OPTIMICA Compiler Toolkit

Version 1.48.1

OPTIMICA Compiler Toolkit: Version 1.48.1

Acknowledgements

This document is produced with DocBook 5 using XMLMind XML Editor for authoring, Norman Walsh's XSL stylesheets and a GNOME xsltproc + Apache fop toolchain. Math contents are converted from LaTeX using the TeX/LaTeX to MathML Online Translator by the Ontario Research Centre for Computer Algebra and processed by JEuclid.

Table of Contents

Chapter 1. Introduction

The OPTIMICA Compiler Toolkit (OCT) is a computational platform for model-based systems design leveraging the open standards Modelica and the Function Mock-up Interface (FMI), see Section 1.2. It offers a versatile environment for posing and solving dynamic and steady-state simulation and optimization problems throughout the product development process. The OPTIMICA Compiler Toolkit comes with a Modelica compiler with capabilities beyond dynamic simulation by offering unique features for optimization and steady-state computations.

The OPTIMICA Compiler Toolkit builds on Python and MATLAB® for user-interaction. User-friendly scripting APIs enable custom workflows to support flexible design flows that integrate sequences of computations.

In early stages of the design cycle, steady-state simulation and optimization support architectural exploration and selection. During detailed design, model fidelity is increased and transient simulation and steady-state performance computations are used to refine the design. Dynamic optimization is a tool to assess limits of performance during control systems design and is also a cornerstone in advanced control strategies such as non-linear model predictive control. The OPTIMICA Compiler Toolkit unifies simulation and optimization for transient and steady-state computations throughout the design process.

The OPTIMICA Compiler Toolkit offers the following main features:

- A Modelica compiler compliant with the Modelica language specification 3.4 (notable limitations: *Synchronous Language Elements* and *State Machines*, cf. Chapter 20) and supporting both the Modelica Standard Library version 3.2.3 build 3 as well as the Modelica Standard Library version 4.0.0. The compiler generates Functional Mock-up Units (FMUs), including Model Exchange and Co-simulation as well as version 1.0 and 2.0 of the FMI standard.
- Dynamic simulation algorithms for integration of large-scale and stiff systems. Algorithms include CVode and Radau.
- Dynamic optimization algorithms based on collocation for solving optimal control and estimation problems. Dynamic optimization problems are encoded in Optimica, an extension to Modelica.
- A derivative-free model calibration algorithm to estimate model parameters based on measurement data.
- Support for generation of source code FMUs.
- A non-linear solver for solving large-scale systems of equations arising, e.g., in steady-state applications. Efficient and robust steady-state problem formulation is enable by Hand Guided Tearing, which enables user specified selection of residuals and iteration variables.
- Support for encrypted and licensed Modelica libraries.
- Support for state-of-the-art numerical algorithms for dynamic optimization, notably the HSL solver MA57 that provides improved robustness and performance.

- A compiler API is available to extract information, e.g., packages, models, parameters and annotations, from Modelica libraries.
- Scripting APIs in Python and MATLAB® are available to script automation of compilation, simulation and optimization of Modelica and FMI models.

The OPTIMICA Compiler Toolkit is based on OCT technology.

1.1. License Options

The OPTIMICA Compiler Toolkit is available in two versions:

- OPTIMICA Compiler Toolkit Base version which supports compilation of FMUs and dynamic simulation in Python and MATLAB® (with the addition of FMI Toolbox for MATLAB®) as well as dynamic optimization based on open source solvers.
- OPTIMICA Compiler Toolkit Full version which in addition supports steady-state computations, including Hand-Guided Tearing in Modelica models and non-linear solver integration in MATLAB®.

In addition, there are two additional license options available:

- A license option for the linear solver MA57 for industrial grade dynamic optimization applications.
- A license option for generation of source code FMUs.

The OPTIMICA Compiler Toolkit is available for end users as well as for integration in software and custom tool-chains.

1.2. Modelica and the Functional Mock-up Interface

Modelica is a language where a system can be modeled by connecting components across many domains such as thermodynamics and automatic control. The components can be rapidly assembled by reusing components that are already tested and available in a rich set of libraries. The Modelica language and the Modelica Standard Library are actively developed by the Modelica Association, see http://www.modelica.org.

Modelica components are based on a mathematical model defined by hybrid differential-algebraic equations. This means that the model of the system can contain differential equations and also leverage other constraints on the variables, such as conservation of flow, as well as capture discrete events that can occur during simulation.

The easy reuse of components inherent to the language enables a clear separation of the equations that concern a modeler when looking at a component. The components are connected through well-defined connectors to ensure that only compatible components can be connected to each other. This mean that we can model a system by picking and choosing its components and then declare which components are connected without concerning ourselves with how they are connected.

To ease the exchange of components between users and tools, the components can be packaged in a Functional Mock-up Unit. A Functional Mock-up Unit(FMU) is a binary, such as a Modelica component in compiled form, that complies with the Functional Mock-up Interface standard. Any binary that complies with the standard can be used by tools that implement it. The Functional Mock-up Interface(FMI) standard specifies an interface for connecting components that is tool-independent. This means that any tool that implements the FMI standard can use the component to solve a simulation problem. It is maintained by the Modelica Association, see http://www.fmi-standard.org.

Components can be distributed in FMU form without having to reveal their source code. The public connectors and parameters of the components are declared in the FMU so that it can be simulated or connected to other components. The FMU can include only the model, which is referred to as an FMU for Model Exchange (FMU-ME). It can also include a solver and is then referred to as a FMU for Co-Simulation (FMU-CS).

Chapter 2. Installation

2.1. General Prerequisites

2.1.1. Supported platforms

The OPTIMICA Compiler Toolkit can be installed on the following platforms:

- Windows 10
- CentOS 7, 64-bit

2.1.2. C compiler

A C compiler is required by the Modelica compiler provided by OCT.

The supported C compilers for OCT are:

- *TDM-GCC (5.1.0)* is shipped with the OCT installer.
- *Microsoft Visual C++ 2010 (Visual C++ 10.0), express or professional*
- *Microsoft Visual C++ 2012 (Visual C++ 11.0), express or professional*
- *Microsoft Visual C++ 2015 (Visual C++ 14.0), express, community or professional*
- *Microsoft Visual C++ 2017 (Visual C++ 15.0), express, community or professional*
- *Microsoft Visual C++ 2019 (Visual C++ 16.0), community or professional*

Note that *Microsoft Visual C++ 2010 express* can't be used to compile 64 bit FMUs. Also note that the C++ compiler is not installed by default when installing Microsoft Visual Studio. It needs to be chosen manually.

2.2. Installation of OPTIMICA Compiler Toolkit

To install the OPTIMICA Compiler Toolkit start by double-clicking `OCT-X.X.exe` and follow the instructions of the setup wizard.

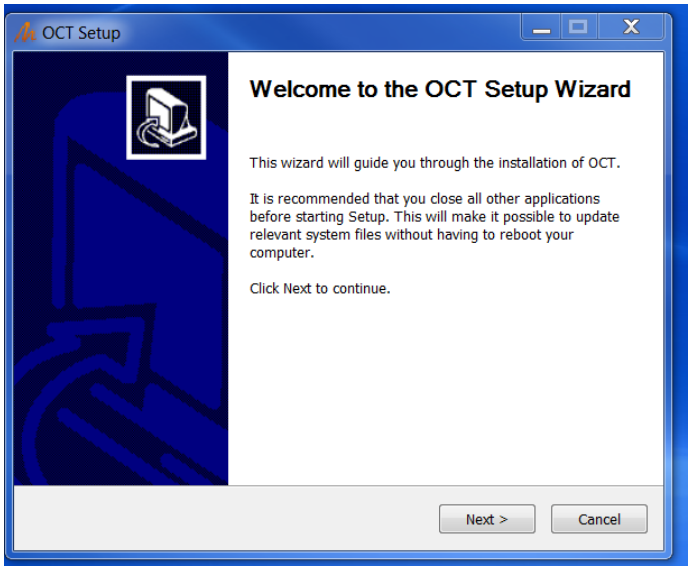


Figure 2.1 Start page of the installation wizard.

When the wizard has finished OCT has been installed and the Python interface is ready to be used. All Python packages and other thirdparty dependencies are listed in Section C.1.To be able run from MATLAB®, follow the instructions in Section 2.2.1.

2.2.1. Setting up the MATLAB® Interface

Make sure the prerequisites defined below are met before enabling the toolbox as described in Section 2.2.1.2.

2.2.1.1. Prerequisites

Supported MATLAB® versions

The OCT Modelica toolbox for MATLAB® is supported only for 64 bit versions of MATLAB® from R2014b (MATLAB 8.4) and later. The OCT Modelica toolbox for MATLAB® relies on FMI Toolbox, see the section called “FMI Toolbox”, and hence on its supported MATLAB® versions.

FMI Toolbox

To be able to solve steady-state problems as well as simulate dynamic systems in MATLAB® FMI Toolbox is required. Supported versions are FMI Toolbox 2.4 and newer.

(Optional) Dymola

Dymola 2014 FD01 or later is required to use the MATLAB® interface for the Dymola Modelica compiler.

2.2.1.2. Enabling the toolbox in MATLAB®

First, the toolbox must be added to the MATLAB® search path. *This only needs to be done once*, see Section 2.2.1.3 for instructions.

Before using the toolbox functions, an initialization script, `oct.initOCT`, must be run. *This must be done every time a new MATLAB® session is started*, see Section 2.2.1.4 for more details.

2.2.1.3. Adding the toolbox to MATLAB® search path

Follow these steps to enable the OCT Modelica toolbox in MATLAB®.

1. Start MATLAB®.

2. Open the **Set Path** dialog box, either from the menu or ribbon in the GUI or by typing `pathtool` in the MATLAB® command window.

3. Select the button **Add folder...**

4. Browse to the OCT Modelica toolbox for MATLAB® installation folder (the folder where the OPTIMICA Compiler Toolkit was installed and therefrom "`install/MATLAB`").

5. Select the "`toolbox`" folder found in "`MATLAB`" and click **OK** followed by **Save** and **Close**.

2.2.1.4. Initializing the toolbox in MATLAB®

In order to use the toolbox, the initialization script `oct.initOCT` must be executed at MATLAB® startup. The script adds some JAR files that are needed to run the toolbox to the Java class path in MATLAB®. Since the script must be run for every startup of MATLAB®, it is recommended to add the call to `oct.initOCT` either to your user specific `startup.m` file or system-wide `matlabrc.m` file. In the MATLAB® console, type `userpath` to query the directory in which startup.m should be located. Note that '`clear`' is called as a part of the `oct.initOCT` script.

The `oct.initOCT` script will throw an exception if a JAR file can not be found.

If the `DYMOLA_INSTALL_DIR` environment variable has not been set (see Section 7.2.1.2 for an example how to do this), the JAR file needed for the Dymola compilation will not be loaded. In this case, you will see a warning in the MATLAB® command window. Before the environment variable `DYMOLA_INSTALL_DIR` has been set, it is not possible to use Dymola for the compilation. Note that the `oct.initOCT` script must always be run after `DYMOLA_INSTALL_DIR` is set. Therefore, when using Dymola it is recommended that the environment variable `DYMOLA_INSTALL_DIR` is also set in your user specific `startup.m` file or system-wide `matlabrc.m` file before the call to `oct.initOCT`.

The content of a `startup.m` file that enables Dymola compilation and runs the initialization script `oct.initOCT` could look like:

```
setenv('DYMOLA_INSTALL_DIR','C:\Program Files (x86)\Dymola 2014 FD01')
oct.initOCT
```

Chapter 3. Getting started

This chapter is intended to give a brief introduction to using the OCT Python packages and the MATLAB® interface and will therefore not go into any details. Please refer to the other chapters of this manual for more information on each specific topic.

3.1. The OCT Python packages

The OCT Python interface enables users to use Python scripting to interact with Modelica and Optimica models. The interface consists of three packages:

- **PyModelica** Interface to the compilers. Compile Modelica and Optimica code into model units, FMUs. See Chapter 4 for more information.
- **PyFMI** Work with models that have been compiled into FMUs (Functional Mock-up Units), perform simulations, parameter manipulation, plot results etc. See Chapter 5 for more information.
- **PyJMI** Work with models that are represented in symbolic form based on the automatic differentiation tool CasADi. This package is mainly used for solving optimization problems. See Chapter 6 for more information.

3.1.1. Starting a Python session

There are three different Python shells available under the OCT start menu.

- **Python** Normal command shell started with Python.
- **IPython** Interactive shell for Python with, for example, code highlighting and tab completion.
- **pylab** IPython shell which also loads the numeric computation environment PyLab.

It is recommended to use either the IPython or pylab shell.

3.1.1.1. Running an example

The Python packages `pyfmi` and `pyjmi` each contain a folder called `examples` in which there are several Python example scripts. The scripts demonstrate compilation, loading and simulation or optimization of models. The corresponding model files are located in the subdirectory `files`. The following code demonstrates how to run such an example. First a Python session must be started, see Section 3.1.1 above. The example scripts are preferably run in the pylab Python shell.

The following code will run the RLC example and plot some results.

```
# Import the RLC example
```

```
from pyjmi.examples import RLC

# Run the RLC example and plot results
RLC.run_demo()
```

Open `RLC.py` in a text editor and look at the Python code to see what happens when the script is run.

3.2. The OCT MATLAB® Interface

The OCT compiler can be used from within MATLAB®, both for compilation of dynamic models as well as models intended for steady-state initialization, see Chapter 7.

Dynamic models can be simulated in MATLAB® through FMI Toolbox, see Chapter 8. A basic example for solving steady-state problems is provided in Section 3.2.1, further details are found in Chapter 9.

3.2.1. Running a steady-state example

In `install/MATLAB/examples/nlesol/example_SimpleSteadyState.m` an example is given of how to compile and solve a simple steady state model.

Open the file in the MATLAB® Editor and run the example. The solver trace, i.e., the steps of the Newton solver, will be displayed in the command window. The compiled FMU will be present in the same folder as the example file.

```
    >> example_SimpleSteadyState

Model name.......................: ExampleModels.SimpleSteadyState
Number of iteration variables......: 1
Number of discontinuity switches...: 2

Switch iteration 1
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   1Js   1.0000e+00  1.0000e+00:   1     0    0   2.0000e-01:   1r  2.0000e-01
   2     8.0000e-01  8.0000e-01:   1     0    0   2.0000e-01:   1r  2.0000e-01
   3     6.4000e-01  6.4000e-01:   1     0    0   2.0000e-01:   1r  2.0000e-01
   4     5.1200e-01  5.1200e-01:   1     0    0   2.0000e-01:   1r  2.0000e-01
   5     4.0960e-01  4.0960e-01:   1     0    0   2.4414e-01:   1r  2.4414e-01
   6     3.0960e-01  3.0960e-01:   1     0    0   3.2300e-01:   1r  3.2300e-01
   7     2.0960e-01  2.0960e-01:   1     0    0   4.7710e-01:   1r  4.7710e-01
   8     1.0960e-01  1.0960e-01:   1     0    0   9.1241e-01:   1r  9.1241e-01
   9     9.6000e-03  9.6000e-03:   1     0    0   1.0000e+00        1.0000e+00
  10     0.0000e+00  0.0000e+00:   1
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   1s    0.0000e+00  0.0000e+00:   1
Switch iteration 2
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   1     6.6667e-01  6.6667e-01:   1     0    0   3.0000e-01:   1r  3.0000e-01
```

```
   2     6.0000e-01  6.0000e-01:   1     0    0   3.3333e-01:   1r  3.3333e-01
   3     5.3333e-01  5.3333e-01:   1     0    0   3.7500e-01:   1r  3.7500e-01
   4     4.6667e-01  4.6667e-01:   1     0    0   4.2857e-01:   1r  4.2857e-01
   5     4.0000e-01  4.0000e-01:   1     0    0   5.0000e-01:   1r  5.0000e-01
   6     3.3333e-01  3.3333e-01:   1     0    0   6.0000e-01:   1r  6.0000e-01
   7     2.6667e-01  2.6667e-01:   1     0    0   9.0000e-01:   1r  9.0000e-01
   8     1.8667e-01  1.8667e-01:   1     0    0   1.0000e+00         1.0000e+00
   9     1.2444e-01  1.2444e-01:   1     0    0   1.0000e+00         1.0000e+00
  10J    8.2963e-02  8.2963e-02:   1     0    0   1.0000e+00         1.0000e+00
  11     2.8553e-10 -2.8553e-10:   1     0    0   1.0000e+00         1.0000e+00
  12     0.0000e+00  0.0000e+00:   1
Switch iteration 3
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   1     6.6667e-01 -6.6667e-01:   1     0    0   2.0000e-01:   1r  2.0000e-01
   2     2.6667e-01 -2.6667e-01:   1     0    0   4.0000e-01:   1r  4.0000e-01
   3     5.3333e-02  5.3333e-02:   1     0    0   1.0000e+00         6.0000e-01
   4     4.2667e-02 -4.2667e-02:   1     0    0   1.0000e+00         6.0000e-01
   5     3.4133e-02  3.4133e-02:   1     0    0   1.0000e+00         6.0000e-01
   6     2.7307e-02 -2.7307e-02:   1     0    0   1.0000e+00         6.0000e-01
   7     2.1845e-02  2.1845e-02:   1     0    0   1.0000e+00         6.0000e-01
   8     1.7476e-02 -1.7476e-02:   1     0    0   1.0000e+00         6.0000e-01
   9     1.3981e-02  1.3981e-02:   1     0    0   1.0000e+00         6.0000e-01
  10J    1.1185e-02 -1.1185e-02:   1     0    0   1.0000e+00         1.0000e+00
  11     7.7968e-11  7.7968e-11:   1     0    0   1.0000e+00         1.0000e+00
  12     7.4015e-17  7.4015e-17:   1
Number of function evaluations: 56
Number of jacobian evaluations: 3
Solver finished
Total time in solver: 0.56 s
```

Chapter 4. Working with Models in Python

4.1. Introduction to models

Modelica and Optimica models can be compiled and loaded as model objects using the OCT Python interface. These model objects can be used for both simulation and optimization purposes. This chapter will cover how to compile Modelica and Optimica models, set compiler options, load the compiled model in a Python model object and use the model object to perform model manipulations such as setting and getting parameters.

4.1.1. The different model objects in OCT

There are several different kinds of model objects that can be created with OCT: `FMUModel(ME/CS)(1/2)` (i.e. FMUModelME1, FMUModelCS1, FMUModelME2, and FMUModelCS2) and `OptimizationProblem`. The `FMUModel(ME/CS)(1/2)` is created by loading an *FMU* (Functional Mock-up Unit), which is a compressed file compliant with the FMI (Functional Mock-up Interface) standard. The `OptimizationProblem` is created by transferring an optimization problem into the CasADi-based optimization tool chain.

FMUs are created by compiling Modelica models with OCT, or any other tool supporting FMU export. OCT supports export of FMUs for Model Exchange (FMU-ME) and FMUs for Co-Simulation (FMU-CS), version 1.0 and 2.0. Import of FMU-CS version 2.0 is also supported. Generated FMUs can be loaded in an `FMUModel(ME/CS)(1/2)` object in Python and then be used for simulation purposes. Optimica models can not be compiled into FMUs.

`OptimizationProblem` objects for CasADi optimization do not currently have a corresponding file format, but are transferred directly from the OCT compiler, based on Modelica and Optimica models. They contain a symbolic representation of the optimization problem, which is used with the automatic differentiation tool CasADi for optimization purposes. Read more about CasADi and how an `OptimizationProblem` object can be used for optimization in Section 6.5 in Chapter 6.

4.2. Compilation

This section brings up how to compile a model to an FMU-ME / FMU-CS. Compiling a model to an FMU-ME / FMU-CS will be demonstrated in Section 4.2.1 and Section 4.2.2 respectively.

For more advanced usage of the compiler functions, there are compiler options and arguments which can be modified. These will be explained in Section 4.2.4.

4.2.1. Simple FMU-ME compilation example

The following steps compile a model to an FMU-ME version 2.0:

1. Import the OCT compiler function `compile_fmu` from the package `pymodelica`.

2. Specify the model and model file.

3. Perform the compilation.

This is demonstrated in the following code example:

```
# Import the compiler function
from pymodelica import compile_fmu

# Specify Modelica model and model file (.mo or .mop)
model_name = 'myPackage.myModel'
mo_file = 'myModelFile.mo'

# Compile the model and save the return argument, which is the file name of the FMU
my_fmu = compile_fmu(model_name, mo_file)
```

There is a compiler argument `target` that controls whether the model will be exported as an FMU-ME or FMU-CS. The default is to compile an FMU-ME, so `target` does not need to be set in this example. The compiler argument `version` specifies if the model should be exported as an FMU 1.0 or 2.0. As the default is to compile an FMU 2.0, `version` does not need to be set either in this example. To compile an FMU 1.0, `version` should be set to `'1.0'`.

Once compilation has completed successfully, an FMU-ME 2.0 will have been created on the file system. The FMU is essentially a compressed file archive containing the files created during compilation that are needed when instantiating a model object. Return argument for `compile_fmu` is the file path of the FMU that has just been created, this will be useful later when we want to create model objects. More about the FMU and loading models can be found in Section 4.3.

In the above example, the model is compiled using default arguments and compiler options - the only arguments set are the model class and file name. However, `compile_fmu` has several other named arguments which can be modified. The different arguments, their default values and interpretation will be explained in Section 4.2.4.

4.2.2. Simple FMU-CS compilation example

The following steps compiles a model to an FMU-CS version 2.0:

1. Import the OCT compiler function `compile_fmu` from the package `pymodelica`.

2. Specify the model and model file.

3. Set the argument `target = 'cs'`

4. Perform the compilation.

This is demonstrated in the following code example:

```
# Import the compiler function
from pymodelica import compile_fmu

# Specify Modelica model and model file (.mo or .mop)
model_name = 'myPackage.myModel'
mo_file = 'myModelFile.mo'

# Compile the model and save the return argument, which is the file name of the FMU
my_fmu = compile_fmu(model_name, mo_file, target='cs')
```

In a Co-Simulation FMU, the integrator for solving the system is contained within the FMU. With an FMU-CS exported with OCT, four different solvers are supported: CVode, Explicit Euler, Runge-Kutta (2nd order) and Radau5.

4.2.3. Compiling from libraries

The model to be compiled might not be in a standalone `.mo` file, but rather part of a library consisting of a directory structure containing several Modelica files. In this case, the file within the library that contains the model should *not* be given on the command line. Instead, the entire library should to added to the list of libraries that the compiler searches for classes in. This can be done in several ways (here *library directory* refers to the top directory of the library, which should have the same name as the top package in the library):

- Giving the path to the library directory in the `file_name` argument of the compilation function. This allows adding a specific library to the search list (as opposed to adding all libraries in a specific directory).
- Set the directory containing a library directory via the keyword argument named `modelicapath` of the compilation function. This allows for adding several libraries to the search list. For example if `modelicapath=C:\MyLibs`, then the compiler sets MODELICAPATH to `C:\MyLibs` and all libraries within are added to the search list during compilation.

The Modelica Standard Library (MSL) that is included in the installation is loaded by default when starting the OCT Python shell. The version of MSL that is loaded is based on the compiler option named *msl_version* or any existing *uses* annotations within the model being compiled.

The Python code example below demonstrates these methods:

```
# Import the compiler function
from pymodelica import compile_fmu

# Compile an example model from the MSL
fmu1 = compile_fmu('Modelica.Mechanics.Rotational.Examples.First')

# Compile an example model utilizing the modelicapath keyword argument assuming
# the library MyLibrary is located in C:/MyLibs, i.e. C:/MyLibs/MyLibrary exists
fmu2 = compile_fmu('MyLibrary.MyModel, modelicapath = 'C:/MyLibs')

# Compile a model from the library MyLibrary, located in C:\MyLibs
fmu3 = compile_fmu('MyLibrary.MyModel', 'C:/MyLibs/MyLibrary')
```

4.2.4. Compiler settings

The compiler function arguments can be listed with the interactive help in Python. The arguments are explained in the corresponding Python *docstring* which is visualized with the interactive help. This is demonstrated for `compile_fmu` below. The docstring for any other Python function for can be displayed in the same way.

4.2.4.1. compile_fmu arguments

The `compile_fmu` arguments can be listed with the interactive help.

```
# Display the docstring for compile_fmu with the Python command 'help'
from pymodelica import compile_fmu
help(compile_fmu)
Help on function compile_fmu in module pymodelica.compiler:

compile_fmu(class_name, file_name = [], compiler = 'auto', target = 'me', version = '2.0',
            platform = 'auto', compiler_options = {}, compile_to = '.', compiler_log_level
 = 'warning',
            modelicapath = '', separate_process = True, jvm_args = ''):
Compile a Modelica model to an FMU.

A model class name must be passed, all other arguments have default values.
The different scenarios are:

* Only class_name is passed:
    - Class is assumed to be in MODELICAPATH.

* class_name and file_name is passed:
    - file_name can be a single path as a string or a list of paths
      (strings). The paths can be file or library paths.
    - Default compiler setting is 'auto' which means that the appropriate
      compiler will be selected based on model file ending, i.e.
      ModelicaCompiler if a .mo file and OptimicaCompiler if a .mop file is
      found in file_name list.

The compiler target is 'me' by default which means that the shared
file contains the FMI for Model Exchange API. Setting this parameter to
'cs' will generate an FMU containing the FMI for Co-Simulation API.

Parameters::

    class_name --
        The name of the model class.

    file_name --
        A path (string) or paths (list of strings) to model files and/or
        libraries.
        If file does not exist, an exception of type
 pymodelica.compiler_exceptions.PyModelicaFileError is raised.
        Default: Empty list.
```

```
   compiler --
       The compiler used to compile the model. The different options are:
         - 'auto': the compiler is selected automatically depending on
            file ending
         - 'modelica': the ModelicaCompiler is used
         - 'optimica': the OptimicaCompiler is used
       Default: 'auto'

   target --
       Compiler target. Possible values are 'me', 'cs' or 'me+cs'.
       Default: 'me'

   version --
       The FMI version. Valid options are '1.0' and '2.0'.
       Default: '2.0'

   platform --
       Set platform, controls whether a 32 or 64 bit FMU is generated. This
       option is only available for Windows.
       Valid options are:
         - 'auto': platform is selected automatically. This is the only
           valid option for linux and darwin.
         - 'win32': generate a 32 bit FMU
         - 'win64': generate a 64 bit FMU
       Default: 'auto'

   compiler_options --
       Options for the compiler.
       Default: Empty dict.

   compile_to --
       Specify target file or directory. If file, any intermediate directories
       will be created if they don't exist. Furthermore, the Modelica model will
       be renamed to this name. If directory, the path given must exist and the model
       will keep its original name.
       Default: Current directory.

   compiler_log_level --
       Set the logging for the compiler. Takes a comma separated list with
       log outputs. Log outputs start with a flag :'warning'/'w',
       'error'/'e', 'verbose'/'v', 'info'/'i' or 'debug'/'d'. The log can
       be written to file by appending a colon and file name.
       Example: compiler_log_level='d:debug.txt', sets the log level to debug and writes
the log
       to a file named 'debug.txt'.
       Default: 'warning'

   modelicapath --
       Set the MODELICAPATH to use. Depending on compiler options one or
       more versions of MSL may be added to it as well.
```

```
    separate_process --
        Run the compilation of the model in a separate process.
        Checks the environment variables (in this order):
            1. SEPARATE_PROCESS_JVM
            2. JAVA_HOME
        to locate the Java installation to use.
        For example (on Windows) this could be:
            SEPARATE_PROCESS_JVM = C:\Program Files\Java\jdk1.6.0_37
        Default: True

    jvm_args --
        String of arguments to be passed to the JVM when compiling in a
        separate process.
        Default: Empty string


Returns::

    A compilation result, represents the name of the FMU which has been
    created and a list of warnings that was raised.
```

4.2.4.2. Compiler options

Compiler options can be modified using the `compile_fmu` argument `compiler_options`. This is shown in the example below.

```
# Compile with the compiler option 'enable_variable_scaling' set to True

# Import the compiler function
from pymodelica import compile_fmu

# Specify model and model file
model_name = 'myPackage.myModel'
mo_file = 'myModelFile.mo'

# Compile
my_fmu = compile_fmu(model_name, mo_file,
 compiler_options={"enable_variable_scaling":True})
```

There are four types of options: `string`, `real`, `integer` and `boolean`. The complete list of options can be found in Appendix B.

4.3. Loading models

Compiled models, FMUs, are loaded in the OCT Python interface with the `FMUModel(ME/CS) (1/2)` class from the `pyfmi` module, while optimization problems for the CasADi-based optimization are transferred directly into the `OptimizationProblem` class from the `pyjmi` module. This will be demonstrated in Section 4.3.1 and Section 4.3.2.

The model classes contain many methods with which models can be manipulated after instantiation. Among the most important methods are `initialize` and `simulate`, which are used when simulating. These are explained in Chapter 5 and Chapter 6. For more information on how to use the OptimizationProblem for optimization purposes, see Chapter 6. The more basic methods for variable and parameter manipulation are explained in Section 4.4.

4.3.1. Loading an FMU

An FMU file can be loaded in OCT with the method `load_fmu` in the `pyfmi` module. The following short example demonstrates how to do this in a Python shell or script.

```
# Import load_fmu from pyfmi
from pyfmi import load_fmu
myModel = load_fmu('myFMU.fmu')
```

`load_fmu` returns a class instance of the appropriate FMU type which then can be used to set parameters and used for simulations.

4.3.2. Transferring an Optimization Problem

An optimization problem can be transferred directly from the compiler in OCT into the class `OptimizationProblem` in the `pyjmi` module. The transfer is similar to the combined steps of compiling and then loading an FMU. The following short example demonstrates how to do this in a Python shell or script.

```
# Import transfer_optimization_problem
from pyjmi import transfer_optimization_problem

# Specify Modelica model and model file
model_name = 'myPackage.myModel'
mo_file = 'myModelFile.mo'

# Compile the model, return argument is an OptimizationProblem
myModel = transfer_optimization_problem(model_name, mo_file)
```

4.4. Changing model parameters

Model parameters can be altered with methods in the model classes once the model has been loaded. Some short examples in Section 4.4.1 will demonstrate this.

4.4.1. Setting and getting parameters

The model parameters can be accessed via the model class interfaces. It is possible to set and get one specific parameter at a time or a whole list of parameters.

The following code example demonstrates how to get and set a specific parameter using an example FMU model from the `pyjmi.examples` package.

```
# Compile and load the model
```

```
from pymodelica import compile_fmu
from pyfmi import load_fmu
my_fmu = compile_fmu('RLC_Circuit','RLC_Circuit.mo')
rlc_circuit = load_fmu(my_fmu)

# Get the value of the parameter 'resistor.R' and save the result in a variable
 'resistor_r'
resistor_r = rlc_circuit.get('resistor.R')

# Give 'resistor.R' a new value
resistor_r = 2.0
rlc_circuit.set('resistor.R', resistor_r)
```

The following example demonstrates how to get and set a list of parameters using the same example model as above. The model is assumed to already be compiled and loaded.

```
# Create a list of parameters, get and save the corresponding values in a variable 'values'
vars = ['resistor.R', 'resistor.v', 'capacitor.C', 'capacitor.v']
values = rlc_circuit.get(vars)

# Change some of the values
values[0] = 3.0
values[3] = 1.0
rlc_circuit.set(vars, values)
```

4.5. Debugging models

The OCT compilers can generate debugging information in order to facilitate localization of errors. There are three mechanisms for generating such diagnostics: dumping of debug information to the system output, generation of HTML code that can be viewed with a standard web browser or logs in XML format from the non-linear solver.

4.5.1. Compiler logging

The amount of logging that should be output by the compiler can be set with the argument `compiler_log_level` to the compile-functions (`compile_fmu` and also `transfer_optimization_problem`). The available log levels are `'warning'` (default), `'error'`, `'info'`,`'verbose'` and `'debug'` which can also be written as `'w'`, `'e'`, `'i'`,`'v'` and `'d'` respectively. The following example demonstrates setting the log level to `'info'`:

```
# Set compiler log level to 'info'
compile_fmu('myModel', 'myModels.mo', compiler_log_level='info')
```

The log is printed to the standard output, normally the terminal window from which the compiler is invoked.

The log can also be written to file by appending the log level flag with a colon and file name. This is shown in the following example:

```
# Set compiler log level to info and write the log to a file log.txt
compile_fmu('myModel', 'myModels.mo', compiler_log_level='i:log.txt')
```

It is possible to specify several log outputs by specifying a comma separated list. The following example writes log warnings and errors (log level `'warning'` or `'w'`) to the standard output and a more verbose logging to file (log level `'info'` or `'i'`):

```
# Write warnings and errors to standard output and the log with log level info to log.txt
compile_fmu('myModel', 'myModels.mo', compiler_log_level= 'w,i:log.txt')
```

4.5.2. Runtime logging

Runtime logging refers to logging of data during simulation, this section outlines some methods to retrieve simulation data from an FMU.

4.5.2.1. Setting log level

Many events that occur inside of an FMU can generate log messages. The log messages from the runtime are saved in a file with the default name `<FMU name>_log.txt`. A log file name can also be supplied when loading an FMU, this is shown in the example below:

```
# Load model
model = load_fmu(fmu_name, log_file_name='MyLog.txt')
```

How much information that is output to the log file can be controlled by setting the `log_level` argument to `load_fmu`. `log_level` can be any number between 0 and 7, where 0 means no logging and 7 means the most verbose logging. The log level can also be changed after the FMU has been loaded with the function `set_log_level(level)`. Setting the `log_level` is demonstrated in the following example:

```
# Load model and set log level to 5
model = load_fmu(fmu_name, log_level=5)

# Change log level to 7
model.set_log_level(7)
```

If the loaded FMU is an FMU exported by OCT, the amount of logging produced by the FMU can also be altered. This is done by setting the parameter `_log_level` in the FMU. This log level ranges from 0 to 7 where 0 represents the least verbose logging and 7 the most verbose. The following example demonstrates this:

```
# Load model (with default log level)
model = load_fmu(fmu_name)

# Set amount of logging produced to the most verbose
model.set('_log_level', 6)

# Change log level to 7 to be able to see everything that is being produced
model.set_log_level(7)
```

4.5.2.2. Interpreting logs from FMUs produced by OCT

In OCT, information is logged in XML format, which ends up mixed with FMI Library output in the resulting log file. Example: (the following examples are based on the example `pyjmi.examples.logger_example`.)

```
  1  ...
  2 FMIL: module = FMICAPI, log level = 5: Calling fmiInitialize
  3 FMIL: module = Model, log level = 4: [INFO][FMU status:OK] <EquationSolve>Model
 equations evaluation invoked at<value name="t">        0.0000000000000000E+00</value>
  4 FMIL: module = Model, log level = 4: [INFO][FMU status:OK]
 <BlockEventIterations>Starting block (local) event iteration at<value name="t">
 0.0000000000000000E+00</value>in<value name="block">0</value>
  5 FMIL: module = Model, log level = 4: [INFO][FMU status:OK]     <vector name="ivs">
   0.0000000000000000E+00,         0.0000000000000000E+00,         0.0000000000000000E+00</
vector>
  6 FMIL: module = Model, log level = 4: [INFO][FMU status:OK]     <vector name="switches">
        0.0000000000000000E+00,         0.0000000000000000E+00,         0.0000000000000000E
+00,         0.0000000000000000E+00</vector>
  7 FMIL: module = Model, log level = 4: [INFO][FMU status:OK]     <vector
 name="booleans"></vector>
  8 FMIL: module = Model, log level = 4: [INFO][FMU status:OK]     <BlockIteration>Local
 iteration<value name="iter">1</value>at<value name="t">        0.0000000000000000E+00</
value>
  9 FMIL: module = Model, log level = 4: [INFO][FMU status:OK]
 <JacobianUpdated><value name="block">0</value>
 10 FMIL: module = Model, log level = 4: [INFO][FMU status:OK]         <matrix
 name="jacobian">
 11 FMIL: module = Model, log level = 4: [INFO][FMU status:OK]
 -1.0000000000000000E+00,         4.0000000000000000E+00,          0.0000000000000000E+00;
 12 FMIL: module = Model, log level = 4: [INFO][FMU status:OK]
 -1.0000000000000000E+00,        -1.0000000000000000E+00,         -1.0000000000000000E+00;
 13 FMIL: module = Model, log level = 4: [INFO][FMU status:OK]
 -1.0000000000000000E+00,         1.0000000000000000E+00,         -1.0000000000000000E+00;
 14 FMIL: module = Model, log level = 4: [INFO][FMU status:OK]         </matrix>
 15 FMIL: module = Model, log level = 4: [INFO][FMU status:OK]       </JacobianUpdated>
 16 ...
```

The log can be inspected manually, using general purpose XML tools, or parsed using the tools in `pyfmi.common.log`. A pure XML file that can be read by XML tools can be extracted with

```
# Generate an XML file from the simulation log that was generated by model.simulate()
model.extract_xml_log()
```

The XML contents can also be parsed directly:

```
# Parse an XML log
log = pyfmi.common.log.parse_fmu_xml_log(log_file_name)
```

`log` will correspond to the top level log node, containing all other nodes. Log nodes have two kinds of children: named (with a `name` attribute in the XML file) and unnamed (without).

- Named children are accessed by indexing with a string: `node['t']`, or simply dot notation: `node.t`.
- Unnamed children are accessed as a list `node.nodes`, or by iterating over the node.

There is also a convenience function `gather_solves` to extract common information about equation solves in the log. This function collects nodes of certain types from the log and annotates some of them with additional named children. The following example is from pyjmi.examples.logger_example:

```
 1 # Parse the entire XML log
 2 log = pyfmi.common.log.parse_fmu_xml_log(log_file_name)
 3 # Gather information pertaining to equation solves
 4 solves = pyjmi.log.gather_solves(log)
 5
 6 print('Number of solver invocations:',                      len(solves))
 7 print('Time of first solve:',                               solves[0].t)
 8 print('Number of block solves in first solver invocation:', len(solves[0].block_solves)
 9 print('Names of iteration variables in first block solve:',
solves[0].block_solves[0].variables))
10 print('Min bounds in first block solve:',
solves[0].block_solves[0].min)
11 print('Max bounds in first block solve:',
solves[0].block_solves[0].max)
12 print('Initial residual scaling in first block solve:',
solves[0].block_solves[0].initial_residual_scaling)
13 print('Number of iterations in first block solve:',
len(solves[0].block_solves[0].iterations)
14 print('\n')
15 print('First iteration in first block solve: ')
16 print('  Iteration variables:',
solves[0].block_solves[0].iterations[0].ivs)
17 print('  Scaled residuals:',
solves[0].block_solves[0].iterations[0].residuals)
18 print('  Jacobian:\n',
solves[0].block_solves[0].iterations[0].jacobian)
19 print('  Jacobian updated in iteration:',
solves[0].block_solves[0].iterations[0].jacobian_updated)
20 print('  Residual scaling factors:',
solves[0].block_solves[0].iterations[0].residual_scaling)
21 print('  Residual scaling factors_updated:',
solves[0].block_solves[0].iterations[0].residual_scaling_updated)
22 print('  Scaled residual norm:',
solves[0].block_solves[0].iterations[0].scaled_residual_norm)
```

4.5.3. Compiler Diagnostic Output

By setting the compiler option `generate_html_diagnostics` to true, a number of HTML pages containing diagnostics are generated. The HTML files are generated in the directory `Model_Name_diagnostics`, where `Model_Name` is the name of the compiled model. As compared to the diagnostics generated by the `compiler_log_level` argument, the HTML diagnostics contains only the most important information, but it also provides a better overview. Opening the file `Model_Name_diagnostics/index.html` in a web browser, results in a page with information on number of variables, parameters and equations as well as other statistics about the model.

Note that some of the entries in `Model_Name_diagnostics/index.html`, including `Problems`, `Flattened model`, `Connection sets`, `Transformed model`, `Alias sets`, `BLT diagnostics table`, `BLT for DAE System` and `BLT for Initialization System` are links to sub pages containing additional information. For example, the BLT for DAE System page contains information about in which order the model equations are evaluated and which blocks are present after compilation.

Additionally there is a table view of the `BLT`. It can be found on the `BLT diagnostics table` page. It provides a graphical representation of the `BLT`. The `BLT diagnostics table` is only generated when the model have fewer equations than the limit specified by the option `diagnostics_limit` due to the size of the graph.

In the following section a more thorough description of the HTML diagnostics will be presented.

4.6. HTML diagnostics

The compiler can generate diagnostic output in HTML format which can be viewed in, e.g, a web browser. The generation is enabled through the option `generate_html_diagnostics` and the diagnostic consists of several pages which will be presented in the sections below.

index.html

`index.html` is the index page or, i.e., the start page of the HTML diagnostics. It consists of links to the other diagnostic pages as well as statistics of the compiled model. *Model before transformation* summarizes model statistics of the flattened model. *Model after transformation* gives the statistics after the compiler has done its transformations to the model like, for example, alias elimination. Finally the number of unsolved equation blocks in DAE initialization system and system before and after tearing is applied is presented. Note that nested blocks are not visible in the equation block statistics.

errors.html

The page `errors.html`, which can be reached from `Problems` in the index page, lists all compiler errors and warnings that occurred during compilation.

flattened.html

In `flattened.html`, the flattened model, which the numbers in *Model before transformation* corresponds to, is presented. That includes a listing of all constants, parameters and variables in the model with their `type_prefix`, `type_specifier`, the possible array subscripts, and the `fully qualified name`. If the `type_specifier` is not a built-in type, the defined type will be presented at the end of the page like, e.g.,

```
type Modelica.Units.SI.MassFraction = Real(final quantity = "MassFraction",final unit =
"1",min = 0,max = 1);
```

After all the components, the initial equations are presented followed by the equations in the order they are read in by the compiler. Note that the components are given by their fully qualified name. When functions are used in the

model, the function description with its inputs, outputs and algorithm is given in the end of the page. Definitions of records used in the model can be found there as well.

Note that uses of constants, e.g., `Modelica.Constants.pi`, in

```
parameter Real x=Modelica.Constants.pi;
```

will have been evaluated (to 3.141592653589793) when declared in the flattened model. This is also true for parameters and variables which are necessary to evaluate (for example parameters used as array sizes) or determined to be equivalent to constants (for example a final independent parameter).

connections.html

All connection sets in the model are given in `connections.html`. For each connection set, the connection type, e.g., *potential* or *stream*, is written in parentheses followed by the variables in the set. The (i) and (o) indications give information on whether the variable is an inner or an outer component.

transformed.html

The transformed model, presented in `transformed.html`, has the same structure as the flattened model. The numbers in *Model after transformation* corresponds to this stage of the compilation process. In the transformed model alias variables are removed, temporary variables are introduced and some other symbolic transformations are performed. Furthermore, extra initial equations may have been introduced based on, e.g., start attributes set on variables.

alias.html

In `alias.html` alias variables are listed set by set. Each set is enclosed within curly brackets and the first variable in the list is the variable name used in the transformed model.

blt.html and initBlt.html

In `initBlt.html` and `blt.html` all the equations are sorted in the order in which they are calculated, i.e., by causality. The initialization system is found in `initBlt.html` and the BLT for the DAE system in `blt.html`. In case of an interactive FMU, these two systems coincide.

The BLT consists of solved equations, meta equations and different kinds of blocks. For equations below the *Solved equation* label, the variable on the left hand side is calculated directly through evaluation of the right hand side. *Meta equation block*s hold assert statements etc.

Blocks can be linear, non-linear as well as having discrete parts. The block type is documented in the title, for example, *Torn system (Block 1) of 1 iteration variables and 3 solved variables*. Included in the title is also the name of the block, which in turn is used in the runtime logging. Continuous iteration variables, torn variables and discrete variables are listed in separate columns. So are also the equations corresponding to each of the categories.

When reading the BLT from the interactive FMU perspective, res_i, with i=0,1,2..., corresponds to the residual equations. There is no easy way to detect which variables are the iteration variables of the steady state problem from this view. Nestled blocks will be presented as blocks are presented for segregated FMUs, before the residual equations, since these are to be solved before the residuals can be evaluated.

bltTable.html

The relationship between the equations and the variables is presented in `bltTable.html`. As for the BLT, there exist two tables: one for the initialization system and one for the DAE system. Even for this case, the tables are the same for an interactive FMU.

In the table, the equations are listed in the rows and the variables in the columns. The equation appears in the same form as in transformed.html. There are different colors and symbols in the BLT table. We have

o	The 'o' means that the variable is analytically solvable from the equation if all the other variables are known.
x	The 'x' means that the variable cannot be solved for analytically even if the other variables are known.
green	The green color marks a solved equation.
pink	A pink block shows algebraic equation blocks
dark pink	The dark pink highlights the iteration variables and residual equations of the lighter pink block.
orange	The orange color marks discrete equations and variables.
blue	The blue color marks an equation block where all equations are unsolved.

An example of how the BLT table may look like can be found in Figure 4.1. Note that a legend is also generated in the BLT with an explanation of the symbols.

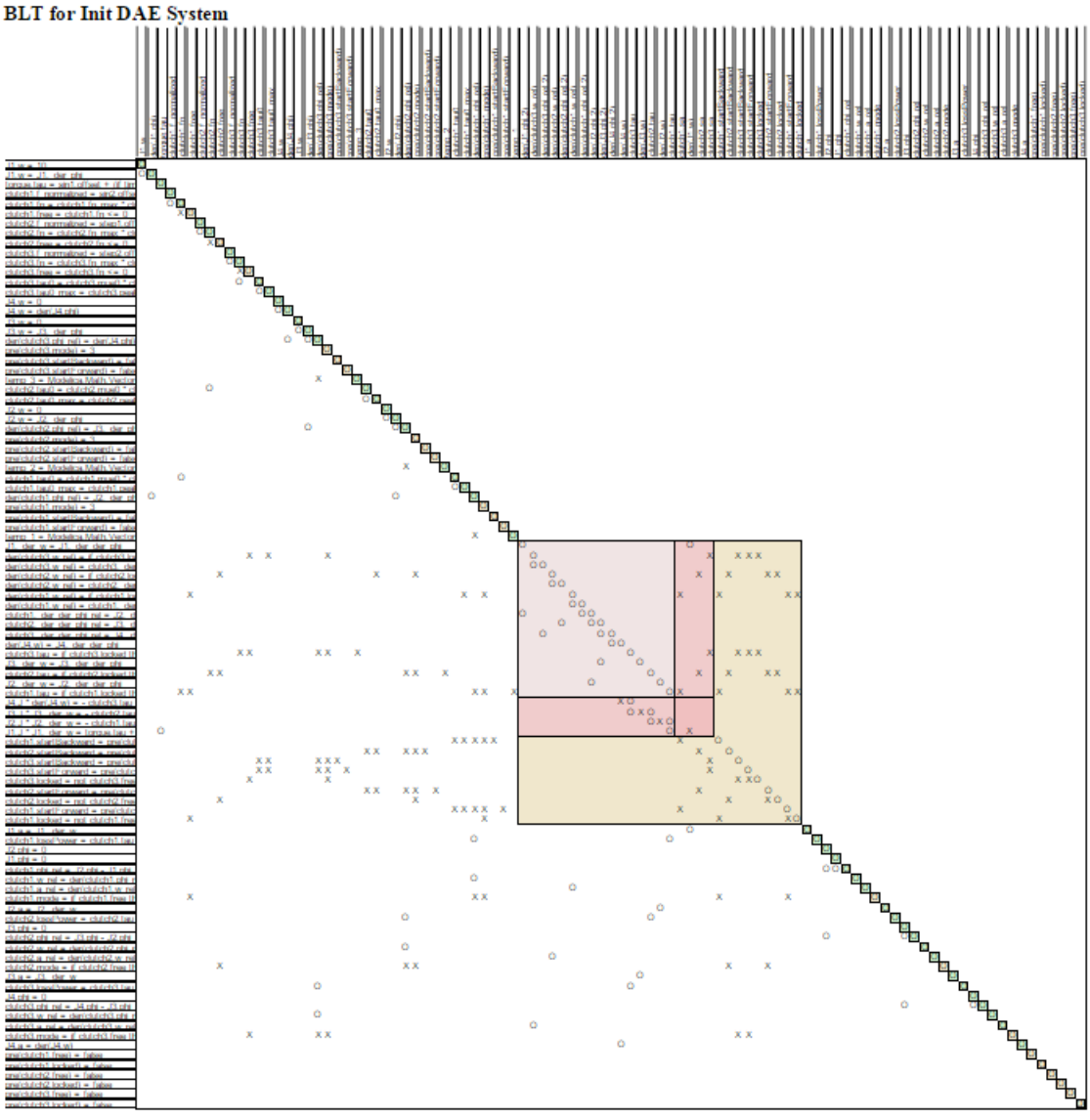


Figure 4.1 BLT table for the initial system of Modelica.Mechanics.Rotational.Examples.CoupledClutches.

blockInfo.html

`blockInfo.html` provides a list of all linear and nonlinear blocks in the initBlt and blt. Each block lists the *iteration variables* and their *start, min, max* and *nominal* values. The block names are also links to the corresponding blocks in `initBlt.html` and `blt.html`, which contain more information.

Chapter 5. Simulation of FMUs in Python

5.1. Introduction

OCT supports simulation of models described in the Modelica language and models following the FMI standard. The simulation environment uses Assimulo as standard which is a standalone Python package for solving ordinary differential and differential algebraic equations. Loading and simulation of FMUs has additionally been made available as a separate Python package, PyFMI.

This chapter describes how to load and simulate FMUs using explanatory examples.

5.2. A first example

This example focuses on how to use OCT's simulation functionality in the most basic way. The model which is to be simulated is the Van der Pol problem described in the code below. The model is also available from the examples in OCT in the file `VDP.mop` (located in `install/Python/pyjmi/examples/files`).

```
model VDP
    // State start values
    parameter Real x1_0 = 0;
    parameter Real x2_0 = 1;

    // The states
    Real x1(start = x1_0);
    Real x2(start = x2_0);

    // The control signal
    input Real u;

  equation
    der(x1) = (1 - x2^2) * x1 - x2 + u;
    der(x2) = x1;
end VDP;
```

Create a new file in your working directory called `VDP.mo` and save the model.

Next, create a Python script file and write (or copy paste) the commands for compiling and loading a model:

```
# Import the function for compilation of models and the load_fmu method
from pymodelica import compile_fmu
from pyfmi import load_fmu
```

```
# Import the plotting library
import matplotlib.pyplot as plt
```

Next, we compile and load the model:

```
# Compile model
fmu_name = compile_fmu("VDP","VDP.mo")

# Load model
vdp = load_fmu(fmu_name)
```

The function `compile_fmu` compiles the model into a binary, which is then loaded when the `vdp` object is created. This object represents the compiled model, an FMU, and is used to invoke the simulation algorithm (for more information about model compilation and options, see Chapter 4):

```
res = vdp.simulate(final_time=10)
```

In this case we use the default simulation algorithm together with default options, except for the final time which we set to 10. The result object can now be used to access the simulation result in a dictionary-like way:

```
x1 = res['x1']
x2 = res['x2']
t  = res['time']
```

The variable trajectories are returned as NumPy arrays and can be used for further analysis of the simulation result or for visualization:

```
plt.figure(1)
plt.plot(t, x1, t, x2)
plt.legend(('x1','x2'))
plt.title('Van der Pol oscillator.')
plt.ylabel('Angle (rad)')
plt.xlabel('Time (s)')
plt.show()
```

In Figure 5.1 the simulation result is shown.

