

# **Dymola**

Dynamic Modeling Laboratory

## User's Manual Dymola 6 Additions

Version 6.0

© Copyright 1992-2006 by Dynasim AB. All rights reserved.  
Dymola™ is a trademark of Dynasim AB.  
Dymola® is a registered trademark of Dynasim AB in Sweden.  
Modelica® is a registered trademark of the Modelica Association.

Dynasim AB  
Research Park Ideon  
SE-223 70 Lund  
Sweden

E-mail: [support@Dynasim.com](mailto:support@Dynasim.com)  
URL: <http://www.Dynasim.com>  
Phone: +46 46 2862500  
Fax: +46 46 2862501

# Contents

<b>Recent features in Dymola.....</b>	<b>9</b>
Graphical editor .....	9
Parameter dialog.....	9
Package browser.....	15
Component browser.....	16
Replaceable components .....	17
Connections .....	18
Graphical editing .....	19
HTML documentation .....	22
Settings and options.....	24
Modelica text editor.....	26
Variable declarations .....	26
Editor context menu .....	28
Other operations in text editor .....	30
Used classes.....	30
Modelica language .....	31
Arrays .....	31
Conditional declarations.....	32
Checking for structural singularities.....	32
Improvements in diagnostics .....	34
Evaluation of parameters .....	34
Dynamics state selection .....	34
Storing of protected variables.....	35
Other .....	35
Simulation .....	36

Commands menu .....	36
Simulation windows .....	36
Improvements in interactive functions .....	37
Minor improvements .....	38
Diagram layer in simulation mode .....	40
Improved experiment setup .....	41
Output of manipulated equations in Modelica format .....	42
Discriminating start values .....	51
Bounds checking for variables .....	53
Traceback message for errors in functions .....	53
Direct link in error log to variables in model window .....	55
Extended online diagnostics for non-linear systems .....	56
Extended diagnostics for stuck simulation .....	58
Ensuring that ‘Stop’ stops the simulation .....	59
New integration algorithms .....	60
Analytic Jacobians .....	62
Commands and Scripting .....	63
Plotting and animation .....	65
Variable browser context menu .....	65
Display units .....	66
Other plotting .....	67
Animation .....	67
Matlab and Simulink .....	68
Libraries .....	68
Modelica Standard Library version 2.2 .....	68
Comparison to Modelica Standard Library 1.6 .....	70
Other libraries .....	71
Library handling improvements .....	72
Installation and setup of Dymola .....	72
<b>Modelica Data Structures and GUI .....</b>	<b>77</b>
Records and dialogs .....	77
Tabs and Groups .....	80
Labels and layout .....	82
Alternative forms for input fields .....	83
Illustrations and formatting in dialogs .....	85
Declare variable dialog .....	87
Specialized GUI widgets .....	88
Checking of input data .....	89
Arrays of records .....	91
<b>Visualize 3D .....</b>	<b>95</b>
Introduction .....	95
Inserting and removing objects .....	97
Basic primitives .....	106
Surface Plots .....	109
<b>Model Experimentation .....</b>	<b>121</b>
Introduction .....	121
Varying parameters of a model .....	122

Case Study: CoupledClutches model.....	122
Perturb parameters.....	123
Sweep One Parameter – two variants.....	128
Sweep Two parameters.....	132
Monte Carlo Analysis.....	134
Data Preprocessing.....	142
Setting up for preprocessing.....	142
Limiting and detrending signals.....	146
Analysing Signals: is there any noise?.....	148
Filtering signals.....	150
<b>Model calibration.....</b>	<b>155</b>
Introduction.....	155
The basics of setting up and executing a calibration task.....	156
Vehicle data.....	157
Vehicle model.....	159
Validation of the nominal model.....	161
Measurement file formats.....	167
Calibration.....	169
Free start values.....	171
Tune the parameters.....	172
Validation using measurements from first gear.....	173
The setup as Modelica code.....	174
Saving the setup for reuse.....	175
Reusing a setup for a similar operation.....	176
Analysing parameter sensitivities and dependencies.....	177
Sweep one parameter – sweepParameter.....	177
Sweep two parameters – sweepTwoParameters.....	181
Response to parameter perturbations - perturbParameters.....	182
Check if tuners can be calibrated – checkCalibrationSensitivity.....	184
<b>Design optimization.....</b>	<b>189</b>
Introduction.....	189
First optimization setup.....	191
Multi-criteria experimenting.....	204
Multi-case optimization.....	205
<b>Model Management.....</b>	<b>213</b>
Version management.....	213
The context of version management.....	213
Scope of implementation.....	215
Supported features.....	215
Selecting version management system.....	217
Version management using CVS.....	218
An example of file management using CVS.....	219
Version management using SVN.....	225
An example of file management using SVN.....	227
References.....	228
Model dependencies.....	229
Cross-reference options.....	230

Encryption in Dymola .....	230
Introduction .....	230
Visible and concealed classes .....	231
Developing encrypted libraries.....	232
Using encrypted components.....	233
Examples .....	234
Special annotations for concealment .....	239
Scrambling in Dymola.....	240

# **Recent features in Dymola**





# Recent features in Dymola

Dymola is constantly improved in order to better support building, browsing, simulating, debugging, and understanding large complex models. This chapter describes features that are not yet included in the manual.

---

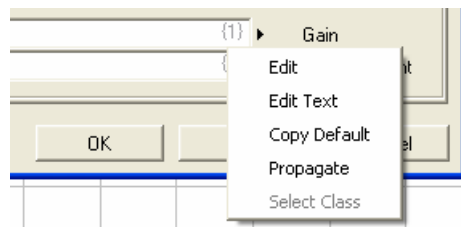
## Graphical editor

The graphical editor has been improved to better support editing and browsing of large complex models.

### Parameter dialog

The parameter dialog automatically gets a scrollbar for any dialog-tab that would otherwise be too large for the screen.

The context menu for a parameter field is also available from a triangular shaped button.



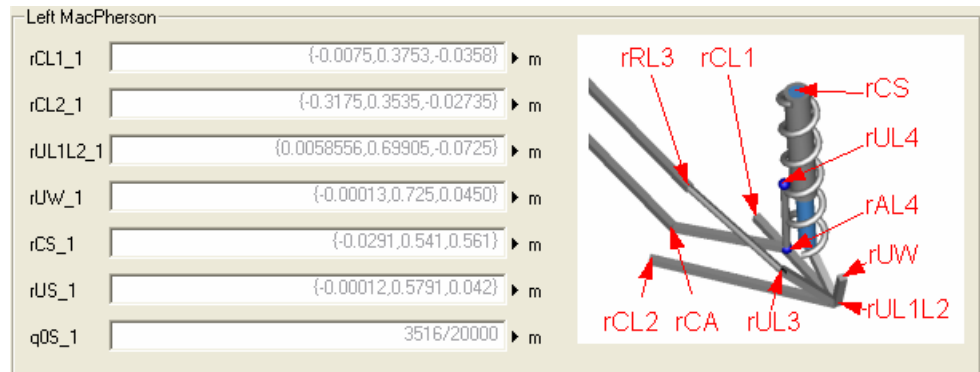
- The command “Copy Default” copies the default value to the input field for further editing.
- A line editor, “Edit Text”, is available for long parameter expressions and redeclarations.
- Command to “Propagate” a parameter in the context menu for parameters, i.e. insert a parameter declaration in the enclosing model and bind the parameter to it. A variable declaration dialog is displayed, see “Variable declarations” below.

The parameter dialog also has an “Info” button that shows Modelica documentation using an external browser.

Graphical illustrations can be included to show meaning of parameters. The syntax for adding images to a group in the parameter dialog is:

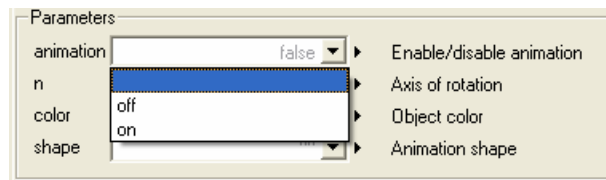
```
annotation ( Images( Parameters( tab="Geometry" ,
    group="Left MacPherson" ,
    source="images/MacPherson_text.png" ) ) );
```

In this example for the Tab “Geometry” and the Group “Left MacPherson” we wanted to add an illustration showing the meaning of parameters.



### Menu for choosing constants

It is possible to annotate parameters or parameter types in order that it's possible to make a selection from a set of values from a pull down menu. For example, setting a parameter true or false can be made by selecting on or off as shown below.



The needed declarations for this appearance are:

```

type OnOff = Boolean annotation (choices(
  choice=false "off",
  choice=true "on"));
parameter OnOff animation=false
  "Enable/disable animation";

```

The following examples show similar choices from a set of predefined vectors representing different common directional axes or commonly used colors. In the example to the right, a selection has been among a set of strings.

animation	false
n	{1,0,0}
color	
shape	{1,0,0} - x axis {0,1,0} - y axis {0,0,1} - z axis any axis

animation	false
n	{1,0,0}
color	{1,0,0}
shape	red green blue

animation	false
n	{1,0,0}
color	{1,0,0,0.9}
shape	none box cylinder

The corresponding declarations are:

```

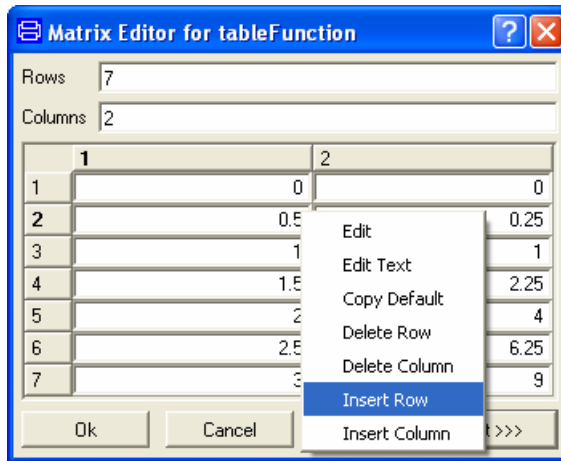
type Axis = Real[3] annotation (choices(
  choice={1,0,0} "{1,0,0} - x axis",
  choice={0,1,0} "{0,1,0} - y axis",
  choice={0,0,1} "{0,0,1} - z axis",
  choice={0,0,0} "any axis"));
parameter Axis n={1,0,0} "Axis of rotation";
type Color = Real[3] annotation (choices(
  choice={1,0,0} "red",
  choice={0,1,0} "green",
  choice={0,0,1} "blue"));
parameter Color color={1,0,0} "Object color";
type Shape = String annotation (choices(
  choice="" "none",
  choice="box" "box",
  choice="cylinder" "cylinder"));
parameter Shape shape="" "Animation shape";

```

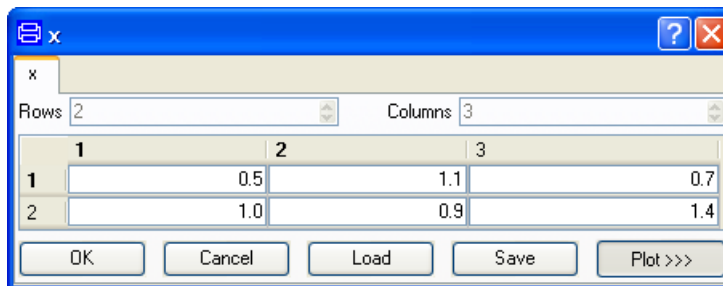
It should be noted that, it's possible to enter any value without using the pull down menu. This enables the use of expressions, for example.

### Structured editor for parameter fields

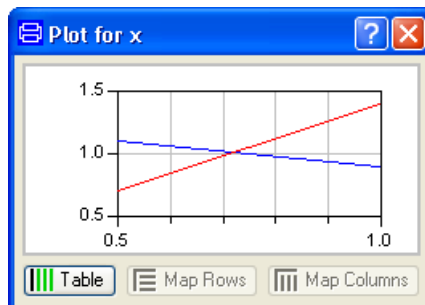
Dymola has specialized array and matrix editors for entering structured parameters. The context menu is available to insert (before selected entry) and delete rows and columns. Insert after the last entry is performed by increasing the size.



Data for the table can be loaded from external file in Matlab, CSV (Comma Separated Values) or text format. The table contents can also be saved on file.



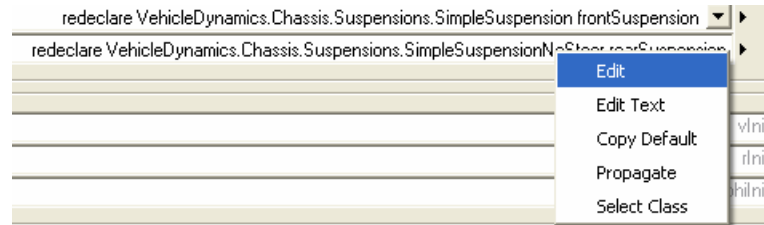
Matrix editor has extended dialog for plotting one- and two-dimensional table functions. The plot is displayed by pressing the “Plot >>>” button.



A special editor for array of records is available if possible as “Edit combined” in context menu.

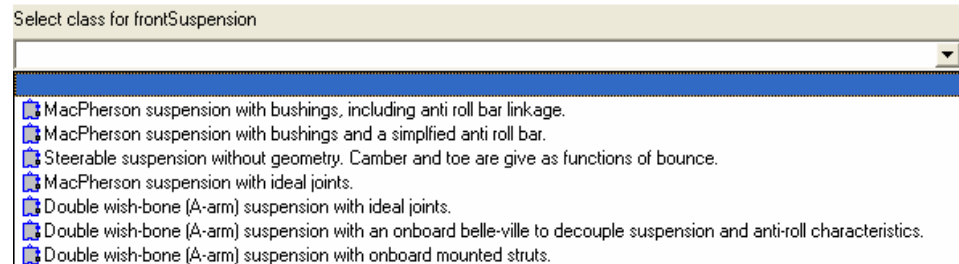
## Replaceable components

The context menu for a replaceable component allows editing of the redeclaration modifier.



This gives the parameter dialog for frontSuspension, see also section ‘Replaceable components’ on page 17.

A class selector for redeclare is available in context menu as ‘Select Class’. All matching classes are listed.

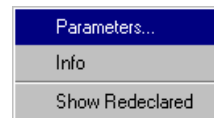


It is possible to set that all choices generated by “choicesAllMatching” and “choicesFromPackage” are replaceable:

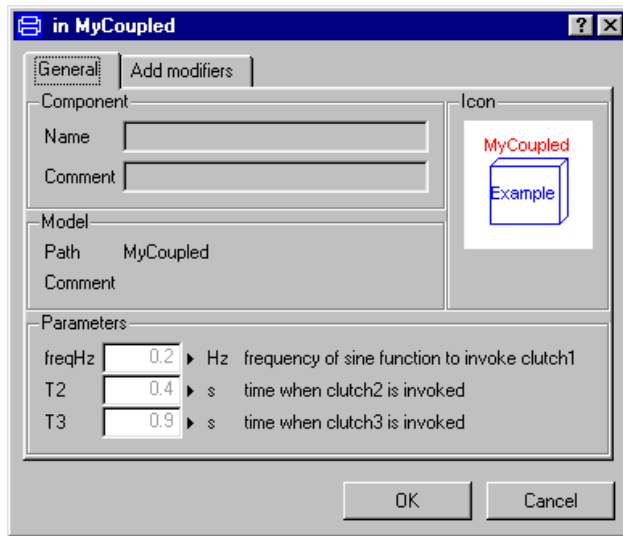
```
Advanced.AutomaticChoicesAreReplaceable=true;
```

## Parameter dialog for inherited parameters

Parameter dialog also for *inherited parameters* of the shown model. This is in the context menu in diagrams if no component is selected:



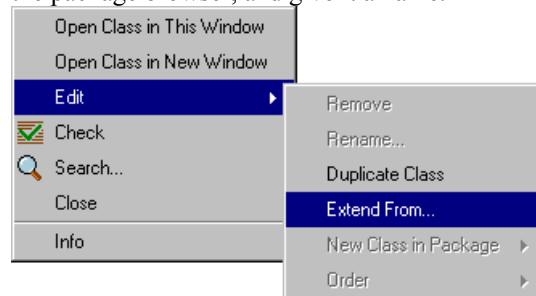
This also enabled at root-model, provided the model solely extend from other models and contain no declarations of its own. For instance if extending from Modelica.Mechanics.Rotational.Examples.CoupledClutches:



*Semantics:* Parameter-changes of the root-model are stored as modifiers on the corresponding extends-clause(s). Parameter-changes for non-root models work in the same way if you use 'Show component' and then 'Parameters' (with no selected component) to bring up the parameters for the component or directly use 'Parameters' for the component.

To experiment with a model such as CoupledClutches you can follow these steps. The experimental changes are then stored in a model. *Note: You can also edit parameters after translation in the plot-selector. Such changes are not stored in the model, but you can store them in a script using 'File/Save Script/Variables'.*

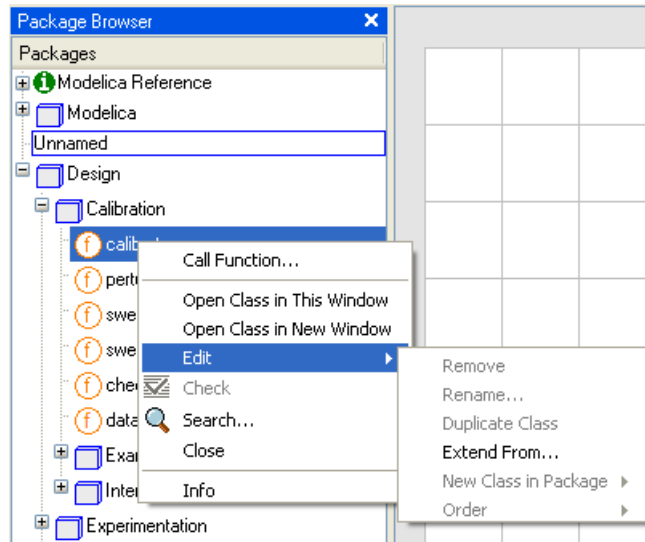
1. Extend from the model CoupledClutches using the context menu on CoupledClutches in the package browser, and give it a name.



2. In the diagram select a component, several components, or no component and use the right mouse button to bring up the parameter-dialog, and modify parameters.
3. Simulate. The model is translated if necessary.
4. Repeat from step 2.

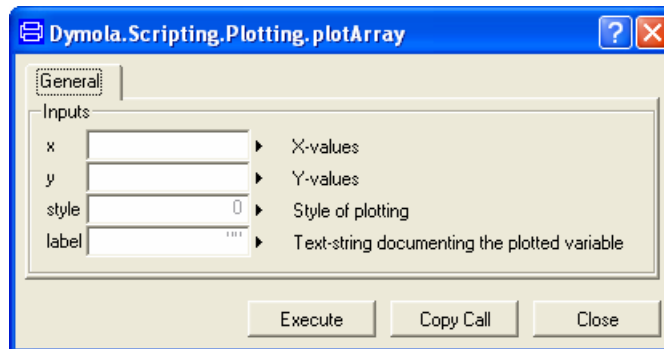
## Package browser

The context menu of the package browser has been extended with new functions:



The principal new operations are:

- Call Function displays a dialog to fill in parameters. The “Execute” button shows the log window also in Modeling mode. The “Copy Call” button copies the function call including parameter list to clipboard



- Check of package.
- Edit/Order to change order of classes.
- Close to close a package and all sub-packages. Clicking on the “+” will close the package, but when “+” is clicked again all sub-packages will be open as before. The “Close” will close all sub-packages so that they will not open when “+” is clicked.

A new annotation from Modelica 2.1 has been introduced:

```
annotation(DocumentationClass=true)
```

It is used to control the browser for documentation classes such as ModelicaReference. A special icon and the description string are shown in the browser, selecting such a class will always show the documentation layer and dragging of such classes is inhibited.

Dymola supports a new annotation to control what layer to show when a class is selected.

```
annotation(preferedView="diagram", "text", "icon", "info")
```

Other minor improvements are

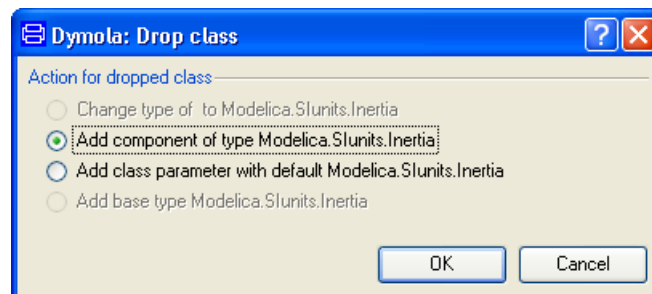
- Protected classes are by default not included in package browser.
- Constants are shown in the browser (the tool-tip gives the value)
- Changed handling of clicking in the package browser. There is a switch to control browser behavior: Advanced.SingleClickOpensTree 0=no open on single click, 1=toggle open, 2=only open; default=2.

## Component browser

It is possible to drag classes from the package browser and drop them into the component browser. This allows you to:

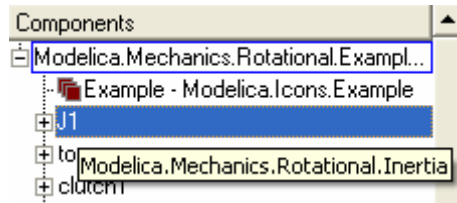
- Change type of declarations (by dropping on top of the component name).
- Declare new variables (see also “Variable declarations”).
- Declare new class parameters.
- Add new base-class.

When something is dropped in the component browser the following dialog is shown:



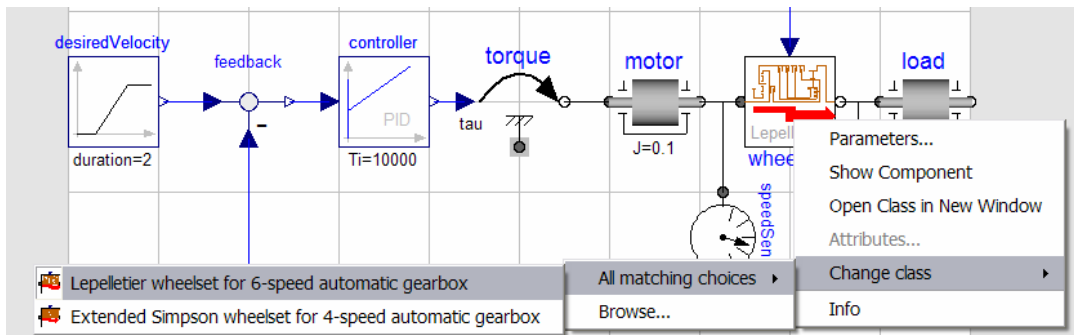
The type of a component is shown as tool tip if you rest the cursor over it.





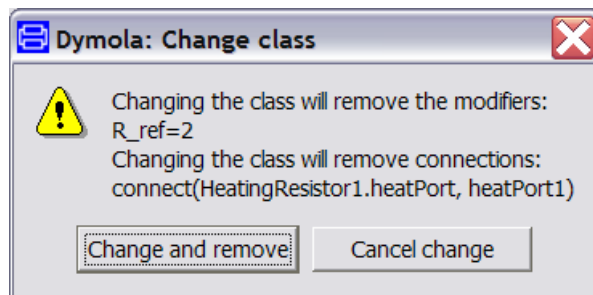
## Replaceable components

It is possible to change class of replaceable component using the context menu for the component. The menu shows a list of matching choices, where the icon of the current choice is shown depressed. Selecting the menu choice “Browse...” displays a normal class browser. If the component is declared in the current class you can only browse for replacements.



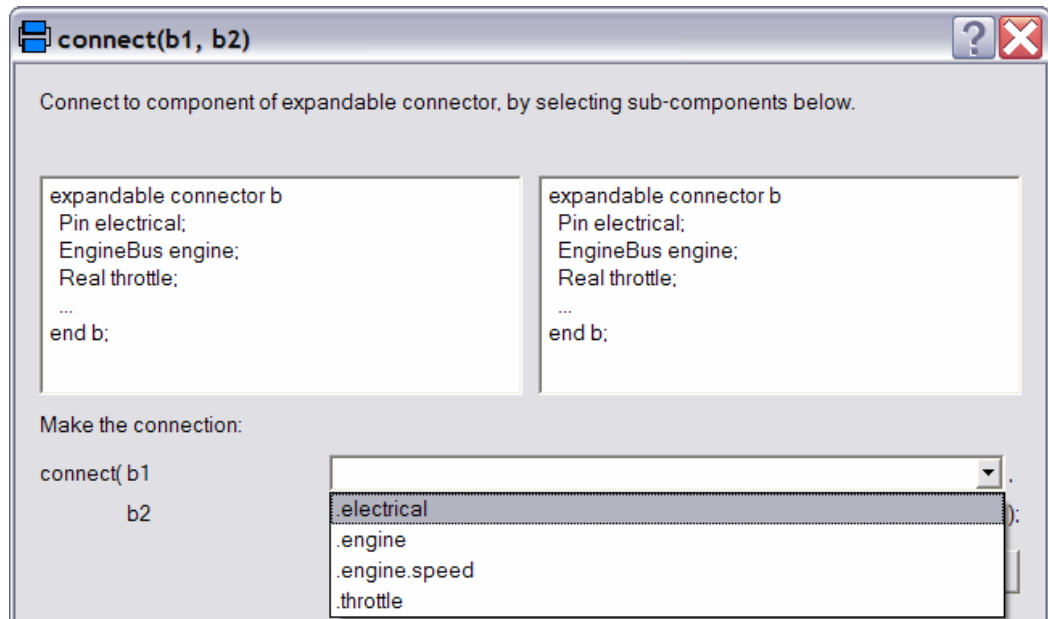
The parameter dialog shows the parameter dialog of the redeclared class (above for Lepelletier) instead of the original class. The text and info layers show the text of the redeclared class.

When changing class of a component (dragging the new class from package browser and dropping it on the component in the *component browser*) and modifiers or connections no longer match, you will be asked to confirm the change, and in that case they are automatically removed.



## Connections

Dymola has support for expandable connectors as defined in Modelica Language version 2.2. Connecting two expandable connectors gives a dialog as shown below, and components of expandable connectors are automatically treated as connectors.



Support for array of connectors has been implemented.


- Vectors of connectors can be defined and connected to in the graphical editor.
- The graphical representation is currently as one large connector, not as "n" individual connectors. For that reason, the extent (e.g. height) of the connector should be increased so it can accommodate a reasonable number of connections.
- When connecting to a connector array, Dymola will present a dialog that lets the user select index.
- Connection lines are evenly distributed on the connector, according to index number.
- The orientation of the connector determines if connections start at the top or the bottom. The connections start from the first extent point of the connector. If a different order is desired, use Edit/Flip Vertical or Edit/Flip Horizontal on the connector.
- The array dimension can either be a number or a simple parameter:

```
ConnType a[3];  
parameter Real n = 2;  
ConnType b[n];
```

In the graphical editor, connection lines are now drawn *over* components and other graphical objects. This means that connections are not hidden by mistake.

## Graphical editing

### Recent models

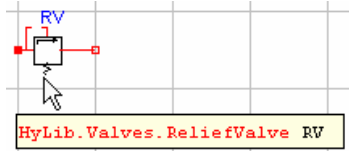
Dymola provides easy access to recently viewed models. Click on the  button to select the previous model, and on the arrow next to this to select one of the most recently viewed models.



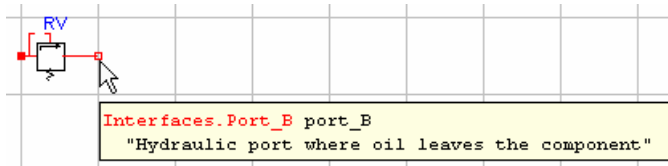
### Dynamic tooltips

Resting the cursor over a model component or connector displays a tooltip with type and name. Over a connection line the tooltip contains the names of the connectors.

#### Component tooltip.

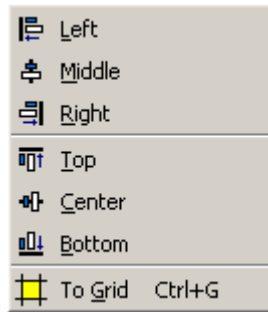


#### Connector tooltip.



### Edit Align

The alignment operations are used to organize objects in the graphical editor. It is an alternative to moving the objects with the mouse or by using arrow keys.



The alignment operations Middle, Center and To Grid use a center point for alignment. For graphical objects, the center point is the center of the bounding box of the object. For components and connectors, the origin of the coordinate system of the component's class is used as center.

### **Align objects**

The majority of commands in Edit/Align align graphical objects with each other. The first selected object is used as reference (and does not move).

First select the reference object. Then select the objects that should be aligned with the reference while holding down the Shift key. This creates a multiple selection with a known reference object. Finally, select the appropriate alignment from the menu. Horizontal alignment is specified with Left, Middle and Right, vertical alignment by Top, Center and Bottom.

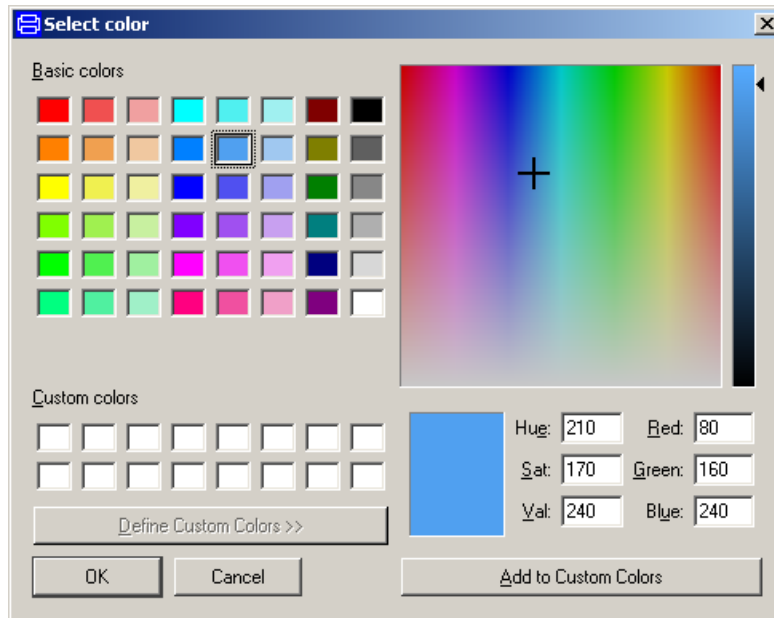
### **Align to gridlines**

Edit/Align/To Grid aligns selected objects to the gridlines. The center is aligned to the nearest gridline or halfway between gridlines; this allows alignment of components of default size on gridlines or between gridlines.

The keyboard shortcut is Ctrl-G.

### **Color selection**

There is an improved color selection dialog with a better choice of standard colors. Dymola supports true RGB color for all graphical objects, offering improved capabilities to design icons. For backward compatibility, color information is stored both in true RGB form and mapped to a color index as in previous versions of Dymola. Note that RGB color is lost if models are edited in an older version of Dymola.



## Bitmaps



There is a button in the drawing tool bar to insert bitmap images. Double click on the bitmap image presents a dialog to specify filename of bitmap. Incorrect filename displays a default bitmap (same as toolbar button). Bitmap is by default scaled preserving aspect ratio, centered in bounding box.

The bitmap annotation also has the attributes:

- `stretch=false` => The bitmap is not scaled to fit the bounding box. The original (pixel) size is preserved.
- `preserveAspectRatio=false` => The bitmap is scaled to fill the bounding box, possibly distorting the image.

The default values are 'true' in both cases.

## Default component size

More flexibility in defining the size of components which are dragged into a model. It is now possible to set a component scale factor in a class. This factor defines the default size of a component of the class. Assuming a model M has defined the scale factor 0.1, dragging class M into another model A will create a component which size is "scale" times the size of the coordinate system extent of M. The default settings are compatible with the default behaviour of previous versions of Dymola.

Previously it was possible to set the default "component size" in Dymola. In this case the receiving class A defined the size of any component that was inserted regardless of

component type. To preserve backward compatibility, the old "component size" attribute is applied if "scale" has not been set.

To manually set "component size" the user must now edit the Modelica text of the model to create the appropriate annotation. For this example, open model A in the Modelica text layer. Then right-click and select Expand/Show entire text. Then add the annotation:

```
model A
  annotation(Coordsys(component=[20, 20]));
  // ....
end A;
```

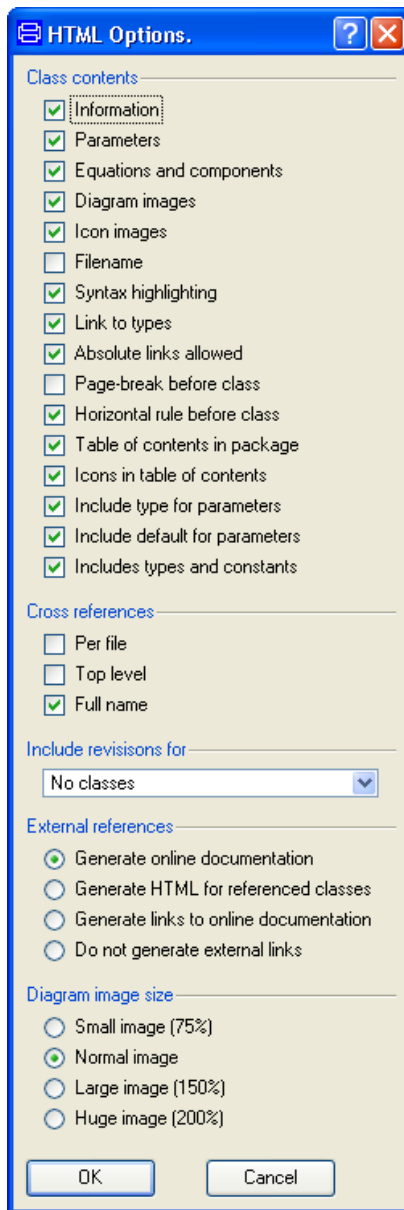
## HTML documentation

The content of the HTML generated by Dymola has been improved in several ways.

- Dymola can generate a table of contents for packages, connectors, and connectors in models, with small icons.
- Improved parameter-tables for generated HTML: tables for function inputs and outputs and type-column in parameter-tables.
- If the diagram layer is in "Show redeclared" mode, the generated diagrams and icons are also shown redeclared.
- Hyperlinks using "Modelica://Package.Class" are supported. Such links makes it easy to reference Modelica classes in the documentation (used for example in the new MultiBody library).
- Protected classes are not included if not shown in package browser.
- Empty pictures removed.

The control over HTML generation (File/Export/Setup HTML) has been extended with more options.

- Automatically generate table of classes with description for every package.
- Can set if revisions shall be included, not included, or included at top-level.
- Control if types and default values are included.
- Switch to insert lines between classes.
- Page break can be inserted before each class.



The documentation layer has been also been improved.

- Documentation window in Dymola starts with description and Information followed by info-text (as does the generated HTML-file).
- Package content, parameters, inputs, and connectors are also included in info-layer.

A Word 2002 template for generation of printed library documentation is available in `dymola/documentation/Documentation template.doc`. Index entries for class declarations is generated according to Word HTML format.

## Settings and options

The Edit/Options dialog has been extended with two new tabs.

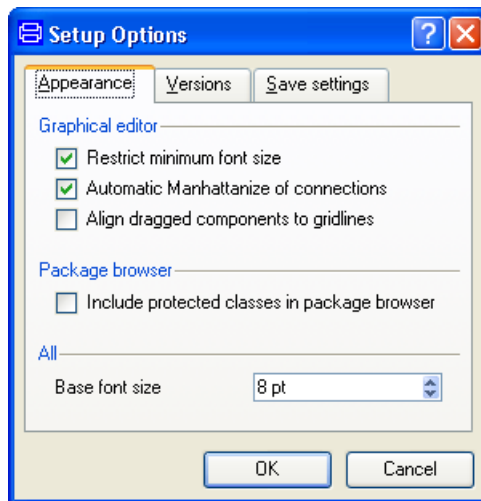
### Appearance

Protected classes are by default not included in package browser, but this can be changed by clicking the checkbox.

Dymola by default uses the system font size for text in menus, Modelica text etc. If this size is too small, for example when presenting Dymola for an audience, the base font size can be changed.

On some computers the it can be difficult to see the difference between the digit “1” (one) and the letter “l” (lower-case L). Increasing the font size often helps, but if needed the font used for Modelica text can be changed by a command line option; which font is suitable depends on the computer and which fons have been installed. Example:

```
dymola.exe -fixedfont "Lucida Console"
```

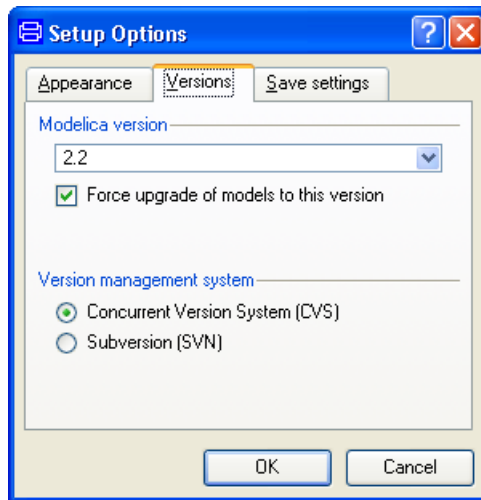


### Versions

Sets the default version of the Modelica Standard Library and whether models should be [upgraded to this version](#). This allows the user to quickly switch between Modelica 1.6 and Modelica 2.2. By setting the "Force" flag, models are required to be updated to the currently selected version of Modelica; if not set, the version of Modelica that the model uses is loaded on demand.



If the “Model Management” option is available, Dymola supports version control of models using two commonly used systems, Concurrent Version System (CVS) and Subversion (SVN).



### Save settings

The command `Save` will save the current window layout or Modelica version in a file associated with the user. The next time Dymola is started, the saved settings are used. The layout information includes:

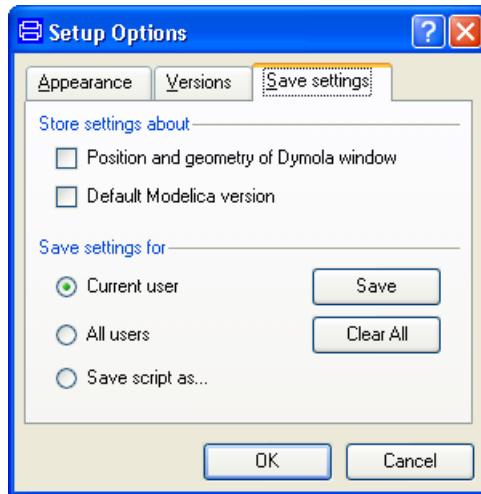
- Dymola window position and size.
- Position and location of command log, browsers and toolbars inside the Dymola window.

The setup information is by default stored in a system directory associated with the user. On Windows, according to the recommendations of Microsoft, the name is typically

```
C:\Documents and Settings\\Application  
Data\Dynasim\settings.mos
```

The directory `Dynasim` is automatically created by Dymola if needed. It is also possible to store the settings in a directory read by all users, or to save the settings as a local script file chosen by the user.

The command `Clear All` will erase the settings file for the current user or all users. Defaults will apply until settings are saved.



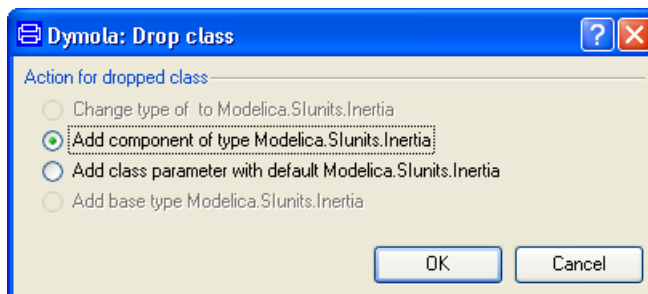
---

## Modelica text editor

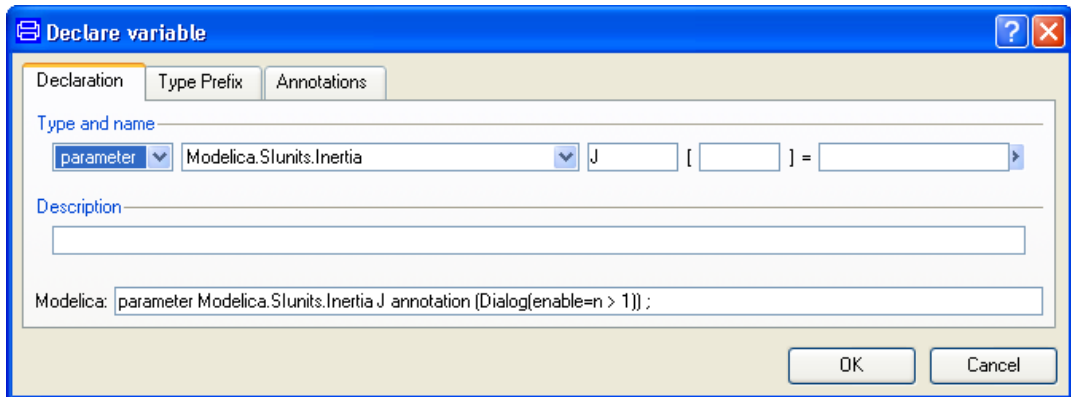
### Variable declarations

There is less need for writing Modelica code because most variable declarations can be generated by a drag-and-drop operation from the package browser.

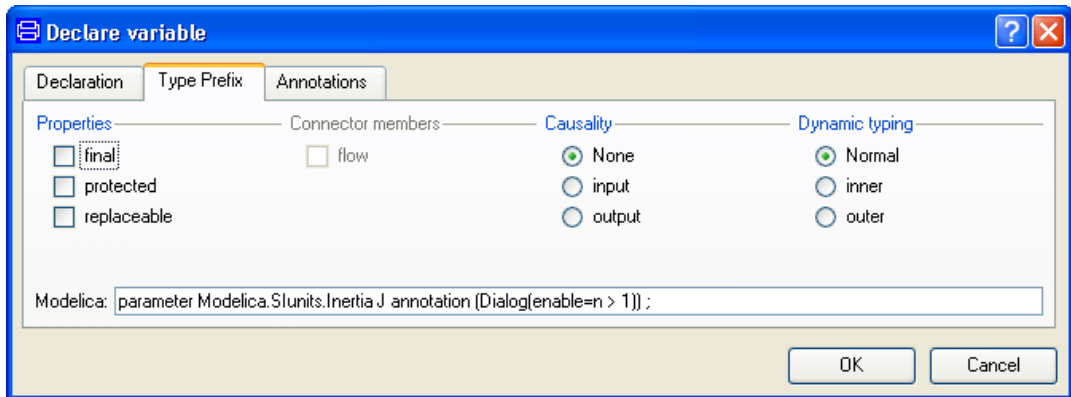
To declare a new variable drag a type from the package browser to the component browser or diagram. Several operations are possible, but the default is to add a new variable of the specified type:



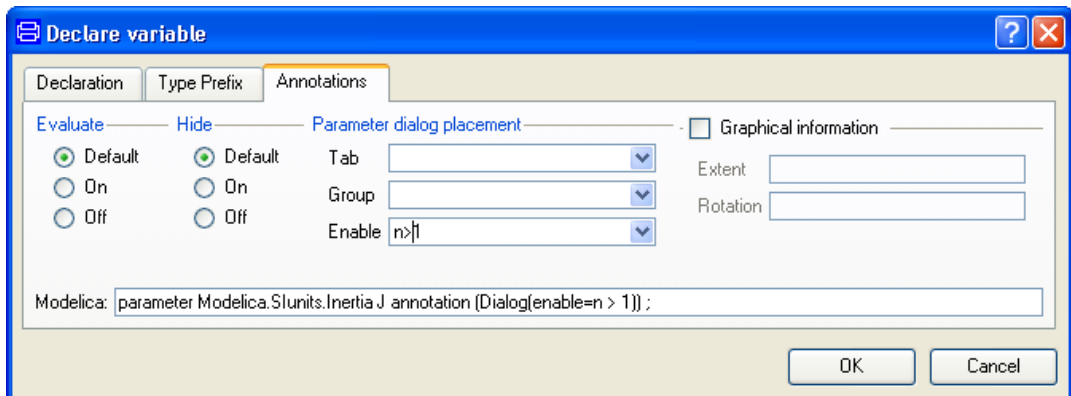
Dymola will then show a declare variable dialog where you can specify name, defaults, variability (parameter, constant, ...).



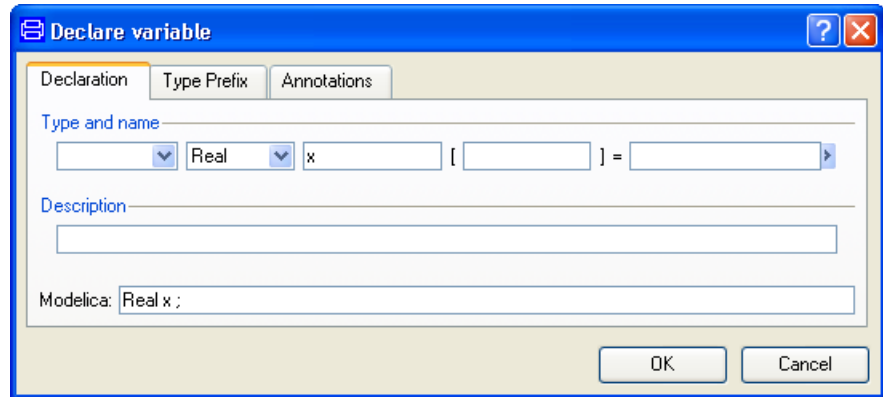
Type prefix, such as, final or protected, causality and dynamic typing is specified in the second tab:



In the annotations you can specify the tab and group of parameters, and also make the input field conditionally enabled depending on other parameters. These are stored as annotations.



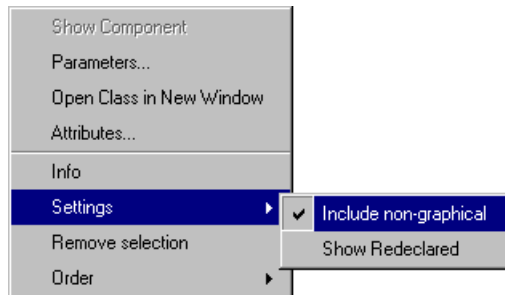
New variables of the built-in types Real, Integer, Boolean and String can be declared with “Edit/Variables/New Variable”.



It is easy access to all variable declarations in a model using “Edit/Variables”. This is allows easy modification of attributes and common annotations.



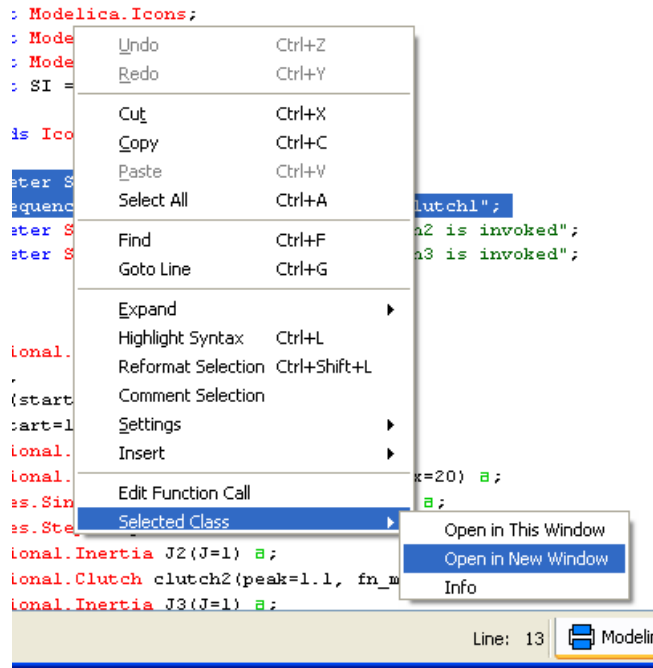
You can also use “Setting/Include non-graphical” in the component browser and select “Parameters” for a variable; this will display the variable dialog.



The operation “Propagate” in the parameter context menu in the parameter dialog also gives a variable declaration dialog. See “Parameter dialog” on page 9.

## Editor context menu

The context menu of the Modelica text layer has been extended with several new operations.



Syntax checking and highlighting (without reformatting) can be done by pressing on the Modelica Text button, in addition to command “Highlight Syntax” in the context menu. The keyboard shortcut is Ctrl-L. Syntax is automatically highlighted when you use the Check command.

It is possible to auto-format (pretty print) a selection using the command “Reformat Selection” in the context menu or by pressing Ctrl-Shift-L.

A command to comment out selected rows is also available in the context menu.

Support for directly setting level of expansion from the context menu of the Modelica Text layer:



Local packages are by default not shown in the Modelica Text view. They can be shown by Expand/Expand Local Packages in the context menu.

It is possible to get a parameter-window for a function call in the Modelica text. Select a function call up to ending parenthesis (or put the cursor inside name), and select “Edit Function Call” in context menu.

It is possible to go to model/function from Modelica text. Select a class name (or put the cursor inside the name). The context menu contains operations on the “Selected Class”.

## Other operations in text editor

The number of the current line is shown in the statusbar. It is also shown as default if you select “Goto Line” from the context menu, shortcut Ctrl-G.

The “Find” operation will automatically scroll the window to make the found text visible. Find/Replace in Modelica text gives warning if search string is not found.

A selection of Modelica text layer can be printed. If you have selected a part of Modelica Text, and then do File/Print, there is an option to print only the selected text instead of the whole model.

The formatting of declarations, equations, etc is kept by default. The formatting is not kept for:

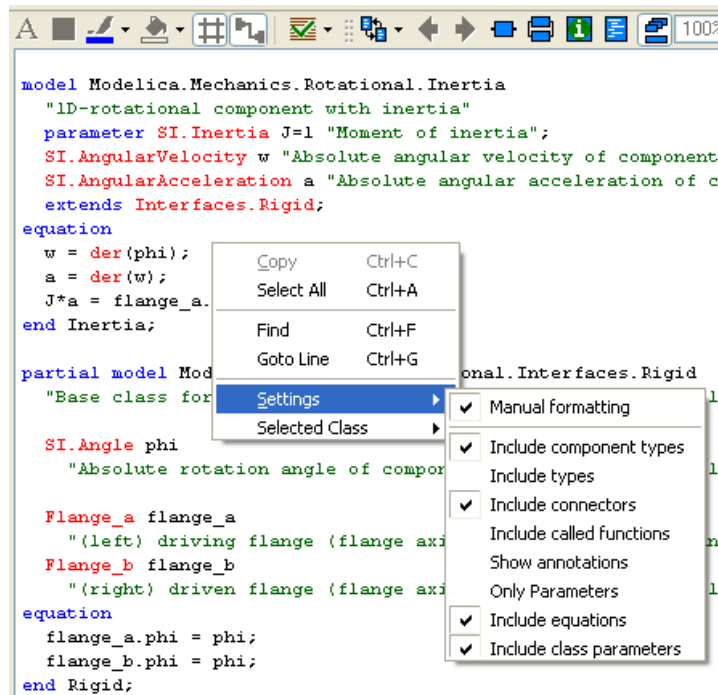
- Multiple statements or declarations on one line.
- Indentation of end-tags (such as ], ), end <class> ).
- Indentation of separators (such as ,, ;, then, else).
- Multiple empty lines.
- Comments inside declaration of components, or type-prefixes of classes (e.g. inner /\*  
\*/ class a end a;)
- Comments inside expressions.
- Equal sign for modifiers.
- Minor errors that are automatically corrected (e.g. incorrect use of = vs. :=).

If a syntax error is detected, the position of the error is shown by the cursor.

It is possible to copy text with hidden annotations to clipboard.

## Used classes

The text of a Modelica class and classes which are used directly or indirectly is available in the “used classes” layer. What is shown can be controlled by the context menu. This layer is read-only, no editing is possible.



## Modelica language

Dymola's support for the Modelica language has been improved to provide better diagnostics and support recent features of the Modelica language.

### Arrays

The support for array of records makes it possible to check the entire Modelica 2.2 library.

Increased support for arrays in the interactive environment.

- The size of arrays of records can depend on the inputs.
- Arrays in functions declared using the size : can be resized by assigning to the entire array. (This applies to both arrays of records and arrays of simple types.):

```

function f
  input Integer n;
  output Real x[:];
algorithm
  for j in 1:n loop
    x:=cat(1, x, {j});
  end for;
end f;

```

Array of records (with literal size) is supported in compiled functions.

## Conditional declarations

Conditional declarations are supported according to the Modelica 2.2 semantics. If the condition is false the component is removed including modifiers and connections to it:

```
model C2
  extends
    Modelica.Mechanics.Rotational.Examples.CoupledClutches;
  parameter Boolean addFriction=true;
  Modelica.Mechanics.Rotational.BearingFriction
  BearingFriction1(tau_pos=[0,2]) if addFriction annotation
  (extent=[62,-62; 82,-42]);
equation
  connect(BearingFriction1.flange_a, J3.flange_a) annotation
  (points=[62,-52;
           50,-52; 50,0; 35,0], style(color=0, rgbcolor={0,0,0}));
end C2;
```

## Checking for structural singularities

Dymola has extended support for checking for structural singularities of the model equations.

When using a model for simulation it is a basic requirement that there are the same number of equations and number of unknown variables and that there is for each variable an equation from which the variable can be solved. This check is now done in more detail as it is done separately for each of the four basic data types Real, Integer, Boolean and String. This supports better checking. For example, let  $r$  be a real variable and let  $i$  be an integer variable and consider the equation  $r = i$ . This equation is a real equation, since we need to use it to solve for  $r$ . We cannot use it to solve for  $i$ . It may be remarked that the checking for structural singularities of the initialization problem has had this more detailed checking from the start. The simulation problem is more complex since it may have high DAE index which means that it is by definition singular if only the derivatives,  $\text{der}(x)$ , and algebraic variables,  $v$ , are considered to be the unknown variables. It is necessary to also consider the appearances of  $x$ .

Using the check command enforces checking for structural singularities. No user is happy when the error message at translation says missing equations or too many equations. It may be that the components are used in a wrong way, for example a component is missing, but it may also be that a used component model is wrong. Dymola has extended the checking of non-partial models and block components to include checking for structural singularities in order to give component or library developers better support. To get a non-trivial result, Dymola puts the component in an environment that should reflect a general use of the model.

- All inputs of the model are considered to be known
- Flow variables that are not connected will at translation be set to zero. However, checking a flow source (components that defines the flow variable) in such a way would



make it singular. Such models are not intended to be used in that way. At checking Dymola instead generates for each flow variable an fictive equation referring all variables of all the connectors of the model component. The aim is to create the most general variable dependence that may be generated by connections.

- The equations of the model component may depend on the cardinality of its connectors (whether a connector is connected or not). The Check operation of a component considers all the connectors of the component to have connections on their outer sides.
- Overdetermined connectors for example in the MultiBody Library are dealt with in the following way. If an overdetermined connector of the model component is part of a connected set that has a root or potential root candidates everything should be fine. Otherwise, Dymola specifies the connector as a potential root and if it is selected as a root, Dymola adds fictive equations referring to all of the variables of the overdetermined connectors to compensate for the missing redundancy.

Checking of model components are done recursively. As indicated above a structural singularity may be caused by improper use of components or come from singular components. When trying to pinpoint a source of singularity we cannot assume that the components are correct because we have checked all model classes. First, the checking of a model component assumes a general use, however, when actually using a component, the environment are more specific and singularities may then show up. Secondly, modifiers with redeclare may imply drastic changes of a component and there may not be an explicit class to make relevant checks on. Thirdly, this is even more accentuated when there are class parameters to set. Fourth, dimension parameters may take other values then assumed when checking the component model. When Dymola finds a component to be singular, it makes a recursive check of the components. Dymola then tries to set up an environment that mimics the real environment in the best way. For example a connector not being connected, the generic equations for the flow variables of that connector are not generated, but zero default values are used. For example, a flow source will then be diagnosed as singular. Also the cardinality is preserved. The error diagnosis output exploits the results. If a component is found to be singular this is reported. If no component is found to be singular the error message focuses on the use of the components.

The extended structural checking is enabled by default, but can be disabled by setting the flag `Advanced.ExtendedStructuralCheck = false`

If a model is found to be singular at translation, the components are checked recursively. This can be disabled by setting the flag `Advanced.ExtendedStructuralDiagnosis = false`

Connectors that are neither physical (matched flow and non-flow) nor causal will assume a suitable number of external conditions on them. If this corrects the problem no recursive check is performed.

Models that by design are non-partial and structurally singular can use the annotation `(structurallyIncomplete)`; This has been added to e.g. `Modelica.Blocks.Math.TwoInputs`, and to the base-classes of `MediaModels`.

Component models of partial classes inhibit the structural check.

## Improvements in diagnostics

Error messages for type errors are grouped together, and start by giving the variable or equation that caused the error. The error messages contain links to the text of classes.

Assertions are evaluated early to improve diagnostics, and thus allow an assertion to guard against a structural error.

The translation log now also includes statistics for the initialization problem.

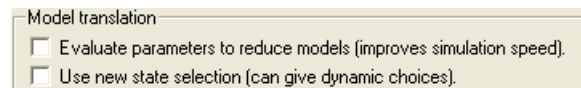
Improved diagnostics for nonlinear system of equations:

- Warnings are output for variables to be solved by a nonlinear solver, but not having an explicit start value.
- The start values for all iteration variables can be logged by setting the flag `Advanced.LogStartValuesForIterationVariables = true`.

## Evaluation of parameters

Top-level parameter records are handled in the same way as simple top-level parameters (they are not evaluated when you specify that parameters should be evaluated).

The simulation menu also allows a global setting for evaluating parameters and dynamics state-selection:



It is also possible to specify Evaluate selectively for a parameter.

- `annotation(Evaluate=true)` forces evaluation if value provided
- `annotation(Evaluate=false)` overrides the flag Evaluate

Top level packages may contain an annotation of the form:

`annotation(Settings(Evaluate=true));` If a model uses anything from that package Dymola automatically uses Evaluate=true.

## Dynamics state selection

Dymola has a new state selection algorithm that may cause dynamic state selection.

Top level packages may contain an annotation of the form:

`annotation(Settings(NewStateSelection=true));` If a model uses anything from that package Dymola automatically uses NewStateSelection=true. Since Modelica 2.2 has this annotation nearly all models will automatically use the new state selection, and may thus cause dynamic state selection.

## Storing of protected variables

It is possible to prevent storing protected and hidden variables during simulation. Not storing protected variables makes it easier to find the relevant information, but in some cases detailed post-processing require access to protected variables.

- Check box in the dialog Experiment Setup/Store/Protected (default is to not store them).
- flag `Advanced.StoreProtectedVariables=false`
- specify that a variable should be hidden:  
`annotation(Hide=true)`  
`annotation(Hide=false)` overrides the flag `Advanced.StoreProtectedVariables`
- This can cause some animation objects to disappear, in particular for the PowerTrain library version 1.0 (the problem is corrected in 1.0a and later versions). You can either enable storing of all protected variables (in Experiment Setup/Store), or contact Dynasim for a new version of the library.

Changing "Protected variables" in Simulation/Setup/Output/Store does not force a recompilation of the model.

## Other

- `semiLinear()` is now a built-in function and redesigned to not generate events.
- Improved symbolic processing for initialization. It includes exploitation of constant expressions. This is in particular useful for steady state initialization, where  $\text{der}(x)=0$  is exploited to reduce the problem.
- Support for record constructor.
- Output arguments to functions may be omitted.
- In order to support inner/outer components some new annotations are supported (can also be useful for other cases):

```
annotation (  
  defaultComponentName="world",  
  defaultAttributes="inner",  
  missingInnerMessage="No \"world\" component is defined. A  
default world  
component with the default gravity field will be used  
(g=9.81 in negative y-axis). If this is not desired,  
drag MultiBody.World into the top level of your model.",
```

- Handling of functions with non-alphanumeric names (e.g. '`<`') has been improved.
- Expandable connectors are supported, see Connections on page 18.

---

## Simulation

The main improvements in the simulation mode are model-specific commands menu, a new simulation window, and improvements in the simulation itself to give increased robustness and diagnostics.

### Commands menu

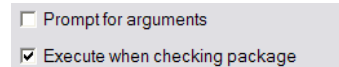
The Commands menu has the choice “Add Command”. This function is used to associate commands (Modelica functions or script files) with models. The Commands menu can in addition to `file="file.mos"` have `executeCall=foo()` and `editCall=bar()`. Both of these calls the given function instead of running the file, and `editCall` allows the caller to modify the function arguments before calling the function.

These commands are added to the Command menu when the model is active. An annotation of the following form is used:

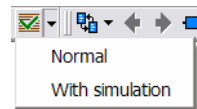
```
annotation(Commands(file="file.mos" "Script to ...",
file="file.ppt" "PowerPoint presentation ..."));
```

The directory where the model was found is used in front of the file-name. Mos-scripts are run everything else is opened. Note: Running a script does not perform `cd` to its directory, thus the script cannot run other local scripts, i.e. a full path to such script is needed.

You can also specify that commands are part of check.

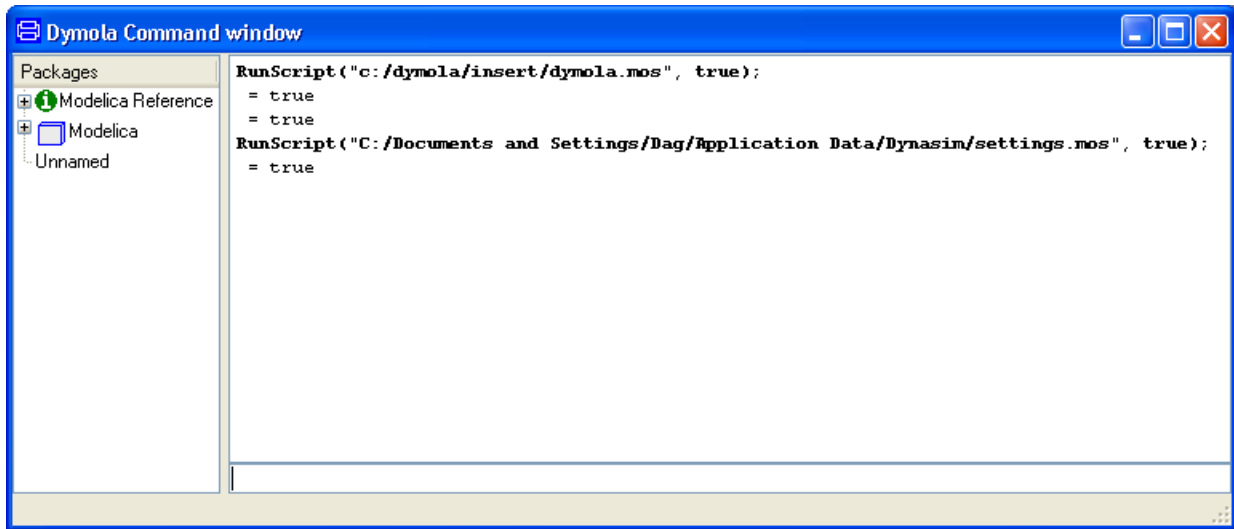


Check has been extended with a drop-down list to select if you also want to run such commands and simulate any model with stored experiment setup.

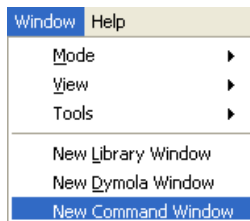


### Simulation windows

Dymola has new separate command window with a package browser.



Selecting Window/New Command Window displays the new command window.



The Window menu has a new command “Tools” which makes it easy to select which browsers and toolbars should be visible. For example, this makes it possible to display the diagram layer of a model in Simulation mode.

## Improvements in interactive functions

Possible to trace function execution using the built-in function

```
trace(variables = false, statements = false, calls = false,
onlyFunction = "");
```

This will optionally trace values of assigned variables, executed statements, calls of the function, and can optionally be overridden for a specific function. Calling trace with default value for onlyFunction overrides the trace-settings for all functions.

Edit Function Call added to context menu of command input line.

Additional function arguments:

```
simulateExtendedModel
```

autoLoad=true. If false the result file is not loaded in the plot-window (and variables are not replotted).

initialized

isInitialized=true. If false it will initialize according to the initial equations at the start of the simulation.

list and variables

Possible to write the variables to a script-file (which can be executed) filename="script.mos", and limit it to certain variables by using variables={"var1","var2"}.

Can specify a model-name with modifiers for translateModel, simulateModel, etc. e.g.

```
for source in {"Step", "Constant", "Ramp", "Sine"} loop
  simulateModel("TestSource(redeclare
Modelica.Blocks.Sources."+source+" Source)");
end for;
```

## Minor improvements

Many minor improvements, in particular

- Performance of when and sample in certain cases
- Global common sub expression elimination is performed for scalar functions.
- Solving of ill-conditioned linear systems
- Improvements necessary to generate efficient code for Modelica.Media.
- Automatic differentiation and partial derivatives of Modelica functions. For analytic Jacobians see section 'Analytic Jacobians' on page 62, please contact Dynasim for further information on partial derivatives.
- The possible number of external objects has been increased.

Improved robustness of non-linear solver:

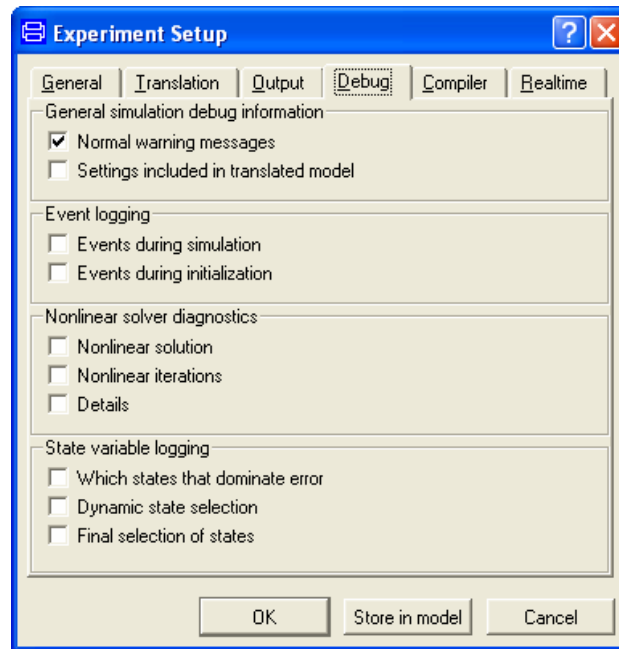
- Initial equations automatically use a homotopy-method to handle difficult cases.
- Automatic test of non-zero start-values in non-linear systems.
- Special handling of scalar systems.
- Improved diagnostics.

