Model-Based Calibration Toolbox™

Reference

MATLAB® & SIMULINK®

R2022b

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

Model-Based Calibration Toolbox™ Reference

Revision History

November 2005	Online only	New for Version 3.0 (Release 14SP3+)
September 2006	Online only	Version 3.1 (Release 2006b)
March 2007	Online only	Version 3.2 (Release 2007a)
September 2007	Online only	Revised for Version 3.3 (Release 2007b)
March 2008	Online only	Revised for Version 3.4 (Release 2008a)
October 2008	Online only	Revised for Version 3.4.1 (Release 2008a+)
October 2008	Online only	Revised for Version 3.5 (Release 2008b)
March 2009	Online only	Revised for Version 3.6 (Release 2009a)
September 2009	Online only	Revised for Version 3.7 (Release 2009b)
March 2010	Online only	Revised for Version 4.0 (Release 2010a)
September 2010	Online only	Revised for Version 4.1 (Release 2010b)
April 2011	Online only	Revised for Version 4.2 (Release 2011a)
September 2011	Online only	Revised for Version 4.3 (Release 2011b)
March 2012	Online only	Revised for Version 4.4 (Release 2012a)
September 2012	Online only	Revised for Version 4.5 (Release 2012b)
March 2013	Online only	Revised for Version 4.6 (Release 2013a)
September 2013	Online only	Revised for Version 4.6.1 (Release 2013b)
March 2014	Online only	Revised for Version 4.7 (Release 2014a)
October 2014	Online only	Revised for Version 4.8 (Release 2014b)
March 2015	Online only	Revised for Version 4.8.1 (Release 2015a)
September 2015	Online only	Revised for Version 5.0 (Release 2015b)
March 2016	Online only	Revised for Version 5.1 (Release 2016a)
September 2016	Online only	Revised for Version 5.2 (Release 2016b)
March 2017	Online only	Revised for Version 5.2.1 (Release 2017a)
September 2017	Online only	Revised for Version 5.3 (Release 2017b)
March 2018	Online only	Revised for Version 5.4 (Release 2018a)
September 2018	Online only	Revised for Version 5.5 (Release 2018b)
March 2019	Online only	Revised for Version 5.6 (Release 2019a)
September 2019	Online only	Revised for Version 5.7 (Release 2019b)
March 2020	Online only	Revised for Version 5.8 (Release 2020a)
September 2020	Online only	Revised for Version 5.9 (Release 2020b)
March 2021	Online only	Revised for Version 5.10 (Release 2021a)
September 2021	Online only	Revised for Version 5.11 (Release 2021b)
March 2022	Online only	Revised for Version 5.12 (Release 2022a)
September 2022	Online only	Revised for Version 5.13 (Release 2022b)

Contents

10

Commands

MBC Model Fitting

Create experimental designs and statistical models for model-based calibration

Description

The **MBC Model Fitting** app enables you to create experimental designs, fit statistical models to engine data, and export the models to generate optimal calibrations in MBC Optimization app. You can also export models to MATLAB® and Simulink® to reuse statistical models for control design, hardware-in-the-loop testing, or powertrain simulation.

Open the MBC Model Fitting App

- MATLAB Toolstrip: On the **Apps** tab, under **Automotive**, click the app icon.
- MATLAB command prompt: Enter `mbcmodel`.

Examples

- “Model Set Up”
- “Multi-Injection Diesel Calibration”
- “Gasoline Engine Calibration”
- “Model Export to MATLAB, Simulink, and CAGE”

Programmatic Use

`mbcmodel` opens the Model Browser or brings an existing Model Browser to the front of the screen.

`mbcmodel fileName.mat` opens the Model Browser and loads the file specified by *`fileName`*`.mat`.

Version History

Introduced before R2006a

See Also

Apps
MBC Optimization

Topics
“Model Set Up”
“Multi-Injection Diesel Calibration”
“Gasoline Engine Calibration”
“Model Export to MATLAB, Simulink, and CAGE”

MBC Optimization

Generate optimal lookup tables for model-based calibration

Description

The **MBC Optimization** app lets you use statistical models created in MBC Model Fitting app to generate optimal calibrations for lookup tables that control engine functions. You can generate calibrations and lookup tables for complex, high-degree-of-freedom engines to identify the optimal balance of engine performance, emissions, and fuel economy.

Open the MBC Optimization App

- MATLAB Toolstrip: On the **Apps** tab, under **Automotive**, click the app icon.
- MATLAB command prompt: Enter `cage`.

Examples

- "Calibration Setup"
- "Optimization"
- "Feature Calibration"
- "Tradeoff Calibration"

Programmatic Use

`cage` opens the CAGE Browser or brings an existing CAGE Browser to the front of the screen. CAGE stands for Calibration Generation.

`cage fileName.cag` opens the CAGE Browser and loads the file specified by *`fileName`*.

Version History

Introduced before R2006a

See Also

Apps
MBC Model Fitting

Topics
"Calibration Setup"
"Optimization"
"Feature Calibration"
"Tradeoff Calibration"

ActiveInputs

Active boundary model inputs

Syntax

```
B.ActiveInputs = [X]
```

Description

`ActiveInputs` is a property of `mbcboundary.Model`.

`B.ActiveInputs = [X]` sets the active inputs for the boundary model. `X` is a logical row vector indicating which inputs to use to fit a boundary. You can build boundary models using subsets of input factors and then combine them for the most accurate boundary. This approach can provide more effective results than including all inputs.

Examples

To make a boundary model using only the first two inputs:

```
B.ActiveInputs = [true true false false];
```

Version History

Introduced in R2009b

Add

Add boundary model to tree and fit to test plan data

Syntax

```
B = Add(Tree,B)
B = Add(Tree,B,InBest)
```

Description

This is a method of `mbcboundary.Tree`.

`B = Add(Tree,B)` adds the boundary model to the tree and fits the boundary model to the test plan data. `Tree` is an `mbcboundary.Tree` object, `B` is a new boundary model object. The boundary model must have the same inputs as the boundary tree. The boundary model is always fitted when you add it to the boundary tree. This fitting ensures that the fitting data is compatible with the test plan data. The method returns the fitted boundary model.

`B = Add(Tree,B,InBest)` adds and fits the boundary model, and `InBest` specifies whether to include the boundary model in the best boundary model for the boundary tree. By default, the best model includes the new boundary model.

Version History

Introduced in R2009b

See Also

`Update` | `Remove` | `CreateBoundary`

AlternativeModelStatistics

Summary statistics for alternative models

Syntax

```
S = AlternativeModelStatistics(R)
```

```
S = AlternativeModelStatistics(R, Name)
```

Description

This is a method of all model objects: `mbcmodel.hierarchicalresponse`, `mbcmodel.localresponse` and `mbcmodel.response`.

This returns an array (`S`) of summary statistics of all the alternative model fits, to be used to select the best model. These are the summary statistics seen in the list view at the bottom of the Model Browser GUI in any model view.

You must use CreateAlternativeModels before you can compare the alternative responses using AlternativeModelStatistics. Then use ChooseAsBest.

`R` is the model object whose alternative response models you want to compare. `R` could be a local (`L`), response feature (`R`) or hierarchical response (`HR`) model.

`S` is a structure containing `Statistics` and `Names` fields.

- `S.Statistics` is a matrix of size (number alternative responses x number of statistics).
- `S.Names` is a cell array containing the names of all the statistics.

The available statistics vary according to what kind of parent model (two-stage, local, response feature or response) produced the alternative models, and include PRESS RMSE, RMSE, and Two-Stage RMSE.

All the available statistics are calculated unless you specify which you want. You can specify only the statistics you require using the following form:

```
S = AlternativeModelStatistics(R, Name)
```

This returns a double matrix containing only the statistics specified in `Name`.

Note that you use `SummaryStatistics` to examine the fit of the current model, and `AlternativeModelStatistics` to examine the fit of several alternative child models.

Examples

```
S = AlternativeModelStatistics(R);
```

Version History

Introduced before R2006a

See Also

`CreateAlternativeModels` | `SummaryStatistics` | `ChooseAsBest`

AlternativeResponses

Array of alternative responses for this response

Syntax

```
altR = R.AlternativeResponses
```

Description

This is a property of the response model object, `mbcmodel.response` (R).

It returns a list of alternative responses used for one-stage or response feature models.

Examples

```
R = testplan.Responses;
TQ = R(1);
AR = TQ.AlternativeResponses;
```

See Also

`LocalResponses` | `ResponseFeatures(Local Response)`

BestModel property

Combined best boundary models

Syntax

`mbcboundary.Tree.BestModel`

Description

This is a property of `mbcboundary.Tree` and `mbcboundary.TwoStageTree`.

`mbcboundary.Tree.BestModel` returns the combined boundary model containing all best boundary models in the tree (read only).

`BestModel` is the boundary model combining the models selected as best. You can select which boundary models to include in the best model with `InBest`. If the best boundary model includes more than one boundary model, that boundary model is an `mbcboundary.Boolean` object.

For `TwoStageTree` objects, the `BestModel` property contains the best boundary model (local, global, and response) (read only). In this case, `BestModel` is the boundary model combining the best local, global and response boundary models. You can select which boundary models to include in the best model with `InBest`. If the best boundary model includes more than one boundary model, that boundary model is an `mbcboundary.Boolean` object.

See Also

`InBest`

ChooseAsBest

Choose best model from alternative responses

Syntax

```
ChooseAsBest(R, Index)
```

Description

This is a method of the response model object, `mbcmodel.response`. This is the same function as selecting the best model in the Model Selection window of the Model Browser GUI. For a local model `MakeHierarchicalResponse` performs a similar function.

`R` is the object containing the response model.

`Index` is the number of the response model you want to choose as best. Use `AlternativeResponses` to find the index for each response model, and use AlternativeModelStatistics to choose the best fit.

Examples

```
ChooseAsBest(R, AlternativeModel)
RMSE = AlternativeModelStatistics(R, 'RMSE');
[mr, Best] = min(RMSE);
ChooseAsBest(R, Best);
```

Version History

Introduced before R2006a

See Also

`AlternativeResponses` | `AlternativeModelStatistics` | `DiagnosticStatistics` | `MakeHierarchicalResponse`

CreateAlgorithm

Create algorithm

Syntax

```
newalg = alg.CreateAlgorithm( AlgorithmName)
```

Description

This is a method of `mbcmodel.fitalgorithm`.

`newalg = alg.CreateAlgorithm( AlgorithmName)` creates an algorithm of the specified type. `alg` is a `mbcmodel.fitalgorithm` object. `AlgorithmName` must be in the list of alternative algorithms given by `alg.getAlternativeNames`.

To change the fit algorithm for a model:

```
>> model = mbcmodel.CreateModel('Polynomial', 2);
>> minpress = model.FitAlgorithm.CreateAlgorithm('Minimize PRESS');
>> model.FitAlgorithm = minpress;
```

The `AlgorithmName` determines what properties you can set. You can display the properties for an algorithm as follows:

```
>> model.FitAlgorithm.properties

Algorithm: Minimize PRESS
Alternatives: 'Least Squares','Forward Selection','Backward
Selection','Prune'
     MaxIter: Maximum Iterations (int: [1,1000])
```

As a simpler alternative to using CreateAlgorithm, you can assign the algorithm name directly to the algorithm. For example:

```
B.FitAlgorithm.BoundaryPointOptions = 'Boundary Only';
```

Or:

```
m.FitAlgorithm = 'Minimize PRESS';
```

The following sections list the properties available for each algorithm type.

Linear Model Algorithm Properties

Linear Models Algorithms

Used by polynomials, hybrid splines and as the StepAlgorithm for RBF algorithms.

Algorithm: Least Squares

Alternatives: `'Minimize PRESS','Forward Selection','Backward Selection','Prune'`

Algorithm: Minimize PRESS

Alternatives: `'Least Squares','Forward Selection','Backward Selection','Prune'`

- MaxIter: Maximum Iterations (int: [1,1000])

Algorithm: Forward Selection

Alternatives: `'Least Squares','Minimize PRESS','Backward Selection','Prune'`

- ConfidenceLevel: Confidence level (%) (numeric: [70,100])
- MaxIter: Maximum Iterations (int: [1,1000])
- RemoveAll: Remove all terms first (Boolean)

Algorithm: Backward Selection

Alternatives: `'Least Squares','Minimize PRESS','Forward Selection','Prune'`

- ConfidenceLevel: Alpha (%) (numeric: [70,100])
- MaxIter: Maximum Iterations (int: [1,1000])
- IncludeAll: Include all terms first (Boolean)

Algorithm: Prune

Alternatives: `'Least Squares','Minimize PRESS','Forward Selection','Backward Selection'`

- Criteria (PRESS RMSE|RMSE|GCV|Weighted PRESS|-2logL|AIC|AICc|BIC|R^2|R^2 adj|PRESS R^2|DW|Cp|cond(J))
- MinTerms: Minimum number of terms (int: [0,Inf])
- Tolerance (numeric: [0,1000])
- IncludeAll: Include all terms before prune (Boolean)
- Display (Boolean)

RBF Algorithm Properties

For information about any of the RBF and Hybrid RBF algorithm properties, see “Radial Basis Functions for Model Building”, and especially “Fitting Routines” in the Model Browser User's Guide.

Algorithm: RBF Fit

- WidthAlgorithm: Width selection algorithm (mbcmodel.fitalgorithm)
- StepAlgorithm: Stepwise (mbcmodel.fitalgorithm)

Width Selection Algorithms

Alternatives: `'WidPerDim','Tree Regression'`

Algorithm: TrialWidths

- NestedFitAlgorithm: Lambda selection algorithm (mbcmodel.fitalgorithm)
- Trials: Number of trial widths in each zoom (int: [2,100])

- Zooms: Number of zooms (int: [1,100])
- MinWidth: Initial lower bound on width (numeric: [2.22045e-016,1000])
- MaxWidth: Initial upper bound on width (numeric: [2.22045e-016,100])
- PlotFlag: Display plots (Boolean)
- PlotProgress: Display fit progress (Boolean)

Algorithm: WidPerDim

Alternatives: `'TrialWidths','Tree Regression'`

- NestedFitAlgorithm: Lambda selection algorithm (mbcmodel.fitalgorithm)
- DisplayFlag: Display (Boolean)
- MaxFunEvals: Maximum number of test widths (int: [1,1e+006])
- PlotProgress: Display fit progress (Boolean)

Algorithm: Tree Regression

Alternatives: `'TrialWidths','WidPerDim'`

- MaxNumRectangles: Maximum number of panels (int: [1,Inf])
- MinPerRectangle: Minimum data points per panel (int: [2,Inf])
- RectangleSize: Shrink panel to data (Boolean)
- AlphaSelectAlg: Alpha selection algorithm (mbcmodel.fitalgorithm)

Lambda Selection Algorithms

Algorithm: IterateRidge

Alternatives: `'IterateRols','StepItRols'`

- CenterSelectionAlg: Center selection algorithm (mbcmodel.fitalgorithm)
- MaxNumIter: Maximum number of updates (int: [1,100])
- Tolerance: Minimum change in log10(GCV) (numeric: [2.22045e-016,1])
- NumberOfLambdaValues: Number of initial test values for lambda (int: [0,100])
- CheapMode: Do not reselect centers for new width (Boolean)
- PlotFlag: Display (Boolean)

Algorithm: IterateRols

Alternatives: `'IterateRidge','StepItRols'`

- CenterSelectionAlg: Center selection algorithm (mbcmodel.fitalgorithm)
- MaxNumIter: Maximum number of iterations (int: [1,100])
- Tolerance: Minimum change in log10(GCV) (numeric: [2.22045e-016,1])
- NumberOfLambdaValues: Number of initial test values for lambda (int: [0,100])
- CheapMode: Do not reselect centers for new width (Boolean)
- PlotFlag: Display (Boolean)

Algorithm: StepItRols

Alternatives: `'IterateRidge','IterateRols'`

- MaxCenters: Maximum number of centers (evalstr)
- PercentCandidates: Percentage of data to be candidate centers (evalstr)
- StartLambdaUpdate: Number of centers to add before updating (int: [1,Inf])
- Tolerance: Minimum change in log10(GCV) (numeric: [2.22045e-016,1])
- MaxRep: Maximum number of times log10(GCV) change is minimal (int: [1,100])

Center Selection Algorithms

Algorithm: Rols

Alternatives: `'RedErr','WiggleCenters','CenterExchange'`

- MaxCenters: Maximum number of centers (evalstr)
- PercentCandidates: Percentage of data to be candidate centers (evalstr)
- Tolerance: Regularized error tolerance (numeric: [2.22045e-016,1])

Algorithm: RedErr

Alternatives: `'Rols','WiggleCenters','CenterExchange'`

- MaxCenters: Number of centers (evalstr)

Algorithm: WiggleCenters

Alternatives: `'Rols','RedErr','CenterExchange'`

- MaxCenters: Number of centers (evalstr)
- PercentCandidates: Percentage of data to be candidate centers (evalstr)

Algorithm: CenterExchange

Alternatives: `'Rols','RedErr','WiggleCenters'`

- MaxCenters: Number of centers (evalstr)
- NumLoops: Number of augment/reduce cycles (int: [1,Inf])
- NumAugment: Number of centers to augment by (int: [1,Inf])

Tree Regression Algorithms

Algorithm: Trial Alpha

Alternatives: `'Specify Alpha'`

- AlphaLowerBound: Initial lower bound on alpha (numeric: [2.22045e-016,Inf])
- AlphaUpperBound: Initial upper bound on alpha (numeric: [2.22045e-016,Inf])
- Zooms: Number of zooms (int: [1,Inf])
- Trials: Trial alphas per zoom (int: [2,Inf])
- Spacing: Spacing (LinearLogarithmic)
- CenterSelectAlg: Center selection algorithm (mbcmodel.fitalgorithm)

Algorithm: Specify Alpha

Alternatives: `'Trial Alpha'`

- Alpha: Width scale parameter, alpha (numeric: [2.22045e-016,Inf])
- NestedFitAlgorithm: Center selection algorithm (mbcmodel.fitalgorithm)

Algorithm: Tree-based Center Selection

Alternatives: `'Generic Center Selection'`

- ModelSelectionCriteria: Model selection criteria (BIC|GCV)
- MaxNumberCenters: Maximum number of centers (evalstr)

Algorithm: Generic Center Selection

Alternatives: `'Tree-based Center Selection'`

- CenterSelectAlg: Center selection algorithm (mbcmodel.fitalgorithm)

Hybrid RBF Algorithms

Algorithm: RBF Fit

- WidthAlgorithm: Width selection algorithm (mbcmodel.fitalgorithm)
- StepAlgorithm: Stepwise (mbcmodel.fitalgorithm)

Width Selection Algorithms

Algorithm: TrialWidths

- NestedFitAlgorithm: Lambda and term selection algorithm (mbcmodel.fitalgorithm)
- Trials: Number of trial widths in each zoom (int: [2,100])
- Zooms: Number of zooms (int: [1,100])
- MinWidth: Initial lower bound on width (numeric: [2.22045e-016,1000])
- MaxWidth: Initial upper bound on width (numeric: [2.22045e-016,100])
- PlotFlag: Display plots (Boolean)
- PlotProgress: Display fit progress (Boolean)

Nested Fit Algorithms

Algorithm: Twostep

Alternatives: `'Interlace'`

- MaxCenters: Maximum number of centers (evalstr)
- PercentCandidates: Percentage of data to be candidate centers (evalstr)
- StartLambdaUpdate: Number of terms to add before updating (int: [1,Inf])
- Tolerance: Minimum change in log10(GCV) (numeric: [2.22045e-016,1])
- MaxRep: Maximum number of times log10(GCV) change is minimal (int: [1,100])
- PlotFlag: Display (Boolean)

Algorithm: Interlace

Alternatives: `'Twostep'`

- MaxParameters: Maximum number of terms (evalstr)
- MaxCenters: Maximum number of centers (evalstr)
- PercentCandidates: Percentage of data to be candidate centers (evalstr)
- StartLambdaUpdate: Number of terms to add before updating (int: [1,Inf])
- Tolerance: Minimum change in log10(GCV) (numeric: [2.22045e-016,1])
- MaxRep: Maximum number of times log10(GCV) change is minimal (int: [1,100])

Boundary Model Fit Algorithm Parameters

The following sections list the available fit algorithm parameters for command-line boundary models. The boundary model fit algorithm parameters have the same fit options as the Boundary Editor GUI. For instructions on using these fit options, see “Editing Boundary Model Fit Options” in the Model Browser documentation.

Convex Hull

KeepAllFacets: Boolean to indicate whether to keep all facets (default is `false`, do not keep all facets).

Tolerance: Tolerance for maximum 1-norm distance allowed for removing facets (numeric: [0,Inf], default 0.02). To remove more facets, increase the tolerance.

For more information, see “Convex Hull Setting” in the Model Browser documentation.

Ellipsoid

Algorithm: Constraint Fitting

BoundaryPointOptions: Boundary Points (mbcmodel.fitalgorithm)

The boundary points algorithm uses optimization to find the best ellipse. These options are from `fmincon`.

Algorithm: Boundary Points

- Display: Display (none|iter|final)
- MaxFunEvals: Maximum function evaluations (int: [1,Inf])
- MaxIter: Maximum iterations (int: [1,Inf])
- TolFun: Function tolerance (numeric: [1e-012,Inf])
- TolX: Variable tolerance (numeric: [1e-012,Inf])
- TolCon: Constraint tolerance (numeric: [1e-012,Inf])

Star-shaped

Algorithm: Constraint Fitting

SpecialPointOptions: Special Points (mbcmodel.fitalgorithm)

BoundaryPointOptions: Boundary Points (mbcmodel.fitalgorithm)

ConstraintFitOptions: Constraint Fit (mbcmodel.fitalgorithm)

Star-shaped—Special Points

Algorithm: Star-shaped Points

CenterAlg: Center (mbcmodel.fitalgorithm)

Algorithm alternatives: 'Mean', 'Median', 'Mid Range', 'Min Ellipse', 'User Defined'

For User Defined only: CenterPoint: User-defined center [X1,X2] (vector: NumberOfActiveInputs)

Star-shaped—Boundary Points

You can choose to find boundary points (use `Interior`) or to assume that all points are on the boundary (use `Boundary Only`). The interior algorithm then has manual and auto options for the dilation radius and ray casting algorithms.

- Algorithm: Boundary Only (no further options)
- Algorithm: Interior. Further options:
  - DilationRadius (mbcmodel.fitalgorithm)
    - Algorithm: Auto
    - Algorithm: Manual
      - radius: Radius (numeric: [0,Inf])
  - RayCasting (mbcmodel.fitalgorithm)
    - Algorithm: From data
    - Algorithm: Manual
      - nrays: Number of Rays (int: [1,Inf])

Star-shaped—Constraint Fit

Algorithm: Star-shaped RBF Fit

Further options:

- Transform (None|Log|McCallum)
- KernelOpts: RBF Kernel (mbcmodel.fitalgorithm)

  Kernel algorithms can be: wendland, multiquadric, recmultiquadric, gaussian, thinplate, logisticrbf. linearrbf, cubicrbf.

  You can specify widths and continuity as sub-properties of particular RBF kernels.

  - You can set widths for wendland, multiquadric, recmultiquadric, gaussian, logisticrbf. Width: RBF Width (numeric: [1.49012e-008,Inf])

    You can set Continuity for wendland. Cont: RBF Continuity (0|2|4|6)

  RbfOpts: RBF Algorithm (mbcmodel.fitalgorithm)

  Algorithm: Interpolation. The following are additional settings for interpolating RBF.

- CoincidentStrategy: Coincident Node Strategy (Maximum|Minimum|Mean)
- Algorithm: Algorithm (Direct|GMRES|BICG|CGS|QMR)
- Tolerance: Tolerance (numeric: [0,Inf])
- MaxIt: Maximum number of iterations (int: [1,Inf])

Examples

First get a fitalgorithm object, F, from a model:

```
M = mbcmodel.CreateModel('Polynomial', 4);
F = M.FitAlgorithm

F =
Algorithm: Least Squares
Alternatives: 'Minimize PRESS','Forward Selection','Backward
Selection','Prune'
1x1 struct array with no fields.
```

Then, to create a new algorithm type:

```
Alg = CreateAlgorithm(F, 'Minimize PRESS')

Alg =
Algorithm: Minimize PRESS
Alternatives: 'Least Squares','Forward Selection','Backward
Selection','Prune'
    MaxIter: 50
```

The `AlgorithmName` determines what properties you can set. You can display the properties for an algorithm as follows:

```
>> model.FitAlgorithm.properties

Algorithm: Minimize PRESS
Alternatives: 'Least Squares','Forward Selection','Backward
Selection','Prune'
     MaxIter: Maximum Iterations (int: [1,1000])
```

As a simpler alternative to using `CreateAlgorithm`, you can assign the algorithm name directly to the algorithm. For example:

```
B.FitAlgorithm.BoundaryPointOptions = 'Boundary Only';
```

Or:

```
m.FitAlgorithm = 'Minimize PRESS';
```

Case and spaces are ignored.

Version History

Introduced in R2007a

See Also

`getAlternativeNames` | `SetupDialog`

CreateAlternativeModels

Create alternative models from model template

Syntax

```
R = CreateAlternativeModels(R, modeltemplate, criteria)
```

```
R = CreateAlternativeModels(R, modellist, criteria)
```

```
R = CreateAlternativeModels(R, LocalModels,LocalCriteria,GlobalModels,GlobalCriteria)
```

Description

This is a method of all model objects: `mbcmodel.hierarchicalresponse`, `mbcmodel.localresponse` and `mbcmodel.response`.

This is the same as the Build Models function in the Model Browser GUI. A selection of child node models are built. The results depend on where you call this method from. Note that the hierarchical model is automatically constructed when `CreateAlternativeModels` is called for a local model.

- This option makes alternative response feature models for each response feature.

  ```
  R = CreateAlternativeModels(R, models, criteria)
  ```

  - `Models` is the list of models. You can use a model template file (`.mbm`) created in the Model Browser, or a cell array of `mbcmodel.model` objects.
  - `Criteria` is the selection criteria for best model (from the statistics available from `AlternativeModelStatistics`).
- This option makes alternative local models as well as alternative response feature models.

  ```
  R = CreateAlternativeModels(R,
  LocalModels,LocalCriteria,GlobalModels,GlobalCriteria)
  ```

  - `LocalModels` is the list of local models - you must pass in an empty matrix).
  - `LocalCriteria` is `'Two-Stage RMSE'`.
  - `GlobalModels` is the list of global models (from the model template).
  - `GlobalCriteria` is the selection criteria for best model.

You construct a model template file (such as `'mymodels.mbm'`) in the Model Browser. From any response (global or one-stage model) with alternative responses (child nodes), select **Model > Make Template**. You can save the child node model types of your currently selected modeling node as a model template. Alternatively from any response click **Build Models** in the toolbar and create a series of alternative response models in the dialog.

Examples

```
mymodels = 'mymodels.mbm';
mlist = {};
```

```
load('-mat', mymodels);
criteria = 'PRESS RMSE';
CreateAlternativeModels(R, [], 'Two-Stage RMSE', mlist,
criteria);
```

Note that the model template contains the variable `mlist`.

```
CreateAlternativeModels( RESPONSE, 'alternative_models.mbm', 'Weighted PRESS' )
```

creates alternative response feature models based upon the model template file `alternative_models.mbt`, and chooses the best model based upon each model's Weighted PRESS statistic.

Version History

Introduced before R2006a

See Also

`AlternativeModelStatistics`

CreateResponseFeature

Create new response feature for local model

Syntax

```
RF = CreateResponseFeature(RF,RFType)
RF = CreateResponseFeature(RF,RFType,EvaluationPoint)
```

Description

This is a method of `mbcmodel.localresponse`.

`RF = CreateResponseFeature(RF,RFType)` creates a response feature for `RFType`.

`RF = CreateResponseFeature(RF,RFType,EvaluationPoint)` creates a response feature for `RFType` at `EvaluationPoint`.

`RFType` is a description character vector belonging to the set of alternative response features for the current local model.

`EvaluationPoint` is a row vector with an element for each model input and is used for response features that require an input value to evaluate the response feature (e.g., function evaluation, derivatives). It is an error to specify an evaluation point for a response feature type that does not require an evaluation point.

You should use this method to add response features without refitting all local and global models.

Examples

```
RF = CreateResponseFeature(RF,'Beta_1')
```

Version History

Introduced in R2007b

See Also

`ResponseFeatures(Local Model)`

Data

Array of data objects in project, boundary tree, or test plan

Syntax

```
allD = project.Data
allD = testplan.Data
```

Description

This is a property of `mbcmodel.project`, `mbcmodel.testplan`, and `mbcboundary.Tree`.

For projects and test plans, it returns an array of `mbcmodel.data` objects. A project can have many data objects, but a test plan can only have one or none.

`Tree.B.Data` returns a double matrix for one-stage, response, and global boundary models. For local boundary models, `Data` is a cell array of double matrices with one cell per test. For boundary models, `Data` is read-only.

Examples

```
allD = P.Data;
```

For a project object `P`, this example returns an `nx1` array of all the data objects.

```
allD = T.Data;
```

For the test plan object T, this example returns a `1x1` array if the test plan has a data object attached, and `0x1` otherwise.

See Also

`CreateData` | `RemoveData` | `CopyData`

DiagnosticStatistics

Diagnostic statistics for response

Syntax

```
S = DiagnosticStatistics(R)
S = DiagnosticStatistics(R, Stats)
S = DiagnosticStatistics(LocalR, TestNumbers)
S = DiagnosticStatistics(LocalR, TestNumbers, Stats)
```

Description

This is a method of the local and response model objects, `mbcmodel.localresponse` and `mbcmodel.response`.

The options available are model-specific and are the same options shown in the drop-down menus of the scatter plots (the top plots) in the local and global (response feature) model views of the toolbox GUI.

`S = DiagnosticStatistics(R)` returns `S`, a structural array containing `Statistics` and `Names` fields. `R` is the response or local response model object.

`S = DiagnosticStatistics(R, Stats)` allows you to specify `Stats`, an optional input that defines which diagnostic statistics you want from the available list. If you don't specify `Stats`, you get all available statistics.

`S = DiagnosticStatistics(LocalR, TestNumbers)` returns `S` for `LocalR`, a local response object, and `Testnumbers` specifies the index into tests for local or hierarchical models.

Use `S = DiagnosticStatistics(LocalR, TestNumbers, Stats)` to specify which diagnostic statistics you want from the available list.

A row is set to `NaN` if that point is removed.

Examples

```
studentRes = DiagnosticStatistics(local, tn, 'Studentized
residuals');
```

Version History

Introduced before R2006a

See Also

`SummaryStatistics` | `AlternativeModelStatistics`

DoubleInputData

Data being used as input to model

Syntax

```
X = DoubleInputData(R, TestNumber)
```

Description

This is a method of all model objects: `mbcmodel.hierarchicalresponse`, `mbcmodel.localresponse` and `mbcmodel.response`. It returns an array (`X`) containing the input data used for fitting the model.

`R` is the response model object.

`TestNumber` is an optional input to specify the tests you want.

Examples

```
X = DoubleInputData(R);
x = DoubleInputData(local, tn);
```

Version History

Introduced before R2006a

See Also

`DoubleResponseData`

DoubleResponseData

Data being used as output to model for fitting

Syntax

```
Y = DoubleResponseData(R, TestNumber)
```

Description

This is a method of all model objects: `mbcmodel.hierarchicalresponse`, `mbcmodel.localresponse` and `mbcmodel.response`. It returns an array (`Y`) containing the response data used for fitting the model.

`R` is the response model object.

`TestNumber` is an optional input to specify the tests you want.

Examples

```
Y = DoubleResponseData(R);
y = DoubleResponseData(local, tn);
```

Version History

Introduced before R2006a

See Also

`DoubleInputData`

Export

Make command-line or Simulink export model

Syntax

```
ExportedModel = Export(model)
ExportedModel = Export(model,format)
```

Description

`ExportedModel = Export(model)` exports the model to MATLAB.

`ExportedModel = Export(model,format)` exports the model in the specified format, which can be `'MATLAB'` or `'Simulink'`.

Examples

Export Model

Export model to MATLAB.

```
M = Export(R2, 'MATLAB');
mbt_model = Export(maxTQ, 'MATLAB');
```

Input Arguments

`model` — Model object
`mbcmodel.linearmodel` object

Model object, specified as a `mbcmodel.linearmodel` object. `model` contains the response models from the node you are exporting from.

`format` — Exported model format
`'MATLAB'` | `'SIMULINK'`

Format of exported model, specified as `'MATLAB'` or `'Simulink'`.

Output Arguments

`ExportedModel` — Model exported
`xregstatsmodel` object

Model exported, specified as an `xregstatsmodel` object. You can use `ExportedModel` to evaluate the model and calculate the prediction error variance. If you convert an `mbcmodel.localresponse` object and you have not created a two-stage model (hierarchical response object), then the output is an `mbcPointByPointModel` object that you can use to evaluate the model and calculate the prediction error variance.

You can evaluate models exported to the MATLAB workspace in the same way as when you export them from the Model Browser. You can save these models as a `*.mat` file and load them into CAGE.

More About

Usage

This is a method of these model objects: `mbcmodel.hierarchicalresponse`, `mbcmodel.localresponse`, `mbcmodel.response` and `mbcmodel.model`.

Version History

Introduced before R2006a

See Also

`mbcmodel.linearmodel` | `mbcmodel.model`

GetAllTerms

List all model terms

Syntax

`Terms = M.Properties.GetAllTerms`

Description

This is a method of `mbcmodel.linearmodelproperties`.

`Terms = M.Properties.GetAllTerms` returns a list of all terms in this model. `M` is an `mbcmodel.linearmodel` object.

`Terms` is a (*numterms*-by-*nfactors*) array. The $(m,n)^{th}$ element is the power of the n^{th} factor in the m^{th} term.