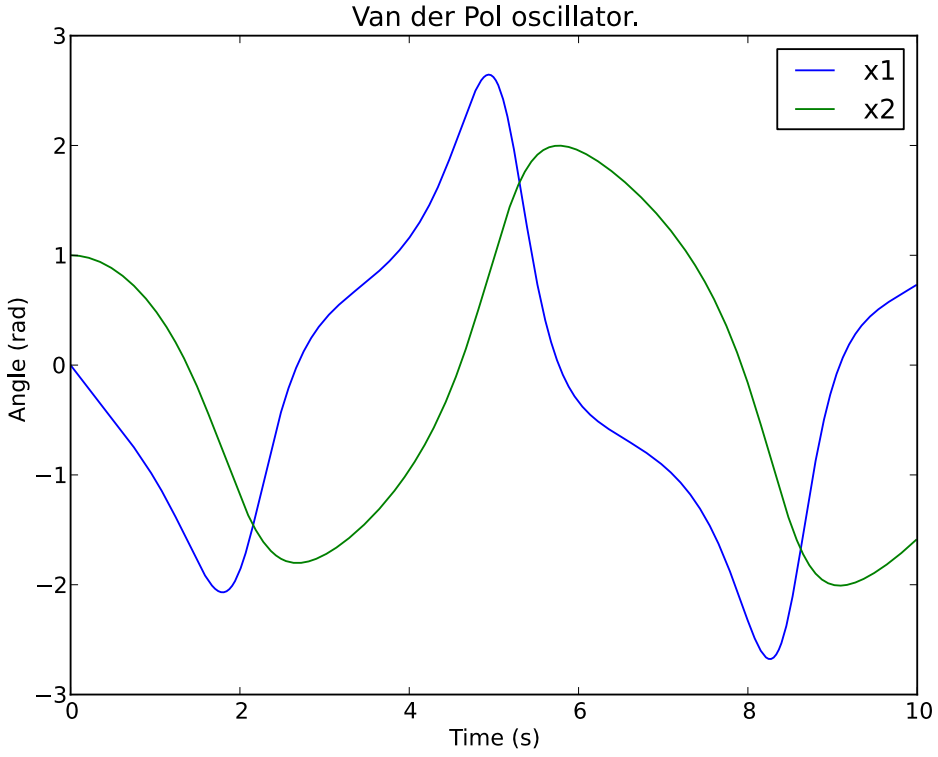


Figure 5.1 Simulation result of the Van der Pol oscillator.

5.3. Simulation of Models

Simulation of models in OCT is performed via the simulate method of a model object. The FMU model objects in OCT are located in PyFMI:

- `FMUModelME1 / FMUModelME2`
- `FMUModelCS1 / FMUModelCS2`

`FMUModelME* / FMUModelCS*` also supports compiled models from other simulation/modelling tools that follow the FMI standard (extension .fmu) (either Model exchange FMUs or Co-Simulation FMUs). Both FMI version 1.0 and FMI version 2.0 are supported. For more information about compiling a model in OCT see Chapter 4.

The simulation method is the preferred method for simulation of models and which by default is connected to the Assimulo simulation package but can also be connected to other simulation platforms. The simulation method for `FMUModelME* / FMUModelCS*` is defined as:

```
class FMUModel(ME/CS)(...)
    ...
    def simulate(self,
             start_time=0.0,
             final_time=1.0,
             input=(),
             algorithm='AssimuloFMIAlg',
             options={}):
```

And used in the following way:

```
res = FMUModel(ME/CS)*.simulate() # Using default values
```

For `FMUModelCS*`, the FMU contains the solver and is thus used (although using the same interface).

5.3.1. Convenience method, load_fmu

Since there are different FMI specifications for Model exchange and Co-Simulation and also differences between versions, a convenience method, `load_fmu` has been created. This method is the preferred access point for loading an FMU and will return an instance of the appropriate underlying `FMUModel(CS/ME)*` class.

```
model = load_fmu("myFMU.fmu")
```

5.3.2. Arguments

The start and final time attributes are simply the time where the solver should start the integration and stop the integration. The input however is a bit more complex and is described in more detail in the following section. The algorithm attribute is where the different simulation package can be specified, however currently only a connection to Assimulo is supported and connected through the algorithm `AssimuloFMIAlg` for `FMUModelME*`.

5.3.2.1. Input

The input argument defines the input trajectories to the model and should be a 2-tuple consisting of the names of the input variables and their trajectories. The names can be either a list of strings, or a single string for setting only a single input trajectory. The trajectories can be given as either a data matrix or a function. If a data matrix is used, it should contain a time vector as the first column, and then one column for each input, in the order of the list of names. If instead the second argument is a function it should be defined to take the time as input and return an array with the values of the inputs, in the order of the list of names.

For example, consider that we have a model with an input variable `u1` and that the model should be driven by a sine wave as input. We are interested in the interval 0 to 10. We will look at both using a data matrix and at using a function.

```
import numpy as N
t = N.linspace(0.,10.,100)          # Create one hundred evenly spaced points
u = N.sin(t)                        # Create the input vector
```

```
u_traj = N.transpose(N.vstack((t,u))) # Create the data matrix and transpose
                                      # it to the correct form
```

The above code have created the data matrix that we are interested in giving to the model as input, we just need to connect the data to a specific input variable, `u1`:

```
input_object = ('u1', u_traj)
```

Now we are ready to simulate using the input and simulate 10 seconds.

```
res = model.simulate(final_time=10, input=input_object)
```

If we on the other hand would have two input variables, u1 and u2 the script would instead look like:

```
import numpy as N
t = N.linspace(0.,10.,100)                    # Create one hundred evenly spaced points
u1 = N.sin(t)                                 # Create the first input vector
u2 = N.cos(t)                                 # Create the second input vector
u_traj = N.transpose(N.vstack((t,u1,u2)))     # Create the data matrix and
                                              # transpose it to the correct form
input_object = (['u1','u2'], u_traj)
res = model.simulate(final_time=10, input=input_object)
```

Note that the variables are now a `List of variables`.

If we were to do the same example using input functions instead, the code would look like for the single input case:

```
input_object = ('u1', N.sin)
```

and for the double input case:

```
def input_function(t):
    return N.array([N.sin(t),N.cos(t)])

input_object = (['u1','u2'],input_function)
```

5.3.2.2. Options for Model Exchange FMUs

The options attribute are where options to the specified algorithm are stored, and are preferably used together with:

```
opts = FMUModelME*.simulate_options()
```

which returns the default options for the default algorithm. Information about the available options can be viewed by typing help on the `opts` variable:

```
>>> help(opts)
   Options for the solving the FMU using the Assimulo simulation package.
   Currently, the only solver in the Assimulo package that fully supports
   simulation of FMUs is the solver CVode.
```

```
...
```

In Table 5.1 the general options for the AssimuloFMIAlg algorithm are described while in Table 5.2 a selection of the different solver arguments for the ODE solver CVode is shown. More information regarding the solver options can be found here, http://www.jmodelica.org/assimulo.

Table 5.1 General options for AssimuloFMIAlg.

Option	Default	Description
solver	"CVode"	Specifies the simulation method that is to be used. Currently supported solvers are, CVode, Radau5ODE, RungeKutta34, Dopri5, RodasODE, LSODAR, ExplicitEuler. The recommended solver is "CVode".
ncp	500	Number of communication points. If ncp is zero, the solver will return the internal steps taken.
initialize	True	If set to True, the initializing algorithm defined in the FMU model is invoked, otherwise it is assumed the user have manually invoked model.initialize()
write_scaled_result	False	Set this parameter to True to write the result to file without taking scaling into account. If the value of scaled is False, then the variable scaling factors of the model are used to reproduced the unscaled variable values.
result_file_name	Empty string (default generated file name will be used)	Specifies the name of the file where the simulation result is written. Setting this option to an empty string results in a default file name that is based on the name of the model class.
filter	None	A filter for choosing which variables to actually store result for. The syntax can be found here. An example is filter = "*der" , store all variables ending with 'der' and filter = ["*der*", "summary*"], store all variables with "der" in the name and all variables starting with "summary".
result_handling	"file"	Specifies how the result should be handled. Either stored to file or stored in memory. One can also use a custom handler. Available options: "file", "memory", "custom"

Lets look at an example, consider that you want to simulate an FMU model using the solver CVode together with changing the discretization method (`discr`) from BDF to Adams:

```
...
```

```
opts = model.simulate_options()          # Retrieve the default options
#opts['solver'] = 'CVode'                # Not necessary, default solver is CVode
opts['CVode_options']['discr'] = 'Adams' # Change from using BDF to Adams
opts['initialize'] = False               # Don't initialize the model
model.simulate(options=opts)             # Pass in the options to simulate and simulate
```

It should also be noted from the above example that the options regarding a specific solver, say the tolerances for `CVode`, should be stored in a double dictionary where the first is named after the solver concatenated with `_options`:

```
opts['CVode_options']['atol'] = 1.0e-6   # Options specific for CVode
```

For the general options, as changing the solver, they are accessed as a single dictionary:

```
opts['solver'] = 'CVode'  # Changing the solver
opts['ncp'] = 1000        # Changing the number of communication points.
```

Table 5.2 Selection of solver arguments for CVode

Option	Default	Description
discr	'BDF'	The discretization method. Can be either 'BDF' or 'Adams'
iter	'Newton'	The iteration method. Can be either 'Newton' or 'FixedPoint'.
maxord	5	The maximum order used. Maximum for 'BDF' is 5 while for the 'Adams' method the maximum is 12
maxh	(final_time-start_time)/ncp	Maximum step-size. Positive float.
atol	rtol*0.01*(nominal values of the continuous states)	Absolute Tolerance. Can be an array of floats where each value corresponds to the absolute tolerance for the corresponding variable. Can also be a single positive float.
rtol	1.0e-4	The relative tolerance. The relative tolerance are retrieved from the 'default experiment' section in the XML-file and if not found are set to 1.0e-4

5.3.2.3. Options for Co-Simulation FMUs

The options attribute are where options to the specified algorithm are stored, and are preferably used together with:

```
opts = FMUModelCS*.simulate_options()
```

which returns the default options for the default algorithm. Information about the available options can be viewed by typing help on the `opts` variable:

```
>>> help(opts)
   Options for the solving the CS FMU.

   ...
```

In Table 5.3 the general options for the FMICSAlg algorithm are described.

Table 5.3 General options for FMICSAlg.

Option	Default	Description
ncp	500	Number of communication points.
initialize	True	If set to True, the initializing algorithm defined in the FMU model is invoked, otherwise it is assumed the user have manually invoked model.initialize()
write_scaled_result	False	Set this parameter to True to write the result to file without taking scaling into account. If the value of scaled is False, then the variable scaling factors of the model are used to reproduced the unscaled variable values.
result_file_name	Empty string (default generated file name will be used)	Specifies the name of the file where the simulation result is written. Setting this option to an empty string results in a default file name that is based on the name of the model class.
filter	None	A filter for choosing which variables to actually store result for. The syntax can be found in http://en.wikipedia.org/wiki/Glob_%28programming%29 . An example is filter = "*der" , store all variables ending with 'der' and filter = ["*der*", "summary*"], store all variables with "der" in the name and all variables starting with "summary".
result_handling	"file"	Specifies how the result should be handled. Either stored to file or stored in memory. One can also use a custom handler. Available options: "file", "memory", "custom"

5.3.3. Return argument

The return argument from the simulate method is an object derived from a common result object `ResultBase` in `algorithm_drivers.py` with a few extra convenience methods for retrieving the result of a variable. The result object can be accessed in the same way as a dictionary type in Python with the name of the variable as key.

```
res = model.simulate()
y = res['y']           # Return the result for the variable/parameter/constant y
dery = res['der(y)']   # Return the result for the variable/parameter/constant der(y)
```

This can be done for all the variables, parameters and constants defined in the model and is the preferred way of retrieving the result. There are however some more options available in the result object, see Table 5.4.

Table 5.4 Result Object

Option	Type	Description
options	Property	Gets the options object that was used during the simulation.
solver	Property	Gets the solver that was used during the integration.
result_file	Property	Gets the name of the generated result file.
is_variable(name)	Method	Returns True if the given name is a time-varying variable.
data_matrix	Property	Gets the raw data matrix.
is_negated(name)	Method	Returns True if the given name is negated in the result matrix.
get_column(name)	Method	Returns the column number in the data matrix which corresponds to the given variable.

5.4. Examples

In the next sections, it will be shown how to use the OCT platform for simulation of various FMUs.

The Python commands in these examples may be copied and pasted directly into a Python shell, in some cases with minor modifications. Alternatively, they may be copied into a file, which also is the recommended way.

5.4.1. Simulation of a high-index model

Mechanical component-based models often result in high-index DAEs. In order to efficiently integrate such models, Modelica tools typically employs an index reduction scheme, where some equations are differentiated, and dummy derivatives are selected. In order to demonstrate this feature, we consider the model

`Modelica.Mechanics.Rotational.Examples.First` from the Modelica Standard library, see Figure 5.2. The model is of high index since there are two rotating inertias connected with a rigid gear.

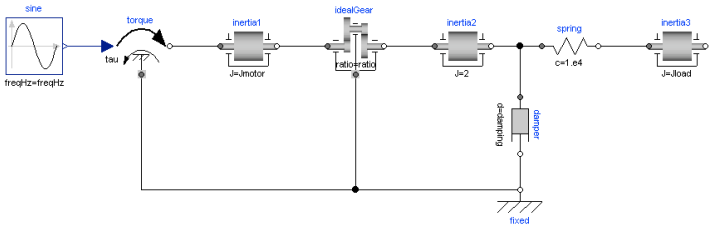


Figure 5.2 Modelica.Mechanics.Rotational.First connection diagram

First create a Python script file and enter the usual imports:

```
import matplotlib.pyplot as plt
from pymodelica import compile_fmu
from pyfmi import load_fmu
```

Next, the model is compiled and loaded:

```
# Compile model
fmu_name = compile_fmu("Modelica.Mechanics.Rotational.Examples.First")

# Load model
model = load_fmu(fmu_name)
```

Notice that no file name, just an empty tuple, is provided to the function `compile_fmu`, since in this case the model that is compiled resides in the Modelica Standard Library. In the compilation process, the index reduction algorithm is invoked. Next, the model is simulated for 3 seconds:

```
# Load result file
res = model.simulate(final_time=3.)
```

Finally, the simulation results are retrieved and plotted:

```
w1 = res['inertia1.w']
w2 = res['inertia2.w']
w3 = res['inertia3.w']
tau = res['torque.tau']
t = res['time']

plt.figure(1)
plt.subplot(2,1,1)
plt.plot(t,w1,t,w2,t,w3)
plt.grid(True)
plt.legend(['inertia1.w','inertia2.w','inertia3.w'])
plt.subplot(2,1,2)
plt.plot(t,tau)
plt.grid(True)
```

```
plt.legend(['tau'])
plt.xlabel('time [s]')
plt.show()
```

You should now see a plot as shown below.

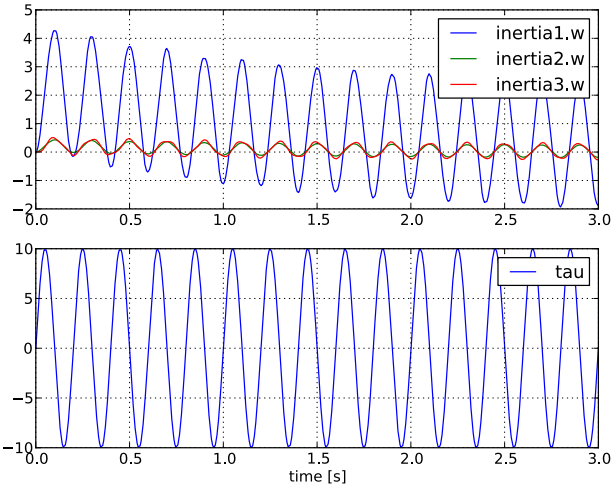


Figure 5.3 Simulation result for Modelica.Mechanics.Rotational.Examples.First

5.4.2. Simulation and parameter sweeps

This example demonstrates how to run multiple simulations with different parameter values. Sweeping parameters is a useful technique for analysing model sensitivity with respect to uncertainty in physical parameters or initial conditions. Consider the following model of the Van der Pol oscillator:

```
model VDP
  // State start values
  parameter Real x1_0 = 0;
  parameter Real x2_0 = 1;

  // The states
  Real x1(start = x1_0);
  Real x2(start = x2_0);

  // The control signal
  input Real u;

equation
  der(x1) = (1 - x2^2) * x1 - x2 + u;
  der(x2) = x1;
end VDP;
```

Notice that the initial values of the states are parametrized by the parameters `x1_0` and `x2_0`. Next, copy the Modelica code above into a file `VDP.mo` and save it in your working directory. Also, create a Python script file and name it `vdp_pp.py`. Start by copying the commands:

```
import numpy as N
import pylab as P
from pymodelica import compile_fmu
from pyfmi import load_fmu
```

into the Python file. Compile and load the model:

```
# Define model file name and class name
model_name = 'VDP'
mofile = 'VDP.mo'

# Compile model
fmu_name = compile_fmu(model_name,mofile)
```

Next, we define the initial conditions for which the parameter sweep will be done. The state `x2` starts at 0, whereas the initial condition for `x1` is swept between -3 and 3:

```
# Define initial conditions
N_points = 11
x1_0 = N.linspace(-3.,3.,N_points)
x2_0 = N.zeros(N_points)
```

In order to visualize the results of the simulations, we open a plot window:

```
fig = P.figure()
P.clf()
P.xlabel('x1')
P.ylabel('x2')
```

The actual parameter sweep is done by looping over the initial condition vectors and in each iteration set the parameter values into the model, simulate and plot:

```
for i in range(N_points):
    # Load model
    vdp = load_fmu(fmu_name)
    # Set initial conditions in model
    vdp.set('x1_0',x1_0[i])
    vdp.set('x2_0',x2_0[i])
    # Simulate
    res = vdp.simulate(final_time=20)
    # Get simulation result
    x1=res['x1']
    x2=res['x2']
    # Plot simulation result in phase plane plot
    P.plot(x1, x2,'b')
P.grid()
P.show()
```

You should now see a plot similar to that in Figure 5.4.

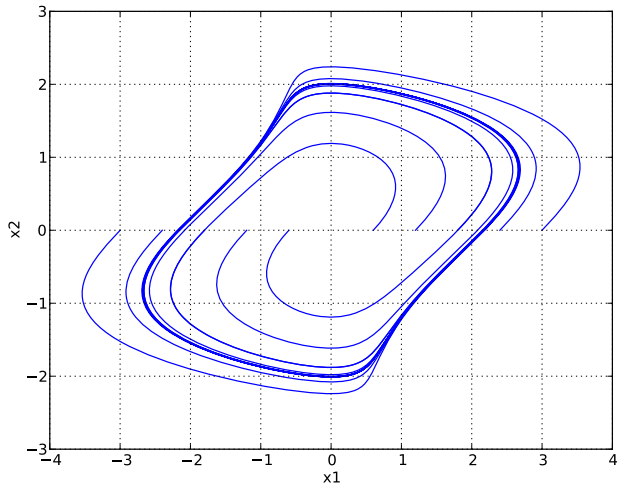


Figure 5.4 Simulation result-phase plane

5.4.3. Simulation of an Engine model with inputs

In this example the model is larger than the previous. It is a slightly modified version of the model EngineV6_analytic from the Multibody library in the Modelica Standard Library. The modification consists of a replaced load with a user defined load. This has been done in order to be able to demonstrate how inputs are set from a Python script. In Figure 5.5 the model is shown.

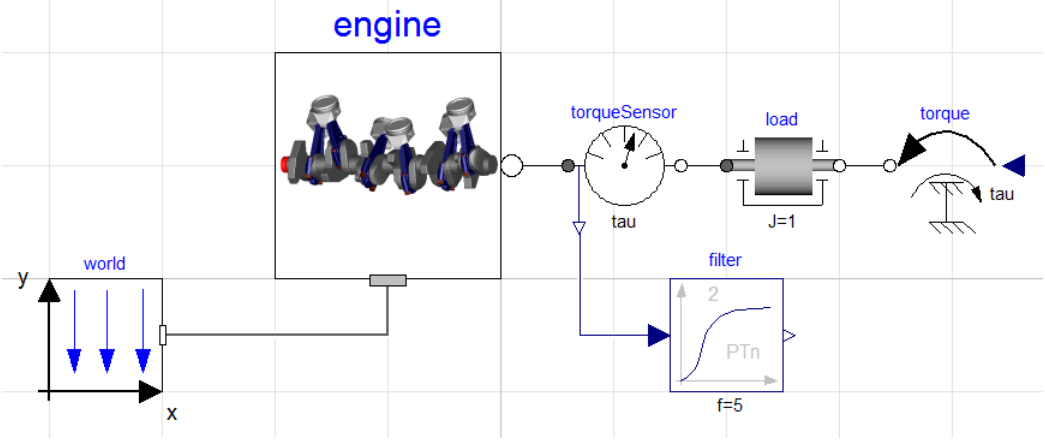


Figure 5.5 Overview of the Engine model

The Modelica code for the model is shown below, copy and save the code in a file named `EngineV6.mo`.

```
model EngineV6_analytic_with_input
  output Real engineSpeed_rpm= Modelica.Units.SI.Conversions.to_rpm(load.w);
  output Real engineTorque = filter.u;
  output Real filteredEngineTorque = filter.y;

  input Real u;

  import Modelica.Mechanics.*;

  inner MultiBody.World world;
  MultiBody.Examples.Loops.Utilities.EngineV6_analytic engine(redeclare
      model Cylinder = MultiBody.Examples.Loops.Utilities.Cylinder_analytic_CAD);

  Rotational.Components.Inertia load(
    phi(start=0,fixed=true), w(start=10,fixed=true),
    stateSelect=StateSelect.always,J=1);
  Rotational.Sensors.TorqueSensor torqueSensor;
  Rotational.Sources.Torque torque;

  Modelica.Blocks.Continuous.CriticalDamping filter(
    n=2,initType=Modelica.Blocks.Types.Init.SteadyState,f=5);

equation
  torque.tau = u;

  connect(world.frame_b, engine.frame_a);
  connect(torque.flange, load.flange_b);
  connect(torqueSensor.flange_a, engine.flange_b);
  connect(torqueSensor.flange_b, load.flange_a);
  connect(torqueSensor.tau, filter.u);
  annotation (experiment(StopTime=1.01));

end EngineV6_analytic_with_input;
```

Now that the model has been defined, we create our Python script which will compile, simulate and visualize the result for us. Create a new text-file and start by copying the below commands into the file. The code will import the necessary methods and packages into Python.

```
from pymodelica import compile_fmu
from pyfmi import load_fmu
import pylab as P
```

Compiling the model is performed by invoking the `compile_fmu` method where the first argument is the name of the model and the second argument is where the model is located (which file). The method will create an FMU in the current directory and in order to simulate the FMU, we need to additionally load the created FMU into Python. This is done with the `load_fmu` method which takes the name of the FMU as input.

```
name = compile_fmu("EngineV6_analytic_with_input", "EngineV6.mo")
```

```
model = load_fmu(name)
```

So, now that we have compiled the model and loaded it into Python we are almost ready to simulate the model. First however, we retrieve the simulation options and specify how many result points we want to receive after a simulation.

```
opts = model.simulate_options()
opts["ncp"] = 1000 #Specify that 1000 output points should be returned
```

A simulation is finally performed using the `simulate` method on the model and as we have changed the options, we need to additionally provide these options to the simulate method.

```
res = model.simulate(options=opts)
```

The simulation result is returned and stored into the `res` object. Result for a trajectory is easily retrieved using a Python dictionary syntax. Below is the visualization code for viewing the engine torque. One could instead use the Plot GUI for the visualization as the result are stored in a file in the current directory.

```
P.plot(res["time"],res["filteredEngineTorque"], label="Filtered Engine Torque")
P.show()
```

In Figure 5.6 the trajectories are shown for he engine torque and the engine speed utilizing subplots from Matplotlib.

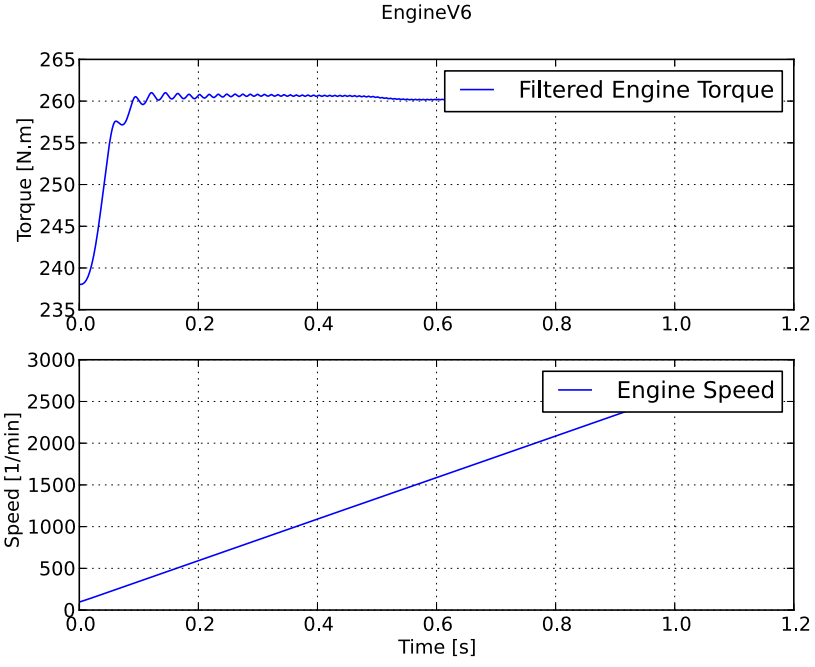


Figure 5.6 Resulting trajectories for the engine model.

Above we have simulated the engine model and looked at the result, we have not however specified any load as input. Remember that the model we are looking at has a user specified load. Now we will create a Python function that will act as our input. We create a function that depends on the time and returns the value for use as input.

```
def input_func(t):
    return -100.0*t
```

In order to use this input in the simulation, simply provide the name of the input variable and the function as the input argument to the simulate method, see below.

```
res = model.simulate(options=opts, input=("u",input_func))
```

Simulate the model again and look at the result and the impact of the input.

Large models contain an enormous amount of variables and by default, all of these variables are stored in the result. Storing the result takes time and for large models the saving of the result may be responsible for the majority of the overall simulation time. Not all variables may be of interest, for example in our case, we are only interested in two variables so storing the other variables are not necessary. In the options dictionary there is a filter option which allows to specify which variables should be stored, so in our case, try the below filter and look at the impact on the simulation time.

```
opts["filter"] = ["filteredEngineTorque", "engineSpeed_rpm"]
```

5.4.4. Simulation using the native FMI interface

This example shows how to use the native OCT FMI interface for simulation of an FMU of version 2.0 for Model Exchange. For the procedure with version 1.0, refer to Functional Mock-up Interface for Model Exchange version 1.0.

The FMU that is to be simulated is the bouncing ball example from Qtronics FMU SDK (http://www.qtronic.de/en/fmusdk.html). This example is written similar to the example in the documentation of the 'Functional Mock-up Interface for Model Exchange' version 2.0 (https://www.fmi-standard.org/). The bouncing ball model is to be simulated using the explicit Euler method with event detection.

The example can also be found in the Python examples catalog in the OCT platform. There you can also find a similar example for simulation with a version 1.0 Model Exchange FMU.

The bouncing ball consists of two equations,

$$\dot{h} = v$$
$$\dot{v} = -g$$

and one event function (also commonly called root function),

$$h > 0$$

Where the ball bounces and lose some of its energy according to,

$v_a = -ev_b$

Here, h is the height, g the gravity, v the velocity and e a dimensionless parameter. The starting values are, h=1 and v=0 and for the parameters, e=0.7 and g = 9.81.

5.4.4.1. Implementation

Start by importing the necessary modules,

```
import numpy as N
import pylab as P                  # Used for plotting
from pyfmi.fmi import load_fmu  # Used for loading the FMU
```

Next, the FMU is to be loaded and initialized

```
# Load the FMU by specifying the fmu together with the path.
bouncing_fmu = load_fmu('/path/to/FMU/bouncingBall.fmu')

Tstart = 0.5                # The start time.
Tend   = 3.0                # The final simulation time.
# Initialize the model. Also sets all the start attributes defined in the
# XML file.
bouncing_fmu.setup_experiment(start_time = Tstart) # Set the start time to Tstart
bouncing_fmu.enter_initialization_mode()
bouncing_fmu.exit_initialization_mode()
```

The first line loads the FMU and connects the C-functions of the model to Python together with loading the information from the XML-file. The start time also needs to be specified by providing the argument `start_time` to setup_experiment. The model is also initialized, which must be done before the simulation is started.

Note that if the start time is not specified, `FMUModelME2` tries to find the starting time in the XML-file structure 'default experiment' and if successful starts the simulation from that time. Also if the XML-file does not contain any information about the default experiment the simulation is started from time zero.

Next step is to do the event iteration and thereafter enter continuous time mode.

```
eInfo = bouncing_fmu.get_event_info()
eInfo.newDiscreteStatesNeeded = True
#Event iteration
while eInfo.newDiscreteStatesNeeded == True:
    bouncing_fmu.enter_event_mode()
    bouncing_fmu.event_update()
    eInfo = bouncing_fmu.get_event_info()

bouncing_fmu.enter_continuous_time_mode()
```

Then information about the first step is retrieved and stored for later use.

```
# Get Continuous States
x = bouncing_fmu.continuous_states
# Get the Nominal Values
x_nominal = bouncing_fmu.nominal_continuous_states
# Get the Event Indicators
event_ind = bouncing_fmu.get_event_indicators()

# Values for the solution
# Retrieve the valuereferences for the values 'h' and 'v'
vref  = [bouncing_fmu.get_variable_valueref('h')] + \
        [bouncing_fmu.get_variable_valueref('v')]
t_sol = [Tstart]
sol = [bouncing_fmu.get_real(vref)]
```

Here the continuous states together with the nominal values and the event indicators are stored to be used in the integration loop. In our case the nominal values are all equal to one. This information is available in the XML-file. We also create lists which are used for storing the result. The final step before the integration is started is to define the step-size.

```
time = Tstart
Tnext = Tend   # Used for time events
dt = 0.01      # Step-size
```

We are now ready to create our main integration loop where the solution is advanced using the explicit Euler method.

```
# Main integration loop.
while time < Tend and not bouncing_fmu.get_event_info().terminateSimulation:
    #Compute the derivative of the previous step f(x(n), t(n))
    dx = bouncing_fmu.get_derivatives()

    # Advance
    h = min(dt, Tnext-time)
    time = time + h

    # Set the time
    bouncing_fmu.time = time

    # Set the inputs at the current time (if any)
    # bouncing_fmu.set_real,set_integer,set_boolean,set_string (valueref, values)

    # Set the states at t = time (Perform the step using x(n+1)=x(n)+hf(x(n), t(n))
    x = x + h*dx
    bouncing_fmu.continuous_states = x
```

This is the integration loop for advancing the solution one step. The loop continues until the final time has been reached or if the FMU reported that the simulation is to be terminated. At the start of the loop the derivatives of the continuous states are retrieved and then the simulation time is incremented by the step-size and set to the model. It could also be the case that the model depends on inputs which can be set using the `set_(real/...)` methods.

Note that only variables defined in the XML-file to be inputs can be set using the `set_(real/...)` methods according to the FMI specification.

The step is performed by calculating the new states (x+h*dx) and setting the values into the model. As our model, the bouncing ball also consist of event functions which needs to be monitored during the simulation, we have to check the indicators which is done below.

```
    # Get the event indicators at t = time
    event_ind_new = bouncing_fmu.get_event_indicators()

    # Inform the model about an accepted step and check for step events
    step_event = bouncing_fmu.completed_integrator_step()

    # Check for time and state events
    time_event  = abs(time-Tnext) <= 1.e-10
    state_event = True if True in ((event_ind_new>0.0) != (event_ind>0.0)) else False
```

Events can be, time, state or step events. The time events are checked by continuously monitoring the current time and the next time event (`Tnext`). State events are checked against sign changes of the event functions. Step events are monitored in the FMU, in the method `completed_integrator_step()` and return `True` if any event handling is necessary. If an event has occurred, it needs to be handled, see below.

```
    # Event handling
    if step_event or time_event or state_event:
        bouncing_fmu.enter_event_mode()
        eInfo = bouncing_fmu.get_event_info()
        eInfo.newDiscreteStatesNeeded = True

        # Event iteration
        while eInfo.newDiscreteStatesNeeded:
            bouncing_fmu.event_update('0')       # Stops at each event iteration
            eInfo = bouncing_fmu.get_event_info()

            # Retrieve solutions (if needed)
            if eInfo.newDiscreteStatesNeeded:
                # bouncing_fmu.get_real,get_integer,get_boolean,get_string(valueref)
                pass

        # Check if the event affected the state values and if so sets them
        if eInfo.valuesOfContinuousStatesChanged:
            x = bouncing_fmu.continuous_states

        # Get new nominal values.
        if eInfo.nominalsOfContinuousStatesChanged:
            atol = 0.01*rtol*bouncing_fmu.nominal_continuous_states

        # Check for new time event
        if eInfo.nextEventTimeDefined:
            Tnext = min(eInfo.nextEventTime, Tend)
        else:
```

```
            Tnext = Tend
        bouncing_fmu.enter_continuous_time_mode()
```

If an event occurred, we enter the iteration loop and the event mode where we loop until the solution of the new states have converged. During this iteration we can also retrieve the intermediate values with the normal `get` methods. At this point `eInfo` contains information about the changes made in the iteration. If the state values have changed, they are retrieved. If the state references have changed, meaning that the state variables no longer have the same meaning as before by pointing to another set of continuous variables in the model, for example in the case with dynamic state selection, new absolute tolerances are calculated with the new nominal values. Finally the model is checked for a new time event and the continuous time mode is entered again.

```
    event_ind = event_ind_new

    # Retrieve solutions at t=time for outputs
    # bouncing_fmu.get_real,get_integer,get_boolean,get_string (valueref)

    t_sol += [time]
    sol += [bouncing_fmu.get_real(vref)]
```

In the end of the loop, the solution is stored and the old event indicators are stored for use in the next loop.

After the loop has finished, by reaching the final time, we plot the simulation results

```
# Plot the height
P.figure(1)
P.plot(t_sol,N.array(sol)[:,0])
P.title(bouncing_fmu.get_name())
P.ylabel('Height (m)')
P.xlabel('Time (s)')
# Plot the velocity
P.figure(2)
P.plot(t_sol,N.array(sol)[:,1])
P.title(bouncing_fmu.get_name())
P.ylabel('Velocity (m/s)')
P.xlabel('Time (s)')
P.show()
```

and the figure below shows the results.

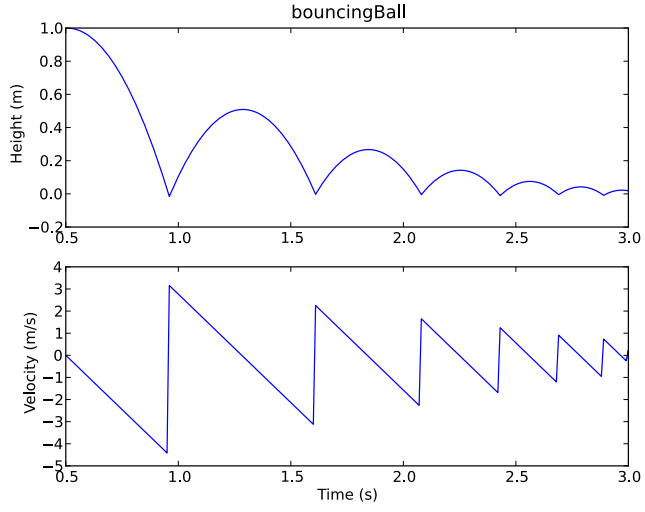


Figure 5.7 Simulation result

5.4.5. Simulation of Co-Simulation FMUs

Simulation of a Co-Simulation FMU follows the same workflow as simulation of a Model Exchange FMU. The model we would like to simulate is a model of a bouncing ball, the file `bouncingBall.fmu` is located in the examples folder in the OCT installation, `pyfmi/examples/files/CS1.0/` for version 1.0 and `pyfmi/examples/files/CS2.0/` for version 2.0. The FMU is a Co-simulation FMU and in order to simulate it, we start by importing the necessary methods and packages into Python:

```
import pylab as P           # For plotting
from pyfmi import load_fmu  # For loading the FMU
```

Here, we have imported packages for plotting and the method `load_fmu` which takes as input an FMU and then determines the type and returns the appropriate class. Now, we need to load the FMU.

```
model = load_fmu('bouncingBall.fmu')
```

The `model` object can now be used to interact with the FMU, setting and getting values for instance. A simulation is performed by invoking the `simulate` method:

```
res = model.simulate(final_time=2.)
```

As a Co-Simulation FMU contains its own integrator, the method simulate calls this integrator. Finally, plotting the result is done as before:

```
# Retrieve the result for the variables
h_res = res['h']
```

```
v_res = res['v']
t     = res['time']
# Plot the solution
# Plot the height
fig = P.figure()
P.clf()
P.subplot(2,1,1)
P.plot(t, h_res)
P.ylabel('Height (m)')
P.xlabel('Time (s)')
# Plot the velocity
P.subplot(2,1,2)
P.plot(t, v_res)
P.ylabel('Velocity (m/s)')
P.xlabel('Time (s)')
P.suptitle('FMI Bouncing Ball')
P.show()
```

and the figure below shows the results.

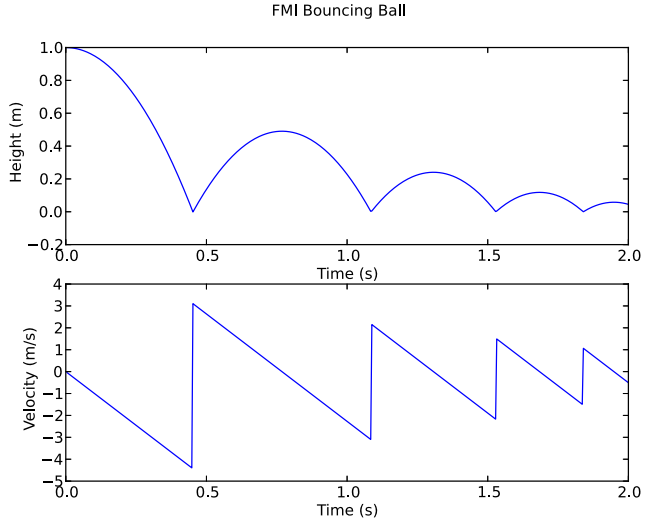


Figure 5.8 Simulation result

Chapter 6. Dynamic Optimization in Python

6.1. Introduction

OCT supports optimization of dynamic and steady state models. Many engineering problems can be cast as optimization problems, including optimal control, minimum time problems, optimal design, and model calibration. These different types of problems will be illustrated and it will be shown how they can be formulated and solved. The chapter starts with an introductory example in Section 6.2 and in Section 6.3, the details of how the optimization algorithms are invoked are explained. The following sections contain tutorial exercises that illustrates how to set up and solve different kinds of optimization problems.

When formulating optimization problems, models are expressed in the Modelica language, whereas optimization specifications are given in the Optimica extension which is described in Chapter 16. The tutorial exercises in this chapter assumes that the reader is familiar with the basics of Modelica and Optimica.

6.2. A first example

In this section, a simple optimal control problem will be solved. Consider the optimal control problem for the Van der Pol oscillator model:

```
optimization VDP_Opt (objectiveIntegrand = x1^2 + x2^2 + u^2,
                      startTime = 0,
                      finalTime = 20)

  // The states
  Real x1(start=0,fixed=true);
  Real x2(start=1,fixed=true);

  // The control signal
  input Real u;

equation
  der(x1) = (1 - x2^2) * x1 - x2 + u;
  der(x2) = x1;
constraint
   u<=0.75;
end VDP_Opt;
```

Create a new file named `VDP_Opt.mop` and save it in you working directory. Notice that this model contains both the dynamic system to be optimized and the optimization specification. This is possible since Optimica is an extension of Modelica and thereby supports also Modelica constructs such as variable declarations and equations. In most cases, however, Modelica models are stored separately from the Optimica specifications.

Next, create a Python script file and a write (or copy paste) the following commands:

```
# Import the function for transfering a model to CasADiInterface
from pyjmi import transfer_optimization_problem

# Import the plotting library
import matplotlib.pyplot as plt
```

Next, we transfer the model:

```
# Transfer the optimization problem to casadi
 op = transfer_optimization_problem("VDP_Opt", "VDP_Opt.mop")
```

The function `transfer_optimization_problem` transfers the optimization problem into Python and expresses it's variables, equations, etc., using the automatic differentiation tool CasADi. This object represents the compiled model and is used to invoke the optimization algorithm:

```
res = op.optimize()
```

In this case, we use the default settings for the optimization algorithm. The result object can now be used to access the optimization result:

```
# Extract variable profiles
x1=res['x1']
x2=res['x2']
u=res['u']
t=res['time']
```

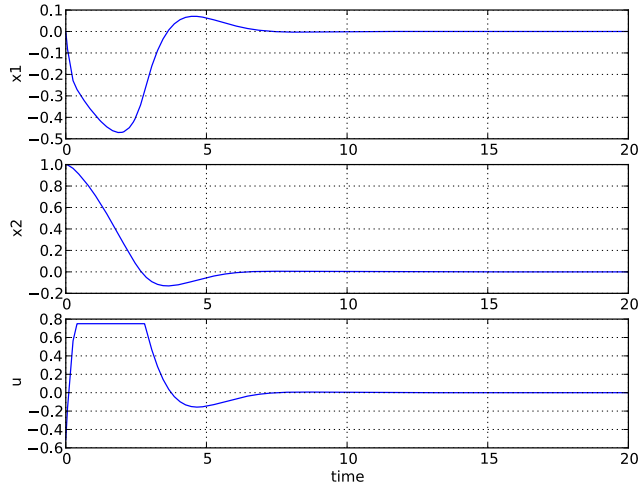
The variable trajectories are returned as NumPy arrays and can be used for further analysis of the optimization result or for visualization:

```
plt.figure(1)
plt.clf()
plt.subplot(311)
plt.plot(t,x1)
plt.grid()
plt.ylabel('x1')

plt.subplot(312)
plt.plot(t,x2)
plt.grid()
plt.ylabel('x2')

plt.subplot(313)
plt.plot(t,u)
plt.grid()
plt.ylabel('u')
plt.xlabel('time')
plt.show()
```

You should now see the optimization result as shown in Figure 6.1.



Optimal control and state profiles for the Van Der Pol optimal control problem.
Figure 6.1 Optimal profiles for the VDP oscillator

6.3. Solving optimization problems

The first step when solving an optimization problem is to formulate a model and an optimization specification and then compile the model as described in the following sections in this chapter. There are currently two different optimization algorithms available in OCT, which are suitable for different classes of optimization problems.

- **Dynamic optimization of DAEs using direct collocation with CasADi.** This algorithm is the default algorithm for solving optimal control and parameter estimation problems. It is implemented in Python, uses CasADi for computing function derivatives and the nonlinear programming solvers IPOPT or WORHP for solving the resulting NLP. Use this method if your model is a DAE and does not contain discontinuities.

- **Derivative free calibration and optimization of ODEs with FMUs.** This algorithm solves parameter optimization and model calibration problems and is based on FMUs. The algorithm is implemented in Python and relies on a Nelder-Mead derivative free optimization algorithm. Use this method if your model is of large scale and has a modest number of parameters to calibrate and/or contains discontinuities or hybrid elements. Note that this algorithm is applicable to models which have been exported as FMUs also by other tools than OCT.

To illustrate how to solve optimization problems the Van der Pol problem presented above is used. First, the model is transferred into Python

```
op = transfer_optimization_problem("VDP_pack.VDP_Opt2", "VDP_Opt.mop")
```

All operations that can be performed on the model are available as methods of the `op` object and can be accessed by tab completion. Invoking an optimization algorithm is done by calling the method `OptimizationProblem.optimize`, which performs the following tasks:

- Sets up the selected algorithm with default or user defined options
- Invokes the algorithm to find a numerical solution to the problem
- Writes the result to a file
- Returns a result object from which the solution can be retrieved

The interactive help for the `optimize` method is displayed by the command:

```
>>> help(op.optimize)
    Solve an optimization problem.

    Parameters::

        algorithm --
            The algorithm which will be used for the optimization is
            specified by passing the algorithm class name as string or
            class object in this argument. 'algorithm' can be any
            class which implements the abstract class AlgorithmBase
            (found in algorithm_drivers.py). In this way it is
            possible to write custom algorithms and to use them with this
            function.

            The following algorithms are available:
            - 'LocalDAECollocationAlg'. This algorithm is based on
              direct collocation on finite elements and the algorithm IPOPT
              is used to obtain a numerical solution to the problem.
            Default: 'LocalDAECollocationAlg'

        options --
            The options that should be used in the algorithm. The options
            documentation can be retrieved from an options object:

                >>> myModel = OptimizationProblem(...)
                >>> opts = myModel.optimize_options()
                >>> opts?

            Valid values are:
            - A dict that overrides some or all of the algorithm's default values.
              An empty  dict will thus give all options with default values.
            - An Options object for the corresponding algorithm, e.g.
              LocalDAECollocationAlgOptions for LocalDAECollocationAlg.
            Default: Empty dict

    Returns::
```

```
        A result object, subclass of algorithm_drivers.ResultBase.
```

The optimize method can be invoked without any arguments, in which case the default optimization algorithm, with default options, is invoked:

```
res = vdp.optimize()
```

In the remainder of this chapter the available algorithms are described in detail. Options for an algorithm can be set using the `options` argument to the `optimize` method. It is convenient to first obtain an options object in order to access the documentation and default option values. This is done by invoking the method `optimize_options`:

```
>>> help(op.optimize_options)
    Returns an instance of the optimize options class containing options
    default values. If called without argument then the options class for
    the default optimization algorithm will be returned.

    Parameters::

        algorithm --
            The algorithm for which the options class should be returned.
            Possible values are: 'LocalDAECollocationAlg'.
            Default: 'LocalDAECollocationAlg'

    Returns::

        Options class for the algorithm specified with default values.
```

The option object is essentially a Python dictionary and options are set simply by using standard dictionary syntax:

```
opts = vdp.optimize_options()
opts['n_e'] = 5
```

The optimization algorithm may then be invoked again with the new options:

```
res = vdp.optimize(options=opts)
```

Available options for each algorithm are documented in their respective sections in this Chapter.

The `optimize` method returns a result object containing the optimization result and some meta information about the solution. The most common operation is to retrieve variable trajectories from the result object:

```
time = res['time']
x1 = res['x1']
```

Variable data is returned as NumPy arrays. The result object also contains references to the model that was optimized, the name of the result file that was written to disk, a solver object representing the optimization algorithm and an options object that was used when solving the optimization problem.

6.4. Scaling

Many physical models contain variables with values that differ by several orders of magnitude. A typical example is thermodynamic models containing pressures, temperatures and mass flows. Such large differences in scales may have a severe deteriorating effect on the performance of numerical algorithms, and may in some cases even lead to the algorithm failing. In order to relieve the user from the burden of manually scaling variables, Modelica offers the `nominal` attribute, which can be used to automatically scale a model. Consider the Modelica variable declaration:

```
Real pressure(start=101.3e3, nominal=1e5);
```

Here, the `nominal` attribute is used to specify that the variable pressure takes on values which are on the order of 1e5. In order to use `nominal` attributes for scaling with CasADi-based algorithms, scaling is enabled by setting the algorithm option `variable_scaling` to `True`, and is enabled by default . When scaling is enabled, all variables with a set nominal attribute are then scaled by dividing the variable value with its nominal value, i.e., from an algorithm point of view, all variables should take on values close to one. Notice that variables typically vary during a simulation or optimization and that it is therefore not possible to obtain perfect scaling. In order to ensure that model equations are fulfilled, each occurrence of a variable is multiplied with its nominal value in equations. For example, the equation:

```
T = f(p)
```

is replaced by the equation

```
T_scaled*T_nom = f(p_scaled*p_nom)
```

when `variable scaling` is enabled.

The algorithm in Section 6.5 also has support for providing trajectories (obtained by for example simulation) that are used for scaling. This means that it usually is not necessary to provide nominal values for all variables, and that it is possible to use time-varying scaling factors.

For debugging purposes, it is sometimes useful to write a simulation/optimization/initialization result to file in scaled format, in order to detect if there are some variables which require additional scaling. The option `write_scaled_result` has been introduced as an option to the `initialize`, `simulate` and `optimize` methods for this purpose.

6.5. Dynamic optimization of DAEs using direct collocation with CasADi

6.5.1. Algorithm overview

The direct collocation method described in this section can be used to solve dynamic optimization problems, including optimal control problems and parameter optimization problems. In the collocation method, the dynamic model variable profiles are approximated by piecewise polynomials. This method of approximating a differential equation corresponds to a fixed step implicit Runge-Kutta scheme, where the mesh defines the length of each step.

Also, the number of collocation points in each element, or step, needs to be provided. This number corresponds to the stage order of the Runge-Kutta scheme. The selection of mesh is analogous to the choice of step length in a one-step algorithm for solving differential equations. Accordingly, the mesh needs to be fine-grained enough to ensure sufficiently accurate approximation of the differential constraint. The nonlinear programming (NLP) solvers IPOPT and WORHP can be used to solve the nonlinear program resulting from collocation. The needed first- and second-order derivatives are obtained using CasADi by algorithmic differentiation. For more details on the inner workings of the algorithm, see [Mag2015] and Chapter 3 in [Mag2016].

The NLP solvers require that the model equations are twice continuously differentiable with respect to all of the variables. This for example means that the model can not contain integer variables or `if` clauses depending on the states.

Optimization models are represented using the class `OptimizationProblem`, which can be instantiated using the `transfer_optimization_problem` method. An object containing all the options for the optimization algorithm can be retrieved from the object:

```
from pyjmi import transfer_optimization_problem
op = transfer_optimization_problem(class_name, optimica_file_path)
opts = op.optimize_options()
opts? # View the help text
```

After options have been set, the options object can be propagated to the `optimize` method, which solves the optimization problem:

```
res = op.optimize(options=opts)
```

The standard options for the algorithm are shown in Table 6.1. Additional documentation is available in the Python class documentation. The algorithm also has a lot of experimental options, which are not as well tested and some are intended for debugging purposes. These are shown in Table 6.2, and caution is advised when changing their default values.

Table 6.1 Standard options for the CasADi- and collocation-based optimization algorithm

Option	Default	Description
`n_e`	50	Number of finite elements.
`hs`	None	Element lengths. Possible values: None, iterable of floats and "free" None: The element lengths are uniformly distributed. iterable of floats: Component i of the iterable specifies the length of element i. The lengths must be normalized in the sense that the sum of all lengths must be equal to 1. "free": The element lengths become optimization variables and are optimized according to the algorithm option free_element_lengths_data. WARNING: The "free" option is very experimental and will not always give desirable results.