Improved behavior and diagnostics for non-linear solver.

Improved debug-facilities while dymosim is running:

- Can enable logging of non-linear systems of equations.
- Can interrupt during initialization.

New tab for selecting logging of Debug information in the Simulation/Setup menu. These correspond to the debug-facilities while dymosim is running, but are easier to access. For details, see the tool tips.



The "continue" command has been extended in the menu to also allow continue from an arbitrary point in a result file (from the same model).

Improved initialization and extended support for mixed equation systems.

Find in log-windows, see context menu or use Control-F.

Improved event handling

- Full support of smooth
- Fewer events generated

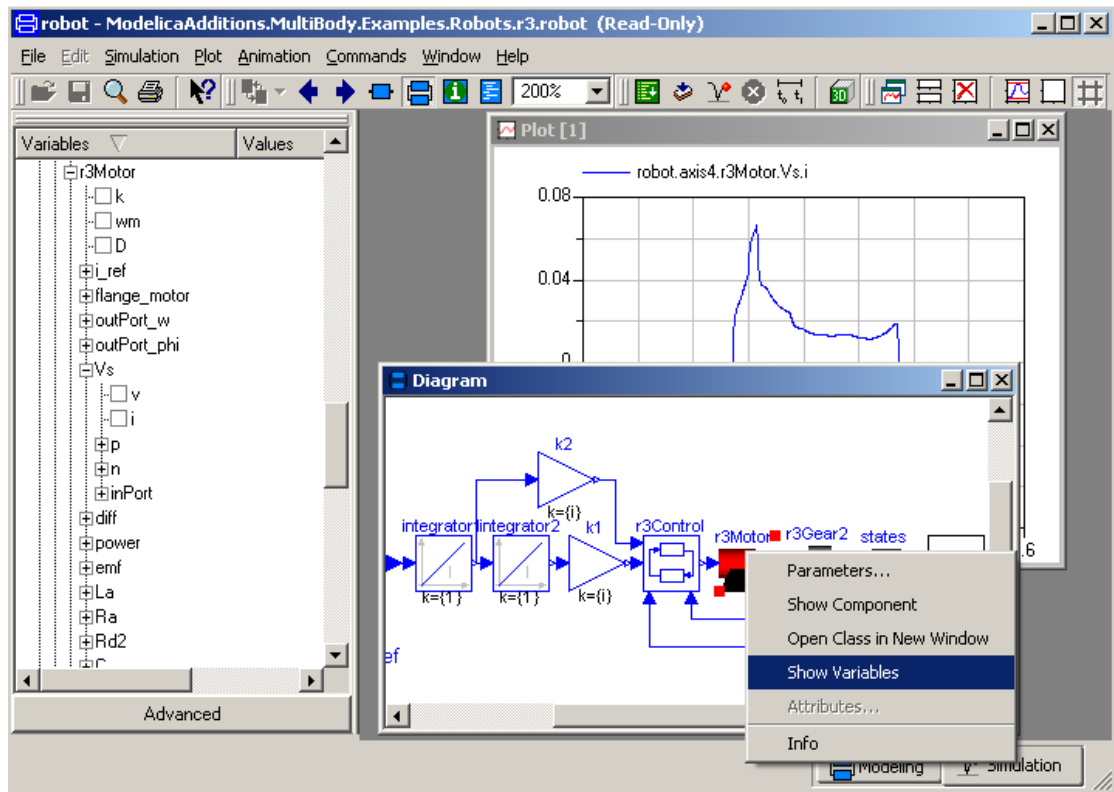
Improved handling of tables:

- Table editor has load/save buttons for Matlab files, CSV-files (for Excel), and Text-files.
- Matlab-routines for easier construction of 2-dimensional, load2DTable.m, save2DTable.m, and n-dimensional, loadNDTable.m save2DTable.m files.
- The n-dimensional routine works in combination n-dimensional table lookup model, TableND. The file is found in [Dymola/Modelica/Library/TableND.mo](#), and will later be integrated into the Tables-library.

## Diagram layer in simulation mode

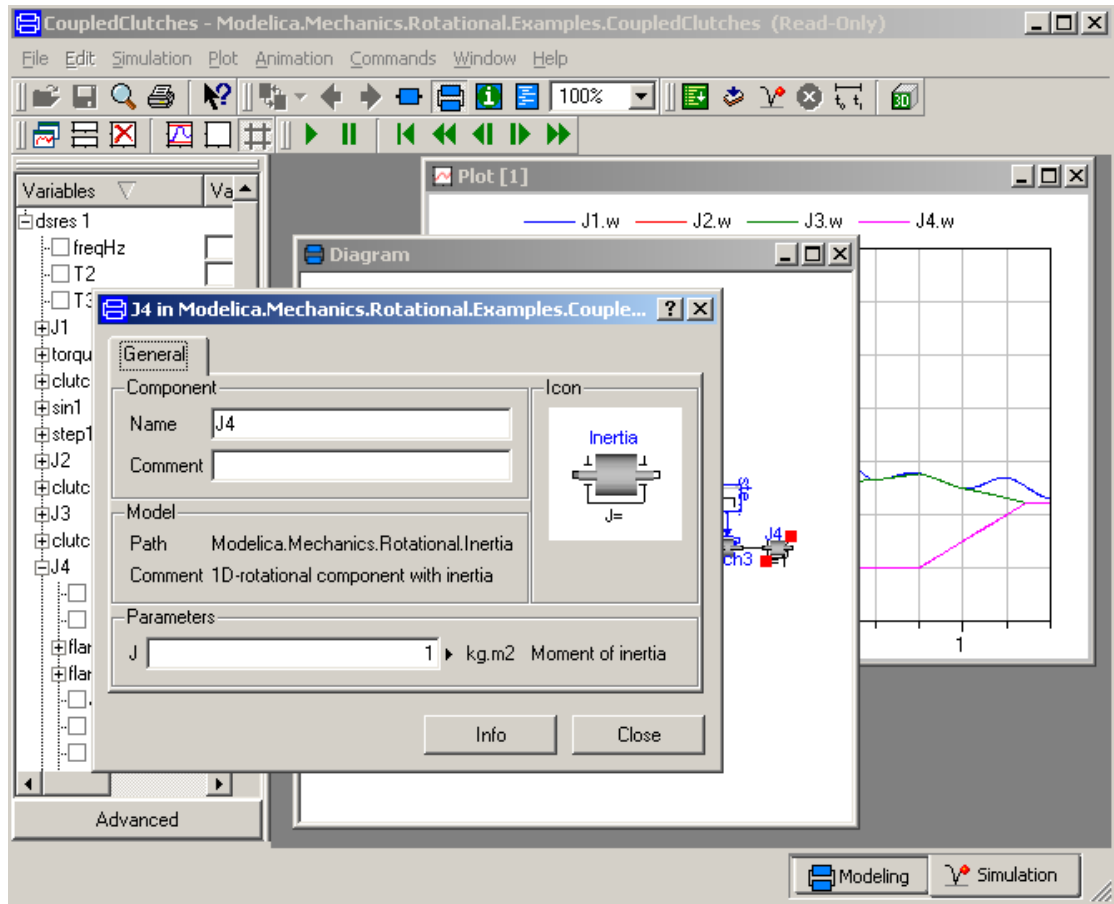
The diagram layer representing the model is available in Simulation mode to enable the user to follow a simulation by displaying variables and to control it by setting parameters. The user can descend into any level of the model in order to plot or display variables. Push the diagram layer button to show the diagram.

The context menu for a component contains the choice Show Variables, which opens the plot selector for the selected component instance.



When double-clicking on a component or selecting Parameters... in the context menu the parameter dialog is displayed.





Changing parameters in the parameters dialog does not always require a new translation (it is still necessary if the modifier is too complex, the parameter had been evaluated during translation, or if parameters of a different model were changed). The model is changed and you will be asked to save it when exiting Dymola. To use this for top-level parameter see section 'The graphical editor has been improved to better support editing and browsing of large complex models.

Parameter dialog' on page 9.

## Improved experiment setup

The experiment setup can be stored in the current model using the button "Store in model". This applies to the General and Output tabs. When the model is later selected, these settings appear in the experiment setup dialog and can be changed before simulation.

A set of new inline integration methods have been introduced for real-time simulation. The following selector is available under the real-time tab and as the built-in variable, Integer

Advanced.InlineMethod:  
0: Inline integration method not used  
1: Explicit Euler  
2: Implicit Euler  
3: Trapezoidal method  
4: Mixed explicit/implicit Euler  
5: Implicit Runge Kutta

The order of the Implicit Runge Kutta method can be set between 2 and 4:

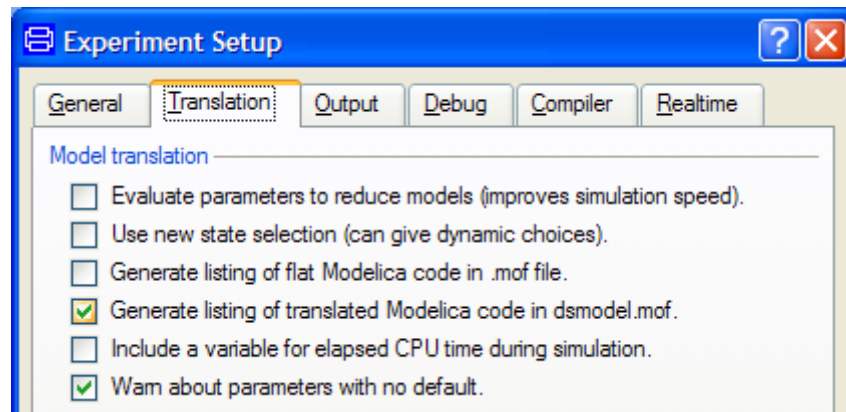
Integer Advanced.InlineOrder=2 "Order of Implicit Runge Kutta method (2-4)"

More information about real-time simulation can be found in the report ["Dymola Application Note HILS"](#).

## Output of manipulated equations in Modelica format

### Description

The result of translating a Modelica model can be listed in a Modelica like representation. The listing is stored in the file dsmodel.mof and is meant to be a more readable version of dsmodel.c. The listing is enabled by ticking "Generate listing of translated Modelica Code in dsmodel.mof" in the Model translation tab of Experiment Setup.



The listing may be useful for users who want to investigate algebraic loops or for other debugging purposes. It gives the correct computational structure including algebraic loops. However, to make it more readable some optimization steps such as elimination of common subexpressions are not done.

It means that Jacobians of algebraic loops are by default not listed, because without common subexpression elimination those expressions may be very long. Listing of the non-zero Jacobian elements may be enabled by issuing the command

```
Advanced.OutputModelicaCodeWithJacobians = true.
```

Information on this is included in the listing if there are algebraic loops.

Listing of eliminated alias variables may also be long. Thus, the listing of these variables is not enabled by default. The listing of alias variables is enabled by setting the flag

```
Advanced.OutputModelicaCodeWithAliasVariables = true
```

Information on this is also included at the end of dsmodel.mof, if listing of alias variables is not enabled.

Below some examples are given for illustration.

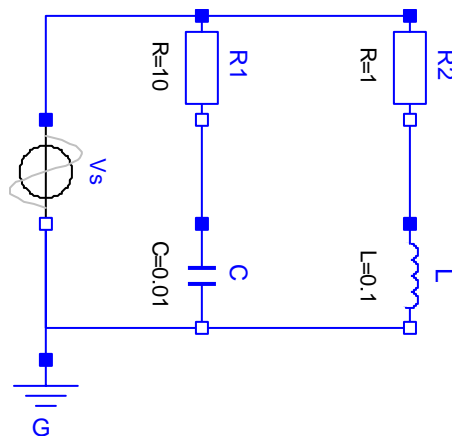
## Examples

Below, some examples will be given to illustrate and to discuss the information given by dsmodel.mof.

- **Simple LC circuit** is a first example to discuss the organization of the manipulated equations and listing of alias variables
- **Two resistors connected in series** illustrates symbolic solution of a linear algebraic loop
- **A simple resistor network** illustrates a manipulated linear system for numeric solution.
- **Diode circuit with two valued resistance diode model** introduces mixed discrete/real algebraic loops
- **Diode circuit with diode exponential diode model** introduces nonlinear algebraic loops

### Simple LC circuit

Consider a simple electric LC circuit with two resistors connected in series.



Translating the model produces a dsmodel.mof file with following contents.

```

// Translated Modelica model generated by Dymola from //
Modelica model
// OutputModelicaCodeExamples.SimpleLC_Circuit
// -----
// Initial Section
Vs.signalSource.pi := 3.14159265358979;
C.n.v := 0;
G.p.v := 0;
L.n.v := 0;
Vs.n.v := 0;
// -----
// Bound Parameter Section
Vs.signalSource.amplitude := Vs.V;
Vs.signalSource.freqHz := Vs.freqHz;
Vs.signalSource.phase := Vs.phase;
Vs.signalSource.offset := Vs.offset;
Vs.signalSource.startTime := Vs.startTime;
// -----
// Dynamics Section
R2.p.v := Vs.signalSource.offset+
  (if time < Vs.signalSource.startTime then 0
   else Vs.signalSource.amplitude*
     sin(6.28318530717959*
       Vs.signalSource.freqHz*
       (time-Vs.signalSource.startTime)
       +Vs.signalSource.phase));
R1.v := R2.p.v-C.v;

// Linear system of equations
// Symbolic solution
/* Original equation
R1.R*C.n.i = -R1.v;
*/
C.n.i := -R1.v/R1.R;
// Torn part
// End of linear system of equations

// Linear system of equations
// Symbolic solution
/* Original equation
C.C*der(C.v) = -C.n.i;
*/
der(C.v) := -C.n.i/C.C;
// Torn part
// End of linear system of equations
R2.v := R2.R*L.i;
L.v := R2.p.v-R2.v;

// Linear system of equations
// Symbolic solution
/* Original equation
L.L*der(L.i) = L.v;
*/
der(L.i) := L.v/L.L;

```

```

        // Torn part
        // End of linear system of equations
        // -----
        // Conditionally Accepted Section
        Vs.p.i := C.n.i-L.i;
        G.p.i := Vs.p.i-C.n.i+L.i;
        // -----
        // Eliminated alias variables
        // To have eliminated alias variables listed, set
        //   Advanced.OutputModelicaCodeWithAliasVariables
        //   = true
        // before translation. May give much output.

```

The manipulated equations are sorted into sections. First, there are calculations of constants and bound parameters. These parts are only executed at initialization. Then the calculation of outputs and the derivatives of the continuous time states follow. The output and dynamics sections are executed during continuous integration. In the general case there is then a section titled Accepted Section. It includes codes for detecting discrete event and updating discrete states that need not be evaluated at continuous integration. This section is executed at the end of each step to check for events. The output, dynamics and accepted sections are executed at event propagation also. Finally, there is the Conditionally Accepted Section. It includes calculation of variables which are not necessary to know when calculating derivatives or updating discrete states. This section is executed when storing values. In some situations for example when simulating on a HIL platforms where only states and outputs may be visible, the conditionally accepted section is not executed at all.

As shown by the listing, Dymola converts this problem symbolically to explicit ODE form (no algebraic loops to solve numerically). We can observe that Ohm's law of the resistor R1 is used to solve for the current through the resistor:

```
C.n.i := -R1.v/R1.R;
```

On the other hand, Ohm's law of the resistor R2 is used to solve for the voltage drop across the resistor :

```
R2.v := R2.R*L.i;
```

The sorting procedure of Dymola automatically finds which variable to solve for from each equation.

Connections between non-flow connectors result in simple equations,  $v_1=v_2$ . A connection between two flow connectors gives  $v_1+v_2=0$ . Dymola exploits such simple equations to eliminate variables. The listing of these variables is not enabled by default as indicated above because it may give much output. If we enable the listing of alias variables by setting the flag

```
Advanced.OutputModelicaCodeWithAliasVariables = true
```

the last part of the listing becomes

```

// Eliminated alias variables
R2.p.i = L.i;

```

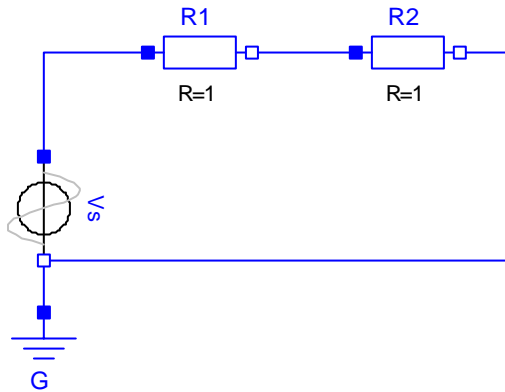
```

R1.i = -C.n.i;
R2.n.v = L.v;
R1.p.i = -C.n.i;
R2.n.i = -L.i;
L.p.v = L.v;
Vs.n.i = -Vs.p.i;
R1.n.i = C.n.i;
C.i = -C.n.i;
R2.i = L.i;
R1.n.v = C.v;
R1.p.v = R2.p.v;
Vs.v = R2.p.v;
C.p.v = C.v;
Vs.i = Vs.p.i;
C.p.i = -C.n.i;
L.p.i = L.i;
Vs.p.v = R2.p.v;
Vs.signalSource.y = R2.p.v;
L.n.i = -L.i;

```

## Two resistors connected in series

Consider a simple electric circuit with two resistors connected in series.



After elimination of alias variables, the problem has a linear algebraic loop with three unknowns. Dymola solves this symbolically as seen from the following excerpt from the `dsmodel.mof`.

```

// Linear system of equations
// Symbolic solution
/* Original equation
R1.R*R1.i = R1.v;
*/
R1.i := Vs.v/(R1.R+R2.R);
// Torn part
R2.v := R2.R*R1.i;
R1.v := Vs.v-R2.v;

```

```
// End of linear system of equations
```

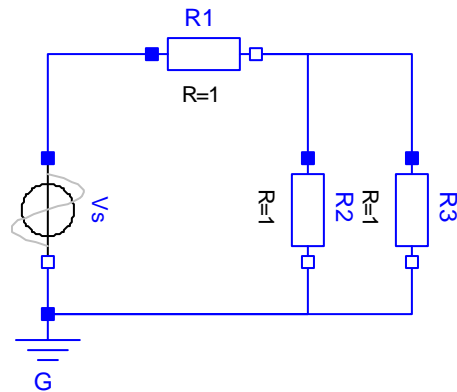
The equation

```
R1.i := Vs.v/(R1.R+R2.R);
```

reveals that the resistance of two resistors connected in series is the sum of the resistances of the two resistors. Dymola “discovered” this law automatically.

### A simple resistor network

Consider a simple resistor network.



After elimination of alias variables the problem has a linear algebraic loop with five unknowns. Dymola reduces it to a linear problem with two unknowns as seen from the following excerpt from the `dsmodel.mof`.

```
// Linear system of equations
// Matrix solution:
/* Original equations:
R1.R*R1.n.i = -R1.v;
R2.R*R2.p.i = R3.v;
*/
// Calculation of the J matrix and the b vector,
// but these calculations are not listed here.
// To have them listed, set
//   Advanced.OutputModelicaCodeWithJacobians =
//   true
// before translation. May give much output,
// because common subexpression elimination is
// not activated.
x := Solve(J, b); // J*x = b
{R3.p.i, R2.p.i} := x;
// Torn part
R1.n.i := -(R2.p.i+R3.p.i);
R3.v := R3.R*R3.p.i;
R1.v := Vs.v-R3.v;
// End of linear system of equations
```

To make the listing more readable, some optimization steps such as elimination of common subexpressions are not done. It means that Jacobians of algebraic loops are by default not listed, because without common subexpression elimination those expressions may be very long. As described above the listing of the non-zero Jacobian elements is enabled by issuing the command

```
Advanced.OutputModelicaCodeWithJacobians = true
```

The manipulated linear system is now output.

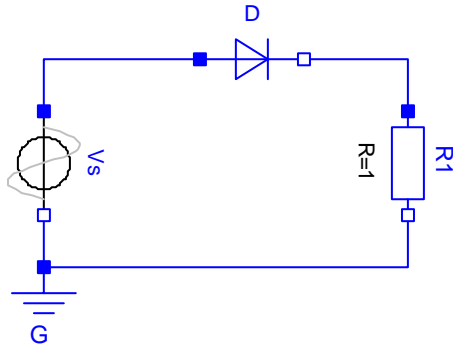
```
// Linear system of equations
// Matrix solution:
/* Original equations:
R1.R*R1.n.i = -R1.v;
R2.R*R2.p.i = R3.v;
*/
J[1, 1] := -(R1.R+R3.R);
J[1, 2] := -R1.R;
J[2, 1] := -R3.R;
J[2, 2] := R2.R;
b[1] := -Vs.v;
x := Solve(J, b); // J*x = b
{R3.p.i, R2.p.i} := x;
// Torn part
R1.n.i := -(R2.p.i+R3.p.i);
R3.v := R3.R*R3.p.i;
R1.v := Vs.v-R3.v;
// End of linear system of equations
```

Please, note that all element of the J matrix are non-literal expressions. Elimination of variables must not introduce divisions by zero. If we would like to use the first remaining equation to solve for the first unknown,  $R3.p.i$ , we need to divide by  $J[1, 1] := -(R1.R+R3.R)$ . Since it cannot be guaranteed that this expression always is non-zero, it is not a good idea to use this equation to eliminate  $R3.p.i$ . Thus to use an equation to eliminate a variable safely, its coefficient must be a non-zero numeric value. Since the Jacobian above has no numeric elements, it is not possible to eliminate variables further. We need to invert the matrix. It is indeed possible to do that for a two by two matrix and Dymola does it in some situations when generating simulation code for real-time and HIL simulation. However, in normal cases Dymola generates code for numeric solution, because it allows better support of singular systems.

### **Diode circuit with two valued resistance diode model**

Consider a simple electric circuit with a diode and a resistor connected in series.





Let the diode be modeled by the model

```
Modelica.Electrical.Analog.Ideal.IdealDiode.
```

It models the diode characteristic as a conductance in off mode and a resistance in leading mode. Thus in each of the two modes the problem is linear. There is an algebraic loop with five unknowns of which one of them, namely  $D.off$ , is a Boolean variable. The algebraic loop is a mixed system with one Boolean equation and four real equations.

```
// Mixed system of equations
// Linear system of equations
// Symbolic solution
/* Original equation
D.v = Vs.v-R1.v;
*/
D.s := (Vs.v-(R1.R*D.Goff*D.Vknee+D.Vknee)) /
      (R1.R*(if D.off then D.Goff else 1)
      +(if D.off then 1 else D.Ron));

// Torn part
D.i := D.s*(if D.off then D.Goff else 1)
      + D.Goff*D.Vknee;
R1.v := R1.R*D.i;
D.v := D.s*(if D.off then 1 else D.Ron)
      + D.Vknee;
// End of linear system of equations

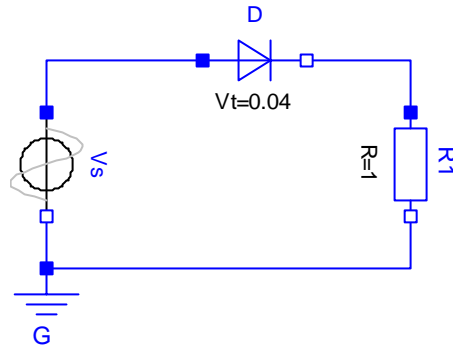
// Torn discrete part
D.off := D.s < 0;
// End of mixed system of equations
```

Dymola solves such a system by iterating. Assuming the Boolean variable to be known and removing the Boolean equation, the rest is a linear problem with four real unknowns. As seen, Dymola solves the linear part symbolically. During simulation, Dymola iterates, if the assignment  $D.off := D.s < 0$  changes the value of  $D.off$ .

### Diode circuit with exponential diode model

Let us revise the diode circuit above to use the diode model

Modelica.Electrical.Analog.SemiConductors.Diode



Its diode characteristic is nonlinear, with exponential terms. There is a nonlinear algebraic loop with three unknowns.

```
// Mixed system of equations
// Nonlinear system of equations
// It depends on the following parameters:
//   D.Ids
//   D.Maxexp
//   D.R
//   D.Vt
//   R1.R
// It depends on the following timevarying
// variables:
//   Vs.v
// discreteexpr_0.
// Unknowns:
//   R1.v(start = 0)
algorithm // Torn part
  D.v := Vs.v-R1.v;
  D.i := (if discreteexpr_0.then
    D.Ids*(exp(D.Maxexp)*
      (1+D.v/D.Vt-D.Maxexp)-1)
    +D.v/D.R
  else D.Ids*(exp(D.v/D.Vt)-1)+D.v/D.R);
equation // Residual equations
  0 = R1.R*D.i-R1.v;
// Non-zero elements of Jacobian
  J[1, 1] := (-1)-
    (if discreteexpr_0.then
      D.Ids*exp(D.Maxexp)/D.Vt+1/D.R
    else D.Ids*exp(D.v/D.Vt)/D.Vt+1/D.R)*R1.R;
// End of nonlinear system of equations
// Torn discrete part
  discreteexpr_0. := D.v/D.Vt > D.Maxexp;
// End of mixed system of equations
```

Dymola reduces it to a nonlinear problem with one iteration variable, namely

```
R1.v(start = 0)
```

The start value is included in the listing, because the nonlinear solver will use it. The diode model includes also an if-then-else expressions, where the condition,  $D.v/D.Vt > D.Maxexp$ , refer to one unknown,  $D.v$ , of the algebraic loop. Dymola introduces an auxiliary variable named “discreteexpr\_0.” for the condition. This transformation removes a possible discontinuity from the real part of the problem.

## Discriminating start values

When there are nonlinear algebraic loops, the nonlinear solver will use the start values of the iteration variables. The possibility of converging to a solution may depend critically on the quality of these start values. The quality of start values may vary considerably between the unknown variables. When it is possible to select between start values, the “best” start value should be chosen.

Dymola has taken the approach to consider a literal start less confident than a literal expression. When defining a basic quantity type or very generic model components, start values, if set, are typically given a numeric value. For example the start value of a pressure variable may be set to  $10^5$  Pascal. In a well-designed model library such as the Modelica Standard Library, the full-fledged components allow the user to specify initial conditions in a flexible way, typically by setting of parameters. These parameters are used to specify hard initial conditions ( $fixed=true$  or to control initial equations). However, the parameters are also used to specify start values when  $fixed=false$ . Thus, such a start value should be a more reliable estimation of the correct value, than a literal value set on a very general level.

Let us consider the situations where Dymola can select between start values. The very first situation is elimination of alias variables. A connection between non-flow connectors gives an equation  $v1 = v2$ . A connection between two flow connectors gives  $v1 + v2 = 0$ . Dymola exploits such simple equations to eliminate variables. In this elimination procedure Dymola keeps the start value.

The sorting procedure of Dymola finds the minimal loops, which means that the sorting is unique and such a loop cannot be made smaller by sorting the variables and the equations in another way. It means that the set of intrinsic unknowns of an algebraic loop is well-defined. In order to obtain efficient simulation, it is very important to reduce the size of the problem sent to a numerical solver. Dymola uses a tearing approach to “eliminate” variables. The numerical solver is only made aware of the remaining variables, the iteration variables, call them  $z$ . A numerical solver provides values for the  $z$  variables and would like to have the residuals of the remaining equations calculated. The tearing procedure has produced a sequence of assignments to calculate the eliminated variables,  $v$ , assuming  $z$  to be known. The start values of the eliminated variables have no influence at all. An aim is of course to make the number of components of  $z$  as small as possible. It is a hard (NP-complete) problem to find the minimum. However, there are fast heuristic approaches to finding good partitions of the unknowns into  $v$  and  $z$ . In order to get good start values for the numerical solver, Dymola tries first to eliminate variable with less reliable start values.

As for usual or iteration variables of nonlinear system of equations, the plot browser provides support for setting start values interactively. If the start value of the unknown is bound to a parameter expression, then setting any of the parameters appearing in the expression will of course influence the start value. If no start value is given or if it is a literal

number, then it is possible to set it interactively. In the plot browser, click on Advanced, and then click the button labeled,  $v_0^{\approx}$ , and the interactively setting is enabled.

Setting

```
Advanced.LogStartValuesForIterationVariables = true;
```

before translation, will make Dymola produce a listing of all iteration variables and their start values.

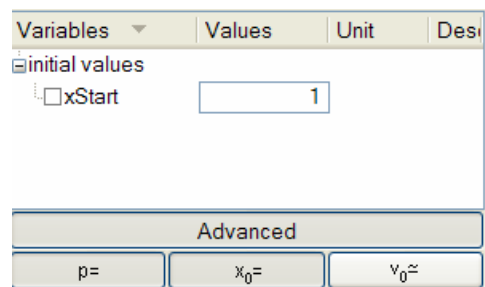
### Example

Consider the following example intended to illustrate both the improved heuristics and setting start-values:

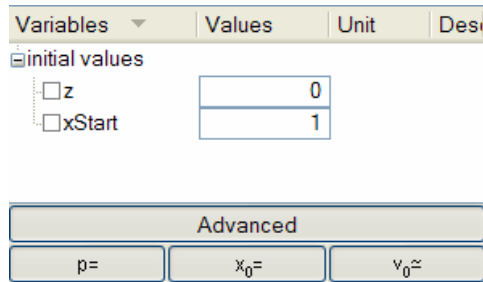
```
model NonLinear
  Real x(start=xStart);
  Real y;
  Real z;
  parameter Real xStart=1;
algorithm
  x:=(y+z)+time;
algorithm
  x:=(y+z)^2;
equation
  0=y-z;
end NonLinear;
```

Note: Using algorithm in this way for actual models is not good since Dymola manipulates algorithms less and thus algorithms often lead to harder numeric problems (larger system of equations, no analytic Jacobian, no alias elimination between  $y$  and  $z$ ). Rewriting them as equations would be a good idea.

Translating this model gives a prompt for initial values:



Enabling guess values ( $v_0^{\approx}$ ) gives the prompt



Setting 'z' to 1 generates another solution for this non-linear system of equations.

## Bounds checking for variables

Bounds checking for variables can be used to ensure that the solution is not only a numerical solution to the equations, but also satisfies additional bounds to and thus is physically correct.

Consider the following model where a length-constraint should be satisfied and the length shall be positive:

```

model LimitProblem
  Real length(min=0);
  equation
    length^2-length=1;
end LimitProblem;

```

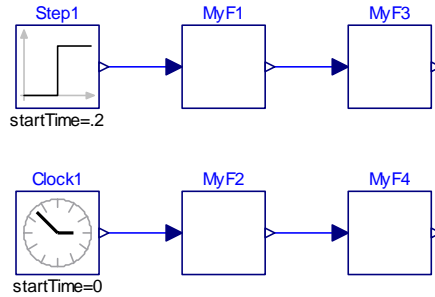
This equation has two solutions: -0.62 and 1.62. However, the solution at -0.62 does not satisfy the min-value. By enabling min/max assertions in Simulation/Setup/Debug (and allowing an error of 1e-6) it is guaranteed that an unphysical solution is not found and instead the physically correct solution at 1.62 is found.



Note that types in Modelica.SIunits contain min-values, and thus by using Modelica.SIunits some min-values are automatically applied. However, based on the actual model it might make sense to add stricter limits.

## Traceback message for errors in functions

When an error occurs in a function call it is important to know in which instance of the function that caused the problem. Consider the following simple model.



Each of the blocks contain the same function call, and in case one of them fails it is important to understand which one. This problem can be even more pronounced in larger models.

```

model MyF
  extends Modelica.Blocks.Interfaces.SISO;
  function f
    input Real x;
    output Real y;
  algorithm
    assert(x<2, "Cannot be larger than 2");
    y:=x*x/(1+x);
  end f;
  Real x;
equation
  der(x)=u;
  y=f(x);
  annotation (uses(Modelica(version="2.2")));
end MyF;

model WhichOne
  import Modelica.Blocks.Sources;
  MyF MyF1 annotation (extent=[-20,20; 0,40]);
  annotation (Diagram, uses(Modelica(version="2.2")));
  Sources.Step Step1(startTime=.2, height=4)
    annotation (extent=[-60,20; -40,40]);
  Sources.Clock Clock1 annotation (extent=[-60,-20; -40,0]);
  MyF MyF2 annotation (extent=[-20,-20; 0,0]);
  MyF MyF3 annotation (extent=[20,20; 40,40]);
  MyF MyF4 annotation (extent=[20,-20; 40,0]);
equation
  connect(Step1.y, MyF1.u)
    annotation (points=[-39,30; -22,30]);
  connect(Clock1.y, MyF2.u)
    annotation (points=[-39,-10; -22,-10]);
  connect(MyF2.y, MyF4.u)
    annotation (points=[1,-10; 18,-10]);
  connect(MyF1.y, MyF3.u)
    annotation (points=[1,30; 18,30]);
end WhichOne;

```

This model fails to simulate because of the assertion in f, but which of MyF1.f, MyF2.f, MyF3.f and MyF4.f is violating the assertion?

The log-message contains

```
Assertion failed: x < 2
The following error was detected at time: 0.7000000000000001
Cannot be larger than 2
The stack of functions is:
MyF.f
MyF.f(MyF1.x)
Integration terminated before reaching "StopTime" at T = 0.7
```

This clearly shows MyF1.f caused the error.

Note: Due to alias elimination the variable will in some cases not be the exact same variable, but can be an identical variable in the same or a connected sub-system.

This is active as default but it is possible to de-activate this additional diagnostics, since it adds to the size of the generated c-code and is only interesting if the model fails in some function.

## Direct link in error log to variables in model window

When looking at the error message for the WhichOne-model there is a tooltip for the component as follows:

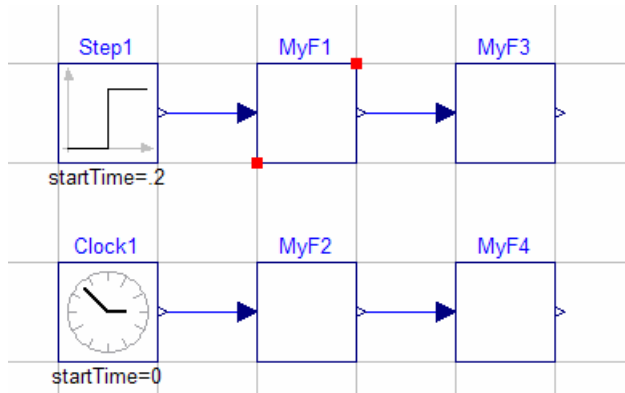
```
Assertion failed: x < 2
The following error was detected at time: 0.7000000000000001
Cannot be larger than 2
The stack of functions is:
MyF.f
MyF.f(MyF1.x)
Integration terminated before reaching "StopTime" at T = 0.7
CPU-time Context menu link: MyF1.x : 0.015 seconds
CPU-time for one GRID interval: 0.429 milli-seconds
```

Right-clicking on MyF1.x brings up the context-menu:

```
Assertion failed: x < 2
The following error was detected at time: 0.7000000000000001
Cannot be larger than 2
The stack of functions is:
MyF.f
MyF.f(MyF1
Integratio
CPU-tim
CPU-tim
Number
Number
Number of F-evaluations
Number of Jacobian-evaluations:

Copy Ctrl+C
Select All Ctrl+A
Find Ctrl+F
Show component ▶
MyF1
.x (Declaration)
```

Selecting MyF1 in this menu highlights the MyF1 component in the diagram:



The intention of highlighting the component is that an error can be due to the component itself, the parameters, or the interaction with connected components. By highlighting the component it is easy to investigate these.

Clicking on the last part (.x) brings up the text-layer of the component, and searches for the declaration of 'x'.

The function part of the function call has a similar link to the function.

## Extended online diagnostics for non-linear systems

When a simulation is slow it can be due to the non-linear systems of equations in the model. This is especially the case if the simulation is not yet started and the problem occurs for the initial equations.

As a contrived example consider:

```

model SlowNonLinear
  Real x[10];
  function multiplySlowly
    input Real x[:];
    output Real y[size(x,1)];
  algorithm
    y:=x;
    for i in 1:1000000 loop
      y:=x+y;
    end for;
  end multiplySlowly;
equation
  multiplySlowly(x)=ones(size(x,1));
end SlowNonLinear;

```

Running this and pressing Ctrl-c at the start gives:

```

Log-file of program dymosim
(generated: Wed Dec 21 11:59:03 2005)

dymosim started
... "dsin.txt" loading (dymosim input file)

```



```

In non-linear solver after 11 function evaluations:
x[10] = 0
x[9] = 9.99999E-007
x[8] = 9.99999E-007
x[7] = 9.99999E-007
x[6] = 9.99999E-007
x[5] = 9.99999E-007
x[4] = 9.99999E-007
x[3] = 9.99999E-007
x[2] = 9.99999E-007
x[1] = 9.99999E-007
Integration status probed at T = 0
  CPU-time for integration      : 2.55 seconds
  Number of result points      : 0
  Number of GRID points       : 1
  Number of (successful) steps : 0
  Number of F-evaluations      : 0
  Number of Jacobian-evaluations: 0
  Number of (model) time events : 0
  Number of (U) time events    : 0
  Number of state events       : 0
  Number of step events        : 0
  Minimum integration stepsize : 0
  Maximum integration stepsize : 0
  Maximum integration order    : 0

```

By pressing Ctrl-C twice instead of once the simulation is stopped and the user is additionally prompted for further commands:

```

Enter command: continue(c), quit(q), stop non-linear with
diagnostics(s), log event(le), log norm(ln), log
singular(ls)=log & allow singular systems log
iterationsnonlinear(li), log debugnonlinear(ld), log
resultnonlinear(lr)
Logging command syntax: log event true, log norm false, and log
norm reset

```

The last three log-commands correspond to the setup in Simulation/Setup/Debug/Non-linear solver diagnostics (iterationsnonlinear=Nonlinear iterations, debugnonlinear=Details, resultnonlinear = Nonlinear solution). To log all subsequent non-linear iterations write:

```

log iterationsnonlinear true
continue

```

To stop the simulation give the command 'quit' instead. The quit-command generates the same diagnostics as if the non-linear system of equations did not converge.

## Extended diagnostics for stuck simulation

When a simulation is stuck (or progressing very slowly) it is important to provide some form of simple diagnostics.

As an example consider the following example:

```
model StuckSimulation
  Real x(start=0.5);
equation
  der(x)=if x>0 then -1 else 1;
end StuckSimulation;
```

This model is a classical example of chattering after 0.5 seconds (when  $x$  has reached zero) since the derivative of  $x$  is  $-1$  for positive  $x$  and  $+1$  for negative  $x$ .

Such models can occur when manually writing e.g. friction elements such as clutches without a stuck mode (use `Modelica.Mechanics.Rotational.Clutch`). Adding ‘noEvent’ around the if-expression is not a solution.

### Diagnostics for example

Running this with `lsodar/dassl` gives a very slow progress, and it thus seems best to press ‘Stop’. This terminates the simulations and gives a log with:

```
Integration started at T = 0 using integration method DASSL
(DAE multi-step solver (dassl/dasslrt of Petzold modified by
Dynasim))
Integration terminated before reaching "StopTime" at T = 0.5
  WARNING: You have many state events. It might be due to
chattering.
  Enable logging of event in Simulation/Setup/Debug/Events
during simulation
  CPU-time for integration      : 9 seconds
```

Re-running and pressing `ctrl-c` in the `dymosim`-window gives a message:

```
Integration status probed at T = 0.5004871181
  WARNING: You have many state events. It might be due to
chattering.
  Enable logging of events by pressing ctrl-c twice and then:
log event true
continue
```

Enabling the logging in one of these ways gives a large number of messages of this type:

```
Expression x > 0 became false ( (x)-(0) = -3.93213e-013 )
Iterating to find consistent restart conditions.
  during event at Time : 0.50000000000003932
Expression x > 0 became true ( (x)-(0) = 1e-010 )
Iterating to find consistent restart conditions.
  during event at Time : 0.5000000001007865
Expression x > 0 became false ( (x)-(0) = -1.0025e-010 )
```

```
Iterating to find consistent restart conditions.  
during event at Time : 0.5000000003010365
```

The ‘x’ variables in the log are links as described in the section ‘Direct link in error log to variables in model window’.

By using a fixed-step solver, euler, the simulation runs to completion, but there is still chattering and thus the warning-message is given at the end of the successful simulation.

## Fast sampling

Another cause for slow simulations is that the model contains sampling at a high speed.

As an example consider

```
model RapidSampling  
  Real x;  
  equation  
    when sample(0, 1e-6) then  
      x=pre(x)+1;  
    end when;  
end RapidSampling;
```

Running this with dassl/lodar generates the diagnostics:

```
Integration terminated successfully at T = 1  
WARNING: You have many time events. This is probably due to  
fast sampling.  
Enable logging of event in Simulation/Setup/Debug/Events  
during simulation  
CPU-time for integration      : 57.1 seconds
```

The simulation does not stop because of this, but especially for a larger system it will be slower than normal, and can be a cause for concern. If the simulation is acceptably fast there is no need to enable the logging and investigate it further.

Using fixed-step-size solvers such as Euler does not generate any diagnostics, since running a sampled system with a step-size corresponding to the sampling rate is normal and not a cause for concern.

## Ensuring that ‘Stop’ stops the simulation

Simulations normally run to completion, but in some cases it is necessary to stop the simulation – either because the setup was incorrect (e.g. stop time 1e10 instead of 10) or because the simulation is progressing too slowly.

Simulations are normally stopped in a nice way in order to ensure that the user gets a complete result file including diagnostics (see ‘Extended diagnostics for stuck simulation’). However, in extreme examples a model might be stuck in an infinite loop.

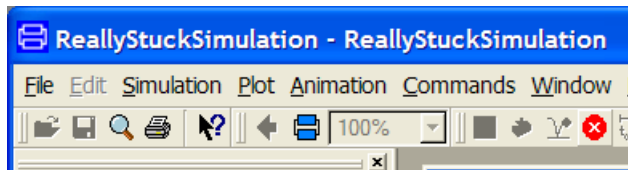
Consider a user writing a variant of the rem-function in Modelica (the intention is that ‘x’ should be a sawtooth-shape), but mixes up plus and minus:

```

model ReallyStuckSimulation
  Real x;
algorithm
  x:=time;
  while x>0.2 loop
    x:=x+0.2; // Subtract offset.
  end while;
end ReallyStuckSimulation;

```

Running this example gives a stuck simulation after 0.2 seconds, and the user should therefore press the red Stop-button highlighted below:

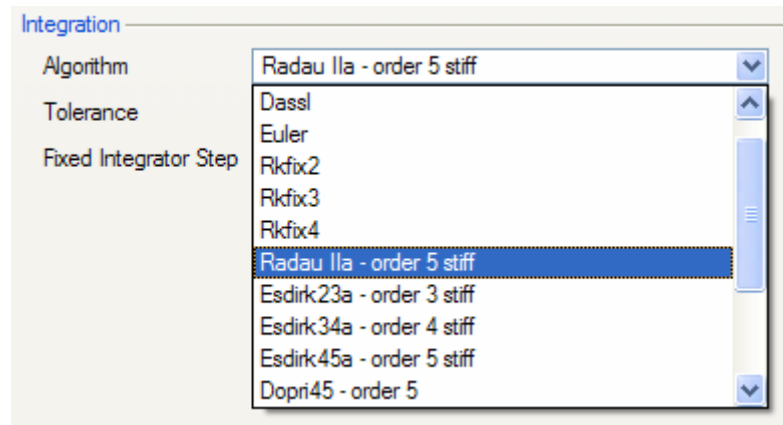


After waiting for a half a minute the simulation-process (dymosim) will be terminated. The waiting period is used to ensure that the dymosim-process is terminated in a nice way if possible, even for larger examples.

## New integration algorithms

In Dymola a new set of integration algorithms has been added. We will here present the algorithms and their advantages in order to allow you make a better choice, both between them and compared to the traditional ones. The main user-benefit is better restart after events, the addition of non-stiff integrators with variable step-size, and solvers better suited for badly damped stiff systems (i.e. with poles close to the imaginary axis).

The new algorithms are included in the integrator setup making it straightforward to switch between the traditional dymosim integrators and the new ones (marked with order and the methods applicable to stiff systems are marked with 'stiff' – the first one is 'Radau ...').



Changing between the new solvers and the normal solvers will cause a recompilation (but not a complete retranslation). Linearizing a model will use the dymosim-solvers and might thus cause a recompilation. For a user this implies that one should preferably select the integration algorithm before pressing ‘Translate’ (or start by pressing ‘Simulate’).

### Overview of the new algorithms

The new algorithms (as well as the remaining traditional methods described in the dymosim-chapter of the manual) support the entire range of events present in Modelica models, i.e. state events, time events, and dynamic state selection. The table below gives their characteristics (the combo-box contains the same information).

Method	Order	Stiff
Radau IIa	5	Yes
Esdirk23a	3	Yes
Esdirk34a	4	Yes
Esdirk45a	5	Yes
Dopri45	5	No
Dopri853	8	No
Sdirk34hw	4	Yes
Cerk23	3	No
Cerk34	4	No
Cerk45	5	No

Variable (or adaptive) step-size implies that the algorithm adapts the step-size to meet a local error criterion based on the tolerance.

One-step (or Runge-Kutta) methods are basically designed such that they start fresh on every step and thus the cost of restarting them after an event is substantially reduced compared to multi-step methods such as lsodar (implementing Adams-methods) and dassl (implementing BDF-methods). However, even if the methods are one-step methods the implementation often uses more information from the previous step.

Fixed order mean that you manually select the method including order (where higher order should be used for stricter tolerance) instead of the solver automatically adapting the order as for lsodar and dassl.

Dense output implies that the method can handle state events efficiently and also produce evenly spaced output. The dense output has traditionally been added as an afterthought to the methods. Good exceptions are the cerk-methods, where the method coefficients were optimized including the dense output.

Most of the solvers are stiff-solvers indicating that they are suited for stiff systems, i.e. systems where the fastest time-scale in the model is substantially faster than the interesting dynamics.

The new stiff algorithms are also designed to be A-stable, i.e. stable for all stable linear systems. This means that the methods are better suited for badly damped stiff systems (i.e. with poles close to the imaginary axis). Furthermore, since they start with higher order they are more suited for systems with discontinuities or events.

## Analytic Jacobians

For non-linear systems of equations it is possible to avoid numeric Jacobians and instead rely on Dymola to automatically differentiate the functions. Compared to writing derivative functions this is much easier for the modeler, easier to understand for the user, and also considerably less error-prone.

In order to enable Dymola's automatic differentiation feature, the modeler writing functions must declare the smoothness of the function by providing a smoothOrder-annotation corresponding to the smooth operator in Modelica. A basic limitation of automatic differentiation is that it can provide a derivative even at points where the function does not have a derivative. Verifying that a function with branches (if-statements, if-expressions, or while-statements) is continuous is a difficult problem. The person providing the smoothOrder-annotation is guaranteeing that the function is at least that smooth. When using the function, its derivative is only constructed if it is found to be needed because of index reduction or to generate an analytic Jacobian.

The basics of automatic differentiation and the implementation choices in Dymola are discussed in H. Olsson, H. Tummescheit and H. Elmqvist: "Using automatic differentiation for partial derivatives in Modelica", Proceedings of the 4<sup>th</sup> International Modelica Conference, Hamburg-Harburg, Germany, 2005, pp. 105-112.

### Example

We will use a simple function that just inverts a strictly positive number for illustration:

```
function MyDivision
  input Real x;
  output Real y;
  annotation (smoothOrder=1000);
algorithm
  assert(x>0, "x should be positive");
  y:=1/x;
end MyDivision;
```

We then write a simple model where this function must be differentiated in order to solve a non-linear equation:

```
model TestDivision3
  Real x;
  equation
  MyDivision(x)=1+time;
```

```
end TestDivision3;
```

Translating this example gives a translation log with

```
...  
Sizes of nonlinear systems of equations: {1}  
Sizes after manipulation of the nonlinear systems: {1}  
Number of numerical Jacobians: 0
```

An analytic Jacobian is constructed and used since needed. The derivative function is:

```
function P.TestDivision3.MyDivision:derf  
  input Real x;  
  protected  
    Real y;  
  public  
    input Real x_der2;  
    output Real y_der2;  
  algorithm  
    assert(x > 0, "x should be positive");  
    y_der2 := -x_der2/x^2;  
  annotation (smoothOrder=999);  
end P.TestDivision3.MyDivision:derf;
```

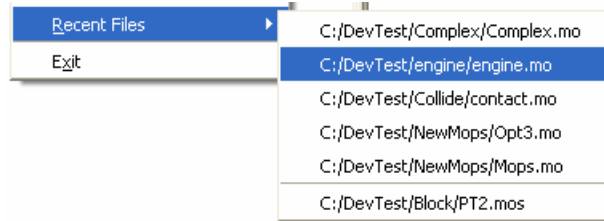
Removing the smoothOrder-annotation instead gives

```
...  
Sizes of nonlinear systems of equations: {1}  
Sizes after manipulation of the nonlinear systems: {1}  
Number of numerical Jacobians: 1
```

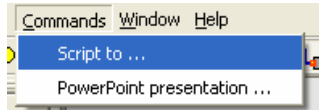
## Commands and Scripting

It is possible to use command line arguments when starting Dymola, for example by making a short cut and associate either a mo- or mos-file. After Dymola has started a mos-file is executed by RunScript() or a mo-file is handled by openModel().

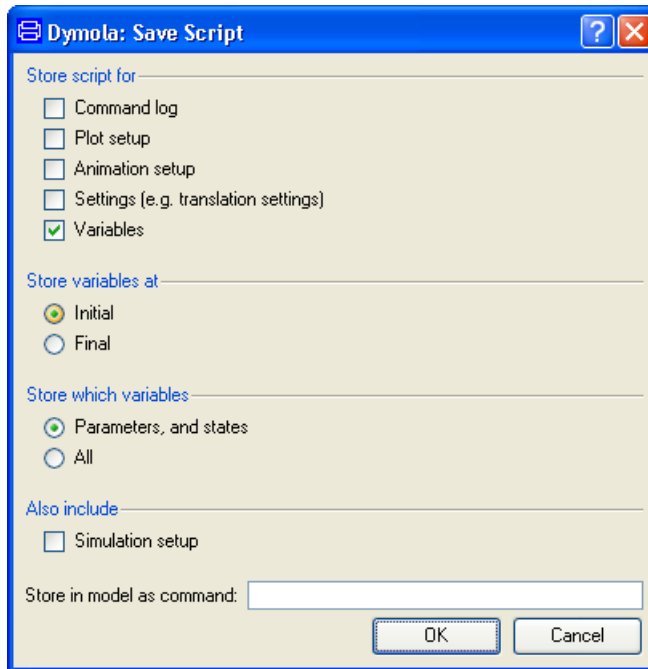
The command File / Recent Files shows a submenu for the most recently opened model files or executed scripts during the current Dymola session. The content of File/Recent Files is saved between Dymola sessions.



Possible to include calls to scripts and opening of documents in menu Commands



Command File / Save script to save various settings in a script. The command log can be saved. Alternatively switch settings, plot and animation setup, and variables can be saved. Either all variables or only parameters and initial values of states can be saved either as specified initial values or as the final values after simulation.



It is possible to specify that the generated script shall be callable from the Commands menu associated with the current model, see section ‘Commands menu’ on page 36.

Selecting models in File/Demo does no longer change current directory.

Import statement can be used on command line.

```
import Modelica.Math.*
= true
sin(1)
= 0.841470984807897
```

Redeclaration of size for variables in work space allowed.

```
a=1:10
Declaring variable: Integer a [10];
```



```
a=1:20
Redeclaring variable: Integer a [20];
```

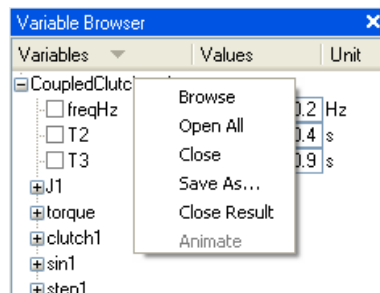
New built-in functions to write to log window: `Utilities.ModelicaMessage("message");`

---

## Plotting and animation

### Variable browser context menu

The top-level nodes in the variable browser represent simulation result files; other nodes represent the component hierarchy and variables and parameters at the lowest level. The variable browser has a context menu with several important operations.



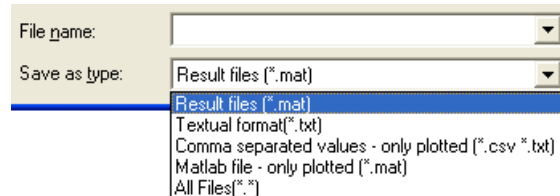
The first three choices are available for all nodes in the variable browser.

- Clicking on a “+” opens one more level of the tree. The “Browse” operation opens two levels, which makes the model structure clearer without creating a huge browse tree.
- The “Open All” operation opens every sub-node in the browse tree. Large sub-trees may become very large and hard to navigate.
- Clicking on “-” will close a node. The “Close” operation will close all nodes in the browse tree. The difference is that the next time you open the node, all nodes will be closed.

The last three commands in the menu are only available for top-level (result file) nodes.

- The “Save As” operation allows the user to save the result file in several different file formats. It is possible to either store the whole result file, or just those signals which are currently plotted.
- The “Close Result” operation will delete the result file from Dymola and free the occupied storage space.
- The “Animate” operation will animate the data in the selected result file. This operations requires that an animation window is open and that the result file contains animation data.

The “Save As” operation can store data in several different formats.



First, data can be stored as Dymola result files (.mat format), as text, or as comma-separated values (suitable for Microsoft Excel and other applications).

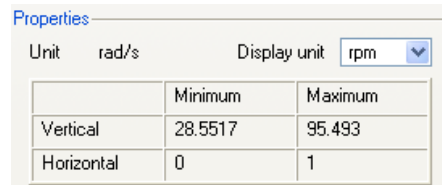
- Comma separated values: For use in e.g. Excel  
Note: some versions of Excel uses the Regional Setting of List Separator when reading CSV-files, please set this to ','
- Matlab-format: easy to use in Matlab and can also be re-opened as a result in Dymola

The file types marked with “only plotted” will only store the signals which are currently plotted.

## Display units

It is possible to change the display unit of a signal in the plot window, for example from *Pa* to *MPa* or from *rad/s* to *rpm*, if suitable unit conversions have been defined. Changing the display unit for a signal will change the display unit for *all signals* in the diagram with the same display unit. Diagram tooltips show units.

After a signal has been plotted, the display unit can be changed in Plot/Setup. The display unit is chosen from a list of known derived units. The display unit must be changed for each plotted signal. The last selected display unit becomes the default display unit when another signal with the same fundamental unit is plotted.



Unit conversions and the initial default display unit can be specified in command scripts as needed. For example,

```
defineUnitConversion("Pa", "MPa", 1e-6, 0);  
// scale and offset  
defineDefaultDisplayUnit("Pa", "MPa");  
// use MPa by default
```

The set of common unit conversions which are used by default can be found in `dymola/insert/displayunit.mos`. Additional US unit conversions are defined in `dymola/insert/displayunit_us.mos`; which are used by default can be changed in `dymola/insert/dymola.mos`.

## Other plotting

The command `plotArray()`, which is used to plot data computed in functions or scripts, has been extended with new parameters.

x		X-values
y		Y-values
style	0	Style of plotting (default automatic)
legend	""	Legend describing plotted data (default none)
id	0	Identity of window (default 0 means last)

The resolution has been improved for plotted variables which are copied to clipboard with `File/Export/To Clipboard` or printed on high-resolution printers.

Some programs (for example Microsoft Word 97) may have problems when high-resolution plots are pasted. We suggest using `Edit/Paste Special` and selecting `Enhanced Metafile Format`. It is possible to set the resolution for high-resolution printing or export to clipboard. The variable is called `Advanced.ClipboardResolution`; default is 600 dpi. It is also possible to export low-resolution plots in Dymola by setting `Advanced.PrintLowRes = true`; this will print and copy the plot window using screen resolution, which is compatible with earlier versions of Dymola.

Derivative variables look smoother in plots. Plotting of derivative-variables when using the Euler method is no longer influenced by interpolation for finding events. This improvement is only needed for plots and not for the actual solution.

## Animation

Direct manipulation of the view in the animation window using the mouse has been implemented. The view can be panned, rotated and zoomed using mouse movements in combination with meta keys:

Operation	Meta key	Mouse move
Pan view	none	Up/Down/Left/Right
Rotate around x-axis	Ctrl	Left/Right
Rotate around y-axis	Ctrl	Up/Down
Roll (rotate around z-axis)	Ctrl+Shift	Clockwise/Counter-clockwise
Zoom in/out	Shift	Up/Down
Zoom in/out	none	Wheel

In addition, arrow keys pan and tilt in fixed increments of 5 degrees, page up/down tilt 45 degrees. The “Home” key resets viewing transformation.

To better support model portability, `dxs`-files are found relative to the directory of the current model.

Texture mapping for animation objects is supported. Please contact Dynasim for more information. Example of animation window:



---

## Matlab and Simulink

Matlab 7.1 (R14SP3) is supported.

Dymola now generates Simulink S-function level 2 (unless you are using Matlab 5).

The library annotation for external functions is automatically used also in Simulink. For those who have worked around this in other ways it can be turned off using

```
Advanced.IncludeLibrariesForSimulink=false;
```

Support for compiling models with dSpace release 4.2.

Dymola-generated models can be run on two more realtime platforms: ADI SimSystem and ETAS LABCAR.

---

## Libraries

### Modelica Standard Library version 2.2

Version 2.2 is backward compatible to version 2.1.

**Modelica.Blocks** has revised table blocks to avoid multiple allocations of table space.

**Modelica.Mechanics.MultiBody** has been revised. See MSL 2.2 Release Notes for details.

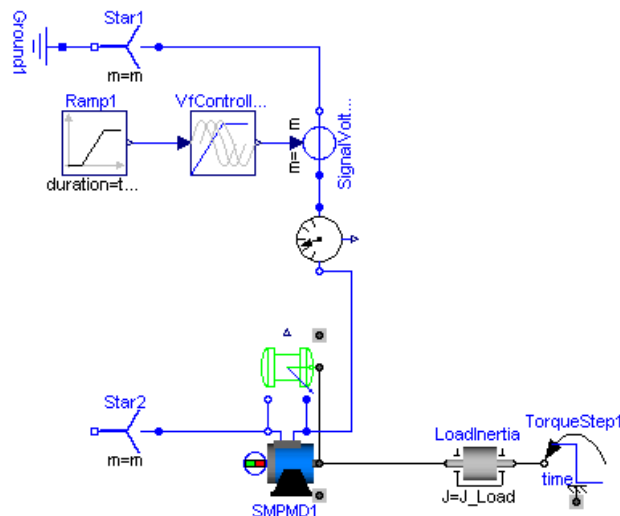
The following new libraries have been added:

**Modelica.Media** - Property models of liquids and gases, especially 1241 detailed gas models, moist air, high precision water model and incompressible media defined by tables. The user can conveniently define mixtures of gases between the 1241 gas models. The models are designed to work well in dynamic simulations. They are based on a new standard interface for media with single and multiple substances and one or multiple phases.

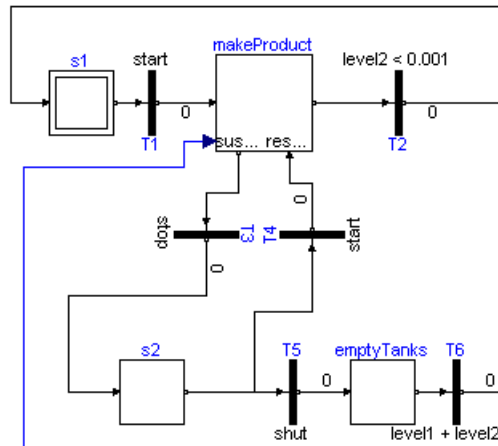
**Modelica.Thermal.FluidHeatFlow** - Simple components for 1-dim., incompressible thermo-fluid flow to model coolant flows, e.g., of electrical machines. Components can be connected arbitrarily together (= ideal mixing at connection points) and fluid may reverse direction of flow.

**Modelica.Electrical.Digital** - Digital electrical components based on 2-,3-,4-, and 9-valued logic according to the VHDL standard

**Modelica.Electrical.Machines** - Asynchronous, synchronous and DC motor and generator models. The example shows a permanent magnet synchronous induction machine with inverter.



**Modelica.StateGraph** - Modeling of discrete event and reactive systems in a convenient way using hierarchical state machines and Modelica as action language. It is based on JGraphChart and Grafcet and has a similar modeling power as StateCharts. It avoids deficiencies of usually used action languages. This library makes the ModelicaAdditions.PetriNets library obsolete. The example shows the controller for a tank system.



**Modelica.Math.Matrices** - Functions operating on matrices such as solve() ( $A*x=b$ ), leastSquares(), norm(), LU(), QR(), eigenValues(), singularValues(), exp(), ...

**Modelica.Utilities** - Functions to operate on files, streams, strings and support for operations in the operating system

## Comparison to Modelica Standard Library 1.6

Modelica 2.1 and 2.2 are major changes with respect to 1.6 (and previous) versions of the Modelica Standard Library, because many new libraries and components are included and because the input/output blocks (Modelica.Blocks) have been considerably simplified:

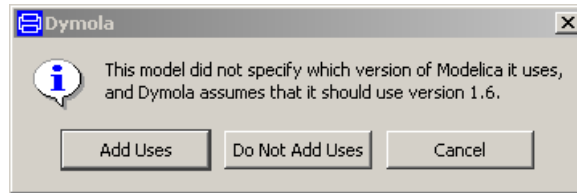
- An input/output connector is defined without a hierarchy (this is possible due to new features of the Modelica language). For example, the input signal of a block "FirstOrder" was previously accessed as "FirstOrder.inPort.signal[1]". Now it is accessed as "FirstOrder.u". This simplifies the understanding and usage especially for beginners.
- De-vectorized the Modelica.Blocks library. All blocks in the Modelica.Blocks library are now scalar blocks. As a result, the parameters of the Blocks are scalars and no vectors any more. For example, a parameter "amplitude" that might had a value of "{1}" previously, has now a value of "1". This simplifies the understanding and usage especially for beginners. If a vector of blocks is needed, this can be easily accomplished by adding a dimension to the instance. For example "Constant const[3](k={1,2,3})" defines three Constant blocks. An additional advantage of the new approach is that the implementation of Modelica.Blocks is much simpler and is easier to understand.

All components of the **ModelicaAdditions** library are included in the Modelica Standard Library in an improved way:

- ModelicaAdditions.Blocks is included in Modelica.Blocks. The logical blocks have a nicer icon layout now.

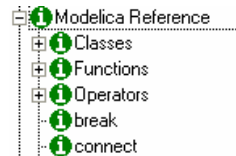
- `ModelicaAdditions.Tables` is included in `Modelica.Blocks.Sources` and `Modelica.Blocks.Tables`.
- `ModelicaAdditions.MultiBody` is obsolete and is replaced by the much more powerful library `Modelica.Mechanics.MultiBody` (this is version 1.0.1 of the `MultiBody` library where the signal connectors have been changed to the new signal connectors).
- `ModelicaAdditions.HeatFlow1D` is obsolete since a long time. It was replaced by the improved library `Modelica.Thermal.HeatTransfer`.
- `ModelicaAdditions.PetriNets` is obsolete and is replaced by the much more powerful library `Modelica.StateGraph`.

When opening models built using early versions of the Modelica Standard Library, the following dialog is shown. Click on “Add Uses” to update the model so this dialog is not shown in the future.



## Other libraries

Modelica Reference documentation as a package. See the package browser.



Library **Modelica\_LinearSystems** is a free Modelica package providing different representations of linear, time invariant differential and difference equation systems, as well as typical operations on these system descriptions. See the Users Guide inside the package for details.

The package “DataFiles” has been updated with functions to write and read CSV (comma separated values)-files. CSV-files are for example, generated and read by MS Excel. The function `DataFiles.readCSVmatrix("fileName")` reads the data into a matrix skipping the first line if containing textual legends. The data can be separated by tab, semicolon or comma.

## Library handling improvements

If the string \$DYMOLA/Modelica/Library is not found in MODELICAPATH it is added first (and not last). The environment variable MODELICAPATH specifies a semi-colon separated list of directories where packages will be searched for.

The handling of conversion to Modelica 2.2 (and also other conversions) has been substantially improved:

- See [separate document for more information](#).
- Sources with vector arguments automatically converted to array of components.
- Automatic save a script for conversion of models using the converted package (when converting a package with its own version number).
- Can convert `a.inPort.signal[1]` to `a.u`, and `a.inPort.signal` to `{a.u}`.
- The conversion script for Modelica 2.2 has been improved.

LAPACK library for GCC and Visual C++ (required by packages Matrices and Sampled).

---

## Installation and setup of Dymola

Dymola can be installed in directories with spaces, for example Program Files. If the default working directory “dymola/work” is not writeable for the current user, Dymola will instead start in the directory “Dymola” in the user’s “My documents” folder (the “Dymola” subdirectory will be created if it does not exist). Please note that this cannot be an UNC-path (i.e. [\\server\...](#)).

If dymola/tmp is not writeable for the current user, Matlab/Simulink compilation will still work but all files will be recompiled. In combination with the change for compiler setup this means that users do not need write-access to the Dymola installation directory.

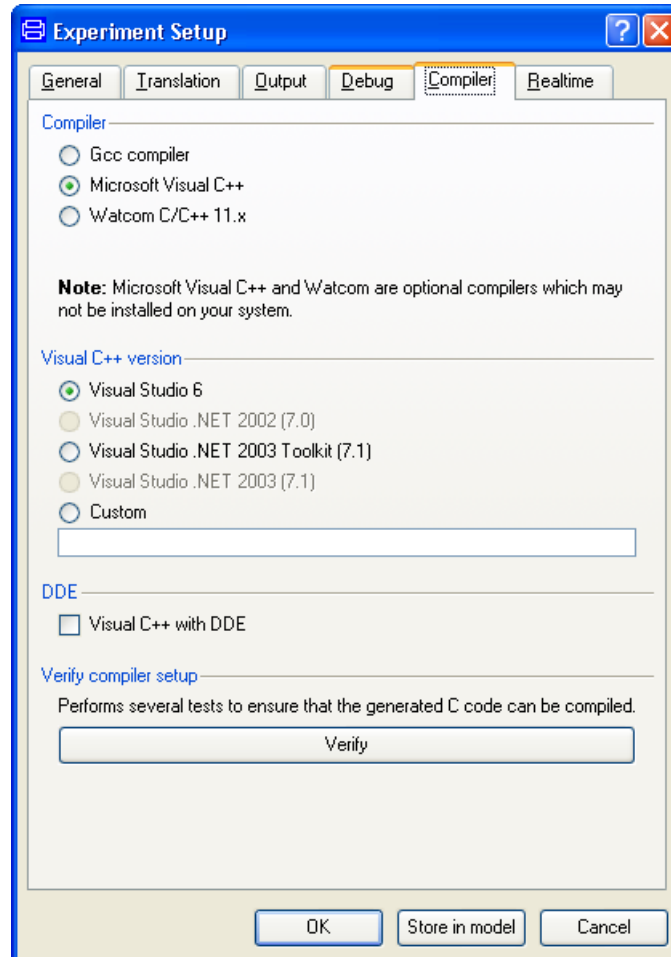
The Dymola 6 executable is now called dymola.exe, and the version number is removed from several files (including this one). The Simulink interface starts Dymola 6.

