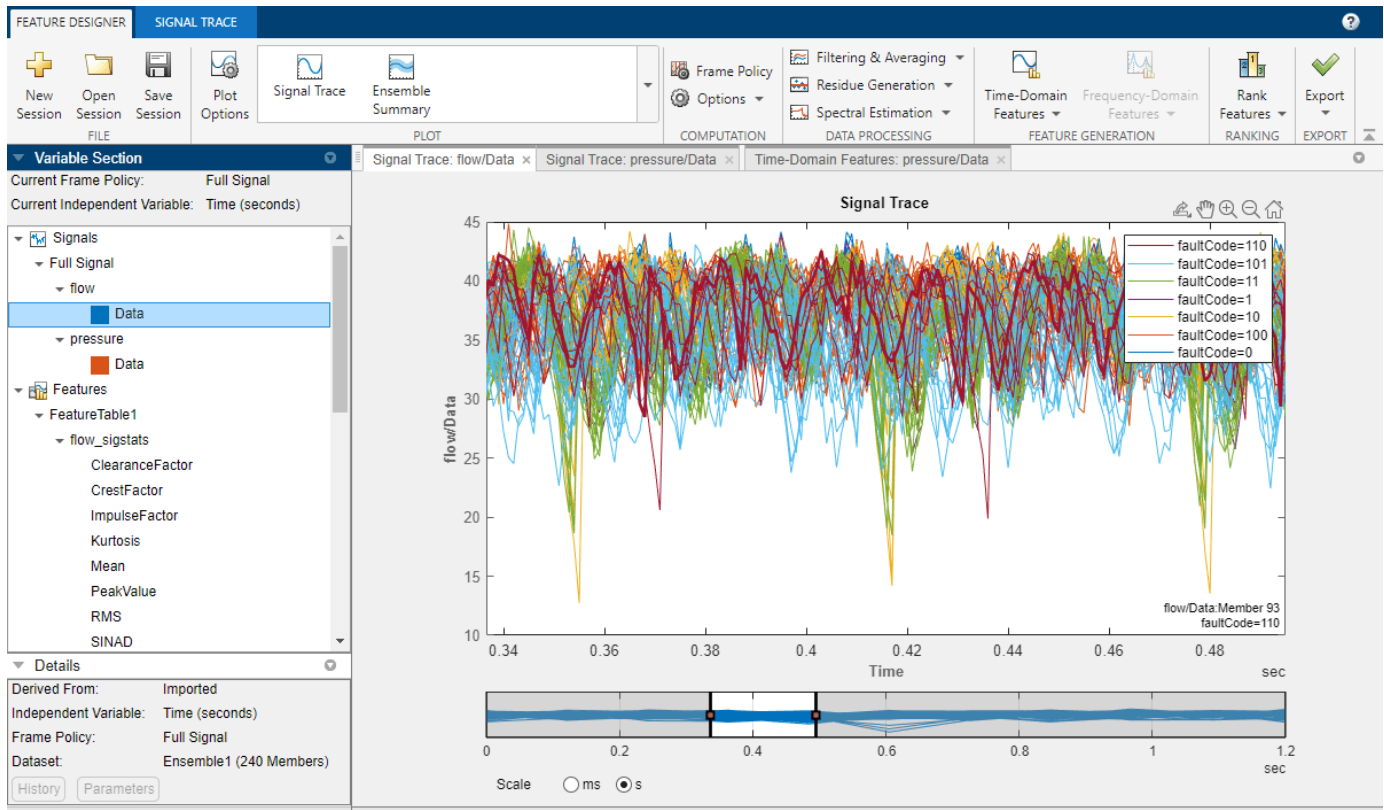
Computing the frequency spectrum of the flow will highlight the cyclic nature of the flow signal and could give better insight into how the flow signal changes under different fault conditions. Estimate the frequency spectra using an autoregressive model.
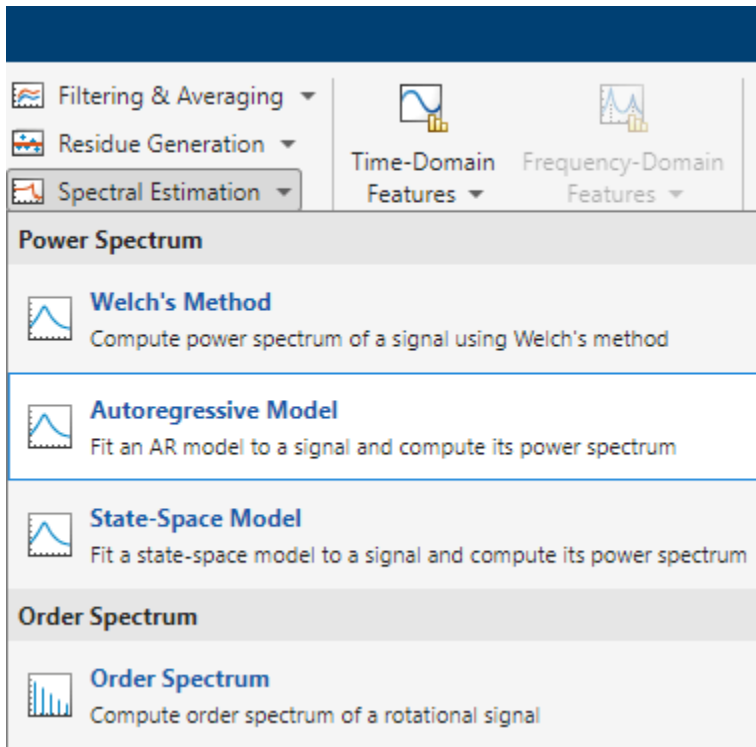
This method fits an autoregressive model of the prescribed order to the data, and then computes the spectrum of that estimated model. This approach reduces any overfitting to the raw data signal. In this case specify a model order of 20. Also set the frequency grid to have a minimum of 0 and a maximum of 500.
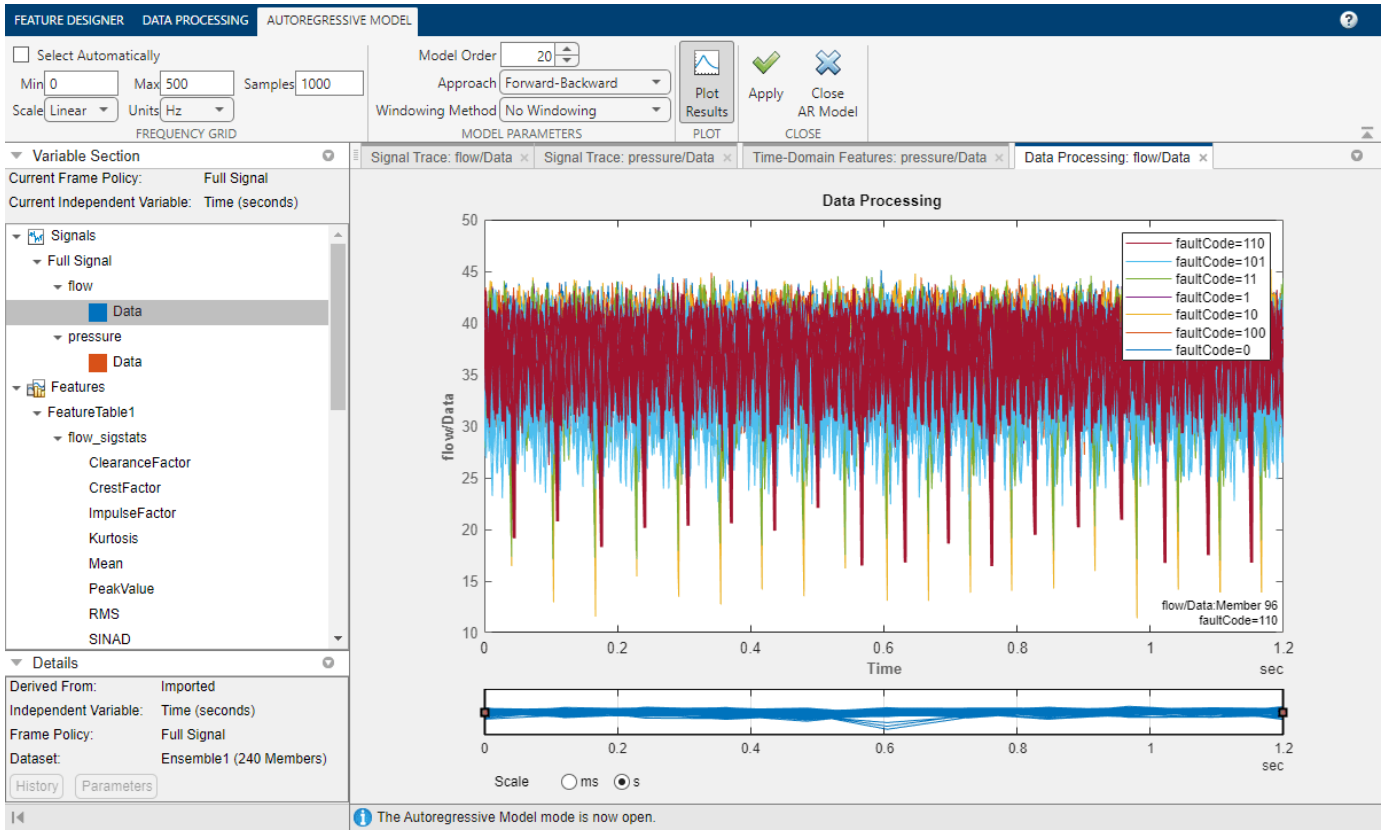
Plotting the computed spectra on a linear scale clearly shows resonant peaks. Grouping by fault code highlights how the spectra change for different fault conditions.
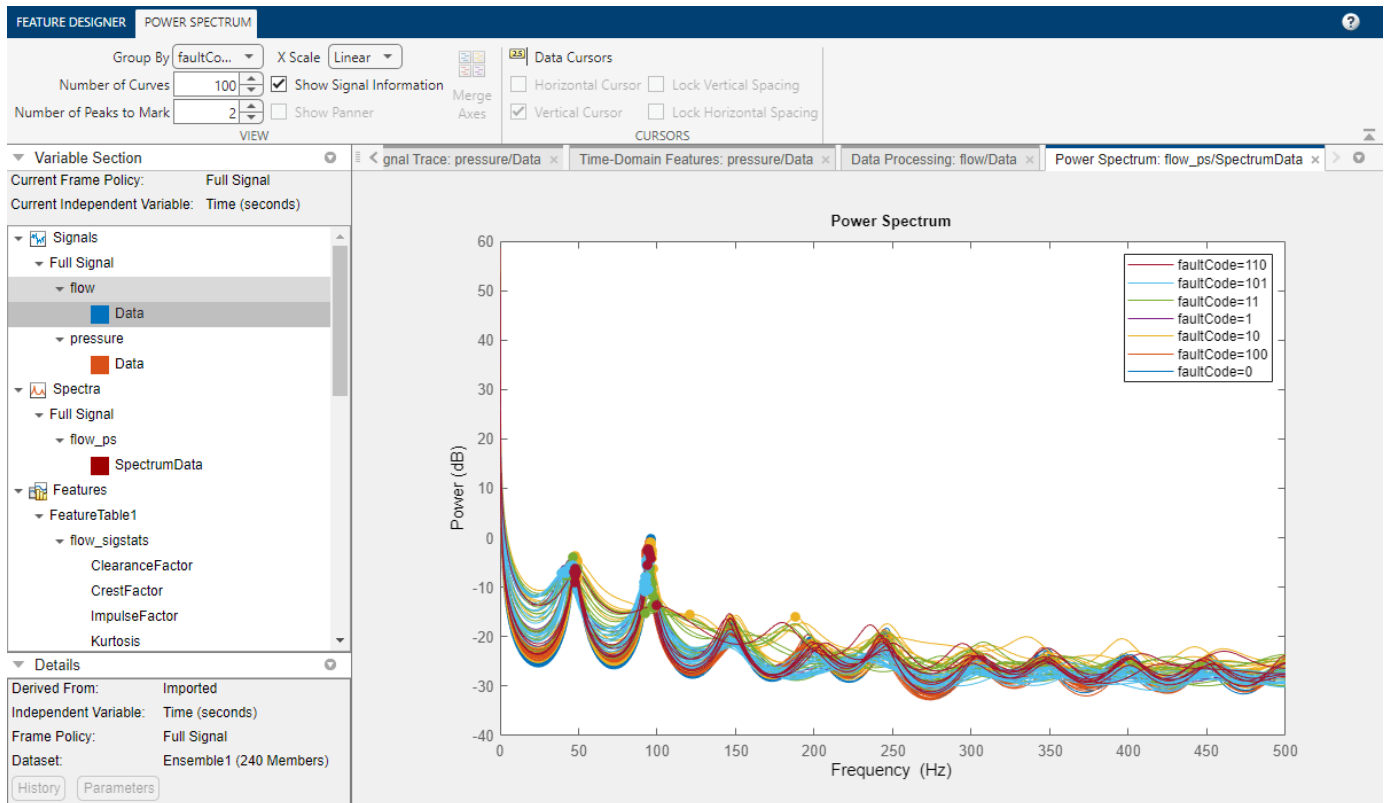
Perform the same computations for the pressure signal as the results will provide additional features to help distinguish different fault conditions.

You can now compute spectral features such as peaks, modal coefficients, and band power.
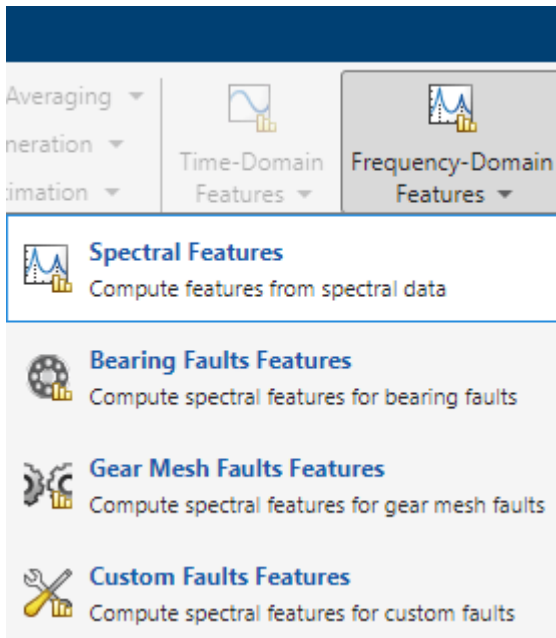
Extract these features in a smaller band of frequencies between 23-250 Hz as the peaks after 250 Hz are smaller. For each signal, extract five spectral peaks. For now, clear the **Plot results** check box. You will plot results later to see if the features help distinguish different fault conditions. Repeat this process for the pressure signal by changing the signal selected at the top of the dialog box.
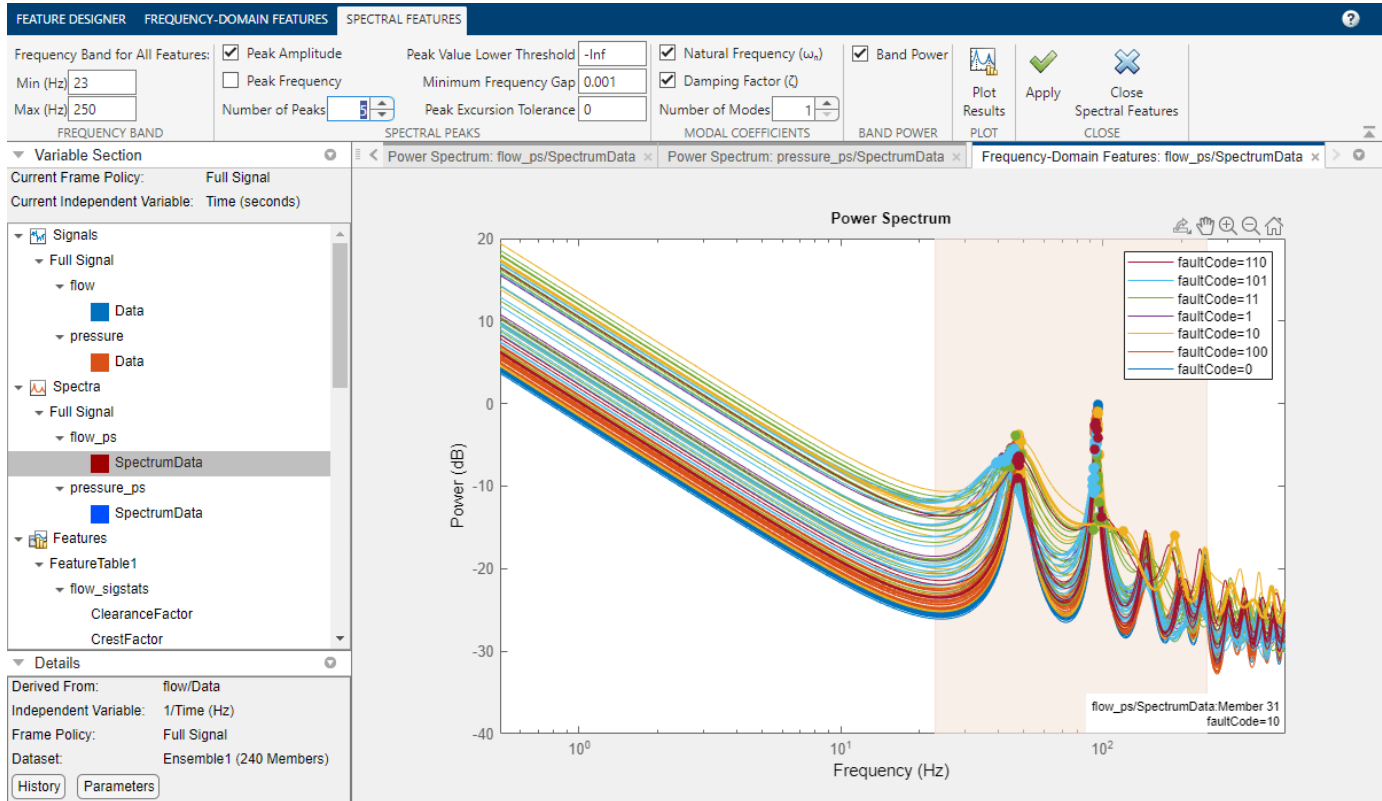
View Features

All the features we have extracted have been collected in a table shown in the **Feature Tables** browser. To view the computed feature data, select `FeatureTable1` from the data browser and click **Feature Table View** in the plot gallery. The fault code is also displayed in the feature table view as the rightmost column in the table. As more features are computed, more columns get appended to the table.

You can see the distributions of the feature values for different condition variable values, in this case, fault types, by viewing the feature table as a histogram. Select `FeatureTable1` and then, click **Histogram** in the plot gallery to create a set of histogram plots. Use the next and previous buttons to show histograms for different features. Histogram plots grouped by fault code can help to determine if certain features are strong differentiators between fault types. If they are strong differentiators, their distributions will be more distant from each other. For the triplex pump data, the feature distributions tend to overlap and there are no features that can clearly be used to identify faults. The next section looks at using automated ranking to find which features are more useful for fault prediction.

Rank and Export Features

From the **Feature Designer** tab, click **Rank Features** and select `FeatureTable1`. The app gathers all the feature data and ranks the features based on a metric such as ANOVA. The app lists the features in order of importance based on the metric value. In this case, the RMS value for the flow signal and the RMS and mean values for the pressure signal are the features that most strongly distinguish different fault types from each other.

After you have ranked your features in terms of importance, the next step is to export them so that you can train a classification model based on these features. Click **Export,** select **Export features to Classification Learner,** and select the features you want to use for classification. In this case, export the top 15 features. The app then sends these features to **Classification Learner** where they can be used to design a classifier to identify different faults.
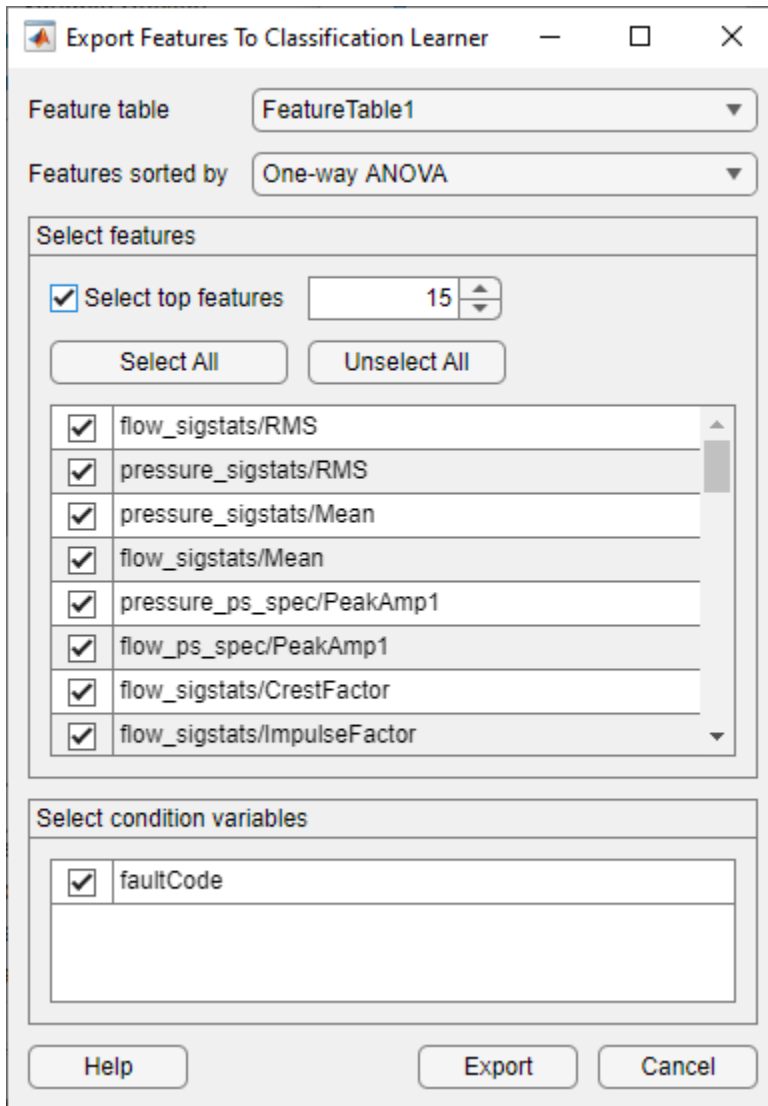
In the **New Session from File** dialog box that **Classification Learner** opens, confirm 5-fold cross-validation and start the session.

New Session from File — □ ✕

Data set

Data Set Variable

| FeatureTable1 | 240x16 table | ▼ |

Response

| faultCode | categorical 8 unique | ▼ |

Predictors

	Name	Type	Range	
☐	faultCode	categorical	8 unique	▲
☑	flow_sigstats/ClearanceFactor	double	1.1397 .. 1.27213	
☑	flow_sigstats/CrestFactor	double	1.1342 .. 1.25801	
☑	flow_sigstats/ImpulseFactor	double	1.13784 .. 1.26731	
☑	flow_sigstats/Mean	double	34.1849 .. 38.4592	
☑	flow_sigstats/RMS	double	34.4394 .. 38.583	▼

[Add All]   [Remove All]

How to prepare data

Validation

Validation Scheme

| Cross-Validation | ▼ |

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

Cross-validation folds: [5] ▲▼

Read about validation

Test

☐ Set aside a test data set

Test Data Percent   [0] ▲▼

Use test set to evaluate model performance. You can import a stand alone test set from the toolstrip after starting a session

Read about test data

[Start Session]   [Cancel]

Classification Learner displays a scatter plot for a single model.
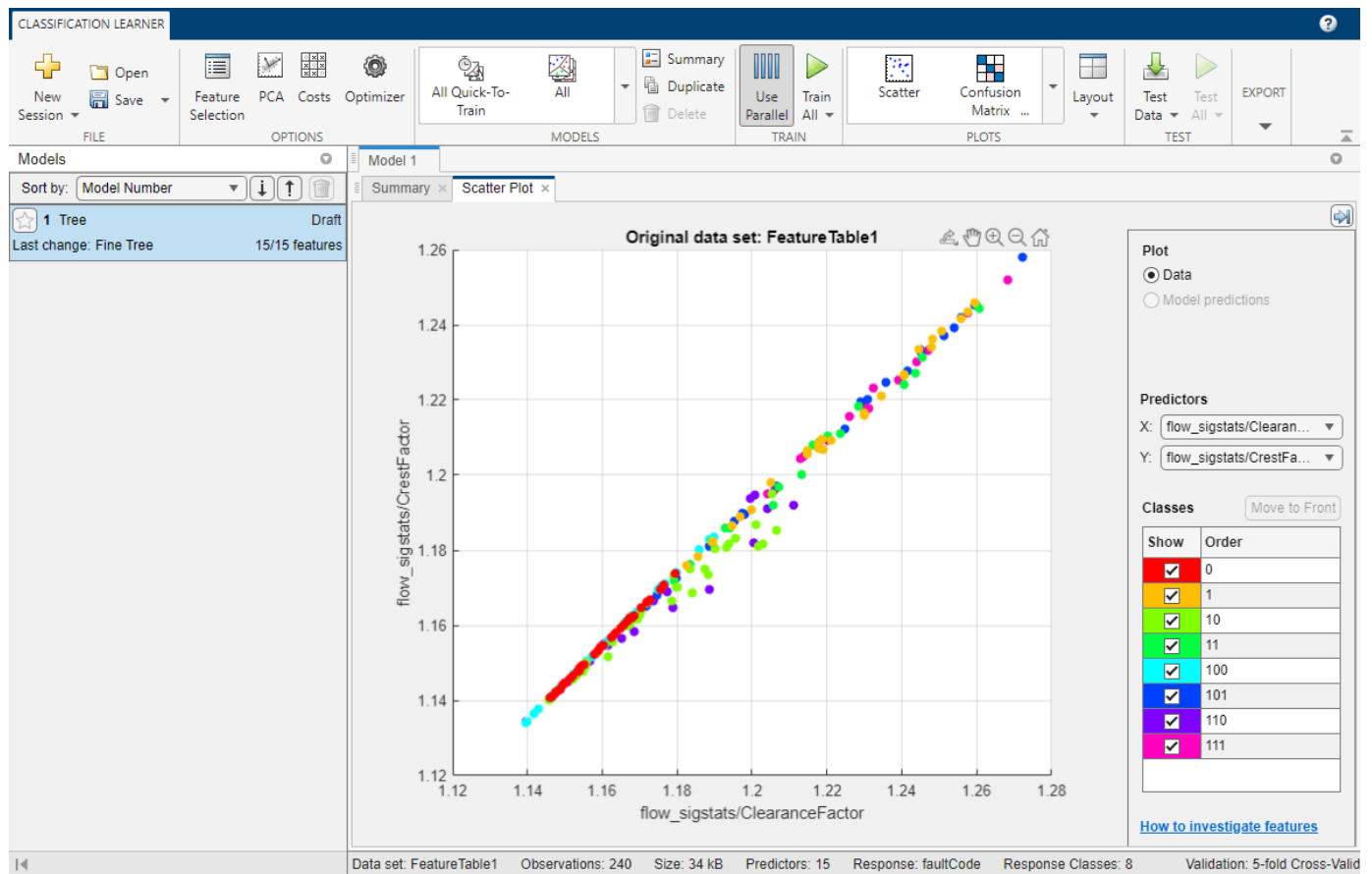
In the **Models** section of the **Classification Learner** tab, select all model types for training.

In the **Models** list, select **Multiple**. Then, click **Train All**.

When the training is complete, **Classification Learner** lists each model in order of model number, along with the model validation accuracy, and displays a confusion matrix for the first model in the set. Change the **Sort by** order to `Accuracy (Validation)`.
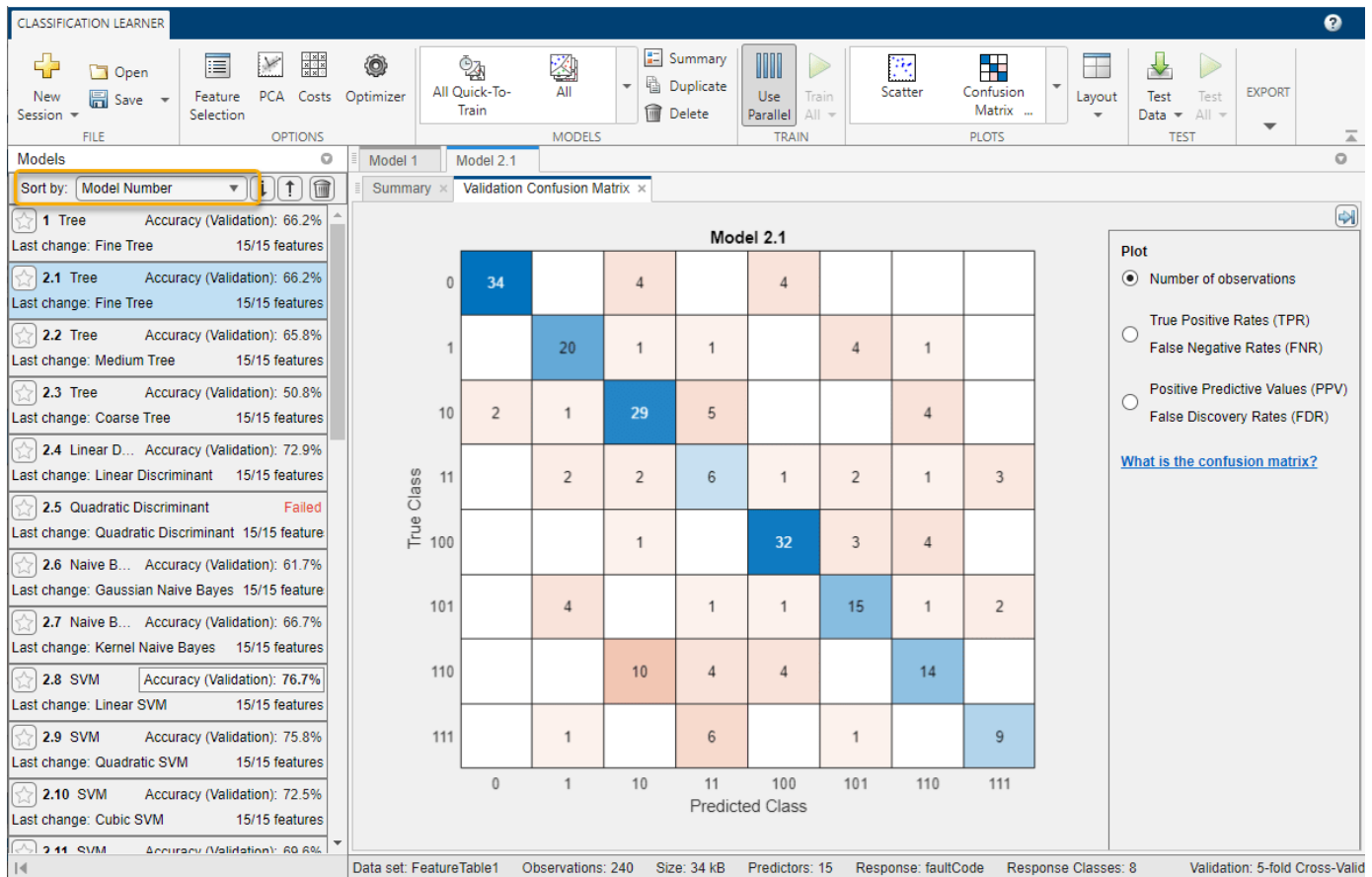
The SVM method has the highest classification accuracy of around 77%. There is some randomness in the process, so your results may be different. Select this model and click **Confusion Matrix**. The confusion matrix illustrates how well this method classifies models for each fault type. The entries on the diagonal represent the number of fault types that are correctly classified. Off-diagonal entries represent fault types for which the predicted and true classes are not the same. To improve accuracy, you can try increasing the number of features. Alternatively, you can iterate on the existing features. Another step would be to iterate on the existing features— especially the spectral features — and perhaps to modify the spectral computation method, change the bandwidth, or use different frequency peaks to improve the classification accuracy.
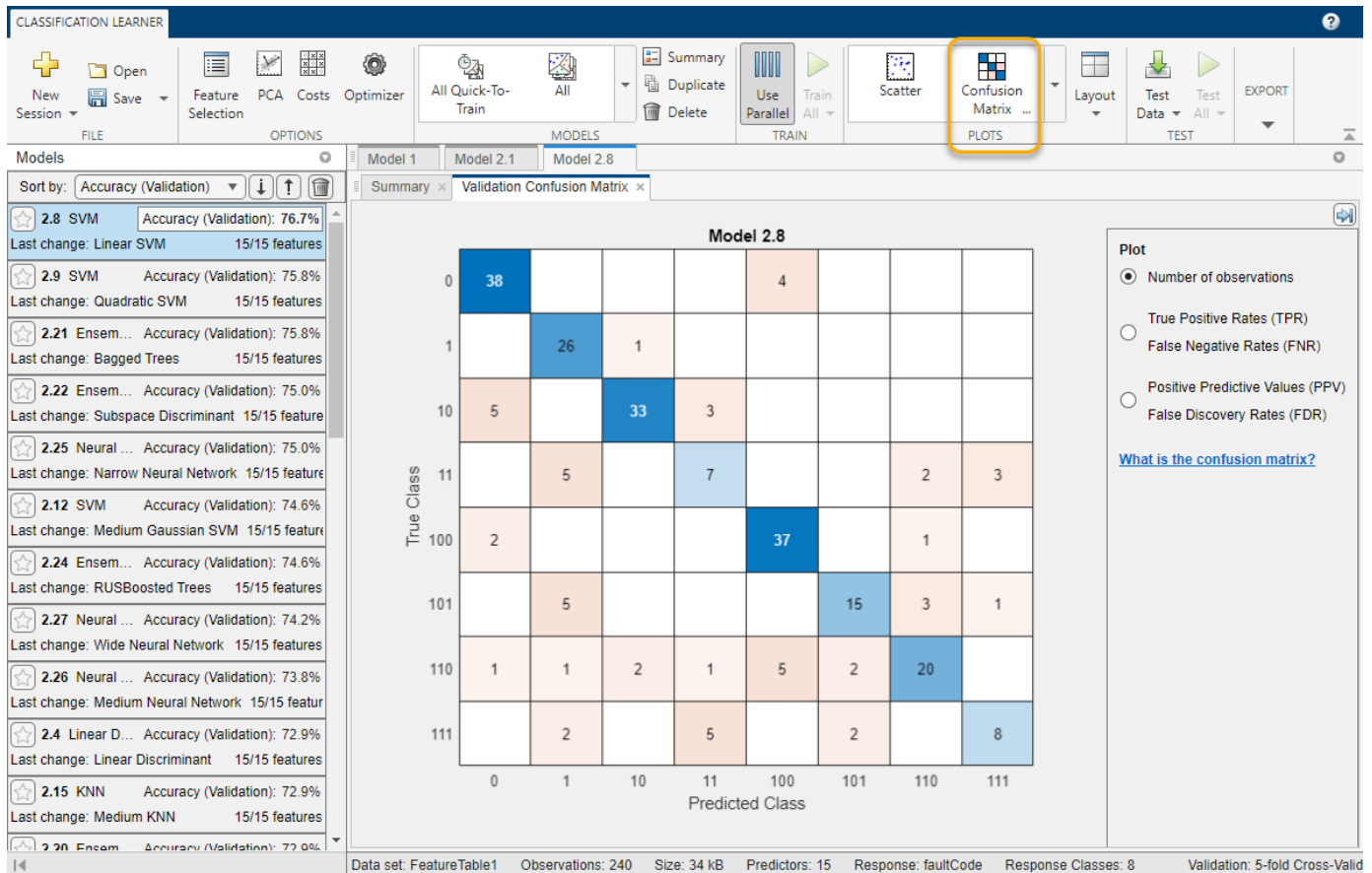
Diagnose Triplex Pump Faults

This example showed how to use **Diagnostic Feature Designer** to analyze and select features and create a classifier to diagnose faults in a triplex reciprocating pump.

See Also
Diagnostic Feature Designer | **Classification Learner**

More About
- "Multi-Class Fault Detection Using Simulated Data" on page 1-43
- "Organize System Data for Diagnostic Feature Designer" on page 7-19
- "Data Ensembles for Condition Monitoring and Predictive Maintenance" on page 1-2

Isolate a Shaft Fault Using Diagnostic Feature Designer

This example shows how to isolate a shaft fault from simulated measurement data for machines with varying rotation speeds and develop features that can help detect the fault.

The example assumes that you are already familiar with basic operations with the app. For a tutorial on using the app, see "Identify Condition Indicators for Predictive Maintenance Algorithm Design".

Model Description

The following figure illustrates a drivetrain with six gears. The motor for the drivetrain is fitted with a vibration sensor and a tachometer. In this drivetrain:

- Gear 1 on the motor shaft meshes with gear 2 with a gear ratio of 17:1.
- The final gear ratio, or the ratio between gears 1 and 2 and gears 3 and 4, is 51:1.
- Gear 5, also on the motor shaft, meshes with gear 6 with a gear ratio of 10:1.
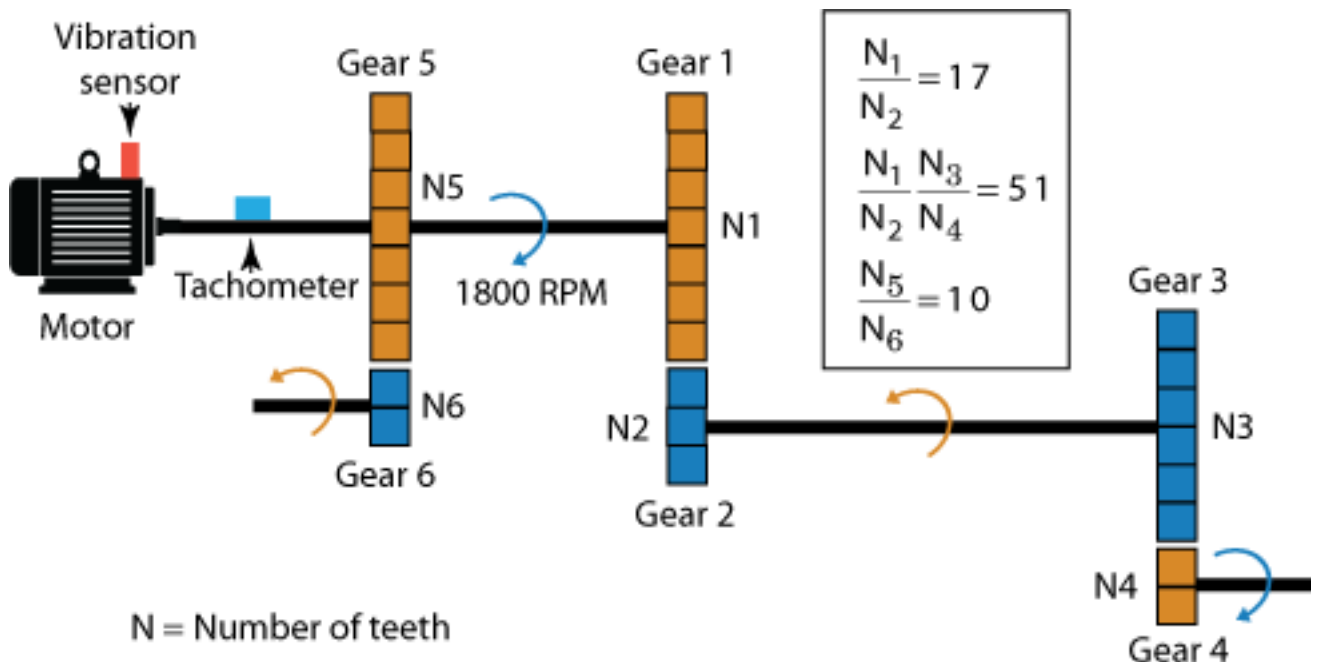
Twenty simulated machines use this drivetrain. Each machine operates with a nominal rotation speed within 1 percent of the design rotation speed of 1800 rpm. Therefore, the nominal rotation speed for each machine ranges between 1782 rpm and 1818 rpm.

Ten of the machines include a fault developing on the shaft of gear 6.

Import and Examine Measurement Data

To start, load the data into your MATLAB workspace and open **Diagnostic Feature Designer**.

```
load(fullfile(matlabroot, 'toolbox', 'predmaint', 'predmaintdemos', ...
    'motorDrivetrainDiagnosis', 'machineData3'), 'motor_data')
diagnosticFeatureDesigner
```

Import the data. To do so, in the **Feature Designer** tab, click **New Session**. Then, in the **Select more variables** of the **New Session** window, select `motor_data` as your source variable.
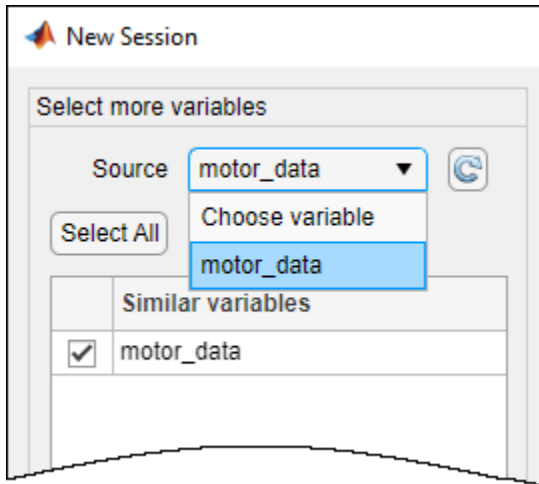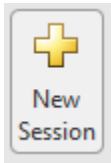
Complete the import process by accepting the default configuration and variables. The ensemble includes two data variables—`Signal/vib`, which contains the vibration signal, and `Tacho/pulse`, which contains the tachometer pulses. The ensemble also includes the condition variable `Health`.

In the **Data Browser**, select both signals and plot them together using **Signal Trace**.
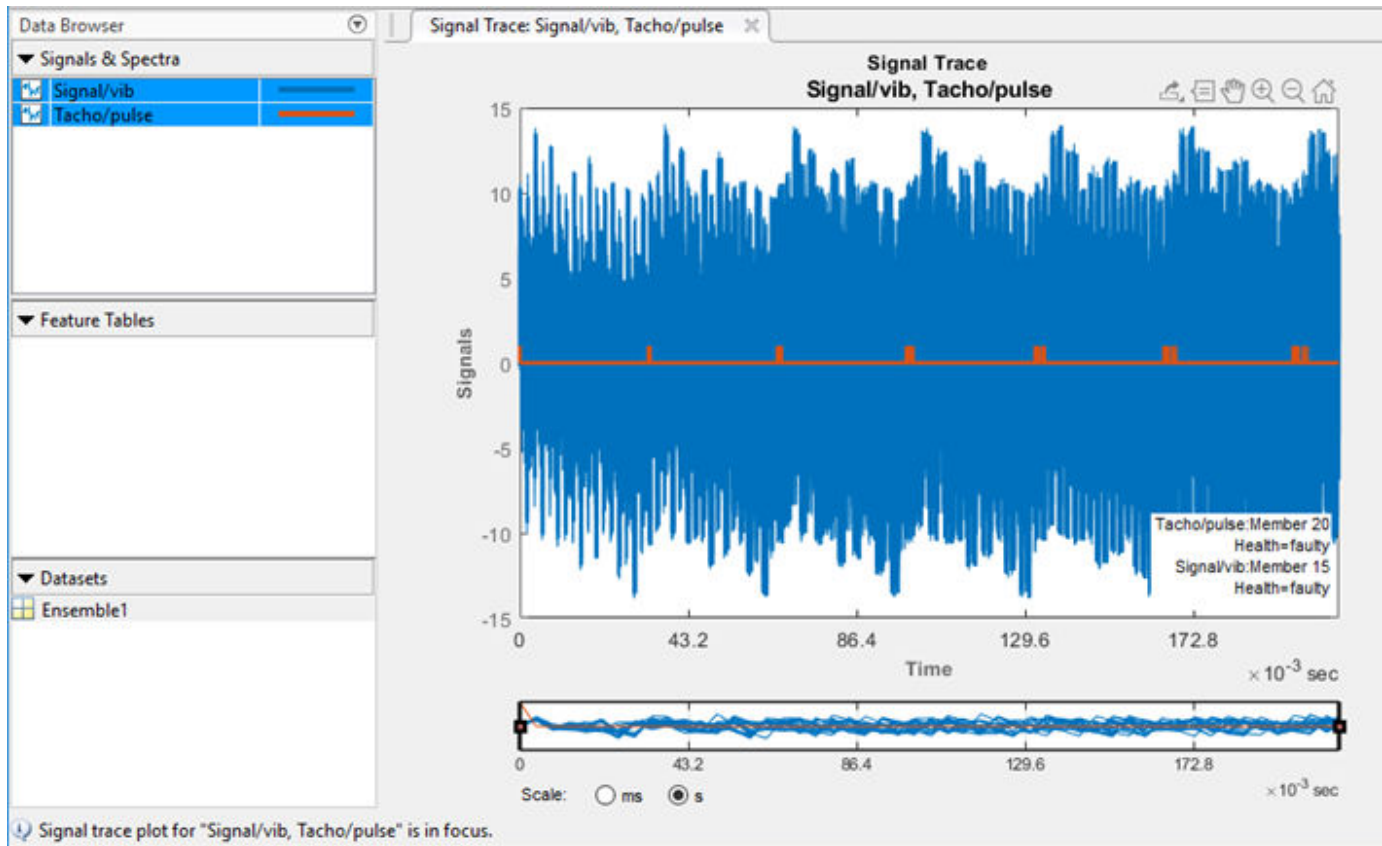
Note that the `Tacho` pulse clusters widen with each pulse because of rotation-speed variation. If the `Tacho` pulses are not visible, click **Group Signals** twice to bring the `Tacho` signal to the front.

Now group the data by fault condition by selecting **Ensemble View Preferences > Group by "Health"**. Use the panner to expand a slice of the signal.
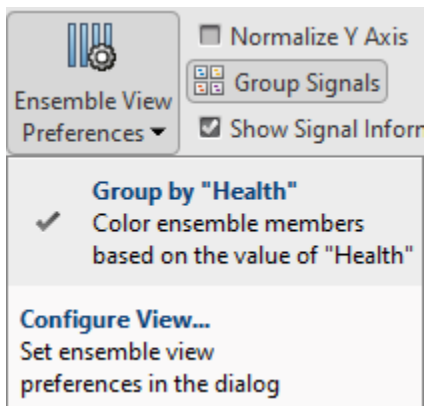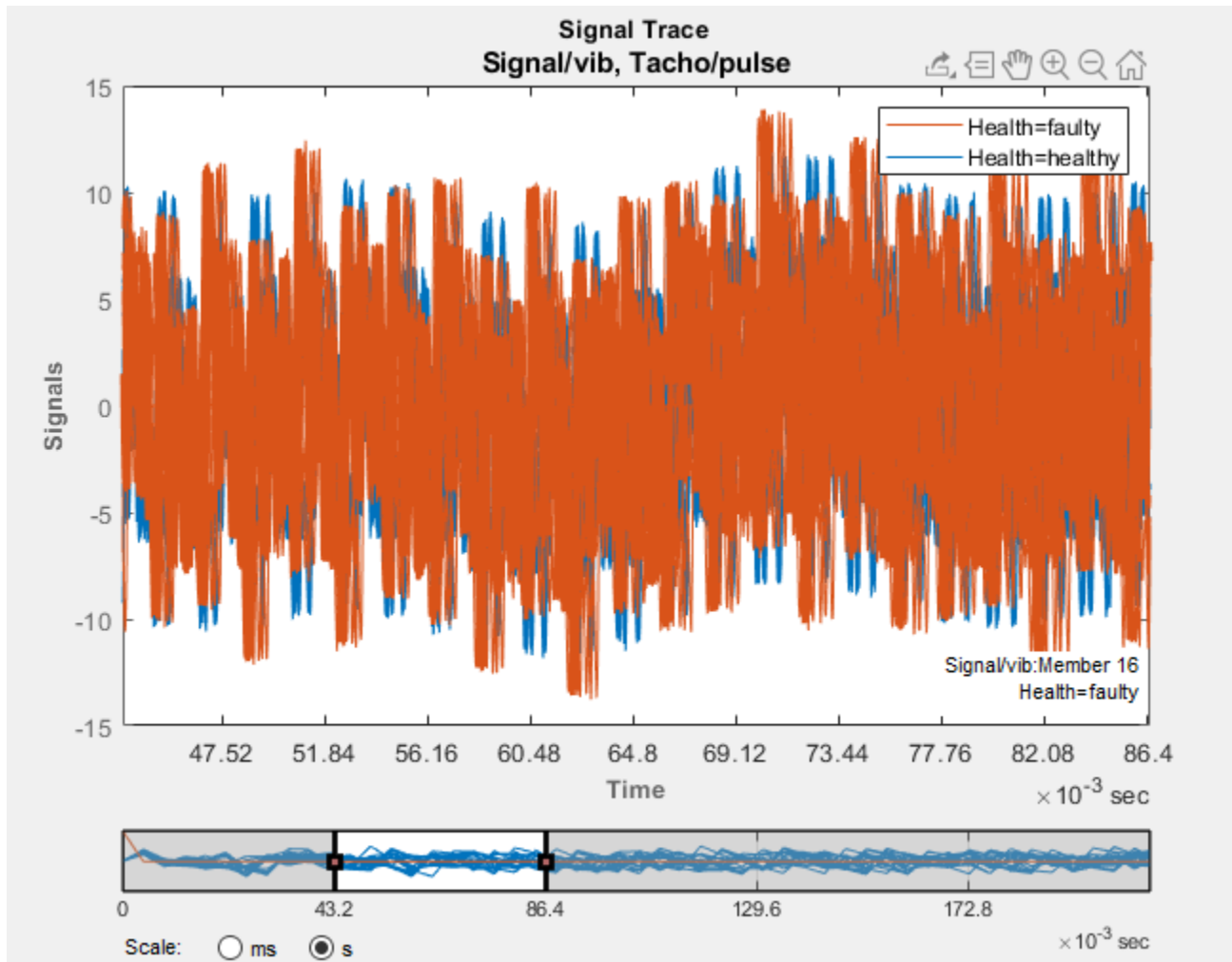
The plot shows small differences in the peaks of the groups, but the signals look similar otherwise.

Perform Time-Synchronous Averaging

Time-synchronous averaging (TSA) averages a signal over one rotation, substantially reducing noise that is not coherent with the rotation. TSA-filtered signals provide the basis for much rotational-machinery analysis, including feature generation.

In this example, the rotation speeds vary within 1 percent of the design value. You capture this variation automatically when you use the `Tacho` signal for TSA processing.

To compute the TSA signal, select **Filtering & Averaging > Time-Synchronous Signal Averaging**. In the dialog box:

- Confirm the selection in **Signal**.
- In **Tacho Information**, select **Tacho signal** and confirm the signal selection.

  Select **Compute nominal speed (RPM)**. This option results in the computation of a set of machine-specific nominal speeds, or operating points, from the `Tacho` signal. You can use this information when you perform follow-on processing, such as TSA signal filtering. Since your

tachometer variable is named `Tacho`, the app stores these values as the condition variable `Tacho_rpm`.

- Accept all other settings.
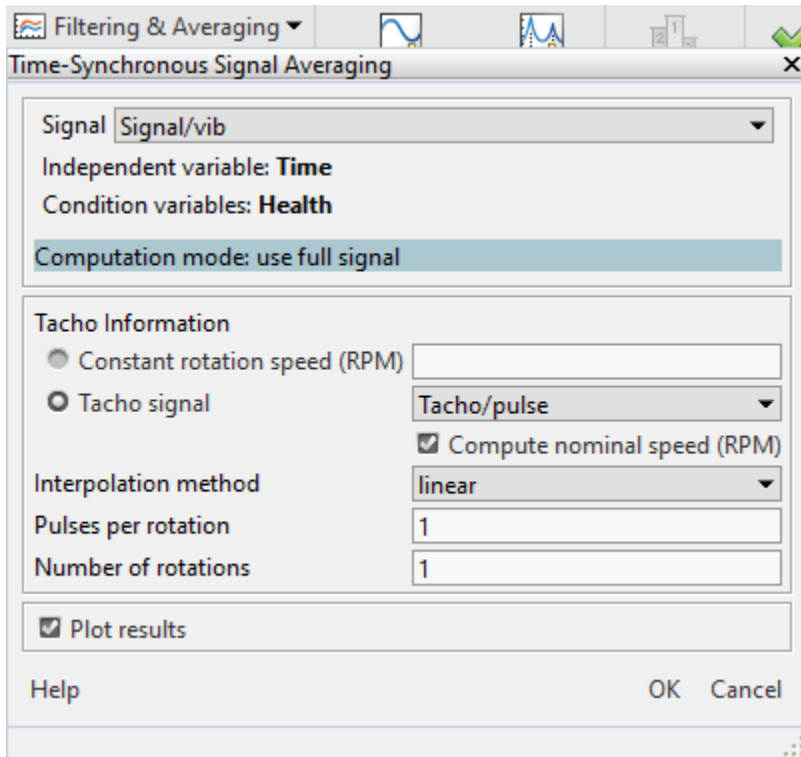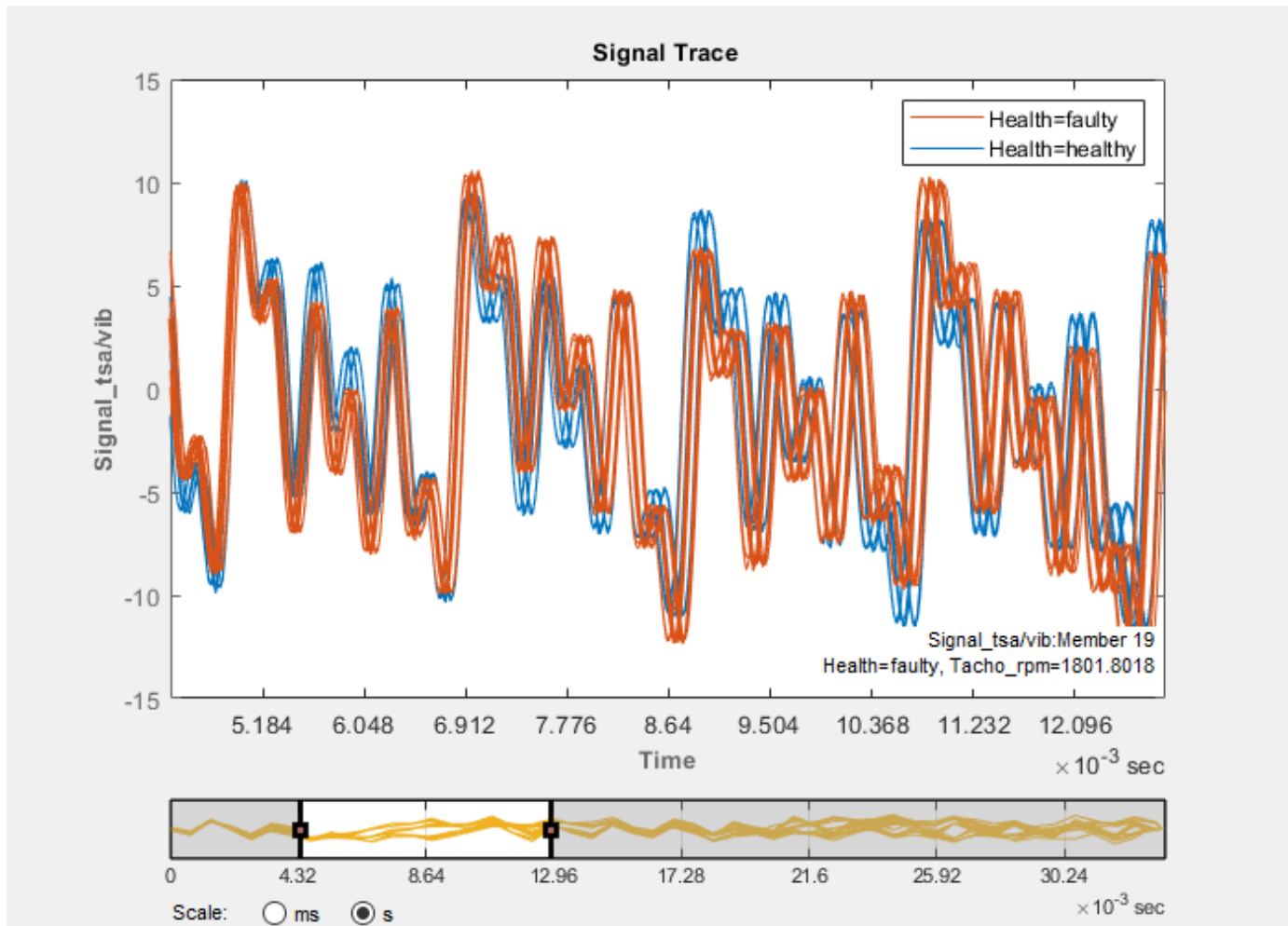
The TSA-averaged signals are cleaner than the raw signals. The peak heights are clustered by health condition, just as they were before, but the plot does not show enough information to indicate the source of the fault.

Compute TSA Difference Signal

The TSA filtering options in the app all start with a TSA signal and subtract various components from that signal to produce a filtered signal. Each filtered signal type yields unique features that you can use to detect specific faults in the gear train. One of these filtered signals is the difference signal. The TSA difference signal contains the components that remain after you subtract all of the components that are due to the drivetrain design for components that are not in your area of interest. Specifically, the TSA difference-signal processing subtracts:

- Shaft frequency and harmonics
- Gear-meshing frequencies and harmonics
- Sidebands at the gear-meshing frequencies and their harmonics

For this example, you are interested in the mesh defect between gear 5 and gear 6. To focus on signals arising from this defect, filter out signals associated with the other gears. To do so, use the

successive gear ratios described in the model description as you move down the drivetrain. The ratio between gears 1 and 2 is 17. The ratio between gears 1/2 and 3/4 is 51. These ratios become your rotation orders.