Option	Default	Description
n_cp	3	Number of collocation points in each element.
expand_to_sx	"NLP"	Whether to expand the CasADi MX graphs to SX graphs. Possible values: "NLP", "DAE", "no". "NLP": The entire NLP graph is expanded into SX. This will lead to high evaluation speed and high memory consumption. "DAE": The DAE, objective and constraint graphs for the dynamic optimization problem expressions are expanded into SX, but the full NLP graph is an MX graph. This will lead to moderate evaluation speed and moderate memory consumption. "no": All constructed graphs are MX graphs. This will lead to low evaluation speed and low memory consumption.
init_traj	None	Variable trajectory data used for initialization of the NLP variables.
nominal_traj	None	Variable trajectory data used for scaling of the NLP variables. This option is only applicable if variable scaling is enabled.
blocking_factors	None (not used)	Blocking factors are used to enforce piecewise constant inputs. The inputs may only change values at some of the element boundaries. The option is either None (disabled), given as an instance of pyjmi.optimization.casadi_collocation.BlockingFactors or as a list of blocking factors. If the options is a list of blocking factors, then each element in the list specifies the number of collocation elements for which all of the inputs must be constant. For example, if blocking_factors == [2, 2, 1], then the inputs will attain 3 different values (number of elements in the list), and it will change values between collocation element number 2 and 3 as well as number 4 and 5. The sum of all elements in the list must be the same as the number of collocation elements and the length of the list determines the number of separate values that the inputs may attain. See the documentation of the BlockingFactors class for how to use it. If blocking_factors is None, then the usual collocation polynomials are instead used to represent the controls.
external_data	None	Data used to penalize, constrain or eliminate certain variables.

Option	Default	Description
`delayed_feedback`	None	If not `None`, should be a `dict` with mappings `'delayed_var': ('undelayed_var', delay_ne)`. For each key-value pair, adds the the constraint that the variable `'delayed_var'` equals the value of the variable `'undelayed_var'` delayed by `delay_ne` elements. The initial part of the trajectory for `'delayed_var'` is fixed to its initial guess given by the `init_traj` option or the `initialGuess` attribute. `'delayed_var'` will typically be an input. This is an experimental feature and is subject to change.
`solver`	'IPOPT'	Specifies the nonlinear programming solver to be used. Possible choices are 'IPOPT' and 'WORHP'.
`verbosity`	3	Sets verbosity of algorithm output. 0 prints nothing, 3 prints everything.
`IPOPT_options`	IPOPT defaults	IPOPT options for solution of NLP. See IPOPT's documentation for available options.
`WORHP_options`	WORHP defaults	WORHP options for solution of NLP. See WORHP's documentation for available options.

Table 6.2 Experimental and debugging options for the CasADi- and collocation-based optimization algorithm

Option	Default	Description
`free_element_lengths_data`	None	Data used for optimizing the element lengths if they are free. Should be None when hs != "free".
`discr`	'LGR'	Determines the collocation scheme used to discretize the problem. Possible values: "LG" and "LGR". "LG": Gauss collocation (Legendre-Gauss) "LGR": Radau collocation (Legendre-Gauss-Radau).
`named_vars`	False	If enabled, the solver will create a duplicated set of NLP variables which have names corresponding to the Modelica/Optimica variable names. Symbolic expressions of the NLP consisting of the named variables can then be obtained using the get_named_var_expr method of the collocator class. This option is only intended for investigative purposes.
`init_dual`	None	Dictionary containing vectors of initial guess for NLP dual variables. Intended to be obtained as the solution of an optimization problem which has an identical structure, which is stored in the dual_opt attribute of

Option	Default	Description
		the result object. The dictionary has two keys, 'g' and 'x', containing vectors of the corresponding dual variable intial guesses. Note that when using IPOPT, the option warm_start_init_point has to be activated for this option to have an effect.
variable_scaling	True	Whether to scale the variables according to their nominal values or the trajectories provided with the nominal_traj option.
equation_scaling	False	Whether to scale the equations in collocated NLP. Many NLP solvers default to scaling the equations, but if it is done through this option the resulting scaling can be inspected.
nominal_traj_mode	{"_default_mode": "linear"}	Mode for computing scaling factors based on nominal trajectories. Four possible modes: "attribute": Time-invariant, linear scaling based on Nominal attribute "linear": Time-invariant, linear scaling "affine": Time-invariant, affine scaling "time-variant": Time-variant, linear scaling Option is a dictionary with variable names as keys and corresponding scaling modes as values. For all variables not occuring in the keys of the dictionary, the mode specified by the "_default_mode" entry will be used, which by default is "linear".
result_file_name	""	Specifies the name of the file where the result is written. Setting this option to an empty string results in a default file name that is based on the name of the model class.
write_scaled_result	False	Return the scaled optimization result if set to True, otherwise return the unscaled optimization result. This option is only applicable when variable_scaling is enabled and is only intended for debugging.
print_condition_numbers	False	Prints the condition numbers of the Jacobian of the constraints and of the simplified KKT matrix at the initial and optimal points. Note that this is only feasible for very small problems.
result_mode	'collocation_points'	Specifies the output format of the optimization result. Possible values: "collocation_points", "element_interpolation" and "mesh_points" "collocation_points": The optimization result is giv-

Option	Default	Description
		en at the collocation points as well as the start and final time point. "element_interpolation": The values of the variable trajectories are calculated by evaluating the collocation polynomials. The algorithm option n_eval_points is used to specify the evaluation points within each finite element. "mesh_points": The optimization result is given at the mesh points.
`n_eval_points`	20	The number of evaluation points used in each element when the algorithm option result_mode is set to "element_interpolation". One evaluation point is placed at each element end-point (hence the option value must be at least 2) and the rest are distributed uniformly.
`checkpoint`	False	If `checkpoint` is set to `True`, transcribed NLP is built with packed MX functions. Instead of calling the DAE residual function, the collocation equation function, and the lagrange term function `n_e * n_cp` times, the check point scheme builds an `MXFunction` evaluating `n_cp` collocation points at the same time, so that the packed `MXFunction` is called only `n_e` times. This approach improves the code generation and it is expected to reduce the memory usage for constructing and solving the NLP.
`quadrature_constraint`	True	Whether to use quadrature continuity constraints. This option is only applicable when using Gauss collocation. It is incompatible with eliminate_der_var set to True. True: Quadrature is used to get the values of the states at the mesh points. False: The Lagrange basis polynomials for the state collocation polynomials are evaluated to get the values of the states at the mesh points.
`mutable_external_data`	True	If true and the `external_data` option is used, the external data can be changed after discretization, e.g. during warm starting.
`explicit_hessian`	False	Explicitly construct the Lagrangian Hessian, rather than rely on CasADi to automatically generate it. This is only done to circumvent a bug in CasADi, see #4313, which rarely causes the automatic Hessian to be incorrect.

Option	Default	Description
`order`	"default"	Order of variables and equations. Requires write_scaled_result! Possible values: "default", "reverse", and "random"

The last standard options, `IPOPT_options` and `WORHP_options`, serve as interfaces for setting options in IPOPT and WORHP. To exemplify the usage of these algorithm options, the maximum number of iterations in IPOPT can be set using the following syntax:

```
opts = model.optimize_options()
opts["IPOPT_options"]["max_iter"] = 10000
```

OCT's CasADi-based framework does not support simulation and initialization of models. It is recommended to use PyFMI for these purposes instead.

Some statistics from the NLP solver can be obtained by issuing the command

```
res_opt.get_solver_statistics()
```

The return argument of this function can be found by using the interactive help:

```
help(res_opt.get_solver_statistics)
Get nonlinear programming solver statistics.

Returns::

    return_status --
        Return status from nonlinear programming solver.

    nbr_iter --
        Number of iterations.

    objective --
       Final value of objective function.

    total_exec_time --
        Execution time.
```

6.5.1.1. Reusing the same discretization for several optimization solutions

When collocation is used to solve a dynamic optimization problem, the solution procedure is carried out in several steps:

- Discretize the dynamic optimization problem, which is formulated in continuous time. The result is a large and sparse nonlinear program (NLP). The discretization step depends on the options as provided to the `optimize` method.

- Solve the NLP.
- Postprocess the NLP solution to extract an approximate solution to the original dynamic optimization problem.

Depending on the problem, discretization may account for a substantial amount of the total solution time, or even dominate it.

The same discretization can be reused for several solutions with different parameter values, but the same options. Discretization will be carried out each time the `optimize` method is called on the model. Instead of calling `model.optimize(options=opts)`, a problem can be discretized using the `prepare_optimization` method:

```
solver = model.prepare_optimization(options=opts)
```

Alternatively, the solver can be retrieved from an existing optimization result, as `solver = res.get_solver()`. Manipulating the solver (e.g. setting parameters) may affect the original optimization problem object and vice versa.

The obtained solver object represents the discretized problem, and can be used to solve it using its own `optimize` method:

```
res = solver.optimize()
```

While options cannot be changed in general, parameter values, initial trajectories, external data, and NLP solver options can be changed on the solver object. Parameter values can be updated with

```
solver.set(parameter_name, value)
```

and current values retrieved with

```
solver.get(parameter_name)
```

New initial trajectories can be set with

```
solver.set_init_traj(init_traj)
```

where `init_traj` has the same format as used with the `init_traj` option.

External data can be updated with

```
solver.set_external_variable_data(variable_name, data)
```

(unless the `mutable_external_data` option is turned off). `variable_name` should correspond to one of the variables used in the `external_data` option passed to `prepare_optimization`. `data` should be the new data, in the same format as variable data used in the `external_data` option. The kind of external data used for the variable (eliminated/constrained/quadratic penalty) is not changed.

Settings to the nonlinear solver can be changed with

```
solver.set_solver_option(solver_name, name, value)
```

where `solver_name` is e g `'IPOPT'` or `'WORHP'`.

6.5.1.2. Warm starting

The solver object obtained from `prepare_optimization` can also be used for *warm starting*, where an obtained optimization solution (including primal and dual variables) is used as the initial guess for a new optimization with new parameter values.

To reuse the solver's last obtained solution as initial guess for the next optimization, warm starting can be enabled with

```
solver.set_warm_start(True)
```

before calling `solver.optimize()`. This will reuse the last solution for the primal variables (unless `solver.set_init_traj` was called since the last `solver.optimize`) as well as the last solution for the dual variables.

When using the IPOPT solver with warm starting, several solver options typically also need to be set to see the benefits, e g:

```
def set_warm_start_options(solver, push=1e-4, mu_init=1e-1):
    solver.set_solver_option('IPOPT', 'warm_start_init_point', 'yes')
    solver.set_solver_option('IPOPT', 'mu_init', mu_init)

    solver.set_solver_option('IPOPT', 'warm_start_bound_push', push)
    solver.set_solver_option('IPOPT', 'warm_start_mult_bound_push', push)
    solver.set_solver_option('IPOPT', 'warm_start_bound_frac', push)
    solver.set_solver_option('IPOPT', 'warm_start_slack_bound_frac', push)
    solver.set_solver_option('IPOPT', 'warm_start_slack_bound_push', push)

set_warm_start_options(solver)
```

Smaller values of the `push` and `mu` arguments will make the solver place more trust in that the sought solution is close to the initial guess, i e, the last solution.

6.5.2. Examples

6.5.2.1. Optimal control

This tutorial is based on the Hicks-Ray Continuously Stirred Tank Reactors (CSTR) system. The model was originally presented in [1]. The system has two states, the concentration, c, and the temperature, T. The control input to the system is the temperature, Tc, of the cooling flow in the reactor jacket. The chemical reaction in the reactor is exothermic, and also temperature dependent; high temperature results in high reaction rate. The CSTR dynamics are given by:

$$\dot{c}(t) = \frac{F_0(c_0 - c(t))}{V} - k_0 c(t) e^{-\mathrm{EdivR}/T(t)}$$
$$\dot{T}(t) = \frac{F_0(T_0 - T(t))}{V} - \frac{dHk_0 c(t)}{\rho C_p} e^{-\mathrm{EdivR}/T(t)} + \frac{2U}{r\rho C_p}(Tc(t) - T(t))$$

This tutorial will cover the following topics:

- How to solve a DAE initialization problem. The initialization model has equations specifying that all derivatives should be identically zero, which implies that a stationary solution is obtained. Two stationary points, corresponding to different inputs, are computed. We call the stationary points A and B respectively. Point A corresponds to operating conditions where the reactor is cold and the reaction rate is low, whereas point B corresponds to a higher temperature where the reaction rate is high.
- An optimal control problem is solved where the objective is to transfer the state of the system from stationary point A to point B. The challenge is to ignite the reactor while avoiding uncontrolled temperature increases. It is also demonstrated how to set parameter and variable values in a model. More information about the simultaneous optimization algorithm can be found at OCT API documentation.
- The optimization result is saved to file and then the important variables are plotted.

The Python commands in this tutorial may be copied and pasted directly into a Python shell, in some cases with minor modifications. Alternatively, you may copy the commands into a text file, e.g., `cstr_casadi.py`.

Start the tutorial by creating a working directory. In your installation there is a directory named *install*, within that folder you find the filepath `Python/pyjmi/examples/files/CSTR.mop`, copy this file to your working directory. If you choose to create a Python script file, save that as well to the working directory.

Compile and instantiate a model object

The functions and classes used in the tutorial script need to be imported into the Python script. This is done by the following Python commands. Copy them and paste them either directly into your Python shell or, preferably, into your Python script file.

```
import numpy as N
import matplotlib.pyplot as plt

from pymodelica import compile_fmu
from pyfmi import load_fmu
from pyjmi import transfer_optimization_problem
```

To solve the initialization problem and simulate the model, we will first compile it as an FMU and load it in Python. These steps are described in more detail in Section 4.

```
# Compile the stationary initialization model into an FMU
init_fmu = compile_fmu("CSTR.CSTR_Init", "CSTR.mop")

# Load the FMU
```

```
init_model = load_fmu(init_fmu)
```

At this point, you may open the file `CSTR.mop`, containing the CSTR model and the static initialization model used in this section. Study the classes `CSTR.CSTR` and `CSTR.CSTR_Init` and make sure you understand the models. Before proceeding, have a look at the interactive help for one of the functions you used:

```
help(compile_fmu)
```

Solve the DAE initialization problem

In the next step, we would like to specify the first operating point, A, by means of a constant input cooling temperature, and then solve the initialization problem assuming that all derivatives are zero.

```
# Set input for Stationary point A
Tc_0_A = 250
init_model.set('Tc', Tc_0_A)

# Solve the initialization problem using FMI
init_model.initialize()

# Store stationary point A
[c_0_A, T_0_A] = init_model.get(['c', 'T'])

# Print some data for stationary point A
print(' *** Stationary point A ***')
print('Tc = %f' % Tc_0_A)
print('c = %f' % c_0_A)
print('T = %f' % T_0_A)
```

Notice how the method `set` is used to set the value of the control input. The initialization algorithm is invoked by calling the method `initialize`, which returns a result object from which the initialization result can be accessed. The values of the states corresponding to point A can then be extracted from the result object. Look carefully at the printouts in the Python shell to see the stationary values. Display the help text for the `initialize` method and take a moment to look it through. The procedure is now repeated for operating point B:

```
# Set inputs for Stationary point B
init_model.reset() # reset the FMU so that we can initialize it again
Tc_0_B = 280
init_model.set('Tc', Tc_0_B)

# Solve the initialization problem using FMI
init_model.initialize()

# Store stationary point B
[c_0_B, T_0_B] = init_model.get(['c', 'T'])

# Print some data for stationary point B
print(' *** Stationary point B ***')
print('Tc = %f' % Tc_0_B)
print('c = %f' % c_0_B)
```

```
print('T = %f' % T_0_B)
```

We have now computed two stationary points for the system based on constant control inputs. In the next section, these will be used to set up an optimal control problem.

Solving an optimal control problem

The optimal control problem we are about to solve is given by

$$\min_{u(t)} \int_0^{150} \left(c^{ref} - c(t)\right)^2 + \left(T^{ref} - T(t)\right)^2 + \left(T_c^{ref} - T_c(t)\right)^2 dt$$

subject to

$$230 \le u(t) = T_c(t) \le 370$$

$$T(t) \le 350$$

and is expressed in Optimica format in the class `CSTR.CSTR_Opt2` in the `CSTR.mop` file above. Have a look at this class and make sure that you understand how the optimization problem is formulated and what the objective is.

Direct collocation methods often require good initial guesses in order to ensure robust convergence. Also, if the problem is non-convex, initialization is even more critical. Since initial guesses are needed for all discretized variables along the optimization interval, simulation provides a convenient means to generate state and derivative profiles given an initial guess for the control input(s). It is then convenient to set up a dedicated model for computation of initial trajectories. In the model `CSTR.CSTR_Init_Optimization` in the `CSTR.mop` file, a step input is applied to the system in order obtain an initial guess. Notice that the variable names in the initialization model must match those in the optimal control model.

First, compile the model and set model parameters:

```
# Compile the optimization initialization model
init_sim_fmu = compile_fmu("CSTR.CSTR_Init_Optimization", "CSTR.mop")

# Load the model
init_sim_model = load_fmu(init_sim_fmu)

# Set initial and reference values
init_sim_model.set('cstr.c_init', c_0_A)
init_sim_model.set('cstr.T_init', T_0_A)
init_sim_model.set('c_ref', c_0_B)
init_sim_model.set('T_ref', T_0_B)
init_sim_model.set('Tc_ref', Tc_0_B)
```

Having initialized the model parameters, we can simulate the model using the `simulate` function.

```
# Simulate with constant input Tc
init_res = init_sim_model.simulate(start_time=0., final_time=150.)
```

The method `simulate` first computes consistent initial conditions and then simulates the model in the interval 0 to 150 seconds. Take a moment to read the interactive help for the `simulate` method.

The simulation result object is returned. Python dictionary access can be used to retrieve the variable trajectories.

```
# Extract variable profiles
t_init_sim = init_res['time']
c_init_sim = init_res['cstr.c']
T_init_sim = init_res['cstr.T']
Tc_init_sim = init_res['cstr.Tc']

# Plot the initial guess trajectories
plt.close(1)
plt.figure(1)
plt.subplot(3, 1, 1)
plt.plot(t_init_sim, c_init_sim)
plt.grid()
plt.ylabel('Concentration')
plt.title('Initial guess obtained by simulation')

plt.subplot(3, 1, 2)
plt.plot(t_init_sim, T_init_sim)
plt.grid()
plt.ylabel('Temperature')

plt.subplot(3, 1, 3)
plt.plot(t_init_sim, Tc_init_sim)
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

Look at the plots and try to relate the trajectories to the optimal control problem. Why is this a good initial guess?

Once the initial guess is generated, we compile the optimal control problem:

```
# Compile and load optimization problem
op = transfer_optimization_problem("CSTR.CSTR_Opt2", "CSTR.mop")
```

We will now initialize the parameters of the model so that their values correspond to the optimization objective of transferring the system state from operating point A to operating point B. Accordingly, we set the parameters representing the initial values of the states to point A and the reference values in the cost function to point B:

```
# Set reference values
op.set('Tc_ref', Tc_0_B)
op.set('c_ref', float(c_0_B))
op.set('T_ref', float(T_0_B))

# Set initial values
op.set('cstr.c_init', float(c_0_A))
op.set('cstr.T_init', float(T_0_A))
```

We will also set some optimization options. In this case, we decrease the number of finite elements in the mesh from 50 to 19, to be able to illustrate that simulation and optimization might not give the exact same result. This is done by setting the corresponding option and providing it as an argument to the `optimize` method. We also lower the tolerance of IPOPT to get a more accurate result. We are now ready to solve the actual optimization problem. This is done by invoking the method `optimize`:

```
# Set options
opt_opts = op.optimize_options()
opt_opts['n_e'] = 19 # Number of elements
opt_opts['init_traj'] = init_res
opt_opts['nominal_traj'] = init_res
opt_opts['IPOPT_options']['tol'] = 1e-10
opt_opts['IPOPT_options']['linear_solver'] = "mumps"

# Solve the optimal control problem
res = op.optimize(options=opt_opts)
```

You should see the output of IPOPT in the Python shell as the algorithm iterates to find the optimal solution. IPOPT should terminate with a message like 'Optimal solution found' or 'Solved to acceptable level' in order for an optimum to have been found. The optimization result object is returned and the optimization data are stored in `res`.

We can now retrieve the trajectories of the variables that we intend to plot:

```
# Extract variable profiles
c_res = res['cstr.c']
T_res = res['cstr.T']
Tc_res = res['cstr.Tc']
time_res = res['time']
c_ref = res['c_ref']
T_ref = res['T_ref']
Tc_ref = res['Tc_ref']
```

Finally, we plot the result using the functions available in matplotlib:

```
# Plot the results
plt.close(2)
plt.figure(2)
plt.subplot(3, 1, 1)
plt.plot(time_res, c_res)
plt.plot(time_res, c_ref, '--')
plt.grid()
plt.ylabel('Concentration')
plt.title('Optimized trajectories')

plt.subplot(3, 1, 2)
plt.plot(time_res, T_res)
plt.plot(time_res, T_ref, '--')
plt.grid()
plt.ylabel('Temperature')
```

```
plt.subplot(3, 1, 3)
plt.plot(time_res, Tc_res)
plt.plot(time_res, Tc_ref, '--')
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

You should now see the plot shown in Figure 6.2.

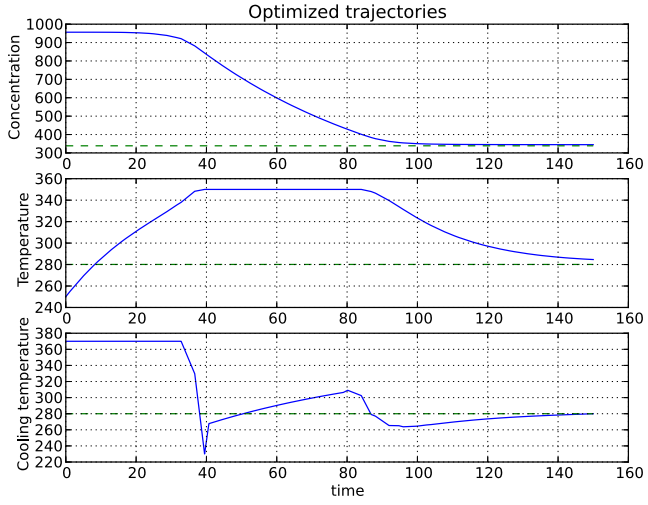


Figure 6.2 Optimal profiles for the CSTR problem.

Take a minute to analyze the optimal profiles and to answer the following questions:

1. Why is the concentration high in the beginning of the interval?

2. Why is the input cooling temperature high in the beginning of the interval?

Verify optimal control solution

Solving optimal control problems by means of direct collocation implies that the differential equation is approximated by a time-discrete counterpart. The accuracy of the solution is dependent on the method of collocation and the number of elements. In order to assess the accuracy of the discretization, we may simulate the system using the optimal control profile as input. With this approach, the state profiles are computed with high accuracy and the result may then be compared with the profiles resulting from optimization. Notice that this procedure does not verify the optimality of the resulting optimal control profiles, but only the accuracy of the discretization of the dynamics.

We start by compiling and loading the model used for simulation:

```
# Compile model
sim_fmu = compile_fmu("CSTR.CSTR", "CSTR.mop")

# Load model
sim_model = load_fmu(sim_fmu)
```

The solution obtained from the optimization are values at a finite number of time points, in this case the collocation points. The CasADi framework also supports obtaining all the collocation polynomials for all the input variables in the form of a function instead, which can be used during simulation for greater accuracy. We obtain it from the result object in the following manner.

```
# Get optimized input
(_, opt_input) = res.get_opt_input()
```

We specify the initial values and simulate using the optimal trajectory:

```
# Set initial values
sim_model.set('c_init', c_0_A)
sim_model.set('T_init', T_0_A)

# Simulate using optimized input
sim_opts = sim_model.simulate_options()
sim_opts['CVode_options']['rtol'] = 1e-6
sim_opts['CVode_options']['atol'] = 1e-8
res = sim_model.simulate(start_time=0., final_time=150.,
                         input=('Tc', opt_input), options=sim_opts)
```

Finally, we load the simulated data and plot it to compare with the optimized trajectories:

```
# Extract variable profiles
c_sim=res['c']
T_sim=res['T']
Tc_sim=res['Tc']
time_sim = res['time']

# Plot the results
plt.figure(3)
plt.clf()
plt.subplot(311)
plt.plot(time_res,c_res,'--')
plt.plot(time_sim,c_sim)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(time_res,T_res,'--')
plt.plot(time_sim,T_sim)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Temperature')
```

```
plt.subplot(313)
plt.plot(time_res,Tc_res,'--')
plt.plot(time_sim,Tc_sim)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

You should now see the plot shown in Figure 6.3.

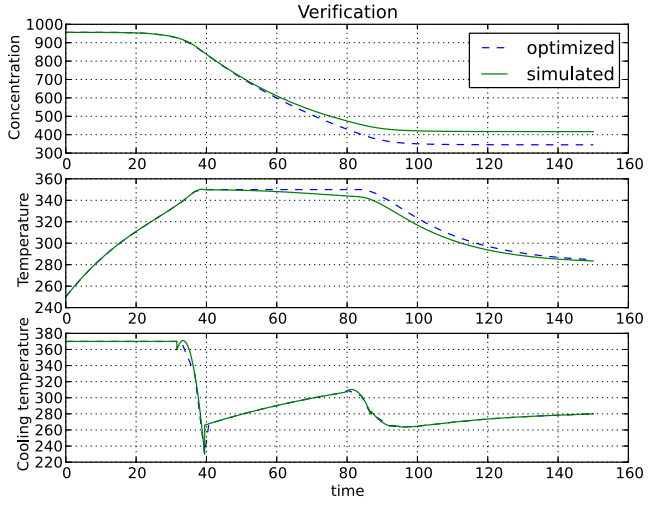


Figure 6.3 Optimal control profiles and simulated trajectories corresponding to the optimal control input.

Discuss why the simulated trajectories differ from their optimized counterparts.

Exercises

After completing the tutorial you may continue to modify the optimization problem and study the results.

1. Remove the constraint on `cstr.T`. What is then the maximum temperature?

2. Play around with weights in the cost function. What happens if you penalize the control variable with a larger weight? Do a parameter sweep for the control variable weight and plot the optimal profiles in the same figure.

3. Add terminal constraints (`cstr.T(finalTime)=someParameter`) for the states so that they are equal to point B at the end of the optimization interval. Now reduce the length of the optimization interval. How short can you make the interval?

4. Try varying the number of elements in the mesh and the number of collocation points in each interval.

References

[1] G.A. Hicks and W.H. Ray. Approximation Methods for Optimal Control Synthesis. *Can. J. Chem. Eng*., 40:522–529, 1971.

[2] Bieger, L., A. Cervantes, and A. Wächter (2002): "Advances in simultaneous strategies for dynamic optimization." *Chemical Engineering Science*, **57**, pp. 575-593.

6.5.2.2. Minimum time problems

Minimum time problems are dynamic optimization problems where not only the control inputs are optimized, but also the final time. Typically, elements of such problems include initial and terminal state constraints and an objective function where the transition time is minimized. The following example will be used to illustrate how minimum time problems are formulated in Optimica. We consider the optimization problem:

$$\min_{u(t)} t_f$$

subject to the Van der Pol dynamics:

$$\dot{x}_1 = \left(1 - x_2^2\right) x_1 - x_2 + u, \quad x_1(0) = 0$$
$$\dot{x}_2 = x_1, \quad x_2(0) = 1$$

and the constraints:

$$x_1(t_f) = 0, \quad x_2(t_f) = 0$$

$-1 \leq u(t) \leq 1$

This problem is encoded in the following Optimica specification:

```
optimization VDP_Opt_Min_Time (objective = finalTime,
                               startTime = 0,
                               finalTime(free=true,min=0.2,initialGuess=1))

  // The states
  Real x1(start = 0,fixed=true);
  Real x2(start = 1,fixed=true);

  // The control signal
  input Real u(free=true,min=-1,max=1);

equation
  // Dynamic equations
  der(x1) = (1 - x2^2) * x1 - x2 + u;
```

```
  der(x2) = x1;

constraint
  // terminal constraints
  x1(finalTime)=0;
  x2(finalTime)=0;
end VDP_Opt_Min_Time;
```

Notice how the class attribute `finalTime` is set to be free in the optimization. The problem is solved by the following Python script:

```
# Import numerical libraries
import numpy as N
import matplotlib.pyplot as plt

# Import the OCT Python packages
from pymodelica import compile_fmu
from pyfmi import load_fmu
from pyjmi import transfer_optimization_problem

vdp = transfer_optimization_problem("VDP_Opt_Min_Time", "VDP_Opt_Min_Time.mop")
res = vdp.optimize()

# Extract variable profiles
x1=res['x1']
x2=res['x2']
u=res['u']
t=res['time']

# Plot
plt.figure(1)
plt.clf()
plt.subplot(311)
plt.plot(t,x1)
plt.grid()
plt.ylabel('x1')

plt.subplot(312)
plt.plot(t,x2)
plt.grid()
plt.ylabel('x2')

plt.subplot(313)
plt.plot(t,u,'x-')
plt.grid()
plt.ylabel('u')
plt.xlabel('time')
plt.show()
```

The resulting control and state profiles are shown in Figure 6.4. Notice the difference as compared to Figure Figure 6.1, where the Van der Pol oscillator system is optimized using a quadratic objective function.

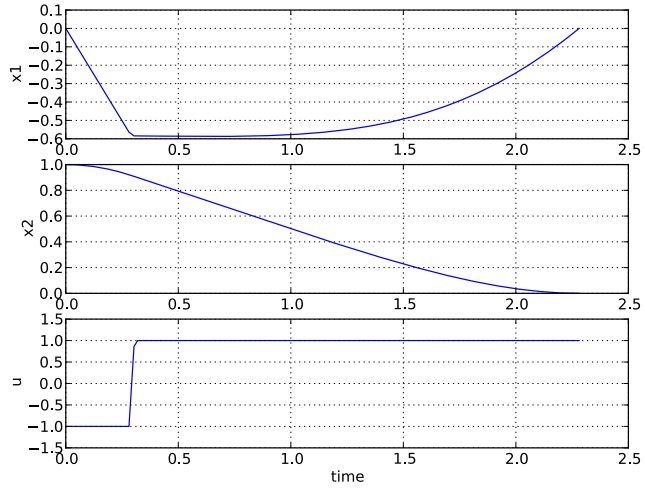


Figure 6.4 Minimum time profiles for the Van der Pol Oscillator.

6.5.2.3. Optimization under delay constraints

In some applications, it can be useful to solve dynamic optimization problems that include time delays in the model. Collocation based optimization schemes are well suited to handle this kind of models, since the whole state trajectory is available at the same time. The direct collocation method using CasADi contains an experimental implementation of such delays, which we will describe with an example. Please note that the implementation of this feature is experimental and subject to change.

We consider the optimization problem

$$\min_{u(t)} \int_0^1 (4x(t)^2 + u(t)_1^2 + u(t)_2^2)\, dt$$

subject to the dynamics

$$\dot{x}(t) = u_1(t) - 2u_2(t)$$
$$u_2(t) = u_1(t - t_{delay})$$

and the boundary conditions

$$x(0) = 1$$
$$x(1) = 0$$
$$u_2(t) = 0.25,\ t < t_{delay}$$

The effect of positive u_1 is initially to increase x, but after a time delay of time t_{delay}, it comes back with twice the effect in the negative direction through u_2.

We model everything except the delay constraint in the Optimica specification

```
optimization DelayTest(startTime = 0, finalTime = 1,
                       objectiveIntegrand = 4*x^2 + u1^2 + u2^2)
    input Real u1, u2;
    Real x(start = 1, fixed=true);
equation
    der(x) = u1 - 2*u2;
constraint
    x(finalTime) = 0;
end DelayTest;
```

The problem is then solved in the following Python script. Notice how the delay constraint is added using the `delayed_feedback` option, and the initial part of u_2 is set using the `initialGuess` attribute:

```
# Import numerical libraries
import numpy as np
import matplotlib.pyplot as plt

# Import OCT Python packages
from pyjmi import transfer_optimization_problem

n_e = 20
delay_n_e = 5
horizon = 1.0
delay = horizon*delay_n_e/n_e

# Compile and load optimization problem
opt = transfer_optimization_problem("DelayTest", "DelayedFeedbackOpt.mop")

# Set value for u2(t) when t < delay
opt.getVariable('u2').setAttribute('initialGuess', 0.25)

# Set algorithm options
opts = opt.optimize_options()
opts['n_e'] = n_e
# Set delayed feedback from u1 to u2
opts['delayed_feedback'] = {'u2': ('u1', delay_n_e)}

# Optimize
res = opt.optimize(options=opts)

# Extract variable profiles
x_res = res['x']
u1_res = res['u1']
u2_res = res['u2']
time_res = res['time']
```

```
# Plot results
plt.plot(time_res, x_res, time_res, u1_res, time_res, u2_res)
plt.plot(time_res+delay, u1_res, '--')
plt.legend(('x', 'u1', 'u2', 'delay(u1)'))
plt.show()
```

The resulting control and state profiles are shown in Figure 6.5. Notice that x grows initially since u_1 is set positive to exploit the greater control gain that appears delayed through u_2. At time $1 - t_{\text{delay}}$, the delayed value of u_1 ceases to influence x within the horizon, and u_1 immediately switches sign to drive down x to its final value $x(1) = 0$.

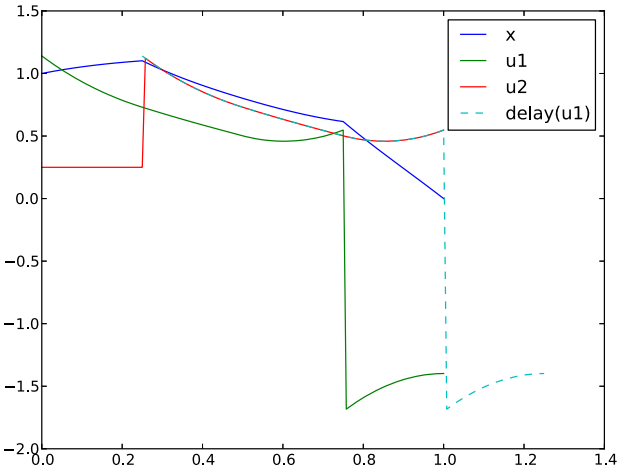


Figure 6.5 Optimization result for delayed feedback example.

6.5.2.4. Parameter estimation

In this tutorial it will be demonstrated how to solve parameter estimation problems. We consider a quadruple tank system depicted in Figure 6.6.

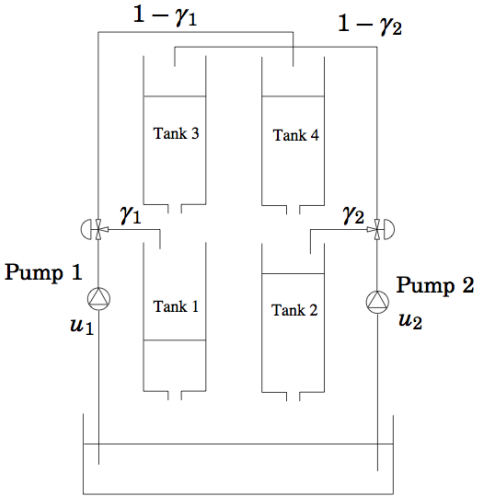


Figure 6.6 A schematic picture of the quadruple tank process.

The dynamics of the system are given by the differential equations:

$$\dot{x}_1 = -\frac{a_1}{A_2}\sqrt{2gx_1} + \frac{a_3}{A_1}\sqrt{2gx_3} + \frac{\gamma_1 k_1}{A_1}u_1$$

$$\dot{x}_2 = -\frac{a_2}{A_2}\sqrt{2gx_2} + \frac{a_4}{A_2}\sqrt{2gx_4} + \frac{\gamma_2 k_2}{A_2}u_2$$

$$\dot{x}_3 = -\frac{a_3}{A_3}\sqrt{2gx_3} + \frac{(1-\gamma_2)k_2}{A_3}u_2$$

$$\dot{x}_4 = -\frac{a_4}{A_4}\sqrt{2gx_4} + \frac{(1-\gamma_1)k_1}{A_4}u_1$$

Where the nominal parameter values are given in Table 6.3.

Table 6.3 Parameters for the quadruple tank process.

Parameter name	Value	Unit
A_i	4.9	cm^2
a_i	0.03	cm^2
k_i	0.56	$cm^2V^{-1}s^{-1}$
$\#_i$	0.3	Vcm^{-1}

The states of the model are the tank water levels x1, x2, x3, and x4. The control inputs, u1 and u2, are the flows generated by the two pumps.

The Modelica model for the system is located in QuadTankPack.mop. Download the file to your working directory and open it in a text editor. Locate the class `QuadTankPack.QuadTank` and make sure you understand the model. In particular, notice that all model variables and parameters are expressed in SI units.

Measurement data, available in `qt_par_est_data.mat`, has been logged in an identification experiment. Download also this file to your working directory.

Open a text file and name it `qt_par_est_casadi.py`. Then enter the imports:

```
import os
from collections import OrderedDict

from scipy.io.matlab.mio import loadmat
import matplotlib.pyplot as plt
import numpy as N

from pymodelica import compile_fmu
from pyfmi import load_fmu
from pyjmi import transfer_optimization_problem
from pyjmi.optimization.casadi_collocation import ExternalData
```

into the file. Next, we compile the model, which is used for simulation, and the optimization problem, which is used for estimating parameter values. We will take a closer look at the optimization formulation later, so do not worry about that one for the moment. The initial states for the experiment are stored in the optimization problem, which we propagate to the model for simulation.

```
# Compile and load FMU, which is used for simulation
model = load_fmu(compile_fmu('QuadTankPack.QuadTank', "QuadTankPack.mop"))

# Transfer problem to CasADi Interface, which is used for estimation
op = transfer_optimization_problem("QuadTankPack.QuadTank_ParEstCasADi",
                                   "QuadTankPack.mop")

# Set initial states in model, which are stored in the optimization problem
x_0_names = ['x1_0', 'x2_0', 'x3_0', 'x4_0']
x_0_values = op.get(x_0_names)
model.set(x_0_names, x_0_values)
```

Next, we enter code to open the data file, extract the measurement time series and plot the measurements:

```
# Load measurement data from file
data = loadmat("qt_par_est_data.mat", appendmat=False)

# Extract data series
t_meas = data['t'][6000::100, 0] - 60
y1_meas = data['y1_f'][6000::100, 0] / 100
y2_meas = data['y2_f'][6000::100, 0] / 100
y3_meas = data['y3_d'][6000::100, 0] / 100
y4_meas = data['y4_d'][6000::100, 0] / 100
```

```
u1 = data['u1_d'][6000::100, 0]
u2 = data['u2_d'][6000::100, 0]

# Plot measurements and inputs
plt.close(1)
plt.figure(1)
plt.subplot(2, 2, 1)
plt.plot(t_meas, y3_meas)
plt.title('x3')
plt.grid()
plt.subplot(2, 2, 2)
plt.plot(t_meas, y4_meas)
plt.title('x4')
plt.grid()
plt.subplot(2, 2, 3)
plt.plot(t_meas, y1_meas)
plt.title('x1')
plt.xlabel('t[s]')
plt.grid()
plt.subplot(2, 2, 4)
plt.plot(t_meas, y2_meas)
plt.title('x2')
plt.xlabel('t[s]')
plt.grid()

plt.close(2)
plt.figure(2)
plt.subplot(2, 1, 1)
plt.plot(t_meas, u1)
plt.title('u1')
plt.grid()
plt.subplot(2, 1, 2)
plt.plot(t_meas, u2)
plt.title('u2')
plt.xlabel('t[s]')
plt.grid()
plt.show()
```

You should now see two plots showing the measurement state profiles and the control input profiles similar to Figure 6.7 and Figure 6.8.

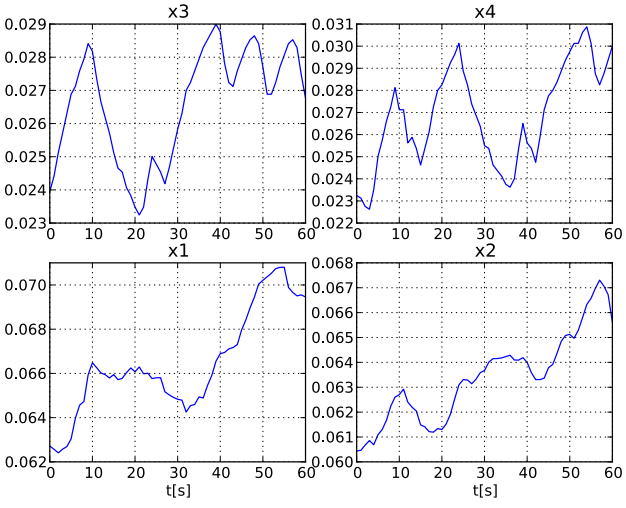


Figure 6.7 Measured state profiles.

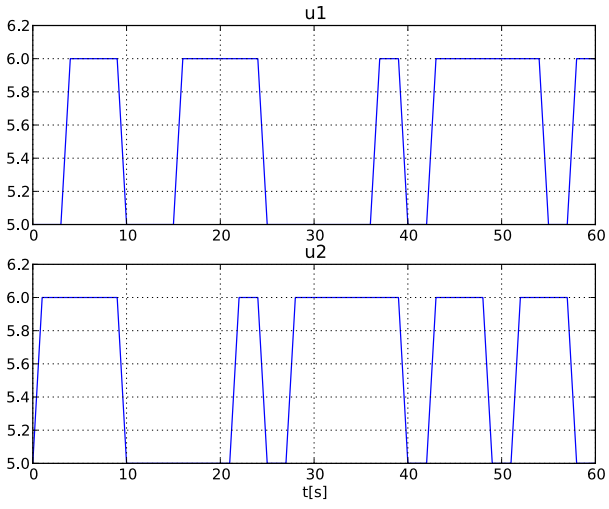


Figure 6.8 Control inputs used in the identification experiment.

In order to evaluate the accuracy of nominal model parameter values, we simulate the model using the same initial state and inputs values as in the performed experiment used to obtain the measurement data. First, a matrix containing the input trajectories is created:

```
# Build input trajectory matrix for use in simulation
```

```
u = N.transpose(N.vstack([t_meas, u1, u2]))
```

Now, the model can be simulated:

```
# Simulate model response with nominal parameter values
res_sim = model.simulate(input=(['u1', 'u2'], u),
                         start_time=0., final_time=60.)
```

The simulation result can now be extracted:

```
# Load simulation result
x1_sim = res_sim['x1']
x2_sim = res_sim['x2']
x3_sim = res_sim['x3']
x4_sim = res_sim['x4']
t_sim  = res_sim['time']
u1_sim = res_sim['u1']
u2_sim = res_sim['u2']
```

and then plotted:

```
# Plot simulation result
plt.figure(1)
plt.subplot(2, 2, 1)
plt.plot(t_sim, x3_sim)
plt.subplot(2, 2, 2)
plt.plot(t_sim, x4_sim)
plt.subplot(2, 2, 3)
plt.plot(t_sim, x1_sim)
plt.subplot(2, 2, 4)
plt.plot(t_sim, x2_sim)

plt.figure(2)
plt.subplot(2, 1, 1)
plt.plot(t_sim, u1_sim, 'r')
plt.subplot(2, 1, 2)
plt.plot(t_sim, u2_sim, 'r')
plt.show()
```

Figure 6.9 shows the result of the simulation.

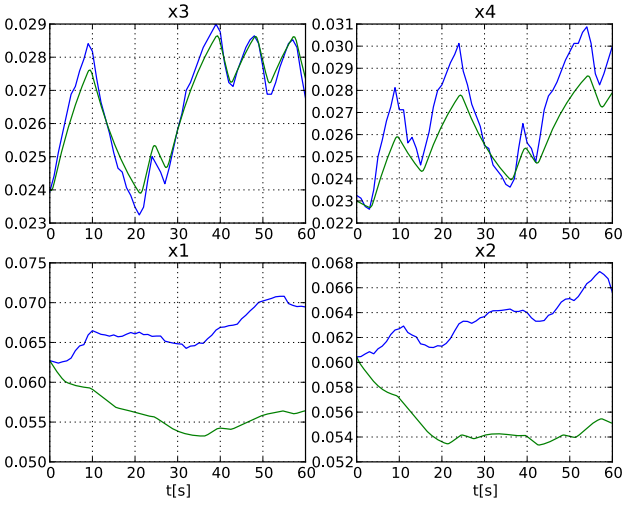


Figure 6.9 Simulation result for the nominal model.

Here, the simulated profiles are given by the green curves. Clearly, there is a mismatch in the response, especially for the two lower tanks. Think about why the model does not match the data, i.e., which parameters may have wrong values.

The next step towards solving a parameter estimation problem is to identify which parameters to tune. Typically, parameters which are not known precisely are selected. Also, the selected parameters need of course affect the mismatch between model response and data, when tuned. In a first attempt, we aim at decreasing the mismatch for the two lower tanks, and therefore we select the lower tank outflow areas, a1 and a2, as parameters to optimize. The Optimica specification for the estimation problem is contained in the class `QuadTankPack.QuadTank_ParEstCasADi`:

```
optimization QuadTank_ParEstCasADi(startTime=0, finalTime=60)

    extends QuadTank(x1(fixed=true), x1_0=0.06255,
                     x2(fixed=true), x2_0=0.06045,
                     x3(fixed=true), x3_0=0.02395,
                     x4(fixed=true), x4_0=0.02325,
                     a1(free=true, min=0, max=0.1e-4),
                     a2(free=true, min=0, max=0.1e-4));

end QuadTank_ParEstCasADi;
```

We have specified the time horizon to be one minute, which matches the length of the experiment, and that we want to estimate a1 and a2 by setting `free=true` for them. Unlike optimal control, the cost function is not specified using Optimica. This is instead specified from Python, using the `ExternalData` class and the code below.