The setup for compilers has been redesigned and more compilers are supported. Dymola dynamically recognizes Visual C++ compilers (no need to install them before Dymola).

- Microsoft Visual Studio 6
- Microsoft Visual Studio .NET 2002 and 2003
- Microsoft Visual Studio .NET 2003 Toolkit. This is a free compiler for Windows 2000 and Windows XP, which can be downloaded from <http://msdn.microsoft.com/visualc/vctoolkit2003>.



The selected compiler is stored as a per-user setting and for the future kept for new installations of Dymola. Switching compiler does not modify dymola/bin.



Dymola displays a specific diagnostic message when the user runs Microsoft Windows Terminal Services, which for security reasons is not supported.

Dymola supports external C libraries on Linux. Classes which contain “Library” annotations to link with external libraries in C are supported on both Windows and Linux.



# **Modelica Data Structures and GUI**



# Modelica Data Structures and GUI

In addition to primitive data types, Real, Integer, Boolean and String and from them derived types, Modelica has records and arrays. We will in this section show how to build graphical user interfaces for models and functions that correspond to these data structuring mechanisms.

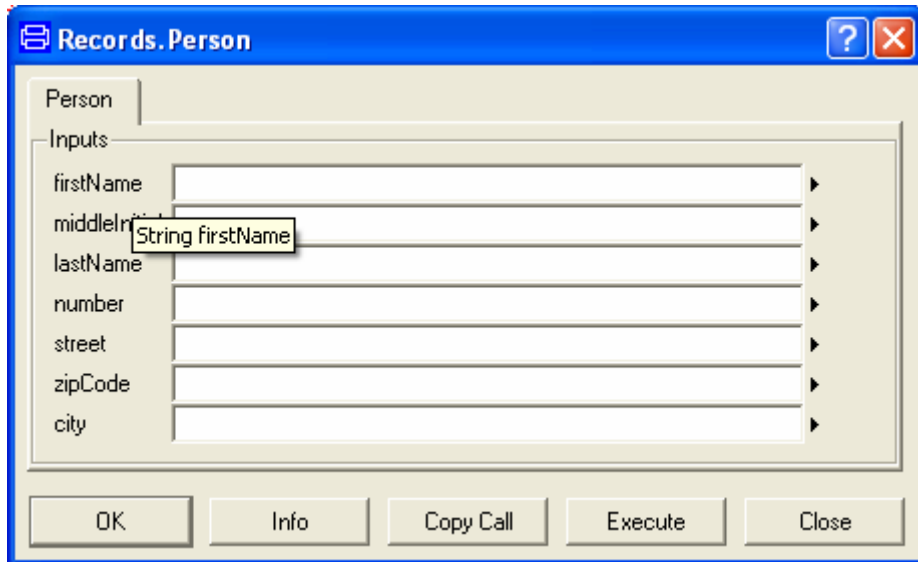
---

## Records and dialogs

As an introductory example, we will consider making a small data base of personal data. Assume that each person is described by the following information:

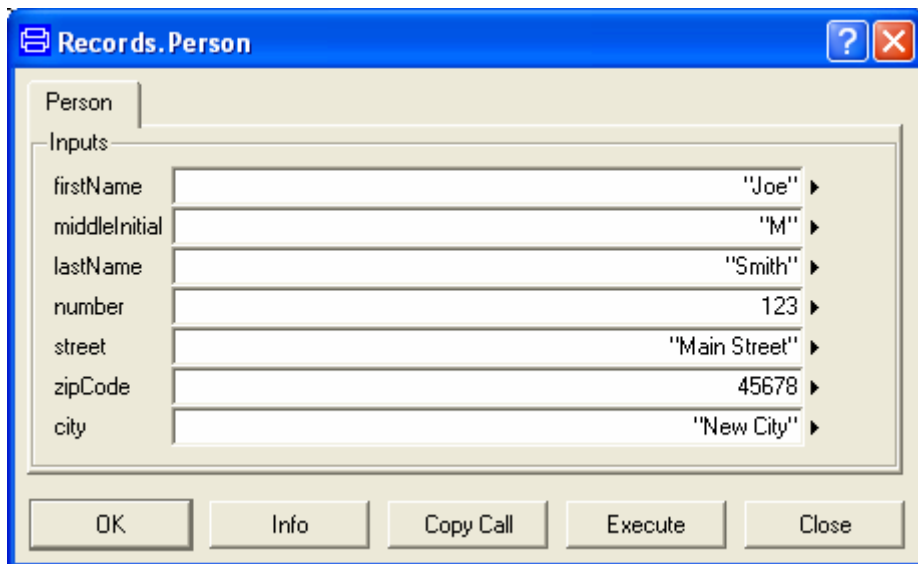
```
record Person
  String firstName;
  String middleInitial;
  String lastName;
  Integer number;
  String street;
  Integer zipCode;
  String city;
end Person;
```

The corresponding automatically constructed GUI dialog for entering data looks as follows:



The tool tip shows the data type of the input field.

Entering the following data:

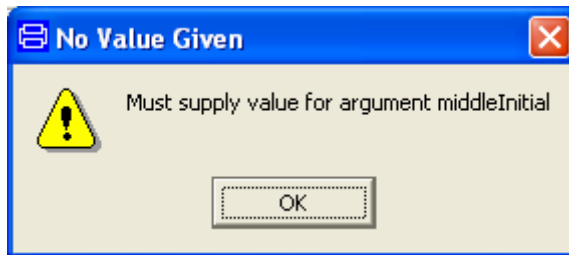


and pressing the "OK" or "Execute" buttons gives the result in the log window as a call to the record constructor `Records.Person` with the name-value pairs for the entered data.

```
Records.Person(  
    firstName = "Joe",  
    middleInitial = "M",
```

```
        lastName = "Smith",  
        number = 123,  
        street = "Main Street",  
        zipCode = 45678,  
        city = "New City"  
    )
```

If we would not fill in any value for middleInitial, the following error message would be generated:



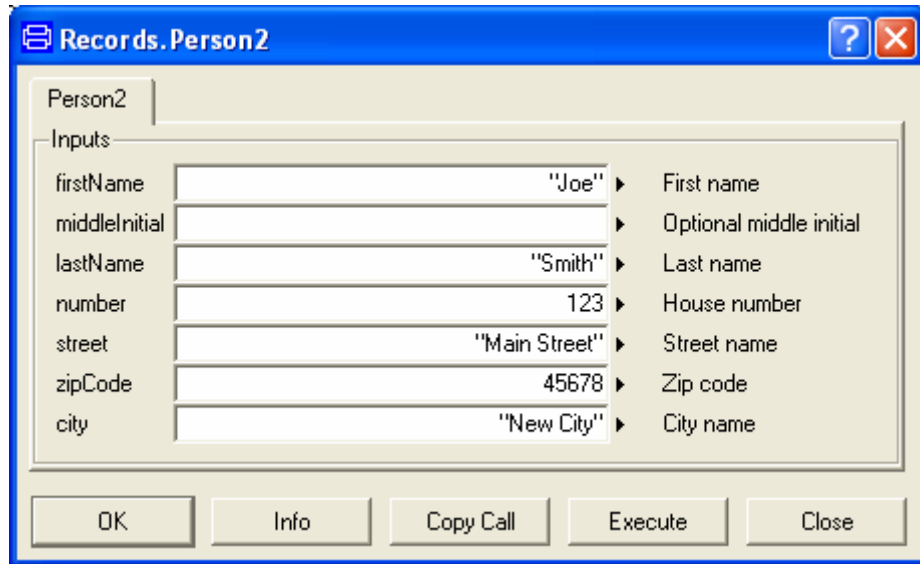
To avoid having to give such data, a default value can be given in the declaration:

```
String middleInitial = "";
```

Modelica allows you to add description strings to all variables:

```
record Person2  
    String firstName "First name";  
    String middleInitial="" "Optional middle initial";  
    String lastName "Last name";  
    Integer number "House number";  
    String street "Street name";  
    Integer zipCode "Zip code";  
    String city "City name";  
end Person2;
```

These are used to annotate the dialog as shown below.

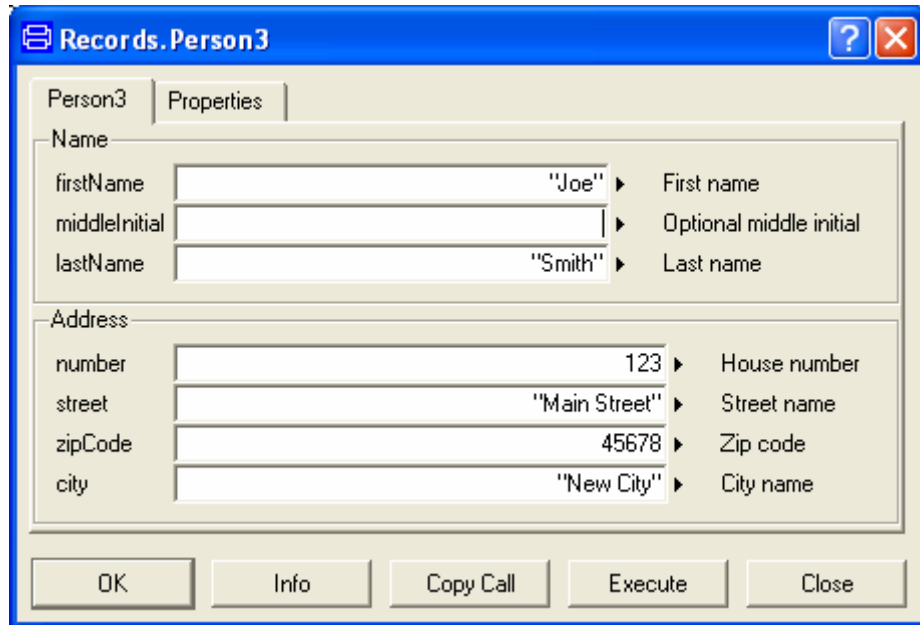


## Tabs and Groups

It is possible to annotate input fields in various ways in order to simplify for the user to enter data.

It is, for example, possible to group record fields together and introduce tabs in the dialog by means of annotations.





These changes are made by adding the following annotations and extending the record with field married.

```

record Person3
  String firstName "First name"
    annotation (Dialog(group="Name"));
  String middleInitial="" "Optional middle initial"
    annotation (Dialog(group="Name"));
  String lastName "Last name"
    annotation (Dialog(group="Name"));
  Integer number "House number"
    annotation (Dialog(group="Address"));
  String street "Street name"
    annotation (Dialog(group="Address"));
  Integer zipCode "Zip code"
    annotation (Dialog(group="Address"));
  String city "City name"
    annotation (Dialog(group="Address"));
  Boolean married "Marital status"
    annotation (Dialog(tab="Properties", group="Marital
      status"));
end Person3;

```

Note, that for the Boolean field married, the combobox with choices false and true appear automatically.

## Labels and layout

By annotating a field, such as `firstName`, with the attribute `joinNext=true`, the next field, `middleInitial`, is put on the same horizontal line as `firstLine`.

Instead of having the variable name in front of the input field, the description string is used if the Dialog annotation: `descriptionLabel=true` is given. The description string is then not shown after the input field. A label with free text can be given by `label="free-text"`. The free text label has precedence over the description label.

The width of the inputs fields can be specified as, for example, `naturalWidth=10`. The width is given in the unit "en", the width of character '0'. The width can also be specified as `absoluteWidth=10`. The difference is that fields with `absoluteWidth` keep their size when the entire dialog is made wider. The fields with `naturalWidth` specification are made wider.

By use of these annotations we can make the dialog much nicer.



The details of the record declaration is given below:

```
record Person4
    String firstName "First name"
        annotation (Dialog(group="Name", joinNext=true,
            naturalWidth=15, descriptionLabel=true));
    String middleInitial="" "Middle initial"
        annotation (Dialog(group="Name", joinNext=true,
            absoluteWidth=3, descriptionLabel = true));
    String lastName "Last name"
        annotation (Dialog(group="Name", naturalWidth=25,
            descriptionLabel = true));

    Integer number "Number"
        annotation (Dialog(group="Address", joinNext=true,
            absoluteWidth = 10, descriptionLabel = true));
```

```

String street "Street name"
    annotation (Dialog(group="Address", descriptionLabel
        = true));
Integer zipCode "Zip code or postal code"
    annotation (Dialog(group="Address", joinNext=true,
        absoluteWidth = 10, descriptionLabel = true,
        label="Postal code"));
String city "City name"
    annotation (Dialog(group="Address",
        descriptionLabel = true));

Boolean married "Marital status"
    annotation (Dialog(tab="Properties",
        group="Marital status", absoluteWidth=10));
end Person4;

```

## Alternative forms for input fields

Sometimes there is a set of frequent input values (enumerations) and in addition free text should be possible. For such cases, it is possible to add a combo box for the frequent choices. This would, for example, be convenient for a sex field:

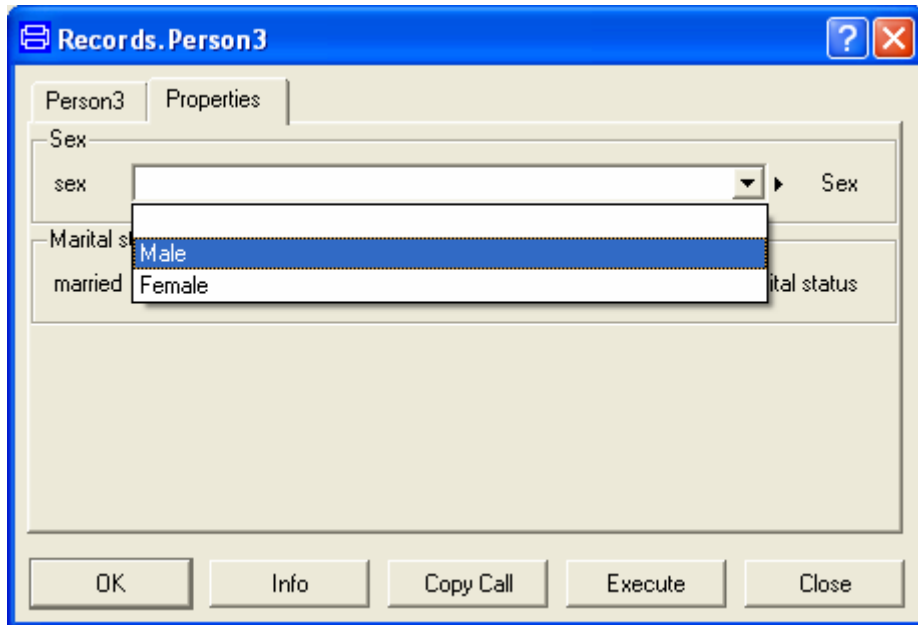
```

Integer sex "Sex"
    annotation (Dialog(tab="Properties", group="Sex"),
        choices(choice=1 "Male", choice=2 "Female"));

```

Associated with each value (1, 2), it's possible to give a description string ("Male", "Female").

The Properties tab has the following layout after this addition.



In the case of only a set of fixed choices, radio buttons are more appropriate. Specification of sex can, for example, be made by radio buttons by adding `radioButtons=true`, i.e. if the following declaration is given:

```
Integer sex "Sex"
  annotation (Dialog(tab="Properties", group="Sex"),
    radioButtons=true,
    choices(choice=1 "Male", choice=2 "Female"));
```

Use of enumeration types would have been more appropriate instead of Integer. However, Dymola does not support enumerations yet. Boolean variables such as

```
Boolean married "Marital status"
  annotation (Dialog(tab="Properties",
    group="Marital status"));
```

give by default a combo box with choices false and true. However, in many cases a check box is more convenient. This is achieved by adding `checkBox=true`, i.e. by giving the declaration

```
Boolean married "Marital status"
  annotation (Dialog(tab="Properties",
    group="Marital status"), choices(checkBox=true));
```

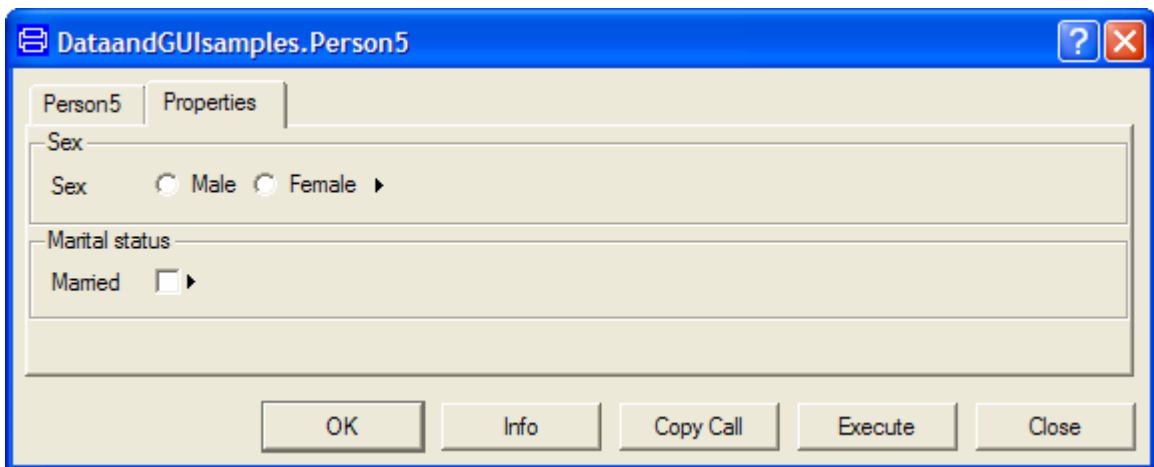
By adding these declarations for sex and married

```
Integer sex "Sex"
  annotation (Dialog(tab="Properties", group="Sex",
```

```
compact=true, descriptionLabel = true),
choices(choice=1 "Male", choice=2 "Female",
radioButtons=true));
```

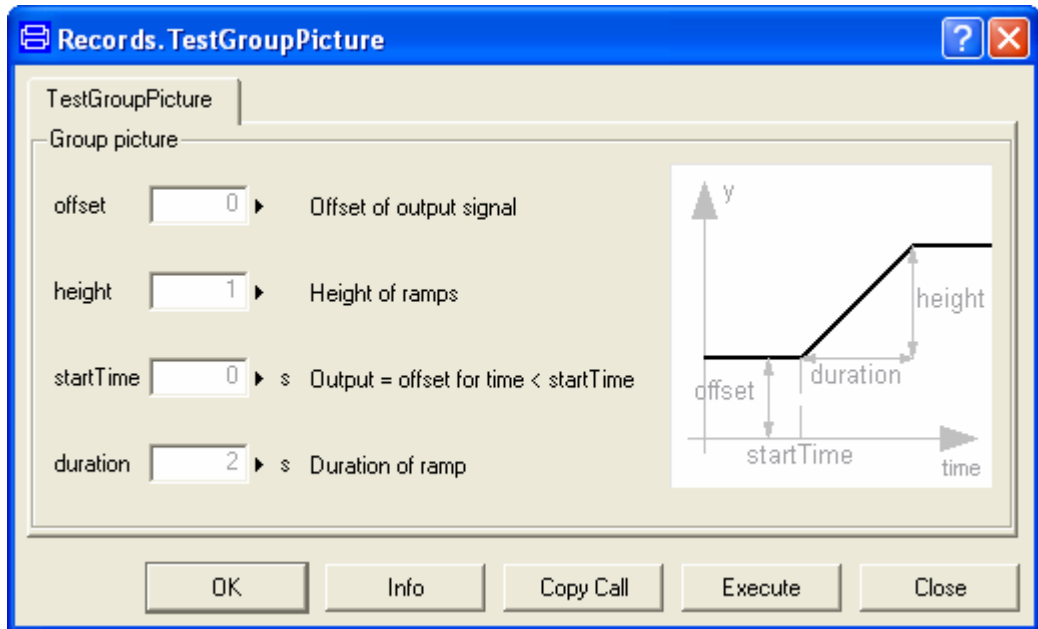
```
Boolean married "Married"
annotation (Dialog(tab="Properties",
group="Marital status",
compact=true, descriptionLabel = true),
choices(checkBox=true));
```

including `compact=true` to move triangle closer to the input field, we obtain the following dialog layout:



## Illustrations and formatting in dialogs

To make it easier to understand the meaning of input data, it's possible to associate a picture with a Group:



The record declaration including the annotation to specify the file name of the picture is shown below:

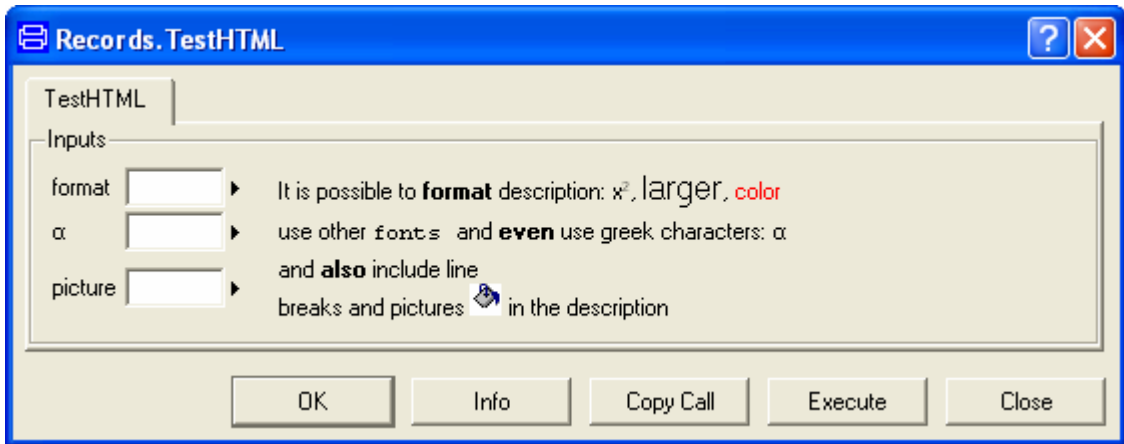
```

record TestGroupPicture
  Real offset=0 "Offset of output signal"
    annotation(Dialog(group="Group picture"));
  Real height=1 "Height of ramps"
    annotation(Dialog(group="Group picture"));
  Modelica.SIunits.Time startTime=0
    "Output = offset for time < startTime"
    annotation(Dialog(group="Group picture"));
  Modelica.SIunits.Time
    duration(min=Modelica.Constants.small) = 2
    "Duration of ramp"
    annotation(Dialog(group="Group picture"));

  annotation (Images(Parameters(group="Group picture",
    source="ramp.png")));
end TestGroupPicture;

```

The description texts and labels may contain HTML formatting tags if the text string is enclosed in <html> ... </html>. The example below shows some of the possibilities.



The corresponding record declaration is given below:

```

record TestHTML
  Real format
    "<html>It is possible to <b>format</b> description:
      x<sup>2</sup>, <font size=\"+2\">larger</font>,
      <font color=\">#ff0000\">color</font> </html>";
  Real alpha
    "<html>use other <font face=\"Courier New, Courier,
      monospace\">fonts </font> and <b>even</b>
      use greek characters: &alpha;</html>"
    annotation(Dialog(label="<html>&alpha;</html>"));
  Real picture
    "<html>and <b>also</b> include line <br> breaks and
      pictures
      <imgsrc=\"C:/Dymola/work/colorfill.png\">&nbsp; in
      the description</html>";
end TestHTML;

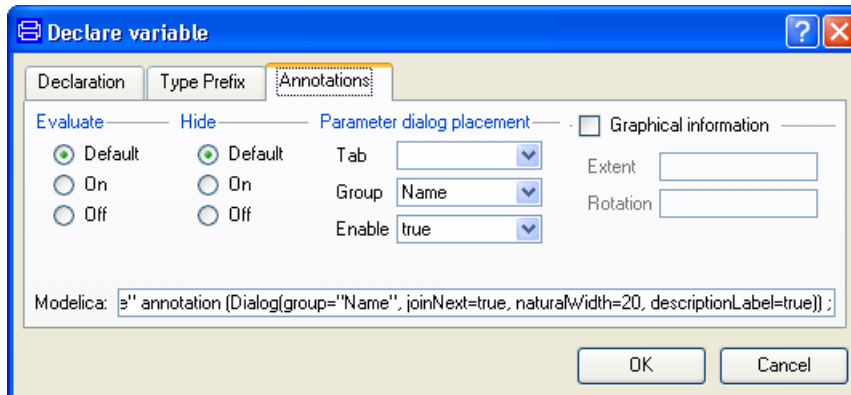
```

Greek symbols can, for example, be found at:

<http://www.htmlhelp.com/reference/html40/entities/symbols.html>

## Declare variable dialog

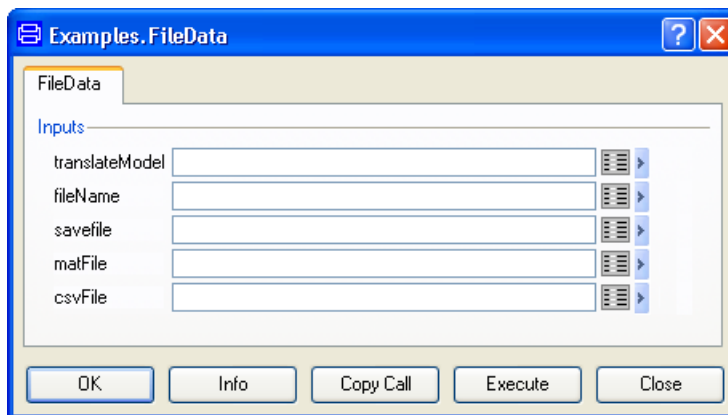
It is possible to introduce groups and tabs in the Annotations tab of the Declare variable editor for the fields firstName, middleInitial and lastName. The rest of the fields are put in the group Address:



This Declare variable editor is reached through Edit/Variables... or, after enabling non-graphical components in the Component browser, using the Parameters... from the context menu.

## Specialized GUI widgets

Declarations of variables can be annotated to provide a convenient user interface, for example to select models or to open files. These annotations are typically used to give inputs to functions or for creating records. The dialog is annotated with edit buttons:



Given the appropriate type definitions (see below), such a record is very easy to declare.

```
record FileData
  Examples.TranslatedModel translateModel;
  Examples.FileName fileName;
  Examples.FileNameOut savefile;
  Examples.MatFileName matFile;
  Examples.CsvFileName csvFile;
```

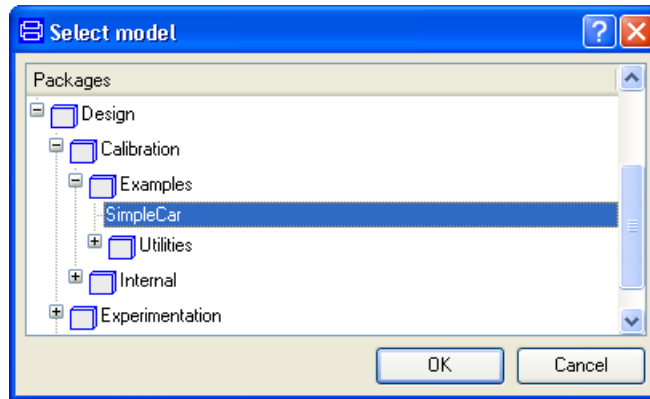


```
end FileData;
```

A string type that presents a dialog for selecting a model is declared as follows:

```
type TranslatedModel=String  
annotation(Dialog(translatedModel));
```

Pressing the edit button for such function argument displays this dialog:



The following declarations use annotations to display different kinds of file dialogs. The first one gets a filename for reading a file:

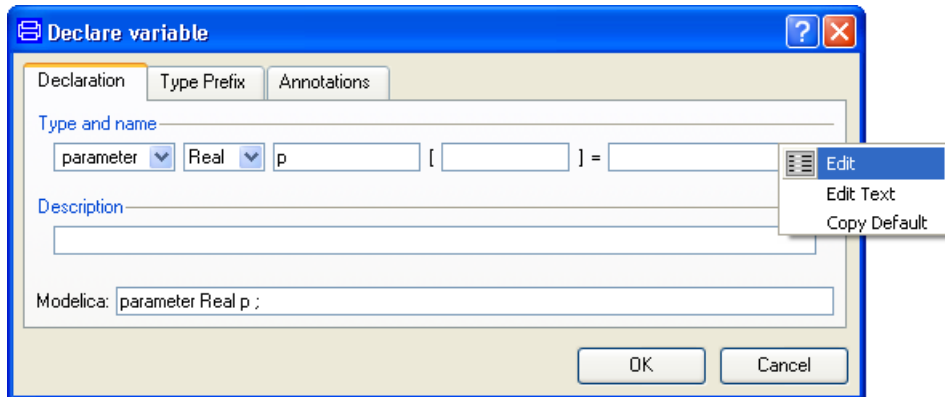
```
type FileName=String  
annotation(Dialog(loadSelector(filter="Matlab files  
(*.mat);;CSV files (*.csv)",caption="Open experiment data  
file")));
```

The second one get a filename for writing a file:

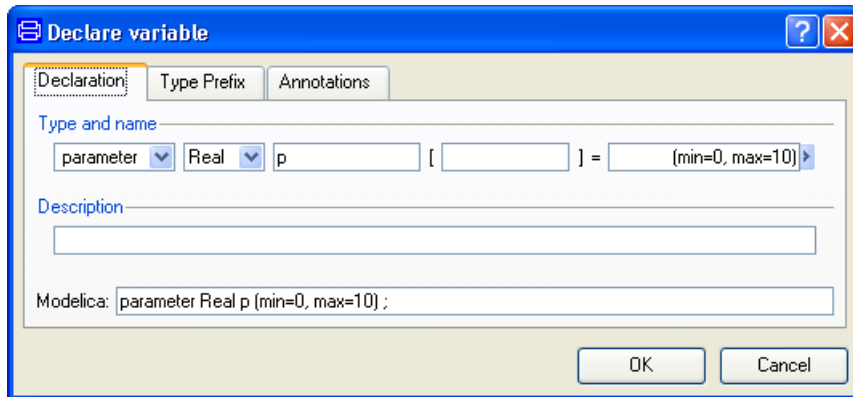
```
type FileNameOut = String  
annotation(Dialog(saveSelector(filter="Matlab files  
(*.mat);;CSV files (*.csv)",caption="Save experiment data  
file")));
```

## Checking of input data

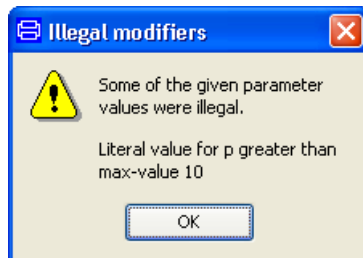
It is possible to declare parameters with minimum and maximum values, which are then checked by Dymola when the user sets a parameter value. The variable is declared with type and name as usual. Then press the edit button (right-arrow) and the end of the value field to present a menu.



Select Edit from the menu and enter the min and max values for the parameter. Assuming that we have specified the range to be [0, 10], the variable dialog shows



If we have a model with such a parameter and try to set a value outside of the valid range, Dymola will display an error message. The parameter dialog cannot be closed until the invalid modifier value has been corrected.



## Arrays of records

A simple address book can be created as an array of Person records as follows:

```
record Addresses
  Person4 persons[:];
end Addresses;
```

The corresponding dialog for such an array of records is:

The dialog box 'Records.Addresses' displays the 'Person4 Properties' view. The left tree browser shows the structure: Addresses > persons > persons[1], persons[2], persons[3]. The main area contains the following fields:

- Name: First name, Middle initial, Last name
- Address: Number, Street name, Postal code, City name

Buttons at the bottom: OK, Info, Copy Call, Execute, Close.

It is also possible to view all person records at the same time by selecting the array 'persons' in the left tree browser:

The dialog box 'Records.Addresses' displays the 'persons' view. The left tree browser shows the structure: Addresses > persons > persons[1], persons[2], persons[3]. The main area shows a table with 3 rows and 5 columns:

	First name	Middle	Last name	Number	Street name
1		"Joe"	"Smith"	123	"Main Street"
2					
3					

Buttons at the bottom: OK, Info, Copy Call, Execute, Close.



**Visualize 3D**



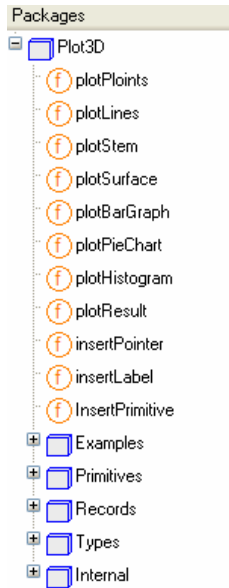
# Visualize 3D

---

## Introduction

Data visualization in 3D is an important way of representation, and it is adequate for understanding and comprehending model behavior. Dymola 6 includes a new 3D graphical tool: Visualize 3D.

Visualize 3D renders 3D scenes and has an associated Modelica package named Plot3D. This new package manipulates and sends the graphical data representation of the scene to Visualize 3D. This guide describes how to use Plot3D to obtain graphs and figures with the Visualize 3D tool in Dymola.

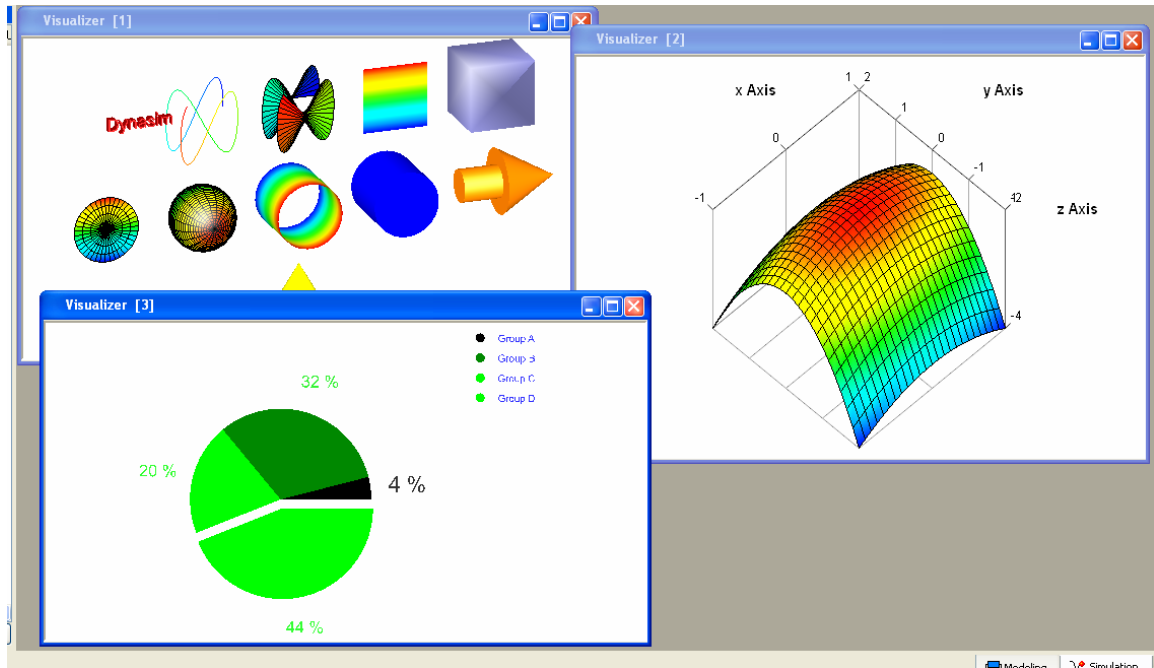


The main functions are at top level of the package: `plotPoints`, `plotLines`, `plotStem`, `plotSurface`, `plotBarGraph`, `plotPieChart`, `plotHistogram`, `plotResult`, `insertPointer`, `insertLabel` and `insertPrimitive`. We recommend strongly using these high-level functions instead of trying to use the low-level ones in `Plot3D.Internal`.

The subpackage `Plot3D.Primitives` contains the basic primitives preset. The subpackages `Records` and `Types` also contain information about the internal representation of a 3D scene. The subpackage `Examples` contains in its turn some of the examples presented here and we will refer to them later on.

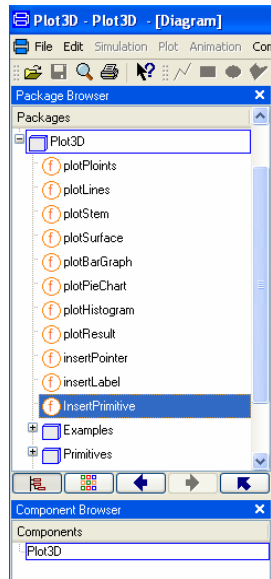
Visualize 3D supports several different types of plots and can be presented separately in their own windows if desired, all integrated in the Simulation tab.



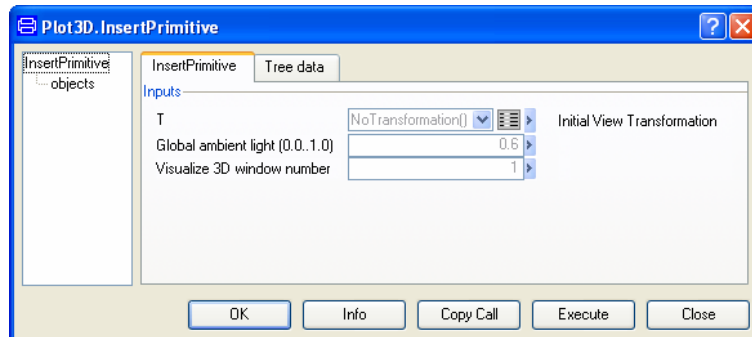


## Inserting and removing objects

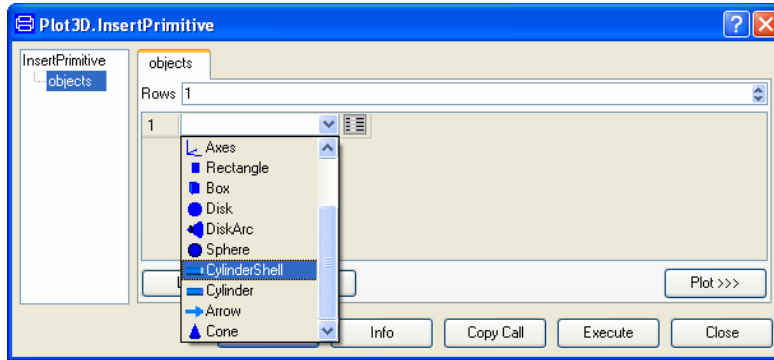
Visualize 3D has several basic primitives that can be combined to construct more complicated scenes. We will start by constructing a simple solid cylinder by combining a cylinder shell with two disks. We start by using the function `Plot3D.InsertPrimitive`



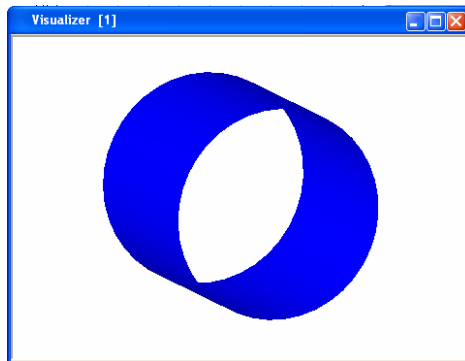
Click right on it and the following dialog will pop.



The first element we observe is the View transform matrix, the global ambient light and Visualize 3D window number. This number identifies the window we want to add some primitive to. We keep the default values just now and click on “objects” field in the tree. There we are to select the primitive forms to be added. Click now on the arrow of the combo box and scroll down until “CylinderShell”.



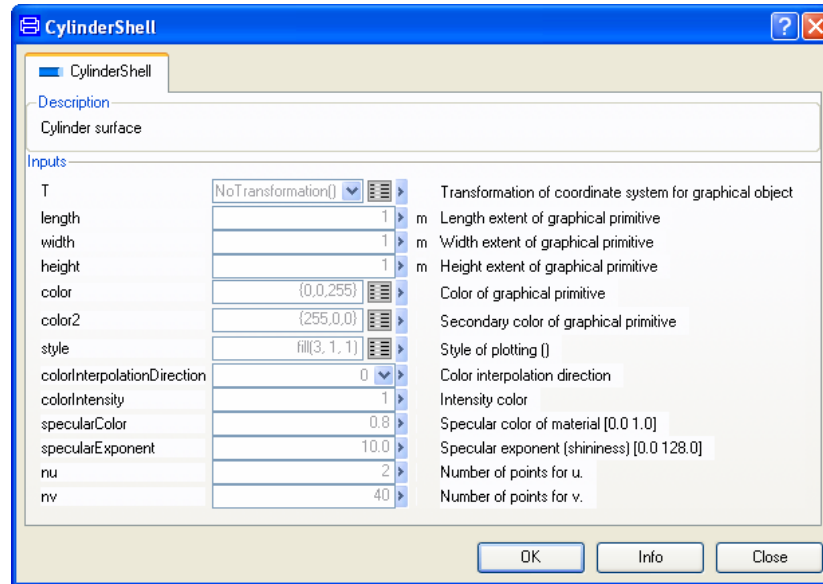
We have now selected a cylinder shell and we can plot it with default values. Press Execute. At first sight there is just an empty Visualize 3D window. Actually, we are looking at the shell with zero thickness along its main axis. To realize this, press the Ctrl key and move the mouse to rotate along the axes x and y. The figure below shows one possible view of the new created cylinder shell.



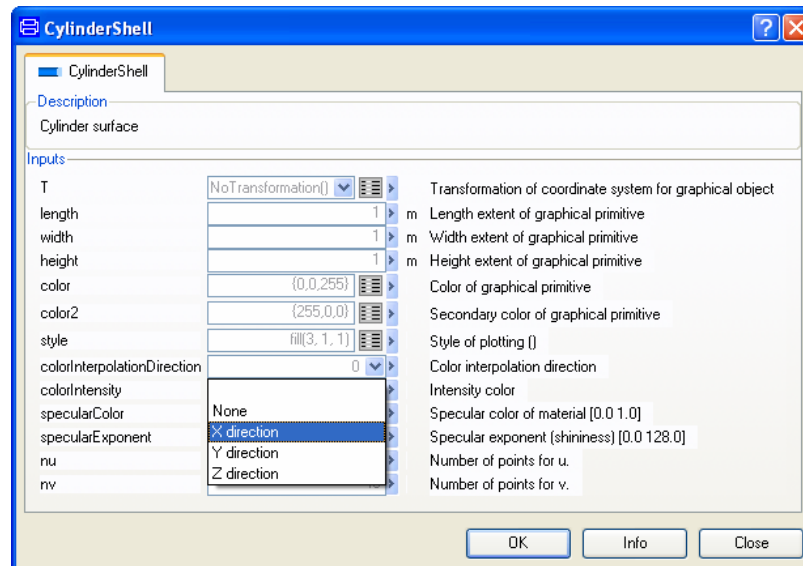
The operation of Visualize 3D can be summarized in the following table

<b>Operation</b>	<b>Meta key</b>	<b>Mouse move</b>	<b>Arrow keys</b>
Pan/Scroll	none	Left/Right/Up/Down	Left/Right/Up/Down
Rotate around x-axis	Ctrl	Up/Down	Up/Down
Rotate around y-axis	Ctrl	Left/Right	Left/Right
Rotate around z-axis	Ctrl+Shift	Clock-wise/ Counter clock-wise	Left/Right
Zoom in/out	Shift	Up/Down	Up/Down or Wheel

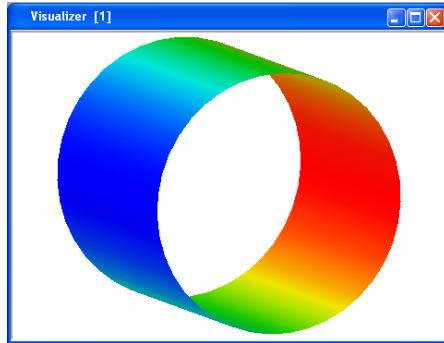
We can also perform other operations on the cylinder shell. Back to the dialog window, and clicking on the edit icon, we get the following dialog window.



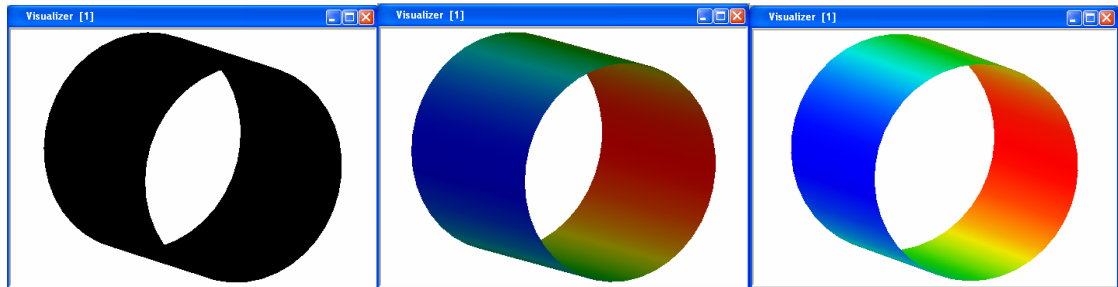
We observe different graphical properties of the cylinder shell primitive. We are now interested in a few: matrix T, length, color, style, colorInterpolationDirection and colorIntensity. Change colorInterpolationDirection to “x direction” and press execute once more.



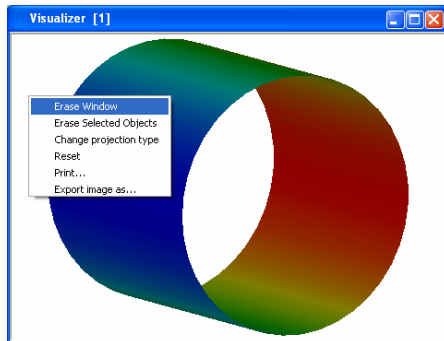
Press control-key and move the mouse. The change is that Visualize3D interpolates the color using the range of the x coordinate of the primitive.



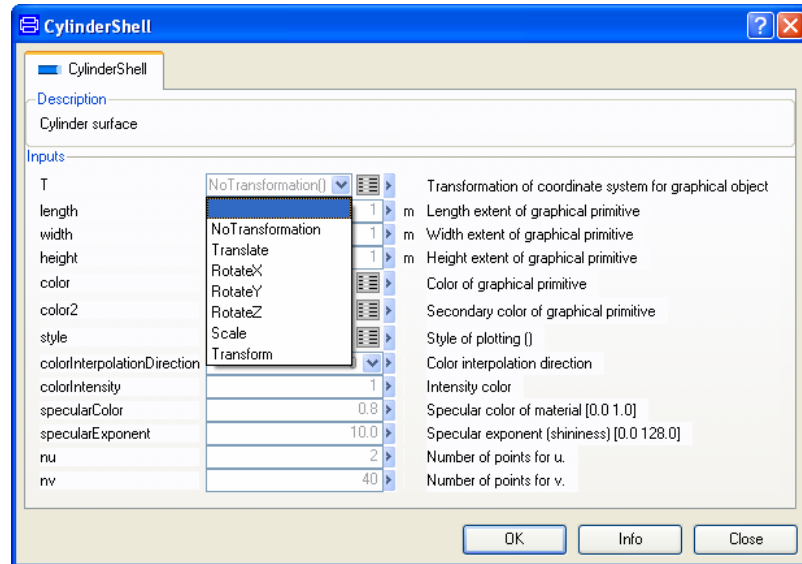
Changing the colorIntensity parameter it is possible to set the brightness of the color scheme applied. This factor is to be in the interval  $[0,1]$ . Below we find depicted the cylinder shell for intensityColor=0, 0.5 and 1.



Remember that we are adding primitives; this means that if the intention is to change and paint again, the Visualize 3D window has to be erased. This can be done by right clicking on the window, the context menu opens and the selecting "Erasing window", as below. This operation will clean the window object list.

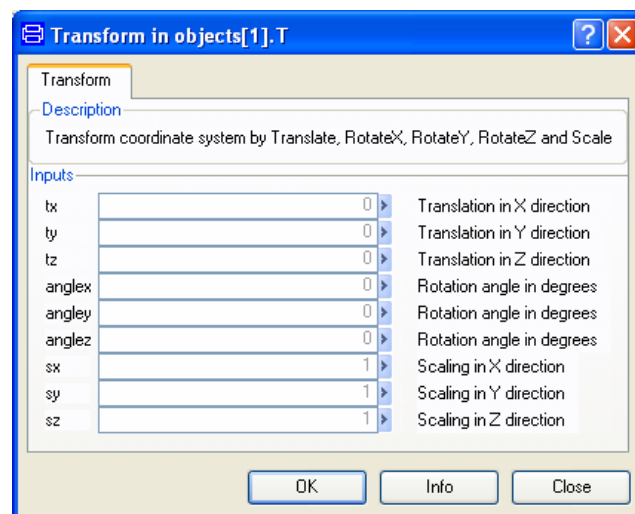


The matrix  $T$  is used to perform transforms on just the associated object. Operations like translation, scaling and rotation of the body respect to the global coordinate system are described with this  $T$  matrix. These transforms are independent of the global view, and are used to construct the 3D scene. Clicking on the combo box arrow shows the predefined possibilities.



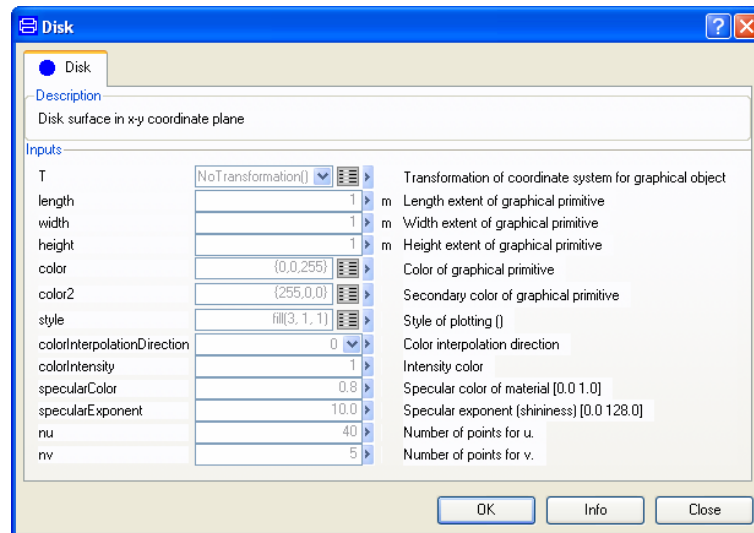
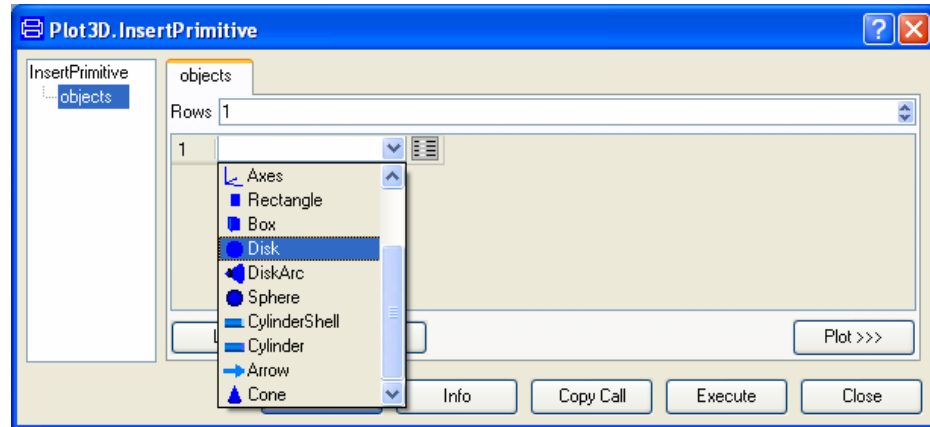
We can select NoTransform, Translate, RotateX, RotateY and RotateZ. They are fairly described by their names. The most general of the operations is Transform, and involves a combination of all the others.

The dialog window for “Transform” is the following

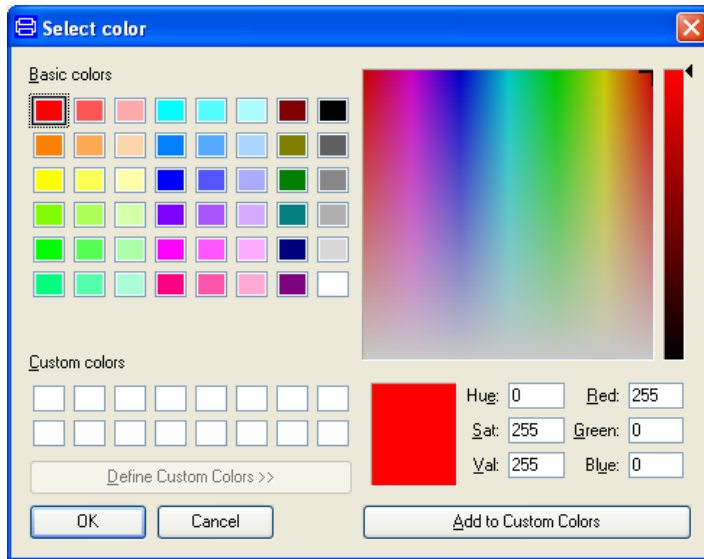


Here we can describe what we want to do with the object. The order is important, since all these operations are not commutative, for instance, it is not the same to rotate and translate as to translate and then rotate. The order preset is scale first, rotate around Z, rotate around Y, rotate around X and then translate.

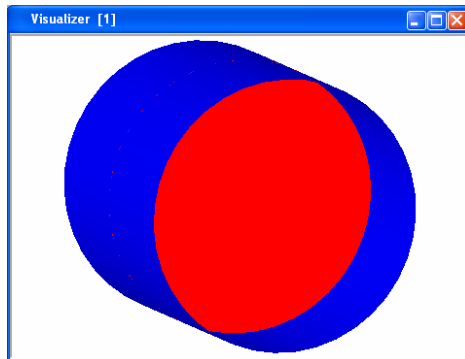
Let us now add the top and bottom of the cylinder. Again in the dialog window, we change the “CylinderShell” primitive to the “Disk” primitive.



Change the color of the “Disk” by pressing the Edit icon, and then the Edit icon of the field “color”. We choose in this case the red color to get a good contrast.

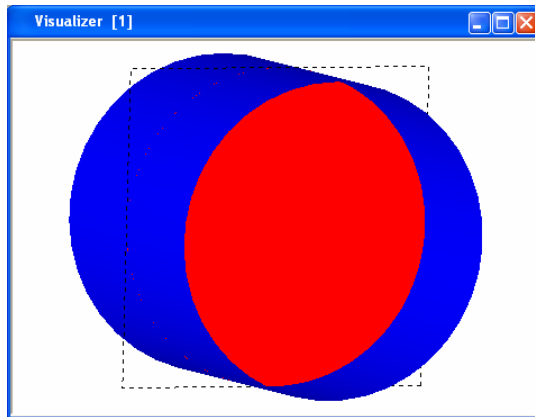


Press Execute and rotate once more using control-Key and moving the mouse. We observe the following result.

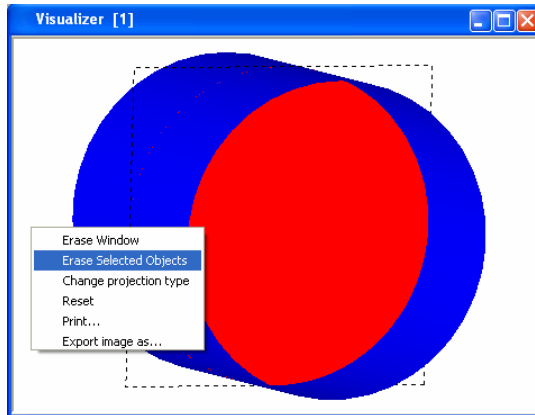


The disk is in the middle by default. We want to place the disks on top and bottom of the cylinder shell. We will therefore erase the disk and place it correctly using the translate transform. To erase an object, we have to select it first by clicking on it pressing Alt-key. The selected object is delimited by a dotted box.

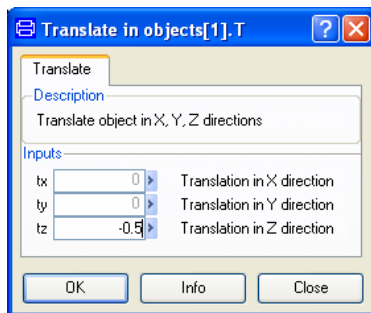




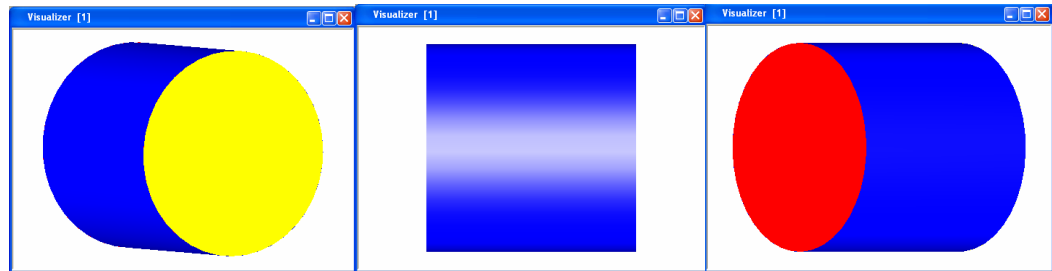
Now we select from the context menu “Erase Selected Objects”, and the disk is erased from the actual view.



To close the cylinder shell, we have then to set the bottom disk at the point  $(0, 0, -0.5)$  and the top disk at  $(0, 0, 0.5)$ , since the cylinder has length 1. Press again on the Edit icon of “Disk”, then change “T” to “Translate” and set “tz” to -0.5.



Press Execute. Then change `tz` to 0.5 and press Execute again. We obtain now a closed cylinder as below. We changed the color of top disk to yellow to show clearly the three components.

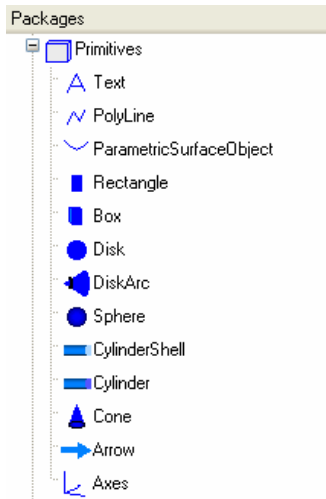


Here we see top, side and bottom of the newly created cylinder. The primitive `Plot3D.Primitives.Cylinder` is constructed with this technique, encapsulating all necessary steps to get a uniform color, size and other properties.

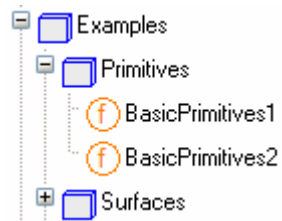
---

## Basic primitives

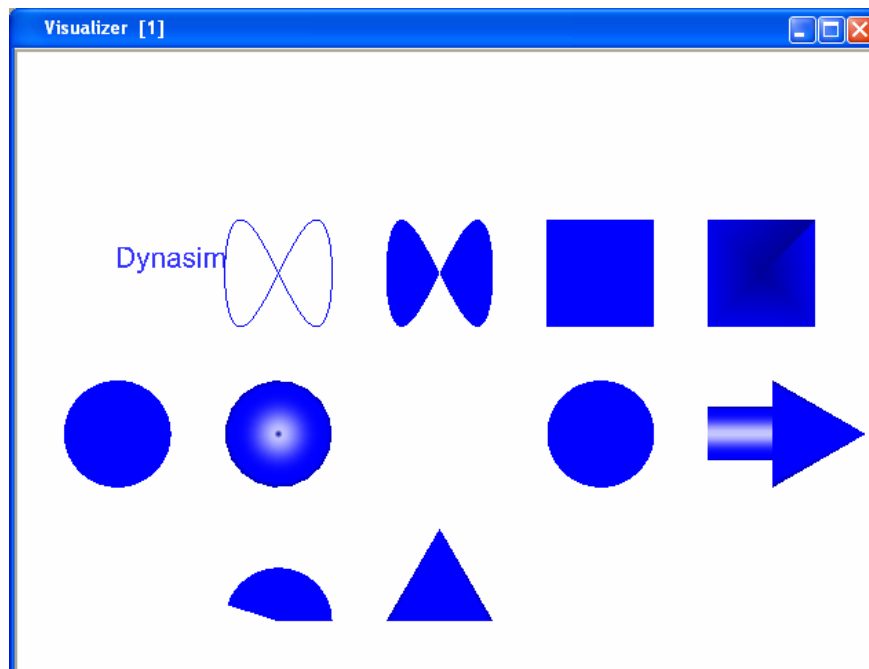
The basic predefined primitives included in `Plot3D` are presented in the figure below.



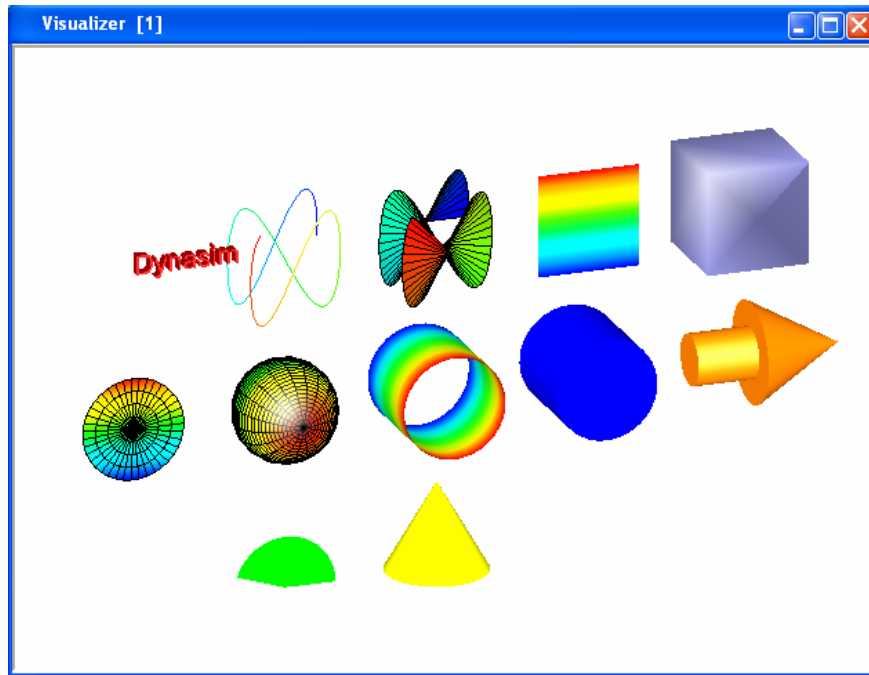
In the package `Plot3D.Examples.Primitives`, the functions `BasicPrimitives1` and `BasicPrimitives2` produce 3D scenes with the primitives.



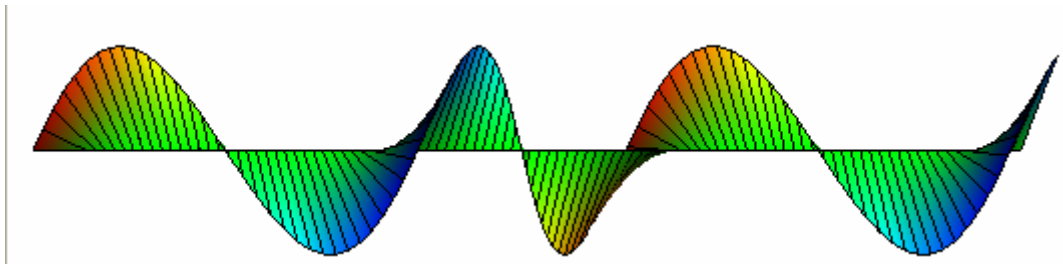
Click right on BasicPrimitives1 and then “Call Function ...”. Then, press Execute. The resulting 3D scene is the following.



Notice that the cylinder shell has no thickness. BasicPrimitives2 is another example showing some of the features of Visualize 3D. Repeat for BasicPrimitives2 as before to get the following 3D scene.



In particular, the third curve at the top line is a Lissajous curve, typically used in electronics and electrotechnique to find frequency and phase of a unknown sine curve, using a known one as reference. If we observe now this curve along its z-axis, the result is the following.



The dialog windows of all primitives are very similar. Each one of them have inherent fields, for instance, `Plot3D.Primitives.Text` has a `String` field called `textString`. In this case, the label we want to render. The primitive `Plot3D.Primitives.Axes` is a very particular one, since it produces a reference coordinate system. We will use it in the next section.

---

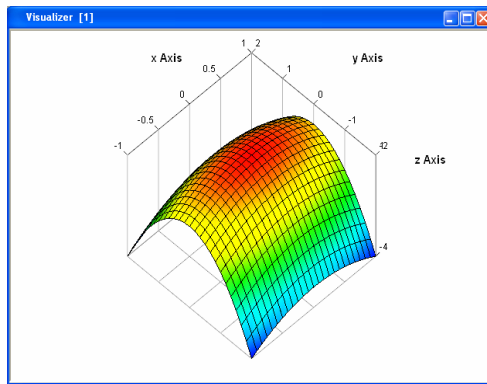
## Surface Plots

Other important feature of Plot3D is the easy user interface and the inclusion of high level help functions that will render surfaces, contour lines, water fall plots and bar graphs from matrix data. In the following, the notation we use is as follows

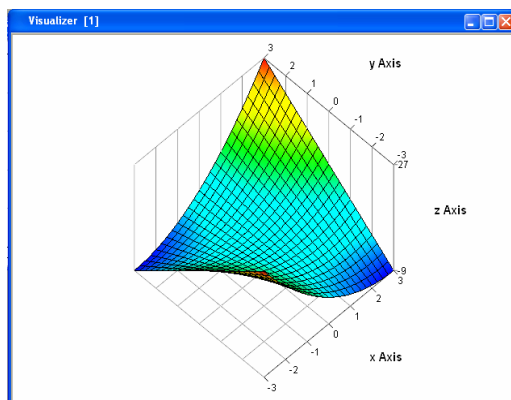
1. The matrices  $x,y,z$  describing a parametric surface of the form  $x=f(t,s),y=g(t,s),z=h(t,s)$ .
2. The matrices  $n_x,n_y,n_z$  describing a vector field  $(n_x,n_y,n_z)$  on the point  $(x,y,z)$ .