To compute the TSA difference signal, select **Filtering & Averaging > Filter Time-Synchronous Averaged Signals**. In the dialog box:

- Set **Signal** to the TSA signal `Signal_tsa/vib`.
- In **Signals to Generate**, select **Difference signal**.
- In **Speed Settings**, select **Nominal rotation speed (RPM)** and then `Tacho_rpm`.
- Confirm that **Domain** is `Order`.
- Set **Rotation orders** to `[17 51]`.
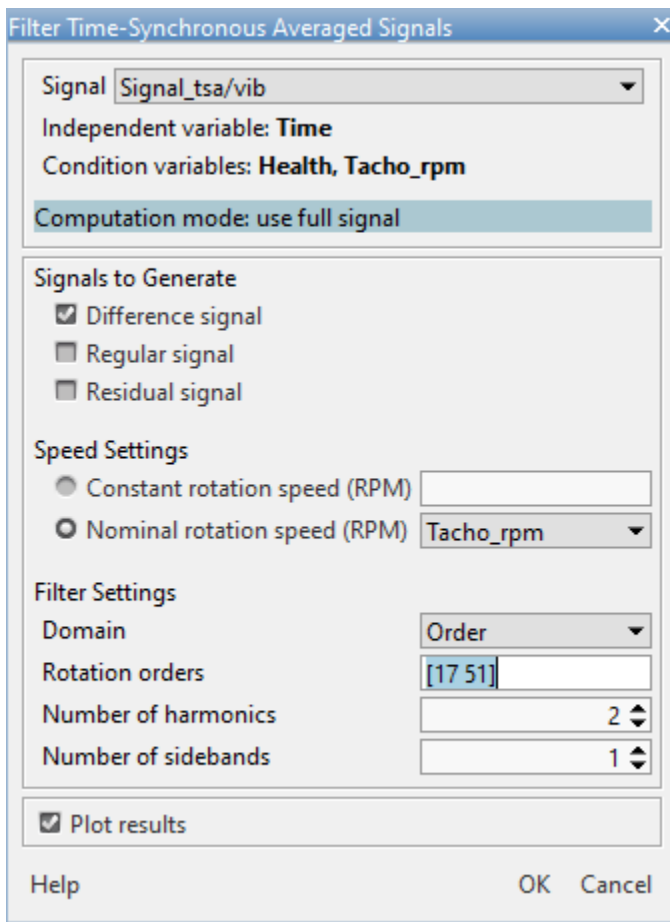
When you group the plotted data by `Health`, you also see the condition variable `Tacho_rpm` in the option list.
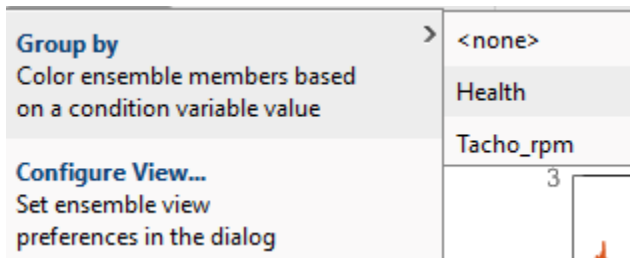
The resulting plot shows a clear oscillation. Group the data by `healthy` and `faulty` labels. The oscillation is present only for the `faulty` machines. Using the data cursors, you can show that the period of this oscillation is about 0.0033 seconds. The corresponding frequency of the oscillation is about 303 Hz or 18,182 rpm. This frequency has roughly a 10:1 ratio with the primary shaft speed of 1800 rpm, and is consistent with the 10:1 gear ratio between gear 5 and gear 6. The difference signal therefore isolates the source of the simulated fault.
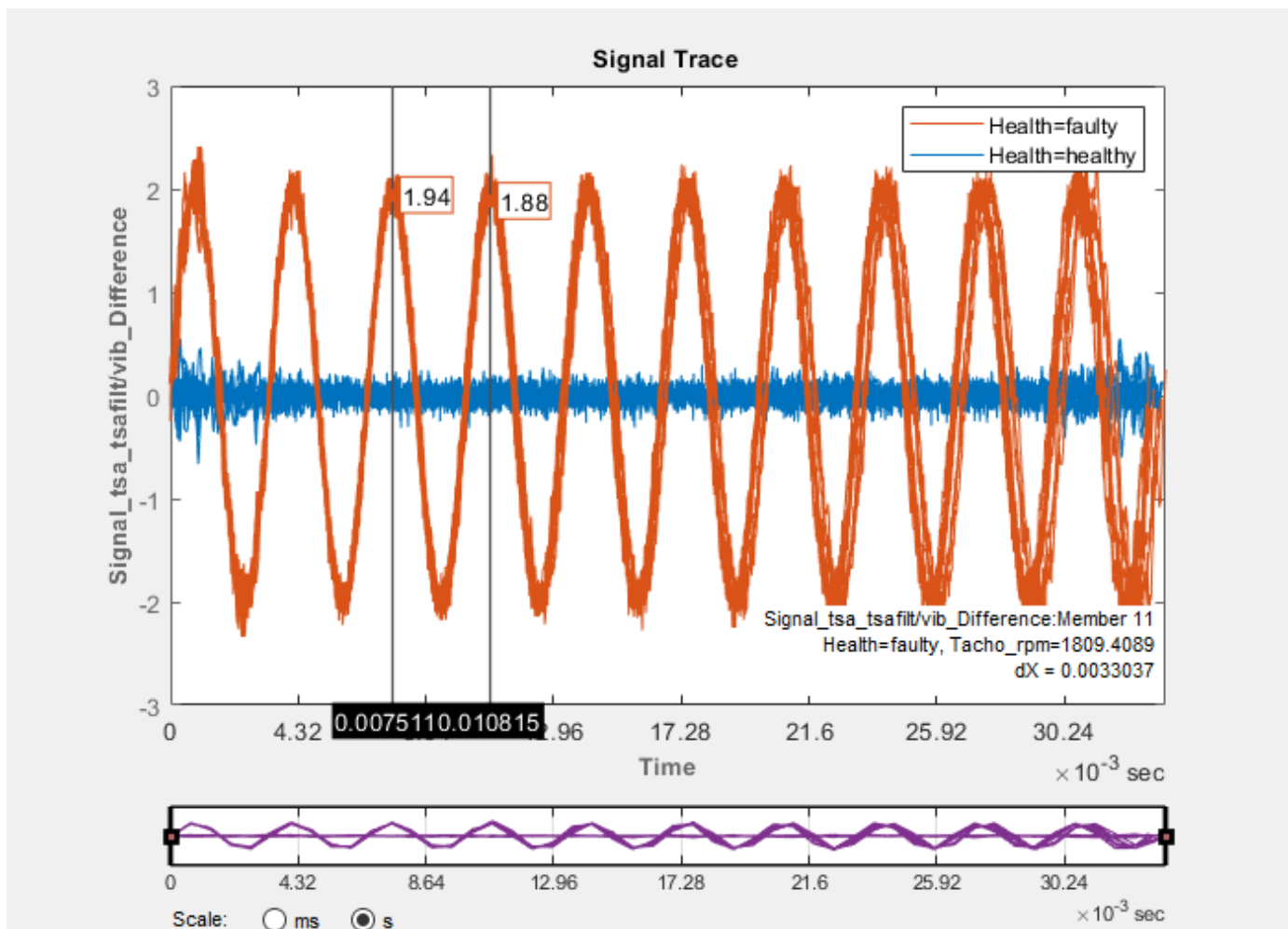
Isolate the Fault Without a Tachometer Signal

In the preceding sections, you use tachometer pulses to precisely generate TSA signals and difference signals. If you do not have tachometer information, you can use a constant rpm value to generate these signals. However, the results are less precise.

See if you can isolate the defect without using the `Tacho` signal. Compute both the TSA signal and the difference signal with an ensemble-wide rotation speed of 1800 rpm.
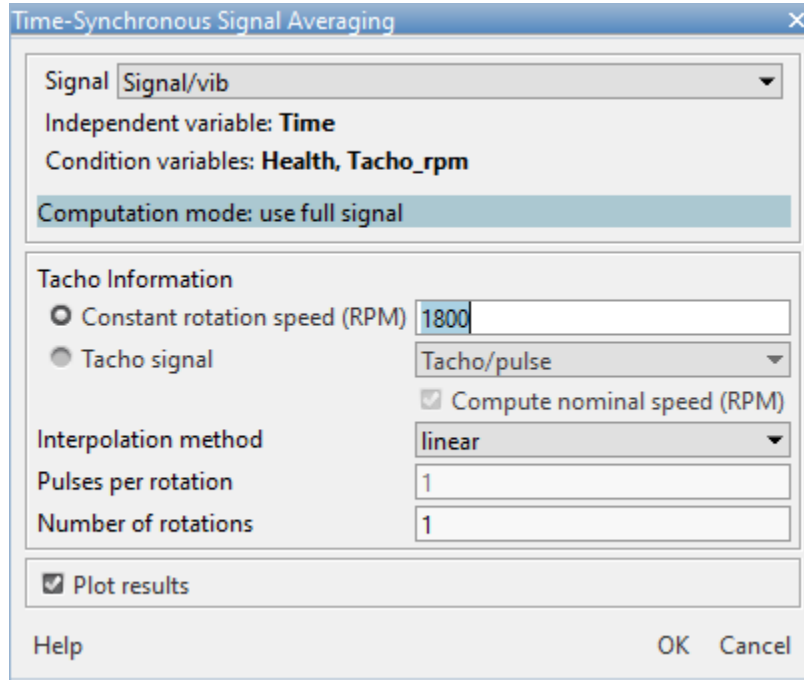
The new TSA signal has the name `Signal_tsa_1/vib`. As the plot shows, generating a TSA signal without tachometer information produces a less clear signal then generating a signal with tachometer information.
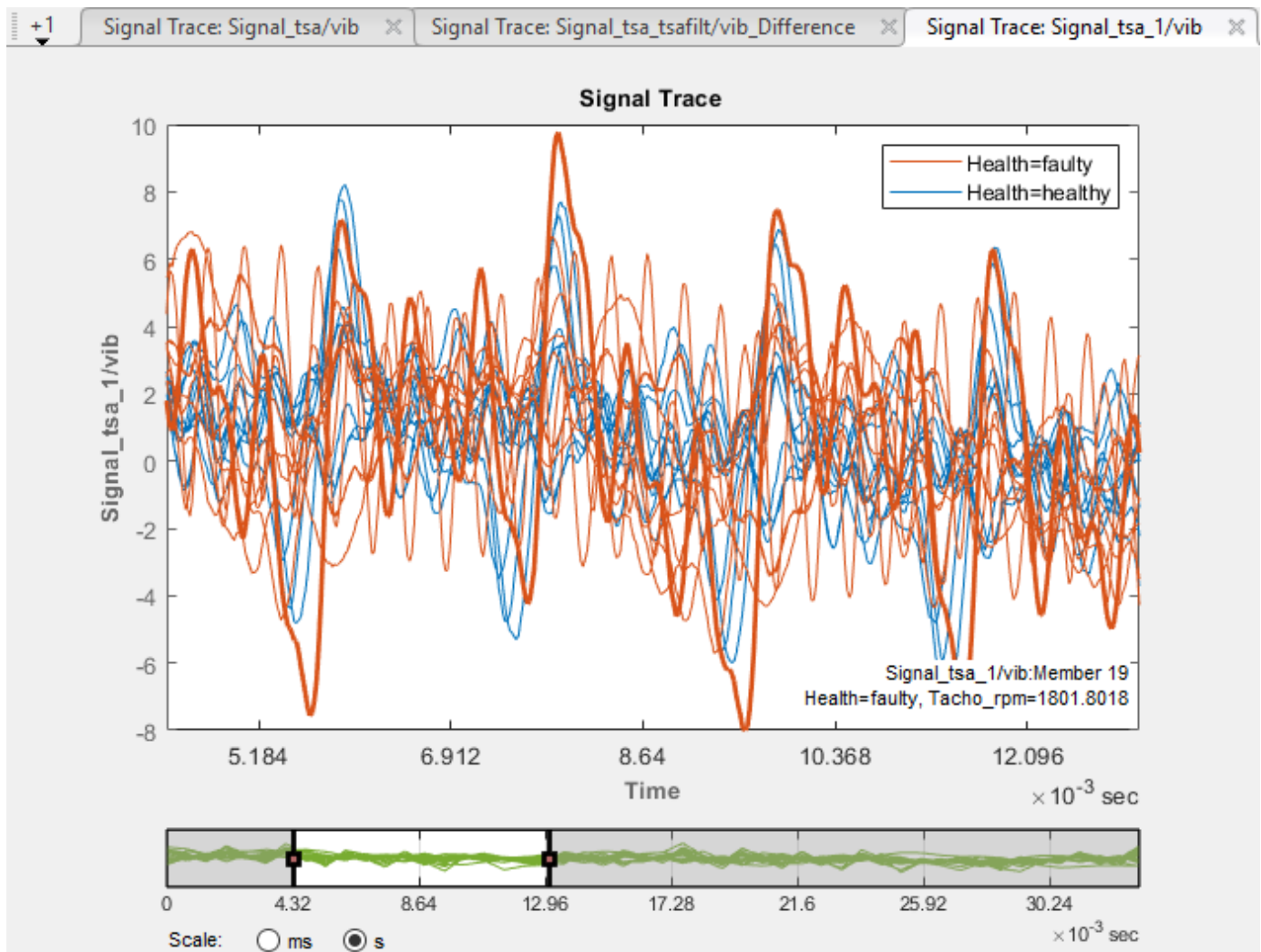
Compute the difference signal using `Signal_tsa_1/vib` and the **Constant rotation speed (RPM)** setting.
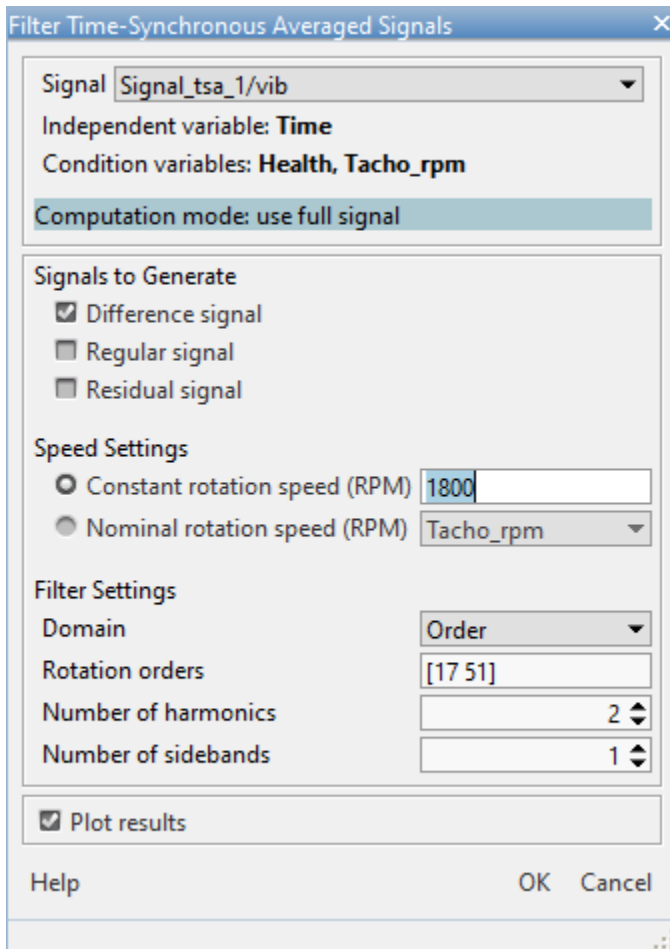
In the resulting plot, you can still see the oscillation for the `faulty` set of machines, but as with the TSA signal, the oscillation is much less clear than the previous difference signal. The unaccounted-for 1-percent rpm variation has a significant impact on the results.
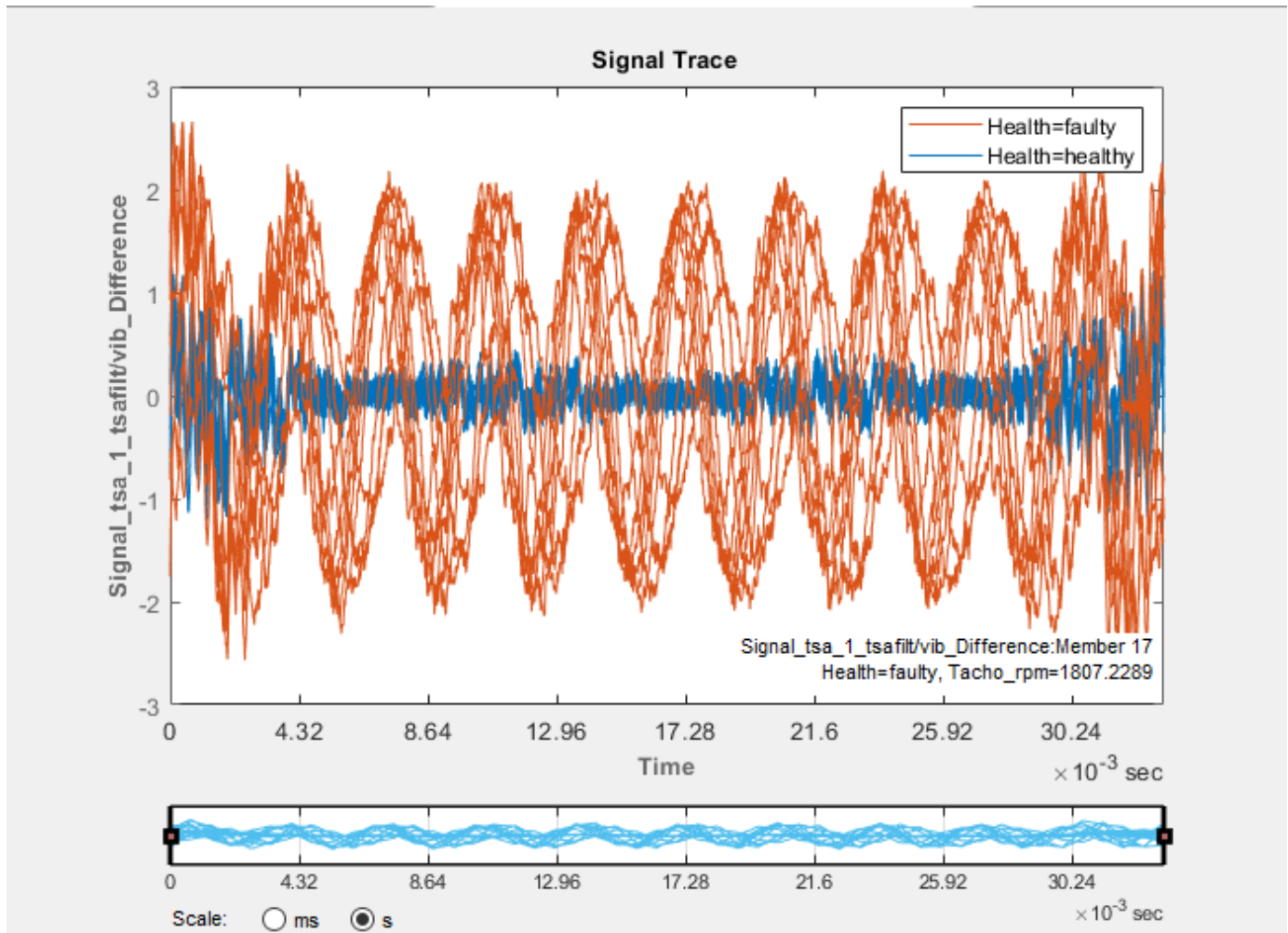
Extract Rotating Machinery Features

Use the TSA and difference signals `Signal_tsa/x` and `Signal_tsa_tsafilt/x_Difference` to compute time-domain rotational machinery features.

To compute these features, select **Time-Domain Features > Rotating Machinery Features**. In the dialog box, select the signals to use for **TSA signal** and **Difference signal** and then select all the feature options that use TSA or difference signals.

Rotating Machinery Features   ✕

Signals to Use
TSA signal   Signal_tsa/vib   ▼
Difference signal   Signal_tsa_tsafilt/vib_Difference ▼
Regular signal   ▼
Computation mode: use full signal

Metrics using TSA signals
☑ Root mean square (RMS)
☑ Kurtosis
☑ Crest factor

Metrics using difference signals
☑ Kurtosis (FM4)
☑ Normalized 6th moment (M6A)
☑ Normalized 8th moment (M8A)

Metrics using regular signals
☐ Zero-order figure of merit (FM0)

Metrics using a mix of signals
☐ Energy ratio

☑ Plot results

Help   OK   Cancel

The resulting histogram plots indicate good separation between `healthy` and `faulty` groups for all of the TSA-signal-based features, and for FM4 (kurtosis) in the difference-signal-based features.
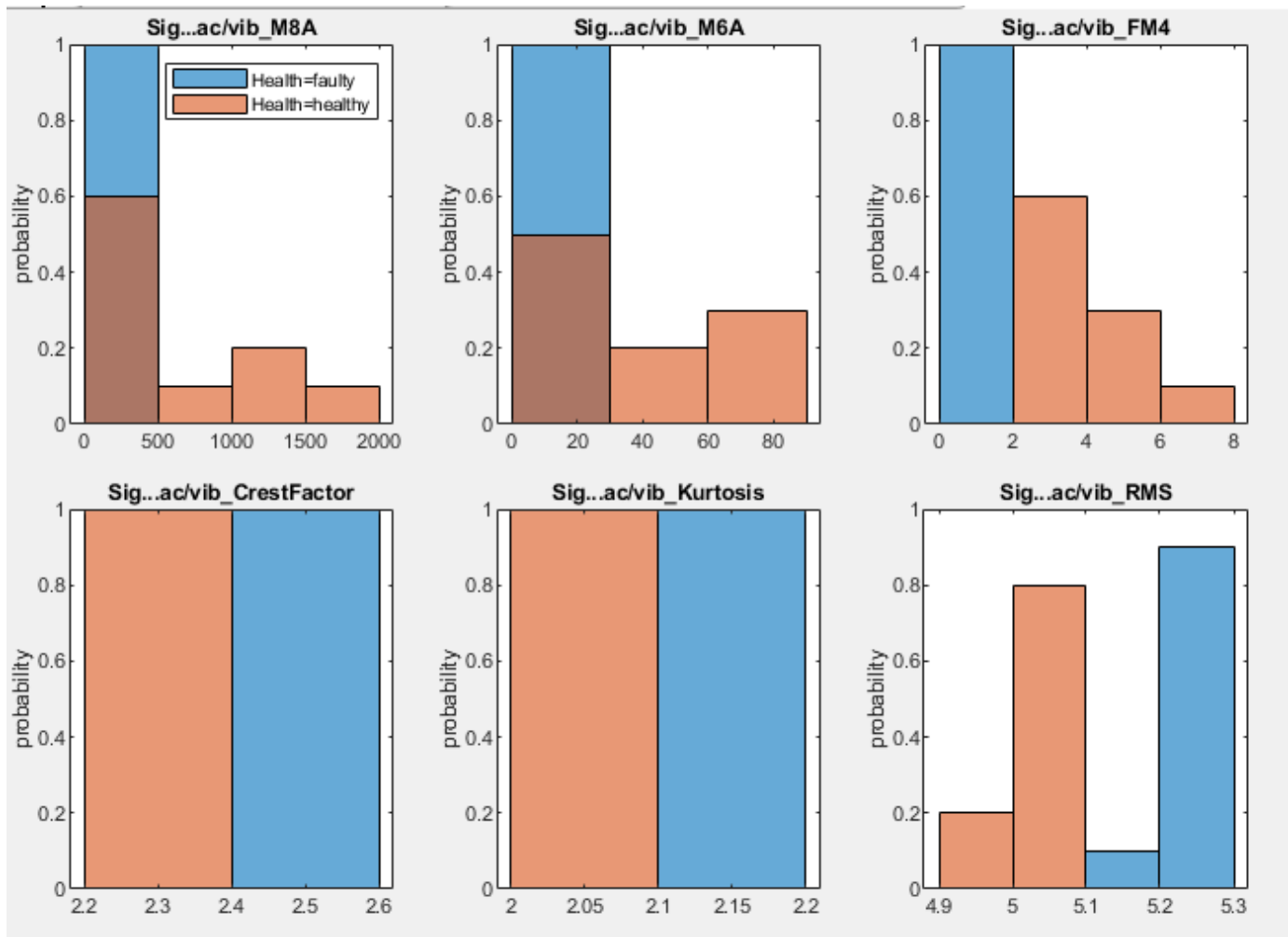
Extract Spectral Features

Since the difference signal displays a clear oscillation that is limited to the `faulty` group, spectral features are also likely to differentiate well between `healthy` and `faulty` groups. To calculate spectral features, you must first compute a spectral model. To do so, click **Spectral Estimation > Order Spectrum**. As before, select your difference signal as **Signal** and your tachometer signal as **Tacho signal**.
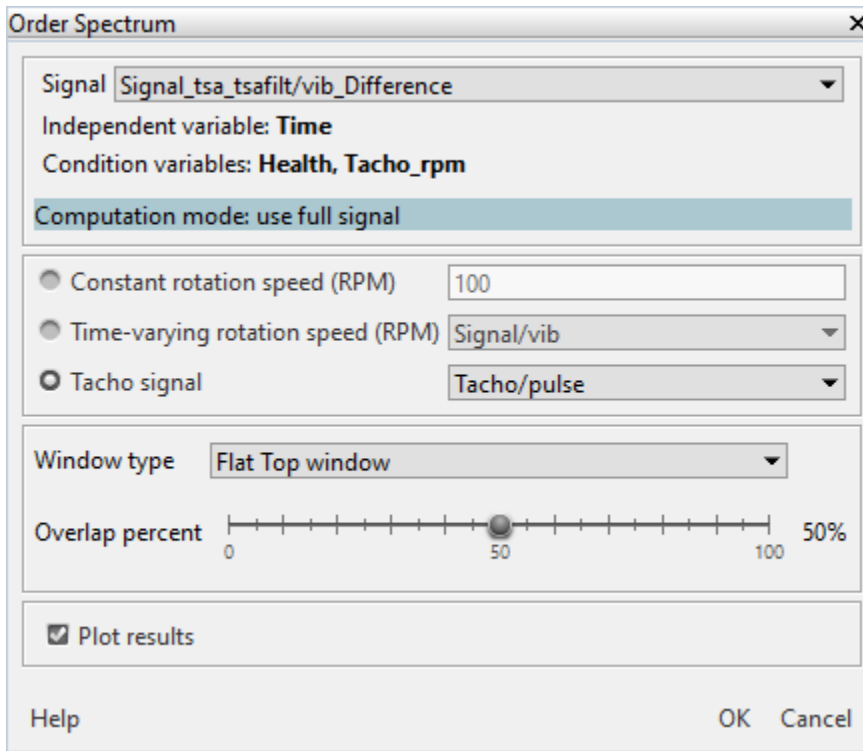
The resulting plot displays the oscillation of the defect as the first peak in the plot at roughly order 10.
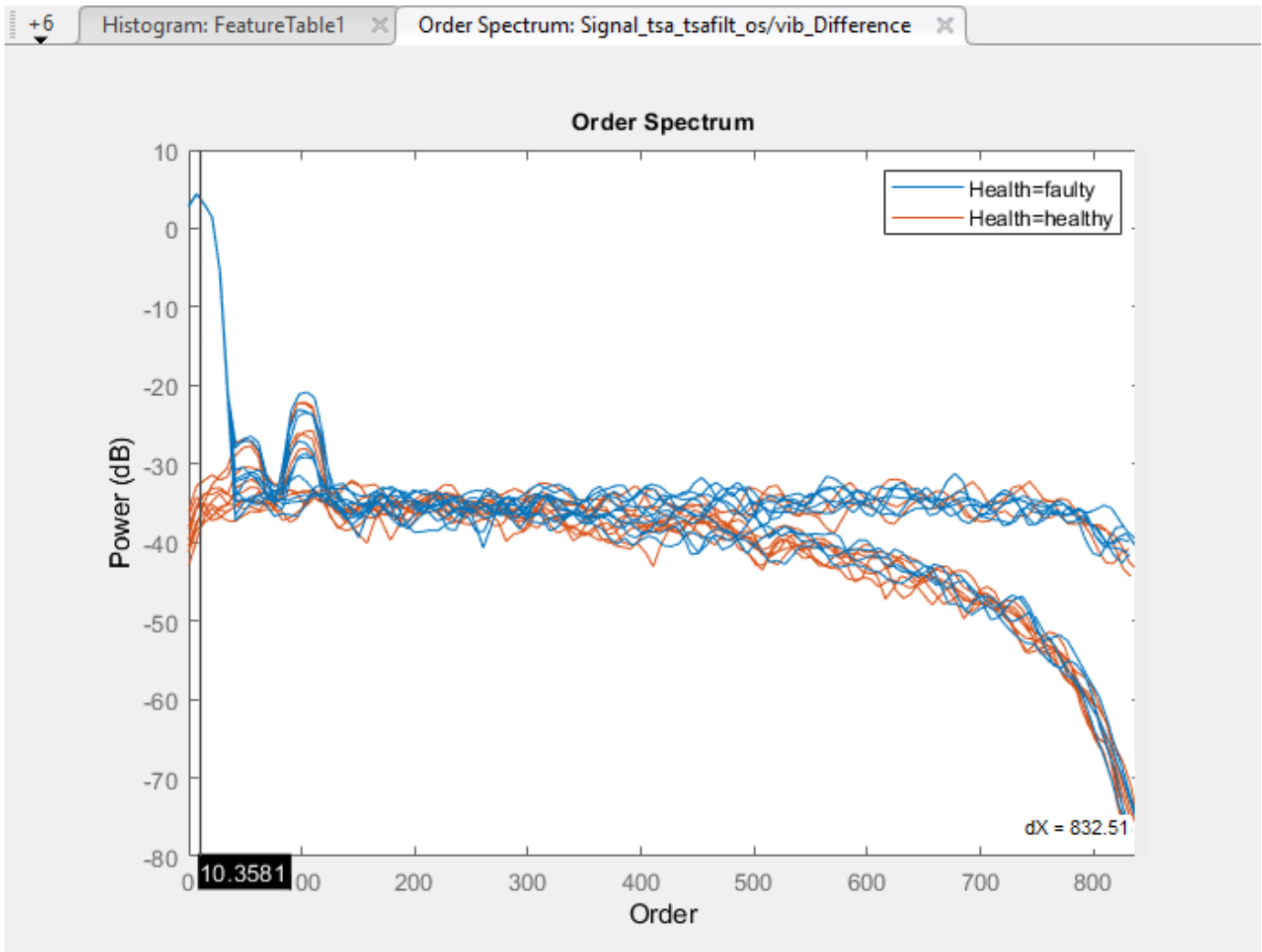
Compute the spectral features by clicking **Spectral Features**. In the dialog box:

- Confirm the selection for **Spectrum**.
- Move the order range slider to cover the range from 0 to 200. As you move the slider, the order spectrum plot incorporates the change.
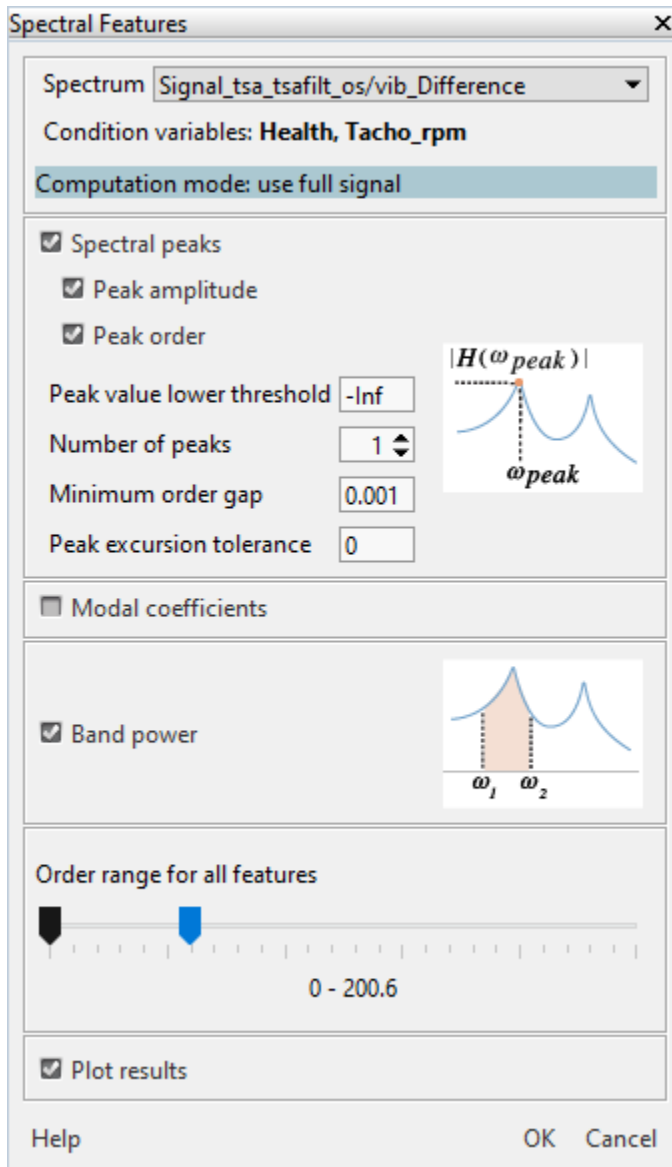
The resulting histograms indicate good differentiation between groups for `BandPower` and `PeakAmp`. `PeakFreq1` shows a small amount of group overlap.
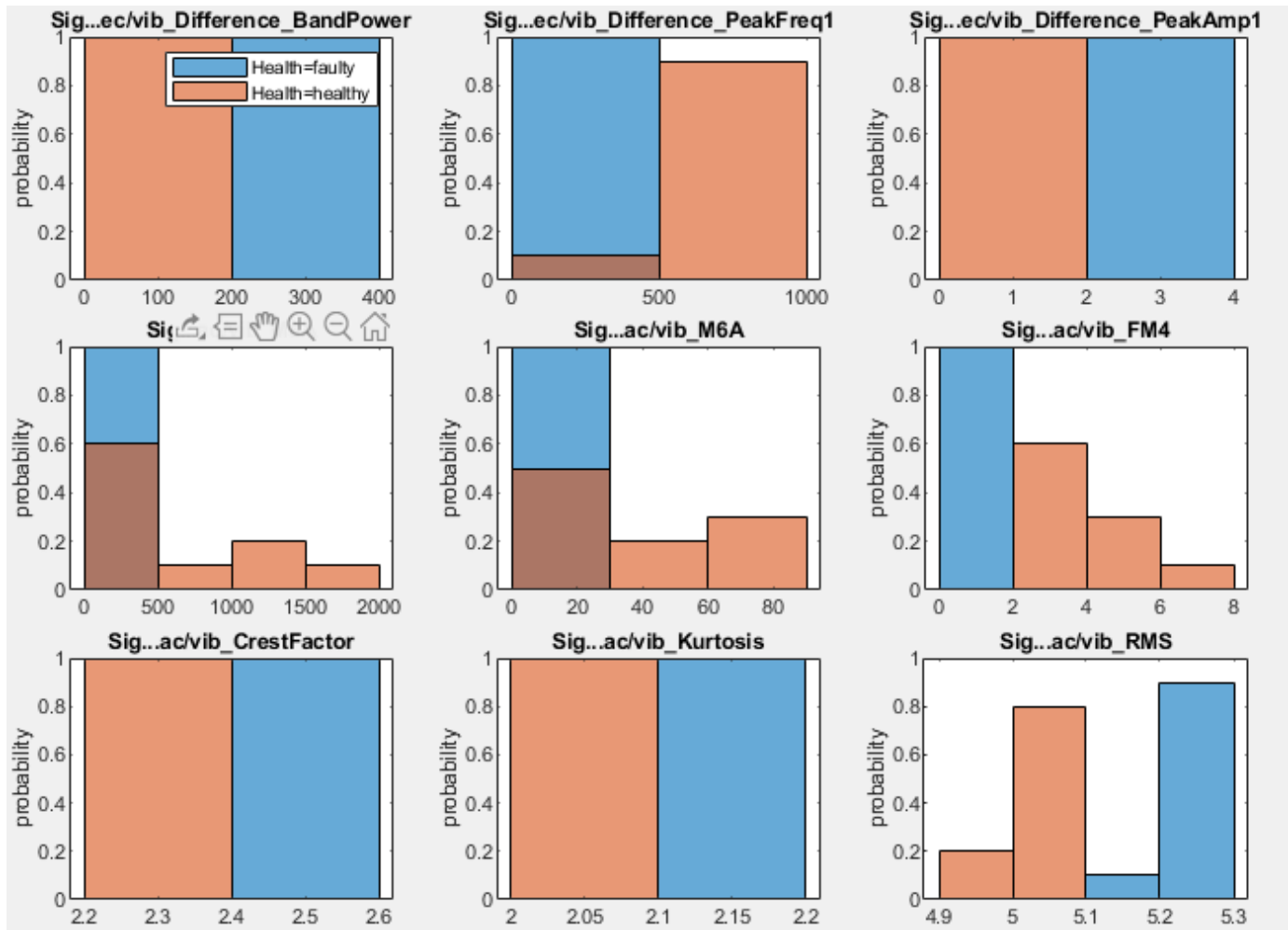
Rank Features

Rank the features using the default T-test ranking. To do so, click **Rank Features** and select `FeatureTable1`. The app automatically ranks the features and plots the scores.

- Spectral features `BandPower` and `PeakAmp` take the two top places with scores significantly higher than the other features.
- Rotational features `Kurtosis` and `CrestFactor` take the third and fourth places with scores much lower than the spectral features, but still significantly higher than the remaining features.
- The remaining features are likely not useful for detecting faults of this type.
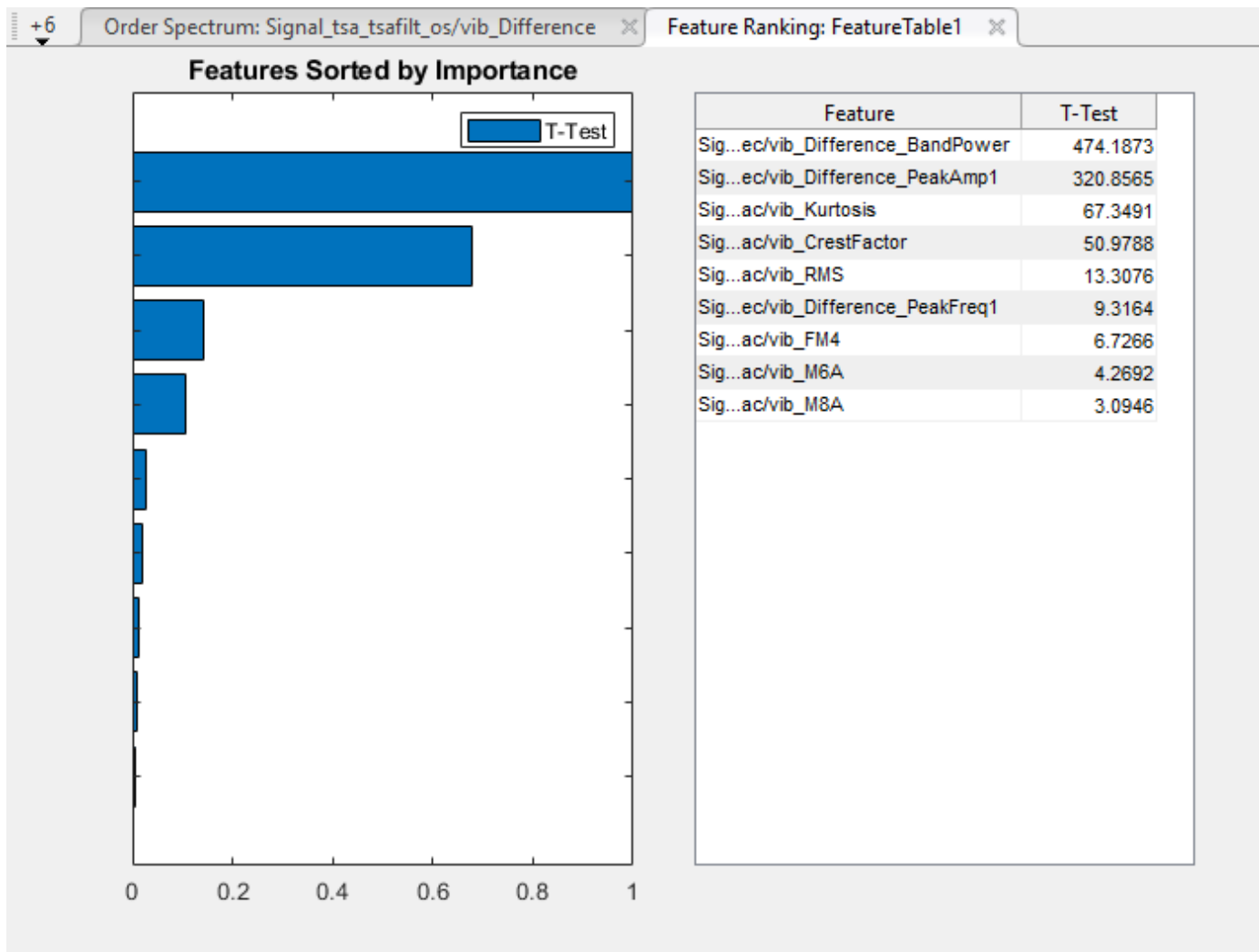
Using these high-ranking features, you could now move on to export the features to **Classification Learner** for training or to your MATLAB workspace for algorithm incorporation.

See Also

Diagnostic Feature Designer | tsa | tsadifference | gearConditionMetrics

More About

- "Identify Condition Indicators for Predictive Maintenance Algorithm Design"
- "Condition Indicators for Gear Condition Monitoring" on page 3-10
- "Explore Ensemble Data and Compare Features Using Diagnostic Feature Designer" on page 7-2
- "Perform Prognostic Feature Ranking for a Degrading System Using Diagnostic Feature Designer" on page 7-65
- "Designing Algorithms for Condition Monitoring and Predictive Maintenance"

Perform Prognostic Feature Ranking for a Degrading System Using Diagnostic Feature Designer

This example shows how to process and extract features from segmented data that contains evidence of a worsening shaft fault, and how to perform prognostic ranking to determine which features are best for determining remaining useful life (RUL). RUL feature development is based on run-to-failure data rather than conditionally grouped data.

The example assumes that you are already familiar with basic operations with the app. For a tutorial on using the app, see "Identify Condition Indicators for Predictive Maintenance Algorithm Design".

Model Description

The following figure illustrates a drivetrain with six gears. The motor for the drivetrain is fitted with a vibration sensor. The drivetrain has no tachometer. The motor drives a constant rotation speed of 1800 rpm with no variation. In this drivetrain:

- Gear 1 on the motor shaft meshes with gear 2 with a gear ratio of 17:1.
- The final gear ratio, or the ratio between gears 1 and 2 and gears 3 and 4, is 51:1.
- Gear 5, also on the motor shaft, meshes with gear 6 with a gear ratio of 10:1.
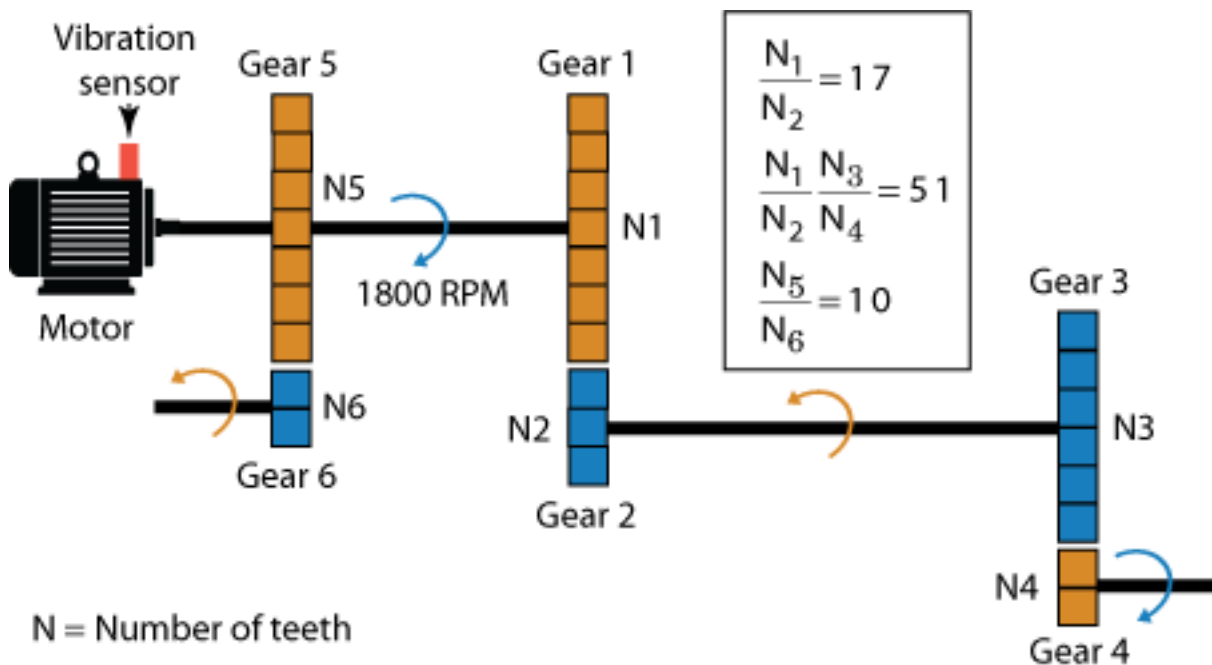
Ten simulated machines use this drivetrain. All of the machines have a fault developing on the shaft of gear 6. This fault becomes worse every day. The rate of the fault progression is fixed for each machine, but varies over the set of machines.

Data has been recorded during one 0.21 s period each day for 15 days. For each machine, these daily segments are stored in adjacent positions within a single variable. The timestamps reflect the data recording time and increase continuously. For instance, if the timestamp on the final sample of day 1 is t_f and the sample time is T_s, then the timestamp on the first sample of day 2 is $t_f + T_s$.

Import and View the Data

To start, load the data into your MATLAB workspace and open **Diagnostic Feature Designer**.

```
load(fullfile(matlabroot, 'toolbox', 'predmaint', 'predmaintdemos', ...
  'motorDrivetrainDiagnosis', 'machineDataRUL3'), 'motor_rul3')
diagnosticFeatureDesigner
```

Import the data. To do so, in the **Feature Designer** tab, click **New Session**. Then, in the **Select more variables** area of the **New Session** window, select motor_rul3 as your source variable.
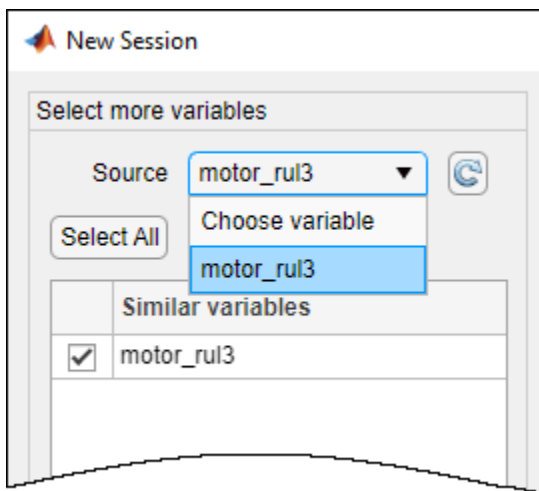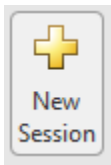
Complete the import process by accepting the default configuration and variables. The ensemble consists of one data variable Signal/vib, which contains the vibration signal. There are no condition variables.

View the vibration signal. To do so, in the **Data Browser**, select the signal and plot it using **Signal Trace**. The amplitude of the signal increases continuously as the defect progresses.
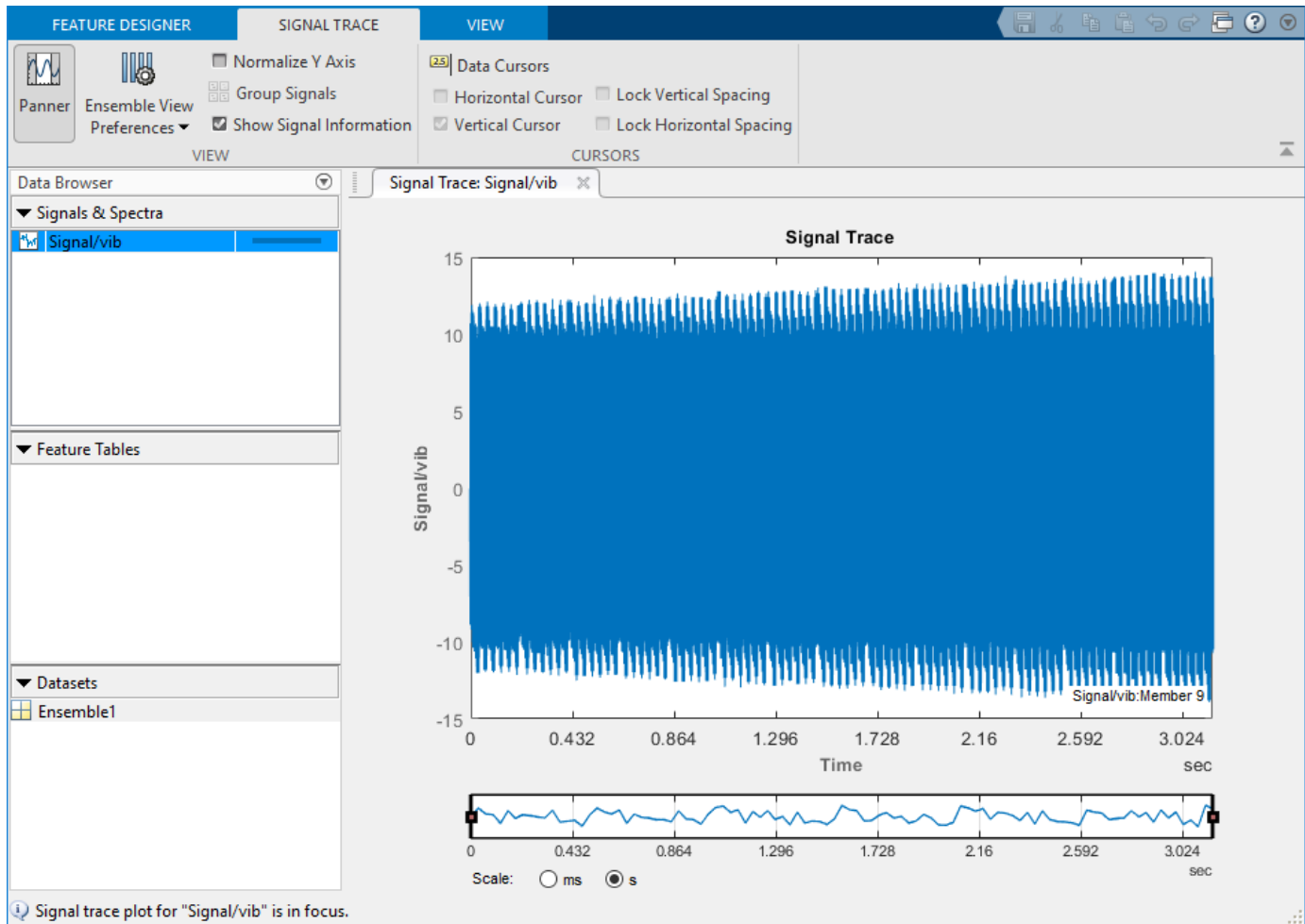
Separate Daily Segments by Frame

When developing features for RUL use, you are interested in tracking the progression of degradation rather than in isolating specific faults. The time history of a useful RUL feature provides visibility into the degradation rate, and ultimately enables the projection of time to failure.

Frame-based processing allows you to track the progression of degradation segment by segment. Small or abrupt changes are captured in the segment that they occur. Segment-based features convey a more precise record of the degradation than features extracted from the full signal can provide. For RUL prediction, the progression rate of the degradation is as important as the magnitude of the defect at a given time.

The data set for each machine in supports segmented processing by providing a segment of data for each day. Specify frame-based processing so that each of these segments is processed separately. Since the data has been collected in 0.21 s segments, separate the data for processing into 0.21 s frames.

Click **Computation Options**. In the dialog box, set **Data Handling Mode** to **Frame-based**. The data segments are contiguous, so set both the frame size and frame rate to `0.21` seconds.
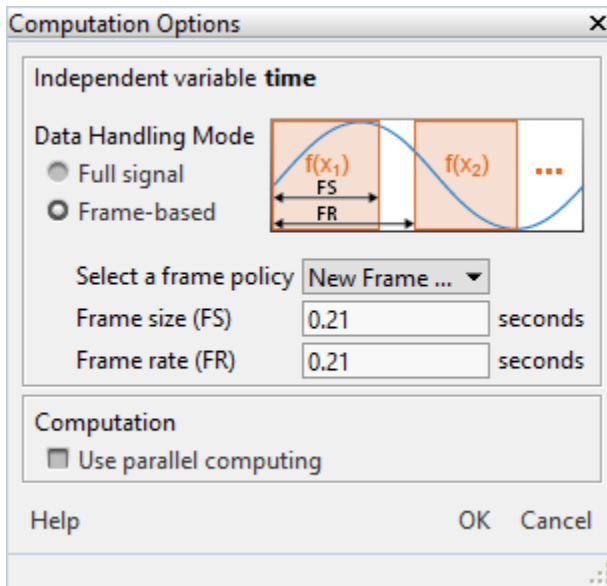
Perform Time-Synchronous Averaging

Time-synchronous averaging (TSA) averages a signal over one rotation, substantially reducing noise that is not coherent with the rotation. TSA-filtered signals provide the basis for much rotational-machinery analysis, including feature generation.

In this example, the rotation speed is fixed to the same 1800 rpm value for each machine.

To compute the TSA signal, select **Filtering & Averaging > Time-Synchronous Signal Averaging**. In the dialog box:

- Confirm the selection in **Signal**.
- In **Tacho Information**, select **Constant rotation speed (RPM)** and set the value to 1800.
- Accept all other settings.

The app computes the TSA signal for each segment separately, and by default plots the first segment.
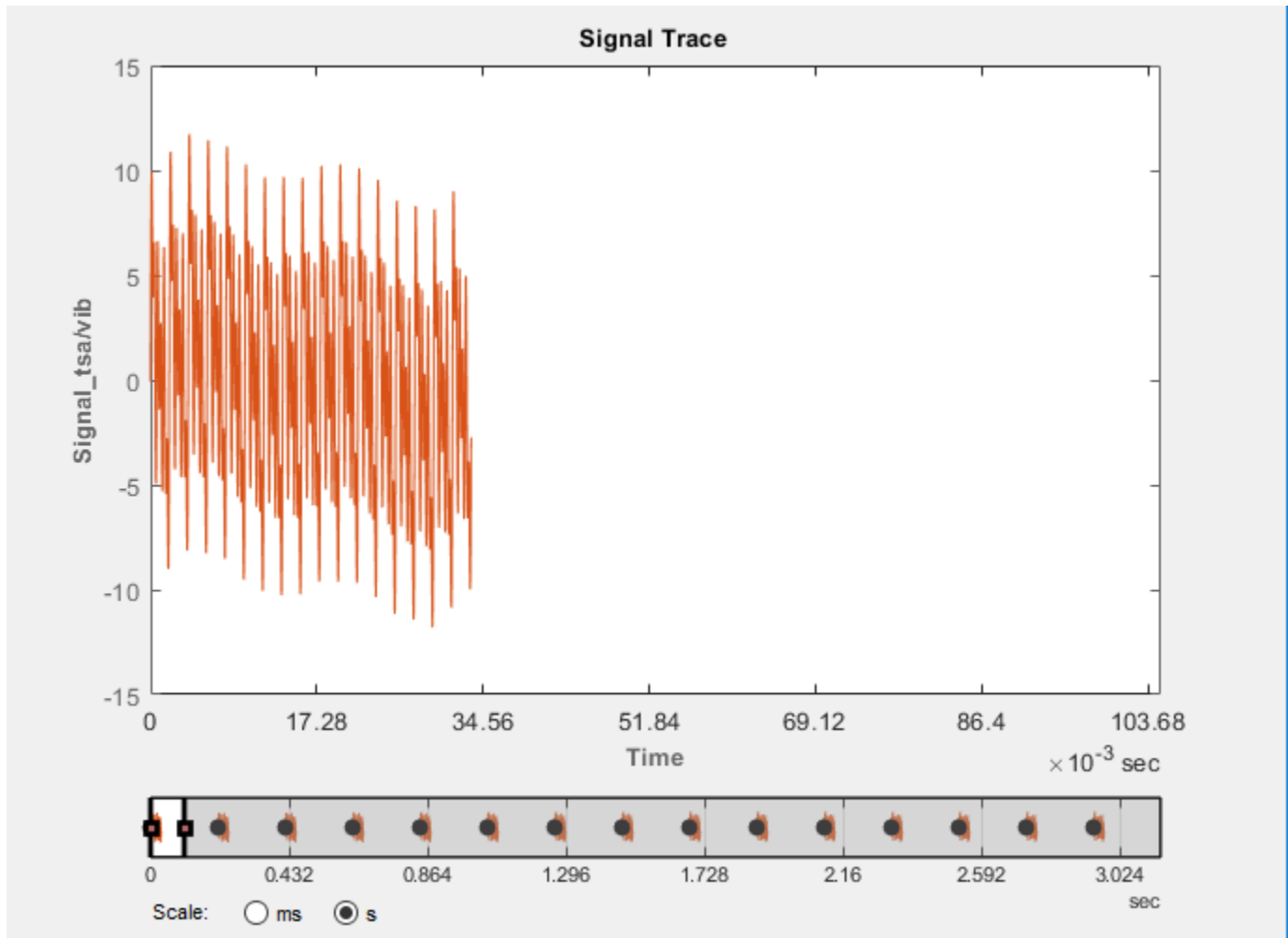
Use the panner to expand the plot to all segments. The plot shows a slightly increasing amplitude.
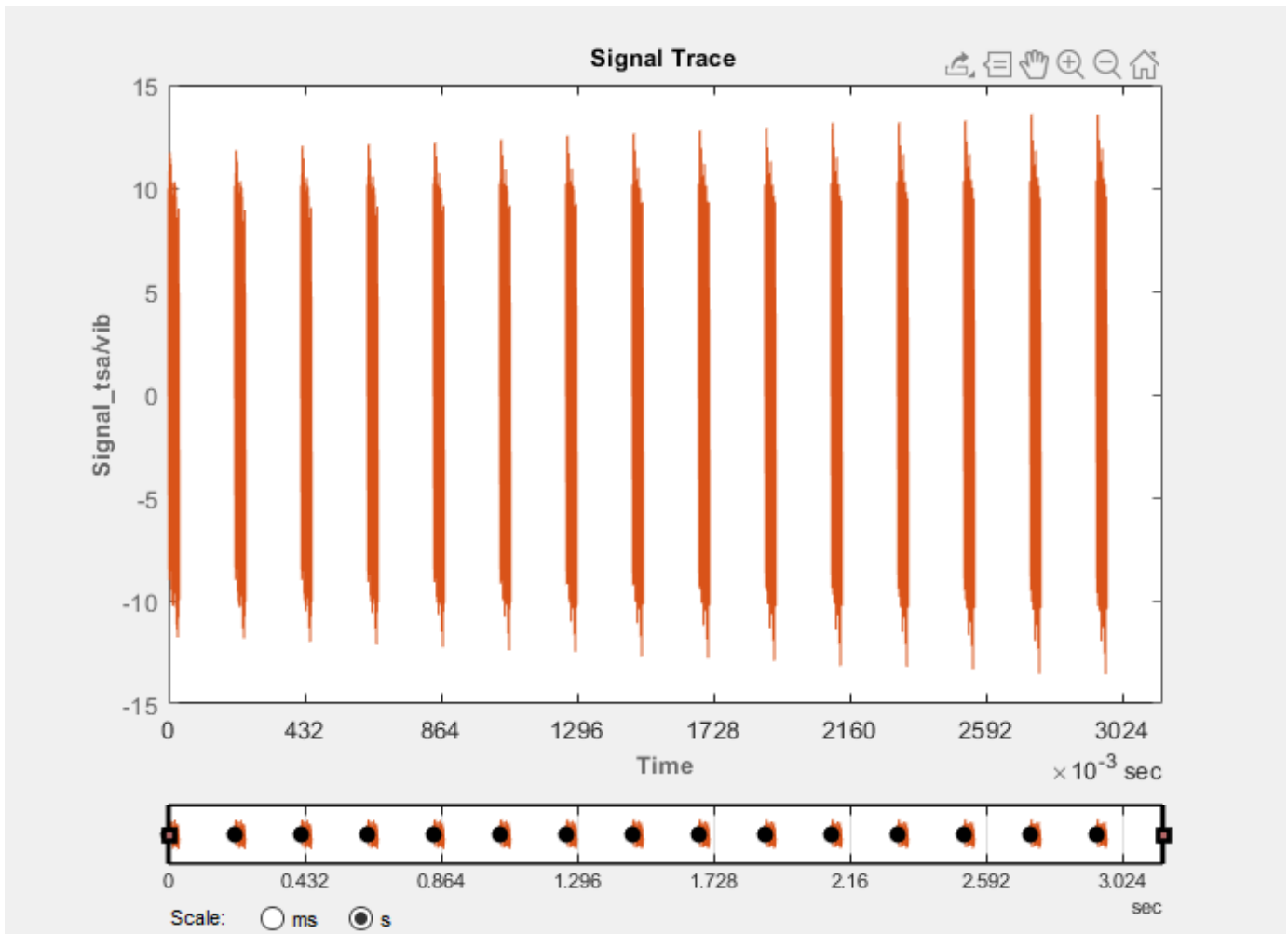
Extract Rotating Machinery Features

Use the TSA signal to compute time-domain rotating machinery features.