```
# Create external data object for optimization
```

```
Q = N.diag([1., 1., 10., 10.])
data_x1 = N.vstack([t_meas, y1_meas])
data_x2 = N.vstack([t_meas, y2_meas])
data_u1 = N.vstack([t_meas, u1])
data_u2 = N.vstack([t_meas, u2])
quad_pen = OrderedDict()
quad_pen['x1'] = data_x1
quad_pen['x2'] = data_x2
quad_pen['u1'] = data_u1
quad_pen['u2'] = data_u2
external_data = ExternalData(Q=Q, quad_pen=quad_pen)
```

This will create an objective which is the integral of the squared difference between the measured profiles for x1 and x2 and the corresponding model profiles. We will also introduce corresponding penalties for the two input variables, which are left as optimization variables. It would also have been possible to eliminate the input variables from the estimation problem by using the `eliminated` parameter of `ExternalData`. See the documentation of `ExternalData` for how to do this. Finally, we use a square matrix Q to weight the different components of the objective. We choose larger weights for the inputs, as we have larger faith in those values.

We are now ready to solve the optimization problem. We first set some options, where we specify the number of elements (time-discretization grid), the external data, and also provide the simulation with the nominal parameter values as an initial guess for the solution, which is also used to scale the variables instead of the variables' nominal attributes (if they have any):

```
# Set optimization options and optimize
opts = op.optimize_options()
opts['n_e'] = 60 # Number of collocation elements
opts['external_data'] = external_data
opts['init_traj'] = res_sim
opts['nominal_traj'] = res_sim
res = op.optimize(options=opts) # Solve estimation problem
```

Now, let's extract the optimal values of the parameters a1 and a2 and print them to the console:

```
# Extract estimated values of parameters
a1_opt = res.initial("a1")
a2_opt = res.initial("a2")

# Print estimated parameter values
print('a1: ' + str(a1_opt*1e4) + 'cm^2')
print('a2: ' + str(a2_opt*1e4) + 'cm^2')
```

You should get an output similar to:

```
a1: 0.0266cm^2
a2: 0.0271cm^2
```

The estimated values are slightly smaller than the nominal values - think about why this may be the case. Also note that the estimated values do not necessarily correspond to the physically true values. Rather, the parameter

values are adjusted to compensate for all kinds of modeling errors in order to minimize the mismatch between model response and measurement data.

Next we plot the optimized profiles:

```
# Load state profiles
x1_opt = res["x1"]
x2_opt = res["x2"]
x3_opt = res["x3"]
x4_opt = res["x4"]
u1_opt = res["u1"]
u2_opt = res["u2"]
t_opt  = res["time"]

# Plot estimated trajectories
plt.figure(1)
plt.subplot(2, 2, 1)
plt.plot(t_opt, x3_opt, 'k')
plt.subplot(2, 2, 2)
plt.plot(t_opt, x4_opt, 'k')
plt.subplot(2, 2, 3)
plt.plot(t_opt, x1_opt, 'k')
plt.subplot(2, 2, 4)
plt.plot(t_opt, x2_opt, 'k')

plt.figure(2)
plt.subplot(2, 1, 1)
plt.plot(t_opt, u1_opt, 'k')
plt.subplot(2, 1, 2)
plt.plot(t_opt, u2_opt, 'k')
plt.show()
```

You will see the plot shown in Figure 6.10.

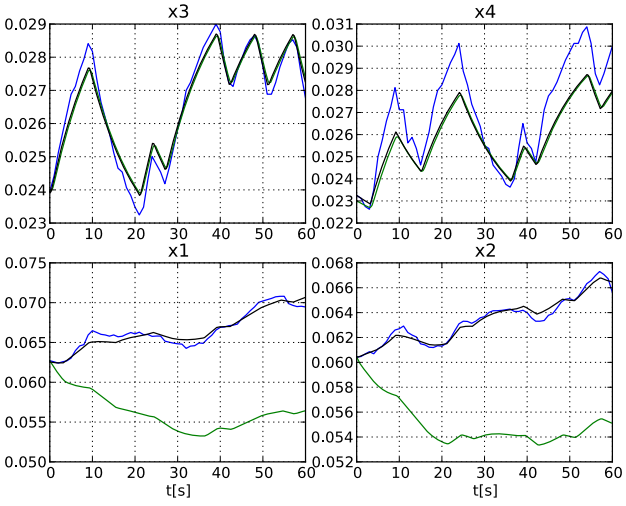


Figure 6.10 State profiles corresponding to estimated values of a1 and a2.

The profiles corresponding to the estimated values of a1 and a2 are shown in black curves. As can be seen, the match between the model response and the measurement data has been significantly improved. Is the behavior of the model consistent with the estimated parameter values?

Nevertheless, there is still a mismatch for the upper tanks, especially for tank 4. In order to improve the match, a second estimation problem may be formulated, where the parameters a1, a2, a3, a4 are free optimization variables, and where the squared errors of all four tank levels are penalized. Do this as an exercise!

6.5.3. Investigating optimization progress

This section describes some tools that can be used to investigate the progress of the nonlinear programming solver on an optimization problem. This information can be useful when debugging convergence problems; some of it (e.g. dual variables) may also be useful to gain a better understanding of the properties of an optimization problem. To make sense of the information that can be retrieved, we first give an overview of the collocation procedure that transcribes the optimization problem into a Nonlinear Program (NLP).

Methods for inspecting progress are divided into low level and high level methods, where the low level methods provide details of the underlying NLP while the high level methods are oriented towards the optimization problem as seen in the model formulation.

All functionality related to inspection of solver progress is exposed through the solver object as returned through the `prepare_optimization` method. If the optimization has been done through the `optimize` method instead, the solver can be obtained as in

```
res = op.optimize(options=opts)
```

```
solver = res.get_solver()
```

6.5.3.1. Collocation

To be able to solve a dynamic optimization problem, it is first discretized through collocation. Time is divided into *elements* (time intervals), and time varying variables are approximated by a low order polynomial over each element. Each polynomial piece is described by sample values at a number of *collocation points* (default 3) within the element. The result is that each time varying variable in the model is *instantiated* into one NLP variable for each collocation point within each element. Some variables may also need to be instantiated at additional points, such as the initial point which is typically not a collocation point.

The equations in a model are divided into initial equations, DAE equations, path constraints and point constraints. These equations are also instantiated at different time points to become constraints in the NLP. Initial equations and point constraints are instantiated only once. DAE equations and path constraints are instantiated at collocation point of each element and possibly some additional points.

When using the methods described below, each model equation is referred to as a pair `(eqtype, eqind)`. The string `eqtype` may be either `'initial'`, `'dae'`, `'path_eq'`, `'path_ineq'`, `'point_eq'`, or `'point_ineq'`. The equation index `eqind` gives the index within the given equation type, and is a nonnegative integer less than the number of equations within the type. The symbolic model equations corresponding to given pairs `(eqtype, eqind)` can be retrieved through the `get_equations` method:

```
eq      = solver.get_equations(eqtype, 0)     # first equation of type eqtype
eqs     = solver.get_equations(eqtype, [1,3]) # second and fourth equation
all_eqs = solver.get_equations(eqtype)        # all equations of the given type
```

Apart from the model equations, collocation may also instantiate additional kinds of constraints, such as *continuity constraints* to enforce continuity of states between elements and *collocation constraints* to prescribe the coupling between states and their derivatives. These constraints have their own `eqtype` strings. A list of all equation types that are used in a given model can be retrieved using

```
eqtypes = solver.get_constraint_types()
```

6.5.3.2. Inspecting residuals

Given a potential solution to the NLP, the *residual* of a constraint is a number that specifies how close it is to being satisfied. For equalities, the residual must be (close to) zero for the solution to be feasible. For inequalities, the residual must be in a specified range, typically nonpositive. The constraint *violation* is zero if the residual is within bounds, and gives the signed distance to the closest bound otherwise; for equality constraints, this is the same as the residual. Methods for returning residuals actually return the violation by default, but have an option to get the raw residual.

For a feasible solution, all violations are (almost) zero. If an optimization converges to an infeasible point or does not have time to converge to a feasible one then the residuals show which constraints the NLP solver was unable to satisfy. If one problematic constraint comes into conflict with a number of constraints, all of them will likely have nonzero violations.

Residual values for a given equation type can be retrieved as a function of time through

```
r = solver.get_residuals(eqtype)
```

where `r` is an array of residuals of shape `(n_timepoints, n_equations)`. There are also optional arguments: `inds` gives a subset of equation indices (e.g. `inds=[0, 1]`), `point` specifies whether to evaluate residuals at the optimization solution (`point='opt'`, default) or the initial point (`point='init'`), and `raw` specifies whether to return constraint violations (`raw=False`, default) or raw residuals (`raw=True`).

The corresponding time points can be retrieved with

```
t, i, k = solver.get_constraint_points(eqtype)
```

where `t`, `i`, and `k` are vectors that give the time, element index, and collocation point index for each instantiation.

To get an overview of which residuals are the largest,

```
solver.get_residual_norms()
```

returns a list of equation types sorted by descending residual norm, and

```
solver.get_residual_norms(eqtype)
```

returns a list of equation indices of the given type sorted by residual norm.

By default, the methods above work with the unscaled residuals that result directly from collocation. If the `equation_scaling` option is turned on, the constraints will be rescaled before they are sent to the NLP solver. It might be of more interest to look at the size of the scaled residuals, since these are what the NLP solver will try to make small. The above methods can then be made to work with the scaled residuals instead of the unscaled by use of the `scaled=True` keyword argument. The residual scale factors can also be retrieved in analogy to `solver.get_residuals` through

```
scales = solver.get_residual_scales(eqtype)
```

and an overview of the residual scale factors (or inverse scale factors with `inv=True`) can be gained from

```
solver.get_residual_scale_norms()
```

6.5.3.3. Inspecting the constraint Jacobian

When solving the collocated NLP, the NLP solver typically has to evaluate the Jacobian of the constraint residual functions. Convergence problems can sometimes be related to numerical problems with the constraint Jacobian. In particular, Ipopt will never consider a potential solution if there are nonfinite (infinity or not-a-number) entries in the Jacobian. If the Jacobian has such entries at the initial guess, the optimizer will give up completely.

The constraint Jacobian comes from the NLP. As seen from the original model, it contains the derivatives of the model equations (and also e.g. the collocation equations) with respect to the model variables at different time points. If one or several problematic entries are found in the Jacobian, it is often helpful to know the model equation and variable that they correspond to.

The set of (model equation, model variable) pairs that correspond to nonfinite entries in the constraint Jacobian can be printed with

```
solver.print_nonfinite_jacobian_entries()
```

or returned with

```
entries = solver.find_nonfinite_jacobian_entries()
```

There are also methods to allow to make more custom analyses of this kind. To instead list all Jacobian entries with an absolute value greater than 10, one can use

```
J = solver.get_nlp_jacobian() # Get the raw NLP constraint Jacobian as a (sparse)
 scipy.csc_matrix

# Find the indices of all entries with absolute value > 10
J.data = abs(J.data) > 10
c_inds, xx_inds = N.nonzero(J)

entries = solver.get_model_jacobian_entries(c_inds, xx_inds) # Map the indices to equations
 and variables in the model
solver.print_jacobian_entries(entries) # Print them
```

To get the Jacobian with residual scaling applied, use the `scaled_residuals=True` option.

6.5.3.4. Inspecting dual variables

Many NLP solvers (including Ipopt) produce a solution that consists of not only the primal variables (the actual NLP variables), but also one *dual variable* for each constraint in the NLP. Upon convergence, the value of each dual variable gives the change in the optimal objective per unit change in the residual. Thus, the dual variables can give an idea of which constraints are most hindering when it comes to achieving a lower objective value, however, they must be interpreted in relation to how much it might be possible to change any given constraint.

Dual variable values for a given equation type can be retrieved as a function of time through

```
d = solver.get_constraint_duals(eqtype)
```

in analogy to `solver.get_residuals`. To get constraint duals for the equation scaled problem, use the `scaled=True` keyword argument. Just as with `get_residuals`, the corresponding time points can be retrieved with

```
t, i, k = solver.get_constraint_points(eqtype)
```

Besides regular constraints, the NLP can also contain upper and lower bounds on variables. These will correspond to the Modelica `min` and `max` attributes for instantiated model variables. The dual variables for the bounds on a given model variable `var` can be retrieved as a function of time through

```
d = solver.get_bound_duals(var)
```

The corresponding time points can be retrieved with

```
t, i, k = solver.get_variable_points(var)
```

6.5.3.5. Inspecting low level information about NLP solver progress

The methods described above generally hide the actual collocated NLP and only require to work with model variables and equations, instantiated at different points. There also exist lower level methods that expose the NLP level information and its mapping to the original model more directly, and may be useful for more custom applications. These include

- `get_nlp_variables`, `get_nlp_residuals`, `get_nlp_bound_duals`, and `get_nlp_constraint_duals` to get raw vectors from the NLP solution.
- `get_nlp_variable_bounds` and `get_nlp_residual_bounds` to get the corresponding bounds used in the NLP.
- `get_nlp_residual_scales` to get the raw residual scale factors.
- `get_nlp_variable_indices` and `get_nlp_constraint_indices` to get mappings from model variables and equations to their NLP counterparts.
- `get_point_time` to get the times of collocation points `(i, k)`.
- `get_model_variables` and `get_model_constraints` to map from NLP variables and constraints to the corresponding model variables and equations.

The low level constraint Jacobian methods `get_nlp_jacobian`, `get_model_jacobian_entries`, and the `print_jacobian_entries` method have already been covered in the section about jacobians above.

See the docstring for the respective method for more information.

6.5.4. Eliminating algebraic variables

When the algorithm of this section is used, it is applied on the full DAE, meaning that all of the algebraic variables and equations are exposed to the numerical discretization and need to be solved by the NLP solver. It is often beneficial to instead solve some of these algebraic equations in a symbolic pre-processing step. This subsection describes how this can be done.

OCT has two different frameworks for performing such eliminations. The first one is not described in this User's Guide, but an example demonstrating its use can be found in `pyjmi.examples.ccpp_elimination`. It is implemented as a part of CasADi Interface, whereas the second framework, which is the focus of this subsection, is implemented in Python. The elimination framework in CasADi Interface has faster pre-processing, but has limitations regarding what kind of algebraic variables it can eliminate and also lacks important features such as tearing and sparsity preservation. For more details on the inner workings of the Python-based framework, see Chapter 4 in [Mag2016].

6.5.4.1. Basic use

To leave everything in the hands of the framework, simply transfer an optimization problem as per usual and use the following Python code snippet.

```
from pyjmi.symbolic_elimination import BLTOptimizationProblem, EliminationOptions
op = transfer_optimization_problem(class_name, file_name) # Regular compilation
op = BLTOptimizationProblem(op) # Symbolically eliminate algebraic variables
```

You can then proceed as usual. There is however one caveat. The min and max attributes of eliminated algebraic variables will not be respected. If this is undesired, these bounds should either be converted into constraints (not recommended), or the corresponding variables should be marked as ineliminable as described in Section 6.5.4.2.

6.5.4.2. Small example

To demonstrate the use and effects of the framework, we consider the example `pyjmi.examples.elimination_example`. Note that this example is intended to be pedagogical, rather than showing the performance gains of the techniques. For a real-world example where the framework offers significant performance gains, see `pyjmi.examples.ccpp_sym_elim`, where the solution time is reduced by a factor of 5.

The following artificial Modelica and Optimica code is used in this example.

```
optimization EliminationExample(finalTime=4,
        objectiveIntegrand=(x1-0.647)^2+x2^2+(u-0.0595)^2+(y1-0.289)^2)
    Real x1(start=1, fixed=true);
    Real x2(start=1, fixed=true);
    Real y1(start=0.3, max=0.41);
    Real y2(start=1);
    Real y3(start=1);
    Real y4(start=1);
    Real y5(start=1);
    input Real u;
equation
    der(x1) = x2;
    der(x2) + y1 + y2 - y3 = u;
    x1*y3 + y2 - sqrt(x1) - 2 = 0;
    2*y1*y2*y4 - sqrt(x1) = 0;
    y1*y4 + sqrt(y3) - x1 - y4 = u;
    y4 - sqrt(y5) = 0;
    y5^2 - x1 = 0;
end EliminationExample;
```

We start as usual by transferring the optimization problem to CasADi Interface.

```
op = transfer_optimization_problem("EliminationExample", file_path, compiler_options={})
```

Next we prepare the symbolic elimination. An important part of this is the manual selection of algebraic variables that are not allowed to be eliminated. In general, it is recommended to not eliminate the following variables:

- **Variables with potentially active bounds (min or max attributes).** When variables are eliminated, their min and max attributes are neglected. This is because many Modelica variables have min and max attributes that are not intended to constrain the optimization solution. Preserving these bounds during elimination is highly inefficient. Since there is no way for the toolchain to know which variables may be actively constrained by their min and max attributes, it is up to the user to provide the names of these variables.

- **Variables that occur in the objective or constraints.** Marking these variables as ineliminable is less important, but can yield performance improvements.
- **Variables that lead to numerically unstable pivots.** When employing tearing, one runs the risk of causing numerically unstable computations. This is difficult to predict, but experienced users may know that certain variables should be selected for tearing to prevent instability, which can be achieved by marking them as ineliminable, which does not require a corresponding tearing residual to be chosen. Further details on manual tearing is described in Section 6.5.4.4.

In our small example, the only thing we have to worry about is y1, which has an upper bound. To mark y1 as ineliminable, we use the following code.

```
elim_opts = EliminationOptions()
elim_opts['ineliminable'] = ['y1'] # List of variable names
```

The elim_opts dictionary object is used to set any other elimination options, which are described in Section 6.5.4.5. For now, we just enable the option to make a plot of the block-lower triangular (BLT) decomposition of the DAE incidence matrix, which gives insight regarding the performed eliminations (see [Mag2016]).

```
elim_opts['draw_blt'] = True
elim_opts['draw_blt_strings'] = True
```

Now we are ready to symbolically transform the optimization problem.

```
op = BLTOptimizationProblem(op, elim_opts)
```

This prints the following simple problem statistics.

```
System has 5 algebraic variables before elimination and 4 after.
The three largest BLT blocks have sizes 3, 1, and 1.
```

Since we enable the BLT drawing, we also get the following plot.

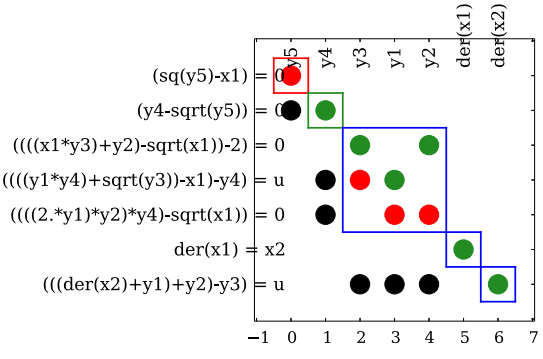


Figure 6.11 Simple example BLT decomposition.

The only variable we were able to eliminate was y4. For details on what all the colors mean in the figure, see Section 6.5.4.3.

6.5.4.3. The many colors of symbolic elimination

In the plots generated by enabling the option `draw_blt`, linear (with respect to the block variables) incidences are marked by green dots, and nonlinear incidences are marked by red dots. There is no distinction made between linear and nonlinear incidences outside of the diagonal blocks. Hence, such incidences are marked by black dots. Torn blocks are marked by red edges. Variables, and their respective matched equations, that have been user-specified as actively bounded (and hence are not eliminated) are marked by orange edges. State variable derivatives (which are not eliminated) and their respective matched equations are marked by blue edges. Blue edges are also used to mark non-scalar blocks that have not been torn. Variable–equation pairs along the diagonal that are not sparsity preserving are marked by yellow edges. The remaining variable–equation pairs along the diagonal are the ones used for elimination, which are marked by green edges.

6.5.4.4. Tearing

By default, tearing is not used in the elimination. The use of tearing enables the elimination of variables in algebraic loops. In this example, we can also eliminate y2 through tearing. Tearing can either be done automatically or manually. Manual tearing is performed on the `OptimizationProblem` object, prior to symbolic transformation. To eliminate y2, we select the other variables in the algebraic loop for y2—that is, y3 and y1—as tearing variables as follows.

```
op.getVariable('y1').setTearing(True)
op.getVariable('y3').setTearing(True)
```

We also have to select tearing residuals. This is less convenient, as there is no straightforward way to identify an equation. We can either manually inspect the equations obtained from `op.getDaeEquations()`, or search through the string representations of all of them. We will adopt the second approach.

```
for eq in op_manual.getDaeEquations():
    eq_string = eq.getResidual().repr()
    if 'y1)*y2)*y4)' in eq_string or 'y1*y4' in eq_string:
        eq.setTearing(True)
```

Once the tearing selection is finished, the symbolic transformation can be performed as before by instantiating `BLTOptimizationProblem`.

For this example, we can get the same result by automatic tearing, which is enabled during compilation. We previously used `compiler_options={}`. By instead using

```
compiler_options = {'equation_sorting': True, 'automatic_tearing': True}
```

tearing will be performed automatically. This will mark the same variables and equations as tearing variables as we just did manually. Hence, it may be a good idea to first perform tearing automatically and then make any needed changes manually, rather than doing manual tearing from scratch. Automatic tearing will yield satisfactory performance for most problems, so manual tearing is only recommended for experts. For this example, we can also eliminate y1 through manual tearing, but since we have a bound on y1, this is not recommended anyway.

6.5.4.5. Available options

The standard elimination options are listed below. All of these have been explained in the above subsections, except for the last two related to sparsity preservation. A higher density tolerance will allow for the elimination of more algebraic variables, but the resulting DAE will be more dense. This parameter thus allows a trade-off between the sparsity and dimension of the DAE, both of which affect the performance of the optimization.

Table 6.4 Standard options for the symbolic elimination.

Option	Default	Description
`draw_blt`	False	Whether to plot the BLT form.
`draw_blt_strings`	False	Whether to annotate plot of the BLT form with strings for variables and equations.
`tearing`	True	Whether to tear algebraic loops.
`ineliminable`	[]	List of names of variables that should not be eliminated. Particularly useful for variables with bounds.
`dense_measure`	'lmfi'	Density measure for controlling density in causalized system. Possible values: ['lmfi', 'Markowitz']. Markowitz uses the Markowitz criterion and lmfi uses local minimum fill-in to estimate density.
`dense_tol`	15	Tolerance for controlling density in causalized system. Possible values: [-inf, inf]

The below table lists the experimental and debugging elimination options, which should not be used by the typical user.

Table 6.5 Experimental and debugging options for the symbolic elimination.

Option	Default	Description
`plots`	False	Whether to plot intermediate results for matching and component computation.
`solve_blocks`	False	Whether to factorize coefficient matrices in non-scalar, linear blocks.
`solve_torn_linear_blocks`	False	Whether to solve causalized equations in torn blocks, rather than doing forward substitution as for nonlinear blocks.
`inline`	True	Whether to inline function calls (such as creation of linear systems).
`linear_solver`	"symbolicqr"	Which linear solver to use. See http://casadi.sourceforge.net/api/html/d8/d6a/classcasadi_1_1LinearSolver.html for possibilities

Option	Default	Description
`closed_form`	False	Whether to create a closed form expression for residuals and solutions. Disables computations.
`inline_solved`	False	Whether to inline solved expressions in the closed form expressions (only applicable if closed_form == True).

6.6. Derivative-Free Model Calibration of FMUs

Figure 6.12 The Furuta pendulum.

This tutorial demonstrates how to solve a model calibration problem using an algorithm that can be applied to Functional Mock-up Units. The model to be calibrated is the Furuta pendulum shown in Figure 6.12. The Furuta pendulum consists of an arm rotating in the horizontal plane and a pendulum which is free to rotate in the vertical plane. The construction has two degrees of freedom, the angle of the arm, φ, and the angle of the pendulum, θ. Copy the file `$JMODELICA_HOME/Python/pyjmi/examples/files/FMUs/Furuta.fmu` to your working directory. **Note that the Furuta.fmu file is currently only supported on Windows.** Measurement data for φ and θ is available in the file `$JMODELICA_HOME/Python/pyjmi/examples/files/FurutaData.mat`. Copy this file to your working directory as well. These measurements will be used for the calibration. Open a text file, name it `furuta_par_est.py` and enter the following imports:

```
from scipy.io.matlab.mio import loadmat
import matplotlib.pyplot as plt
import numpy as N
from pyfmi import load_fmu
from pyjmi.optimization import dfo
```

Then, enter code for opening the data file and extracting the measurement time series:

```
# Load measurement data from file
data = loadmat('FurutaData.mat',appendmat=False)
# Extract data series
t_meas = data['time'][:,0]
phi_meas = data['phi'][:,0]
theta_meas = data['theta'][:,0]
```

Now, plot the measurements:

```
# Plot measurements
plt.figure (1)
plt.clf()
plt.subplot(2,1,1)
plt.plot(t_meas,theta_meas,label='Measurements')
plt.title('theta [rad]')
plt.legend(loc=1)
plt.grid ()
plt.subplot(2,1,2)
plt.plot(t_meas,phi_meas,label='Measurements')
plt.title('phi [rad]')
plt.legend(loc=1)
plt.grid ()
plt.show ()
```

This code should generate Figure 6.13 showing the measurements of θ and φ.

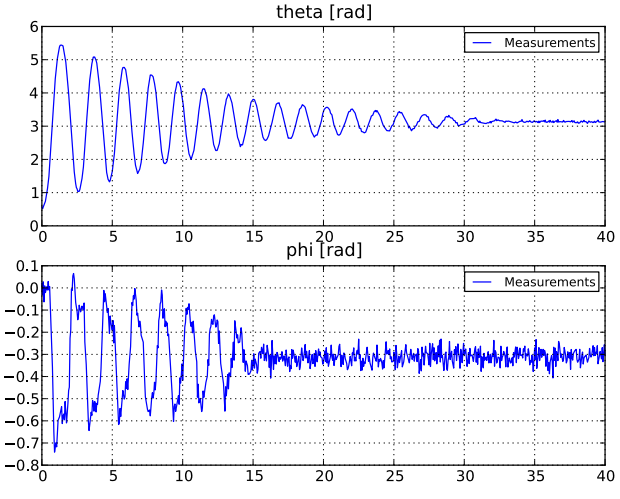


Figure 6.13 Measurements of θ and φ for the Furuta pendulum.

To investigate the accuracy of the nominal parameter values in the model, we shall now simulate the model:

```
# Load model
model = load_fmu("Furuta.fmu")
# Simulate model response with nominal parameters
res = model.simulate(start_time=0.,final_time=40)
# Load simulation result
phi_sim = res['armJoint.phi']
theta_sim = res['pendulumJoint.phi']
t_sim = res['time']
```

Then, we plot the simulation result:

```
# Plot simulation result
plt.figure (1)
plt.subplot(2,1,1)
plt.plot(t_sim,theta_sim,'--',label='Simulation nominal parameters')
plt.legend(loc=1)
plt.subplot(2,1,2)
plt.plot(t_sim,phi_sim,'--',label='Simulation nominal parameters')
plt.xlabel('t [s]')
plt.legend(loc=1)
plt.show ()
```

Figure 6.14 shows the simulation result together with the measurements.

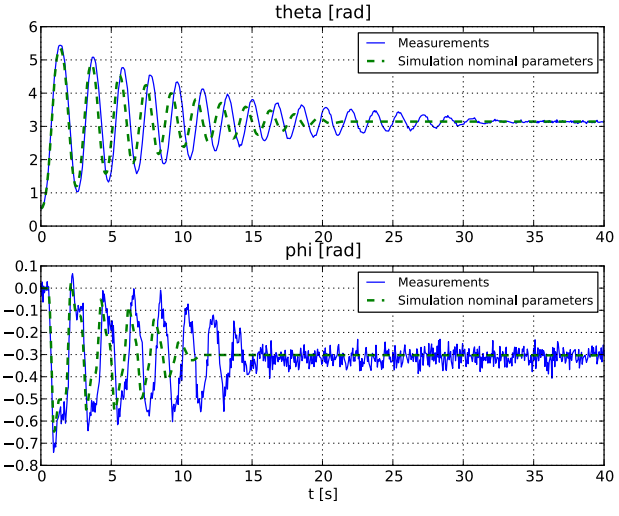


Figure 6.14 Measurements and model simulation result for φ and θ when using nominal parameter values in the Furuta pendulum model.

As can be seen, the simulation result does not quite agree with the measurements. We shall now attempt to calibrate the model by estimating the two following model parameters:

- c_{arm}: arm friction coefficient (nominal value 0.012)
- c_{pend}: pendulum friction coefficient (nominal value 0.002)

The calibration will be performed using the Nelder-Mead simplex optimization algorithm. The objective function, i.e. the function to be minimized, is defined as:

$$f(x) = \sum_{i=1}^{M} \left(\varphi^{sim}(t_i, x) - \varphi^{meas}(t_i) \right)^2 + \sum_{i=1}^{M} \left(\theta^{sim}(t_i, x) - \theta^{meas}(t_i) \right)^2$$

where t_i, i = 1,2,...,M, are the measurement time points and $\begin{bmatrix} c_{\text{arm}} & c_{\text{pend}} \end{bmatrix}^T$ is the parameter vector. φ^{meas} and θ^{meas} are the measurements of φ and θ, respectively, and φ^{sim} and θ^{sim} are the corresponding simulation results. Now, add code defining a starting point for the algorithm (use the nominal parameter values) as well as lower and upper bounds for the parameters:

```
# Choose starting point
x0 = N.array([0.012,0.002])*1e3
# Choose lower and upper bounds (optional)
lb = N.zeros (2)
ub = (x0 + 1e-2)*1e3
```

Note that the values are scaled with a factor 10^3. This is done to get a more appropriate variable size for the algorithm to work with. After the optimization is done, the obtained result is scaled back again. In this calibration problem, we shall use multiprocessing, i.e., parallel execution of multiple processes. All objective function evaluations in the optimization algorithm will be performed in separate processes in order to save memory and time. To be able to do this we need to define the objective function in a separate Python file and provide the optimization algorithm with the file name. Open a new text file, name it `furuta_cost.py` and enter the following imports:

```
from pyfmi import load_fmu
from pyjmi.optimization import dfo
from scipy.io.matlab.mio import loadmat
import numpy as N
```

Then, enter code for opening the data file and extracting the measurement time series:

```
# Load measurement data from file
data = loadmat('FurutaData.mat',appendmat=False)
# Extract data series
t_meas = data['time'][:,0]
phi_meas = data['phi'][:,0]
theta_meas = data['theta'][:,0]
```

Next, define the objective function, it is important that the objective function has the same name as the file it is defined in (except for `.py`):

```
# Define the objective function
def furuta_cost(x):
    # Scale down the inputs x since they are scaled up
```

```
    # versions of the parameters (x = 1e3*[param1,param2])
    armFrictionCoefficient = x[0]/1e3
    pendulumFrictionCoefficient = x[1]/1e3
    # Load model
    model = load_fmu('../Furuta.fmu')
    # Set new parameter values into the model
    model.set('armFriction',armFrictionCoefficient)
    model.set('pendulumFriction',pendulumFrictionCoefficient)
    # Simulate model response with new parameter values
    res = model.simulate(start_time=0.,final_time=40)
    # Load simulation result
    phi_sim = res['armJoint.phi']
    theta_sim = res['pendulumJoint.phi']
    t_sim = res['time']
    # Evaluate the objective function
    y_meas = N.vstack((phi_meas ,theta_meas))
    y_sim = N.vstack((phi_sim,theta_sim))
    obj = dfo.quad_err(t_meas,y_meas,t_sim,y_sim)
    return obj
```

This function will later be evaluated in temporary sub-directories to your working directory which is why the string `'../'` is added to the FMU name, it means that the FMU is located in the parent directory. The Python function `dfo.quad_err` evaluates the objective function. Now we can finally perform the actual calibration. Solve the optimization problem by calling the Python function `dfo.fmin` in the file named `furuta_par_est.py`:

```
# Solve the problem using the Nelder-Mead simplex algorithm
x_opt,f_opt,nbr_iters,nbr_fevals,solve_time = dfo.fmin("furuta_cost.py",
xstart=x0,lb=lb,ub=ub,alg=1,nbr_cores=4,x_tol=1e-3,f_tol=1e-2)
```

The input argument `alg` specifies which algorithm to be used, `alg=1` means that the Nelder-Mead simplex algorithm is used. The number of processor cores (`nbr_cores`) on the computer used must also be provided when multiprocessing is applied. Now print the optimal parameter values and the optimal function value:

```
# Optimal point (don't forget to scale down)
[armFrictionCoefficient_opt, pendulumFrictionCoefficient_opt] = x_opt/1e3
# Print optimal parameter values and optimal function value
print('Optimal parameter values:')
print('arm friction coeff = ' + str(armFrictionCoefficient_opt))
print('pendulum friction coeff = ' + str(pendulumFrictionCoefficient_opt))
print('Optimal function value: ' + str(f_opt))
```

This should give something like the following printout:

```
Optimal parameter values:
arm friction coeff = 0.00997223923413
pendulum friction coeff = 0.000994473020199
Optimal function value: 1.09943830585
```

Then, we set the optimized parameter values into the model and simulate it:

```
# Load model
```

```
model = load_fmu("Furuta.fmu")
# Set optimal parameter values into the model
model.set('armFriction',armFrictionCoefficient_opt)
model.set('pendulumFriction',pendulumFrictionCoefficient_opt)
# Simulate model response with optimal parameter values
res = model.simulate(start_time=0.,final_time=40)
# Load simulation result
phi_opt = res['armJoint.phi']
theta_opt = res['pendulumJoint.phi']
t_opt = res['time']
```

Finally, we plot the simulation result:

```
# Plot simulation result
plt.figure(1)
plt.subplot(2,1,1)
plt.plot(t_opt,theta_opt,'-.',linewidth=3,
label='Simulation optimal parameters')
plt.legend(loc=1)
plt.subplot(2,1,2)
plt.plot(t_opt,phi_opt,'-.',linewidth=3,
label='Simulation optimal parameters')
plt.legend(loc=1)
plt.show()
```

This should generate the Figure 6.15. As can be seen, the agreement between the measurements and the simulation result has improved considerably. The model has been successfully calibrated.

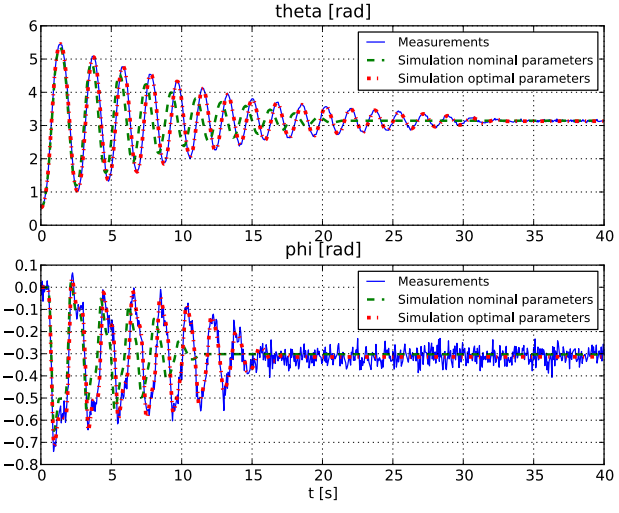


Figure 6.15 Measurements and model simulation results for φ and θ with nominal and optimal parameters in the model of the Furuta pendulum.

Chapter 7. Modelica Compiler Interface for MATLAB®

7.1. Introduction

The Modelica compiler interface for MATLAB® supports compilation of Modelica models to FMUs (Functional Mock-up Units) from the MATLAB® command line or a script. Both FMUs for Model Exchange and Co-simulation standalone, version 1.0 and 2.0, are supported. An FMU can be compiled using either the Modelica compiler provided by OCT or Dymola.

7.2. Getting started

This section introduces the Modelica compiler interface for MATLAB®. The prerequisites for using the interface were described in Section 2.2.1.1 in Chapter 2. In Section 7.2.1 some introductory examples on how to get started follows.

7.2.1. Introductory examples

The following examples introduces how to use the Modelica compiler interface using both the Modelica compiler provided by OCT and Dymola.

7.2.1.1. Modelica compiler provided by OCT

The following example shows how to compile an FMU from the Modelica Standard Library with the Modelica compiler provided by OCT. The code can be executed directly in the MATLAB® command prompt or entered in to a script (m-file).

```
% Call the complilation function with model name and chosen compiler as input arguments.
% The function output argument is the absolute path to the compiled FMU.
modelName = 'Modelica.Mechanics.Rotational.Examples.CoupledClutches'
fmuName = oct.modelica.compileFMU(modelName, 'OCT_Modelica')
```

7.2.1.2. Dymola

This short example shows how to compile an FMU from the Modelica Standard Library with Dymola. Note that before Dymola can be used, the environment variable "`DYMOLA_INSTALL_DIR`" must be set to point at the Dymola installation that should be used for the compilation. After setting `DYMOLA_INSTALL_DIR`, the toolbox initialization script must be run, see more information in Section 2.2.1.4.

```
% Set the Dymola installation to use
```

```
setenv('DYMOLA_INSTALL_DIR','C:\Program Files (x86)\Dymola 2014 FD01')
oct.initOCT()
% Call the complilation function with model name and chosen compiler as input arguments.
% The function output argument is the absolute path to the compiled FMU.
modelName = 'Modelica.Mechanics.Rotational.Examples.CoupledClutches'
fmuName = oct.modelica.compileFMU(modelName, 'Dymola')
```

7.3. Working with the Modelica compiler interface

In Section 7.3.1 the function API is described, followed by examples showing how to use the different input arguments, see Section 7.3.3.

7.3.1. Reference

Only one function is needed to compile FMUs from MATLAB® using the Modelica compiler interface for MATLAB®. This function is called `compileFMU` and resides in the package `oct.modelica`.

There are some required input arguments and additional optional input arguments to `compileFMU`. The optional input arguments are entered in pairs, such as: `'argument name'`, `'value'`. The function has one output argument which is the absolute file path to the compiled FMU.

Table 7.1, summarizes the input and output arguments, with corresponding information about type and default value (where applicable).

Table 7.1 oct.modelica.compileFMU input and output arguments

Argument name	Type	Default value	Description
Required input arguments			
`modelName`	`string`	-	The name of the model to be compiled.
`compiler`	`string`	-	The Modelica compiler to use for the compilation. Can be `'Dymola'` or `'OCT_Modelica'`.
Optional input arguments			
`modelPath`	`cell array`	`{}`	List of any Modelica libraries or model files required to compile the model. Libraries are added with absolute or relative paths to the top directory of the library.
`type`	`string`	`'me'`	FMU type. Can be `'me'` (ModelExchange) or `'cs'` (Co-simulation standalone).
`version`	`string`	`'1.0'`	FMU version. Can be `'1.0'` or `'2.0'`.
`options`	`cell array`	`{}`	Cell array of compiler options. The options are entered in pairs, where the first value specifies the option

Argument name	Type	Default value	Description
			name and the second value the option value, e.g. `{'option1', 'value1', 'option2', 'value2'}`.
`logLevel`	`string`	`'warning'`	The compiler log level. Can be `'warning'`, `'error'`, `'info'` and `'debug'`. Append the log level flag with a colon and a filename to write the log to a file, e.g. `'debug:log.txt'`. This argument will not have any effect when using Dymola.
`outputDir`	`string`	`'.'`	Output directory for the FMU. Can be relative or absolute. The directory must exist.
Output arguments			
`-`	`string`	-	Absolute path to the compiled FMU.

7.3.2. Configuration variables

There are a few important environment variables that can be used to configure the Modelica compiler interface provided by OCT. The table below lists these variables with description and default value.

Table 7.2 Configuration variables

Name	Default value	Description
`DYMOLA_INSTALL_DIR`	-	Path pointing at the Dymola installation to use for the compilation. Must be set before compiling an FMU using Dymola.
`OCT_JVM_ARGS`	'-Xmx1g'	JVM arguments for the Java process used when compiling with the Modelica compiler provided by OCT. Can be used to increase memory when compiling larger models.
`OCT_JAVA_HOME`	-	Path to Java (JRE or JDK) that will be used when compiling with the Modelica compiler provided by OCT. If not set, the Java distribution that is bundled in the OCT installation will be used.

7.3.3. Examples

In this section, a few code examples on how to use the Modelica compiler interface for MATLAB® are shown.

All examples assume that the Modelica toolbox for MATLAB® has been installed and enabled according to the instructions in Section 2.2.1.

7.3.3.1. Compile an FMU using default arguments

In this example, a model will be compiled using default arguments. The code example assumes that the Dymola installation to use has already been set using the environment variable "DYMOLA_INSTALL_DIR".

```
% Compile a model using default arguments using Dymola
% Save the path to the compiled FMU in a variable 'fmu'
fmu = oct.modelica.compileFMU('myPackage.myModel', 'Dymola')
```

If the model resides in a mo-file, the function call is

```
% Compile the model specifying the model file
% Save the path to the compiled FMU in a variable 'fmu'
fmu = oct.modelica.compileFMU('myPackage.myModel', 'Dymola', 'modelPath', {'myModels.mo'})
```

7.3.3.2. Compile an FMU setting input arguments

In this example, a model is compiled by specifying a model file and setting FMU type.

```
% Compile the model specifying the model file and setting FMU type to 'cs'
fmu = oct.modelica.compileFMU('myPackage.myModel', 'OCT_Modelica', 'modelPath',
 {'myModels.mo'}, 'type', 'cs')
```

If many model files or libraries are required to compile a model it might be easier to save the input argument as a variable. This is demonstrated in the following example:

```
% Save array of libraries in a variable
myLibs = {'../lib1', '../lib2'}
% Compile the model specifying the model file and setting FMU type to 'cs'
fmu = oct.modelica.compileFMU('myPackage.myModel', 'OCT_Modelica', 'modelPath', myLibs,
                              'type', 'cs')
```

The next example shows how to set compiler options. Options are entered as key-value pairs in a cell array:

```
% Set a couple of options, entered as key-value pairs
opts = {'string-option', 'stringvalue', 'boolean-option', true}
% Compile the model specifying the model file and compiler options
fmu = oct.modelica.compileFMU('myPackage.myModel', 'OCT_Modelica', 'modelPath',
{'myModels.mo'}, 'options', opts)
```

7.3.3.3. Compile an FMU specifying log level

The following examples will show how to change the compiler log level and how to output the log to file:

```
% Compile a model from MSL, changing compiler log level to 'debug'
modelName = 'Modelica.Mechanics.Rotational.Examples.CoupledClutches'
fmu = oct.modelica.compileFMU(modelName, 'OCT_Modelica', 'logLevel', 'debug')
```

The log level 'debug' produces a lot of output and it is then convenient to save the output in a file. Log entries corresponding to 'warning' will still be printed to the command prompt.

```
% Compile a model from MSL, writing debug log output to file
modelName = 'Modelica.Mechanics.Rotational.Examples.CoupledClutches'
fmu = oct.modelica.compileFMU(modelName, 'OCT_Modelica', 'logLevel', 'debug:log.txt')
```

Chapter 8. Simulation of FMUs in MATLAB®

In this chapter a small example will be shown on how to compile and then simulate a Modelica model in MATLAB®. The simulation relies on FMI Toolbox for MATLAB®. For further documentation please refer to the FMI Toolbox User's Guide.

8.1. Compile an FMU and simulate in the FMI Toolbox

This example demonstrates how to compile a Modelica model to an FMU Model Exchange 1.0 using the Modelica compiler provided by OCT and how to simulate the FMU using FMI Toolbox for MATLAB®.

```
% Set required input arguments
modelname = 'Modelica.Mechanics.Rotational.Examples.First'
compiler = 'OCT_Modelica'
% Compile model with default input arguments
fmuName = oct.modelica.compileFMU(modelname, compiler)
% Load the FMU using FMI toolbox
fmu_me = FMUModelME1(fmuName)
% Specify result variables
outdata.name = {'inertia1.w','inertia2.w','inertia3.w'}
% Instantiate, initialize and simulate for 3s
fmu_me.fmiInstantiateModel()
fmu_me.fmiInitialize()
[tout, yout, yname] = fmu_me.simulate([0,3],'Output',outdata)
% Plot results
plot(tout,yout)
legend(yname)
```

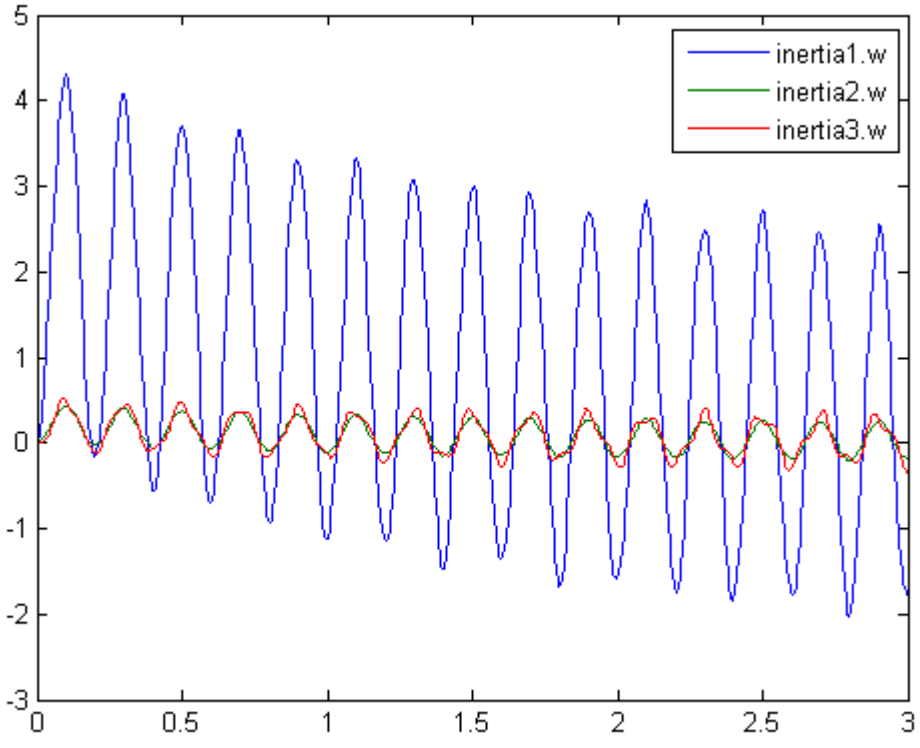


Figure 8.1 Simulation result Modelica.Mechanics.Rotational.Examples.First

Chapter 9. Steady-state Solver Interface for MATLAB® and Python

9.1. Introduction

The non-linear equation solver supports solution of non-linear equation systems with discontinuities. The solver is based on the KINSOL solver that is a part of SUNDIALS, see [Kinsol2011]. The primary application is solution of steady-state model equations packaged into FMUs (Functional Mockup Units), see [FMI2017]. The solver utilizes interfaces provided by the FMI Toolbox for MATLAB®/Simulink (FMIT) and PyFMI (Python) respectively to interact with FMUs.

The solver is intended to be applied for non-linear equation systems on the form:

$$f(x,sw)=0 \tag{9.1}$$

$$g(x,sw)... \tag{9.2}$$

Here x is a column vector of unknown length, sw are the current state of state of switches and f(x,sw) is a column vector of residuals. The number of residuals is equal to the number of unknowns. The equations are solved using an iterative equation solver starting with an initial guess for the iteration variables and accuracy is controlled both by the solver options and nominal values for the iteration variables.

The equations may contain discontinuities that are identified with discontinuity indicator functions g(x, sw), which are real valued functions. Discontinuities occur when an indicator function changes sign. The residual functions are expected to be at least C^1 continuous outside the discontinuities represented with the indicator functions. This implies that there are two separate continuous representations of f(x, sw) around each discontinuity. Explicit call-back function is utilized by the solver to switch between the two continuous representations around a discontinuity.

9.2. Working with the Steady-State Solver MATLAB® and Python Interfaces

Working with the steady-state interface involves setting up problems, e.g. by compiling Modelica models into FMUs, and feeding these to the solver. When the solver solves it actively reports information, which makes it easy to quickly monitor and understand what happens. Once the solver is finished it has created a log file with extensive information about the solution progress, much more detailed than what was provided during the solver process. This information can be used to understand in detail what happened, which is for example useful for debugging non-convergence and finding enhancements. To this end the steady-state solver in MATLAB® and Python ships with a logging functionality that interprets the generated log files and provides an information retrieval API.

To use the steady-state interface `Problem` and `Solver` instances must be created, where the `Problem` is given as an argument to the `Solver` constructor. A typical use case in MATLAB® will include an `FMUProblem` instance that is created by supplying an `FMUModelME1` or `CoupledFMUModelME1` model instance, as provided by FMIT and OCT, as an argument to the `FMUProblem` constructor. Corresponding in Python is to provide the `FMUProblem` constructor with an `FMUModelME2` from PyFMI. `FMUProblem` is a subclass of the `Problem` class that is included in the package, and it uses FMUs that follow the input and output variable conventions as generated when using the Modelica compiler provided by OCT. It is also possible to create custom subclasses of the `Problem` class and provide them to the `Solver`. The log information retrieval API is provided through the `LogViewer` class.

Below a short description of the API is given, followed by some examples: a simple use case, a complete use case including specification of a Modelica model and compilation to an FMU using the Modelica compiler provided by OCT, an example showing how to use the information retrieval API, an example showing how to create a custom subclass to `Problem` and finally an example demonstrating the concept of interactive FMUs.

For Python, the examples below are equivalent to whether a 32-bit or 64-bit version is used. For MATLAB® users this holds only for a 64-bit version. Moreover, the bitness of the solvers is tied to the specific bitness of Python or MATLAB® that is used. Note that FMUs are platform specific, and need to be compiled for the platform used.

9.2.1. Important Interface Features

Described below are some central user aspects of the `Problem`, `Solver` and `LogViewer` classes. The mechanics of log file creation is also described. The classes are located inside the MATLAB® package `oct.nlesol` and Python package `oct.steadystate.nlesol` respectively.

For detailed documentation about classes and methods use the interactive documentation. In MATLAB® this can be retrieved through:

- `help CLASS_NAME` For class level help.
- `help CLASS_NAME.METHOD_NAME` For detailed documentation about methods.
- `doc CLASS_NAME` Explore documentation using MATLAB® documentation browser.

In Python this can be retrieved through:

- `help(object)` can be utilized to access docstrings. For example by typing in a Python-shell: `help(Solver)`.
- `object?`, e.g. `Solver?`, can also be utilized to access docstrings if using IPython or PyLab.

Note that the solver uses Math Kernel Library (MKL) and that it can slightly affect the results of a solve. For further information, see Section C.5.

9.2.1.1. Problem and FMUProblem

The `Problem` class in itself can not be used with the solver as it provides no way to populate it with problem information. The intended use is via subclasses, either through a custom implementation or by using the provided

class `FMUProblem`, which uses interactive FMUs. An interactive FMU is designed to expose the system to an external solver, and this is achieved by transforming the iteration variables to inputs and the residual variables to outputs on the FMU, see example in Section 9.2.2.1 below for further details.

`FMUProblem` takes an FMU as an argument to the constructor, and it may also take names of iteration and residual variables. `FMUProblem` provides functionality to hold iteration variables and residuals, see examples in the section called “Hold Iteration Variables (Python)” and the section called “Hold Iteration Variables (MATLAB)”. `FMUProblem` also provides functionality for parametric holding, see examples in the section called “Parametric Hold Iteration Variables (Python)” and the section called “Parametric Hold Iteration Variables (MATLAB)” and see Section 14.1 for a specification over how parametric holding is specified in models.

`FMUProblem` supports handling of initial guess based on iteration variable start attribute that is symmetric to the handling done in FMUs with integrated solver. Specifically, if start attribute is defined via a parametric expression the expression is evaluated and used during FMU initialization. See example in the section called “Parametric Start Attributes (MATLAB)” for more details.

A convenient way to view information about the `FMUProblem` is through the printInfo and print_info methods for MATLAB® and Python respectively.

9.2.1.2. Solver

`Solver` takes a `Problem` instance as an argument to the constructor. This instance represents the steady-state problem to be solved. The most important methods are the solve method, which invokes the solver, and the setOptions method in MATLAB® and the corresponding functionality with solve_options() and the options argument to the solve method in Python. Possible options to the solver are given in the table below.

Table 9.1 Solver Options

Option	Default	Description
active_bounds_mode	0	0 - project Newton step, 1 - use steepest descent in case of non-descent direction. (Corresponding compiler option: nle_active_bounds_mode)
check_jac_cond	true	Calculate jacobian condition number and write it out to the log. (Corresponding compiler option: nle_solver_check_jac_cond)
Brent_ignore_error	false	Ignore Brent solve error for debugging purposes. (Corresponding compiler option: nle_brent_ignore_error)
discontinuities_tolerance (MATLAB only)	1e-10	Tolerance used to decide if discontinuity was crossed [eps(1), 0.1] (Corresponding compiler option: None).
enforce_bounds	true	Enforce bounds on iteration variables flag (true/false) (Corresponding compiler option: enforce_bounds)

iteration_variable_scaling	2	Iteration variable scaling mode:0 - no scaling, 1 - heuristic scaling, 2 - nominal based scaling. (Corresponding compiler option: iteration_variable_scaling)
jacobian_calculation_mode	0	Mode for how to calculate the Jacobian: 0 - onesided differences, 1 - central differences, 2 - central differences at bound, 3 - central differences at bound and 0, 4 - central differences in second Newton solve, 5- central differences at bound in second Newton solve, 6 - central differences at bound and 0 in second Newton solve, 7 - central differences when small residual, 8 - calculate Jacobian through MATLAB®, 9 - Jacobian compression. (Corresponding compiler option: nle_jacobian_calculation_mode)
jacobian_check	false	Compare the Jacobian calculated through MATLAB® with the finite differences Jacobian. (Corresponding compiler option: None)
jacobian_check_tolerance	1e-6	Maximal allowed relative error between the Jacobians compared through option 'check_jacobian' [eps(1), 1] (Corresponding compiler option: None)
jacobian_finite_difference_delta	sqrt(eps)	Delta to use when calculating finite difference Jacobians [eps(1), 0.1] (Corresponding compiler option: nle_jacobian_finite_difference_delta)
jacobian_update_mode	2	Mode for how to update the Jacobian: 0 - full Jacobian, 1 - Broyden update, 2 - reuse Jacobian (Corresponding compiler option: nle_jacobian_update_mode)
log_level	4	Log level for the solver [0, 8] (Corresponding compiler option: log_level)
max_iter	100	Maximum number of iterations in the non-linear solver [1, 1000] (Corresponding compiler option: nle_solver_max_iter)
max_iter_no_jacobian	10	Maximum number of iterations without jacobian update. Value 1 means an update in every iteration. [1, 1000] (Corresponding compiler option: nle_solver_max_iter_no_jacobian)
max_residual_scaling_factor	1e10	Maximum allowed scaling factor for residuals [1, 1e32] (Corresponding compiler option: nle_solver_max_residual_scaling_factor)

min_residual_scaling_factor	1e-10	Minimal allowed scaling factor for residuals [1e-32, 1] (Corresponding compiler option: nle_solver_min_residual_scaling_factor)
regularization_tolerance	1e-10	Tolerance for deciding when jacobian regularization should kick in (i.e. when condition number is larger than 1/regularization_tolerance [eps(1), 0.1] (Corresponding compiler option: nle_solver_regularization_tolerance)
rescale_after_singular_jac	true	Update scaling factors after passing a singular point. (Corresponding compiler option: rescale_after_singular_jac)
rescale_each_solve	false	Update scaling factors at the beginning of each solve() call. (Corresponding compiler option: rescale_each_step)
residual_equation_scaling	1	Residual equation scaling mode: 0 - no scaling, 1 - automatic scaling, 2 - manual scaling, 3 - hybrid scaling, 4 - aggressive automatic scaling, 5 - automatic rescaling at full Jacobian update. (Corresponding compiler option: residual_equation_scaling)
solver_exit_criterion	3	Exit criterion mode for the solver: Valid values: 0 - step length and residual based, 1 - only step length based, 2 - only residual based, 3 - hybrid. (Corresponding compiler option: nle_solver_exit_criterion)
tolerance	1e-6	Relative tolerance [eps(1), 0.1]. (Corresponding compiler option: nle_solver_default_tol)
silent_mode	false	No solve information written to the screen, but still to log (true/false) (Corresponding compiler option: None).
step_limit_factor	0.2	Factor used to limit the step size based on nominal and min/max range [0.01, 100]. Newton step length is limited so that for any iteration variable `xi` it is not larger than `step_limit_factor` times `min(max(abs(nominal), abs(xi)),(xi_max-xi_min))`. (Corresponding compiler option: nle_solver_step_limit_factor)
use_Brent_in_1d	false	Use Brent as a backup method for solving 1D equations. (Corresponding compiler option: use_Brent_in_1d)
use_jacobian_equilibration	false	Use Jacobian equilibration to improve linear solver accuracy (true/false). (Corresponding compiler option: use_jacobian_equilibration)

As the solver progresses it will print out status messages to the MATLAB® terminal/Python console. These are provided to give a quick overview of the process.

9.2.1.3. LogViewer

`LogViewer` works with log files, both from logs generated by the `Solver` and logs generated by FMIT and PyFMI, during e.g. initialization. In the case of logs generated by the `Solver` they are stored as XML files. Log files generated by FMIT and PyFMI are stored as .txt files that contain XML data. The log files contain a lot of information, and the amount of information saved in them can be regulated by setting the log level on the `Solver` and on the FMUs, see section Section 9.2.1.4 for further details. `LogViewer` provides an API for retrieving informations from the log. Example of methods it provides are in the MATLAB® case: getIterationVariables, getResiduals, getJacobian, getErrors, getStep and getWarnings. Corresponding Python methods are: get_iteration_variables, get_residuals, get_jacobian, get_errors, get_step and get_warnings.

A `LogViewer` can be created either by providing a `Solver` object or the name of a log-file as an argument to the constructor.

9.2.1.4. How to Control Information Saved in Log Files

The OCT steady-state packages contain a powerful logging framework. The framework is particularly useful when applied to FMUs compiled with OCT Modelica compiler. This section will explain how to determine what data should be logged. It may be necessary to determine what data is logged for large models, since logging can use a lot of resources.

When using oct.nlesol with FMIT in MATLAB® there are three ways to change what log data will be saved:

- Set an option on the oct.nlesol.Solver object.
- Give an argument to the constructor of the FMIT FMU object.
- Set a parameter on the FMU.

When using oct.steadystate.nlesol in Python together with PyFMI there are three ways to change what log data will be saved:

- Provide an option on the oct.steadystate.nlesol.Solver object.
- Give the argument `log_level` to PyFMI's `load_fmu`
- Set a parameter on the FMU.

These values will change different aspects of what is logged, and they may also affect each other. Below an overview of these ways and their connections is presented. Note that the logging framework can also be used with logs produced by FMUs with an integrated solver, in that case the log level can only be changed directly on the FMU as no Solver object is used.

Events that trigger logging mostly include the solution of equations blocks, but also other information such as ModelicaMessage/ModelicaError in external C-libs, IO messages about external files such as XML files used by the FMU and asserts in Modelica code. There are two places where equation blocks may be solved:

- In the segregated MATLAB®/Python solver.
- In code that is built into the FMU.

In the case of FMUs with an integrated solver, blocks are only solved inside the FMU. For FMUs used together with the segregated MATLAB®/Python solver, blocks are typically solved only in the segregated solver. When solving FMUs with local iterations, the blocks for local iterations are solved inside the FMU even if the segregated solver is used. The blocks for local iterations are typically solved using a Brent solver.

In MATLAB® the log level can be set by:

- `Solver.setOptions('log_level')`: For the segregated solver this is the main log level and will determine what data is emitted about the block solved by the segregated solver. This will not affect what data is emitted by the FMU.
- As a parameter on the FMU: `fmu.setValue('_log_level', nnn)`, where nnn is an integer. The log level can only be changed before a call to fmiInitialize. Since fmiInitalize is called on the FMU as part of the FMUProblem constructor it can only be set before the FMUProblem is created. The default log level for an FMU is set with a compiler option 'log_level'.
- In the constructor to an FMIT FMU object, e.g. `fmu = FMUModelME1(fmuName, 'log_level')` . Where log_level is: 'all', 'verbose', 'info', 'warning', 'error', 'fatal', 'nothing'. See FMIT docs for further details.

In Python the log level can be set by:

- Changing the solver option *log_level* by first retrieving a `SolverOptions` object through