Examples

The following example creates a model, and finds which terms are quadratic in the first input factor (X1):

```
model = mbcmodel.CreateModel('Polynomial', 2)

model =

   1 + 2*X1 + 8*X2 + 3*X1^2 + 6*X1*X2 + 9*X2^2 + 4*X1^3
     + 5*X1^2*X2 + 7*X1*X2^2 + 10*X2^3
   InputData: [0x2 double]
   OutputData: [0x1 double]
   Status: Not fitted
   Linked to Response: <not linked>

>>terms = model.Properties.GetAllTerms;
>>x1quadraticterms = find(terms(:,1)==2)

x1quadraticterms =

     4
     8
```

Version History

Introduced in R2007a

See Also

`GetIncludedTerms`

getAlternativeNames

List alternative algorithm names

Syntax

```
F.getAlternativeNames
AltList = getAlternativeNames(F)
```

Description

This is a method of `mbcmodel.fitalgorithm`.

`F.getAlternativeNames` or `AltList = getAlternativeNames(F)` return a cell array of alternative algorithm names. `F` is a `mbcmodel.fitalgorithm` object.

Examples

```
model = mbcmodel.CreateModel('Polynomial', 2);
F = model.FitAlgorithm;
altAlgs = F.getAlternativeNames

altAlgs =

    'Least Squares'    'Minimize PRESS'    'Forward Selection'
'Backward Selection'    'Prune'
```

Version History

Introduced in R2007a

See Also

`CreateAlgorithm` | `IsAlternative`

GetIncludedTerms

List included model terms

Syntax

`Terms = M.Properties.GetIncludedTerms`

Description

This is a method of `mbcmodel.linearmodelproperties`.

`Terms = M.Properties.GetIncludedTerms` returns a list of those terms that will be used to fit the model. `M` is an `mbcmodel.linearmodel` object.

`Terms` is a (*numincludedterms*-by-*nfactors*) array. The $(m,n)^{th}$ element is the power of the n^{th} factor in the m^{th} included term.

Examples

```
>>model = mbcmodel.CreateModel('Polynomial', 2);


>>includedterms = model.Properties.GetIncludedTerms;
>>x1quadraticterms = find(includedterms(:,1)==2)

x1quadraticterms =

     4
     8
```

Version History

Introduced in R2007a

See Also

`GetAllTerms` | `SetTermStatus`

GetTermLabel

List labels for model terms

Syntax

```
Labels = M.Properties.GetTermLabel
Labels = M.Properties.GetTermLabel( Terms )
Labels = M.Properties.GetTermLabel( Terms, 'Format',OutputFormat )
```

Description

This is a method of `mbcmodel.linearmodelproperties`, which returns a user-friendly label for one or more specified terms.

`Labels = M.Properties.GetTermLabel` lists the labels.

`Labels = M.Properties.GetTermLabel( Terms )` lists the labels with the specified terms.

`Labels = M.Properties.GetTermLabel( Terms, 'Format',OutputFormat )` lists the labels with the specified terms and format.

`M` is an `mbcmodel.linearmodel` object.

The specified terms form a row where each value gives the power of that parameter. `OutputFormat`

can be `'List'` or `'Formula'`.

Examples

```
model = mbcmodel.CreateModel('Polynomial', 2);
model.Properties.GetTermLabel([1 2; 1 0] )
```

produces `{'X1*X2^2';'X1'}` and

```
model.Properties.GetTermLabel([1 2; 1 0], 'Format', 'Formula' )
```

produces `'X1*X2^2 + X1'`.

Version History

Introduced in R2007a

See Also

`GetAllTerms` | `GetIncludedTerms`

GetTermStatus

List status of some or all model terms

Syntax

```
Status = M.Properties.GetTermStatus
Status = M.Properties.GetTermStatus(Terms)
```

Description

This is a method of `mbcmodel.linearmodelproperties`.

`Status = M.Properties.GetTermStatus` returns the status of all of the terms in this model. `Status` is a cell array of status character vectors. `M` is an `mbcmodel.linearmodel` object.

`Status = M.Properties.GetTermStatus(Terms)` returns the status of the specified terms in this model.

The stepwise status for each term can be 'Always', 'Never' or 'Step'. The status determines whether you can use the StepwiseRegression function to throw away terms in order to try to improve the predictive power of the model.

Examples

```
model = mbcmodel.CreateModel('Polynomial', 2);
```

Get status of $X2^3$ term:

```
status = model.Properties.GetTermStatus([0 3])

status =

    'Step'
```

Get status of all terms linear in X1:

```
status = model.Properties.GetTermStatus([1 0; 1 1; 1 2])

status =

    'Step'
    'Step'
    'Step'
```

Version History

Introduced in R2007a

See Also

`SetTermStatus` | `StepwiseStatus`

InBest

Boundary models selected as best

Syntax

`mbcboundary.Tree.InBest`

Description

This is a property of `mbcboundary.Tree` and `mbcboundary.TwoStageTree`.

`mbcboundary.Tree.InBest` Specify a logical array indicating which boundary models to select as best.

You can combine models into a single boundary model for the boundary tree. The logical array specifies which models to include in the best boundary model. The `BestModel` property gives the best boundary model for the boundary tree.

Including boundary models `InBest` corresponds to combining boundary models in best in the Boundary Editor. For further information, see“Combining Best Boundary Models” in the Model Browser documentation.

See Also

`BestModel`

Inputs property

Inputs for test plan, model, boundary model, design, or constraint

Syntax

`testplan.Inputs`

`model.Inputs`

`design.Inputs`

`boundary.Inputs`

Description

This is a property of `mbcmodel.testplan`, `mbcmodel.model`, `mbcdoe.design`, `mbcdoe.designconstraint`, and boundary model object `mbcboundary.AbstractBoundary` and all its subclasses.

For `mbcmodel.testplan`, this property returns a cell array of `mbcmodel.modelinput` objects (one array for each stage). You cannot change the number of stages after creation of the test plan.

For `mbcmodel.model` and `mbcboundary` objects, this property returns an `mbcmodel.modelinput` object. You cannot edit this object when it is attached to a response. You cannot change number of inputs after creation.

In both cases, verification of valid variable names and symbols occurs before assigning inputs to model at the command line. Names and Symbols must be unique.

Boundary model inputs use an array of `mbcmodel.modelinput` objects. You set the number of boundary model inputs when you create the boundary model. You can change the name, symbol, and range of the inputs.

For `mbcdoe.design`, `D.Inputs = NewInputs` updates the inputs. You cannot change the number of design inputs. Many designs have `Limits` properties in addition to model input ranges. These properties allow you to restrict the range of the design without changing the model or losing points via a constraint.

See Also

`CreateTestplan` | `modelinput` | `mbcdoe.design`

InputSignalNames property

Names of signals in data that are being modeled

Syntax

```
inputs = A.InputSignalNames
```

Description

This is a property of `mbcmodel.testplan` and the modeling objects `mbcmodel.hierarchicalresponse`, `mbcmodel.localresponse` and `mbcmodel.response`.

A can be a test plan (T) or model (L, R, HR) object.

Examples

```
inputs =  T.'InputSignalNames;
```

```
InputFactors = thisRF.InputSignalNames';
```

See Also

`mbcmodel.data`

IsAlternative

Test alternative fit algorithm

Syntax

```
OK = IsAlternative(F1, F2)
```

Description

This is a method of `mbcmodel.fitalgorithm`.

`OK = IsAlternative(F1, F2)` tests whether F is an alternative `mbcmodel.fitalgorithm` for F1.

Version History

Introduced in R2007a

See Also

`CreateAlgorithm` | `getAlternativeNames`

Level

Level in test plan of response

Syntax

```
level = R.Level
```

Description

This is a property for all model objects: `mbcmodel.hierarchicalresponse`, `mbcmodel.localresponse` and `mbcmodel.response`.

`R` is the response for which you want the level.

The level is usually 0 for hierarchical models, usually 1 for local models, and usually 2 or 1 for response models. See “Understanding Model Structure for Scripting” for an explanation of what `Level` indicates about a response.

Examples

```
level = R.Level;
```

See Also

`mbcmodel.testplan`

LocalBoundaries

Array of local boundary models for each operating point

Syntax

```
LocalBoundaries(B)
```

Description

This is a property of `mbcboundary.PointByPoint`.

`LocalBoundaries(B)` returns a cell array of local boundary models for each operating point (read only).

Version History

Introduced in R2009b

LocalModel Properties

Edit local model properties

Syntax

```
Props = localmodel.Properties
```

Description

This is a property of the `mbcmodel.localmodel` object, which is a subclass of mbcmodel.model.

See “Understanding Model Structure for Scripting” for an explanation of the relationship between the different response types.

Every local model object has an mbcmodel.modelproperties object (within the Properties property). In this object, each local model type has specific properties, as described in the following tables.

Local Polynomial Properties

Property	Description
Order	Polynomial order (vector int: {[0,Inf],2})
InteractionOrder	Maximum order of interaction terms (int: [0,Inf])
TransformInputRange	Transform inputs (Boolean)
ParameterNames	List of parameter names (read-only)
StepwiseStatus	Stepwise status `{'Always','Never','Step'}` (cell)
Transform	Transform function (char) or empty (`''`)
CovarianceModel	Covariance Model (enum: `{'None','Power', 'Exponential','Mixed'}`)
CorrelationModel	Correlation Model (enum: `{'None','MA(1)','AR(1)', 'AR(2)'}`)

Local Hybrid Spline Properties

Property	Description
Order	Spline and polynomial order (vector int: `{[0,3],2}`)
SplineVariable	Spline variable
SplineInteraction	Order of interaction between spline and polynomial (int: `[0,3]`)
Knots: Position of knots (vector real)	ParameterNames: List of parameter names (read-only)
StepwiseStatus	Stepwise status `{'Always','Never','Step'}` (cell)
Transform	Transform function (char) or empty (`''`)
CovarianceModel	Covariance Model (enum: `{'None','Power', 'Exponential','Mixed'}`)
CorrelationModel	Correlation Model (enum: `{'None','MA(1)','AR(1)', 'AR(2)'}`)

Local Polynomial Spline Properties

Property	Description
HighOrder	Polynomial order above knot (int: `[2,Inf]`)
LowOrder	Polynomial order below knot (int: `[2,Inf]`)
Transform	Transform function (char) or empty (`''`)
CovarianceModel	Covariance Model (enum: `{'None','Power', 'Exponential','Mixed'}`)
CorrelationModel	Correlation Model (enum: `{'None','MA(1)','AR(1)', 'AR(2)'}`)
DatumType	Datum Type (enum: `{'None','Maximum','Minimum', 'Linked'}`)

Local Polynomial With Datum Properties

Property	Description
Order	Polynomial order (int: `[0,Inf]`)
Transform	Transform function (char) or empty (`''`)
CovarianceModel	Covariance Model (enum: `{'None','Power', 'Exponential','Mixed'}`)
CorrelationModel	Correlation Model (enum: `{'None','MA(1)','AR(1)', 'AR(2)'}`)
DatumType	Datum Type (enum: `{'None','Maximum','Minimum', 'Linked'}`)

Local Free Knot Spline Properties

Property	Description
Order	Spline Order (int: `[0,Inf]`)
NumKnots	Number of knots (int: `'Positive'`)
Transform	Transform function (char) or empty (`''`)
CovarianceModel	Covariance Model (enum: `{'None','Power', 'Exponential','Mixed'}`)
CorrelationModel	Correlation Model (enum: `{'None','MA(1)','AR(1)', 'AR(2)'}`)

Local Truncated Power Series Properties

Property	Description
Order	Polynomial order (int: `'Positive'`)
NumKnots	Number of knots (int: `'Positive'`)
Transform	Transform function (char) or empty (`''`)
CovarianceModel	Covariance Model (enum: `{'None','Power', 'Exponential','Mixed'}`)
CorrelationModel	Correlation Model (enum: `{'None','MA(1)','AR(1)', 'AR(2)'}`)

Local Growth Properties

Property	Description
Model	Growth model (enum: {'expgrowth','gomp', 'logistic','logistic4', 'mmf','richards', 'weibul'})
AlternativeModels	List of growth models (read-only)
Transform	Transform function (char) or empty ('')
TransformBothSides	Transform both sides (Boolean)
CovarianceModel	Covariance Model (enum: {'None','Power', 'Exponential','Mixed'})
CorrelationModel	Correlation Model (enum: {'None','MA(1)','AR(1)', 'AR(2)'})

Local User-Defined Properties

Property	Description
Model	Name of user-defined model (enum: {'exponential'})
AlternativeModels	List of registered user-defined models (read-only)
Transform	Transform function (char) or empty ('')
TransformBothSides	Transform both sides (Boolean)
CovarianceModel	Covariance Model (enum: {'None','Power', 'Exponential','Mixed'})
CorrelationModel	Correlation Model (enum: {'None','MA(1)','AR(1)', 'AR(2)'})

Local Transient Properties

Property	Description
Model	Name of transient model (enum: `{'fuelPuddle'}`)
AlternativeModels	List of registered transient models (read-only)
Transform	Transform function (char) or empty (`''`)
TransformBothSides	Transform both sides (Boolean)
CovarianceModel	Covariance Model (enum: `{'None','Power', 'Exponential','Mixed'}`)
CorrelationModel	Correlation Model (enum: `{'None','MA(1)','AR(1)', 'AR(2)'}`)

Local Multiple Models Properties

Property	Description
ModelCandidates	List of candidate models (cell)
SelectionStatistic	Selection statistic for automatic model selection (char). See below for input names and descriptions. The list of valid statistics is the summary statistics in common with all model candidates (e.g., if an interpolating RBF is one of the candidates, only RMSE will be available).
AutomaticInputRanges	Use data range as model input ranges (Boolean)
Transform	Transform function (char) or empty (`''`)

Model Type	List of `SelectionStatistic` Inputs
Polynomial,Hybrid Spline, RBF, Hybrid RBF	`'PRESS RMSE','RMSE','GCV','Weighted PRESS','-2logL','AIC','AICc', 'BIC','R^2','R^2 adj', 'PRESS R^2','DW','Cp','cond(J)'`
Neural Network	`'RMSE','R^2','R^2 adj','-2logL','AIC','AICc','BIC'`
Free Knot Spline	`'PRESS RMSE','RMSE','GCV','Weighted PRESS','-2logL','AIC','AICc', 'BIC','R^2','R^2 adj', 'PRESS R^2','DW','Cp'`
Interpolating RBF	`'RMSE'`

`SelectionStatistic` Input Argument	Description	
`'PRESS RMSE'`	Predicted Standard Error	`'sqrt(PRESS/N)'`
`'RMSE'`	Root Mean Square Error	`'sqrt(SSE/(N-p))'`

SelectionStatistic Input Argument	Description	
'GCV'	Generalized Cross-validation Variance	'N*SSE/(N-p)^2'
'Weighted PRESS'	Weighted Predicted Standard Error	'sqrt(PRESS/(N-p-1))'
'-2logL'	-2 * log likelihood	'N*log(SSE/N)'
'AIC'	Akaike Information Criteria	'-2logL + 2*(p+1)'
'AICc'	Small Sample Akaike Information Criteria	'-2logL + 2(p+1)*N/(N-p)'
'BIC'	Bayesian Information Criteria	'-2logL + 2*log(N)*(p+1)'
'R^2'	R^2	'1 - SSE/SST'
'R^2 adj'	Adjusted R^2	'1 - SSE/SST*(N-1)/(N-p)'
'PRESS R^2'	PRESS R^2	'1 - PRESS/SST'
'DW'	Durbin-Watson Statistic	'sum((e_i-e_{i+1})^2)/sum(e_i^2) '
'Cp'	Mallow's Statistic	'SSE/(SSEmax/(N-pmax)) - N + 2*p'
'cond(J)'	Condition of Regression Matrix	'cond(J)'

Local Average Fit Properties

Property	Description
Model	[1x1 mbcmodel.linearmodel]
Transform	Transform function (char) or empty ('')

Examples

To create a local model object, create a model specifying any model Type that begins with the word “local”, e.g.,

```
L = mbcmodel.CreateModel('Local Polynomial',2);
```

To show properties, at the command line enter:

```
P = L.Properties

P =
Local Polynomial Properties
                  Order: [3 3]
       InteractionOrder: 3
    TransformInputRange: 1
         ParameterNames: {10x1 cell}
         StepwiseStatus: {10x1 cell}
              Transform: ''
        CovarianceModel: 'None'
       CorrelationModel: 'None'
```

To set the Order property to a quadratic, enter:

```
>> P.Order = [2,2]

P =
Local Polynomial Properties
                  Order: [2 2]
       InteractionOrder: 2
    TransformInputRange: 1
         ParameterNames: {6x1 cell}
         StepwiseStatus: {6x1 cell}
              Transform: ''
        CovarianceModel: 'None'
       CorrelationModel: 'None'
```

To update the local model, the properties object must be reassigned to the model as follows:

```
>> L.Properties = P

L =

   1 + 2*X1 + 5*X2 + 3*X1^2 + 4*X1*X2 + 6*X2^2
   InputData: [0x2 double]
   OutputData: [0x1 double]
   Status: Being Edited
   Linked to Response: not linked
```

See Also

`CreateModel` | `mbcmodel.model` | `mbcmodel.modelproperties` | `ResponseFeatures(Local Model)`

LocalResponses

Array of local responses for response

Syntax

```
local = response.LocalResponses
```

Description

This is a property of the `mbcmodel.hierarchicalresponse` object.

It returns the local model response objects that belong to the hierarchical response `R`.

See "Understanding Model Structure for Scripting" for an explanation of the relationship between the different response types.

Examples

```
local = response.LocalResponses;
```

MakeHierarchicalResponse

Build two-stage model from response feature models

Syntax

```
OK = MakeHierarchicalResponse(L,MLE)
```

Description

This method of `mbcmodel.localresponse` builds a two-stage model from the response feature models and optionally runs MLE (Maximum Likelihood Estimation). If there are more response features than the number of parameters in the local model, the subset of response features that leads to the best hierarchical response is chosen. The best hierarchical response is chosen using `PRESS RMSE` (root mean square prediction error — see “PRESS statistic”) if all the response feature models are linear. Otherwise, the best hierarchical response is chosen using `Two-stage RMSE`.

This performs a similar function to `ChooseAsBest` for response models. You can call `MakeHierarchicalResponse` directly or indirectly by calling `CreateAlternativeModels` for a local model. If you call `CreateAlternativeModels` for a local model, `MakeHierarchicalResponse` is called automatically.

If the local and response models are not ready to calculate a two-stage model, an error is generated. This situation can occur if you have created alternative models and not chosen the best. A sufficient number of response features models to calculate the two-stage model must be selected.

`L` is the local model object.

`MLE` can be `true` or `false`. If `true`, MLE will be calculated.

Examples

```
OK = MakeHierarchicalResponse(L, true)
```

Version History

Introduced before R2006a

See Also

`ChooseAsBest`

MatchInputs

Match design constraint inputs

Syntax

```
C = MatchInputs(C,DesignInputs)
C = MatchInputs(C,DesignInputs,mapping)
```

Description

`MatchInputs` is a method of `mbcdoe.designconstraint`. Use it to match inputs for constraints from different sources.

`C = MatchInputs(C,DesignInputs)` matches `DesignInputs` and inputs in `C`.

`C = MatchInputs(C,DesignInputs,mapping)` matches inputs where `mapping` defines the relationship between the inputs in `C`, and `DesignInputs`.

Examples

A design constraint does not have required inputs EXH_RET and INT_ADV. Use `MatchInputs` to match the constraint inputs to the design inputs as follows:

```
c = BoundaryModel(p.Testplans,'all')
c =
Star(N-3.5e+003,L-0.54)

originalInputs=c.Inputs
originalInputs =
   SPEED (N) [rpm] [500,6000]
   LOAD (L) [%] [0.06,0.95]

designInputs = Design.Inputs
designInputs =
   SPEED (N) [rpm] [500,6000]
   LOAD (L) [%] [0.06,0.95]
   EXH_RET (ECP) [DegCrank] [-5,50]
   INT_ADV (ICP) [DegCrank] [-5,50]

c2=MatchInputs(c,designInputs,[1 2]);
newInputs=c2.Inputs
newInputs =
   SPEED (N) [rpm] [500,6000]
   LOAD (L) [%] [0.06,0.95]
   EXH_RET (ECP) [DegCrank] [-5,50]
   INT_ADV (ICP) [DegCrank] [-5,50]
```

Version History

Introduced in R2008a

See Also

`CreateConstraint`

mbcboundary.AbstractBoundary

Base boundary model class

Description

Do not use this class directly because the `mbcboundary.AbstractBoundary` class is the base class for all boundary model classes in the Model-Based Calibration Toolbox software.

The following subclasses inherit all the properties and methods of the `mbcboundary.AbstractBoundary` class:

- `mbcboundary.Model`
- `mbcboundary.Boolean`
- `mbcboundary.PointByPoint`
- `mbcboundary.TwoStage`

Properties of `mbcboundary.AbstractBoundary`

Inputs	Inputs for test plan, model, boundary model, design, or constraint
Name	Name of object
NumInputs	Number of model, boundary model, or design object inputs

Methods of `mbcboundary.AbstractBoundary`

Evaluate	Evaluate model, boundary model, or design constraint

Version History
Introduced in R2009b

mbcboundary.Boolean

Boolean boundary model class

Description

You can create Boolean boundary models, which are useful as design constraints, in two ways. You can either use logical operators (&,|,~) on other boundary models, or you can include more than one boundary model in the best boundary model for a boundary tree. If you combine boundary models using logical operators you cannot add the resulting Boolean boundary model to a boundary tree.

When working in projects, you can combine boundary models by including them `InBest`. For example, you can use subsets of input factors to build boundary models (see `ActiveFactors`). You can then combine the subset boundary models for the most accurate boundary. This approach can provide more effective results than including all inputs. If the `BestModel` property of the boundary tree includes more than one boundary model, then the boundary model is an `mbcboundary.Boolean` object.

The mbcboundary.Boolean class is a subclass of `mbcboundary.AbstractBoundary`. The `mbcboundary.AbstractBoundary` class is the base class for all boundary model classes in the Model-Based Calibration Toolbox software.

Properties of `mbcboundary.Boolean`

Inputs	Inputs for test plan, model, boundary model, design, or constraint
Name	Name of object
NumInputs	Number of model, boundary model, or design object inputs

Methods of `mbcboundary.Boolean`

Evaluate	Evaluate model, boundary model, or design constraint

Version History

Introduced in R2009b

mbcboundary.Model

Boundary model class

Description

The `mbcboundary.Model` class represents the basic boundary model types in the Model-Based Calibration Toolbox software.

You can fit boundary models in `mbcmodel` projects using the boundary tree class `mbcboundary.Tree`, or you can fit boundary models directly to data.

You can combine boundary models using the logical operators &, | and ~.

The mbcboundary.Model class is a subclass of `mbcboundary.AbstractBoundary`. The `mbcboundary.AbstractBoundary` class is the base class for all boundary model classes in the Model-Based Calibration Toolbox software.

Properties of `mbcboundary.Model`

ActiveInputs	Active boundary model inputs
Inputs	Inputs for test plan, model, boundary model, design, or constraint
Name	Name of object
NumInputs	Number of model, boundary model, or design object inputs

Methods of `mbcboundary.Model`

Evaluate	Evaluate model, boundary model, or design constraint

Version History

Introduced in R2009b

mbcboundary.PointByPoint

Point-by-point boundary model class

Description

You can only create and fit point-by-point boundary models in the local boundary tree in two ways. You can use either a two-stage test plan or an existing boundary of type, either `'Point-by-point'` or `'Two-stage'`. You cannot create or fit these types of boundary models outside a project. Fit them by adding to the boundary model to the boundary tree.

A separate boundary model is fitted to each operating point. Point-by-point boundary models are only valid at the observed operating points.

The mbcboundary.PointByPoint class is a subclass of `mbcboundary.AbstractBoundary`. The `mbcboundary.AbstractBoundary` class is the base class for all boundary model classes in the Model-Based Calibration Toolbox software.

Properties of `mbcboundary.PointByPoint`

Inputs	Inputs for test plan, model, boundary model, design, or constraint
LocalBoundaries	Array of local boundary models for each operating point
Name	Name of object
NumInputs	Number of model, boundary model, or design object inputs
OperatingPoints	Model operating point sites

Methods of `mbcboundary.PointByPoint`

Evaluate	Evaluate model, boundary model, or design constraint

Version History

Introduced in R2009b

mbcboundary.Tree

Boundary tree class

Description

The boundary `Tree` is a container for all the boundary models you create. You access the boundary tree from the Boundary property of `mbcmodel.testplan`. The root of the boundary tree for a one-stage test plan is an `mbcboundary.Tree` object. The root of the boundary tree for a two-stage test plan is a `mbcboundary.TwoStageTree`, and this object has `mbcboundary.Tree` objects in its `Local`, `Global` and `Response` properties.

Use the `Models` and `BestModel` properties of the boundary `Tree` to access your boundary models.

The mbcboundary.Tree class is a subclass of `mbcboundary.AbstractBoundary`. The `mbcboundary.AbstractBoundary` class is the base class for all boundary model classes in the Model-Based Calibration Toolbox software.

Properties of `mbcboundary.Tree`

BestModel	Combined best boundary models
Data	Array of data objects in project, boundary tree, or test plan
InBest	Boundary models selected as best
Models	Array of boundary models
TestPlan	Test plan containing boundary tree

Methods of `mbcboundary.Tree`

Add	Add boundary model to tree and fit to test plan data
Remove	Remove test plan, model, or boundary model
Update	Update boundary model in tree and fit to test plan data

Version History

Introduced in R2009b

mbcPointByPointModel

Class for evaluating point-by-point models and calculating PEV

Description

If you convert an `mbcmodel.localresponse` object using Export and you have not created a two-stage model (hierarchical response object), then the output is an `mbcPointByPointModel` object. Point-by-point models are created from a collection of local models for different operating points. `mbcPointByPointModel` objects share all the same methods as `xregstatsmodel` except `dferror`. See `xregstatsmodel`.

Version History

Introduced in R2010a

Model Object

Model object within response object

Syntax

```
M = response.Model
```

Description

This is a property of all mbcmodel.response objects.

Each response contains a model object (`mbcmodel.model`) that can be extracted and manipulated independently of the project.

Extract a model object from any response object, and then:

- Fit to new data (`fit`).
- Change model type, properties, and fit algorithm settings (`ModelSetup`; `mbcmodel.modelproperties`, `CreateAlgorithm`).
- Create a copy of the model with the same inputs (`CreateModel`).
- Include and exclude terms to improve the model (`StepwiseRegression`).
- Examine coefficient values, predicted values, and regression matrices (`ParameterStatistics`; `PredictedValue`; `Jacobian`).
- If you change the model you need to use `UpdateResponse` to replace the new model back into the response object in the project. When you use `UpdateResponse` the new model is fitted to the response data.

Examples

```
M = response.Model;
```

ModelForTest

Model for specified test

Syntax

`model = ModelForTest(L,TestNo);`

Description

This is a method of `mbcmodel.localresponse`.

`model = ModelForTest(L,TestNo);` gets the model for test `TestNo`.

Examples

To get the model for test 22, enter:

```
model = ModelForTest(L,22);
```

Version History

Introduced in R2007b

modelinput

Create modelinput object

Syntax

```
Inputs = mbcmodel.modelinput('Property1',value1,'Property2',value2,...);
Inputs = mbcmodel.modelinput(NUMINPUTS);
Inputs = mbcmodel.modelinput(INPUTCELLARRAY);
```

Description

This is the constructor for the `mbcmodel.modelinput` object.

`Inputs = mbcmodel.modelinput('Property1',value1,'Property2',value2,...);` creates the `mbcmodel.modelinput` object.

You can set the properties shown in the following table.

Property	Description
Range	[min,max]
NonlinearTransform	`{'','1./x','sqrt(x)', 'log10(x)','x.^2', 'log(x)'}`
Name	Character vector. Signal name from dataset. Inputs for a test plan must be set before selecting data.
Symbol	Character vector. Short name for plot labels and for use in CAGE.
Units	Character vector. Units are overwritten from the dataset units when a data is selected.

Specify "property, value" pairs as follows:

```
Inputs = mbcmodel.modelinput('Symbol',{'A','B'},...
 'Range',{[0 100],[-20 20]});
```

Scalar expansion of properties is supported, e.g.,

```
Inputs = mbcmodel.modelinput('Symbol',{'A','B'},...
 'Range',[0 100]);
```

`Inputs = mbcmodel.modelinput(NUMINPUTS);` creates the `mbcmodel.modelinput` object with the specified number inputs.

`NUMINPUTS` is the number of inputs. Symbols are automatically set to `'X1', 'X2',...,'Xn'`. The default range is [-1,1]. For example:

```
Inputs = mbcmodel.modelinput(2);
```

`Inputs = mbcmodel.modelinput(INPUTCELLARRAY);` creates the `mbcmodel.modelinput` object with `INPUTCELLARRAY` inputs.

`INPUTCELLARRAY` is a cell array with one row per input and 5 columns to specify factor names, symbols, ranges and nonlinear transforms as follows.

The columns of `INPUTCELLARRAY` must be:

1. Factor symbol (character vector)
2. Minimum (double)
3. Maximum (double)
4. Transform (character vector) — empty for none
5. Signal name

These columns are the same as the columns in the Model Factor Setup dialog box, which can be launched from the test plan in the model browser.

Examples

To create a modelinput object with 2 inputs, enter:

```
Inputs = mbcmodel.modelinput(2);
```

To create a modelinput object and define symbols and ranges, enter:

```
Inputs = mbcmodel.modelinput('Symbol',{'A','B'},...
 'Range',{[0 100],[-20 20]});

Inputs = mbcmodel.modelinput('Symbol',{'A','B'},...
 'Range',[0 100]);
```

To create a modelinput object and define inputs with a cell array, enter:

```
Inputs = mbcmodel.modelinput( {...
 'N',   800, 5000, '', 'ENGSPEED'
 'L',   0.1,    1, '', 'LOAD'
'EXH',  -5,   50, '', 'EXHCAM'
'INT',  -5,   50, '', 'INTCAM'} );
```

Version History

Introduced in R2007b

See Also

`CreateModel` | `CreateTestplan`

Models

Array of boundary models

Syntax

```
Models(T)
```

Description

This is a property of `mbcboundary.Tree`.

`Models(T)` returns a cell array of boundary models (read only).

Version History

Introduced in R2009b

MultipleVIF

Multiple VIF matrix for linear model parameters

Syntax

`vif = MultipleVIF(linearmodel)`

Description

`vif = MultipleVIF(linearmodel)` calculates the multiple Variance Inflation Factor (VIF) matrix for the linear model parameters.

Examples

Calculate VIF of Linear Model

Calculate multiple VIF of knot model.

```
VIF = MultipleVIF(knot_model)
```

Input Arguments

`linearmodel` — Model object
`mbcmodel.linearmodel` object

Model object, specified as a `mbcmodel.linearmodel` object.

Output Arguments

`vif` — Multiple variance inflation factor
matrix

Multiple variance inflation factor, returned as a matrix.

Version History

Introduced in R2007a

See Also

`ParameterStatistics`

Name

Name of object

Syntax

```
name = A.Name
```

Description

This is a property of project, data, test plan, input, model, fitalgorithm, design, design constraint, and boundary model objects.

`A` can be any test plan (`T`), data (`D`), project (`P`) model (`L`, `R`, `HR`), fitalgorithm (`F`), design (`D`), design constraint (`C`) or boundary model (`B`) object.

You can change the names of these objects as follows:

```
A.Name =  newName
```

For response (output or Y data) signal names, see `ResponseSignalName`.

For `mbcmodel.model.Name`, the Name property refers to the model output name. The toolbox sets this property to the data signal name when the response is created or if you assign a model to a response. You cannot set this property when a response is attached to the model.

For model parameter names, see `Names`.

For testplan and response object input names, see `InputSignalNames`, and for data objects, see `mbcmodel.data`.

Names of boundary models are read only and provide a description of the boundary model type and active inputs.

Examples

```
ResponseFeatureName = thisRF.Name;
```

See Also

`Names` | `InputSignalNames` | `mbcmodel.data` | `mbcdoe.design` | `ResponseSignalName`

Names

Model parameter names

Syntax

```
N = params.Names
```

Description

This is a property of `mbcmodel.modelparameters`. It returns the names of all the parameters in the model. These are read-only.

Examples

```
N = paramsknot.Names
N =
'1'
'N'
'N^2'
'N*L'
'N*A'
'L'
'L^2'
'L*A'
'A'
'A^2';
```

See Also

`NumberOfParameters` | `Values` | `Name`

NumInputs

Number of model, boundary model, or design object inputs

Syntax

```
N = model.NumInputs
```

Description

This is a property of

- `mbcmodel.model` and `mbcmodel.modelproperties`
- The design objects `mbcdoe.design`, `mbcdoe.generator`, `mbcdoe.candidateset`, and `mbcdoe.designconstraint`
- The boundary model object `mbcboundary.AbstractBoundary` and all its subclasses: `mbcboundary.Model`, `mbcboundary.Boolean`, `mbcboundary.PointByPoint` and `mbcboundary.TwoStage`. You set the number of boundary model inputs when you create the boundary model.

It returns the number of inputs to the model, boundary model, or design object.

Examples

```
N = knot.NumInputs;
```

`mbcdoe.design`

NumberOfParameters

Number of included model parameters

Syntax

```
N = knotparams.NumberOfParameters
```

Description

This is a read-only property of `mbcmodel.linearmodelparameters`, for linear models only.

The number returned is the number of parameters currently in the model (you can remove some parameters by using StepwiseRegression). To see which parameters are currently in the model, use StepwiseSelection. Only parameters listed as '`in`' are currently included.

To see the total possible number of parameters in a linear model, use SizeOfParameterSet.

Use Names and Values to get the parameter names and values.

Examples

```
N = knotparams.NumberOfParameters;
```

See Also

`SizeOfParameterSet` | `StepwiseSelection` | `StepwiseRegression` | `Names` | `Values`

NumberOfTests property

Total number of tests being used in model

Syntax

```
numtests = A.NumberOfTests
```

Description

This is a property of all model objects: `mbcmodel.hierarchicalresponse`, `mbcmodel.localresponse` and `mbcmodel.response`, and data objects `mbcmodel.data`. 'A' can be any model or data object.