Rotating Machinery Features ✕

Signals to Use

TSA signal   [Signal_tsa/vib   ▼]
Difference signal [▼]
Regular signal   [▼]
Computation mode: use signal frames

Metrics using TSA signals
☑ Root mean square (RMS)
☑ Kurtosis
☑ Crest factor

Metrics using difference signals
☐ Kurtosis (FM4)
☐ Normalized 6th moment (M6A)
☐ Normalized 8th moment (M8A)

Metrics using regular signals
☐ Zero-order figure of merit (FM0)

Metrics using a mix of signals
☐ Energy ratio

☑ Plot results

Help   OK   Cancel

Since you have no condition variables, the resulting histograms display only the distribution of the feature values across the segments.
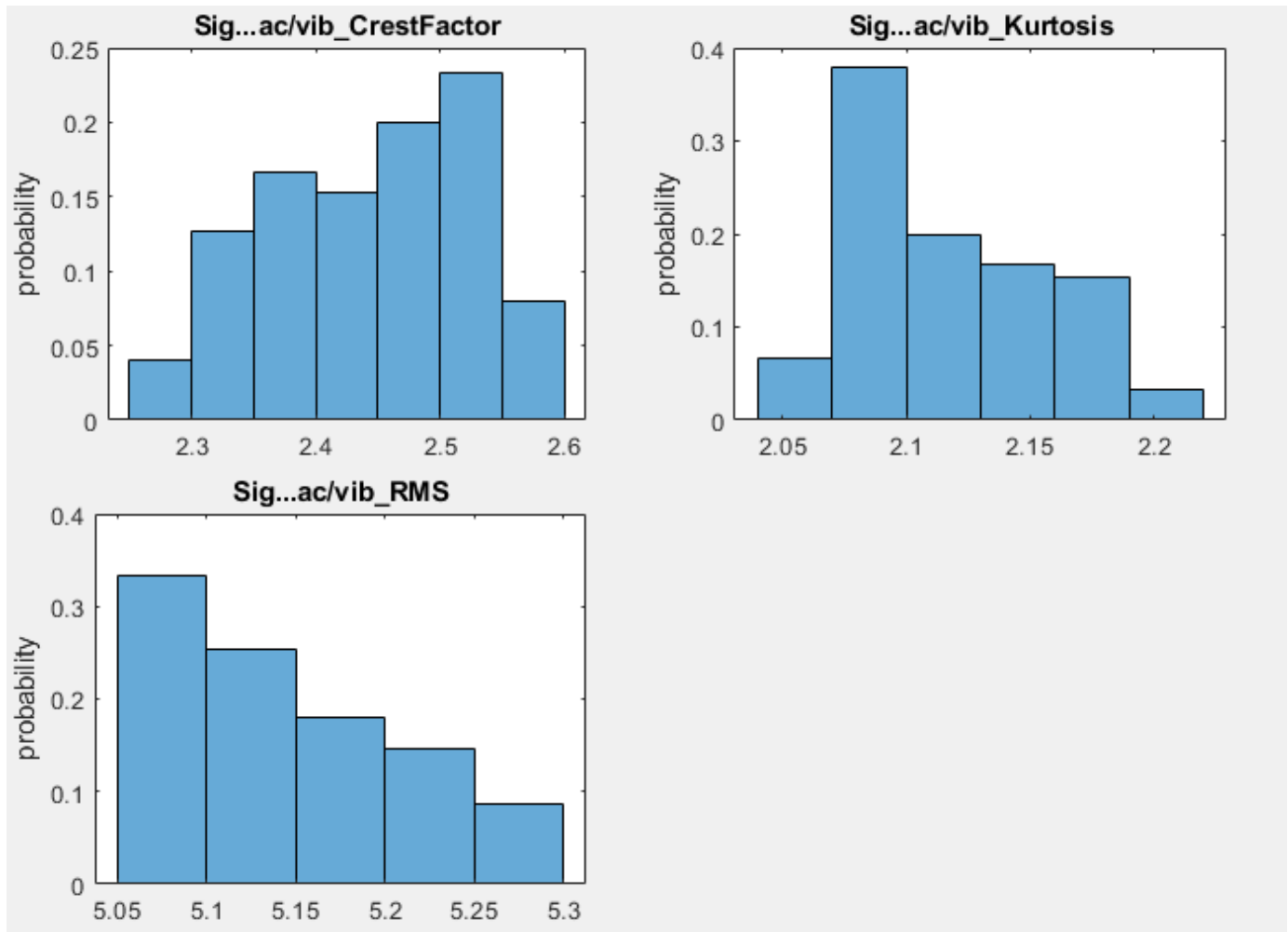
You can also look at the feature trace plots to see how the features change over time. To do so, in **Feature Tables**, select `FeatureTable1`. In the plot gallery, select **Feature Trace**.
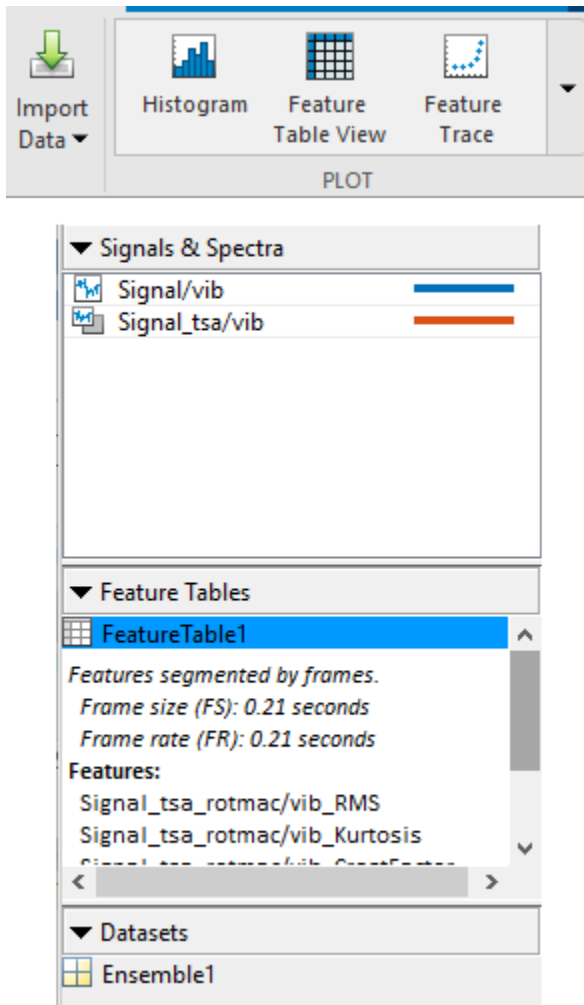
In the feature trace plot, all three features show an upward slope corresponding to the continuing degradation. The values of the features relative to one another have no meaning, as the features represent different metrics that are not normalized.
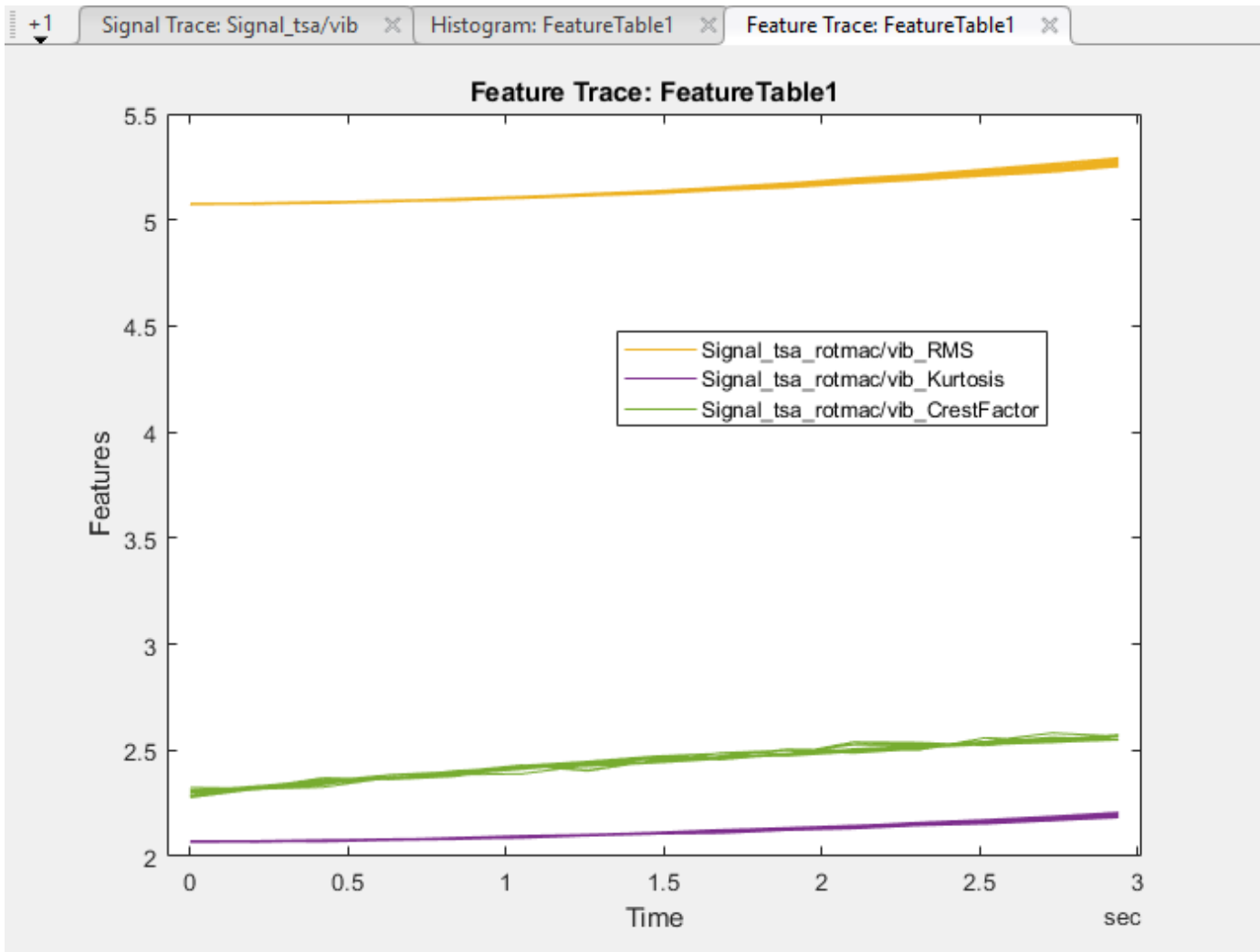
Extract Spectral Features

Spectral features generally work well when a defect results in a periodic oscillation. Extract spectral features from your TSA signal. Start by computing a power spectrum. To do so, select **Spectral Estimation > Power Spectrum**. Select the TSA signal and change **Algorithm** to `Welch's method`.
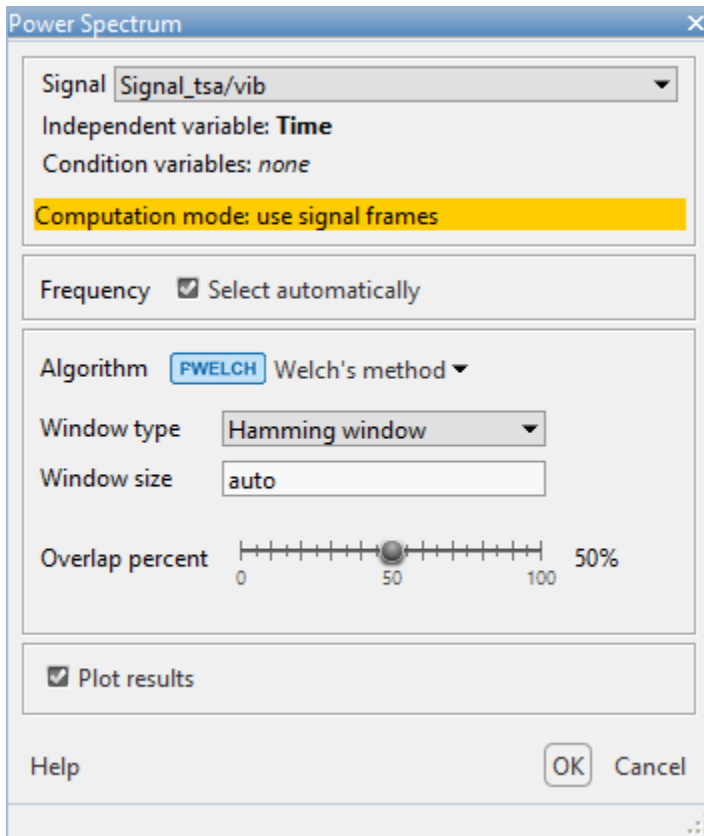
The spectrum for the first segment includes distinct peaks at around 500 Hz and 1540 Hz. The rotation speed is 1800 rpm or 30 Hz. The ratios between these peak frequencies are roughly 17 and 51, consistent with the gear ratios. The intervening peaks are additional harmonics of those frequencies.
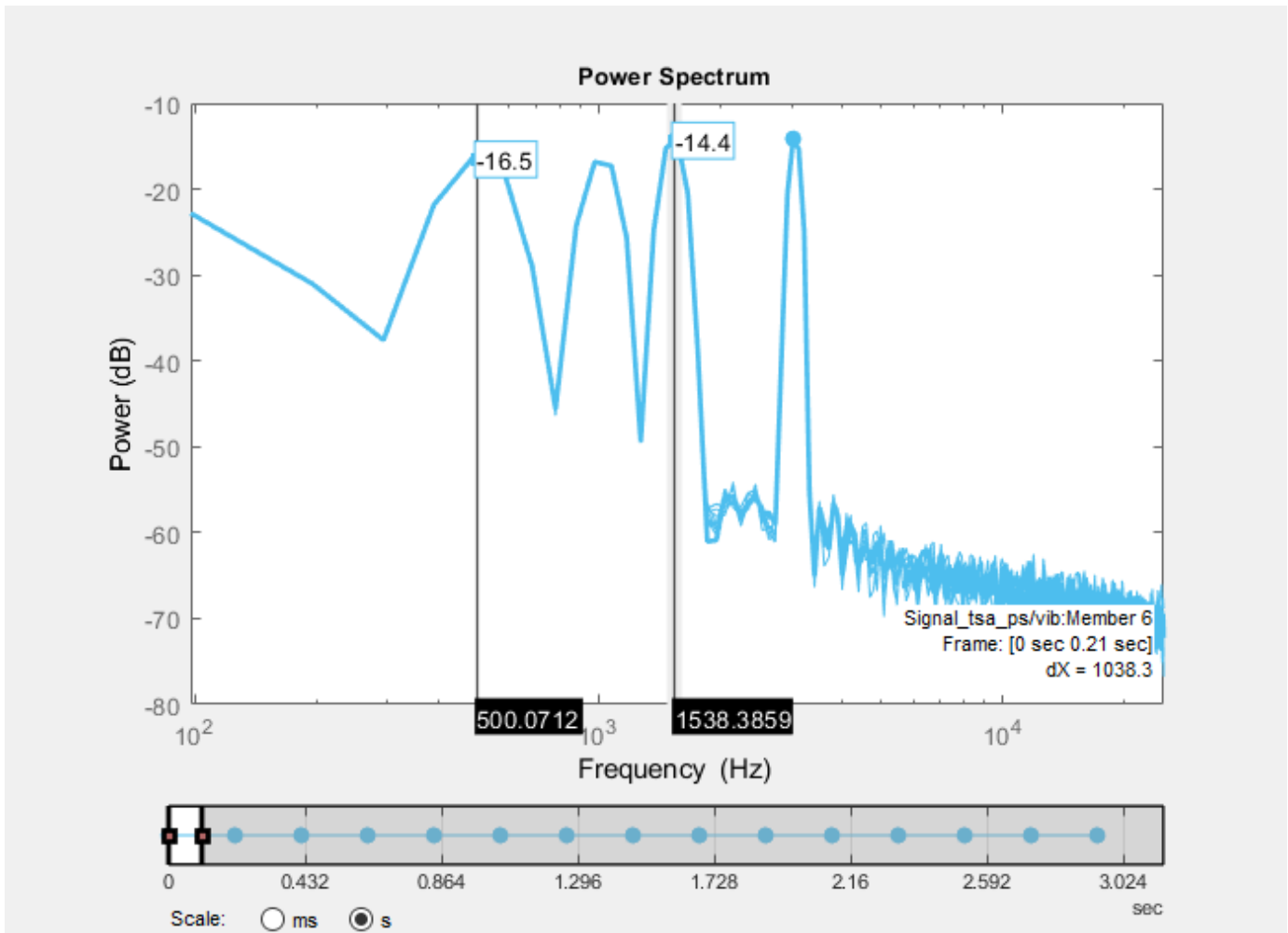
In the order and frequency domains, the segment spectra are superposed. The panner allows you to select multiple segments just as it does in the time domain. Set the panner to cover all the segments. As you expand the number of segments, the power increases at 300 Hz. This frequency corresponds to an order of 10 with respect to the 30 Hz rotation rate, and represents the increasing defect.

Power Spectrum

Signal_tsa_ps/vib:Member 10
Frame: [2.73 sec 2.94 sec]
dX = 24703

Scale: ○ ms ● s
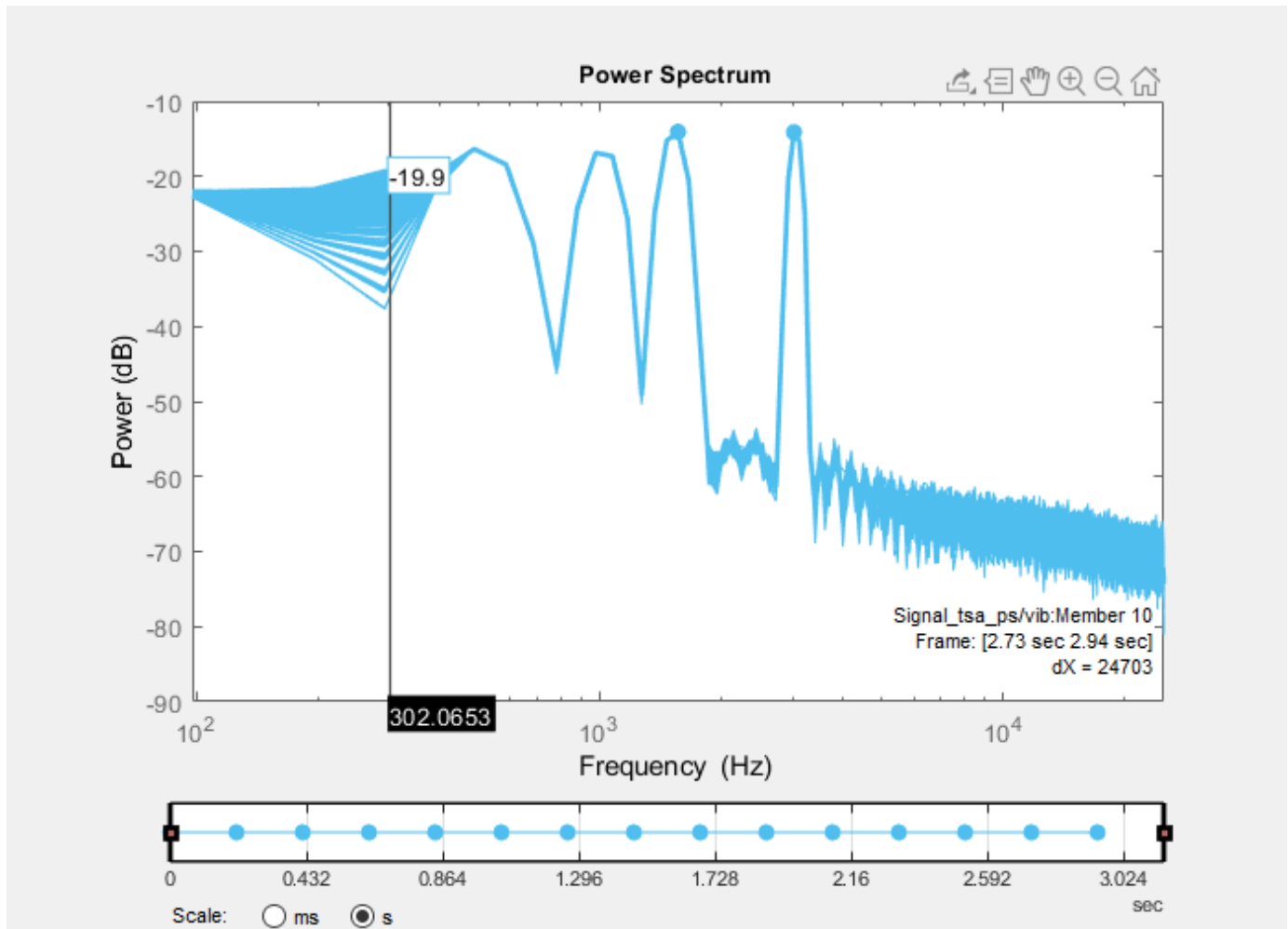
Extract spectral features. To do so, click **Spectral Features** and confirm that **Spectrum** is set to your power spectrum. Using the slider, limit the range to about 4000 Hz to bound the region to the peaks. The power spectrum plot automatically changes from a log to a linear scale and zooms in to the range you select.
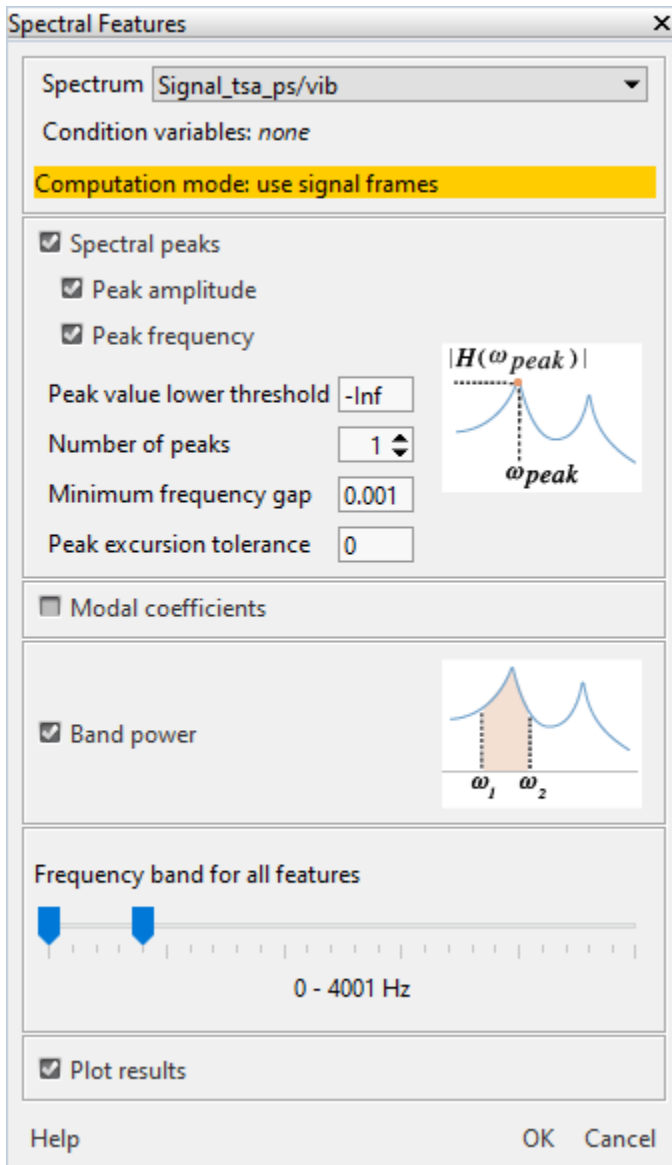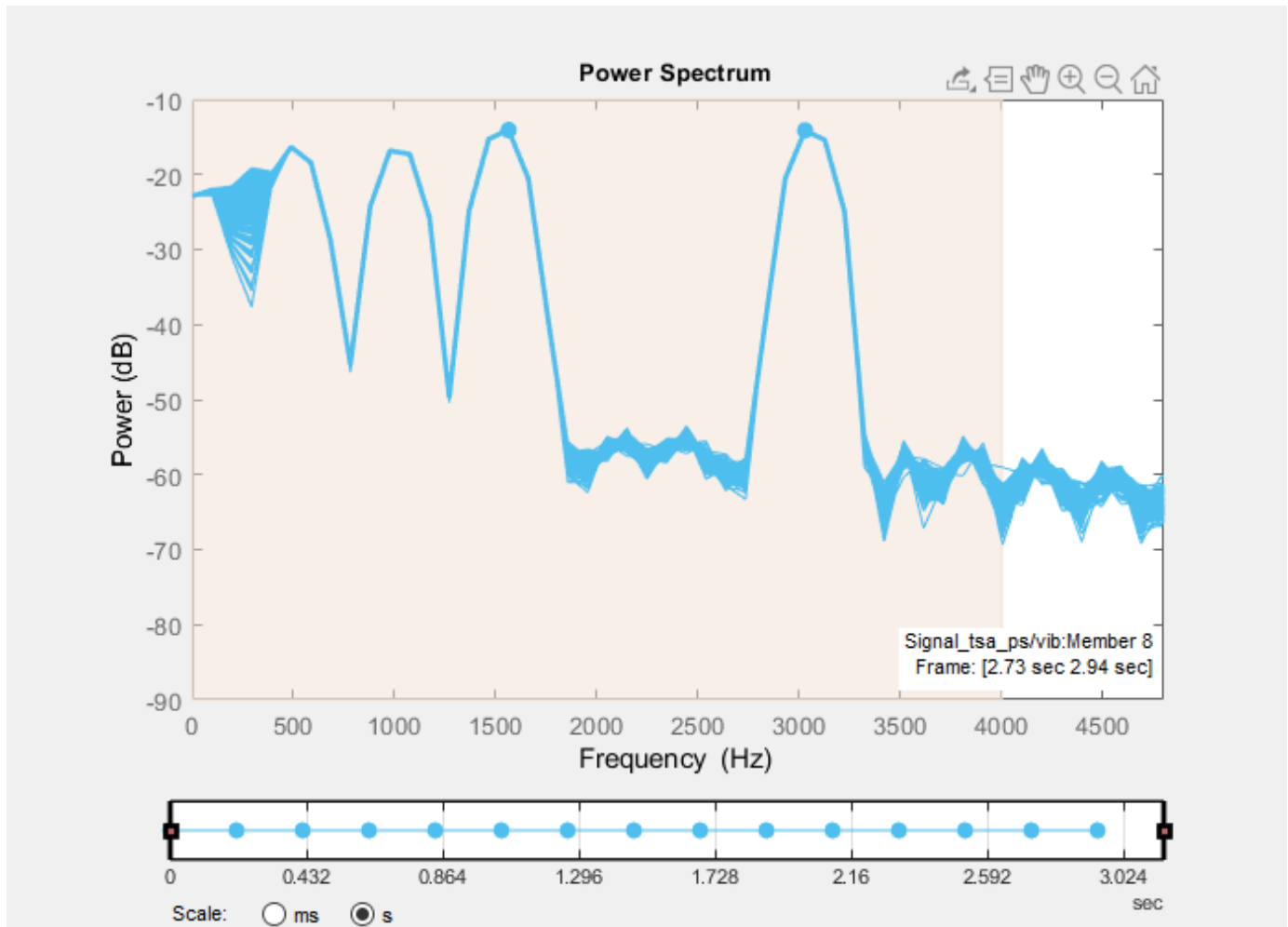
The resulting histogram plot now includes the spectral features.
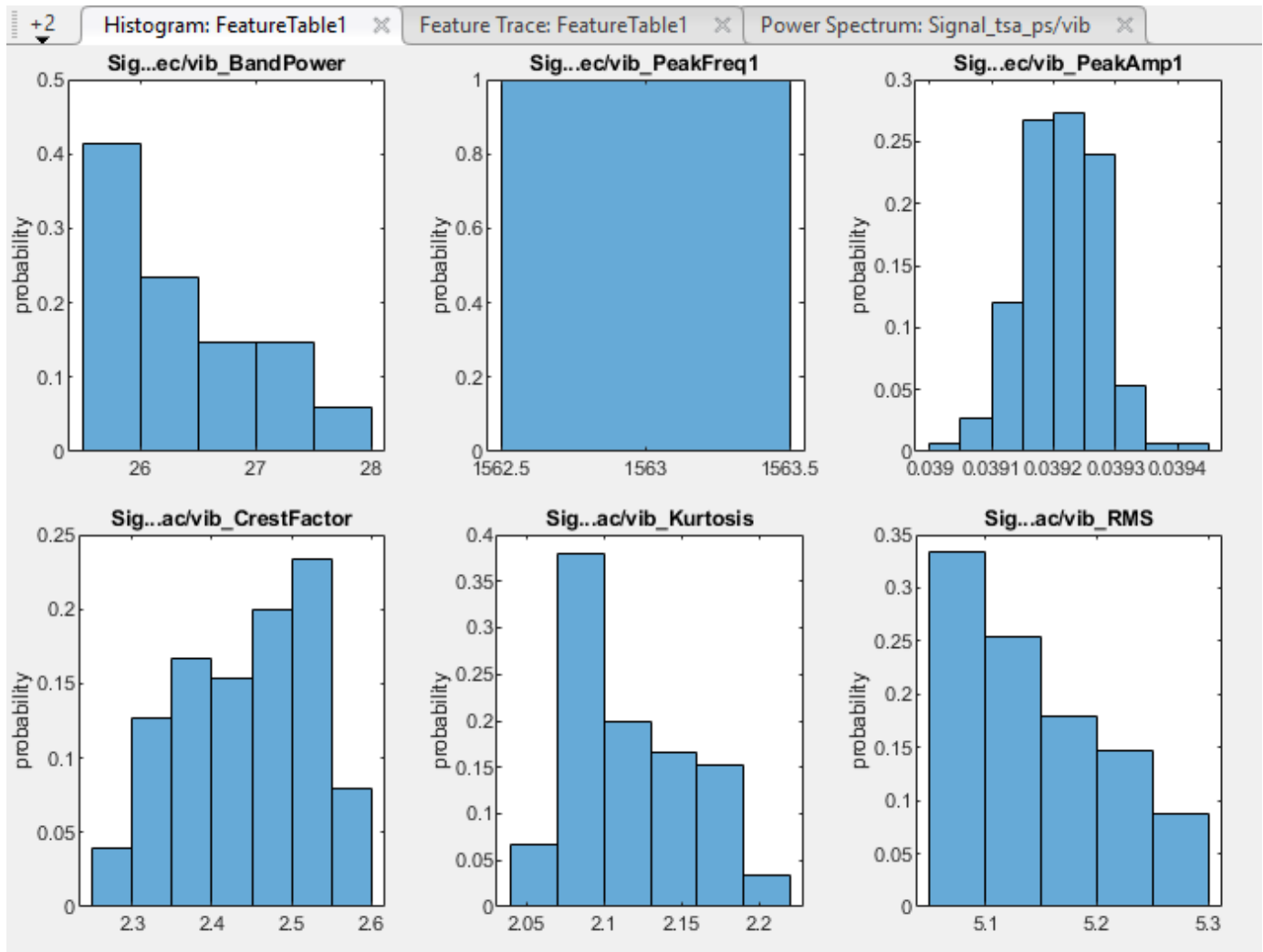
Plot the band-power feature trace to see how it compares with the all-segment power spectrum. Use **Select Features** to clear the other feature traces.
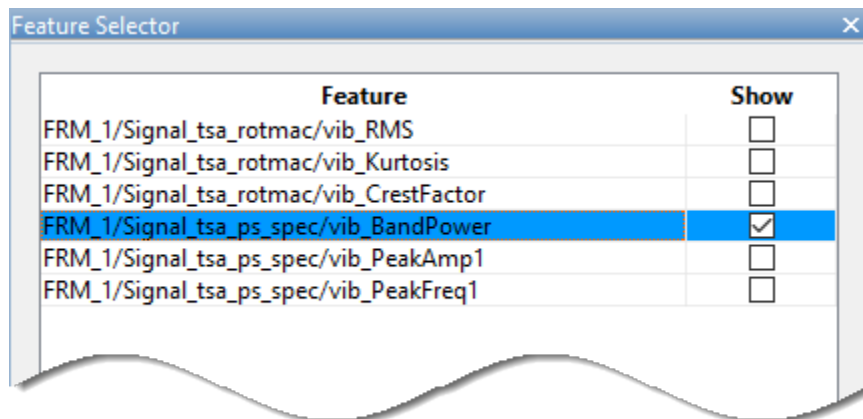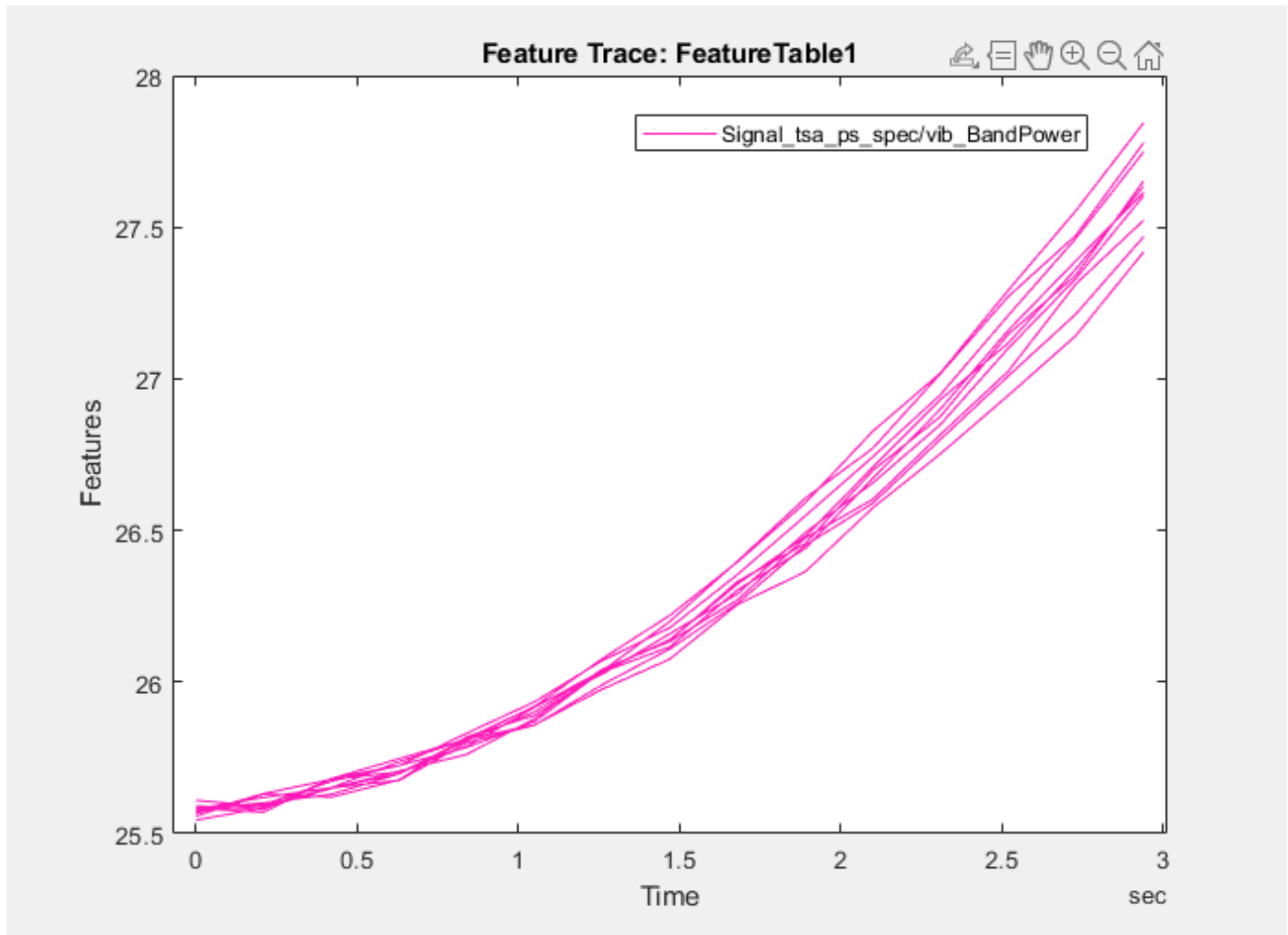
The band-power feature captures the progression of the defects in each machine. The traces of the other two spectral features do not track defect progression.
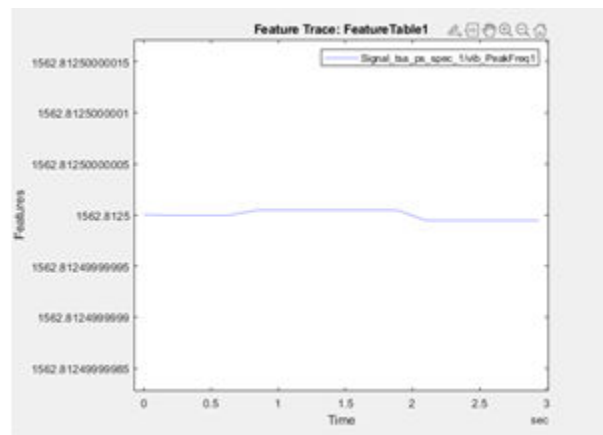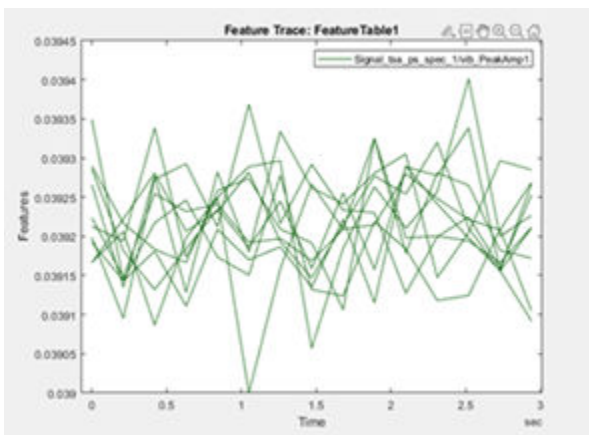
Rank Features with Prognostic Ranking Methods

Rank the features to see which ones perform best for predicting RUL. The app provides three prognostic ranking methods:

- Monotonicity characterizes the trend of a feature as the system evolves toward failure. As a system gets progressively closer to failure, a suitable condition indicator has a monotonic positive or negative trend. For more information, see `monotonicity`.

- Trendability provides a measure of similarity between the trajectories of a feature measured in multiple run-to-failure experiments. The trendability of a candidate condition indicator is defined as the smallest absolute correlation between measurements. For more information, see `trendability`.

- Prognosability is a measure of the variability of a feature at failure relative to the range between its initial and final values. A more prognosable feature has less variation at failure relative to the range between its initial and final values. For more information, see `prognosability`.

Click **Rank Features** and select `FeatureTable1`. Because you have no condition variables, the app defaults to the prognostic ranking technique `Monotonicity`.

Features Sorted by Importance

Feature	Monotonicity
Sig...ac/vib_RMS	1
Sig...ac/vib_CrestFactor	1
Sig...ec/vib_BandPower	1
Sig...ac/vib_Kurtosis	0.9857
Sig...ec/vib_PeakFreq1	0.2143
Sig...ec/vib_PeakAmp1	0.1429
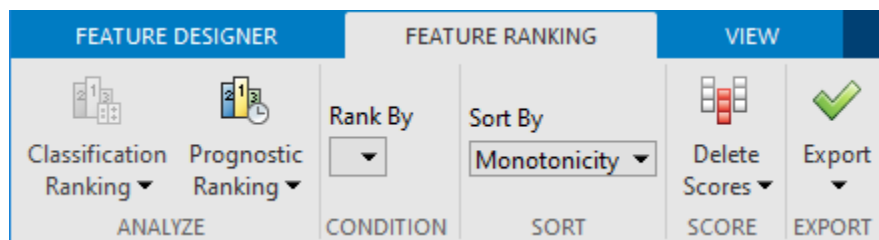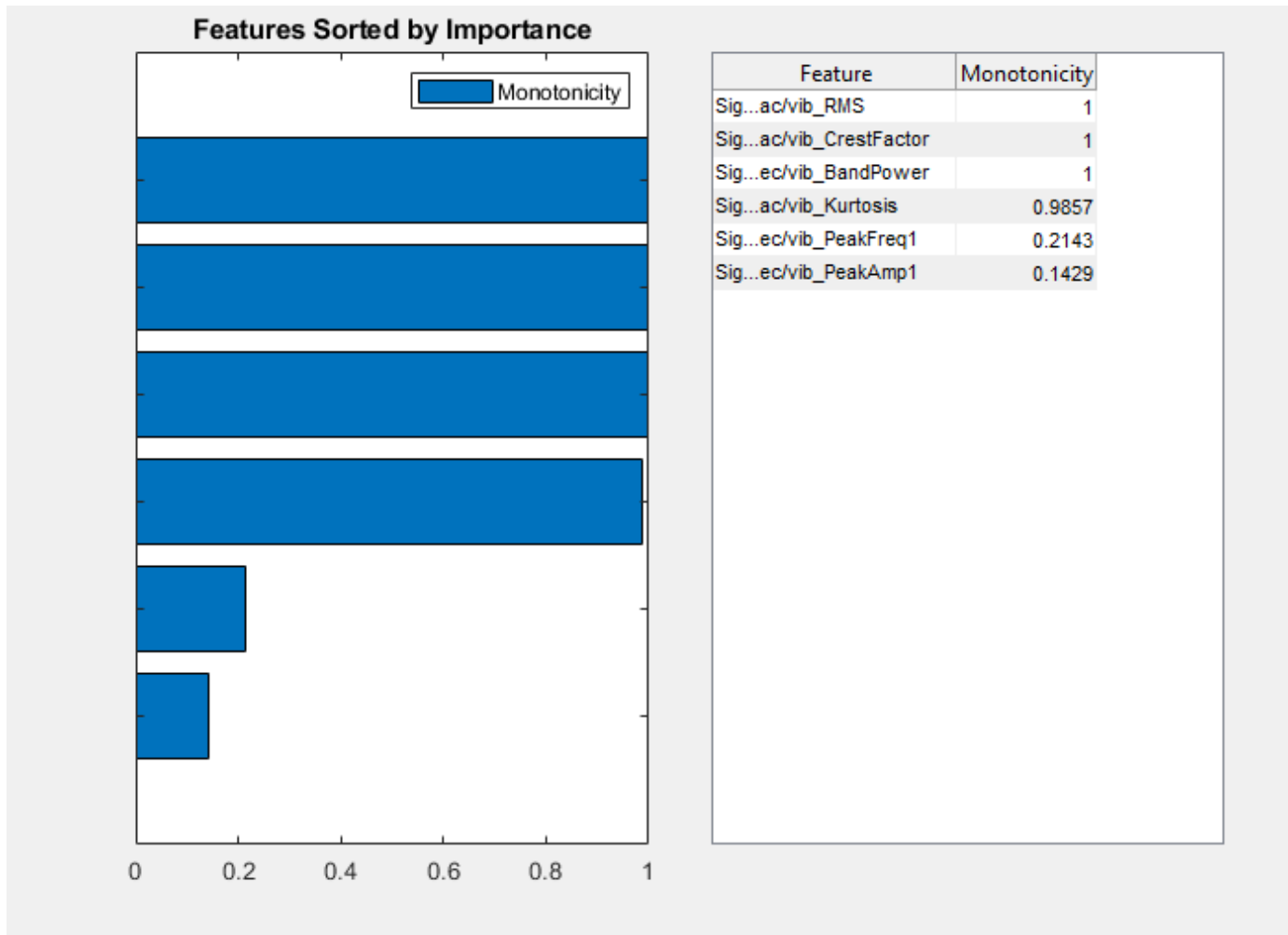
Four of the features score at or close to the maximum. Two features, `PeakAmp1` and `PeakFreq1`, have considerably lower scores.

Add rankings for the other two prognostic methods. Click **Prognostic Ranking** and select `Trendability`. Click **Apply** and then **Close Trendability**.

Repeat the previous step for `Prognosability`. The ranking plot now contains the results of all three ranking methods.

Features Sorted by Importance

Feature	Monotonicity	Trendability	Prognosability
Sig...ac/vib_RMS	1	0.9993	0.9362
Sig...ac/vib_CrestFactor	1	0.9897	0.9627
Sig...ec/vib_BandPower	1	0.9992	0.9360
Sig...ac/vib_Kurtosis	0.9857	0.9982	0.9389
Sig...ec/vib_PeakFreq1	0.2143	1	1
Sig...ec/vib_PeakAmp1	0.1429	9.4885e-04	0.3847
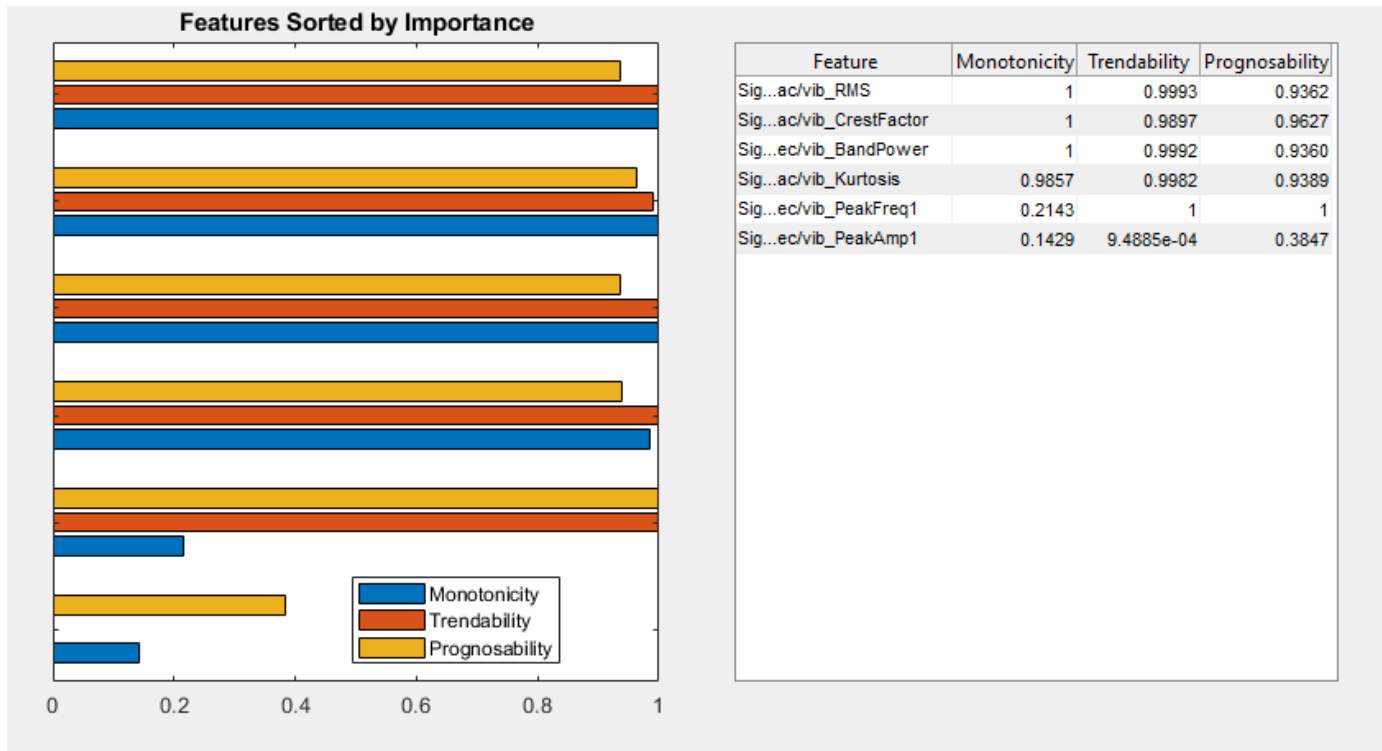
The ranking results are consistent with the feature traces plotted in "Extract Spectral Features" on page 7-75.

- The features that track the worsening fault have high scores for `Monotonicity`. These features also have high scores for the other two methods.
- `PeakFreq1`, which has the second lowest ranking `Monotonicity` score, has high scores for both `Trendability` and `Prognosability`. These high scores result from the close agreement among feature trajectories and low variability at the end of the simulation, where the fault is greatest.
- `PeakAmp1` has low scores for all rankings, reflecting both the insensitivity of this feature to defect progression and the variation in the machine values for this feature.

Since you have four features which scored well in all categories, choose these features as the feature set to move ahead with in an RUL algorithm.

See Also
Diagnostic Feature Designer | monotonicity | trendability | prognosability

More About
- "Models for Predicting Remaining Useful Life" on page 5-4
- "Feature Selection for Remaining Useful Life Prediction" on page 5-2
- "Identify Condition Indicators for Predictive Maintenance Algorithm Design"

Automatic Feature Extraction Using Generated MATLAB Code

In **Diagnostic Feature Designer**, you explore features interactively, using tools for signal processing, feature generation, and ranking. Once you determine which features perform best, you can generate code that reproduces your interactive computations and allows you to automate feature extraction on similar input data. Select among your features, computed variables, and ranking tables to specify what the code includes.

With the generated code, you can:

- Apply the code directly to a larger set of measurement data that includes more members and, therefore, increase the number of members in your feature set. Using more members improves model training in **Classification Learner**.
- Modify the function to suit your application. For example, you might add signal processing or features that are not available in **Diagnostic Feature Designer**.
- Incorporate portions of the function into another set of code that you are developing.

Generate a Function for Features

The simplest way to generate code for automatic feature extraction is to use the **Export** button in the **Feature Designer** tab and select **Generate Function for Features**.
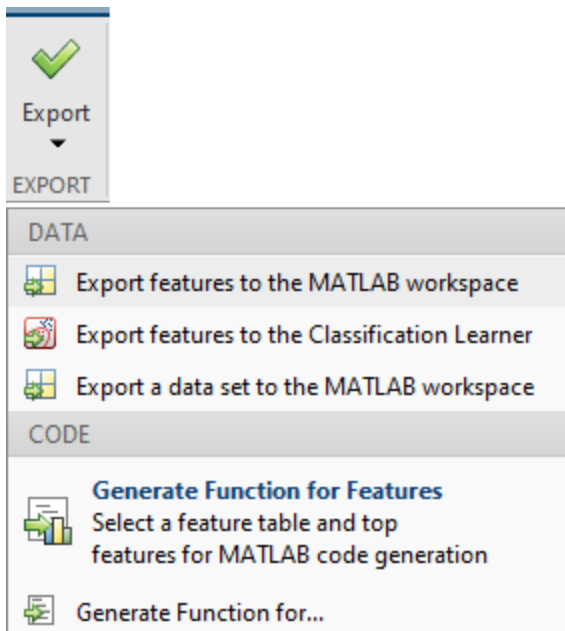
Your selection opens a set of options that allow you to specify the features to include from the feature table that you select. Code generation is possible for only one feature table at a time.

You can generate code for all your features or, if you have performed ranking, you can choose the number of top-ranked features to include. If you have performed ranking, you can also generate the function using the **Export** button in the **Feature Ranking** tab. The generated code includes the calculations for any computed signals or spectra that the feature requires. The code includes a preamble that summarizes the computations that the code performs.

```
%DIAGNOSTICFEATURES recreates results in Diagnostic Feature Designer.
%
% Input:
%  inputData: A table or a cell array of tables/matrices containing the
%  data as those imported into the app.
%
% Output:
%  featureTable: A table containing all features and condition variables.
%  outputTable: A table containing the computation results.
%
% This function computes signals:
%  Vibration_tsa/Data
%
% This function computes spectra:
%  Vibration_ps/SpectrumData
%
% This function computes features:
%  Vibration_sigstats/Mean
%  Vibration_tsa_rotmac/RMS
%  Vibration_tsa_rotmac/CrestFactor
%  Vibration_ps_spec/PeakAmp1
%  Vibration_ps_spec/PeakFreq1
%  Vibration_ps_spec/BandPower
%
% Organization of the function:
% 1. Compute signals/spectra/features
% 2. Extract computed features into a table
%
% Modify the function to add or remove data processing, feature generation
% or ranking operations.
```

Generate a Function for Specific Variables, Features, and Ranking Tables

If you want to customize your selections for code generation, use the **Export > Generate Function for...** option. With this option you can:

- Select code generation for any outputs that the app computes, including computed signals and spectra, ensemble statistics, and ranking tables.
- Filter your outputs so that you can choose among features with specific characteristics, such as input signal or variable-name text.

You can generate a function for features from one feature table at a time. If you are using frame-based processing, each feature table is associated with one frame policy, or combination of frame size and frame rate. Therefore, if you want to generate code for features computed with two different frame policies, or with both full-signal mode and a frame-based mode, you must generate a separate function for each feature table.

When you generate code for frame-based signals that depend on derived full signals, computations for the full signals appear also in the generated code.
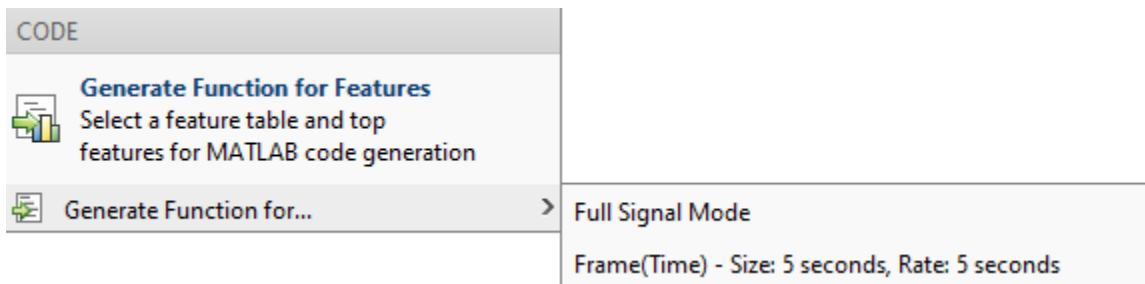
When you select **Export > Generate Function for...** and choose a feature source, a window containing candidate selection outputs opens. Each output row includes additional information on how the output was computed.

	Output	Method	Input	Analysis Type
☐	Vibration_sigstats/Mean	Mean	Vibration/Data	Feature Generation
☑	Vibration_sigstats/Skewness	Skewness	Vibration/Data	Feature Generation
☐	Vibration_sigstats/Std	Standard Deviation	Vibration/Data	Feature Generation
☐	Vibration_tsa/Data	TSA	Vibration/Data, Tacho/Data	Data Processing
☐	Tacho_rpm	RPM	Vibration/Data, Tacho/Data	Data Processing
☑	Vibration_tsa_rotmac/RMS	RMS	Vibration_tsa/Data	Feature Generation
☐	Vibration_tsa_rotmac/Kurtosis	Kurtosis	Vibration_tsa/Data	Feature Generation
☐	Vibration_tsa_rotmac/CrestFactor	Crest Factor	Vibration_tsa/Data	Feature Generation
☑	Vibration_ps/SpectrumData	Power Spectrum	Vibration_tsa/Data	Data Processing
☐	Vibration_ps_spec/PeakAmp1	Peak Amplitude	Vibration_ps/SpectrumData	Feature Generation
☐	Vibration_ps_spec/PeakFreq1	Peak Frequency	Vibration_ps/SpectrumData	Feature Generation
☐	Vibration_ps_spec/BandPower	Band Power	Vibration_ps/SpectrumData	Feature Generation
☐	FeatureRankingTable	T-Test	FeatureTable1	Ranking
☐	FeatureRankingTable	Bhattacharyya	FeatureTable1	Ranking

Details

Output: Vibration_ps/SpectrumData
Input: Vibration_tsa/Data

Code Will Be Generated For

Vibration_ps/SpectrumData
Vibration_sigstats/Skewness
Vibration_tsa_rotmac/RMS

In the figure, the skewness and RMS features and the power spectrum are selected. The **Details** pane displays the output and input for the most recently selected item. The **Code Will Be Generated For** pane contains your selections.

Along with the selection window, selecting **Export > Generate Function for...** opens the **Code Generation** tab, which contains filtering options.

For example, to select only features that are computed directly from the TSA signal, select **Input > Vibration_tsa/Data**, as the figure shows.

The filtered selection window displays the three features that are based directly on the TSA signal, along with the power spectrum, which is also based directly on the TSA signal.

	Output	Method	Input	Analysis Type
☑	Vibration_tsa_rotmac/RMS	RMS	Vibration_tsa/Data	Feature Generation
☐	Vibration_tsa_rotmac/Kurtosis	Kurtosis	Vibration_tsa/Data	Feature Generation
☐	Vibration_tsa_rotmac/CrestFactor	Crest Factor	Vibration_tsa/Data	Feature Generation
☑	Vibration_ps/SpectrumData	Power Spectrum	Vibration_tsa/Data	Data Processing

You can also filter on output, method, and analysis type. For example, if you want to generate code for a ranking table, select **Analysis Type > Ranking**.
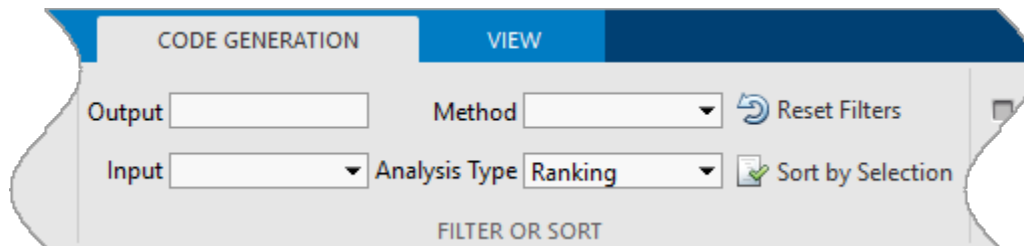
The selection list includes the T-Test and Bhattacharyya ranking tables.