  ```
  opts = solver.solve_options()
  ```

  Then the log level is set through:

  ```
  opts['log_level'] = val
  ```

  and `opts` is passed to the `solve(options=opts)` method. It is also possible to set the log level without retrieving a `SolverOptions` object using `solve(options = {'log_level': val})` where `val` is a suitable integer.

  This affects what data is emitted about the block solved by the segregated solver and does not affect what data is emitted by the FMU.

- As a parameter on the FMU: `fmu.set("_log_level", nnn)`, where nnn is an integer. The log level can only be changed before a call to initialize. Since initialize is called on the FMU as part of the FMUProblem constructor it can only be set before the FMUProblem is created. The default log level for an FMU is set with a compiler option 'log_level'.
- Invoking the command `load_fmu(fmu_name, log_level=XX)` and setting `XX` to any integer in the range zero to seven.

  With PyFMI it is also possible to set the log level of a model that has already been loaded, this is done by using the `set_log_level(param)` method. E.g. `model.set_log_level(7)`.

As mentioned above, the log level set on the solver object does not affect the log information emitted by the FMU. However, in MATLAB® the log level set as an argument to the constructor of the FMIT FMU object affects what happens when the log level is set as a parameter on the FMU. The constructor log level determines what information can be logged by the FMU. For example, if it is set to 'warning' then only log messages that are considered warnings or more severe can be logged by the FMU. Thus setting a very high log level as a parameter on the FMU would not give a log with much information if FMIT log level is not set at the same time. Examples:

```
fmu = loadFMU(fmuName, 'error').
fmu.setValue('_log_level', 8) % No effect of high log level since only errors are passed
 through by FMIT
fmu.setValue('_log_level', 0) % No information emitted by the FMU

fmu = loadFMU(fmuName, 'all')
fmu.setValue('_log_level', 8) % All possible information emitted by the FMU
fmu.setValue('_log_level', 0) % No information emitted by the FMU
```

In PyFMI, the log_level that is set either as an argument to the load_fmu method or through fmu.set_log_level() determines what information will be passed through. That means, like in the MATLAB® case above that:

```
fmu = load_fmu(fmu_name, log_level=2)
fmu.set('_log_level', 8) # No effect of high log level since only errors are passed through
 by PyFMI
fmu.set('_log_level', 0) # No information emitted by the FMU

fmu = load_fmu(fmuName, log_level=7)
fmu.set('_log_level', 8) % All possible information emitted by the FMU
fmu.set('_log_level', 0) % No information emitted by the FMU
```

To get information about the blocks solved in the FMU the log level set in the constructor must be 'info'/4 or above. Messages that are considered 'info'/4 include all types of information emitted in the solution of blocks. So if an FMU object is created with log level 'info'/4 then the log level set as a parameter on the FMU determine exactly what messages on the level of info that are emitted. For example, to get full information from the blocks solved by Brent requires log level 8. But if blocks are solved using a Newton solver only log level 6 is needed.

9.2.1.5. What Log Files are Created

There are two different log files written for a ModelA.fmu compiled from ModelA in Modelica:

- ModelA_log.txt – produced by FMIT/PyFMI. The detail level is controlled by setting the log level on the FMU in one of the two ways described above. The log can be used by LogViewer that retrieves and parses the XML information in the file. This is used when the system is solved with an integrated solver.
- ModelA_log.xml (MATLAB)/ModelA_solver_log.xml (Python) – produced by LogMonitor in MATLAB®/Python segregated solver. Detail level is controlled by log_level option of the solver, and by the log level set on the FMU as detailed above. The log can be used by LogViewer, which parses the XML information in the file.

9.2.1.6. Control Log Information with Log Level

A summary of the OCT log levels and what information they are associated with is given below:

- **0**: Log nothing
- **1**: Errors
- **2**: Warnings
- **3**: Base-line information
- **4**: Detailed information including basic Newton solver traces
- **5**: Detailed information including Newton solver traces
- **6**: Detailed information including Jacobians
- **7**: Detailed information including basic information on Brent
- **8**: Detailed information including Brent traces

9.2.2. Examples

In this section we show examples similar for both Python and MATLAB®, starting with Python and continuing with MATLAB®. These examples cover basic functionality such as solving the problem, creating custom problems, working with the LogViewer, and parametric holding. The aim of this section is to familiarize the user with the procedure of using FMUs for steady-state solving and accessing the results by means of basic examples.

First we will, however, go through some concepts which are shared between Python and MATLAB®; how to create an interactive FMU and how to interpret the solver trace.

9.2.2.1. Interactive FMU

This example shows how the compiler creates an FMU from a steady-state Modelica model. The model:

```
model twoEqSteadyState
```

```
    Real x (start = 1);
    Real T (start = 400);
equation
    0 = 120*x - 75*(0.12*exp(12581*(T - 298)/(298*T)))*(1 - x);
    0 = -x*(873 - T) + 11.0*(T - 300);
end twoEqSteadyState;
```

Compiling this model with the interactive fmu option set to `true` will produce the following flattened model:

```
fclass twoEqSteadyState
 input Real x(start = 1);
 input Real T(start = 400);
 parameter Real iter_0 "T";
 output Real res_0(nominal = 0) "0 = (- x) * (873 - T) + 11.0 * (T - 300)";
 parameter Real iter_1 "x";
 output Real res_1(nominal = 0) "0 = 120 * x - 75 * (0.12 * exp(12581 * (T - 298) / (298 *
 T))) * (1 - x)";
parameter equation
 iter_0 = 400;
 iter_1 = 1;
equation
 res_1 = 75 * (0.12 * exp(12581 * (T - 298) / (298 * T))) * (1 - x) - 120 * x;
 res_0 = - (- x) * (873 - T) - 11.0 * (T - 300);
end twoEqSteadyState;
```

Where we see that the iteration variables are now inputs and there are residual variables as outputs.

9.2.2.2. Interpreting the Solver Trace

A typical solver trace generated (in this case from the MATLAB® example in the section called “Complete Example (MATLAB)”) by the steady-state solver looks like:

```
Model name.....................: ExampleModels.SimpleSteadyState
Number of iteration variables......: 1
Number of discontinuity switches...: 2

Switch iteration 1
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   1Js   1.0000e+00  1.0000e+00:   1     0    0   2.0000e-01:   1r  2.0000e-01
   2     8.0000e-01  8.0000e-01:   1     0    0   2.0000e-01:   1r  2.0000e-01
   3     6.4000e-01  6.4000e-01:   1     0    0   2.0000e-01:   1r  2.0000e-01
   4     5.1200e-01  5.1200e-01:   1     0    0   2.0000e-01:   1r  2.0000e-01
   5     4.0960e-01  4.0960e-01:   1     0    0   2.4414e-01:   1r  2.4414e-01
   6     3.0960e-01  3.0960e-01:   1     0    0   3.2300e-01:   1r  3.2300e-01
   7     2.0960e-01  2.0960e-01:   1     0    0   4.7710e-01:   1r  4.7710e-01
   8     1.0960e-01  1.0960e-01:   1     0    0   9.1241e-01:   1r  9.1241e-01
   9     9.6000e-03  9.6000e-03:   1     0    0   1.0000e+00        1.0000e+00
  10     0.0000e+00  0.0000e+00:   1
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   1s    0.0000e+00  0.0000e+00:   1
Switch iteration 2
```

```
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   1     6.6667e-01  6.6667e-01:   1     0    0   3.0000e-01:   1r  3.0000e-01
   2     6.0000e-01  6.0000e-01:   1     0    0   3.3333e-01:   1r  3.3333e-01
   3     5.3333e-01  5.3333e-01:   1     0    0   3.7500e-01:   1r  3.7500e-01
   4     4.6667e-01  4.6667e-01:   1     0    0   4.2857e-01:   1r  4.2857e-01
   5     4.0000e-01  4.0000e-01:   1     0    0   5.0000e-01:   1r  5.0000e-01
   6     3.3333e-01  3.3333e-01:   1     0    0   6.0000e-01:   1r  6.0000e-01
   7     2.6667e-01  2.6667e-01:   1     0    0   9.0000e-01:   1r  9.0000e-01
   8     1.8667e-01  1.8667e-01:   1     0    0   1.0000e+00         1.0000e+00
   9     1.2444e-01  1.2444e-01:   1     0    0   1.0000e+00         1.0000e+00
  10J    8.2963e-02  8.2963e-02:   1     0    0   1.0000e+00         1.0000e+00
  11J    2.8553e-10 -2.8553e-10:   1     0    0   1.0000e+00         1.0000e+00
  12     0.0000e+00  0.0000e+00:   1
Switch iteration 3
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   1     6.6667e-01 -6.6667e-01:   1     0    0   2.0000e-01:   1r  2.0000e-01
   2     2.6667e-01 -2.6667e-01:   1     0    0   4.0000e-01:   1r  4.0000e-01
   3     5.3333e-02  5.3333e-02:   1     0    0   1.0000e+00         6.0000e-01
   4     4.2667e-02 -4.2667e-02:   1     0    0   1.0000e+00         6.0000e-01
   5     3.4133e-02  3.4133e-02:   1     0    0   1.0000e+00         6.0000e-01
   6     2.7307e-02 -2.7307e-02:   1     0    0   1.0000e+00         6.0000e-01
   7     2.1845e-02  2.1845e-02:   1     0    0   1.0000e+00         6.0000e-01
   8     1.7476e-02 -1.7476e-02:   1     0    0   1.0000e+00         6.0000e-01
   9     1.3981e-02  1.3981e-02:   1     0    0   1.0000e+00         6.0000e-01
  10J    1.1185e-02 -1.1185e-02:   1     0    0   1.0000e+00         1.0000e+00
  11J    7.5318e-11  7.5318e-11:   1     0    0   1.0000e+00         1.0000e+00
  12     7.4015e-17  7.4015e-17:   1
Number of function evaluations: 60
Number of jacobian evaluations: 5
Solver finished
Total time in solver: 0.35 s
```

The solver outputs brief progress messages to allow to follow the solution progress in real time. Solution is performed using a sequence of Newton iterations; each produces a line of status values. Newton iterations are grouped into runs that start from iteration number `iter=1` (which the gives status before the first iteration). Each run of Newton iterations belongs to a switch iteration, printed as a header.

`iter` gives the number of the current Newton iteration in the current run. It can be followed by one or more letters:

- `J`: The Jacobian was updated at the beginning of the iteration.
- `s`: The residual scaling was updated at the beginning of the iteration.
- `r`: Regularization is used during the iteration due to a singular Jacobian.
- `x`: Iteration was retried due to line search failure and Jacobian not up-to-date.
- `m`: A minimum norm approach was used to calculate the step.
- `d`: Steepest descent direction used.

`res_norm` is the 2-norm of the current residual according to the current residual scaling. `max_res` is the value of the scaled residual that currently has the greatest magnitude, and the corresponding `ind` is its index.

At the start of each Newton iteration, the solver computes a Newton step based on the current residuals and Jacobian. The search direction is updated to move along active variable bounds and the step length is limited based on the two criteria:

- Limiting bounds: full step violates min/max bounds on some iteration variables.
- Range limit: full step length in a variable is too long as compared to the variable specific range limit based on nominal value and max-min range (see `step_limit_factor` solver option).

`nlb` column gives the current number of limiting bounds including active ones, and `nab` gives the number of active bounds. Line search is performed with the (projected) Newton step as search direction; the actual step taken is `lambda` times the (projected) Newton step.

The `lambda_max` value gives the upper bound on `lambda` from the step limiting criteria. The index of the most limiting variable in the block is given by the corresponding `ind`, if there is one. The `r` suffix indicates that the range limit condition reduced the step the most. The step length may be further reduced as a part of line-search if `lambda_max` does not give sufficient residual decrease.

The step lengths and number of limiting/active bounds reported on an iteration are for the step that is taken at the iteration. This is why they are not printed at the last iteration.

9.2.2.3. Python Examples

Introductory Example (Python)

The simple introductory example below shows how an `FMUProblem` instance is created, where PyFMI is used to load an FMU model. A solver instance is created from the `FMUProblem` instance, and the steady-state solver is invoked.

```
from pyfmi import load_fmu
from oct.steadystate.nlesol import FMUProblem, Solver
fmu_model = load_fmu('SteadyStateProblem.fmu')
fmu_problem = FMUProblem(fmu_model)
solver = Solver(fmu_problem)
sol = solver.solve()
```

Complete Example (Python)

This model is included in ExampleModels.mo-package, and to run this example begin by importing the example by

```
from oct.steadystate.examples import simple_steadystate
```

and then run

```
simple_steadystate.run_demo()
```

First, create a steady-state Modelica model and save it to a file SimpleSteadyState.mo.

```
model SimpleSteadyState
  parameter Real  a = 2;
  input Real  b(start = 2);
  Real x (start = -2);
 equation
  b = if (x < -1.0)
    then x + (x + 1)*a
   else
    if (x > 1)
     then x + (x - 1)*a
    else
     x;
end SimpleSteadyState;
```

Next we compile the FMU with the required interactive argument and imports.

```
import os
import numpy as np
from pymodelica import compile_fmu
from pyfmi import load_fmu
from oct.steadystate.nlesol import FMUProblem, Solver
compiler_opts = {"interactive_fmu": True,
     "expose_scalar_equation_blocks_in_interactive_fmu": True}
name = compile_fmu("SimpleSteadyState",
     "SimpleSteadyState.mo",
     compiler_options=compiler_opts)
```

Next we use PyFMI to load the FMU and pass the FMUModelME2 instance to FMUProblem.

```
model = load_fmu(name, log_level=4)
problem = FMUProblem(model)
```

Note that it is also possible to supply the names of iteration and residual variables when the FMUProblem is created, for the model above the following would have worked as well:

```
problem = FMUProblem(model, iteration_variables=["x"], residual_variables=["res_0"])
```

This feature is useful when not all iteration and residual variables in an interactive FMU are used.

Next we create a Solver instance and invoke the steady-state solver.

```
solver = Solver(problem)
res = solver.solve()
```

When calling for the `solve()` command, it produces the following output (described in more detail in Section 9.2.2.2).

```
Model name.............................: ExampleModels.SimpleSteadyState
Number of iteration variables..........: 1
Number of discontinuity switches.......: 2

Event iteration 1
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   0Js  1.0000e+000  1.0000e+000:   0     0    0  2.0000e-001:   0r 2.0000e-001
   1    8.0000e-001  8.0000e-001:   0     0    0  2.0000e-001:   0r 2.0000e-001
   2    6.4000e-001  6.4000e-001:   0     0    0  2.0000e-001:   0r 2.0000e-001
   3    5.1200e-001  5.1200e-001:   0     0    0  2.0000e-001:   0r 2.0000e-001
   4    4.0960e-001  4.0960e-001:   0     0    0  2.4414e-001:   0r 2.4414e-001
   5    3.0960e-001  3.0960e-001:   0     0    0  3.2300e-001:   0r 3.2300e-001
   6    2.0960e-001  2.0960e-001:   0     0    0  4.7710e-001:   0r 4.7710e-001
   7    1.0960e-001  1.0960e-001:   0     0    0  9.1241e-001:   0r 9.1241e-001
   8    9.6000e-003  9.6000e-003:   0     0    0  1.0000e+000       1.0000e+000
   9    0.0000e+000  0.0000e+000:   0
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   0s   0.0000e+000  0.0000e+000:   0
Event iteration 2
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   0    6.6667e-001  6.6667e-001:   0     0    0  3.0000e-001:   0r 3.0000e-001
   1    6.0000e-001  6.0000e-001:   0     0    0  3.3333e-001:   0r 3.3333e-001
   2    5.3333e-001  5.3333e-001:   0     0    0  3.7500e-001:   0r 3.7500e-001
   3    4.6667e-001  4.6667e-001:   0     0    0  4.2857e-001:   0r 4.2857e-001
   4    4.0000e-001  4.0000e-001:   0     0    0  5.0000e-001:   0r 5.0000e-001
   5    3.3333e-001  3.3333e-001:   0     0    0  6.0000e-001:   0r 6.0000e-001
   6    2.6667e-001  2.6667e-001:   0     0    0  9.0000e-001:   0r 9.0000e-001
   7    1.8667e-001  1.8667e-001:   0     0    0  1.0000e+000       1.0000e+000
   8    1.2444e-001  1.2444e-001:   0     0    0  1.0000e+000       1.0000e+000
   9J   8.2963e-002  8.2963e-002:   0     0    0  1.0000e+000       1.0000e+000
  10    2.8553e-010 -2.8553e-010:   0     0    0  1.0000e+000       1.0000e+000
  11    0.0000e+000  0.0000e+000:   0
Event iteration 3
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   0    6.6667e-001 -6.6667e-001:   0     0    0  2.0000e-001:   0r 2.0000e-001
   1    2.6667e-001 -2.6667e-001:   0     0    0  4.0000e-001:   0r 4.0000e-001
   2    5.3333e-002  5.3333e-002:   0     0    0  1.0000e+000       6.0000e-001
   3    4.2667e-002 -4.2667e-002:   0     0    0  1.0000e+000       6.0000e-001
   4    3.4133e-002  3.4133e-002:   0     0    0  1.0000e+000       6.0000e-001
   5    2.7307e-002 -2.7307e-002:   0     0    0  1.0000e+000       6.0000e-001
```

```
     6    2.1845e-002  2.1845e-002:   0     0    0  1.0000e+000       6.0000e-001
     7    1.7476e-002 -1.7476e-002:   0     0    0  1.0000e+000       6.0000e-001
     8    1.3981e-002  1.3981e-002:   0     0    0  1.0000e+000       6.0000e-001
     9J   1.1185e-002 -1.1185e-002:   0     0    0  1.0000e+000       1.0000e+000
    10    7.7968e-011  7.7968e-011:   0     0    0  1.0000e+000       1.0000e+000
    11    7.4015e-017  7.4015e-017:   0
  Number of function evaluations.....: 56
  Number of jacobian evaluations.....: 3
  Solver finished
  Total time in solver.....: 0.22 s
```

Note that indexing of iteration variables and residuals starts from 0 here in Python instead of 1 as compared to MATLAB®.

Log Information Retrieval (Python)

Below an example is presented where a steady-state problem is solved and the generated log-file provided to the `LogViewer`. This model is included in the ExampleModels.mo-package and the corresponding script to run it is in *log_viewer.py*.

```
compiler_opts = {"interactive_fmu": True,
                 "expose_scalar_equation_blocks_in_interactive_fmu": True}
name = compile_fmu("ExampleModels.twoEqSteadyState",
                   os.path.join(curr_dir, "files", "example_models.mo"),
                   compiler_options=compiler_opts)

# Use pyfmi to create an fmu model object.
model = load_fmu(name, log_level=4)
model.set("_log_level", 4)

# Create the problem class instance from the fmu and supply it to the
# solver.
problem = FMUProblem(model)
solver = Solver(problem)

# Solve the problem with a log level sufficiently high to get
# values of iteration variables emitted to the log file.
solver.solve({"log_level": 6})

# Create the LogViewer from the solver class instance we
# created above as such
log_viewer = LogViewer(solver)

# LogViewer can also be supplied with the log-file, e.g.
# log_viewer = LogViewer("twoEqSteadyState_log.xml")

# It is now possible to play around and retrieve information from the log.
# E.g. retrieve the values of the iteration variables at different
# times during the solver invocation.
```

```
iter_data_second_newton = log_viewer.get_iteration_variables()
iv_names = log_viewer.get_iteration_variable_names()[0]
residuals = log_viewer.get_residuals()

if with_plots:
    import matplotlib.pyplot as plt
    # Plot the data
    f, (ax1, ax2, ax3) = plt.subplots(3, sharex=True)
    ax1.plot(iter_data_second_newton[0][:, :])
    ax1.margins(0,0.1)
    ax1.set_title("IV values as a function of iterations in Newton solve invocation")
    ax1.set_ylabel("Values")
    ax1.legend(iv_names)
    ax2.semilogy(np.abs(iter_data_second_newton[0][:, :]-iter_data_second_newton[0]
[-1, :]))
    ax2.margins(0,0.1)
    ax2.set_ylabel("Distance")
    ax3.semilogy(np.abs(residuals[0]))
    ax3.set_ylabel("Residuals")
    ax3.set_xlabel("Iteration #")
    plt.tight_layout()

# For several functions one may specify which iteration variables one is interested in.
nominal_data_second_newton =
 log_viewer.get_iteration_variable_nominal(iv_names=[iv_names[0]])

# Retrieve the solver trace
solver_trace = log_viewer.get_solver_trace()


# If the steady-state solver was used several times on the same problem it will have
# filled the log file with information from the different solver invocations. To get
# information about a particular solver invocation and/or solve provide the solve index
# as argument. Solve indices can be found through
print("Solve info:")
print(log_viewer.get_solve_info())

# E.g. to get the values of iteration variables for the second
# solver invocation log present in the log the following is used:
# Solve index is 2 since two Newton solves have been performed in solver invocation 0.
iter_data_second_newton = log_viewer.get_iteration_variables(2)
# Or to only get information for a particular variable
iter_data_second_newton = log_viewer.get_iteration_variables(2, iv_names=[iv_names[0]])

# Other functions provided by LogViewer
res_var_data_second_newton = log_viewer.get_residuals()
res_data_second_newton = log_viewer.get_scaled_residual_norm()
jac_data_second_newton = log_viewer.get_jacobians()
iter_num_second_newton = log_viewer.get_number_of_iterations()

# For some of LogViewer's functions it is possible to specify a solve index
```

```
# together with an iteration number to retrieve data only from there.
# Note from the get_solve_info() output that solve index represents a Newton
# solve
newton_step = log_viewer.get_step("Newton", 1, 0, scaled_step=False)
lambda_max = log_viewer.get_step_ratio("lambda_max", 1, 0)
```

The following plot was generated in the example:

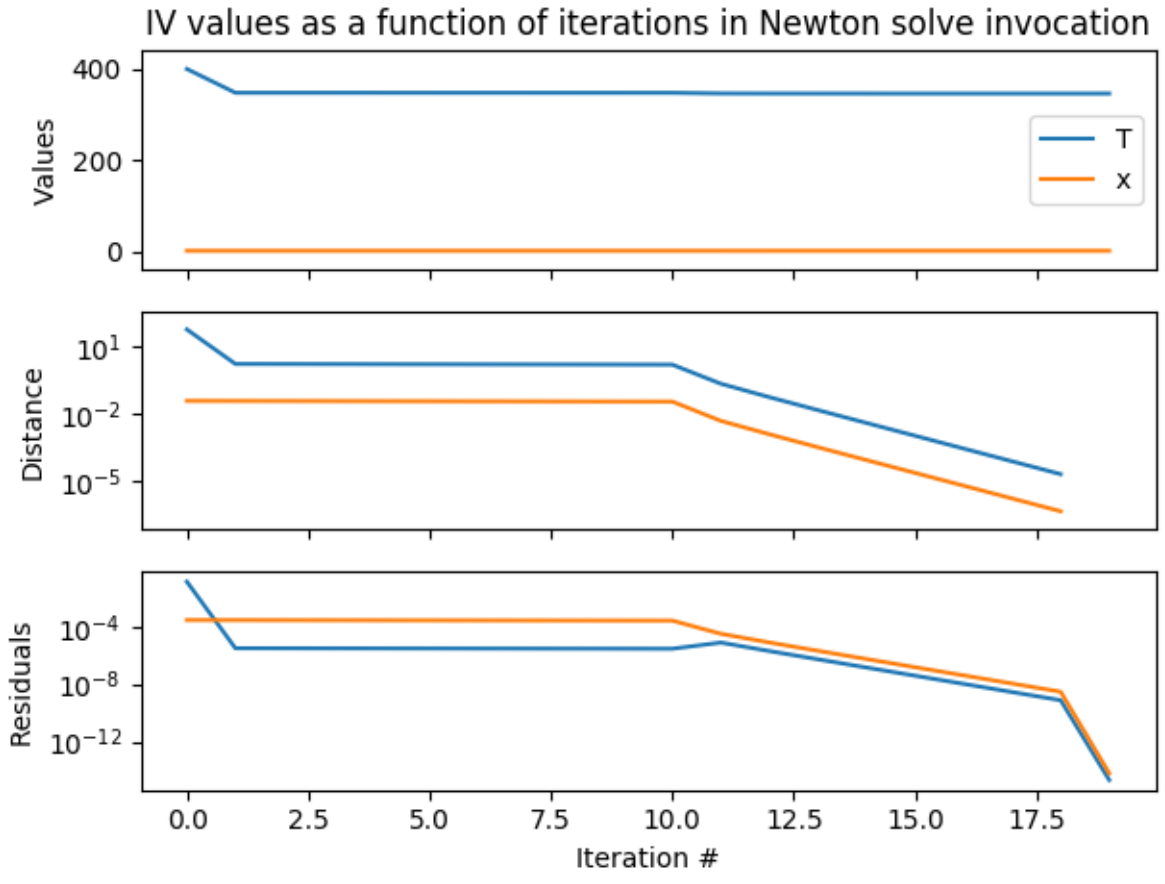


Figure 9.1 Using LogViewer to extract data from a log file in Python.

Creating Custom Problem Class (Python)

It is also possible to implement custom subclasses by utilizing inheritance of the `Problem` class. Below we show a complete class.

```
# Solving:
# FX = if(x < -1)
#         x + (x+1)*COEFF;
#     else
#         if(x > 1)
#             x + (x - 1)*COEFF;
#         else
#             x;

import numpy as np
from oct.steadystate.nlesol.problem import Problem

class TestProblem(Problem):

 def __init__(self):
  Problem.__init__(self)
  self.FX = 2
  self._COEFF = 2
  self._n_iteration_variables = 1
  self._n_discontinuity_indicators = 2
  self._x_initial = np.array([2])
  self._x = self._x_initial
  self.sw0 = False
  self.sw1 = False


 def get_number_of_iteration_variables(self):
  return self._n_iteration_variables

 def get_number_of_discontinuity_indicators(self):
  return self._n_discontinuity_indicators

 def get_initial_guess(self):
  return np.array([self._x_initial])

 def get_iteration_variables(self):
  return np.array([self._x_initial])

 def evaluate_discontinuity_indicators(self, ivs = None):
  if ivs is None:
   ivs = self._x
  return np.array([-ivs[0] - 1, ivs[0] - 1])

 def make_discontinuous_change(self):
  self.sw0 = np.all(np.less(self._x, -1))
  self.sw1 = np.all(np.greater(self._x, 1))

 def set_iteration_variables(self, x):
  self._x = x

 def evaluate_residuals(self, x):
```

```
   if self.sw0:
    val = x + (x + 1)*self._COEFF
   elif self.sw1:
    val = x + (x - 1)*self._COEFF
   else:
    val = x
   return_val = self.FX - val
   self.set_iteration_variables(x)
   return return_val
```

This class can then be sent to the solver in much the same manner as above:

```
from oct.steadystate.nlesol import Solver
problem = TestProblem()
solver = Solver(cls.problem)
```

Hold Iteration Variables (Python)

In this example we show the functionality of holding iteration variables for `FMUProblem`. It uses the model from Section 9.2.2.1, which is included in the ExampleModels.mo-package. The Python script to run this example is *hold_IVs.py*.

```
# The FMUProblem in this example contains two iteration variables. Using
# printInfo its structure is shown:

problem.print_info()
:::::::: Iteration variables ::::::::
   1: 4.0000000000000000E+02  T
   2: 1.0000000000000000E+00  x
:::::::: Residual equations ::::::::
   0: -6.2700000000000000E+02    0 = (- x) * (873 - T) + 11.0 * (T - 300)
   1: -1.2000000000000000E+02    0 = 120 * x - 75 * (0.12 * exp(12581 * (T - 298) / (298
* T))) * (1 - x)


# To hold an iteration variable the method "hold_iteration_variables" is
# provided. Let's hold variable T
problem.hold_iteration_variables([0])
print('problem.hold_iteration_variables([0])')
# We could also use {'T'} as an argument

# Using printInfo again we see that T, and the corresponding residual
# equation is removed.
problem.print_info()

problem.hold_iteration_variables([0])
:::::::: Iteration variables ::::::::
```

```
    1: 1.0000000000000000E+00     x
 :::::::: Residual equations ::::::::
    1: -1.2000000000000000E+02    0 = 120 * x - 75 * (0.12 * exp(12581 * (T - 298) / (298
* T))) * (1 - x)

 # To restore the system we can use the method "releaseIterationVariables"
 problem.release_iteration_variables([0])
 print('\nproblem.release_iteration_variables([0])')
 problem.print_info()

 problem.release_iteration_variables([0])
 :::::::: Iteration variables ::::::::
    0: 4.0000000000000000E+02     T
    1: 1.0000000000000000E+00     x
 :::::::: Residual equations ::::::::
    0: -6.2700000000000000E+02    0 = (- x) * (873 - T) + 11.0 * (T - 300)
    1: -1.2000000000000000E+02    0 = 120 * x - 75 * (0.12 * exp(12581 * (T - 298) / (298
* T))) * (1 - x)


 # We can specify which residual eqautions that should be removed
 problem.hold_iteration_variables([1], [0]) # Remove T, and res. eq 1
 print('\nproblem.hold_iteration_variables([1], [0])')
 problem.print_info()

 problem.hold_iteration_variables([1], [0])
 :::::::: Iteration variables ::::::::
    0: 4.0000000000000000E+02     T
 :::::::: Residual equations ::::::::
    1: -1.2000000000000000E+02    0 = 120 * x - 75 * (0.12 * exp(12581 * (T - 298) / (298
* T))) * (1 - x)

 # Most methods on FMUProblem also lets you specify if you want to apply the
 # method for active, held or all variables. Default is active.
 print('\nproblem.printInfo("held")')
 problem.print_info('held')

 problem.printInfo("held")
 :::::::: Iteration variables ::::::::
    1: 1.0000000000000000E+00     x
 :::::::: Residual equations ::::::::
    0: -6.2700000000000000E+02    0 = (- x) * (873 - T) + 11.0 * (T - 300)

 print('\nproblem.printInfo("all")')
 problem.print_info('all')

 problem.printInfo("all")
 :::::::: Iteration variables ::::::::
    0: 4.0000000000000000E+02     T
    1: 1.0000000000000000E+00     x
 :::::::: Residual equations ::::::::
```

```
    0: -6.2700000000000000E+02    0 = (- x) * (873 - T) + 11.0 * (T - 300)
    1: -1.2000000000000000E+02    0 = 120 * x - 75 * (0.12 * exp(12581 * (T - 298) / (298
* T))) * (1 - x)


 # It also possible change min/max/nominal attributes on the iteration
 # variables. This can e.g. be used if the initial guess is close
 # to the solution, where tighter bounds provides additional
 # guidance to the solver. This may be particularly useful in case of
 # multiple solutions.
```

Parametric Hold Iteration Variables (Python)

In addition to manually held variables there is functionality for setting up hold specifications based on parameters on the FMU. The Modelica compiler provided by OCT supports special annotations for marking variables for parametric holding, see Section 14.1 for a complete specification. The basic methodology behind parametric holding is that iteration variables and residuals are bound to boolean parameters. When the problem is constructed the variables and residuals bound to boolean parameters that are true are automatically held.

The model for this example is included in the ExampleModels.mo-package and the script to run the example is parametric_hold_IV.py. In the model, parameters for parametric holding is specified for the iteration variables and residuals as follows:

```
model twoEqSteadyStateParaHold
 Real x (start = 1);
 Real T (start = 400);
 parameter Boolean testHold1 = false;
 parameter Boolean testHold2 = false;
equation
 0 = 120*x - 75*(0.12*exp(12581*(T - 298)/(298*T)))*(1 - x)
  annotation(__Modelon(ResidualEquation(hold = testHold1,
   iterationVariable(hold = testHold2)=x), name = LongEq));
 0 = -x*(873 - T) + 11.0*(T - 300)
  annotation(__Modelon(ResidualEquation(hold = testHold2,
   iterationVariable(hold = testHold1)=T), name = LongEq2));
end twoEqSteadyStateParaHold;
```

We begin by setting up the problem.

```
import os
from pymodelica import compile_fmu
from pyfmi import load_fmu
from oct.steadystate.nlesol import FMUProblem

compiler_opts = {"interactive_fmu": True,
```

```
      "expose_scalar_equation_blocks_in_interactive_fmu": True,
      "hand_guided_tearing": True}
 model_name = 'ExampleModels.twoEqSteadyStateParaHold'
 name = compile_fmu(model_name, "/../files/example_models.mo",
compiler_options=compiler_opts)

 model = load_fmu(name, log_level=4)
 model.set("testHold1", True)

 problem = FMUProblem(model)
```

Now utilizing `print_info()` we get the following.

```
 print('problem.print_info()')
 problem.print_info()

 problem.print_info()
 :::::::: Iteration variables ::::::::
    1: 1.0000000000000000E+00     x
 :::::::: Residual equations ::::::::
    0: -6.2700000000000000E+02    LongEq2

 print('problem.print_info("parametric_hold")')
 problem.print_info("parametric_hold")

 problem.print_info("parametric_hold")
 :::::::: Iteration variables ::::::::
    0: 4.0000000000000000E+02     T
 :::::::: Residual equations ::::::::
    1: -1.2000000000000000E+02    LongEq

 print('problem.print_info("all")')
 problem.print_info("all")

 problem.print_info("all")
 :::::::: Iteration variables ::::::::
    0: 4.0000000000000000E+02     T
    1: 1.0000000000000000E+00     x
 :::::::: Residual equations ::::::::
    0: -6.2700000000000000E+02    LongEq2
    1: -1.2000000000000000E+02    LongEq
```

9.2.2.4. MATLAB Examples

These examples assume that the +oct folder, as provided in the installation, is on the MATLAB® path. Some of the examples described are included in the `install/MATLAB/examples/nlesol` folder.

Introductory Example (MATLAB)

The simple introductory example below shows how an `FMUProblem` instance is created, where FMIT is used to load an FMU model. A solver instance is created from the `FMUProblem` instance, and the steady-state solver is invoked.

```
import oct.nlesol.*; % Import nlesol package.
fmu = FMUModelME1('SteadyStateProblem.fmu');
fmuProblem = FMUProblem(fmu);
solver = Solver(fmuProblem);
sol = solver.solve();
```

Complete Example (MATLAB)

This model is included in the ExampleModels.mo-package and the corresponding script to run it is *example_SimpleSteadyState.m*. In a first step, create a steady-state Modelica model and save it to a file SimpleSteadyState.mo.

```
model SimpleSteadyState
        parameter Real  a = 2;
        input Real  b(start = 2);
        Real x (start = -2);
    equation
        b = if (x < -1.0)
                then x + (x + 1)*a
            else
                if (x > 1)
                    then x + (x - 1)*a
                else
                    x;
end SimpleSteadyState;
```

Next we compile the FMU with the required interactive argument using the MATLAB® OCT compiler interface.

```
import oct.modelica.*; % Import compiler package.
import oct.nlesol.*; % Import nlesol package.
modelName = 'ExampleModels.SimpleSteadyState';
compiler = 'OCT_Modelica'; % To compile with the OCT provided Modelica compiler that
                      % supports special handling of iteration and residual variables.
exampledir = fileparts(mfilename('fullpath')); % Path to example directory
lib = {[exampledir, '\ExampleModels.mo']}; % Absolute path to model

% The options needed for special handling of iteration and residual variables.
opt = {'interactive_fmu', true, 'expose_scalar_equation_blocks_in_interactive_fmu', true};
fmuName = compileFMU(modelName, compiler, 'libs', lib, 'options', opt);
```

Next we use FMIT to create an FMU model instance. Note that we create an FMU for Model Exchange in order to match the FMU generated by the compiler.

```
fmu = FMUModelME1(fmuName);
```

```
fmuProblem = FMUProblem(fmu);
```

Note that it is also possible to supply the names of iteration and residual variables when the `FMUProblem` is created, for the model above the following would have worked as well:

```
fmuProblem = FMUProblem(fmu, {'x'}, {'res_0'});
```

This feature is useful when not all iteration and residual variables in an interactive FMU are used.

Next we create a `Solver` instance and invoke the steady-state solver.

```
solver = Solver(fmuProblem);
solver.solve()
```

This will produce the following output (described in more detail in Section 9.2.2.2).

```
Model name.....................: ExampleModels.SimpleSteadyState
Number of iteration variables......: 1
Number of discontinuity switches...: 2

Switch iteration 1
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   1Js   1.0000e+00  1.0000e+00:   1     0    0   2.0000e-01:   1r  2.0000e-01
   2     8.0000e-01  8.0000e-01:   1     0    0   2.0000e-01:   1r  2.0000e-01
   3     6.4000e-01  6.4000e-01:   1     0    0   2.0000e-01:   1r  2.0000e-01
   4     5.1200e-01  5.1200e-01:   1     0    0   2.0000e-01:   1r  2.0000e-01
   5     4.0960e-01  4.0960e-01:   1     0    0   2.4414e-01:   1r  2.4414e-01
   6     3.0960e-01  3.0960e-01:   1     0    0   3.2300e-01:   1r  3.2300e-01
   7     2.0960e-01  2.0960e-01:   1     0    0   4.7710e-01:   1r  4.7710e-01
   8     1.0960e-01  1.0960e-01:   1     0    0   9.1241e-01:   1r  9.1241e-01
   9     9.6000e-03  9.6000e-03:   1     0    0   1.0000e+00        1.0000e+00
  10     0.0000e+00  0.0000e+00:   1
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   1s    0.0000e+00  0.0000e+00:   1
Switch iteration 2
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
   1     6.6667e-01  6.6667e-01:   1     0    0   3.0000e-01:   1r  3.0000e-01
   2     6.0000e-01  6.0000e-01:   1     0    0   3.3333e-01:   1r  3.3333e-01
   3     5.3333e-01  5.3333e-01:   1     0    0   3.7500e-01:   1r  3.7500e-01
   4     4.6667e-01  4.6667e-01:   1     0    0   4.2857e-01:   1r  4.2857e-01
   5     4.0000e-01  4.0000e-01:   1     0    0   5.0000e-01:   1r  5.0000e-01
   6     3.3333e-01  3.3333e-01:   1     0    0   6.0000e-01:   1r  6.0000e-01
   7     2.6667e-01  2.6667e-01:   1     0    0   9.0000e-01:   1r  9.0000e-01
   8     1.8667e-01  1.8667e-01:   1     0    0   1.0000e+00        1.0000e+00
   9     1.2444e-01  1.2444e-01:   1     0    0   1.0000e+00        1.0000e+00
  10J    8.2963e-02  8.2963e-02:   1     0    0   1.0000e+00        1.0000e+00
  11J    2.8553e-10 -2.8553e-10:   1     0    0   1.0000e+00        1.0000e+00
  12     0.0000e+00  0.0000e+00:   1
Switch iteration 3
iter       res_norm      max_res: ind   nlb  nab   lambda_max: ind      lambda
```

```
   1     6.6667e-01 -6.6667e-01:   1     0    0   2.0000e-01:   1r  2.0000e-01
   2     2.6667e-01 -2.6667e-01:   1     0    0   4.0000e-01:   1r  4.0000e-01
   3     5.3333e-02  5.3333e-02:   1     0    0   1.0000e+00        6.0000e-01
   4     4.2667e-02 -4.2667e-02:   1     0    0   1.0000e+00        6.0000e-01
   5     3.4133e-02  3.4133e-02:   1     0    0   1.0000e+00        6.0000e-01
   6     2.7307e-02 -2.7307e-02:   1     0    0   1.0000e+00        6.0000e-01
   7     2.1845e-02  2.1845e-02:   1     0    0   1.0000e+00        6.0000e-01
   8     1.7476e-02 -1.7476e-02:   1     0    0   1.0000e+00        6.0000e-01
   9     1.3981e-02  1.3981e-02:   1     0    0   1.0000e+00        6.0000e-01
  10J    1.1185e-02 -1.1185e-02:   1     0    0   1.0000e+00        1.0000e+00
  11J    7.5318e-11  7.5318e-11:   1     0    0   1.0000e+00        1.0000e+00
  12     7.4015e-17  7.4015e-17:   1
Number of function evaluations: 60
Number of jacobian evaluations: 5
Solver finished
Total time in solver: 0.35 s

solution = 1.3333
```

Log Information Retrieval (MATLAB)

Below an example is presented where a steady-state problem is solved and the generated log-file provided to the `LogViewer`. This model is included in the ExampleModels.mo-package and the corresponding script to run it is *example_LogViewer.m*. Here we assume that the model `ExampleModels.twoEqSteadyState` has been compiled exactly as in the previous example.

```
% Use FMIT to create an fmu model object.
% Verbose is an argument to FMIT to indicate that we wish
% to retain all log information generated in the FMU
fmu = FMUModelME1(fmuName, 'verbose');

% Create the problem class instance from the fmu and supply it to the
% solver.
fmuProblem = FMUProblem(fmu);
solver = Solver(fmuProblem);
% Set a log level sufficiently high to get values of iteration
% variables emitted to the log file.
solver.setOptions('log_level', 5)

% Solve the problem
solver.solve();

% Create the LogViewer from the solver class instance we
% created above as such
logViewer = LogViewer(solver)

% LogViewer can also be supplied with the log-file, e.g.
% LogViewer('twoEqSteadyState_log.xml')
```

```
% It is now possible to play around and retrieve information from the log.
% E.g. retrieve the values of the iteration variables at different
% times during the solver invocation.
iterDataLatest = logViewer.getIterationVariables()

% Plot the data
iterVarNames = fmuProblem.getIterationVariableNames();
fig1 = figure(1);
plot(iterDataLatest{1}(:, 1))
xlabel(['Values for: ', iterVarNames{1}, ...
     ' as a function of iterations in newton solve invocation'])

% For several functions one may specify which iteration variables one is interested in.
nominalDataLatest = logViewer.getIterationVariableNominal({iterVarNames{1}})

% Retrieve the solver trace
solverTrace = logViewer.getSolverTrace()

%% If the steady-state solver was used several times on the same problem it will have
% filled the log file with information from the different solver invocations. To get
% information about a particular solver invocation and/or solve provide the solve index
% as argument. Solve indices can be found through
logViewer.getSolveInfo();
% E.g. to get the values of iteration variables for the second
% solver invocation log present in the log the following is used:
% Solve index is 4 since two Newton solves have been performed in solver invocation 1.
iterDataLatest = logViewer.getIterationVariables(4)
% Or to only get information for a particular variable
iterDataLatest = logViewer.getIterationVariables({iterVarNames{1}}, 4)

% Other functions provided by LogViewer
resVarDataLatest = logViewer.getResiduals()
resDataLatest = logViewer.getScaledResidualNorm()
jacDataLatest = logViewer.getJacobians()
iterNumDataLatest = logViewer.getNumberOfIterations()

%% For some of LogViewer's functions it is possible to specify a solve index
% together with an iteration number to retrieve data only from there.
% Note from the getSolveInfo() output that solve index represents a Newton
% solve
newtonStep = logViewer.getStep('Newton', 'unscaled', 2, 1)
lambdaMax = logViewer.getStepRatio('lambda_max', 2, 1)
```

The following plot was generated in the example:

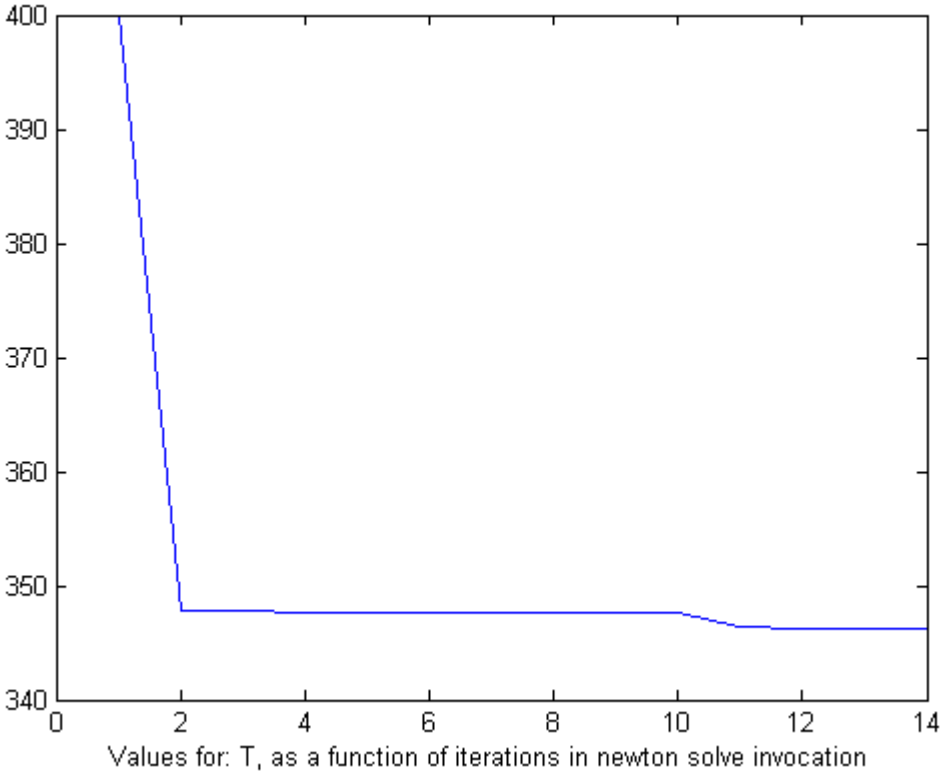


Figure 9.2 Using LogViewer to extract data from a log file in MATLAB®.

For log files generated by FMIT:

```
% FMIT generates logs in .txt format. Supply these to LogViewer which will
% extract the XML data present in them.
LogViewer('twoEqSteadyState_log.txt');
% LogViewer will create a new file, typically names 'twoEqSteadyState_log_extracted.xml'
% in this case and load it as well.
```

Creating Custom Problem Class (MATLAB)

It is also possible to implement subclasses to the `Problem` class. Below, a complete class TestProblem which is a subclass of `Problem` is given.

```
% Solving:
% FX = if(x < -1)
%         x + (x+1)*COEFF;
```

```
%     else
%         if(x > 1)
%             x + (x - 1)*COEFF;
%         else
%             x;
%     FX, COEFF defined below, INITX - initial X

classdef TestProblem < oct.nlesol.Problem

    properties (Access = public)
        sw0;
        sw1;
        COEFF = 2;
        FX = 2;
    end

    methods(Access = public)
       function this = TestProblem()
           this.sw0 = false;
           this.sw1 = false;
       end

        function [numberOfIterationVariables] = getNumberOfIterationVariables(this)
            numberOfIterationVariables = 1;
        end

        function [numDiscontinuities] = getNumberOfDiscontinuities(this)
            numDiscontinuities = 2;
        end

        function [indicators] = evaluateDiscontinuityIndicators(this, x)
            indicators = [-x - 1; x - 1 ];
        end

        function [residual] = evaluateResiduals(this, x)
            if(this.sw0)
                v = x + (x + 1)*this.COEFF;
            elseif(this.sw1)
                v = x + (x - 1)*this.COEFF;
            else
                v = x;
            end
            residual = this.FX - v;
        end

        function [xInitial] = getInitialGuess(this)
            xInitial = 2;
        end

        function makeDiscontinuousChange(this, x)
            this.sw0 = (x < -1);
```

```
            this.sw1 = (x > 1);
        end
    end
end
```

This class can then be sent to the solver in much the same manner as above:

```
import oct.nlesol.*; % Import nlesol package.
testProblem = TestProblem();
solver = Solver(testProblem);
solution = solver.solve()
```

Hold Iteration Variables (MATLAB)

This example shows how the hold iteration variable functionality works for `FMUProblem`. It uses the model from Section 9.2.2.1, which is included in the ExampleModels.mo-package. The MATLAB® script to run this example is *example_HoldIVs.m*.