Examples

```
numTests = TQ_response.NumberOfTests;
```

See Also

`DefineTestGroups` | `mbcmodel.data`

OperatingPoints

Model operating point sites

Syntax

`OperatingPoints(B)`

Description

This is a property of `mbcboundary.PointByPoint`.

`OperatingPoints(B)` returns the operating point sites for models (read only).

Version History

Introduced in R2009b

OutlierIndices

Indices of `DoubleInputData` marked as outliers

Syntax

```
indices = OutlierIndices(R)
```

Description

This is a method of all model objects: `mbcmodel.hierarchicalresponse`, `mbcmodel.localresponse` and `mbcmodel.response`.

Examples

```
ind = OutlierIndices(R);
bad = OutlierIndices(thisRF);
```

Version History

Introduced before R2006a

See Also

`DoubleInputData`

OutlierIndicesForTest

Indices marked as outliers for test

Syntax

```
indices = OutlierIndicesForTest(R, TestNumber)
```

Description

This is a method of the local model object, `mbcmodel.localresponse`.

This shows the current records discarded as outliers.

You can use ':' to use all tests.

Examples

```
ind = OutlierIndicesForTest(R, ':');
bad = OutlierIndicesForTest(local, tn);
```

Version History

Introduced before R2006a

See Also

`OutlierIndices`

Parameters

Model parameters

Syntax

```
P = model.Parameters
```

Description

This is a property of `mbcmodel.model`, that contains an object `mbcmodel.modelparameters`. This object contains a number of read-only parameters that describe the model.

All models have these properties:

- `SizeOfParameterSet`
- `Names`
- `Values`

Linear models also have these properties:

- `StepwiseStatus`
- `NumberOfParameters`
- `StepwiseSelection`

Examples

```
P = model.Parameters;
```

See Also

`SizeOfParameterSet` | `Names` | `Values` | `StepwiseStatus` | `NumberOfParameters` | `StepwiseSelection`

PartialVIF

Partial VIF matrix for linear model parameters

Syntax

`vif = PartialVIF(linearmodel)`

Description

`vif = PartialVIF(linearmodel)` calculates the partial Variance Inflation Factor (VIF) matrix for the linear model parameters.

Examples

Calculate Partial VIF

Calculate partial VIF of knot model.

```
VIF = PartialVIF(knot_model)
```

Input Arguments

`linearmodel` — Model object
`mbcmodel.linearmodel` object

Model object, specified as a `mbcmodel.linearmodel` object.

Output Arguments

`vif` — Partial variance inflation factor
matrix

Partial variance inflation factor, returned as a matrix.

Version History

Introduced in R2007a

See Also

`ParameterStatistics`

ParameterStatistics

Calculate parameter statistics for linear model

Syntax

```
values = ParameterStatistics(linearmodel)
values = ParameterStatistics(linearmodel,statType)
```

Description

`values = ParameterStatistics(linearmodel)` calculates the parameter statistics for `linearmodel`.

`values = ParameterStatistics(linearmodel,statType)` calculates the parameter statistics using the specified `statType`.

Examples

Calculate Statistics Model

Calculate parameter statistics of knot model.

```
values = ParameterStatistics(knot)
```

```
values =
          Alias: [7x3 double]
     Covariance: [7x7 double]
    Correlation: [7x7 double]
      VIFsingle: [5x5 double]
    VIFmultiple: [7x1 double]
     VIFpartial: [5x5 double]
       Stepwise: [10x4 double]

values.Stepwise
ans =
  1.0e+003 *
    0.0190    0.0079    0.0210       NaN
    0.0000    0.0000    0.0210    1.9801
    0.0000    0.0000    0.0200    0.2984
   -0.0000    0.0000    0.0200    0.2768
    0.0000    0.0000    0.0200    0.2890
   -0.0526    0.0367    0.0210    0.2679
    0.0911    0.0279    0.0210    0.3837
   -0.0041    0.0024    0.0210    0.2728
   -0.0178    0.0095    0.0200    0.2460
```

```
    0.0001    0.0000    0.0210    0.3246
```

Input Arguments

`linearmodel` — Model object
`mbcmodel.linearmodel` object

Model object, specified as a `mbcmodel.linearmodel` object.

`statType` — Type of parameter statistic
character vector | cell array

Type of parameter statistics, specified as a character vector specifying a particular statistic or a cell array of character vectors specifying a number of statistics to output. The valid data types are

- '`Alias`'
- '`Covariance`'
- '`Correlation`'
- '`VIFsingle`'
- '`VIFmultiple`'
- '`VIFpartial`'
- '`Stepwise`'

All of these types (except '`Stepwise`') appear in the Design Evaluation tool. See documentation for more details of these matrices.

The `Stepwise` field contains the values found in the Stepwise table. In this array (and in the Stepwise GUI) you can see for each parameter in the model: the value of the coefficient, the standard error of the coefficient, the `t` value and Next PRESS (the value of PRESS if the status of this term is changed at the next iteration). See the documentation for the Stepwise table. You can also see these Stepwise values when you use `StepwiseRegression`.

Output Arguments

`values` — Parameter statistics values
array

Parameter statistics values, returned as a structure.

If `statType` is a character vector, then `values` is an array of doubles. If `statType` is a cell array of character vectors, then `values` is a cell array of array of doubles.

Version History

Introduced before R2006a

See Also

`StepwiseRegression`

PEVForTest

Local model predicted error variance for test

Syntax

```
pev = PEVforTest(L, TestNumber, X)
```

Description

This is a method of the local model object, `mbcmodel.localresponse`.

`L` is the local model object.

`TestNumber` is the test for which you want to evaluate the model PEV.

`X` is the array of inputs where you want to evaluate the PEV of the model.

Examples

```
pev = PEVforTest(L, TestNumber, X);
```

Version History

Introduced before R2006a

See Also

`pev`

PredictedValueForTest

Predicted local model response for test

Syntax

```
y = PredictedValueForTest(L, TestNumber, X)
```

Description

This is a method of the local model object, `mbcmodel.localresponse`.

`L` is a local model object.

`TestNumber` is the test for which you want to evaluate the model.

`X` is the array of inputs where you want to evaluate the output of the model.

Examples

```
y = PredictedValueForTest(L, TestNumber, X);
```

Version History

Introduced before R2006a

See Also

`PredictedValue`

Properties (for candidate sets)

View and edit candidate set properties

Syntax

```
properties(CS)
CS.PropertyName = NewValue
```

Description

“`Properties`” is a method of `mbcdoe.candidateset`, which returns a list of properties.

`properties(CS)` lists the candidate set properties.

`CS.PropertyName = NewValue` sets the candidate set property.

The candidate set `Type` determines which properties you can set.

The following table lists the properties available for each candidate set type.

Candidate Set Properties (for Optimal Designs)

Candidate Set Type	Property	Description
All built-in: Grid/ Lattice, Grid, Lattice, Stratified Lattice, Sobol, Halton	NumberOfPoints (read-only for Grid and Grid/Lattice)	Number of points (int: [0,Inf])
	Limits	Design Limits
Grid	Levels	Selection criteria for best LHS design (cell)
	NumberPerLevel	Symmetric design (vector int: {[-Inf,Inf], NumInputs})
Lattice	Generators	Prime number generators for lattice (vector int: {[0,Inf], NumInputs})
Stratified Lattice	StratifyLevels	Number of levels for each factors (vector int: {[0,Inf], NumInputs})
Sobol Sequence	Scramble	Scramble method (enum: {‘none’, ’MatousekAffineOwen’}
	SkipMode	Skip mode options (enum: {'None','2^k','Custom'})
	Skip	Skip size (int: [0,Inf])
Halton Sequence	Scramble	Scrambling method for sequence (enum: {'None','RR2'})
	PrimeLeap	Leap sequence points using prime number (boolean)
	SkipZero	Skip zero point (boolean)
User-defined	NumberOfPoints	User-defined points (read-only)
	Points	User-defined points

Examples

You can use property value pairs to specify candidate set properties as part of the `CreateCandidateSet` command, or you can set properties individually.

To create a candidate set with type grid and specified grid levels:

```
CandidateSet = augmentedDesign.CreateCandidateSet...
( 'Type', 'Grid' );
CandidateSet.NumberOfLevels = [21 21 21 21];
```

Version History

Introduced in R2008a

See Also

`CreateCandidateSet`

Properties (for design constraints)

View and edit design constraint properties

Syntax

```
properties(C)
C.PropertyName = NewValue
```

Description

“`Properties`” is a method of `mbcdoe.designconstraint`, which returns a list of properties.

`properties(C)` lists the constraint properties.

`C.PropertyName = NewValue` sets the constraint property.

The constraint `Type` determines which properties you can set. For more information, see the following table or `Type (for design constraints)`.

The following table lists the properties available for each constraint type.

Constraint Properties

Constraint Type	Property	Description
Linear design constraint: 1*Input1 + 1* Input2 + 1* Input3 <= 0	A	Matrix for linear constraint (matrix: [1,NumInputs])
	b	Bound for linear constraint (double)
Ellipsoid design constraint: Ellipsoid at (Input1=0, Input2=0, Input3=0)	CenterPoint	Center of ellipse (vector: NumInputs)
	Matrix	Ellipsoid form matrix (positive semi-definite) (matrix: [NumInputs, NumInputs])
1D Table design constraint: InputY(InputX) <= InputY max	Table	Table constraint (vector)
	Breakpoints	Breakpoints for rows (vector)
	Inequality	Relational Operator (enum: `{'<=','>='}`)
	InputFactor	Column input symbol (enum: `{ 'InputX','InputY'}`)
	TableFactor	Table input symbol (enum: `{'InputX','InputY '}`)
2D Table design constraint: InputZ(InputX,InputY) <=InputZmax	Table	: Table constraint (matrix))
	RowBreakpoints	Breakpoints for rows (vector)
	ColumnBreakpoints	Breakpoints for columns (vector)
	Inequality	Relational operator (enum: `{'<=','>='}`)
	RowFactor	Row input symbol (enum: `{'InputX','InputY, 'InputZ'}`)
	ColumnFactor	Column input symbol (enum: `{'InputX','InputY, 'InputZ'}`)
	TableFactor	Table input symbol (enum: `{'InputX','InputY', 'InputZ'}`

Examples

You can use property value pairs to specify constraint properties as part of the `CreateConstraint` command, or you can set properties individually.

For examples, see `CreateConstraint`.

Version History

Introduced in R2008a

See Also

`CreateConstraint`

Properties (for design generators)

View and edit design generator properties

Syntax

```
properties(Generator)
Generator.PropertyName = NewValue
```

Description

"`properties`" (lower case p) is a method of `mbcdoe.generator`, which returns a list of properties.

`properties(Generator)` lists the generator properties.

`Generator.PropertyName = NewValue` sets the generator property.

The design generator object `Type` determines which properties you can set. For more information, see `Type (for designs and generators)`.

The settings are applied immediately, you do not need to call `generate` on the design object.

The following tables list the properties available for each design type.

Optimal Design Properties (D-, V- and A-Optimal)

Property	Description
NumberOfPoints	Number of points (int: [0,Inf])
InitialPoints	Initial design points (Matrix)
CandidateSet	Candidate set (mbcdoe.candidateset)
AllowReplicates	Allow replicate points (boolean)
AugmentMethod	Methods to add points (enum: `{'random','optimal'}`)
Tolerance	Tolerance (numeric: `'positive'`)
MaxIterations	Maximum Iterations (int: `'positive'`)
NumberOfPointsToAlter	Number of points to alter per iteration using the random augment method (p) (int: `'positive'`)
NoImprovement	Number of iterations with no improvement using the random augment method (p) (int: `'positive'`)

Note Optimal designs have dependencies between `NumberOfPoints`, `InitialPoints` and `CandidateSets`. When you change `NumberOfPoints`, an initial point is drawn from the existing candidate set. Setting `NumberOfPoints` updates `InitialPoints`. Likewise setting `InitialPoints` updates `NumberOfPoints`. When changing the candidate set a new initial design is drawn from the new candidate set.

Space-Filling Design Properties

Design Type	Property	Description
All space-filling design types (Lattice, Latin Hypercube Sampling, Stratified Latin Hypercube, Sobol, Halton)	NumberOfPoints	Number of points (int: [0,Inf])
	Limits	Design Limits (matrix: [NumInputs,2])
	BoundaryPercent	Limits the maximum number of boundary points as a percentage of the total number of design of experiment (DoE) points (int: `'positive'`)
Lattice	PrimeGenerators	Prime number generators for lattice for each input (vector int: [0,Inf])
Latin Hypercube Sampling and Stratified Latin Hypercube	SelectionCriteria	Selection criteria for best LHS design (enum: `{'discrepancy', 'minimax', 'maximin', 'cdfvariance', 'cdfmaximum'}`)
	Symmetry	Symmetric design (boolean)
Stratified Latin Hypercube	StratifyLevels	Number of levels for each factors (vector int: {[0,Inf], NumInputs})
	StratifyValues	Stratify levels (cell)
Sobol Sequence	Scramble	Scramble method (enum: `{'none', 'MatousekAffineOwen'}`)
	SkipMode	Skip mode options (enum: `{'None','2^k', 'Custom'}`)
	Skip	Skip size (int: [0,Inf])
Halton Sequence	Scramble	Scrambling method for sequence (enum: `{'None','RR2'}`)
	PrimeLeap	Leap sequence points using prime number (boolean)
	SkipZero	Skip zero point (boolean)

Classical Design Properties

Design Type	Property	Description
All (Box-Behnken, Central Composite, Full Factorial, Plackett-Burman, Regular Simplex)	NumberOfPoints (read-only)	Number of points (int: [0,Inf])
	Limits	Design limits
All except Plackett-Burman	NumberOfCenterPoints	Number of center points (int: [0,Inf])
Central Composite	StarPoints	Star point position (enum: `{'FaceCenteredCube', 'Spherical', 'Rotatable', 'Custom'}`)
	Inscribe	Inscribe points (boolean)
	Alpha	Specify `'Custom'` star point location: (vector: `{'positive',` NumInputs}) For `'FaceCenteredCube', alpha = 1` For `'Spherical', alpha = sqrt(nf)` For `'Rotatable', alpha = 2^(nf/4)`
Full Factorial	Levels	Cell array of levels for each input (cell)
	NumberOfLevels	Number of levels for each input (vector int: `{'positive',` NumInputs })

Examples

You can use property value pairs to specify design generator properties as part of the `Generate` and `Augment` commands. You can also set properties individually. Some examples:

To create a full factorial design and specify the number of levels when generating the design:

```
design = CreateDesign( inputs, 'Type', 'Full Factorial' );
design = Generate( design, 'NumberOfLevels', [50 50] );
```

To create a latin hypercube sampling design:

```
globalDesign = TP.CreateDesign(2,...
'Type', 'Latin Hypercube Sampling');
```

To create and generate a halton design with 50 points:

```
haltonDesign = CreateDesign( inputs, 'Type',...
 'Halton Sequence', 'Name', 'Halton' );
haltonDesign = Generate( haltonDesign, 50 );
```

To explicitly specify the `NumberOfPoints` property:

```
haltonDesign = Generate( haltonDesign, 'NumberOfPoints', 50 );
```

To create and generate a halton design with specified scrambling and other properties:

```
haltonDesignWithScrambling = haltonDesign.CreateDesign...
( 'Name', 'Scrambled Halton' );
haltonDesignWithScrambling = Generate...
( haltonDesignWithScrambling,...
 'Scramble', 'RR2', 'PrimeLeap', true );
```

To generate an optimal design with specified properties:

```
OptDesign = Generate(OptDesign,...
    'Type','V-optimal',...
    'CandidateSet',C,...
    'MaxIterations',200,...
    'NoImprovement', 50,...
    'NumberOfPoints',200);
```

The previous code is equivalent to setting the properties individually and then calling `Generate` as follows:

```
P = OptDesign.Generator;
P.Type = 'V-optimal';
P.CandidateSet.NumberOfLevels(:)=21;
P.MaxIterations = 200;
P.NumberOfPoints = 200;
P.NoImprovement = 50;
OptDesign.Generator = P;
```

To augment a design optimally with 20 points:

```
OptDesign = Augment(OptDesign,...
    'Type','V-optimal',...
    'MaxIterations',200,...
    'NoImprovement', 50,...
    'NumberOfPoints',20);
```

Version History

Introduced in R2008a

See Also

`CreateDesign` | `Generate` | `Augment` | `Properties (for candidate sets)` | `Properties (for design constraints)`

Remove

Remove test plan, model, or boundary model

Syntax

```
OK = Remove(A)
OK = Remove(BoundaryTree,Index)
```

Description

`OK = Remove(A)` removes test plan, or model object `A`. This is a method of all the nondata objects: projects, test plans, all models, and boundary trees. You cannot remove datum models if other models use them.

`OK = Remove(BoundaryTree,Index)` removes the boundary model at `Index`.

Input Arguments

`A` — Object to be removed

test plan | model object

Object to be removed, specified as either a test plan, or a model object.

`BoundaryTree` — BoundaryTree

nondata object

Boundary model to be removed.

`Index` — Index

real positive integer

Index of boundary model to be removed.

Version History

Introduced before R2006a

See Also

RemoveOutliers

Remove outliers in input data by index or rule, and refit models

Syntax

```
R = RemoveOutliers(R, Selection);
```

```
R = RemoveOutliers(L, LocalSelection, GlobalSelection)
```

Description

This is a method of the local model object, `mbcmodel.localresponse` and the response feature model object `mbcmodel.response`.

All the response feature models are refitted after the local models are refitted. Outlier selection is applied to all tests.

For a response model:

- `R` is a response object.
- `Selection` specifies either a set of indices or the name of an outlier selection function, of the following form:

  ```
  Indices = myMfile(model, data, factorName)
  ```

  The factors are the same as defined in `DiagnosticStatistics`.
- `data` contains the factors as columns of a matrix.
- `factorNames` is a cell array of the names for each factor.

For a local model:

- `LocalSelection` is the local outlier selection indices or function.
- `GlobalSelection` is the global outlier selection indices or function.

  Outlier selection functions must conform to this prototype:

  ```
  Indices = myMfile(model, data, factorName)
  ```

  The factors are the same as appear in the scatter plot in the Model Browser.
- `data` contains the factors as columns of a matrix.
- `factorNames` is a cell array of the names for each factor.

Examples

```
outlierind = [1 4 6 7];
RemoveOutliers(thisRF, outlierind);
```

Version History

Introduced before R2006a

See Also

`RemoveOutliersForTest`

RemoveOutliersForTest

Remove outliers on test by index or rule and refit models

Syntax

```
L = RemoveOutliersForTest(LOCALRESPONSE, TESTNUMBER, LOCALSELECTION)
L = RemoveOutliersForTest(LOCALRESPONSE, TESTNUMBER, LOCALSELECTION,
doUpdate)
```

Description

This is a method of `mbcmodel.localresponse`.

`L = RemoveOutliersForTest(LOCALRESPONSE, TESTNUMBER, LOCALSELECTION)` removes outliers, refits the local model, and refits the response feature models.

`L = RemoveOutliersForTest(LOCALRESPONSE, TESTNUMBER, LOCALSELECTION, doUpdate)` removes outliers and if `doUpdate` is `true`, refits all response features after the local model is refitted.

`TESTNUMBER` is the single test number to refit.

`LOCALSELECTION` can either be a set of indices or a function name.

An outlier selection function must take the following form:

```
INDICES = MYMFILE(MODEL, DATA, FACTORNAME);
```

The factors are the same as defined in `DiagnosticStatistics`.

`DATA` contains the factors as columns of a matrix, and `FACTORNAME` is a cell array of the names for each factor.

Examples

For a local response `LOCALRESPONSE`, to remove first two data points and do not update response features:

```
RemoveOutliersForTest(LOCALRESPONSE,1,1:2,false);
```

To find list of indices of removed data points:

```
indices = OutliersForTest(LOCALRESPONSE,1);
```

To restore first data point:

```
RestoreDataForTest(LOCALRESPONSE,1,1,false);
```

To restore all data:

```
RestoreDataForTest(LOCALRESPONSE,1,':',false);
```

To update response features:

```
UpdateResponseFeatures(LOCALRESPONSE);
```

Version History

Introduced before R2006a

See Also

`UpdateResponseFeatures` | `RestoreDataForTest` | `OutlierIndicesForTest` | `RemoveOutliers`

ResponseFeatures(Local Model)

Set of response features for local model

Syntax

```
RFs = L.ResponseFeatures
```

Description

This is a property of the local model object, `mbcmodel.localmodel`.

`RFs = L.ResponseFeatures` returns a `mbcmodel.responsefeatures` object. `L` is the local model.

See "Understanding Model Structure for Scripting" in the Getting Started documentation for an explanation of the relationships between local models, local responses, and other responses.

Available properties and methods are described in the following tables.

Property	Description
EvaluationPoints	Cell array of evaluation points for the response feature set (read-only). An element of `EvaluationPoints` is empty if the response feature does not use the Evaluation point. This property is set up when the response feature is created (see the Add method).
Types	Cell array of types for response feature set (read-only). This property is set up when the response feature is created (see the Add method).
NumberOfResponseFeatures	Number of response features in set (read-only).
IsFitted	The local model has been fitted.

Method	Description

Add	Add new response feature to response feature set
	
	`RF = Add(RF,RFtype)`
	
	`RFtype` is a description character vector belonging to the set of alternative response features. See getAlternativeTypes.
	
	`RF = Add(RF,RFtype,EvaluationPoint)`
	
	`EvaluationPoint` is a row vector with an element for each model input and is used for response features that require an input value to evaluate the response feature (e.g., function evaluation, derivatives). It is an error to specify an evaluation point for a response feature type that does not require an evaluation point.

Method	Description
Remove	Remove a response feature from the response feature set `RF = Remove(RF,index)`
Select	Select a subset of response features from the response feature set `RF = Select(RF,indices)`
getDefaultSet	List of default response features `RF = getDefaultSet(RF)` Returns an `mbcmodel.responsefeatures` object with the default set of response features for the local model.
getAlternativeTypes	List of all alternative response feature types for local model `RFtypes = getAlternativeTypes(RF)` Returns a cell array of response feature type character vectors for the local model.

Evaluate	Evaluate response features
	
	`rfvals = Evaluate(RF);`
	
	Returns the values for the response features for the current local model.
	
	`[rfvals,stderr] = Evaluate(RF)`
	
	Also returns the standard errors for the response features for the current local model. The local model must be fitted before evaluating response features.

Jacobian	Jacobian matrix of response features with respect to parameters `J = Jacobian(RF)` The local model must be fitted before calculating the Jacobian matrix.
Covariance	Covariance matrix for response features `rfvals = Covariance(RF);` The local model must be fitted before calculating the covariance matrix.
Correlation	Correlation matrix for response features `rfvals = Correlation(RF)` Errors occur if model is not fitted.

Method	Description
ReconstructSets	List of subsets of response features which can be used to reconstruct the local model `RFlist = ReconstructSets(RF)` `RFlist` is a cell array of `mbcmodel.responsefeatures`. Each element of `RFlist` can be used to reconstruct the local model from response feature values.

Examples

First, create a local model object:

```
L = mbcmodel.CreateModel('Local Polynomial',2)


L =

   1 + 2*X1 + 8*X2 + 3*X1^2 + 6*X1*X2 + 9*X2^2 + 4*X1^3...
 + 5*X1^2*X2 + 7*X1*X2^2 +
   10*X2^3
   InputData: [0x2 double]
   OutputData: [0x1 double]
   Status: Not fitted
   Linked to Response: not linked
```

The properties of the local model object are the same as the properties of an `mbcmodel.model` object with the additional property “ResponseFeatures”. Look at the response features property as follows:

```
>> RFs = L.ResponseFeatures

RFs =

Response features for Polynomial
    'Beta_1'
    'Beta_X1'
    'Beta_X1^2'
    'Beta_X1^3'
    'Beta_X1^2*X2'
    'Beta_X1*X2'
    'Beta_X1*X2^2'
    'Beta_X2'
    'Beta_X2^2'
    'Beta_X2^3'

% Set up response features
RFtypes = getAlternativeTypes(RFs);
RF = Add(RF, RFtypes{end},-10);

% assign to local model
L.ResponseFeatures = RFs;
```

ResponseFeatures(Local Response)

Array of response features for local response

Syntax

```
RFs = L.ResponseFeatures
```

Description

This is a property of the local model object, `mbcmodel.localresponse`.

`L` is the local response.

See “Understanding Model Structure for Scripting” in the Getting Started documentation for an explanation of the relationships between local responses and other responses.

Examples

```
RFs = Local.ResponseFeatures;
```

ResponseSignalName

Name of signal or response feature being modeled

Syntax

```
ysignal = R.ResponseSignalName
```

Description

This is a property of all response objects: `mbcmodel.hierarchicalresponse`, `mbcmodel.localresponse` and `mbcmodel.response`.

`R` can be a hierarchical response, local response or response.

Examples

```
yName = local.ResponseSignalName;
```

See Also

`InputSignalNames`

RestoreData

Restore removed outliers

Syntax

```
R = RestoreData(RESPONSE)
R = RestoreData(RESPONSE, OUTLIERINDICES)
```

Description

This is a method of `mbcmodel.localresponse` and `mbcmodel.response`.

`R = RestoreData(RESPONSE)` restores all data previously removed as outliers.

`R = RestoreData(RESPONSE, OUTLIERINDICES)` restores all removed data specified in `OutlierIndices`. For a local response, the indices refer to record numbers for all tests.

Examples

```
RemoveOutliers(R, 1:5)
RestoreData(R, 1:2)
```

Version History

Introduced in R2007a

See Also

`RemoveOutliersForTest` | `RemoveOutliers` | `OutlierIndices`

RestoreDataForTest

Restore removed outliers for test

Syntax

```
L = RestoreDataForTest(LOCALRESPONSE, TESTNUMBER, Indices)
L = RestoreDataForTest(LOCALRESPONSE, TESTNUMBER, Indices, doUpdate)
```

Description

This is a method of `mbcmodel.localresponse`.

`L = RestoreDataForTest(LOCALRESPONSE, TESTNUMBER, Indices)` restores all removed data for `TESTNUMBER` specified in `Indices`.

`L = RestoreDataForTest(LOCALRESPONSE, TESTNUMBER, Indices, doUpdate)` restores all specified removed data and if `doUpdate` is `true`, refits all response features. By default, all response feature models will be updated. If a number of tests are being screened it is more efficient to set `doUpdate` to `false` and call `UpdateResponseFeatures` when all the tests have been screened.

`Indices` must be numbers and must belong to the set of outliers in `OutliersForTest`.

Examples

For a local response `LOCALRESPONSE`, to remove first two data points without updating response features:

```
RemoveOutliersForTest(LOCALRESPONSE,1,1:2,false);
```

To find list of indices of removed data points:

```
indices = OutliersForTest(LOCALRESPONSE,1);
```

To restore first data point:

```
RestoreDataForTest(LOCALRESPONSE,1,1,false);
```

To restore all data:

```
RestoreDataForTest(LOCALRESPONSE,1,':',false);
```

To update response features:

```
UpdateResponseFeatures(LOCALRESPONSE);
```

Version History

Introduced in R2007a

See Also

UpdateResponseFeatures | RemoveOutliersForTest | OutlierIndicesForTest

SetTermStatus

Set status of model terms

Syntax

```
M.Properties = M.Properties.SetTermStatus(Terms, Status)
```

Description

This is a method of `mbcmodel.linearmodelproperties`.

`M.Properties = M.Properties.SetTermStatus(Terms, Status)` sets the status of the specified terms in this model. `Status` must be a cell array of status character vectors.

The stepwise status for each term can be Always, Never or Step. The status determines whether you can use the StepwiseRegression function to throw away terms in order to try to improve the predictive power of the model.

`M` is an `mbcmodel.linearmodel` object.

Examples

```
M = mbcmodel.CreateModel('Polynomial', 2);
M.Properties = M.Properties.SetTermStatus([1 2; 1 0],
 {'Never',  'Always'});
```

This example sets the status of the `X1*X2^2` term to `Never` and the `X1` term to `Always`.

Version History

Introduced in R2007a

See Also

`GetTermStatus` | `StepwiseStatus`

SetupDialog

Open fit algorithm setup dialog box

Syntax

`[OPT,OK]= SetupDialog(F)`

Description

This is a method of `mbcmodel.fitalgorithm`.

`[OPT,OK]= SetupDialog(F)` opens the fit algorithm setup dialog box, where you can edit the algorithm parameters. `F` is a `mbcmodel.fitalgorithm` object.

If you click **Cancel** to dismiss the dialog, `OK = false` and no changes are made. If you click **OK** to close the dialog box, then `OK = true` and your new chosen algorithm parameters are set up.

Examples

```
[OPT,OK]= SetupDialog(F)
```

Version History

Introduced in R2007a

See Also

`CreateAlgorithm` | `getAlternativeNames`

SingleVIF

Single VIF matrix for linear model parameters

Syntax

```
vif = SingleVIF(linearmodel)
```

Description

`vif = SingleVIF(linearmodel)` calculates the single Variance Inflation Factor (VIF) matrix for the linear model parameters.

Examples

Calculate Single VIF

Calculate single VIF of knot model.

```
VIF = SingleVIF(knot_model)
```

Input Arguments

`linearmodel` — Model object
`mbcmodel.linearmodel` object

Model object, specified as a `mbcmodel.linearmodel` object.

Output Arguments

`vif` — Single variance inflation factor
matrix

Single variance inflation factor, returned as a matrix.

Version History

Introduced in R2007a

See Also

`ParameterStatistics`

SizeOfParameterSet

Number of model parameters

Syntax

```
N = params.SizeOfParameterSet
```

Description

This is a property of `mbcmodel.linearmodelparameters`, for linear models only. It returns the total possible number of parameters in the model. Note that not all of these terms are necessarily currently included in the model, as you may remove some using StepwiseRegression.

Call NumberOfParameters to see how many terms are currently included in the model. Call StepwiseSelection to see which terms are included and excluded.

Use Names and Values to get the parameter names and values.

Examples

```
N = knotparams.SizeOfParameterSet
```

See Also

`NumberOfParameters` | `StepwiseSelection` | `Names` | `Values`

StepwiseRegression

Change stepwise selection status for specified terms

Syntax

```
[s, outmodel] = StepwiseRegression(inmodel)
[s, outmodel] = StepwiseRegression(inmodel,toggleTerms)
```

Description

`[s, outmodel] = StepwiseRegression(inmodel)` returns the Stepwise table (as in the current stepwise values for `ParameterStatistics`). You can choose to remove or include parameters using StepwiseRegression, as long as their `StepwiseStatus` is `Step`.

`[s, outmodel] = StepwiseRegression(inmodel,toggleTerms)` toggles between `in` and `out` for particular parameters.

Examples

Stepwise Regression of Knot Model

Return the stepwise table of the knot model.

```
[S, knot] = StepwiseRegression(knot)

S =

  1.0e+003 *

    0.1316    0.0606    0.0200       NaN
    0.0000    0.0000    0.0200    2.0919
    0.0000    0.0000    0.0190    0.2828
   -0.0000    0.0000    0.0190    0.2531
    0.0000    0.0000    0.0190    0.2680
   -0.0551    0.0347    0.0200    0.2566
    0.0919    0.0264    0.0200    0.3672
   -0.0040    0.0023    0.0200    0.2564
   -0.0178    0.0095    0.0200    0.2644
    0.0008    0.0004    0.0200    0.2787
```

This example shown toggles parameter 4, after inspection of the Next PRESS column indicates changing the status of this term will result in the lowest PRESS.

```
[S, knot] = StepwiseRegression(knot, 4)
params = knot.Parameters;
N = params.StepwiseSelection

S =

  129.8406   60.1899   19.0000       NaN
    0.0048    0.0008   19.0000  662.3830
```

```
    0.0000    0.0000   18.0000  290.8862
   -0.0021    0.0019   19.0000  245.9833
    0.0001    0.0002   18.0000  281.4104
  -50.4091   34.7401   19.0000  262.8346
   94.9675   26.3690   19.0000  400.6572
   -4.0887    2.2488   19.0000  262.6588
  -17.9412    9.4611   19.0000  276.7535
    0.8229    0.3734   19.0000  292.0827

N =
    'in'
    'in'
    'out'
    'in'
    'out'
    'in'
    'in'
    'in'
    'in'
    'in'
```

Input Arguments

`inmodel` — Input model object
`mbcmodel.linearmodel` object

Input model object, specified as a `mbcmodel.linearmodel` object.

`toggleTerms` — Parameters to toggle
real positive integer | array | `0` (`False`) | `1` (`true`)

Parameters to toggle, specified as an array, a real positive integer, `0` (`False`) or `1` (`true`).

Output Arguments

`outmodel` — Model object
`mbcmodel.linearmodel` object

Output model object, specified as a `mbcmodel.linearmodel` object.

`s` — New stepwise values
`MATLAB` table

New stepwise values, returned as a table.

The Stepwise values returned are the same as those found in the table in the Stepwise GUI. For each parameter, the columns are: the value of the coefficient, the standard error of the coefficient, the t value and Next PRESS (the value of PRESS if the status of this term is changed at the next iteration). Look for the lowest Next PRESS to indicate which terms to toggle in order to improve the predictive power of the model.

More About

Next Steps

After making changes to the model using StepwiseRegression you must call `UpdateResponseFeatures`.

Use `StepwiseStatus` (on the child `modelparameters` object) to see which parameters have a status of `Step`; these can be toggled between `in` and `out` using StepwiseRegression (on the parent model object).

Use `StepwiseSelection` (on the child `modelparameters` object) to view which terms are `in` and `out`.

Version History

Introduced before R2006a

See Also

`StepwiseSelection` | `StepwiseStatus` | `ParameterStatistics`

StepwiseSelection

Model parameters currently included and excluded

Syntax

```
N =  paramsknot.StepwiseSelection
```

Description

This is a read-only property of `mbcmodel.linearmodelparameters`, for linear models only. It returns a status for each parameter in the model, `in` or `out`, depending on whether the term is included or excluded. You can choose to remove or include parameters using StepwiseRegression, as long as their StepwiseStatus is `Step`. Call StepwiseRegression (on the parent model object) to toggle between `in` and `out` for particular parameters. You must then call UpdateResponse before calling StepwiseSelection.