☐	FeatureRankingTable	T-Test	FeatureTable1	Ranking
☑	FeatureRankingTable	Bhattacharyya	FeatureTable1	Ranking

In addition to the filter lists, you can filter for text within variable names by typing the text into the filter. For example, if you type `peak` into the **Output** filter, as the following figures show, the filtered list now includes the spectral features for peak amplitude and peak frequency. Text matching is case insensitive.
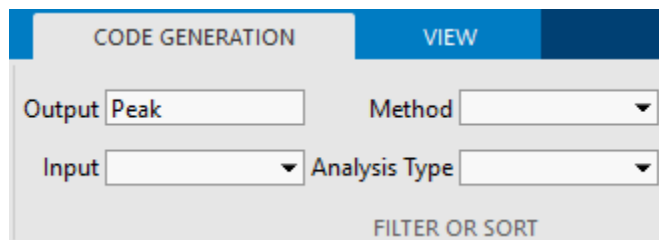
	Output	Method	Input	Analysis Type
☑	Vibration_ps_spec/PeakAmp1	Peak Amplitude	Vibration_ps/SpectrumData	Feature Generation
☐	Vibration_ps_spec/PeakFreq1	Peak Frequency	Vibration_ps/SpectrumData	Feature Generation

To clear all the filters, click **Reset Filters**.
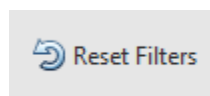
Each filtered view displays a subset of available outputs. To display all your selections together, click **Sort by Selection**.
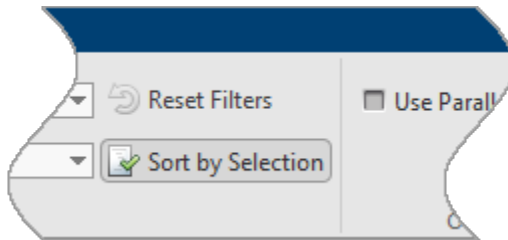
The items that you selected in the filtered views appear in one group at the top of the selection list.

	Output	Method	Input	Analysis Type
☑	Vibration_sigstats/Skewness	Skewness	Vibration/Data	Feature Generation
☑	Vibration_tsa_rotmac/RMS	RMS	Vibration_tsa/Data	Feature Generation
☑	Vibration_ps/SpectrumData	Power Spectrum	Vibration_tsa/Data	Data Processing
☑	Vibration_ps_spec/PeakAmp1	Peak Amplitude	Vibration_ps/SpectrumData	Feature Generation
☑	FeatureRankingTable	Bhattacharyya	FeatureTable1	Ranking
☐	Vibration_sigstats/Mean	Mean	Vibration/Data	Feature Generation
☐	Vibration_sigstats/Std	Standard Deviation	Vibration/Data	Feature Generation
☐	Vibration_tsa/Data	TSA	Vibration/Data, Tacho/Data	Data Processing
☐	Tacho_rpm	RPM	Vibration/Data, Tacho/Data	Data Processing
☐	Vibration_tsa_rotmac/Kurtosis	Kurtosis	Vibration_tsa/Data	Feature Generation
☐	Vibration_tsa_rotmac/CrestFactor	Crest Factor	Vibration_tsa/Data	Feature Generation
☐	Vibration_ps_spec/PeakFreq1	Peak Frequency	Vibration_ps/SpectrumData	Feature Generation
☐	Vibration_ps_spec/BandPower	Band Power	Vibration_ps/SpectrumData	Feature Generation
☐	FeatureRankingTable	T-Test	FeatureTable1	Ranking

To generate code for these items, click **Generate Function**. This action produces a function with a preamble that includes the following information.

```
%
% This function computes signals:
%  Vibration_tsa/Data
%
% This function computes spectra:
%  Vibration_ps/SpectrumData
%
% This function computes features:
%  Vibration_sigstats/Skewness
%  Vibration_tsa_rotmac/RMS
%  Vibration_ps_spec/PeakAmp1
%
% This function ranks computed feautres using algorithms:
%  Bhattacharyya
%
```

Even though the `Vibration_tsa/Data` is not selected in the sorted filter view, the function computes this signal because other signals that are selected require it.

Save and Use Generated Code

The app assigns the default name of `diagnosticFeatures` to the function. You can save the function as a file with this name, or rename the function and file name. To run the function, follow the syntax in the function line of the code. For example, suppose the function line is:

```
function [featureTable,outputTable] = diagnosticFeatures(inputData)
```

The two outputs of this function are a feature table that contains the features and condition variables, and an output table that contains all of the variables in the ensemble. The second output argument is optional. You can use this function on any input data that has the same input variables as the data that you originally imported into the app. For instance, suppose that your data is in `fullDataTable` and you need only a feature table `features`. Use:

```
features = diagnosticFeatures(fullDataTable)
```

If you originally imported individual tables and want to validate the code against the original data, you must combine the tables. For example, if you imported tables `t1`, `t2`, and `t3`, where each table represents a different ensemble member, first combine the tables into a single input table, and then run the generated function.

```
inputData = {t1,t2,t3};
features = diagnosticFeatures(inputData);
```

For an example of generating code and validating the code with the original data, see "Generate a MATLAB Function in Diagnostic Feature Designer" on page 7-94. For an example that applies code to a new dataset, see "Apply Generated MATLAB Function to Expanded Data Set" on page 7-101. For a description of generated code itself and how it performs its computations, see "Anatomy of App-Generated MATLAB Code" on page 7-114.

Change Options for Generated Code

When you develop your features, you have options for whether to use parallel computing and, during the import process, how to handle ensemble datastores. In some cases, you may want the generated code to use a different option than the option you used to compute your features in the app. For instance, you might want to invoke parallel computing in the code when you did not use that option originally. If you originally chose to store computation results in local memory when you imported an ensemble datastore, you might want your code to append results directly to the external files instead when you run the code. You can set these options for code generation in the **Code Generation** tab.
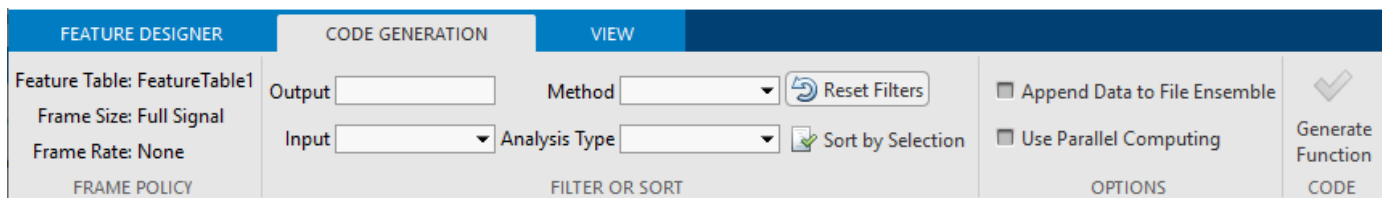
In the **Options** section, the following options are available under the following conditions:

- **Append Data to File Ensemble** — Available if your data source is a `fileEnsembleDatastore` or a `simulationEnsembleDatastore`
- **Use Parallel Computing** — Available if you have a license for Parallel Computing Toolbox

See Also
Diagnostic Feature Designer

More About

- "Generate a MATLAB Function in Diagnostic Feature Designer" on page 7-94
- "Apply Generated MATLAB Function to Expanded Data Set" on page 7-101
- "Export Feature Extraction Function and Simulink Model for Streaming Data" on page 7-179
- "Anatomy of App-Generated MATLAB Code" on page 7-114
- "Data Ensembles for Condition Monitoring and Predictive Maintenance" on page 1-2
- "Import Data into Diagnostic Feature Designer" on page 7-127

Generate a MATLAB Function in Diagnostic Feature Designer

In **Diagnostic Feature Designer**, you explore features interactively, using tools for signal processing, feature generation, and ranking. Once you select the set of the features that perform best, you can generate a MATLAB function that reproduces the calculations for those features. You can apply this function directly to a larger set of measurement data and increase the number of members in your feature set. You can also modify the function to suit your application, and incorporate part or all of the function into other code.

This example shows how to generate a MATLAB function to compute a set of features, and how to validate that function with the original data set.

The example assumes that you are familiar with ensemble data concepts and with basic operations in the app, such as data import, signal processing, and feature generation. For more information on these concepts and operations, see "Identify Condition Indicators for Predictive Maintenance Algorithm Design".

Import the Transmission Model Data

This example uses ensemble data generated from a transmission system model in "Using Simulink to Generate Fault Data" on page 1-25. Outputs of the model include:

- Vibration measurements from a sensor monitoring casing vibrations
- Tachometer sensor, which issues a pulse every time the shaft completes a rotation
- Fault code indicating the presence of a modeled fault

In your MATLAB command window, load the transmission data, which is stored in the table `dataTable`.

```
load dfd_Tutorial dataTable
```

`dataTable` is an ensemble table that contains 16 members, each of which represents one simulated transmission system. Each row of the table corresponds to one member. Each column of the table corresponds to one variable, such as `Vibration` or `FaultCode`. All ensemble members have the same variables.

Open **Diagnostic Feature Designer**.

```
diagnosticFeatureDesigner
```

In the app, import `dataTable`. During the import process, set the `faultCode` type to a condition variable. When the import is complete, the **Signals** list in the **Variables** pane displays the vibration and tacho data. For information on the import process, see "Import and Visualize Ensemble Data in Diagnostic Feature Designer".
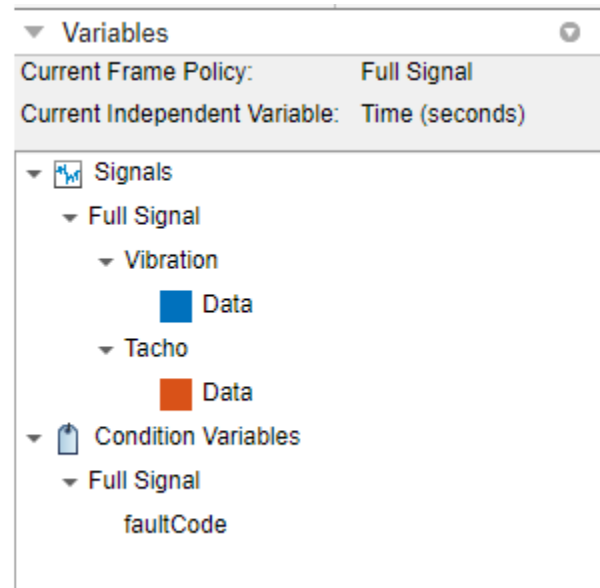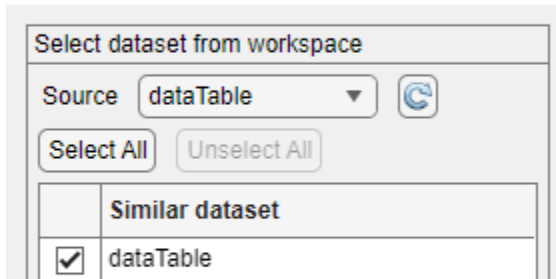
Compute a TSA Signal

Compute a time-synchronous average (TSA) signal from your vibration and tacho signals. To do so, first select `Vibration/Data` in the **Variables** pane. Then, in the **Feature Designer** tab, select **Filtering & Averaging > Time-Synchronous Averaging**. Set the parameters as shown in the following figure and click **OK**.

The new signal appears in the **Variables** pane.

For information on TSA signals, see `tsa`.

Extract Features from the TSA Signal

In the **Variables** pane, select the TSA signal. Then, in the **Feature Designer** tab, select **Time-Domain Features > Signal Features** to open the set of available signal features. Select the features for **Mean**, **Standard Deviation**, and **Kurtosis**.

View the feature values. In the **Variables** pane, select `FeatureTable1`. Then, in the plot gallery, click **Feature Table View**. These steps open a table containing the feature values for each member along with the condition variable `faultCode`.
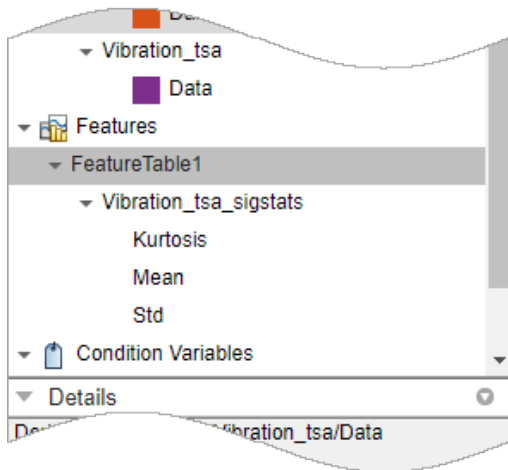
	faultCode	Vibration_tsa_sigstats/Kurtosis	Vibration_tsa_sigstats/Mean	Vibration_tsa_sigstats/Std
1	0	2.2516	0.0221	0.9995
2	1.0000	2.2526	-0.0273	0.9990
3	1.0000	2.2571	-0.4548	0.9963
4	1.0000	2.2526	0.4742	0.9990
5	1.0000	2.2529	0.3733	0.9990
6	1.0000	2.2526	-0.1418	0.9990
7	1.0000	2.2529	0.4064	0.9990
8	1.0000	2.2529	-0.4749	0.9992
		2.2586	0.0501	0.9954

Generate a MATLAB Function

Generate a MATLAB function that reproduces the calculations for these features. In the **Feature Designer** tab, select **Export > Generate Function for Features**.
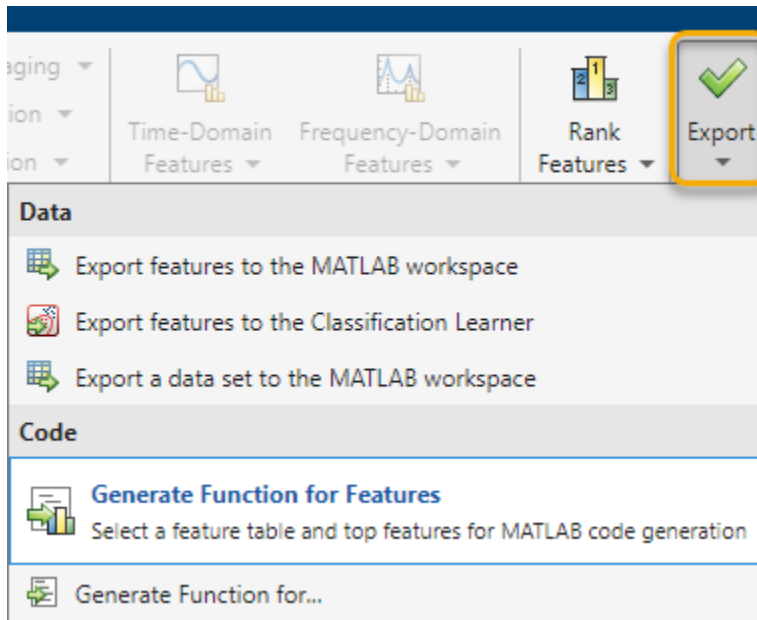
Your selection opens a dialog box that allows you to specify the feature table and the features. Because you have performed no ranking, the app configures the dialog box to export all three features.
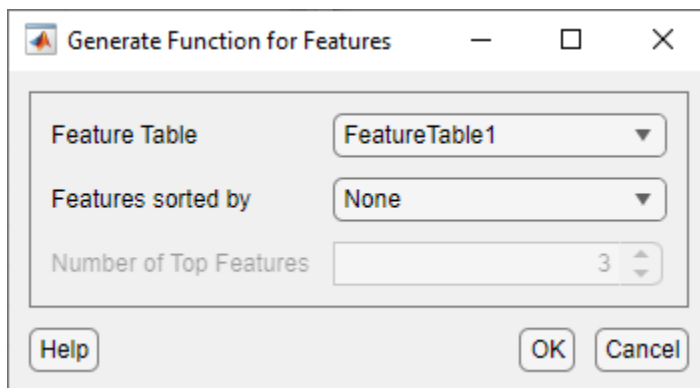
When you click **OK**, a function script opens in the MATLAB editor that begins with the following lines.

```
function [featureTable,outputTable] = diagnosticFeatures(inputData)
%DIAGNOSTICFEATURES recreates results in Diagnostic Feature Designer.
%
% Input:
%  inputData: A table or a cell array of tables/matrices containing the
%  data as those imported into the app.
%
% Output:
%  featureTable: A table containing all features and condition variables.
%  outputTable: A table containing the computation results.
%
% This function computes signals:
%  Vibration_tsa/Data
%
```

```
% This function computes features:
%  Vibration_tsa_sigstats/Kurtosis
%  Vibration_tsa_sigstats/Mean
%  Vibration_tsa_sigstats/Std
%
% Organization of the function:
% 1. Compute signals/spectra/features
% 2. Extract computed features into a table
%
% Modify the function to add or remove data processing, feature generation
% or ranking operations.
```

The preamble describes what the function computes. In this case, the function computes the features along with the TSA processing that produced the signal source for these features. Save the script as `diagnosticFeatures.m`.

For more information on the code content, see "Anatomy of App-Generated MATLAB Code" on page 7-114.

Validate Function with the Original Data

Run the function using `dataTable` to create a new feature table `featuretable`.

```
featuretable = diagnosticFeatures(dataTable)
```

Compare the first eight feature values to the corresponding feature values in the app. At the level of the displayed precision, the values are identical.

```
16×4 table

  faultCode    Vibration_tsa_stats/Kurtosis    Vibration_tsa_stats/Mean    Vibration_tsa_stats/
  _____    _____    _____    _____

      0                 2.2516                        0.022125                    0.99955
      1                 2.2526                       -0.027311                    0.999
      1                 2.2571                       -0.45475                     0.99629
      1                 2.2526                        0.47419                     0.999
      1                 2.2529                        0.37326                     0.999
      1                 2.2526                       -0.14185                     0.999
      1                 2.2529                        0.40644                     0.999
      1                 2.2529                       -0.47485                     0.99915
```

See Also
`readMemberData` | `read` | `workspaceEnsemble` | `writeToLastMemberRead` | `readFeatureTable` | `tsa`

More About
- "Identify Condition Indicators for Predictive Maintenance Algorithm Design"
- "Automatic Feature Extraction Using Generated MATLAB Code" on page 7-86
- "Apply Generated MATLAB Function to Expanded Data Set" on page 7-101
- "Anatomy of App-Generated MATLAB Code" on page 7-114

Apply Generated MATLAB Function to Expanded Data Set

This example shows how to use a small set of measurement data in **Diagnostic Feature Designer** to develop a feature set, generate and run code to compute those features for a larger set of measurement data, and compare model accuracies in **Classification Learner**.

Using a smaller data set at first has several advantages, including faster feature extraction and cleaner visualization. Subsequently generating code so that you can automate the feature computations with an expanded set of members increases the number of feature samples and therefore improves classification model accuracy.

The example, based on "Analyze and Select Features for Pump Diagnostics" on page 7-24, uses the pump fault data from that example and computes the same features. For more detailed information about the steps and the rationale for the feature development operations using the pump-fault data in this example, see "Analyze and Select Features for Pump Diagnostics" on page 7-24. This example assumes that you are familiar with the layout and operations in the app. For more information on working with the app, see the three-part tutorial in "Identify Condition Indicators for Predictive Maintenance Algorithm Design".

Load Data and Create Reduced Data Set

Load the data set `pumpData`. `pumpData` is a 240-member ensemble table that contains simulated measurements for flow and pressure. `pumpData` also contains categorical fault codes that represent combinations of three independent faults. For example, a fault code of `0` represents data from a system with no faults. A fault code of `111` represents data from a system with all three faults.

```
load savedPumpData pumpData
```

View a histogram of original fault codes. The histogram shows the number of ensemble members associated with each fault code.

```
fcCat = pumpData{:,3};
histogram(fcCat)
title('Fault Code Distribution for Full Pump Data Set')
xlabel('Fault Codes')
ylabel('Number of Members')
```

Fault Code Distribution for Full Pump Data Set
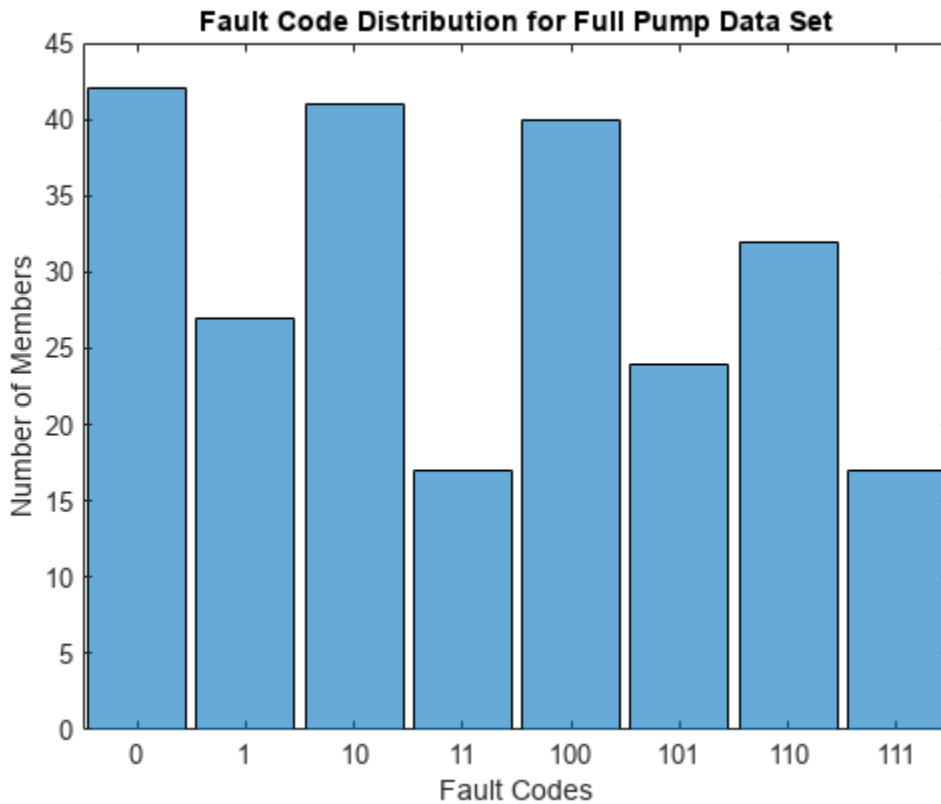
Create a subset of this data set that contains 10% of the data, or 24 members. Because simulation data is often clustered, generate a randomized index with which to select the members. For the purposes of this example, first use `rng` to create a repeatable random seed.

```
rng('default')
```

Compute a randomized 24-element index vector `idx`. Sort the vector so that the indices are in order.

```
pdh = height(pumpData);
nsel = 24;
idx = randi(pdh,nsel,1);
idx = sort(idx);
```

Use `idx` to select member rows from `pumpData`.

```
pdSub = pumpData(idx,:);
```

View a histogram of the fault codes in the reduced data set.

```
fcCatSub = pdSub{:,3};
histogram(fcCatSub)
title('Fault Code Distribution for Reduced Pump Data Set')
xlabel('Fault Codes')
ylabel('Number of Members')
```

Fault Code Distribution for Reduced Pump Data Set
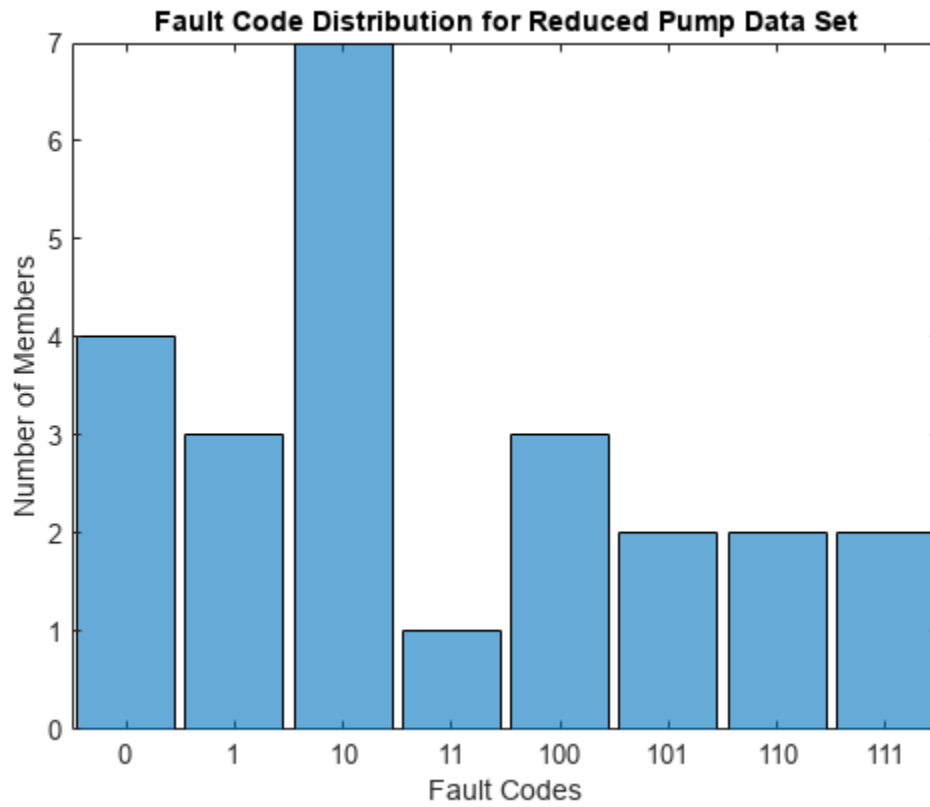
All the fault combinations are represented.

Import Reduced Data Set into Diagnostic Feature Designer

Open **Diagnostic Feature Designer** by using the `diagnosticFeatureDesigner` command. Import `pdSub` into the app.
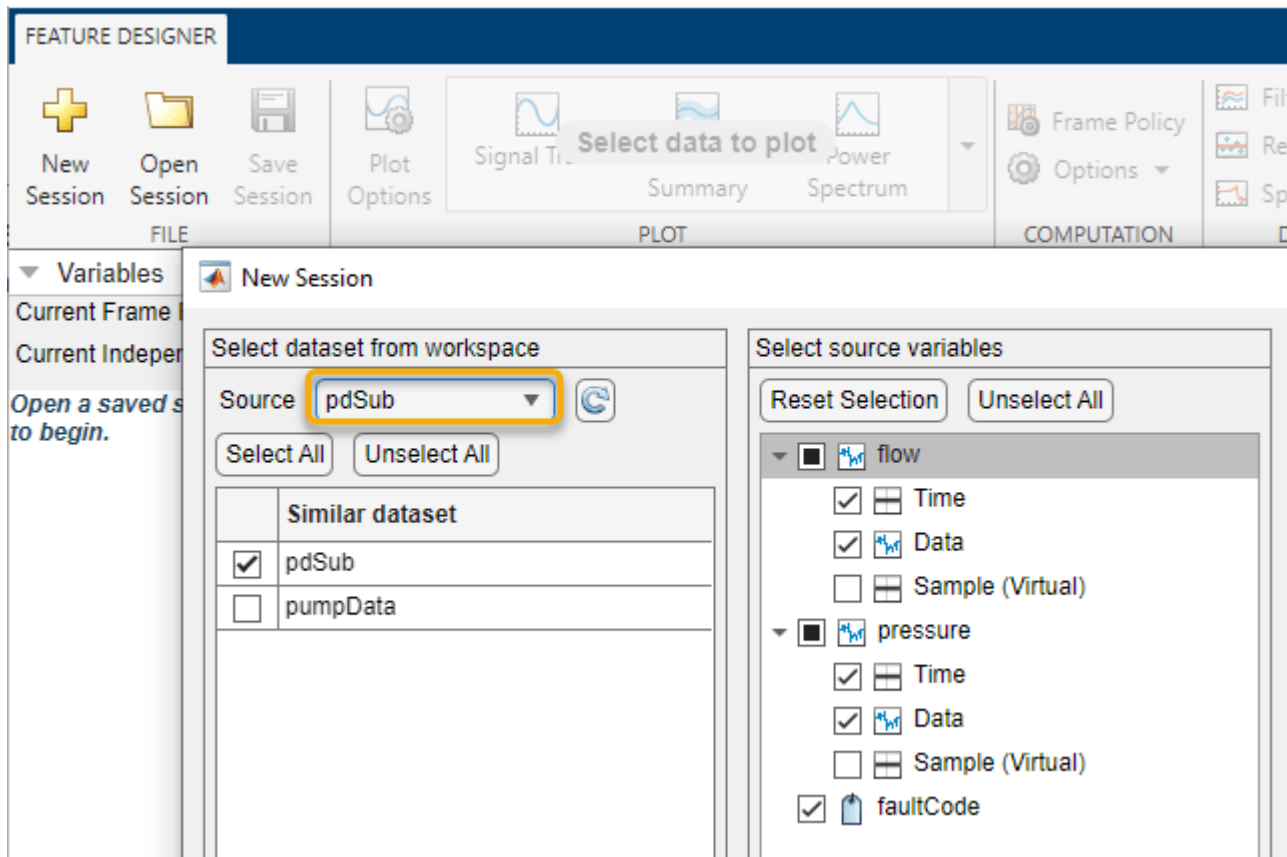
Extract Time-Domain Features

Extract the time-domain signal features from both the `flow` and `pressure` signals. For each signal, first, select the signal. Then, in the **Feature Designer** tab, select **Time Domain Features > Signal Features** and select all features.
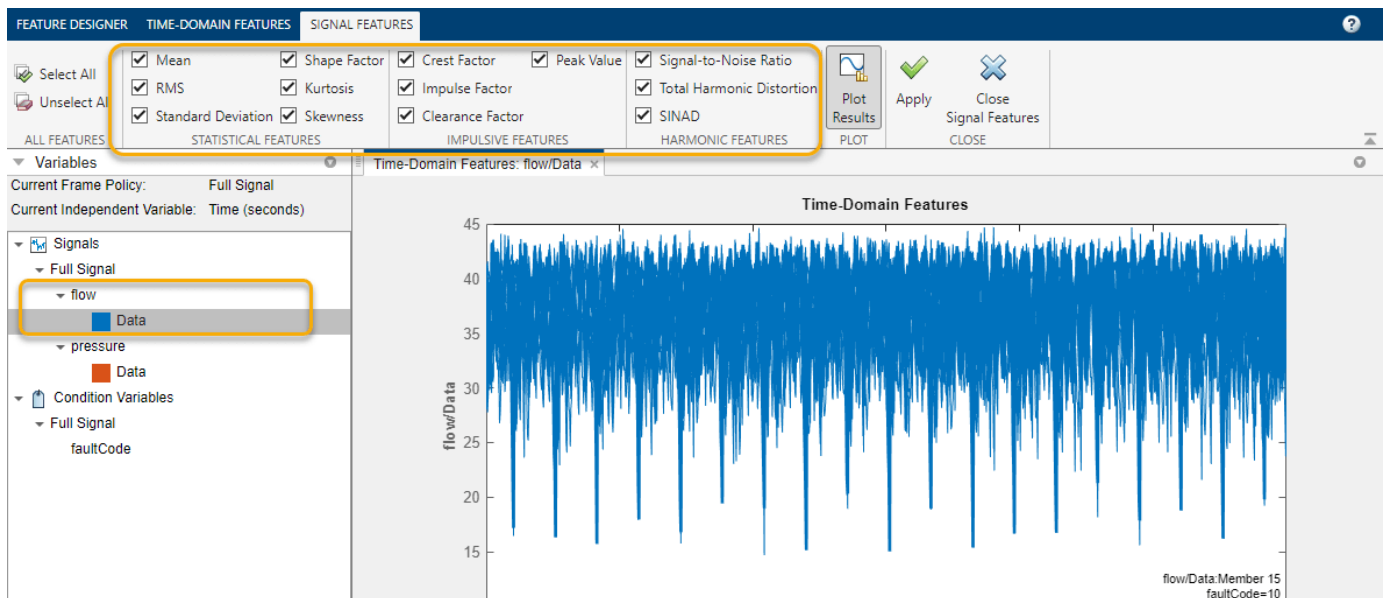
Extract Frequency Domain Features

As "Analyze and Select Features for Pump Diagnostics" on page 7-24 describes, computing the frequency spectrum of the flow highlights the cyclic nature of the flow signal. Estimate the frequency spectrum by selecting **Spectral Estimation > Autoregressive model** and using the options shown for both `flow` and `pressure`.
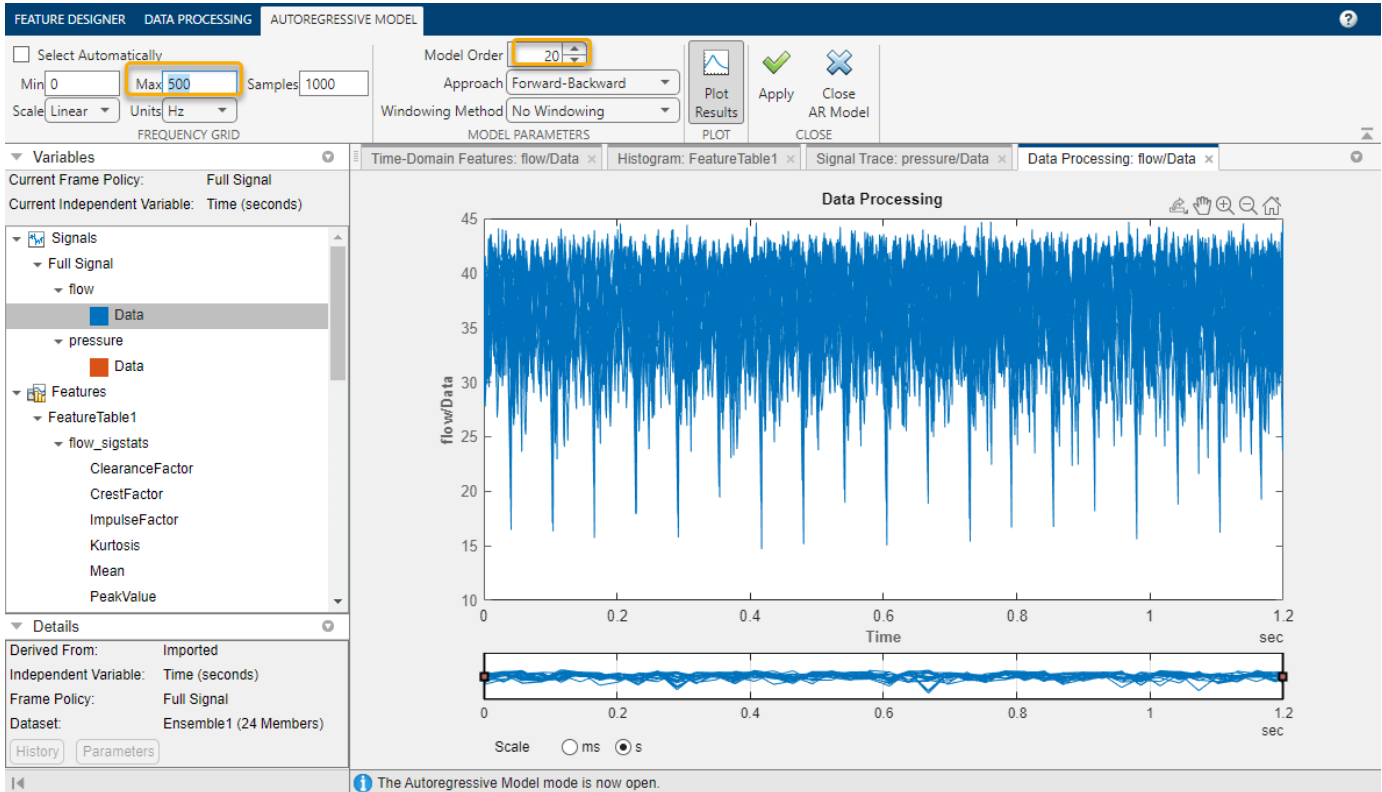
From the derived `flow` and `pressure` spectra, compute spectral features in the band 23–250 Hz, using the options shown.
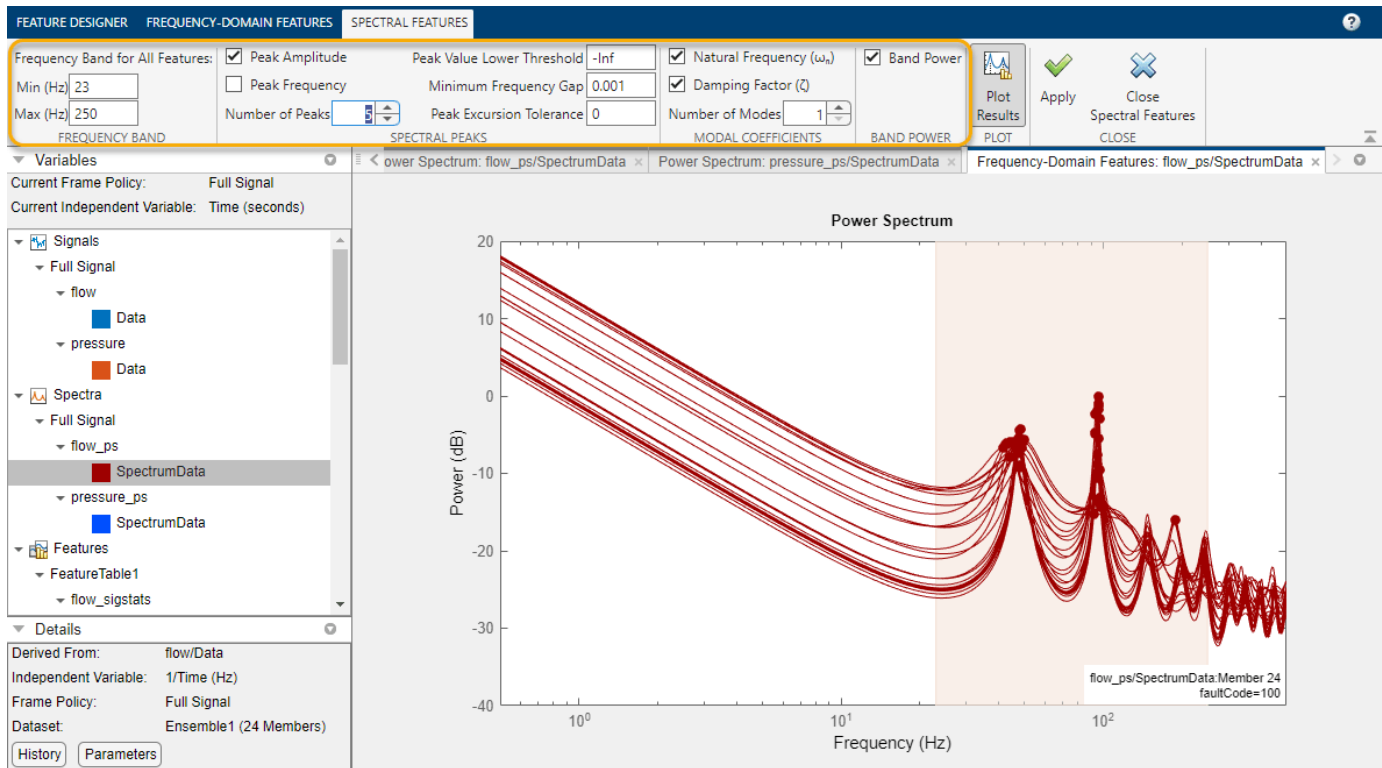
Rank Features

Rank your features by selecting **Rank Features > FeatureTable1**. Because `faultCode` contains multiple possible values, the app defaults to the `One-Way ANOVA` ranking method.

Features Sorted by Importance

Feature	One-way ANOVA
pressure_ps_spec/PeakAmp5	15.4364
flow_sigstats/RMS	13.2686
pressure_sigstats/RMS	13.2353
pressure_sigstats/Mean	13.1782
flow_ps_spec/PeakAmp3	12.6677
pressure_ps_spec/PeakAmp1	11.9333
flow_sigstats/Mean	11.8681
pressure_ps_spec/PeakAmp4	11.4467
flow_ps_spec/PeakAmp4	10.4217
pressure_sigstats/CrestFactor	10.4063
pressure_sigstats/ImpulseFactor	10.3924
pressure_sigstats/ClearanceF...	10.3853
pressure_sigstats/PeakValue	10.3025
flow_ps_spec/PeakAmp5	10.1433
flow_sigstats/THD	10.0790
flow_ps_spec/PeakAmp1	8.9746
flow_sigstats/CrestFactor	8.6567
flow_sigstats/ImpulseFactor	8.3026
flow_sigstats/ClearanceFactor	8.0534
flow_ps_spec/PeakAmp2	8.0166

Export Features to Classification Learner

Export the features set to **Classification Learner** so that you can train a classification model. In the **Feature Ranking** tab, click **Export > Export Features to the Classification Learner**. Select the top 15 features by selecting **Select top features** and typing 15.
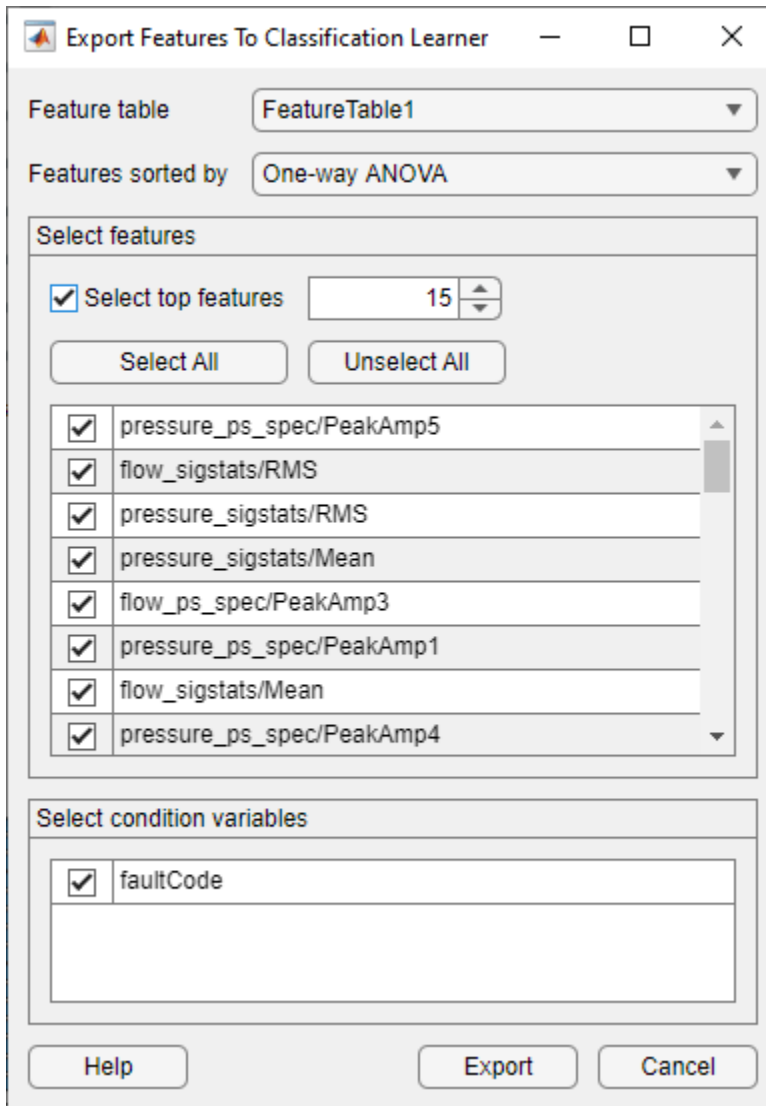
Train Models in Classification Learner

Once you click **Export**, **Classification Learner** opens a new session using the data you exported.
Start the session by clicking **Start Session**.

Train all available models by clicking **All i**n the **Classification Learner** tab, and then **Train All.**

Classification Learner trains all the models and initially sorts then by name. Use the **Sort by** menu to sort by `Accuracy (Validation)`. For this session, the highest scoring model, KNN, has an accuracy of about 63%. Your results may vary. Click **Confusion Matrix** to view the confusion matrix for this model.
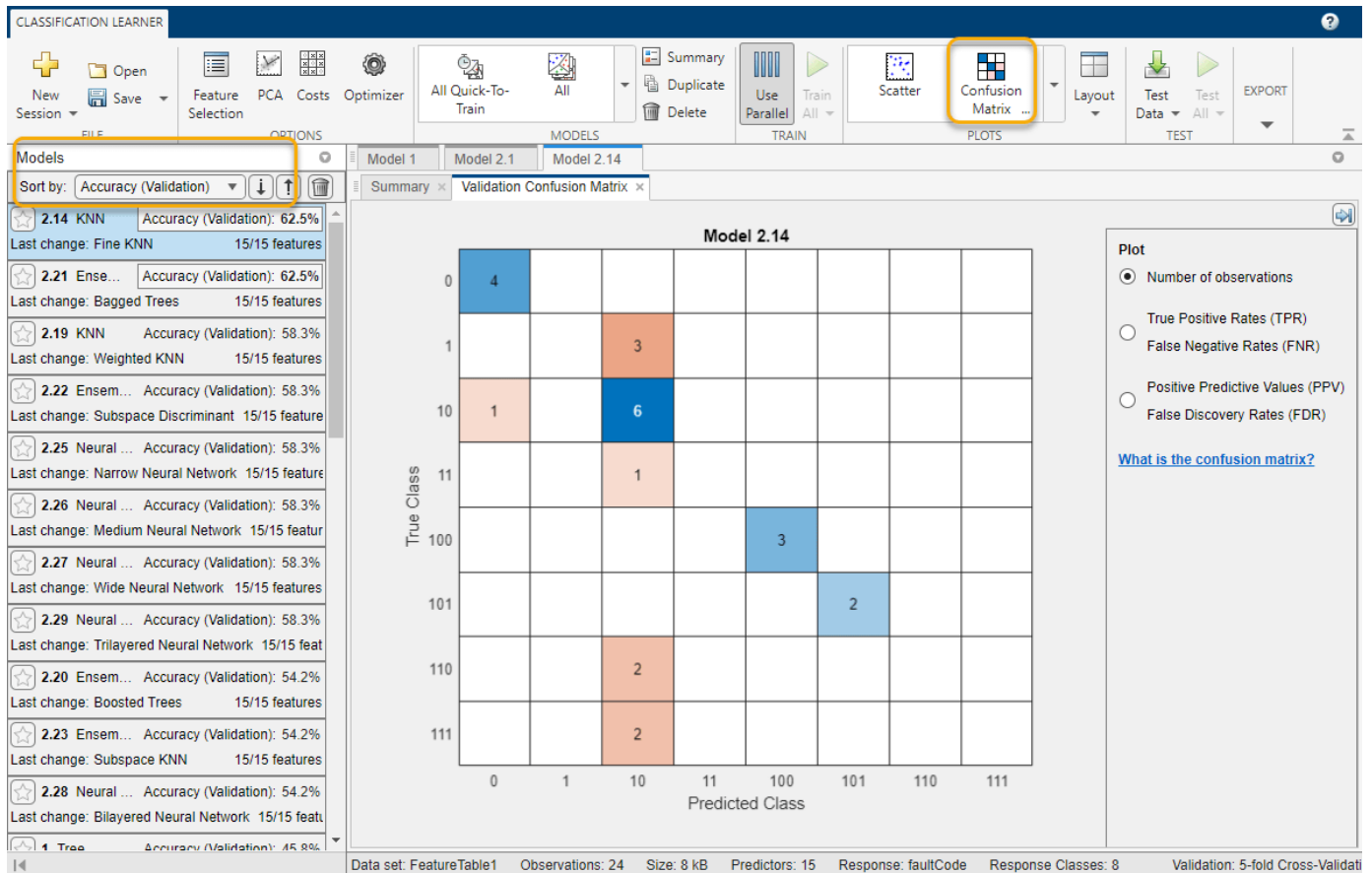
Generate Code to Compute Feature Set

Now that you have completed your interactive feature work with a small data set, you can apply the same computations to the full data set using generated code. In **Diagnostic Feature Designer**, generate a function to calculate the features. To do so, in the **Feature Ranking Tab,** select **Export > Generate Function for Features**. Select the same 15 features that you exported to **Classification Learner.**
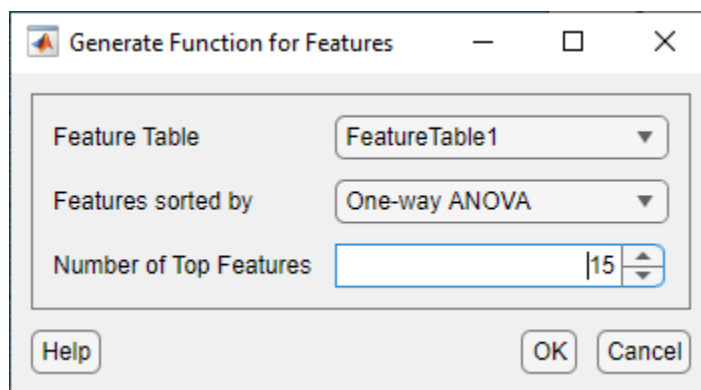
When you click **OK**, a function appears in the editor.

```
function [featureTable,outputTable] = diagnosticFeatures(inputData)
%DIAGNOSTICFEATURES recreates results in Diagnostic Feature Designer.
%
% Input:
%  inputData: A table or a cell array of tables/matrices containing the
%  data as those imported into the app.
%
% Output:
%  featureTable: A table containing all features and condition variables.
%  outputTable: A table containing the computation results.
%
% This function computes spectra:
%  flow_ps/SpectrumData
%  pressure_ps/SpectrumData
%
% This function computes features:
%  flow_sigstats/Mean
%  flow_sigstats/RMS
%  flow_sigstats/THD
%  pressure_sigstats/ClearanceFactor
%  pressure_sigstats/CrestFactor
%  pressure_sigstats/ImpulseFactor
%  pressu
```

Save the function to your local folder as `diagnosticFeatures`.

Apply the Function to Full Data Set

Execute `diagnosticFeatures` with the full `pumpData` ensemble to get the 240-member feature set. Use the following command.

```
feature240 = diagnosticFeatures(pumpData);
```

`feature240` is a 240-by-16 table. The table includes the condition variable `faultCode` and the 15 features.

Train Models in Classification Learner with Larger Feature Table

Train classification models again in **Classification Learner**, using `feature240` this time. Open a new session window using the following command.

```
classificationLearner
```

In the **Classification Learner** window, click **New Session > From Workspace**. In the **New Session** window, in **Data Set > Data Set Variable**, select `feature240`.

Repeat the steps you performed with the 24-member data set. Start the session and then train all models. Sort the models by `Accuracy (Validation)`. In this session, the highest scoring model is `Bagged Trees`, with an accuracy of about 73%, roughly 10% higher than the model computed using the reduced data. Again, your results may vary, but they should still reflect the increase in best accuracy
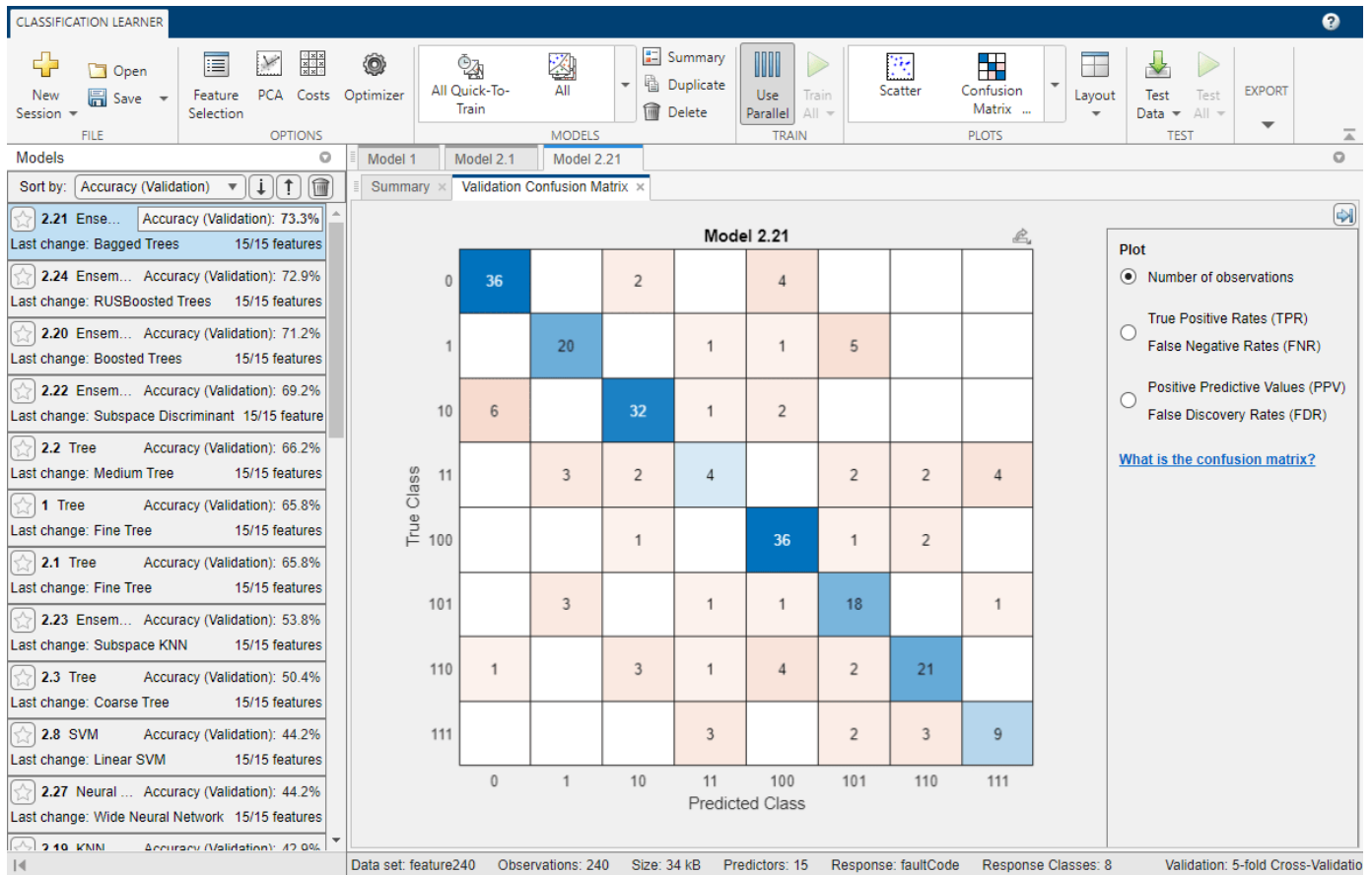
For this session, the highest model accuracy, achieved by both `Bagged Trees` and `RUSBoosted Trees`, is around 80%. Again, your results may vary, but they should still reflect the increase in best accuracy.

Anatomy of App-Generated MATLAB Code

With **Diagnostic Feature Designer**, you can generate MATLAB code that automates the computations for the features and variables you choose. This generated code accepts any ensemble data that is configured the same way as the ensemble data you imported into the app, and generates a new feature table, as well as computed signals, spectra, and ranking tables, that can be used for feature analysis or model training. The code replicates various options that you set within the app, and can perform:

- Full-signal computation of signals and features for each member
- Ranking for features
- Ensemble-level computations for characterizing ensemble behavior
- Parallel processing
- Processing of segmented signals, also known as frame-based processing

Ensemble management is a fundamental component of the generated code. For information about data ensembles and ensemble variable types, see "Data Ensembles for Condition Monitoring and Predictive Maintenance" on page 1-2.

You can compare this functional description with actual code by generating your own code in the app. For more information, see "Automatic Feature Extraction Using Generated MATLAB Code" on page 7-86. For an example showing how to generate code, see "Generate a MATLAB Function in Diagnostic Feature Designer" on page 7-94. For an example that provides a setup for frame-based code generation, see "Perform Prognostic Feature Ranking for a Degrading System Using Diagnostic Feature Designer" on page 7-65.

Basic Function Flow

The figure illustrates the basic functional flow of generated code. In this diagram, the function returns both features and derived variables, uses serial processing, and operates on full signals.
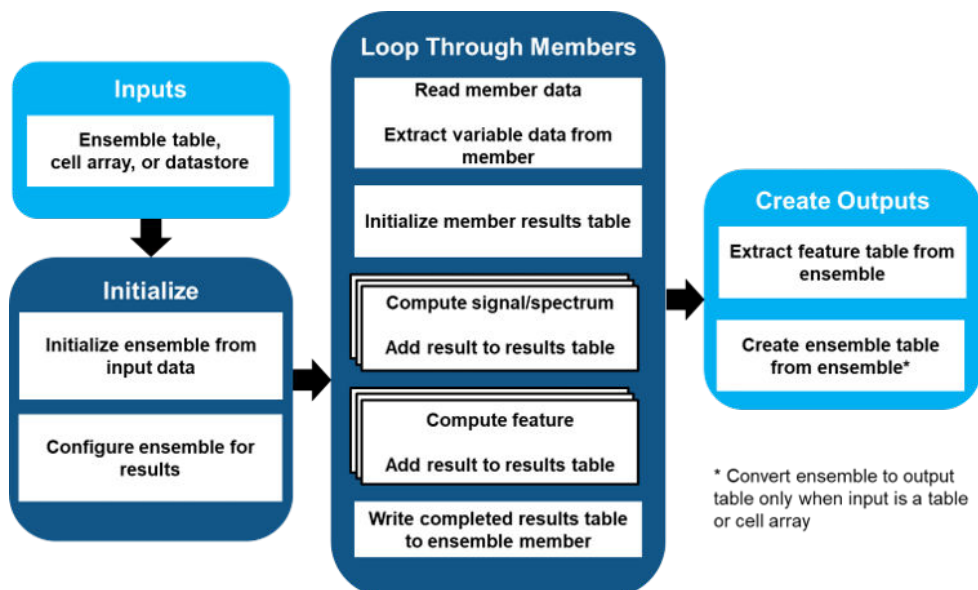
The figure breaks the code flow into three major sections: Initialize, Loop through Members, and Create Outputs.

- The Initialize block performs initial configuration. Specific operations depend on the type of data you originally imported and the variables and features that you specified for code generation.
- Loop Through Members block operations execute all the variable and feature computations one member at a time.
- Create Outputs extracts and formats the feature table and the full ensemble.

Inputs

The function operates on input data that is consistent with the data you initially imported into the app. When you plan to generate code from the app, importing your data in the same format as the data that you plan to apply your code to is recommended.

- If you imported a workspace variable such as an ensemble table or cell array from memory, the function requires an ensemble table or cell array.
- If you imported a file or simulation ensemble datastore, the function requires a file or simulation ensemble datastore.

The input data for the code must have a variable structure that is similar to the data that you imported into the app. Your input ensemble can include additional variables as well. The code ignores the additional variables and does not flag them as errors.

Initialization

In the Initialize block, the code configures an ensemble that contains variables both for the inputs and for the outputs that the function computes in the Loop Through Members block. These computed outputs include the variables and features you explicitly selected when you generated the code and any additional variables, such as a `tsa` signal, that any of your features require.

- If the input data is a table or cell array, the code creates a `workspaceEnsemble` object that includes variables corresponding to the input data variables. This object is similar to an ensemble datastore object, but it operates on data in memory rather than in external files.
- If the input data is a `simulationEnsembleDatastore` or a `fileEnsembleDatastore` object, the code operates on the object directly.

Once the code initializes the ensemble, the code appends all the variables and features to be computed during member computations. The code eliminates redundant variables with the `unique` function.

The figure shows an example of a workspace ensemble and its data variables. The data variables identify the input signals, output signals and spectra, and features.