```
% The FMUProblem in this example contains two iteration variables. Using
% printInfo its structure is shown:

fmuProblem.printInfo()
:::::::: Iteration variables ::::::::
   1: 4.0000000000000000E+02  T
   2: 1.0000000000000000E+00  x
:::::::: Residual equations ::::::::
   1: -6.2700000000000000E+02  0 = (- x) * (873 - T) + 11.0 * (T - 300)
   2: -1.2000000000000000E+02  LongEq


% To hold an iteration variable the method "holdIterationVariables" is
% provided. Let's hold variable T
fmuProblem.holdIterationVariables([1])
% We could also use {'T'} as an argument

% Using printInfo again we see that T, and the corresponding residual
% equation is removed.
fmuProblem.printInfo()
:::::::: Iteration variables ::::::::
   2: 1.0000000000000000E+00  x
:::::::: Residual equations ::::::::
   2: -1.2000000000000000E+02  LongEq

% To restore the system we can use the method "releaseIterationVariables"
fmuProblem.releaseIterationVariables([1])

fmuProblem.printInfo()
:::::::: Iteration variables ::::::::
   1: 4.0000000000000000E+02  T
   2: 1.0000000000000000E+00  x
:::::::: Residual equations ::::::::
```

```
   1: -6.2700000000000000E+02  0 = (- x) * (873 - T) + 11.0 * (T - 300)
   2: -1.2000000000000000E+02  LongEq


% We can specify which residual equations that should be removed

fmuProblem.holdIterationVariables([2], [1]) % Remove x, and res. eq 1
fmuProblem.printInfo()
:::::::: Iteration variables ::::::::
   1: 4.0000000000000000E+02  T
:::::::: Residual equations ::::::::
   2: -1.2000000000000000E+02  LongEq


% Most methods on FMUProblem also lets you specify if you want to apply the
% method for active, held or all variables. Default is active.

fmuProblem.printInfo('held')
:::::::: Iteration variables ::::::::
   2: 1.0000000000000000E+00  x
:::::::: Residual equations ::::::::
   1: -6.2700000000000000E+02  0 = (- x) * (873 - T) + 11.0 * (T - 300)

fmuProblem.printInfo('all')
:::::::: Iteration variables ::::::::
   1: 4.0000000000000000E+02  T
   2: 1.0000000000000000E+00  x
:::::::: Residual equations ::::::::
   1: -6.2700000000000000E+02  0 = (- x) * (873 - T) + 11.0 * (T - 300)
   2: -1.2000000000000000E+02  LongEq

% It also possible change min/max/nominal attributes on the iteration
% variables. This can e.g. be used if the initial guess is close
% to the solution, where tighter bounds provides additional
% guidance to the solver. This may be particularly useful in case of
% multiple solutions.
```

Parametric Hold Iteration Variables (MATLAB)

In addition to manually held variables there is functionality for setting up hold specifications based on parameters on the FMU. The Modelica compiler provided by OCT supports special annotations for marking variables for parametric holding, see Section 14.1 for a complete specification. The basic methodology behind parametric holding is that iteration variables and residuals are bound to boolean parameters. When the problem is constructed the variables and residuals bound to boolean parameters that are true are automatically held.

The model for this example is included in the ExampleModels.mo-package and the script to run the example is example_ParametricHoldIV.m. In the model, parameters for parametric holding is specified for the iteration variables and residuals as follows:

```
model twoEqSteadyStateParaHold
    Real x (start = 1);
    Real T (start = 400);
    parameter Boolean testHold1 = false;
    parameter Boolean testHold2 = false;
equation
    0 = 120*x - 75*(0.12*exp(12581*(T - 298)/(298*T)))*(1 - x)
        annotation(__Modelon(ResidualEquation(hold = testHold1,
            iterationVariable(hold = testHold2)=x), name = LongEq));
    0 = -x*(873 - T) + 11.0*(T - 300)
        annotation(__Modelon(ResidualEquation(hold = testHold2,
            iterationVariable(hold = testHold1)=T), name = LongEq2));
end twoEqSteadyStateParaHold;
```

This model could be used to set up parametric holding:

```
fmu = loadFMU(fmuName);
fmu.setValue('testHold1', true);
fmuProblem = FMUProblem(fmu);

% Print the active variables - notice that an iteration/residual
% pair is held
fmuProblem.printInfo()
:::::::: Iteration variables ::::::::
   2: 1.0000000000000000E+00  x
:::::::: Residual equations ::::::::
   1: -6.2700000000000000E+02  LongEq2

% Print the parametrically held variables
fmuProblem.printInfo('parametricHold')
:::::::: Iteration variables ::::::::
   1: 4.0000000000000000E+02  T
:::::::: Residual equations ::::::::
   2: -1.2000000000000000E+02  LongEq
```

Parametric Start Attributes (MATLAB)

If the iteration variables were automatically found as part of the call to the constructor of `FMUProblem`, then the initial guess for all iteration variables that have start values that depend on a parametric expression will be altered. The parametric expression will be evaluated and the value set as the initial guess for all such variables.

It is possible to bypass this behavior and to set the initial guess explicitly if the values are set on the FMU followed by initialization of the FMU before it is passed as an argument to the constructor.

The model for this example is included in the ExampleModels.mo-package and the script to run the example is *example_ParametricStart.m.* In the model, one of the iteration variables has a parametric expression in its start value:

```
model twoEqSteadyStateParaStart
    parameter Real x_start = 1;
    Real x (start = x_start);
    Real T (start = 400);
equation
    0 = 120*x - 75*(0.12*exp(12581*(T - 298)/(298*T)))*(1 - x)
        annotation(__Modelon(ResidualEquation(iterationVariable=x)));
    0 = -x*(873 - T) + 11.0*(T - 300)
        annotation(__Modelon(ResidualEquation(iterationVariable=T)));
end twoEqSteadyStateParaStart;
```

This model would have the initial guess for `x` altered as part of the call to the constructor of `FMUProblem`:

```
% Change intial guesses for the iteration variables. The
% variables whose start values are not determined by a
% parameteric are not changed directly since FMU not initialized before
% call to FMUProblem constructor. Fixed start values will be used if there
% are any, otherwise parametric.
fmu = loadFMU(fmuName);
fmu.setValue('x_start', 10); % This will change guess for x
fmu.setValue('T', 20); % This will not change guess for T
fmuProblem = FMUProblem(fmu);

fmu.getValue('x')
10
fmu.getValue('T')
400

% Change intial guesses for the iteration variables. The
% variables whose start values are set as parameteric are not
% changed directly since fmu initialized
fmu = loadFMU(fmuName);
fmu.setValue('x_start', 10); % This will not change guess for x
fmu.setValue('T', 20); % This will change guess for T
fmu.fmiInitialize();
fmuProblem = FMUProblem(fmu);
fmu.getValue('x')
1
fmu.getValue('T')
20

% If the value of an iteration variable with a parametric start
% expression is explicitly set it will not be used since fmu not initialized:
fmu = loadFMU(fmuName);
fmu.setValue('x', 10); % This will not change guess for IterVar1
fmuProblem = FMUProblem(fmu);
fmu.getValue('x')
1
```

```
% However, if fmiInitialize was called before instantiation of the FMUProblem
% then the behavior would be bypassed and the above would work.
fmu = loadFMU(fmuName);
fmu.setValue('x', 10);
fmu.fmiInitialize(); % Bypass parameteric start behavior
fmuProblem = FMUProblem(fmu);
fmu.getValue('x')
10

% The same holds if the FMUProblem constructor is called with the
% argument pair ('guesses', 'interactive').
fmu = loadFMU(fmuName);
fmu.setValue('x', 10);
fmuProblem = FMUProblem(fmu, 'guesses', 'interactive');
fmu.getValue('x')
10

% No initialization but FMUProblem interactive
fmu = loadFMU(fmuName);
fmu.setValue('x_start', 10); % This will not change guess for x
fmuProblem = FMUProblem(fmu, 'guesses', 'interactive');
fmu.getValue('x')
1
fmu.getValue('T')
400

% Initialization but FMUProblem parametric, change of x_start has effect
% since GUESS_KIND set to 'parametric'
fmu = loadFMU(fmuName);
fmu.setValue('x_start', 10); % This will change guess for x
fmu.fmiInitialize();
fmuProblem = FMUProblem(fmu, 'guesses', 'parametric');
fmu.getValue('x')
10
fmu.getValue('T')
400
```

Chapter 10. Interactive Continuation Solver in MATLAB® and Python

10.1. Introduction

A continuation solver is intended for solution of parameterized steady-state problems formalized as non-linear equation systems. The solver is primarily intended to be used with the non-linear equation solver that is a part of OPTIMICA Compiler Toolkit. The intended application is solution of steady-state model equations packaged into FMUs (Functional Mockup Units) ([FMI2017]). The solver utilizes the interfaces provided by the FMI Toolbox for MATLAB® and PyFMI for Python, to interact with FMUs.

The solver is intended to be applied for parameterized equation systems on the form:

$$F(xa, G(a))=0,\ xmin<=x<=xmax,\ a0<=a<=a1 \tag{10.1}$$

Here xa is a column vector of unknowns of length nx and F(x,G(a)) is a column vector of residuals. The number of residuals is equal to the number of unknowns x. a is the continuation parameter that defines the boundary conditions G(a). The continuation solver rely on an external solver for finding the solution to the steady-state equations for any specified, fixed, value of the continuation parameter. Continuation solver assumes that solution for a0 is available or can be easily computed. It then construct a sequence of steady-state problems by modifying the continuation parameter, striving to reach a=a1, while starting from a0. The equations are solved using the external solver starting with an initial guess for the iteration variables: xa = x0. Accuracy is controlled both by the solver options and nominal values for the iteration variables, xnom.

The function F(xa,G(a)) is expected to be C^1 continuous with respect to x inside the bounds and with respect to the continuation parameter a on the processed interval.

10.2. Solver Interface

The solver interface is object-oriented and is contained in the MATLAB® package: oct.nlesol and Python package: oct.steadystate.nlesol, respectively. Detailed documentation of the methods of the provided classes is available at the MATLAB® command prompt via help and doc commands and through Python docstrings.

The package contains the following classes:

`ContinuationProblem` is an abstract interface class defining the parameterized non-linear equation system as used by the ContinuationSolver. It is possible to define custom problems by sub-classing this class and overloading the required methods.

`ContinuationFMUProblem` is a sub-class of the ContinuationProblem that provides the means to interact with an FMU. This class relies on an instance of oct.nlesol.Solver/oct.steadystate.nlesol.Solver class attached to a

oct.nlesol.FMUProblem/oct.steadystate.nlesol.FMUProblem class to process steady-state problems in the FMU. Note that in order to use the default constructors the FMU needs to follow the interactive FMU variable naming convention as implemented when using the interactive_fmu flag to the OCT compiler. That is, the iteration variables are identified by variables named iter_<nnn> where nnn is the iteration variable index. The description of each such variable corresponds to a name of the actual iteration variable that is defined as an input for the FMU. The corresponding residuals are identified by the output variables named res_<nnn>. Remaining input variables on the FMU are assumed to define the boundary conditions used for continuation.

`ContinuationSolver` is a class that encapsulates the continuation solver. There is functionality to retrieve and set solver options and run a continuation on equations that are defined by an instance of the ContinuationProblem class. The solution is returned from a call to the solve function. It can also be retrieved from the ContinuationProblem class on solver exit.
Continuation solver is controlled by a number of options listed in the table below

Table 10.1 Solver options

Option	Default	Description
init_step	0.5	Initial step length in continuation parameter [eps(1), 1000]
log_level	4	Log level for the solver [0, 6]
min_step	0.001	Minimum step legth in continuation parameter [eps(1), 1000]
max_step	10	Maximum step length in continuation parameter [eps(1), 1000]
step_increase_factor	1.5	Factor used to increase the step length on success [1, 10]
step_decrease_factor	2	Factor used to decrease the step length on failure (1, 10)
save_intermediate_solutions (Python only)	True	Flag indicating if intermediate results should be stored.

10.3. Example Python script

The example below demonstrates the intended call sequence for the solver.

```
# Load the
fmuname = 'model.fmu'
fmu = load_fmu(fmuname)

# Import needed classes
from oct.steadystate.nlesol import FMUProblem, Solver, ContinuationSolver,
 ContinuationFMUProblem
```

```
# Construct the FMUProblem class providing FMU as an argument
fmu_p = FMUProblem(fmu)
# Create the steady-state solver
ss_solver = Solver(fmu_p)

# Create the continuation problem
# By default all FMU inputs are taken as boundary conditions.
# Alternatively, second argument can be used to specify the list
# of variable names representing boundary conditions.
problem = ContinuationFMUProblem(ss_solver)

# Set the continuation target boundary conditions
# Set target values for continuation on boundary conditions
# Here we choose 2 as target and 1 as starting point (optional)
problem.setup_continuation_boundary([2],[1])

# Construct the ContinuationSolver class for the ContinuationProblem
solver = ContinuationSolver(problem);
# Retrieve the default solver options
opt = solver.solve_options()
# Set maximum step in continuation to 0.2
opt['max_step'] = 0.2
# Set low log level
opt['log_level'] = 2

# Run continuation
solution = solver.solve(options=opt)
```

10.4. Example MATLAB® script

The example below demonstrates the intended call sequence for the solver.

```
% Instantiate and initialize FMU using FMI Toolbox
fmuname = 'model.fmu';
fmu = FMUModelME1(fmuname);
fmu.fmiInstantiateModel();
fmu.fmiInitialize();

% Import the package namespace
import oct.nlesol.*
% Construct the FMUProblem class providing FMU as an argument
fmu_p = FMUProblem(fmu);
% Create the steady-state solver and set low log level
ssSolver = Solver(fmu_p);
ssSolver.setOptions('log_level',2);

% Create the continuation problem
% By default all FMU inputs are taken as boundary conditions.
% Alternatively, second argument can be used to specify the list
% of variable names representing boundary conditions.
```

```
problem = ContinuationFMUProblem(ssSolver);

% Set the continuation target boundary conditions
% Set target values for continuation on boundary conditions
% targetBoundaryVector = [<value>; <value>;…;<value>];
problem.setupContinuationBoundary(targetBoundaryVector);

% Construct the ContinuationSolver class for the ContinuationProblem
solver = ContinuationSolver(problem);
% Retrieve the default solver options
opt = solver.getOptions();
% Set maximum step in continuation to 0.2
opt.max_step = 0.2;
solver.setOptions(opt);

% Run continuation between continuation parameter values 0 and 0.5
% The arguments are optional and defaults to continuation between 0 and 1
solution = solver.solve(0,0.5);
```

Chapter 11. Steady-State Diagnostics in Python

In the steady-state package there are several different diagnostic tools available to aid the analysis of solving and converging models using the steady-state solver interface from Section 9.1. In this chapter we briefly discuss what they can do and how to apply them. Among the steady-state examples you can find an example notebook named `diagnostics-template.ipynb`. In the example notebook two different models are analyzed to demonstrate the application of the diagnostic tools.

11.1. Examples of Using The Diagnostic Tools

The diagnostics driver scripts provide pretty-printed ASCII output (when in the console) and HTML output, e.g. in Jupyter Notebook, allowing for easy overview of the problem properties suitable to the different diagnostic modes. We continue by going through each diagnostic tool, with some examples of how they can be used. The documentation for each method can be accessed using the built-in python function `help`, where you can see all available input parameters, as well as a brief explanation for each.

11.1.1. Plot and Analyze Residuals

The tool `analyze_residuals` allows us to plot residuals as a function of a specific variable. To do this we require an instance of the object `IndependentVariable`. An `IndependentVariable` is used to initialize an independent variable to plot residuals against. We can import these simply by typing the following:

```
from oct.steadystate.diagnostics.analyze_residuals import analyze_residuals,
 IndependentVariable
```

When we create an instance of `IndependentVariable` we also have to decide the range and the number of data points to plot. In this example we create two instances of `IndependentVariable` for our plots. We use different ranges for each axis and 100 points for each:

```
T = IndependentVariable("T", 0, 1000, 100)
x = IndependentVariable("x", 0, 10, 100)
```

For a given instance of `oct.steadystate.nlesol.Problem` denoted by `problem`, we can plot the iteration variables `T` and `x` against residuals with indices 0 and 1.

```
ares = analyze_residuals(problem, [0,1], [T,x])
```

This generates the results in Figure 11.1

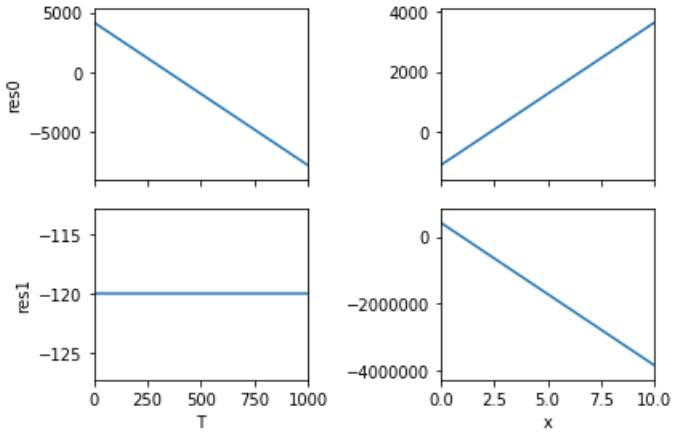


Figure 11.1 A plot of the residuals with indices 0 and 1 as a function of the iteration variables `T` and `x` respectively.

We can also plot each residual as a multivariate function of the variables `T` and `x`, results seen in Figure 11.2.

```
ares = analyze_residuals(problem, [0,1], T, x)
```

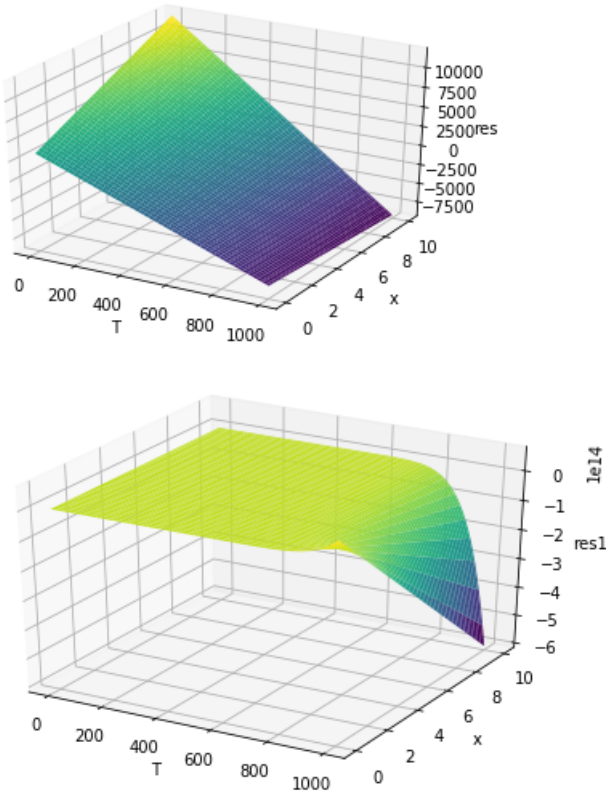


Figure 11.2 A plot of the residuals with indices 0 and 1 as a multivariate function the iteration variables `T` and `x`.

The object returned by `analyze_residuals` can be used to change the type of the plot. For example, using the object `ares` from above:

```
ares.plot(plot_mode="map")
```

Which generates the results seen in Figure 11.3.

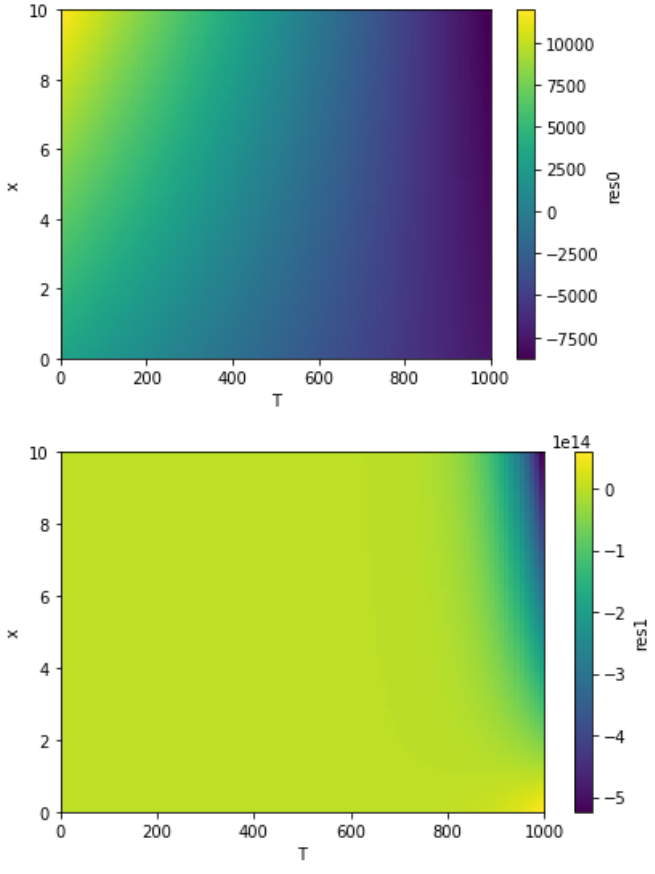


Figure 11.3 A heatmap of the residuals with indices 0 and 1 as a multivariate function the iteration variables `T` and `x`.

11.1.2. Diagnose Singular System

To diagnose reasons behind a singular system one can use the diagnostic tool `diagnose_singular_system`. We import it simply by:

```
from oct.steadystate.diagnostics.diagnose_singular_system import diagnose_singular_system
```

This tool uses instances of `oct.steadystate.nlesol.LogViewer` or `oct.steadystate.nlesol.Problem`. In this example we use an instance of `oct.steadystate.nlesol.LogViewer` denoted by `lv`. Then, the diagnostic tool is applied as follows:

```
diagnose_singular_system(lv, solve_index=1, iteration=5)
```

Note however that solve index and iteration are only required when using an instance of `oct.steadystate.nlesol.LogViewer`. This generates the results in Figure 11.4.

Characteristics of 3x3 Jacobian for Problem

	Unscaled	Scaled
Rank	2	2
Condition number	4.51130514745e+16	1.21203083593e+16
σ_{max}	3.43964211388	2.27251722396
σ_{min}	7.62449446769e-17	1.87496650794e-16

Linearly dependent groups

#	Scaled σ	Group		Var name
0	1.87496650794e-16			
		IVs	0	y
			1	z
		Residuals	0	res_1
			1	res_2
			2	res_0

Groups belonging to the maximum singular value

#	Scaled σ	Group		Var name
0	2.27251722396			
		IVs	0	x
			1	z
			2	y
		Residuals	0	res_2
			1	res_1
			2	res_0

Figure 11.4 HTML output from the diagnostic tool `diagnose_singular_system`.

11.1.3. Get Active and Limiting Bounds

The diagnostic tool `get_bounds` displays a nicely formatted table of all active/limiting bounds. If `lv` is an instance of `oct.steadystate.nlesol.LogViewer`, we can use `get_bounds` accordingly:

```
from oct.steadystate.diagnostics.get_bounds import get_bounds
get_bounds(lv)
```

This generates the results in Figure 11.5.

Limiting and Active Bounds

Solve index	Iteration	Limiting min	Limiting max	Active min	Active max	Interval		IV name
1	0	o				[9.00000e-01,	1.79769e+308]	x
			o		o	[-1.79769e+308,	1.00000e+01]	T
	1	o		o		[9.00000e-01,	1.79769e+308]	x
			o		o	[-1.79769e+308,	1.00000e+01]	T
2	0	o		o		[9.00000e-01,	1.79769e+308]	x
			o		o	[-1.79769e+308,	1.00000e+01]	T

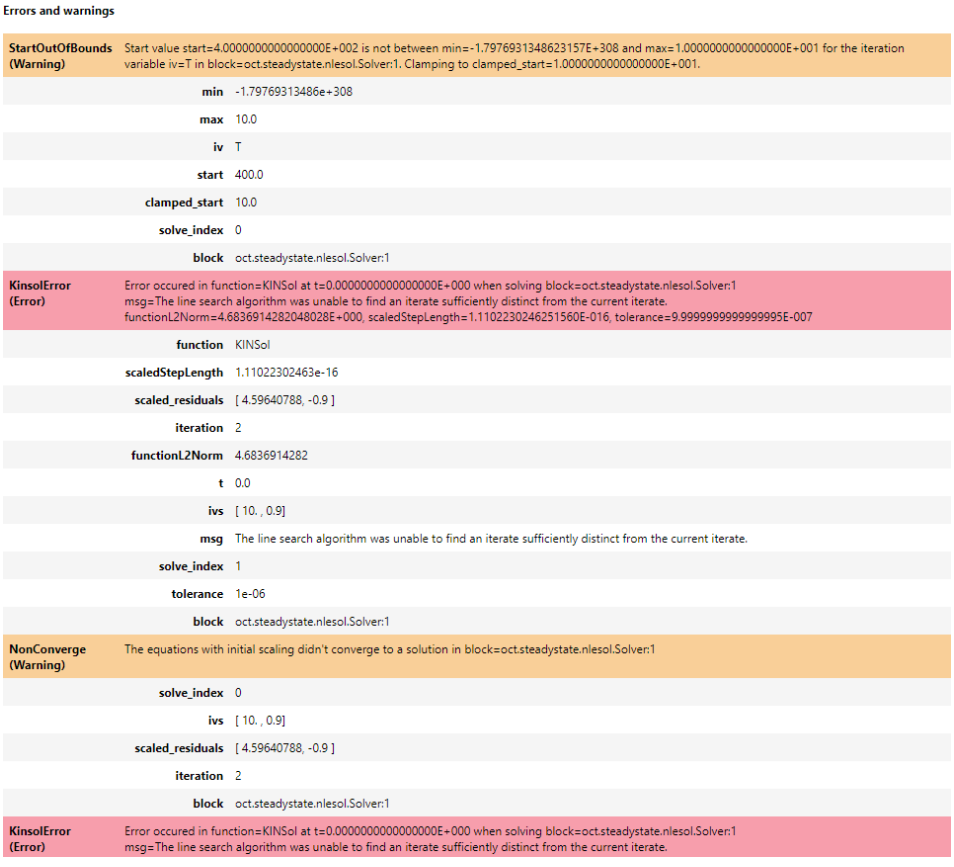
Figure 11.5 HTML output from the diagnostic tool `get_bounds`. When viewing the output in an HTML format, the interval values are colored orange when variables are on limiting bounds and red for active bounds. Note that we have a limiting bound in the first iteration, and active bounds in the other iterations.

11.1.4. Get Errors and Warnings

Using the diagnostic tool `get_errors_and_warnings` it is possible to summarize all errors and warnings encountered throughout a log. For a given instance of `oct.steadystate.nlesol.LogViewer`, which we denote by `lv`, we write:

```
from oct.steadystate.diagnostics.get_errors_and_warnings import get_errors_and_warnings
get_errors_and_warnings(lv)
```

The number of errors and warnings displayed can be controlled with the input keyword `limit_amount`. Using the default value, the results can be seen in Figure 11.6 This generates the results in Figure 11.6.



Errors and warnings

StartOutOfBounds (Warning)	Start value start=4.0000000000000000E+002 is not between min=-1.79769313486231570E+308 and max=1.0000000000000000E+001 for the iteration variable iv=T in block=oct.steadystate.nlesol.Solver:1. Clamping to clamped_start=1.0000000000000000E+001.
min	-1.79769313486e+308
max	10.0
iv	T
start	400.0
clamped_start	10.0
solve_index	0
block	oct.steadystate.nlesol.Solver:1
KinsolError (Error)	Error occured in function=KINSol at t=0.0000000000000000E+000 when solving block=oct.steadystate.nlesol.Solver:1 msg=The line search algorithm was unable to find an iterate sufficiently distinct from the current iterate. functionL2Norm=4.6836914282048028E+000, scaledStepLength=1.1102230246251560E-016, tolerance=9.9999999999999995E-007
function	KINSol
scaledStepLength	1.11022302463e-16
scaled_residuals	[4.59640788, -0.9]
iteration	2
functionL2Norm	4.6836914282
t	0.0
ivs	[10. , 0.9]
msg	The line search algorithm was unable to find an iterate sufficiently distinct from the current iterate.
solve_index	1
tolerance	1e-06
block	oct.steadystate.nlesol.Solver:1
NonConverge (Warning)	The equations with initial scaling didn't converge to a solution in block=oct.steadystate.nlesol.Solver:1
solve_index	0
ivs	[10. , 0.9]
scaled_residuals	[4.59640788, -0.9]
iteration	2
block	oct.steadystate.nlesol.Solver:1
KinsolError (Error)	Error occured in function=KINSol at t=0.0000000000000000E+000 when solving block=oct.steadystate.nlesol.Solver:1 msg=The line search algorithm was unable to find an iterate sufficiently distinct from the current iterate.

Figure 11.6 HTML output from the diagnostic tool `get_errors_and_warnings`. When viewing the output in an HTML format, warnings are colored orange and errors are colored red.

11.1.5. Illegal Residuals in Solve

When running into illegal residuals, the diagnostic tool `illegal_residuals` is convenient because it tabulates the residual equations with illegal values. This tool works with an instance of `oct.steadystate.nlesol.LogViewer`. Optionally, if also given the corresponding instance of `oct.steadystate.nlesol.Problem` a more detailed description is displayed for the residual equations. For a given instance of `oct.steadystate.nlesol.LogViewer` which we denote by `lv`, and an instance of `oct.steadystate.nlesol.Problem` denoted by `problem`, we use the diagnostic tool accordingly:

```
from oct.steadystate.diagnostics.illegal_residuals import illegal_residuals
illegal_residuals(lv, problem)
```

This generates the output in Figure 11.7. In this example the initial evaluation resulted in illegal residuals before any iteration was performed.

Illegal residuals

Solve index	Iteration	Value	Residual index	Description
0		NaN	0	x = acos(y)
		∞	1	y = y/z
		NaN	3	x1 = acos(y1)
		∞	4	y1 = y1/z1
		NaN	0	x = acos(y)
		∞	1	y = y/z
		NaN	3	x1 = acos(y1)
		∞	4	y1 = y1/z1

Figure 11.7 HTML output from the diagnostic tool `illegal_residuals`.

11.1.6. Nominals and Iteration Variable Diagnostics

The diagnostic tool `analyze_iteration_variable_magnitudes` makes it possible to diagnose nominals whose magnitude differ a lot compared to the magnitude of its corresponding iteration variable. The factor that defines too large nominals is easily changed through the input keyword `factor`. For example, for a given instance of `oct.steadystate.nlesol.LogViewer` which we denote by `lv`, we can use this tool as:

```
from oct.steadystate.diagnostics.iv_diagnostics import \
                                    analyze_iteration_variable_magnitudes
analyze_iteration_variable_magnitudes(lv)
```

This generates the results in Figure 11.8.

Suspicious Nominals

Summary

IV nominal	IV name
1.345000000000000e-07	y
1.123456789000000e+08	x
1.000000000000000e+01	z

Details

solve_index	iteration	IV value	IV nominal	Ratio	IV name
1	0	1.200000000000000e+01	1.123456789000000e+08	1.068131869200000e-07	x
		1.000000000000000e-02	1.345000000000000e-07	7.434944237918216e+04	y
		1.000000000000000e-04	1.000000000000000e+01	1.000000000000000e-05	z
	1	1.199975747760534e+01	1.123456789000000e+08	1.068110282041773e-07	x
		1.200000000000000e-02	1.345000000000000e-07	8.921933085501859e+04	y
		3.175615690872780e-04	1.000000000000000e+01	3.175615690872780e-05	z
	2	1.199946644984185e+01	1.123456789000000e+08	1.068084377372688e-07	x
		1.440000000000000e-02	1.345000000000000e-07	1.070631970260223e+05	y
		5.786354667000580e-04	1.000000000000000e+01	5.786354667000580e-05	z
	3	1.199911721521620e+01	1.123456789000000e+08	1.068053291653231e-07	x
		1.728000000000000e-02	1.345000000000000e-07	1.284758364312268e+05	y
		8.919241654670848e-04	1.000000000000000e+01	8.919241654670849e-05	z
2	0	1.199911721521620e+01	1.123456789000000e+08	1.068053291653231e-07	x
		1.728000000000000e-02	1.345000000000000e-07	1.284758364312268e+05	y
		8.919241654670848e-04	1.000000000000000e+01	8.919241654670849e-05	z
	1	1.199869813173144e+01	1.123456789000000e+08	1.068015988617738e-07	x
		2.073600000000000e-02	1.345000000000000e-07	1.541710037174721e+05	y
		1.267870635918494e-03	1.000000000000000e+01	1.267870635918494e-04	z

Figure 11.8 HTML output from diagnostic tool `analyze_iteration_variable_magnitudes`.

11.1.7. Non-zero Residuals and Their Dependencies

This tool lists non-zero residuals sorted by magnitude, along with their affecting iteration variables. If also given the optional argument `log_viewer` the residuals are sorted according to the magnitudes of their scaled values. The number of non-zero residuals that is listed can be controlled through the input keyword `n`. We demonstrate this using an instance of `oct.steadystate.nlesol.Problem` denoted by `problem`, and an instance of `oct.steadystate.nlesol.LogViewer` denoted by `lv`, as well as the optional parameters `solve_index` and `iteration`.

```
from oct.steadystate.diagnostics.non_zero_residuals import non_zero_residuals
non_zero_residuals(problem, lv, solve_index=1, iteration=4)
```

This generates the results in Figure 11.9.

Non-zero residuals

Residual values

Unscaled	Scaled	Index	Description
-115.691828626	-0.000271324009619	1	▶ `0 = 120 * x - 75 * (0.12 * exp(12581 * (T - 298) /`

Affecting IVs	Value	Index	Name
	347.70258622	0	T
	0.998889591422	1	x

-0.0143293912432	-2.98528984234e-06	0	`0 = (- x) * (873 - T) + 11.0 * (T - 300)`

Affecting IVs	Value	Index	Name
	347.70258622	0	T
	0.998889591422	1	x

Figure 11.9 HTML output from the diagnostic tool `non_zero_residuals`.

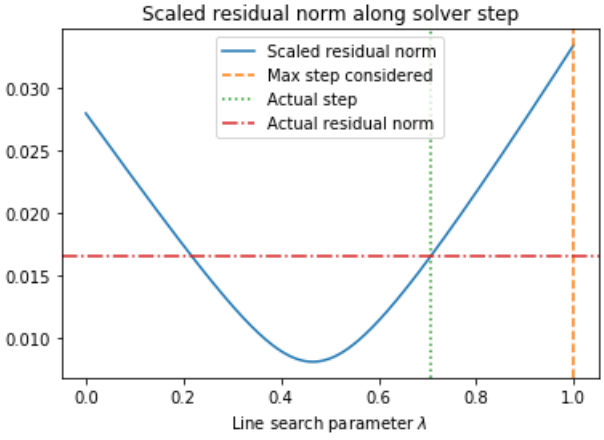
11.1.8. Line-search Plot

It is possible to perform line-search diagnostics at a specific iteration during the solve using the diagnostic script `line_search_plot`. It can help to understand how the norm of the residuals behaves along either the Newton step without taking iteration variable bounds into account or along the projected step. For a given instance of `oct.steadystate.nlesol.Problem` denoted by `problem`, and `lv` an instance of `oct.steadystate.nlesol.LogViewer`, we write:

```
from oct.steadystate.diagnostics.line_search_plot import line_search_plot
```

```
line_search_plot(problem, log_viewer, solve_index=1, iteration=25,
                 step_type="projected", scaled_residuals=True)
```

This generates the results in Figure 11.10.



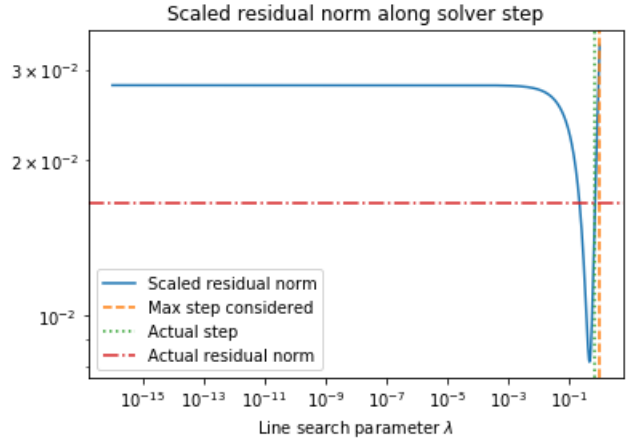
Line search analysis

λ_{max}	1.0
λ_{actual}	0.705841394003
$\lvert Step_{Newton} \rvert$	0.38376119473
$\lvert Step_{projected} \rvert$	0.38376119473

Figure 11.10 A plot generated by using the diagnostic tool `line_search_plot`.

We can also plot using a logarithmic grid, shown in Figure 11.11.

```
line_search_plot(problem, log_viewer, solve_index=1, iteration=25,
                 step_type="projected", scaled_residuals=True,
                 lambda_axis_log=True)
```



Line search analysis

λ_{max}	1.0
λ_{actual}	0.705841394003
$\lvert Step_{Newton} \rvert$	0.38376119473
$\lvert Step_{projected} \rvert$	0.38376119473

Figure 11.11 A plot with similar input as the Figure shown in Figure 11.10 but using a logarithmic grid.

11.1.9. Solver Report

The diagnostic tool `generate_nle_solver_report` gives a brief summary of different properties of a given problem such as max, min, nominal values, convergence information, variable names, non-zero residuals and more. The amount of information to display is controlled through the keyword `mode`. The diagnostic tool will also display information about occured errors and warnings if the FMU is loaded with a minimium log level "error". For a given instance of `oct.steadystate.nlesol.Problem` denoted by `problem` and an instance of `oct.steadystate.nlesol.LogViewer` denoted by `lv`, the diagnostics is displayed by writing:

```
from oct.steadystate.diagnostics.nle_solver_report import generate_nle_solver_report
generate_nle_solver_report(lv, problem, mode="convergence")
```

The results are shown in Figure 11.12, note that `mode="all"` is the default option.

Solver Report

Iteration variables convergence information

Index	Start	End	IV scaled	Residual	Name
0	0.00000e+00	7.07107e-01	nan	-2.91308e-04	y
1	1.50000e+00	1.87075e+00	nan	-2.91767e-04	x

Non-zero residuals

Residual values

Unscaled	Scaled	Index	Description
-0.000291766546438	-4.16844146594e-05	1	▶ x ^ 2 + y ^ 2 = 4 annotation(__Modelon(ResidualEqu

Affecting IVs	Value	Index	Name
	1.8707507751	1	x
	0.707106619194	0	y

Unscaled	Scaled	Index	Description
-0.0002913083617	-4.16189542327e-05	0	▶ x ^ 2 - y ^ 2 = 3 annotation(__Modelon(ResidualEqu

Affecting IVs	Value	Index	Name
	1.8707507751	1	x
	0.707106619194	0	y

Figure 11.12 HTML output from diagnostic tool `generate_nle_solver_report` when using `mode="convergence"`.

Chapter 12. FMU Aggregation Interface for MATLAB®

12.1. Introduction

The primary goal of the FMU aggregation framework is to enable coupled simulation of steady state models represented as interactive FMUs within the Non-linear equation solver MATLAB® interface framework (see Chapter 9). This is achieved by providing a package that contains an external API to a set of connected FMUs that mimics the FMI Toolbox for MATLAB® mex-interfaces for a single FMU.

The FMU aggregation support covers:

- Functionalities necessary to combine a number of FMUs under a single class interface including variable name and value references mangling as well as data routing from the aggregate to individual FMUs and back.
- Support for top-level variables that are aliased to one or several variables of the coupled FMUs.
- Automatic identification of the FMU execution order in the coupled model.
- Support for the debug logging when running simulation with aggregate model.

Current version of the framework has following limitations:

- There is no support for aggregation of dynamics models. In particular ODE state variables and derivatives as well as potential dependencies between them are not supported.
- Only FMI 1.0 Model exchange FMUs are supported.
- The framework does not support “real” algebraic loops over FMUs (see FMI 2.0 specification Section 3.1 for the definition). Artificial algebraic loops are fully supported.
- Coupling of discrete signals (integer and Boolean) is supported but not recommended between FMUs due to limited support in FMI 1.0 specification.
- Top level variables are only supported for inputs and outputs of the aggregated FMUs.

12.2. The MATLAB® interface

FMU aggregation is supported by the `oct.cosimulation` package. The package consists of a number of public classes and functions, as well as a few internal classes and scripts used by the data propagation infrastructure.

The main class is `CoupledFMUModelME1`, which has been implemented to mimic the public interface of `FMUModelME1` class from the FMI Toolbox for Matlab (FMIT). Separate versions of the auxiliary classes `VariableList` and `ScalarVariable1` are also provided as `CoupledScalarVariable1` and `CoupledVariableList`. By retaining the public interfaces these classes are directly usable with other classes in OCT, e.g., `FMUProblem`, `Solver` and `LogViewer`. The classes should also enable easy transition of all the scripts utilizing FMIT mex interfaces to access steady state models from single FMU to an aggregated model.

Below a short overview of the package is given. For details please consult the interactive help in Matlab using the `help` and `doc` commands

`CoupledFMUModelME1` provides an API to access an aggregated model composed of several coupled FMUs. To setup an aggregated model one needs to supply information about FMUs and connections between them. Specification of connections is done at scalar signal level where output of an FMU is connected to one or several inputs on other FMUs. Such flat connection specification may be very long for a large model. In such cases utility function `createFMUConnectionMap` may be utilized.

The `createFMUConnectionMap` function generates a flat connection map based on supplied rules and actual signals available on the FMUs. There are two ways to instruct the algorithm: either look for input-output pairs according to a predefined pattern, or supply the pattern.

The top-level variables are the ones associated with the aggregated model as a whole and not individual FMUs. In essence, the top-level variables are aliases that may point to one or more actual input variables on one or several FMUs, or to an output variable. Top-level variables are generated through the `createTopLevelDefinitions` function .

An alternative to explicitly calling `CoupledFMUModelME1` constructor is to utilize `loadCoupledFMU` function. The function mirrors `loadFMU`, but takes the same arguments as the `CoupledFMUModelME1` constructor. The function returns a `CoupledFMUModelME1` instance on which `fmiInstantiateModel` has been called.

FMU aggregation requires special handling when addressing variables using variable names and value references. Each contained FMU receives an unique instance name. Variables within FMUs are then addressed by prepending their name with the FMU instance name, i.e., `FMUInstanceName.variableName`.

For value references the problem is solved by encoding additional information about FMU instances together with the variable reference within a specific FMU. A few extra public methods are added to work with this intermediate level, e.g., `getFMUInstanceName`, `getFMU`, `getFMUId`, `translateCoupledName`, `translateCoupledVariableRef` and `translateRefToCoupled`. It is possible to use these to address individual FMUs within `CoupledFMUModelME1`. The recommended and more convenient usage is through the auxiliary class `CoupledVariableList` that handles underlying conversion automatically.

`CoupledScalarVariable1` replaces `ScalarVariable1` class from FMIT when working with the aggregated model. Instances of this class are returned from a number of functions, most notably `getModelVariables`. The class provides access to all the scalar variable attributes as defined in FMI specification, except for the `directDependency`, which is not supported.

`CoupledVariableList` replaces `VariableList`, and supports both variables from the different sub-models and top level variables.

12.3. Examples

12.3.1. Simple example for predefined connection map algorithm

Below an example is presented where the connection graph is generated based on the default mapping algorithm. See interactive help on `createFMUConnectionMap` for more details.

```
% Load FMUs and create units argument
fmuA = loadFMU('fmuAFileName', 'LOGNAME', 'A');
fmuB = loadFMU('fmuBFileName', 'LOGNAME', 'B');
units = {{fmuA, 'A'}, {fmuB, 'B'}};
% Create connection map according to predefined pattern
connectionMap2FMUs = createFMUConnectionMap(units, 'fmu2fmuBuses');
% Create CoupledFMU.
CoupledFMU = CoupledFMUModelME1(units, connectionMap2FMUs);
% Construct FMUProblem, Solver and solve the non-linear system.
fmuProblem = oct.nlesol.FMUProblem(coupledFMU);
solver = oct.nlesol.Solver(fmuProblem);
solution = solver.solve();
```

12.3.2. Simple example for user supplied rules for connection map algorithm

Below an example is presented where the rules for generating the connection graph are altered by providing buses argument.