We will consider three test cases with their respective plots:

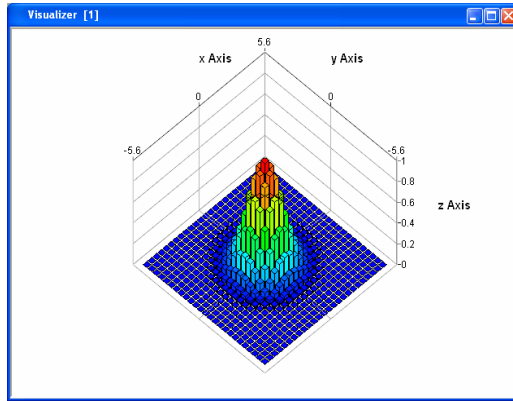
- parabolic function  $z=1-x^2-y^2$  on the interval  $[-1,1] \times [-2,2]$



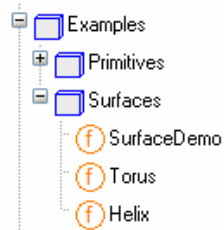
- hyperbolic function  $z = x^2 + 2xy$  on the interval  $[-3,3] \times [-3,3]$



- bivariate non-normalized Gaussian distribution  $z = e^{-\frac{x^2+y^2}{2}}$  on the interval  $[-5, 5] \times [-5, 5]$ .



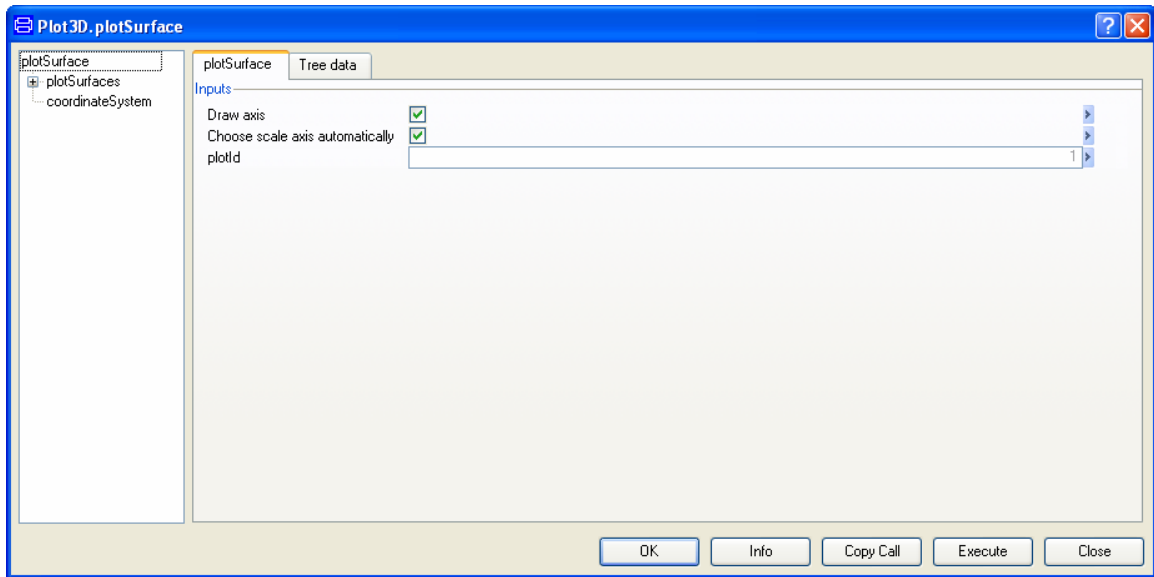
The function `Plot3D.Examples.Surfaces.surfaceDemo` runs all test cases. We will consider two of them here and just show the rest. The functions `Torus` and `Helix` are further examples of closed surfaces in different styles.



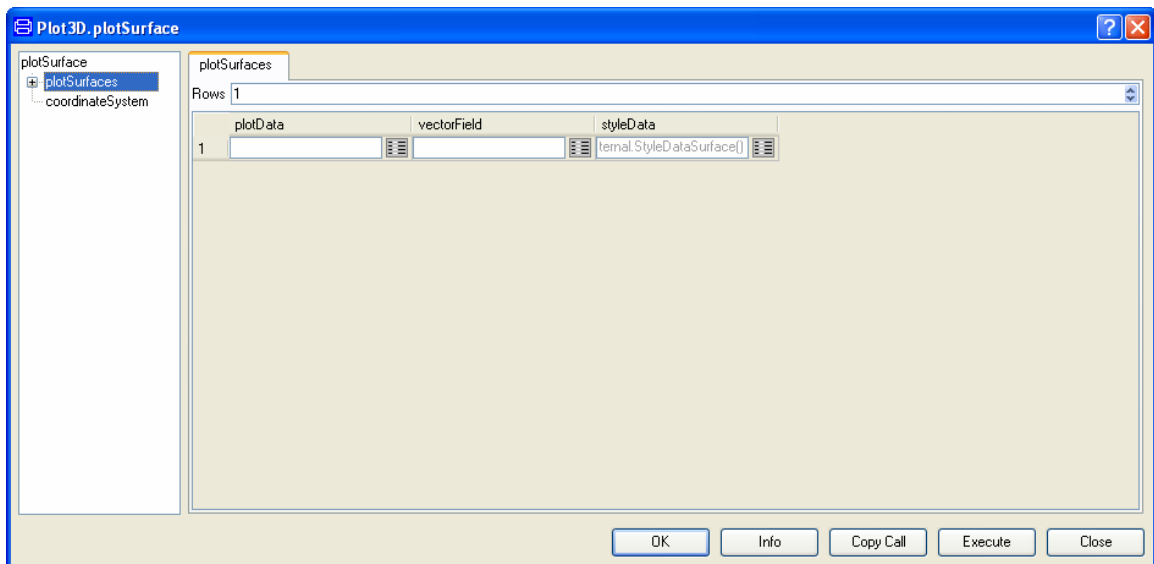
Let us plot the first test function. Execute the following command in the command window to create the matrices `x,y,z,nx,ny` and `nz`

```
(x,y,z,nx,ny,nz):=Plot3D.Utilities.SurfaceTest1(25);
```

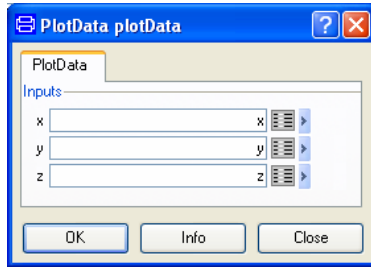
Now, click right on the function `Plot3D.plotSurface`. Click “Call Function ...” item. The following dialog window appears



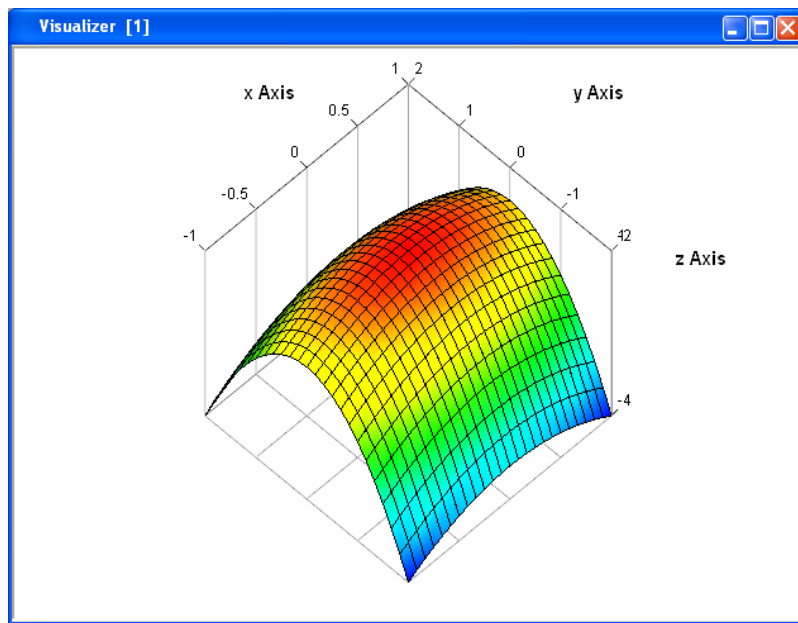
In this dialog we can set whether we want the axes automatically constructed or not. Furthermore, we can indicate a Visualize window number in “plotId”. Click now on “plotSurfaces”. The following dialog pops



To set the parametric surface matrices x,y and z we click on the edit icon of “plotData”. The following window pops

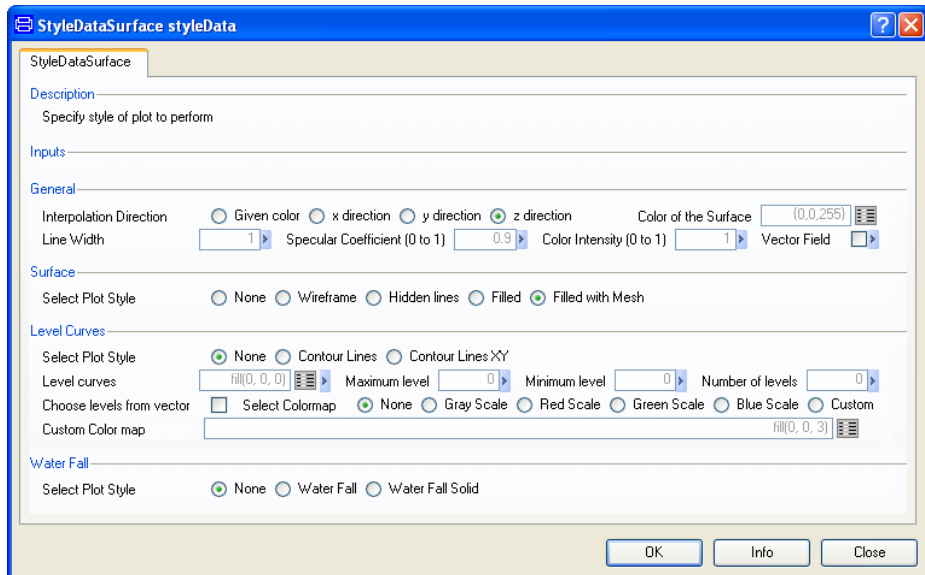


The only needed to plot is to fill in the matrices. We write x,y and z in their corresponding places and click OK. Then, back in the main Dialog, we click on Execute and obtain the following plot.

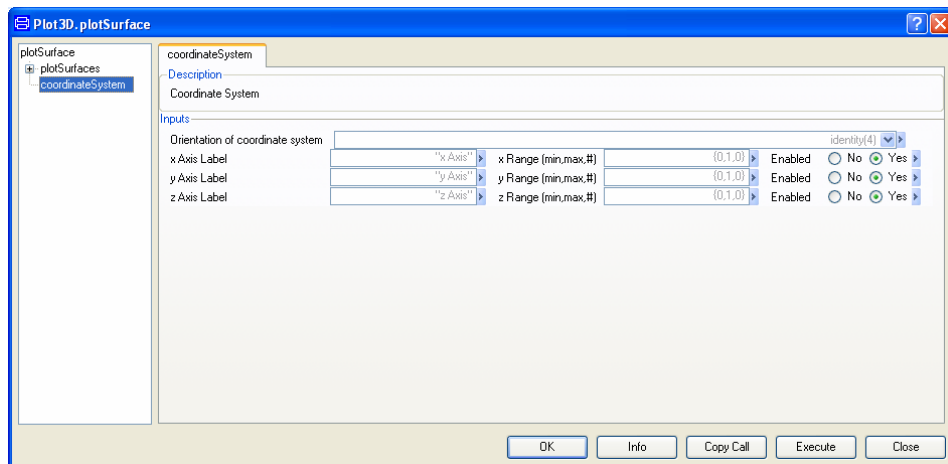


This is the default plot style (Filled with Mesh), and with default names for the X, Y and Z axes. If we want to change the style of the plots, the data has to be filled in the “styleData” field, using its Edit icon. The dialog follows





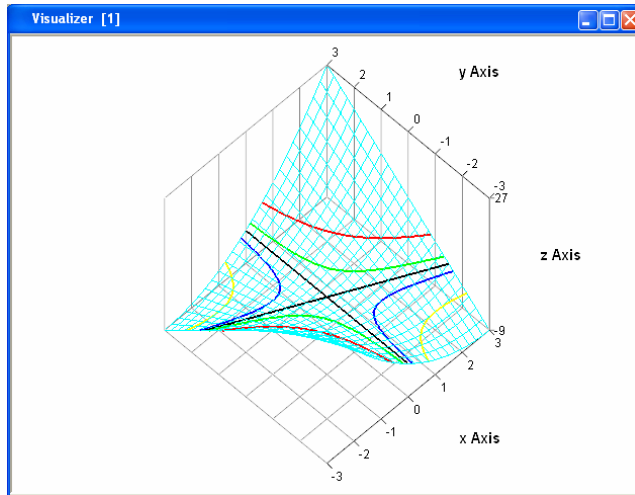
We observe the different alternatives. We can combine independently four groups of data: Surface (Wireframe, Hidden Lines, Filled and Filled with Mesh), Level Curves (Contour Lines and Contour lines XY), Water Fall (Normal and Solid disks) and Vector field (check box in General group). To change the axes properties, we click on “coordinateSystem” on the tree. The following dialog window pops



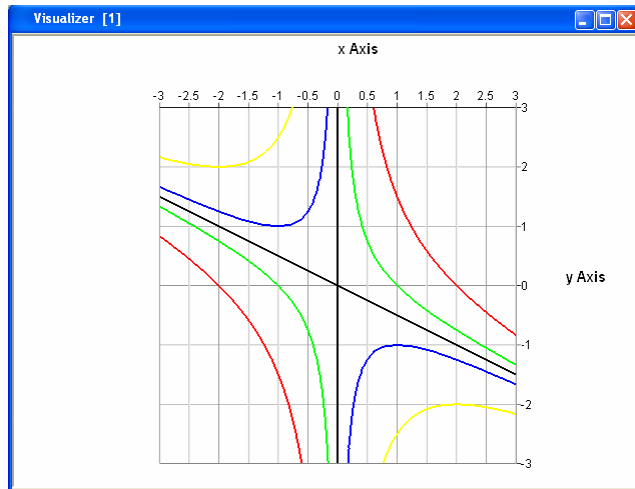
We observe here the fields “Axis label”, “Range” and “Enabled” for X,Y and Z axis.

Using the functions plotPoints, plotLines, plotStem, plotSurface and plotBarGraph follows the same lines, with particular variations.

We want to emphasize the combination of contour plots with Wireframe. This combination is particularly interesting to show interesting features of a function. For instance, the contour lines of the hyperbolic function  $z = x^2 + 2xy$  for  $z = 0$  yield two straight lines and constitute a degenerated transition case between the hyperbolic lines in two quadrants (above of  $z = 0$ ) and hyperbolic lines in the other two quadrants (below of  $z = 0$ ). The resulting plot follows

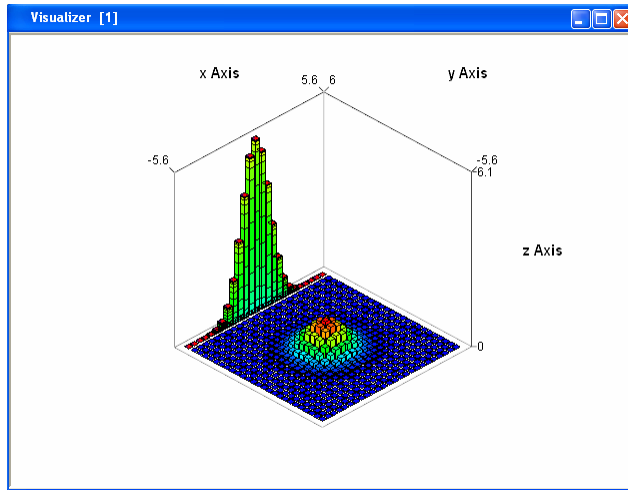


Or easily viewed, projected on the XY plane without Z axis ticks

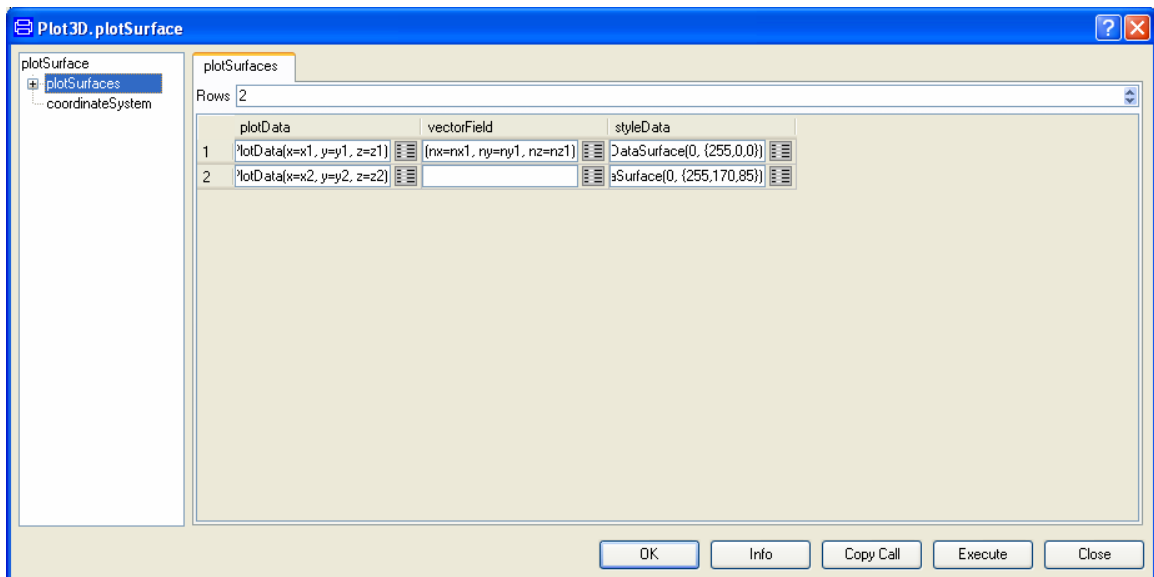


The black lines are the asymptotes of both sets of hyperbolic contour lines (red means  $z = 4$ , green  $z = 1$ , blue  $z = -1$  and yellow  $z = 4$ ).

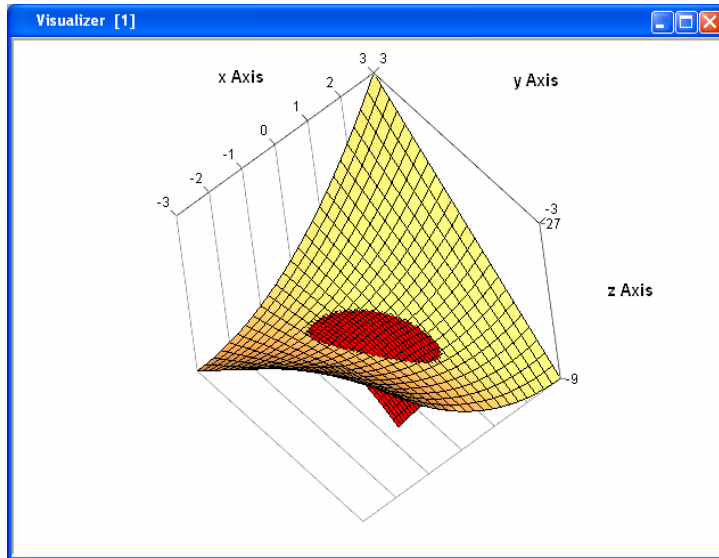
Other combinations can be useful to explain features too. For instance, when considering the Gaussian bivariate probability distribution. If we integrate one of the variables (let us integrate the  $y$  variable in this case) the resulting univariate function is also a Gaussian distributed variable. Combining the “Rectangle on Top” with the “Rectangle” plots of Plot3D.plotBarGraph function we can illustrate just that. The result follows.



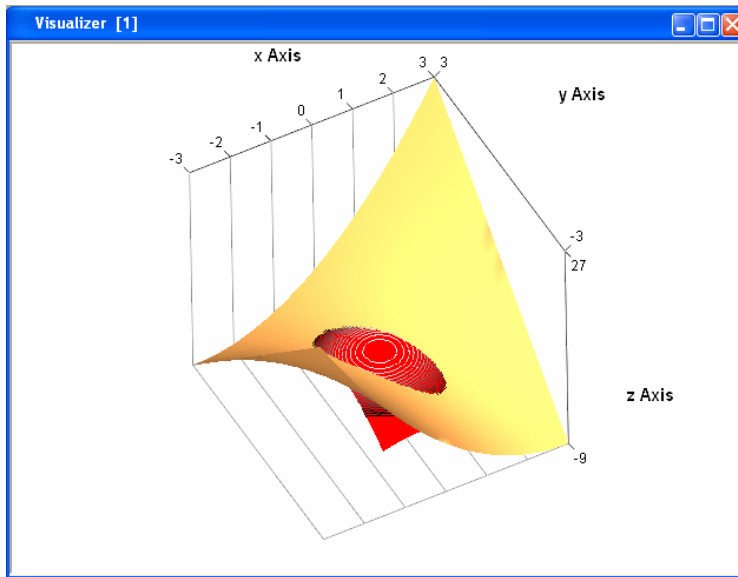
The intersection of surfaces using Plot3D.plotSurface is also possible. The only thing we have to do is to increment the number of elements to plot in the dialog. We can also set color and style to identify easily the functions and delimit the intersection area.



One possibility is to have different colors for the surfaces. The intersection of the parabolic surface and the hyperbolic surface with different colors will look as follows.

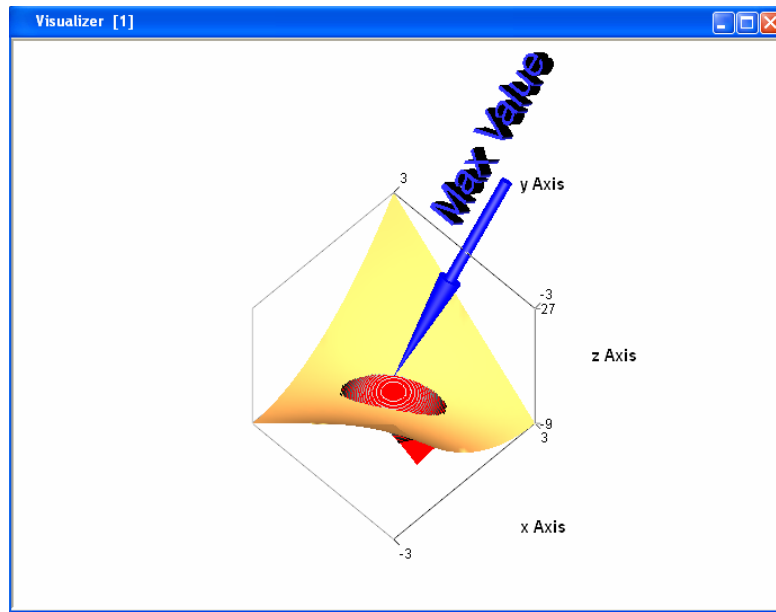


Combining different styles, we can obtain the following graph.

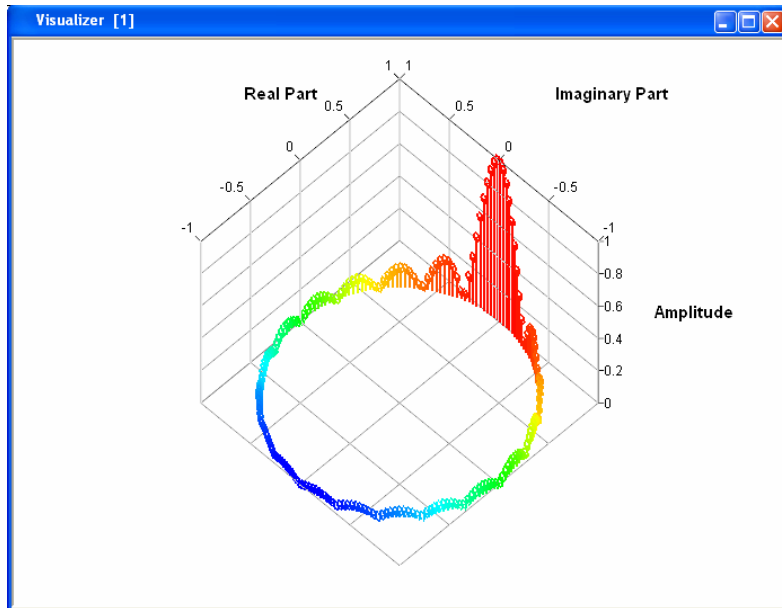


The contour lines in the parabolic surface (red) are used to illustrate that the intersection does not happen on a plane. The black color corresponds to the level  $z=-1.5$  and the white color corresponds to the level  $z=1$ .

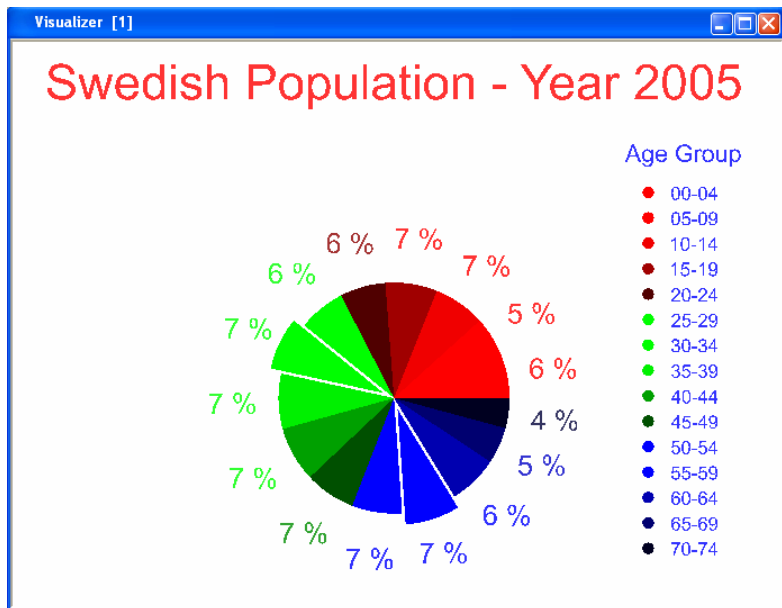
We can add a pointer to show where the maximum of the red surface occurs. Using the function `Plot3D.insertPointer` directly on the last image we can add text and an arrow. The resulting figure follows.



To plot discrete data, the alternative is to use `plotStem` function. This function considers each point by itself and puts a triangle, square or circle at the data point and adds a line from the point to the plane XY. We plot here down as example the amplitude or absolute value of the discrete Fourier Transform of a pulse. Putting these values in the unit circle of the complex plane relates the Z-transform to the discrete Fourier transform. The color is interpolated in the x direction.



Other alternative that Plot3D provides is to make pie charts. Statistiska Centralbyrån (Central statistics office) in Sweden reports the following population distribution by age in 2005. Two age groups are separated (30-34 and 55-59) to distinguish them.



# **Model Experimentation**





# Model Experimentation

---

## Introduction

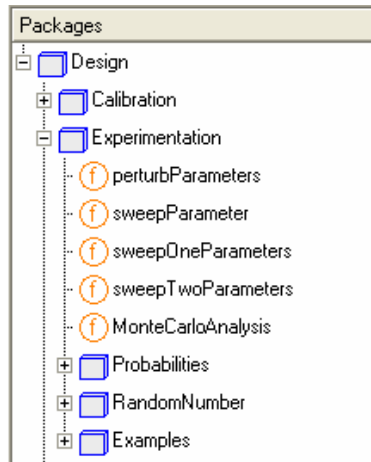
Dymola provides the Experimentation package as a feature of the Design package. The main purpose of this package is to allow the user to vary parameters of the system to get an intuitive knowledge of the behavior of the model. Some of the functionalities of this package are related to other functions of the Calibration package.

The main difference is that those are coupled to the calibration setup, while the functions in Experimentation are independent and can be used to illustrate phenomena of the system. One of the functionalities of Experimentation package is essentially different: Monte Carlo simulation.

---

## Varying parameters of a model

The Experimentation package provides several ways of analyzing the behavior of a model. The main functions are `perturbParameter`, `sweepParameter`, `sweepOneParameter`, `sweepTwoParameters` and `MonteCarloAnalysis`.

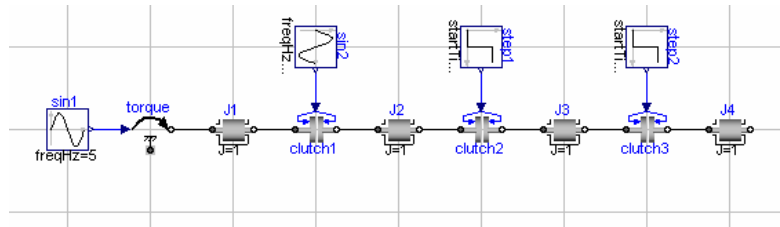


The functions `perturbParameters`, `sweepParameter` and `sweepTwoParameters` have corresponding in the Calibration package and can be used for more general parameter studies. The main difference in this package compared to Calibration is that the resulting output is the response of the model. We give a short overview of these functions now.

The functions `sweepOneParameter` and `MonteCarlo Analysis` complete the set, giving the possibility of plotting the response at the end of the integration interval and random draws of numbers for the parameters in Monte Carlo simulations. The example studied for this package is the model `Design.Experimentation.CoupledClutches`. This example is an extension of `Modelica.Mechanics.Rotational.CoupledClutches`.

### Case Study: CoupledClutches model

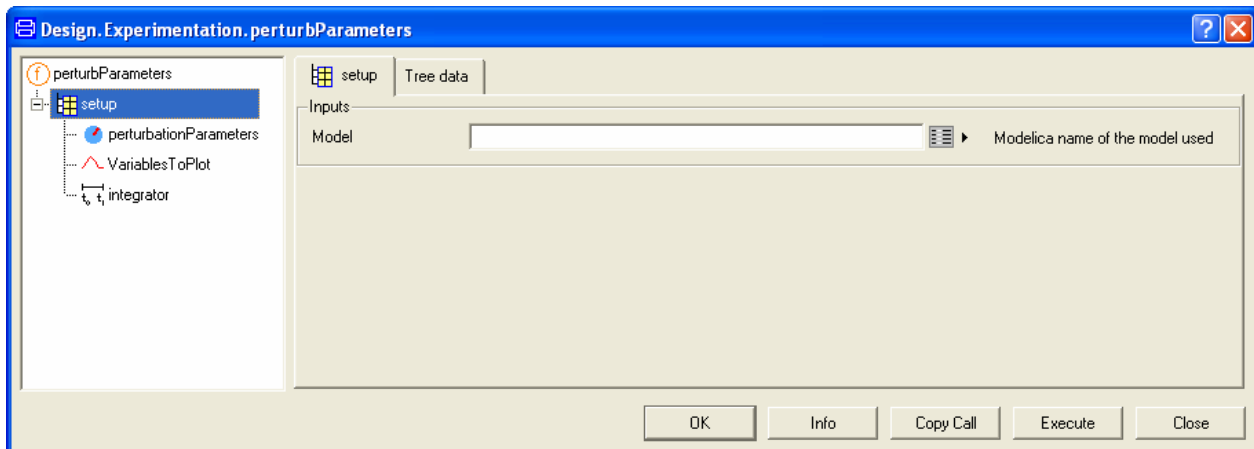
The model `CoupledClutches` is composed by four rotating inertias  $J_1$ ,  $J_2$ ,  $J_3$  and  $J_4$  coupled by three clutches that make them interact. The diagram looks as follows.



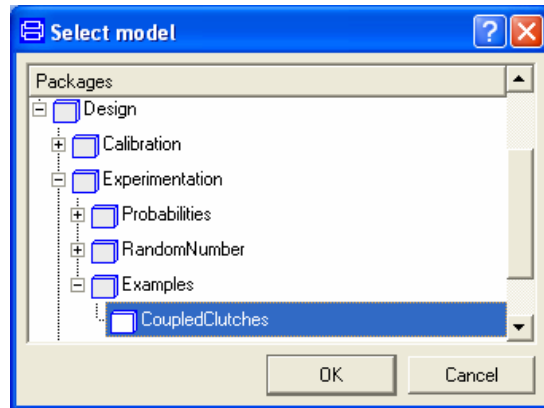
The parameters of the model to explore are the inertia values J1.J, J2.J, J3.J and J4.J. The observed variables are the rotational speeds J1.w, J2.w, J3.w and J4.w. The setups of the functions are very similar and their description will therefore be brief.

## Perturb parameters

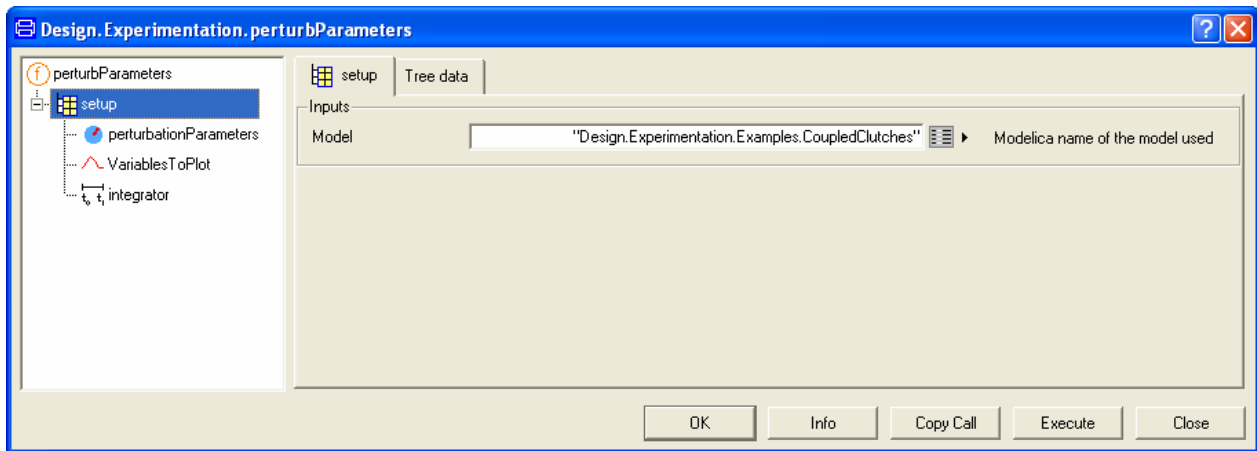
Let us check the behavior of the model if we perturb the nominal values of the parameters. Select the function `Design.Experimentation.perturbParameter` in the package browser. Click right mouse button and select “Call Function ...”. The following menu pops



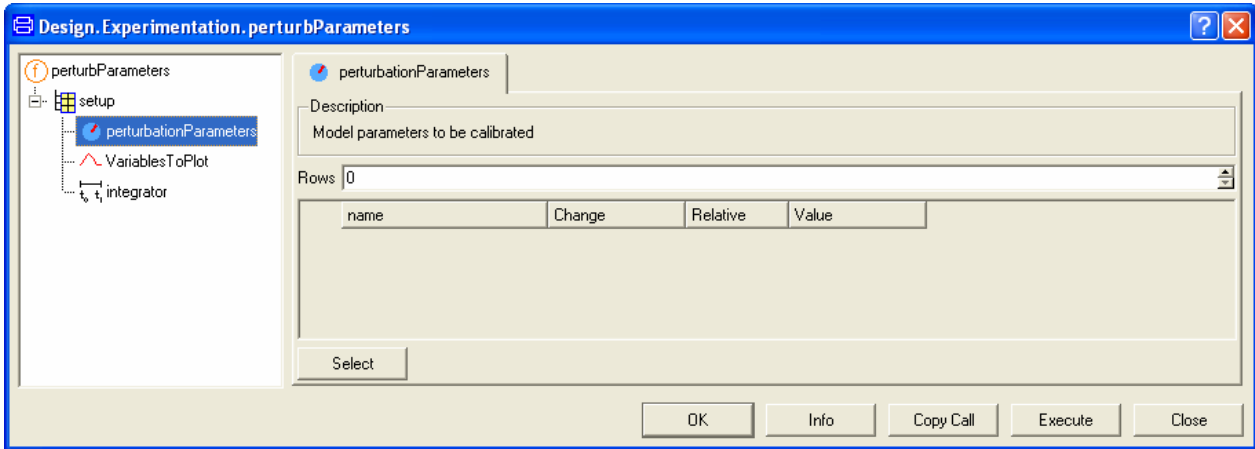
Now, to specify the model to use, click on Edit icon to the left of the input field. A package browser pops up. Use it to select the model.



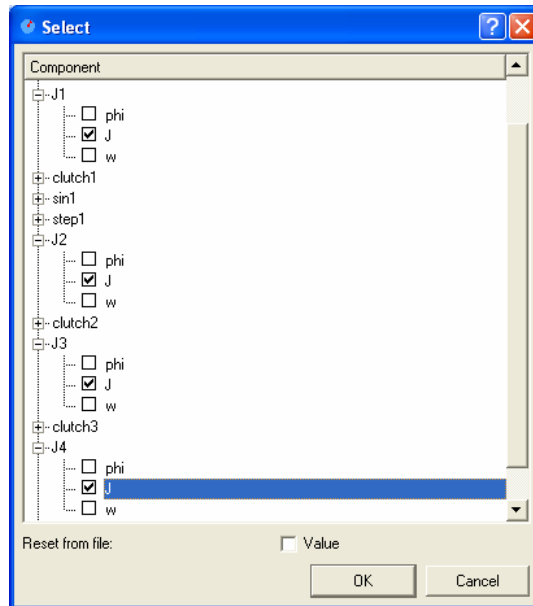
Click OK. The model is now translated in order to gather information needed to build browsers and selectors to support the remaining setting up. If Dymola already has a translated model, then this model appears as the default model.



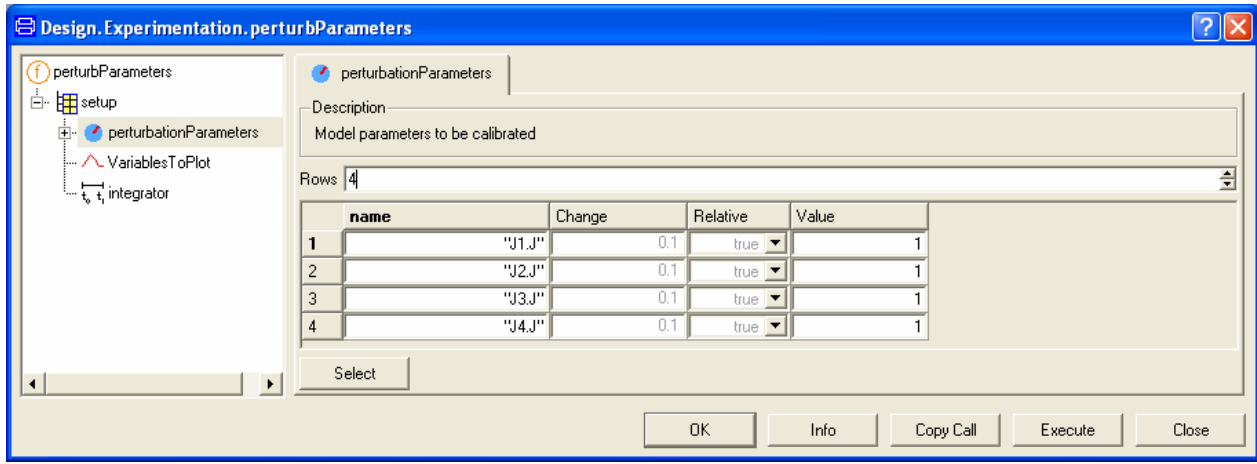
The next task is to select the parameters to perturb and the variables to observe and plot. Click on `perturbationParameters`, and then on the “Select” button.



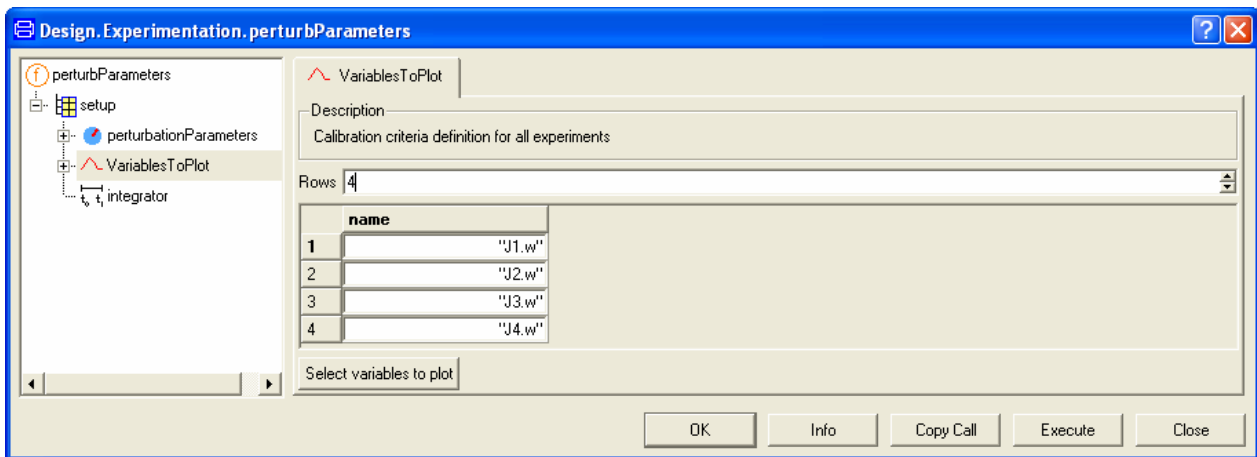
The following browser pops and the parameters J1.J, J2.J, J3.J and J4.J can be selected as perturbation parameters. Their nominal value is 1 for all of them. The perturbation is by default 10 percent.



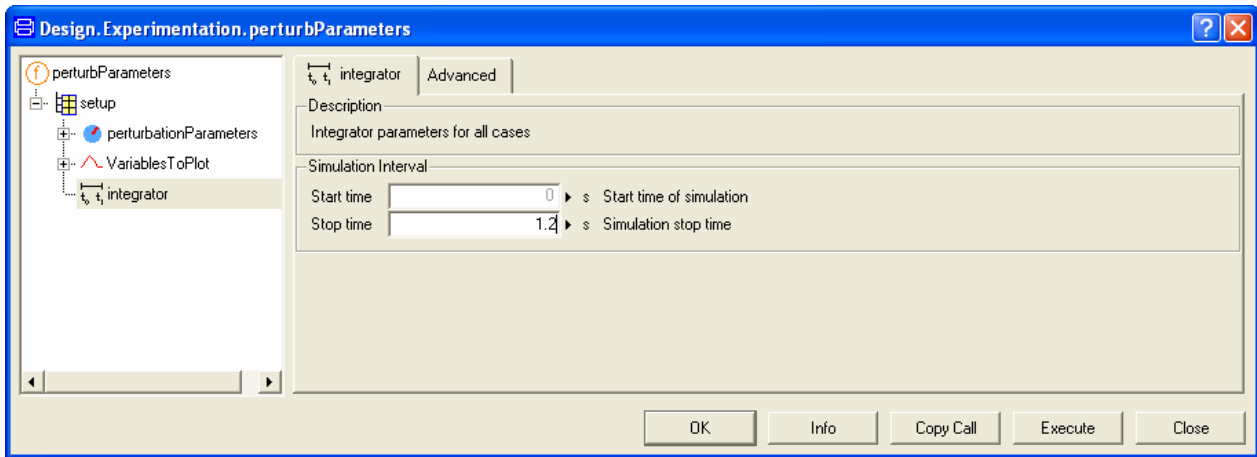
We can select a percent change of absolute change if we like. In the setup presented, the parameters are perturbed 10 percent from their nominal value.



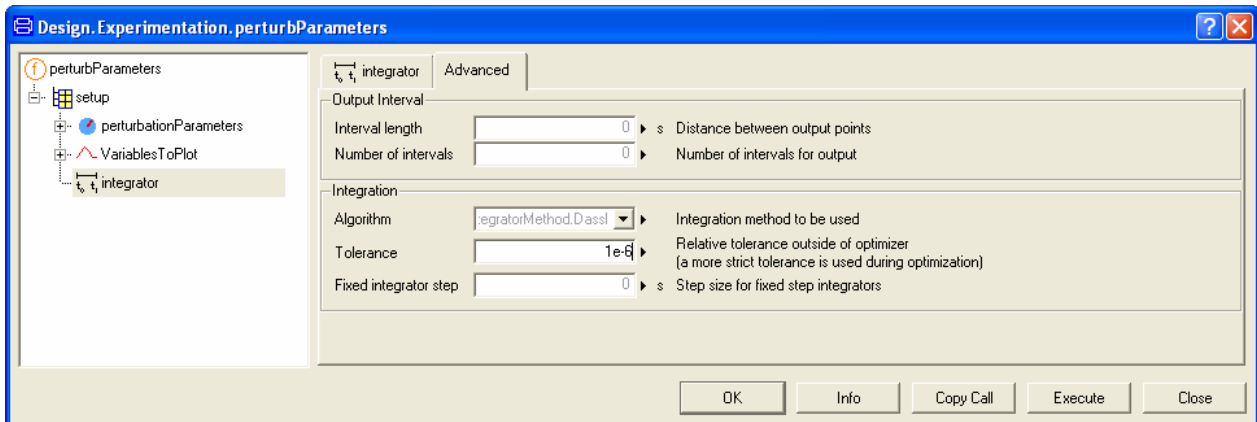
Now, let us select the variables to plot. Click on VariablesToPlot and then clicking on “Select variables to plot” button we get a variable browser where the selection of J1.w, J2.w, J3.w and J4.w is possible. The resulting menu looks as following.



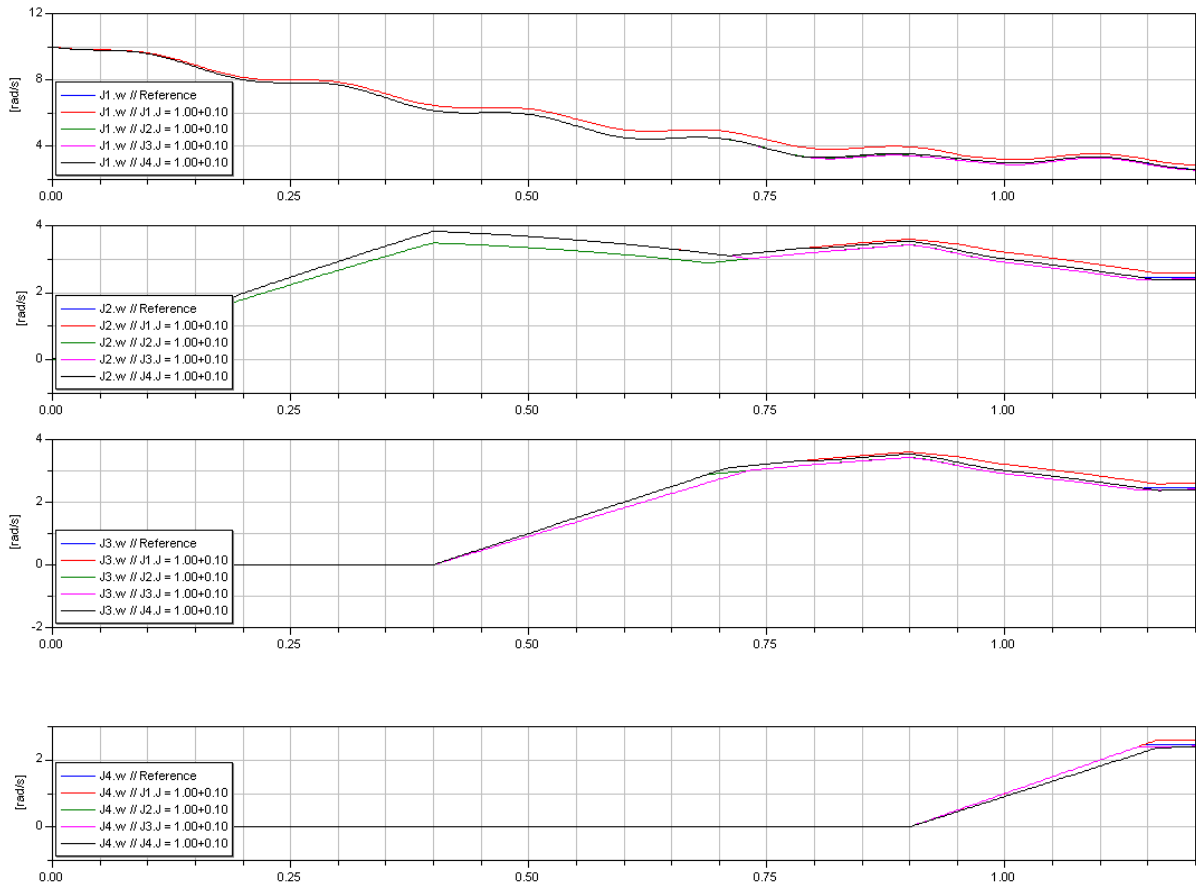
Finally, the setup for the integrator is to be done. Click on “integrator” and set as stop time 1.2



and the default tolerance for the integrator lowered to 1e-6.



Now we can run the command. Click on “Execute”. After the simulations, and moving the legends to the appropriate place, we get the following sequence of images.



The plots show the variation of every variable when varying the parameters J1.J, J2.J, J3.J and J4.J 10 percent, one at a time. We observe, for instance, in the first plot that only the variation of J1.J affects the response on J1.w.

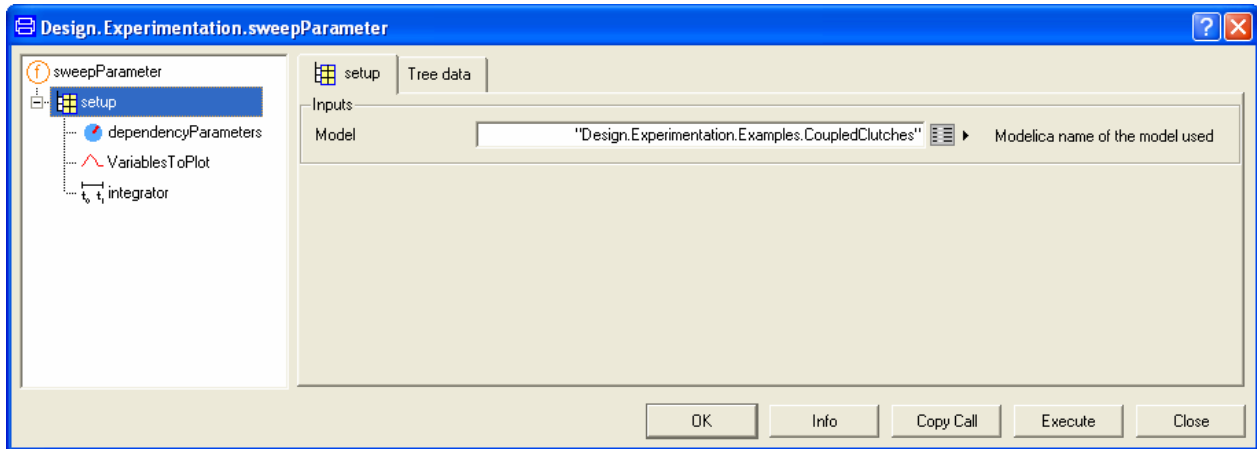
## Sweep One Parameter – two variants

The phenomenon described before can be observed in another fashion. We can sweep one parameter and observe the result along the whole interval from 0 to 1.2, or just at the final time of 1.2 seconds. These variants are implemented in two functions `sweepParameter` and `SweepOneParameter`.

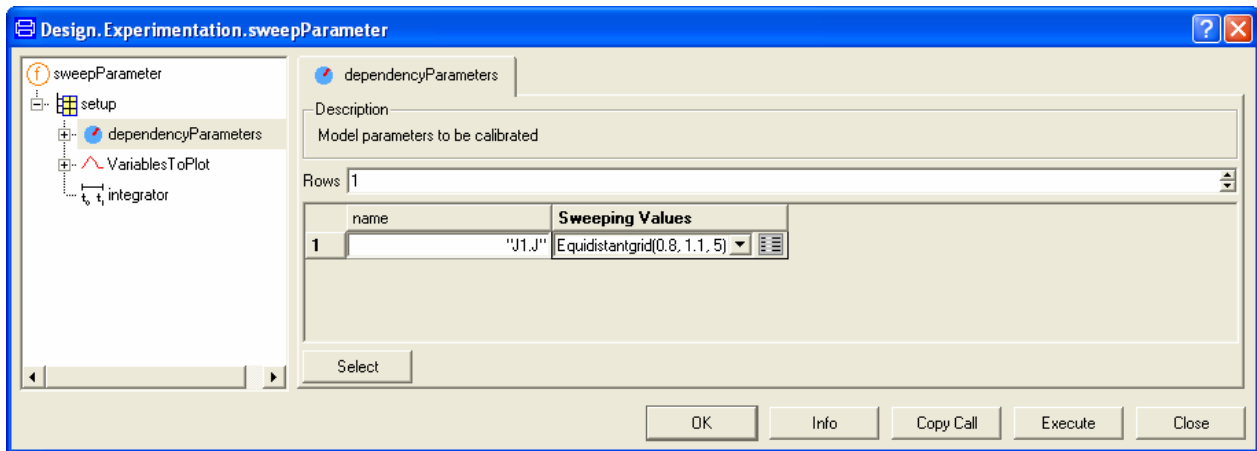
`sweepParameter`



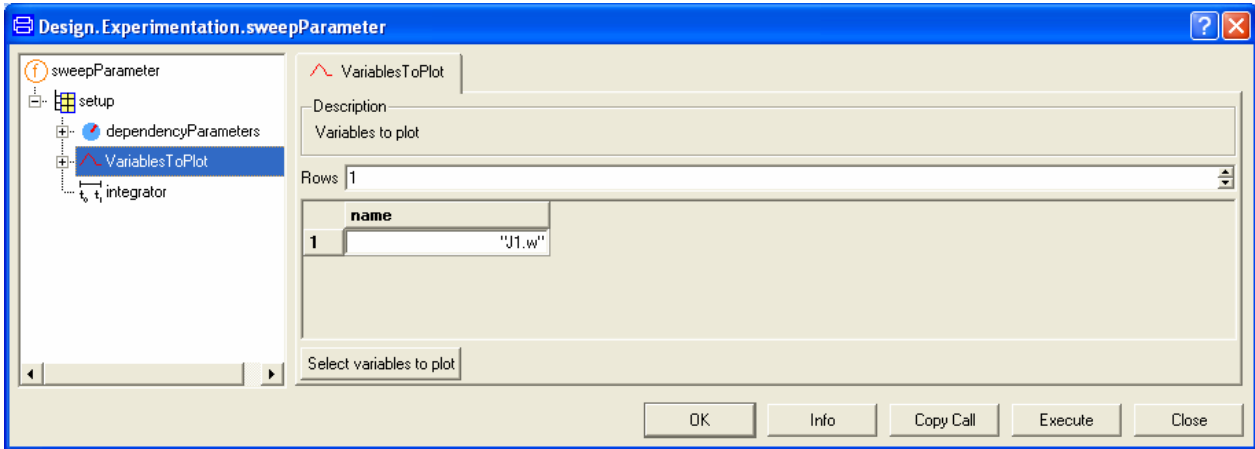
The setup of this function is very similar to perturbParameter. Click on Design.Experimentation.sweepParameter to get the setup menu. The model is already filled in.



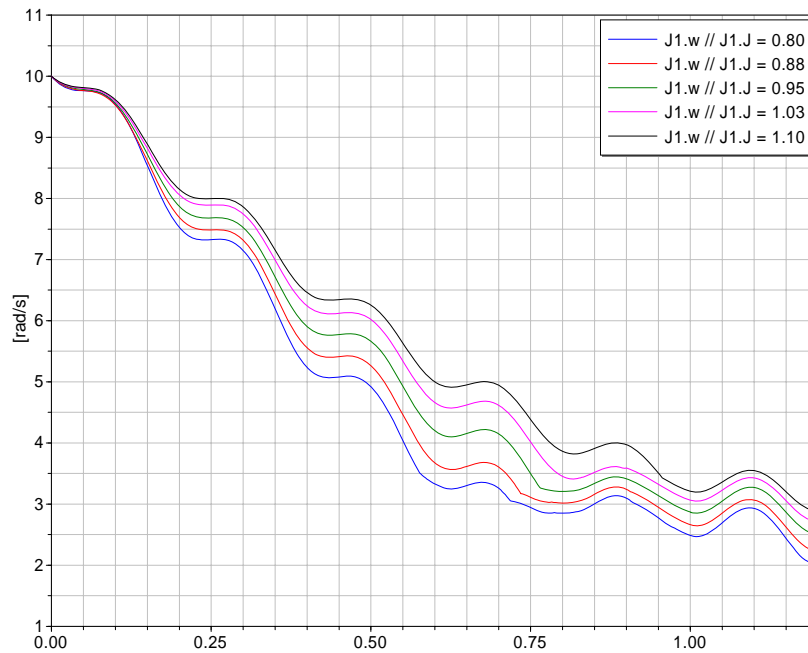
We have to select the dependency parameter and the variable to plot. The way is the same as before. We just present the menus as a guide.



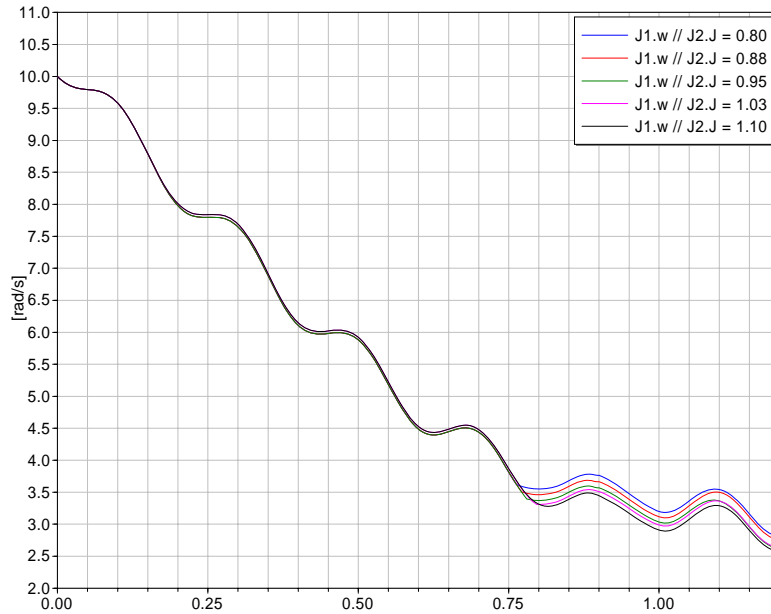
In this case, we are selecting five equidistant values between 0.8 and 1.1 for J1.J. The variable to plot is J1.w



Don't forget to set the Stop Time to 1.2 in the integration setup and the tolerance to 1e-6! Press "Execute" and the result follows.



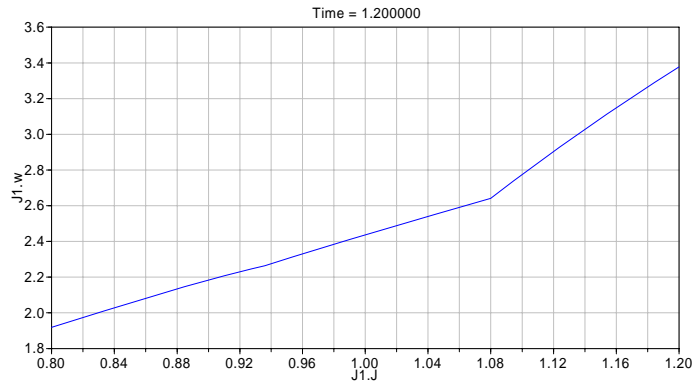
Let us observe now  $J1.w$  and vary  $J2.J$ . Change in the setup  $J1.J$  with  $J2.J$ , in dependencyparameters setup. Press Execute again.



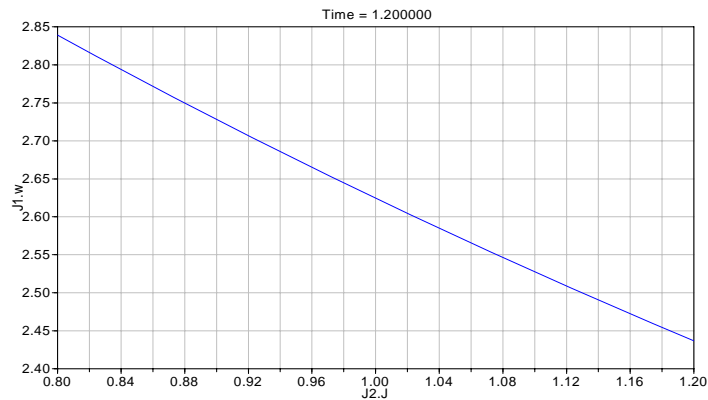
The response  $J1.w$  is less sensitive at the beginning of the interval to variations of  $J2.w$ . At the end, when all inertias are coupled, the variation is larger.

### **sweepOneParameter**

If our interest is just the response at end point of the interval, we use `sweepOneParameter`. This setup is the same as for `sweepOneParameter`. Just choose  $J1.J$  as dependency variable in the same way, take 51 values between 0.8 and 1.2 and use  $J1.w$  as variable to plot. The following curve is obtained when the command is executed. Once more, don't forget to set the Stop Time to 1.2 in the integration setup and the tolerance to  $1e-6$ .

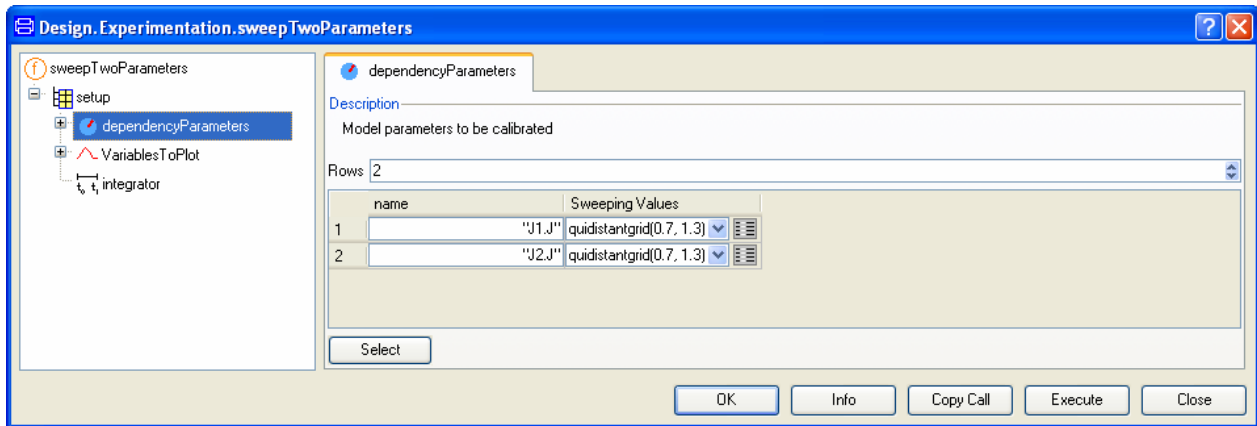


This curve relates at  $t=1.2$  the parameter  $J1.J$  and the response  $J1.w$ . The same situation can be depicted for  $J2.J$  as parameter and  $J1.w$  as response. The figure follows.

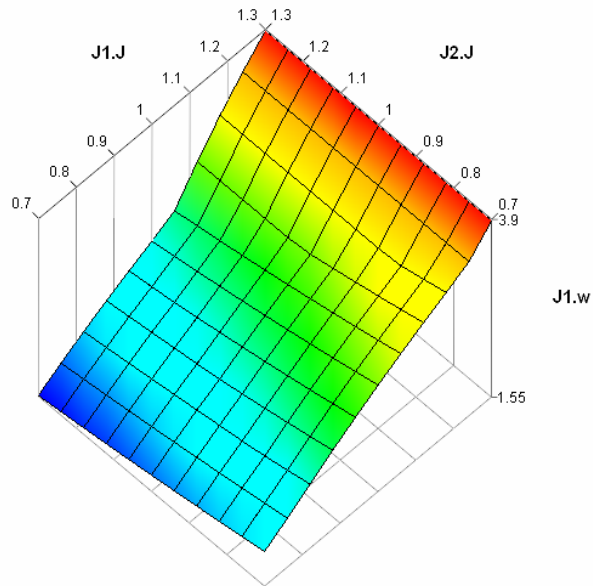


## Sweep Two parameters

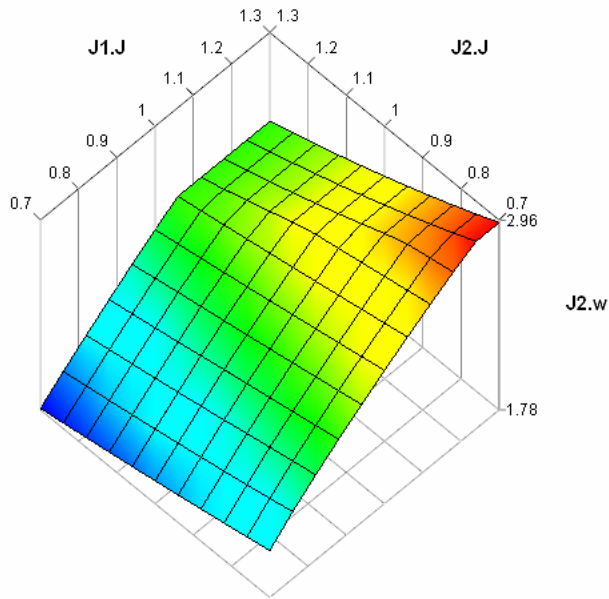
To study the dependence of one response with respect to two parameters at the end of the integration interval, the function `sweepTwoParameters` is to be used. The setup is almost identical to `sweepParameter` and `sweepOneParameter`. The only difference is that two dependency variables are to be selected instead.



We observe now J1.w against J1.J and J2.J. The values chosen for J1.J and J2.J are eleven values between 0.7 and 1.3 for both variables. Even for this case, the Stop time is 1.2 and the tolerance is 1e-6 in the integrator tab.