Key ensemble-related functions during initialization include:

- `reset` — Reset ensemble to its original unread state so that the code reads from the beginning
- `workspaceEnsemble` — Ensemble object that manages data in memory
- `fileEnsembleDatastore` — Ensemble object that manages data in external files
- `simulationEnsembleDatastore` — Ensemble object that manages simulated data in external logs or files

---

Note During initialization, the function does not preallocate arrays to use during processing. This lack of preallocation is for clarity and flexibility, since the code must operate on an input ensemble with any number of members. During follow-on computation cycles, which append newly computed data to intermediate results tables, the code suppresses MATLAB Code Analyzer warnings about preallocation using the in-line comment `%#ok<AGROW>`. For information about Code Analyzer message preferences, see "Code Analyzer Preferences".

Member Computation Loop

In the member computation loop, the function performs all member-specific computations, one member at a time.

A series of `read` function calls initiates the loop, reading each ensemble member in succession until there are no ensemble members left. The computations that follow each `read` command provide, for that member, all the specified variables and features.

A running member-level results table collects the results as each variable or feature set is computed.

The figure shows an example of a member-level results table. Here, the results table contains two embedded tables that contain features and an embedded timetable that contains a computed signal.

Once all computations are complete, the code appends the full member results table back to the main ensemble.

Member computations use a `try/catch` combination to handle input data that cannot be processed. This approach prevents bad data from halting code execution.
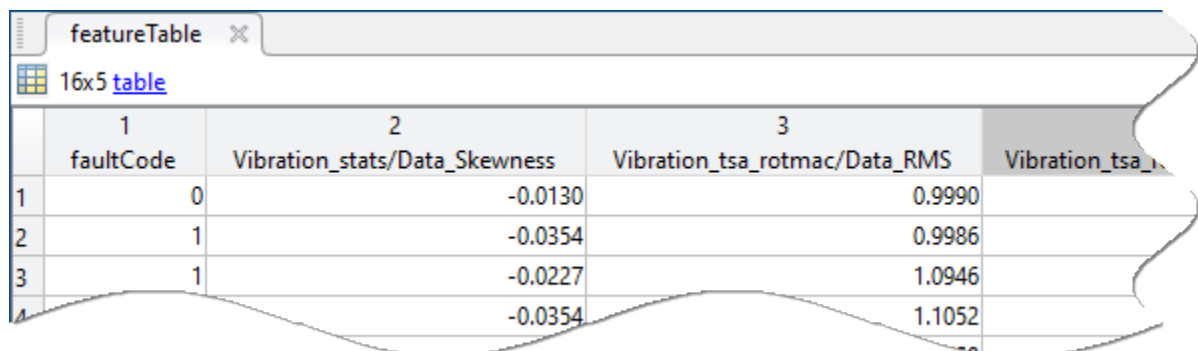
The code uses these key ensemble-management functions:

- `read` — Read the next ensemble member
- `readMemberData` — Extract data from an ensemble member for a specific variable
- `writeToLastMemberRead` — Write data to ensemble member
- `table` — Ensemble array that contains variables and features in named columns and members in rows
- `array2table` — Convert an array to a table
- `timetable` — Specialized member-specific table that contains signals in named variable columns and a specific time for each row
- `array2timetable` — Convert an array to a timetable

Outputs

The main output of the generated function is a feature table, which the code extracts using the function `readFeatureTable`. This output is the same whether you are using a workspace ensemble or an ensemble datastore as input. The feature table contains the scalar features themselves as well as the condition variables.

The figure shows an example of a feature table. Each row represents a member. The first column contains the condition variable, and subsequent columns contain a scalar feature value.

	1	2	3	
	faultCode	Vibration_stats/Data_Skewness	Vibration_tsa_rotmac/Data_RMS	Vibration_tsa_
1	0	-0.0130	0.9990	
2	1	-0.0354	0.9986	
3	1	-0.0227	1.0946	
		-0.0354	1.1052	

featureTable — 16x5 table

Use the optional second output argument to return the ensemble itself. If the input to your function is a table or cell array, the function converts the workspace ensemble into a table using the `readall` function, and returns the table.

The figure shows an example of an output table. Each row represents a member. The first two columns are the input variables, and the remaining columns contain features or computed variables.

If your function is based on an original import of an ensemble datastore object, the function returns the updated datastore object.

The code includes these key functions for outputs:

- `readFeatureTable` — Read condition variable and feature data from an ensemble data set into a table
- `readall` — Read all data from an ensemble data set into a table

Ranking

When you select one or more ranking tables when you generate code, the function includes a ranking section that follows the extraction of the feature table, as shown in the figure. The figure shows detail only for the portions of the flowchart that change from the Basic Function Flow figure. In Create Outputs, the figure shows all output arguments when using ranking.
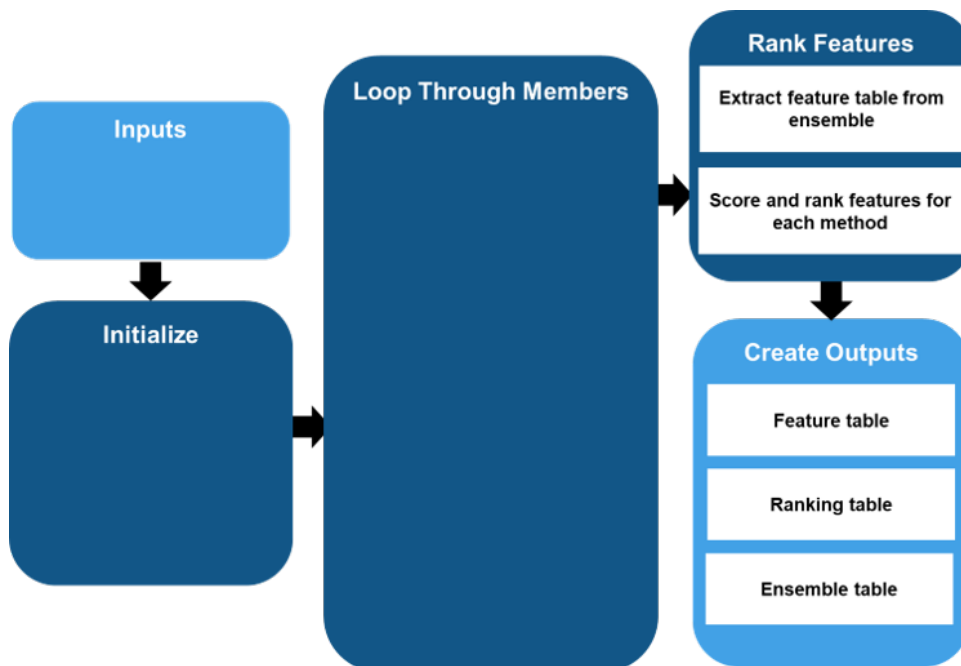
To initialize ranking, the code extracts the feature values and the labels (condition variable values) from the feature table. The code then defines class groups by converting the labels into numeric
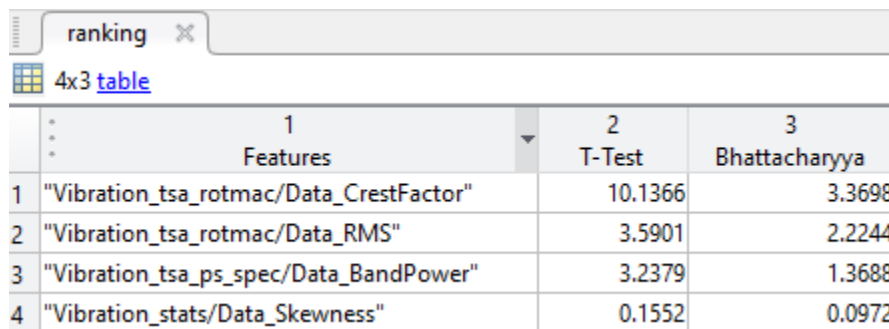
values using the function `grp2idx` to assign a group index to each feature. For example, if the condition variable `FaultCode` has the labels `"Faulty"`, `"Degraded"`, and `"Healthy"`, `grp2idx` groups the members with these labels into groups 1, 2, and 3.

For each ranking method, the code computes a score for each feature with these steps:

1 Normalize the features using the specified normalization scheme.

2 Call the function for the ranking method, using a group-index mask to separate the groups. The specific syntax depends on the ranking method function.

3 If a correlation-importance factor is specified, update the score using `correlationWeightedScore`. Correlation weighting lowers the scores of features that are highly correlated to higher ranking features, and that therefore are redundant.

4 Append the scores to the scoring matrix and the method to the method list.

The code then creates a ranking table by using `sortrows` to sort the rows by the scores of the **Sort By** method specified in the app during code generation.

The figure shows an example of a ranking table for four features, sorted by T-Test results.

	1 Features		2 T-Test	3 Bhattacharyya
1	"Vibration_tsa_rotmac/Data_CrestFactor"		10.1366	3.3698
2	"Vibration_tsa_rotmac/Data_RMS"		3.5901	2.2244
3	"Vibration_tsa_ps_spec/Data_BandPower"		3.2379	1.3688
4	"Vibration_stats/Data_Skewness"		0.1552	0.0972

The code uses these key functions for managing ranking:

- `grp2idx` — Convert labels to numeric values
- `correlationWeightedScore` — Weight feature-ranking scores with correlation factor
- `sortrows` — Rank features by sorting the rows by score

Ensemble Statistics and Residues

An ensemble statistic is a statistical metric that represents the entire ensemble rather than an individual member. For example, in the app, you can specify the ensemble maximum for a vibration signal. The resulting single-member statistic contains, for each time sample, the vibration value that is the maximum of all the member vibration values.

You can use ensemble statistics to compute residues by subtracting the same ensemble metric from all the member signals for a specific variable. For example, if the ensemble mean represents an average operating point, you can subtract the mean from all members to isolate behavior around the operating point. The isolated signal is a form of a residue.

The figure illustrates the code flow when you specify features based on a mean residual signal.
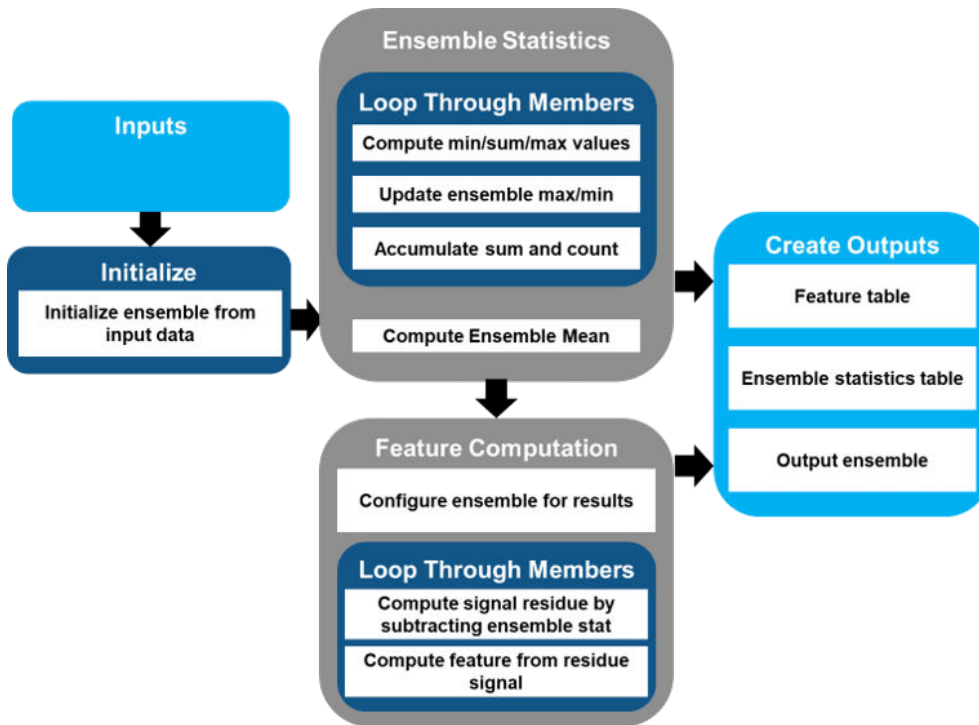
In this flow, there are two separate member loops. The first member loop computes the ensemble statistics. The second member loop performs the signal, spectrum, and feature processing. In the flowchart, the second member processing loop illustrates the residue signal and residue-based feature processing steps.

Loop 1: Ensemble Statistics Processing

To compute ensemble statistics for a specified variable, the code first loops through the members while maintaining an accumulator. At a given point in the looping sequence, the accumulator might contain, for example:

- The maximum signal value calculated so far
- A running sum of all the data values and an iteration count
- The minimum signal value calculated so far

The figure shows an example of accumulator contents and the running sum and count in the `mean` variable.

At the end of the loop iterations, the code transfers the ensemble max and min signals from the accumulator to the ensemble statistics max and min variables. The code calculates the ensemble mean by dividing the ensemble sum by the number of counts.

The figure shows an example of final ensemble statistics table and the final `mean` variable that now contains the mean signal.

Loop 2: Residue Processing

In the main member processing loop, the code creates residue signal variables by subtracting the specified statistics from the specified signals, and packages these residues in the same manner as other signals and features.

The figure shows an example of a member result table with residues. The table contains two residue signals and two feature sets computed from those signals.

Parallel Processing

When you specify parallel processing, the code partitions the ensemble members into subensembles and executes the full member processing loop in parallel for each subensemble, as the figure shows.
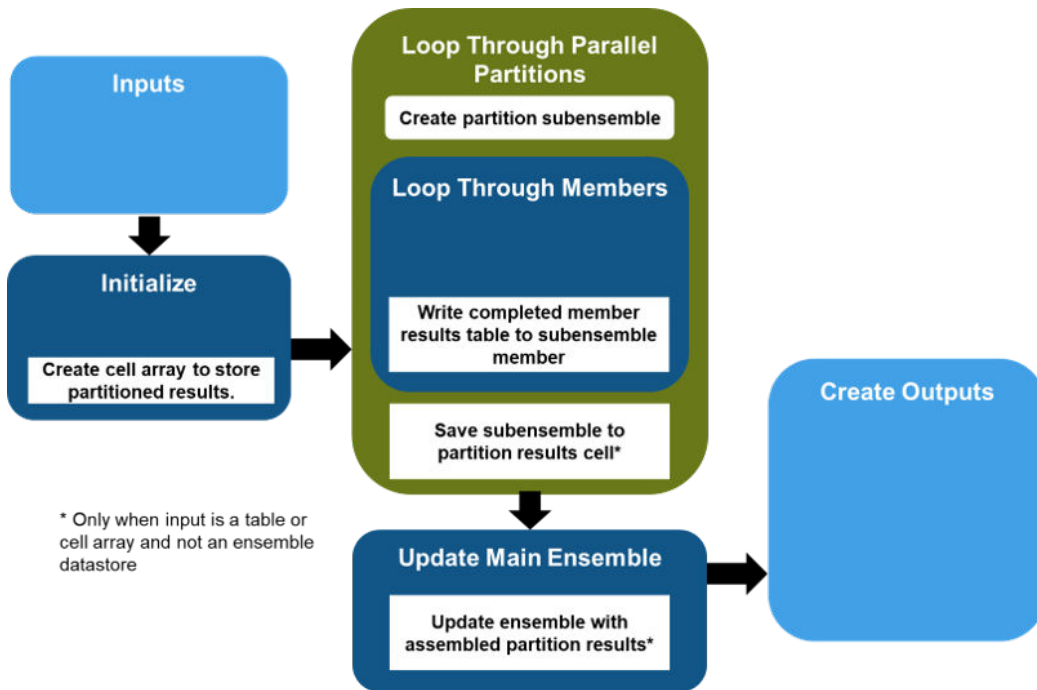
If the main ensemble is a `workspaceEnsemble` object, then at the end of each partition-processing cycle, the code saves the updated subensemble as a cell in an array that stores the results for all the subensembles. The following figure shows an example of this array along with the first two cells. In this figure, each partition contains 13 members.
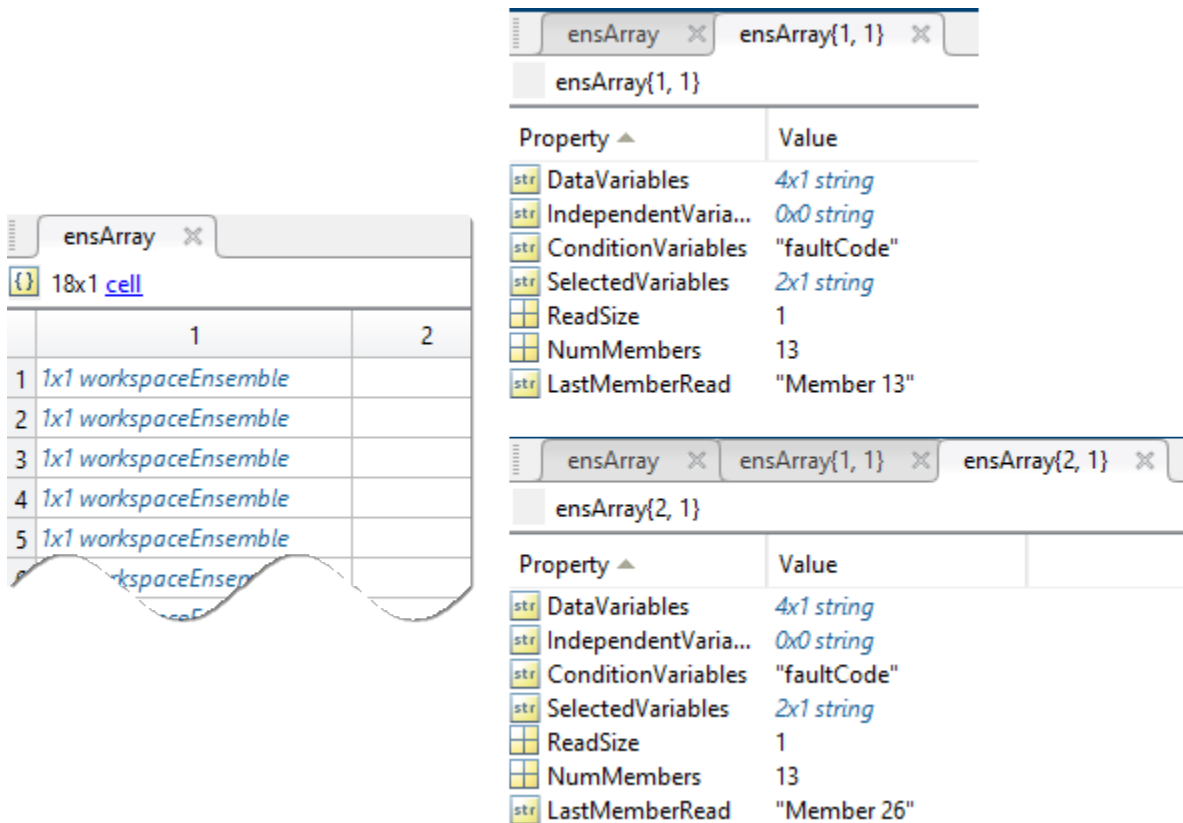
If the main ensemble is a workspace ensemble, then once all partition processing is complete, the code reassembles the result partitions and updates the main ensemble using the `refresh` command.

If the main ensemble is an ensemble datastore object, then the code updates the object directly when it writes results to the subensemble member at the end of each member loop.

The code uses these key functions for parallel processing:

- `numpartitions` — Number of partitions to separate ensemble members into
- `partition` — Partition an ensemble
- `refresh` — Update a workspace ensemble with reassembled partition results

Frame-Based Processing

When you have specified frame-based processing in the app, the generated code divides each full member signal into segments, or frames. The size and frequency of these frames are stored in the frame policy.

The figure illustrates the flow. The code executes a frame loop within each member loop. When you are selecting features for generating code, the app constrains your feature selections to a single frame policy. The generated function therefore never contains more than one frame loop.
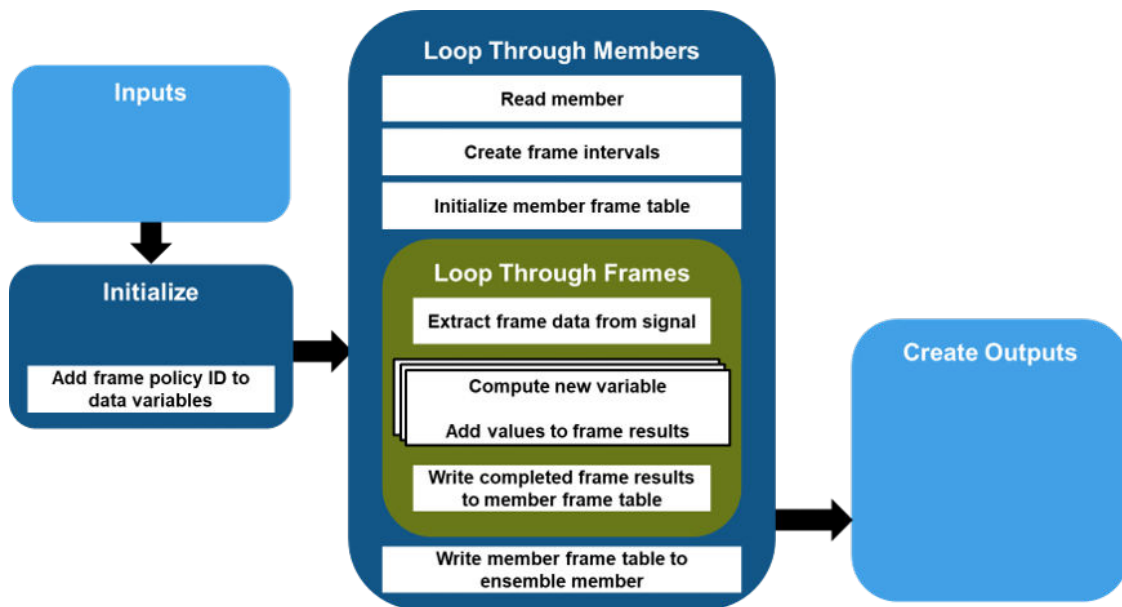
During the initialization portion, the code adds only the input variables and the frame policy id, such as FRM_2, to the data variables. The code does not add the variables to be computed. Those variables are stored in the FRM_ variable.

During the first part of the member loop, the code:

1 Reads the full member signal.

2 Creates a frame interval table that spans the time range of the full signal and which contains the start and stop times for each frame, using `frameintervals`.

3 Initializes a frame table at the member level. This table ultimately contains computed variable values for all the frames in the member.

The second part of the member loop is the frame loop. For each frame, the code:

1 Uses the frame interval information to extract the data for that frame from the full signal.

2 Computes the signals, spectra, and features in the same manner as for full-signal processing at the member level. After computing each new variable, the code appends the variable to the frame results table. The figure shows an example of a member frame table. The first two elements contain the start and stop time of the frame interval. The final element contains the features computed for that frame.

3 When the variable computations are complete, the code appends the completed frame results table to the member-level frame table. The figure shows an example of the member-level table, which contains the frame results for all members.

frames

8x3 table

	1 TimeStart	2 TimeEnd	3 Signal_stats
1	0 sec	0.42 sec	1x3 table
2	0.42 sec	0.84 sec	1x3 table
3	0.84 sec	1.26 sec	1x3 table
4	1.26 sec	1.68 sec	1x3 table
5	1.68 sec	2.1 sec	1x3 table
	2.1 sec		1x3 table

The final operation in the member loop is to write the completed member frame table to the ensemble member.

Creation of the feature table output is essentially the same as for the basic case, but each member variable now includes all the segments.

featureTable

80x6 table

	1 EnsembleID_	2 FRM_2/TimeStart	3 FRM_2/TimeEnd	4 FRM_2/Signal_stats/vib_Mean	5 FRM_2/Signal_stats/vib_S
1	"Member 1"	0 sec	0.42 sec	0.0624	
2	"Member 1"	0.42 sec	0.84 sec	0.0618	
3	"Member 1"	0.84 sec	1.26 sec	0.0605	
4	"Member 1"	1.26 sec	1.68 sec	0.0620	
5	"Member 1"	1.68 sec	2.1 sec	0.0648	
6	"Member 1"	2.1 sec	2.52 sec	0.0621	
7	"Member 1"	2.52 sec	2.94 sec	0.0606	
8	"Member 1"	2.94 sec	3.36 sec	0.0654	
9	"Member 2"	0 sec	0.42 sec	0.0627	
10	"Member 2"	0.42 sec	0.84 sec	0.0618	
11	"Member 2"	0.84 sec	1.26 sec	0.0634	
12	"Member 2"	1.26 sec	1.68 sec	0.0642	
13	"Member 2"	1.68 sec	2.1 sec	0.0620	
14	"Memb_"	2.1 sec	2.52 sec	0.0621	
		2.52 sec		0608	

See Also

frameintervals | workspaceEnsemble | grp2idx | readall | correlationWeightedScore | simulationEnsembleDatastore | fileEnsembleDatastore | reset | unique | read | readMemberData | writeToLastMemberRead | refresh

More About

Import Data into Diagnostic Feature Designer

Diagnostic Feature Designer is an interactive tool for processing ensemble measurement data and extracting features that indicate the condition, such as `healthy` or `faulty`, of the machines that produced the data. The data can come from measurements on systems using sensors such as accelerometers, pressure gauges, thermometers, altimeters, voltmeters, and tachometers. The main unit for organizing and managing multifaceted data sets in the app is the data ensemble. An ensemble is a collection of data sets, created by measuring or simulating a system under varying conditions. Each row within the ensemble is a member. Each member of the ensemble contains the same variables, such as `Vibration` or `Tacho`.

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

Before importing your data, it must already be clean, with preprocessing such as outlier and missing-value removal. For more information, see "Data Preprocessing for Condition Monitoring and Predictive Maintenance" on page 2-2.

During the import process, you select which variables you want to import, specify the variable types, and perform other operations to create the ensemble that you want to work with. When you click import, the app applies your specifications and creates an ensemble that contains your selected data. The following figure illustrates the overall import flow.
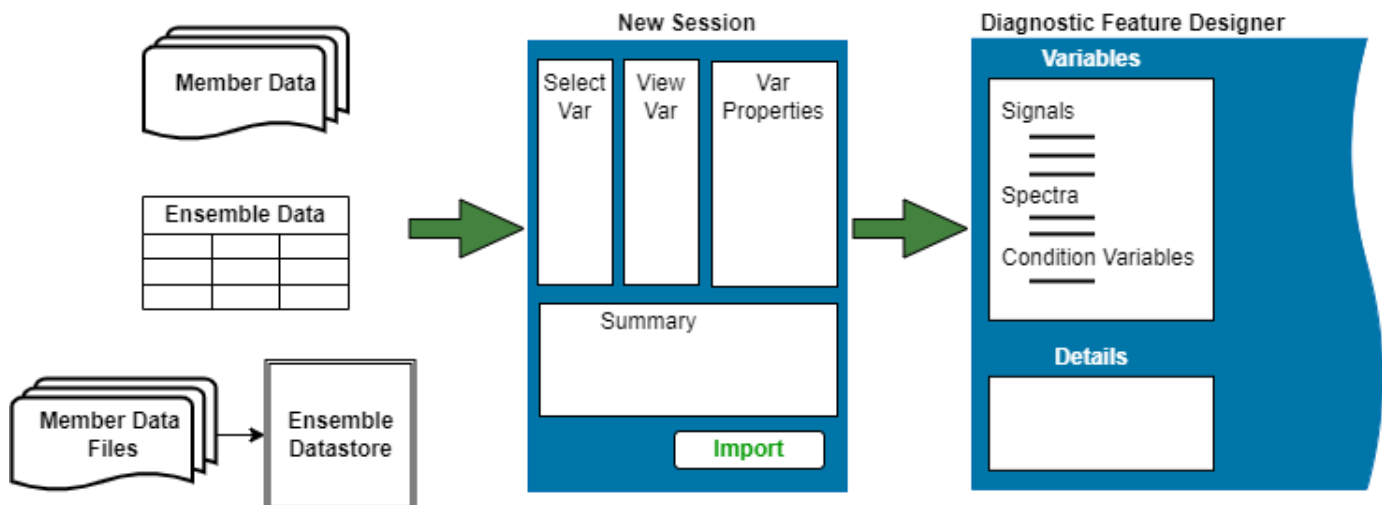
After you import your data once, you do not need to import it again for subsequent sessions. Save your session to store both the initial data ensemble and any derived variables and features you compute prior to saving. You can also export your data set to the MATLAB workspace, and save your data as a file that you can import in a subsequent session.

If you have a large number of ensemble members, consider creating a representative subset of members when you first start exploring the data and potential features in the app. Because the app is

interactive, importing a large number of members can result in slower performance. Instead, you can develop and rank features interactively with the smaller data set, and then generate code that repeats the computations on the original data set.

For more information on data sources, ensembles, and variable types in predictive maintenance, see "Data Ensembles for Condition Monitoring and Predictive Maintenance" on page 1-2.

Source Data Requirements

For signals, the app accepts individual member `table` arrays, `timetable` arrays, cell arrays, or numeric matrices, each member containing the same independent variables, data variables, and condition variables. For spectral data, the app accepts individual member `table` arrays or `idfrd` objects. This table describes the data requirements for variables within ensemble members.

Input Item	Content	Notes
Signal data	Timetables, tables, cell arrays, or numeric arrays	For time-based data, timetables are recommended.
Signal independent variables (IVs)	Double, `duration`, or `datetime`	For each signal variable, all member IVs must be of the same type, whether your signal is based on time or on another IV such as consumption or duty cycles. If your member data is stored in a matrix, you must have only one IV that applies to the full set of signals that you are importing. If your data was uniformly sampled in time and you do not have recorded timestamps, you can construct a uniform timeline during the import process.
Spectral data	Numeric (double) table or `idfrd` object	Each `idfrd` object must contain data for only one spectrum (the first dimension is 1). You cannot import spectra from matrices.
Condition variables (CVs)	Scalar — Numeric, string, cell, or categorical	You can import condition variables along with your data in tables, timetables, or cell arrays, but not in matrices.
Features	Scalar — Numeric, string, or cell	You can import features that you computed previously, either externally or within the app itself.

Input Item	Content	Notes
Matrices	Purely numeric array that contains columns representing a single IV and any number of signals that share that IV. Cannot accommodate spectra. Cannot generally accommodate condition variables or features.	Matrices cannot accommodate variable names. You can import condition variables and features from a matrix if your data set contains only scalars and does not contain signals.

You can import data members individually or as an ensemble that contains all your data members. This ensemble can be any of the following:

- An ensemble table containing `table` arrays, cell arrays, or matrices. Table rows represent individual members.

- An ensemble cell array containing tables, cell arrays, or matrices. Cell array rows represent individual members.

- An ensemble datastore object such as a `fileEnsembleDatastore` or `simulationEnsembleDatastore` object that contains the information necessary to interact with files stored externally. Use an ensemble datastore object especially when you have too much data to fit into app memory. Reading and writing to external files during computations does impact performance, however. To create a representative subset of the ensemble datastore files to work with, see `subset`.

- An ensemble matrix that contains only condition variables and features. Matrix rows represent individual members.

- A labeled signal set, which is a `labeledSignalSet` object, that contains labeled data for signals, condition variables, and features.

For more information about organizing your data for import, see "Organize System Data for Diagnostic Feature Designer" on page 7-19.

Typical Workflows for Data Import

The following sections describe typical workflows for different types of data import. The first workflow describes the overall import process and the subsequent workflows focus on variations that are specific to the import data format. The following table summarizes the cases for the workflows.

Import Type	Summary
Ensemble table on page 7-130	Core workflow that describes the import of an ensemble table, including ensemble specification, configuration, import, and confirmation in the app
Individual Ensemble Members on page 7-136	Use of the filtering capability within the import dialog process to quickly select and import individual ensemble members that are represented by tables or timetables
Matrices on page 7-139	Use of matrix column indices for variable identification. Alternative specification of matrix as feature rather than signal.
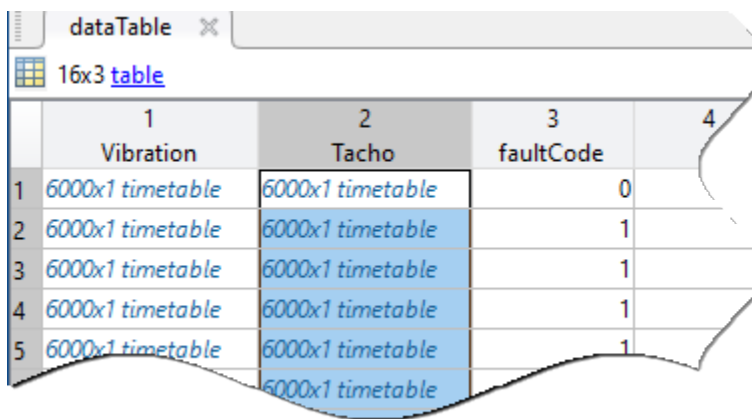
Import Type	Summary
Spectral Data on page 7-141	Import of spectral data in `idfrd` object or in a table with columns that contain frequency or order values and corresponding data values
Signal with no IV on page 7-145	Import of signal with no explicit time stamps by generating virtual IV
Signal with multiple IVs on page 7-147	Specification of an alternative IV so that the signal can be analyzed in the app using either IV
Ensemble Datastore on page 7-149	Specification of an ensemble datastore that contains information on interacting with external files.
"Import Labeled Signal Set (LSS)" on page 7-151	Import of signal, condition variable, and feature information from a `labeledSignalSet` object.

Core Workflow—Import Ensemble Table

This workflow illustrates the steps for importing an ensemble table into the app.

Load Ensemble Table into MATLAB Workspace

Load your ensemble table into the MATLAB workspace. You can preview the data in the workspace variable browser, as this example shows. In this case, the data set contains two time-based signals and a scalar condition variable named `faultCode`.

Open App and Start New Session

Open the app by entering `diagnosticFeatureDesigner` at the command line. Then, click **New Session**. This action opens the import dialog box.
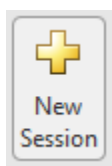
Select Ensemble Table from Source Variables

In the **Select dataset from workspace** pane, select your ensemble table as the **Source**.
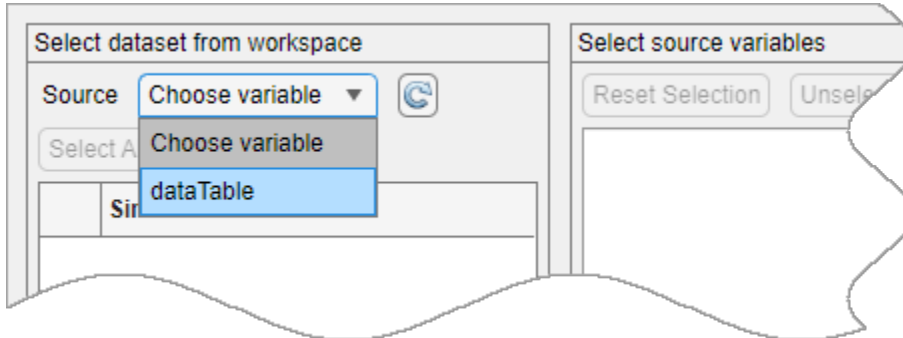
View Source Variable Components and Change Variable Type and Units

The **Select source variables** pane displays the variables from your ensemble. In the following example figure, the app identifies the `Vibration` and `Tacho` variables as time-based signals that each contain `Time` and `Data` variables. The third variable in the variable set for each signal, `Sample (Virtual)`, is not checked. `Sample (Virtual)` allows you to generate an IV, especially when your source data set does not contain an explicit IV. For more information on `Sample (Virtual)`, see "Import Signal with No Time Variable" on page 7-145.
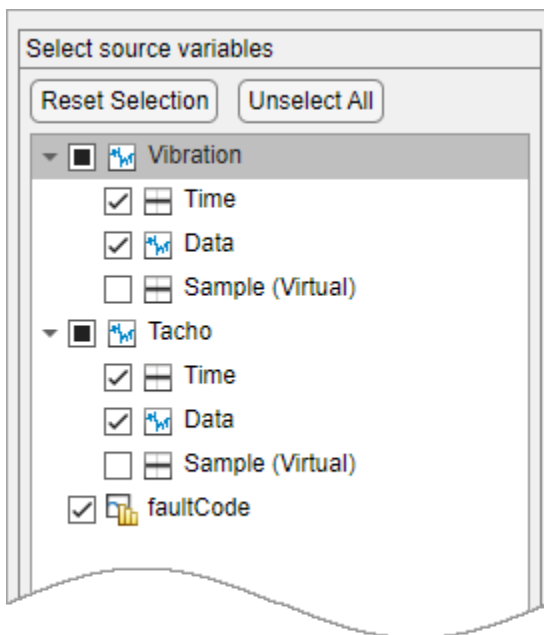
The icons identify the variable type that the app assumes. The icon next to `faultCode`, which illustrates a histogram, represents a feature. Both features and condition variables are scalars, and the app cannot distinguish between the two unless the condition variable is categorical. To change a variable type, click the variable name to open the variable properties in the **Configure source variable properties** pane. Then, in **Variable type**, change **Feature** to **Condition Variable**.
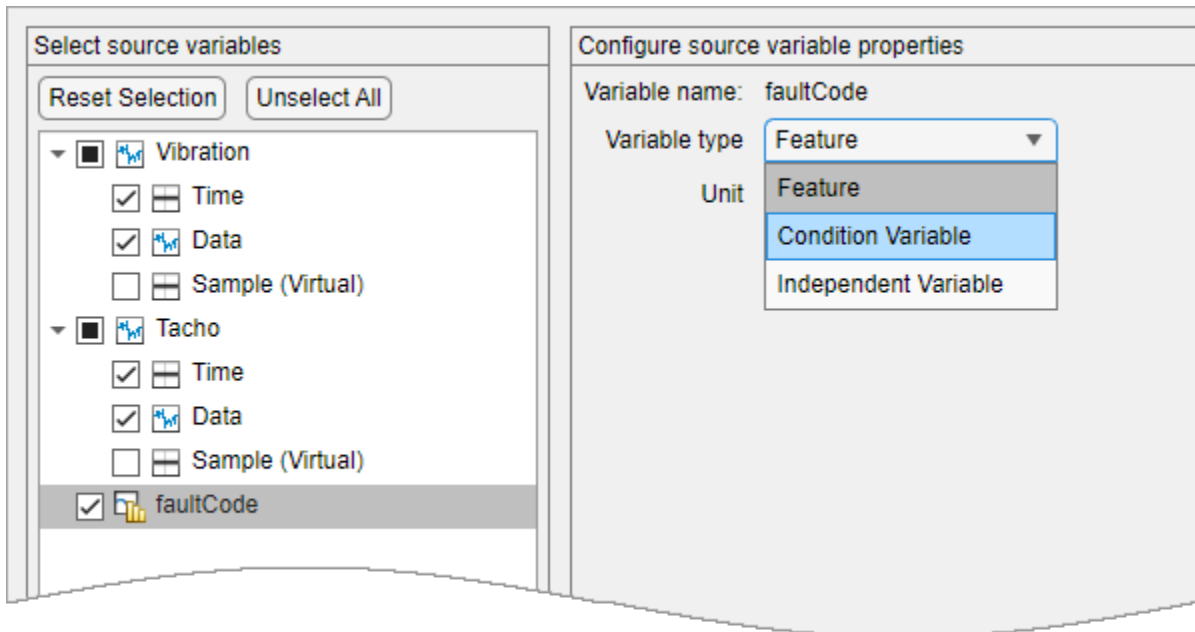
The icon for `faultCode` now illustrates a label, which represents a condition variable.
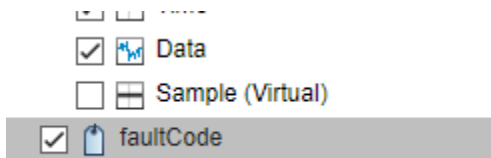
For signal and spectrum variables, you can also change the units that the app uses for plotting and for other operations within the app. To do so, in the lower level variable list of the signal or spectrum variable, click the name of the IV or data variable. The **Configure source variable properties** pane provides a menu of options for each property that you can modify. In the following example figure, **Configure source variable properties** displays properties for the `Time` variable of the `Vibration` signal. The figure illustrates the selection of **Minutes** from the menu of **Units** options.
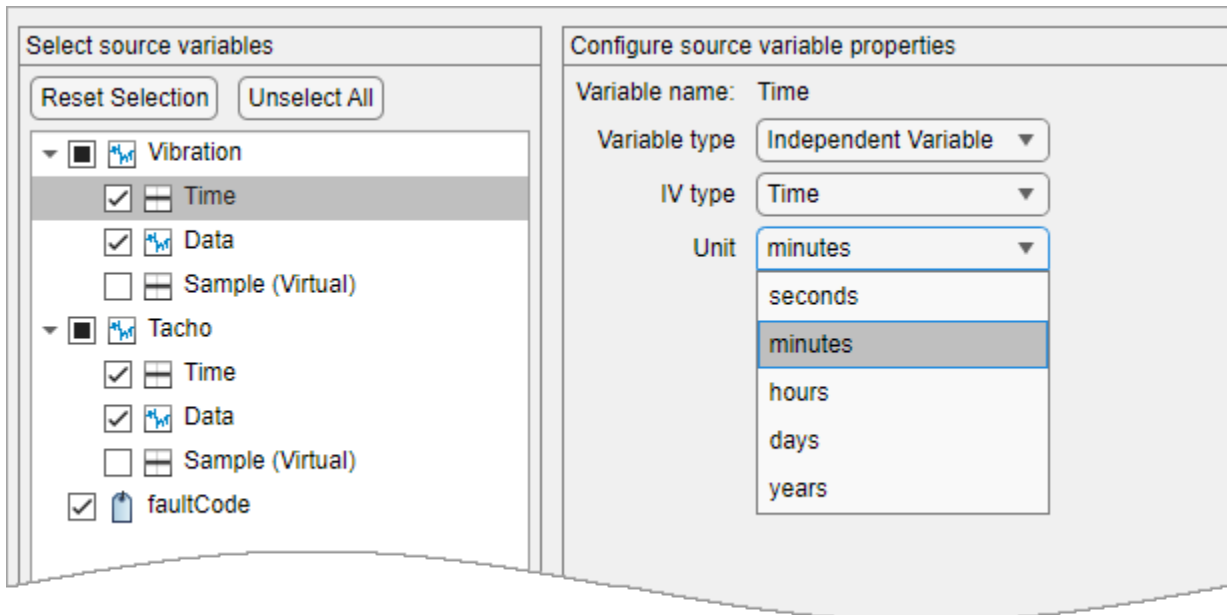
Preview Data Variables

In addition to providing options for variable properties such as `Type`, the **Configure source variable properties** pane displays a preview of your import data when you click the name of a signal or spectrum variable. The following example figure shows the preview of the `Vibration` data. The preview pane in the figure displays source data for the first ten `Vibration` samples of the first ensemble member, and includes values for IV, data, and the sample index.

The preview pane displays source properties only. The preview pane does not reflect any property modifications that you make in the pane. For example, if you change the units of the `Vibration` signal from seconds to minutes, the preview pane still displays source units of seconds. When you complete the import, the app converts the time data to minutes for use in the app.

Confirm Ensemble Specification and Execute Import

Confirm the ensemble specification in the Summary pane at the bottom of the dialog box. Click **Import** to execute the data import.

Confirm Successful Import into App

Confirm the import in the variables pane. In this example, the two signals appear in the **Signals** list. The condition variable `FaultCode` appears in the **Condition Variables** list. Beneath the variables pane is the **Details** pane, which provides additional information about the selected variable.

Select `Vibration/Data` and click **Signal Trace** to plot the data and view the imported signals, as the following example figure shows. For more information on plotting data, see "Import and Visualize Ensemble Data in Diagnostic Feature Designer".
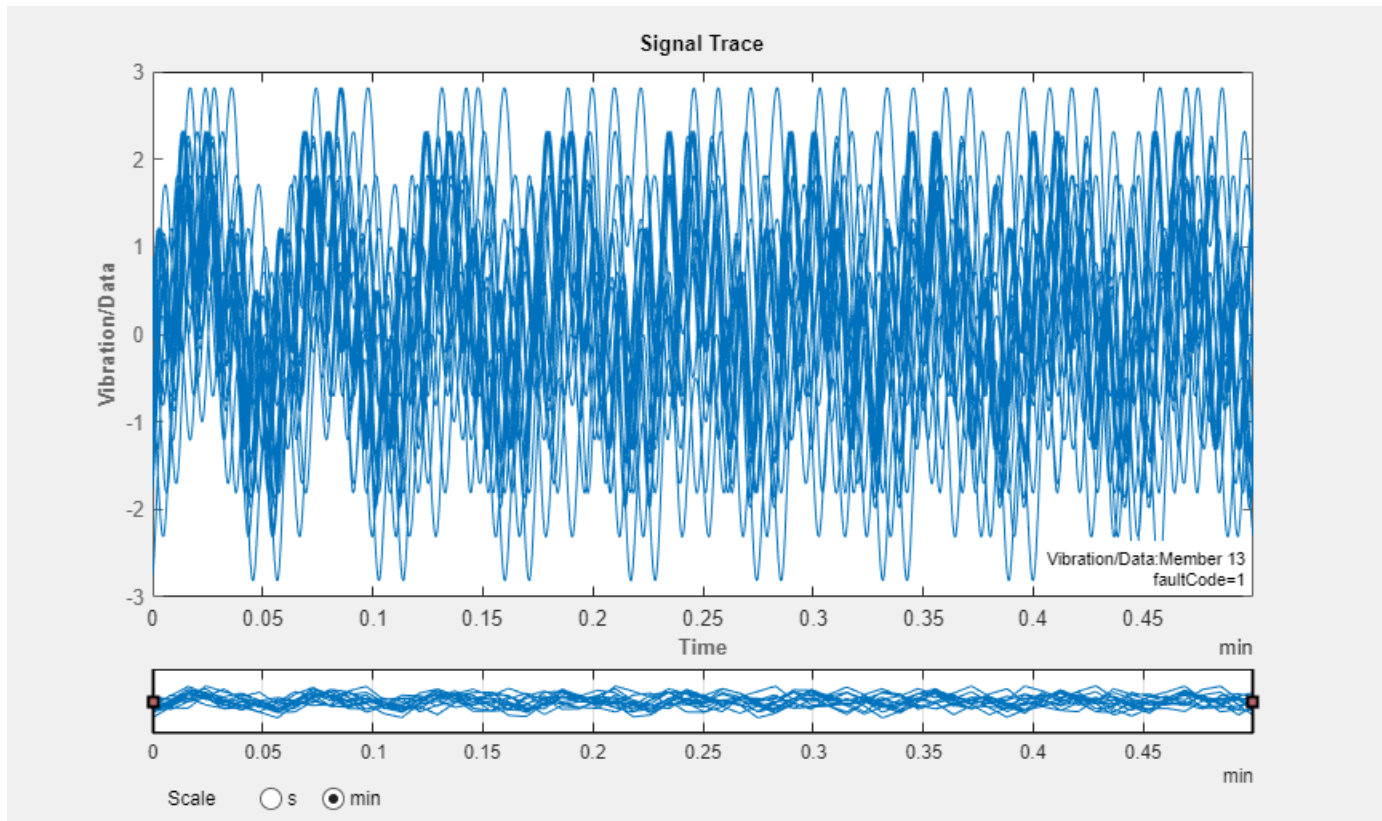
Import Individual Members

This workflow describes the steps associated with importing ensemble members individually.

Initiate Import Process

Load the individual member variables into your MATLAB workspace, open the app, and click **New Session**. In the **Select dataset from workspace** pane, the **Source** menu displays a list of the files in your workspace. The following example figure shows 10 member files and one additional file, sens2, that is not an ensemble member.
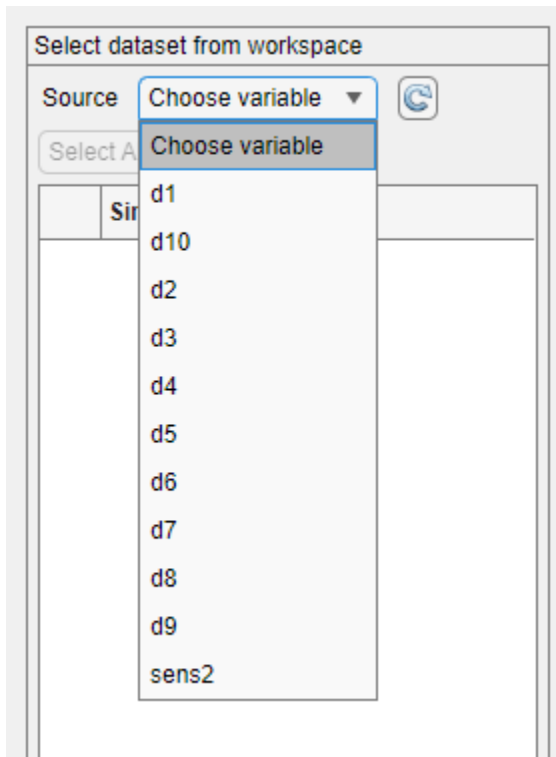
Select Ensemble Members

Select one of the variables that represents an ensemble member. In this example, select d1. The app opens a list of all compatible workspace variables that contain the same internal variables. You can select any combination of these variables or click **Select All** to select all of them at once.

Import Individual Timetables

If your individual members are packaged as timetables with a scalar value for each time point, specify **Use as signal** to have the app interpret the timetable variable as a signal rather than as a feature.

Complete Import Process

Once you select your variables, the remaining import steps are the same as in "Core Workflow— Import Ensemble Table" on page 7-130. The app combines the members you import into a single ensemble.

Import Matrix Data

This workflow describes the import of signals from individual member matrices. When you import data in matrices, each signal in the ensemble must share one independent variable, such as a time variable, with all signals in the ensemble. You cannot import condition variables, features, or spectral data in matrices.

Select and Preview Matrices to Import

Load your matrices into the MATLAB workspace, and then initiate the import process by clicking **New Session**. In the **Select dataset from workspace** pane, select one of your matrices and click **Select All**. For matrices, this pane also displays a **Use as feature** option. This option applies to the special case where a matrix contains only scalar condition variables and features, and no signal data.

Because matrices are numeric, the app identifies each variable column by its column index. The following example figure shows four member matrices. The first column of each matrix contains the IV representing time, and the second and third columns contain the data values for vibration and tacho data. To preview the contents of the ensemble, in the **Select Source variables** pane, select the `Matrix` row.

Confirm Variable Types

The app interprets `Col1` as the IV because it is monotonic, and `Col2` and `Col3` as signal data variables.

If you cannot accurately represent your signals with a single time variable, convert your matrices into tables before import.

Complete and Confirm Import

Click **Import** to complete the import. Confirm that the variables pane contains the signals you want, as shown in the following example figure.
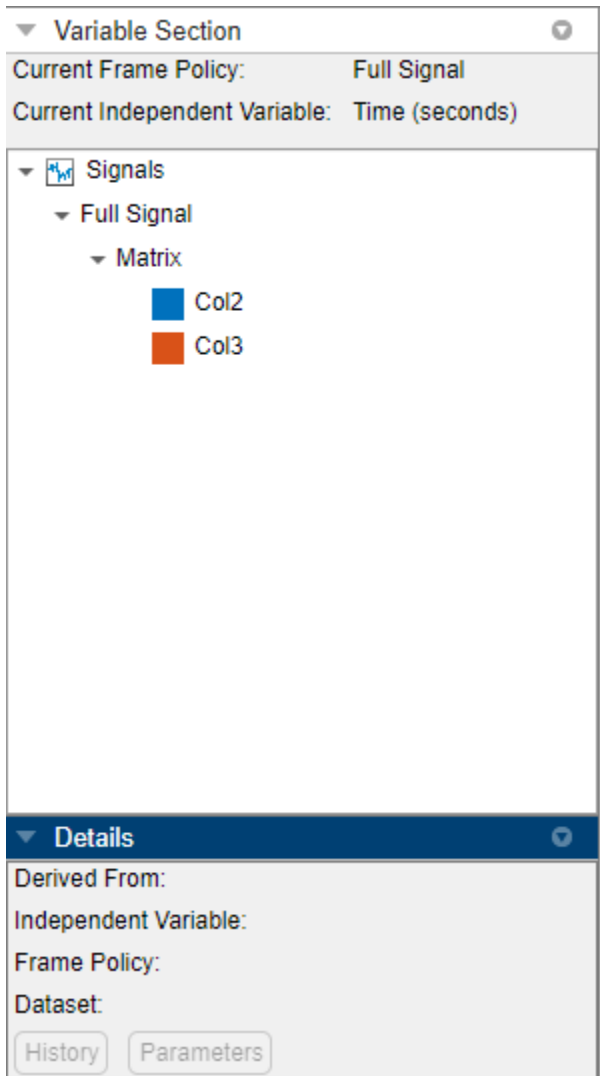
The app merges the matrices into an ensemble data set that contains the four matrices.

This workflow demonstrates that you can import matrices, but only with limitations. If you want to identify your variables by name, import condition variables or features, or use independent timelines for independent signals, convert your matrices to tables or cell arrays prior to import. For an example of converting a set of matrices to an ensemble table, see "Prepare Matrix Data for Diagnostic Feature Designer" on page 7-10.

Import Spectral Data

This workflow illustrates how to import spectral data. You can import spectral data in two forms:

1   An `idfrd` object that contains frequency and spectrum data for a single spectrum in the `Frequency` and `SpectrumData` properties, respectively
2   A table that contains columns with the frequency and spectral data

When you import an `idfrd` object, the app recognizes that the data source is spectral, and defaults to the **Spectrum** variable type, as the following example figure shows.

When you import spectral data in a table, the app defaults to the **Signal** variable type, but provides additional options for the **Spectrum** and **Order Spectrum** variable types. The following example figure illustrates the import of spectral data in a table.

You can also import order spectra that contain order rather than frequency information. Order spectra are useful for analyzing rotating machinery. Each order is a multiple of a reference frequency, such as the primary-shaft rotational frequency. The following example figure illustrates the import of an order spectrum.

Select source variables

Reset Selection   Unselect All

- ▼ ■ ⩘ Tacho
  - ☑ ⊞ Time
  - ☑ ⩙ Data
  - ☐ ⊞ Sample (Virtual)
- ▼ ☑ ⩗ Vibration_ps
  - ☑ ⊞ Frequency
  - ☑ ⩙ SpectrumData
- ☑ ⬚ faultCode
- ▼ ■ ⩗ Vibration_ps_1
  - ☑ ⊞ Frequency
  - ☑ ⩙ SpectrumData
  - ☐ ⊞ Frequency_1 (Virtual)
- ▼ ■ ⩘ Vibration_os
  - ☑ ⊞ Order
  - ☑ ⩙ SpectrumData
  - ☐ ⊞ Sample (Virtual)

Configure source variable properties

Variable name: Vibration_os

Variable type   Signal   ▼

Order	Signal	ata	Sample
	Spectrum	1.1121	0
	Order Spectrum	1.9208	1.0000
5.4846		1.1624	2.0000
8.2269		0.5490	3.0000
10.9692		0.1581	4.0000
13.7115		0.0103	5.0000
16.4538		0.0001	6.0000
19.1960		0.0000	7.0000
21.9383		0.0000	8.0000
24.6806		0.0000	9.0000

When you complete the import, the **Variables** pane displays the spectra, as the following example figure shows.
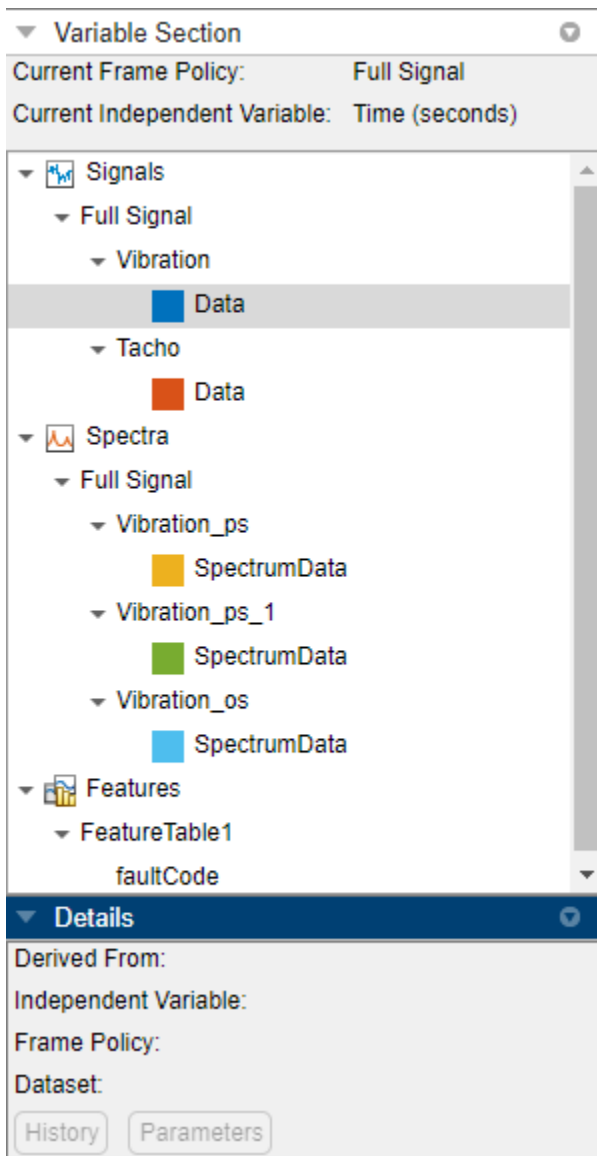
Import Signal with No Time Variable

This workflow illustrates how to generate a virtual IV if your data set includes signals that contain no IV. For instance, you might have a signal that was measured or generated in uniform time samples, but which does not include a vector of the actual time stamps. The app can generate a virtual timeline that contains the same sample rate.

Select Data Source and View Source Variables

Initiate the import process and select workspace variables to import. In the following example figure, the data source is a table that contains `Vibration` and `Tacho` variables. However, these variables contain only measurement data and no time information. As always, the app provides a `Sample (Virtual)` option. In this case, since the data has no IV, the app automatically selects this variable.

View Virtual IV Properties

The default unit for a virtual IV is the sample index. You can modify this default setting by selecting the `Sample (Virtual)` name, which opens the source properties. The following example figure displays the default properties.