Examples

```
 N = paramsknot.StepwiseSelection
N =
    'in'
    'in'
    'out'
    'out'
    'out'
    'in'
    'in'
    'in'
    'in'
    'in'
```

See Also

`StepwiseRegression` | `StepwiseStatus` | `NumberOfParameters` | `UpdateResponse`

StepwiseStatus

Stepwise status of parameters in model

Syntax

```
N = paramsknot.StepwiseStatus
```

Description

This is a method of `mbcmodel.linearmodelparameters`, for linear models only. It returns the stepwise status of each parameter in the model.

The stepwise status for each term can be Always, Never or Step. The status determines whether you can use the `StepwiseRegression` function to throw away terms in order to try to improve the predictive power of the model.

- `Always` - Always included in the model.
- `Never` - Never included in the model.
- `Step` - You can choose whether to include or exclude this term. Do this by using StepwiseRegression to toggle between `in` and `out` for particular parameters.

Use StepwiseSelection to find out which terms are currently included and excluded.

Examples

```
N = paramsknot.StepwiseStatus
N =
    'Always'
    'Step'
    'Step'
    'Step'
    'Step'
    'Step'
    'Step'
    'Step'
    'Step'
    'Step'
```

Version History

Introduced before R2006a

See Also

`StepwiseRegression` | `StepwiseSelection`

SummaryStatisticsForTest

Statistics for specified test

Syntax

```
SS = SummaryStatisticsForTest( LocalResponse, TestNumber )
SS = SummaryStatisticsForTest(LocalResponse,TestNumber,Names)
```

Description

This is a method of `mbcmodel.localresponse`.

`SS = SummaryStatisticsForTest( LocalResponse, TestNumber )` returns a structure array containing Statistics and Names fields values for the local model for test TestNumber.

`SS = SummaryStatisticsForTest(LocalResponse,TestNumber,Names)` returns an array of the statistics specified by `Names`. `Names` can be a char array, or a cell array of character vectors.

Examples

```
SS = SummaryStatisticsForTest( L, 22 )
```

Version History

Introduced in R2007b

See Also

`SummaryStatistics`

TestPlan

Test plan containing boundary tree

Syntax

```
Tree.TestPlan
```

Description

This is a property of `mbcboundary.Tree` and `mbcboundary.TwoStageTree`.

`Tree.TestPlan` returns the test plan object that contains this boundary tree (read only).

Type (for candidate sets)

Candidate set type

Syntax

```
C.Type
```

Description

This is a property of `mbcdoe.candidateset`.

`C.Type` returns the candidate set type. You can only choose a type when you create the candidate set, when calling `CreateCandidateset`.

You can specify the candidate set type during creation by using the `Type` property, e.g.,

```
CandidateSet = augmentedDesign.CreateCandidateSet...
( 'Type', 'Grid' );
```

Other available properties depend on the candidate set type. To see the properties you can set, see the table of candidate set properties, Candidate Set Properties (for Optimal Designs).

See Also

`CreateCandidateSet`

Type (for designs and generators)

Design type

Syntax

```
D.Type
G.Type = NewType
```

Description

This is a read-only property of `mbcdoe.design`, and a settable property of `mbcdoe.generator`.

`D.Type` returns the design type. You can only choose a type when you create designs. After design creation, you can only set the `Type` of a `mbcdoe.generator` object, or when calling `Generate` or `Augment`.

`G.Type = NewType` changes the Type, where `G` is a `mbcdoe.generator` object.

The design Type determines which properties you can set. To set properties, see `Properties (for design generators)`.

Get a list of types which could be used as alternative designs for current design, using `getAlternativeTypes`, by entering the following syntax:

```
Dlist = getAlternativeTypes(D)
```

where `D` is an `mbcdoe.design` object.

The design Type must be one shown in the following table. The read-only Style property is derived from the Type.

Style	Type
Optimal	D-Optimal
	V-Optimal
	A-Optimal
Classical	Box-Behnken
	Central Composite
	Full Factorial
	Plackett-Burman
	Regular Simplex
Space-filling	Lattice
	Latin Hypercube Sampling
	Stratified Latin Hypercube
	Sobol Sequence
	Halton Sequence

Style	Type
Experimental data	Design points replaced by data points
Custom	Any design with a mix of Types (eg an optimally augmented space-filling design)

Examples

To specify the Type while creating and then generating a design of a given size:

```
D = CreateDesign(model,'Type','Sobol Sequence')
D = Generate(D,128);
```

See Also

Properties (for design generators) | Generate | Augment

Type (for design constraints)

Design constraint type

Syntax

`C.Type`

Description

This is a property of `mbcdoe.constraint`.

`C.Type` returns the design constraint type. You can only choose a type when you create the constraint, when calling `CreateConstraint`.

You can specify the constraint type during creation by using the `Type` property, e.g.,

```
c = D.CreateConstraint('Type','Linear')
```

Other available properties depend on the constraint type. See the table Constraint Properties.

The constraint Type must be one shown in the following table.

Constraint Type	Description
'Linear'	Linear design constraint: 1*Input1 + 1* Input2 + 1* Input3 <= 0
'Ellipsoid'	Ellipsoid design constraint: Ellipsoid at (Input1=0, Input2=0, Input3=0)
'1D Table'	1D Table design constraint: InputY(InputX) <= InputY max
'2D Table'	2D Table design constraint: InputZ(InputX,InputY) <=InputZmax

See Also

`CreateConstraint` | Constraint Properties

Units

Model output units

Syntax

```
model.Units
modelinput.Units
```

Description

This is a property of `mbcmodel.model` and `mbcmodel.modelinput` objects.

`model.Units` or `modelinput.Units` return the units of the model or modelinput object.

This property is set to the data signal units when the response is created or if a model is assigned to a response. This property cannot be set when a response is attached to the model.

Update

Update boundary model in tree and fit to test plan data

Syntax

```
B = Update(Tree,Index,B)
B = Update(Tree,Index,B,InBest)
```

Description

This is a method of `mbcboundary.Tree`.

`B = Update(Tree,Index,B)` updates the boundary model `B` in the boundary tree `Tree`, and fits the boundary model to the test plan data. `Tree` is an `mbcboundary.Tree` object, `Index` is the index to boundary model in the tree, and `B` is a boundary model object. The boundary model must have the same inputs as the boundary tree. The boundary model is always fitted when you add it to the boundary tree. This fitting ensures that the fitting data is compatible with the test plan data. The method returns the fitted boundary model.

`B = Update(Tree,Index,B,InBest)` updates the boundary model in the tree and `InBest` specifies whether to include the boundary model in the best boundary model for the boundary tree. By default, the boundary model retains its previous `InBest` status after calling `Update`.

Version History

Introduced in R2009b

See Also

`Add` | `Remove` | `CreateBoundary`

UpdateResponse

Replace model in response

Syntax

```
UpdateResponse(model)
M = UpdateResponse(model,R)
```

Description

`UpdateResponse(model)` updates the model in the response associated with the model.

`M = UpdateResponse(model,R)` updates the response specified by `R`.

Examples

Update Response Associated with Model

Update response associated with knot model.

```
UpdateResponse(knot);
```

Input Arguments

`model` — Model object
`mbcmodel.linearmodel` object | `mbcmodel.model` object

Model object, specified as `mbcmodel.linearmodel` or `mbcmodel.model` objects.

`R` — Response
`mbcmodel.linearmodel` object | `mbcmodel.model` object

Response object.

Output Arguments

`M` — Updated response model
`mbcmodel.linearmodel` object | `mbcmodel.model` object

Updated response model object, returned as `mbcmodel.linearmodel` or `mbcmodel.model` objects.

More About

Usage

This is a method of `mbcmodel.model`. This takes the model and places it back into the response it came from. Appropriate action is taken if a refit is necessary because you have modified either the model, response data or model data in the interim. For example, if you have changed the model type, the new model is fitted to the response data. If you have changed the response data (e.g. removed an outlier), the model is fitted to the new response data.

Note that when changing the model type or settings (using the `ModelSetup` command) the response is not refitted until you call UpdateResponse. If you have changed the model by using `StepwiseRegression` you must call UpdateResponse.

Version History

Introduced before R2006a

See Also

UpdateResponseFeatures

Refit response feature models

Syntax

```
UpdateResponseFeatures(L)
```

Description

This is a method of `mbcmodel.localresponse`.

`UpdateResponseFeatures(L)` refits all response feature models. You need to call this if you used RemoveOutliersForTest without specifying refitting the response features (`doUpdate` set to false).

Examples

For a local response `LOCALRESPONSE`, to remove first two data points without updating response features:

```
RemoveOutliersForTest(LOCALRESPONSE,1,1:2,false);
```

To update response features:

```
UpdateResponseFeatures(LOCALRESPONSE);
```

Version History

Introduced in R2007a

See Also

`RemoveOutliersForTest` | `RestoreDataForTest`

Values

Values of model parameters

Syntax

```
vals = paramsknot.Values
```

Description

This is a read-only property of `mbcmodel.modelparameters`. It returns the value of each parameter in the model. Use Names to find out the names of these terms.

Examples

```
vals = paramsknot.Values;
```

See Also

`Names`

xregstatsmodel

Class for evaluating models and calculating PEV

Syntax

```
y = StatsModel(X)
Y = EvalModel(StatsModel, X)
[pev, Y] = pev(StatsModel, X)
C = ceval(StatsModel, X)
df = dferror(StatsModel)
Interval = predint(StatsModel,X,Level);
n = nfactors(StatsModel)
[n,symbols,units] = nfactors(StatsModel)
```

Description

Use the `xregstatsmodel` class to evaluate a model and calculate the prediction error variance.

You can create an `xregstatsmodel` object by either:

- Exporting a model from the Model Browser to the workspace.
- Converting any command line response or model object to an `xregstatsmodel` by using the Export method.

  Use the Export method to convert `mbcmodel.hierarchicalresponse`, `mbcmodel.localresponse`, `mbcmodel.response` and `mbcmodel.model` objects to `xregstatsmodel` objects. Use the syntax `ExportedModel = Export(MODEL)`. The default format is `'MATLAB'` so you do not need to specify the format.

After you create an `xregstatsmodel` object, you can use the following methods to evaluate the model and calculate the prediction error variance:

- `EvalModel` — evaluate model

  `pev` — evaluate prediction error variance

  `ceval` – evaluate boundary model

  `dferror` — degrees of freedom for error

  `predint` — calculate confidence intervals for model prediction

  `nfactors` — get number of input factors

If you convert an `mbcmodel.localresponse` object using Export and you have not created a two-stage model (hierarchical response object), then the output is an `mbcPointByPointModel` object. Point-by-point models are created from a collection of local models for different operating points. `mbcPointByPointModel` objects share all the same methods as `xregstatsmodel` except `dferror`.

`y = StatsModel(X)` evaluates the `xregstatsmodel` model object `StatsModel` at input values `X`. `X` is a `(N-by-NF)` array, where `NF` is the number of inputs, and `N` the number of points to evaluate the model at.

`Y = EvalModel(StatsModel, X)` evaluates the model at input values X. You can also evaluate models using parentheses, e.g., `y = StatsModel(X)`

`[pev, Y] = pev(StatsModel, X)` calculates the prediction error variance of the model at `X`, `pev`, and also returns `Y` the evaluated model at `X`.

`C = ceval(StatsModel, X)` evaluates the boundary model constraints at `X`.

`df = dferror(StatsModel)` gets the degrees of freedom for the model.

`Interval = predint(StatsModel,X,Level);` calculates the confidence interval for model prediction. A `Level` confidence interval of the predictions is calculated about the predicted value. The default value for `Level` is `99`. `Interval` is a `Nx2` array where the first column is the lower bound and the second column is the upper bound.

`n = nfactors(StatsModel)` gets the number of input factors of the model. `[n,symbols,units] = nfactors(StatsModel)` returns the number, symbols and units of the input factors in the model.

Version History
Introduced in R2010a

See Also
`Export`

MBCModel.Project

mbcmodel.project

Properties and methods for project objects

Description

Use these properties and object functions to create and examine project objects.

Creation

Create a mbcmodel.project object using `CreateProject`.

Properties

`Name` — Project object name
character vector

Project object name, specified as a character vector.

Data Types: `char` | `string`

`Filename` — Project file path
character vector

This property is read-only.

Project file path, specified as a character vector.

Data Types: `char` | `string`

`Modified` — Project modification status
`0` or `false` | `0` or `true`

This property is read-only.

Project modification status, specified as `0` (`false`) or `1` (`true`).

Data Types: `double` | `logical`
Complex Number Support: Yes

`Data` — Data objects
array

This property is read-only.

Data objects to be returned to `mbcmodel.project`, specified as an array.

`Testplans` — Test plan objects
array

This property is read-only.

Test plan objects to be returned to `mbcmodel.project`, specified as an array.

Object Functions

CopyData	Create data object from copy of existing object
CreateProject	Create project object for model
CreateTestplan	Create new test plan
Load	Load existing project file
Save	Save project
RemoveData	Remove data from project
Remove	Remove project model
New	Create new project file

Examples

Create Data from MBC Project

Create a data from an existing project object ProjObj.

```
data = CreateData(P, 'D:\MBCWork\data1.xls');
D = mbcmodel.CreateData;
D = mbcmodel.CreateData('D:\MBCWork\data.xls');
```

Version History

Introduced before R2006a

See Also

CreateProject

Create project object for model

Syntax

```
ProjectObj = CreateProject(Name)
```

Description

`ProjectObj = CreateProject(Name)` creates a project object called `Name` for a `mbcmodel` object.

Examples

Create Project and Test Plan

Speed (*N*) and fuel (*F*) are global inputs. Injection (*soi*), fuel pressure (*fuelpress*), variable geometry turbo rack position (*grackmea*) and exhaust gas recirculation (*EGR*) are local inputs.

```
project = mbcmodel.CreateProject('DieselMulti');

% Define Inputs for test plan
LocalInputs = mbcmodel.modelinput('Symbol',{'S','P','G','E'},...
    'Name',{'soi','fuelpress','grackmea','egrlift'},...
    'Units',{'deg','MPa','ratio','mm'},...
    'Range',{[-9 3],[60 160],[0.2 0.9],[0.5 5]});
GlobalInputs = mbcmodel.modelinput('Symbol',{'N','F'},...
    'Name',{'measrpm','basefuelmass'},...
    'Units',{'rpm','mg/stroke'},...
    'Range',{[1600 2200],[20 200]});
% create test plan
TP = CreateTestplan( project, {LocalInputs,GlobalInputs} );
```

Input Arguments

Name — Project name
character vector

Project name.

Example: `'DieselMulti'`

Output Arguments

`ProjectObj` — Project object
project object

Output project object created by `CreateProject`, returned as a design object.

Version History

Introduced in R2008a

See Also

`mbcmodel.project`

Load

Load existing project file

Syntax

```
P = Load(P, Filename)
```

Description

`P = Load(P, Filename)` loads the existing project in the `Filename`.

Examples

Load Project from File

```
P2 = Load(P2, 'D:/MBCwork/TQproject2.mat');
```

Input Arguments

P — Project
object

Existing project object.

`Filename` — Path to project file
character vector

Path to project file to load, specified as a character vector.

Version History

Introduced before R2006a

See Also

New | `mbcmodel.project`

CopyData

Create data object from copy of existing object

Syntax

```
newD = CopyData(PrjObj,D)
newD = CopyData(PrjObj,I)
```

Description

`newD = CopyData(PrjObj,D)` copies data from the data object `D`.

`newD = CopyData(PrjObj,I)` copies data from the data object whose index is `I`.

Input Arguments

`D` — Data object
data object

Data object to copy, specified as a data object.

`I` — Index
real positive integer

Index of data object to copy, specified as a real positive integer.

`PrjObj` — Project object
project object

Existing project, specified as a project object.

Output Arguments

`newD` — Duplicate data object
data object

Duplicate data object of `D`, returned as a data object. Use this syntax to duplicate data, for example, if you want to make changes for further modeling but want to retain the existing dataset.

Version History

Introduced before R2006a

See Also

`mbcmodel.project`

New

Create new project file

Syntax

```
ModP = New(PrjObj)
```

Description

`ModP = New(PrjObj)` modifies a project object to make a new project from scratch. Note the current project gets removed from memory when you open a new one.

Input Arguments

`PrjObj` — Project object
project object

Existing project, specified as a project object.

Output Arguments

`ModP` — Modified project
project object

Modified project object created from `P`, returned as a project object.

Version History

Introduced before R2006a

See Also

`mbcmodel.project`

Remove

Remove project model

Syntax

`OK = Remove(A)`

Description

`OK = Remove(A)` removes the project object `A`.

Input Arguments

`A` — Object to be removed
project object | test plan object | model object

Object to be removed, specified as a project, test plan, or model object.

Version History

Introduced before R2006a

See Also

`mbcmodel.project`

RemoveData

Remove data from project

Syntax

```
NewP = RemoveData(PrjObj, D)
NewP = RemoveData(PrjObj, Index)
```

Description

`NewP = RemoveData(PrjObj, D)` removes data object `D` from project object `P`.

`NewP = RemoveData(PrjObj, Index)` removes data object `D` at `Index`.

Input Arguments

`Index` — Index
real positive integer

Index of the data object you want to remove.

`D` — Data object
object

Data object you want to remove.

`PrjObj` — Project object
project object

Existing project, specified as a project object.

Output Arguments

`NewP` — New project object
object

New project object, with data object `D` removed.

Version History

Introduced before R2006a

See Also

`mbcmodel.project`

Save

Save project

Syntax

```
OK = Save(PrjObj)
OK = Save(PrjObj, filename)
```

Description

`OK = Save(PrjObj)` saves the project `P` to the currently selected `filename`. The project `name` is used as the `Filename` if none has been specified previously. If neither has been specified, you will see a warning that your project has been saved to `Untitled.mat`.

`OK = Save(PrjObj, filename)` saves the project `PrjObj` with the name specified by `filename`.

Examples

Save Project to Specified Filename

```
OK = Save(proj, 'Example.mat');
```

Input Arguments

`PrjObj` — Project object
project object

Existing project, specified as a project object.

`filename` — Name of project file
character vector

Name of project file where you want to save your project, specified as a character vector.

Version History

Introduced before R2006a

See Also

`mbcmodel.project`

CreateTestplan

Create new test plan

Syntax

```
TPObj = CreateTestplan(PrjObj,TestPlanTemplate)
TPObj = CreateTestplan(PrjObj,TestPlanTemplate,newtestplanname)
TPObj = CreateTestplan(PrjObj,InputsPerLevel)
TPObj = CreateTestplan(PrjObj,InputsPerLevel,newtestplanname)
TPObj = CreateTestplan(PrjObj,Inputs)
TPObj = CreateTestplan(PrjObj,Inputs,newtestplanname)
```

Description

`TPObj = CreateTestplan(PrjObj,TestPlanTemplate)` creates a test plan.

`TPObj = CreateTestplan(PrjObj,TestPlanTemplate,newtestplanname)` creates a test plan with a name.

`TPObj = CreateTestplan(PrjObj,InputsPerLevel)` creates a test plan with the number of inputs per level.

`TPObj = CreateTestplan(PrjObj,InputsPerLevel,newtestplanname)` creates a test plan with the inputs per level and a name.

`TPObj = CreateTestplan(PrjObj,Inputs)` creates a test plan with the number of inputs.

`TPObj = CreateTestplan(PrjObj,Inputs,newtestplanname)` creates a test plan with the number of inputs and a name.

Examples

Create Test Plan Using Template

To create a test plan using a test plan template, enter:

```
T = CreateTestplan(P1, 'd:\MBCwork\TQtemplate1', 'newtestplan')
testplan = CreateTestplan(P, 'example_testplan')
```

To create a test plan using inputs per level, enter:

```
T = P.CreateTestplan([1,2])
```

To specify the input information in a cell array of `mbcmodel.modelinput` objects, enter:

```
% Define Inputs for test plan
LocalInputs = mbcmodel.modelinput('Symbol','S',...
    'Name','SPARK',...
    'Range',[0 50]);
GlobalInputs = mbcmodel.modelinput('Symbol',{'N','L','ICP',...
```

```
'ECP'},'Name',{'SPEED','LOAD','INT_ADV','EXH_RET'},...
    'Range',{[500 6000],[0.0679    0.9502],[-5 50],[-5 50]});
% create test plan
testplan = CreateTestplan( project, {LocalInputs,...
GlobalInputs} );
```

Or

```
T = P.CreateTestplan({LocalInputs,GlobalInputs})
```

To specify the input information in a cell array, enter:

```
localInputs = {'S',0,50,'','SPARK'};
globalInputs = {'N',   800, 5000, '', 'ENGSPEED'
           'L',   0.1,    1, '', 'LOAD'
           'EXH',  -5,   50, '', 'EXHCAM'
           'INT',  -5,   50, '', 'INTCAM'};

T = CreateTestplan(P,{localInputs,globalInputs});
```

Input Arguments

`Inputs` — Input information

cell array

Input information, specified as a cell array. The input information can be specified as a cell array of `mbcmodel.model` input objects (one for each level), or as a cell array of cell arrays (one for each level).

`InputsPerLevel` — Number of inputs for each stage

row vector

Number of inputs for each stage, specified as a row vector.

`newtestplanname` — Optional name

character vector

Optional name for the new test plan object, specified as a character vector.

`TestPlanTemplate` — Template and test plan

character vector

Template and test plan name and path, specified as a character vector. The test plan template file is created in the Model Browser.

`PrjObj` — Project object

project object

Existing project, specified as a project object.

Output Arguments

`TPObj` — Test plan object

test plan object

New test plan, returned as a project object.

More About

Creating test plans

You can use this method with a test plan template or input information.

You set up templates in the Model Browser. This setup includes number of stages, inputs, base models, and designs. If the test plan is used as part of a previous project, then you can save response models in the test plan. You cannot change the number of stages after creation of the test plan.

After you create a new test plan, you can add data to model, and new responses. Note that the model input signal names specified in the template *must* match the signal names in the data.

Version History

Introduced before R2006a

See Also

`mbcmodel.project` | `mbcmodel.testplan` | `AttachData` | `CreateResponse` | `Data` | `InputSignalNames` | `Inputs` | `modelinput`

MBCModel.Model

mbcmodel.model

Properties and methods for model objects

Description

Use these properties and object functions to create and examine model objects.

Creation

Create a mbcmodel.model object using `CreateModel`.

Properties

`Data` — Data stored in model
array

Data stored in model, returned as an array.

Type — Type of model objects
vector

This property is read-only.

Type of model objects to be returned to `mbcmodel.project`, specified as a vector. `model.Type` returns the model type.

Note Spaces and case in model `Type` are ignored.

The model type must be one in this table.

Type	Model Object
Polynomial	`mbcmodel.linearmodel`
Hybrid Spline	`mbcmodel.linearmodel`
RBF	`mbcmodel.linearmodel`
Hybrid RBF	`mbcmodel.linearmodel`
Polynomial-RBF	`mbcmodel.linearmodel`
Hybrid Spline-RBF	`mbcmodel.linearmodel`
Multiple Linear	`mbcmodel.linearmodel`
Gaussian Process	`mbcmodel.model`
Free Knot Spline	`mbcmodel.model`
Transient	`mbcmodel.model`

Type	Model Object
User-Defined	`mbcmodel.model`
Neural Network	`mbcmodel.model`
Interpolating RBF	`mbcmodel.model`
Local Polynomial Spline	`mbcmodel.localmodel`
Local Polynomial with Datum	`mbcmodel.localmodel`
Local Polynomial	`mbcmodel.localmodel`
Local Hybrid Spline	`mbcmodel.localmodel`
Local Truncated Power Series	`mbcmodel.localmodel`
Local Free Knot Spline	`mbcmodel.localmodel`
Local Multiple Models	`mbcmodel.localmodel`
Local Growth	`mbcmodel.localmodel`
Local User-Defined	`mbcmodel.localmodel`
Local Transient	`mbcmodel.localmodel`
Local Average Fit	`mbcmodel.localmodel`

You can get a list of types by using `getAlternativeTypes`. Use this syntax.

```
Mlist = getAlternativeTypes(M)
```

where `M` is an `mbcmodel.model` object.

Data Types: `char` | `string`

`Inputs` — Model input
`mbcmodel.modelinput` object

Model input, specified as a `modelinput` object.

`Status` — Status of model fit
`Not Fitted` | `Fitted` | `Best`

This property is read-only.

Status of model fit, specified as either `Not Fitted`, `Fitted`, or `Best`.

Data Types: `char` | `string`

`NumInputs` — Number of inputs to model
real positive scalar

This property is read-only.

Number of inputs to model, specified as a real positive scalar.

Data Types: `double` | `single`

`InputData` — Input training data
matrix

This property is read-only.

Input training data, specified as a matrix. `InputData` is specified when calling fit.

Data Types: `double`

`OutputData` — Output or response data
matrix

This property is read-only.

Output or response data, specified as a matrix. `OutputData` is specified when calling fit.

Data Types: `double`

`FitAlgorithm` — Fit algorithm for model
array

Fit algorithm for model, specified as an array.

`FitAlgorithm` is a property of `mbcmodel.model`, and boundary model objects `mbcboundary.AbstractBoundary` and all subclasses.

An `mbcmodel.model.FitAlgorithm` object is contained within the `FitAlgorithm` property of an `mbcmodel.model` object or `mbcboundary` object.

As an alternative to using `CreateAlgorithm`, you can assign the algorithm name directly to the algorithm.

```
B.FitAlgorithm.BoundaryPointOptions = 'Boundary Only';
```

```
m.FitAlgorithm = 'Minimize PRESS';
```

Case and spaces are ignored.

To get a `fitalgorithm` object, `F`, from a model, use this code.

```
M = mbcmodel.CreateModel('Polynomial', 4);
F = M.FitAlgorithm
```

```
F =
Algorithm: Least Squares
Alternatives: 'Minimize PRESS','Forward Selection','Backward
Selection','Prune'
1x1 struct array with no fields.
```

`Name` — Model object name
character vector

Model object name, specified as a character vector.

Data Types: `char` | `string`

`Units` — Model output unit
vector

Unit of model output, specified as a vector.

Data Types: `double` | `single`

Response — Response object
object

This property is read-only.

Response object in `mbcmodel.project` object, specified as an object.

`IsBeingEdited` — Boolean indicating if model is being edited
`true` or `1` | `false` or `0`

This property is read-only.

Boolean indicating if model is being edited, specified as either `true` (`1`) or `false` (`0`).

Example: 0

Data Types: `logical`

`IsEditable` — Boolean indicating if model is editable
`true` or `1` | `false` or `0`

This property is read-only.

Boolean signaling if model is editable, specified as either `true` (`1`) or `false` (`0`). The following rules apply:

- If the model was created using `mbcmodel.CreateModel` and is not `Attached` to a test plan it is editable.
- If the model was created or retrieved from the project and was not `Attached` to a test plan, it is editable.
- If the data was `Attached` to a test plan and was subsequently retrieved from that test plan, it is editable.

Data Types: `logical`

Object Functions

CreateModel	Create new model
CreateDesign	Create design object for test plan or model
evaluate	Evaluate model, boundary model, or design constraint
Export	Make command-line or Simulink export model
fit	Fit model or boundary model to new or existing data, and provide summary statistics
InputSetupDialog	Open Input Setup dialog box to edit inputs
Jacobian	Calculate Jacobian matrix for model at existing or new data points
ModelSetup	Open Model Setup dialog box where you can alter model type
pev	Predicted error variance of model at specified inputs
PredictedValue	Predicted value of model at specified inputs
StatisticsDialog	Open summary statistics dialog box
SummaryStatistics	Summary statistics for response
UpdateResponse	Replace model in response
getAlternativeTypes	Alternative model or design types
ValidationRMSE	Calculates the validation RMSE for model data

Version History

Introduced before R2006a

See Also

`mbcmodel.project` | `mbcdoe.design` | `mbcmodel.data`

CreateModel

Create new model

Syntax

```
ModelObj = CreateModel(Type,Inputs)
ModelObj = CreateModel(Model,Type)
```

Description

`ModelObj = CreateModel(Type,Inputs)` creates an `ModelObj` object of the specified `Type`.

`ModelObj = CreateModel(Model,Type)` creates a new model (of the specified `Type`) with the same inputs as an existing `Model`, where `Model` is a `ModelObj` object.

Examples

Create Hybrid Spline

Create a hybrid spline with four input factors.

```
M = mbcmodel.CreateModel('Hybrid Spline', 4)
```

Create RBF

Create an RBF with four input factors.

```
Inputs = mbcmodel.modelinput('Symbol',{'N','L','EXH','INT'}',...
   'Name',{'ENGSPEED','LOAD','EXHCAM','INTCAM'}',...
   'Range',{[800 5000],[0.1 1],[-5 50],[-5 50]}');

RBFModel = mbcmodel.CreateModel( 'RBF', Inputs);
```

Create Polynomial

Create a polynomial with the same input factors as the previously created RBF.

```
PolyModel = CreateModel(RBFModel,'Polynomial')
```

Input Arguments

Type — Type of model
vector

Type of model objects.

Note Spaces and case in model `Type` are ignored.

The model type must be one in this table.

Type	Model Object
Polynomial	`mbcmodel.linearmodel`
Hybrid Spline	`mbcmodel.linearmodel`
RBF	`mbcmodel.linearmodel`
Hybrid RBF	`mbcmodel.linearmodel`
Polynomial-RBF	`mbcmodel.linearmodel`
Hybrid Spline-RBF	`mbcmodel.linearmodel`
Multiple Linear	`mbcmodel.linearmodel`
Gaussian Process	`mbcmodel.model`
Free Knot Spline	`mbcmodel.model`
Transient	`mbcmodel.model`
User-Defined	`mbcmodel.model`
Neural Network	`mbcmodel.model`
Interpolating RBF	`mbcmodel.model`
Local Polynomial Spline	`mbcmodel.localmodel`
Local Polynomial with Datum	`mbcmodel.localmodel`
Local Polynomial	`mbcmodel.localmodel`
Local Hybrid Spline	`mbcmodel.localmodel`
Local Truncated Power Series	`mbcmodel.localmodel`
Local Free Knot Spline	`mbcmodel.localmodel`
Local Multiple Models	`mbcmodel.localmodel`
Local Growth	`mbcmodel.localmodel`
Local User-Defined	`mbcmodel.localmodel`
Local Transient	`mbcmodel.localmodel`
Local Average Fit	`mbcmodel.localmodel`

Data Types: `char` | `string`

`Inputs` — Model input
`mbcmodel.modelinput` object

Model input, specified as a `modelinput` object.

`Model` — Model object
`mbcmodel.model` object

Model object.

An existing `ModelObj` object.

Output Arguments

`ModelObj` — Model object
`mbcmodel.model` object

Model created by `CreateModel`, returned as a model object.

Version History

Introduced in R2007a

See Also

`CreateData` | `mbcmodel.model`

CreateDesign

Create design object for test plan or model

Syntax

```
D = CreateDesign(Testplan)
D = CreateDesign(Testplan,Level)
D = CreateDesign(Testplan,Level,Name1,Value1,...)
D = CreateDesign(Model)
D = CreateDesign(Model,Name1,Value1,...)
D = CreateDesign(Inputs)
D = CreateDesign(Inputs,Name1,Value1,...)
D = CreateDesign(Design)
```

Description

`D = CreateDesign(Testplan)` creates a design for the test plan, where `Testplan` is an `mbcmodel.testplan` object.

`D = CreateDesign(Testplan,Level)` creates a design for the specified level, `Level`, of the test plan.

If you do not specify any properties, the method creates a default design type. The default design types are a Sobol Sequence for two or more inputs and a Full Factorial for a single input.

`D = CreateDesign(Testplan,Level,Name1,Value1,...)` creates a design for the specified level of the `mbcmodel.testplan` object, with the specified name-value pairs.

`D = CreateDesign(Model)` creates a design based on the inputs of the `mbcmodel.model` object `Model`.

`D = CreateDesign(Model,Name1,Value1,...)` creates a design based on the inputs of the `mbcmodel.model` object, with the specified name-value pairs.

`D = CreateDesign(Inputs)` creates a design based on the inputs of the `mbcmodel.modelinput` object `Inputs`.

`D = CreateDesign(Inputs,Name1,Value1,...)` creates a design based on the inputs of the `mbcmodel.modelinput` object, with the specified name-value pairs.

`D = CreateDesign(Design)` creates a copy of an existing design, `Design`.

Examples

Create Space Filling Design

Create a space-filling design for the test plan `TP`.

```
sfDesign = CreateDesign(TP, ...
    'Type', 'Latin Hypercube Sampling',...
    'Name', 'Space Filling');
```

Create Optimal Design

Create an optimal design based on the inputs of a model.

```
optimalDesign = CreateDesign( model,...
    'Type', 'V-optimal',...
    'Name', 'Optimal Design' );
```

Create Classical Full Factorial Design

Create a classical full factorial design based on the inputs defined by a `mbcmodel.modelinput` object.

```
design = CreateDesign( inputs, 'Type', 'Full Factorial' );
```

Create New Design

Create a new design, `augmentedDesign`, based on an existing design, `ActualDesign`, in order to augment the design.

```
augmentedDesign = ActualDesign.CreateDesign('Name',...
 'Augmented Design');
```

Create Local-Level Design

Create a local-level design for the two-stage test plan `TP`.

```
localDesign = TP.CreateDesign(1,'Type',...
'Latin Hypercube Sampling');
```

Create Global-Level Design

Create a global-level design for the two-stage test plan `TP`.

```
globalDesign = TP.CreateDesign(2, 'Type',...
 'Latin Hypercube Sampling');
```

Input Arguments

`Testplan` — Test plan objects

`mbcmodel.Testplan` object

Test plan objects, specified as a `mbcmodel.Testplan` object created by `CreateTestplan`.

`Level` — Test plan level
outer level (default) | `1` | `2`

Test plan level, specified as either `1` or `2`. By default, the level is the outer level: 1 for one-stage design and 2 (global) for two-stage design.