```
% Load FMU and set up units
fmuA = loadFMU('fmuAFileName', 'LOGNAME', 'A');
fmuB = loadFMU('fmuBFileName', 'LOGNAME', 'B');
units = {{fmuA, 'A'}, {fmuB, 'B'}};
% Set up buses
buses = {{'A', 'B', 'A_outs', 'B_ins'}};
% The connection graph algorithm will now look for
% outputs named according to the pattern 'A_outs.RESTOFNAME'
% on A and connect them to input variables named according
% to the pattern 'B_ins.RESTOFNAME' on B.

connectionMap2FMUs = createFMUConnectionMap(units, 'causalBuses', buses);
% Create CoupledFMU.
CoupledFMU = CoupledFMUModelME1(units, connectionMap2FMUs);
% Construct FMUProblem, Solver and solve the non-linear system.
fmuProblem = oct.nlesol.FMUProblem(coupledFMU);
solver = oct.nlesol.Solver(fmuProblem);
solution = solver.solve();
```

12.3.3. Complete example including models

The Modelica models used in the example

```
within ;
package TestUnpaired

  connector RealSignal = Real;

model Splitter
  RealSignal x;
  parameter Real k = 1.0;
  parameter Real factor_guess = 0.5;
  Real factor(min = 0, max = 1, start = factor_guess)
           annotation(__Modelon(IterationVariable(enabled = true)));
  RealSignal x1 = factor * k *  x;
  RealSignal x2 = (1-factor) * k* x;
end Splitter;

model Mixer
  RealSignal  x1;
  RealSignal  x2;
  RealSignal  y;
equation
  0 = x1 - x2 annotation(__Modelon(ResidualEquation, name=dx));
  y = x1;
end Mixer;

model SplitterSeparate
  input RealSignal x(start = 1.0);
  output RealSignal x1;
  output RealSignal x2;
  Splitter splitter;

  input Real extraIV;
equation

  x1 = extraIV annotation(__Modelon(ResidualEquation, name=extraDV));

  connect(x, splitter.x);
  connect(splitter.x1, x1);
  connect(splitter.x2, x2);

end SplitterSeparate;

model MixerSeparate
  output RealSignal x1;

  input RealSignal x2;
  output RealSignal y;
  Mixer mixer;
```

```
  Real extraIV annotation(__Modelon(IterationVariable));
equation
  x1 = extraIV;
  connect(x1, mixer.x1);
  connect(x2, mixer.x2);
  connect(y, mixer.y);
end MixerSeparate;

model SimpleComposit
  input RealSignal x( start = 1.0);
  output RealSignal y;
  Splitter splitter;
  Mixer mixer;
equation
  connect(x, splitter.x);
  connect(splitter.x1, mixer.x1);
  connect(splitter.x2, mixer.x2);
  connect(y, mixer.y);
end SimpleComposit;

model SplitterSeparate
  input RealSignal x(start = 1.0);
  output RealSignal x1;
  output RealSignal x2;
  Splitter splitter;

  input Real extraIV;
equation

  x1 = extraIV annotation(__Modelon(ResidualEquation, name=extraDV));

  connect(x, splitter.x);
  connect(splitter.x1, x1);
  connect(splitter.x2, x2);

end SplitterSeparate;

  annotation (uses(Modelica(version="3.2.1")));
end TestUnpaired;
```

Code that compiles the partitioned models and compares solution results from a monolithic model

```
splitterModel = 'TestUnpaired.SplitterSeparate';
mixerModel = 'TestUnpaired.MixerSeparate';
combinedModel = 'TestUnpaired.SimpleComposit';

compiler = 'OCT_Modelica';
lib = {[testdir, '\TestUnpaired.mo']};
opt = {'interactive_fmu', true,'expose_scalar_equation_blocks_in_interactive_fmu',
 true, ...
```

```
             'hand_guided_tearing', true, 'merge_blt_blocks', true};
splitterFN = oct.modelica.compileFMU(splitterModel, compiler, 'modelPath', lib, ...
             'options', opt);
mixerFN = oct.modelica.compileFMU(mixerModel, compiler, 'modelPath', lib, ...
            'options', opt);
combinedFN = oct.modelica.compileFMU(combinedModel, compiler, 'modelPath', lib, ...
            'options', opt);

combinedFmu = loadFMU(combinedFN);
combinedP = oct.nlesol.FMUProblem(combinedFmu);
combinedS = oct.nlesol.Solver(combinedP);
sol = combinedS.solve;

mixerFmu = loadFMU(mixerFN);
splitterFmu = loadFMU(splitterFN);
% Set up CoupledFMUModelME1 with a manual created connection map.
coupledFmu = oct.cosimulation.CoupledFMUModelME1(...
    { {splitterFmu, 's',}, {mixerFmu, 'm'} }, ...
    {'s', 'm', 'x2', 'x2'; 'm','s', 'x1', 'extraIV'} ...
    );
coupledP = oct.nlesol.FMUProblem(coupledFmu);
coupledS = oct.nlesol.Solver(coupledP);
sol1 = coupledS.solve;
factor = coupledFmu.getValue('s.splitter.factor');
relativeDiff = (sol - factor)./((abs(sol + factor) + 1e-16)/2);

if (all(abs(relativeDiff) < 1e-6))
  disp('Success');
else
  disp('Failure');
end
```

Chapter 13. Graphical User Interface for Visualization of Results

13.1. Plot GUI

OCT comes with a graphical user interface (GUI) for displaying simulation and / or optimization results. The GUI supports result files generated by OCT and Dymola (both binary and textual formats).

The GUI is located in the module `(pyjmi/pyfmi).common.plotting.plot_gui` and can be started by Windows users by selecting the shortcut located in the start-menu under OCT. The GUI can also be started by typing the following commands in a Python shell:

```
from pyfmi.common.plotting import plot_gui   # or pyfmi.common.plotting import plot_gui
plot_gui.startGUI()
```

Note that the GUI requires the Python package wxPython which is installed by default in the Python version that comes with OCT.

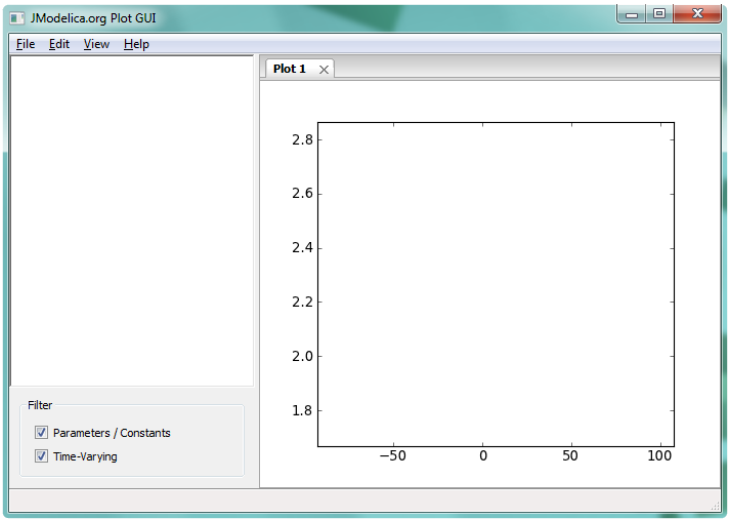


Figure 13.1 Overview of OCT Plot GUI

13.1.1. Introduction

An overview of the GUI is shown in Figure 13.1. As can be seen, the plot figures are located to the right and can contain multiple figures in various configurations. The left is dedicated to show the loaded result file(s) and corresponding variables together with options for filtering time-varying variables and parameters/constants.

Loading a result file is done using the `File` menu selection `Open` which opens a file dialog where either textual (.txt) results or binary (.mat) results can be chosen. The result is then loaded into a tree structure which enables the user to easily browse between components in a model, see Figure 13.2 . Multiple results can be loaded either simultaneously or separately by using the `File` menu option `Open` repeatedly.

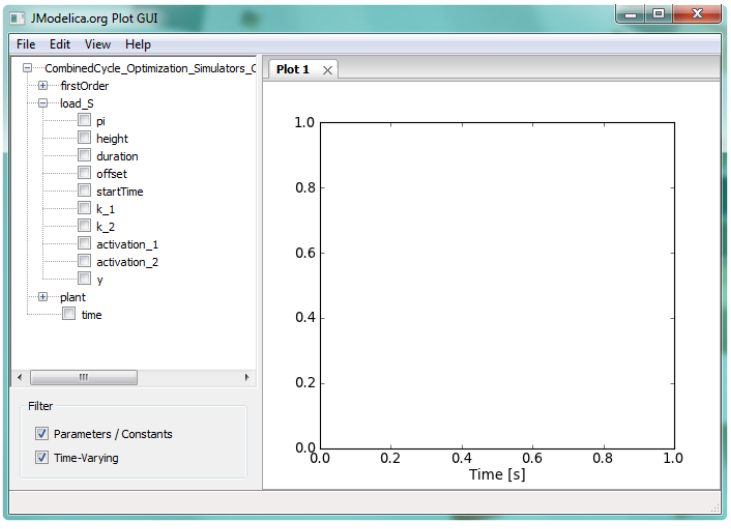


Figure 13.2 A result file has been loaded.

Displaying trajectories is done by simply checking the box associated with the variable of interest, see Figure 13.3. Removing a trajectory follows the same principle.

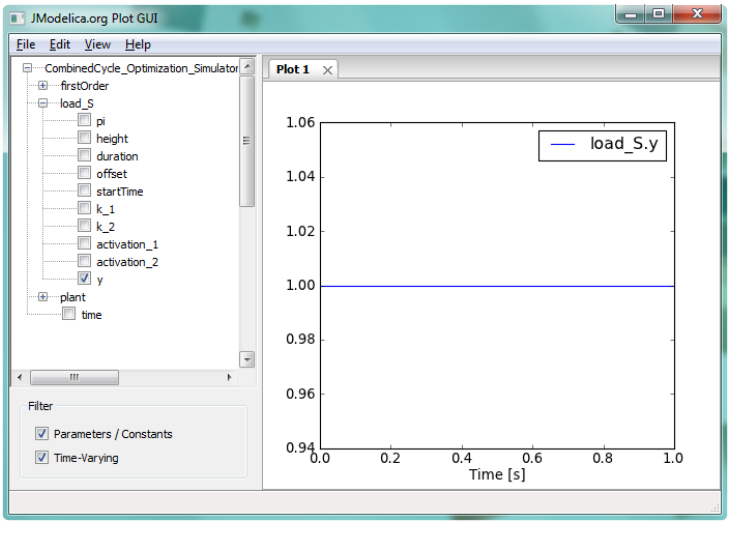


Figure 13.3 Plotting a trajectory.

A result can also be removed from the tree view by selecting an item in the tree and by pressing the delete key.

13.1.2. Edit Options

The GUI allows a range of options, see Figure 13.4, related to how the trajectories are displayed such as line width, color and draw style. Information about a plot can in addition be defined by setting titles and labels. Options related to the figure can be found under the `Edit` menu as well as adding more plot figures.

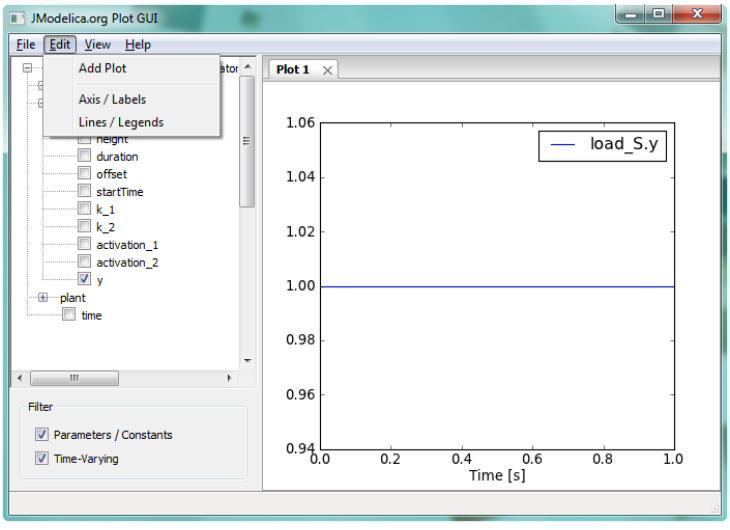


Figure 13.4 Figure Options.

Under `Axis/Labels`, see Figure 13.5, options such as defining titles and labels in both X and Y direction can be found together with axis options.

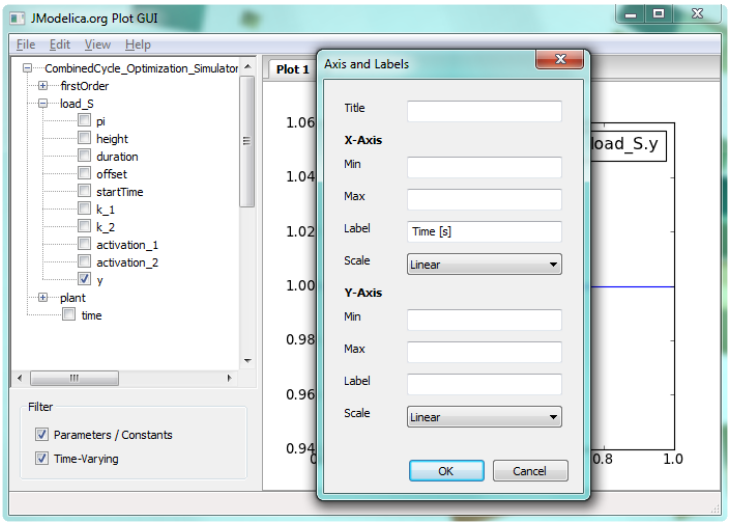


Figure 13.5 Figure Axis and Labels Options.

Under `Lines/Legends`, options for specifying specific line labels and line styles can be found, see Figure 13.6. The top drop-down list contains all variables related to the highlighted figure and the following input fields down to `Legend` are related to the chosen variable. The changes take effect after the button `OK` has been pressed. For changing multiple lines in the same session, the `Apply` button should be used.

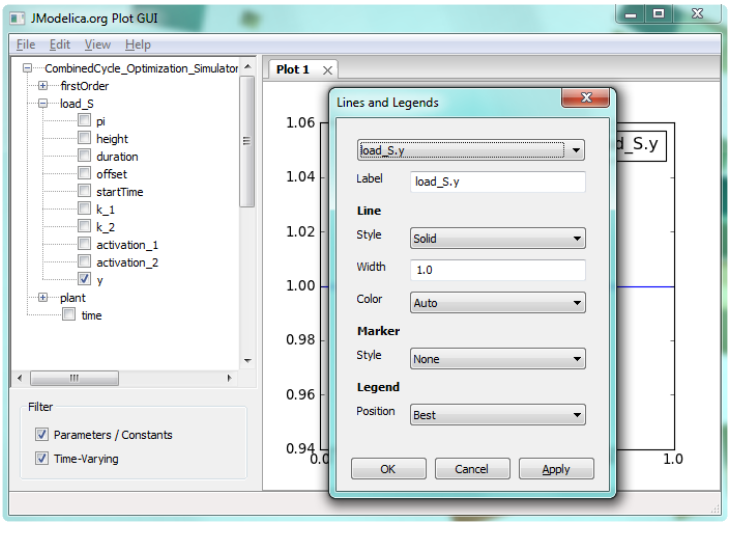


Figure 13.6 Figure Lines and Legends options.

Additional figures can be added from the `Add Plot` command in the `Edit` menu. In Figure 13.7 an additional figure have been added.

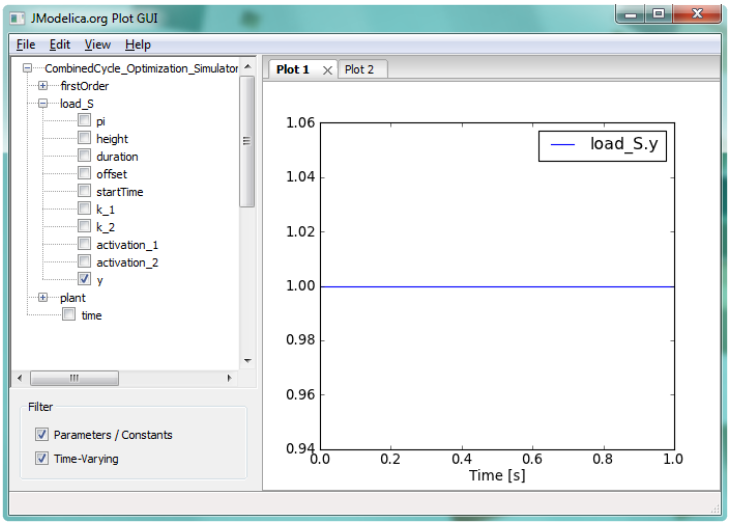


Figure 13.7 An additional plot has been added.

The figures can be positioned by choosing a figure tab and moving it to one of the borders of the GUI. In Figure 13.8 "Plot 1" have been dragged to the left side of the figure and a highlighted area has emerged which shows where "Plot 1" will be positioned. In Figure 13.9 the result is shown.

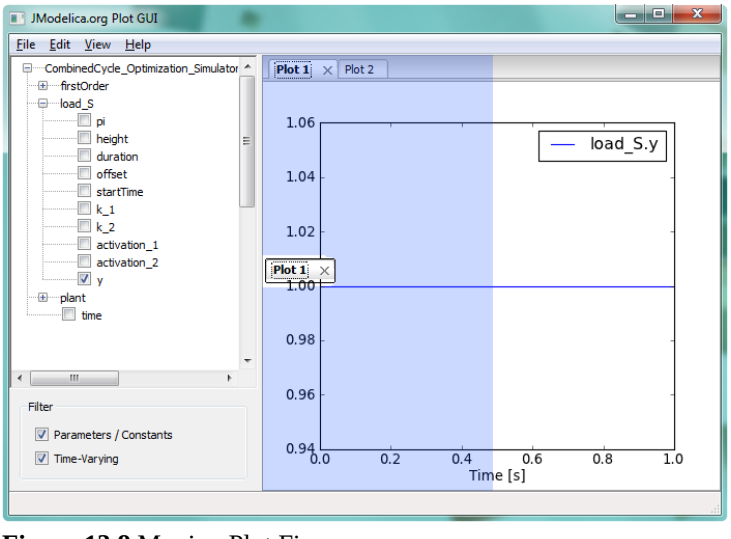


Figure 13.8 Moving Plot Figure.

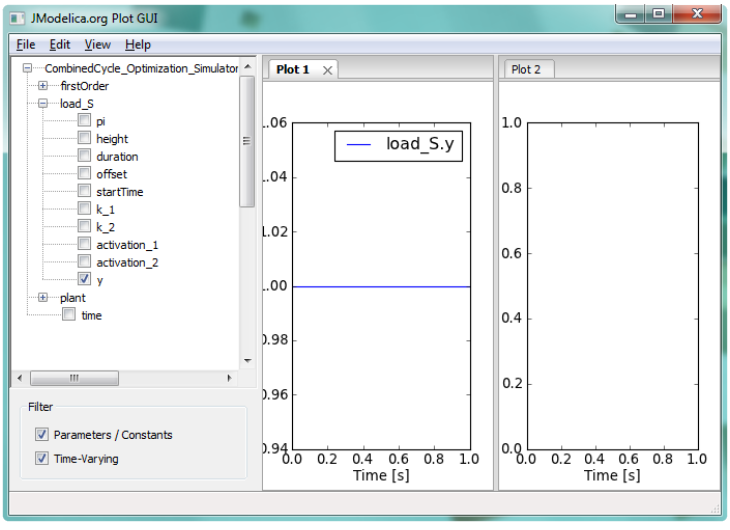


Figure 13.9 GUI after moving the plot window.

If we are to add more figures, an increasingly complex figure layout can be created as is shown in Figure 13.10 where figures also have been dragged to other figure headers.

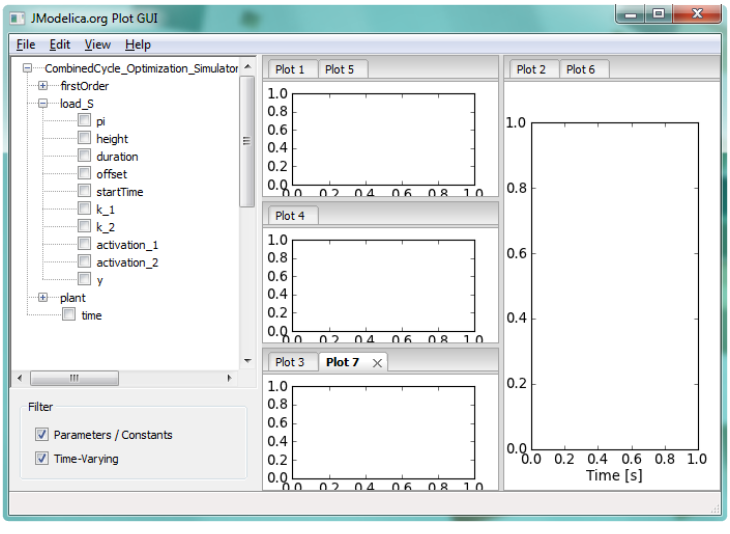


Figure 13.10 Complex Figure Layout.

13.1.3. View Options

Options for interacting with a figure and changing the display can be found under the `View` menu. The options are to show/hide a grid, either to use the mouse to move the plot or to use the mouse for zooming and finally to resize the plot to fit the selected variables.

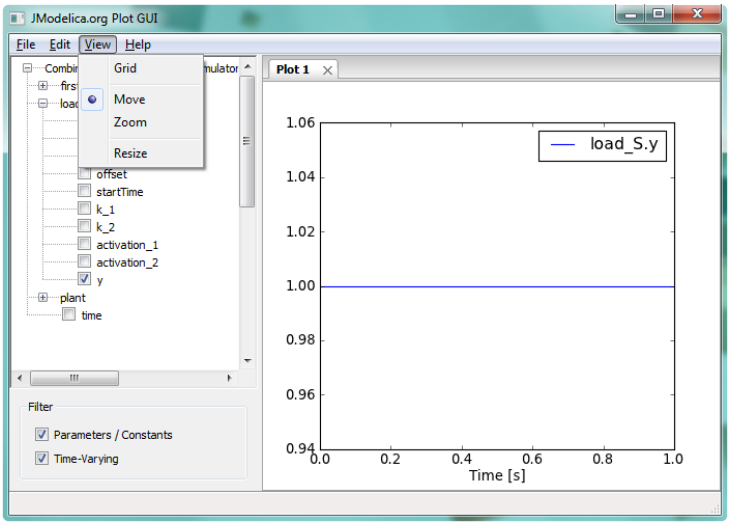


Figure 13.11 Figure View Options.

Moving a figure with the `move` option is performed by simply pressing the left mouse button and while still holding down the button, dragging the plot to the area of interest. A zoom operation is performed in a similar fashion.

13.1.4. Example

Figure 13.12 shows an example of how the GUI can be used to plot four different plots with different labels. Some of the lines have also been modified in width and in line style. A grid is also shown.

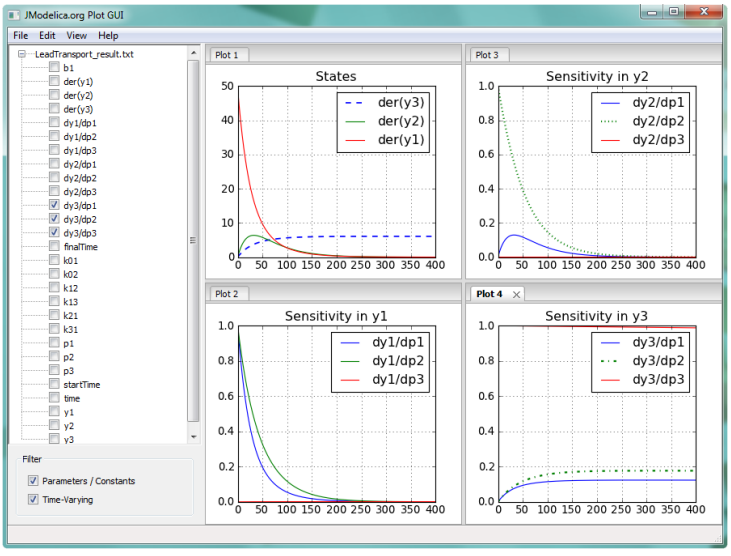


Figure 13.12 Multiple figure example.

Chapter 14. Steady-state Modelica Modeling with Hand Guided Tearing

In the Optimica Compiler Toolkit there are extensions to the Modelica language which gives the possibilities to utilize hand guided tearing. With hand guided tearing the user can specify certain variables and equations which should be chosen as iteration variables and residuals respectively. Normally this choice is made automatically by the compiler. In this chapter the syntax and the method will be explained.

14.1. Specification of Hand Guided Tearing

There are two ways to use hand guided tearing in OCT:

- As pairing where an equation is bound to a variable
- As unpaired variables and equations where pairs are bound by the compiler

14.1.1. Identification of Equations

In some situations, it is necessary to identify an equation so that it can be referenced.

Syntax

It is possible to place annotations for equation name in the annotation block for the equation.

```
"annotation" "("
  "__Modelon" "("
    "name" "=" IDENT
  ")"
")"
```

Example

```
x = y + 1 annotation(__Modelon(name=res));
```

14.1.2. Paired Tearing

In some situations it is crucial that an equation and a variable form a tearing pair. This is where the hand guided tearing pair annotations comes into play. It allows the user to specify exactly which tearing pairs to form. The tearing pairs that are specified are torn before any automatic tearing comes into play. The pairs are also torn without any regard for solvability of the system. This means that if the user specifies to many pairs, they will all be used and the torn block becomes unnecessarily complex. If the final system is unsolvable after all pairs are torn, the automatic algorithm will kick in and finalize the tearing.

There are two ways to specify hand guided tearing pairs.

- On component level
- On system level

14.1.2.1. Specify Tearing Pairs on Component Level

Tearing pairs can be specified in the annotation for the equation that should become residual equation. This type of hand guided tearing is limited to the name scope that is visible from the equation. In other words, the equation has to be able to "see" the variable that should be used as iteration variable.

Syntax

It is possible to place annotations for tearing pairs in the annotation block for the residual equation. The syntax for tearing pair on component level has the following syntax:

```
"annotation" "("
  "__Modelon" "("
    ResidualEquation
  ")"
")"
```

Where `ResidualEquation` is defined by the following record declaration:

```
record ResidualEquation
  parameter Boolean enabled = true;
  parameter Integer level(min=1) = 1;
  Real nominal = 1;
  IterationVariable iterationVariable;
  parameter Boolean hold = false;
end ResidualEquation;
```

Where `IterationVariable` is defined by the following record declaration:

```
record IterationVariable
  parameter Real name; // Accessed without dot-notation
  Real max;
  Real min;
  Real nominal;
  Real start;
  parameter Boolean hold = false;
end IterationVariable;
```

Example

```
model A
  ...
```

```
  parameter Boolean subSystem1Hold = true;
  Real z;
  Real q;
  ...
equation
  ...
  x = y + 1 annotation(__Modelon(ResidualEquation(iterationVariable=z)));

  p = x + q annotation(__Modelon(ResidualEquation(
                                   iterationVariable(
                                     start = 1,
                                     nominal = 10,
                                     hold = subSystem1Hold
                                   ) = q,
                                   nominal = 100,
                                   hold = subSystem1Hold,
                                 )));
  ...
end A;
```

14.1.2.2. Specify Tearing Pairs on System Level

Tearing pairs on system level are necessary when the residual equation and iteration variable are located in different name scopes. In other words, the equation can not "see" the iteration variable.

Before it is possible to specify tearing pairs on system level it is necessary to define a way to identify equations.

Syntax

It is possible to place annotations for tearing pairs on system level in the annotation block for the class deceleration.

```
"annotation" "("
  "__Modelon" "("
    "tearingPairs" "(" Pair* ")"
  ")"
")"
```

Where `Pair` is defined by the following record declaration:

```
record Pair
  parameter Boolean enabled = true;
  parameter Integer level(min=1) = 1;
  ResidualEquation residualEquation;
  IterationVariable iterationVariable;
end Pair;
```

Where ResidualEquation is defined by the following record declaration:

```
record ResidualEquation
```

```
  parameter Equation name; // Accessed without dot-notation
  Real nominal = 1;
  parameter Boolean hold = false;
end ResidualEquation;
```

Where IterationVariable is equal to the record declaration with the same name in section Section 14.1.2.1:

Example

Here follows an example where the equation is identified by a name tag and then paired with a variable.

```
model A
  model B
    ...
    x = y + 1 annotation(__Modelon(name=res));

    p = x + q annotation(__Modelon(name=res2));
    ...
  end B;
  model C
    ...
    Real z;
    Real q;
    ...
  end C;
  parameter Boolean subSystem2Hold = true;
  B b;
  C c;
  ...
  annotation(__Modelon(tearingPairs(Pair(residualEquation=b.res,iterationVariable=c.z))));

  annotation(__Modelon(tearingPairs(Pair(
                         residualEquation(
                           nominal = 10,
                           hold    = subSystem2Hold
                         )  = b.res2,
                         iterationVariable(
                           start = 2,
                           hold    = subSystem2Hold
                         ) = c.q
                       ))));
end A;
```

14.1.3. Unpaired Tearing

It is also possible to specify that an equation or variable should be used in tearing without pairing. This is useful when there is no requirement that a certain equation is bound to a specific variable. The pairing is instead done by the compiler during compilation. An error is given If the number of unpaired equations is unequal to the number of unpaired variables.

14.1.3.1. Specify an Equation as Unpaired Residual Equation

By marking an equation as unpaired residual equation it will be paired to an unpaired iteration variable during tearing.

Syntax

It is possible to place annotations for residual equations in the annotation block for an equation. The syntax for residual equation annotation has the following syntax:

```
"annotation" "("
  "__Modelon" "("
    ResidualEquation
  ")"
")"
```

Where `ResidualEquation` is equal to that of component level pairs (see section Section 14.1.2.1) with one exception; the `iterationVariable` field is left unspecified.

Example

```
x = y + 1 annotation(__Modelon(ResidualEquation));
```

14.1.3.2. Specify a Variable as Unpaired Iteration Variable

By marking a variable as unpaired iteration variable it will be paired to an unpaired residual equation during tearing.

Syntax

It is possible to place annotations for unpaired iteration variable in the annotation block for a variable. The iteration variable annotation has the following syntax:

```
"annotation" "("
  "__Modelon" "("
    IterationVariable
  ")"
")"
```

Where `IterationVariable` is defined by the following record declaration:

```
record IterationVariable
  parameter Boolean enabled = true;
  parameter Integer level(min=1) = 1;
  Real max;
  Real min;
  Real nominal;
  Real start;
```

```
  parameter Boolean hold = false;
end IterationVariable;
```

Example

```
Real x annotation(__Modelon(IterationVariable));
```

14.2. Nested Hand Guided Tearing

The record definitions `ResidualEquation` and `IterationVariable` in section Section 14.1 have the field declaration `level`. This field specifies on which level the equation or variable should be torn. Equations and variables with the same level will be torn into the same torn block. It is possible to have nested torn blocks by specifying different levels for different equations and variables.

14.3. Hand Guided Tearing Attributes

The record definitions `ResidualEquation` and `IterationVariable` in section Section 14.1 have the field declarations `max`, `min`, `start` and `nominal`. These fields are optional. If left unspecified, the value is retrieved from the corresponding attributes in the variable declaration specified by the `name` field. It is possible to use continuous variables in the expressions for the fields `max`, `min`, `nominal`, `start` and equation `nominal` with two restrictions; the variable must be solved before the computation of the equation block start and the block must be torn on HGT `level` two or greater.

14.4. Extended Example

14.4.1. NHGT with fixed bounds

Consider the following two-dimensional nonlinear algebraic test problem. It depends on two unknowns x1 and x2. The first residual equation is

$$0 = \frac{3\exp\left(-\frac{1}{r+0.1}\right)}{r+0.1} - \frac{1}{r+2}$$

It involves a sub-expression *r*.

$$r = \frac{x1^2}{100} + \frac{x2^2}{10}$$

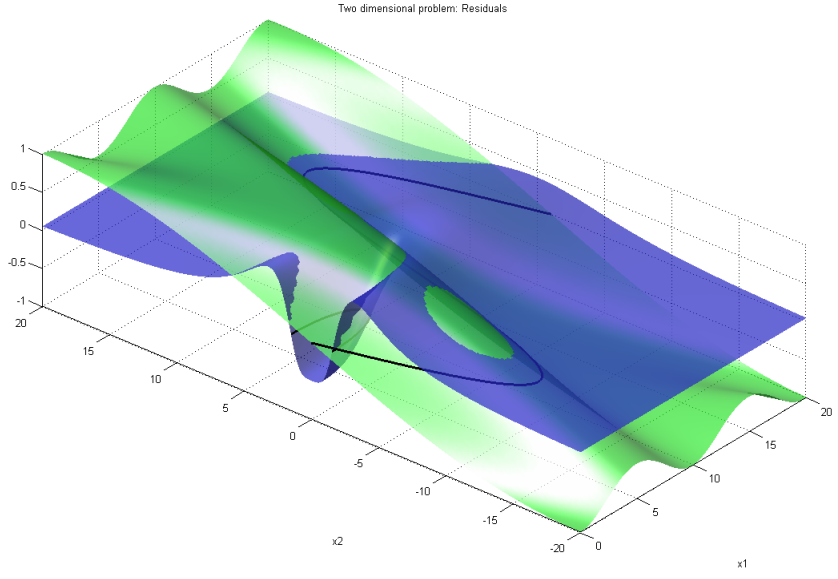
The second residual equation is

$$0 = \sin(a+b)$$

Again, sub-expressions are used.

$$a = \frac{\pi x2}{40},\ b = \sin\left(\frac{x1}{\pi}\right)$$

The shape of the residuals is as follows (the first residual is blue, the second green, and black lines illustrate the intersection with the zero plane). Note how the blue residual is non-convex. This means that a gradient-based algorithm will have difficulties to reach the solution inside the “valley” for small x1 values unless the algorithm is started inside (i.e., the start attributes correspond to an iterate inside the “valley”).

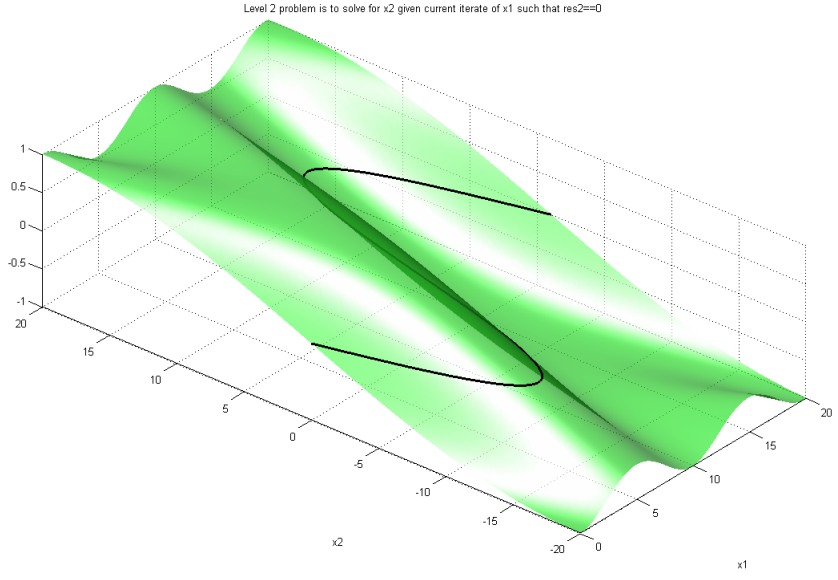


The Modelica code for such a model can be written as follows. This model is included in the ExampleModels.mo-package and the script to run this example is example_NHGT.

```
model NonConvex
  import Modelica.Math.*;
  import Modelica.Constants.pi;
  Real r;
  Real a;
  Real b;
  Real x1(min=0, max=20, start=15);
  Real x2(min=-20, max=20, start=15);
equation
  r = x1^2/100+x2^2/10;
  0 = 3*exp(-1/(r+0.1))/(r+0.1)-1/(r+2);
  a = pi*x2/40;
  b = sin(x1/pi);
  0 = sin(a + b);
```

```
end NonConvex;
```

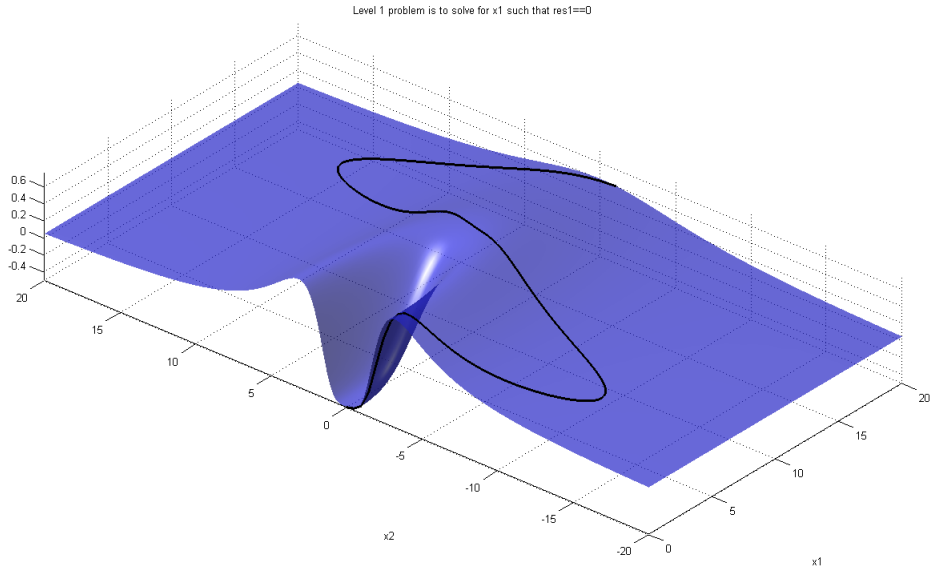
How can we solve such a problem using nested hand-guided tearing, ideally such that the robustness is high? We split the problem into two levels. On the inner level ("level 2"), the solver finds the solution x2 to the second residual based on the current iterate of x1. In other words, the solver iterates are constrained to the following one dimensional manifold (see the black line in the following illustration).



To achieve this, we add a nested hand-guided tearing annotation to the residuals equation as described in the introductory section (see bold text).

```
model NonConvex
  import Modelica.Math.*;
  import Modelica.Constants.pi;
  Real r;
  Real a;
  Real b;
  Real x1(min=0, max=20, start=15);
  Real x2(min=-20, max=20, start=15);
equation
  r = x1^2/100+x2^2/10;
  0 = 3*exp(-1/(r+0.1))/(r+0.1)-1/(r+2);
  a = pi*x2/40;
  b = sin(x1/pi);
  0 = sin(a + b)
    annotation(__Modelon(ResidualEquation(iterationVariable=x2,level=2)));
end NonConvex;
```

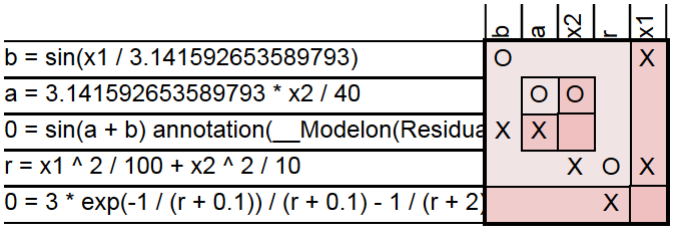
Finally, in the outer “level 1” problem, the solver drives the first residual to zero. This problem can be solved robustly (i.e., without suffering from the non-convex shape of the residual) using a computationally efficient algorithm. The algorithm moves along the same one-dimensional manifold highlighted with a black line in the following illustration. Residual 1 is evaluated along this manifold, and a solution is computed.



This is implemented via the following additional annotation (again, see the bold text). The model with these annotations is also included in the ExampleModels.mo-package.

```
model NonConvexNHGT
  import Modelica.Math.*;
  import Modelica.Constants.pi;
  Real r;
  Real a;
  Real b;
  Real x1(min=0, max=20, start=15);
  Real x2(min=-20, max=20, start=15);
equation
  r = x1^2/100+x2^2/10;
  0 = 3*exp(-1/(r+0.1))/(r+0.1)-1/(r+2)
    annotation(__Modelon(ResidualEquation(iterationVariable=x1,level=1)));
  a = pi*x2/40;
  b = sin(x1/pi);
  0 = sin(a + b)
    annotation(__Modelon(ResidualEquation(iterationVariable=x2,level=2)));
end NonConvexNHGT;
```

In the compilation log files we get the following visualization of the problem structure.



14.4.2. NHGT with adaptive bounds

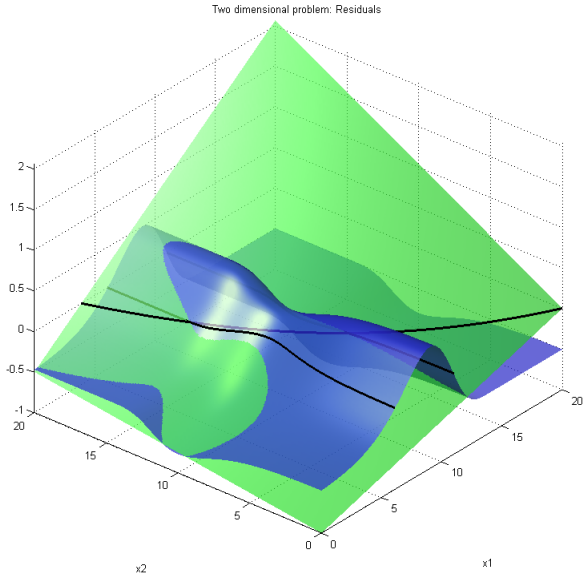
Additionally to allowing to the modeler to specify in what hierarchy to solve an equation system, Nested Hand Guided Tearing allows to adapt Real variable attributes such as the start value. Consider the following two dimensional problem. The residual equations are

$$0 = \exp(-\exp(k) + k + 1) - 0.3\exp(-\exp(l) + l + 1) - 0.5 \qquad 0 = s + \mathrm{x1} \times \frac{\mathrm{x2}}{16^2} - 1$$

The equation system involves three further equations (and unknowns k, l, s), which we will ask the compiler to solve for as needed to express the problem in terms of two residuals. The variable f is a problem-specific parameter.

$$0 = k + \frac{\mathrm{x1} - 9}{-1}, \quad 0 = l + \frac{\mathrm{x2} - f}{-2}, \quad 0 = \frac{\mathrm{x2}}{40} + \frac{\mathrm{x1}}{20} - s$$

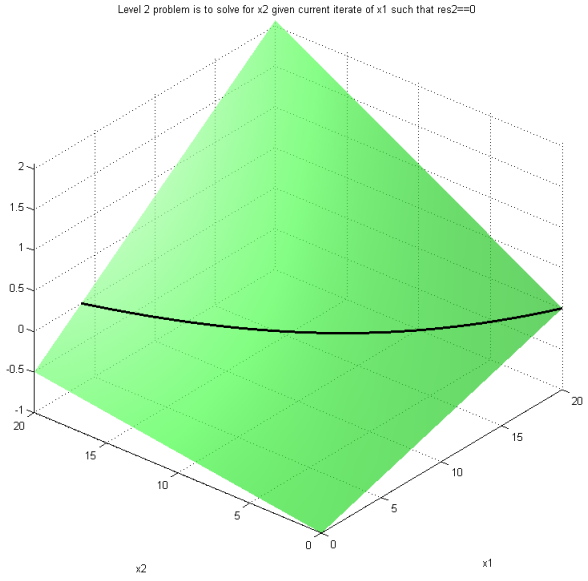
Substituting the three equations as required into the two residual equations, we are able to visualize the problem as follows (again, the blue surface is the first and the green surface the second residual). Note how the blue residual admits two one-dimensional manifolds to satisfy it.



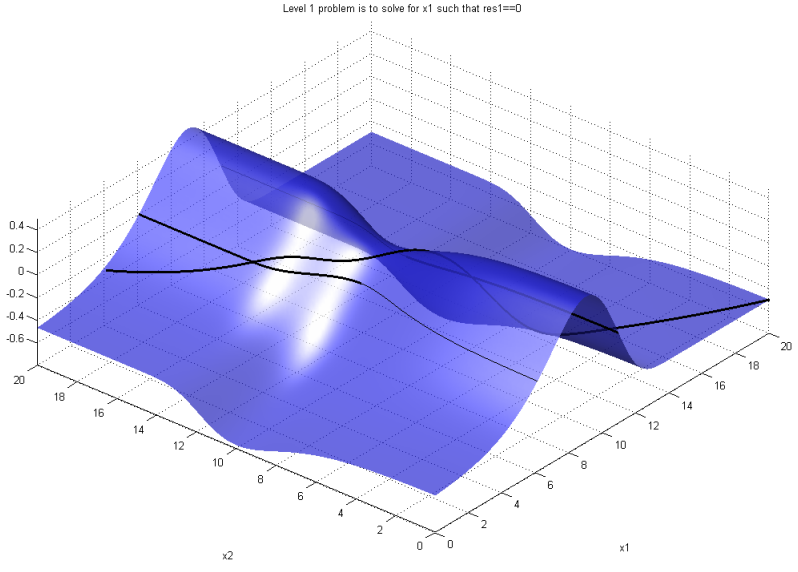
Without Hand Guided Tearing we formulate this problem in Modelica as follows.

```
model Test
  parameter Real f = 10;
  Real x1(min=0, max=20, start=15);
  Real x2(min=0, max=20, start=15);
  Real k;
  Real l;
  Real s;
equation
  0 = k +(x1-9)/(-2);
  0 = l +(x2-f)/(-2);
  0 = x2/40 + x1/20 - s;
  0 = 1 * exp(-exp(k)+k+1.0) - 0.3 * exp(-exp(l)+l+1.0) - 0.5;
  0 = s + x1*x2/16^2 - 1;
end Test;
```

Assume that, based on an understanding of the problem (usually given the physics-based characteristics of it), we are able to infer that solving residual two in a nested fashion is particularly beneficial for robustness or computational efficiency. The “level 2” problem shall be to solve residual two for x2 given the current iterate of x1. In the following illustration, the one-dimensional manifold is shown as a black line.



The “level 1” problem is now to vary variable x1 until the problem is solved. Again, the algorithm moves along the same one-dimensional manifold highlighted with a black line in the following residual 1 illustration.

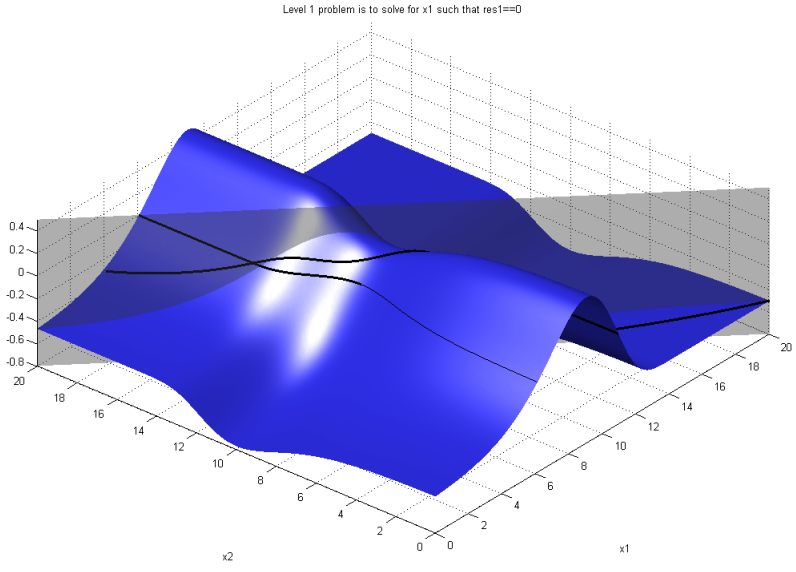


These two different levels can be implemented with the following annotations (see text in bold font).

```
model Test
  parameter Real f = 10;
  Real x1(min=0, max=20, start=15);
  Real x2(min=0, max=20, start=15);
  Real k;
  Real l;
  Real s;
equation
  0 = k +(x1-9)/(-2);
  0 = l +(x2-f)/(-2);
  0 = x2/40 + x1/20 - s;
  0 = 1 * exp(-exp(k)+k+1.0) - 0.3 * exp(-exp(l)+l+1.0) - 0.5
    annotation(__Modelon(ResidualEquation(iterationVariable=x1,level=1)));
  0 = s + x1*x2/16^2 - 1
    annotation(__Modelon(ResidualEquation(iterationVariable=x2,level=2)));
end Test;
```

This problem does however exhibit two different solutions. In engineering, one of the multiple solutions is usually superior if compared to the others. In some cases, the solutions differ in efficiency or a similar metric, in others some of the solutions may even be mathematical artifacts that are not physically possible (because they violate fundamental laws such as the Second Law of Thermodynamics). We assume that a condition can be written to express such a condition, and, based on its value, the attributes of the real iteration variables is adapted.

In this simple example we assume that any solution with $x1 + x2 > 20$ can be excluded based on physical insight. We therefore introduce an adaptive bound on the “level 2” problem by setting the lower bound to $x2$, $\max = 20 - x1$. This condition is illustrated below with the grey transparent surface; all points facing the reader “before the surface” are valid.



These adaptive bounds are implemented as follows.

```
model Test
  parameter Real f = 10;
  Real x1(min=0, max=20, start=15);
  Real x2(min=0, max=20, start=15);
  Real k;
  Real l;
  Real s;
equation
  0 = k +(x1-9)/(-2);
  0 = l +(x2-f)/(-2);
  0 = x2/40 + x1/20 - s;
  0 = 1 * exp(-exp(k)+k+1.0) - 0.3 * exp(-exp(l)+l+1.0) - 0.5
    annotation(__Modelon(ResidualEquation(iterationVariable=x1,level=1)));
  0 = s + x1*x2/16^2 - 1
    annotation(__Modelon(ResidualEquation(iterationVariable(max=20-x1)=x2,level=2)));
end Test;
```

The compilation log again contains a visualization of the equation structure with the BLT table.

	s	x2	_	k	x1
0 = x2 / 40 + x1 / 20 - s	O	O			O
0 = s + x1 * x2 / 256.0 - 1 annotation(__Mode	O	O			O
0 = l + (x2 - f) / -2		X	O		
0 = k + (x1 - 9) / -2				O	X
0 = exp(- exp(k) + k + 1.0) - 0.3 * exp(- exp(l)			X	X	

Chapter 15. Modelica Smoothness Check

Smoothness check introduces a set of checking modes which detects non-smooth Modelica constructs. Some examples of relevant checking modes:

- Steady-state models with discrete switches. Model equations are `C1`, explicitly modeled discrete elements are allowed. Allowed elements:

  1. Equations with discrete constructs such as if expressions
  2. Relation expressions generating events
  3. Algorithmic functions with `smoothOrder` annotation with `order >=1`

  Some elements that would not be allowed in this mode are:

  1. Non-C1 operators which do not generate events such as `min`, `max` or `abs` functions
  2. When clauses in equations and algorithms
  3. Boolean and integer variables

- Steady-state models without discrete switches. Model equations are `C1` without discrete elements. Forbidden elements:

  1. Equations with discrete constructs such as if expressions if they are not in `smooth(1,..)` operators
  2. Relation expressions generating events
  3. Algorithmic functions with `smoothOrder` annotation with `order <1`
  4. Non-C1 operators which do not generate events such as `min`, `max` or `abs` functions
  5. When clauses in equations and algorithms
  6. Boolean and integer variables

- Optimization models `Cn, n>=1` models without events. In most cases for optimization `n>=2` to fulfill the requirements of the numerical algorithms. Allowed elements:

  1. Equations with sufficient smooth order n. Relational expressions are allowed inside `noEvent` and smooth (with sufficiently high level of smoothness) operators

2. Algorithmic functions with `smoothOrder` annotation with `order >=n` Some elements that would not be allowed in this mode are:

3. `Non-Cn, n>=1` functions which do not generate events such as `min`, `max` or `abs` functions

4. When clauses in equations and algorithms

5. Boolean and integer variables

6. Relation expressions outside of `noEvent` and `smooth` (with smooth order not sufficiently high) operators

All of these use cases can be covered by combining and settings the options described in this chapter.

The check is done during normal error checking of the model which means that no variability propagation or other model transformation has been performed. So for example, some variables or expressions that later will be found to be of constant or parameter variability will be reported as discontinuous.

15.1. Options Flags

There are several compiler option flags that control which language constructs that should be allowed.

`allow_discrete_variables`
: Boolean option that controls whether discrete variables such as Integers, Booleans, Strings or Enumerations should be allowed in the model. This option is true by default.

`allow_when_clauses`
: Boolean option that controls whether when clauses should be allowed in the model. This option is true by default.

`allow_discrete_switches`
: Boolean option that controls whether discrete switches should be allowed in the model. This option is true by default.

`system_continuity_order`
: Require the system to be at least n times continuous differentiable where n is an integer given by this option. Default value is -1 which means that discontinuations are allowed.

`smoothness_check_as_warnings`
: Boolean option that controls whether smoothness check problems should be given as warning instead of errors. This option is false by default which means that problems are given as errors.

15.2. Annotations

In addition to the options, smoothness check also introduces a set of annotations for handling special use cases which aren't covered by the language specification.

15.2.1. Function smooth order

The smooth order annotation that is defined by the language specification only allow literal integer expressions. However there are situations where the smoothness order of the function depends on the input arguments. This OCT specific smooth order annotation addresses this issue.

```
"annotation" "("
  "__Modelon" "("
    "smoothOrder" "=" EXP
  ")"
")"
```

The integer typed expression `EXP` may contain an arbitrary expression that refer input variable or package constants whit the restriction that the referenced variables must have known value at compile time. I.e., when referring an input variable, the corresponding function call argument must be of constant variability. If both the smooth order annotation as defined in language specification and the OCT specific smooth order annotation is supplied, then the maximum value of the two will be used.

15.2.1.1. Example 1

This example illustrates usage of the `smoothOrder` annotation where the model passes the smoothness check.

```
model A
  function F
    input Real x;
    input Integer smooth;
    output Real y;
  algorithm
    y := x;
    annotation(__Modelon(smoothOrder=smooth));
  end F;
  constant Integer smooth = 1;
  Real r = F(time, smooth);
end A;
```

15.2.1.2. Example 2

This example illustrates usage of the `smoothOrder` annotation where the smoothness check will give an error. The variability of the function call argument that corresponds to the referenced function input is too high.

```
model A
  function F
```

```
    input Real x;
    input Integer smooth;
    output Real y;
  algorithm
    y := x;
    annotation(__Modelon(smoothOrder=smooth));
  end F;
  parameter Integer smooth = 1;
  Real r = F(time, smooth);
end A;
```

15.2.2. Design parameters

The need to mark design parameters in Modelica code arises when working with Design of Experiment (DOE), sensitivity analysis and optimization. The design parameters may affect the continuity of the system and thus needs to be handled as regular variables during smoothness check.

The OCT specific annotation `DesignParameter`, addresses this problem by allowing the user to mark parameters as design parameters. The annotation syntax is as follows:

```
"annotation" "("
  "__Modelon" "("
    DesignParameter
  ")"
")"
```

Where `DesignParameter` is defined by the following record:

```
record DesignParameter
  parameter Boolean enabled = true;
end DesignParameter;
```

15.2.2.1. Example 1

The following model gives an error when compiled or checked with `allow_disctete_switches` set to false.

```
model A
    parameter Real p annotation(__Modelon(DesignParameter));
    Real r;
equation
    r = if 0 > p then time else -time;
end A;
```

Normally the test in the if-expression wouldn't give an error for expressions referencing parameters. But an error is given in this case since the parameter p is marked as design parameter.

Chapter 16. The Optimica Language Extension

In this chapter, the Optimica extension will be presented and informally defined. The Optimica extension in described in detail in [Jak2008a], where additional motivations for introducing Optimica can be found. The presentation will be made using the following dynamic optimization problem, based on a double integrator system, as an example:

$$\min_{u(t)} t_f$$

subject to the dynamic constraint

$$\dot{x}(t) = v(t) \quad , \; x(t) = 0$$
$$\dot{v}(t) = u(t) \quad , \; v(t) = 0$$

and

$$v(t_f) = 0 \qquad x(t_f) = 1$$
$$1 < u(t) < -1 \quad v(t) < 0.5$$

In this problem, the final time, *tf*, is free, and the objective is thus to minimize the time it takes to transfer the state of the double integrator from the point (0,0) to (1,0), while respecting bounds on the velocity *v(t)* and the input *u(t)*. A Modelica model for the double integrator system is given by:

```
model DoubleIntegrator
  Real x(start=0);
  Real v(start=0);
  input Real u;
equation
  der(x)=v;
  der(v)=u;
end DoubleIntegrator;
```

In summary, the Optimica extension consists of the following elements:

- A new specialized class: `optimization`
- New attributes for the built-in type Real: `free` and `initialGuess`
- A new function for accessing the value of a variable at a specified time instant
- Class attributes for the specialized class `optimization`: `objective`, `startTime`, `finalTime` and `static`

- A new section: `constraint`
- Inequality constraints

16.1. A new specialized class: optimization

A new specialized class, called `optimization`, in which the proposed Optimica-specific constructs are valid is supported by Optimica. This approach is consistent with the Modelica language, since there are already several other specialized classes, e.g., `record`, `function` and `model`. By introducing a new specialized class, it also becomes straightforward to check the validity of a program, since the Optimica-specific constructs are only valid inside an `optimization` class. The `optimization` class corresponds to an optimization problem, static or dynamic, as specified above. Apart from the Optimica-specific constructs, an `optimization` class can also contain component and variable declarations, local classes, and equations.

It is not possible to declare components from `optimization` classes in the current version of Optimica. Rather, the underlying assumption is that an `optimization` class defines an optimization problem, that is solved off-line. An interesting extension would, however, be to allow for `optimization` classes to be instantiated. With this extension, it would be possible to solve optimization problems, on-line, during simulation. A particularly interesting application of this feature is model predictive control, which is a control strategy that involves on-line solution of optimization problems during execution.

As a starting-point for the formulation of the optimization problem consider the `optimization` class:

```
optimization DIMinTime
  DoubleIntegrator di;
  input Real u = di.u;
end DIMinTime;
```

This class contains only one component representing the dynamic system model, but will be extended in the following to incorporate also the other elements of the optimization problem.

16.2. Attributes for the built in class Real

In order to superimpose information on variable declarations, two new attributes are introduced for the built-in type Real. Firstly, it should be possible to specify that a variable, or parameter, is free in the optimization. Modelica parameters are normally considered to be fixed after the initialization step, but in the case of optimization, some parameters may rather be considered to be free. In optimal control formulations, the control inputs should be marked as free, to indicate that they are indeed optimization variables. For these reasons, a new attribute for the built-in type Real, `free`, of boolean type is introduced. By default, this attribute is set to `false`.

Secondly, an attribute, `initialGuess`, is introduced to enable the user to provide an initial guess for variables and parameters. In the case of free optimization parameters, the `initialGuess` attribute provides an initial guess to the optimization algorithm for the corresponding parameter. In the case of variables, the `initialGuess` attribute is used to provide the numerical solver with an initial guess for the entire optimization interval. This is particularly

important if a simultaneous or multiple-shooting algorithm is used, since these algorithms introduce optimization variables corresponding to the values of variables at discrete points over the interval. Note that such initial guesses may be needed both for control and state variables. For such variables, however, the proposed strategy for providing initial guesses may sometimes be inadequate. In some cases, a better solution is to use simulation data to initialize the optimization problem. This approach is also supported by the Optimica compiler. In the double integrator example, the control variable *u* is a free optimization variable, and accordingly, the `free` attribute is set to `true`. Also, the `initialGuess` attribute is set to 0.0.