Reconstruct Signal Time Variable Using Sample Time

If you know the sample time of the signal, you can reconstruct the time variable. To do so, change **Independent variable name** to the name you want, **IV type** to **Time**, **Unit** to the time units you want, and **Sampling interval** to the sample time. For example, consider that you know the sample time for both `Vibration` and `Tacho` is 0.001 seconds. The following figure shows how to set this sample time for `Vibration`. Note that these settings do not affect `Tacho`.
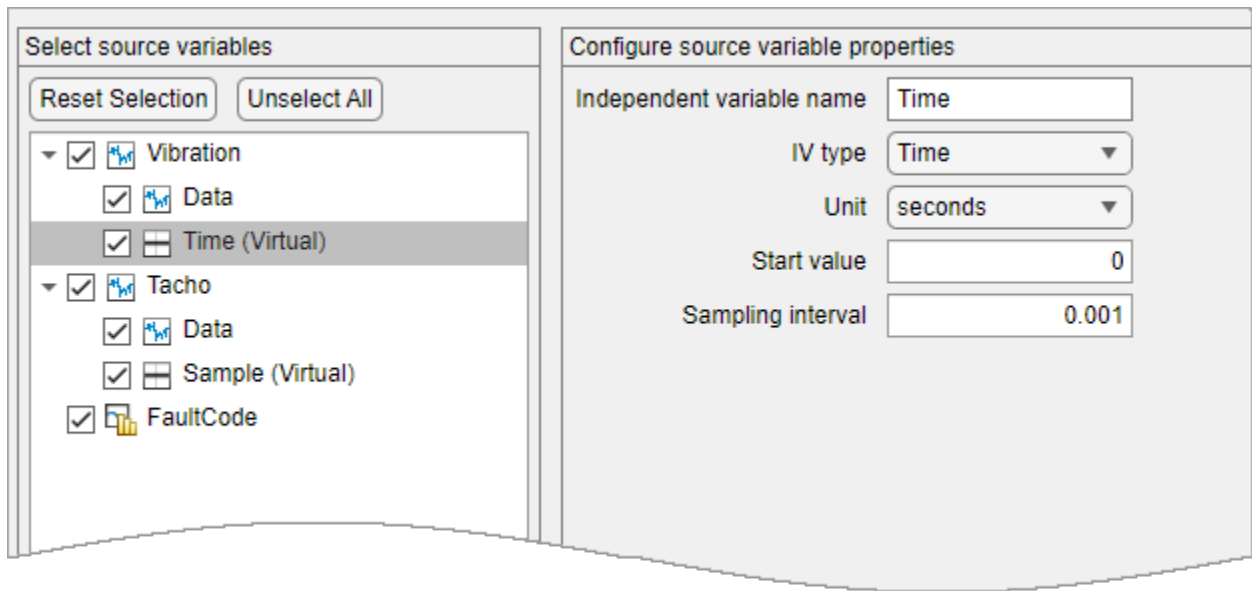
Once you have reconstructed the IVs that you want, complete the import process. You can view your reconstructed timelines by plotting your imported signals in the app. The following figure shows plots for `Vibration`, which has the reconstructed timeline, and `Tacho`, which retains the default IV of `Sample`.
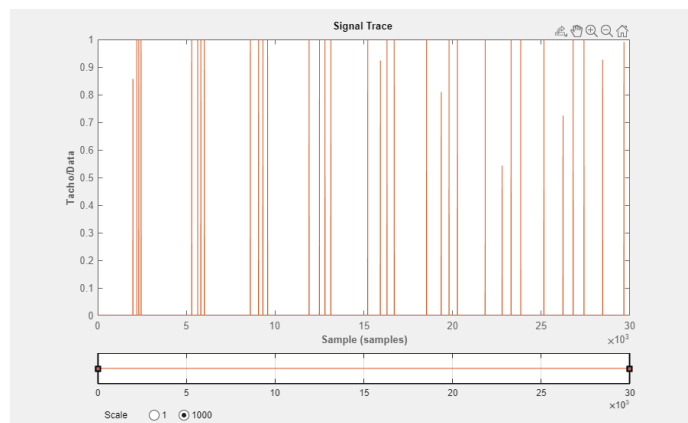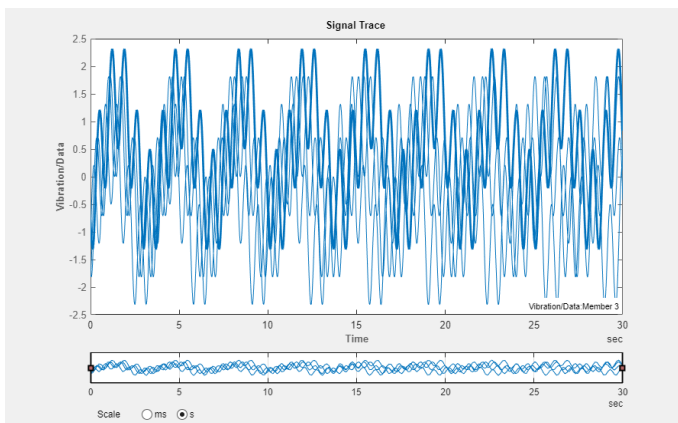
Specify Sample Index as Alternative IV

This workflow describes the steps for specifying the signal sample index as an alternative IV when you also import a time variable or some other signal IV.

Specify Sample Index as IV

Initiate the import and select the data to import. In the **Select source variables** pane, select `Sample (Virtual)` and view the properties. The following example figure illustrates this step. In this figure, `Vibration` now has all three lower level variables selected. The **Configure source variable properties** pane displays the default IV type and unit for `Sample (Virtual)`, which are **Index** and **samples**, respectively.
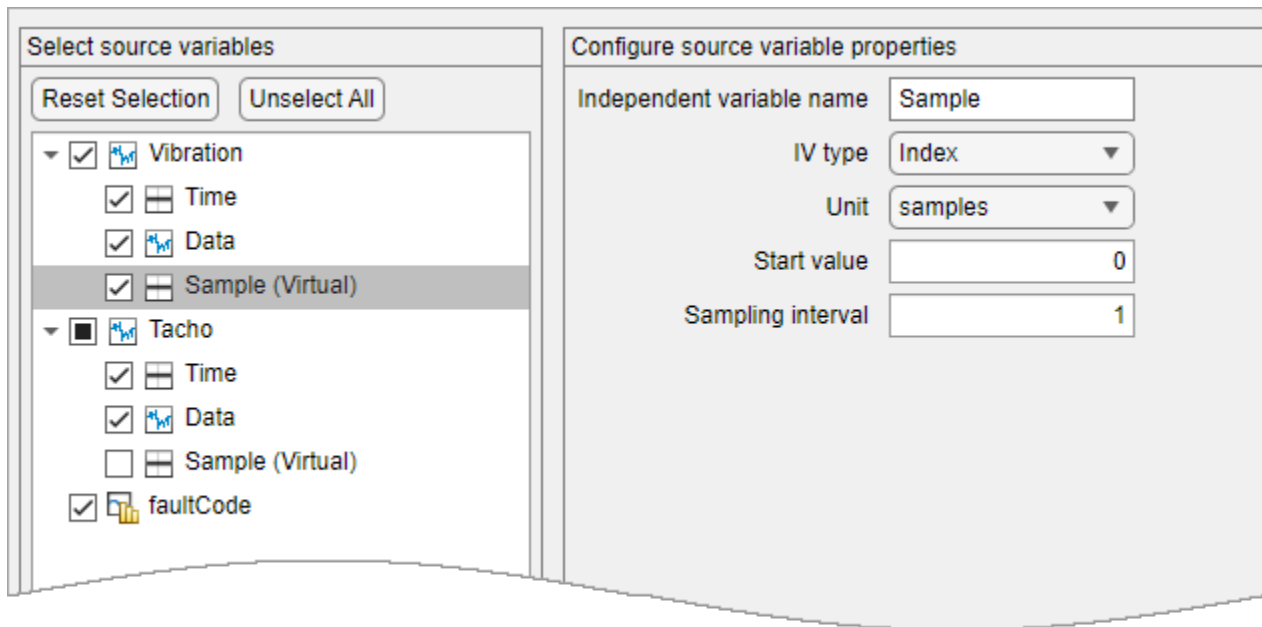
Perform the same operation on all variables for which you want to include the sample index. Complete the import process.

Switch to Sample Index the App

The app defaults to the IV type that you imported with the data. To switch to the sample index, in **Options**, select **Index**, as the example figure illustrates.
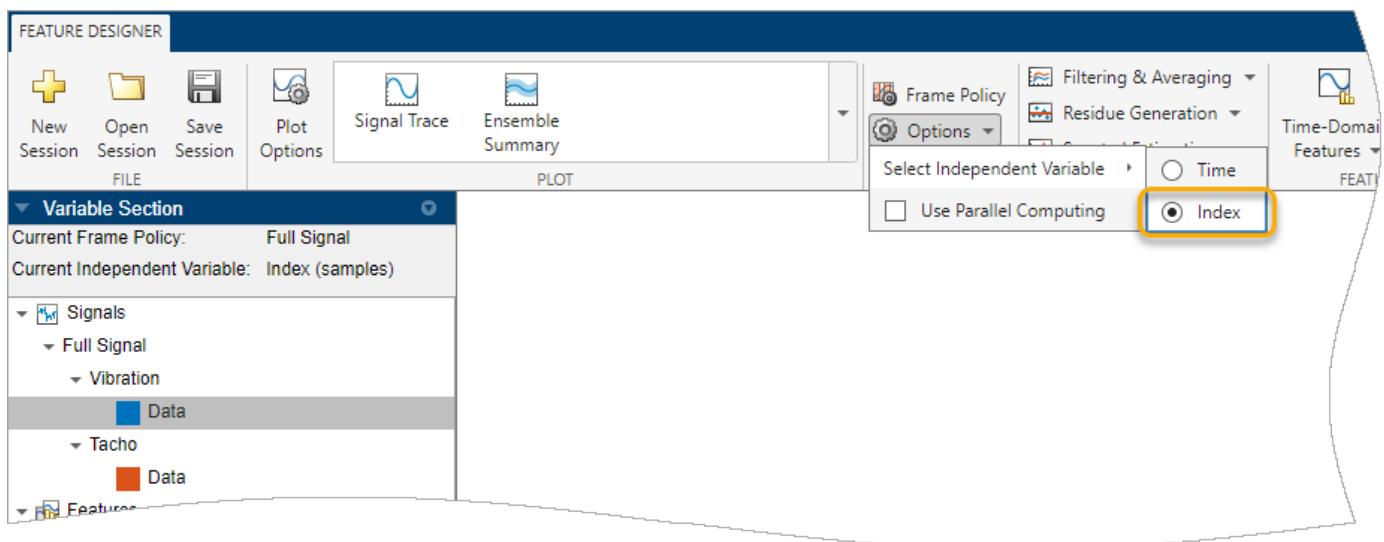
All signals that include **Index** as an alternative IV type now use the sample indices rather than the time values. When you plot the data, the signal trace uses the sample data, as the following example figure shows.
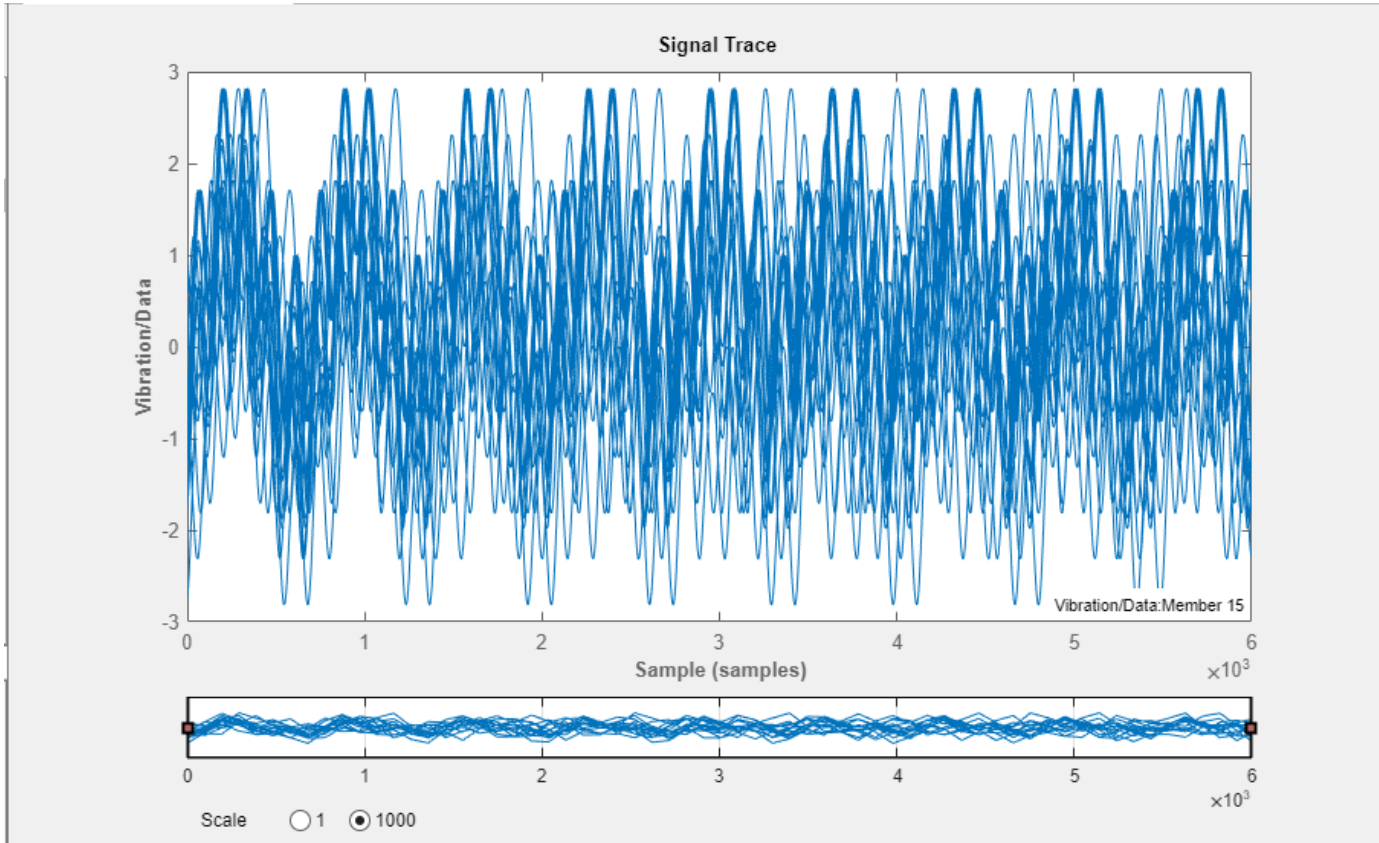
Import Ensemble Datastore

This workflow describes the steps for importing a `fileEnsembleDatastore` object or a `simulationEnsembleDatastore` object. Ensemble datastore objects provide information that allows the app to interact with external files. They include specifications for the variables you want to read, the variable types, and the source file locations. When you import an ensemble datastore, you can choose whether to store results within the app memory or write the results back to the ensemble datastore. For more information about ensemble datastores, see "Data Ensembles for Condition Monitoring and Predictive Maintenance" on page 1-2.

Select Data Source and View Source Variables

Initiate the import process. In the **Select dataset from workspace** pane, select the ensemble datastore. The app uses the ensemble datastore properties for `SelectedVariables` to select the variables to display. The app also uses the `DataVariables`, `IndependentVariables`, and `ConditionVariables` properties to determine which variables belong to which of these variable types. The example figure illustrates the import of a `simulationEnsembleDatastore` object `ens`.

In the preceding example figure, the app interprets `ens` as follows:

- The data variables `Flow` and `Pressure` appear identical in form to timetable-based variables extracted from tables.

- `ens` includes the standard `simulationEnsembleDatastore` variable `SimulationInput` in the `SelectedVariables` property. However, the app does not support the `SimulationInput` data format and displays an orange warning icon. The app also automatically clears the selection and deletes `SimulationInput` from `ens.SelectedVariables`.

- `CombinedFlag` appears as a condition variable in accordance with `ens.ConditionVariables`.

Choose How App Interacts with External Files

You can choose whether the app interacts with the external files referenced in your ensemble datastore. In the Select more variables pane, use **Append data to file ensemble** to specify your choice.
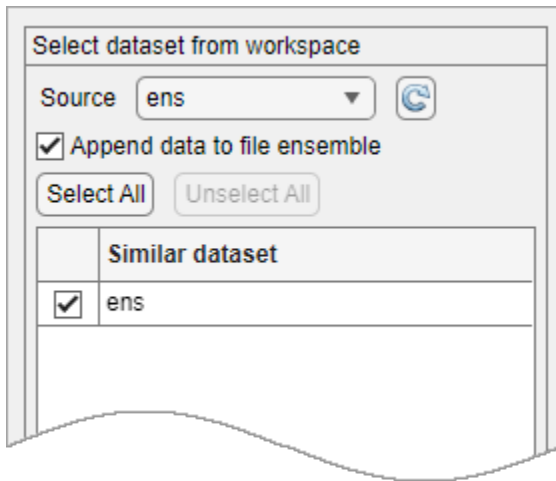
- If you select this option, the app interacts directly with the external files and writes results, such as derived variables or features, to the same folder as the original data. If you are using a `fileEnsembleDatastore` object, the object must include a reference to a `write` function that is specific to your data structure. You do not need a `write` function if you are using a `simulationEnsembleDatastore` object.

- If you clear this option, the app stores results in local app memory for the duration of the session. Select this option if, for example:

  - You want to keep your source files pristine at least until you have finalized your processing and feature generation.
  - You do not have write permission for your source files.
  - You do not have a `write` function, and you are using a `fileEnsembleDatastore`.
  - The process of writing the results back to the source files is slow.

  To retain your local results at the end of the session, use **Save Session**. You can also export your results as a `table` to the MATLAB workspace. From the workspace, you can store the results in the file, or integrate results selectively using ensemble datastore commands.

Import Labeled Signal Set (LSS)

This workflow describes the steps for importing an LSS, or `labeledSignalSet` object, from **Signal Labeler** in Signal Processing Toolbox. The signals you import must be full signals and not framed (segmented) signals.

Select Data Source and View Contents

Initiate the inport process. In the **Select dataset from workspace** pane, select the LSS. The app retains the LSS labels and interprets the variables as signals, condition variables, and features, according to the variable roles in the LSS and their data type.
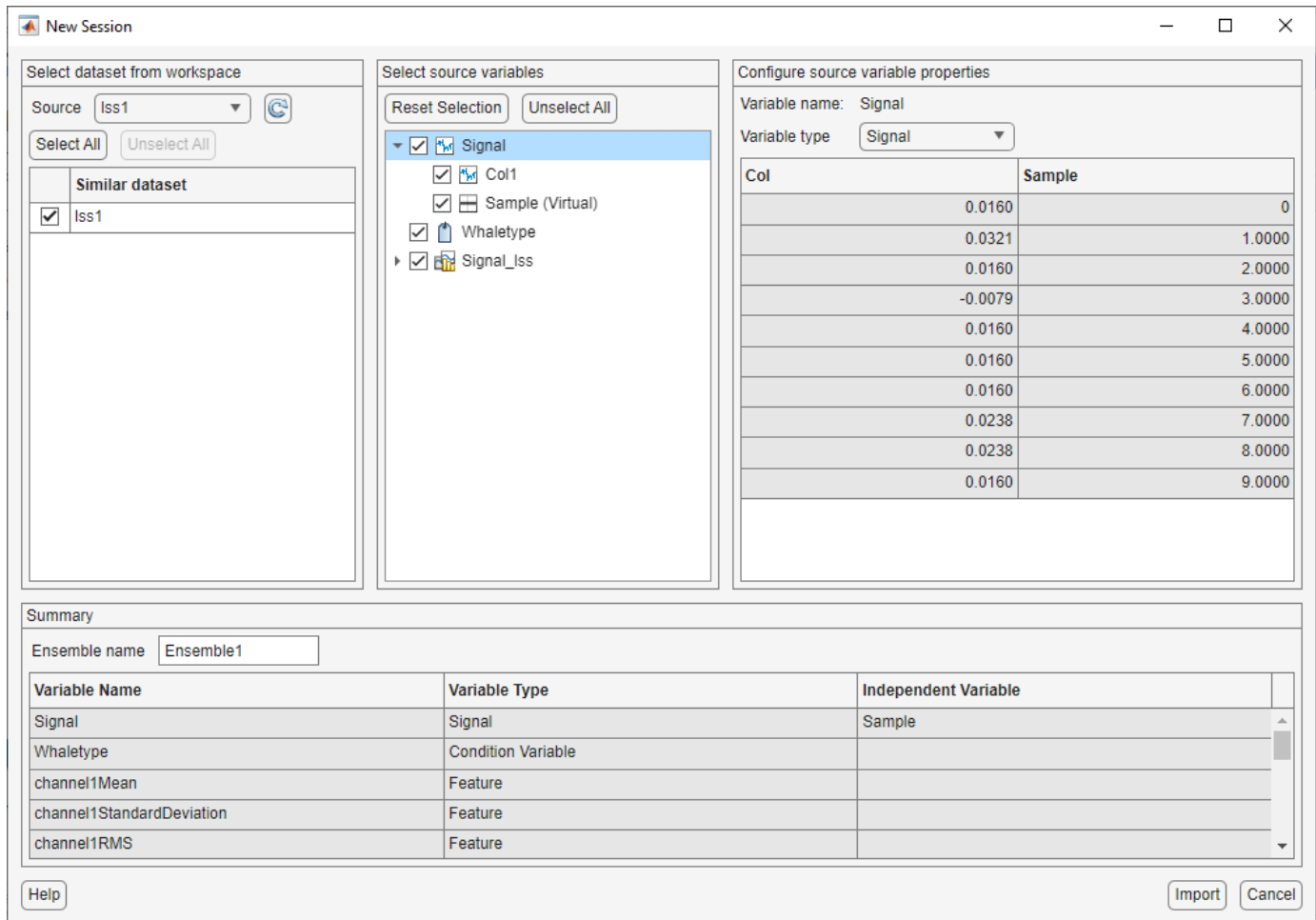
View Imported Variables and Features in Variables Pane

View the imported variables and features in the variables pane. The imported data retains the labels and the variable and feature designations.
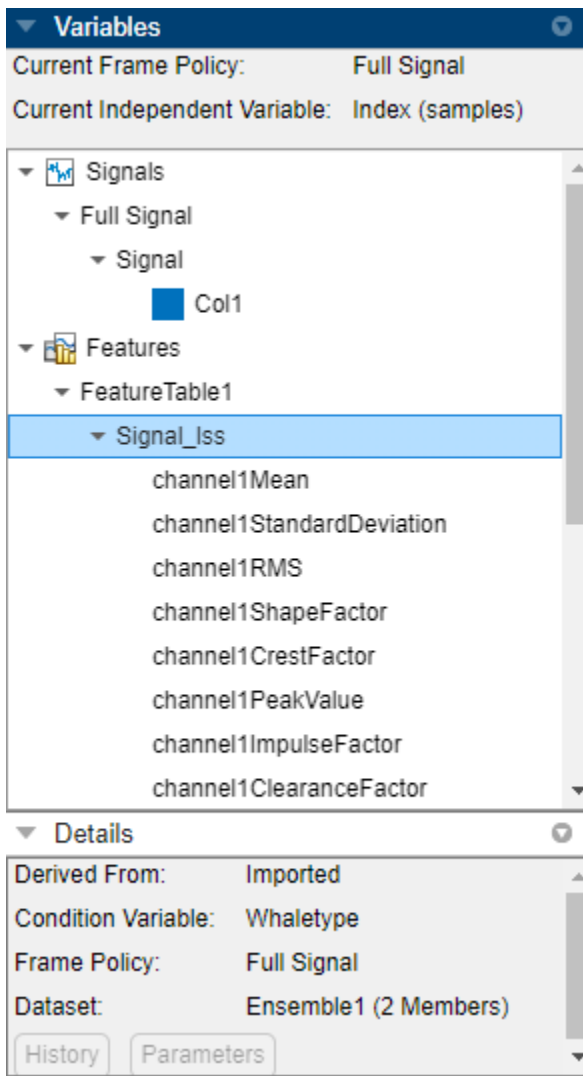
See Also

Diagnostic Feature Designer | `table` | `timetable` | `simulationEnsembleDatastore` | `fileEnsembleDatastore` | `idfrd` | `subset` | `labeledSignalSet`

Related Examples

- "Data Ensembles for Condition Monitoring and Predictive Maintenance" on page 1-2
- "Organize System Data for Diagnostic Feature Designer" on page 7-19
- "Import and Visualize Ensemble Data in Diagnostic Feature Designer"
- "Prepare Matrix Data for Diagnostic Feature Designer" on page 7-10
- "Data Preprocessing for Condition Monitoring and Predictive Maintenance" on page 2-2

Generate Features Automatically in Diagnostic Feature Designer

When you import data into **Diagnostic Feature Designer**, you can apply specific data processing and feature extraction options to generate and rank a feature set from your data. You can also generate and rank a feature set automatically using **Auto Features**. When you select one or more signals or spectra, **Auto Features** computes a predefined set of features that are appropriate for the variable type. The automatic computations include:

- Deriving intermediate variables to use for feature extraction, such as spectra and time series signals
- Extracting the features from the expanded variable set
- Ranking the features and plotting the histograms of top-ranked features.

With **Auto Features**, you can generate both general and rotating machinery features.

- General features are standard metrics that can be applied to any signal (for signal features) or spectrum (for spectral features). The software uses default parameters for these features.
- Rotating machinery features are derived from specialized computations that incorporate rotation rate.

The following figure illustrates the computation flow for general and rotating machinery features.
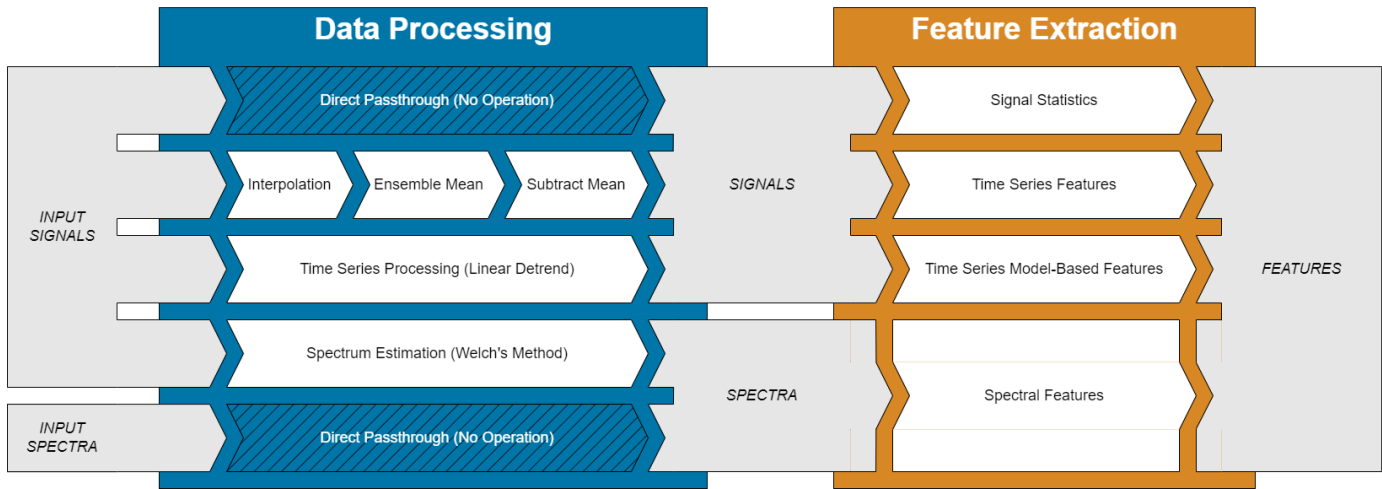
General

Rotating Machinery
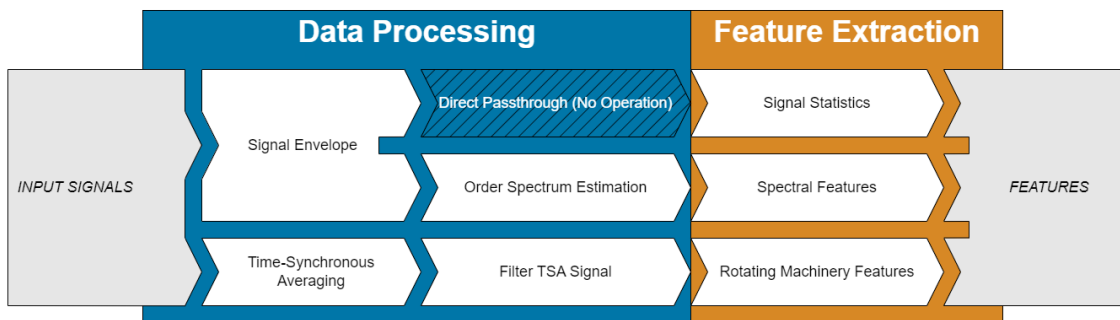
To use **Auto Features**, perform the following general steps:

1 Select a variable, such as a signal or spectrum, in the **Variables** pane. To select more than one variable, use **Ctrl**-click to add to the variable selection. The variables do not have to have the same data type.

2 In the **Feature Designer** tab, click **Auto Features**.

3 Confirm the computational settings in the **Configuration** pane of the **Auto Features** dialog box.

4 In the **Features to Generate** pane, select **Standard** for general features, **Rotating Machinery** for rotating machinery features, or both.

5 In the **Settings** pane, specify general and, if you are selecting rotating machinery features, rotating machinery information.

- **General** tab: choose whether to plot the histograms and specify the number of top-ranked features to plot.

- **Rotating Machinery** tab:

    - Specify the rotation rate as a constant RPM value or as a tacho signal from your variables list. if you specify a tacho signal, also specify the number of pulses per rotation

- Specify the TSA filter domain as either order or frequency and provide the orders of rotation in the machinery.

6 Click **Compute**.

Once you have generated the feature set, you can continue to derive new variables and add new features. Alternatively, you can immediately export the feature set directly to **Classification Learner**.

The following sections provide more information about the general workflow.

Set Up Auto Features Computation

To set up an **Auto Features** computation, select one or more variables and, in the **Feature Designer** tab, click **Auto Features**.
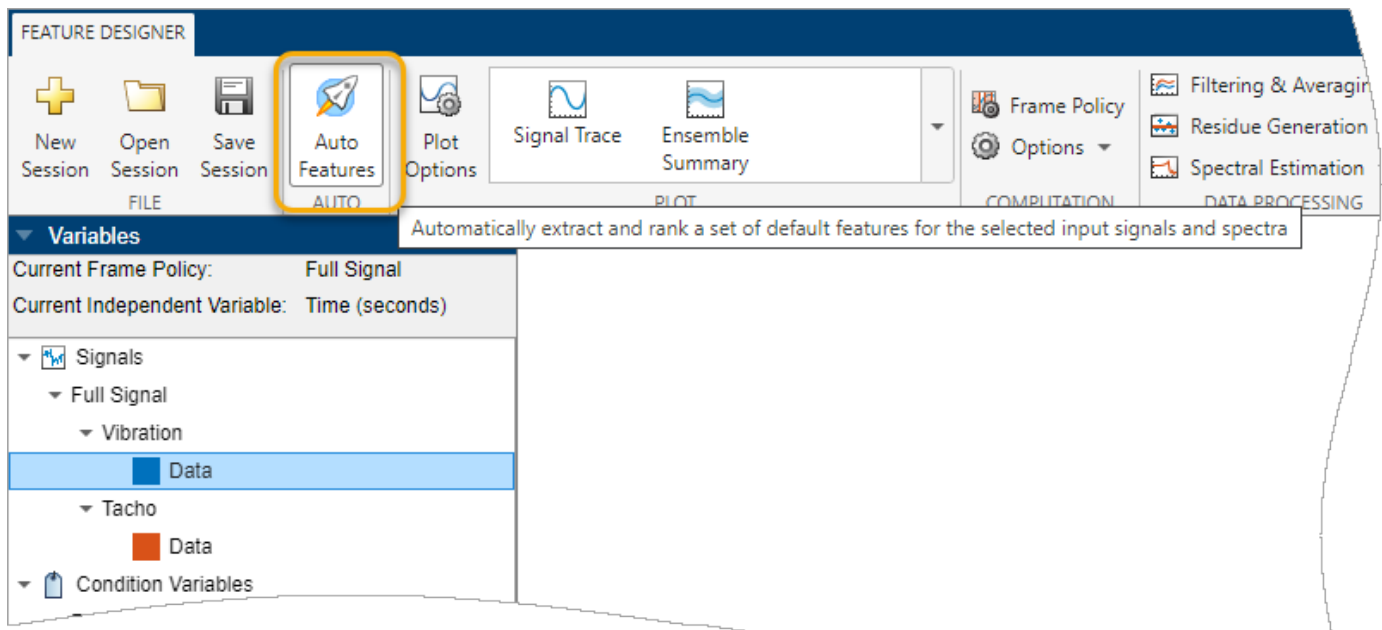
The **Configuration** pane of the **Auto Features** dialog box displays information that includes the selected variables, the feature table to add features to, and the number of features to generate. In the following example figure, a selection of a specific signal results in the app identifying 108 time-domain features and 6 frequency-domain features for generation. These features include both general and rotating machinery features.
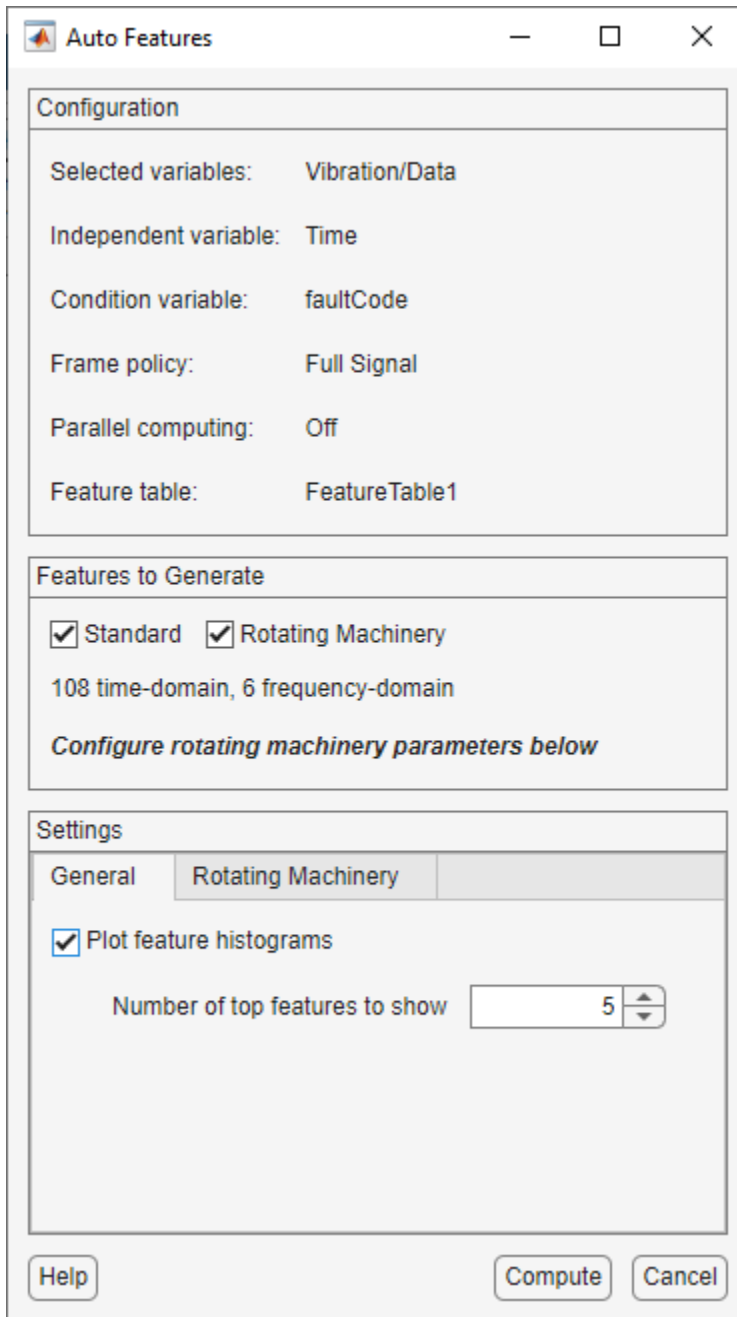
The **Configuration** pane also lists the status of computational options such as parallel computing. If you want to change any of these options or if you want to add an additional variable for feature generation, click **Cancel**, and make the necessary changes in the app options and variable selections. Then click **Auto Features** once more to proceed.

The **General** tab of the **Settings** pane lets you choose to plot the histograms automatically after the features are generated and ranked. To do so, select **Plot feature histograms**, and then specify the number of top features to show.

The **Rotating Machinery** tab of the **Settings** pane lets you specify the rotation rate and filter domain. In the following example figure, the rotation rate is set to the tacho signal `Tacho/Data` and the TSA filter domain is set to `Order`, with a rotation order of 1.

Once you are satisfied with the configuration and the settings, click **Compute**.

View New Variables and Ranked Features

When the feature computations are complete, the app plots the feature ranking. The **Variables** pane includes new derived variables that the app computed to support feature generation. For example, in the following figure, two new derived spectrum variables provide sources for spectral features.

You can use `Compact view` to view more variables in the same space. Right-click on any variable to access the compact view option. In the following figure, selecting `Compact view` collapses each variable to the main variable name.
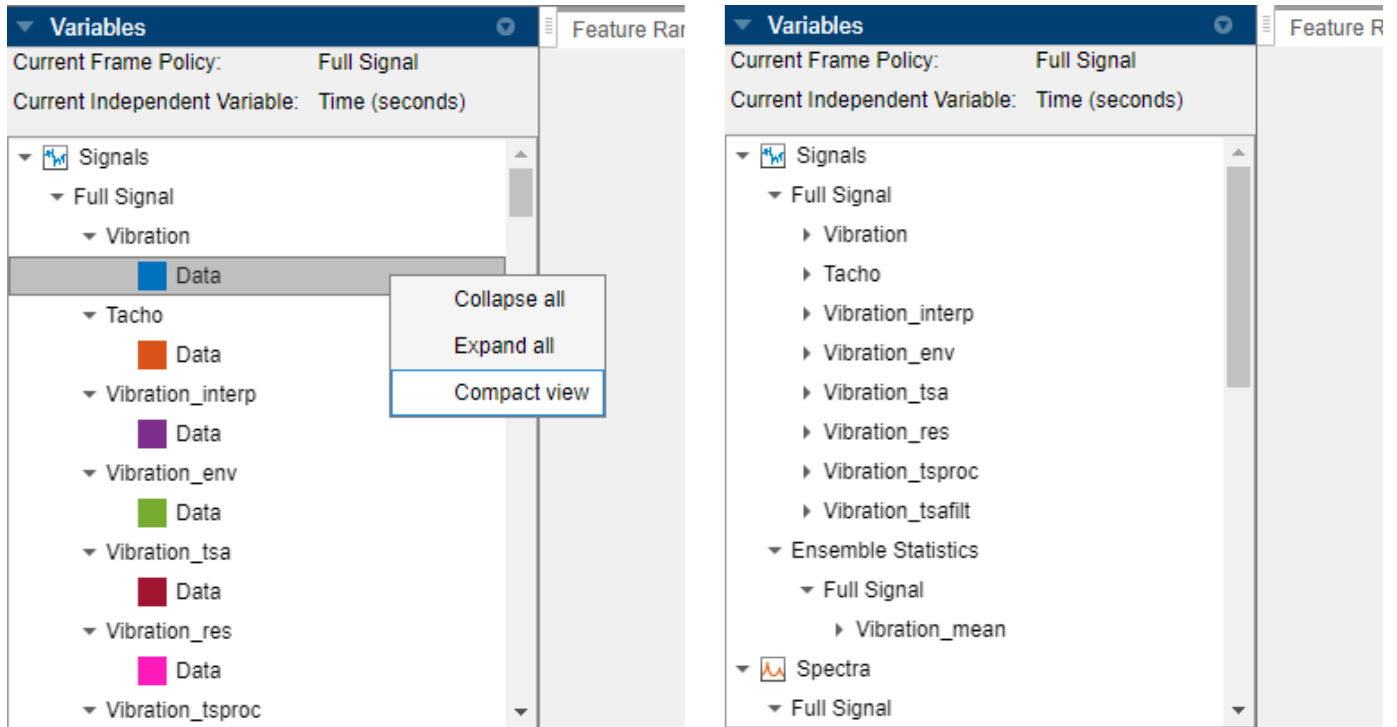
The derived variable names include the last processing step that was used to compute them. For example, in the following figure, `Vibration_res` is a residual signal that is computed by subtracting a reference signal such as the mean value of the signal ensemble.
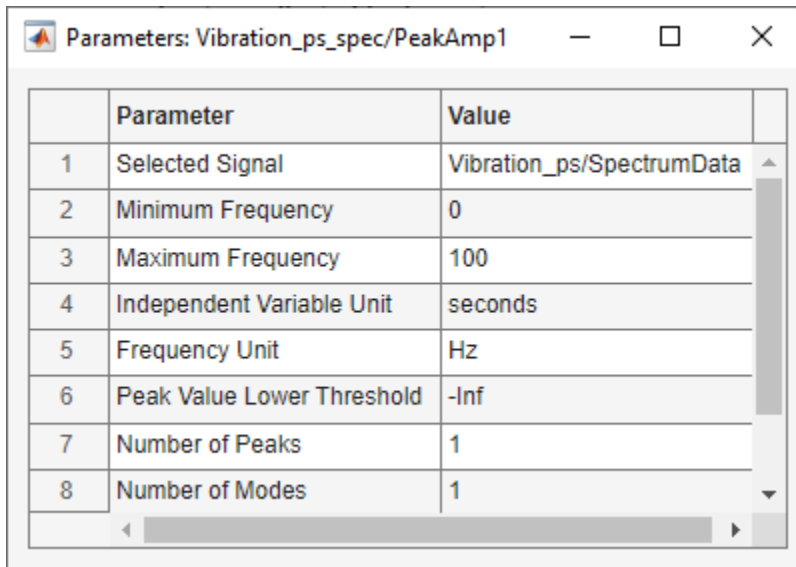
To view the generated features, scroll down in the **Variables** pane. To obtain more information about a feature, such as what variable the feature is derived from, select the feature and then view the feature information in the **Details** pane. You can also use the **Details** pane for information on variables. In the following figure, the **Details** pane shows that the `ClearanceFactor` feature from the `Vibration_res_sigstats` feature group is derived from the `Vibration_res` signal.
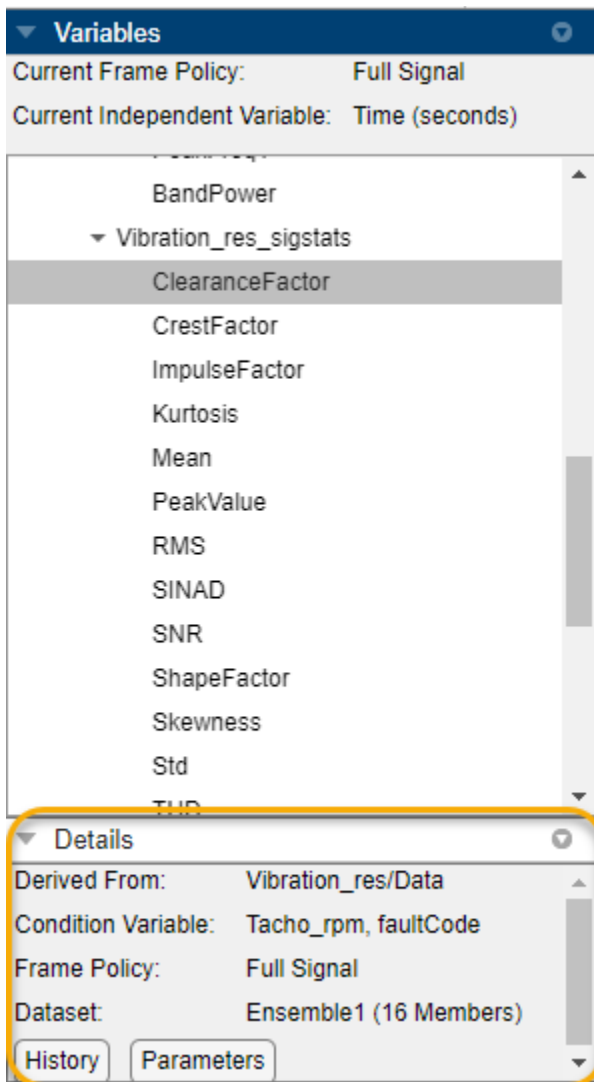
To view more information about the processing history of a feature, click **History**. The following figure illustrates the sequence of serial and parallel processing steps that led to `ClearanceFactor`.
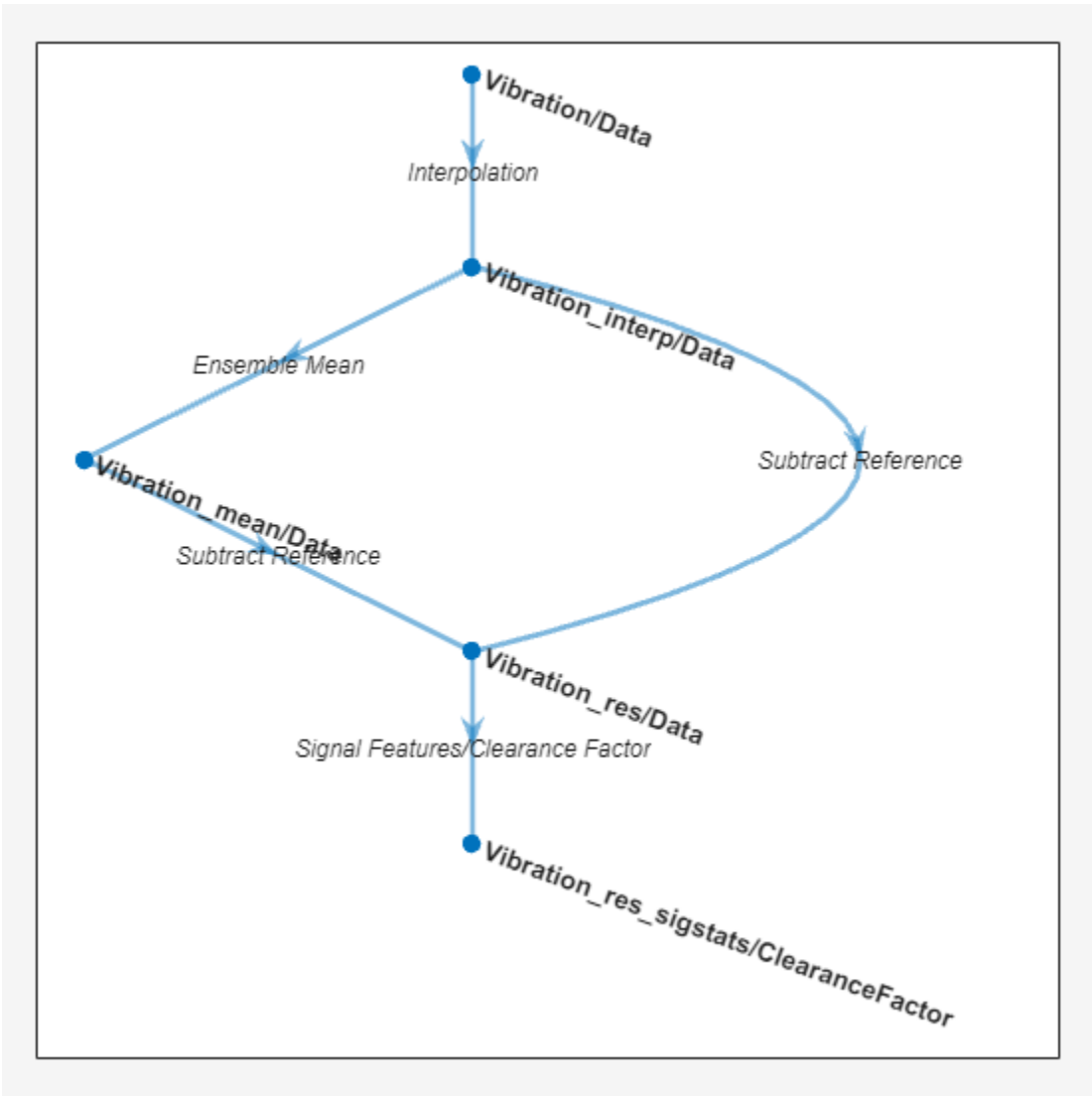
For some features, the app uses tunable parameters in the computation. To view the parameter values, click **Parameters**. The following figure shows the parameter values for the spectral feature `PeakAmp1`.

If you selected the option to plot histograms automatically, you can view them by selecting the **Histogram** plot tab next to the **Feature Ranking** plot tab.

Add Additional Variables and Features

After you generate your first set of features using **Auto Features**, you can continue to select variables for automatic or manual processing and feature extraction. If you apply **Auto Features** to variables that **Auto Features** previously derived, the new features will be duplicates. However, if you create new variables yourself, **Auto Features** generates distinct features.

For example, the default spectral processing option that **Auto Features** uses is **Welch**. Suppose that you create a separate spectrum variable using the **Autoregressive** option. Select that new spectrum, `Vibration_ps_1`, for **Auto Features** to see that there are three available features. Since **Auto Features** did not use this spectrum variable to generate the initial feature set, these features are distinct from the spectral features that **Auto Features** generated previously.
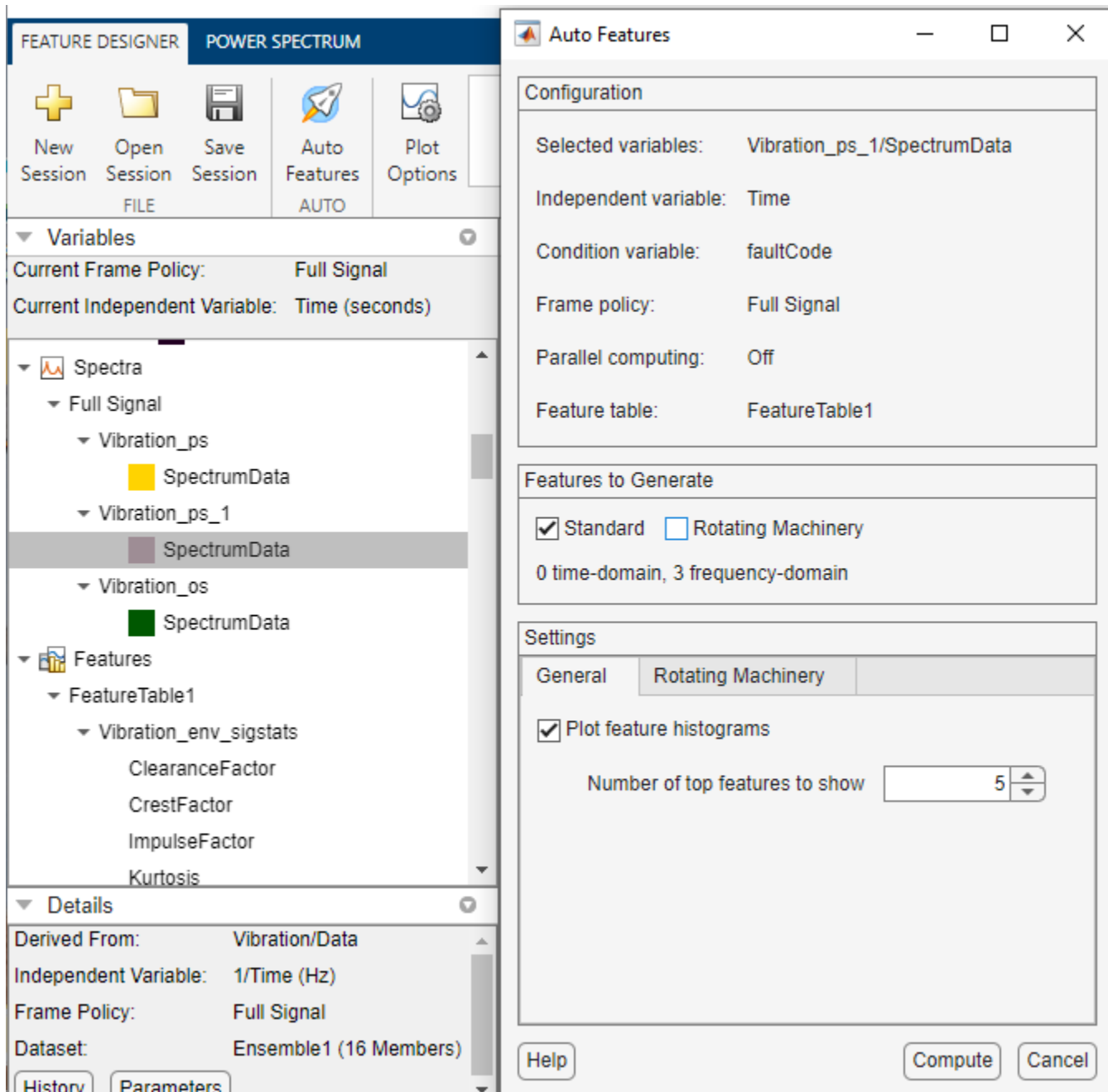
The new features are ranked along with the existing features. In the following figure, two of the new spectral features are ranked eighth and ninth.
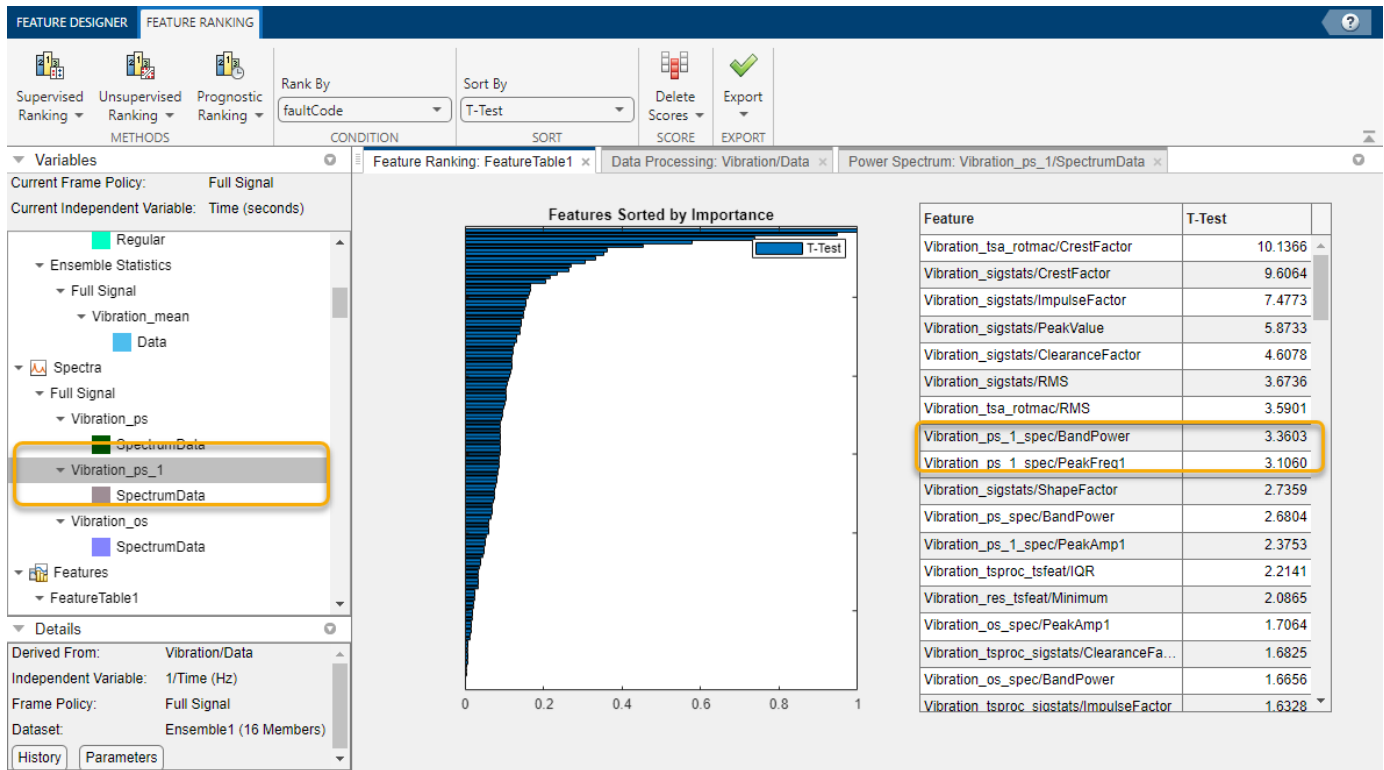
Because **Auto Features** dialog box for this example is set to the default value of 5, the set of histogram plots for `FeatureTable1` does not include the new features. If you decide you want to see additional histograms, select the feature table. Then, in the **Histogram** tab, click **Select Features**.
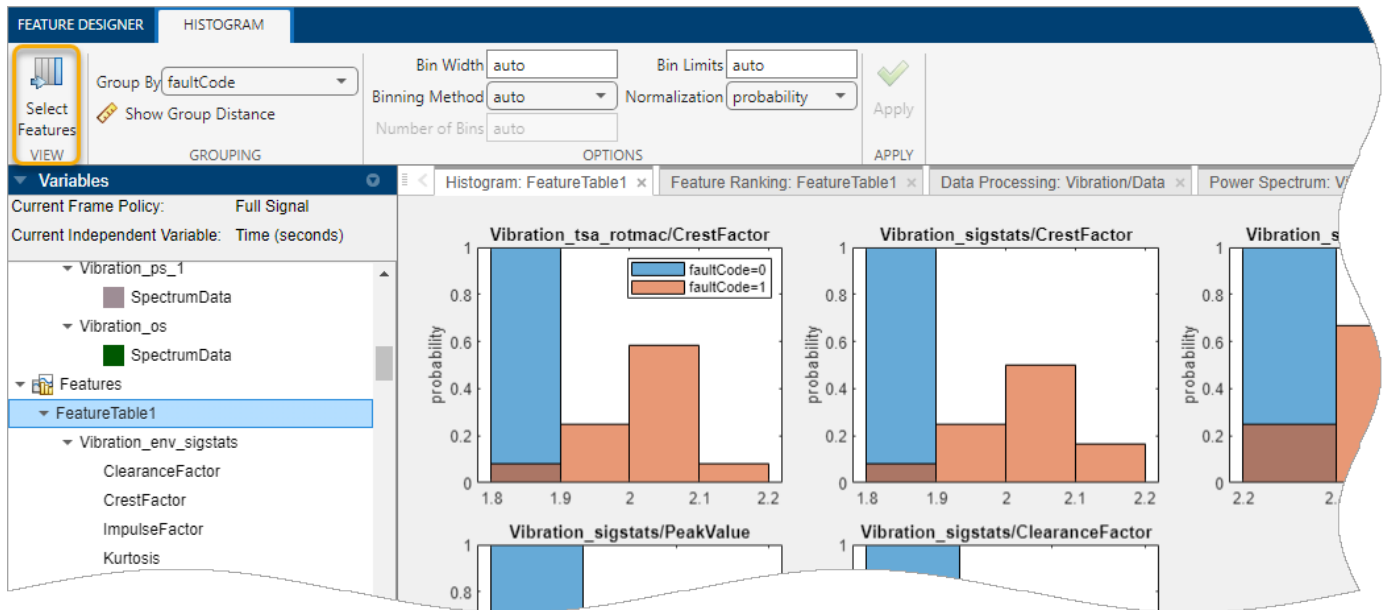
A dialog box that displays the features in ranked order allows you to select additional features.
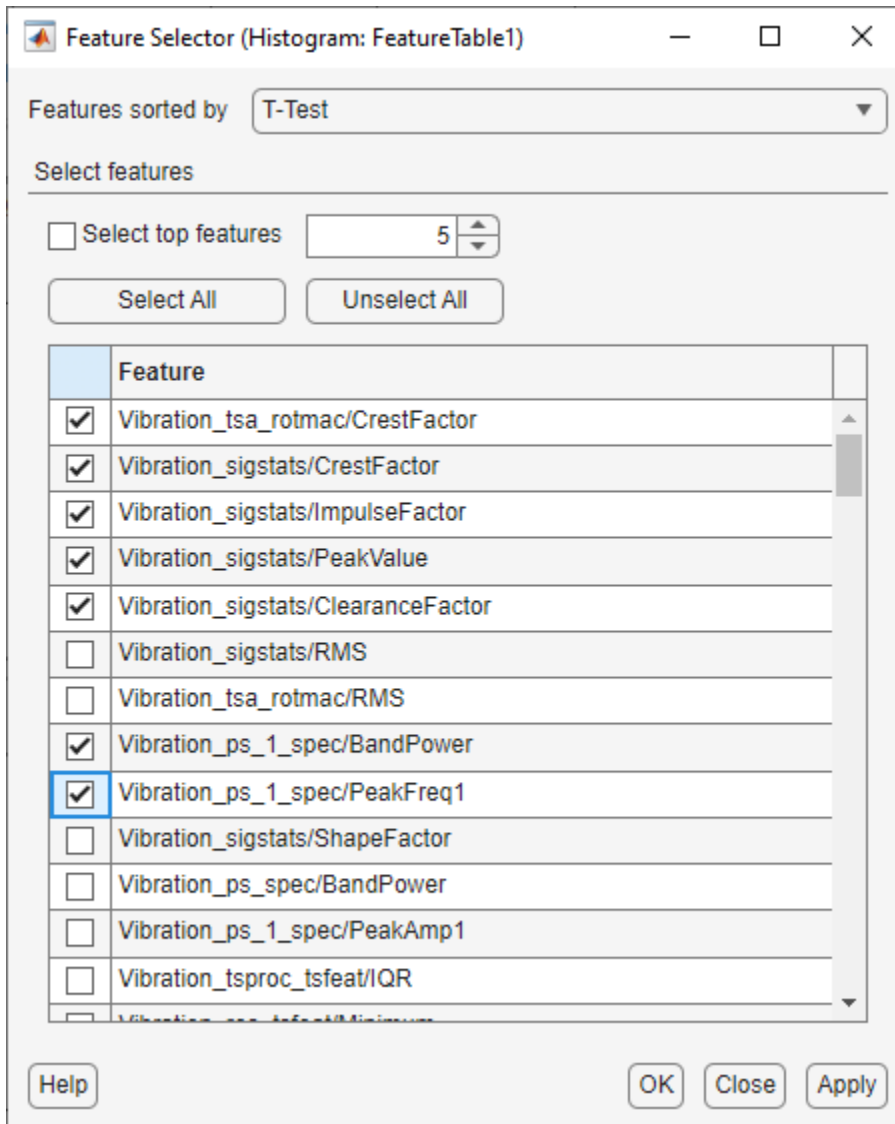
For more information on **Select Features**, see "Feature Selector" on page 8-33.