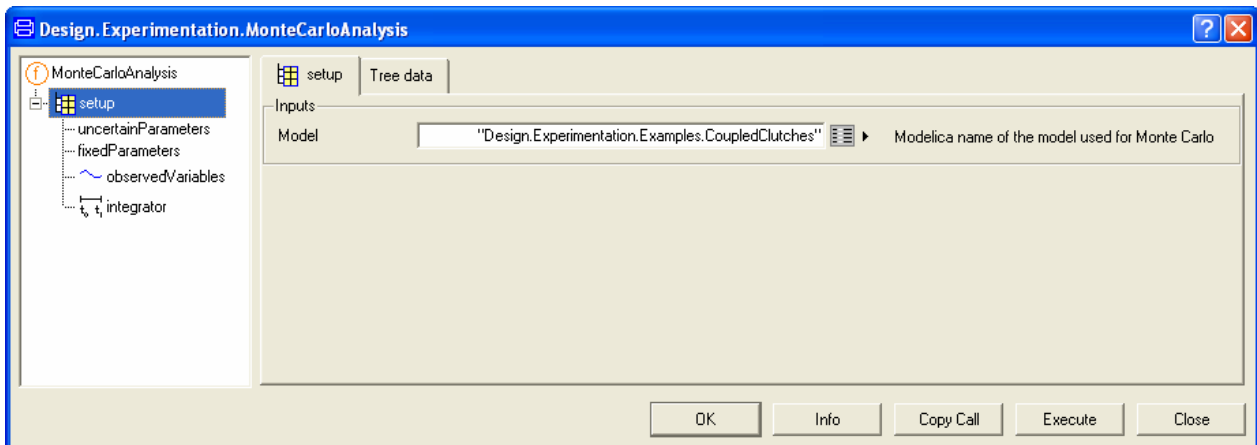


Observing J2.w gives the following result.

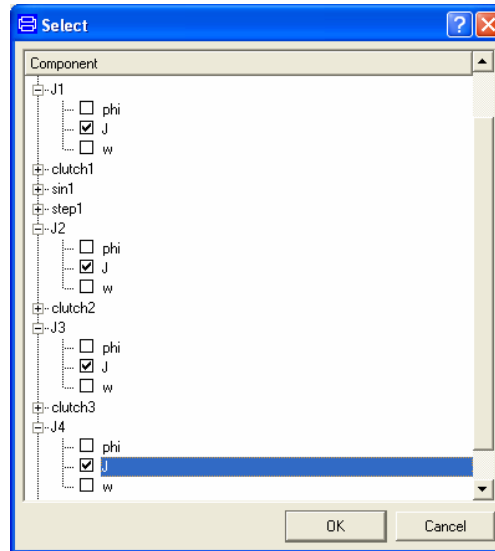


## Monte Carlo Analysis

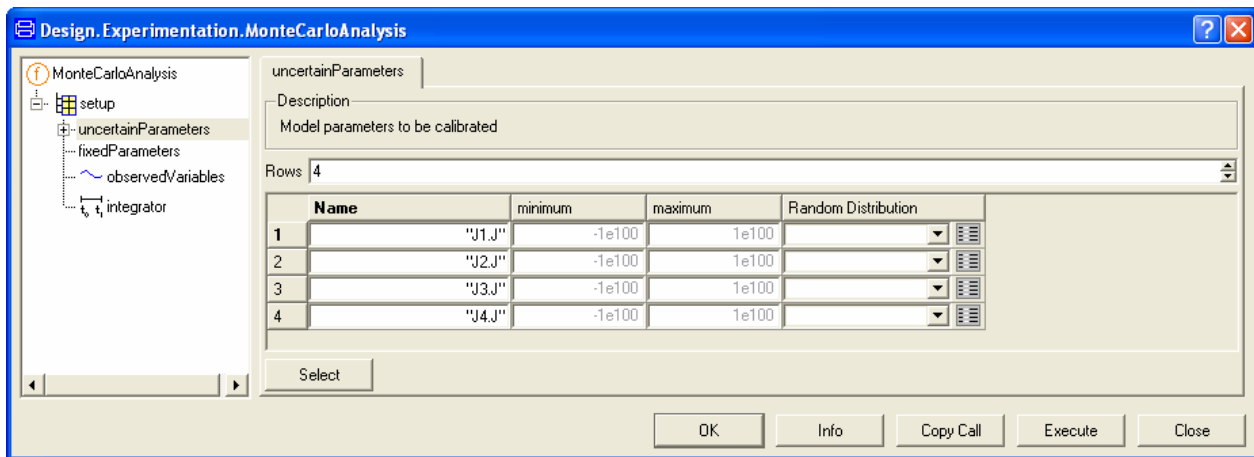
Monte Carlo Analysis is widely used to explore the behavior of a model when the input parameters are multidimensional. We will set up now the command MonteCarloAnalysis to observe the model response when varying J1.J, J2.J, J3.J and J4.J at the same time. As before, select Design.Experimentation.MonteCarloAnalysis function. The following menu pops.



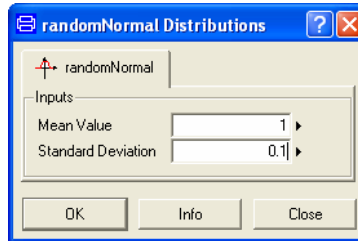
The task now, as before, is to select the uncertain parameters. Click on uncertainParameters and click on “Select”-button. Select in the browser J1.J to J4.J.



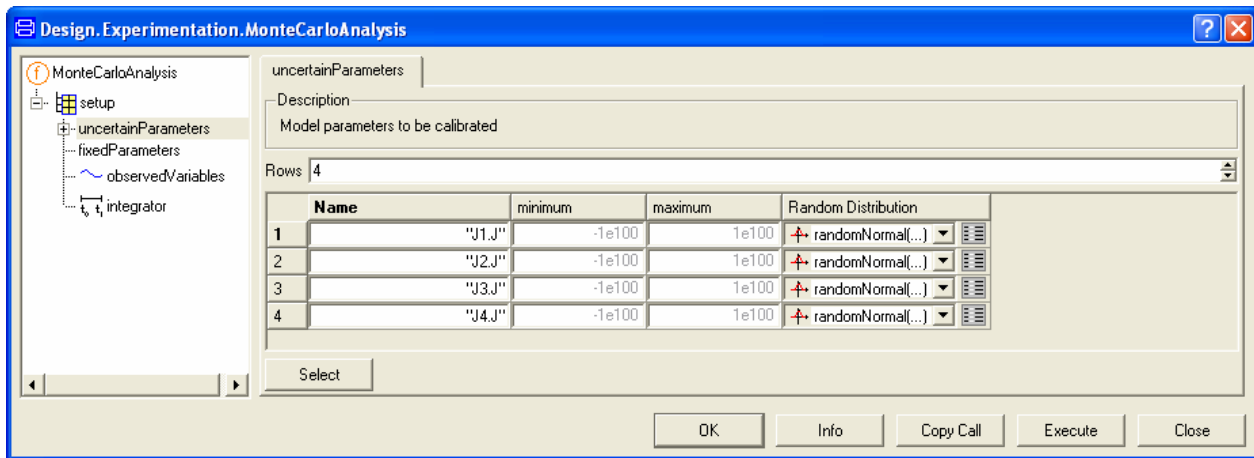
And click OK. The next step is to select a random distribution for every inertia.



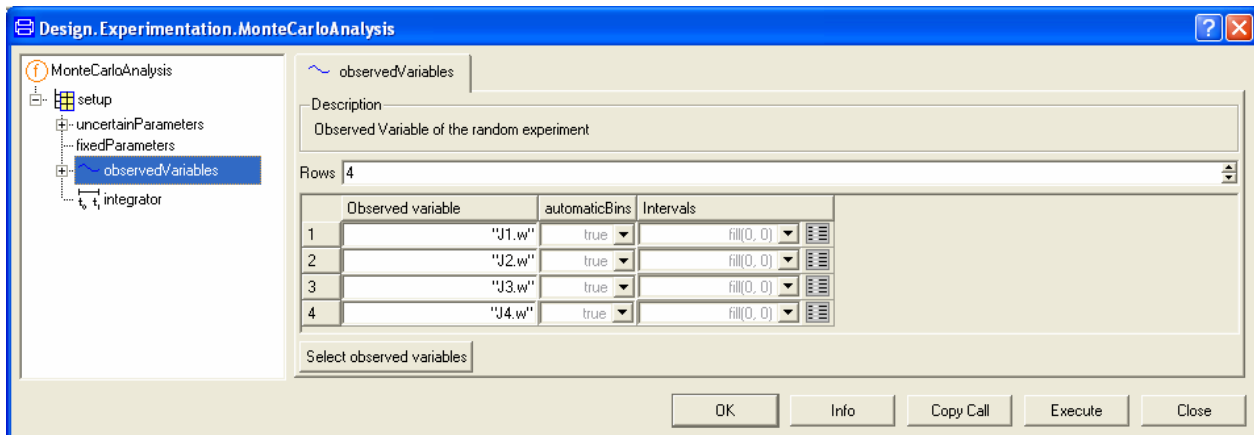
Click on the arrow of the combo box and select randomNormal for J1.J. Another menu pops up asking for values for “mean” and “standard deviation”. Those values characterize the normal distribution to be used. Set mean to 1 and standard deviation to 0.1.



Click OK. Repeat the same process for J2.J to J4.J.

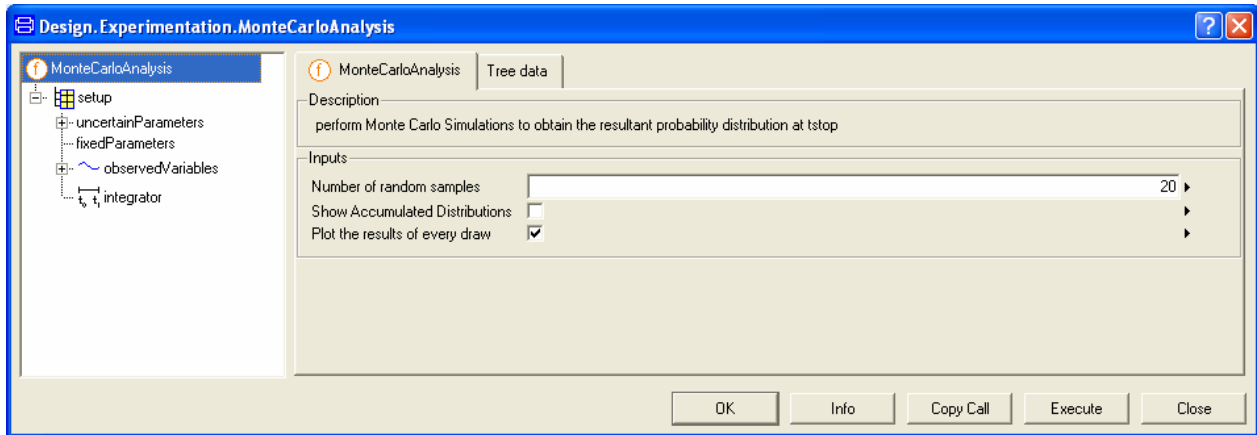


The setup for fixedParameters is used if we want to specify other simulation situations than the nominal values written in the model. For instance, if the initial angle J1.phi is specified and different from zero, we should add it there. In our case, we don't have such fixed parameters so we just go directly to observed variables. Click on observedVariables and press the button "Select observed variables". Mark in the browser J1.w, J2.w, J3.w and J4.w.

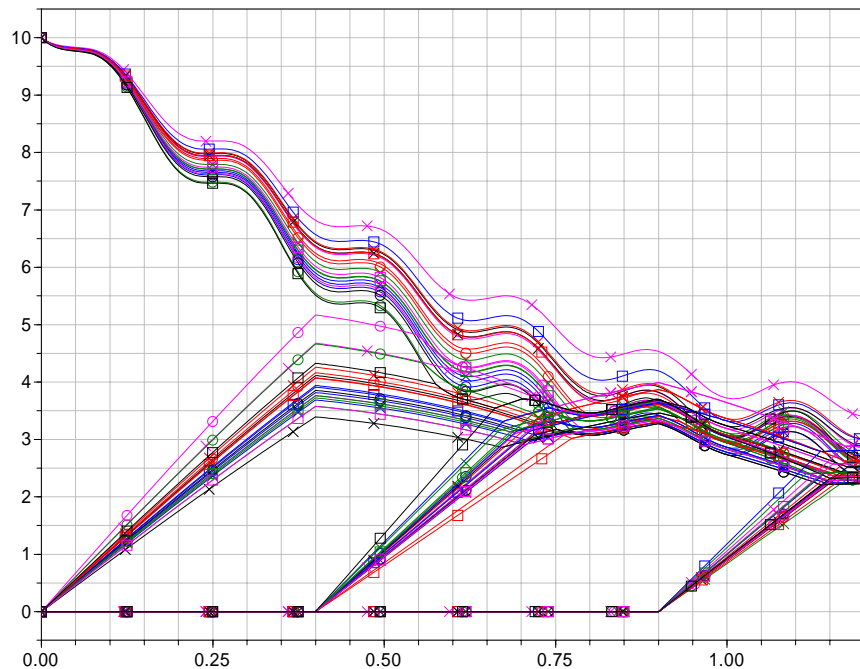




The flag `automaticBins` set to `true` allows the algorithm to choose automatically an appropriate set of bins, according to the maximum and minimum values observed in the result. It takes also into account the total number of samples to set the appropriate resolution. Set the integrator stop time once more to 1.2. To set up the type of desired result, click on `MonteCarloAnalysis`.

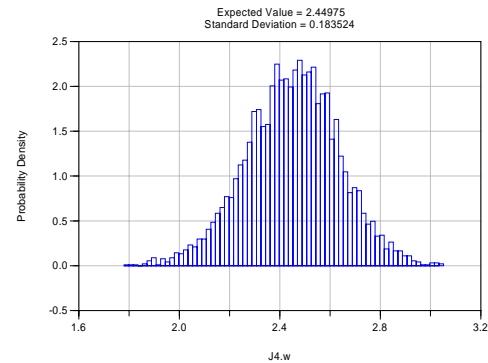
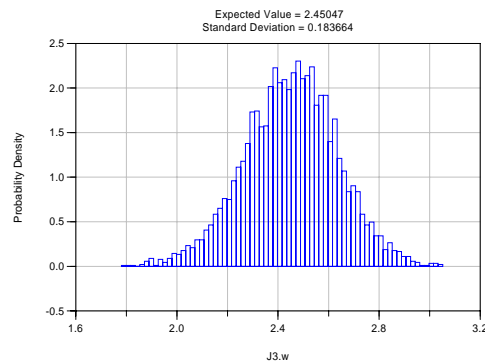
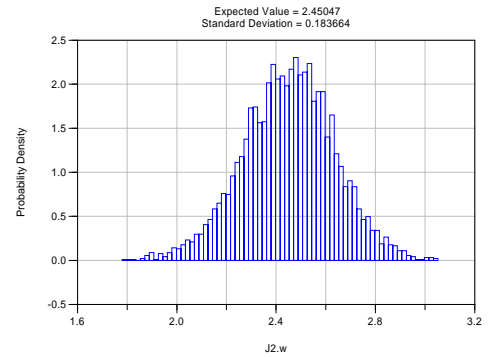
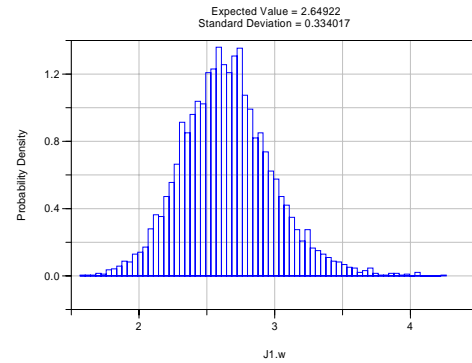


We set the number of draws in the field “Number of random samples”. As we want to plot the result of every draw, only twenty draws are needed. Check also “Plot the results of every draw” to obtain the plot of the responses and the density of probability.

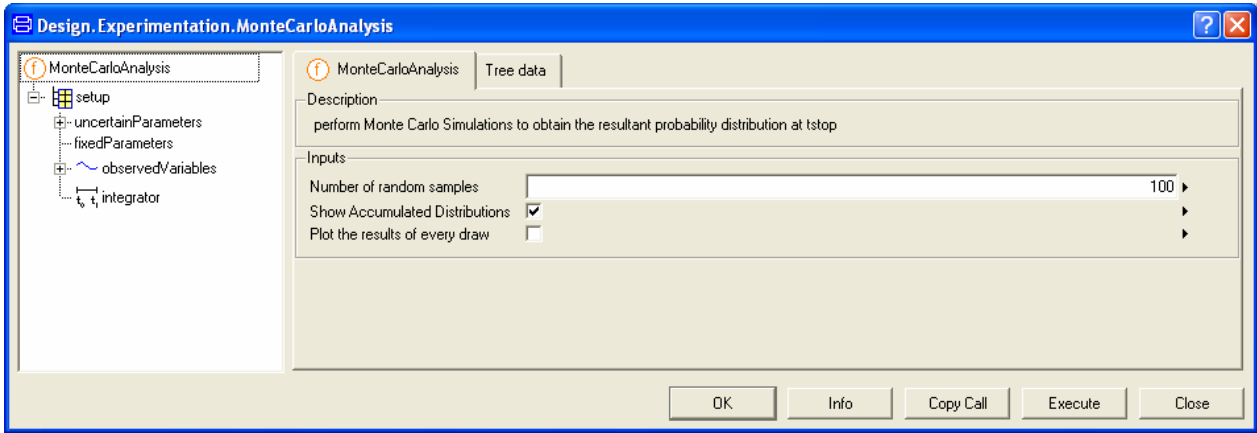


In this graph we observe the variation of slope and behavior produced by random sampling of the values of J1.J, J2.J J3.J and J4.J in time.

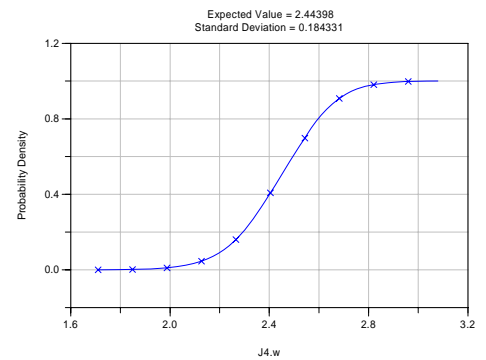
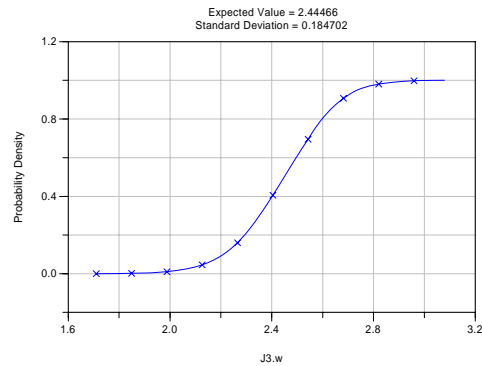
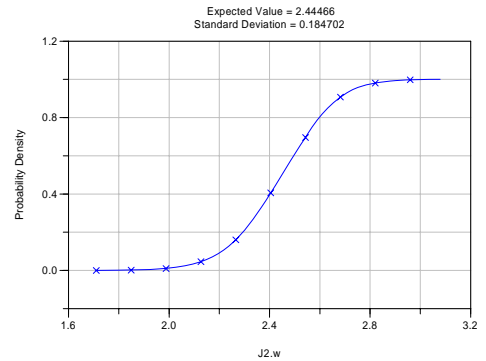
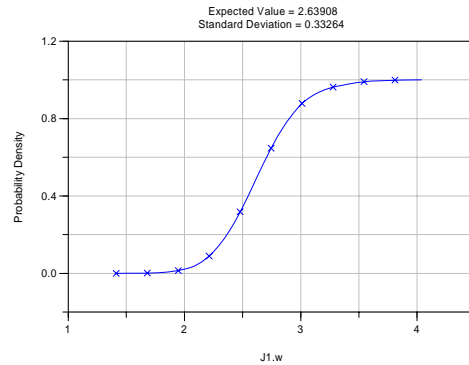
If the plots of the density of probability or accumulated probability are important, we change the setup to plot those with more samples. To plot the densities, we take five thousand samples and uncheck the flag “Plot results of every draw”. Press Execute to obtain the plots.



To plot the accumulated distributions, check the flag “Show accumulated distributions”.



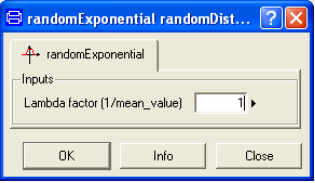
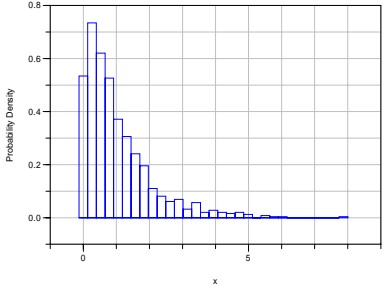
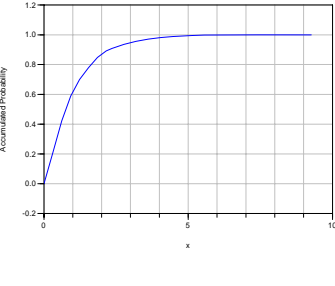
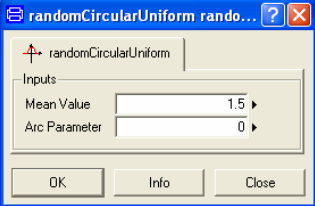
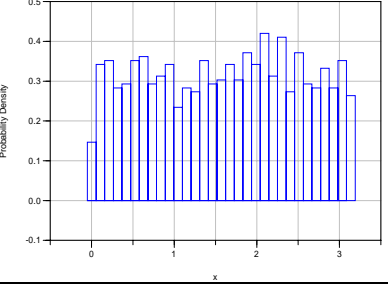
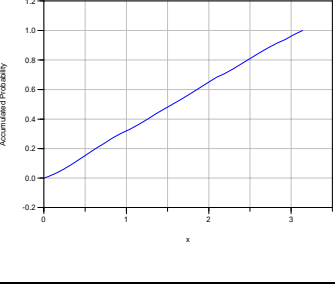
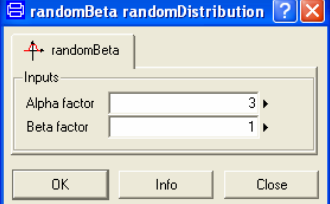
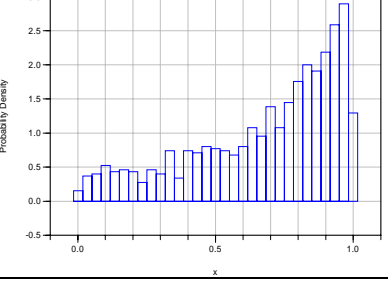
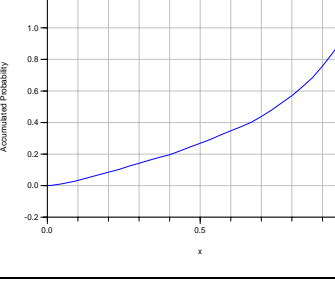
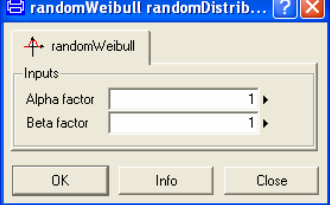
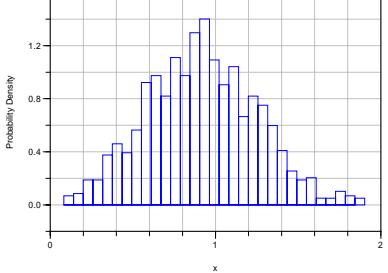
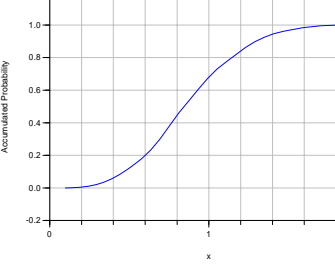
Click on Execute. The result plots follow.

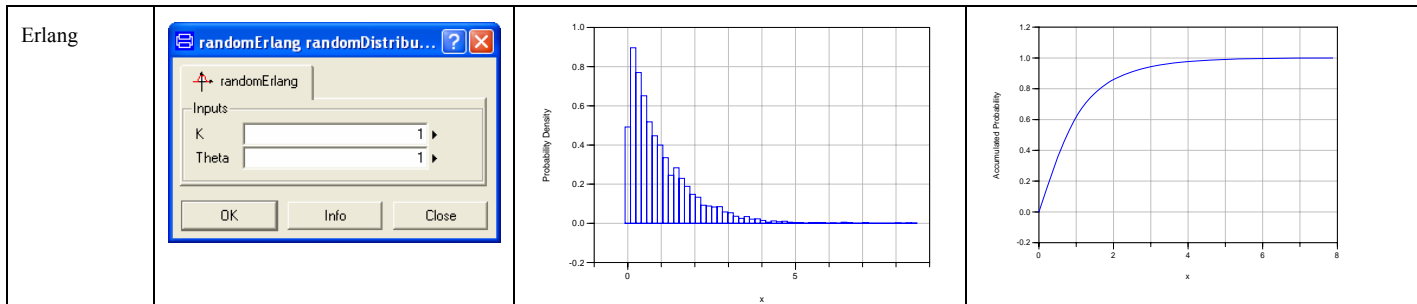


## Random Distributions available and their parameters

The following table reviews briefly the random distributions in Experimentation package that can be used together with MonteCarloAnalysis.

Distribution	Parameters	Probability density	Accumulated probability
Normal			
Uniform			
Logarithmic Normal			
Pareto			

<p>Exponential</p>			
<p>Circular Uniform</p>			
<p>Beta</p>			
<p>Weibull</p>			



---

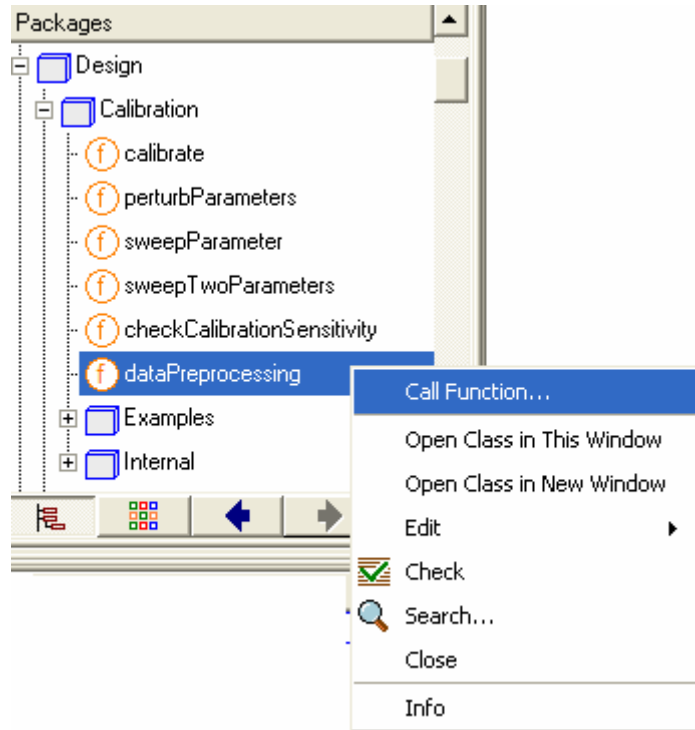
## Data Preprocessing

The quality of the calibration process is directly related to the quality of the measured data used as input to the calibration tool. Any factor that perturbs the data will cause directly distortion of the final result of the calibration tool. It is important then to preprocess the data, that is, to adjust the data eliminating noise, zones where the model is not valid and erroneous or not representative measurements.

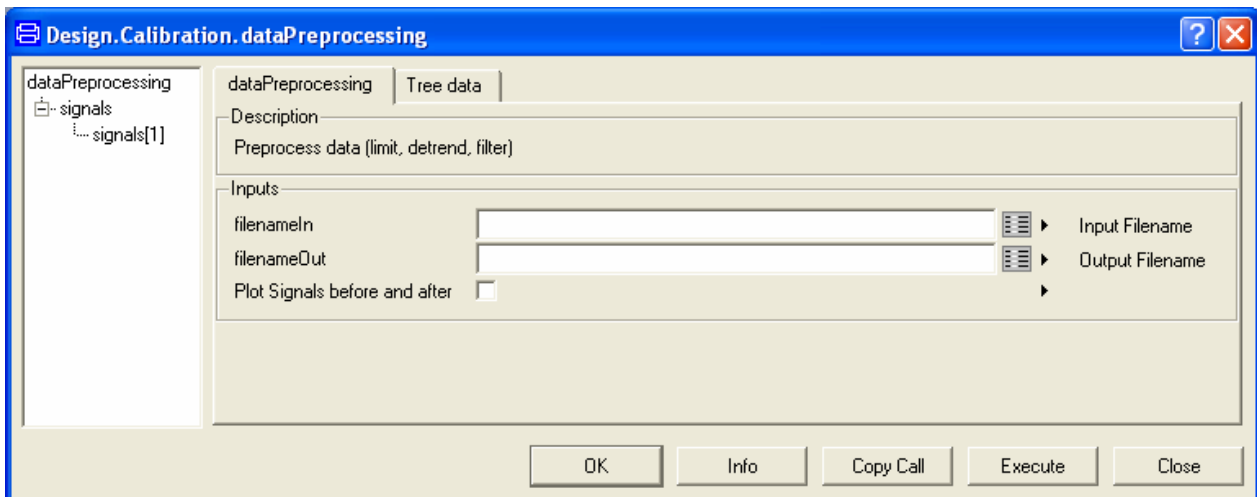
The Design.Calibration package incorporates the function `dataPreprocessing` with this objective: preprocess data for calibration.

### Setting up for preprocessing

Select `dataPreprocessing` in the class browser under `Design.Calibration`. Click right mouse button, select the command "Call function..."

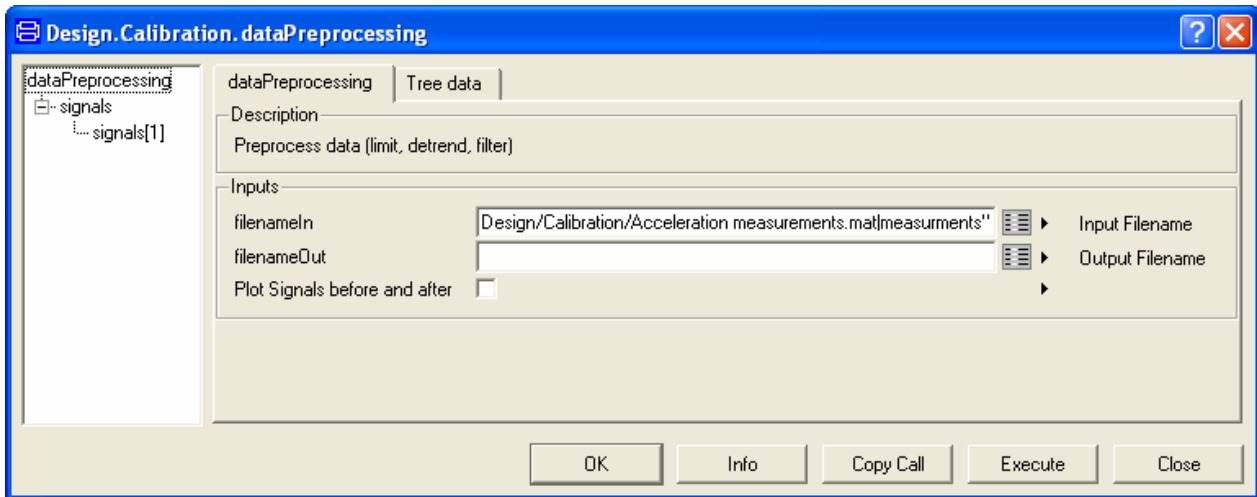


The dialog of dataPreprocessing pops. We select now the file we want to process and the file that will contain the output.



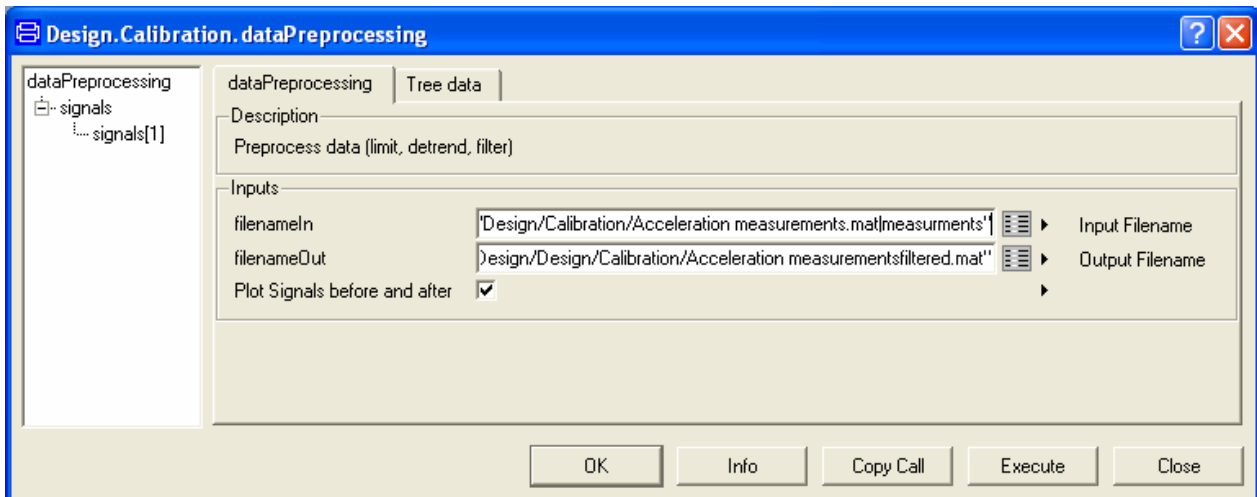
To introduce an experiment file, click on the Edit icon in the “Input Filename”. A file browser pops up. Use it to select the file

.../Design/Calibration/Acceleration measurements.mat.



The same procedure has to be repeated to select an output file. In this case, the file does not exist. We choose as name for this example

.../Design/Calibration/Acceleration measurementsfiltered.mat.

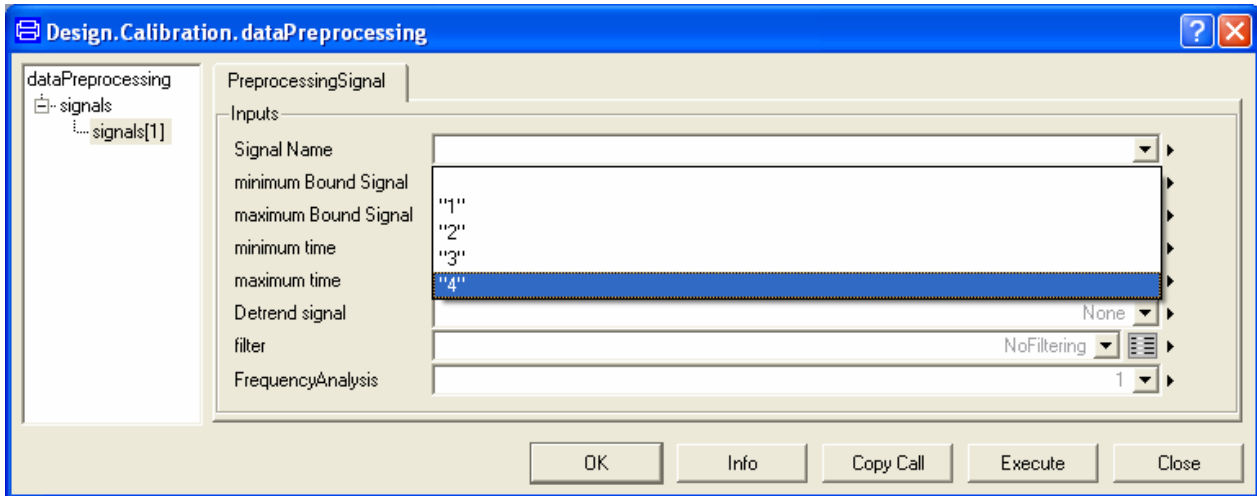


and check the field “Plot Signals before and after” to obtain a plot of the original signal and the result of the preprocessing.

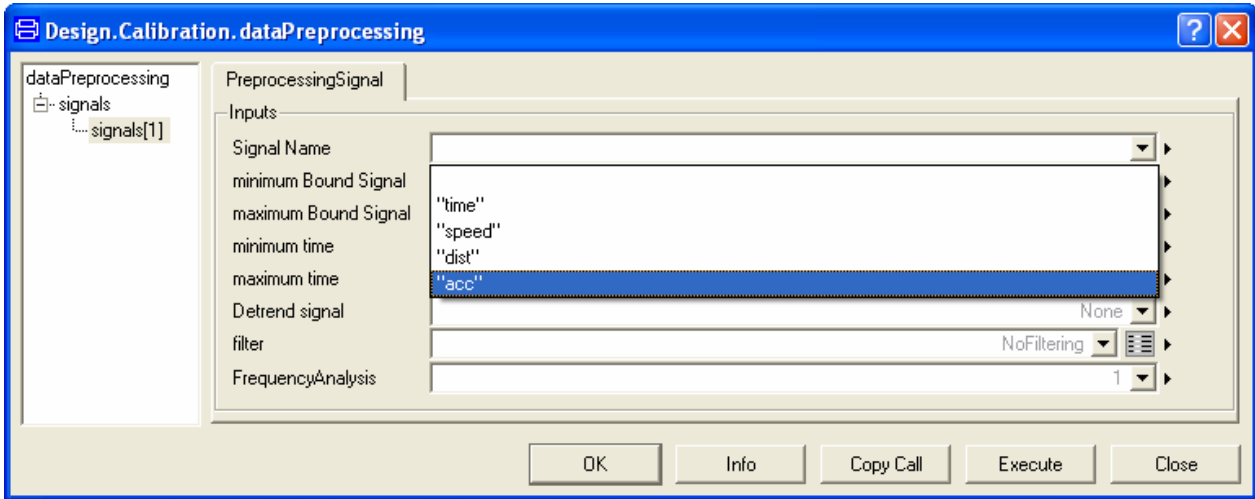


It is possible to overwrite the original file, there is no restriction for that. But the data is destroyed. This means that the original data is gone, and no possibility of recovery is possible. A good practice is always to change the name slightly, since we might want to adjust later on the preprocessing parameters.

We select now a signal from the file. Since the input filename is a .mat file, we don't have access to the name of the variables, but we know the position of the acceleration signal, that is, number four in the matrix. Choose "4" from the combo box in "Signal Name".



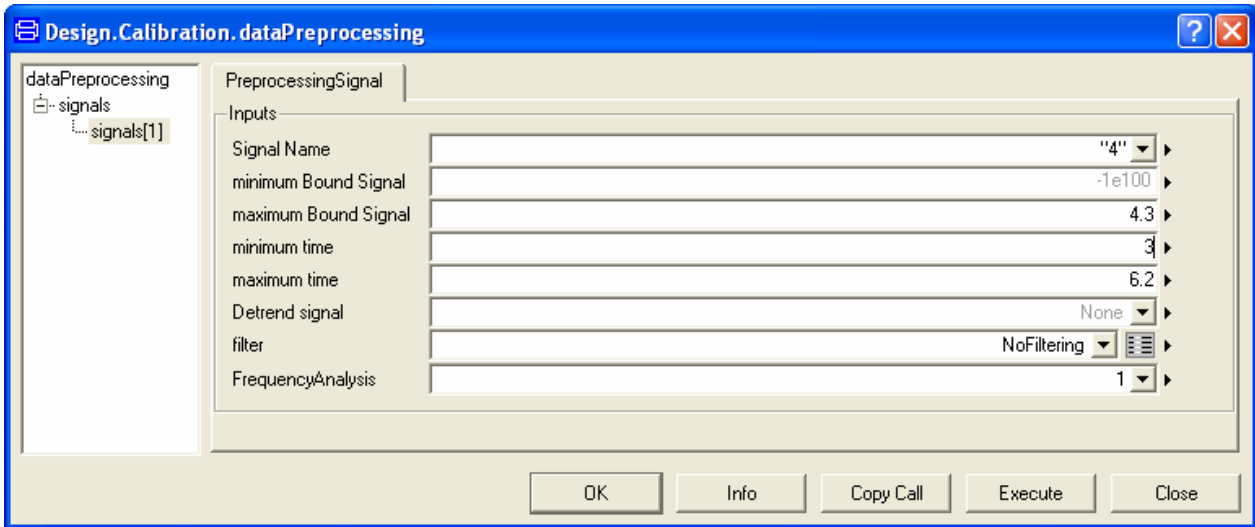
The dataPreprocessing tool assumes for .mat files that time is in the first column always. This is a cornerstone of the function, since all functionalities are relying on time. This is very important. If we instead choose to process a .csv file, we get the names in the combo box. We can simply then select "acc". And dataPreprocessing will seek the keyword "time" and "Time".



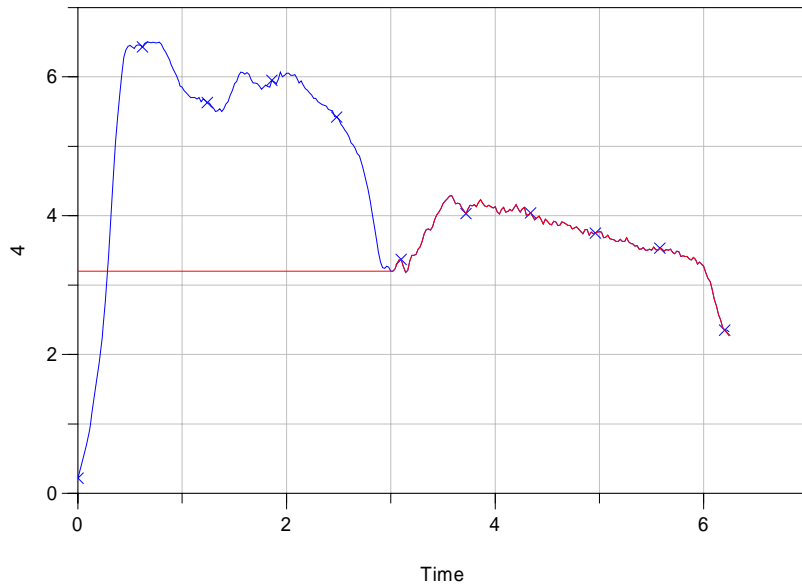
We are ready now to run the preprocessor function. We start limiting and detrending functions.

## Limiting and detrending signals

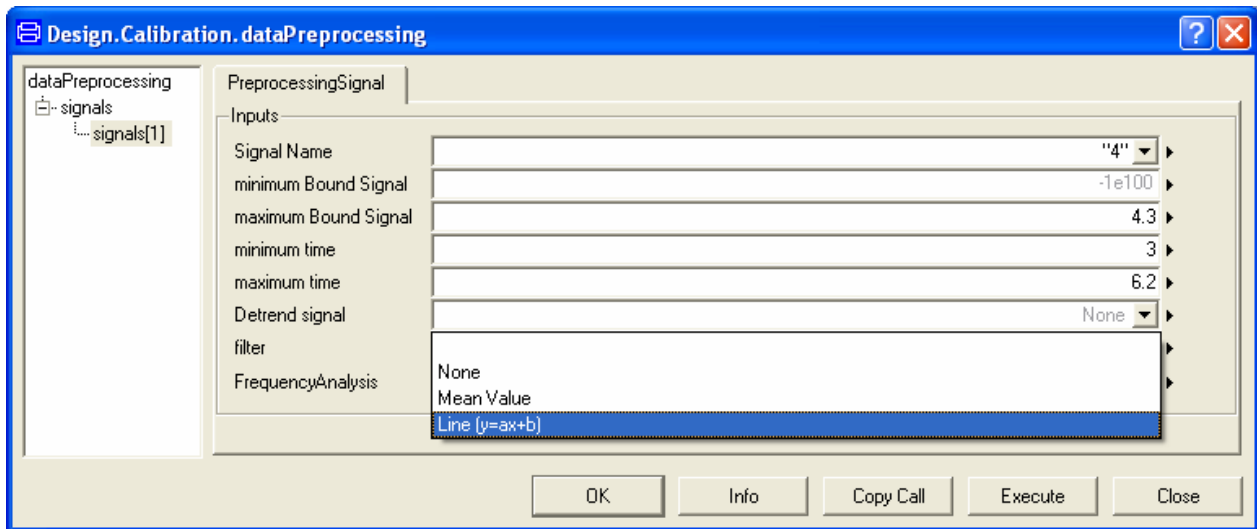
Limiting and detrending the signals is also very important. To limit a signal in time and amplitude it is enough to write the desired values in the fields “minimum Bound Signal” and “maximum Bound of Signal” for the amplitude and “minimum time” and “maximum time” for time axis. The data outside of these limits is taken away and interpolated or extrapolated linearly.



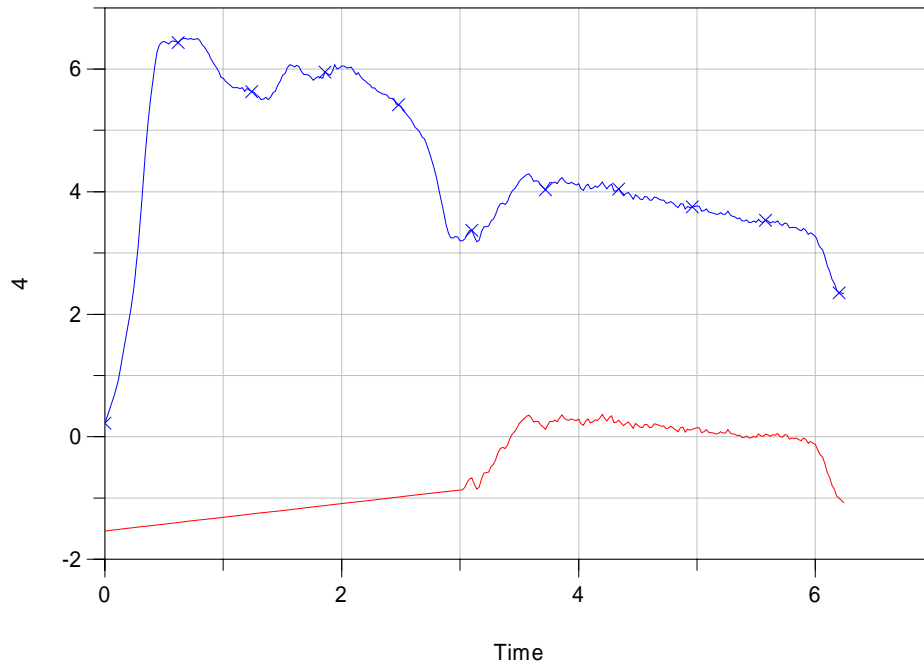
Just to demonstrate this feature, take as limits the interval  $[-1e100, 4.3]$  for the amplitude and  $[3, 6.2]$  for time. Press Execute and the result is presented.



All values outside of the range have been substituted by linear interpolation. Now, we choose “Line( $y=a*x+b$ )” in the combo box “Detrend signal”. This will fit in least squares sense a straight line and subtract it from the data.



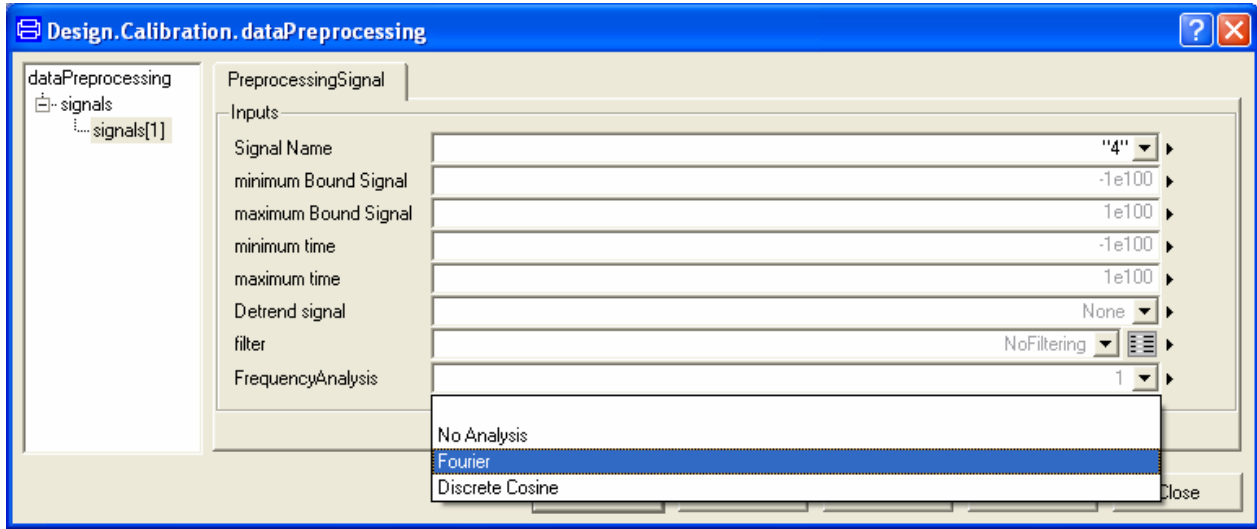
Press Execute and the result follows.



We observe the resulting curve. The other possibility for detrending is “Mean Value”, that subtracts the mean value of the function. Reset the values of the limits and set detrending to “None”. We are now into frequency analysis and filtering of signals.

## **Analysing Signals: is there any noise?**

Let us analyse the function in the frequency domain. The point now is to filter out noise. Such a noisy perturbation is normally easy to get in the measured data and complicates later on the calibration process unnecessary. Select “Fourier” from the combo box of “FrequencyAnalysis”.



We are about to perform the discrete Fourier transform (DFT) of the acceleration signal. The DFT is defined as follows. Assume we have  $N$  samples of a function  $x_n = x(t_n)$  at  $t_n = nT_s$ , where  $T_s$  is the sampling time. The DFT is a set of complex numbers  $c_k$  such that

$$x_n = \sum_{k=1}^N c_k \exp(i \omega_k n T_s)$$

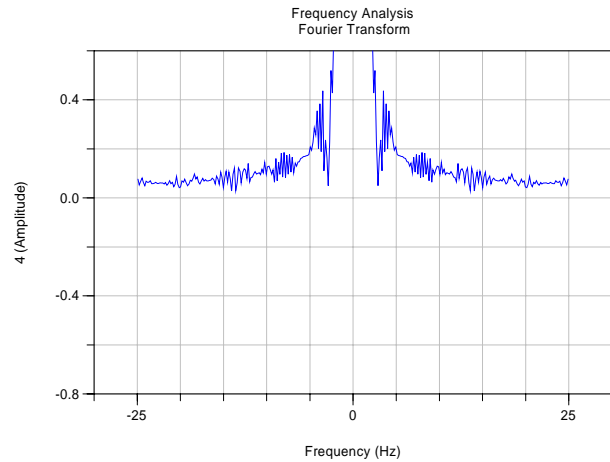
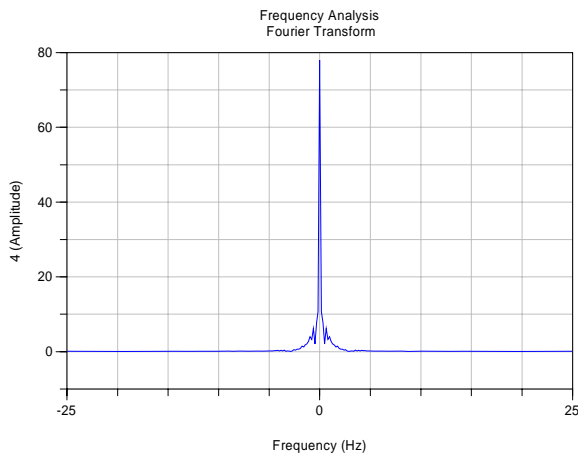
for all sampling  $x_n$  points and frequencies  $\omega_k = \frac{2\pi k}{NT_s}$ , and  $i$  the imaginary unit. The coefficients  $c_k$  can be calculated explicitly by matrix-vector multiplication or by more effective algorithms in case of large amounts of data.

The frequencies are discrete equidistant points distributed in the interval  $\left[0, \frac{2\pi}{T_s}\right]$ . Since

the complex exponential function  $\exp(i \omega_k n T_s) = \exp\left(\frac{i 2\pi k n}{N}\right)$  is periodic, we choose

a representation in the interval  $\left[-\frac{\pi}{T_s}, \frac{\pi}{T_s}\right]$ , to have the highest frequencies farther at the

boundary, instead of in the middle of the graph. Now, we press Execute and obtain the graph at the left side. The right side graph is a zooming.



Since the coefficients are complex numbers, we present their modulus. In the log of the command window we observe also the following report

```
Processing signal 4
```

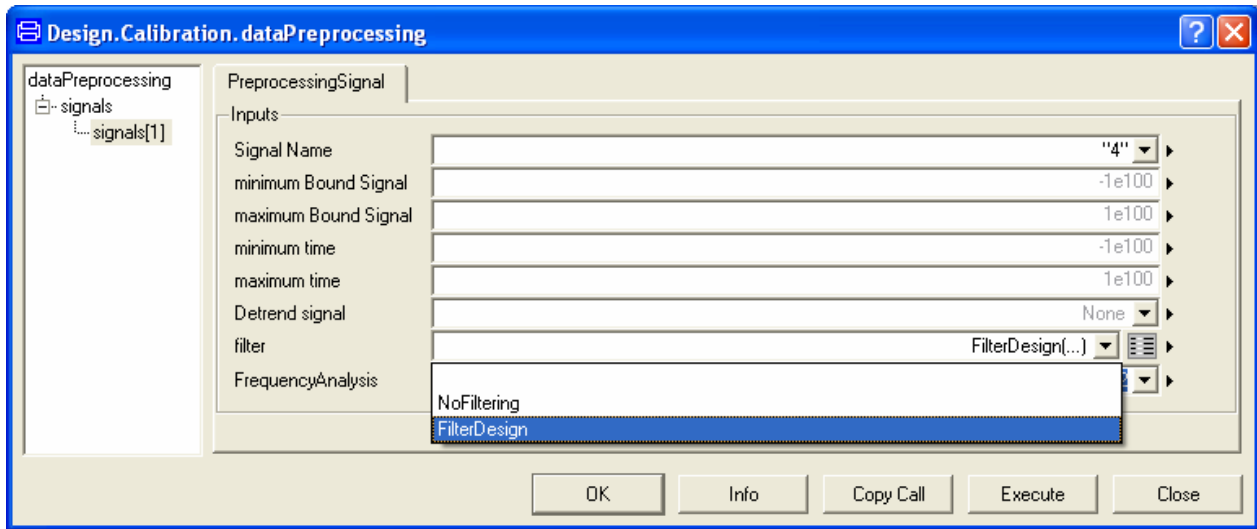
```
Signal 4 has the 99.5341% of its power under 2.21519 Hz
```

This is an important piece of information. The tool detected that the energy of the signal is almost condensed in the interval  $[-2.2159, 2.2159]$ . In the graphs before (right) we observe the behavior of the coefficients, and it is less smooth and more erratic outside of the interval reported by the tool.

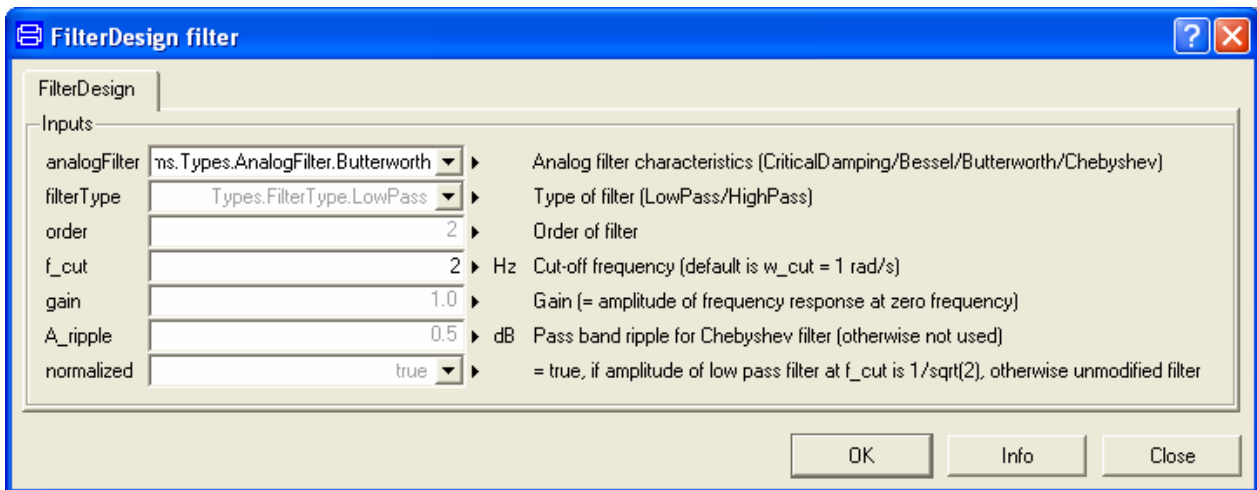
We can therefore suspect that additive noise is present in this range of frequencies. We can now design a filter and get rid of these noisy oscillations.

## Filtering signals

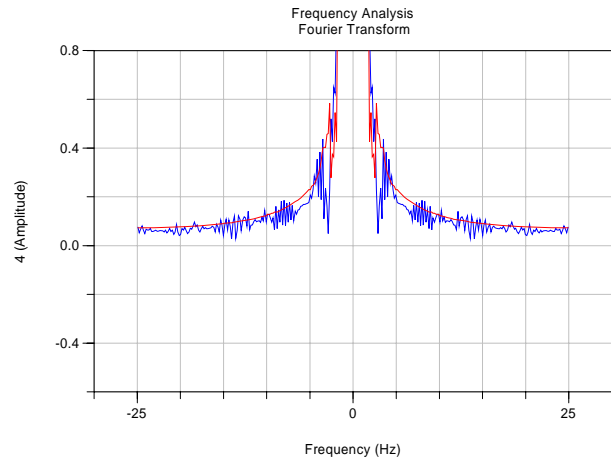
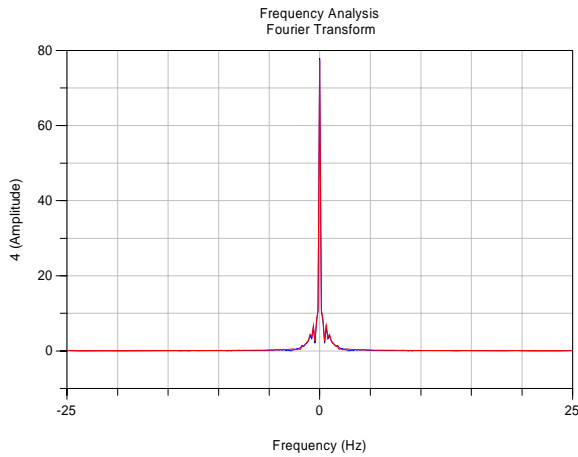
Back to the GUI, click on “filter” combo box and choose “FilterDesign”.



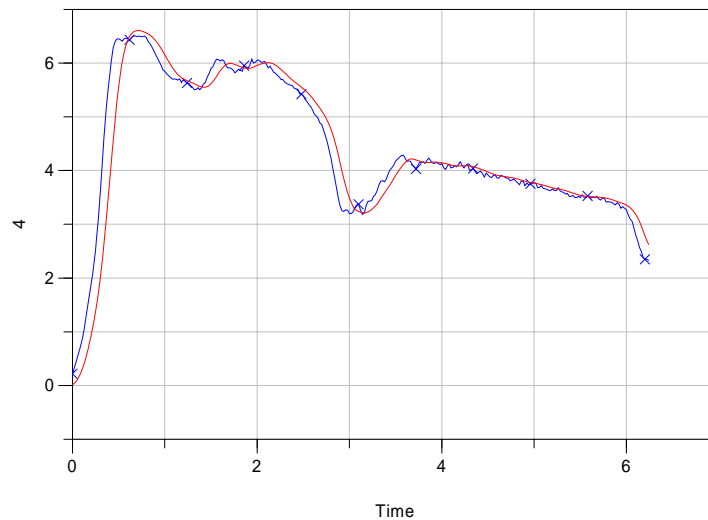
The GUI for filter Design from the LinearSystems library pops up now.



We choose a Butterworth filter in “analogFilter” and as cut frequency we choose 2 Hz. It is enough to choose 2 Hz and not exactly 2.2159 since the filter is not ideal and we will smooth out the spectrum of the signal around those frequencies too. The type of the filter has to be lowpass since the signal is clustered around zero frequency. Press OK. Then Execute in the main GUI. The resultant spectra are presented.



We observe how high oscillatory modes are smoothed out. This means that the signal in time is also smoother. The result is presented in the next picture.



The filtered signal (red) has less noise than the original one (blue). This makes the calibration process easier. The filters are constructed using the LinearSystems library from the Modelica Standard Library 2.2. These are discretised versions of continuous systems, with a discretisation in such a way that the ramp response is exact. The possible filters are four: Critically damped, Bessel, Butterworth and Chebyshev.



# **Model Calibration**



# Model calibration

---

## Introduction

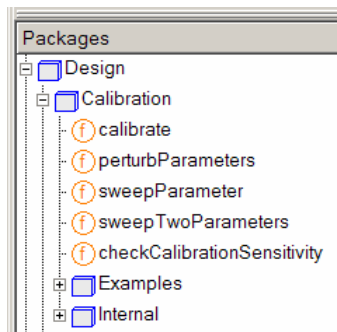
Dymola includes features to perform integrated computer experiments with Modelica models. This document describes the features to calibrate and to assess models. The functions described in this document are parts of the Design.Calibration package. The ModelCalibration option is required for problems with more than one tuner. However, the examples given below can be run without the ModelCalibration option.

Consider a Modelica model describing a physical system. Such a model includes typically many parameters, which have to be set. Some parameter values can be found from design sheets. Some parameters such as physical dimensions may be easy to measure on the system. Direct measurements of the weights of the parts are more difficult since it requires the system to be dismounted. Moreover, it is for example not simple to measure the inertia of a part. Friction and loss parameters are good examples of parameters that often are unknown.

Model calibration (parameter estimation) is the process where measured data from a real device is used to tune parameters such that the simulation results are in good agreement with the measured data. The parameters that we tune are often referred to as tuners. Dymola varies the tuners and simulates when it searches for satisfactory solutions. Mathematically, the tuning procedure is an optimization procedure to minimize the error between the simulation results and the measurements.

When tuning parameters from measurements, a basic question is “Which parameters can be estimated from the measurements available?” Changing a parameter to be estimated must of course influence the output. However, this is not enough. Two or several parameters may influence the result in a similar way such that it is not possible to estimate them individually. Dymola includes function to analyze and to plot parameter sensitivities. When a set of parameters have been tuned, it is recommendable to validate the model and the tuned parameters against other measured data to check that there is a good agreement between the simulation result and the new measurements. For a specific series of measured data it is possible to get good fits by increasing the model complexity and the number of tuned parameters. However, this does guarantee that the result is that good for other operating conditions.

To load Design.Calibration, select File/Libraries and click Design.



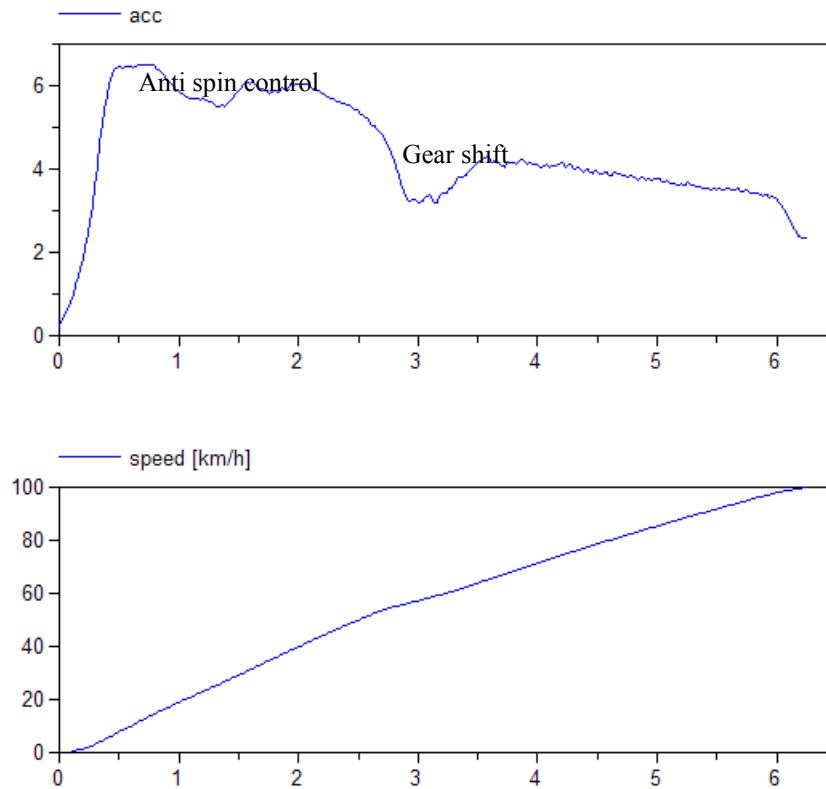
The function `Design.Calibration.calibrate` is the main function for calibration and validation of models. There is also a set of functions for analyzing parameter sensitivities and dependencies of calibration tasks. For parameter studies in general see `Design.Experimentation`.

This document uses a simple car model describing translational motion to illustrate how a basic calibration task is set up and executed. How to store a setup for later reuse is described. Then this document describes the set of functions to analyze parameter sensitivities and dependencies of calibration tasks.

---

## The basics of setting up and executing a calibration task

We have acceleration and speed measurements from a BMW 645i at full throttle as shown in the plot below. For further information we refer to Auto Mobil, Issue 1, 2005.



Anti spin control and gear shifting make the acceleration curve complex. Here we will focus on the time interval 3.8-6 seconds when the second gear is engaged.

We need to describe how the generated torque makes the car move. Thus we need to make a simple powertrain model including gearbox and rotating elements which make the wheels rotate.