`Model` — Model objects
`mbcmodel.model` object

Model objects, specified as a `mbcmodel.model` object.

`Inputs` — Inputs objects
`mbcmodel.modelinput` object

Input objects, specified as a `modelinput` object.

`Design` — Design object
design object

Design object being copied, specified as a design object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `'Type','V-optimal'`

`Type` — Design time
character vector

Design time, specified as the comma-separated pair consisting of 'Type' and a value.

Example: `'Type','Latin Hypercube Sampling'`

`Name` — Design name
character vector

Design name, specified as the comma-separated pair consisting of 'Name' and a value.

Example: `'Name','Optimal Design'`

Output Arguments

`Design` — Output design object
design object

Output design object created by `CreateDesign`, returned as a design object.

More About

Usage

`CreateDesign` is a method of `mbcmodel.testplan`, `mbcmodel.model`, and `modelinput`. Property value arguments can be specified at creation time. The property arguments are properties of `mbcdoe.design`.

Version History

Introduced in R2008a

See Also

`Generate` | `modelinput` | `mbcdoe.design` | `mbcmodel.testplan` | `mbcmodel.model`

Evaluate

Evaluate model, boundary model, or design constraint

Syntax

```
y = Evaluate(model,x)
y = Evaluate(c,x)
y = Evaluate(b,x)
```

Description

`y = Evaluate(model,x)` evaluates the model `model` at `X`.

`y = Evaluate(c,x)` evaluates the design constraint `c` at `x` (negative results are within the constraint).

`y = Evaluate(b,x)` evaluates the boundary model `b` at `x`. A positive value indicates that the point is outside the boundary. The method cannot evaluate a boundary model until it is fitted.

Input Arguments

`model` — Model object
`mbcmodel.model` object

Model being evaluated, specified as an `mbcmodel.model` object.

x — Data
array

Data where model is being evaluated, specified as an array of size *numpoints*-by-*nfactors*. *nfactors* is the number of inputs to the model specified using NumInputs and *numpoints* is the number of row in `x`.

For boundary model `b`, `x` is a matrix with `b.NumInputs` columns. All boundaries use the form `g(x)=0`.

c — Design constraint
`mbcdoe.designconstraint` object

Design constraint, specified as a `mbcdoe.designconstraint` object.

b — Boundary model
`mbcboundary.model` object

Boundary model, specified as a `mbcboundary.Model` object.

Output Arguments

y — Predicted value
array

Predicted value of the model at `x`, specified as an array of size (*numpoints*-by-1.

More About

Usage

This is a method of `mbcmodel.model`, `mbcdoe.designconstraint`, and boundary model object `mbcboundary.AbstractBoundary` and all its subclasses.

Version History

Introduced in R2007a

See Also

`PredictedValue` | `pev` | `mbcboundary.TwoStage`

fit

Fit model or boundary model to new or existing data, and provide summary statistics

Syntax

```
[outmodel,statistics] = fit(inmodel,X,Y)
[outmodel,statistics] = fit(inmodel)
```

Description

`[outmodel,statistics] = fit(inmodel,X,Y)` This fits the model to the specified data.

`[outmodel,statistics] = fit(inmodel)` refits the model if data for fit has already been supplied.

Examples

Calculate Fit of Knot

Calculate fit of a model using this syntax.

```
statistics = fit(knot)
statistics =
   27.0000    7.0000    1.0000    3.0184    2.6584
```

Input Arguments

`inmodel` — Model being fitted
`mbcmodel.model` object

Model being fitted, specified as an `mbcmodel.model` object.

X, Y — Data
matrix

Data being used to fit the model, specified as a matrix.

Output Arguments

`outmodel` — Fitted model
`mbcmodel.model` object

Fitted model, returned as an `mbcmodel.model` object.

`statistics` — Data
structure

Statistics of the fitted model, returned as a structure.

More About

Creation

This is a method of `mbcmodel.model` and `mbcboundary.Model`.

Version History

Introduced in R2007a

See Also

`SummaryStatistics` | `UpdateResponse`

InputSetupDialog

Open Input Setup dialog box to edit inputs

Syntax

```
[newmodel, OK] = InputSetupDialog(oldmodel)
[newtestplan, OK] = InputSetupDialog(oldtestplan)
```

Description

`[newmodel, OK] = InputSetupDialog(oldmodel)` opens the Input Setup dialog box, where you can edit the `oldmodel` model inputs: names, symbols, and ranges.

`[newtestplan, OK] = InputSetupDialog(oldtestplan)` opens the Input Setup dialog box, where you can edit the `oldtestplan` test plan inputs: names, symbols, and ranges.

Input Arguments

`oldmodel` — Input model
`mbcmodel.model` object

Input model that is being updated using Input Setup Dialog, specified as a `mbcmodel.model` object.

`oldtestplan` — Input test plan
`mbcmodel.testplan` object

Input test plan that is being updated using Input Setup Dialog, specified as a `mbcmodel.testplan` object.

`OK` — Changes to make in dialog box
`false` | `true`

Changes to make in dialog box, specified as either `false` or `true`.

- If you click **Cancel** to dismiss the dialog box, this argument is set to `OK = false` and `newmodel = oldmodel`.
- If you click **OK** to close the dialog box, this argument is `OK = true` and `newmodel` is your new chosen model setup. The new model is refitted when you click OK.

Output Arguments

`newmodel` — Output model
`mbcmodel.model` object

Output model with inputs set up using Input Setup Dialog, returned as a `mbcmodel.model` object.

`newtestplan` — Output test plan
`mbcmodel.testplan` object

Output test plan with inputs set up using Input Setup Dialog, returned as a `mbcmodel.testplan` object.

Version History

Introduced in R2007a

See Also

`CreateTestplan` | `mbcmodel.model` | `mbcmodel.testplan`

Jacobian

Calculate Jacobian matrix for model at existing or new data points

Syntax

```
J = Jacobian(model,x)
```

Description

`J = Jacobian(model,x)` calculates the Jacobian matrix for the model at existing or new data points `x`. If `x` is not specified then the existing data is used.

Input Arguments

`model` — Model input
`mbcmodel.model` object

Model whose Jacobian matrix is being computed, specified as a `mbcmodel.model` object.

x — New data points
matrix

New data points where the Jacobian of `model` is being computed, specified as a matrix.

Output Arguments

J — Jacobian
matrix

Jacobian of the matrix at designated data points, returned as a matrix. The Jacobian matrix (for linear and RBF models) is the same as the Regression Matrix in the GUI. These matrices only include the terms currently selected in the model.

If all terms are included (none removed by Stepwise) then the Jacobian (for linear and RBF models) is the same as the Full FX matrix found in the “Design Evaluation Tool” GUI. The Jacobian matrix only includes the currently selected model terms.

To determine the condition number, use the MATLAB command `cond(J)`.

Version History

Introduced before R2006a

See Also

ModelSetup

Open Model Setup dialog box where you can alter model type

Syntax

```
[newModel, OK] = ModelSetup(oldModel)
```

Description

`[newModel, OK] = ModelSetup(oldModel)` opens the **Model Setup** dialog box where you can choose new model types and settings.

Input Arguments

`oldModel` — Input model
`mbcmodel.model` object

Input model being set up, specified as a `mbcmodel.model` object.

`OK` — Changes to make in dialog box
`false` | `true`

Changes to make in the dialog box, specified as either `false` or `true`.

- If you click **Cancel** to dismiss the dialog, this argument is set to `OK = false` and `newModel = oldModel`.
- If you click **OK** to close the dialog box, then `OK = true` and `newModel` is your new chosen model setup. Data and response remain the same as `oldModel`.

The new model is refitted when you click OK.

Output Arguments

`newModel` — Output model
`mbcmodel.model` object

Out model with updated type and settings, returned as a `mbcmodel.model` object.

Call `UpdateResponse` to put the new model type back into the response.

Version History

Introduced in R2006a

See Also

`UpdateResponse` | `StatisticsDialog` | `fit`

pev

Predicted error variance of model at specified inputs

Syntax

```
p = pev(model,X)
```

Description

`p = pev(model,X)` calculates the Predicated Error Variance at `X`. If `X` is not specified, then PEV is calculated using the existing input values.

Input Arguments

`model` — Model object
`mbcmodel.model` object

Model whose Predicated Error Variance is being computed, specified as an `mbcmodel.model` object.

`X` — Input values
array

Input values where PEV of the model is evaluated, specified as an array. For a local response, the predicted value uses the hierarchical model.

Note For `mbcmodel.model` and `mbcmodel.response` objects input `X` is optional.

Output Arguments

p — Predicted error variance
array

Predicted error variance values, returned as an array.

More About

Usage

This is a method of the hierarchical, local response, response, and model objects: `mbcmodel.hierarchicalresponse`, `mbcmodel.response` and `mbcmodel.model`.

Version History

Introduced before R2006a

See Also

`PEVForTest` | `mbcdoe.design` | `mbcmodel.model`

PredictedValue

Predicted value of model at specified inputs

Syntax

`y = PredictedValue(model,x)`

Description

`y = PredictedValue(model,x)` evaluates the model `model`at `x`.

Examples

Compare Predicted Values

Compare predicted values of two models.

```
y = PredictedValue(R, X);
modelPred = PredictedValue(thisRF, x);
```

Input Arguments

`model` — Model object
`mbcmodel.model` object

Model being evaluated, specified as an `mbcmodel.model` object.

x — Input data
array

Input data where you want to evaluate the output of the model, specified as an array

Output Arguments

y — Predicted value
array

Predicted value of the model at the input data points `x`, returned as an array.

Note To evaluate model output for a local response or hierarchical response, you have to construct it using `MakeHierarchicalResponse` (or `CreateAlternativeModels`). If you have created alternative response feature models then a best model must be selected. If you have made changes such as removing outliers since choosing a model as best, you may need to choose a new best model. For a local response, the predicted value uses the hierarchical model. If no data is specified then the data from all tests is used.

More About

Usage

This is a method of the hierarchical, response, local response, and model objects: `mbcmodel.hierarchicalresponse`, `mbcmodel.response`, `mbcmodel.localresponse`, and `mbcmodel.model`.

Version History

Introduced before R2006a

See Also

`ChooseAsBest` | `pev` | `PredictedValueForTest`

StatisticsDialog

Open summary statistics dialog box

Syntax

```
[model_out,OK]= StatisticsDialog(model_in)
```

Description

`[model_out,OK]= StatisticsDialog(model_in)` opens the **Summary Statistics** dialog box, where you can select the summary statistics you want to use.

Input Arguments

`model_in` — Input model
`mbcmodel.model` object

Input model whose statistics are being used, specified as a `mbcmodel.model` object.

`OK` — Changes to make in dialog box
`false` | `true`

Changes to make in the dialog box, specified as either `false` or `true`.

- If you click **Cancel** to dismiss the dialog, this item is set to`OK = false` and no changes are made.
- If you click **OK** to close the dialog box, this item is set to `OK = true` and your new chosen summary statistics are set up.

Output Arguments

`model_out` — Output model
`mbcmodel.model` object

Out model whose statistics are being updated, returned as a `mbcmodel.model` object.

Version History

Introduced in R2007a

See Also

`SummaryStatistics`

SummaryStatistics

Summary statistics for response

Syntax

```
s = SummaryStatistics(model)
s = SummaryStatistics(model, Names)
```

Description

`s = SummaryStatistics(model)` returns summary statistics for the model or response.

`s = SummaryStatistics(model, Names)` returns summary statistics specified by `Names` for the model or response in an array.

Input Arguments

`model` — Model
`mbcmodel.model` object | `mbcmodel.response` object

Model whose summary statistics is being displayed, specified as a `mbcmodel.model` or `mbcmodel.response` object.

`Names` — Names of model or response
array

Names of `mbcmodel.model` or `mbcmodel.response` objects, specified as a char array or a cell array of character vectors..

Output Arguments

`s` — Summary of statistics
structure

Summary of statistics, returned as a structure with fields `Statistics` and `Names`.

More About

Usage

This is a method of all model objects (`mbcmodel.model` and `mbcmodel.linearmodel`) and response objects (`mbcmodel.hierarchicalresponse`, `mbcmodel.localresponse`, and `mbcmodel.response`).

These statistics appear in the Summary Statistics pane of the Model Browser GUI.

Version History

Introduced before R2006a

See Also

`DiagnosticStatistics` | `AlternativeModelStatistics`

ValidationRMSE

Calculates the validation RMSE for model data

Syntax

```
s = ValidationRMSE(model,X,Y)
```

Description

`s = ValidationRMSE(model,X,Y)` calculates the root mean square error (RMSE) of a validation set.

Input Arguments

`model` — Model object
`mbcmodel.model` object

Model whose root mean square error is being computed, specified as a `mbcmodel.model` object.

`X`, `Y` — Validation data
`table` object | numeric array

Validation data to calculate RMSE, specified as a `table` object or numeric array.

Output Arguments

`s` — RMSE error
array

Root mean square error of the model compared to the validation set.

More About

Root Mean Square Error

`rmse = sqrt(sum((Y - evaluate(model,X)).^2)/N)`, where `N` is the number of data points.

Version History

Introduced in R2019a

See Also

`SummaryStatistics` | `UpdateResponse`

MBCModel.Data

mbcmodel.data

Properties and methods for data objects

Description

Use these properties and object functions to create and examine data objects.

Creation

Create a mbcmodel.data object using `CreateData`.

Properties

`Name` — Data object name
character vector

Name of the data object.

Example: `'holliday_data.mat'`

Data Types: `char` | `string`

`NumRecords` — Total number of records in data object
scalar

This property is read-only.

Total number of records in data object.

Example: `270`

Data Types: `integer`

`NumSignals` — Number of signals contained in each record
scalar

This property is read-only.

Number of signals contained in each record.

Example: `7`

Data Types: `integer`

`NumTests` — Total number of tests used in model
scalar

This property is read-only.

Total number of tests used in model.

Example: `27`

Data Types: `double`

`RecordsPerTest` — Number of records in each test
1-by-n array

This property is read-only.

Number of records in each test, where *n* is the number of tests.

Example: `[10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10]`

Data Types: `double`

`IsEditable` — Boolean indicating if data is editable
`true` or `1` | `false` or `0`

This property is read-only.

Boolean indicating if data is editable. The following rules apply:

- If the data was created using `mbcmodel.CreateData` and is not `Attached` to a test plan it is editable.
- If the data was created or retrieved from the project and was not `Attached` to a test plan, it is editable.
- If the data was `Attached` to a test plan and was subsequently retrieved from that test plan, it is editable.

Data Types: `logical`

`IsBeingEdited` — Boolean indicating if data or model is being edited
`true` or `1` | `false` or `0`

This property is read-only.

Boolean indicating if data or model is being edited.

Example: 0

Data Types: `logical`

`Owner` — `mbcmodel.project` object containing data
`0`-by-1 array

This property is read-only.

`mbcmodel.project` or `mbcmodel.testplan` object containing the data. The array is:

- Empty if `mbcmodel.CreateData` created the data.
- An `mbcmodel.project` object if a project contained the data.
- An `mbcmodel.testplan` object if a test plan contained the data.

Data Types: `function_handle`

`SignalNames` — Signal names in data
n-by-1 array

This property is read-only.

n-by-1 array of character vectors that contains the data signal names, where *n* is the number of signals.

Example: `["afr" "egr" "load" "n" "spark" "logno" "tq"]`

Data Types: `string`

`SignalUnits` — Signal units in data
n-by-1 array

This property is read-only.

n-by-1 array of character vectors that contains the data signal units, where *n* is the number of signals.

Example: `[ "%" "%" "ratio" "rpm" "deg" "none" "ft lbf"]`

Data Types: `string`

`Filters` — Structure array containing user-defined filters
array

This property is read-only.

Structure array holding user-defined filters. The array is the same length as the number of currently defined filters, with the following fields for each filter:

- `Expression` — The character vector expression as defined in `AddFilter` or `ModifyFilter`.
- `AppliedOK` — Boolean indicating that the filter was successfully applied.
- `RemovedRecords` — Boolean vector indicating which records the filter removed. Note that many filters could remove the same record.
- `Message` — Character vector holding information on the success or otherwise of the filter.

See also `AddFilter`, `ModifyFilter`, and `RemoveFilter`.

Data Types: `struct`

`TestFilters` — Structure array containing user-defined test filters
array

This property is read-only.

Structure array holding user-defined test filters. The array is the same length as the number of currently defined test filters, with the following fields for each filter:

- `Expression` — The character vector expression as defined in `AddTestFilter` or `ModifyTestFilter`.
- `AppliedOK` — Boolean indicating that the filter was successfully applied.
- `RemovedTests` — Boolean vector indicating which records the filter removed. Note that many filters could remove the same test.
- `Message` — Character vector holding information on the success or otherwise of the test filter.

See also `AddTestFilter`, `ModifyTestFilter`, and `RemoveTestFilter`.

Data Types: `struct`

`UserVariables` — Structure array holding user-defined variables
array

This property is read-only.

Structure array holding user-defined variables. The array is the same length as the number of currently defined variables, with the following fields for each filter:

- `Variable` — Variable Name
  - `Expression` — The character vector expression as defined in `AddVariable` or `ModifyVariable`.
  - `Units` — The character vector defining the units.
  - `AppliedOK` — Boolean indicating that the variable expression was successfully applied.
  - `Message` — Character vector holding information on the success or otherwise of the variable.

See also `AddVariable`, `ModifyVariable`, and `RemoveVariable`.

Data Types: `struct`

Object Functions

AddFilter	Add filter to data set
AddTestFilter	Add test filter to data set
AddVariable	Add variable to data set
Append	Append data to data set
BeginEdit	Begin editing a data object
CreateData	Create data object
CommitEdit	Apply changes in data
DefineNumberOfRecordsPerTest	Define exact number of records per test
DefineTestGroups	Define rule-based test groupings
ExportToTable	Export data to table object
ExportToMBCDataStructure	Export data to MBC data structure
ImportFromFile	Import data from file
ImportFromMBCDataStructure	Load data from MBC data structure
ImportFromTable	Load data from a table object
ModifyFilter	Modify filter in data set
ModifyTestFilter	Modify test filter in data set
ModifyVariable	Modify variable in data set
RemoveFilter	Remove filter from data set
RemoveTestFilter	Remove test filter from data set
RemoveVariable	Remove variable from data set
RollbackEdit	Undo most recent changes to data
Value	Extract data values from data object

Examples

Create a Data Object

This example shows how to create a new data object that is independent of a project. `filename` is a character vector specifying the full path to the file. To use the data object in another test plan, use `AttachData`.

```
DataObj = mbcmodel.CreateData(filename);
testplan.AttachData(DataObj);
```

Create a Data Object in a Project Object

This example shows how to create a data object in an existing project object `ProjObj`.

```
data = CreateData(ProjObj,'D:\MBCWork\data1.xls');
DataObj = mbcmodel.CreateData;
DataObj = mbcmodel.CreateData('D:\MBCWork\data.xls');
```

Version History

Introduced before R2006a

See Also

`CreateData`

Topics

"Load and Modify Data"

CreateData

Create data object

Syntax

```
DataObj = CreateData(ProjObj)
DataObj = CreateData(ProjObj,Filename)
DataObj = CreateData(ProjObj,Table)
DataObj = CreateData(ProjObj,mbcStruct)
DataObj = CreateData(ProjObj,Filename,Filetype)
```

Description

`DataObj = CreateData(ProjObj)` creates a data object in a project object *`ProjObj`*.

`DataObj = CreateData(ProjObj,Filename)` creates a data object in a project object *`ProjObj`*. The data is in a file, *`filename`*, specified as a character vector containing the full path to the file.

`DataObj = CreateData(ProjObj,Table)` creates a data object in a project object *`ProjObj`*. The data is contained in a table object, *`Table`*.

`DataObj = CreateData(ProjObj,mbcStruct)` creates a data object for an MBC data structure in a project object *`ProjObj`*. *`mbcStruct`* is the MBC data structure name.

`DataObj = CreateData(ProjObj,Filename,Filetype)` creates a data object in a project object *`ProjObj`*. The data is in a file, *`Filename`*, specified as a character vector containing the full path to the file. *`Filetype`* is a character vector specifying the file type.

Examples

Create Data Object in Project Object

Create data object in a project object from file.

```
data = CreateData(ProjObj, 'D:\MBCWork\data1.xls');
D = mbcmodel.CreateData;
D = mbcmodel.CreateData('D:\MBCWork\data.xls');
```

`ProjObj` is an `mbcmodel.project` object.

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`Filename` — Data file path
character vector

Project file path, specified as a character vector.

If you do not specify a `Filename`, no data is loaded into the new data object. Load data using `ImportFromFile`, provided that you enable editing of the data object by using `BeginEdit`. Call `CommitEdit` to apply edits.

If you create the data object specifying a `filename`, then the Name property is set to the filename. However, if you use `ImportFromFile` after creation to load data from a file, the name of the data object does not change.

Data Types: `char` | `string`

`Table` — Table object
table object

Table object.

Data Types: `char` | `string`

`mbcStruct` — MBC data structure
`mbcmodel.data` object

An MBC data structure is a structure array that contains these fields:

- `varNames` — Cell array of character vectors that hold the names of the variables in the data (`1xn` or `nx1`).
- `varUnits` — Cell array of character vectors that hold the units associated with the variables in `varNames` (`1xn` or `nx1`). If array is empty, no units are defined.
- `data` — Array that holds the values of the variables (`mxn`).
- `comment` — Optional character vector holding comment information about the data.

`Filetype` — Data file path
character vector

Character vector specifying the file type. See `DataFileTypes` for the specification of allowed file types. If `filetype` is not provided, the software infers the file type from the file extension.

Data Types: `char` | `string`

Output Arguments

`ProjObj` — Project object
project object

Output project object created by `CreateData`, returned as a design object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `mbcmodel.project` | `DataFileTypes`

AddFilter

Add filter to data set

Syntax

`DataObjMod = AddFilter(DataObj,Expr)`

Description

`DataObjMod = AddFilter(DataObj,Expr)` adds a filter to the dataset that exclude some records. Define the filter using logical operators or a logical function on the existing variables.

Examples

Add Filter

Add a filter that keeps records when `AFR < AFR_CALC +10`.

```
DataObjMod = AddFilter(DataObj, 'AFR < AFR_CALC + 10');
```

Add a filter that uses the function `MyFilterFunction`. The function uses the variables *AFR*, *RPM*, *TQ*, and *SPK*.

```
DataObjMod = AddFilter(DataObj, 'MyFilterFunction(AFR, RPM, TQ, SPK)');
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`Expr` — Expression
character vector

Input character vector containing the expression. To define the expression, use logical operators or a logical function on the existing variables.

Example: `'AFR < AFR_CALC + 10'`

Data Types: `char`

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `AddTestFilter` | `ModifyFilter` | `ModifyTestFilter` | `RemoveFilter`

AddTestFilter

Add test filter to data set

Syntax

`DataObjMod = AddTestFilter(DataObj,Expr)`

Description

`DataObjMod = AddTestFilter(DataObj,Expr)` adds a test filter to the dataset that excludes some tests. Define the filter using logical operators or a logical function on the existing variables.

Examples

Add Test Filter

Include all tests in which all records have speed, *n*, greater than 1000.

```
DataObjMod = AddTestFilter(DataObj, 'any(n>1000)');
```

Include all tests with more than 6 records.

```
DataObjMod = AddTestFilter(DataObj, 'length(LOGNO) > 6');
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`Expr` — Expression
character vector

Input character vector containing the expression. To define the expression, use logical operators or a logical function on the existing variables.

Example: `'AFR < AFR_CALC + 10'`

Data Types: `char`

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `AddFilter` | `RemoveTestFilter` | `ModifyTestFilter`

AddVariable

Add variable to data set

Syntax

```
DataObjMod = AddVariable(DataObj,Expr,Units)
```

Description

`DataObjMod = AddVariable(DataObj,Expr,Units)` adds a variable to the data set. Variable names are case sensitive.

Examples

Add New Variable

Add a variable, *MY_NEW_VARIABLE*, that depends on *TQ* and *AFR*.

```
DataObjMod = AddVariable(DataObj,'MY_NEW_VARIABLE = TQ*AFR/2');
```

Add a variable that uses the function `MyVariableFunction`. The function uses the variables *TQ*, *AFR*, and *RPM*.

```
DataObjMod = AddVariable(DataObj,'funcVar = MyVariableFunction(TQ, AFR, RPM)',
'lb');
```

Add a variable if the signal names in data do not match the model input factors in the test plan template file.

```
DataObjMod = AddVariable(DataObj,'TQ=tq');
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`Expr` — Expression
character vector

Input character vector containing the expression. To define the expression, use logical operators or a logical function on the existing variables.

Example: `'AFR < AFR_CALC + 10'`

Data Types: `char`

`Units` — Units, optional
character vector

Input character vector containing the expression that defines the variable units.

Example: `'lb'`

Data Types: `char`

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `ModifyVariable` | `RemoveVariable`

Append

Append data to data set

Syntax

`DataObjMod = Append(DataObj,otherData)`

Description

`DataObjMod = Append(DataObj,otherData)` adds new data to an existing data set.

Examples

Append Data

```
DataObjMod = Append(DataObj,CreateData('aDataFile.xls'));
DataObjMod = Append(DataObj,rand(10,100));
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`otherData` — Expression
`mbcmodel.data` object | `array`

If `otherData` is an `mbcmodel.data` object, then `Append` looks for common `SignalNames` between the two sets of data. If the method does not find common `SignalNames`, the method throws an error. The method appends any common signals to the existing data and fills other signals with `NAN`.

If `otherData` is an array, then it must have exactly the same number of columns as there are `SignalNames` in the data. The method applies `vertcat` (vertical concatenation) between the existing data and `otherData`.

Example: `rand(10,100)`

Data Types: `function_handle` | `double`

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`CreateData` | `mbcmodel.data`

BeginEdit

Begin editing a data object

Syntax

`DataObjMod = BeginEdit(DataObj)`

Description

`DataObjMod = BeginEdit(DataObj)` allows you to edit a data object. You must call `BeginEdit` before to modifying a data object. `BeginEdit` throws an error if you cannot edit the data.

You can use `BeginEdit` if the `DataObj` property `Editable` is true. Otherwise, `BeginEdit` throws an error.

Examples

Begin Editing

This example shows you how to use `BeginEdit`. Use `CommitEdit` to commit edited data to data object, *`DataObj`*, in a project object, *`ProjObj`*.

```
DataObj = ProjObj.Data;
BeginEdit(DataObj);
AddVariable(DataObj, 'TQ = tq', 'lbft');
AddFilter(DataObj, 'TQ < 200');
DefineTestGroups(DataObj, {'RPM' 'AFR'}, [50 10], 'MyLogNo');
CommitEdit(DataObj);
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

mbcmodel.data | CommitEdit | AddVariable | AddFilter | DefineTestGroups

CommitEdit

Apply changes in data

Syntax

`DataObjMod = CommitEdit(DataObj)`

Description

`DataObjMod = CommitEdit(DataObj)` applies data changes to a data object. For example, use `CommitEdit` after you create new variables or apply filters to remove unwanted data.

You can use `CommitEdit` if the `DataObj` properties `Editable` and `IsBeingEdited` are both true. Otherwise, `CommitEdit` throws an error.

Examples

Commit Edited Data

This example shows you how to commit edited data to data object, *`DataObj`*, in a project object, *`ProjObj`*.

```
DataObj = ProjObj.Data;
BeginEdit(DataObj);
AddVariable(DataObj, 'TQ = tq', 'lbft');
AddFilter(DataObj, 'TQ < 200');
DefineTestGroups(DataObj, {'RPM' 'AFR'}, [50 10], 'MyLogNo');
CommitEdit(DataObj);
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `BeginEdit` | `AddVariable` | `AddFilter` | `DefineTestGroups`

DataFileTypes

Data file types

Syntax

```
DataTypes = DataFileTypes(ModelObj)
```

Description

`DataTypes = DataFileTypes(ModelObj)` returns a list of data file types for *`ModelObj`*.

Examples

Return Data File Types

Return a list of data file types for model object `mbcmodel`.

```
f = mbcmodel.DataFileTypes
```

```
f =

  Columns 1 through 4
    'Excel file'    'FT/DB data files'    'Delimited Text File'
  [1x25 char]
  Column 5
    'MATLAB Data File'
```

Input Arguments

`ModelObj` — Model object
`mbcmodel.model` object

Model object.

Output Arguments

`DataTypes` — Data file types
character vector

Character vector specifying the data file type.

Data Types: `char` | `string`

Version History

Introduced in R2007a

See Also

`mbcmodel.data` | `ImportFromFile` | `CreateData` | `CreateModel`

DefineNumberOfRecordsPerTest

Define exact number of records per test

Syntax

```
DataObjMod = DefineNumberOfRecordsPerTest(DataObj,Number,TestNumAlias)
```

Description

`DataObjMod = DefineNumberOfRecordsPerTest(DataObj,Number,TestNumAlias)` defines the number of records per test. Use `DefineNumberOfRecordsPerTest` to set one test per record for one-stage modeling.

Examples

Define Number of Records Per Test

```
DataObjMod = DefineNumberOfRecordsPerTest(DataObj, 1);
DataObjMod = DefineNumberOfRecordsPerTest(DataObj, 10, 'MYLOGNO');
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`Number` — Number of records
`scalar`

Number of records to include in each test. Usually, this is one test per record.

Example: `1`

Data Types: `double`

`TestNumAlias` — Test number alias, optional
character vector

Optional character vector input to define the `SignalName` that the software should use as the test number. Defaults to the test index.

Note For the test number, `testnumAlias` uses the first record in the test. Test numbers are unique, so the method does modify duplicates.

Data Types: `char`

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `DefineTestGroups`

DefineTestGroups

Define rule-based test groupings

Syntax

```
DataObjMod = DefineTestGroups(DataObj,Variables,Tolerances,TestNumAlias,
Reorder)
```

Description

`DataObjMod = DefineTestGroups(DataObj,Variables,Tolerances,TestNumAlias, Reorder)` defines a rule-based test group. You can impose rules to sort records of `DataObj` into groups. The groups are tests. Two-stage models use the test groupings to define hierarchical structure in the data.

Use `DefineTestGroups` to set variables to group. The method uses the tolerance to define groups. When the value of any specified variable changes by more than the tolerance, the method defines a new group.

Examples

Define Test Groups

```
DataObjMod = DefineTestGroups(DataObj, {'AFR' 'RPM'}, [0.1 30], 'MYLOGNO', false);
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`Variables` — Variables
character vector

Character vector containing the `SignalNames` that define the test groupings.

Data Types: `char`

`Tolerances` — Variable tolerances
array

Array containing the tolerances for the test grouping definition. Same length as Variables vector.

Data Types: `double`

`TestNumAlias` — Test number alias, optional
character vector

Optional character vector input to define the `SignalName` that the software should use as the test number. Defaults to the test index.

Note For the test number, `testnumAlias` uses the first record in the test. Test numbers are unique, so the method does modify duplicates.

Data Types: `char`

`Reorder` — Boolean to reorder
`false`or `0` (default) | `true` or `1`

Set to `true` to reorder data.

Data Types: `logical`

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `DefineNumberOfRecordsPerTest`

ExportToMBCDataStructure

Export data to MBC data structure

Syntax

```
mbcStruct = ExportToMBCDataStructure(DataObj)
```

Description

`mbcStruct = ExportToMBCDataStructure(DataObj)` converts the data object to the MBC data structure format.

Examples

Export Data to MBC Data Structure

```
mbcStruct = ExportToMBCDataStructure(DataObj);
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

Output Arguments

`mbcStruct` — MBC data structure
`mbcmodel.data` object

An MBC data structure is a structure array that contains these fields:

- `varNames` — Cell array of character vectors that hold the names of the variables in the data (`1xn` or `nx1`).
- `varUnits` — Cell array of character vectors that hold the units associated with the variables in `varNames` (`1xn` or `nx1`). If array is empty, no units are defined.
- `data` — Array that holds the values of the variables (`mxn`).
- `comment` — Optional character vector holding comment information about the data.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `ImportFromMBCDataStructure`

ImportFromFile

Import data from file

Syntax

`DataObjMod = ImportFromFile(DataObj,filename,filetype,sheetname)`

Description

`DataObjMod = ImportFromFile(DataObj,filename,filetype,sheetname)` imports data on `sheetname` from `filename` of `filetype` to your data object.

Before using the method, use `CreateData` and `BeginEdit` so that you can add data to the data object.

Examples

Import Data to Data Object

```
DataObjMod = ImportFromFile(DataObj, filename, filetype)
DataObjMod = ImportFromFile(DataObj, filename, 'Excel file', SHEETNAME)
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`filename` — File name
character vector

Input character vector containing the path and name of the file.

Example: `'D:\MBCData\Raw Data\testdata.xls'`

Data Types: `char`

`filetype` — File type, optional
character vector

Input character vector containing the file type. For allowed file types, see `DataFileTypes`.

If you do not enter a `filetype`, the method uses the file extension to provide the filetype. If the file extension is `.xls`, then the method uses Excel.

Example: `'Excel file'`

Data Types: `char`

`sheetname` — Sheet name, optional
character vector

Input character vector containing the sheet name.

Example: `'lb'`

Data Types: `char`

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `Append` | `BeginEdit` | `CreateData` | `ImportFromMBCDataStructure`

ImportFromMBCDataStructure

Load data from MBC data structure

Syntax

```
DataObjMod = ImportFromMBCDataStructure(DataObj,mbcStruct)
```

Description

`DataObjMod = ImportFromMBCDataStructure(DataObj,mbcStruct)` imports an MBC data structure to your `DataObj`.

Before using the method, use CreateData and BeginEdit so that you can add data to the data object.