Next Steps

You can now continue to process your variables and add new features. You can also export your feature set to your MATLAB workspace or to **Classification Learner**.
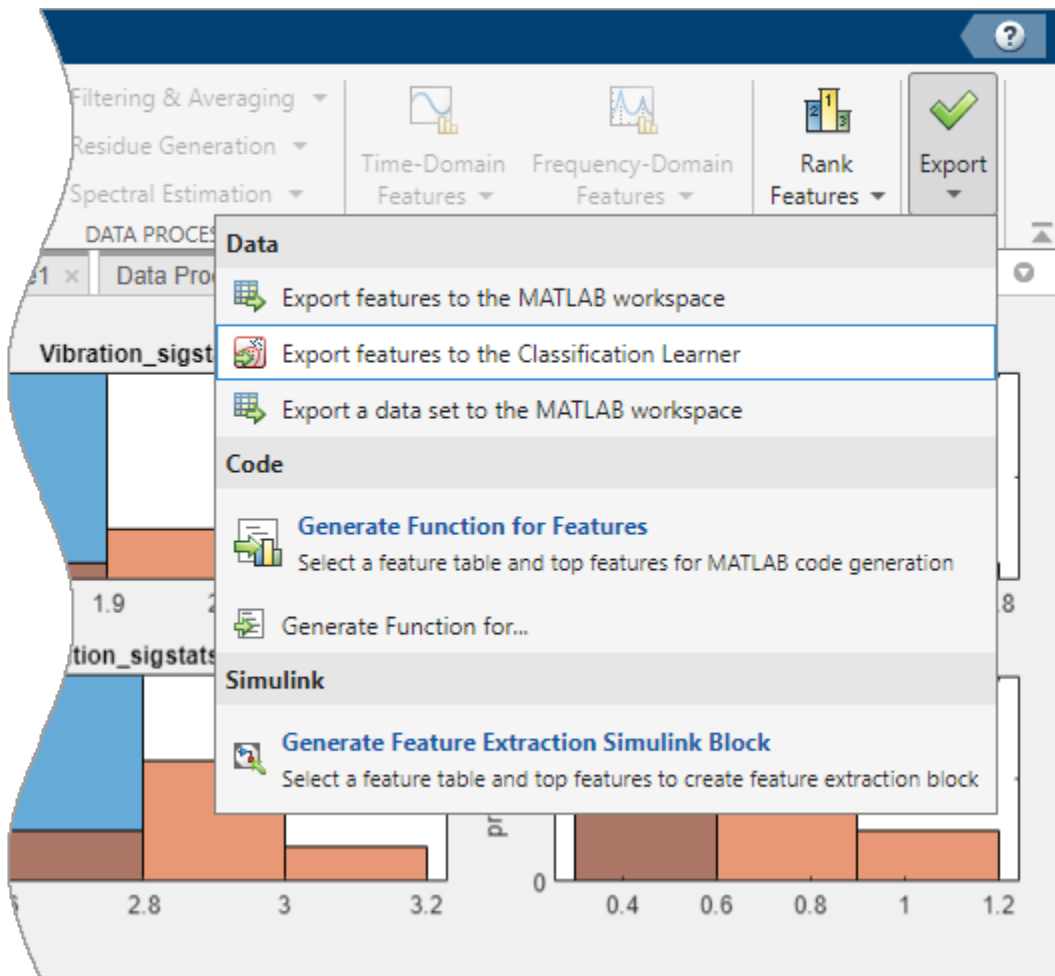
See Also
Diagnostic Feature Designer

Related Examples

- "Identify Condition Indicators for Predictive Maintenance Algorithm Design"
- "Condition Indicators for Gear Condition Monitoring" on page 3-10

Create Custom Features in Diagnostic Feature Designer

Diagnostic Feature Designer provides a set of built-in processing functions that allow you to extract features from your ensemble data. You can also add additional functions to extract custom features and work with these features just as you would with any other features in the app. For example, your custom feature processing might include:

- Existing command-line functions that the app does not incorporate, such as `meanfreq`
- Specialized functions that combine a number of operations

The one requirement is that the custom feature syntax follows the following form.

```
function Feature = myCustomFunction(DataVariable,IndependentVariable,varargin)
```

When you select a variable to process, such as a variable named `Vibration` that contains a data variable and a time variable, the custom function uses these two variables as the first two input arguments. You can specify additional arguments in `varargin` to support the processing or to select specific outputs.

To use **Custom Features**, perform the following general steps:

1 Select a variable, such as a signal or a spectrum, in the **Variables** pane.
2 Select either **Time-Domain Features** > **Custom Features** or **Frequency-Domain Features** > **Custom Features**.
3 In the **Custom Features** tab, click **Add Function**.
4 In **Create New Function**, provide the name you want for the new function. The app brings up a template for you to create it. If the function already exists, provide that in **Add Function**.
5 Use **Check Function** to test your function on the first member of your ensemble.
6 Set **Plot** to select whether to automatically plot histograms when you apply the function.
7 Click **Apply**.

The app adds the custom features to the feature table alongside built-in features that you compute and includes them when performing feature ranking and when exporting features to **Classification Learner** or to your MATLAB workspace.

General Workflow Description

Add Custom Function

Start by selecting a signal or spectrum from the **Variables** pane. Then, select either **Time-Domain Features** > **Custom Features** or **Frequency-Domain Features** > **Custom Features**. In the following figure, the **Vibration** signal is selected, so the available **Custom Features** option is in **Time-Domain Features**.
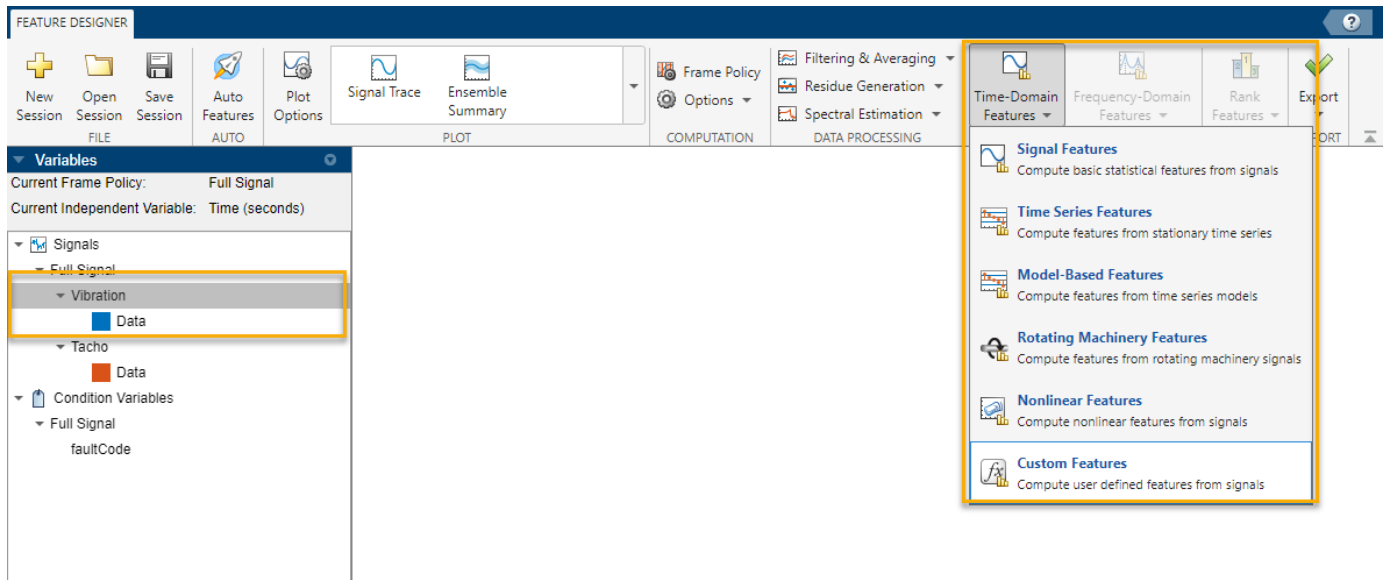
The **Custom Features** tab includes the operations for adding a function, checking the function, and, finally, applying the function and creating the new features.
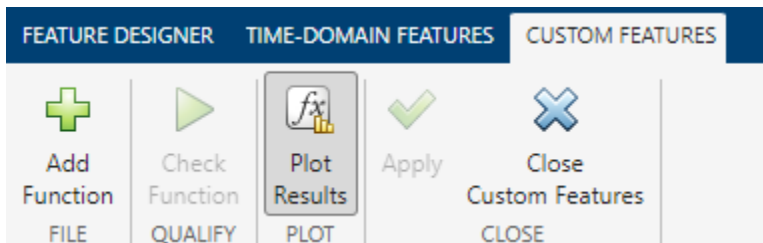
Select **Plot Results** if you want the app to automatically plot histograms once you have applied the function. To initiate the process of adding a function, click **Add Function**. The app then brings up a dialog box that lets you start the process of creating a function by specifying a name. You can also add an existing function if that function already exists on your path. In the following figure, a new function named `customZeroCrossings` will be created.
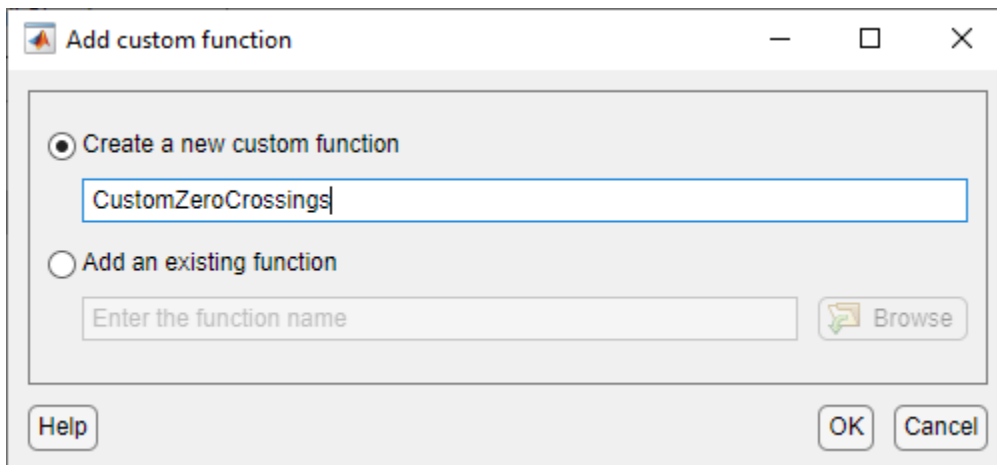
Clicking **OK** creates a new template script.

```
function Feature = CustomZeroCrossings(DataVariable,IndependentVariable,varargin)
% Input:
% DataVariable: Selected Signal
% IndependentVariable: The current independent variable associated with the selected signal
% varargin: The additional input arguments to the function
% Output:
% Feature: A numeric scalar or vector value
Feature = [];    %Assign the computed feature value here
end
```

All custom functions use the same basic argument list: the data variable (for example, `Vibration/Data`), the independent variable (for example, `Vibration/Time`), and any additional optional arguments that the function requires. For examples of typical custom functions, see "Examples of Custom Feature Functions" on page 7-172. Edit the script to perform the custom feature processing that you want.

Test Custom Function

To qualify your new function, click **Check Function**. The app applies the processing only to the first member of your ensemble to verify that the feature can be computed without an error. If the function succeeds, the app shows a **Successfully qualified** message that contains the feature value for the first member or first frame. The app does not save this qualification value for any future use.
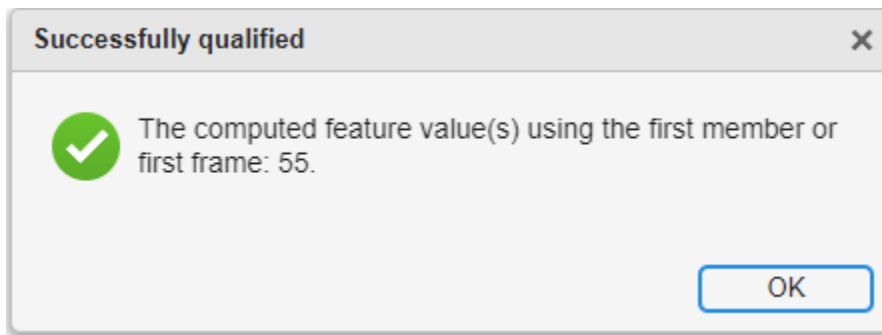
If the function fails, the app shows an error message.
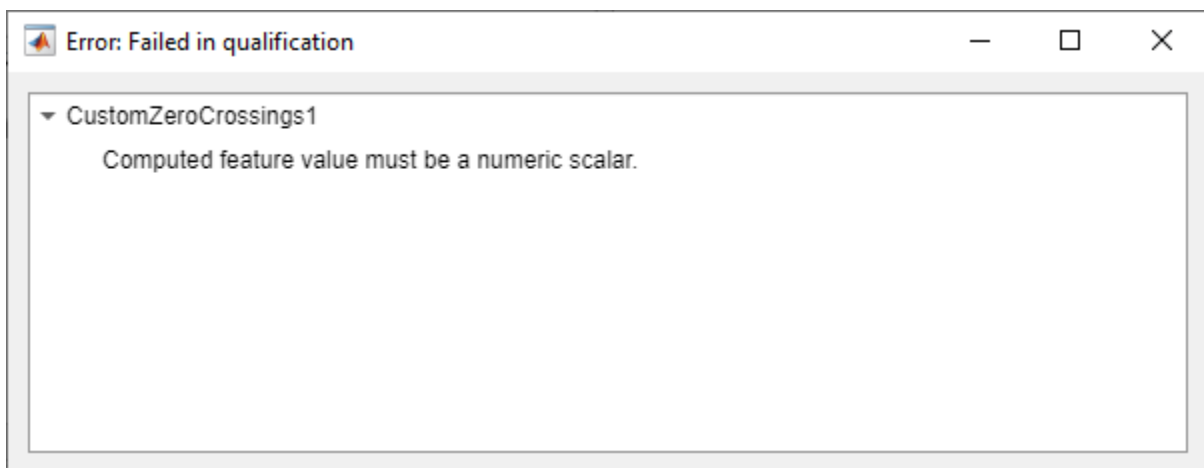
Apply Custom Function

Once your function test is successful, you can click **Apply** to generate your new feature and, if you have selected **Plot**, plot the histogram. In the following figure, the new feature has been added to `FeatureTable1`, and the histogram shows the distribution.
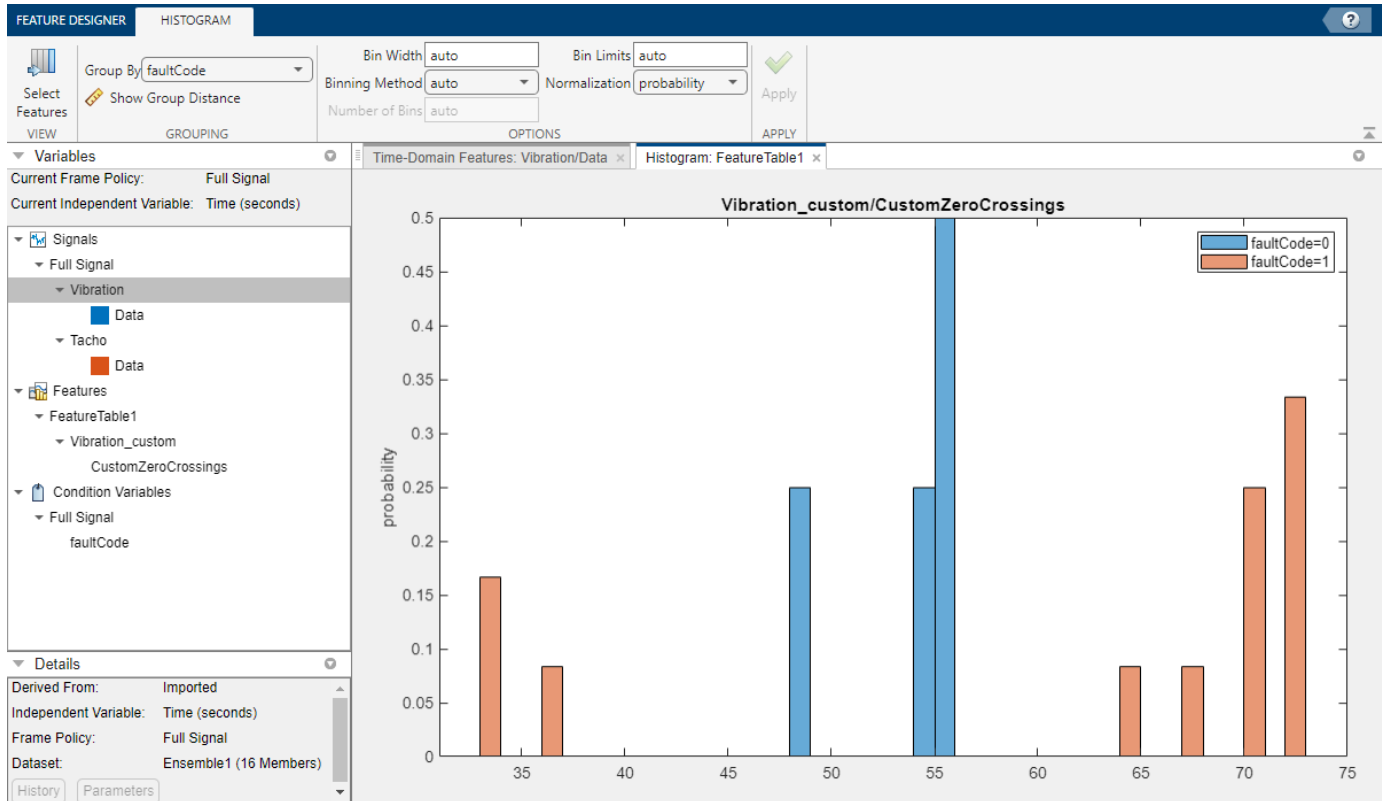
Use Custom Feature Table to Record Feature Information and Specify Arguments and Names

The app maintains a record of the custom functions you have added in a table that you can access by returning to the same **Custom Feature** tab that you used to create the feature. The table includes the function name, the optional arguments, the name of the feature that the app generates, and a description of the feature. You can modify any of the fields but the function name.

- In **Optional Arguments**, you can provide additional information that your function requires such as a constant or method. You can also add multiple versions of the same function, but change the feature computation by specifying different argument values for the arguments for each field.

- For the **Generated Feature Name**, you can specify the name that the app assigns to the feature.

- In **Description**, you can capture any information that pertains to the custom function.

The following table illustrates a custom feature table with one entry. Select the row to perform operations such as editing the function script or deleting the function from the table. You can also create a new custom function from the feature table options. Select the check box to select the function for computation.

From the custom feature table, you can also select multiple functions to compute multiple features at once, add new features, edit features that are already in the table, and delete features from the table.

The custom feature table lets you specify arguments to pass to the function. In the following figure, the custom function `myCustomFunc` computes the number of zero crossings when the argument is `Crossing` and the number of sign changes when the argument is `Sign`. The generated feature name is unique for each argument specification.

The app stores custom signal features and custom spectral features in separate tables. To restore the time-domain **Custom Feature** tab and custom feature table, select any signal variable in the **Variables** pane and select **Time-Domain Features > Custom Features**. Similarly, for spectral features, select any spectrum variable and then select **Time-Domain Features > Custom Features**.

You can restore the feature table only during the session in which you create it. Saved sessions retain all the features you computed, but not the table itself.

Examples of Custom Feature Functions

Possible custom feature functions range from direct use of existing MATLAB functions to multipurpose functions that can provide a variety of feature types depending on what optional arguments you specify.

Use MATLAB Functions Directly

Some MATLAB functions already use the syntax that custom feature functions require. For example, consider the function `meanfreq` in Signal Processing Toolbox. This function uses the syntax `freq = meanfreq(x,fs)`. In this syntax, `x` is the spectral data variable and `fs` is the independent variable. You can use this function to generate frequency-domain custom features simply by typing `meanfreq`, as the following figure shows.
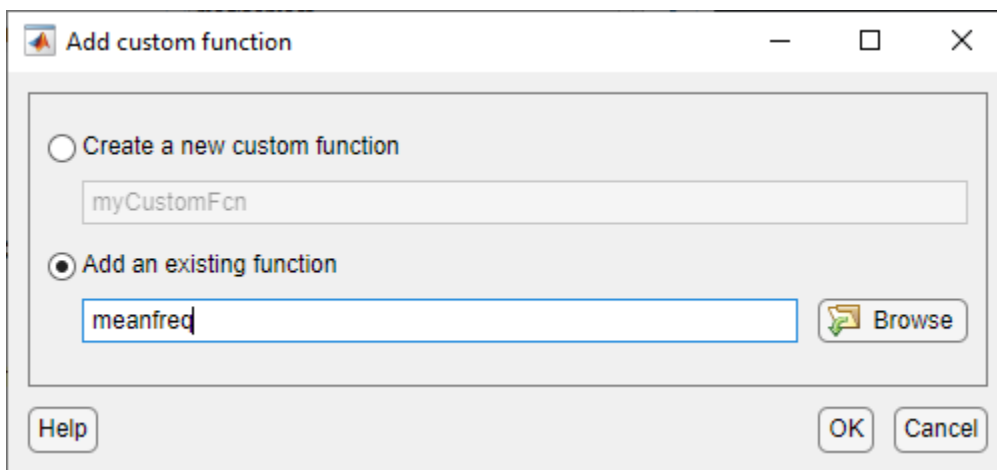
Create Custom Features That Require Only Data Variables

Although the custom feature template includes arguments for both the data variable and the independent variable, you can create functions that require just the data variable. For example, the following code creates the feature `customZeroCrossings`, which requires only signal data and not time. Although the function syntax includes all arguments, the feature value calculation uses only the signal data. This code excludes the template text comments, but you can choose whether to include them in or exclude them from your function.

```
function Feature = CustomZeroCrossings(DataVariable,IndependentVariable,varargin)
zcd = dsp.ZeroCrossingDetector;
Feature = double(zcd(DataVariable));    %Assign the computed feature here
end
```

Create Custom Features That Require Both Data Variables and Independent Variables

The following code example shows a more complex custom feature, `mySignChange`, that uses both the signal values and the time.

```
function Feature = mySignChange(DataVariable,IndependentVariable,varargin)
timeVector = time2num(IndependentVariable, "seconds");
dt = diff(timeVector);
dx = diff(DataVariable);
slope = dx./dt;
slope_sign = sign(slope);
Feature = numel(find(diff(slope_sign)));
end
```

Use Additional Arguments

You can use the optional arguments to provide special data constants or to select processing options. In the following function, myCustomFunction, you can compute either slope sign changes or zero crossings, depending on whether you set the optional argument to Sign or Crossing.

```
function Feature = myCustomFunc(DataVariable,IndependentVariable,varargin)

% Extract underlying additional input from varargin
if ~isempty(varargin)
    FeatureType = varargin{:};CustomZeroCrossing1
    FeatureType = FeatureType{:};
end

% Compute feature based on the FeatureType that has been passed in as input
switch FeatureType
    case {'Sign','sign'}
        timeVector = time2num(IndependentVariable, "seconds");
        dt = diff(timeVector);
        dx = diff(DataVariable);
        slope = dx./dt;
        slope_sign = sign(slope);
        Feature = numel(find(diff(slope_sign)));
    case {'Crossing', 'crossing'}
        zcd = dsp.ZeroCrossingDetector;
        Feature = double(zcd(DataVariable));
end
end
```

To compute both types of feature from this code, add the feature to custom feature table twice, using the first argument for the first entry and the second argument for the second entry. Then select both and click **Apply**.

See Also
Diagnostic Feature Designer

Extract Signal Envelopes

When you import high-bandwidth signal data into **Diagnostic Feature Designer**, you can extract low-bandwidth signal envelopes from the signal data to isolate low-frequency vibration modulation near machine resonances. This low-frequency signal can be useful for detecting faults in rotating machinery.

Parameter Settings and General Workflow

The **Signal Envelope** tab, which you access from the **Filtering & Averaging** menu, provides the parameters for extracting the envelope of a selected signal. You can extract the envelope using the full bandwidth of the signal. However, results are often better if you first apply a bandpass filter to the signal to isolate the signal components near the resonances.

The following paragraphs describe the available parameters and controls.

Envelope Settings

- **Envelope** — Choose Upper or Lower.
- **Method** and method parameter

  - **Hilbert** — No parameter
  - **FIR** — **Filter Order**
  - **RMS** — **Window Length** in samples or time units
  - **Peak** — **Maximum Separation** in samples or time units

Bandpass Filtering

- **Use Bandpass Filter** — Select to specify bandpass filtering
- **Center Frequency** — Center frequency of the bandpass filter in selectable units
- **Bandwidth** — Bandwidth of the bandpass filter in selectable units

You can iterate on your settings and preview the results without creating a new variable by clicking **Preview**. The app shows the envelope corresponding to the current settings alongside the original signal so that you can visually compare the two plots.

When you are satisfied with shape of the envelope, click **Apply** to create the new envelope variable. You can process and extract features from the signal envelope as you would with other time-domain signals. If you have selected **Plot Results**, then the app generates an envelope plot.
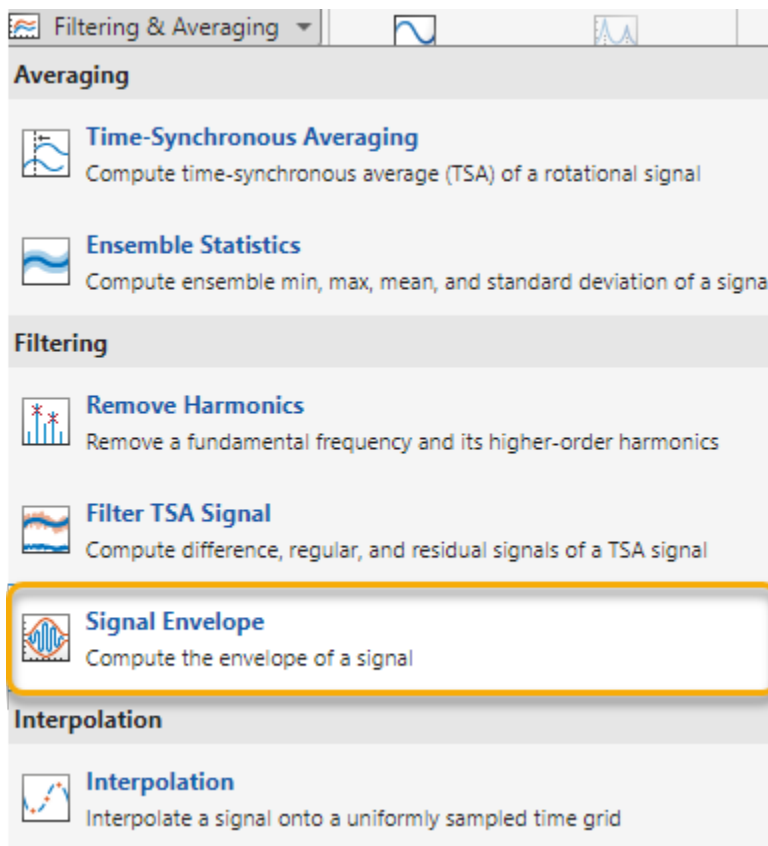
Extract a Signal Envelope

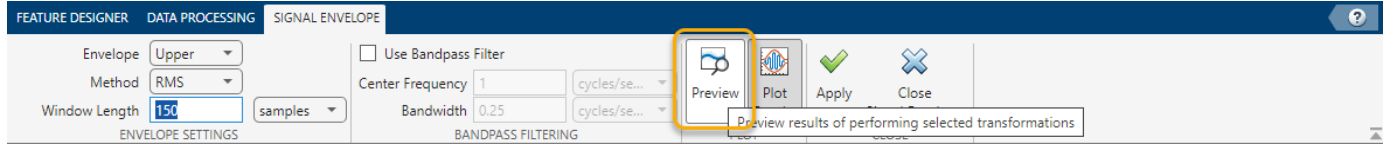As an example, consider the following single signal which has been imported into the app.
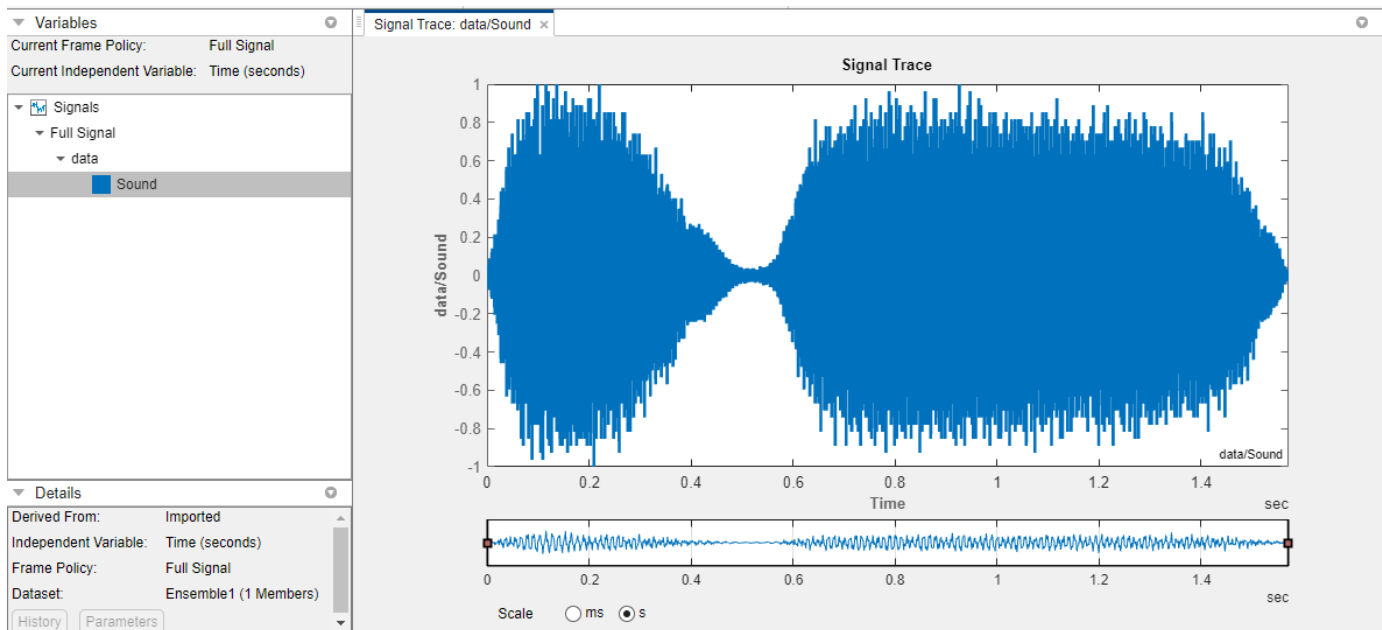
Select Signal Envelope Option

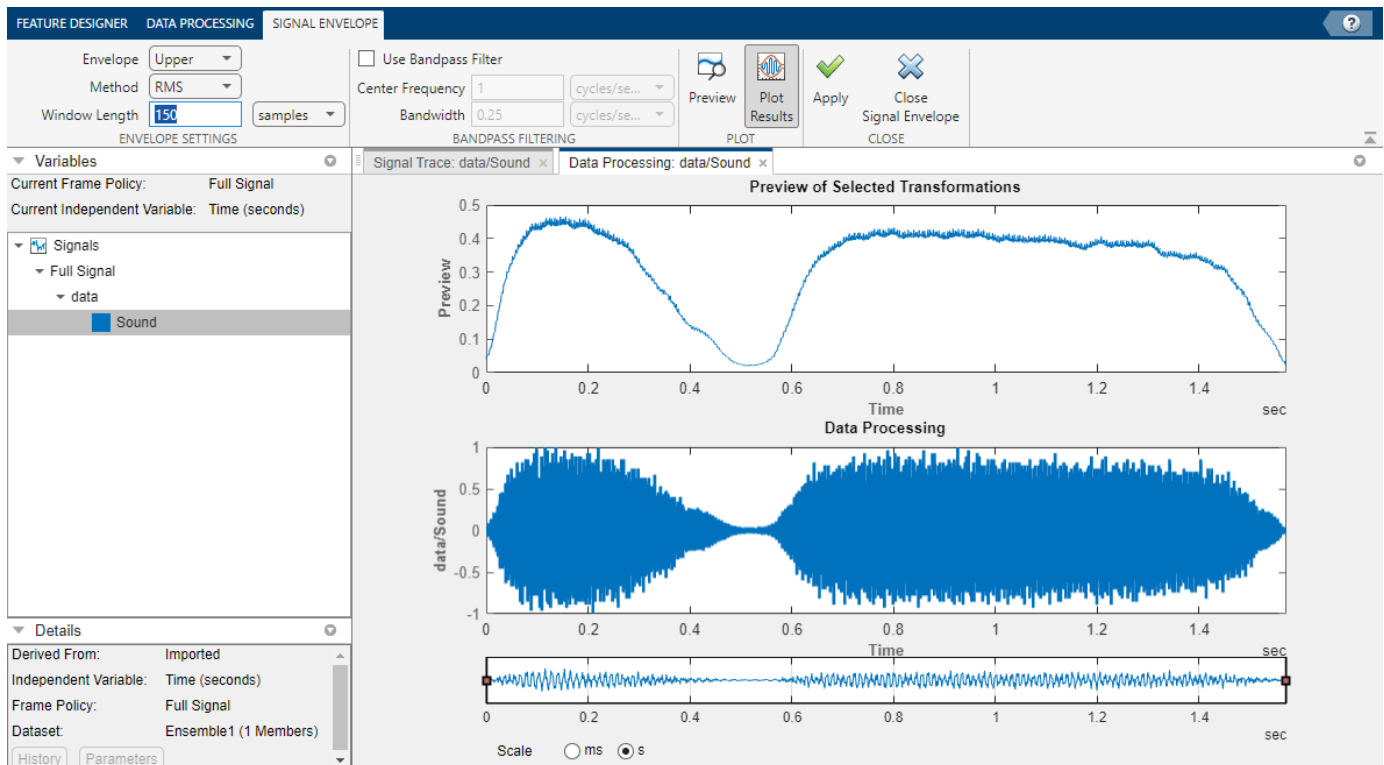From the **Filtering & Averaging** menu, select the `Signal Envelope` option.

Specify and Preview Full Bandwidth Signal

Specify a signal envelope using the full bandwidth of the signal and click **Preview**.
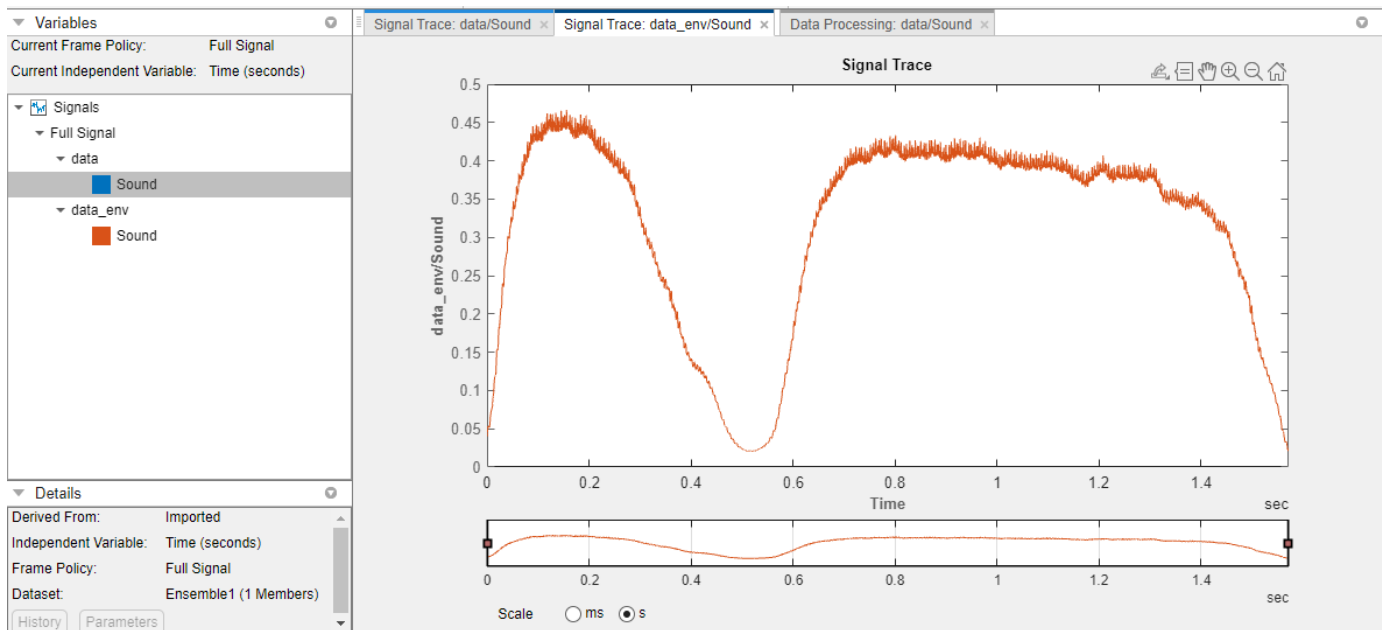

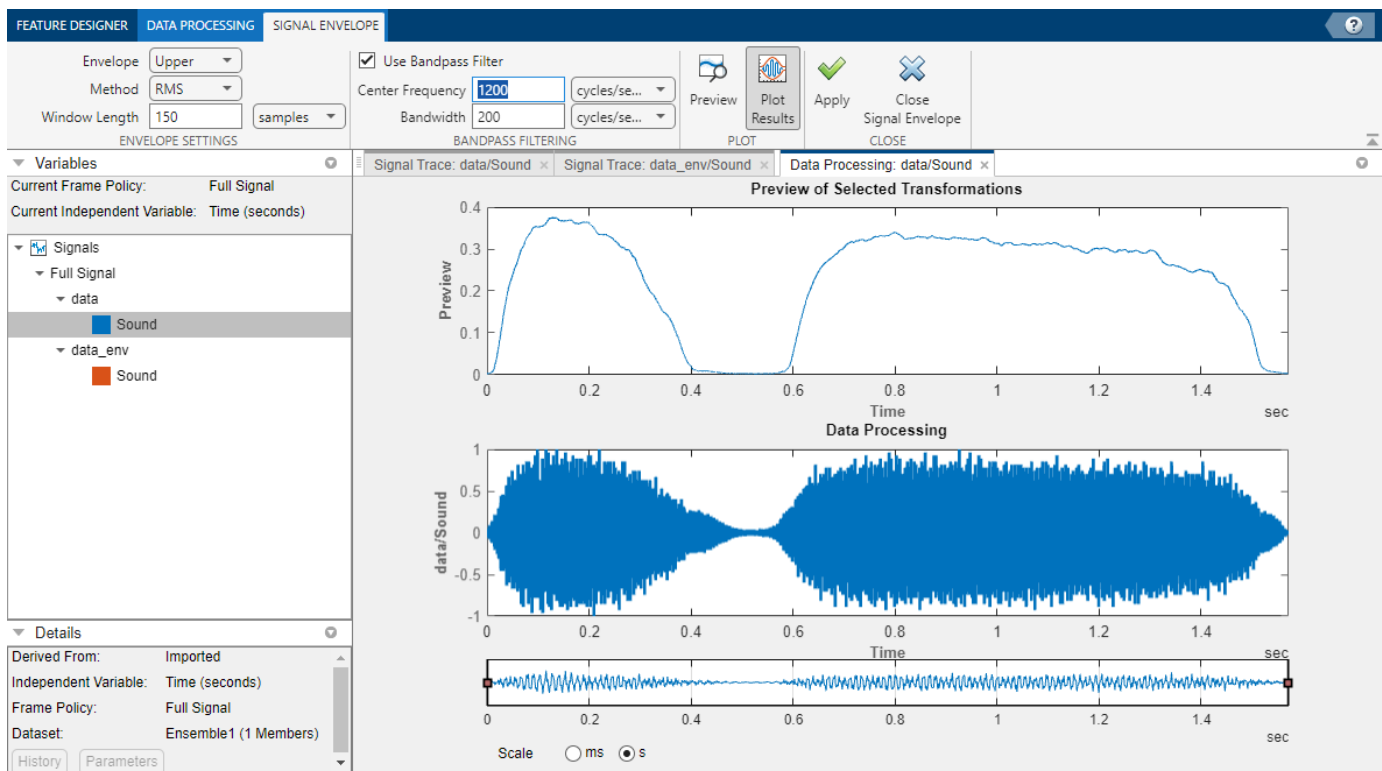The previewed envelope captures the general shape of the original signal.

Apply Envelope Settings

Click **Apply** to create and plot the new envelope variable.

Use Bandpass Filter

Applying a bandpass filter can improve results. Specify and preview another envelope that includes a bandpass filter of 1200 Hz, with a center frequency of 200 Hz.

The envelope computed using a band-limited filter captures the shape of the signal more accurately than the unfiltered envelope.

See Also

envelope | envspectrum | hilbert

Related Examples

- "Identify Condition Indicators for Predictive Maintenance Algorithm Design"

Export Feature Extraction Function and Simulink Model for Streaming Data

In **Diagnostic Feature Designer**, you can generate a MATLAB function that automates your interactive computations for the features you choose. The function that you generate can be formatted in two ways:

1 Automate feature extraction from multimember data ensembles, as described in "Automatic Feature Extraction Using Generated MATLAB Code" on page 7-86.

2 Automate feature extraction for streaming data.

- Generate a MATLAB function that is MATLAB Coder compliant and use `codegen` to convert the function to C/C++.
- Generate a Simulink model that uses the MATLAB function.

Spectral features that you create using autoregressive or state-space modeling are not supported for streaming data. To generate code for spectral features, you must use the Welch method to extract them. Features derived from ensemble statistics variables or frame-based processing are also not supported for code generation.

Generate MATLAB Function for Streaming

The following figure shows the ranking table for a set of features. These features include spectral features that were computed using an autoregressive model.
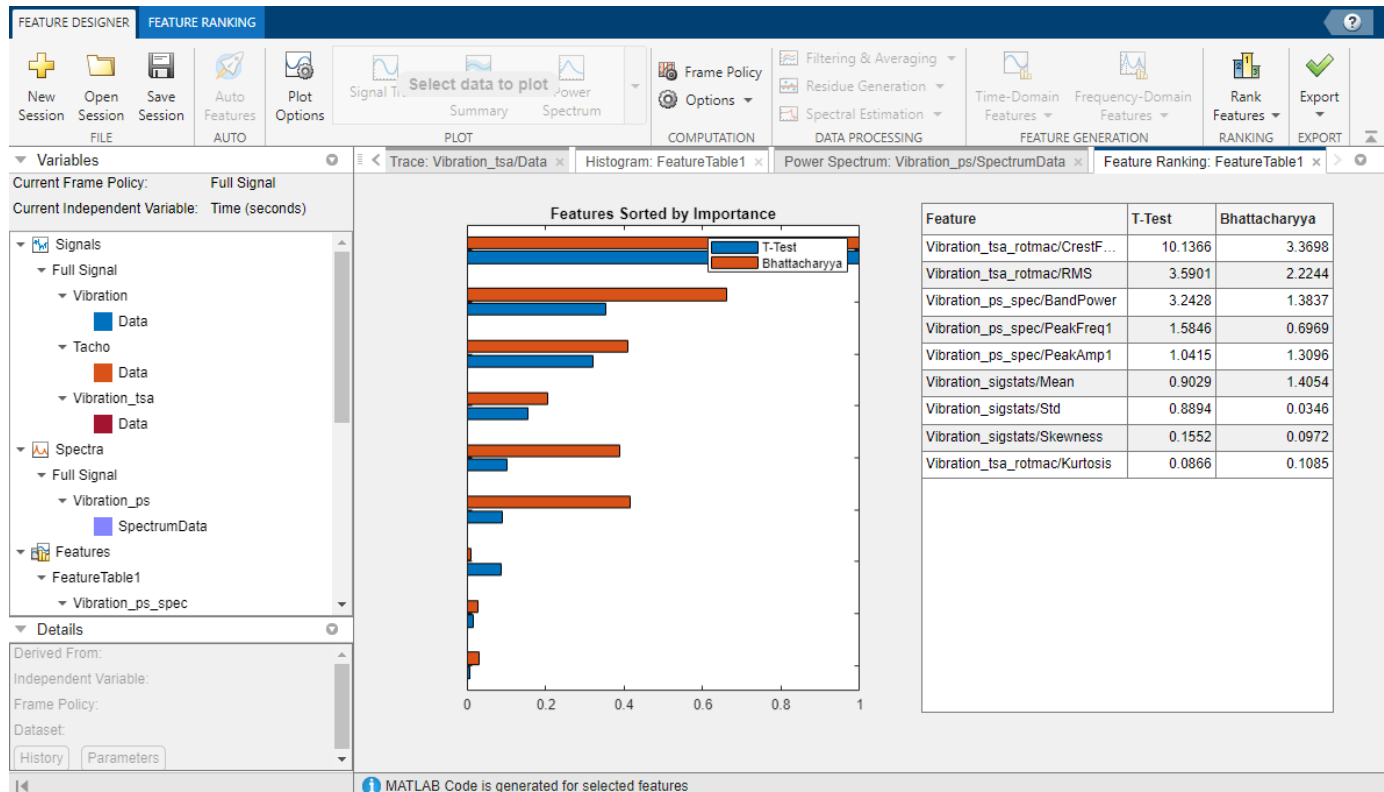
Click **Export** to access the export options and select `Generate Function for Features`.
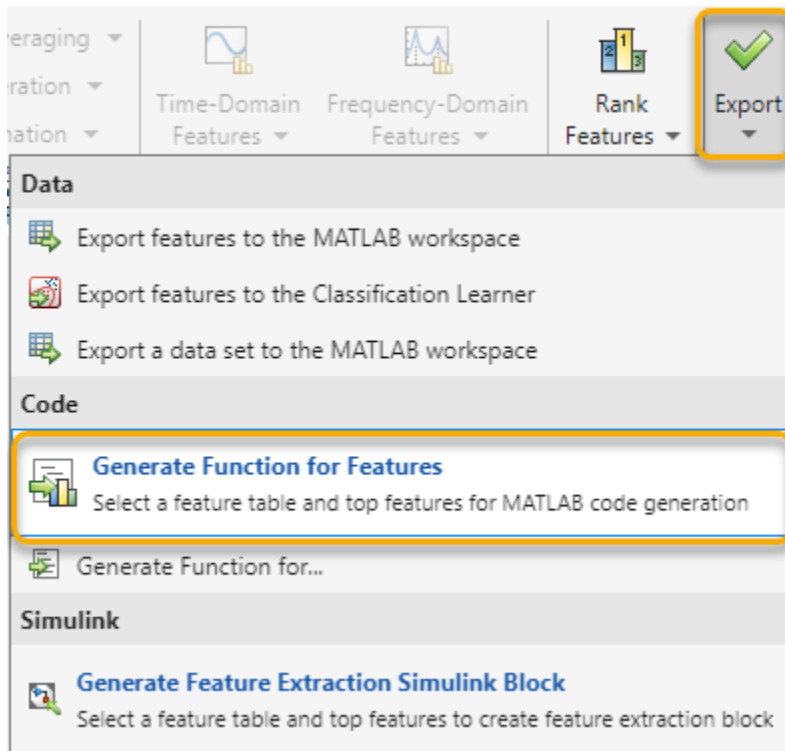
The dialog box allows you to choose the number of top ranked features and whether you want to format the function for streaming data.
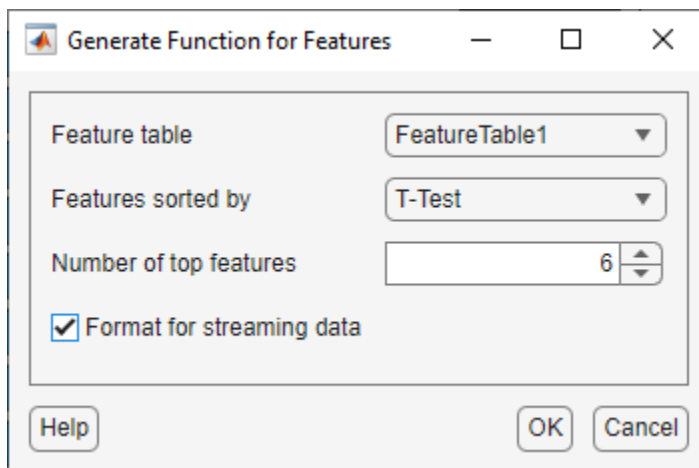
When you click **OK**, the app generates the streaming-formatted function. The following code is the descriptive header at the top of the function.

```
%DIAGNOSTICFEATURES recreates results in Diagnostic Feature Designer.
%
% Input:
%   inputData: A timetable.
%
% Output:
%   featureTable: A table containing all features.
```

```
%
% This function computes signals:
%  Vibration_tsa/Data
%
% This function computes features:
%  Vibration_sigstats/Mean
%  Vibration_tsa_rotmac/RMS
%  Vibration_tsa_rotmac/CrestFactor
%
% Organization of the function:
% 1. Compute signals/spectra/features
% 2. Extract computed features into a table
%
% Modify the function to add or remove data processing, feature generation
% or ranking operations.

% Auto-generated by MATLAB
```

This function accepts input data in the form of a timetable and returns a feature table. The function generates only three features from the app session. This is because the other three features were extracted from a power spectrum that was computed using an autoregressive model, which is not compatible with C/C++.

For comparison, the exported code description from the same session data, but without streaming format, is provided in "Automatic Feature Extraction Using Generated MATLAB Code" on page 7-86.

Generate Streaming Code

From this new MATLAB function, you can now generate C or C++ code using `codegen`.

For example, if your function has the file name `diagnosticFeatures.m` and the representative function inputs are in the timetable `tt`, then generate C code using the following command.

```
codegen diagnosticFeatures.m -args {tt}
```

Export Simulink Block for Streaming Data

In addition to exporting a MATLAB function, you can also export a Simulink block that uses that function.

In the **Export** menu, select `Generate Feature Extraction Simulink Block`.
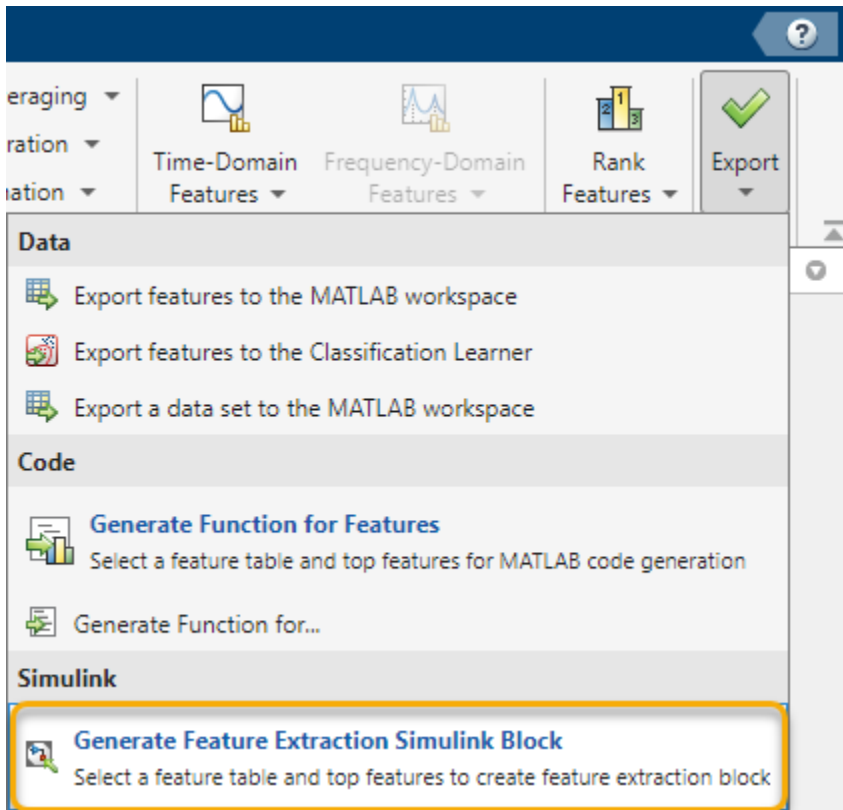
The resulting dialog box is similar to the **Generate Function for Features** dialog box but does not include a checkbox for streaming. This is because the Simulink Block code is always formatted for streaming.
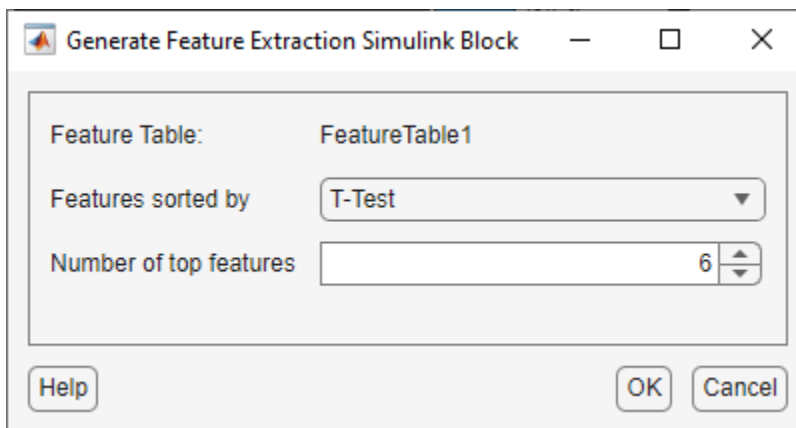
When you click **OK**, the app generates a template Simulink model that includes blocks for the vibration and tacho signal inputs, the exported **Feature Extraction** block, and an output scope.
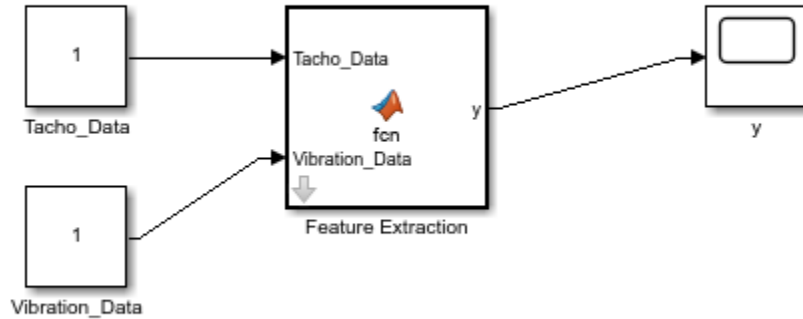
Double-click the **Feature Extraction** block to open the block parameters and confirm or specify the feature extraction function as well as buffering, independent variable, and sample time and start time information.

Block Parameters: Feature Extraction ✕

Subsystem (mask)

Imitates the feature extraction process performed in the Diagnostic Feature Designer (DFD) app. Only computes a set of top features that the block was created to extract and provides the result as a vector output.

Streaming Function Name: The name of the function that will be used for feature computation.

Buffer Size: The size of the signals/spectra on which feature computation will be performed.

Buffer Overlap: The size of the signals/spectra that will be re-used for consecutive feature computation.

IV Type: Select an Independent Variable for feature computation. It can be either Time (with its own Start Time and Sampling Rate) or an additional Signal input to the block.

Parameters

Streaming Function Name: diagnosticFeatures   Browse

Buffering

Buffer Size: 6000

Buffer overlap: 0

Independent Variable

IV type

◉ Time   ○ Signal   ○ None

Start Time: 0

Sampling Rate: 0.005

OK   Cancel   Help   Apply

Double click an input block to specify the input parameters. The block includes tabs for main parameters and signal attributes.
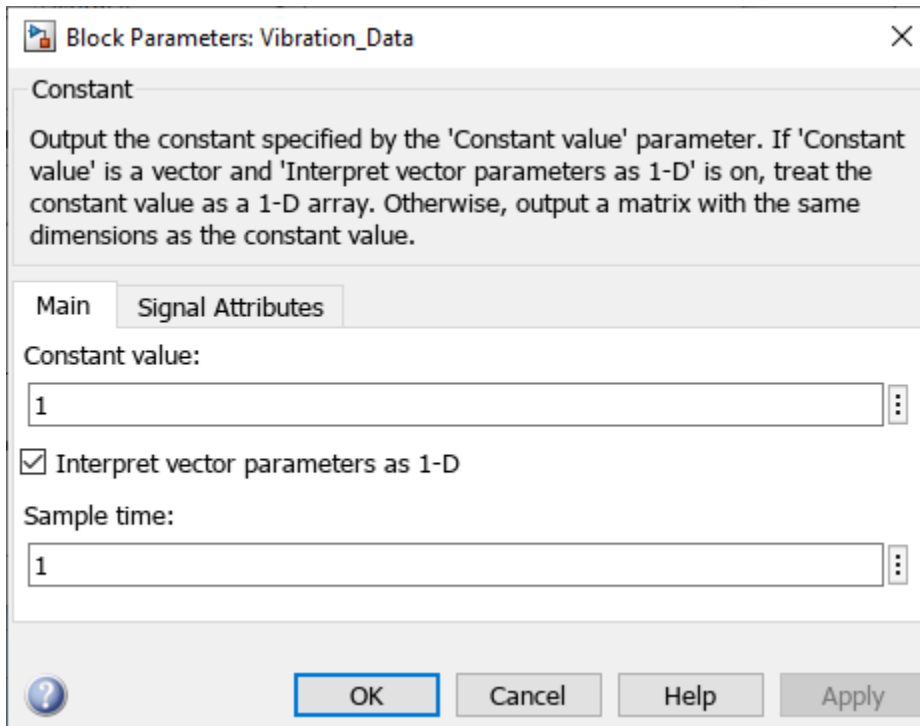
You can incorporate additional blocks as well such as a **Classification Learner Predict** block.