## Vehicle data

By searching on the web we can find the following data for the car

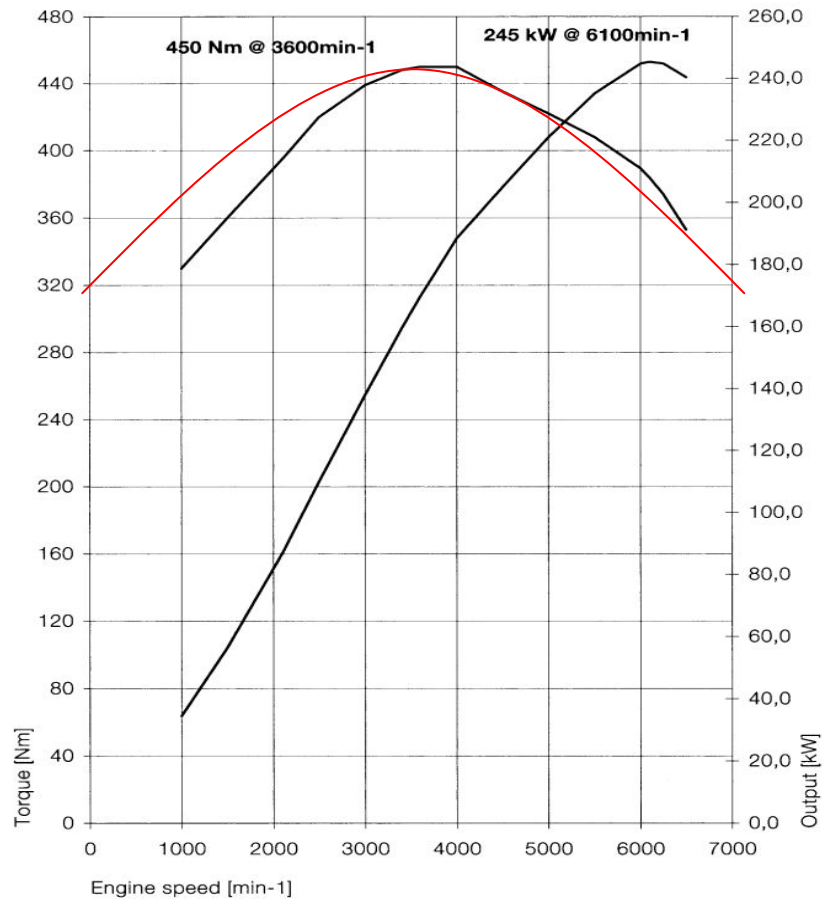
Engine torque at 3600 rpm [Nm]	450
Engine inertia [ $\text{kgm}^2$ ]	0.4
Gearbox and cardan inertia [ $\text{kgm}^2$ ]	0.01
Wheel inertia [ $\text{kgm}^2$ ]	4* 1
Wheel radius [m]	0.34
Car mass [kg]	1690
Automatic gear ratios I-VI	{4.17, 2.34, 1.52, 1.14, 0.87, 0.69}

Gear ratio of final gear	3.46
--------------------------	------

The wheel radius is calculated for 245/45 R18 W saying that the radius is  $18''/2 + 0.45*245 = 0.338$  m.

Engine characteristics at full throttle for a BMW 545i were found at

<http://www.e60.net/information/options/engines/N62B44/>



BMW 545i and BMW 645i have the same 4.4-liter V8 engine. The black lines in the plot above show the torque and power characteristics.

As a first approximation we fit a quadratic characteristic:

$$\tau = \tau_0 + (\tau_{\max} - \tau_0) * (1 - ((w - w_{\max}) / w_{\max})^2) ;$$

The parameter  $w_{\max}$  is  $3600 * 2\pi / 60$  rad/s and  $\tau_{\max}$  is 450 Nm. Choosing  $\tau_0$  to 320 gives the red curve in the plot above.

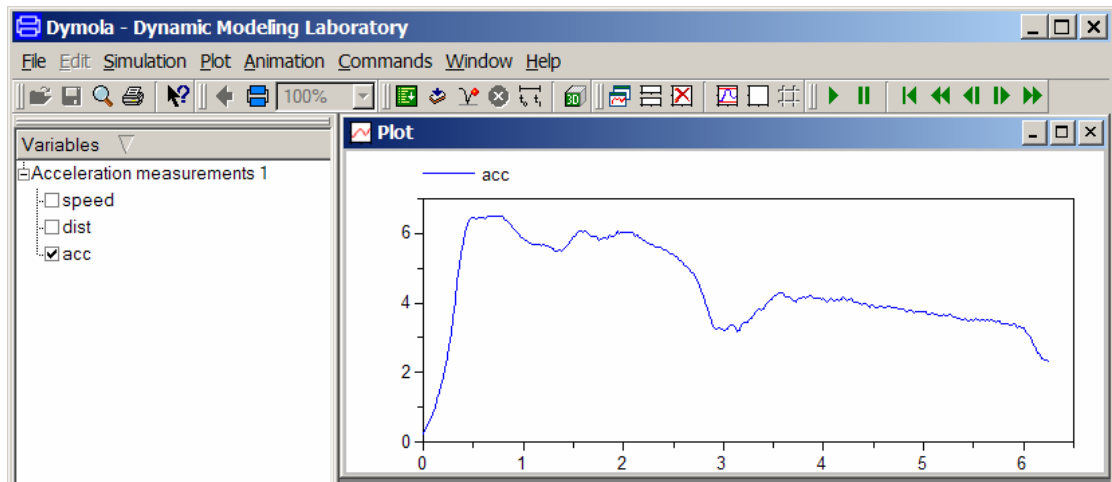
The velocity and acceleration measurements are stored simply as a csv file

*Program Files/Dymola/*

*Modelica/Library/Design/Acceleration measurements.csv*

	A	B	C	D
1	time	speed	dist	acc
2	0	0	0	0.22
3	0.02	0.2	0	0.33
191	3.78	68.1	37.84	4.14
192	3.8	68.4	38.22	4.16
193	3.82	68.7	38.6	4.13

The first row of the file includes the column headings and then the data follow. Dymola supports plotting of such a csv file. Select “Plot/Open Results...” and a file browser pops. Use it to select the csv file. The file and its variables appear in the plot browser and can be plotted in the usual way.



## Vehicle model

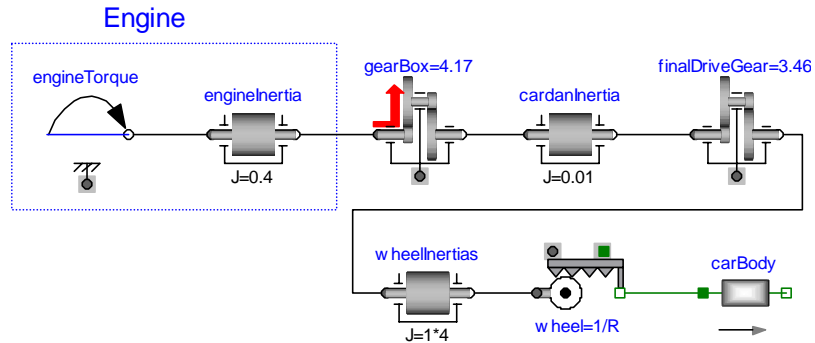
The model we are going to build is available as:

`Design.Calibration.Example.SimpleCar`

Useful modeling components are found in

`Modelica.Mechanics.Rotational`

`Modelica.Mechanics.Translational`



To the left there is the engine driving the gearbox, which is connected to the cardan system giving a final drive to the four wheels. The rotational motion of the wheels results in a translational motion of the car. Let  $R$  be the wheel radius then  $1/R$  gives the ratio between the driving rotational motion and the resulting translational motion where  $R$  is the wheel radius. The model defines

```
parameter Real R=0.34;
```

and binds the parameter `wheel.ratio = 1/R`. Setting of parameters are indicated by the diagram. Additionally the mass of the car, `carBody.m` is set to `1690+70+50` kg to include the weight of the driver and measurement equipment.

The quadratic torque characteristics at full throttle is modeled by extending from

`Modelica.Mechanics.Rotational.Interfaces.PartialSpeedDependentTorque`

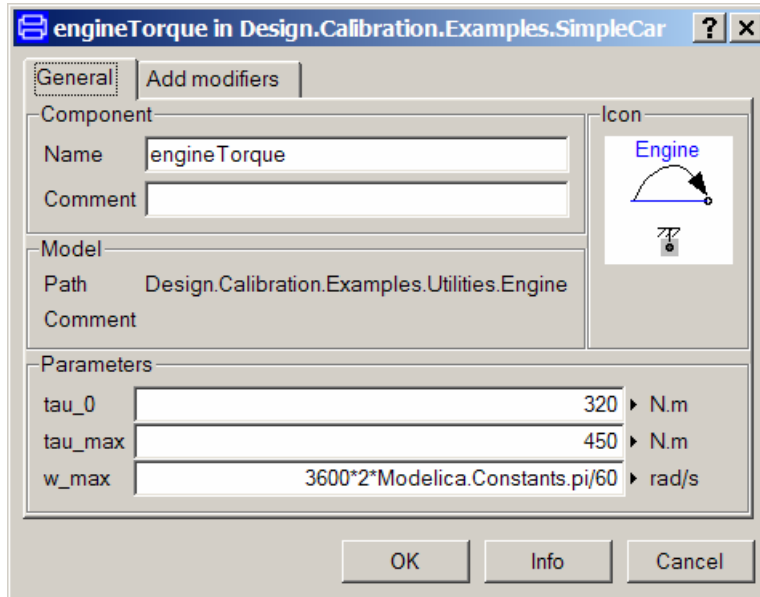
and adding the quadratic torque characteristics and the definitions of its parameters

```
model Engine
  extends
    Modelica.Mechanics.Rotational.Interfaces.PartialSpeedDependentT
    orque;
  parameter Modelica.SIunits.Torque tau_0;
  parameter Modelica.SIunits.Torque tau_max;
  parameter Modelica.SIunits.AngularVelocity w_max;
  equation
    tau = - (tau_0 + (tau_max-tau_0)*(1-((w-w_max)/w_max)^2));
end Engine;
```

Please, note minus sign for the torque to specify that the torque is a driving torque and not a reaction torque.

The parameters of the component `engineTorque` are then set as shown by its parameter dialogue

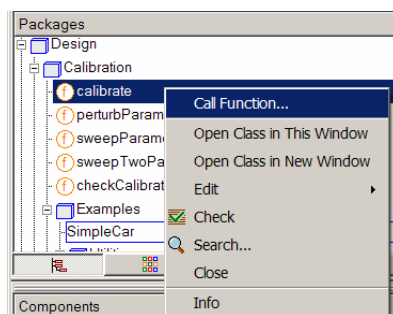




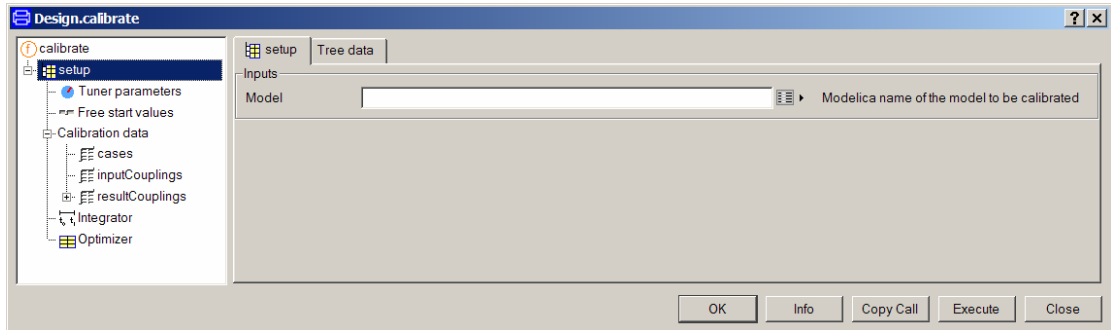
## Validation of the nominal model

Let us first check how the model with nominal parameters compares with measured data. Validation is set up very similar to calibration. A basic difference is of course that no tunable parameters need to be specified for the validation. The functions described in this document are parts of the Design package. To load it, select File/Libraries and click Design.

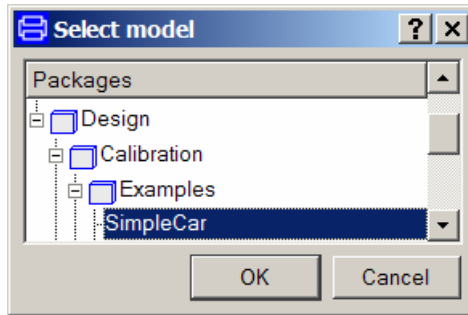
To set up the calibration, select Design.Calibration.calibrate in the package browser. Click right mouse button, select the command “Call function...”



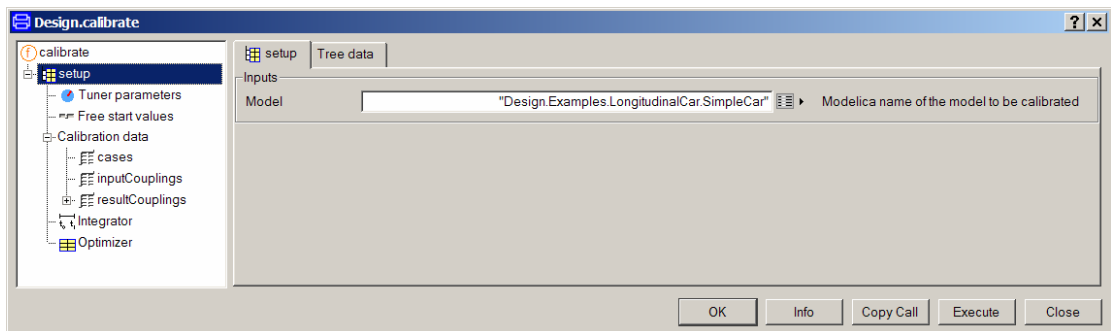
The following menu pops:



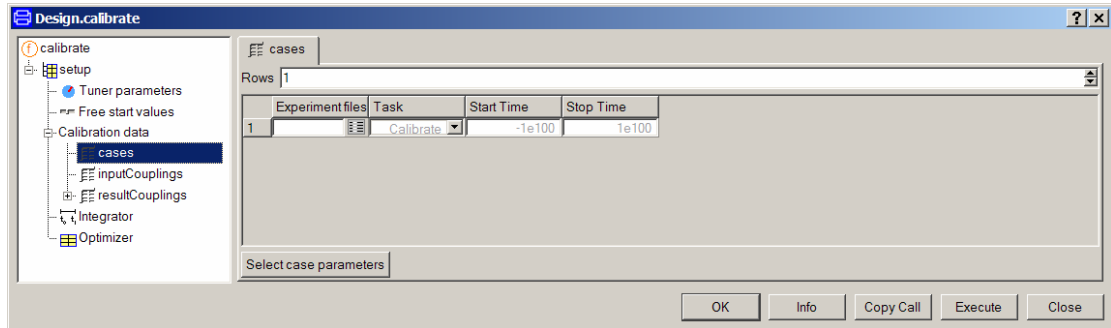
To specify the model to be calibrated, click on the Edit icon to the left of the input field. A package browser pops up. Use it to select the model.



Click OK! The model is now translated in order to gather information needed to build browsers and selectors to support the remaining setting up of the calibration task. If Dymola already has a translated model, then this model appears as the default model.

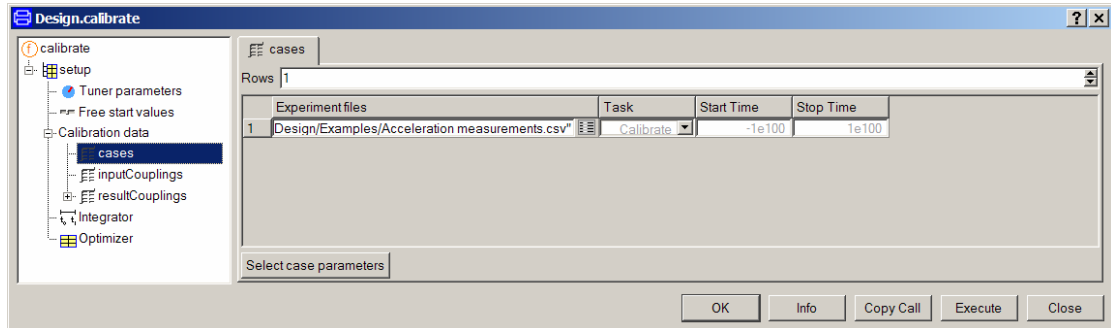


The next task is to specify the measurements and how they are stored. Consider the tree browser to the left. Select cases under Calibration data.



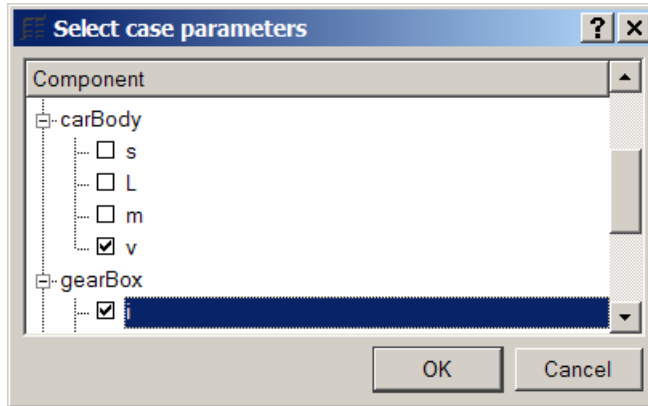
To introduce an experiment file, click on the Edit icon of the first element in the “Experiment files” column. A file browser pops up. Use it to select the file

*Program Files/Dymola/  
Modelica/Library/Design/Acceleration measurements.csv*

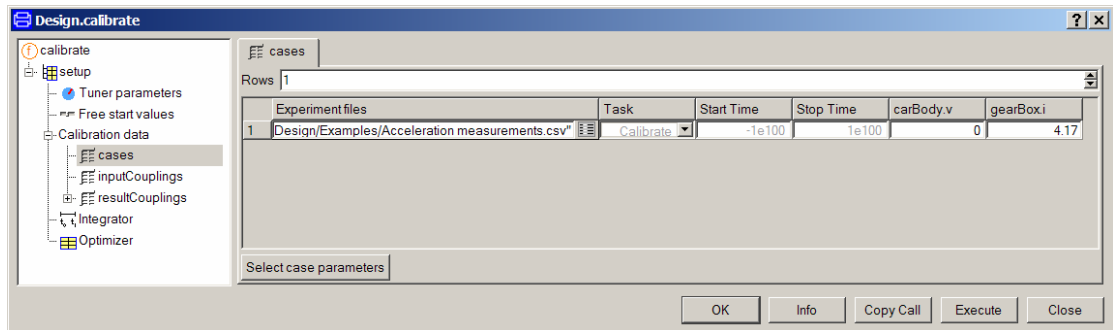


If we had had more measurement files we had increased the number of rows and selected more measurement files. In this case the measurements are stored in a csv file as described above. Dymola supports some common ways of storing measurements, see further below. An advanced user can replace the calibration data input with a routine accessing data in different formats, without having to change the underlying calibration routines.

The different cases may need individual parameter settings or individual initial values for some of the states. Recall that we are to use the measurements from the time interval 3.8-6 seconds when second gear is engaged. Thus we need to use the gear ratio of the second gear and an initial velocity. To enter this information, click on “Select case parameters”. Use the browser to select gearBox.i and carBody.v



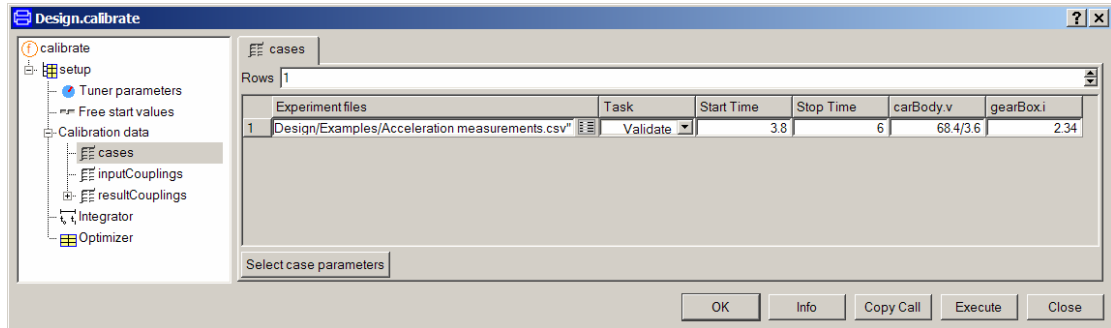
Click OK. The default values appear in the new columns.



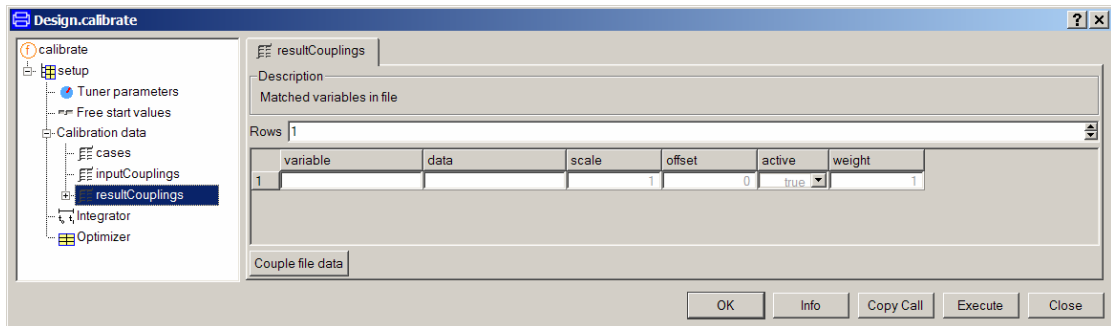
From the measurement file we can find that the velocity at time 3.8s is 68.4 km/hour = 68.4/3.6 m/s.

	A	B	C	D
1	time	speed	dist	acc
191	3.78	68.1	37.84	4.14
192	3.8	68.4	38.22	4.16
193	3.82	68.7	38.6	4.13

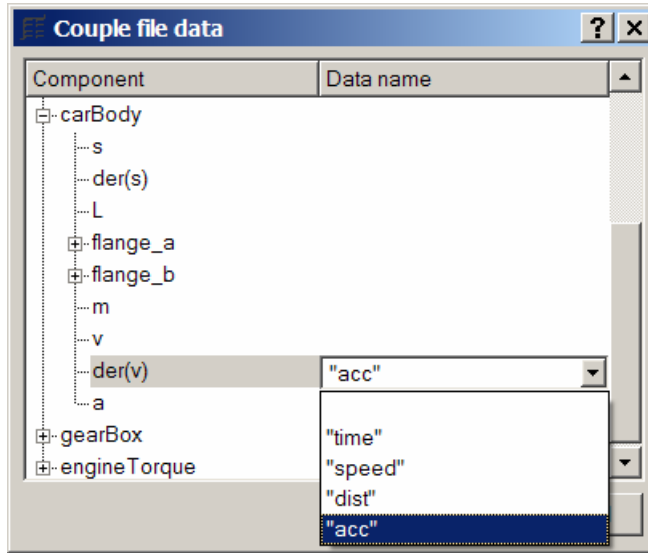
Enter this value for carBody.v and set gearBox.i to have the value of the second gear, namely 2.34. Enter also start time (3.8) and stop time (6) and set task to Validate.



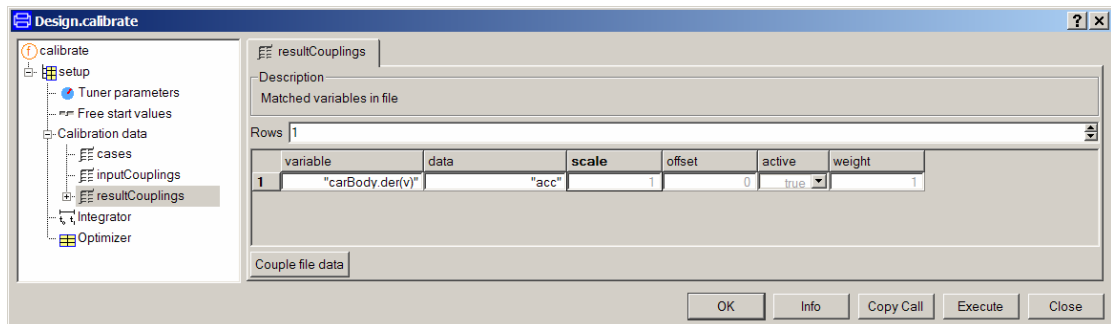
The files may include input signals to drive the model, parameter values to be used and measured data that the model shall reproduce. In this case the file includes measured speed, distance and acceleration for each 20 ms in the time interval 0-6.24 seconds. The acceleration measurements will be used for the calibration criterion. To specify that click on resultCouplings in the browser to the left



Click on “Couple file data”.



Use the browser to select the car acceleration, `carBody.der(v)`, and then click to the right to see the names of the data series in the input files. Select “acc”. We could also have chosen `carBody.a`, because `carBody.a = carBody.der(v)`. Click OK.

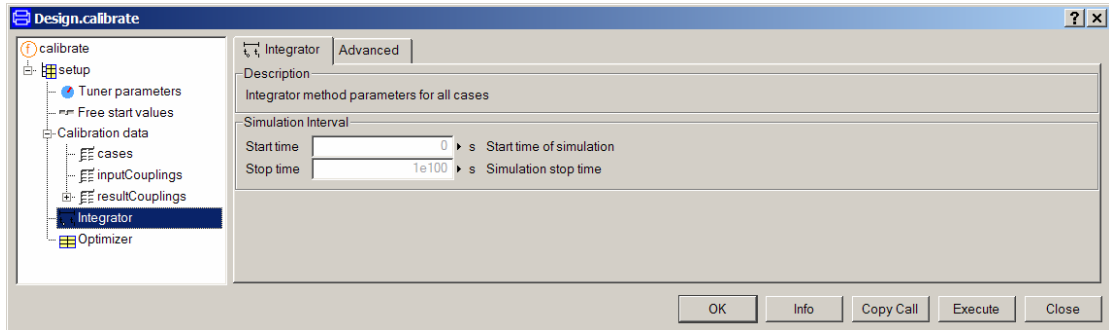


We have now specified that the difference between `carBody.der(v)` and the data column “acc” shall be used as the criterion for calibration. If the measured data are given in some unit different than that used in the model, the scale column allows scaling of the measurements:  $\text{variable} = \text{data} * \text{scale} + \text{offset}$

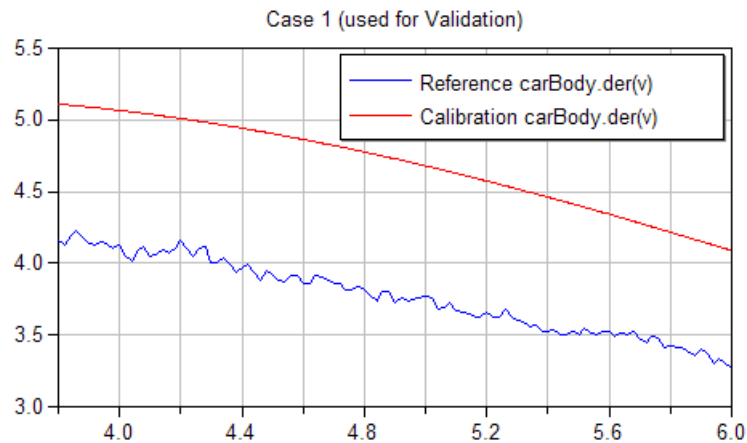
In case the deviations of several variables shall be used to specify the criterion, the weight column allows the user to give them different weights.

The model SimpleCar has no inputs. In case the model has inputs, click on “inputCouplings” and couple them to the file data in a similar way as done for the outputs.

The integrator element allows specification of a global simulation interval.



To perform the validation, click Execute.



The result is plotted above. The curves have similar shapes, but there is an offset. The model gives a higher acceleration than measured. This may make you think of losses not being modeled. In the next section we will discuss calibration.

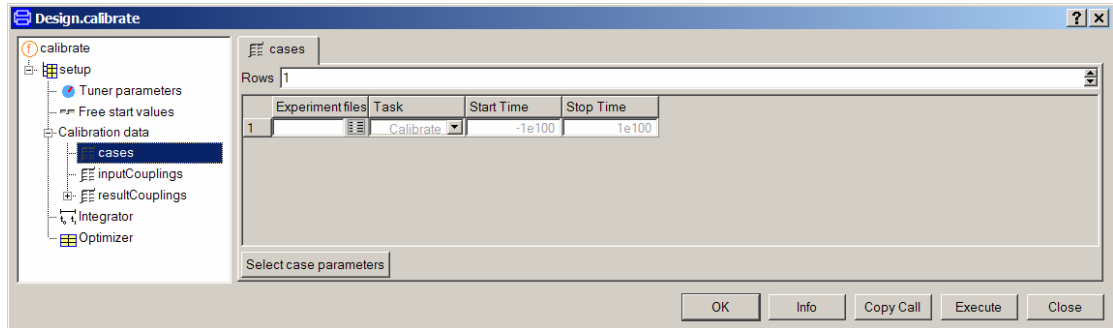
## Measurement file formats

In the example above the measurements are stored in a csv file as described. Dymola supports some common ways of storing measurements and allows users to specify their own storage formats. The measurement files must have the same format.

In case the measurement data are stored in (Matlab 4) mat files, we need to specify the name of the matrix containing the measurement data to be used and the data are referred by column number. The acceleration measurements are also available as

```
Program Files/Dymola/
  Modelica/Library/Design/Acceleration measurements.mat
```

Let us use this file instead.

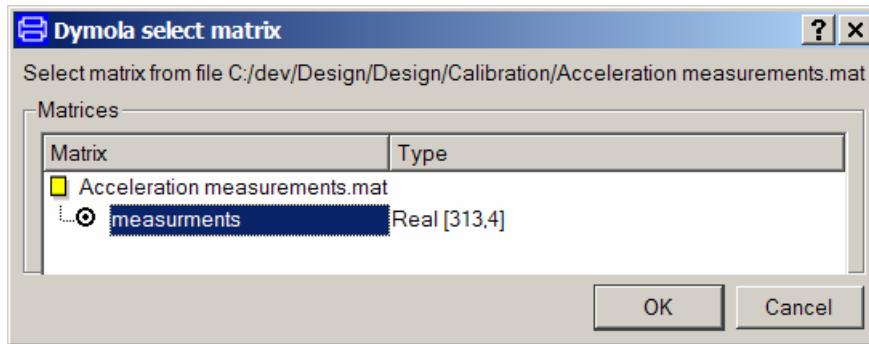


As previously, click on cases. Click on the Edit icon of the first element in the “Experiment files” column. A file browser pops up. Use it to select the file

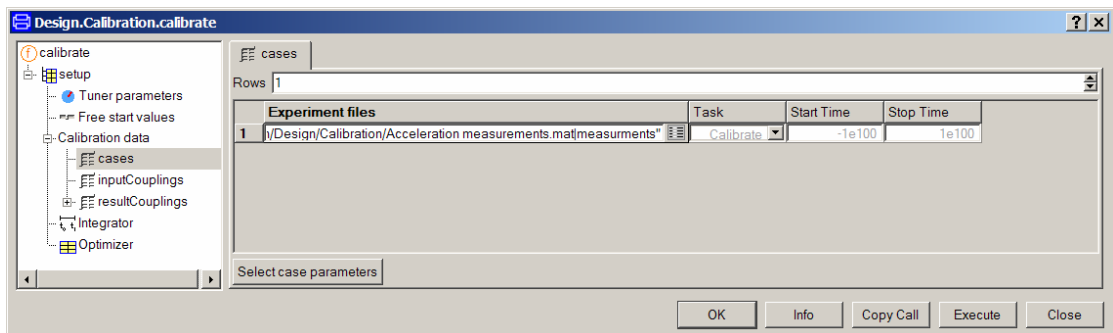
*Program Files/Dymola/*

*Modelica/Library/Design/Acceleration measurements.csv*

Dymola then pops a menu to select the appropriate matrix

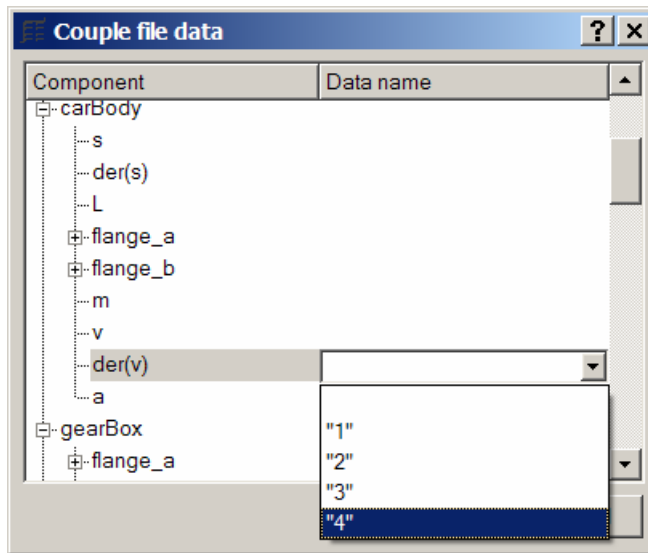


Click OK.

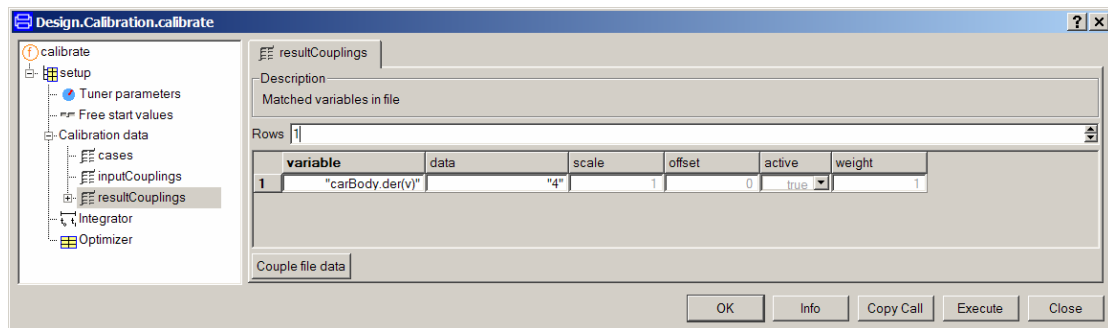




Proceed as previously to select case parameters, setting their values and start and stop time. The specification of result couplings is slightly different, because the data is referenced by column number. The acceleration measurements is column 4.



The result of the coupling now becomes



The data field has "4" instead of "acc".

The simulation results of Dymola are stored as mat files, which includes information on the name of the variables. If such trajectory files are used as measurement files then the information on variable names are used. The user will not be prompted for matrix name. When couplinging inputs or results, the browser will display variable names.

## Calibration

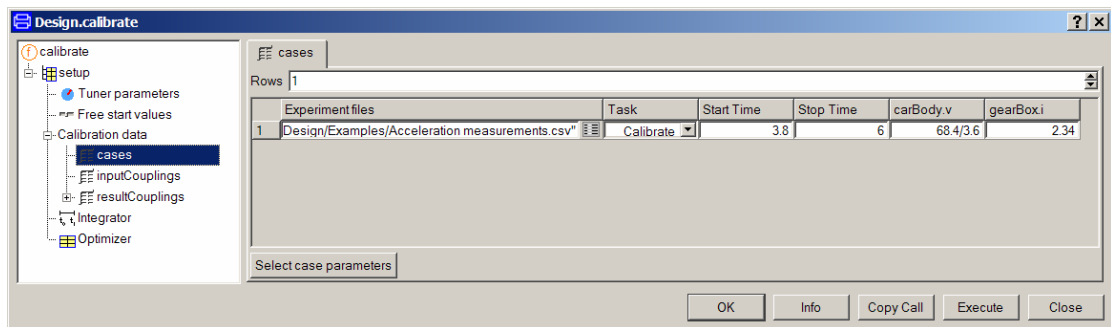
The task of a calibration is to tune some parameters to obtain a better agreement between measured behavior and behavior predicted by the model. Thus, we need to address the

question, which parameters to tune. When deciding which parameters to tune, it is good to consider the question: Which parameter values are most uncertain? In the model above, friction and losses in the gearbox elements have been neglected. Frictions and other losses are good examples where calibration is useful. There are for instance losses in both gearBox and finalDriveGear, however, having only measurements of the translational motion of the car, it is not possible to decide the individual losses of these two elements. Thus, it is necessary to aggregate all losses to one element and gearBox is selected, since it has provisions to model efficiency. The efficiency is given by gearBox.lossTable[1,2], see the documentation of Modelica.Mechanics.Rotational.LossyGear.

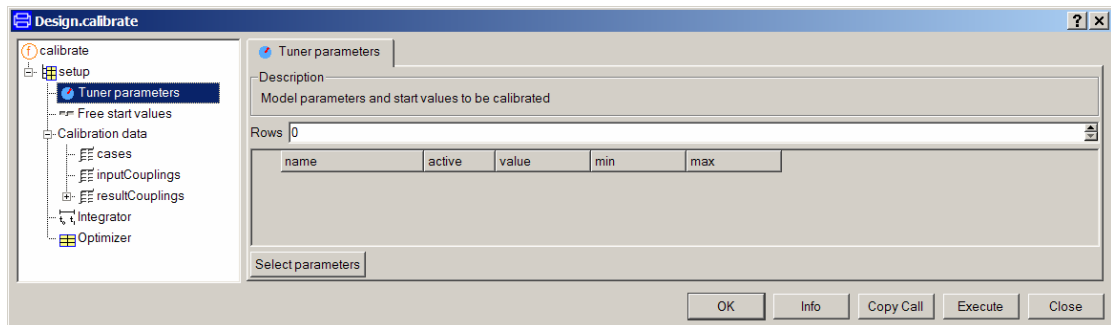
The parameter tau\_0 was manually selected to 320, so it is a good candidate for tuning.

Dymola supports an interactive explorative approach to this problem. Dymola has powerful functions to perform parameter sweeps and to analyze parameter sensitivities and possible couplings between parameters with respect to the result variables to eliminate irrelevant parameters and to diagnose over-parameterization. However, let us come back to these later and first try tuning the two parameters.

First we have to set the task to Calibrate. Click on cases in the tree browser to the left and set task to Calibrate.

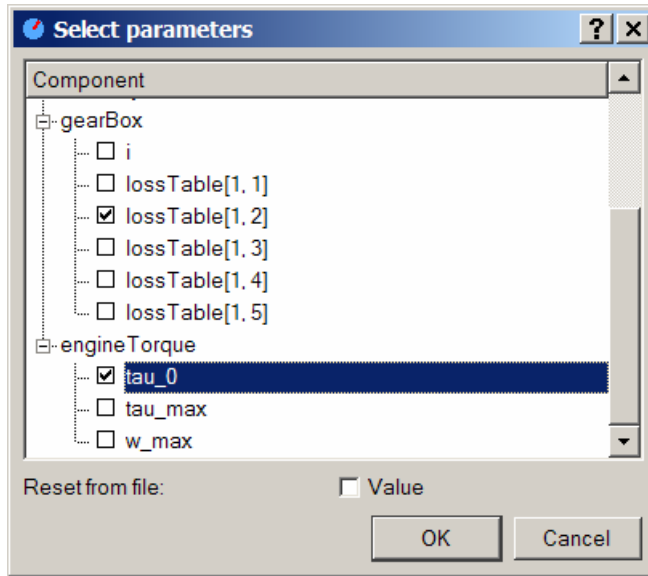


Select Tuner parameters in the left browser.

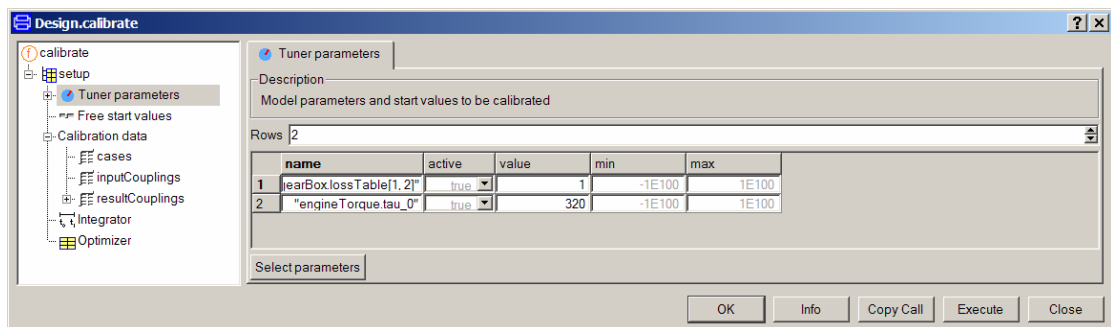


Click “Select parameters”. A menu pops. Select gearBox.lossTable[1, 2]

engineTorque.tau\_0

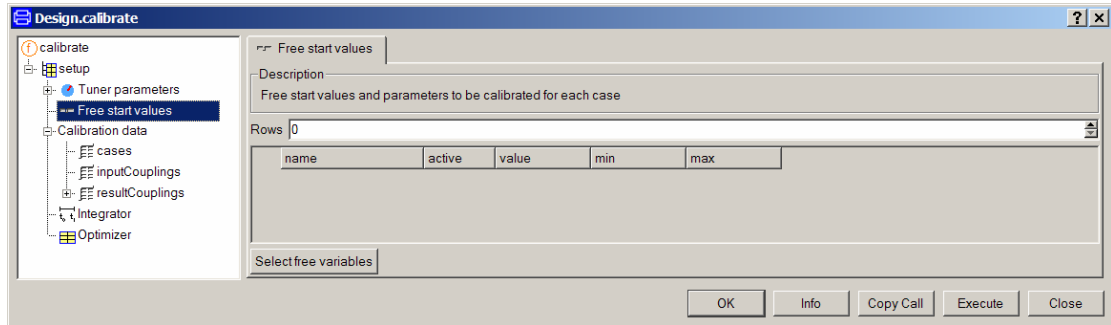


Click OK.



## Free start values

The start value of a state may be unknown. By including the state as a tuner, the start value is estimated automatically. However, in case we have several measurement series, it may be necessary to tune or estimate these initial values individually for each calibration case. Dymola supports individual tuning of parameters and start values of states and they are specified as freeStartValues.

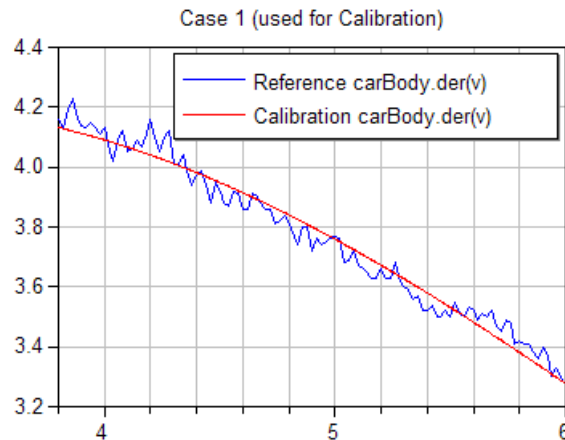


Such parameters or states are selected by clicking “Select free variables” which pops a variable selector as when selecting case parameters or tuners. The variable selector includes parameters and states. These values are also tuned for cases having Task = Validate.

## Tune the parameters

It is time to do the first calibration. Click Execute. During the calibration, results are plotted. After 25 fast iterations, we obtain the result.

```
gearBox.lossTable[1, 2]    0.794
engineTorque.tau_0.      260.7
criterion                 0.218
```



A passenger car has normally an efficiency of 0.90 at high gears in normal operation. The measurements are made at full throttle to give maximum acceleration. It means for example that the tires are slipping say 4%, which of course is increasing the losses.

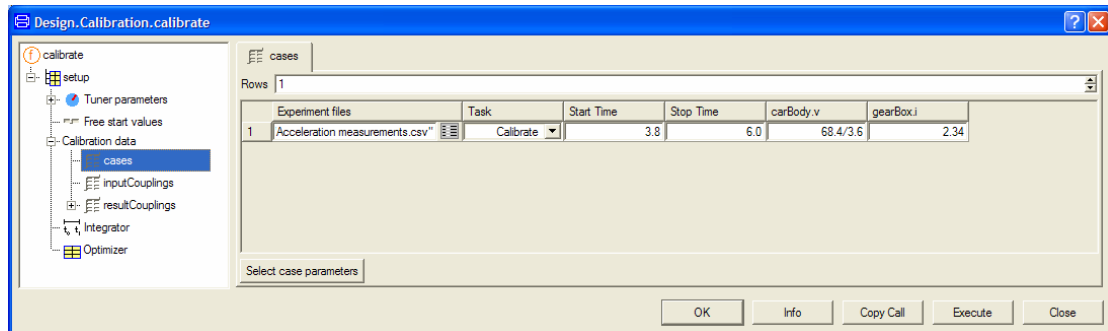
It is very easy to add new tuners. Just click “Select parameters” and select new parameters. By changing active from true to false or vice versa it is easy to experiment with different set

of tuners. Having a parameter as an inactive tuner is a good way to set a parameter to have a value different from the value given by the model.

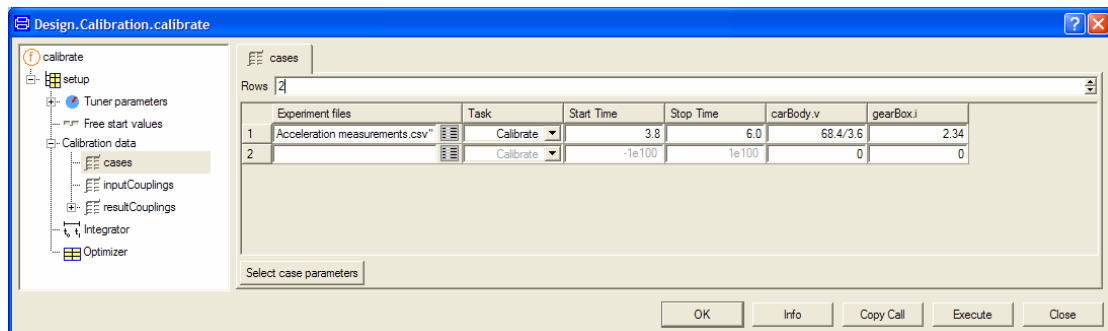
## Validation using measurements from first gear

It is recommended to validate against other measurements. Unfortunately, we do not have another measurement series in this case, but for validation we can use the data from the time interval where first gear is used.

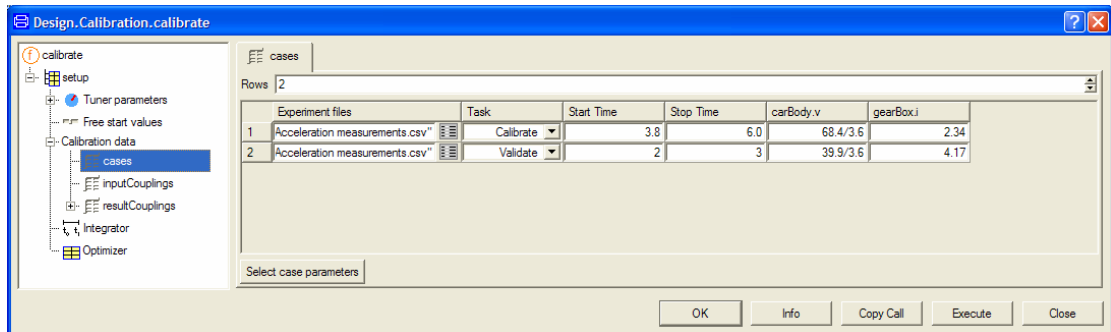
We do this by specifying another case. Click on cases.



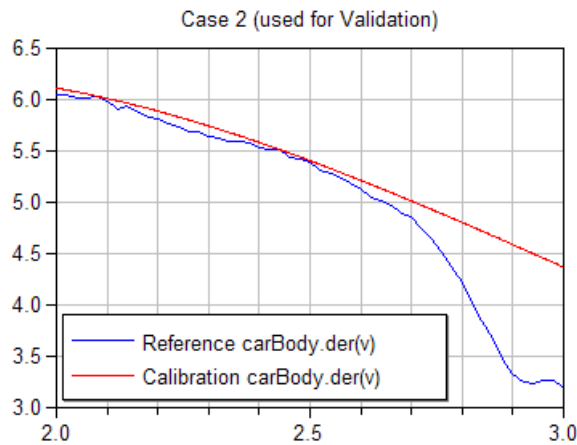
Put the cursor in the input field for Rows and hit the uparrow keyboard button once to increase the value by one. You may also use the up arrow of Rows to increase Rows to 2.



As previously, use the Edit button of Experiment files to select experiment file. You can also copy and paste the file name. Enter the values for start time (2.0), stop time (3.0), carBody.v (39.9/3.6) and gearBox.i (4.17) as illustrated below.



Click Execute! The calibration starts and gives the same results as previously, but also the plot below for the validation case (the criterion is 13.88).



The agreement for the interval 2.0-2.7 s is very good. If we rerun the validation having set the stop time of the second case to 2.7, the criterion is 0.44. As indicated above the tires are slipping when the car is run to accelerate as fast as possible. If the wheels slip too much, the anti spin control system gets active and the result is reduced acceleration after 2.7 seconds as shown by the measured data.

As illustrated, Dymola supports a flexible and incremental way of working. We need not define this total setup in one step. First we made the model and validated the nominal model against the measured data, then selected turners and calibrated. Finally we validated the calibrated model. Dymola also provides support for sentivity analysis as will be discussed below.

## The setup as Modelica code

The calibration setup is represented in Modelica in the following way. It is a function call, where nested record constructors build the needed input arguments.

```

Design.Calibration.calibrate(Design.Internal.Records.ModelCalibrationSetup(
  Model="Design.Calibration.Examples.SimpleCar",
  tunerParameters={
    Design.Internal.Records.TunerParameter(name= "gearBox.lossTable[1, 2]", Value=1),
    Design.Internal.Records.TunerParameter(name="engineTorque.tau_0", Value=320)},
  calibrationData=Design.Calibration.Internal.Dynamic_common(
    Design.Internal.Records.DynamicCommonCalibrationCases(
      experimentNames={"Acceleration measurements.csv",
        "Acceleration measurements.csv"},
      task={1,2},
      startTime={3.8,2},
      stopTime={6.0,3},
      parameterNames={"carBody.v", "gearBox.i"},
      parameterValues=[68.4/3.6,2.34; 39.9/3.6,4.17]),
  resultCouplings={Design.Internal.Records.DynamicCalibrationResultCoupling(
    variable="carBody.der(v)", data="acc"})),
  integrator=Design.Internal.Records.CalibrationIntegrator(stopTime=6.2),
  optimizer=Design.Internal.Records.Optimizer()))

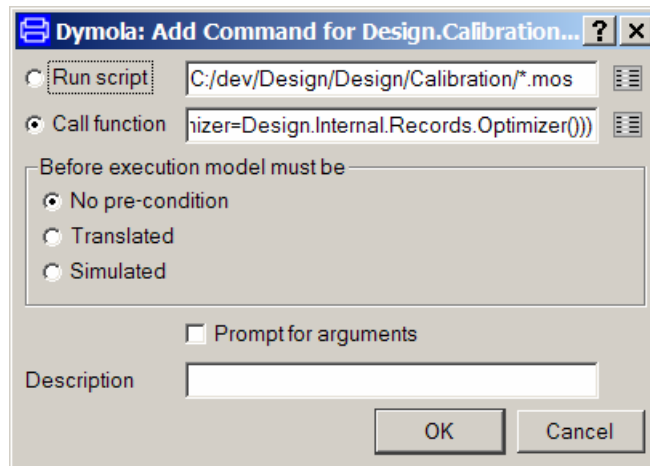
```

---

## Saving the setup for reuse

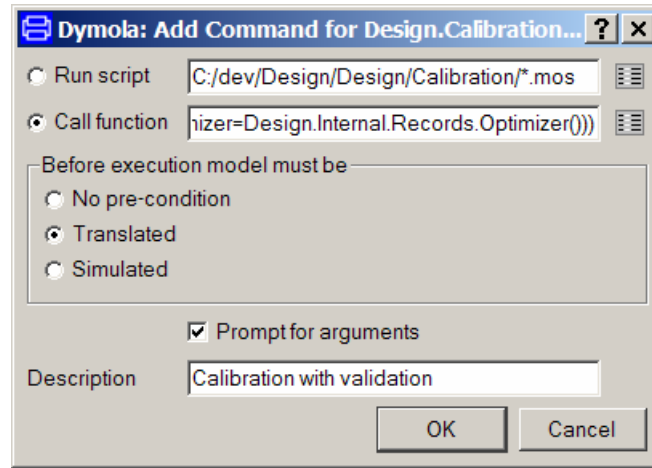
**Note: This cannot be done for SimpleCar, because it is read-only**

After an execution of a command we can save it in the model for later reuse. Select Commands/Add command. A menu pops up



Tick “Prompt for arguments” and enter a description, which will be used in the commands menu. Since the model needs to be translated in order to get the select browsers we tick that model shall be translated. This is not critical, butter only a matter of convenience. If we do not tick Translated, then when a browser need s to be popped, Dymola will give a prompt pointing out that the model needs to be translated. If we just select the command and then click Execute there will be no prompt, but function is executed as expected. The model is

translated when needed. The edit button next to the function call allows you browse or edit the function call once more.



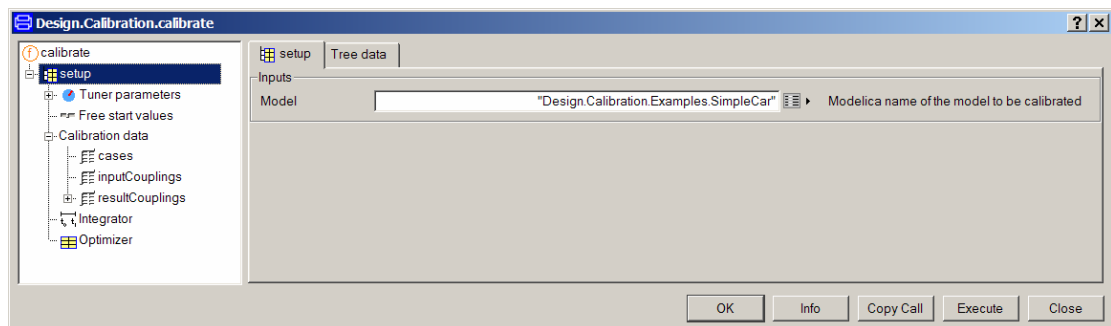
Click OK.

A function call menu as for calibrate has an Execute button. Clicking this button start an execution of the function and the menu stays popped. If we click Close, the menu is closed without any execution. If we click OK, the function is executed and the menu is closed. You click OK by mistake when you meant Execute, you can fix the situation. Click in the command input line. Press the up arrow once to scroll back in the commands given. Click right mouse button and select “Edit function Call” and the function call menu pops. This can be done for any function call in the command log.

---

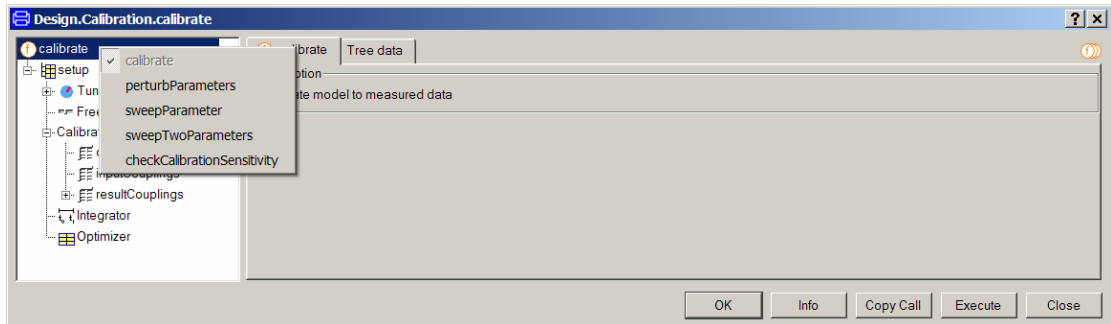
## Reusing a setup for a similar operation

A setup can be reused for a similar operation. Assume that we just have made a calibration. The menu is then





Select calibrate (at the top in the tree browser), click right mouse button to get the context menu.



The menu offers a selection of analysis and plotting functions that can exploit the calibration setup. We will describe these functions further below.

---

## Analysing parameter sensitivities and dependencies

Dymola provides a set of functions to analyze parameter sensitivities and dependencies. Below the functions `perturbParameters`, `sweepParameter`, `sweepTwoParameters` and `checkCalibrationSensitivity` will be described.

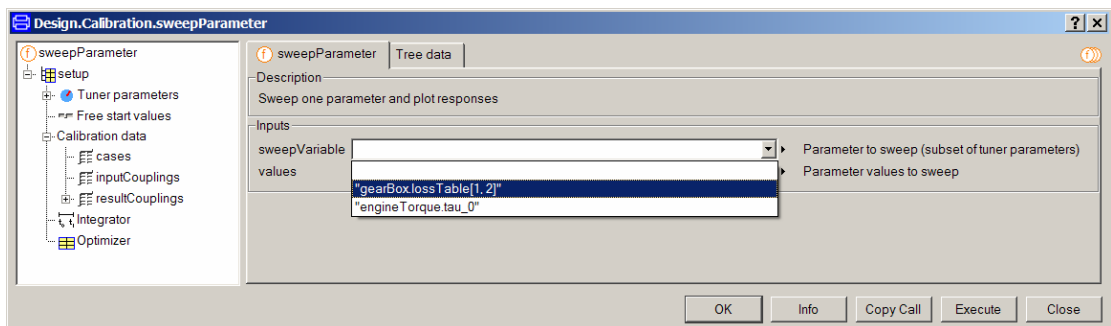
### Sweep one parameter – `sweepParameter`

The function `sweepParameter` sweeps a tuner and plots the responses.

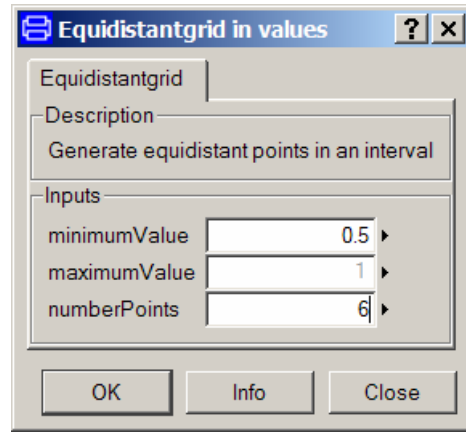
As an example select the command “Calibration with validation” of the model

`Design.Calibration.Examples.SimpleCar`

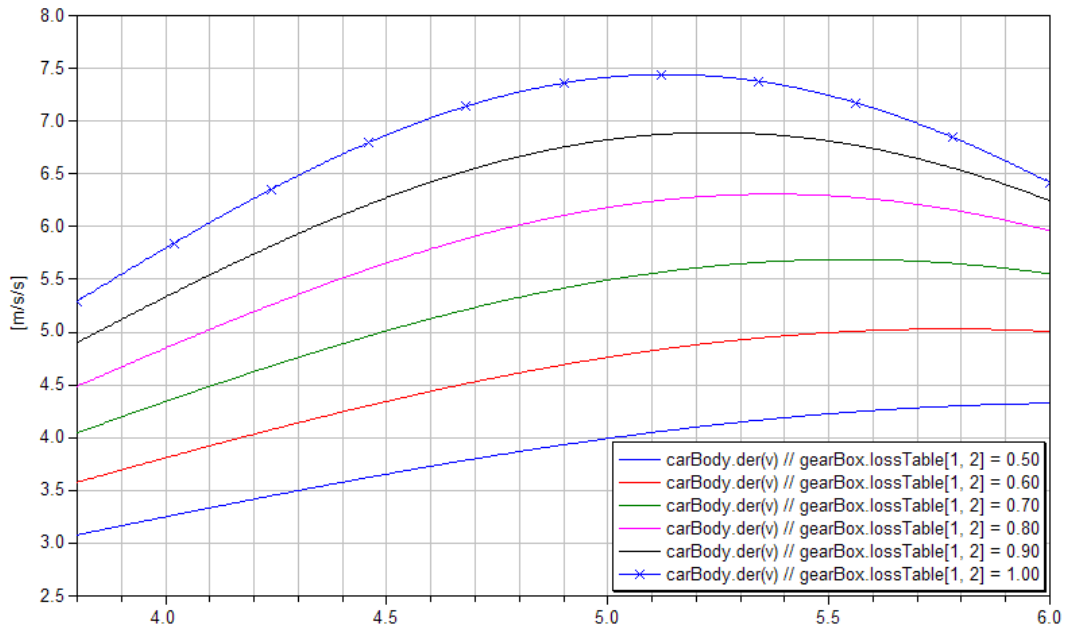
Select `calibrate` (at the top in the tree browser), click right mouse button and select `sweepParameter`. The menu changes since additional parameters needs to be provided.



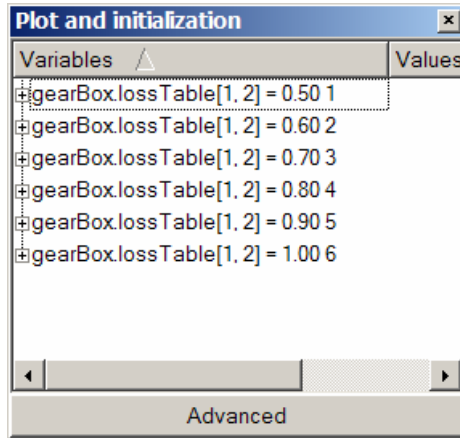
Select gearBox.lossTable[1,2] in the combobox of sweepVariable and select Equidistant grid for values



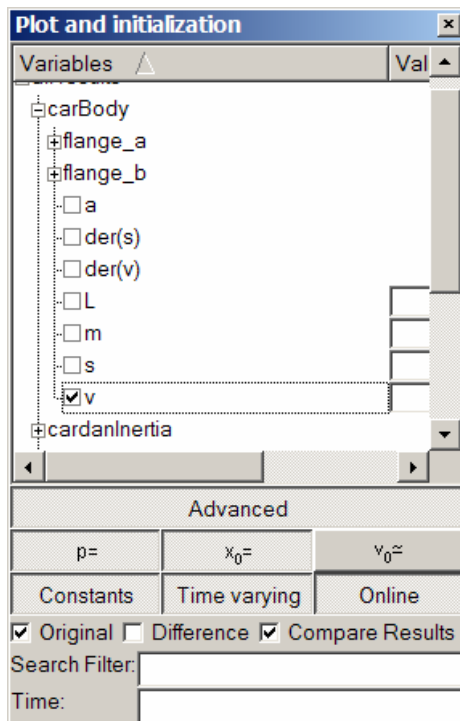
Enter minimum and maximum values and number of points and click OK. Click Execute. The result is plots of the result variables, which in this case is the acceleration. As expected, higher efficiency gives higher acceleration.



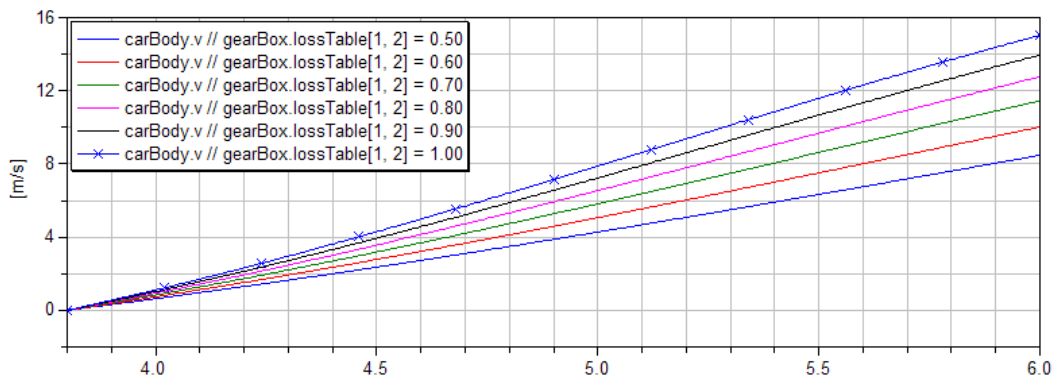
There is also a corresponding plot for the validation case. Moreover, all results of the simulations are available for access in the plot browser



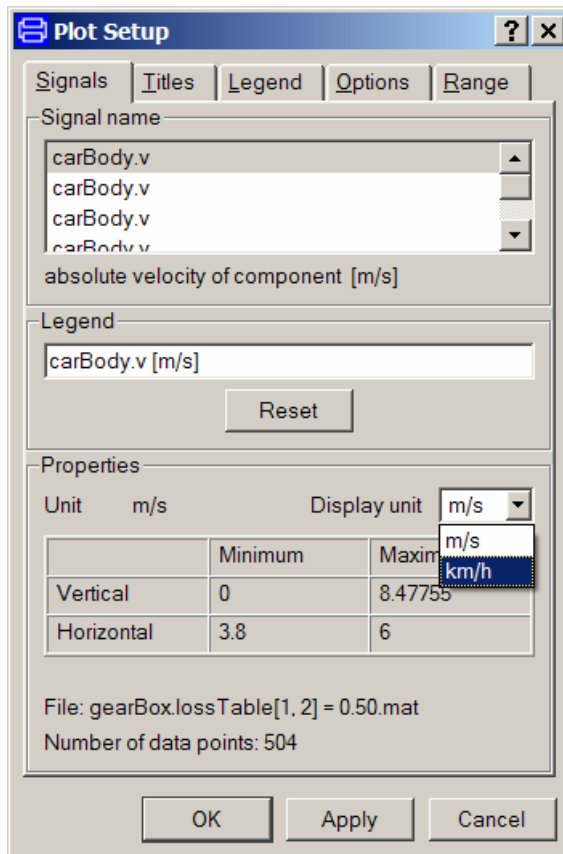
It means that all Dymola's plotting facilities can be used to produce other plots from the sweep. It is for example easy to get a similar plot with the velocity of the car. Click on Advanced and tick Compare results. Then select carBody.v



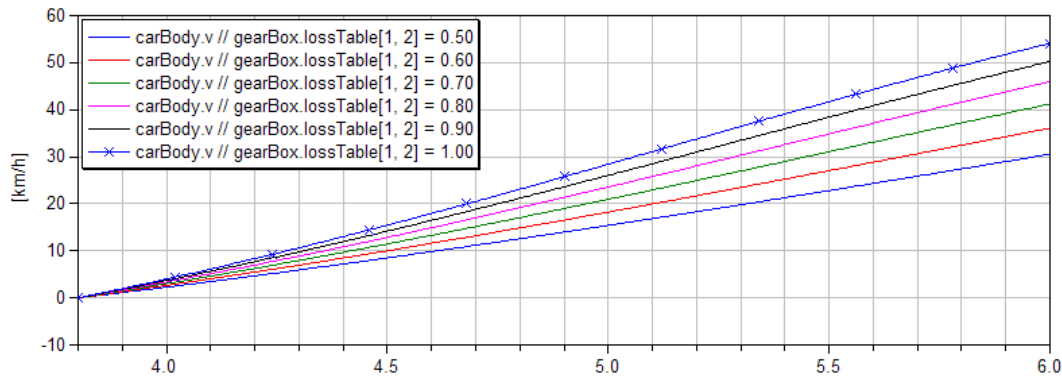
and the plot becomes



The velocities are in m/s, but it is easy to get them in km/h. Select Plot/Setup



Select Display to km/h. The plot becomes now



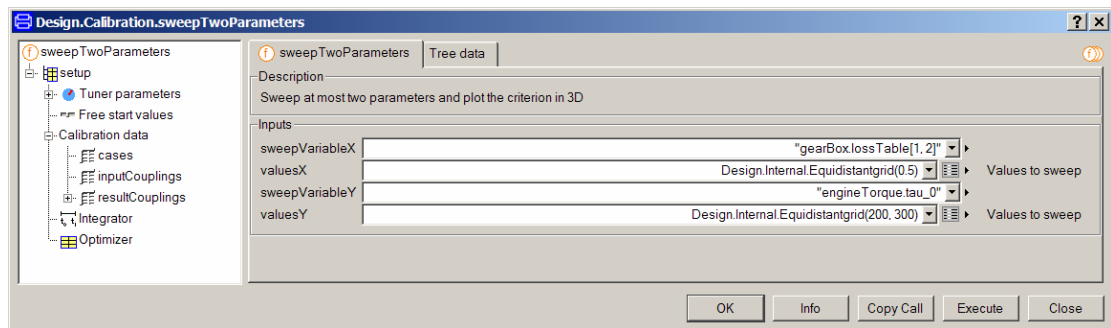
## Sweep two parameters – sweepTwoParameters

The function `sweepTwoParameters` sweeps two tuners and produces a 3D plot of the criterion.