Examples

Import from MBC Data Structure

```
DataObjMod = ImportFromMBCDataStructure(DataObj,mbcStruct);
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`mbcStruct` — MBC data structure
`mbcmodel.data` object

An MBC data structure is a structure array that contains these fields:

- `varNames` — Cell array of character vectors that hold the names of the variables in the data (`1xn` or `nx1`).
- `varUnits` — Cell array of character vectors that hold the units associated with the variables in `varNames` (`1xn` or `nx1`). If array is empty, no units are defined.
- `data` — Array that holds the values of the variables (`mxn`).
- `comment` — Optional character vector holding comment information about the data.

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `Append` | `BeginEdit` | `CreateData` | `ExportToMBCDataStructure`

ModifyFilter

Modify filter in data set

Syntax

```
DataObjMod = ModifyFilter(DataObj,Index,Expr)
```

Description

`DataObjMod = ModifyFilter(DataObj,Index,Expr)` modifies a dataset filter. Define the filter using logical operators or a logical function on the existing variables.

Examples

Modify Filter

Modify filter number 3 to keep all records where `AFR < AFR_CALC + 20`.

```
DataObjMod = ModifyFilter(DataObj, 3, 'AFR < AFR_CALC + 20');
```

Modify filter number 2 to apply the function `MyNewFilterFunction`.

```
ModifyFilter(DataObj, 2, 'MyNewFilterFunction(AFR, RPM, TQ, SPK)');
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`Index` — Input index
scalar

Input index to indicate which of the available filters you want to modify. Use the `mbcmodel.data` object `Filters` property to find the index for each filter.

Example: 2

Data Types: `int`

`Expr` — Expression
character vector

Input character vector containing the expression. To define the expression, use logical operators or a logical function on the existing variables.

Example: `'AFR < AFR_CALC + 10'`

Data Types: `char`

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `AddFilter` | `RemoveFilter`

ModifyTestFilter

Modify test filter in data set

Syntax

`DataObjMod = ModifyTestFilter(DataObj,Index,Expr)`

Description

`DataObjMod = ModifyTestFilter(DataObj,Index,Expr)` modifies a dataset test filter. Define the test filter using logical operators or a logical function on the existing variables.

Examples

Modify Test Filter

Modify test filter number 2 to include all tests in which any records have speed, *n*, greater than 2000.

```
DataObjMod = ModifyTestFilter(DataObj, 2, 'any(n>2000)');
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`Index` — Input index
scalar

Input index to indicate which of the available filters you want to modify. Use the `mbcmodel.data` object `Filters` property to find the index for each filter.

Example: 2

Data Types: `int`

`Expr` — Expression
character vector

Input character vector containing the expression. To define the expression, use logical operators or a logical function on the existing variables.

Example: `'AFR < AFR_CALC + 10'`

Data Types: `char`

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `AddTestFilter` | `RemoveTestFilter`

ModifyVariable

Modify variable in data set

Syntax

```
DataObjMod = ModifyVariable(DataObj,Index,Expr,Units)
```

Description

`DataObjMod = ModifyVariable(DataObj,Index,Expr,Units)` modifies a variable in the data set. Variable names are case sensitive.

Examples

Modify Variable

```
DataObjMod = ModifyVariable(DataObj, 2, 'MY_NEW_VARIABLE = TQ*AFR/2');
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`Index` — Input index
scalar

Input index to indicate which of the available filters you want to modify. Use the `mbcmodel.data` object `Filters` property to find the index for each filter.

Example: 2

Data Types: `int`

`Expr` — Expression
character vector

Input character vector containing the expression. To define the expression, use logical operators or a logical function on the existing variables.

Example: `'AFR < AFR_CALC + 10'`

Data Types: `char`

`Units` — Units, optional
character vector

Input character vector containing the expression that defines the variable units.

Example: `'lb'`

Data Types: `char`

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History
Introduced before R2006a

See Also
`mbcmodel.data` | `AddVariable` | `RemoveVariable`

RemoveFilter

Remove filter from data set

Syntax

`DataObjMod = RemoveFilter(DataObj,Index)`

Description

`DataObjMod = RemoveFilter(DataObj,Index)` removes a dataset filter.

Examples

Remove Filter

Remove filter number 3.

```
RemoveFilter(DataObj,3);
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`Index` — Input index
scalar

Input index to indicate which of the available filters you want to modify. Use the `mbcmodel.data` object `Filters` property to find the index for each filter.

Example: 2

Data Types: `int`

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `AddFilter`

RemoveTestFilter

Remove test filter from data set

Syntax

`DataObjMod = RemoveTestFilter(DataObj,Index)`

Description

`DataObjMod = RemoveTestFilter(DataObj,Index)` removes a dataset test filter.

Examples

Remove Test Filter

Remove test filter number 2.

```
RemoveTestFilter(DataObj,2);
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`Index` — Input index
scalar

Input index to indicate which of the available filters you want to modify. Use the `mbcmodel.data` object `Filters` property to find the index for each filter.

Example: 2

Data Types: `int`

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `AddTestFilter` | `ModifyTestFilter`

RemoveVariable

Remove variable from data set

Syntax

```
DataObjMod = RemoveVariable(DataObj,Index)
```

Description

`DataObjMod = RemoveVariable(DataObj,Index)` removes a variable from a dataset.

Examples

Remove Variable

Remove variable number 2.

```
RemoveVariable(DataObj,2);
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`Index` — Input index
scalar

Input index to indicate which of the available filters you want to modify. Use the `mbcmodel.data` object `Filters` property to find the index for each filter.

Example: 2

Data Types: `int`

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `AddVariable` | `ModifyVariable`

RollbackEdit

Undo most recent changes to data

Syntax

```
DataObjMod = RollbackEdit(DataObj)
```

Description

`DataObjMod = RollbackEdit(DataObj)` reverts changes that you made to the data since you called `BeginEdit`. For example, use `RollbackEdit` to revert edits you made importing or appending data, applying filters or creating new user variables.

Examples

Revert Edits

This example shows how to revert edits for a data object in a project object *`ProjObj`*.

```
DataObj = ProjObj.Data;
BeginEdit(DataObj);
AddVariable(DataObj, 'TQ = tq', 'lbft');
AddFilter(DataObj, 'TQ < 200');
DefineTestGroups(DataObj, {'RPM' 'AFR'}, [50 10], 'MyLogNo');
RollbackEdit(DataObj);
```

This example shows how to revert edits when `IsEditable` is `false` and `IsBeingEdited` is `true`. *`ProjObj`* is a `mbcmodel.project` object, `d` and `d1` are `mbcmodel.data` objects.

```
D = ProjObj.Data;
D1 = ProjObj.Data;
BeginEdit(D1);
tp = ProjObj.Testplan;
Attach(tp, D);
```

This example shows how to revert edits when `IsEditable` for `D1` is `false` because it is `Attached` to the test plan. To modify `D1`, use the test plan. However:

```
OK = D1.IsBeingEdited
```

Is `true`. Calling `CommitEdit` will fail.

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced before R2006a

See Also

`mbcmodel.data` | `BeginEdit` | `CommitEdit`

Value

Extract data values from data object

Syntax

`val = Value(DataObj,varNames,testNumbers)`

Description

`val = Value(DataObj,varNames,testNumbers)` extracts data values from the data object.

Examples

Extract Data Values

Extract values from a data object.

```
val = Value(DataObj,'SPK',1);
val = Value(DataObj,{'SPK' 'AFR'},':');
val = Value(DataObj,[1 3 4 5]);
val = Value(DataObj,':',[1 4 6 8]);
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`varNames` — Variable names, optional
array

Optional input that specifies either the name of the signal that you want to extract or an array of names. Defaults to `':'`.

Example: `'SPK'`

Example: `{'SPK' 'AFR' 'TQ'}`

Example: `':'`

Data Types: `char`

`testNumbers` — Test numbers, optional
scalar | vector

Optional input that specifies which test indices you want. Defaults to `':'`.

Example: `1`

Example: `[1 3 4 5]`

Example: `':'`

Data Types: `uint`

Output Arguments

`val` — Data values
vector

Extracted data object values.

Data Types: `double`

Version History

Introduced before R2006a

See Also

`mbcmodel.data`

ExportToTable

Export data to table object

Syntax

`TableObj = ExportToTable(DataObj)`

Description

`TableObj = ExportToTable(DataObj)` exports data to a table object.

Examples

Export Data to Table Object

```
TableObj = ExportToTable(DataObj);
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

Output Arguments

`TableObj` — Table object
`table` object

Table object.

Version History

Introduced in R2019a

See Also

`mbcmodel.data` | `CreateData` | `ImportFromTable`

ImportFromTable

Load data from a table object

Syntax

`DataObjMod = ImportFromTable(DataObj,TableObj)`

Description

`DataObjMod = ImportFromTable(DataObj,TableObj)` loads data from a table object.

Examples

Import Data from Table Object

```
DataObjMod = ImportFromTable(DataObj,TableObj);
```

Input Arguments

`DataObj` — Instance of `mbcmodel.data` class
`mbcmodel.data` object

`mbcmodel.data` data object.

`TableObj` — Table object
`table` object

Table object.

Output Arguments

`DataObjMod` — Modified instance of `mbcmodel.data` class
`mbcmodel.data` object

Modified `mbcmodel.data` object.

Version History

Introduced in R2019a

See Also

`mbcmodel.data` | `ExportToTable` | `CreateData`

MBCdoe.Design

mbcdoe.design

Properties and methods for design of experiment (doe) design objects

Description

Use these properties and object functions to create and examine doe design objects.

Creation

Create a mbcdoe.design object using `CreateDesign`.

Properties

`Constraints` — `mbcdoe.designconstraint` object containing constraints
1-by-n array

`mbcdoe.designconstraint` object containing one or more constraints, specified as a 1-by-n array.

`Generator` — `mbcdoe.generator` object containing new design type parameters
1-by-n array

`mbcdoe.generator` object containing new design type parameters based on the new design generator, specified as a 1-by-n array. Design generators provide the properties for all the design types.

The properties you can set depend on the design `Type`. To view the properties for generating designs, see `Properties (for design generators)`.

Use `getAlternativeTypes` to get a list of alternative generators.

`Inputs` — Design inputs
1-by-n array

This property is read-only.

Design inputs, specified as a 1-by-n array of `mbcmodel.modelinput` objects. For `mbcdoe.design`, `D.Inputs = NewInputs` updates the inputs. You cannot change the number of design inputs. Many designs have `Limits` properties in addition to model input ranges. These properties allow you to restrict the range of the design without changing the model or losing points via a constraint.

Data Types: `integer`

`Model` — `mbcmodel.model` or `mbcmodel.linearmodel` object containing the design model
1-by-n array

`mbcmodel.model` or `mbcmodel.linearmodel` object containing the design model, specified as a 1-by-n array.

`D.Model = NewModel` changes the model for the design to `NewModel`.

Setting this property changes optimal designs to `custom` if the new model does not support optimal designs.

Name — Design object or design constraint object name
character vector

Design object or design constraint object name, specified as a character vector.

Data Types: `char` | `string`

`NumInputs` — Number of design object inputs
scalar

This property is read-only.

Number of design object inputs, specified as a scalar.

Data Types: `integer`

`NumPoints` — Number of design points
scalar

This property is read-only.

Number of points in the design after applying the constraints, specified as a scalar.

You can specify the number of points for a design using the generator object. The `NumberOfPoints` property of `mbcdoe.generator` is the number of points before any constraints are applied. You cannot specify the number of points for all design types (e.g., it is not allowed for Central Composite, Box Behnken). To see which design types have an editable `NumberOfPoints` property, see the tables in `Type (for designs and generators)`.

Data Types: `integer`

`Points` — Design points
array

Design points, specified as an array. You can perform any valid MATLAB operation on this matrix. The number of columns of the points matrix must be the same as the number of inputs when setting `Points`. If you make an assignment to the `Points`, the design type changes to `Custom`. Points are only updated in the underlying design if they have changed.

Data Types: `double`

`PointTypes` — Fixed and free point status
array

Fixed and free point status, specified as an array. Each point has a type of `free`, `fixed` or `data`.

You can specify `fixed` points. `free` is the default. If a point is matched to data, then it is of type `data`.

`D.PointTypes` returns a cell array of PointTypes, one for each design point. You cannot change a PointType of `data` to something else as the `data` is set by the test plan when matching the design to `data`.

To fix all the points in a design, use the `FixPoints` method

Data Types: `cell`

`Style` — Design type style
`n`-by-`1` array

This property is read-only.

Design type style, specified as a `n`-by-`1` array of character vectors that contains the design type style, where *n* is the number of designs.

The style of the design style is one of the following:

- 'User-defined'
- 'Optimal'
- 'Space-filling'
- 'Classical'
- 'Experimental data'

Data Types: `char` | `string`

Type — Design type
array

This property is read-only.

Design type, specified as an array. To set the property, use the `mbcdoe.generator` object.

`D.Type` returns the design type. You can only choose a type when you create designs. You can only set the `Type` of a `mbcdoe.generator` object after design creation, or when calling `Generate` or `Augment`.

`G.Type = NewType` changes the Type, where `G` is a `mbcdoe.generator` object.

The design type determines which properties you can set. To set properties, see `Properties (for design generators)`.

To get a list of types to use as alternative designs for the current design using `getAlternativeTypes`, enter this command. `D` is an `mbcdoe.design` object.

```
Dlist = getAlternativeTypes(D)
```

To use the alternative designs, the design Type must be one shown in the following table. The Type property determines the Style property.

Style	Type
Optimal	D-Optimal
	V-Optimal
	A-Optimal
Classical	Box-Behnken
	Central Composite
	Full Factorial

Style	Type
	Plackett-Burman
	Regular Simplex
Space-filling	Lattice
	Latin Hypercube Sampling
	Stratified Latin Hypercube
	Sobol Sequence
	Halton Sequence
Experimental data	Design points replaced by data points
Custom	Any design with a mix of Types (eg an optimally augmented space-filling design)

To specify the Type while creating and then generating a design of a given size:

```
D = CreateDesign(model,'Type','Sobol Sequence')
D = Generate(D,128);
```

Data Types: `char` | `string`

Object Functions

AddConstraint	Add design constraint
Augment	Add design points
ConstrainedGenerate	Generate constrained space-filling design of specified size
CreateCandidateSet	Create candidate set for optimal designs
CreateConstraint	Create design constraint
Discrepancy	Discrepancy value
FixPoints	Fix design points
Generate	Generate design points
getAlternativeTypes	Alternative model or design types
Maximin	Maximum of minimum of distance between design points
Merge	Merge designs
Minimax	Minimum of maximum distance between design points
OptimalCriteria	Optimal design criteria
RemovePoints	Remove all nonfixed design points
Scatter2D	Plot design points

Examples

Create Space Filling Design

To create a space-filling design for a test plan `TP`.

```
sfDesign = CreateDesign(TP, ...
    'Type', 'Latin Hypercube Sampling',...
    'Name', 'Space Filling');
```

Version History

Introduced before R2006a

See Also

`CreateDesign`

Topics

"Create Local Designs"
"Create Optimal Designs"

AddConstraint

Add design constraint

Syntax

```
DoeObjMod = AddConstraint(DoeObj,Constr)
```

Description

`DoeObjMod = AddConstraint(DoeObj,Constr)` adds a constraint to the design. You must call `AddConstraint` to apply the constraint and remove points outside the constraint.

Input Arguments

`DoeObj` — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

`Constr` — Design constraint
`mbcdoe.designconstraint` object | boundary model object

Design constraint, specified as a `mbcdoe.designconstraint` object or boundary model object.

If `Constr` is a boundary model, `AddConstraint` also converts the boundary model object to a `mbcdoe.designconstraint` object.

Output Arguments

`DoeObjMod` — Modified instance of `mbcdoe.design` class
`mbcdoe.design` object

Modified instance of `mbcdoe.design` class, returned as a `mbcdoe.design` object.

Version History

Introduced in R2008a

See Also

`mbcdoe.design` | `CreateConstraint`

Augment

Add design points

Syntax

```
DoeObjMod = Augment(DoeObj,NumPoints)
DoeObjMod = Augment(DoeObj,'Name1','Value1',…)
```

Description

`DoeObjMod = Augment(DoeObj,NumPoints)` augments the design with the number of points specified by `NumPoints` using the current generator settings.

`DoeObjMod = Augment(DoeObj,'Name1','Value1',…)` augments the design with the generator specified by the name-value pairs.

Examples

Add Points to Existing Type Using a Different Type

```
OptDesign = Augment(OptDesign,...
    'Type','V-optimal',...
    'MaxIterations',200,...
    'NoImprovement', 50,...
    'NumberOfPoints',20);
```

Optimally Augment Existing Design

Use `FixPoints` to set all the designs points to fixed and then optimally augment an existing design.

```
OptDesign = FixPoints(OptDesign);
OptDesign = Augment(OptDesign,...
    'Type','V-optimal',...
    'MaxIterations',200,...
    'NoImprovement', 50,...
    'NumberOfPoints',20);
```

Optimally Add Points and Keep Fixed Points

Use `RemovePoints` to optimally add points and keep only fixed points.

```
OptDesign = RemovePoints(OptDesign,'free');
OptDesign = Augment(OptDesign,...
    'Type','V-optimal',...
    'MaxIterations',200,...
    'NoImprovement', 50,...
    'NumberOfPoints',20);
```

To get a candidate set object for use with an optimal design, use this code.

```
C = CreateCandidateSet(OptDesign,'Type', 'Grid',...
       'NumberOfLevels',[21 21 21]);
```

You see an error if you try to call `Augment` when the design `Style` is User-defined or Experimental data.

Optimally Augment Design with 10 Points

Create a candidate set and optimally augment it with 10 points.

```
CandidateSet = augmentedDesign.CreateCandidateSet...
( 'Type', 'Grid' );
CandidateSet.NumberOfLevels = [21 21 21 21];
augmentedDesign = Augment( augmentedDesign,...
    'Type', 'V-optimal',...
    'NumberOfPoints', 10,...
    'CandidateSet', CandidateSet,...
    'MaxIterations', 200,...
    'NoImprovement', 50 );
```

Input Arguments

`DoeObj` — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

`NumPoints` — Number of design points
`mbcdoe.designconstraint` object

Number of design points, specified as a `mbcdoe.designconstraint` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

The design generator properties determines the applicable name-value pairs.

For a complete list of the properties for each design type, see `Properties (for design generators)`.

Example: `'Type','V-optimal'`

`Type` — Generator type
character vector

Generator type, specified as the comma-separated pair consisting of 'Type' and a character vector.

Example: `'Type','V-optimal'`

`NumberOfPoints` — Number of points
scalar

Number of points, specified as the comma-separated pair consisting of 'NumberOfPoints' and an integer.

Example: `'NumberOfPoints',20`

Output Arguments

`DoeObjMod` — Modified instance of `mbcdoe.design` class
`mbcdoe.design` object

Modified instance of `mbcdoe.design` class, returned as a `mbcdoe.design` object.

Version History

Introduced in R2008a

See Also

`mbcdoe.design` | `Generate` | `CreateCandidateSet`

ConstrainedGenerate

Generate constrained space-filling design of specified size

Syntax

```
DoeObjMod = ConstrainedGenerate(DoeObj,NumPoints)
DoeObjMod = ConstrainedGenerate(DoeObj,NumPoints,'Name1','Value1',…)
```

Description

`DoeObjMod = ConstrainedGenerate(DoeObj,NumPoints)` generates a space-filling design with the number of constrained points specified by `NumPoints`.

Use `ConstrainedGenerate` for only space-filling designs. It may not be possible to achieve a specified number of points, depending on the generator settings and constraints.

`ConstrainedGenerate` calls `Generate` and uses this formula to update `UnconstrainedSize`.

```
UnconstrainedSize = ceil(UnconstrainedSize * NumPoints/D.NumberOfPoints);
```

`DoeObjMod = ConstrainedGenerate(DoeObj,NumPoints,'Name1','Value1',…)` augments the generated space-filling design with the options specified by the name-value pairs.

Examples

Use Space-Filling Design to Create 200-Point Design

Use an existing space-filling design to create a 200-point design, then you inspect the constrained design and the total number of points.

```
sfDesign = ConstrainedGenerate( sfDesign, 200, 'UnconstrainedSize', 800, 'MaxIter',10  );

% How did we do?
finalNumberOfPoints = sfDesign.NumberOfPoints
% How many points did we need in total?
totalNumberOfPoints = sfDesign.Generator.NumberOfPoints

finalNumberOfPoints =
   200
totalNumberOfPoints =
   839
```

Input Arguments

`DoeObj` — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

`NumPoints` — Number of design points
`mbcdoe.designconstraint` object

Number of design points, specified as a `mbcdoe.designconstraint` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

The design generator properties determines the applicable name-value pairs. For a complete list of the properties, see `Properties (for design generators)`.

Example: `'Type','V-optimal'`

`MaxIterations` — Maximum number of iterations
`20` (default) | scalar

Maximum number of iterations, specified as the comma-separated pair consisting of 'MaxIterations' and an integer.

Example: `'MaxIterations',15`

`UnconstrainedSize` — Number of points in unconstrained design
`Numpoints` (default) | scalar

Total number of points in the unconstrained design, specified as the comma-separated pair consisting of 'UnconstrainedSize' and an integer.

Example: `'UnconstrainedSize',10`

Output Arguments

`DoeObjMod` — Modified instance of `mbcdoe.design` class
`mbcdoe.design` object

Modified instance of `mbcdoe.design` class, returned as a `mbcdoe.design` object.

Version History

Introduced in R2008a

See Also

`mbcdoe.design` | `CreateConstraint` | `Generate`

CreateCandidateSet

Create candidate set for optimal designs

Syntax

```
DoeObjMod = CreateCandidateSet(DoeObj)
DoeObjMod = CreateCandidateSet(DoeObj,'Name1','Value1',…)
```

Description

`DoeObjMod = CreateCandidateSet(DoeObj)` creates a candidate set object for the design.

Use `ConstrainedGenerate` to create a candidate set for optimal designs. Candidate sets are similar to design generators. The software does not use a candidate set to specify a design. Instead, candidate sets specify the set of all possible points to consider as part of an optimal design.

`DoeObjMod = CreateCandidateSet(DoeObj,'Name1','Value1',…)` creates a candidate set object for the design with the options specified by the name-value pairs.

Examples

Create Candidate Set

```
mdl = CreateModel(mbcmodel,'Polynomial',4)
ActualDesign = CreateDesign(mdl,'Type','V-optimal')

c = CreateCandidateSet(ActualDesign,'Type','Grid')
c.NumberOfLevels = [11 11 11 11];

ActualDesign = Generate(ActualDesign,'NumberOfPoints',50,'CandidateSet',c)
```

Input Arguments

`DoeObj` — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

The design candidate properties determines the applicable name-value pairs. This table provides a complete list of the properties.

Candidate Set Properties (for Optimal Designs)

Candidate Set Type	Property	Description
All built-in: Grid/ Lattice, Grid, Lattice, Stratified Lattice, Sobol, Halton	NumberOfPoints (read-only for Grid and Grid/Lattice)	Number of points (int: [0,Inf])
	Limits	Design Limits
Grid	Levels	Selection criteria for best LHS design (cell)
	NumberPerLevel	Symmetric design (vector int: {[-Inf,Inf], NumInputs})
Lattice	Generators	Prime number generators for lattice (vector int: {[0,Inf], NumInputs})
Stratified Lattice	StratifyLevels	Number of levels for each factors (vector int: {[0,Inf], NumInputs})
Sobol Sequence	Scramble	Scramble method (enum: {‘none’, ’MatousekAffineOwen’}
	SkipMode	Skip mode options (enum: {'None','2^k','Custom'})
	Skip	Skip size (int: [0,Inf])
Halton Sequence	Scramble	Scrambling method for sequence (enum: {'None','RR2'})
	PrimeLeap	Leap sequence points using prime number (boolean)
	SkipZero	Skip zero point (boolean)
User-defined	NumberOfPoints	User-defined points (read-only)
	Points	User-defined points

Example: `'Type','Grid'`

Type — Generator type
character vector

Generator type, specified as the comma-separated pair consisting of 'Type' and a character vector.

Example: `'Type','V-optimal'`

Output Arguments

`DoeObjMod` — Modified instance of `mbcdoe.design` class
`mbcdoe.design` object

Modified instance of `mbcdoe.design` class, returned as a `mbcdoe.design` object.

Version History

Introduced in R2008a

See Also

`mbcdoe.design` | `Augment`

Topics

Candidate Set Properties (for Optimal Designs)
`Properties (for candidate sets)`

CreateConstraint

Create design constraint

Syntax

```
DoeObjMod = CreateConstraint(DoeObj)
DoeObjMod = CreateConstraint(DoeObj,'Name1','Value1',…)
```

Description

`DoeObjMod = CreateConstraint(DoeObj)` creates a default constraint for the design.

Designs have a `constraints` property that is initially empty.

```
constraints = design.Constraints

constraints =
0x0 array of mbcdoe.designconstraint
```

`DoeObjMod = CreateConstraint(DoeObj,'Name1','Value1',…)` creates a constraint with properties specified by the name-value pairs.

By default, `CreateConstraint` creates a 1D table constraint for designs with two or more inputs. For a design with one input, `CreateConstraint` creates a linear constraint by default. You can specify the constraint type using the `Type` property.

`CreateConstraint` does not add the constraint to the design. To add a constraint to the design, use `AddConstraint`.

Examples

Create Linear Constraint

```
cLinear = CreateConstraint(design,'Type','Linear');
cLinear.A = [-2.5e-4, 1];
cLinear.b = 0.25;
cLinear
design.Constraints = cLinear;
design = Generate(design);
```

Create 1D Table Constraint

Create and apply a 1D table constraint.

```
cTable1d = CreateConstraint(design,'Type','1D Table');
cTable1d.Table = [0.9 0.5];
cTable1d.Breakpoints = [500 6000];
cTable1d
```

```
design.Constraints = cTable1d;
design = Generate(design);
```

Combine Constraints

```
design.Constraints = [cLinear, cTable1d];
constraints = design.Constraints
design = Generate(design);

constraints =
1x2 array of mbcdoe.designconstraint
Linear design constraint: -0.00025*N + 1*L <= 0.25
1D Table design constraint: L(N) <= Lmax
```

Load Boundary Constraints

Load boundary constraints from another project file and add to them to the design.

```
otherProject = mbcmodel.LoadProject( [matlabroot,'\toolbox\',...
'mbc\mbctraining\Gasoline_project.mat']);
mytestplan = otherProject.Testplans(1);
boundaryConstraints = BoundaryModel(mytestplan,'global');
Design.Constraints = boundaryConstraints;
```

Input Arguments

DoeObj — Instance of `mbcdoe.design` class

`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

The design generator properties determines the applicable name-value pairs. This table provides a complete list of the properties.

Constraint Properties

Constraint Type	Property	Description
Linear design constraint: 1*Input1 + 1* Input2 + 1* Input3 <= 0	A	Matrix for linear constraint (matrix: [1,NumInputs])
	b	Bound for linear constraint (double)
Ellipsoid design constraint: Ellipsoid at (Input1=0, Input2=0, Input3=0)	CenterPoint	Center of ellipse (vector: NumInputs)
	Matrix	Ellipsoid form matrix (positive semi-definite) (matrix: [NumInputs, NumInputs])
1D Table design constraint: InputY(InputX) <= InputY max	Table	Table constraint (vector)
	Breakpoints	Breakpoints for rows (vector)
	Inequality	Relational Operator (enum: `{'<=','>='}`)
	InputFactor	Column input symbol (enum: `{ 'InputX','InputY'}`)
	TableFactor	Table input symbol (enum: `{'InputX','InputY '}`)
2D Table design constraint: InputZ(InputX,InputY) <=InputZmax	Table	: Table constraint (matrix))
	RowBreakpoints	Breakpoints for rows (vector)
	ColumnBreakpoints	Breakpoints for columns (vector)
	Inequality	Relational operator (enum: `{'<=','>='}`)
	RowFactor	Row input symbol (enum: `{'InputX','InputY, 'InputZ'}`)
	ColumnFactor	Column input symbol (enum: `{'InputX','InputY, 'InputZ'}`)
	TableFactor	Table input symbol (enum: `{'InputX','InputY', 'InputZ'}`

Example: `'Type','Linear'`

Type — Constraint type
character vector

Constraint type, specified as as the comma-separated pair consisting of 'Type' and a character vector.

Example: `'Type','V-optimal'`

Output Arguments

DoeObjMod — Modified instance of mbcdoe.design class
mbcdoe.design object

Modified instance of mbcdoe.design class, returned as a mbcdoe.design object.

Version History

Introduced in R2008a

See Also

mbcdoe.design | AddConstraint

Topics

Properties (for design constraints)

Discrepancy

Discrepancy value

Syntax

```
Discrep = Discrepancy(DoeObj)
```

Description

`Discrep = Discrepancy(DoeObj)` returns the design object discrepancy. The discrepancy is a measure of the deviation from the average point density. The discrepancy is defined over the unconstrained design and is available for only space-filling designs.

Input Arguments

`DoeObj` — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

Output Arguments

`Discrep` — Design discrepancy
array

Design discrepancy, returned as an array.

Version History

Introduced in R2008a

See Also

`mbcdoe.design` | `Maximin` | `Minimax`

FixPoints

Fix design points

Syntax

```
DoeObjMod = FixPoints(DoeObj)
DoeObjMod = FixPoints(DoeObj,indices)
```

Description

`DoeObjMod = FixPoints(DoeObj)` fixes all points in the design object.

`DoeObjMod = FixPoints(DoeObj,indices)` fixes all points in the design object specified by `indices`.

Input Arguments

`DoeObj` — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

`indices` — Design object indices
array

Design object indices, specified as an array.

Output Arguments

`DoeObjMod` — Modified instance of `mbcdoe.design` class
`mbcdoe.design` object

Modified instance of `mbcdoe.design` class, returned as a `mbcdoe.design` object.

Version History

Introduced in R2008a

See Also

`mbcdoe.design` | `RemovePoints`

Generate

Generate design points

Syntax

```
DoeObjMod = Generate(DoeObj)
DoeObjMod = Generate(DoeObj,NumPoints)
DoeObjMod = Generate(DoeObj,'Name1','Value1',…)
```

Description

`DoeObjMod = Generate(DoeObj)` regenerates the design with the current design properties and number of points. Calling `Generate` for Latin Hypercube Sampling can result in a different design.

`DoeObjMod = Generate(DoeObj,NumPoints)` generates the number of points specified by `NumPoints` using the current generator settings. You cannot specify the number of points for all design types (e.g., Central Composite, Box Behnken). Therefore, `NumPoints` is not supported for all design types.

The design `Type` must have a writable property `'NumberOfPoints'` to use this syntax. See `Type (for designs and generators)`.

Using `Generate` with constrained space-filling is not guaranteed to produce a design with the specified number of points. Use `ConstrainedGenerate` instead.

`DoeObjMod = Generate(DoeObj,'Name1','Value1',…)` augments the design with the generator specified by the name-value pairs.

Examples

Generate Design with 10 Points

```
DoeObjMod = Generate(DoeObj,10);
```

Generate Latin Hypercube Sampling Designs

Generate a 15-point Latin Hypercube Sampling design.

```
globalDesign = TP.CreateDesign(2, 'Type',...
 'Latin Hypercube Sampling');
globalDesign = Generate(globalDesign, 15)
```

Use this code to regenerate the design and get a different 15-point Latin Hypercube Sampling design.

```
globalDesign = Generate(globalDesign);
```

Generate Halton and Full Factorial Designs

Use this code to create and generate a Halton design with 50 points.

```
haltonDesign = CreateDesign( inputs, 'Type',...
 'Halton Sequence', 'Name', 'Halton' );
haltonDesign = Generate( haltonDesign, 'NumberOfPoints', 50 );
```

Use this code to create and generate a Halton design with specified scrambling and other properties.

```
haltonDesignWithScrambling = haltonDesign.CreateDesign...
( 'Name', 'Scrambled Halton' );
haltonDesignWithScrambling = Generate...
(haltonDesignWithScrambling,
 'Scramble', 'RR2', 'PrimeLeap', true );
```

Use this code to create a full factorial design and specify the number of levels when generating the design.

```
design = CreateDesign( inputs, 'Type', 'Full Factorial' );
design = Generate( design, 'NumberOfLevels', [50 50] );
```

Specify Design Generator Properties

You can use name-value pair arguments to specify design generator properties.

```
C = OptDesign.CreateCandidateSet(OptDesign,...
       'Type', 'Grid',...
       'NumberOfLevels',[21 21 21]);

OptDesign = Generate(OptDesign,...
    'Type','V-optimal',...
    'CandidateSet',C,...
    'MaxIterations',200,...
    'NoImprovement', 50,...
    'NumberOfPoints',200);
```

The preceding code is equivalent to the following code that sets the properties individually and assigns the updated object to the design.

```
P = OptDesign.Generator;
P.Type = 'V-optimal';
P.CandidateSet.NumberOfLevels(:)=21;
P.MaxIterations = 200;
P.NumberOfPoints = 200;
P.NoImprovement = 50;
OptDesign.Generator = P;
```

You see an error if you call `Generate` when the design Style is User-defined or Experimental data.

Input Arguments

`DoeObj` — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

`NumPoints` — Number of design points
`mbcdoe.designconstraint` object

Number of design points, specified as a `mbcdoe.designconstraint` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

The design generator properties determines the applicable name-value pairs.

For a complete list of the properties for each design type, see `Properties (for design generators)`.

Example: `'Type','V-optimal'`

Type — Generator type
character vector

Generator type, specified as the comma-separated pair consisting of 'Type' and character vector.

Example: `'Type','V-optimal'`

`NumberOfPoints` — Number of points
scalar

Number of points, specified as the comma-separated pair consisting of 'NumberOfPoints' and an integer.

Example: `'NumberOfPoints',20`

Output Arguments

`DoeObjMod` — Modified instance of `mbcdoe.design` class
`mbcdoe.design` object

Modified instance of `mbcdoe.design` class, returned as a `mbcdoe.design` object.

Version History

Introduced in R2008a

See Also

`mbcdoe.design` | `Augment` | `CreateDesign` | `ConstrainedGenerate`

Topics

`Type (for designs and generators)`

getAlternativeTypes

Alternative model or design types

Syntax

```
List = getAlternativeTypes(Model)
List = getAlternativeTypes(Boundary)
List = getAlternativeTypes(Design)
List = getAlternativeTypes(Design,Style)
List = getAlternativeTypes(DesignGenerator)
List = getAlternativeTypes(DesignGenerator,Style)
List = getAlternativeTypes(CandidateSet)
List = getAlternativeTypes(DesignConstraint)
```

Description

`List = getAlternativeTypes(Model)` returns a cell array of alternative model types with the same number of inputs as `Model`.

`List = getAlternativeTypes(Boundary)` returns a cell array of alternative boundary models with the same number of inputs as `Boundary`.

`List = getAlternativeTypes(Design)` returns a cell array of alternative designs with the same number of inputs as `Design`.

`List = getAlternativeTypes(Design,Style)` returns a cell array of alternative designs with the same number of inputs as `Design` with `Style`.

`List = getAlternativeTypes(DesignGenerator)` returns a cell array of alternative design generators with the same number of inputs as `DesignGenerator`.

`List = getAlternativeTypes(DesignGenerator,Style)` returns a cell array of alternative design generator types of the specified style.

`List = getAlternativeTypes(CandidateSet)` returns a cell array of alternative candidate sets.

`List = getAlternativeTypes(DesignConstraint)` returns a cell array of design constraint types.

Examples

Obtain List of Alternative Models

```
model = mbcmodel.CreateModel('RBF', 2);
altmodels = getAlternativeTypes(model)
```

The preceding code produces this output.