See Also

Diagnostic Feature Designer | **MATLAB Coder** | codegen | **Classification Learner**

Related Examples

- "Automatic Feature Extraction Using Generated MATLAB Code" on page 7-86
- "Generate a MATLAB Function in Diagnostic Feature Designer" on page 7-94
- "Predict Class Labels Using ClassificationTree Predict Block"

Diagnostic Feature Designer Help

Import Single-Member Datasets

The app accepts input data in the form of individual single-member datasets or an ensemble dataset. Import single-member datasets when:

- Your source data in the MATLAB workspace consists of an individual workspace variable for each machine member.
- The size and number of your member datasets are small enough for app memory to accommodate

Before importing your data, it must already be clean, with preprocessing such as outlier and missing-value removal. For more information, see "Data Preprocessing for Condition Monitoring and Predictive Maintenance" on page 2-2.

The app accepts individual member `table` arrays, `timetable` arrays, or numeric matrices, each containing the same independent variables, data variables, and condition variables.

Input Item	Content	Notes
Data Variables	Can contain timetables, tables, cell arrays, or numeric arrays	
Independent Time Variables	Double, `duration`, or `datetime`	All time variables must be of the same type. If your data was uniformly sampled and you don't have recorded timestamps, you can construct a uniform timeline during the import process.
Condition Variables	Scalar — Numeric, string, cell, or categorical	You can import condition variables along with your data in tables, timetables, or cell arrays, but not in matrices.
Member Matrices	Purely numeric array. Only one independent variable, but any number of data variables tied to that independent variable	Cannot accommodate variable names

For more information about organizing your data for import, see "Organize System Data for Diagnostic Feature Designer" on page 7-19.

Selection — Select Data to Import

Select the same-size datasets you want to import from your workspace. Import all the datasets you want to use in your session at one time. You cannot import data incrementally.

Configuration — Configure Ensemble Variables

Review and modify the variable types and units that **Diagnostic Feature Designer** associates with your imported variables.

In **Variable Name**, the configuration view displays the name of the variable as it will appear after the import. If a table variable consists of a timetable or table with its own variable names, then the app combines these variable names into a new name. For example, if table variable `Vibration` is a

`timetable` with `Time` and `Data` variables, then the imported variable names are `Vibration/Time` and `Vibration/Data`.

In **Variable Type**, the app infers the variable type from its source. Sometimes, the variable type or unit is ambiguous, and you must update the default setting.

- Numeric scalars can represent either condition variables or features. By default, the app assumes numeric scalars are of type `feature`. If your scalar is actually a condition variable, change the variable type to `Condition Variable`.

- Independent variable assignment is explicit in timetables, but not in tables or matrices. If the configuration table shows the wrong variable type for an independent variable, select the correct variable type.

  In a matrix, you can use only one independent variable. If you have multiple identical independent variables, such as a timeline that applies to all the data, select **Skip** for the redundant variables.

- In **Unit**, the configuration view displays the units associated with the variables. If the unit specification for a variable is incorrect, update **Unit** by selecting or entering an alternative.

Uniformly sampled data does not always have explicitly recorded timestamps. The app detects when your imported data does not contain an explicit independent variable and allows you to create a uniform one. Specify the type, starting value, and sampling interval.

Review — Review and Import Variables

Review the ensemble variables that result from your import. Each of these variables is an ensemble signal, spectrum, or feature that contains information from all your imported members. The app maintains these variables in an ensemble with the name specified in **Ensemble name**. Update the default name if you want to use a different ensemble name.

Click **Import** once you are confident your ensemble is complete. If, after completing the import, you find that you need additional datasets, you must perform a fresh import that includes everything you want. This fresh import deletes existing imported variables, derived variables, and features.

if you plan to explore the data in multiple sessions, consider saving your session immediately after you import. Saving your session after import provides you with an option for a clean start for new sessions without needing to import your separate files again. You can save additional sessions after you have generated derived variables and features.

See Also
`table` | `timetable`

More About
- "Data Ensembles for Condition Monitoring and Predictive Maintenance" on page 1-2
- "Organize System Data for Diagnostic Feature Designer" on page 7-19
- "Prepare Matrix Data for Diagnostic Feature Designer" on page 7-10
- "Import and Visualize Ensemble Data in Diagnostic Feature Designer"

Import Multimember Ensemble

Import Multimember Ensemble

The app accepts input data in the form of individual member datasets or ensemble datasets. Import multimember ensemble datasets when your source data is combined into one collective dataset that includes data for all members. This collective dataset can be any of the following:

- An ensemble table containing `table` arrays or matrices. Table rows represent individual members.
- An ensemble cell array containing tables or matrices. Cell array rows represent individual members.
- An ensemble datastore object that contains the information necessary to interact with files stored externally to the app. Use an ensemble datastore object especially when you have too much data to fit into app memory.

The members in the collective dataset must all contain the same independent variables, data variables, and condition variables.

- All independent time variables must be of the same type — either all double or all `duration` or all `datetime`. If your original data was uniformly sampled and timestamps were not recorded, the app prompts you to construct a uniform timeline during the import process
- Embedded matrices can contain only one independent variable, but can have any number of data variables tied to that independent variable.
- Condition variables in a member dataset contain a single scalar. The form of the scalar can be numeric, string, cell, or categorical.

If you are using an ensemble datastore object, you must set the `ReadSize` property to `1`. This property determines how many members the software reads in one operation. The app reads one member at a time. For more information on `ReadSize`, see the property description in `fileEnsembleDatastore` or `simulationEnsembleDatastore`.

For more information about organizing your data for import, see "Organize System Data for Diagnostic Feature Designer" on page 7-19.

Before importing your data, it must already be clean, with preprocessing such as outlier and missing-value removal. For more information, see "Data Preprocessing for Condition Monitoring and Predictive Maintenance" on page 2-2.

Selection

Select a single dataset from **Ensemble variable**. You cannot import data incrementally.

Configuration

Review and modify the variable types and units that **Diagnostic Feature Designer** associates with your imported variables.

If a table variable consists of a `timetable` array or a `table` array with its own variable names, then the imported variable name combines these names. For example, if table variable `Vibration` is a `timetable` with `Time` and `Data` variables, then the imported variable names are `Vibration/Time` and `Vibration/Data`.

The import process infers the variable type from its source and type. Sometimes, the type or the unit is ambiguous, and you must update the default setting.

- Numeric scalars represent either condition variables or features. By default, when you import tables, the app treats numeric scalars as features. If the default type is incorrect, select the correct variable type.
- Independent variable type is explicit in timetables, but not in tables or matrices. Select the correct independent variable type for any unidentified independent variables.
- Update units for variables if necessary by selecting or entering alternatives within **Units**.

Uniformly sampled data does not always have explicitly recorded timestamps. The app detects when your imported data does not contain an explicit independent variable and allows you to create a uniform one. Specify the type, starting value, and sampling interval.

Review

Review the ensemble variables that result from your import. Each of these variables is an ensemble variable that contains information from all your imported members. The app maintains these variables in **Ensemble name**. Update the default if you want to use a different ensemble name.

Click **Import** once you are confident your ensemble is complete. If, after completing the import, you find that you need additional data, you must perform a fresh import that includes everything you want. This fresh import deletes existing imported variables, derived variables, and features.

Consider saving your session immediately after you import if you plan to explore the data in multiple sessions. Saving your session after import provides you with an option for a clean start for new sessions without needing to import your ensemble dataset again. You can save additional sessions after you have generated derived variables and features.

Additional Information

For more information on:

- Data ensembles — See "Data Ensembles for Condition Monitoring and Predictive Maintenance" on page 1-2.
- Tables and Timetables — See `table` and `timetable`.
- Creating an ensemble table from member matrices — See "Prepare Matrix Data for Diagnostic Feature Designer" on page 7-10.
- File ensemble datastore objects — See `fileEnsembleDatastore`, "File Ensemble Datastore with Measured Data" on page 1-17.
- Simulation ensemble datastore objects — See `simulationEnsembleDatastore`, "Generate and Use Simulated Data Ensemble" on page 1-10.

Ensemble View Preferences

Modify how the app displays ensemble members.

Group By

Use **Group by** when you want to visually separate members by condition variable label. The app uses a different color for each label. For example, if your condition variable is `Fault Code` with labels `Healthy` and `Unhealthy`, members from healthy systems are a different color than data from unhealthy systems. The plot legend shows the color associated with each label value.

Ensemble Representation by Members or Statistics

- **Show a curve for each member signal** — Display each member individually.

  - **Maximum number of curves to show** — Reduce the number of curves the app displays. Use this option when you want to clarify the shape of representative member traces, or when your ensemble contains many members that take a long time to plot.

- **Show signal variation among ensemble members as confidence regions** — Display the mean and standard deviation of the ensemble. Use this option when you are interested in signal variation rather than the specific behavior of individual member signals.

  The app saves the statistics that it calculates for the variation plot in single-member variables, and stores those variables in the `SummaryData` dataset. These variables are equivalent to the variables you would generate using **Filtering & Averaging > Ensemble Statistics**. You can use these ensemble statistics to generate signal residues in **Residue Generation**.

  - **Number of standard deviations** — Modify the number of standard deviations to display in the shaded confidence region.
  - **Show minimum/maximum boundaries** — Display the envelope of the source member signals.

Plot Options

Specify default plotting options for all the plots that you generate during your app session. You can set these options before you generate your first plot, or at any time during your session. You can override these settings for individual plots without changing your specified defaults for subsequent plots. When you click **Plot Options**, you open a dialog that allows you to set the following options:

- **General** — These options apply to all signal and spectrum plots.

  - **Group by** — Group data by a condition variable label. The app uses color to distinguish label groups. For example, if your condition variable is `faultCode` with labels `healthy` and `degraded`, the app uses one color for member data with the `healthy` label and another color for member data with the `degraded` label

  - **Number of curves** — Specify the number of members to plot. Set this option when you have a large number of ensemble members and you want to plot only a subset of the members. Using this option reduces the plotting time and allows you to assess individual member behavior more easily.

- **Spectrum** — These options apply only to spectral plots.

  - **Number of peaks to mark** — Specify the number of peaks to mark. Set this option to limit the number of spectral peaks that are marked to highlight only the most significant peaks.

- **Ensemble Summary** — These options apply only to the ensemble summary plot, which is a special plot that displays the mean and standard deviation of the ensemble as a whole.

  - **Number of standard deviations** — Specify the number of standard deviations that the ensemble summary plot displays.

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

Data Handling Mode and Frame Policy

By default, the app processes entire signals. You can also divide your signal into uniform segments — or frames — for sequential processing. Use frame-based processing, for example, to localize abrupt signal changes or compute the full set of prognostic ranking features for remaining-useful-life calculations.

- **Full signal** — Process entire signal in one continuous time series. For example, suppose that you have an ensemble of 20 vibration signals and select the feature **Time Domain Features > Signal Features > Mean**. You create 20 values of the mean, one for each member signal.
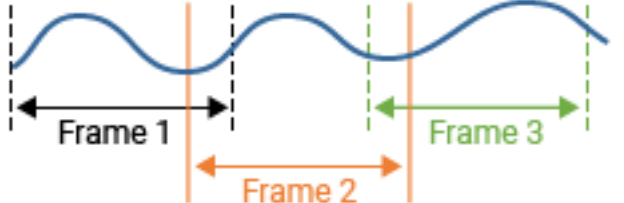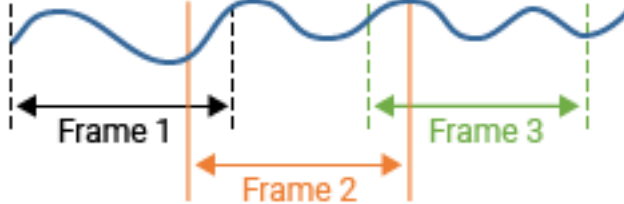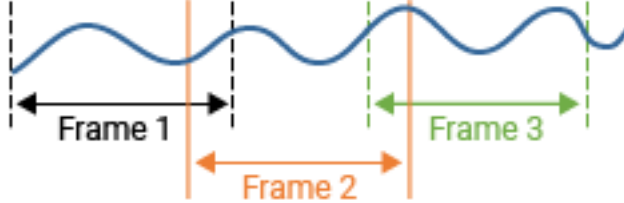

- **Frame-based** — Process signal as a set of individual segments defined by frame size and rate.

  - **Select a frame policy** — Select a previously specified frame size and frame rate pair or create a new frame specification. The app stores each frame setting pair in a frame policy. You can maintain multiple frame policies, and select from them using this menu. To create a new frame policy from your current frame size and frame rate entries, select New Frame....

  - **Frame size (FS)** — Specify the time interval in seconds over which data is provided.

  - **Frame rate (FR)** — Specify the time interval in seconds between frame start times. This interval is equivalent to the frequency with which new frames begin.

  For example, suppose that you have an ensemble of 20 vibration signals, and each signal is at least 100 seconds long. If you enter **Frame size (FS)** as 50 seconds and **Frame rate (FR)** as 40 seconds, each signal contains at least three frames:

  - Frame 1: 0–50 seconds

  - Frame 2: 40–90 seconds

  - Frame 3: 80–L, where L is the minimum of 130 seconds or the signal end time.

  If you compute the mean feature in frame-based data handling mode, each signal contains at least three mean values, with one mean value for each frame.

Member No.	Signal (3 frames, 20% overlap)	Mean
1		0.22 0.3 0.48
2		1.2 2.9 2.5
⋮		
20		-0.12 -0.01 0.07

Remove Harmonics

Remove Harmonics creates a series of notch filters to remove specific frequencies from your signal.

Use **Remove Harmonics**, for example, if you are studying one rotating gear shaft in a gear box, and want to isolate it from the propagated effects of other gear shafts within the box. Also remove harmonics to filter out harmonically related signal components such as AC line components of 60 Hz, 120 Hz, and so forth.

Filter Settings

Use the filter settings to specify the parameters that define the fundamental target frequency, notch bandwidth, and number of harmonics. The sample frequency F_s constrains the allowable combinations to the range [0 F_s/2]. Choose your settings so that the notch of your highest filtered harmonic falls completely within this range. Specifically, $nf + bw/2$ must fall within [0 F_s/2]. Here, n is the number of harmonics, f is the fundamental target frequency, and bw is the notch bandwidth.

- **Fundamental frequency** — Choose the fundamental frequency based on your knowledge of the lowest harmonic frequency that you want to remove.
- **Notch bandwidth** — Set the notch bandwidth:
  - Wide enough to remove sidebands and leakage around the target frequency along with the target frequency
  - Narrow enough that the notch does not filter out nearby harmonic information that you want to retain.
- **Number of Harmonics** — Set the number of harmonics according to your knowledge of how much influence the higher harmonics have on your signal. You can try different harmonic levels and see how much influence they have on feature and computational performance.

The software stores the results of the computation in a new variable. The new variable name includes the source signal name with the suffix `noharm`.

For more information on notch – or stopband – filters, see `butter`.

Time-Synchronous Averaging

Use time-synchronous averaging (TSA) when you are working with data from rotating machinery.

With rotating machinery, a single rotation period encapsulates the full gamut of machinery component interaction. Averaging over uniform rotation angles or complete rotations, rather than averaging over more arbitrary time segments, rejects any noise, disturbance, or periodic signal content that is not coherent with the rotation. This rotation-based averaging is called time-synchronous averaging, or TSA.

- **Tacho Information** — Rotation rate, specified as one of the following:

  - **Constant rotation speed (RPM)** — Use constant RPM when your rotation rate is fixed both in time and across all ensemble members.

  - **Tacho signal** — Use a tacho signal when you have a tachometer that transmits pulses at specific angular points in the rotation. Your tacho signal must have the same length as your data signal.

  - **Compute nominal speed (RPM)** — Compute the nominal rotation speed for each individual machine when processing the tacho signal. Select this option if the rotation speeds of your machines vary across the ensemble and you plan on filtering the TSA signal (see "Filter TSA Signals" on page 8-12). The app stores the computed rotation speeds in a condition variable with the suffix `rpm`.

    For an example of performing TSA processing with varying nominal wheel speeds, see "Isolate a Shaft Fault Using Diagnostic Feature Designer" on page 7-46

- **Interpolation method** — Select the interpolation to use when computing the TSA signal. For information on the interpolation methods and the algorithms for using them, see `tsa`.

- **Pulses per rotation** — Number of tachometer pulses in each rotation. Set this parameter only when using a tacho signal.

- **Number of rotations** — Typically, a TSA signal averages over a single rotation. Increase the number of rotations when you want to improve frequency resolution for follow-on filtering of the TSA signal.

The software stores the results of the computation in a new variable. The new variable name includes the source signal name with the suffix `tsa`.

Additional Information

For more information on:

- TSA signals — See `tsa`.
- Tachometer signals — See `tachorpm`.
- Rotating machinery features — See "Condition Indicators for Gear Condition Monitoring" on page 3-10.

Filter TSA Signals

Apply filtering to a TSA signal to generate additional signals that isolate the response of specific components within a harmonically interrelated system such as a gearbox. These isolated signals can then be used for rotating machinery features that detect faults in specific physical components and locations.

Use TSA signal filtering only if shaft speeds are constant over time.

Signals to Generate

The filtered signals each retain a subset of the original TSA signal components. The TSA signal components are:

- Primary shaft speed and specified higher-order harmonics (*sh*)
- Gear meshing vibrations (fundamental frequencies and their higher-order harmonics) (*gh*)
- Specified gear meshing sidebands (*ss*)
- All other vibration components, including degradation, faults, noise, and unspecified gear-meshing sidebands and harmonics (*ovc*)

- **Difference signal** — Contains *ovc* only. Computed by $diff = tsa-(sh+gh+ss)$
- **Regular signal** — Computed by $reg = sh+gh+ss$
- **Residual signal** — Contains *ss* and *ovc*. Computed by $res = tsa-(sh+gh)$

Speed Settings

- **Constant rotation speed (RPM)** — Enter a constant rotation speed in RPM. This shaft speed applies to all members.
- **Nominal rotation speed (RPM)** — Use an individual rotation speed for each member by selecting the condition variable that stores these values. You create this condition variable when you first compute the TSA signal if you select **Compute nominal speed (RPM)**. For more information, see "Time-Synchronous Averaging" on page 8-11.

Filter Settings

- **Domain** — Specify `Order` or `Frequency`.

  - Specify the order domain when you know the specific ratios between gear sizes in the gearbox. For example, if a gear rotates at three times the primary shaft speed, the order of the gear rotation is 3. Order is independent of absolute shaft speed.
  - Specify the frequency domain when you know the absolute frequencies of the rotating gears. For example, you might have a power spectrum whose peaks correspond to gear rotation.

- **Rotation orders** or **Frequencies (Hz)** — Specify the order list or frequencies list for the known harmonics, in the form `[o1 o2 o3 ...]` for order domain and `[f1 f2 f3 ....]` for frequency domain. `o1` and `f1` both must represent the primary shaft speed, which produces the fundamental frequency.
- **Number of harmonics** — Number of shaft and gear meshing frequency harmonics to be retained or filtered. This parameter includes the fundamental frequency represented by `o1` or `f1`.

- **Number of sidebands** — Number of sidebands to be retained (regular signal) or filtered (difference signal) from the order or frequency list. Modify **Number of Sidebands** to filter out known wideband frequency components around gear harmonics. As machines degrade, these wideband components typically grow. Filtering the known components at a given degradation state can improve detection sensitivity to further degradation.

The filter settings apply to all the filtered signals you select. If you want to apply a unique setting to each filtered signal, such as the **Number of Sidebands** setting, process the signals separately by selecting one filtered signal at a time.

Additional Information

The software stores the results of the computation in new variables. The new variable names include the source signal name with the suffix `tsafilt`.

For more information on filtered TSA signals, see:

- `tsadifference`
- `tsaregular`
- `tsaresidual`

Ensemble Statistics

Calculate ensemble statistics when you want to characterize ensemble behavior as a whole. Also calculate ensemble statistics when you plan to generate residual signals for feature extraction. Unlike ensemble signals that contain multiple members, the resulting statistics signals each contain only one member. If you want to simply view the ensemble statistics without generating any new variables, use the **Ensemble Summary** plot in the plot gallery instead.

- **Ensemble Statistics** — Select which statistics you want to generate. For each statistic that you select, the value is calculated at each time point across all members. The result is a single-member signal for that statistic. For example, if you select **Ensemble mean**, the single computed value at each time point is the mean of the values of all ensemble members at that time point.

- **Interpolate data** — Select this option to provide a common time base when the different members in the ensemble have different sampling points.

  - **Interpolation method** — Select the interpolation method to use. For information on interpolation methods, see the Interpolation Method section of `interp1`.

  - **Sampling frequency** — The default `auto` setting calculates the mean sampling frequency across all members.

- **Group by** — Select a condition variable to group by when you want to separate the statistics corresponding to different groups. For example, if your condition variable indicates `healthy` or `faulty`, you can separate the healthy statistics from the faulty statistics.

- **Plot results** — Select this option if you want to plot all statistical results in the same plot. If you want to choose which results you plot, use **Signal Trace** with the resulting statistics signals.

An alternative method of generating ensemble statistics is by showing signal variation in a signal trace plot of your ensemble signal. That plotting option generates identical statistics to using **Ensemble Statistics**.

The software stores the results of the computation in the `SummaryData` dataset. The new variable names include the source signal name with the suffix `mean`, `min`. `max`, or `std`.

Interpolation

Choose interpolation when your signal members have individual timestamps that vary with each member. Interpolation resamples members to a uniformly sampled data grid.

- **Interpolation Method** — For information on interpolating to a uniform grid, see the Interpolation Method section of `interp1`.
- **Sampling frequency** — The default `auto` setting calculates the mean sampling frequency across all members.

The software stores the results of the computation in a new variable. The new variable name includes the source signal name with the suffix `interp`.

Use interpolation only if you can consider your signals to be concurrent. If you use signals that are intentionally staggered in start time, interpolation expands each member signal from its original size to the full length of the entire sequence using `NaN` fill. For example, suppose that you want to maintain a sequence of three different measurement days. Each day has the same amount of data and sampling frequency, but each has a different start time. Grid interpolation expands each data member from one day to three days. This signal expansion and `NaN` handling increase processing overhead and slows computation.

For more information on gridded interpolation, see "Interpolating Gridded Data".

Subtract Reference

Generate residues by subtracting a reference signal when you want to focus your analysis on signal variation by removing normal ensemble-level behavior such as the ensemble mean.

To compute residues, you must already have at least one `Ensemble Statistics` signal the statistical signal that you want to remove. To generate these signals, use **Filtering & Averaging > Ensemble Statistics**.

- **Signal to subtract** — Choose from the list of available statistical signals.

The software stores the results of the computation in a new variable. The new variable name includes the source signal name with the suffix `res`.

Time Series Processing and Time Series Features

Transform your signals into stationary time series, and from the time series, extract specialized features. Time series features provide unique insights into the data.

Obtain Stationary Time Series from Signal Data

A stationary time series has no trends or periodic fluctuations, and constant variance and autocorrelation over time. Second-order stationary signals, which are usually sufficient for engineering applications, have zero mean and constant variance. The app provides a set of processing transformations that you can use to eliminate nonstationary components and reduce time-dependent variance. These transformations include:

- **Difference** — Remove nonstationary level (first order) or slope (second order).
- **Seasonal** — Remove periodic components such as seasonal variations. Specify the seasonal period in number of samples in one period.
- **Detrend** — Remove deterministic polynomial trend. Specify the polynomial type as constant, linear, cubic, or quadratic.
- **Natural Logarithm** — Stabilize variance over time.
- **Box-Cox Transform** — Stabilize variance over time, using a power level that you specify. Using a power level of 0 is equivalent to taking the logarithm. The data is normalized and shifted to have a minimum of 1 so that negative powers and logarithms are always meaningful.

To specify an operation, select a transformation and click **Add Selected**. You can select a single transformation or stack multiple transformations in the order that you choose. You can experiment with different transformation selections and different orders. Click **Preview** to view intermediate results so that you can assess how well the processing performs. For example, you can visually assess the general flatness of the signal for the absence of obvious trends. Once the preview shows acceptable results, click **Apply** to create the time series variable.

To access this processing in the app, select the source signal, and then, in the **Feature Designer** tab, in the **Data Processing** section, select **Residue Generation > Time-Series Processing**.

Extract Features from Stationary Time Series

Time series features in the app include distribution features, autocorrelation features, and partial autocorrelation features.

- The **Distribution Features** section contains standard statistical features that characterize the overall dispersion of the data. These features include the overall minimum, median, maximum, quartile statistics, and custom quantiles.
- The **Autocorrelation Features** section contains features that describe the linear dependence of a variable with itself at two points in time. For stationary processes, the autocorrelation between any two points depends only on the time lag between them. The autocorrelation function (ACF) is the sequence containing the autocorrelation values that correspond to each possible lag value. The sum of squares for a specified value of n is the sum of the squares of the first n autocorrelations.
- The **Partial Autocorrelation Features** section contains features that are similar to autocorrelation features, but account for the effects of mutual linear dependence on other variables in the sequence. The partial autocorrelation function (PACF) is the sequence containing

the partial autocorrelations for each lag value. The sum of squares for a specified value of n is the sum of the squares of the first n partial autocorrelations.

To access these features in the app, select the source signal, and then, in the **Feature Designer** tab, in the **Feature Generation** section, select **Time-Domain Features > Time-Series Features**.

Order Spectrum

An order spectrum is like a power spectrum in that it displays frequency content. However, an order spectrum provides additional insight for harmonically interrelated systems in rotating machinery.

An order refers to a frequency that is a certain multiple of a reference rotational speed. For example, a vibration signal with a frequency equal to twice the rotational frequency of a motor corresponds to an order of 2. Likewise, a vibration signal that has a frequency equal to 0.5 times the rotational frequency of the motor corresponds to an order of 0.5. For a rotating system, the primary shaft rotation drives the fundamental frequency. The order spectrum for this system quantifies the relative strength of the system harmonics with respect to shaft rotation. This approach has the following benefits:

- Because the spectrum contains frequency ratios rather than absolute frequencies, the computation is insensitive to variations in shaft speed. This insensitivity decouples the change in the frequency location of a certain harmonic from the harmonic amplitude. In contrast, a power spectrum would be time-varying with shaft-speed variation.
- Different harmonics point to different rotating components within the system. Features that detect changes in a single harmonic can therefore potentially isolate individual components or locations within the system.

Rotation Information

Specify the source of rotation rate.

- **Constant rotation speed** — Use this option when you can represent the rotation speed of all members with a single scalar.
- **Time-varying rotation speed (RPM)** — Use this option when you have a signal containing rpm information for every member.
- **Tacho signal** — Use this option when you have a signal containing tachometer pulses.

Window Settings

- **Window type** — For information on order spectrum window types, see `'Window'` in the Name-Value Pair Arguments section of `orderspectrum`.
- **Overlap percent** — A value of 0 for overlap percent means that adjoining segments do not overlap. A value of 100 means that adjoining segments shift by one sample. A larger overlap percentage produces a smoother result but increases the computation time.

Additional Information

The software stores the results of the computation in a new variable. The new variable name includes the source signal name with the suffix `os`.

For more information on order spectrum computation, see `orderspectrum`.

Power Spectrum

A power spectrum characterizes frequency content and resonances within a system. Because degradation usually causes changes in the spectral signature, spectral behavior provides a rich source of information for feature generation.

Select from nonparametric and parametric algorithms. The nonparametric option is **Welch's Method**. The parametric options are **Autoregressive Model** and **State-space Model**. For the parametric methods, **Diagnostic Feature Designer** fits a parametric model to the signal. The software then uses this model to compute the spectral density.

When you select an algorithm, a new algorithm-specific tab that contains processing parameters opens.

Algorithm

- **Welch's Method** — The app calculates the power spectrum from the source signal using Welch's method. For information on setting window parameters, see `pwelch`.
- **Autoregressive Model** — The app fits an AR model to the signal and uses this model to compute the spectral density. For information on setting model order, approach, and windowing method, see `ar`.
- **State-Space Model** — The app fits a state-space model to the signal and uses this model to compute the spectral density.

  - **Model Order** — Specify the model order directly, or specify a range of orders for automatic order selection. With automatic order selection, the software automatically selects the smallest order that leads to a good fit to the data.

  - **Improve results using nonlinear least squares search** — Selecting this option improves estimation results for specific scenarios, at the cost of additional computational time. For more information, see the `'SearchMethod'` option in `ssestOptions`.

  - **Maximum number of iterations** — Increase the number of iterations to improve result accuracy. Decrease the number to improve computational speed.

  For more information on state-space modeling, see `ssest`.

Frequency Settings

- **Frequency Grid** — Frequency settings for the frequency axis. To set these values manually, clear **Select automatically** and update the parameters for the frequency vector generation.

Additional Information

The software stores the results of the computation in a new variable. The new variable name includes the source signal name with the suffix `ps`.

Signal Features

Signal features provide general signal-based statistical metrics that can be applied to any kind of signal, including a time-synchronized average (TSA) vibration signal. Changes in these features can indicate changes in the health status of your system. **Diagnostic Feature Designer** provides a set of feature options .

Statistical Features

The statistical features include basic mean, standard deviation, and root mean square (RMS) metrics. In addition, the feature set includes shape factor and the higher order kurtosis and skewness statistics. All these statistics can be expected to change as a deteriorating fault signature intrudes upon the nominal signal.

Shape factor — RMS divided by the mean of the absolute value. Shape factor is dependent on the signal shape while being independent of the signal dimensions.

$$x_{SF} = \frac{x_{rms}}{\frac{1}{N}\sum_{i=1}^{N}|x_i|}$$

The higher-order statistics provide insight to system behavior through the fourth moment (kurtosis) and third moment (skewness) of the vibration signal.

- **Kurtosis** — Length of the tails of a signal distribution, or equivalently, how outlier prone the signal is. Developing faults can increase the number of outliers, and therefore increase the value of the kurtosis metric. The kurtosis has a value of 3 for a normal distribution. For more information, see `kurtosis`.

$$x_{kurt} = \frac{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^4}{\left[\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2\right]^2}$$

- **Skewness** — Asymmetry of a signal distribution. Faults can impact distribution symmetry and therefore increase the level of skewness.

$$x_{skew} = \frac{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^3}{\left[\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2\right]^{3/2}}$$

For more information, see `skewness`.

Impulsive Metrics

- Impulsive Metrics are properties related to the peaks of the signal.

  - **Peak value** — Maximum absolute value of the signal. Used to compute the other impulse metrics.

$$x_p = \max_i |x_i|$$

- **Impulse Factor** — Compare the height of a peak to the mean level of the signal.

$$x_{IF} = \frac{x_p}{\frac{1}{N}\sum_{i=1}^{N}|x_i|}$$

- **Crest Factor** — Peak value divided by the RMS. Faults often first manifest themselves in changes in the peakiness of a signal before they manifest in the energy represented by the signal root mean squared. The crest factor can provide an early warning for faults when they first develop. For more information, see `peak2rms`.

$$x_{crest} = \frac{x_p}{\sqrt{\frac{1}{N}\sum_{i=1}^{N}x_i^2}}$$

- **Clearance Factor** — Peak value divided by the squared mean value of the square roots of the absolute amplitudes. For rotating machinery, this feature is maximum for healthy bearings and goes on decreasing for defective ball, defective outer race, and defective inner race respectively. The clearance factor has the highest separation ability for defective inner race faults.

$$x_{clear} = \frac{x_p}{(\frac{1}{N}\sum_{i=1}^{N}\sqrt{|x_i|})^2}$$

Signal Processing Metrics

The signal processing metrics consist of distortion measurement functions. System degradation can cause an increase in noise, a change in a harmonic relative to the fundamental, or both.

- **Signal-to-Noise Ratio (SNR)** —Ratio of signal power to noise power
- **Total Harmonic Distortion (THD)** — Ratio of total harmonic component power to fundamental power
- **Signal to Noise and Distortion Ratio (SINAD)** — Ratio of total signal power to total noise-plus-distortion power

For more information on these metrics, see `snr`, `thd`, and `sinad`.

Additional Information

The software stores the results of the computation in new features. The new feature names include the source signal name with the suffix `stats`.

For information on interpreting feature histograms, see "Interpret Feature Histograms in Diagnostic Feature Designer" on page 7-13.

See Also

`kurtosis` | `skewness` | `snr` | `thd` | `sinad` | `peak2rms`

Related Examples

- "Signal-Based Condition Indicators" on page 3-4
- "Interpret Feature Histograms in Diagnostic Feature Designer" on page 7-13
- "Explore Ensemble Data and Compare Features Using Diagnostic Feature Designer" on page 7-2
- "Rotating Machinery Features" on page 8-24
- "Nonlinear Features" on page 8-26

Rotating Machinery Features

The rotating machinery features describe aspects of gearbox machinery, where one primary shaft drives other rotating gears. The system is harmonically interrelated, and this interrelationship allows metrics that can not only detect, but also locate the source of the fault.

The statistical rotating machinery features are similar in nature to their general statistical counterparts in **Signal Statistics**. The remaining features were derived through research in the literature, and empirically determined to be effective for differentiating or isolating specific types of faults.

Signals to Use

- **TSA Signal** — A time-synchronous averaged (TSA) signal is essential to calculating rotating machinery features. This option is read-only. You must select the TSA signal in the variable browser or the **Time-Domain Features** tab prior to selecting this feature option. To generate a TSA signal, use **Filtering & Averaging > Time-Synchronous Averaging**.
- **Difference Signal, Regular Signal** — These filtered TSA signals provide the source for the specialized rotating machinery metrics. To generate these signals, use **Filtering & Averaging > Filter TSA Signal** .

Metrics Using TSA Signal

- **Root Mean Square (RMS)** — Indication of the overall condition of the gearbox
- **Kurtosis** — Indication of major peaks in the signal
- **Crest Factor (CF)** — Peak-to-RMS ratio, which is an indication of gear damage in its early stages, especially where vibration signals exhibit impulsive traits

Metrics Using Difference Signal

- **Kurtosis (FM4)** — Detect faults isolated to only a limited number of teeth in a gear mesh
- **Normalized 6th moment (M6A)** — Indication of surface damage on the rotating machine components
- **Normalized 8th moment (M8A)** — An improved version of the M6A indicator

Metrics Using Mixed Signals

- **Zero-order figure of merit (FM0)** — Ratio of the standard deviations of the difference and regular signals, which is an indication of heavy wear and tooth breakage
- **Energy Ratio (ER)** — Indication of heavy uniform wear, where multiple teeth on the gear are damaged

Additional Information

The software stores the results of the computation in new features. The new feature names include the source signal name with the suffix `rotmac`.

For information on using these metrics for evaluating rotating machinery, see "Condition Indicators for Gear Condition Monitoring" on page 3-10. For information on specific rotating metrics, see `gearConditionMetrics`.

Nonlinear Features

Nonlinear features provide metrics that characterize chaotic behavior in vibration signals. These features can be useful in analyzing vibration and acoustic signals from systems such as bearings, gears, and engines. Nonlinear feature generation is more computationally intensive than the generation of any other features in the app.

The unique benefit of nonlinear features is that these features reflect changes in phase space trajectory of the underlying system dynamics. These changes can surface even before the occurrence of a fault condition. Thus, monitoring system dynamic characteristics using nonlinear features can help identify potential faults earlier, such as when a bearing is slightly worn.

Phase-Space Reconstruction Parameters

All the nonlinear features rely on phase-space reconstruction. Phase space is a multidimensional mapping of all possible variable states. This mapping provides a portrait of the fully dynamic behavior of a system. Phase-space reconstruction is a technique that recreates the multidimensional phase space from a single one-dimensional signal.

- **Embedding dimension** — Dimension of the phase space, equivalent to the number of state variables in the dynamic system
- **Lag** — Delay value used to perform the reconstruction

The default `'Auto'` setting results in estimation of these parameters. Vary the parameters manually to explore the impact of the settings on the effectiveness of the resulting features.

For more information on phase-space reconstruction, see `phaseSpaceReconstruction`.

Approximate Entropy

Approximate entropy measures the regularity in the signal, or conversely, the signal unpredictability. Degradation within a system typically increases the approximate entropy.

- **Radius** — Similarity criterion that identifies a meaningful range in which fluctuations in data are to be considered similar. The `'Auto'` setting invokes the default, which is based on the variance or covariance of the signal.

Correlation Dimension

Correlation dimension measures chaotic signal complexity, which reflects self-similarity. Degradation typically increases signal complexity and in doing so increases the value of this metric.

- **Similarity radius** — Bounding range for points to be included in the correlation dimension calculation. The default values are based on the signal covariance.

  Explore radius values visually using **Explore**. **Explore** brings up a plot of correlation integral versus radius. The correlation integral is the mean probability that the states of a system are close at two different time intervals. This integral reflects self-similarity. You can modify the similarity range by moving either of the vertical bounding lines, as shown in the following figure. The goal is to bound a linear portion of the curve. The bounding values transfer automatically to the Correlation dimension settings when you close the figure.
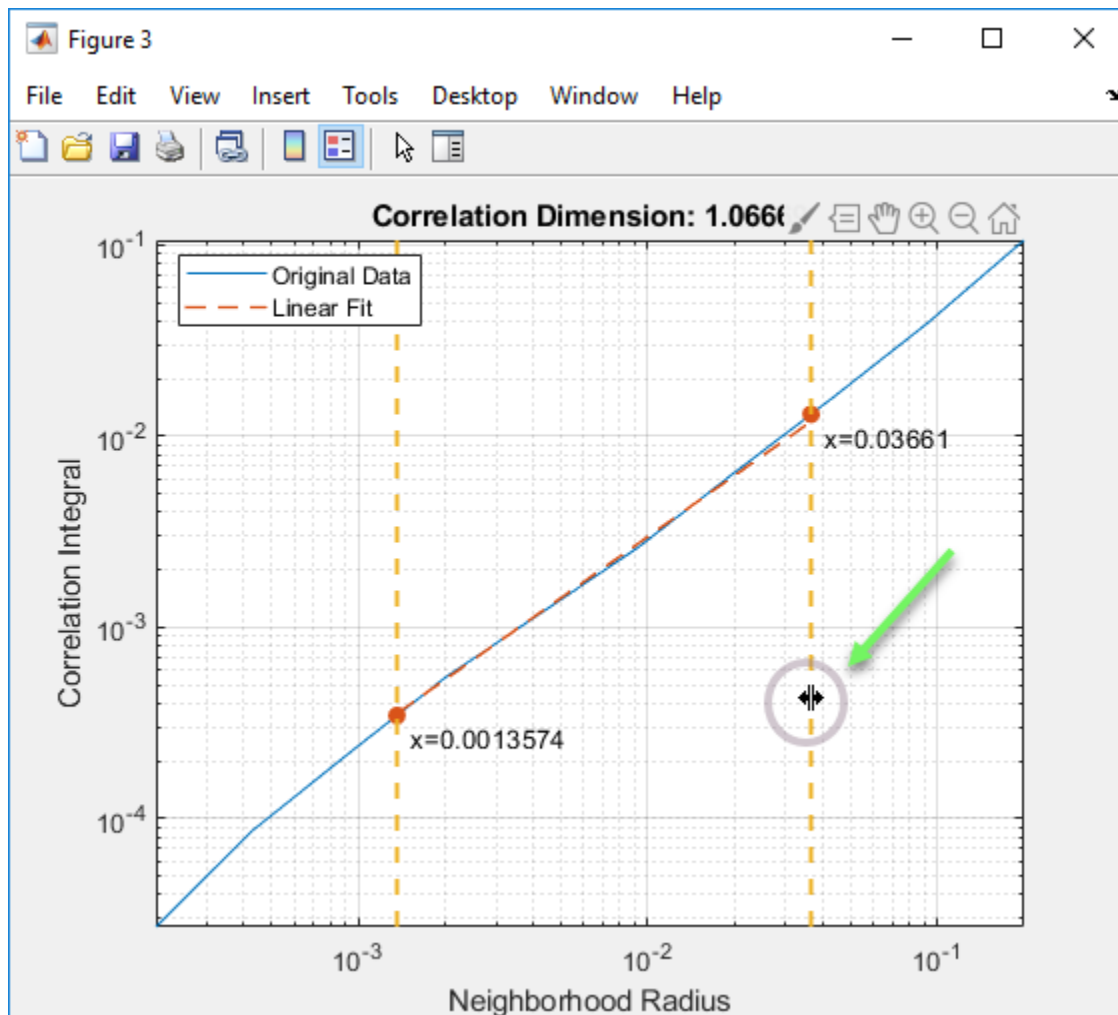
- **Number of points** — Number of points between the min and max range values. This setting drives the resolution of the calculation.

For more information on correlation dimension, see `correlationDimension`.

Lyapunov Exponent

The Lyapunov exponent measures the degree of chaos due to signal abnormality, based on the rate of separation of infinitesimally close trajectories in phase space. Degradation within the system increases this value. A positive Lyapunov exponent indicates the presence of chaos, with degree related to the magnitude of the exponent. A negative exponent indicates a nonchaotic signal.

- **Expansion range** – Bounding integer range that delimits the points to be used to estimate the local expansion rate. This rate is then used to calculate the Lyapunov exponent.

  Explore the relationship between the expansion range and the expansion rate (average log divergence) visually by using **Explore**. Select a portion of the plot that is linear, using integers to bound the region. The bounding values transfer automatically to the **Min** and **Max Expansion range** settings when you close the figure.
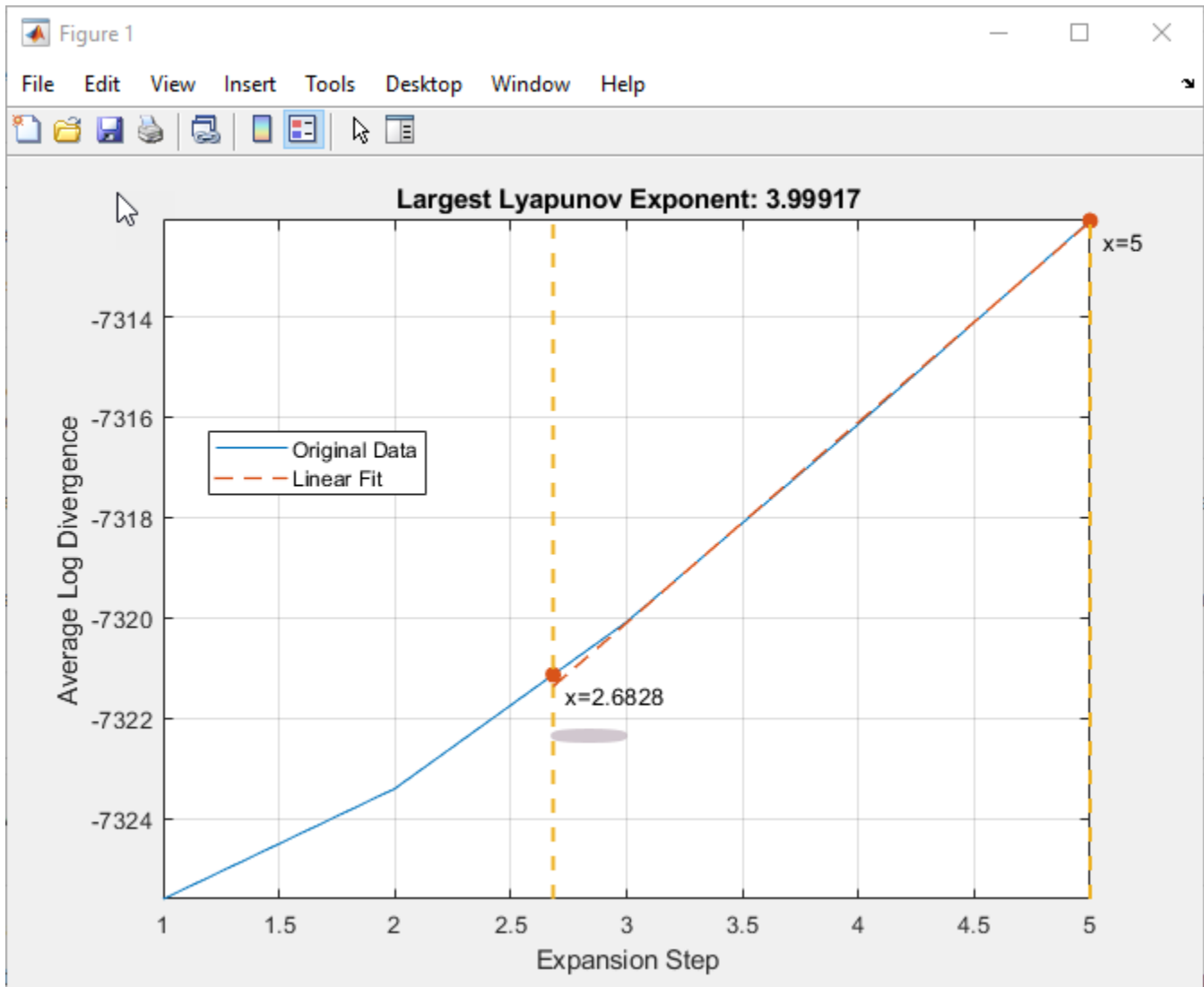
- **Mean period** — Threshold integer value used to find the nearest neighbor for a specific point to estimate the largest Lyapunov exponent. The software bases the default value on the mean frequency of the signal.

For more information on the Lyapunov exponent, see `lyapunovExponent`.

Additional Information

The software stores the results of the computation in new features. The new feature names include the source signal name with the suffix `nonlin`.

Spectral Features

Spectral features provide general frequency-domain metrics on your data. To compute spectral features, you must already have a power spectrum or an order spectrum variable.

Frequency Band

Set the minimum and maximum values for the frequency range within which the spectral features are computed.

Spectral Peaks

- **Peak amplitude** — Generate a feature based on the amplitude of the peaks.
- **Peak frequency** — Generate a feature based on the frequency of the peaks.
- **Number of peaks** — Number of peaks to generate features for. The software selects N most prominent peaks in the chosen frequency band, going in the descending amplitude order. For more information, in `findpeaks`, see the `NPeaks` name-value pair argument.
- **Peak value lower threshold** — Constrain peak size to exclude low-amplitude peaks. For more information, in `findpeaks`, see the `MinPeakHeight` name-value pair argument.
- **Minimum frequency gap** — Specify a minimum frequency gap. If the gap between two peaks is less than this specification, the software ignores the smaller peak of the pair. For more information, in `findpeaks`, see the `MinPeakDistance` name-value pair argument.
- **Peak excursion tolerance** — Specify the minimum prominence of a peak. The prominence of a peak measures how much the peak stands out due to its intrinsic height and its location relative to other peaks. For more information, in `findpeaks`, see the `MinPeakProminence` name-value pair argument.

For more information, see `findpeaks`.

Modal Coefficients

Modal coefficient features extract modal parameters from frequency response information. The setting for **Number of peaks** determines the number of modal coefficients to return.

- **Natural frequency** — Natural frequencies corresponding to each mode
- **Damping factor** — Damping factor for each mode

For more information, see `modalfit`.

Band Power

Band power represents the power of the signal in the selected frequency band. It is defined as the area under the spectrum curve within the chosen band limits.

Additional Information

The software stores the results of the computation in new features. The new feature names include the source signal name with the suffix `spec`.

Spectral Features Based on Fault Bands

Rotating machinery produces a spectrum that reflects the harmonics of the system. These harmonics result from the relationships among the rotating components, such as gear ratios and bearing ball size. When you know the physical characteristics of your bearing or gear mesh component, you can identify the characteristic frequencies of the component where failures can occur. Fault bands are frequency bands that bound these characteristic frequencies. Rather than compute spectral metrics across the entire frequency range, you can concentrate app analysis on frequencies within the fault bands, and extract more focused and effective features that indicate specific mechanical faults.

The app provides three types of fault band-based metrics.

- **Bearing Faults**
- **Gear Mesh Faults**
- **Custom Faults**

Use **Bearing Faults Features** and **Gear Mesh Faults Features** when you know the physical parameters of your bearing or gear mesh component and want the app to compute fault bands from those parameters. Use **Custom Faults Features** to specify the fault bands directly. Selecting any of these options opens a tab for you to specify the parameters. When you use specify rotation or fundamental frequency using **Constant (rpm)** or **Constant (Hz)** respectively, the app overlays a preview of the fault bands on the spectral plot. If you change a parameter on the corresponding tab, the app updates the preview. In the **Features** section of the tab, you can select the features that you want.

The **Bearing Faults Features** and **Gear Mesh Fault Features** tabs contain similar specifications, as the following table shows.

Type	Rotation	Physical Parameters	Defect Types	Fault Band Settings	Features
Bearing Faults Features	Rotation in rpm	Bearing parameters • Number of balls • Ball and pitch diameters • Contact angle	• Outer race • Inner race • Rolling element • Cage	• Harmonics • Sidebands (inner race, ball) • Fault band width • Fault band folding • Number of generated features	• Peak frequency • Band power • Peak amplitude • Total band power

| Gear Mesh Faults Features | | Gear mesh parameters

• Number of input and output teeth | • Input shaft
• Output shaft
• Assembly phase pass
• Gear mesh | • Harmonics
• Sidebands (input gear, output gear)
• Fault band width
• Fault band folding
• Number of generated features | |

The **Custom Faults Features** tab contains the specifications shown in the following table.

Type	Fundamental Frequency	Fault Band Settings	Advanced Settings	Features
Custom Faults Features	Fundamental frequency in Hz	• Harmonics • Sidebands • Fault band width • Sideband separation	• Sideband separation type (additive or multiplicative) • Fault band folding • Number of generated features	• Peak frequency • Band power • Peak amplitude • Total band power

For more information about the parameters used for fault band metrics, see the Live Editor task description in **Extract Spectral Features**. For more information about the commands that the app uses, see

- `faultBands`
- `faultBandMetrics`
- `gearMeshFaultBands`
- `bearingFaultBands`

Group Distances

Select group distance to determine the separation between data groups for each condition label pair. This separation metric — the KS statistic — indicates numerically how effective a feature is at differentiating between, say, faulty and healthy data.

Group distance is especially useful when you have more than two labels for a condition variable, as histograms become harder to interpret when data from multiple color groups combine.

Select the feature you want to examine from **Show grouping for feature**. The table shows the KS statistic for each label pairing. This statistic ranges from 0 to 1.

- A value of 0 means that the data groups are completely mixed, and therefore that the condition value is completely ambiguous. The associated feature has no differentiation capability for this data
- A value of 1 means that the data groups are well separated, and that the associated feature has complete differentiation capability for this data.

Additional Information

The KS statistic indicates how well separated the cumulative distribution functions of the distributions of the two states are, using the two-sample Kolmogorov-Smirnov test. For more information on this test, see `kstest2`.

Feature Selector

Use the **Feature Selector** dialog box when you want to focus on a subset of the features that you have computed. You can specify the features you want to view in the histogram set or in the feature trace plots. By default, the app selects all features. To filter out features that you do not want to view, clear those selections. Alternatively, you can use **Unselect All** to clear all the selections, and then reselect the features that you do want to view. Your selections for histograms and for feature trace plots are independent of each other.

Clearing a feature does not delete the feature from the feature table. You can always reselect the feature by clicking **Select Features** in the **Histogram** tab or the **Feature Trace** tab to restore **Feature Selector**. If you generate new features after using **Feature Selector**, the app automatically adds these features to the selection.

Feature Order

The order that the features are listed depends on where they are displayed.

- **Variables** pane — Feature groups are listed alphabetically. Individual features within the groups are in computation order.
- **Feature selector** — All features are listed together in ranked or alphabetical order, depending on the specification in **Features sorted by**. Specify `Name` to sort features alphabetically. Specify one of the ranking methods, such as `T-Test`, to sort features by ranking.
- **Histogram** plots — By default, the selected features are shown in ranked order if you have performed ranking, or in alphabetical order if you have not. You can change the histogram order with **Feature selector**. Each time you plot histograms within a session, the newest plot retains the order setting of the previous plot.
- **Feature Trace** plots — The selected features are shown together in ranked or alphabetical order, as specified in **Features sorted by**. Specify `Name` for alphabetical order. Specify a listed ranking method, such as `T-Test`, for ranked order.
- **Feature Table View** — All feature values are arranged in computation order.

For an example of using **Select Features** to filter a large feature set, see "Generate Features Automatically in Diagnostic Feature Designer" on page 7-154

Export Features to MATLAB Workspace

Export features to the MATLAB workspace when:

- You want to perform further analysis and visualization in the command window.
- You want to save the feature set outside of the app, or export the feature set into a different app.
- You want to incorporate the features into external data files. For example, suppose that you are working through an ensemble datastore, and had elected in **Computation Options** to write processing results to the local `inAppData` dataset. You can export your selected features to the MATLAB workspace. Then take the steps to incorporate the new feature values into your external files from the MATLAB Command Window.

When you export features, the app brings up a selectable list of features to export. The specific list depends on where you execute the export.

- If you export from the **Feature Designer** tab, the list is in alphabetical order, with all features preselected. This approach allows you to export all features at once from the main tab. You can also tailor the selections at this level if you know which features you want.

- If you export from the **Feature Ranking** tab, the list is in ranked order, based on the ranking method in **Features sorted by**. In this ranked list, the top five features are preselected. This approach allows you to export only your highest-ranked features. You can also tailor the selections at if you want to export more than the top five features.

You can export your entire dataset, including derived variables and all your features, to the MATLAB workspace as well. To do so, in the **Feature Designer** tab, use **Export > Export a dataset to the MATLAB workspace**.

More Information

For more information on writing to ensemble datastore files in the command line, see "File Ensemble Datastore with Measured Data" on page 1-17.

Export Features to Classification Learner

Classification Learner is an app in the Statistics and Machine Learning Toolbox that trains models to classify data. **Classification Learner** uses automated methods to create and test different types of classification models using labeled datasets. For predictive maintenance, the goal of using **Classification Learner** is to select and train a model that uses the exported feature set to discriminate between data from healthy and from faulty systems. You can incorporate this model into an algorithm for fault detection and prediction.

Use **Export features to Classification Learner** when:

- You want to obtain more insight on the relative effectiveness of your features.
- You are developing a predictive maintenance algorithm and want to select and train the best model for the algorithm to use.

When you export features, the app brings up a selectable list of features to export. The specific list depends on where you execute the export.

- If you export from the **Feature Designer** tab, the list is in alphabetical order, with all features preselected. This approach allows you to export all features at once from the main tab. You can also tailor the selections at this level if you know which features you want.

- If you export from the **Feature Ranking** tab, the list is in ranked order, based on the ranking method in **Features sorted by**. In this ranked list, the top five features are preselected. This approach allows you to export only your highest-ranked features. You can also tailor the selections at if you want to export more than the top five features.

More Information

For more information, see **Classification Learner**.

Export a Dataset to the MATLAB Workspace

Export your dataset to the MATLAB workspace when:

- You want to perform further analysis and visualization in the command window.
- You want to save derived variables and features outside of the app.
- You want to incorporate features and derived variables into external data files. For example, suppose that you are working through an ensemble datastore, and had elected in **Computation Options** to write processing results to the local `inAppData` dataset. At the end of your session, you can export that dataset to the MATLAB workspace. Then, take the steps to incorporate the variables and feature values that you want to retain into your external files from the MATLAB Command Window.

When you export data to the MATLAB workspace, you export the entire dataset that you select, including variables, features, and condition variables.

More Information

For more information on writing in-app data to ensemble datastore files at the command line, see "File Ensemble Datastore with Measured Data" on page 1-17.

Generate Function for Features

Generate MATLAB code for features you select when you want a command-line version of the computations you performed interactively in the app. The app provides code that reproduces both the preliminary data processing and the feature extraction itself. For example, if you select features that are based on power spectra that you computed in the app, the generated code includes the spectral processing as well as the feature processing.

You can choose between two code formats:

1 Function formatted for use with ensemble data sets. This is the default format. Use this option when you want to apply your feature extraction computations to other ensemble data sets. For example, use this option when you want to apply features developed using a small data set to a larger data set.

2 Function formatted for streaming. Use this option when you want to perform feature extraction on streaming data. The function accepts streaming data and is MATLAB Coder compliant so you can convert the function to C or C++. You cannot export computations based on spectral variables that are computed using autoregressive or state-space models. To export spectral features in a streaming function, use the Welch method to compute the spectrum variable. You also cannot export features derived from ensemble statistics variables or features derived from frame-based data.

You can select all the features in a single feature table, or select only the top-ranked features in that table.

In **Feature table**, select the feature table you want to work with.

In **Features sorted by**, choose the ranking algorithm you want to use to order the features for selection. Only the ranking algorithms that you applied during your session appear as options. If you want to generate code for all features regardless of ranking, select `Use All Features`.

In **Number of top features**, choose how many features you want to generate code for. This option is active only when you are exporting ranked features. If you do not specify a ranking method, the app generates code for all features.

In **Format for streaming data**, select the option if you want to use the function on streaming data rather than on an ensemble data set.

When you click **OK**, the app generates the code in a function in the editor.

If you want to customize your selection further, such as by filtering the features for specific inputs or methods, use **Generate Function for...**. Selecting this option opens a list of all the signals, features, and ranking tables you can choose from, and also opens the **Code Generation** tab. The **Code Generation** tab allows you to filter the items in the selection list to refine what the generated code includes. View all your selections together with no filters by clicking **Sort by Selection**. When your selections are complete, click **Generate Function**. **Generate Function for...** does not support streaming data.

For more information, see:

- "Generate a MATLAB Function in Diagnostic Feature Designer" on page 7-94
- "Automatic Feature Extraction Using Generated MATLAB Code" on page 7-86

- "Export Feature Extraction Function and Simulink Model for Streaming Data" on page 7-179

Generate Feature Extraction Simulink Block

Generate MATLAB code for features you select when you want a Simulink block that incorporates the feature computations you performed interactively in the app.

You can select all the features in a single feature table, or select only the top-ranked features in that table.

In **Feature table**, select the feature table you want to work with.

In **Features sorted by**, choose the ranking algorithm you want to use to order the features for selection. Only the ranking algorithms that you applied during your session appear as options. If you want to generate code for all features regardless of ranking, select None.

In **Number of top features**, choose how many features you want to generate code for. This option is active only when you are exporting ranked features. If you do not specify a ranking method, the app generates code for all features but features that are restricted for streaming code, as discussed in the following paragraph.

You cannot export computations based on spectral variables that are computed autoregressive or state-space models. To export spectral features in a streaming function, use the Welch method to compute the spectrum. You also cannot export features based on ensemble statistics variables.

When you click **OK**, the app generates a Simulink block that includes the streaming code.

For more information, see "Export Feature Extraction Function and Simulink Model for Streaming Data" on page 7-179.