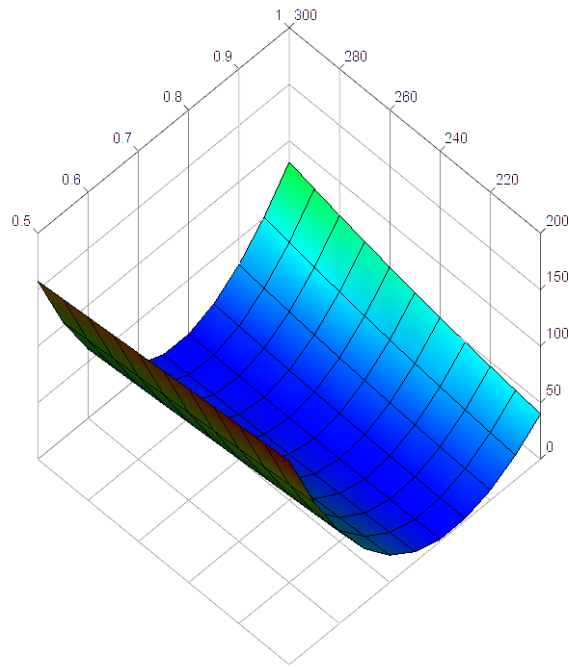
As an example select the command “Calibration with validation” of the model

`Design.Calibration.Examples.SimpleCar`

Select `calibrate` (at the top in the tree browser), click right mouse button and select `sweepTwoParameters`. The menu changes. Since we have just two tuners, we select the efficiency as `sweepVariableX` and 11 values in the interval 0.5-1. We select `tau_0` as `sweepVariableY` and 11 values in the interval 200-300.



Click `Execute` and Dymola produces the plot below



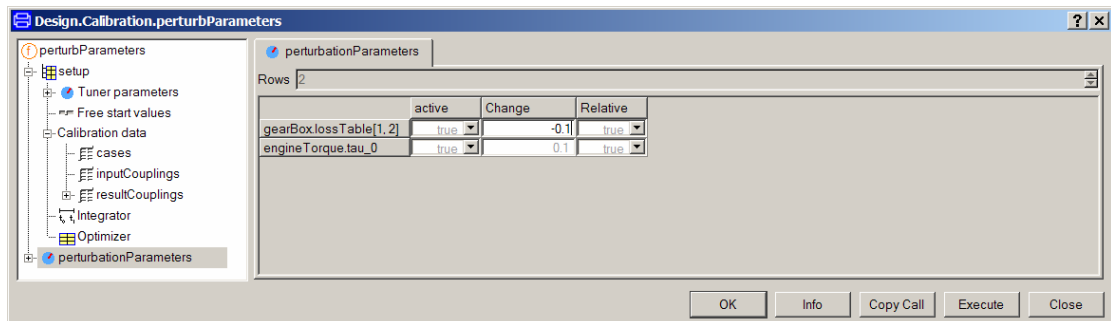
## Response to parameter perturbations - perturbParameters

The function perturbParameters plots the responses to perturbations in the tuners.

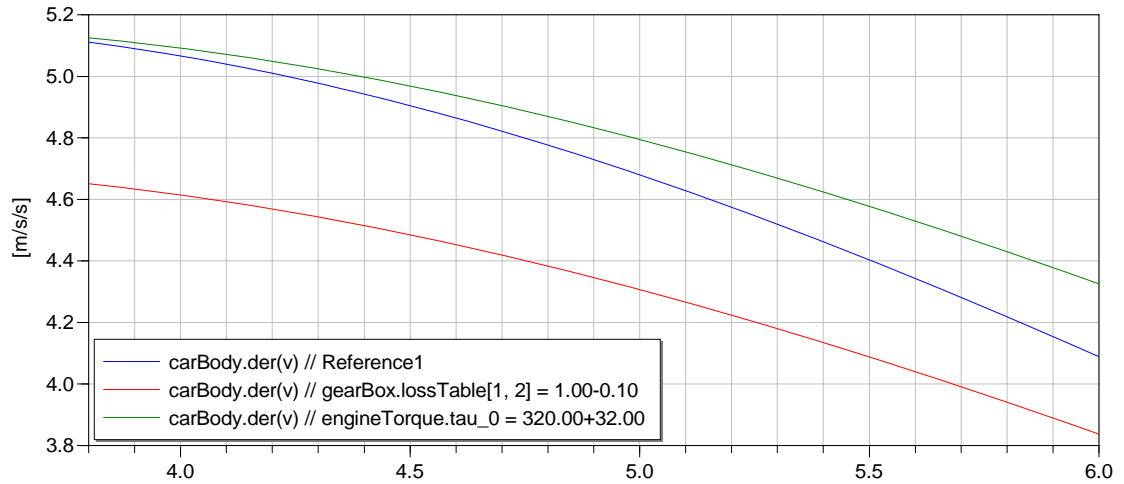
As an example select the command “Calibration with validation” of the model

Design.Calibration.Examples.SimpleCar

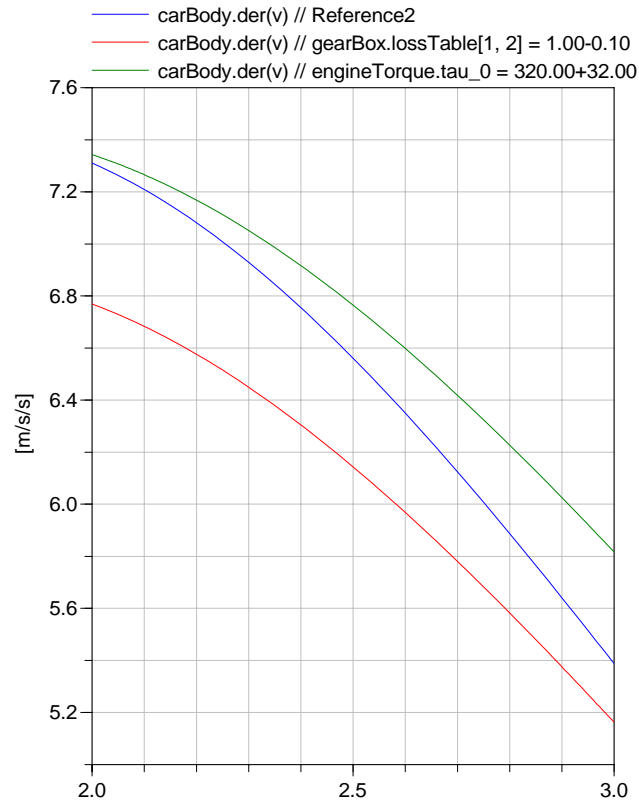
Select calibrate (at the top in the tree browser), click right mouse button and select perturbParameters. The menu changes. Click on perturbationParameters



The function exploits the setup of calibration, but needs some additional input, `perturbationParameters`, which by default are the same as the tuner parameters of setup. When executing the function it perturbs the parameters in turn. The default perturbation is 10%. Note that the efficiency has a nominal value of 1 meaning a default perturbation to 1.1, which is not a physical value. Thus, we change it to  $-10\%$  to get an efficiency of 0.9. The results are plots of the result variables as shown below.



Both tuners influence the acceleration. The responses for the validation case are also plotted



## Check if tuners can be calibrated – checkCalibrationSensitivity

When tuning parameters from measurements, a basic question is “Which parameters can be estimated from the measurements available?” Changing a parameter to be estimated must of course influence the output. However, this is not enough. Two or several parameters may influence the result in a similar way such that it is not possible to estimate them individually.

Assume that our nominal model is the correct model and we had used it to produce a “measurement” file. If we make small perturbations of the values of some parameters and use the “measurement” file for calibration, we would like the result of the calibration procedure to be that the perturbed parameters are tuned to their original values. The function `checkCalibrationSensitivity` checks if this is the case. If not, it lists tuners that do not influence the criterion and linear combinations of parameters where changes of the appearing parameters do not influence the criterion, if the linear expression remains constant.

As an example select the command “Calibration with validation” of the model

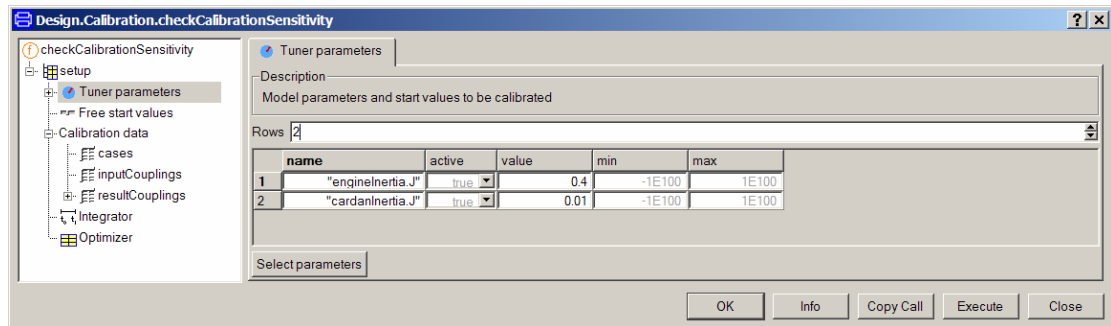
```
Design.Calibration.Examples.SimpleCar
```



Select calibrate (at the top in the tree browser), click right mouse button and select checkCalibrationSensitivity. Click Execute! Dymola outputs a positive message

The calibration criteria are sensitive for small variations around the nominal values in all tuner parameters and in all their linear combinations.

Let us try some other tuners. Can we tune the engine and cardan inertia? Select them as tuners.



Click Execute! Dymola outputs the message

The calibration criteria are insensitive for small variations around the nominal values in the following linear parameter combinations:

$$-\text{engineInertia.J} - 0.1826 * \text{cardanInertia.J}$$

The message says that if we change the two values, but keep

$$-\text{engineInertia.J} - 0.1826 * \text{cardanInertia.J}$$

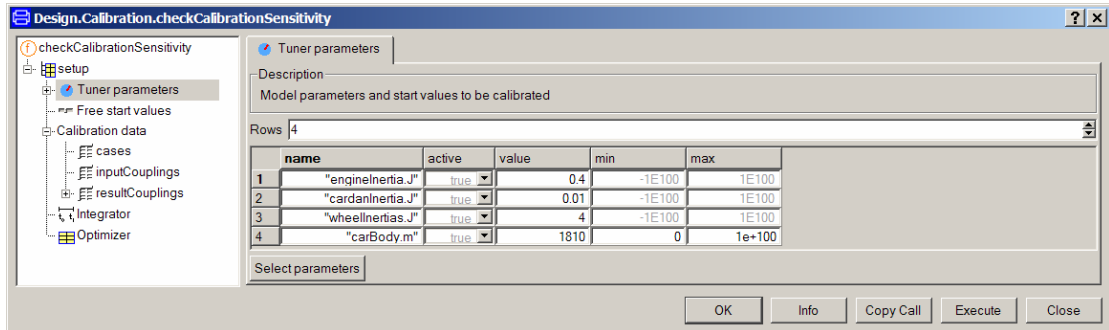
constant, then the criterion does not change. In other words we cannot tune the inertias individually, but we can tune the combination given. The engine and the cardan are rigidly coupled. It means that the inertia for those two bodies sensed from the engine is

$$J_e + J_c / i^2$$

where  $J_e$  is the inertia of the engine and  $J_c$  is the inertia of the cardan and  $i$  is the gear ratio. Using  $i = 2.34$ , we get

$$J_e + J_c / i^2 = J_e + 0.1826 J_c$$

This is consistent with what Dymola told us. In fact the engine, cardan, wheels and the car body are rigidly connected. It means that we can only estimate a total inertia for example reduced to the engine side or a mass equivalent reduced to car body. Let us specify the inertias and the car mass as tuners.



Clicking Execute gives the expected answer

The calibration criteria are insensitive for small variations around the nominal values in the following linear parameter combinations:

$$-\text{engineInertia.J} - 0.1826 * \text{cardanInertia.J} - 0.0153 * \text{wheelInertias.J} - 0.0018 * \text{carBody.m}$$

If we multiply by  $-1$ , this is the total inertia reduced to the engine side.

# **Design optimization**



# Design optimization

---

## Introduction

Dymola includes features to perform integrated computer experiments with Modelica models. This document describes the features to determine improved values of model parameters by multi-criteria optimization based on simulation runs. The functions and models described in this document are parts of the Design.Optimization package. The Optimization option is required. However, the optimization examples given below can be run without the Optimization option.

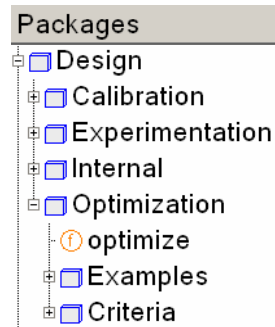
Consider a Modelica model describing a technical system that shall be improved. Such a model includes typically many parameters that can still be changed, for example the spring constants of a car, the gear ratio of a gear box, or parameters of a controller. Some parameters might be determined by using heuristic design rules, by adjusting them by “trial and error” using simulation runs or by using simplified linear models and apply the well established synthesis procedures for linear systems.

Design optimization is an approach to tune parameters such that the system behavior is improved. The parameters that are tuned are often referred to as tuners. Mathematically, the tuning procedure is formulated as multi-criteria parameter optimization: Parameters are calculated to minimize criteria which express in mathematical terms what “improvement” shall mean. Criteria values are usually derived from simulation results, e.g., the overshoot or rise time of a response, but they can also be derived by other analysis procedures, such as frequency responses or eigenvalue analysis.

The typical setup described below consists in defining the most important operating points of a model, and to define criteria for every operating point. This means that usually several simulation runs are needed to compute the criteria values. This setup is called multi-criteria, multi-case optimization. The different operating points are the “cases” under consideration. The major goal is to minimize all criteria and/or to keep them below required bounds. Other types of demands, e.g., criteria that shall be maximized, have to be reformulated.

Since several criteria shall be minimized there is usually no unique mathematical solution. Instead, the criteria have to be weighted with respect to each other and the goal is to find the best compromise solution that minimizes all criteria in the “designer’s sense”. The “weighting” technique described in the next sections is a proven technology developed by DLR and it has been applied in many industrial projects in the last 10 years.

To load Design.Optimization, select File/Libraries and click Design.



The function `Design.Optimization.optimize` is the main function for design optimization via multi-criteria, multi-case parameter optimization. There is also a set of functions and of models to define criteria. For parameter studies in general, see `Design.Experimentation`. To determine model parameters using measurement data, see `Design.Calibration`.

This document uses the design of a control system for a very simple model of an F14 aircraft (see figure below) to illustrate how a basic design optimization task is set up and executed, and how the setup is stored for later reuse.

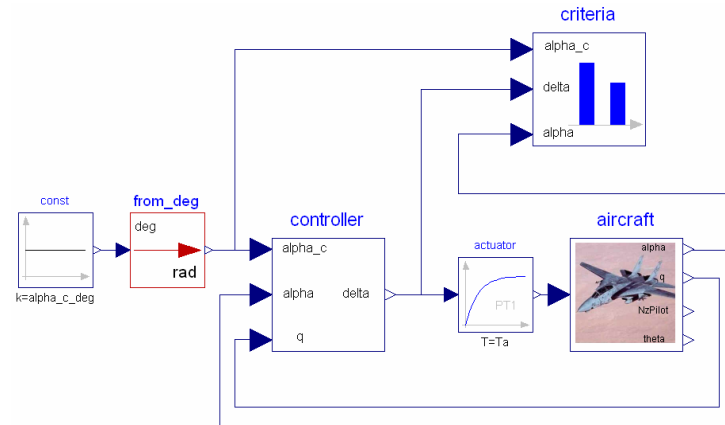


---

## First optimization setup

In this section the first setup of the design optimization of the F14 controllers is shown.

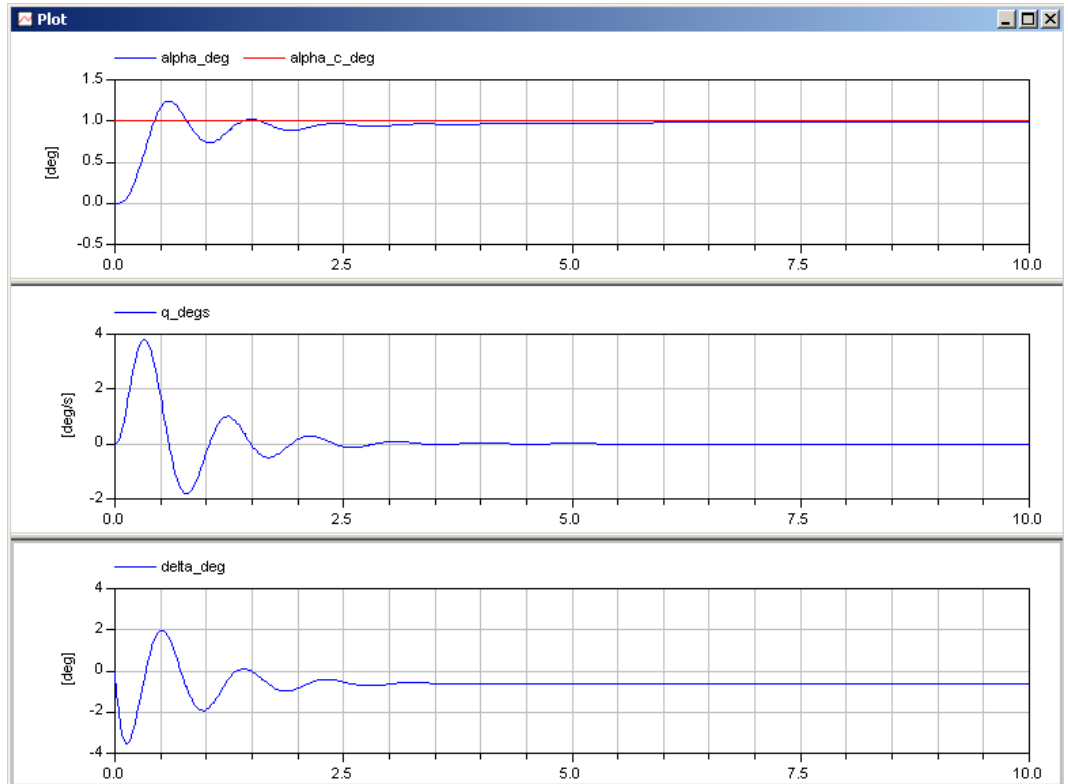
Open model Design.Optimization.Examples.ControllerDesign\_F14.



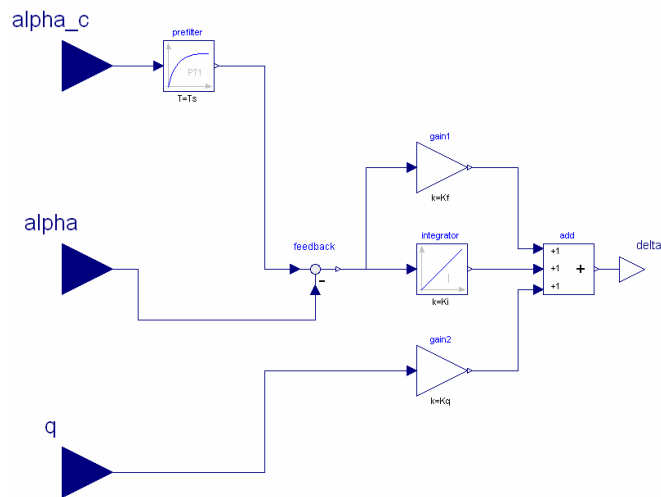
Component “aircraft” contains the dynamic equations of the aircraft. Component “controller” is the control system for the longitudinal motion, and component “criteria” contains the criteria computation.

This model is used for simulation and analysis of the closed loop step response of the longitudinal motion of a very simple F14 aircraft model. A linear controller with fixed controller parameters is used for tracking the reference motion of the angle of attack,  $\alpha$ . The goal is to determine the controller parameters such that the step response is reasonable in the operation region of the aircraft.

Simulate this model for 10 s and plot  $\text{alpha\_c\_deg}$  (= commanded angle of attack),  $\text{alpha\_deg}$  (angle of attack),  $q\_deg$  (pitch rate) and  $\text{delta\_deg}$  (elevator deflection):



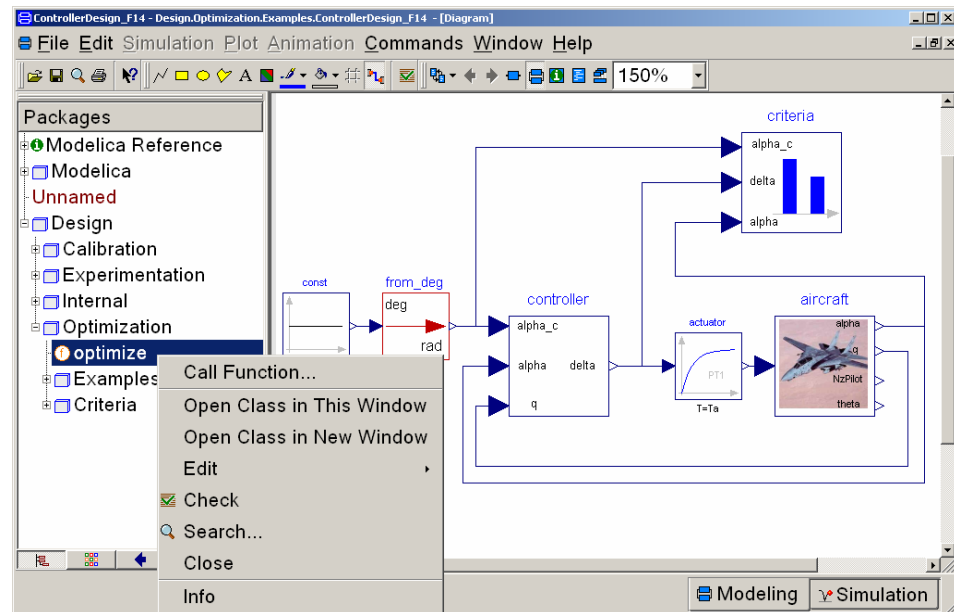
The controller is shown in the next figure:



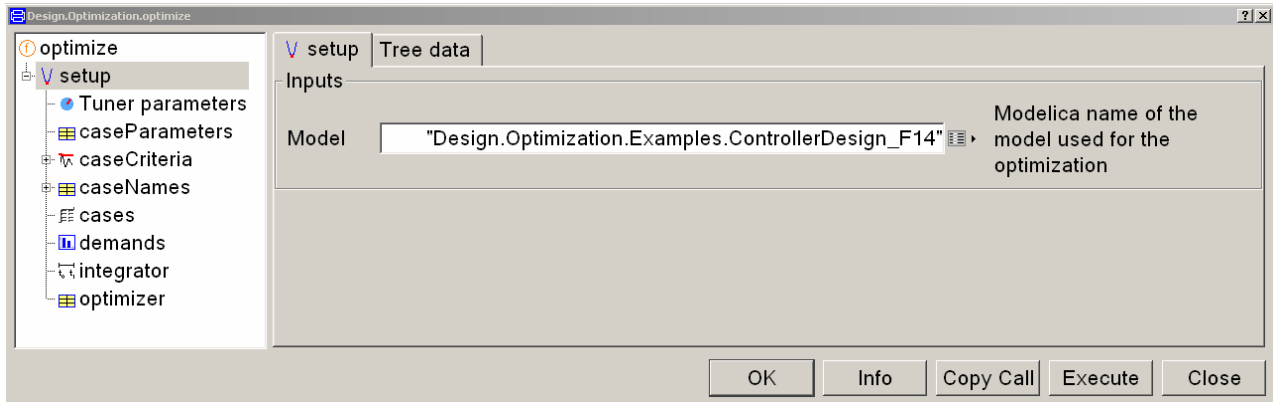


The values of the controller parameters are  $K_f = -6$ ,  $K_i = -2$ ,  $K_q = 0.5$ . The desired reference value for alpha is  $\alpha_{c\_deg} = 1^\circ$ . The initial value for  $\alpha(t)$  is  $\alpha(0) = 0$ . This arbitrarily chosen stabilizing set of controller parameters leads to a large overshoot of alpha and a significant maximum elevator deflection. The design objectives will be to reduce overshoot of alpha below 1 % and to reduce the maximum elevator deflection below  $2^\circ$ .

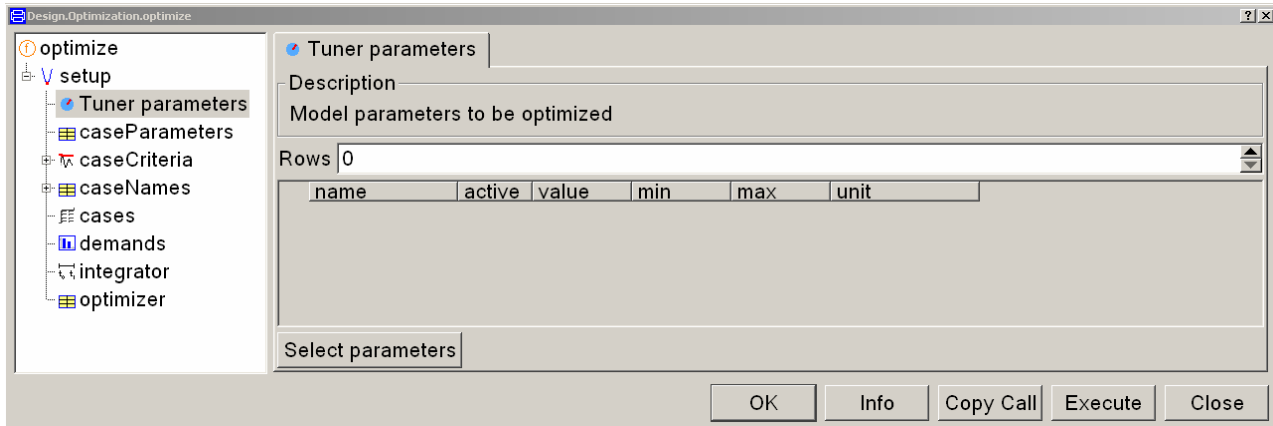
The design problem is now translated into a setup for optimization based parameter tuning. In the “Commands” menu in the toolbar you can find all setups described in this tutorial. We will perform now the setups manually. Right click on function “optimize” in the package browser and select “Call Function ...”:



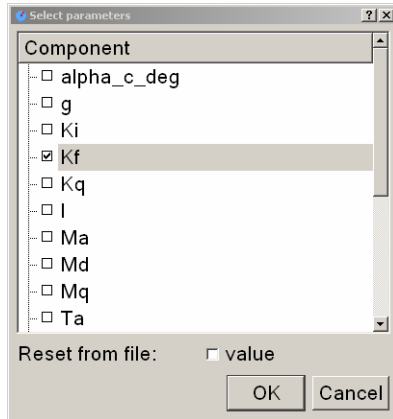
In the appearing dialog window, the model name of the last translated model is automatically inserted (there is also a browser for selecting the model):



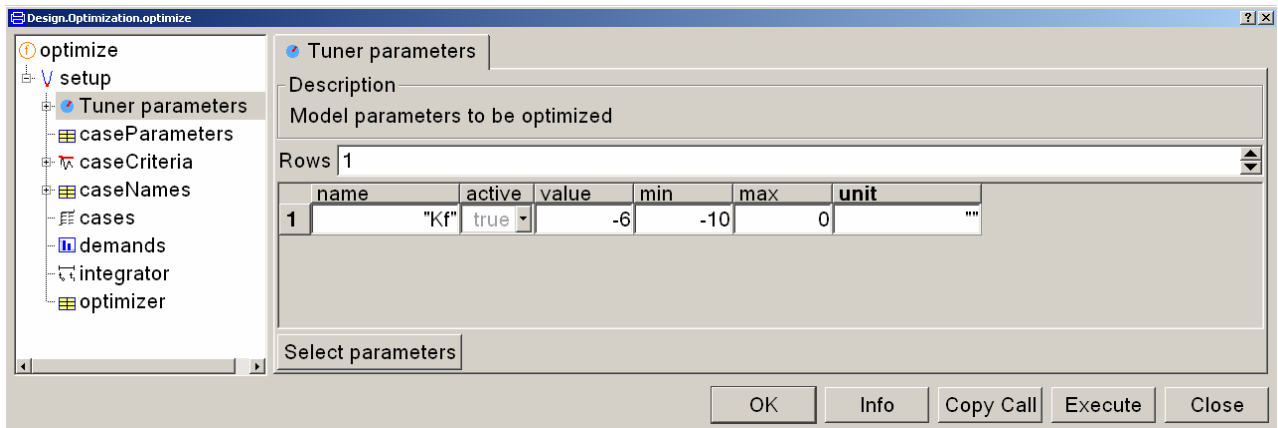
Select “Tuner parameters” in order to define the model parameters that shall be determined by the optimizer.



By clicking on the “Select parameters” button a variable tree browser of the selected models opens. Select the controller parameter Kf as a tuner that is being optimized:

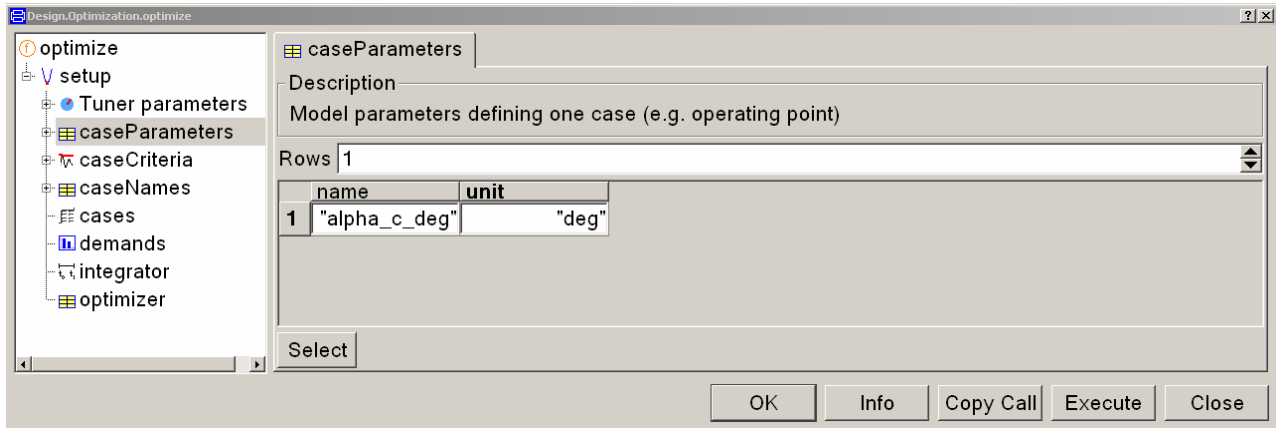


The actual tuner value and the corresponding minimum and maximum values as well as the unit (if defined in the model) are read from the last simulation run

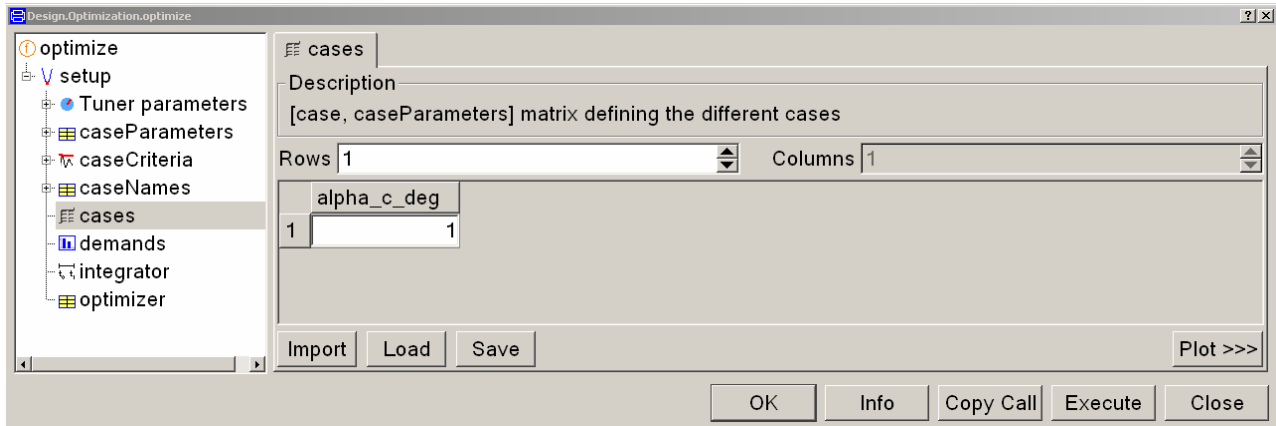


Note, minimum and maximum values should always be defined for tuner parameters in order to ease the task for the optimizer.

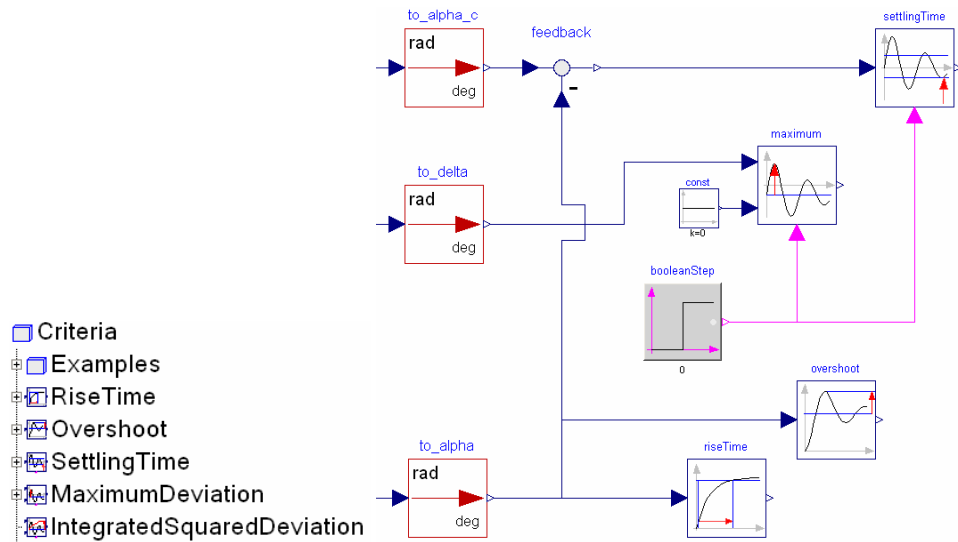
Optionally, case parameters can be specified to define the “operating conditions”. Here, parameter “alpha\_c\_deg” is selected from the tree browser via button “Select parameters”:



The value of “alpha\_c\_deg = 1” of this operating conditions has to be given under “cases”:



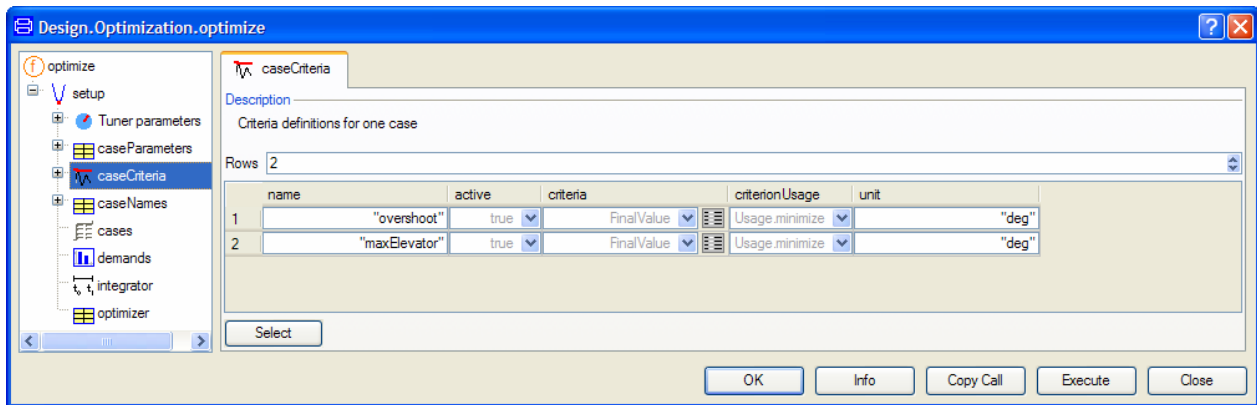
In the model ControllerDesign\_F14, criteria blocks from the Design.Criteria sublibrary are used to compute how well the controller works (see next figure). In the F14 model, for example the criteria block “Criteria.MaximumDeviation” is used with the component name “maximum” (see figure below). Since in the F14 model the “maximum” block is in a block called “criteria”, and the criterion is always the output y from a criteria block, the criterion of the maximum deviation is accessed as “criteria.maximum.y”.



In order to access this variable a bit easier, in the top level text layer of the F14 example an alias variable “maxElevator” is defined as:

```
output Real maxElevator(unit="deg") = criteria.maximum.y;
```

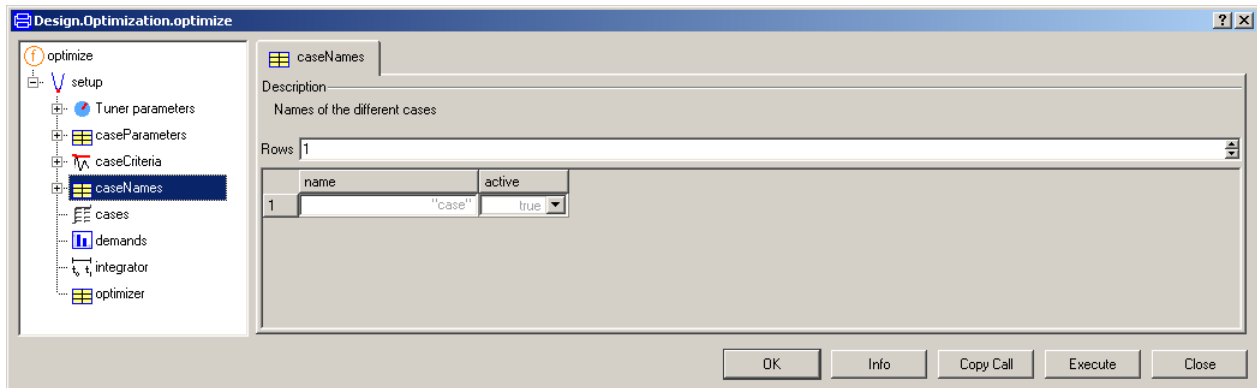
In other words, “maxElevator” is the maximum deviation of the elevator signal from zero. In the variable tree browser of “caseCriteria” the used criteria might be defined by selecting again variables (here: “overshoot” and “maxElevator”):



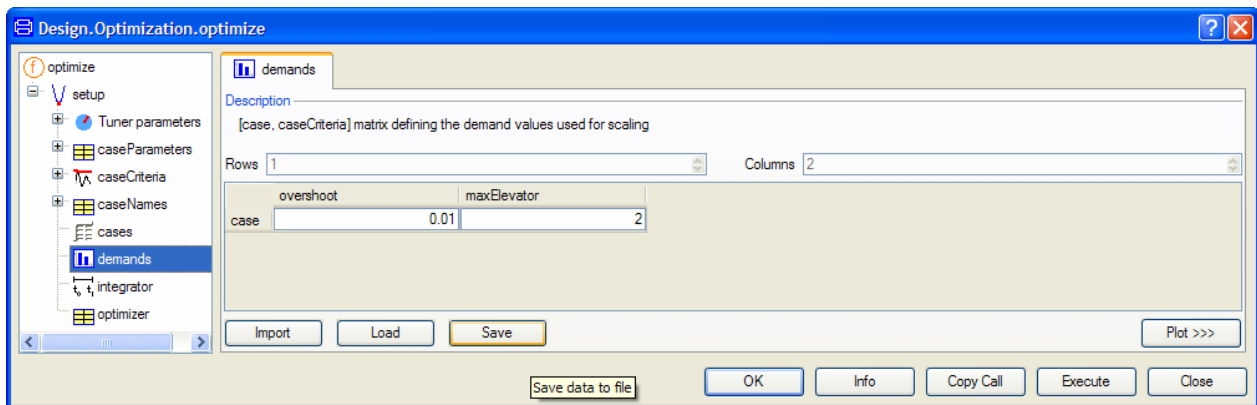
The default value in column “criteria” of the above menu is “FinalValue”, i.e., the final value of a variable in the model is used as criterion. Alternatively, this pop up menu also allows selecting other criteria that are not defined in the model but are deduced from simulation results. In some cases this is more convenient. Criteria based on linearization of the model around an operating point (e.g., maximum real part of all eigen values) can only

be selected from this menu and cannot be defined in the model (this function criterion is not yet supported). Column “criteriaUsage” remains unchanged for the moment.

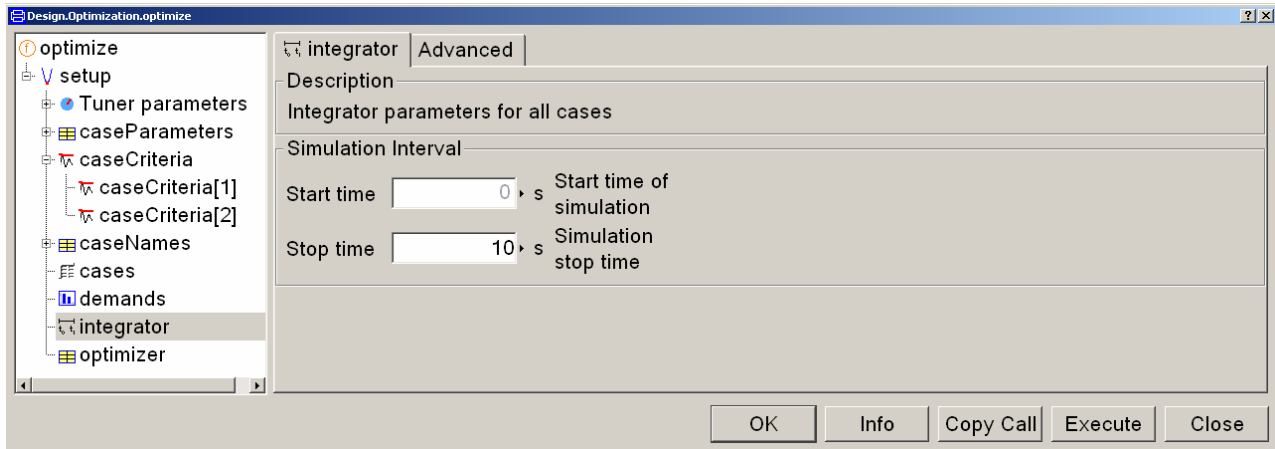
As already mentioned, it is possible to define multiple cases representing, e.g., different working conditions. We could provide different names for them, but for our first optimization run we simply use the default name.



Since we are using several criteria, they have to be weighted with regards to each other. In optimize(), the value “criterion / demand” is minimized, i.e., “demand” is used as scaling factor of “criterion”. A demand value has the same unit as the corresponding criterion. For this first setup, we use a demand value of 0.01° for the overshoot (=1 % overshoot) and 2° for the maximum absolute elevator deflection:



Finally, a simulation time of 10 s has to be defined under “integrator”:

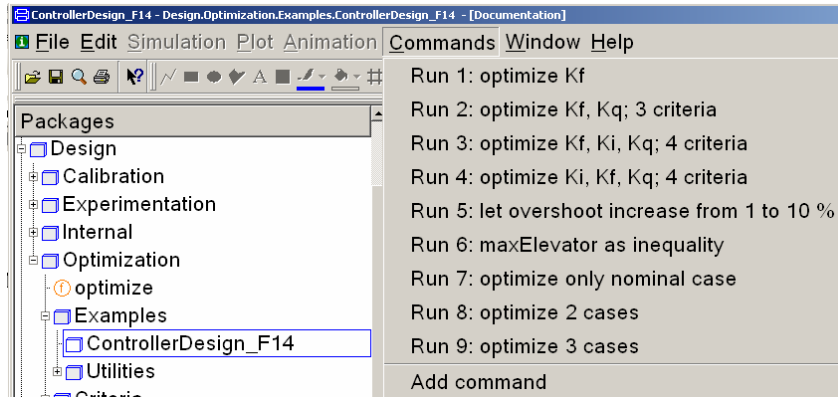


The optimization problem defined and to be solved by the Design package optimizer is now:

$$\min(\max(\text{overshoot}(K_f)/(0.01^\circ), \text{maxElevator}(K_f)/(2^\circ)), K_f \text{ in } [-10;0])$$

By clicking on the “Execute” button, the optimization is started.

The same setup could be obtained by selecting the first command in the “Commands” menu:



After “Execute” is pressed, the optimization is started. The Dymola command log shows the iterations and the final output.

```

Iteration 21
=====
Tuner parameters
+-----+-----+-----+-----+-----+-----+
| name          | value          | abs. diff.    | min           | max           | unit         |
+-----+-----+-----+-----+-----+-----+
| Kf            | -1.685169e+000 | 4.314831e+000 | -1.000000e+001 | 0.000000e+000 |             |
+-----+-----+-----+-----+-----+-----+
Criteria
+-----+-----+-----+-----+-----+-----+
| signal name   | criteria       | scaled        | diff.         | unscaled      | demand       | usac
+-----+-----+-----+-----+-----+-----+
| overshoot     | FinalValue     | max 2.95456  | -87.59%      | 2.954561e-002 deg | 1.000000e-002 deg | mini
| maxElevator  | FinalValue     | 0.59558      | -66.26%      | 1.191153e+000 deg | 2.000000e+000 deg | mini
+-----+-----+-----+-----+-----+-----+
Summary      23.81173      Maximum scaled criterion at start
              2.95456      Maximum scaled criterion in this iteration
              2.95456      Maximum scaled criterion in best iteration (#20)
=====
Optimization terminated successfully.
= { (-1.68517406292052) }

```

All iterations that are better than all previous ones are shown in the log. The following information are given in the log for iteration 21:

Tuner “Kf” is active (see last column in the figure above) and has the value “-1.68517”. This is a change of +4.31 with regards to the value of Kf before the optimization started. The search interval for the optimizer for this tuner is [-10 ... 0] (see min/max columns).

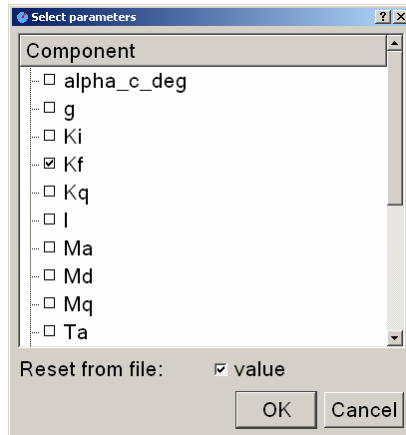
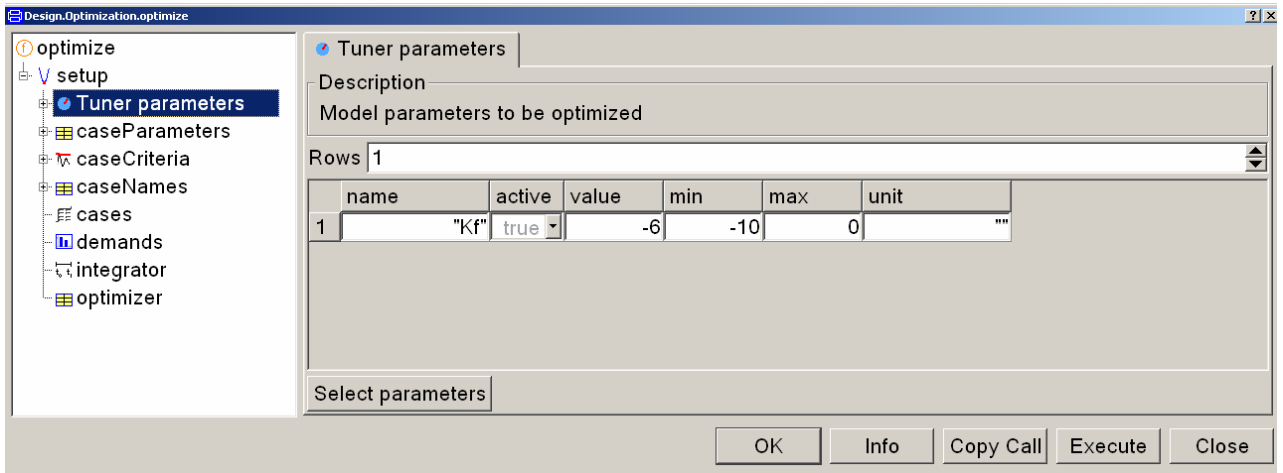
There is only one simulation case defined and therefore no additional information is given for the cases.

The scaled criteria “overshoot” is currently the largest of the scaled criteria (due to the “max” in front of the scaled value) and has a scaled value overshoot/demand = 2.95456. This is a change of -87.59 % with regards to the initial value. The actual value of overshoot = 0.0295456 (= 2.95 % overshoot). This criteria is minimized due to “minimize” and has a demand value of 0.01 (see column “demand”).

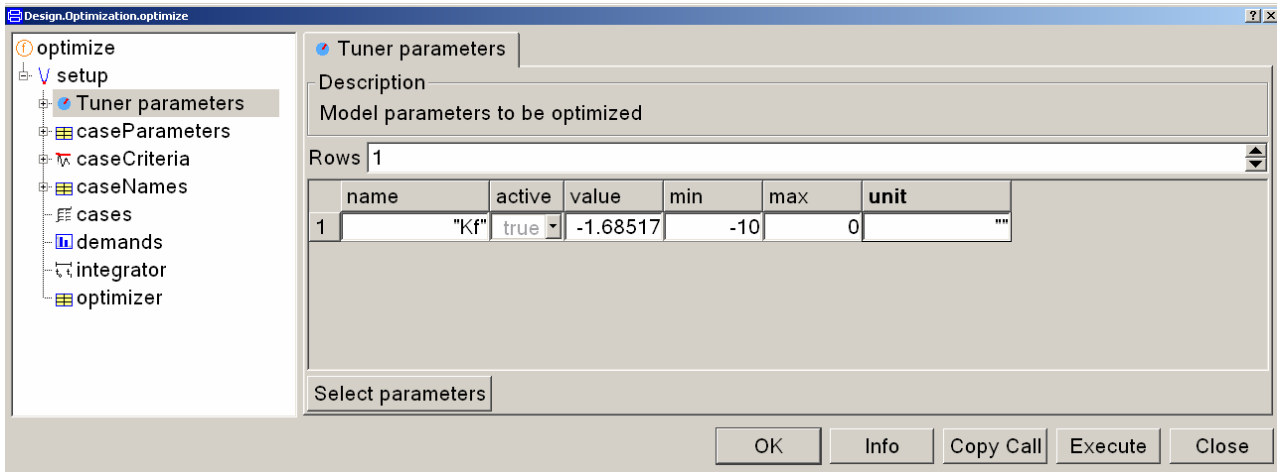
The best values of all tuners are also given in the log as last output. The best value of Kf is -1.68517406292052. It can be seen, that the overshoot is reduced by 87.59 % and the control activity by 66.26 % (see column “diff”). However, by tuning only Kf, the overshoot could not be reduced below the requested demand value of 1 %. Note, the scaled criteria is below 1, if the demand value is fulfilled. Therefore in the next steps the controller parameters Ki and Kq will be also optimized.

After the optimization is finalized, the Design.optimize() menu remains open (when using Execute). Nothing in the setup has changed. In order that the result of the last optimization run is included in the setup, it is necessary to select the “Select” button of “tunerParameters”,





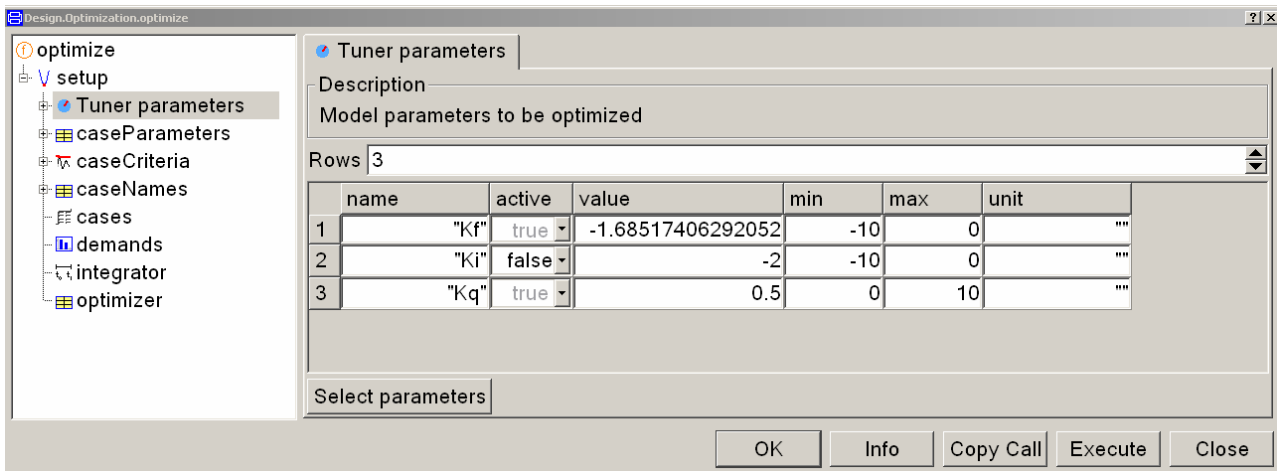
mark “value” and click on “OK”. This will load the values of all tuners from the last simulation run:



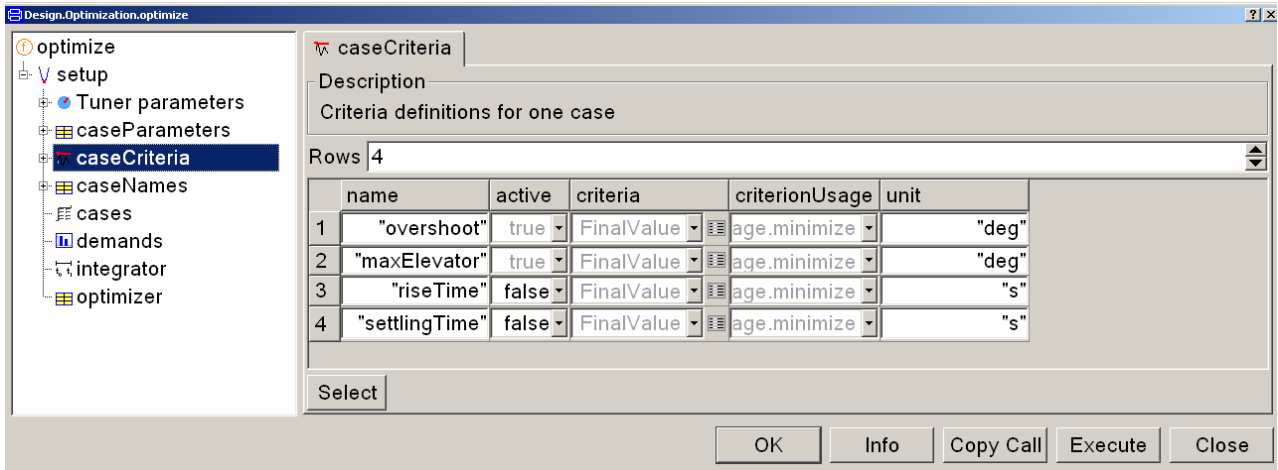
Since the optimizer performs a last simulation run with the best tuner values, these tuner values are the ones from the best iteration of the last optimization.

The setup is changed such that all controller parameters Kf, Ki, Kq are defined as tuners. Furthermore rise time and settling time are introduced as further criteria. These criteria are introduced to counteract the effect that the reduction of control activity and overshoot may lead to very long rise and settling times in the alpha step response.

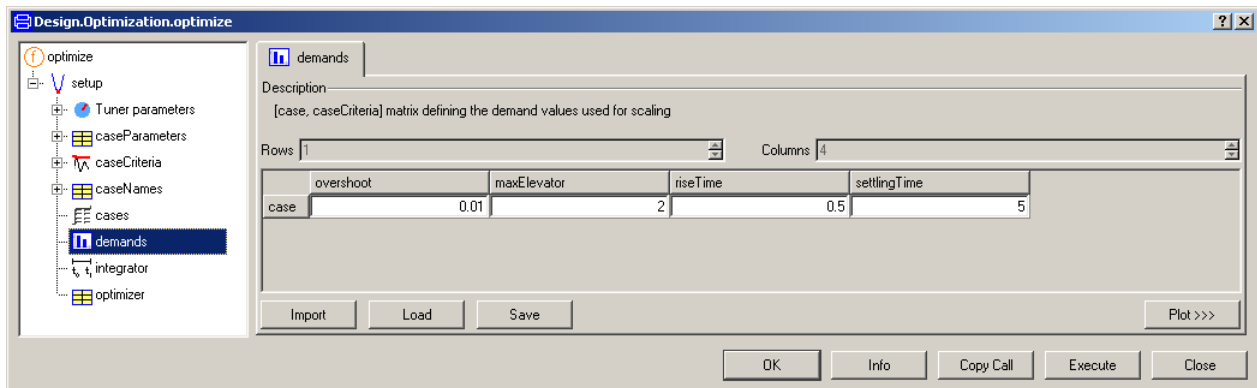
The controller parameters Kf and Kq are defined with active = **true** (default value in the second column of the Tuner parameters table). In turn, we set active = **false** for Ki. This means that the value of Ki is not changed by the optimizer. By increasing Kq it should be possible to reduce the overshoot.



In a first step we set active = **false** for the riseTime and settlingTime criteria (via the second column of the caseCriteria). Therefore, these criteria are shown in the log output, but are not utilized in the optimization.



Although the riseTime and the settlingTime criteria are not used in the optimization itself, we have to provide demand values for them (0.5 and 5).



After the optimization run (same as “Commands / Run 2”), Kq has been increased to 0.725383834580606 and Kf has been increased to -1.33686637964285. As a consequence, the overshoot is now below the demand value of 1 %. However due to the high value of Kq the rise time increased.

Finally we update the tuner values again (with the values from the previous optimization run), set the riseTime and settlingTime criteria active and make the controller parameter Ki a tuner as well, i.e., set active = **true** as done in “Commands / Run 3” After the optimization is finished, all demand values are fulfilled.

---

## Multi-criteria experimenting

The Design optimize function provides features for criteria weighting and scaling by demand values as well as the possibility to use criteria as a value to be **minimized** or to use them as **constraints**.

During an optimization run, the optimization criteria are scaled with their demand values, i.e. the value delivered to the optimization method is  $\text{criterion\_value}/\text{demand\_value}$ . By changing the demand value of a criterion, a differently weighted optimization task is defined and therefore normally a different solution is obtained. In the following, the effect of demand value variation on the multi-criteria controller parameter optimization for the F14 aircraft is shown. Furthermore, the effect of using an optimization criterion as inequality constraint is demonstrated.

The controller parameters Kf, Ki, Kq are defined as tuners (using the best values of the last run by “Reset from file”; you may also execute “Commands / Run 4” to achieve this result) and the final values of overshoot, riseTime, settlingTime and maxElevator as optimization criteria (c1(Kf,Kq,Ki)..c4(Kf,Ki,Kq)) with their new demand values  $d_i = \{0.01, 0.5, 2.5, 3\}$ . For a first optimization of the controller parameters all criteria are defined as minimum (default), i.e. the optimization task is to solve the min-max problem:

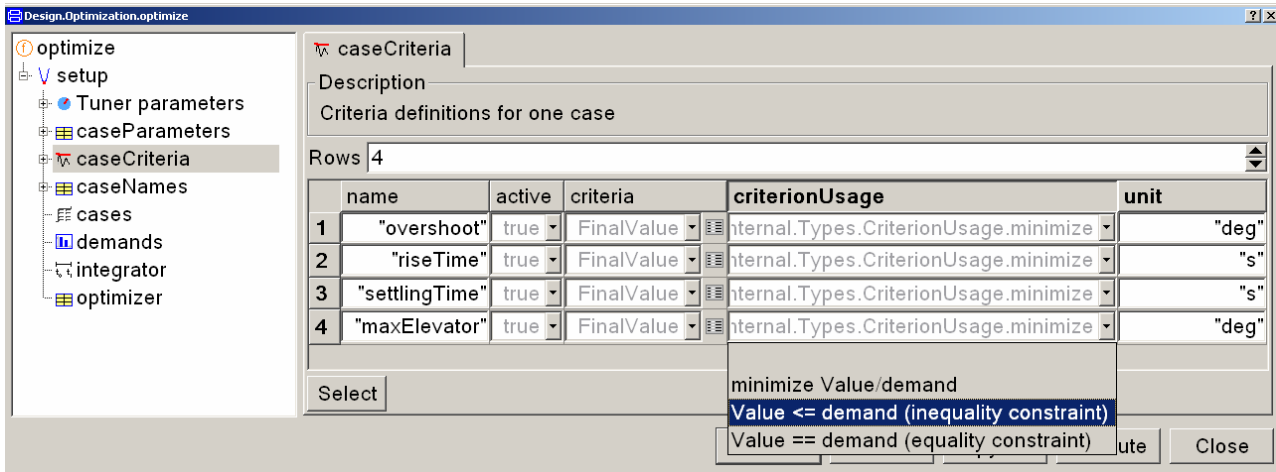
$$\min(\max(c1/d1, c2/d2, c3/d3, c4/d4)) \text{ over } Kf, Ki, Kq$$

A solution for the controller parameters Kf, Ki, Kq was found (-4.25146525932043, -4.23395327838336, 1.00358897271209) such that all criteria are reduced below their demand values. The scaled criteria all have nearly the same value (within computational accuracy): 0.7829. This result indicates that the solution is a Pareto-optimal solution, where no criterion can be further minimized without degrading at least one other criterion (provided this is not a local minimum).

As a first variation, we will change the demand value for the overshoot criterion from 1 % to 10 %. This means that more overshoot in the alpha step response is allowed and we expect that the other criteria improve. After updating the tuner values by “Reset from file”, we start the optimization (you can get this result also by executing “Commands / Run 5”).

In the command log output of this optimization run you can see that the overshoot increased with the effect that all the other criteria could be improved. We obtain again a Pareto-optimal solution among all criteria as their scaled criterion values are nearly identical: 0.73. This demonstrates how different compromise solutions can be found by variation of the demand values.

As a next modification of the optimization task we change the type of the maxElevator criterion (c4) from minimum to inequality:



This means, that this criterion is not minimized any more but taken as inequality constraint. The new problem to solve is

$$\min(\max(c1/d1, c2/d2, c3/d3)), \text{ subject to } c4/d4 \leq 1 \text{ over } K_f, K_i, K_q$$

After updating the tuners (“Reset from file”) and starting the optimization, the result is (this result can also be obtained with “Commands / Run 6”:

$$\{K_f, K_i, K_q\} = \{-5.52323684440722, -5.30428537997218, 0.992903698285534\}.$$

The change of the type of the maxElevator criterion from minimum to inequality yields a new controller parameter set. You can see from the simulation results that due to the new criterion formulation the elevator is now deflected to the maximum allowed value of 3° during the step response. This is an increase of 37.18 % in the maximum elevator deflection compared to the solution of the previous optimization (see the command log output). However due to the increased maximum elevator deflection other criteria could be decreased.

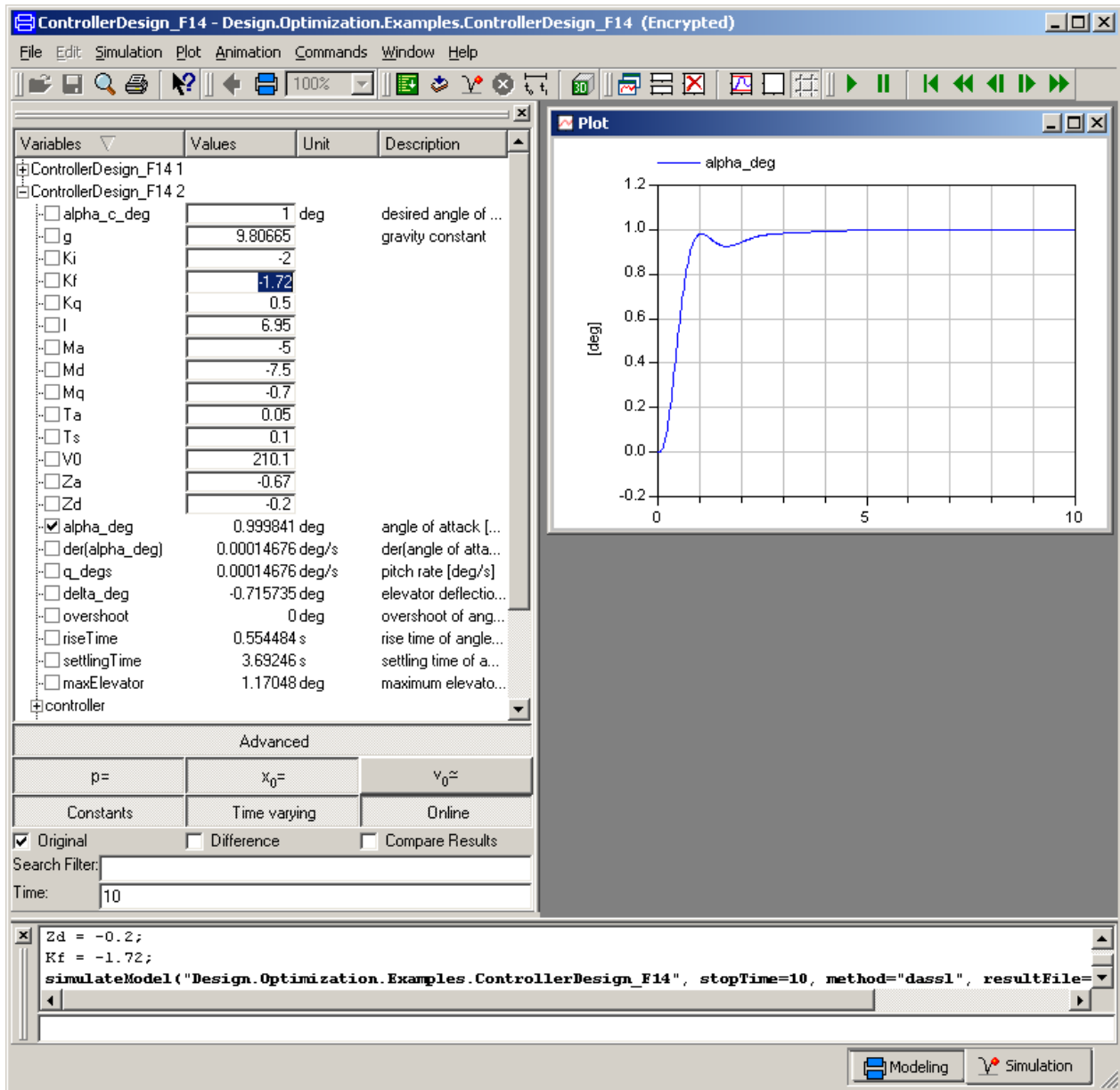
## Multi-case optimization

In this section, we start a new optimization task in order to extend the controller parameter synthesis to a multi-case optimization. As all aerodynamic parameters  $M_a$ ,  $M_q$ ,  $M_d$ ,  $Z_a$ ,  $Z_d$  of the F14 aircraft may vary within  $\pm 10\%$  of their nominal value,

$$\{M_a, M_d, M_q, Z_a, Z_d\}_{\text{nominal}} = \{-5, -7.5, -0.7, -0.67, -0.2\},$$

known worst-case scenarios are simultaneously considered in addition to the nominal case. A controller parameter set stabilizing all these cases shall be found.