```
altmodels =
```

```
1×10 cell array

Columns 1 through 2

  {'Polynomial'}    {'Hybrid Spline'}

Columns 3 through 5

  {'Gaussian Proces…'}    {'RBF'}    {'Polynomial-RBF'}

Columns 6 through 7

  {'Hybrid Spline-RBF'}    {'Multiple Linear'}

Columns 8 through 9

  {'Neural Network'}    {'Interpolating RBF'}

Column 10

  {'Transient'}
```

Input Arguments

Model — Instance of `mbcmodel.model` class
`mbcmodel.model` object

Instance of `mbcmodel.model` class, specified as a `mbcmodel.model` model object.

Boundary — Instance of `mbcboundary.AbstractBoundary` class or subclass
`mbcboundary.AbstractBoundary` object

Instance of `mbcboundary.AbstractBoundary` class or subclass, specified as a `mbcboundary.AbstractBoundary` object.

Design — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` object.

Style — Design type style
n-by-1 array

Design type style, specified as an `n-by-1` array of character vectors, where *n* is the number of designs.

The design style is one of the following:

- `'Optimal'`
- `'Space-Filling'`
- `'Classical'`
- `'Candidate Set'` (for design generator styles)

Data Types: `char` | `string`

`DesignGenerator` — Instance of `mbcdoe.generator` class
`mbcdoe.generator` object

Instance of `mbcdoe.generator` class, specified as a `mbcdoe.generator` object.

`CandidateSet` — Instance of `mbcdoe.candidateset` class
`mbcdoe.candidateset` object

Instance of `mbcdoe.candidateset` class, specified as a `mbcdoe.candidateset` object. You can obtain the candidate set from an optimal design generator or use `mbcdoe.design.CreateCandidateSet`.

`DesignConstraint` — Instance of `mbcdoe.designconstraint` class
`mbcdoe.designconstraint` object

Instance of `mbcdoe.designconstraint` class, specified as a `mbcdoe.designconstraint` object.

Output Arguments

`List` — List of alternative candidate sets
`n`-by-1 array

List of alternative candidate for the current candidate set, returned as an `n`-by-`1` array.

Version History

Introduced in R2007a

See Also

`mbcdoe.design` | `CreateModel` | `mbcmodel.modelproperties`

Maximin

Maximum of minimum of distance between design points

Syntax

```
Max = Maximin(DoeObj)
```

Description

`Max = Maximin(DoeObj)` returns the maximum of the minimum distance between design points. `Maximin` is defined over the unconstrained design and is only available for space-filling design types.

Input Arguments

`DoeObj` — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

Output Arguments

`Max` — Maximum of minimum
scalar

Maximum of minimum distance between design points, returned as a scalar.

Version History

Introduced in R2008a

See Also

`mbcdoe.design` | `Minimax`

Merge

Merge designs

Syntax

```
DoeObjMod = Merge(DoeObj1,DoeObj2,...)
```

Description

`DoeObjMod = Merge(DoeObj1,DoeObj2,...)` merges multiple design objects into a single design. The resulting design is a custom design `Style`.

Input Arguments

`DoeObj1` — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

`DoeObj2` — Other instance of `mbcdoe.design` class
`mbcdoe.design` object

Other instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

Output Arguments

`DoeObjMod` — Modified instance of `mbcdoe.design` class
`mbcdoe.design` object

Modified instance of `mbcdoe.design` class, returned as a `mbcdoe.design` object.

Version History

Introduced in R2008a

See Also

`mbcdoe.design` | `Augment`

Minimax

Minimum of maximum distance between design points

Syntax

```
Min = Minimax(DoeObj)
```

Description

`Min = Minimax(DoeObj)` returns the minimum of the maximum distance between design points. `Minimax` is defined over the unconstrained design and is only available for space-filling design types.

Input Arguments

`DoeObj` — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

Output Arguments

`Min` — Minimum of maximum
scalar

Minimum of maximum distance between design points, returned as a scalar.

Version History

Introduced in R2008a

See Also

`mbcdoe.design` | `Maximin`

OptimalCriteria

Optimal design criteria

Syntax

```
OptCrit = OptimalCriteria(DoeObj)
OptCrit = OptimalCriteria(DoeObj,Criteria)
```

Description

`OptCrit = OptimalCriteria(DoeObj)` returns an array with the values of optimal criteria [V,D,A,G].

`OptCrit = OptimalCriteria(DoeObj,Criteria)` returns the specified optimal criteria. `Criteria` must be one of `V`, `D`, `A`, or `G`.

Input Arguments

`DoeObj` — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

`Criteria` — Optimal design criteria
`'V'` | `'D'` | `'A'` | `'G'`

Optimal design criteria, specified as `'V'`, `'D'`, `'A'`, or `'G'`.

Output Arguments

`OptCrit` — Optimal criteria
`n`-by-`1` array

Optimal criteria, returned as an `n`-by-`1` array containing either `V`, `D`, `A`, or `G`.

Version History

Introduced in R2008a

See Also

`mbcdoe.design` | `Maximin`

RemovePoints

Remove all nonfixed design points

Syntax

```
DoeObjMod = RemovePoints(DoeObj)
DoeObjMod = RemovePoints(DoeObj,PointType)
DoeObjMod = RemovePoints(DoeObj,indices)
```

Description

`DoeObjMod = RemovePoints(DoeObj)` removes all non-fixed points from the design.

`DoeObjMod = RemovePoints(DoeObj,PointType)` removes the specified type of points, where `PointType` is `'free'`,`'fixed'` or `'data'`.

`DoeObjMod = RemovePoints(DoeObj,indices)` removes the points specified by `indices`.

Examples

Remove Free Points

```
Design = RemovePoints(Design,'free');
```

Input Arguments

`DoeObj` — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

`PointType` — Design point types
`'free'` | `'fixed'` | `'data'`

Design point types, specified as `'free'`, `'fixed'`, or `'data'`.

`indices` — Design object indices
array

Design object indices, specified as an array.

Output Arguments

`DoeObjMod` — Modified instance of `mbcdoe.design` class
`mbcdoe.design` object

Modified instance of `mbcdoe.design` class, returned as a `mbcdoe.design` object.

Version History

Introduced in R2008a

See Also

`mbcdoe.design` | `FixPoints`

Scatter2D

Plot design points

Syntax

```
Scatter2D(DoeObj,Xindex,Yindex)
Scatter2D(DoeObj,Xindex,Yindex,plotArguments)
```

Description

`Scatter2D(DoeObj,Xindex,Yindex)` creates a scatter plot of the design points in design `D`, where X and Y are the indices or symbols for the input factors to plot on the X and Y axis.

`Scatter2D(DoeObj,Xindex,Yindex,plotArguments)` creates a scatter plot with additional arguments.

Examples

Plot Design Data

```
Scatter2D( mainDesign, 1, 2 );
```

Input Arguments

`DoeObj` — Instance of `mbcdoe.design` class
`mbcdoe.design` object

Instance of `mbcdoe.design` class, specified as a `mbcdoe.design` doe design object.

`Xindex` — X value indices
array

X value indices, specified as an array.

`Yindex` — Y value indices
array

Y value indices, specified as an array.

`plotArguments` — Plot command arguments
array

Additional plot command arguments, specified as an array. See `plot`. Scatter2D uses this command:

```
plot(D.Points(:,v1),D.Points(:,v2),varargin{:})
```

The default for `varargin` is `'.'`.

Version History

Introduced in R2008a

See Also

`mbcdoe.design` | `FixPoints` | `plot`

MBCModel.LinearModel

mbcmodel.linearmodel

Properties and methods for linear model objects

Description

Use these properties and object functions to create and examine model objects.

Creation

Create a mbcmodel.linearmodel object using `CreateModel`.

Properties

`Data` — Data stored in model
array

Data stored in model, returned as an array.

`FitAlgorithm` — Fit algorithm for model
array

Fit algorithm for model, specified as an array.

`FitAlgorithm` is a property of `mbcmodel.model`, and boundary model objects `mbcboundary.AbstractBoundary` and all subclasses.

An `mbcmodel.model.FitAlgorithm` object is contained within the `FitAlgorithm` property of an `mbcmodel.model` object or `mbcboundary` object.

As an alternative to using `CreateAlgorithm`, you can assign the algorithm name directly to the algorithm.

```
B.FitAlgorithm.BoundaryPointOptions = 'Boundary Only';

m.FitAlgorithm = 'Minimize PRESS';
```

Case and spaces are ignored.

To get a `fitalgorithm` object, `F`, from a model, use this code.

```
M = mbcmodel.CreateModel('Polynomial', 4);
F = M.FitAlgorithm
```

```
F =
Algorithm: Least Squares
Alternatives: 'Minimize PRESS','Forward Selection','Backward
Selection','Prune'
1x1 struct array with no fields.
```

InputData — Input training data
matrix

This property is read-only.

Input training data, specified as a matrix. `InputData` is specified when calling fit.

Data Types: `double`

Inputs — Model input
`mbcmodel.modelinput` object

Model input, specified as a `modelinput` object.

IsBeingEdited — Boolean indicating if model is being edited
`true` or `1` | `false` or `0`

This property is read-only.

Boolean indicating if model is being edited, specified as either `true` (`1`) or `false` (`0`).

Example: 0

Data Types: `logical`

IsEditable — Boolean indicating if model is editable
`true` or `1` | `false` or `0`

This property is read-only.

Boolean signaling if model is editable, specified as either `true` (`1`) or `false` (`0`). The following rules apply:

- If the model was created using `mbcmodel.CreateModel` and is not `Attached` to a test plan it is editable.
- If the model was created or retrieved from the project and was not `Attached` to a test plan, it is editable.
- If the data was `Attached` to a test plan and was subsequently retrieved from that test plan, it is editable.

Data Types: `logical`

Name — Model object name
character vector

Model object name, specified as a character vector.

Data Types: `char` | `string`

NumInputs — Number of inputs to model
real positive scalar

This property is read-only.

Number of inputs to model, specified as a real positive scalar.

Data Types: `double` | `single`

OutputData — Output or response data
matrix

This property is read-only.

Output or response data, specified as a matrix. `OutputData` is specified when calling fit.

Data Types: `double`

Response — Response object
object

This property is read-only.

Response object in `mbcmodel.project` object, specified as an object.

Status — Status of model fit
`Not Fitted` | `Fitted` | `Best`

This property is read-only.

Status of model fit, specified as either `Not Fitted`, `Fitted`, or `Best`.

Data Types: `char` | `string`

Type — Type of model objects
vector

This property is read-only.

Type of model objects to be returned to `mbcmodel.project`, specified as a vector. `model.type` returns the model type.

The model `Type` determines which properties you can set. To set properties, see `mbcmodel.modelproperties` and `LocalModel Properties`.

Note Spaces and case in model `Type` are ignored.

The model type must be one in this table.

Type	Model Object
Polynomial	`mbcmodel.linearmodel`
Hybrid Spline	`mbcmodel.linearmodel`
RBF	`mbcmodel.linearmodel`
Hybrid RBF	`mbcmodel.linearmodel`
Polynomial-RBF	`mbcmodel.linearmodel`
Hybrid Spline-RBF	`mbcmodel.linearmodel`
Multiple Linear	`mbcmodel.linearmodel`

You can get a list of types by using `getAlternativeTypes`. Use this syntax.

```
Mlist = getAlternativeTypes(M)
```

where `M` is an `mbcmodel.model` object.

Data Types: `char` | `string`

`Units` — Model output unit
vector

Unit of model output, specified as a vector.

Data Types: `double` | `single`

Object Functions

AliasMatrix	Alias matrix for linear model parameters
BoxCoxSSE	SSE and confidence interval for Box-Cox transformations
CreateDesign	Create design object for test plan or model
evaluate	Evaluate model, boundary model, or design constraint
Export	Make command-line or Simulink export model
fit	Fit model or boundary model to new or existing data, and provide summary statistics
InputSetupDialog	Open Input Setup dialog box to edit inputs
Jacobian	Calculate Jacobian matrix for model at existing or new data points
ModelSetup	Open Model Setup dialog box where you can alter model type
pev	Predicted error variance of model at specified inputs
PredictedValue	Predicted value of model at specified inputs
StatisticsDialog	Open summary statistics dialog box
SummaryStatistics	Summary statistics for response
UpdateResponse	Replace model in response
getAlternativeTypes	Alternative model or design types
Correlation	Correlation matrix for linear model parameters
Covariance	Covariance matrix for linear model parameters
ValidationRMSE	Calculates the validation RMSE for model data
MultipleVIF	Multiple VIF matrix for linear model parameters
ParameterStatistics	Calculate parameter statistics for linear model
SingleVIF	Single VIF matrix for linear model parameters
PartialVIF	Partial VIF matrix for linear model parameters
StepwiseRegression	Change stepwise selection status for specified terms

Version History

Introduced before R2006a

See Also

`mbcmodel.project` | `mbcdoe.design` | `mbcmodel.data`

AliasMatrix

Alias matrix for linear model parameters

Syntax

```
A = AliasMatrix(model>)
```

Description

A = AliasMatrix(model>) calculates the alias matrix for the linear model parameters (where model> is a linear model).

Examples

Calculate Alias Matrix

Calculate alias matrix of knot model using this syntax.

```
A = AliasMatrix(knot_model)
```

Input Arguments

model — Linear model
mbcmodel.linearmodel object

Linear model, specified as a mbcmodel.linearmodel object.

Output Arguments

A — Alias matrix
matrix

Alias matrix of the linear model model>.

Version History

Introduced in R2007a

See Also

mbcmodel.linearmodel | CreateDesign | mbcmodel.model

BoxCoxSSE

SSE and confidence interval for Box-Cox transformations

Syntax

```
[sse, ci, lambda_out] = BoxCoxSSE(model, lambda_in)
[sse, ci, lambda_out] = BoxCoxSSE(Model)
BoxCoxSSE(model, ...)
```

Description

`[sse, ci, lambda_out] = BoxCoxSSE(model, lambda_in)` computes the sum of squares error `sse` and confidence interval `ci` for values of the model under different Box-Cox transforms as given by the parameter `lambda`).

`[sse, ci, lambda_out] = BoxCoxSSE(Model)` returns the default value in `lambda_out` when `lambda_in` is not specified.

`BoxCoxSSE(model, ...)` If no output arguments are requested then a plot of SSE versus lambda is displayed. The confidence intervals are also displayed on this plot.

Examples

Plot results of Box-Cox values

To try several different values of the Box-Cox parameter and plot the results:

```
lambda = -3:0.5:3;
[sse, ci] = BoxCoxSSE( M, lambda);
semilogy( lambda, sse, 'bo-', lambda([1,end]), [ci, ci], 'r--' );
xlabel( 'Box-Cox parameter, \lambda' );
ylabel( 'SSE' );
```

Note that `BoxCoxSSE` does not set a Box-Cox transform in the model. To do this use:

```
M.Properties.BoxCox = 0;
[S,M] = M.Fit;
```

Input Arguments

`model` — Model object
`mbcmodel.linearmodel` object

Model object, specified as a `mbcmodel.linearmodel` object.

`lambda_in` — Input data
vector

Input data used to fit the model, specified as a vector.

Output Arguments

`sse` — Sum of squares error
real vector

Sum of squares error, returned as a real scalar. `sse` is the same size as `lamba_in` and `lamba_out`.

`ci` — Confidence interval
real scalar

Confidence interval of sum of squares error, returned as a real scalar. There is no statistical difference between the Box-Cox transforms where `sse` less than `ci`.

`lambda_out` — Output data
vector

Output data used to fit the model, returned as a vector. If no `lambda_in` is specified, then default values are returned.

Version History

Introduced in R2007a

See Also

`ParameterStatistics`

Covariance

Covariance matrix for linear model parameters

Syntax

`stats = Covariance(linearmodel)`

Description

`stats = Covariance(linearmodel)` calculates the covariance matrix for the linear model parameters.

Examples

Calculate Covariance

Calculate covariance of knot linear model.

```
Stats = Covariance(knot_model)
```

Input Arguments

`linearmodel` — Linear model
`mbcmodel.linearmodel` object

Linear model whose covariance is being calculated, specified as a `mbcmodel.linearmodel` object.

Output Arguments

`stats` — Covariance
matrix

Covariance of linear model, specified as a matrix.

Version History

Introduced in R2007a

See Also

`ParameterStatistics`

Correlation

Correlation matrix for linear model parameters

Syntax

```
stats = Correlation(linearmodel)
```

Description

`stats = Correlation(linearmodel)` calculates the correlation matrix for the linear model parameters.

Examples

Calculate Correlation

Calculate correlation of knot linear model.

```
Stats = Correlation(knot_model)
```

Input Arguments

`linearmodel` — Linear model
`mbcmodel.linearmodel` object

Linear model whose correlation is being calculated, specified as a `mbcmodel.linearmodel` object.

Output Arguments

`stats` — Correlation
matrix

Correlation of linear model, specified as a matrix.

Version History

Introduced in R2007a

See Also

`ParameterStatistics`

MBCModel.ModelProperties

mbcmodel.modelproperties

Properties and methods for viewing and editing model properties

Description

Use these properties and object functions to create and edit model properties.

Creation

Syntax

```
ModelPropObj = ModelObj.Properties
properties(ModelObj.Properties)
ModelObj.Properties.PropertyName = NewValue
```

Description

`ModelPropObj = ModelObj.Properties` creates an `mbcmodel.modelproperties` object for `ModelObj`.

`properties(ModelObj.Properties)` lists the property names, types, and allowed values.

`ModelObj.Properties.PropertyName = NewValue` changes the `ModelObj` property name `PropertyName` to the new value.

Properties

Every model object `mbcmodel.model` has an `mbcmodel.modelproperties` object. In the `mbcmodel.modelproperties` object, each model type has specific properties.

These tables provide the properties available for each model type.

Gaussian Process Models

Property	Description	Values
KernelFunction	Kernel function	String containing one of these enumerated values: • 'Exponential' • 'ARDExponential' • 'SquaredExponential' • 'ARDSquaredExponential' • 'Matern32' • 'ARDMatern32' • 'Matern52' • 'ARDMatern52' • 'RationalQuadratic' • 'ARDRationalQuadratic'
BasisFunction	Explicit basis function	String containing one of these enumerated values: • 'None' • 'Constant' • 'Linear' • 'PureQuadratic'
Threshold	Threshold to switch to large data fitting algorithm	Integer greater than or equal to 1
ActiveSetSize	Active set size	Integer greater than or equal to 1
ActiveSetMethod	Large scale active set method	String containing one of these enumerated values: • 'SGMA' • 'Entropy' • 'Likelihood' • 'Random'
FitMethod	Large data fit method	String containing one of these enumerated values: • 'SD' • 'FIC' • 'SR'

Property	Description	Values
PredictMethod	Large data predict method	String containing one of these enumerated values: • 'Exact' • 'BCD' • 'SD' • 'FIC' • 'SR'
InitializeMethod	Initialize hyperparameters	String containing one of these enumerated values: • 'LOO-loss' • 'logML' • 'none'
BoxCox	Box-Cox transform (power)	Numeric number between -3 and 3, inclusive

Linear Models – Polynomial

Property	Description	Values
Order	Polynomial order	n-by-1 vector of integers greater than or equal to 0, where *n* is the number of inputs
InteractionOrder	Maximum order of interaction terms	Integer greater than or equal to 0
TransformInputRange	Transform inputs	Boolean
ParameterNames	List of parameter names (read-only)	NA
StepwiseStatus	Stepwise status	Cell containing one of these values: • 'Always' • 'Never' • 'Step'
BoxCox	Box-Cox transform (power)	Numeric number between -3 and 3, inclusive

Linear Models – Hybrid Spline

Property	Description	Values
`Order`	Spline and polynomial order vector	n-by-1 vector of integers greater than or equal to 0 and less than or equal to 3, where *n* is the number of inputs
`SplineVariable`	Spline variable	NA
`SplineInteraction`	Order of interaction between spline and polynomial	Integer greater than or equal to 0 and less than or equal to 3
`Knots`	Position of knots	Real vector
`ParameterNames`	List of parameter names (read-only)	NA
`StepwiseStatus`	Stepwise status	Cell containing one of these values: • `'Always'` • `'Never'` • `'Step'`
`BoxCox`	Box-Cox transform (power)	Numeric number between -3 and 3, inclusive

Linear Models – RBF

Property	Description	Values
`Kernel`	RBF kernel	String containing one of these enumerated values: • `'multiquadric'` • `'recmultiquadric'` • `'gaussian'` • `'thinplate'` • `'logisticrbf'` • `'wendland'` • `'linearrbf'` • `'cubicrbf'`
`Continuity`	Continuity for Wendland kernel	Integer greater than or equal to 0 and less than or equal to 6
`ParameterNames`	List of parameter names (read-only)	NA
`StepwiseStatus`	Stepwise status	Cell containing one of these values: • `'Always'` • `'Never'` • `'Step'`
`BoxCox`	Box-Cox transform (power)	Numeric number between -3 and 3, inclusive

Linear Models – Polynomial RBF

Property	Description	Values
`Order`	Polynomial order vector	n-by-1 vector of integers greater than or equal to 0, where *n* is the number of inputs
`InteractionOrder`	Maximum order of interaction terms	Integer greater than or equal to 0
`Kernel`	RBF kernel	String containing one of these enumerated values: • `'multiquadric'` • `'recmultiquadric'` • `'gaussian'` • `'thinplate'` • `'logisticrbf'` • `'wendland'` • `'linearrbf'` • `'cubicrbf'`
`Continuity`	Continuity for Wendland kernel	Integer greater than or equal to 0 and less than or equal to 6
`ParameterNames`	List of parameter names (read-only)	NA
`StepwiseStatus`	Stepwise status	Cell containing one of these values: • `'Always'` • `'Never'` • `'Step'`
`BoxCox`	Box-Cox transform (power)	Numeric number between -3 and 3, inclusive

Linear Models – Hybrid Spline-RBF

Property	Description	Values
`Order`	Spline and polynomial order vector	`n`-by-`1` vector of integers greater than or equal to 0 and less than or equal to 3, where *n* is the number of inputs
`SplineVariable`	Spline variable	NA
`SplineInteraction`	Order of interaction between spline and polynomial	Integer greater than or equal to 0 and less than or equal to 3
`Knots`	Position of knots	Real vector
`Kernel`	RBF kernel	String containing one of these enumerated values: • `'multiquadric'` • `'recmultiquadric'` • `'gaussian'` • `'thinplate'` • `'logisticrbf'` • `'wendland'` • `'linearrbf'` • `'cubicrbf'`
`Continuity`	Continuity for Wendland kernel	Integer greater than or equal to 0 and less than or equal to 6
`ParameterNames`	List of parameter names (read-only)	NA
`StepwiseStatus`	Stepwise status	Cell containing one of these values: • `'Always'` • `'Never'` • `'Step'`
`BoxCox`	Box-Cox transform (power)	Numeric number between -3 and 3, inclusive

Nonlinear Models – Free Knot Spline

Property	Description	Values
`Order`	Spline and polynomial order vector	`n`-by-`1` vector of integers greater than or equal to 0 and less than or equal to 3, where *n* is the number of inputs
`NumKnots`	Number of knots	Positive integer

Nonlinear Models – Neural Network

Property	Description	Values
`HiddenLayers`	Number of hidden layers	Integer value 1 or 2
`Neurons`	Number of Neurons in each hidden layer	Vector of positive integers

Type — Type of model objects
vector

This property is read-only.

Type of model objects to be returned to `mbcmodel.project`, specified as a vector. `model.Type` returns the model type.

Note Spaces and case in model `Type` are ignored.

The model type must be one in this table.

Type	Model Object
Polynomial	`mbcmodel.linearmodel`
Hybrid Spline	`mbcmodel.linearmodel`
RBF	`mbcmodel.linearmodel`
Hybrid RBF	`mbcmodel.linearmodel`
Polynomial-RBF	`mbcmodel.linearmodel`
Hybrid Spline-RBF	`mbcmodel.linearmodel`
Multiple Linear	`mbcmodel.linearmodel`
Gaussian Process	`mbcmodel.model`
Free Knot Spline	`mbcmodel.model`
Transient	`mbcmodel.model`
User-Defined	`mbcmodel.model`
Neural Network	`mbcmodel.model`
Interpolating RBF	`mbcmodel.model`
Local Polynomial Spline	`mbcmodel.localmodel`
Local Polynomial with Datum	`mbcmodel.localmodel`
Local Polynomial	`mbcmodel.localmodel`
Local Hybrid Spline	`mbcmodel.localmodel`
Local Truncated Power Series	`mbcmodel.localmodel`
Local Free Knot Spline	`mbcmodel.localmodel`
Local Multiple Models	`mbcmodel.localmodel`
Local Growth	`mbcmodel.localmodel`
Local User-Defined	`mbcmodel.localmodel`

Type	Model Object
Local Transient	mbcmodel.localmodel
Local Average Fit	mbcmodel.localmodel

You can get a list of types by using `getAlternativeTypes`. Use this syntax.

```
Mlist = getAlternativeTypes(M)
```

where `M` is an `mbcmodel.model` object.

Data Types: `char` | `string`

`NumInputs` — Number of inputs to model
real positive scalar

This property is read-only.

Number of inputs to model, specified as a real positive scalar.

Data Types: `double` | `single`

Examples

Obtain `mbcmodel.modelproperties` Object From Model

Obtain a `mbcmodel.modelproperties` object from a model.

```
>> ModelObj = mbcmodel.CreateModel('Polynomial', 4);
>> ModelProp = ModelObj.Properties

ModelProp =

Polynomial Properties
                   Type: 'Polynomial'
              NumInputs: 4
                  Order: [3 3 3 3]
       InteractionOrder: 3
    TransformInputRange: 1
         ParameterNames: {35×1 cell}
         StepwiseStatus: {35×1 cell}
                 BoxCox: 1
```

Display the model object.

```
>> disp(ModelObj)
   1 + 2*X1 + 17*X2 + 27*X3 + 33*X4 + 3*X1^2 + 8*X1*X2 + 12*X1*X3 + 15*X1*X4 +
   18*X2^2 + 22*X2*X3 + 25*X2*X4 + 28*X3^2 + 31*X3*X4 + 34*X4^2 + 4*X1^3 +
   5*X1^2*X2 + 6*X1^2*X3 + 7*X1^2*X4 + 9*X1*X2^2 + 10*X1*X2*X3 + 11*X1*X2*X4 +
   13*X1*X3^2 + 14*X1*X3*X4 + 16*X1*X4^2 + 19*X2^3 + 20*X2^2*X3 + 21*X2^2*X4 +
   23*X2*X3^2 + 24*X2*X3*X4 + 26*X2*X4^2 + 29*X3^3 + 30*X3^2*X4 + 32*X3*X4^2 +
   35*X4^3
   InputData: [0x4 double]
   OutputData: [0x1 double]
   Status: Not fitted
   Linked to Response: <not linked>
```

Obtain Model Object Properties

Obtain the model object properties.

```
>> ModelObj = mbcmodel.CreateModel('Polynomial', 4);
>> ModelProp = properties(ModelObj)

ModelProp = 

  13×1 cell array

    {'Type'         }
    {'Status'       }
    {'NumInputs'    }
    {'InputData'    }
    {'OutputData'   }
    {'Properties'   }
    {'FitAlgorithm' }
    {'Inputs'       }
    {'Name'         }
    {'Units'        }
    {'Parameters'   }
    {'Response'     }
    {'IsBeingEdited'}
```

List and Change Model Properties

Create a model and list the model properties.

```
>> ModelObj = mbcmodel.CreateModel('RBF',2)

ModelObj =

   A radial basis function network using a multiquadric kernel
   with 0 centers and a global width of 2.
   The regularization parameter, lambda, is 0.0001.
   InputData: [0x2 double]
   OutputData: [0x1 double]
   Status: Not fitted
   Linked to Response: <not linked>

>> properties(ModelObj.Properties)
RBF Properties
              Type: 'RBF'
         NumInputs: 2
            Kernel: 'multiquadric'
        Continuity: 4
    ParameterNames: {0×1 cell}
    StepwiseStatus: {0×1 cell}
            BoxCox: 1

     Kernel: RBF kernel (enum: {'multiquadric','recmultiquadric','gaussian','thinplate',...
          'logisticrbf','wendland','linearrbf','cubicrbf'})
     Continuity: Continuity for Wendland kernel (0,2,4,6) (int: [0,6])
     ParameterNames: List of parameter names (read-only)
     StepwiseStatus: Stepwise status {'Always','Never','Step'} (cell)
     BoxCox: Box-Cox transform (power) (numeric: [-3,3])
```

Return the properties as a cell array.

```
>> f=ModelObj.Properties.properties

f =
```

```
5×1 cell array

  {'Kernel'        }
  {'Continuity'    }
  {'ParameterNames'}
  {'StepwiseStatus'}
  {'BoxCox'        }
```

Version History

Introduced before R2006a

See Also

mbcmodel.project | mbcdoe.design | mbcmodel.data | mbcmodel.model

MBCModel.Testplan

mbcmodel.testplan

Properties and methods for test plan objects

Description

Use these properties and object functions to create and examine test plan objects.

Creation

Create a mbcmodel.testplan object using `CreateTestplan`.

Properties

`BestDesign` — Best design objects
n-by-1 cell array of `mbcdoe.design` objects

n-by-1 cell array of `mbcdoe.design` objects per model level, where *n* is the number of model levels. There can be one best design for each level, but the software uses the best global (2) level design to match to data when you call `AttachData`.

To set the design `globalDesign` as the best design at the global (2) level:

```
>> T.BestDesign{2} = globalDesign
```

Data Types: `double`

`Boundary` — Boundary tree object
scalar `mbcboundary.Tree` object

This property is read-only.

`BoundaryTree = mbcmodel.testplan.Boundary` returns the boundary tree for the test plan. The `BoundaryTree` is a container for all the boundary models you create. `BoundaryTree` is an `mbcboundary.Tree` object.

To get the boundary tree from the test plan `Boundary` property:

```
BoundaryTree = mbcmodel.testplan.Boundary
```

Data Types: `char` | `string`

`Data` — Data objects in test plan
1-by-1 array | 0-by-1 array

This property is read-only.

Data objects to be returned to `mbcmodel.testplan`, specified as an array.

For the test plan object *T*, this code returns a 1-by-1 array if the test plan has a data object attached. Otherwise, the code returns a 0-by-1 array.

```
allD = T.Data;
```

Data Types: char | string

DefaultModels — Default model object per level
n-by-1 cell array of mbcmodel.model objects

n-by-1 cell array of default model objects for each model level, where *n* is the number of model levels.

To get the default model objects for use in creating a response, use this code:

```
>> models = T.DefaultModels
>> LocalModel = CreateModel(models{1}, 'Local Polynomial Spline');
>> GlobalModel = CreateModel(models{2}, 'RBF');
>> R = CreateResponse(T, 'TQ', LocalModel, GlobalModel, 'Maximum')
```

Data Types: char | string

Designs — Design object per level
n-by-1 cell array of mbcdoe.design objects

n-by-1 cell array that contains the number design objects per model level, where *n* is the number of model levels.

When using designs at the command-line, the software treats designs as an array. In the Design Editor, you can build a design tree, where child designs inherit characteristics from the parent design. At the command-line, you can copy and modify designs. By default, designs are added to the top-level of the design tree. To build tree structures at the command-line, use the Parent argument of the AddDesign method. The tree structure cannot be used at the command-line any further, but you can use the design tree in the Design Editor after you load the project into the Model Browser.

To get local designs only:

```
LocalDesigns = T.Designs{1}
```

To get global designs only:

```
GlobalDesigns = T.Designs{2}
```

To get the fifth global design:

```
D = T.Design {2}(5)
```

After modifying the design, you must call UpdateDesign, or reassign to the test plan as follows:

```
T.Design {2}(5) = D
```

Data Types: double

Inputs — Model input
n-by-1 cell array of mbcdoe.modelinput objects

n-by-1 cell array of mbcdoe.modelinput objects per model level, where *n* is the number of model levels.

InputSignalNames — Input signal names in data
n-by-1 array

This property is read-only.

n-by-1 array of character vectors that contains the data signal names that the software is modeling, where *n* is the number of modeled signals.

Example: `["afr" "egr" "load" "n" "spark" "logno" "tq"]`

Data Types: `string`

`InputsPerLevel` — Number of inputs for each level
n-by-1 array

This property is read-only.

n-by-1 array that contains the number of inputs per model level, where *n* is the number of model levels.

For the test plan object *T*, this code indicates that the test plan has 2 local and 4 global inputs.

```
>> L = T.InputsPerLevel
L =
     2  4
```

Data Types: `double`

`Levels` — Number model levels
scalar

This property is read-only.

Number of levels in the hierarchical model.

For more information about levels, see “Understanding Model Structure for Scripting”.

Example: `levels = T.Levels;`

Data Types: `double`

`Name` — Test plan object name
`mbcmodel.testplan` object

Test plan object name, specified as a `mbcmodel.testplan` object.

Example: `'holliday_data.mat'`

Data Types: `char` | `string`

`Project` — Project object name
`mbcmodel.project` object

This property is read-only.

Project object name, specified as a character vector.

Data Types: `char` | `string`

`Responses` — Test plan response objects
n-by-1 array of `mbcmodel.response` objects

This property is read-only.

`n`-by-`1` array that contains the number of test plan responses per model level, where *n* is the number of model levels.

For more information about the relationship between test plans and responses, see “Understanding Model Structure for Scripting”.

Array of test plan, *T*, responses:

```
>>R = T.Responses
```

Data Types: `char` | `string`

`SummaryStatistics` — Summary statistics for all responses
structure

This property is read-only.

Summary of `mbcmodel.response` object statistics, specified as a structure with fields `Statistics` and `Names`. Statistics include validation RMSE.

For more information, see `SummaryStatistics`.

Data Types: `char` | `string`

`ResponseNames` — Response model names
array of `mbcmodel.response` object names

Array of `mbcmodel.response` object names. Use only before attaching data to the test plan.

The response names should match the channel names in the response data set.

Data Types: `string`

`ValidationData` — Test plan validation data
array of `mbcmodel.data` objects

Array of `mbcmodel.data` object test plan validation data.

Data Types: `char` | `string`

Object Functions

AddDesign	Add design to test plan
AttachData	Attach data from project to test plan
BoundaryModel	Get boundary model from test plan
CreateDesign	Create design object for test plan or model
CreateResponse	Create response model for test plan
DetachData	Detach data from test plan
FindDesign	Find test plan design by name
InputSetupDialog	Open Input Setup dialog box to edit inputs
RemoveDesign	Remove design from test plan
UpdateDesign	Update design in test plan

Examples

Create Test Plan Using Template

To create a test plan using a test plan template, enter:

```
T = CreateTestplan(P1, 'd:\MBCwork\TQtemplate1', 'newtestplan')
testplan = CreateTestplan(P, 'example_testplan')
```

To create a test plan using inputs per level, enter:

```
T = P.CreateTestplan([1,2])
```

To specify the input information in a cell array of `mbcmodel.modelinput` objects, enter:

```
% Define Inputs for test plan
LocalInputs = mbcmodel.modelinput('Symbol','S',...
    'Name','SPARK',...
    'Range',[0 50]);
GlobalInputs = mbcmodel.modelinput('Symbol',{'N','L','ICP',...
'ECP'},'Name',{'SPEED','LOAD','INT_ADV','EXH_RET'},...
    'Range',{[500 6000],[0.0679    0.9502],[-5 50],[-5 50]});
% create test plan
testplan = CreateTestplan( project, {LocalInputs,...
GlobalInputs} );
```

Or

```
T = P.CreateTestplan({LocalInputs,GlobalInputs})
```

To specify the input information in a cell array, enter:

```
localInputs = {'S',0,50,'','SPARK'};
globalInputs = {'N',   800, 5000, '', 'ENGSPEED'
           'L',   0.1,    1, '', 'LOAD'
           'EXH',  -5,   50, '', 'EXHCAM'
           'INT',  -5,   50, '', 'INTCAM'};

T = CreateTestplan(P,{localInputs,globalInputs});
```

Version History

Introduced before R2006a

See Also

`CreateTestplan`

Topics

"Load and Modify Data"

AddDesign

Add design to test plan

Syntax

```
DObjMod = AddDesign(TPObj,DObj)
DObjMod = AddDesign(TPObj,Level,DObj)
DObjMod = AddDesign(TPObj,Level,DObj,Parent)
```

Description

`DObjMod = AddDesign(TPObj,DObj)` adds the design object, `DObj`, to the test plan, `TPObj`.

`DObjMod = AddDesign(TPObj,Level,DObj)` adds a design to the test plan and specifies the level, `Level`.

`DObjMod = AddDesign(TPObj,Level,DObj,Parent)` adds a child design to the test plan.

Examples

Add Design to Test Plan

Add three designs to the test plan global (2) level.

```
D = AddDesign(TP, [sfDesign, parkedCamsDesign, mainDesign])
```

Input Arguments

`TPObj` — Test plan
test plan object

Test plan, specified as a project object.

`DObj` — Design objects
`n`-by-`1` array of `mbcdoe.design` objects

Design objects in test plan, specified as a `n`-by-`1` array, where *n* is the number of designs.

`Level` — Test plan level
scalar

Test plan level, specified as a scalar. By default, the level is the outer level: 1 for one-stage designs and 2 (global) for two-stage designs.

`Parent` — Parent design in design tree
`mbcdoe.design` object

Parent design in design tree, specified as a `mbcdoe.design` object. By default, designs are added to the top level of the design tree.

Output Arguments

`DObjMod` — Modified design objects
`n`-by-`1` array of `mbcdoe.design` objects

Modified design objects in test plan, returned as a `n`-by-`1` array of `mbcdoe.design` design objects, where *n* is the number of designs.

Version History

Introduced in R2008a

See Also

`mbcmodel.testplan`

AttachData

Attach data from project to test plan

Syntax

```
DataObjMod = AttachData(TPObj,DataObj)
DoeObjMod = AttachData(TPObj,DataObj,'Name1','Value1',…)
```

Description

`DataObjMod = AttachData(TPObj,DataObj)` attaches data, `DataObj`, from a project to the test plan, `TPObj`.

`DoeObjMod = AttachData(TPObj,DataObj,'Name1','Value1',…)` attaches data from a project to a test plan with values specified by name-value arguments.

When you attach data to a test plan, the `Name` property of the test plan inputs determines the data channels. If `Name` is empty, then `Name` is set to `Symbol`. If `Name` does not exist in the dataset, an error results.

When a test plan has attached data, you can change only the symbols, ranges, or nonlinear transforms of the test plan inputs.

Examples

Attach Data to Test Plan

Use all the data in `DATA` in the test plan `TESTPLAN` and set the input ranges to the data range.

```
newD = AttachData(TESTPLAN, DATA,'usedatarange',true);
```

Match data in `DATA` to the best design in the test plan `TESTPLAN` within specified tolerances.

```
Datatol = [0.075, 100, 1, 2];
unmatch = 'all';
moredata = 'all';
moredes = 'none';
AttachData(testplan, data ,...
    'tolerances',    tol,...
    'unmatcheddata', unmatch,...
    'moredata',      moredata,...
    'moredesign',    moredes);
```

Use data from one project in another project.

```
p1 = mbcmodel.LoadProject( filename );
p2 = mbcmodel.LoadProject( filename2 );
p1.Testplan.AttachData( p2.Data );
```

Input Arguments

`TPObj` — Test plan
test plan object

Test plan, specified as a project object.

`DataObj` — Data objects
`n`-by-`1` array of data objects

Data objects, specified as a `n`-by-`1` array, where *n* is the number of data objects in the test plan.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use the optional name-value arguments to specify how the software attaches the data to a design.

Example: `'usedatarange',true`

`unmatcheddata` — Unmatched data
`all` (default) | `none`

Unmatched data to attach to design, specified as all or none.

Example: `'unmatcheddata',all`

`moredata` — Additional data
`all` (default) | `closest`

Additional data to attach to design, specified as all or closest.

Example: `'unmatcheddata',all`

`moredesign` — Additional designs
`none` (default) | `closest`

Additional designs to attach to design objects, specified as all or closest.

Example: `'moredesign',none`

`tolerances` — Data tolerances
`1`-by-`NumInputs` array

Data tolerances, specified as a `1`-by-`NumInputs` array, where `NumInputs` of is the number of data inputs.

Data Types: `double`

`usedatarange` — Data range
`false` (default) | `true`

Data range for test plan input ranges, specified as false or true.

Example: `'usedatarange',true`

Data Types: `logical`

Output Arguments

`DataObjMod` — Data objects
`n`-by-`1` array of data objects

Data objects in test plan, returned as a `n`-by-`1` array, where *n* is the number of data objects.

Version History

Introduced before R2006a

See Also

`mbcmodel.testplan`

BoundaryModel

Get boundary model from test plan

Syntax

```
Best = BoundaryModel(TPObj)
Best = BoundaryModel(TPObj,Type)
```

Description

`Best = BoundaryModel(TPObj)` returns the best boundary model for the test plan, `TPObj`.

`Best = BoundaryModel(TPObj,Type)` is the best boundary model for the specified type, `Type`, associated with the test plan.

Examples

Add Boundary Constraints to Design

Load boundary constraints from another project file and add the constraints to the design.

```
otherProject = mbcmodel.LoadProject( [matlabroot,...
'\toolbox\mbc\mbctraining\Gasoline_project.mat']);
boundaryConstraints = otherProject.Testplans(1).Boundary.Global.BestModel
Design.Constraints = boundaryConstraints;
```

When you add the constraints to the design, the boundary model object converts to an `mbcdoe.designconstraint` object.

Input Arguments

`TPObj` — Test plan
test plan object

Test plan, specified as a project object.

`Type` — Best model for test plan type
`'all'` (default) | `'local'` | `'global'`

Best boundary model for the test plan type, specified as:

- `'all'`: Best boundary model for all inputs
- `'local'`: Best local boundary model
- `'global'`: Best global boundary model

Output Arguments

`Best` — Best boundary model
boundary model objects

Best boundary model for test plan, returned as boundary model objects. `Best` is a boundary model subclass of `mbcboundary.AbstractBoundary`: `mbcboundary.Model`, `mbcboundary.Boolean`, `mbcboundary.PointByPoint`, or `mbcboundary.TwoStage`.

Note Before release 2009b, `BoundaryModel` returned an `mbcdoe.designconstraint` object. Use `designconstraint` to convert a boundary to a design constraint.

Version History

Introduced in R2008a

See Also

`mbcmodel.testplan`

CreateResponse

Create response model for test plan

Syntax

```
R = CreateResponse(TPObj,VarName)
R = CreateResponse(TPObj,VarName,Model)
R = CreateResponse(TPObj,VarName,LocalModel,GlobalModel)
R = CreateResponse(TPObj,VarName,LocalModel,GlobalModel,DatumType)
```

Description

`R = CreateResponse(TPObj,VarName)` creates a response model, `VarName`, using the test plan default models, `TPObj`.

`R = CreateResponse(TPObj,VarName,Model)` creates a one-stage response model, `Model`, where `TPObj` is a one-stage test plan object.

`R = CreateResponse(TPObj,VarName,LocalModel,GlobalModel)` creates a two-stage response model specified by `LocalModel` and `GlobalModel`.

`R = CreateResponse(TPObj,VarName,LocalModel,GlobalModel,DatumType)` creates a two-stage response model with a local datum model specified by `DatumType`.

Examples

Create Response Models

Create a response using the default models.

```
R = CreateResponse(T, 'torque');
TQ_response = CreateResponse(testplan, 'TQ');
```

Create a response and specify the local and global model types.

```
models = T.DefaultModels
LocalModel = CreateModel(models{1}, 'Local Polynomial Spline');
GlobalModel = CreateModel(models{2}, 'RBF');
R = CreateResponse(T, 'TQ', LocalModel, GlobalModel, 'Maximum')
```

Input Arguments

`TPObj` — Test plan
test plan object

Test plan, specified as a project object.

`VarName` — Response model variable name
`character vector`

Response model variable name, specified as a character vector.

Data Types: `char` | `string`

`Model` — One-stage model object
`mbcmodel.model` object

One-stage model object, specified as a `mbcmodel.model` object.

Data Types: `char` | `string`

`LocalModel` — Local model object
`mbcmodel.localmodel` object

Local model object, specified as a `mbcmodel.localmodel` object.

Data Types: `char` | `string`

`GlobalModel` — Response feature model object
`mbcmodel.response` object

Response feature model object, specified as a `mbcmodel.response` object.

Data Types: `char` | `string`

`DatumType` — Datum type
`'None'` | `'Maximum'` | `'Minimum'` | `'Linked'`

Datum type, specified as 'None', 'Maximum', 'Minimum', or 'Linked'. You can specify a datum type if the local model type allows a datum model. Model types Polynomial Spline and Polynomial with Datum allow datum models.

Data Types: `char` | `string`

Output Arguments

`R` — Response model object
`mbcmodel.response` object

Response model object, returned as a `mbcmodel.response` object.

Data Types: `char` | `string`

Version History

Introduced before R2006a

See Also

`mbcmodel.testplan`

DetachData

Detach data from test plan

Syntax

```
TPObjMod = DetachData(TPObj)
```

Description

`TPObjMod = DetachData(TPObj)` detaches data from the test plan `TPObj`. A test plan can only use a single dataset, so you do not need to specify the data object.

Input Arguments

`TPObj` — Test plan
test plan object

Test plan, specified as a project object.

Output Arguments

`TPObjMod` — Modified test plan object
test plan object

Modified test plan object, returned as a test plan object.

Version History

Introduced before R2006a

See Also

`mbcmodel.testplan`

RemoveDesign

Remove design from test plan

Syntax

```
DObjMod = RemoveDesign(TPObj,DObj)
DObjMod = RemoveDesign(TPObj,Level,DObj)
DObjMod = RemoveDesign(TPObj,Parent,DObj)
DObjMod = RemoveDesign(TPObj,Name)
DObjMod = RemoveDesign(TPObj,Level,Name)
```

Description

`DObjMod = RemoveDesign(TPObj,DObj)` removes the design, `DObj`, from the test plan, `TPObj`.

`DObjMod = RemoveDesign(TPObj,Level,DObj)` removes a design at the specified level, `Level`.

`DObjMod = RemoveDesign(TPObj,Parent,DObj)` removes the child design from the test plan.

`DObjMod = RemoveDesign(TPObj,Name)` removes a design with the matching name, `Name`, from the test plan.

`DObjMod = RemoveDesign(TPObj,Level,Name)` removes a design with a matching name from the specified level of the test plan.

Input Arguments

`TPObj` — Test plan
test plan object

Test plan, specified as a project object.

`DObj` — Design objects
`n`-by-1 array of `mbcdoe.design` objects

Design objects in test plan, specified as a `n`-by-`1` array, where *n* is the number of designs.

`Level` — Test plan level
scalar

Test plan level, specified as a scalar. By default, the level is the outer level: 1 for one-stage designs and 2 (global) for two-stage designs.

`Parent` — Parent design in design tree
`mbcdoe.design` object

Parent design in design tree, specified as a `mbcdoe.design` object. By default, designs are added to the top level of the design tree.

`Name` — Data name
character vectors

Data name, specified as character vectors.

Data Types: `string`

Output Arguments

`DObjMod` — Modified design objects
`n`-by-`1` array of `mbcdoe.design` objects

Modified design objects in test plan, returned as a `n`-by-`1` array of `mbcdoe.design` design objects, where *n* is the number of designs.

Version History

Introduced in R2008a

See Also

`mbcmodel.testplan`

FindDesign

Find test plan design by name

Syntax

```
DObj = FindDesign(TPObj,Name)
DObj = FindDesign(TPObj,Level,Name)
```

Description

`DObj = FindDesign(TPObj,Name)` finds a design with the matching name, `Name`, from the test plan, `TPObj`.

`DObj = FindDesign(TPObj,Level,Name)` finds a design with a matching name from the specified test plan level, `Level`.

Input Arguments

`TPObj` — Test plan
test plan object

Test plan, specified as a project object.

`Name` — Test plan object name
`mbcmodel.testplan` object

Test plan object name, specified as a `mbcmodel.testplan` object.

Example: `'holliday_data.mat'`

Data Types: `char` | `string`

`Level` — Test plan level
scalar

Test plan level, specified as a scalar. By default, the level is the outer level: 1 for one-stage designs and 2 (global) for two-stage designs.

Output Arguments

`DObj` — Design objects
n-by-1 array of `mbcdoe.design` objects

Design objects in test plan, specified as a n-by-1 array, where *n* is the number of designs.

Version History

Introduced in R2008a

See Also

`mbcmodel.testplan`

UpdateDesign

Update design in test plan

Syntax

```
DObjMod = UpdateDesign(TPObj,DObj)
DObjMod = UpdateDesign(TPObj,Level,DObj)
```

Description

`DObjMod = UpdateDesign(TPObj,DObj)` updates the design, `DObj`, in the test plan, `TPObj`.

`DObjMod = UpdateDesign(TPObj,Level,DObj)` updates the design at the specified level, `Level`.

You must call `UpdateDesign` to replace an edited design in the test plan.

Input Arguments

`TPObj` — Test plan
test plan object

Test plan, specified as a project object.

`DObj` — Design objects
`n`-by-`1` array of `mbcdoe.design` objects

Design objects in test plan, specified as a `n`-by-`1` array, where *n* is the number of designs.

`Level` — Test plan level
scalar

Test plan level, specified as a scalar. By default, the level is the outer level: 1 for one-stage designs and 2 (global) for two-stage designs.

Output Arguments

`DObjMod` — Modified design objects
`n`-by-`1` array of `mbcdoe.design` objects

Modified design objects in test plan, returned as a `n`-by-`1` array of `mbcdoe.design` design objects, where *n* is the number of designs.

Version History

Introduced in R2008a

See Also

`mbcmodel.testplan`

MBCModel.RBFModelParameters

mbcmodel.rbfmodelparameters

Properties and methods for RBF model objects

Description

Use these properties and object functions to create and examine Radial Basis Function (RBF) model objects.

Creation

Create a mbcmodel.rbfmodelparameters object using `CreateModel`.

Properties

`Centers` — Model centers
`c`-by-`n` array

This property is read-only.

Model centers, specified as a `c`-by-`n` array, where `n` is the number of model variables and `c` is the number of centers.

Data Types: `double`

`Widths` — Model width
scalar | `1`-by-`n` array | `c`-by-`n` array

This property is read-only.

Widths is usually a scalar, but it can also be a:

- `1`-by-`n` array, for width per dimension algorithms, where `n` is the number of variables.
- `c`-by-`n`, for tree regressions, where `n` is the number of variables and `c` is the number of centers.

Data Types: `double`

Version History

Introduced before R2006a

See Also

`mbcmodel.linearmodel`

mbcboundary

mbcboundary.TwoStageTree

Root boundary tree class in two-stage test plans

Description

The `mbcboundary.TwoStageTree` class is a subclass of the `mbcboundary.AbstractBoundary` class. The `mbcboundary.AbstractBoundary` class is the base class for all boundary model classes in the Model-Based Calibration Toolbox.

Creation

You access the boundary tree from the `Boundary` property of `mbcmodel.testplan`. The root of the boundary tree for two-stage test plans contains boundary trees specified as `mbcboundary.Tree` objects for local, global, and response boundary models in the `Local`, `Global`, and `Response` properties, respectively.

Properties

`Local` — Local boundary model tree
object

This property is read-only.

Local boundary model tree, specified as an object. Point-by-point and two-stage boundary models are fitted in the local boundary model tree. These boundary models fit local boundary models for each operating point and combine into a single boundary model that includes the global inputs.

`Global` — Global boundary model tree
object

This property is read-only.

Global boundary model tree, specified as an object. Boundary models in the global model boundary tree are fitted with one point per test, specifically, the average value of the global variables for that test.

`Response` — Response boundary model tree
object

This property is read-only.

Response boundary model tree, specified as an object. Boundary models in the response model boundary tree are fitted with all local and global input data for the test plan.

`BestModel` — Combined best boundary models
object

This property is read-only.

Combined best boundary models, specified as an object. `BestModel` is the boundary model combining the best local, global, and response boundary models. You can select which boundary models to include in the best model with `InBest`. If the best boundary model includes more than one boundary model, that boundary model is an `mbcboundary.Boolean` object.

`InBest` — Boundary models selected as best

array

Boundary models selected as best, specified as an array. You can combine local, global, and response boundary models into a single boundary model for the test plan. The logical array specifies whether to include, in order, the best local, global, and response boundary models, in the best boundary model for the test plan. The `BestModel` property gives the best boundary model for the test plan.

Data Types: `logical`

`TestPlan` — Test plan containing boundary tree

character vector

This property is read-only.

Test plan containing boundary tree, specified as a character vector. You can combine local, global, and response boundary models into a single boundary model for the test plan. The logical array specifies whether to include, in order, the best local, global, and response boundary models, in the best boundary model for the test plan. The `BestModel` property gives the best boundary model for the test plan.

Data Types: `string` | `char`

More About

Usage

`mbcboundary.TwoStageTree` is a method of the `mbcboundary.AbstractBoundary` class and all its subclasses.

Version History

Introduced in R2009b

See Also

`mbcboundary.Tree` | `mbcboundary.TwoStage`

mbcboundary.TwoStage

Two-stage boundary model class

Description

The `mbcboundary.TwoStage` class is a subclass of the `mbcboundary.AbstractBoundary` class. The `mbcboundary.AbstractBoundary` class is the base class for all boundary model classes in the Model-Based Calibration Toolbox.

Creation

You can only create and fit two-stage boundary models in the local boundary tree. You cannot create or fit two-stage boundary models outside of a project. Fit one of these boundary models by adding the boundary model to the boundary tree.

Local boundary model parameters are fitted using interpolating RBFs for global inputs. Two-stage boundary models are valid at any operating point.

For an example, see "Create Two-Stage Boundary Model" on page 10-6.

Properties

`FitAlgorithm` — Fit algorithm for model
array

This property is read-only.

Fit algorithm for model, specified as an array.

`FitAlgorithm` is a property of the `mbcboundary.TwoStage` class, and boundary model objects `mbcboundary.AbstractBoundary` and all subclasses.

An `mbcmodel.model.FitAlgorithm` object is contained within the `FitAlgorithm` property of an `mbcmodel.model` object or `mbcboundary` object.

As an alternative to using `CreateAlgorithm`, you can assign the algorithm name directly to the algorithm.

```
B.FitAlgorithm.BoundaryPointOptions = 'Boundary Only';
```

```
m.FitAlgorithm = 'Minimize PRESS';
```

Case and spaces are ignored.

To get a `fitalgorithm` object, `F`, from a model, use this code.

```
M = mbcmodel.CreateModel('Polynomial', 4);
F = M.FitAlgorithm
```

```
F =
Algorithm: Least Squares
Alternatives: 'Minimize PRESS','Forward Selection','Backward
Selection','Prune'
1x1 struct array with no fields.
```

`Fitted` — Whether boundary is fitted
logical

This property is read-only.

`Fitted` is a property of the `mbcboundary.AbstractBoundary` class and all its subclasses.

`Fitted(B)` indicates whether boundary model `B` has been fitted. You cannot evaluate the boundary model unless `Fitted` equals `true`.

`GlobalModel` — Interpolating global boundary model definition
`mbcmodel` object

This property is read-only. You cannot change the `GlobalModel`, however, you can change the model properties or fit algorithm.

Interpolating global boundary model definition, specified as a `mbcmodel` object. `B.GlobalModel` returns the definition of global boundary model. `GlobalModel` requires the type `Interpolating RBF`.

`LocalModel` — Local boundary model definition
`mbcboundary` object

This property is read-only.

Local boundary model definition, specified as a `mbcboundary` object. `B.LocalModel` returns the boundary model that fits each test. Two-stage boundary models support only `Range` and `Ellipsoid` local boundary models.

Use `CreateBoundary` to change the local boundary model, for example:

```
b.LocalModel = CreateBoundary(b.LocalModel,'Range')
```

Note `Ellipsoid` requires two inputs.

`Inputs` — Boundary model input
`mbcboundary` object

This property is read-only.

Boundary model input, specified as an `mbcboundary` object.

For `mbcboundary` objects, this property returns an `mbcmodel.modelinput` object. You cannot edit this object when it is attached to a response. You cannot change the number of inputs after creation.

`Name` — Boundary model object name
`mbcboundary` object

This property is read-only.

Boundary model object name, specified as an `mbcboundary` object. Names of boundary models are read-only and provide a description of the boundary model type and active inputs.

`NumInputs` — Number of boundary model object inputs

`mbcboundary` object

This property is read-only.

Number of boundary model object input, specified as a `mbcboundary` object.

`Type` — Type of boundary model

`mbcboundary` object

`Type` is a property of the `mbcboundary.AbstractBoundary` class and all subclasses.

`B.Type` returns the boundary model type.

Use `getAlternativeTypes` to find out what types are available for the specified boundary model.

Available types depend on the boundary model.

- For `mbcboundary.TwoStage`, the `LocalModel` requires a type of either `Range` or `Ellipsoid`. `GlobalModel` requires a type of `Interpolating RBF` only.
- For `mbcboundary.PointByPoint`, the `LocalModel` type can be any valid type for `mbcboundary.Model`.

Note You can only create boundaries of type `'Point-by-point'` or `'Two-stage'` from a local boundary tree or from an existing boundary of type `'Point-by-point'` or `'Two-stage'`. You cannot create or fit these types of boundary models outside of a project. Fit them by adding the boundary model to the boundary tree.

Object Functions

CreateBoundary	Create boundary model
designconstraint	Convert boundary model to design constraint
evaluate	Evaluate model, boundary model, or design constraint
getAlternativeTypes	Alternative boundary models
getLocalBoundary	Get local boundary model at operating point

Examples

Create Two-Stage Boundary Model

Create a two-stage boundary model and add it to a project.

1. Open a project with a test plan.

```
proj = mbcmodel.LoadProject(fullfile(mbcpath,'mbctraining','Gasoline_project.mat'));
testPlan = proj.Testplans(1);
tree = testPlan.Boundary
```

```
tree =

  TwoStageTree with properties:
```

```
          Local: [1×1 mbcboundary.Tree]
         Global: [1×1 mbcboundary.Tree]
       Response: [1×1 mbcboundary.Tree]
      BestModel: [1×1 mbcboundary.Boolean]
         InBest: [1 1 1]
       TestPlan: [1×1 mbcmodel.testplan]
```

2. Crete a two-stage local boundary model.

```
twostageBoundary = CreateBoundary(tree.Local,'Two-stage');
```

3. You can modify the `twostageBoundary` properties. However, the `twostageBoundary` boundary model is not fitted until it is added to the `Local` part of the two-stage tree. Use `Add` or `Update` to add it to the two-stage tree.

```
twostageBoundary = Add(tree.Local,twostageBoundary);
```

4. Extract the local boundary at an operating point (Speed=3000, load=0.7, ICP=0, ECP=50). The result is a `Range` boundary model.

```
localBoundary = getLocalBoundary(twostageBoundary,[3000,0.7,0,50])
```

```
localBoundary = 

  Model with properties:

    ActiveInputs: 1
            Name: '8.55<=S<=50'
       NumInputs: 1
            Type: 'Range'
          Inputs: [1×1 mbcmodel.modelinput]
    FitAlgorithm: [1×1 mbcmodel.fitalgorithm]
```

More About

Usage

`mbcboundary.TwoStage` is a method of the `mbcboundary.AbstractBoundary` class and all its subclasses.

Version History

Introduced in R2009b

See Also

`mbcmodel.testplan` | `mbcboundary.TwoStageTree` | `mbcmodel.project`

CreateBoundary

Create boundary model

Syntax

```
BndMdl = CreateBoundary(Type,Inputs)
BndMdl = CreateBoundary(Type,Inputs,Property,Value,...)
BndMdl = CreateBoundary(Tree)
BndMdl = CreateBoundary(Tree,Type)
BndMdl = CreateBoundary(Tree,Type,Property,Value,...)
NewBndMdl = CreateBoundary(OrgBndMdl,Type)
NewBndMdl = CreateBoundary(OrgBndMdl,Type,Property,Value,...)
```

Description

`BndMdl = CreateBoundary(Type,Inputs)` is a static package function that creates an `mbcboundary.Model` object of the specified `Type`. Use this syntax to create a new boundary model object independent of any project. See `fit` for alternatives.

`BndMdl = CreateBoundary(Type,Inputs,Property,Value,...)` creates a boundary with the specified properties.

`BndMdl = CreateBoundary(Tree)` creates a new boundary model, `BndMdl`, from the `mbcboundary.Tree` object, `Tree`. The method uses the test plan inputs to define the boundary model inputs. You must call `Add` to add the new model to the tree.

`BndMdl = CreateBoundary(Tree,Type)` creates a new boundary model of the specified `Type`.

`BndMdl = CreateBoundary(Tree,Type,Property,Value,...)` creates a boundary with the specified properties.

`NewBndMdl = CreateBoundary(OrgBndMdl,Type)` creates a new boundary model with the same inputs as the current boundary model, `OrgBndMdl`.

`NewBndMdl = CreateBoundary(OrgBndMdl,Type,Property,Value,...)` creates a new boundary model with the specified properties.

Examples

Create Boundary Model Outside Project

You can create a boundary model outside of a project in either of these ways.

```
B = mbcboundary.Fit(Data,Type);
```

```
B = mbcboundary.CreateBoundary(Type,Inputs)
```

Create Boundary Model In Project

Create a new boundary model, B, from the mbcboundary.Tree object, Tree. The method uses the test plan inputs to define the boundary model inputs.

```
Tree = testplan.Boundary
B = CreateBoundary(Tree)
```

Create a star-shaped global boundary model for a test plan.

```
B = CreateBoundary(testplan.Boundary.Global,'Star-shaped');
```

Add the boundary model to the test plan. Fit the boundary model. The best boundary model for the tree includes this boundary model.

```
B = Add(testplan.Boundary.Global,B);
```

Create Boundary Models for Point-by-Point Test Plan

Create boundary models for a point-by-point test plan.

```
B = TP.Boundary.Local.CreateBoundary('Point-by-point');
% Use convex hull type for the local boundaries
B.LocalModel = CreateBoundary(B.LocalModel,'Convex hull');
% Add point-by-point boundary model to project.
TP.Boundary.Local.Add(B);
```

Create Point-by-Point Boundary Model from Local Boundary

Creates a point-by-point boundary model from the local boundary tree.

```
B = CreateBoundary(T.Boundary.Local,'Point-by-point');
```

Create a local boundary with type Range.

```
B.LocalModel = (B.LocalModel,'Range');
```

Input Arguments

Type — Type of boundary model
mbcboundary object

Type is a property of the mbcboundary.AbstractBoundary class and all subclasses.

B.Type returns the boundary model type.

Use getAlternativeTypes to find out what types are available for the specified boundary model.

Available types depend on the boundary model.

- For mbcboundary.TwoStage, the LocalModel requires a type of either Range or Ellipsoid. GlobalModel requires a type of Interpolating RBF only.

- For `mbcboundary.PointByPoint`, the `LocalModel` type can be any valid type for `mbcboundary.Model`.

Note You can only create boundaries of type `'Point-by-point'` or `'Two-stage'` from a local boundary tree or from an existing boundary of type `'Point-by-point'` or `'Two-stage'`. You cannot create or fit these types of boundary models outside of a project. Fit them by adding the boundary model to the boundary tree.

`Inputs` — Model input
`mbcmodel.modelinput` object

Model input, specified as a `modelinput` object.

`Tree` — Instance of `mbcboundary.Tree` class object
`mbcboundary.Tree` object

Instance of `mbcboundary.Tree` class object, specified as an `mbcboundary.Tree` object.

Data Types: `char` | `string`

`Property` — Boundary model property name
character vector

Boundary model property name, specified as a character vector. The allowed values depend on the boundary model type.

Example: `Global,'Star-shaped'`

`OrgBndMdl` — Boundary model
boundary model object

Boundary model, specified as a boundary model object.

Data Types: `char` | `string`

Output Arguments

`BndMdl` — Boundary model
object

Boundary model object created by `CreateBoundary`, returned as a boundary model object.

Data Types: `char` | `string`

`NewBndMdl` — New boundary model
object

New boundary model object created by `CreateBoundary`, returned as a boundary model object.

Data Types: `char` | `string`

Version History

Introduced in R2009b

See Also

`mbcboundary.Model` | `mbcboundary.Tree` | `mbcboundary.TwoStageTree` | `mbcboundary.TwoStage` | `Add` | `fit`

designconstraint

Convert boundary model to design constraint

Syntax

`C = designconstraint(OrgBndMdl)`

Description

`C = designconstraint(OrgBndMdl)` converts the boundary model `OrgBndMdl` to an `mbcdoe.designconstraint` object. Convert boundary models to use them as a design constraint. You cannot convert the boundary model to a design constraint until it is fitted (`Fitted=true`).

Input Arguments

`OrgBndMdl` — Boundary model
boundary model object

Boundary model, specified as a boundary model object.

Data Types: `char` | `string`

Output Arguments

`C` — Design constraint
object

Design constraint object created by `designconstraint`, returned as a design constraint object.

Data Types: `char` | `string`

Version History

Introduced in R2009b

See Also

`mbcboundary.Model` | `mbcboundary.Tree` | `mbcboundary.TwoStageTree` | `mbcboundary.TwoStage`

getLocalBoundary

Get local boundary model at operating point

Syntax

`getLocalBoundary(TwoStageBoundary,OperatingPoint)`

Description

`getLocalBoundary(TwoStageBoundary,OperatingPoint)` returns the local two-stage boundary model, `TwoStageBoundary`, at the operating point, `OperatingPoint`.

Input Arguments

`TwoStageBoundary` — Two-stage boundary model
two-stage boundary model object

Two-stage boundary model, `Range` or `Ellipsoid`, specified as a boundary model object.

Data Types: `char` | `string`

`OperatingPoint` — Operating point
global inputs for test plan

Operating points, specified as the global inputs for the test plan.

Data Types: `char` | `string`

Version History

Introduced in R2009b

See Also

`mbcboundary.TwoStage`

getAlternativeTypes

Alternative boundary models

Syntax

`List = getAlternativeTypes(Boundary)`

Description

`List = getAlternativeTypes(Boundary)` returns a cell array of alternative boundary models with the same number of inputs as `Boundary`.

Input Arguments

`Boundary` — Instance of `mbcboundary.AbstractBoundary` class or subclass
`mbcboundary.AbstractBoundary` object

Instance of `mbcboundary.AbstractBoundary` class or subclass, specified as a `mbcboundary.AbstractBoundary` object.

Output Arguments

`List` — List of alternative candidate sets
n-by-1 array

List of alternative candidate for the current candidate set, returned as an n-by-1 array.

Version History

Introduced in R2007a

See Also

`CreateBoundary` | `mbcboundary.TwoStage`