First, a simulation with the current controller parameters  $K_i = -2$ ,  $K_f = -1.72$  and  $K_q = 0.5$  for the nominal case is performed.



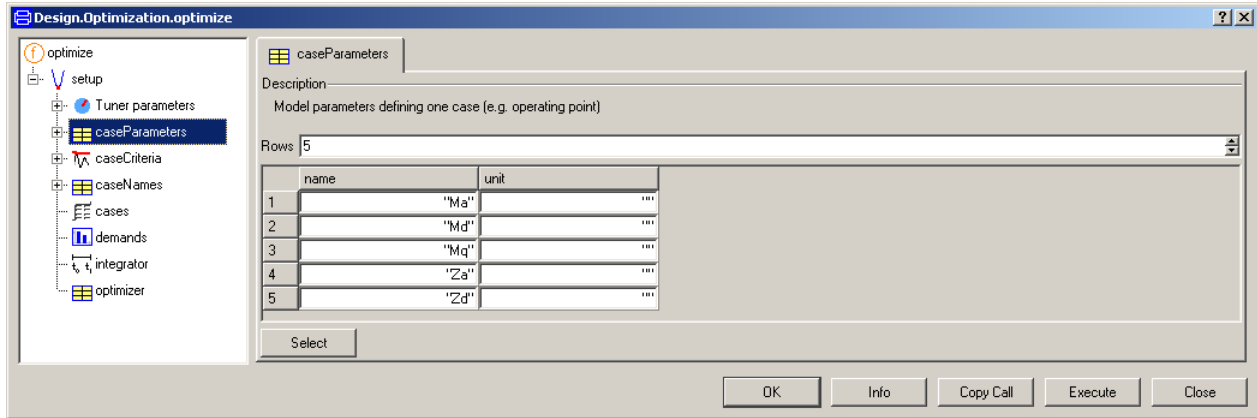
The plot browser shows the corresponding criteria values:

	overshoot	riseTime	settlingTime	maxElevator
demand value	0.01	0.5	4	2
current value	0	0.55	3.7	1.2

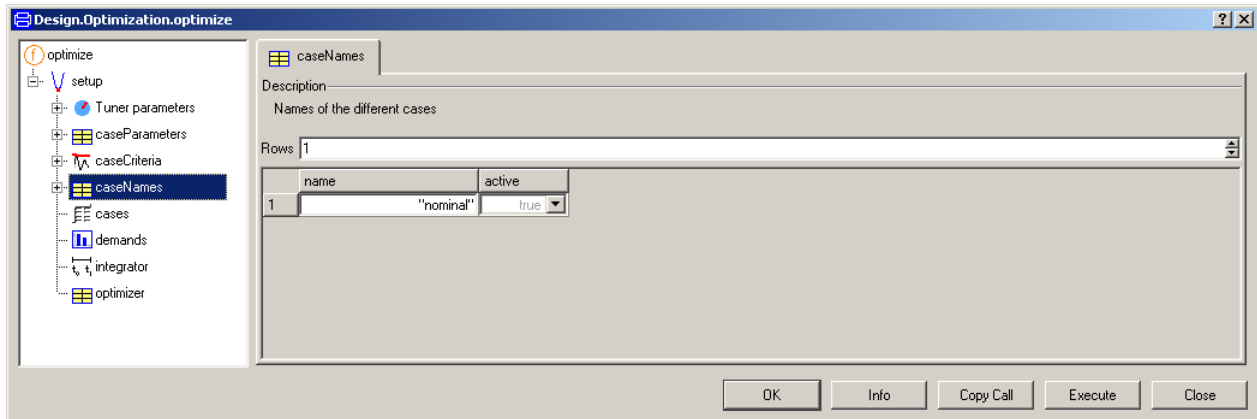
You see that the riseTime criterion is not yet satisfied. Therefore a new optimization task will be defined (you may use “Commands / Run 7” instead). In this particular step we only consider the nominal case.

It should be straightforward to define the tuners {Kf, Ki, Kq} and the criteria {overshoot, riseTime, settlingTime, maxElevator} with the demand values given in the above table.

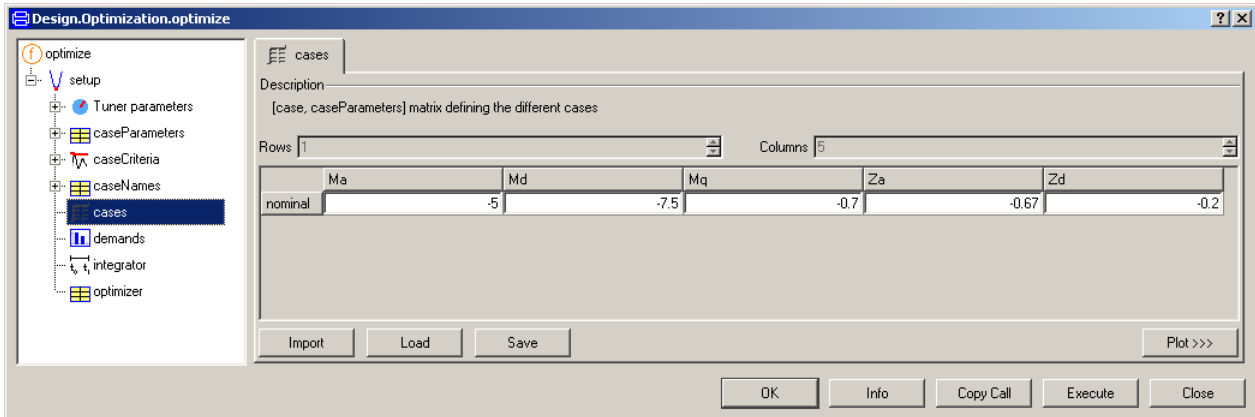
Further we define the aerodynamics parameters as case parameters.



An appropriate name is assigned to the currently considered case.



We provide the case parameters' values for the nominal case.



Finally, we set the simulation time (“integrator”) to 10 seconds.

The obtained optimization result satisfies all criteria and gives the following tuner values:

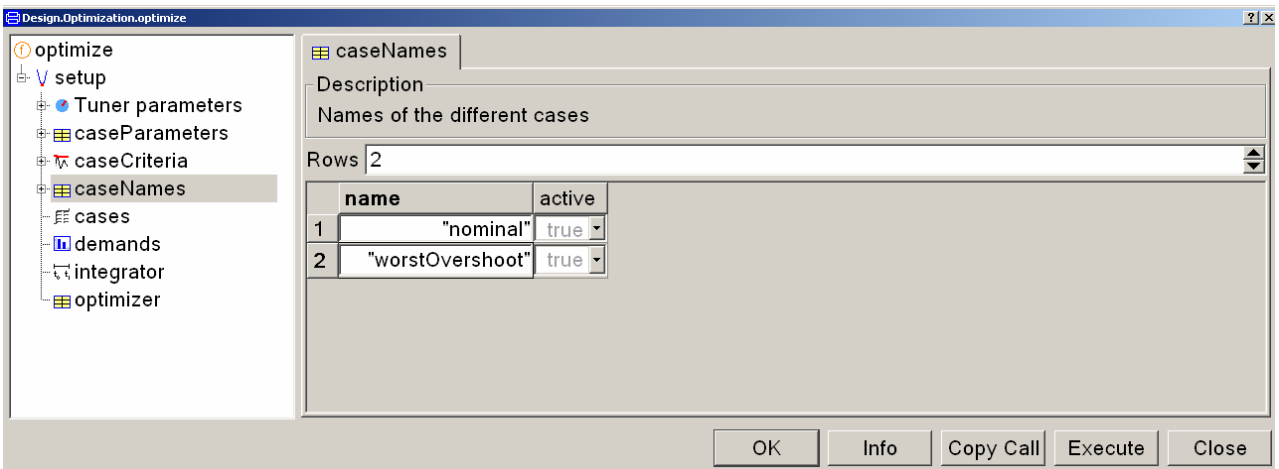
$$\{K_f, K_i, K_q\} = \{-2.7644705121894, -2.73348743052164, 0.61829473903147\}$$

From a different analysis, it is known, that the obtained controller set does not satisfy the criteria in the case

$$\{Ma, Md, Mq, Za, Zd\}_{\text{worstOvershoot}} = \{-4.5, -6.75, -0.63, -0.603, -0.18\} .$$

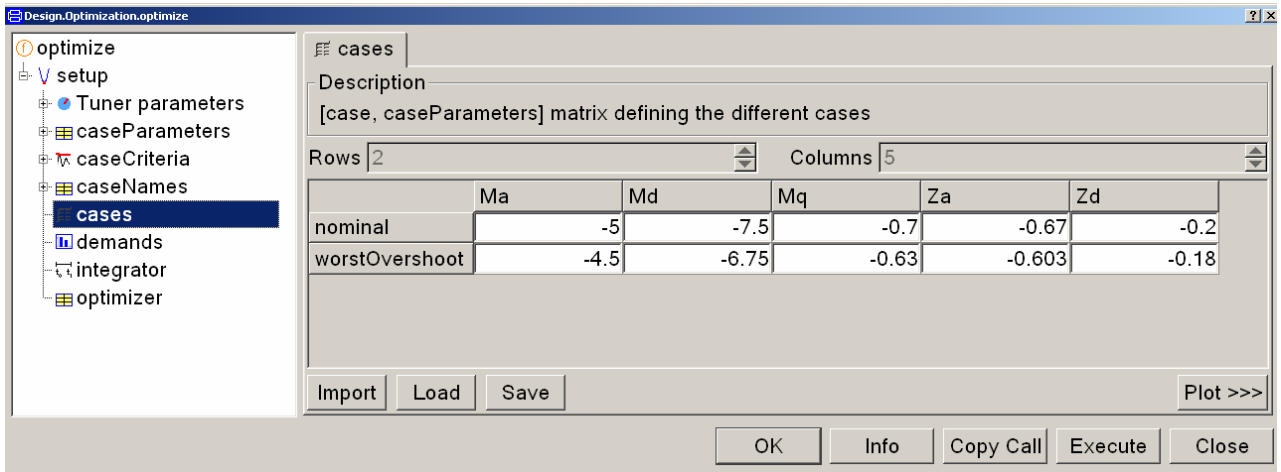
This set of aerodynamic parameters will be used to define a case “worstOvershoot” besides the “nominal” case

In “caseNames” we define the additional case:

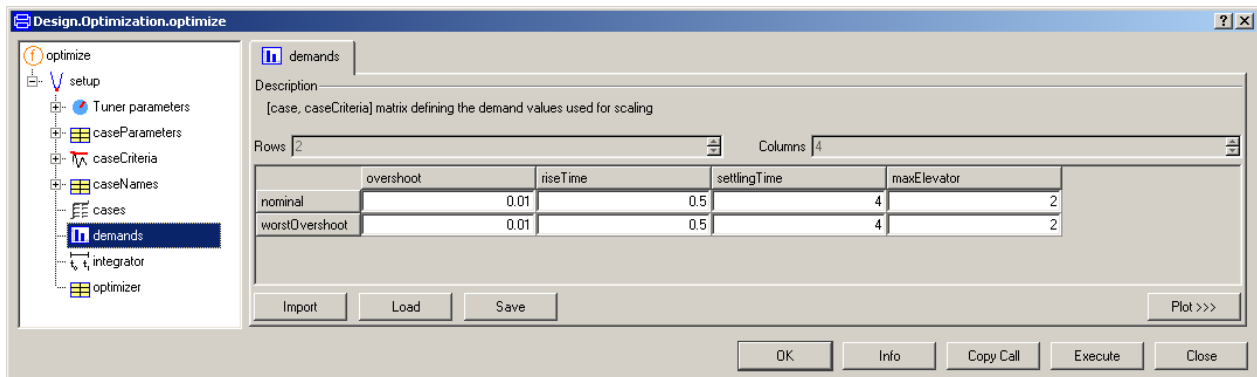


The different values for the parameters defining the new case are given in the cases [case,caseParameters] matrix:





We have to provide demand values for the new case.



An optimization will be performed now (i.e., after having updated the tuner values using “Reset from file”) for this multi-case scenario. The objective is to find controller parameter values for  $K_i$ ,  $K_f$ ,  $K_q$  such that all criteria are satisfied simultaneously for both cases "nominal" and "worstOvershoot" (you may execute this optimization task by “Commands / Run 8”).

You can see that for both cases a controller parameter set could be determined that satisfies all criteria:

$$\{K_f, K_i, K_q\} = \{-2.95319096183149, -2.71504063778775, 0.700519275733324\}$$

Again, it can be shown, that a controller using these parameters is not robust against the uncertainties in the aerodynamic parameters. The settling time criterion is not satisfied for the aerodynamic parameters

$$\{Ma, Md, Mq, Za, Zd\}_{\text{worstSettlingTime}} = \{-5.5, -6.75, -0.63, -0.737, -0.18\}$$

This set enters the setup as new case “worstSettlingTime”. Another optimization step will be performed (we do not forget to update the tuner values and to provide demand values for the new case) to determine controller parameter values to simultaneously satisfy all criteria for the three cases “nominal”, “worstOvershoot”, “worstSettlingTime”(you may execute this optimization task by “Commands / Run 9”). The optimizer is able to find a controller parameter set that satisfies all demands in all three operating points:

$$\{K_f, K_i, K_q\} = \{-3.15272731479831, -3.05878111172574, 0.764835115980916\}$$

By further analysis, it could be shown, that this controller parameter set stabilizes the aircraft robustly to the aerodynamics parameters.

Although the F14 example is very simple, many essential ingredients have been shown. They can all be applied also to much more complicated cases.

# **Model management**



# Model Management

The model management package includes version management, automatic documentation of model dependencies and encryption of models.

Use of the model management package requires the Model Management option.

---

## Version management

### **The context of version management**

In developing model components for a complex system such as a vehicle, many different kinds of competence are needed. Experts in engines, transmissions and chassis etc. are needed to develop a drivetrain. Because several people are involved in the process, it becomes essential to break up or decompose the overall problem into modular units during development.

As more people are involved in the process, the development is geographically and chronologically distributed because it is natural to have centers with specific core-competencies. This implies that the modular units developed must be seamlessly integrated to solve the overall problem, and the partitioning should be able to reflect the organizational structure of the model development teams.

In order to increase quality and reduce development time, tools should be made available to

- Provide a structure for organizing, storing and retrieving information (models, simulation results, documentation, experiment data).
- Support the exchange of information and simplify reuse of models throughout the organization.
- Ensure that correct information is available to each user (versions of libraries, corresponding experiments).

A version control system provides means to track changes to a set of files. A “commit” operation associates a developer and documentation with each change to the common storage of files. The Modelica text of two versions can be compared, and it is possible to back up to any previous version.

The underlying version control system must be able to support multiple concurrent developers working on the same set of models. Extensive locking of files is undesirable in a collaborative environment, and more recent tools also support concurrent development of closely related parts (with appropriate safety nets). A single physical person may have multiple roles in the development or use of the library; one role as a developer for new features of the library, and one role fixing bugs in a release version of the library.

Traceability is essential for maintaining quality over time. Tool enforcement to document modifications before they become publicly available gives the opportunity to review changes and improves quality. The development history (documentation of changes) may also be needed for tracing model incompatibilities, for example.

Model testing should be integrated with model development, which implies that the version control system must be able to handle test scripts, support utilities and binary test data. Regression testing, where models are simulated and compared with known good simulation results, is very powerful in detecting involuntary changes to model libraries. A failed regression test may cause either a change of a model, or the revision of the test itself.

Multiple libraries are often used together. In this case, version compatibility across libraries becomes essential. It must be possible to “tag” releases of multiple libraries to indicate compatibility at the project level.

Dymola supports storing, retrieving, etc. of models in version control systems such as CVS (Concurrent Versions System) or SVN (subversion). We have deliberately chosen to build on existing version control systems, which offers greater flexibility and better integration than a proprietary system. Because of the textual representation of models in the Modelica language, existing text-based tools can be used, for example, to compare versions. To browse changes in large systems, support in the graphical environment of Dymola would be needed.

The use of public libraries has increased in industry over several years. More recent is “open source development”, which can be described as the loosely organized development (typically of software) by several geographically separated parties. Public websites, such as SourceForge, support Open Source development with web-based tools and CVS/SVN. The Modelica Standard Library is maintained as a project on a server.

## Scope of implementation

This is a description of minimal support for version management in Dymola. The strategy is to provide a relatively thin layer on top of an existing version management system, such as, CVS (Concurrent Versions System) or SVN (subversion).

The added value for the user, compared to using existing graphical user interfaces e.g. WinCVS or TortoiseSVN, is:

- Commands are integrated in the Dymola environment. No need to swap between different applications. Some information is easily accessible in Dymola, e.g. version number and date.
- Some steps have been automated. For example, Dymola knows the filename of the current class, knows if there are files which have been modified, etc.
- Files are automatically reloaded into Dymola after updates from the repository.

However, there is no need to provide a comprehensive version management environment in Dymola. More complex tasks are better performed in specialized tool such as WinCVS, TortoiseCVS, TortoiseSVN or RapidSVN.

## Supported features

Dymola provides a graphical user interface to the most basic CVS and SVN commands, where the principal automatic step is to provide the correct file name in which the model the user is located.

The primary commands (in CVS terminology) are:

### Update

Updates your local copy of the file with changes from the repository. If your file has been changed since it was last updated, your changes are merged with the changes made to the repository. After a successful update the file is reloaded into Dymola.

If conflicts arise during the merge, this is noted in the message window, and the file is not reloaded into Dymola.

See Query Update for an explanation of the status code displayed in the message window.

### Commit...

Updates the repository with changes you have made in your local file. Your file is first checked to make sure that you have an up-to-date copy. You are then asked to enter a description of the changes, which is later available through the Log command.

### Add Model

Makes a new model's file known to the underlying version management system. The user must then perform a Commit on the model.

### **Add File...**

Makes an arbitrary file known to the version management system. The user must select the file using a file browser.

### **Diff**

Displays the textual differences between your local file and the corresponding version in the repository.

### **Query Update**

Displays which files in the model's directory are

- Locally modified compared to the corresponding version in the repository (marked by “M” before the filename).
- Changed in the repository compared to the version that was checked out (“U” or “P”).
- Caused a conflict during an Update operation, or which could potentially create a conflict because it is both locally modified and changed in the repository (“C”).
- Added but not yet committed (“A”).
- Unknown to the version management system (“?”).

Local files are not updated. The repository is not changed.

### **Status**

Displays version status of the file. The information includes:

- If the file is up-to-date, needs an Update, or has been locally changed.
- Revision of your local file and the repository file.
- A list of all symbolic tags and which revisions they refer to.

### **Log**

Displays log messages which were entered every time the file was committed, and a list of all symbolic tags and which revisions they refer to

### **Revert**

Deletes your local file and retrieves the latest version from the repository. All changes to your local file are lost.

All version management systems operate on files. An environment which would allow version management of individual models even when several models are stored in the same file could be implemented on top of external tools, but would be quite complex. However, Dymola can easily map from model to the corresponding filename, and also knows when a



model is part of a larger package comprising several files (in which case updates probably should be made on all files).

Also note that CVS can update special “keywords” in the Modelica text, which can be used to automatically insert information in the model documentation. They include version number, date of last change, and a log describing all changes. An example of this is given below.

### **Conflict handling after update**

If several users have modified the file, the “update” command of CVS or SVN will attempt to merge the changes. If they have modified the same lines of code, CVS will detect a conflict. After a conflict the original modified file is kept as backup, and the merged file contains both sets of changes marked by special indicators inserted into the text. It is then up to the user to resolve the conflicts.

An important issue here is that Dymola cannot use the file until conflicts have been resolved. Initially we do nothing, i.e., require that the user edits the Modelica file with some external text editor to delete conflicting lines and their indicators. At some future point in time Dymola could be extended to parse Modelica text with CVS conflict indicators, and the resolution could be handled from within Dymola (which of course has better support for analyzing the conflicts). An intermediate step is to rename the file with conflicts and restore the backup; this will at least maintain consistency between the Dymola environment internally and the corresponding file externally.

It should be noted that merge conflicts arise from a people management problem, and are rare in practice. Normally people working on a project do not edit the same code.

### **Version management of non-model files**

The discussion of version management is naturally focused on Modelica code, but the facilities also handle parameter sets, experiments and trajectories in large projects.

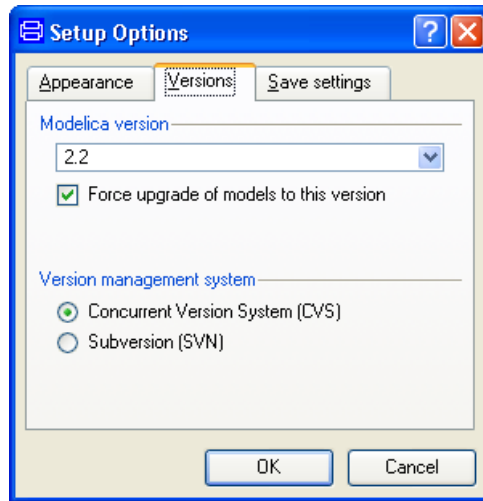
Parameter sets and experiments can be represented by Modelica models. In this case the experiment extends from the top-level model and provides experiment-specific parameters through modifiers of the extends clause. An alternative is to set up the experiment with a Modelica script file (extension .mos). Trajectories are represented by binary files (extension .mat).

Files which are not Modelica text can be stored in hierarchical Modelica packages. These files are added to the repository using the Add File... command. Operations performed on the package will then also operate on the contained .mos and .mat files. Operations supported include “update”, “commit” and “status”.

## **Selecting version management system**

Dymola supports two version control systems, CVS and SVN, and generates the appropriate external commands to perform operations on the version control system. Which system is used is set in Edit/Options/Version.

**Setting version management system.**



The setting for version management system is stored when you do a “Save” in Edit/Options/Save settings.

## Version management using CVS

Version management support in Dymola assumes that there exists a functional CVS environment. In its simplest form there exists a CVS repository on a local disk. More advanced installations maintain a CVS server on a separate UNIX system; one such setup is the use of the SourceForge server to maintain the Modelica standard library. Two examples are given below.

It is worth pointing out that Dymola and the underlying CVS system supports development of libraries maintained at several different servers concurrently. For example, the Modelica standard library may be maintained at SourceForge, other libraries proprietary to the company, and still others by the user on a local disk. In this fashion version management also facilitates effective distribution of updates as they become available from the vendor.

**Location of the CVS command.**

Note: in several places the user is asked to execute the CVS command. The file cvs.exe is located in the Dymola distribution, typically \dymola\bin. For conciseness it is written as cvs in these examples.

### Local CVS repository

To set up a local CVS repository, first choose the machine and disk on which you want to store the revision history of the source files. CPU and memory requirements are modest, so most machines should be adequate.

To create a repository with a set of configuration files, run the “cvs init” DOS command to set up an empty repository in the designated directory. For example,

```
mkdir \cvs
```

```
cvs -d \cvs init
```

These steps complete the initialization of the CVS repository. The “cvs init” command is careful to never overwrite any existing files in the repository, so no harm is done if you run cvs init on an already set-up repository.

Note that if you use a Windows drive letter, you must write a slightly longer repository specification because the “cvs” command interprets the colon after the drive letter:

```
cvs -d ":local:c:\cvs" init
```

The CVS repository is initially empty. It is necessary to create one or more directories which act as top-level directories for further development. For example, we may create a “models” directory:

```
mkdir \cvs\models
```

To use a CVS repository it is necessary to initially perform a “checkout” operation:

```
cvs -d \cvs checkout models
```

This command creates a “models” sub-directory with all models currently stored in the corresponding top-level directory in the CVS repository. It also creates extra directories called “CVS” at each level, which are used to maintain CVS status information. The files inside the “CVS” directories should never be manipulated by hand.

## Access to servers via CVS

Projects maintained at SourceForge (<http://www.sourceforge.net>) or other servers can be accessed via CVS. To access the Modelica area via CVS, you set up your CVSROOT when the files are initially checked out, and do a “cvs login” with an empty password. After that the usual CVS commands work as expected.

If you work against a single CVS repository it may be convenient to set the CVSROOT environment variable to the value below, as an alternative to using the -d command line switch:

```
:pserver:anonymous@cvs.modelica.sourceforge.net:/cvsroot/modelica
```

To use it you must first login and then check out using these DOS commands:

```
cvs login // empty password
cvs checkout Modelica // check out standard library
```

This will checkout the entire Modelica standard library in the current directory.

## An example of file management using CVS

In this example we will demonstrate the basic version management operations provided by Dymola. It is divided into several different steps to setup a local CVS repository, to create a new model, and to make changes to an existing model.

## Setting up the CVS repository

A local CVS repository is set up, and then a new top-level directory called “models” is created. Finally the new top-level directory needs to be checked out in the current working directory. Execute these DOS commands:

```
mkdir \cvs
cvs -d \cvs init
mkdir \cvs\models
cvs -d \cvs checkout models
```

A new (empty) directory has now been created in our working directory.

## Creating a new model

We can start by creating a model in Dymola and saving it in the “models” directory. For our example we will use the simple model:

```
model Decay
  Real x(start=2);
equation
  der(x) = -x;
end Decay;
```

Initially the model is unknown to the version management system. For example, a Version/Status command returns this information in the message window:

```
C:\util\dymola\bin\cvs.exe status -v Decay.mo
(in directory C:/dev/Proj/DymolaQt/models)
cvs.exe status: use `cvs.exe add' to create an entry for
Decay.mo
=====
File: Decay.mo           Status: Unknown
  Working revision:      No entry for Decay.mo
  Repository revision:   No revision control file
Command finished.
```

Next we perform the Version/Add Model command to make the model’s file known to the version management system.

```
C:\util\dymola\bin\cvs.exe add Decay.mo
(in directory C:/dev/Proj/DymolaQt/models)
cvs.exe add: scheduling file `Decay.mo' for addition
cvs.exe add: use 'cvs.exe commit' to add this file permanently
Command finished.
```

The information from Version/Status is now different, but there is no file in the repository yet (not until we commit the file).

```
C:\util\dymola\bin\cvs.exe status -v Decay.mo
(in directory C:/dev/Proj/DymolaQt/models)
=====
File: Decay.mo           Status: Locally Added
```

```

Working revision:    New file!
Repository revision: No revision control file
Sticky Tag:        (none)
Sticky Date:       (none)
Sticky Options:    (none)
Command finished.

```

When we perform Version/Commit..., the user is asked to enter a log message describing what changes are committed. The lines beginning with CVS are generated to help us remember the nature of the commit.

```

This is the first version of our test example.
CVS: -----
CVS: Enter Log.  Lines beginning with `CVS:' are removed
automatically
CVS:
CVS: Added Files:
CVS:      Decay.mo
CVS: -----

```

The message after the commit operation has finished looks like this:

```

C:\util\dymola\bin\cvs.exe commit Decay.mo
(in directory C:/dev/Proj/DymolaQt/models)
RCS file: \cvs/models/Decay.mo,v
Checking in Decay.mo;
\cvs/models/Decay.mo,v <-- Decay.mo
initial revision: 1.1
done
Command finished.

```

The output from Version/Status now contains more information, in particular the version number of the file and the date it was last changed in the repository.

```

C:\util\dymola\bin\cvs.exe status -v Decay.mo
(in directory C:/dev/Proj/DymolaQt/models)
=====
File: Decay.mo          Status: Up-to-date
Working revision:      1.1          Fri Oct 04 09:34:02 2005
Repository revision:  1.1          \cvs/models/Decay.mo,v
Sticky Tag:           (none)
Sticky Date:          (none)
Sticky Options:       (none)
Existing Tags:
      No Tags Exist
Command finished.

```

It is also possible to view the change log with Version/Log. The change log contains all messages entered during commit operations.

```

C:\util\dymola\bin\cvs.exe log Decay.mo
(in directory C:/dev/Proj/DymolaQt/models)
RCS file: \cvs/models/Decay.mo,v
Working file: Decay.mo

```

```

head: 1.1
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 1;      selected revisions: 1
description:
-----
revision 1.1
date: 2005/10/04 09:34:02; author: Dag; state: Exp;
This is the first version of our test example.
=====
Command finished.

```

The output from both Status and Log contain information specific to the underlying CVS system, which is beyond the scope of this report. For non-expert users it would be beneficial to filter the raw output.

## Changing an existing model

Starting with the model created above, we now modify it by adding a time constant  $T_i$ . The revised Modelica text looks like this:

```

model Decay
  Real x(start=2);
  parameter Real Ti=1;
equation
  der(x) = -x/Ti;
end Decay;

```

The Version/Diff command will display the differences between the model stored in the repository and the current model. Changed lines are indicated by “!”, added lines by “+” and any removed lines by “-” (this is the so-called “context diff” format).

```

C:\util\dymola\bin\cvs.exe diff -c Decay.mo
(in directory C:/dev/Proj/DymolaQt/models)
Index: Decay.mo
=====
RCS file: \cvs/models/Decay.mo,v
retrieving revision 1.1
diff -c -w -r1.1 Decay.mo
*** Decay.mo      2005/10/04 09:34:02 1.1
--- Decay.mo      2005/10/04 09:43:12
*****
*** 1,5 ****
   model Decay
     Real x(start=2);
   equation
!  der(x) = -x;
   end Decay;
--- 1,6 ----
   model Decay

```

```

    Real x(start=2);
+   parameter Real Ti=1;
    equation
!   der(x) = -x/Ti;
    end Decay;
Command finished.

```

The Version/Query Update command is used to quickly list which files have been locally modified (indicated by “M”) or need to be updated from the repository (“U”).

```

C:\util\dymola\bin\cvs.exe -qn update
(in directory C:/dev/Proj/DymolaQt/models)
M Decay.mo
Command finished.

```

The Version/Query Update command does not operate only on the file of the model. Instead it operates on the entire directory and all sub-directories; this makes it particularly useful to concisely review the status of all files in a complex model hierarchy.

The model is then committed to the repository with Version/Commit, as shown above. If we review the log with Version/Log, we see that the new revision comment is also listed. The listing also shows the number of changed Modelica text lines.

```

C:\util\dymola\bin\cvs.exe log Decay.mo
(in directory C:/dev/Proj/DymolaQt/models)
RCS file: \cvs/models/Decay.mo,v
Working file: Decay.mo
head: 1.2
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 2;      selected revisions: 2
description:
-----
revision 1.2
date: 2005/10/04 09:44:57;  author: Dag;  state: Exp;  lines:
+2 -1
Added time constant Ti.
-----
revision 1.1
date: 2005/10/04 09:34:02;  author: Dag;  state: Exp;
This is the first version of our test example.
=====
Command finished.

```

This concludes the demonstration of how models are edited in co-operation with the version management facilities in Dymola.

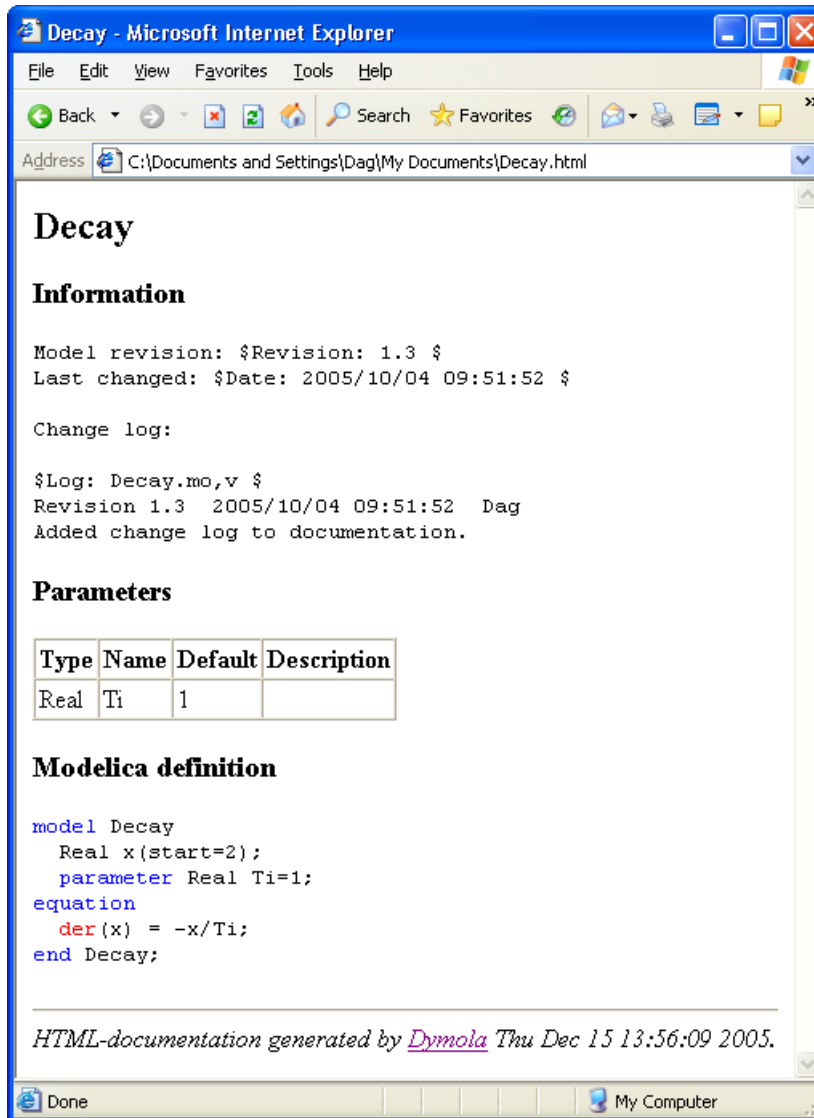
## Use of revision information

The underlying CVS system supports expansion of particular keywords, for example to automatically document the revision or commit date of the model. We could for example enter this text in the revision part of the documentation layer of our model:

```
Model revision: $Revision$  
Last changed: $Date$  
Change log:  
$Log$
```

The keywords indicated by \$ will be expanded at the next commit operation. The result is shown in the following example of HTML documentation:





## Version management using SVN

Version management support in Dymola assumes that there exists a functional SVN (subversion) environment. In its simplest form there exists a SVN repository on a local disk. More advanced installations maintain a SVN server on a separate UNIX system.

It is worth pointing out that Dymola and the underlying SVN system supports development of libraries maintained at several different servers concurrently. For example, the Modelica standard library may be maintained at [svn.modelica.org](http://svn.modelica.org), other libraries proprietary to the

company, and still others by the user on a local disk. In this fashion version management also facilitates effective distribution of updates as they become available from the vendor.

#### **SVN editor setup.**

Some SVN operations require input from the user, for example a log message when a file is committed. To enable this feature the user must set either of the environment variables SVN\_SETUP, EDITOR or VISUAL to the name of a text editor. On Windows “notepad” will be sufficient for most uses.

#### **Location of the SVN command.**

Note: in several places the user is asked to execute SVN commands. The files svn.exe and svnadmin.exe should be available from the command line if you have performed the default installation of SVN (see “References” below).

### **Local SVN repository**

To set up a local SVN repository, first choose the machine and disk on which you want to store the revision history of the source files. CPU and memory requirements are modest, so most machines should be adequate.

To create a repository with a set of configuration files, run the “svnadmin create” DOS command to set up an empty repository in the designated directory. For example,

```
svnadmin create \svn
```

The SVN documentation suggests that you populate the repository with three directories called “branches”, “tags” and “trunks”. The easiest way to do that is to create these directories locally and then import them:

```
mkdir models
cd models
mkdir branches
mkdir tags
mkdir trunk
cd ..
svn import models file:///svn/models -m "Initial import"
```

SVN will report that it has imported the directories as revision 1. It is worth noting that SVN manages directories as well as files, whereas CVS only manages files directly and implicitly creates directories as needed.

These steps complete the initialization of the SVN repository. Remove the local “models” directory to start over.

```
rmdir /S models
```

To use a SVN repository it is necessary to initially perform a “checkout” operation to create a local copy with files that can be modified

```
svn checkout file:///svn/models/trunk models
```

This command creates a “models” sub-directory with all models currently stored in the corresponding top-level directory in the SVN repository. It also creates extra directories called “.svn” at each level, which are used to maintain SVN status information. The files inside the “.svn” directories should never be manipulated by hand.

## An example of file management using SVN

In this example we will demonstrate the basic version management operations provided by Dymola. The example shows the first steps from the CVS-based example above.

Setup the SVN repository with initial directories, and check it out. This is described above.

### Creating a new model

We can start by creating a model in Dymola and saving it in the “models” directory. For our example we will use the simple model:

```
model Decay
  Real x(start=2);
equation
  der(x) = -x;
end Decay;
```

Initially the model is unknown to the version management system. For example, a Version/Status command returns this information:

```
svn.exe status Decay.mo
(in directory C:/Dag/models)

?      Decay.mo

Command finished.
```

Next we perform the Version/Add Model command to make the model’s file known to the version management system.

```
svn.exe add Decay.mo
(in directory C:/Dag/models)

A      Decay.mo

Command finished.
```

We can now perform a Version/Query Update command to get some more information.

```
svn.exe status --verbose --show-updates
(in directory C:/Dag/models)

A          0      ?  ?      Decay.mo
          1          1 Dag      .
Status against revision:      1

Command finished.
```

When we perform Version/Commit..., the user is asked to enter a log message describing what changes are committed. The lines at the end are generated by SVN to help us remember which file is committed.

```
This is the first version of our example.  
--This line, and those below, will be ignored--
```

```
A    Decay.mo
```

The message after the commit operation has finished looks like this:

```
svn.exe commit Decay.mo  
(in directory C:/Dag/models)  
  
Adding          Decay.mo  
Transmitting file data .  
Committed revision 2.  
  
Command finished.
```

It is also possible to view the change log with Version/Log. The change log contains all messages entered during commit operations.

```
svn.exe log Decay.mo  
(in directory C:/Dag/models)  
  
-----  
r2 | Dag | 2005-12-15 11:59:22 (Thu, 15 Dec 2005) | 2 lines  
  
This is the first version of our example.  
  
-----  
  
Command finished.
```

The output from both Status and Log contain information specific to the underlying SVN system, which is beyond the scope of this report. For non-expert users it would be beneficial to filter the raw output.

## Changing an existing model

Changing the model follows the same pattern as for the CVS-based example above. The main difference is that SVN log message are different from those produced by CVS.

## References

The primary reference to the CVS version management system is

- Per Cederqvist et al. (1993): “Version Management with CVS”.

CVS binaries for several platforms and documentation (including Cederqvist et al.) are available for downloading from the official CVS homepage:

```
http://www.cvshome.org/
```

The primary source on Subversion is the homepage. The SVN command line tools used by Dymola are available here.

<http://subversion.tigris.org/>

Graphical user interfaces to SVN are available for downloading. Two of the more popular are TortoiseSVN (an extension to Windows Explorer)

<http://tortoisesvn.tigris.org/>

and RapidSVN

<http://rapidsvn.tigris.org/>

which is a free-standing application.

---

## Model dependencies

Dymola can export documentation of models and packages in HTML format. The HTML documentation contains information extracted from Modelica classes. For example, model parameters and functions inputs and outputs are tabulated for easy reading without any need to understand the Modelica text.

Dymola can also make tables of cross-references in HTML. Such a table clearly shows dependencies to other packages, and in some cases incorrect references can be found. The following is an example from the design calibration package:

These classes have been referenced in this package.

Class	Referenced From
<a href="#">Plot3D</a>	<a href="#">sweepTwoParameters</a>
<a href="#">Design.Internal.Records.MatCsvFileName</a>	<a href="#">dataPreprocessing</a>
<a href="#">Design.Internal.Records.MatCsvFileNameOut</a>	<a href="#">dataPreprocessing</a>
<a href="#">Design.Internal.Records.ModelCalibrationSetup</a>	<a href="#">calibrate</a> , <a href="#">checkCalibrationSensitivity</a> , <a href="#">perturbParameters</a> , <a href="#">sweepParameter</a> , <a href="#">sweepTwoParameters</a>
<a href="#">Design.Internal.Records.PerturbationParameter</a>	<a href="#">perturbParameters</a>
<a href="#">Design.Internal.Records.PreprocessingSignal</a>	<a href="#">dataPreprocessing</a>
<a href="#">Modelica.Utilities.Streams</a>	<a href="#">dataPreprocessing</a>
<a href="#">Modelica.Utilities.Streams.print</a>	<a href="#">checkCalibrationSensitivity</a> , <a href="#">sweepTwoParameters</a>
<a href="#">Modelica.Utilities.Strings</a>	<a href="#">checkCalibrationSensitivity</a> , <a href="#">perturbParameters</a> , <a href="#">sweepTwoParameters</a>
<a href="#">Modelica.LinearSystems</a>	<a href="#">dataPreprocessing</a>

<a href="#">Modelica_LinearSystems.StateSpace</a>	<a href="#">dataPreprocessing</a>
<a href="#">Modelica_LinearSystems.TransferFunction</a>	<a href="#">dataPreprocessing</a>
<a href="#">Modelica_LinearSystems.ZerosAndPoles</a>	<a href="#">dataPreprocessing</a>
<a href="#">Modelica_LinearSystems.Types</a>	<a href="#">dataPreprocessing</a>
extends <a href="#">Modelica.Icons.Function</a>	<a href="#">calibrate</a> , <a href="#">checkCalibrationSensitivity</a> , <a href="#">perturbParameters</a> , <a href="#">sweepParameter</a> , <a href="#">sweepTwoParameters</a>

The left column shows all classes that have been referenced, for example in import statements or as the type of a component; extends clauses are specially marked. The right column show the classes in this package which contain some kind of reference. To see what the reference is, click on the link and view the Modelica text.

## Cross-reference options

The generation of cross-references is controlled by options in File/Export/Setup HTML.

### Per file

Generate cross-references to classes in HTML documentation in each HTML-file. This is typically a subpackage in a larger library.

### Top level

Generate cross-references to classes in HTML documentation for top-level package. Because this often is quite large, the cross-references are stored in a separate file which is linked from the top-level HTML file (near the end).

### Full name

Generate HTML cross-references to classes using full name (the default). When checking consistency of referencing to classes it may be useful to disable this option, because inconsistent naming will show up as multiple cross-reference entries.

---

## Encryption in Dymola

### Introduction

There are many closed simulation packages on the market where you are not able to see the details of the models. Modeling is an art in the sense of describing the relevant aspects of

the object under observation. It is thus very important to be able to see what assumptions and approximation the author of a model made.

Dymola is open to view all and possibly modify the details of models by showing graphical representations and, if all details are wanted, the Modelica code itself. However, Dymola also supports concealment of model details, if, for example, a supplier wants to protect proprietary information when shipping models.

A classical way of protecting software is to distribute only executable programs or object code and no source code. This approach is not useful for Modelica models. To achieve robust and efficient simulation, it is important that Dymola can make a global analysis and manipulation of all equations. It is thus highly desirable to give Dymola access to the equations in their original form. Encryption of the textual Modelica representation of the model supports concealment of internal parts such as the equations, while still allowing Dymola internally to access the equations as if the model was not encrypted.

There are also other aspects of protecting models and model libraries. Prevention of unauthorized modification of models, but still having unrestricted viewing and use is supported by including checksums. Another aspect of library protection is to ensure authorized use. In this case, any use of the library is controlled by options in a license file.

Encryption requires the “Model Management” option in Dymola.

## **Visible and concealed classes**

The basic idea of the protection of models is to hide some information while making it possible to use the model components.

A protected library typically consists of parts that are open, and other parts that are protected.

Protected parts may require different degree of information hiding, e.g.:

- The model is regarded as a “black box”. The icon, its connectors and parameters as well as documentation are available to the user to allow use of the model as a component, but model structure and equations are concealed.
- The model is completely concealed from external use.

Dymola supports concealment by encrypting models or libraries and the use of protected code sections, and special annotations to allow more information to be revealed. The special annotations are grouped in “Protection” group (similar to e.g. “Diagram”).

There are several kinds of classes in an encrypted library starting from the most open:

- Example classes that are completely open, such that a user can duplicate it and use it as a basis for their own work. They can still refer to concealed classes.
- Classes that that can be viewed completely (including the entire Modelica text), but cannot be copied.
- Classes where the diagram is visible (but not the text).
- Classes where only the interface is visible. [This is the normal case].

- Concealed classes are completely hidden for the users, who shall not be aware of the existence of such components at all. They are not shown in the package browser and they cannot be inspected.

A class or a component is defined as concealed if one of these conditions is fulfilled:

- It is declared in the protected section of an encrypted class.
- Its lexically enclosing scope is concealed.
- It has the Protection-annotation: `hideFromBrowser=true`.

Dymola supports encryption on file basis, which means that all parts of an encrypted package must be stored in one file. Storing an encrypted package in several files or in subdirectories would reveal structural information. Instead it is possible to reveal the contents of encrypted packages.

## Developing encrypted libraries

To allow visible components to be used in the normal way to compose models, set parameters and initial values, the developer of such components must make a careful design. The public part must provide all necessary parameters, necessary control of initialization and variables to inspect and plot. Nested modifiers cannot be used to modify concealed parameters.

Instead new parameters have to be declared and propagated down the hierarchy. Parameters for initial conditions need to be introduced and propagated to start values or used in “initial sections”.

The procedure for developing an encrypted library is:

- The developer maintains an unencrypted library, which is easy to modify and easy to maintain in a version control system. All parts which should be concealed in the finished library must be declared as protected.
- When the unencrypted library has been finished for release double-click on the package to show it in Dymola
- If developing a licensed library add the following to the Modelica text in the Modelica text window: `annotation(Dymola(checkSum="", Library="MyLib"));`
- Select ‘File/Export/Encrypted model’ which produces the encrypted file `myPackage.moe`.
- The encrypted file, i.e., the `.moe` file, is distributed. The original `.mo` files for the encrypted parts are never distributed outside of the development group.

It is worth pointing out that external decrypting of a `.moe` file is not supported by Dymola, but all development work must be performed in the original unencrypted `.mo` file. In Dymola all encrypted files are by definition read-only.



## Using encrypted components

Dymola must not reveal any concealed information when encrypted components are used to compose a model and as well as at simulation (unless the library developers has decided otherwise). It means that some commands or operations are disabled or have modified effects or results. Also some diagnostics and error messages must be less informative.

Let us first discuss the use of encrypted components in Modeling mode.

### File menu

An important and basic restriction is that encrypted components are read-only and cannot be modified. The commands Save, Save All, Save As, and Save Total are not available for encrypted components. The Duplicate command is only available if duplication is explicitly enabled.

The commands Print, Export/Image and Export/Animation are not changed in the meaning that they output what is visible on the screen.

The command Open reads encrypted files in the usual way, when the file type “Encrypted Modelica files (\*.moe)” is selected. This file-type is visible for all users – not only the ones who have enabled encryption of models.

### Package and component browsers

Concealed classes are never shown in the package browser. The component browser does not show components or extends of a concealed class.

### Editor (graphics and text)

Encrypted models are read-only and concealed models are never visible in the editor. Dymola implements the following restrictions on what is shown in the graphical and textual layers of the editor.

- The icon layer is empty for concealed classes. Also, it does not show protected connectors (regardless of encryption). Note that these rules for the icon layer also apply to icons as they are shown in the diagram layer of some other class.
- The diagram layer is as default empty for encrypted classes (not even public ones). However, models enclosed in a package called “Examples” or “Tutorial” are shown as default.
- Modelica text (declarations and equations) is as default empty if the class is encrypted.
- The documentation layer is empty for concealed classes, but is otherwise shown.

The window title says “Encrypted” instead of “Read-Only” for all encrypted classes.

## Simulation mode

The aim of translating a model is to perform consistency checks and analyze and manipulate the equations to generate efficient code for simulation. This procedure is not affected by the fact that components are encrypted or concealed with the following natural modifications:

- Diagnostics and error messages during translation and simulation do as default not reveal concealed information. Warnings and error messages are issued as for non-encrypted models, but they may be less informative. An extreme is “Error in ConcealedEquation”.
- The generated simulation code as default prohibits storing, plotting or other access to simulation results for concealed variables by the use of their names.

## Examples

### Encrypted transfer function

To illustrate the basics of using an encrypted model component and how encryption changes error messages, let us develop a simple encrypted model and use it in some simple contexts.

The model `Modelica.Blocks.Continuous.TransferFunction` defines the transfer function between a scalar input,  $u$ , and a scalar output,  $y$ . Transfer functions may be realized in different ways. Assume that we have invented a new good way to realize transfer functions and that we have developed a new model `MyTransferFunction` that exploits our ideas. We have also decided to protect our intellectual property by encrypting the model `MyTransferFunction` before making it available to others.

The model `MyTransferFunction` may look like

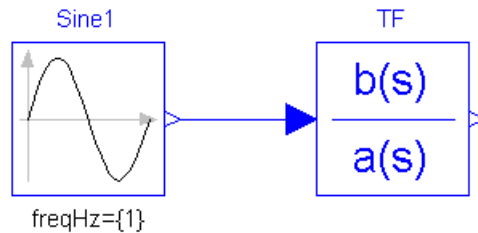
```
block MyTransferFunction "Linear transfer function"
  extends Modelica.Blocks.Interfaces.SISO;
  parameter Real b[:]={1} "Numerator coefficients.";
  parameter Real a[:]={1,1} "Denominator coefficients.";
protected
  Real x[size(a, 1) - 1] "State";
  parameter Integer na=size(a, 1);
  parameter Integer nb=size(b, 1);
  parameter Integer nx=size(a, 1) - 1;
  Real xldot;
  Real xn;
equation
  [der(x); xn] = [xldot; x];
  [u] = transpose([a])*[xldot; x];
  [y] = transpose([zeros(na - nb, 1); b])*[xldot; x];
end MyTransferFunction;
```

This is very similar to the model in the Modelica Standard Library. However, there is one very important difference. The model in the Modelica Standard Library, declares the state  $x$  in the public sections as

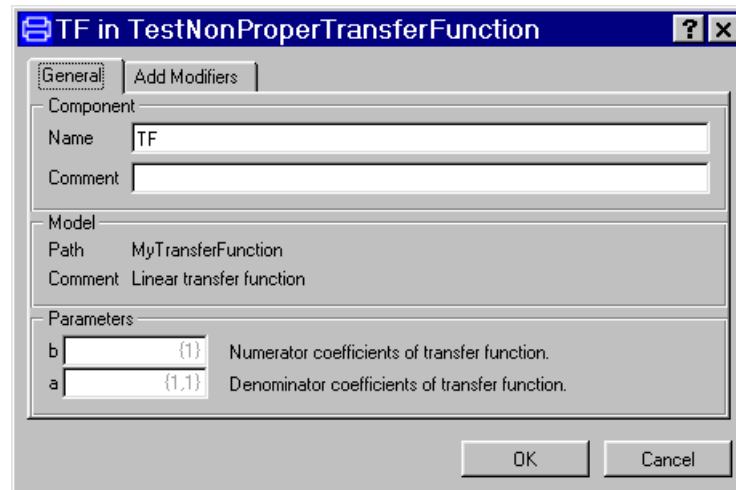
```
output Real x[size(a, 1) - 1] "State";
```

If it is possible to store the time trajectories of  $x$ , it is possible to find out how we realize the transfer function by simulating different transfer functions. In our MyTransferFunction the state is protected, which prevents users to store, plot or otherwise inspect the simulation results for the state.

Let us test the model by connecting the input to the source to the sine signal generator of the type Modelica.Block.Sources.Sine.

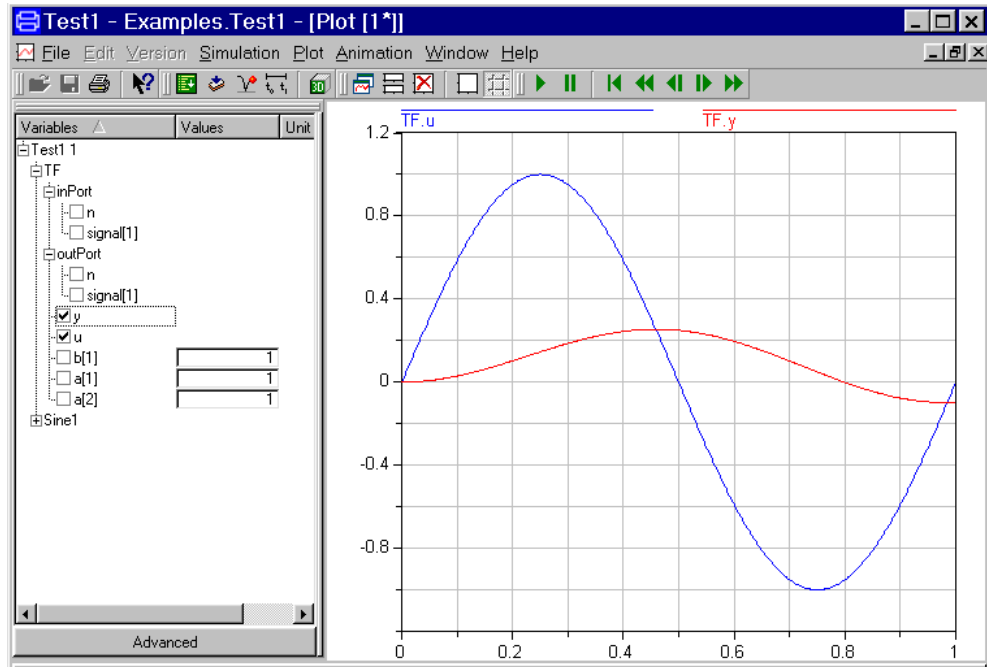


Such a model is built in the usual way by dragging and dropping components and connecting them together. The encrypted model MyTransferFunction is available in the package browser for dragging but it cannot be displayed or inspected in the editor. The connectors are public and thus available for connection. Selecting the component and clicking the right mouse button pops the context menu in the usual way and selecting the alternative Parameter displays



and it is possible to enter values for the coefficient parameters.

The result of a simulation is shown below Please; note that the state  $x$  components are not available in the plot selector.



The sine generator may produce multiple output signals, while the transfer function assumes a scalar input. Let us see what happens if we let the sine generator produce two signals. This can be achieved by setting the value of its parameter amplitude to  $\{1, 2\}$ .

Translation gives the error message

```
Error: The parts of
(Sine1.outPort.signal) = (TF.inPort.signal)
have incompatible sizes: [2] and [1]
```

```
Errors or failure to expand the equation:
Sine1.outPort.signal = TF.inPort.signal;
Errors or failure to expand vector or matrix expressions.
```

This error message does not reveal any concealed information. In fact the same error message is given also when MytransferFunction is not encrypted.

MyTransferFunction assumes that the transfer function is proper, i.e. the degree of the nominator polynomial is equal to or less than the degree of the denominator polynomial. As shown above the parameter  $a = \{1, 1\}$ . If we set  $b = \{1, 1, 1\}$  and translate, Dymola issues the error message:

```
Error: in concealed equation.
Errors or failure to expand vector or matrix expressions.
```

For a non-encrypted MyTransferFunction the error message is more informative

```
Error: Negative sizes in
```

```

zeros(TF.na-TF.nb, 1)
The sizes are: -1, 1
Errors or failure to expand the equation:
[TF.y]=transpose([zeros(TF.na-TF.nb,1);TF.b])*[TF.xldot;TF.x];
Found in class MyTransferFunction, MyTransferFunction.mo
at line 78, and used in component TF.
Errors or failure to expand vector or matrix expressions.

```

However, such an error message cannot be output for the encrypted version, because it reveals concealed information.

## Coupled clutches

We will use the example `Modelica.Mechanics.Rotational.Examples.CoupledClutches` and exchange components to illustrate various possibilities to provide or conceal information.

Let us make an encrypted package `ConcealedMechanics` where we put the components developed

First, let us just make an identical copy of `Modelica.Mechanics.Rotational.Inertia`, call it simply `Inertia`.

It is most simply done using `New/Model` to insert it into `ConcealedMechanics` and extending from and extending from `Modelica.Mechanics.Rotational.Inertia`

```

model Inertia
  extends Modelica.Mechanics.Rotational.Inertia;
end Inertia;

```

This model will reveal all public components

```

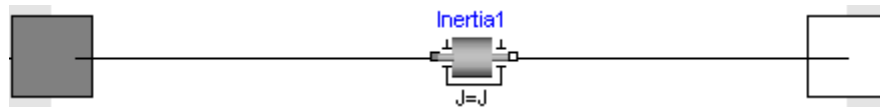
model Inertia "1D-rotational component with inertia"
  extends Interfaces.Rigid;
  parameter SIunits.Inertia J=1 "Moment of inertia";
  SIunits.AngularVelocity w "Absolute angular velocity";
  SIunits.AngularAcceleration a "Absolute acceleration";

```

We could restrict this by putting `w` and `a` in a protected section.

Another approach is to encapsulate the model and design a new interface. In Dymola we make a new model extending from `Modelica.Mechanics.Interfaces.TwoFlanges`.

We drag in a component of class `Modelica.Mechanics.Rotational.Inertia` and connect it.



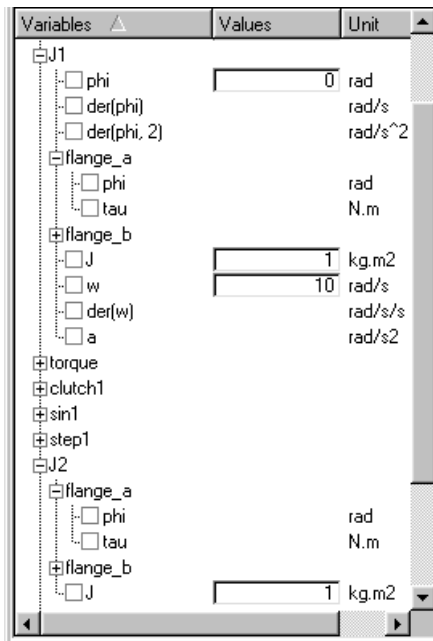
To declare a parameter `J` we select `Inertia1`, click the right mouse button, select the `Parameter` alternative and set its parameter `J` and in the right-click menu select `Propagate`. To make the component `Inertia1` protected, we once again click the right mouse button, select the `Attributes` alternative and check `Protected`. The resulting Modelica model is

```

model Inertia2
  extends Modelica.Mechanics.Rotational.Interfaces.TwoFlanges;
  parameter Modelica.SIunits.Inertia J=1 "Moment of inertia";
protected
  Modelica.Mechanics.Rotational.Inertia Inertial(J=J);
equation
  connect(Inertial.flange_a, flange_a);
  connect(Inertial.flange_b, flange_b);
end Inertia2;

```

Now we encrypt the package ConcealedMechanics. For the model CoupledClutches we let J1 be of class ConcealedMechanics.Inertia and J2 be of class ConcealedMechanics.J2. Simulation of the model CoupledClutches gives the following variable browser



For J1 and J2 it is possible to plot the connector variables and set the moment of inertia J.

However, for J1 it is also possible to plot velocity and acceleration. What to do if we would like to plot the velocity of J2? The velocity can be made available by connecting a Speed-Sensor

For J1 it is possible to set an initial value for w. For J2 the situation is more complex. By just looking at it we cannot tell whether there is some internal initialization. When translating the model, Dymola issues a warning that initial conditions are not fully specified. The documentation of Inertia2, needs thus to include documentation on initial conditions. In this case we know that there are no initial conditions are stated for J2, so we may introduce an initial equation section in the CoupledClutches model containing for example

```

initial equation

```

```
J2.flange_a.phi = ... "start angle";
der(J2.flange_a.phi) = ... "start velocity";
```

to specify the initial position and velocity of J2.

## Special annotations for concealment

These special annotations are all grouped inside:

```
annotation(Protection(...));
```

The annotations are designed based on the following basic principles:

- Security by default – the default is to not reveal information for encrypted packages. You as the library developer have to enable each of these flags.
- It is more important to protect an entire package from being viewed than individual classes in the package.
- Easy-to-use.
- Simple logic to make it easy to verify the behavior. Thus you can enable duplication, but hide diagrams even though this does not make sense.
- Only applied after encryption – thus they can be present in the original library.

The behavior can be summarized in the following table (the missing entries are not implemented):

Show/allow	Annotation		Default
	All classes	Non-packages	
Duplicate	allowDuplicate	nestedAllowDuplicate	false
Diagram	showDiagram	nestedShowDiagram	false(*)
Text(*)	showText	nestedShowText	false
Icon			true
Documentation			true
In class browser/ choices all matching	hideFromBrowser		false

The annotation applies hierarchically to all classes (unless overridden by a similar annotation).

Notes:

- false(\*) indicates that the default for Examples and Tutorial-packages is showDiagram=true.

- The text window has copying disabled (unless duplicate is allowed), but there are ways of circumventing this.
- The logic for the browser is reversed.

In addition there are several package-wide settings (also inside Protection) as follows:

Show/allow	Annotation	Default
Plotting of variables	showVariables	false
Diagnostics with variable	showDiagnostics	false
Statistics (e.g. #states)	showStatistics	false
Flat-modelica	showFlat	false

Note that if several encrypted packages are used they must all enable e.g. statistics for the statistics to be shown.

## Scrambling in Dymola

Encryption of a package/model is a useful way of making a package useable without revealing information. However, in certain scenarios it is not the ideal choice when sending one (or a few) component models that shall only be used directly.

In such cases the most important information to conceal is data and internal structure, and there is no need to keep ‘replaceable’ components or classes.

The ideal choice would in that case be to send something that:

- Does not contain internal structure and original data.
- Automatically hides all internal components.
- Can be used as any other model in Dymola (including differentiation for state-selection).
- Allows you to see exactly what is sent.

This is accomplished using ‘Export/Encrypted total model’ and can be done either on a model/block or for a package, where each public non-partial model/block is scrambled individually and then placed together in a package.

Each individual model is scrambled as explained in the next to remove unnecessary information and the resulting file is then encrypted as an additional safety precaution.

### Example of scrambling

We continue with the inertia example, but now rewrite the Inertia model by replacing the parameter ‘J’ by two variables ‘r’ and ‘m’ and computing the inertia based on these as follows:

```
model Inertia3
```



```

    extends Modelica.Mechanics.Rotational.Interfaces.Rigid;
    parameter Modelica.SIunits.Length r=1 "Radius";
protected
    constant Modelica.SIunits.Mass m=0.5 "Mass";
    Modelica.SIunits.AngularVelocity w;
    annotation (Documentation(info="<html>
An inertia of a certain shape with settable radius.
</html>"));
equation
    w = der(phi);
    m*r^2/12*der(w) = flange_a.tau + flange_b.tau;
end Inertia3;

```

The mass and the shape should be hidden from the user of the model. By selecting 'Export/Encrypted total model' the model is first scrambled and then encrypted.

The procedure gives the messages:

```

Will encrypt to file C:/dymola/work/Inertia3.moe.
First scrambling to file C:/dymola/work/Inertia3.tmp.mo.
Scrambling Inertia3.
The scrambling should preserve the following top-level
variables:
    connector flange_a
    connector flange_b
    parameter r
Scrambling complete, verifying it.
Encrypting.
Encryption complete, file can be found in
C:/dymola/work/Inertia3.moe.

```

The scrambling indicate which variables should be kept, and include a tag before the variable to explain why.

Users can examine the Inertia3.tmp.mo file to verify that the no vital information is present:

```

model Inertia3
encapsulated connector r0
Real phi(unit = "rad") "Absolute rotation angle of flange";
flow Real tau(unit = "N.m") "Cut torque in the flange";
    annotation(Hide=true, Coordsys(extent=[-100, -100; 100, 100],
grid=[2, 2], component=[20, 20]), Icon(Rectangle(extent=[-100,
-100; 100, 100], style(color=0, fillColor=10))));
    end r0;
r0 flange_a annotation (extent=[-110, -10; -90, 10]);
encapsulated connector r1
Real phi(unit = "rad") "Absolute rotation angle of flange";
flow Real tau(unit = "N.m") "Cut torque in the flange";
    annotation(Hide=true, Coordsys(extent=[-100, -100; 100, 100],
grid=[2, 2], component=[20, 20]), Icon(Rectangle(extent=[-100,
-100; 100, 100], style(color=0, fillColor=7))));
    end r1;
r1 flange_b annotation (extent=[90, -10; 110, 10]);
parameter Real r(unit = "m") = 1 "Radius";

```

```

protected
Real z1;
Real z2;
  annotation(Coordsys(extent=[-100, -100; 100, 100], grid=[2, 2],
component=[20, 20]), Documentation(info="<html>
An inertia of a certain shape with settable radius.
</html>"));
  protected equation
flange_a.phi = z1;
flange_b.phi = z1;
z2 = der(z1);
0.0416666666666667*r^2*der(z2) = flange_a.tau+flange_b.tau;
end Inertia3;

```

As can be seen the mass and shape have been constant-evaluated making it impossible to determine their individual values. In addition the names of all internal variables are replaced by scrambled names (if the variable is preserved at all).

The encrypted file only contains this information, but is in addition encrypted. Encryption prevents disclosure of even the scrambled information and also makes the model read-only.