```
optimization DIMinTime
  DoubleIntegrator di(u(free=true,
                        initialGuess=0.0));
  input Real u = di.u;
end DIMinTime;
```

16.3. A Function for accessing instant values of a variable

An important component of some dynamic optimization problems, in particular parameter estimation problems where measurement data is available, is variable access at discrete time instants. For example, if a measurement data value, y_i, has been obtained at time t_i, it may be desirable to penalize the deviation between y_i and a corresponding variable in the model, evaluated at the time instant t_i. In Modelica, it is not possible to access the value of a variable at a particular time instant in a natural way, and a new construct therefore has to be introduced.

All variables in Modelica are functions of time. The variability of variables may be different-some are continuously changing, whereas others can change value only at discrete time instants, and yet others are constant. Nevertheless, the value of a Modelica variable is defined for all time instants within the simulation, or optimization, interval. The time argument of variables are not written explicitly in Modelica, however. One option for enabling access to variable values at specified time instants is therefore to associate an implicitly defined function with a variable declaration. This function can then be invoked by the standard Modelica syntax for function calls, `y(t_i)`. The name of the function is identical to the name of the variable, and it has one argument; the time instant at which the variable is evaluated. This syntax is also very natural since it corresponds precisely to the mathematical notation of a function. Note that the proposed syntax `y(t_i)` makes the interpretation of such an expression context dependent. In order for this construct to be valid in standard Modelica, `y` must refer to a function declaration. With the proposed extension, `y` may refer either to a function declaration or a variable declaration. A compiler therefore needs to classify an expression `y(t_i)` based on the context, i.e., what function and variable declarations are visible. This feature of Optimica is used in the constraint section of the double integrator example, and is described below.

16.4. Class attributes

In the optimization formulation above, there are elements that occur only once, i.e., the cost function and the optimization interval. These elements are intrinsic properties of the respective optimization formulations, and should be specified, once, by the user. In this respect the cost function and optimization interval differ from, for example, constraints, since the user may specify zero, one or more of the latter.

In order to encode these elements, class attributes are introduced. A class attribute is an intrinsic element of a specialized class, and may be modified in a class declaration without the need to explicitly extend from a built-in class. In the Optimica extension, four class attributes are introduced for the specialized class `optimization`. These are `objective`, which defines the cost function, `startTime`, which defines the start of the optimization interval, `finalTime`, which defines the end of the optimization interval, and `static`, which indicates whether the class defines a static or dynamic optimization problem. The proposed syntax for class attributes is shown in the following `optimization` class:

```
optimization DIMinTime (
        objective=finalTime,
        startTime=0,
        finalTime(free=true,initialGuess=1))
 DoubleIntegrator di(u(free=true,
                       initialGuess=0.0));
 input Real u = di.u;
end DIMinTime;
```

The default value of the class attribute `static` is `false`, and accordingly, it does not have to be set in this case. In essence, the keyword `extends` and the reference to the built-in class have been eliminated, and the modification construct is instead given directly after the name of the class itself. The class attributes may be accessed and modified in the same way as if they were inherited.

16.5. Constraints

Constraints are similar to equations, and in fact, a path equality constraint is equivalent to a Modelica equation. But in addition, inequality constraints, as well as point equality and inequality constraints should be supported. It is therefore natural to have a separation between equations and constraints. In Modelica, initial equations, equations, and algorithms are specified in separate sections, within a class body. A reasonable alternative for specifying constraints is therefore to introduce a new kind of section, `constraint`. Constraint sections are only allowed inside an `optimization` class, and may contain equality, inequality as well as point constraints. In the double integrator example, there are several constraints. Apart from the constraints specifying bounds on the control input *u* and the velocity *v*, there are also terminal constraints. The latter are conveniently expressed using the mechanism for accessing the value of a variable at a particular time instant; `di.x(finalTime)=1` and `di.v(finalTime)=0`. In addition, bounds may have to be specified for the `finalTime` class attribute. The resulting optimization formulation may now be written:

```
optimization DIMinTime (
    objective=finalTime,
    startTime=0,
    finalTime(free=true,initialGuess=1))
 DoubleIntegrator di(u(free=true,
                       initialGuess=0.0));
 input Real u = di.u;
constraint
 finalTime>=0.5;
 finalTime<=10;
```

```
 di.x(finalTime)=1;
 di.v(finalTime)=0;
 di.v<=0.5;
 di.u>=-1; di.u<=1;
end DIMinTime;
```

The Optimica specification can be translated into executable format and solved by a numerical solver, yielding the result seen in Figure 16.1.

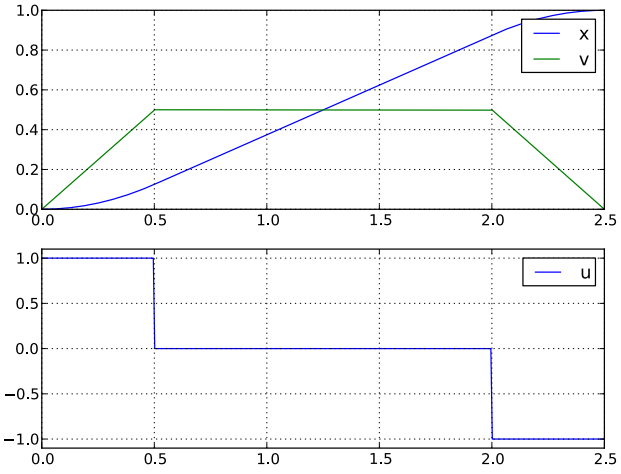


Figure 16.1 Optimization result

Chapter 17. The OPTIMICA Compiler Toolkit API

The OPTIMICA Compiler Toolkit API allows easy use of the compiler's features - compilation of `Modelica`

To start working with the API, refer to its Javadoc, located in `<Installation folder>\doc\api\javadoc`. Therein is an overview of the API's packages when opening `index.html` (see Figure 17.1) An introduction to the features described above is in the package `com.modelon.oct.modelica.api` in the class `API`.

Packages

Package	Description
com.modelon.oct.modelica.api	The main API package, contains the core of the API; authentication and protection annotation management as well as API exceptions.
com.modelon.oct.modelica.api.compiler	The compiler package contains classes related to compiling Modelica models to FMUs.
com.modelon.oct.modelica.api.compiler.options	Functionality for compiler options, used via the `CompilerOptionsManager` and the `RuntimeOptionsManager`.
com.modelon.oct.modelica.api.generated	Option classes created during build time.
com.modelon.oct.modelica.api.model.equations	
com.modelon.oct.modelica.api.model.filter	Contains classes related to filtering of elements in an instance- or source tree.
com.modelon.oct.modelica.api.model.instance	Describes classes related to the instance tree, with `InstanceElement` as top-level parent class.
com.modelon.oct.modelica.api.model.prefixes	Classes for describing Modelica prefixes and the identity of Modelica classes and components:
com.modelon.oct.modelica.api.model.source	Describes classes related to the source tree, with `SourceElement` as top-level parent class.
com.modelon.oct.modelica.api.model.types	The type package contains classes representing the built-in types in Modelica, used in `Expressions`.
org.jmodelica.api.problemHandling	This package contains classes representing `Problems`, that is, issues with the source code detected during compilation.

Figure 17.1 The packages in the OCT API, as seen from its Javadoc.

17.1. Using the API from the command line

Java programs that use the API can be compiled and run from the command line in a two-step process. The program is first compiled using `javac` and then executed using `java`. Make sure that the directory containing these programs is listed in the system environment variable `PATH`. Additionally, the process depends on one environment variable and a few JAR-files. The environment variable `JMODELICA_HOME` points to the install directory in the OCT binary distribution. There are four JAR-files that programs using the API depend on: `ModelicaCompiler.jar`, `separateProcess.jar`, `util.jar` and `beaver-rt.jar` The absolute paths to the JAR-files are concatenated with a semicolon to make up the classpath. The classpath also contains the current working directory, symbolized by the "." character. It is passed as a parameter to `javac` and `java` using the `-classpath` switch. A batch script to compile and run a program `MyTest.java` that uses the API is shown below.

```
setlocal
IF NOT DEFINED JMODELICA_HOME set JMODELICA_HOME=C:\OCT-x.x\install
```

```
set "JMODELICA_CP=%JMODELICA_HOME%\lib\ModelicaCompiler.jar;%JMODELICA_HOME%\lib
\separateProcess.jar;%JMODELICA_HOME%\lib\util.jar;%JMODELICA_HOME%\ThirdParty\Beaver\lib
\beaver-rt.jar;."
javac -classpath %JMODELICA_CP% MyTest.java
java -classpath %JMODELICA_CP% MyTest
endlocal
```

The java program `MyTest.java` is an example of how to use the API to easily retrieve and print all components of the `PID_Controller` model from the Modelica Standard Library.

```
import java.nio.file.Path;

import com.modelon.oct.modelica.api.API;
import com.modelon.oct.modelica.api.APIException;
import com.modelon.oct.modelica.api.model.source.SourceClass;
import com.modelon.oct.modelica.api.model.source.SourceComponent;
import com.modelon.oct.modelica.api.model.source.SourceElement;

class MyTest {
 public static void main(String[] args) {
  API api = new API(null);
  try {
   String msl = System.getenv("JMODELICA_HOME") + "\\ThirdParty\\MSL\\MSL400\\Modelica";
   api.loadLibrary(Path.of(msl), true);
   System.out.println("loaded library " + msl);

   SourceClass pidController =
 api.getSourceClass("Modelica.Blocks.Examples.PID_Controller");
   System.out.println("retrieved model " + pidController.getDeclaredName() + ", contains
 the components:");
   for (SourceElement sourceElement : pidController.getElements(SourceComponent.class)) {
    System.out.println("  " + sourceElement.getDeclaredName());
   }
  } catch (IllegalStateException e) {
   e.printStackTrace();
  } catch (APIException e) {
   e.printStackTrace();
  } finally {
   api.tearDownCompiler();
  }
 }
}
```

The output of running `MyTest.java` is shown below.

```
loaded library C:\OCT-x.x\install\ThirdParty\MSL\MSL400\Modelica
retrieved model PID_Controller, contains the components:
  driveAngle
  PI
  inertia1
  torque
```

```
  spring
  inertia2
  kinematicPTP
  integrator
  speedSensor
  loadTorque
```

Refer to the javadoc of the `API` class for additional usage examples.

Chapter 18. Source Code FMUs

OCT supports generation of source code FMUs following the FMI 2.0 standard. The functionality supports both Model Exchange FMUs and Co-Simulation FMUs. Generating a source code FMU is determined by the compiler option `source_code_fmu`. The support should be considered experimental.

If the source code FMUs are to be used in a real-time setting and Co-Simulation FMUs are used, then it is recommended to switch the default solver from CVode to Explicit Euler or Runge-Kutta (2nd order) by setting the parameter `_cs_solver` to `1` or `2` respectively (if set to `0` then CVode is used). The internal step-size can be changed by setting the parameter `_cs_step_size`.

Given below is an example for generating a source code FMU for the CoupledClutches model available in the Modelica Standard Library.

```
from pymodelica import compile_fmu

#Model name
name = "Modelica.Mechanics.Rotational.Examples.CoupledClutches"

#Specify option to generate a source code FMU
options = {"source_code_fmu":True}

# Compile model
fmu_name = compile_fmu(name, target="cs", compiler_options=options)
```

18.1. Linear algebra routines

The generated FMUs contains the standard implementation of the linear algebra routines included in Blas and Lapack. If the target machine has an optimized Blas and Lapack library available, it is benefical to use this instead of the included version for increased performance. To replace the included Blas/Lapack, the files `f2c.c`, `blas.c` and `lapack.c` should be removed and the remaining `*.c` files should be compiled and linked to the optimized Blas/Lapack library.

18.2. External Code

Many Modelica libraries use external functions or external objects. A source code FMU needs access to all the source code which includes the source code for these external functions / objects. When generating a source code FMU, the compiler detects if the model includes any external functions or objects from any library. If it does, the compiler tries to find the the source code for these in the path `"LibraryName/Resources/C-Sources"` and includes these in the resulting FMU. If there is no source at this location the FMU will not work, there will be a compilation error when trying to compile the source code due to missing funtions.

18.3. Running on dSpace DS1006

The generated source code FMUs have been tested to work on dSpace DS1006 using MATLAB® Simulink and the FMI Toolbox to compile the model compatible with DS1006.

The following steps are recommended in order to use an OCT generated source code FMU on DS1006.

1. Generate a Co-Simulation (FMI2) from OCT by using the compiler option `source_code_fmu` set to True and the target set to `cs`.

2. Open MATLAB® (with the addon FMI Toolbox installed) and open a new Simulink model.

3. Drag the FMI Co-Simulation block from the FMI Toolbox into the model and load the source code FMU into the block.

4. Set the parameter `_cs_solver` to `1` or `2` in the FMU, which indicates usage of the solver Explicit Euler or Runge-Kutta (2nd order). If needed - also adapt the specified step-size by changing the parameter `_cs_step_size`.

5. Still in the configuration window for the FMU block. Go to advanced and set the sample time equal to the step-size.

6. Add outputs and scope to visualize the results.

7. In the Simulink model go to configuration of parameters.

   a. Set the solver to fixed-step and ode1 (Euler). Furthermore, set the step-size to the same as was set previously in the FMU.

   b. Go to code generation and set the system target file to point to the appropriate for DS1006 (`rti1006.tlc`).

   c. Go to the general build options and in the text box for compiler options input the following line (update the model name): `-DFMI2_FUNCTION_PREFIX=<ModelName>_ -DNO_FILE_SYSTEM -DNO_PID -DNO_TIME -DNO_MUTEX -D_SSIZE_T -D__int8_t_defined -DDUMMY_FUNCTION_USERTAB`

   d. Goto code generation and build the model.

8. The model should now compile successfully and be available for use on a dSpace DS1006 machine.

18.3.1. dSpace configuration defines

Compiling on DS1006, the following defines has to be set:

- `FMI2_FUNCTION_PREFIX=<ModelName>_`
- `NO_FILE_SYSTEM`

- NO_PID
- NO_TIME
- NO_MUTEX
- _SSIZE_T
- DUMMY_FUNCTION_USERTAB_

18.4. Limitations

The generated source code FMUs contain a few limitations which are listed below:

- No support for the nonlinear solver MINPACK.
- On Windows, only the Microsoft Visual Studio compiler is supported.
- On machines without file system, the define NO_FILE_SYSTEM has to be set.
- All source code for external functions/objects has to be available.
- If the FMI functions need to be prefixed with the model name, the define FMI2_FUNCTION_PREFIX has to be set.
- Only one source code FMU may be used simultaneously in a Simulink model (when used together with FMI Toolbox). This is due to that the define FMI2_FUNCTION_PREFIX has to be set for each FMU which is not possible (it can only be set globally).

Chapter 19. Cross-platform generation of FMUs

While FMUs are generally specific to the platform they have been compiled on, OCT supports the generation of *Windows* FMUs on *CentOS*.

In this chapter, we describe how to generate an Windows FMU (compiled with *clang* using libraries from *Microsoft Visual C (MSVC) 2015*), from CentOS.

19.1. Prerequisites and setup

This feature requires the environment to be setup to support the use of *clang*, version 7. Installation and activation can be done as follows:

```
yum install centos-release-scl-rh
yum install llvm-toolset-7.0
./opt/rh/llvm-toolset-7.0/enable ## activation
clang --version ## check that installation has been successful
```

Next, one needs to download the necessary MSVC 2015 libraries required for compilation. These can be downloaded using the tools and instructions at https://github.com/modelon-community/xwin.

OCT locates the MSVC 2015 libraries when compiling a model, by checks in the following order:

1. The path specified in the compiler option *target_platform_packages_directory*.
2. The environment variable *OCT_CROSS_COMPILE_MSVC_DIR*.
3. The default location *<OCT_Install_folder>/lib/win64*.

Modelica libraries with external code require compiled binaries compatible with the target platform, Windows, and that the external code is compiled using the Microsoft Visual C (MSVC) 2015 compiler. Furthermore, the external code needs to be additionally compiled to work on the native platform, CentOS.

19.2. Limitations

The following limitations apply to the cross-platform compilation of FMUs:

- OCT only supports cross-platform generation to Windows 64bit.

19.3. Example using the Python API

The following exampls demonstrates how to compile a Windows FMU on CentOS, using the OCT Python API:

```
from pymodelica import compile_fmu
import os

msvc_dir_path = os.path.join('path', 'to', 'your', 'MSVC2015', 'libraries')
compiler_options = {'target_platform_packages_directory' : msvc_dir_path}
model = 'Modelica.Mechanics.Rotational.Examples.CoupledClutches'
fmu = compile_fmu(model, platform = 'win64', compiler_options = compiler_options)
```

Here, *platform = 'win64'* specifies that the target platform is Windows and the compiler option *target_platform_packages_directory* provides the compiler with the necessary MSVC libraries for a successful cross-platform compilation.

Chapter 20. Limitations

This page lists the current limitations of the OCT platform.

The following limitations apply to the language elements that can be used:

- Support for partial function calls is limited.
- For Modelica 3.4, the language elements described in the specification in chapter 16 - Synchronous Language Elements and chapter 17 - State Machines, are not supported.
- No support for *multiple definition import*.
- The following built-in functions are not supported:

`terminal()`

- The following built-in functions are only supported in FMUs:

`ceil(x)`	`integer(x)`	`reinit(x, expr)`	`div(x,y)`
`mod(x,y)`	`sample(start,interval)`	`edge(b)`	`pre(y)`
`semiLinear(...)`	`floor(x)`	`rem(x,y)`	`sign(v)`
`initial()`	`delay(...)`	`spatialDistribu-tion(...)`	`change(v)`

- The following operators are only partially supported:

`homotopy()`

- The following annotations are not supported:

`arrayLayout`	`obsolete`
`unassignedMessage`	

- The following annotations are limited:
  - The `zeroDerivative` annotation is treated the same as the `noDerivative` annotation.
  - The `inverse` annotation does not support using references with array indices or dotted references as arguments to the inverse function.
- There is limited support for using arrays in record arrays.

- Partial derivatives are not supported.
- There is limited support for quoted identifiers (i.e. the secondary form for identifiers, that is enclosed in single quotes, e.g. 'a quoted name').
- No support for non-parameter array size declarations. This means that all array size declarations must be of constant or parameter variability.
- Index reduction fails in some complex cases. It is possible to work around this by setting the `stateSelect` attribute manually.
- Deduction of expandable connector causality for array variables requires same causality for all elements.

In the Optimica front-end the following constructs are not supported:

- Annotations for transcription information.

The following limitations apply to optimization using CasADi-based collocation with OCT:

- Incomplete support for the `Integer` and `Boolean` types: To the extent that they are supported, they are treated more or less like reals.
- No support for `String` and `limited support for enumeration` types.
- Attributes with any name can be set on any type of variable.
- The property of whether an optimization problem has free or fixed time horizon cannot be changed after compilation.
- Limited support for external objects / functions. Only one external object of the same type is supported. Limited support for arrays and records. Limited support for derivative annotations.

The following limitations apply to FMUs compiled with OCT:

- Directional derivatives are known to have limitations in some cases.
- Asynchronous simulation is not supported.

The following restrictions apply to importing CS FMUs:

- Models imported with fmumodelica cannot be compiled with the compiler option *c_compiler* set to *msvs*.

The following restrictions apply to exported FMUs following FMI 2.0:

- Models with external objects may lead to undesired behavior when used together with set/get of FMU states as well as the serialization and de-serialization of FMU states.

The following restrictions apply to OCT MATLAB® Interface:

- The features in the optimization framework of `oct.nlpsol.*` should be regarded as experimental.

The following restrictions apply to the hand guided tearing feature:

- Hand guided tearing is only supported for steady state simulation. Using hand guided tearing in conjunction with other types of simulation (e.g. dynamic simulation) is untested and has undefined behavior.

The following restrictions apply to encrypted libraries:

- It is not possible to optimize models that use encrypted libraries.

The following are known issues in the API:

- getAllMatchingRedeclareChoices() performs a global analysis of all modelica classes the first time it is called. The result of the analysis is flushed after editing operations. Therefore, getAllMatchingRedeclareChoices() can be expensive to use if calls to it are interleaved with editing operations.

- There are known issues with how qualifiedName() works with respect to elements under simple short classes and arrays. Both relative (QualifiedNameProvider.qualifiedName(editingClass)) and non-relative (InstanceElement.qualifiedName()) versions of qualifiedName() are affected, but not in the same way. Erroneous names and/or oddly formatted names may be produced. This affects the Instance-API but not the Source-API.

- If a file contains a line that is at least 4096 characters long, all formatting, including whitespace and comments, is lost in that file. Operations retrieving source text from such a file, including saving the file through the OCT API, will show the source text with OCT's default formatting.

- After performing an editing operation, the results of the SourceElement methods getSourceBeginLine(), getSourceBeginColumn(), getSourceEndLine() and getSourceEndColumn() are undefined for all elements in all changed files.

Chapter 21. Common Functionality

21.1. Obfuscating Variables in a Functional Mock-up Unit

A model has a number of variables. When the model is compiled by OCT, the default behavior is to make all variables visible in the output Functional Mock-up Unit(FMU) with the structured Modelica naming intact. The information about the Modelica source code that can be derived from the FMU can be reduced by limiting what variables are exposed in the xml (see Section 21.1.1) and by automatically renaming variables (see Section 21.1.2).

21.1.1. Restricting Exposed Variables in a Functional Mock-up Unit

A variable is an internal variable if it is not an input, output, or state variable or a runtime option. All internal variables can be hidden by using the compiler option. Note that an internal variable here is not the same as an internal variable in the FMI specification.

```
exclude_internal_variables="*"
```

which using the Python interface, see Section 4.2.4, corresponds to

```
from pymodelica import compile_fmu
my_fmu = compile_fmu('Modelica.Mechanics.Rotational.Examples.First',
 compiler_options={'exclude_internal_variables':'*'})
```

Some hidden variables can then be selectively exposed by using the compiler option `include_internal_variables` where one or more variable name patterns can be specified. The variable name patterns are specified using GLOB patterns[1]. Specifying the compiler option

```
include_internal_variables="name*"
```

will expose all internal variables that have a name that begins with `name`. To specify several GLOB patterns, separate them by a space character. For instance

```
include_internal_variables="x* y*"
```

will expose all variables which begin with the letter `x` or `y`. Using the Python interface, this would correspond to

```
from pymodelica import compile_fmu
my_fmu = compile_fmu('Modelica.Mechanics.Rotational.Examples.First',
 compiler_options={'include_internal_variables':'x* y*'})
```

[1] https://en.wikipedia.org/wiki/Glob_%28programming%29

The GLOB pattern is matched with the names of the FMI variables, not the Modelica variables. For Modelica arrays, the glob pattern should match the names of the elements, rather than the name of the array variable. Note that GLOB syntax include brackets as a wildcard definition so when matching with Modelica array variables the brackets have to be escaped using back slash.

```
include_internal_variables="x\\[1\\"
```

In addition, GLOB patterns can also be added using the options `include_internal_variables_from` and `exclude_internal_variables_from`. The values of these options should be set to paths to files that contain lines similar to the input of the `include_internal_variables` and `exclude_internal_variables` options. The total lists of GLOB patterns used for filtering is built using both the base option and the *_from option. We could achieve the same effect as the previous example by pointing `include_internal_variables_from` to a file with the following contents:

```
x* y*
```

Since the file also separates by newlines it could also be represented as:

```
x*
y*
```

21.1.2. Automatic Renaming of Variables in a Functional Mock-up Unit

The default behavior of the compiler is to maintain the structured names derived from the Modelica source code. This can be configured by setting the options `obfuscate_variables`, `obfuscate_variables_from`, `keep_variables`, and `keep_variables_from`. The options are configured in a similar way as `exclude_internal_variables` etc. from the previous section. The following example will rename all variables that does not end their name with `x`.

```
obfuscate_variables="*"
keep_variables="*x"
```

When renaming is performed the compiler also generates a file with a mapping from the new names to the old names. For an FMU `model.fmu` the mapping file is put in the same directory with the name `model.fmu.varMap.txt`. The first line starts with `GUID` and what follows is the same guid that is written into the FMU, this can be used to validate that a map file belongs to a specific FMU. Each line after that contains the new and old name of a renamed variable, separated by a space. Below is an example of what a mapping file might look like.

```
GUID abc321
input_0 m.a[1]
state_0 m.a[2]
var_0 m.a[3]
var_1 x
```

Appendix A. License Installation

A.1. Introduction

Modelon provides products that are licensed with the FlexNet Publisher license system. Modelon generates compliant license files and vendor daemons via FlexNet for their suite of products. This documentation guides you through some basic steps for installing license files and license servers. Detailed information about the FlexNet Publisher license system can be found at *http://learn.flexerasoftware.com/content/ELO-LMGRD*.

For instructions on how to retrieve a license file, see Section A.2.

For instructions on how to install a license file, see Section A.3.

For instructions on how to install a license server, see Section A.4.

For instructions on how to borrow licenses from a server for offline use, see Section A.5.

For trouble shooting and contacting Modelon support, see Section A.6.

A.2. Retrieving a license file

There are different types of license models that can be used with Modelon products.

- *Node-locked* (No license server required)

  This license enables use on a single computer. The license cannot be moved from one computer to another. The license is locked for use on a computer with a specific MAC address.

- *Server* (Requires a license server)

  This licensing model represents a classic network configuration with a server and users. The server grants or denies requests from computers in the network to use a program or feature. The license file specifies the maximum number of concurrent users for a program or feature. There is no restriction for which computer is using the program or feature, only in the number of programs and features that can be used simultaneously.

  The computer on which the server is running cannot be changed. The server computer's MAC address must be provided to Modelon to generate the license file.

- Evaluation license (*Node Locked*)

  This license enables a program or feature for a limited amount of time and is the same as a node-locked license.

Please contact the Modelon sales department at <sales@modelon.com> to purchase a license or to get an evaluation license. In order to obtain a license file for a node-locked license, you must provide the MAC address of your computer. If you are using a license server, you must provide the MAC address of the server. In Section A.2.1 below, you will find instructions for how to retrieve the MAC address of a computer.

A.2.1. Get MAC address

Modelon uses the Ethernet address (MAC address), also called the host ID, to uniquely identify a specific computer. Therefore, you must provide the MAC address of the computer on which you want to use the program or feature. For a server license, the MAC address for the server computer is required, not all the client computers in the network that will use the program or feature. For a node-locked license, the MAC address of the computer on which the license will be used must be provided.

Note: Modelon only allows ONE MAC address for each computer. Please disable and unplug all network devices that are not permanently connected to the computer such as laptop docking stations, virtual machines and USB network cards.

- Windows

1. Open **cmd**

   Windows 7 and Vista

   a. Click the **Start** button

   b. Type **cmd** in the search bar and press enter.

   Windows XP

   a. Click the **Start** button.

   b. Click on **Run...**.

   c. Type **cmd** in the text box and click **OK**.

2. Run `lmhostid.exe`.

   Type the full path to lmhostid.exe within quotes and press enter. `lmhostid.exe` is normally located in `<installation folder>\license_tools\lmhostid.exe`.

3. Use this hostid when you are in contact with Modelon. If multiple hostids are listed, select one that is permanent for the computer.

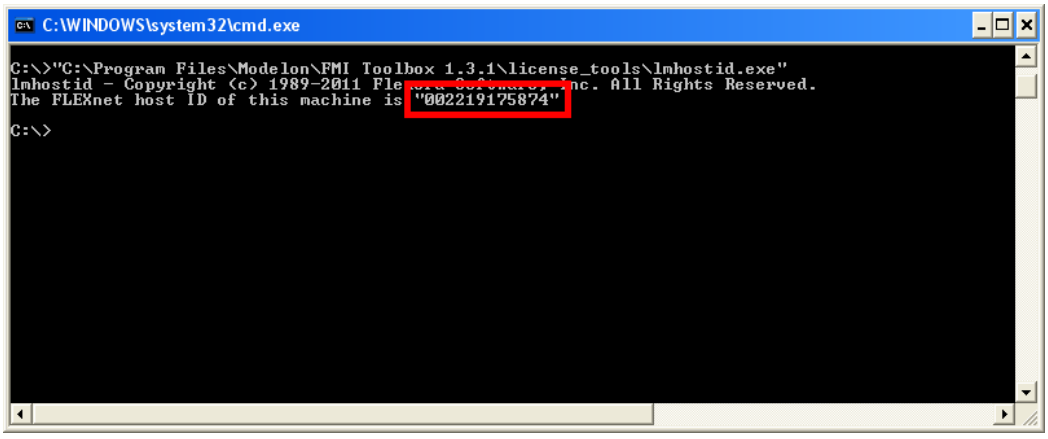


Figure A.1 Lmhostid.exe run on Windows listing the computer's MAC address.

- Unix

1. Open a terminal and change directory to the `<installation folder>/license_tools/`.

   Run `lmhostid` and use the hostid listed when you are in contact with Modelon. If multiple hostids are listed, select one that is permanent for the computer.

A.3. Install a license

After purchasing a license, you should receive a license file with the file extension `*.lic`. This file must be put in a specific folder for the application to find it.

A.3.1. Installing a node-locked license

A.3.1.1. Windows

1. Close the application if it is already running.
2. Open the *Application Data* folder.

   Windows 7 and Windows Vista

   a. Click the **Start** button.

   b. Type `shell:AppData` in the search bar and press enter.

   Windows XP

a. Click the Start button.

b. Click on **Run...**.

c. Type `shell:AppData` in the text box and click **OK**.

3. The *Application Data* folder should now be open. Check that its path is of the form *C:\Users\YourUserName\AppData\Roaming*.

4. Create the folder `Modelon\Licenses\Nodelocked` if it does not exist already.

5. Put your license file in the folder `Nodelocked`.

A.3.1.2. Unix

- Copy your license file to the folder `<installation folder>\Licenses\Nodelocked`.

A.3.1.3. Updating the license

To update the license file, you should overwrite the old license file with the new one. Ensure that the old license file is overwritten or removed from the folder since it may otherwise be used instead of the new one, and the application may fail to check out a license. Note that you must restart the program for license changes to take effect.

A.3.2. Installing a server license

Note that these are not instructions for installing a license file on a server. These are instructions for the end user of the program or feature. The assumption is that the server is already up and running and that the IP address to the server and the port number is already known. The IP address and the port number, if needed, should be provided by the license server administrator.

The application can connect to the license-server and daemon either by reading a license file or an environment variable.

A.3.2.1. Windows

1. Close the application if it is already running.

2. Create an empty text file

   Windows 7 and Windows Vista

   a. Click the **Start** button.

   b. Type `Notepad` in the search bar and press enter.

Windows XP

a. Click the **Start** button.

b. Click on **Run...**.

c. Type `Notepad` in the text box and click **OK**.

3. Configure the license file.

a. Copy the following text in to the text document

```
SERVER <ip-address> ANY <port>
USE_SERVER
```

b. Change `<ip-address>` to the IP address of the server.

c. Change `<port>` to the port number that is being used. If you do not have a port number, you can remove the whole `<port>`. For example, the license file should look like the following for a license server with IP address 192.168.0.12 using port 1200.

```
SERVER 192.168.0.12 ANY 1200
USE_SERVER
```

d. Save the file with a filename with the extension `*.lic` in a temporary place. The file will be moved in a later step. You can now close Notepad.

4. Open the *Application Data* folder.

Windows 7 and Windows Vista

a. Click the **Start** button.

b. Type `shell:AppData` in the search bar and press enter.

Windows XP

a. Click the **Start** button.

b. Click on **Run...**.

c. Type `shell:AppData` in the text box and click **OK**.

The *Application Data* folder should now open.

5. Create the folder `Modelon\Licenses\Server` if it does not exist already.

6. Put the license file you just created in the folder `Server`.

A.3.2.2. Unix

1. Close the program if it is already running.
2. Create an empty file with the file extension name `*.lic`.
3. Configure the license file.
   a. Copy the following text in to the text document

```
SERVER <ip-address> ANY <port>
USE_SERVER
```

   b. Change `<ip-address>` to the IP address of the server.
   c. Change `<port>` to the port number that is being used. If you do not have a port number, you can remove the whole `<port>`. For example, the license file should look like the following for a license server with IP address 192.168.0.12 using port 1200.

```
SERVER 192.168.0.12 ANY 1200
USE_SERVER
```

4. Copy your license file to the folder `<installation folder>\Licenses\Server`.

A.3.2.3. Using the environment variable

An alternative to specify how the application should connect to the license server is to set the environment variable MODELON_LICENSE_FILE. The value can be set to *port@host*, where port and host are the TCP/IP port number and host name from the SERVER line in the license file. Alternatively, use the shortcut specification, @host, if the license file SERVER line uses a default TCP/IP port or specifies a port in the default port range (27000#27009).

A.3.2.4. Updating the license

To update the license file, you can either redo the installation instructions described above or make the changes in the license file directly. Ensure that the old license file is overwritten or removed from the folder since it may otherwise be used instead of the new one, and the application may fail to check out a license. Note that you must restart the program before the changes can take effect.

A.4. Installing a license server

To install a license server, you must have a server license file. Please contact `<sales@modelon.com>` to obtain the server license file. This license file must also be configured prior to use by by setting the IP address and port as shown in Section A.4.1

Modelon products use a licensing solution provided by Flexera Software. It is recommended that you install the latest version of the server software, which is available from http://learn.flexerasoftware.com/content/ELO-LM-GRD . Modelon products require a license server version number v11.10.0.0 or later. A license server and a license daemon are required and are distributed with the product you are installing. If you have not received the server application or license daemon with your product, please contact `<sales@modelon.com>`.

The following step by step instructions for installing a license server assume that no other Flexera license server is already installed.

A.4.1. Configure the license file

When a license server is installed, the server needs a license file provided by Modelon. This file must be configured before it can be used.

1. Open the license file in a text editor. The file may look like the example below:

```
SERVER 192.168.0.1 080027004ca5 25012
VENDOR modelon
FEATURE FMI_TOOLBOX modelon 1.0 3-feb-2012 12 SIGN="0076 305..."
```

2. Edit the SERVER line where the IP address, `192.168.0.1`, should be replaced with the IP address of the server. Also change the port address, `25012`, to the desired port or remove it to use default ports. The IP address and potentially also the port address should be provided to the end users so they can configure their license files to connect to the server.

A.4.2. Installation on Windows

In the `<installation folder>\license_tools` folder that is distributed with your product, you will find the files listed below.

The listed files are used to set up and configure the license server.

- *lmgrd.exe* (license server)
- *modelon.exe* (license daemon)
- *lmutils.exe* (configure- and utility functions)
- *lmtools.exe* (Windows GUI for setting up the license server as a Windows service)

To configure a license server manager (`lmgrd`) as a service, you must have Administrator privileges. The service will run under the LocalSystem account. This account is required to run this utility as a service.

1. Make sure that license daemon `modelon.exe` is in the same folder as the license server, `lmgrd.exe`.

2. Run `lmtools.exe`

3. Click the **Configuration using Services** button, and then click the **Config Services** tab.

4. In the **Service Name**, type the name of the service that you want to define, for example, `Modelon License Server`.

5. In the **Path to the lmgrd.exe** file field, enter or browse to `lmgrd.exe`.

6. In the **Path to the license file field**, enter or browse to the server license file.

7. In the **Path to the debug log file**, enter or browse to the debug log file that this license server should write. Prepending the debug log file name with the + character appends logging entries. The default location for the debug log file is the `c:\winnt\System32 folder`. To specify a different location, be careful to specify a fully qualified path.

8. Make this license server manager a Windows service by selecting the **Use Services** check box.

9. **Optional**. Configure the license server to start at system startup time by selecting the **Start Server at Power Up** check box.

10. To save the new `Modelon License Server` service, click **Save Service**.

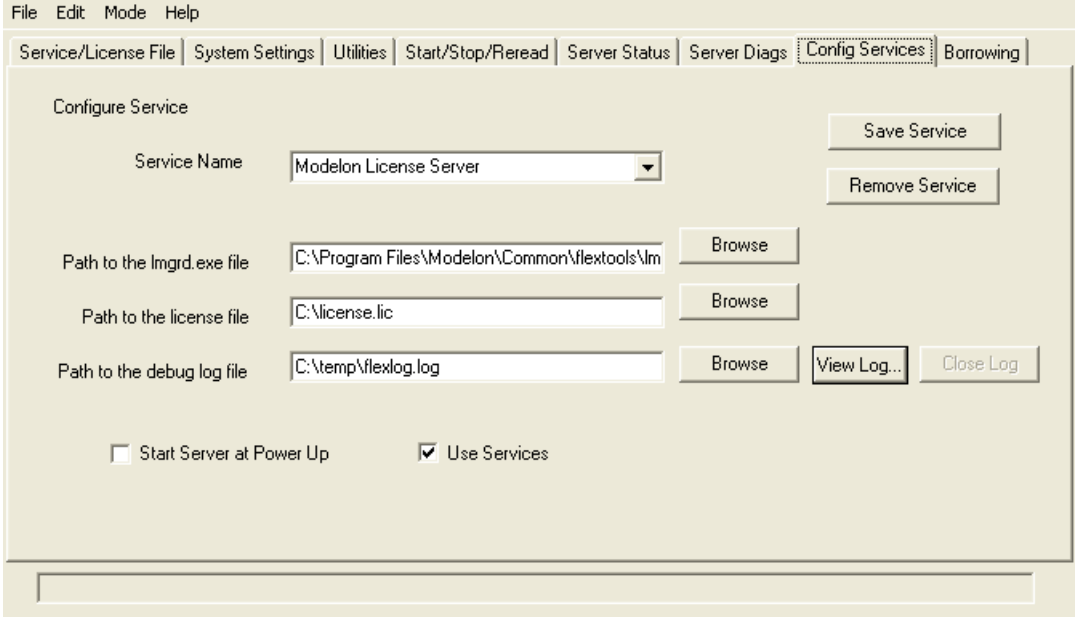


Figure A.2 Setup the license server with lmtools.exe

11.Click the **Service/License File** tab. Select the service name from the list presented in the selection box. In this example, the service name is `Modelon License Server`.

12.Click the **Start/Stop/Reread tab**.

13.Start `Modelon License Server` by clicking the **Start Server** button. `Modelon License Server` license server starts and writes its debug log output to the file specified in the **Config Services** tab.

A.4.3. Installation on Unix

In the `<installation folder>\license_tools` folder that is distributed with the product, you can find the files listed below.

- *lmgrd* (license server)
- *modelon* (license daemon)
- *lmutil* (configure- and utility functions)

Before you start the license server, *lmgrd*, make sure that license daemon *modelon* is in the same folder.

Start `lmgrd` from the UNIX command line using the following syntax:

```
lmgrd -c license_file_list -L [+]debug_log_path
```

where `license_file_list` is either the full path to a license file or a directory containing license files where all files named `*.lic` are used. If the license_file_list value contains more than one license file or directory, they must be separated by colons. `debug_log_path` is the full path to the debug log file. Prepending `debug_log_path` with the + character appends logging entries.

Starting `lmgrd` from a root account my introduce security risks, and it is therefore recommended that a non-root account is used instead. If `lmgrd` must be started by the root user, use the `su` command to run `lmgrd` as a non-privileged user:

```
su username -c "lmgrd -c license_file_list -l debug_log_path"
```

Ensure that the vendor daemons listed in the license file have execute permissions for `username`.

A.5. License borrowing

License borrowing allows users to check out features from a license server for offline use for a limited period of time. License borrowing is only enabled if it was specified in the license order. If you would have a server license and want to enable license borrowing, contact `<sales@modelon.com>`.

A.5.1. Borrowing licenses for offline use

License borrowing is initiated with the `lmborrow` utility in `lmutil`, distributed with your modelon product. License borrowing must be initiated while having a connection to the license server. `lmborrow` is invoked called from the command line, and the command syntax is

```
lmutil lmborrow modelon enddate [endtime]
```

The end date specifies for how long the license should be borrowed, and is on the form *dd-mmm-yyyy*, e.g. *01-dec-2016*. The end time of the borrow period is optional, and if given it should be on the form *hh:mm*. If no end time is specified, it defaults to 23:59, meaning that the license is borrowed for the full duration of the end date.

The default limit on the borrow time for Modelon products is 420 hours (2.5 weeks) from when license borrowing is initiated. License administrators may lower this limit, but it cannot be increased.

Once license borrowing is initiated, and until the end of the day that it is initiated, any license for a Modelon product that is checked out will be borrowed for offline use until the end of the specified borrow period. Each license will be borrowed only once even if it is checked out multiple times after initiating license borrowing.

In order to prevent borrowing of licenses that are not needed, the borrow status should be cleared after checking out the licenses you need. As with initiating borrowing, clearing it is done with the `lmborrow` utility by issuing the command

```
lmutil lmborrow -clear
```

Clearing the borrow status does *not* return any borrowed licenses.

To see which licenses are currently borrowed on your system, run the command

```
lmutil lmborrow -status
```

This will display all licenses currently borrowed, along with the end dates for their borrow periods.

A.5.2. Returning a borrowed license

Borrowed licenses can be returned before the specified end date by running

```
lmutil lmborrow -return featurename
```

`featurename` is the name of the specific feature in the license file, as shown by `lmborrow -status`. A single product can contain multiple features, which have to be returned separately. Note that a connection to the license server must exist in order to return borrowed licenses.

A.5.3. Options for the Vendor Daemon

The following options for the vendor daemon can be used to configure license borrowing.

A.5.3.1. BORROW_LOWWATER

Specifies a number of licenses that can not be borrowed. This will ensure that there are always at least BORROW_LOWWATER licenses to use on the license server.

The option syntax is

```
BORROW_LOWWATER feature n
```

where *feature* is the name of the feature to limit and *n* is the amount of non-borrowable licenses for that feature.

A.5.3.2. EXCLUDE_BORROW

Specify a user or group who can not borrow licenses for a certain feature. EXCLUDE_BORROW takes precedence over INCLUDE_BORROW, meaning that in case conflicting options for inclusion and exclusion are set, the ones for exclusion will be used.

The option syntax is

```
EXCLUDE_BORROW feature type name
```

where *feature* is the feature to prevent user(s) from borrowing, *type* is one of USER, HOST, DISPLAY, INTERNET, PROJECT, GROUP or HOST_GROUP, and *name* is the name of the item to be excluded.

A.5.3.3. INCLUDE_BORROW

Specify a user or group who can borrow licenses for a certain feature. If options specified here conflict with those for EXCLUDE_BORROW, the options for exclusion are used.

The option syntax is the same as for EXCLUDE_BORROW.

A.5.3.4. MAX_BORROW_HOURS

Set the longest allowed borrow period. The default for Modelon products is 420 hours (2.5 weeks), and it is only possible to specify a lower max value than this.

The option syntax is

```
MAX_BORROW_HOURS feature n
```

where *feature* is the feature to set a borrow period limit for, and *n* is the limit, in hours.

A.6. Troubleshooting license installation

If you experience any problems with the license, the error messages are usually descriptive enough to provide hints as to the root cause of the problem. If the problem persists, please contact Modelon at <support@modelon.com>. Before contacting Modelon, support you should run `lmdiag` and provide the resulting information. Follow the instructions below to run `lmdiag`.

A.6.1. Running lmdiag

- Windows

  1. Open **cmd**

     Windows 7 and Vista

     a. Click the **Start** button.

     b. Type **cmd** in the search bar and press enter.

     Windows XP

     a. Click the **Start** button.

     b. Click on **Run...**.

     c. Type **cmd** in the text box and click **OK**.

  2. Run `lmdiag.exe`.

     Type the full path to `lmdiag.exe` within quotes and press enter. `lmdiag.exe` is normally located in `<installation folder>\license_tools\lmdiag.exe`.

- Unix

  - Open a terminal and change directory to the `<installation folder>/license_tools/`.

    Run `lmdiag` with the `./lmutil lmdiag` command.

  Note: `lmutil` requires LSB (Linux Standard Base) compliance to run. Some distributions, e.g. Ubuntu, do not have LSB compliance by default and can thus not run the program. `./lmutil` then fails with a message like

  ```
  $> ./lmutil lmdiag
   bash: ./lmutil: No such file or directory
  ```

  If this error occurs, please check if the required interpreter is installed on your system. The requirement can be found with the `readelf` command, and the output should look similar to

```
$> readelf -a lmutil | grep interpreter
    [Requesting program interpreter: /lib64/ld-lsb-x86-64.so.3]
```

LSB should be available for install through a package manager. If installing it is not an alternative, a quick fix is to symlink the required interpreter to the one on your system, i.e.

```
$> ln -s <your ld> <required ld>
```

Note 2: FlexLM requires that network devices are named *eth0*, *eth1*, etc. When other names are used, `lmhostid` will always return 0 as host ID. Device names can be shown with the `ifconfig` command. If your Linux distribution uses a different naming scheme, it needs to be changed. The steps to change the naming scheme depend on the distribution and release.

Appendix B. Compiler options

B.1. List of options that can be set in compiler

Table B.1 Compiler options

Option	Option type / Default value	Description
`allow_discrete_switches`	`boolean` / `true`	If enabled, then discrete switches are allowed in the model.
`allow_discrete_variables`	`boolean` / `true`	If enabled, then discrete variables such as Integers, Booleans Strings or Enumerations are allowed in the model.
`allow_library_version_ mismatch`	`boolean` / `false`	If enabled, versions given in uses-annotations do not need to match the version in the used library. If there are uses-annotations that specify different versions of the same library, then only the first loaded will be used. Warnings will be given instead of errors.
`allow_when_clauses`	`boolean` / `true`	If enabled, then when-clauses are allowed in the model.
`api_allow_missing_elements`	`boolean` / `true`	If the API should allow getting only partial information from component and classes due to missing classes. The API will otherwise throw ModelicaNotFoundException if any of the extended classes are missing.
`automatic_tearing`	`boolean` / `true`	If enabled, then automatic tearing of equation systems is performed.
`check_inactive_ contitionals`	`boolean` / `false`	If enabled, check for errors in inactive conditional components when compiling. When using check mode, this is always done.
`component_names_in_errors`	`boolean` / `true`	If enabled, the compiler will include the name of the component where the error was found, if applicable.
`convert_free_dependent_ parameters_to_algebraics`	`boolean` / `true`	If enabled, then free dependent parameters are converted to algebraic variables.
`divide_by_vars_in_tearing`	`boolean` / `false`	If enabled, a less restrictive strategy is used for solving equations in the tearing algorithm. Specifically, division by parameters and variables is permitted, by default no such divisions are made during tearing.

Option	Option type / Default value	Description
enable_block_function_ extraction	boolean / false	Looks for function calls in blocks. If a function call in a block doesn't depend on the block in question, it is extracted from the block.
enable_lazy_evaluation	boolean / false	If this option is set to true (default is false), then the set of equation blocks that are evaluated with respect to the value of a dependent variable is limited to the minimal set.
event_indicator_structure	boolean / false	If enabled, additional event indicator dependency information is written to the model description in the FMU. This is relevant for, e.g., solvers of QSS type. Only valid for FMI 2.0.
event_output_vars	boolean / false	If enabled, output variables are added to model description for each generated state event indicator.
expose_scalar_equation_ blocks_in_interactive_fmu	boolean / false	If enabled, unsolved scalar equations will be exposed to the external solver when generating interactive FMU.
external_constant_ evaluation	integer / 5000	Time limit (ms) when evaluating constant calls to external functions during compilation. 0 indicates no evaluation. -1 indicates no time limit.
filter_warnings	string / ''	A comma separated list of warning identifiers that should be omitted from the logs.
generate_block_jacobian	boolean / false	If enabled, then code for computing block Jacobians is generated. If blocks are needed to compute ODE Jacobians they will be generated anyway.
generate_html_diagnostics	boolean / false	If enabled, model diagnostics are generated in HTML format. This includes the flattened model, connection sets, alias sets and BLT form.
generate_html_diagnostics_ output_directory	string / '.'	Path to directory where compiler output should be generated for the option 'generate_html_diagnostics'. Default value is '.'. Directory is created if it doesn't exist. The html diagnostics is generated in a subdirectory of this directory.
generate_json_statistics	boolean / false	If enabled, model statistics are generated in JSON format.
generate_json_statistics_ output_directory	string / '.'	Path to directory where compiler output should be generated for the option 'generate_json_statistics'. Default value is '.'. Directory is created if it doesn't exist.

Option	Option type / Default value	Description
		The JSON statistics is generated in a subdirectory of this directory.
`generate_ode_jacobian`	`boolean` / `false`	If enabled, then code for computing ODE Jacobians is generated.
`generate_svg_metadata_tag`	`boolean` / `true`	Controls if the metadata tag is added during SVG generation. The tag has information about connectors, such as the name and coordinates
`halt_on_warning`	`boolean` / `false`	If enabled, compilation warnings will cause compilation to abort.
`hand_guided_tearing`	`boolean` / `false`	If enabled, hand guided tearing of equation system is performed.
`html_diagnostics_contents`	`string` / `'statistics'`	Space-separated list of content that will be generated if option '`generate_html_diagnostics`' is enabled. Supported values: `'statistics'`,`'build'`, `'model'` and `'full'`. Value `'statistics'` is the standard diagnostics. Value `'build'` is build information including compiler version, platform information, dependent libraries. Value `'model'` is model documentation. Value `'full'` implies all the other values. Default is `'statistics'`.
`index_reduction`	`boolean` / `true`	If enabled, then index reduction is performed for high-index systems.
`init_nonlinear_solver`	`enumeration` / `kinsol`	Decides which nonlinear equation solver to use in the initial system. Alternatives are `'kinsol'`, `'realtime'` or `'minpack'` (DEPRECATED).
`inline_integration`	`boolean` / `false`	If enabled, inline integration is applied.
`interactive_fmu`	`boolean` / `false`	If enabled, the DAE system is converted into an interactive FMU where all residual equations and iteration variables have been changed into top level outputs and inputs.
`merge_blt_blocks`	`boolean` / `false`	If this option is set to `true` (default is `false`), BLT blocks will be constructed so that all level one HGT pairs and all unpaired HGT will reside inside the same BLT block.
`msvs_path`	`string` / `''`	Path to the Microsoft Visual Studio Compiler to compile C code with. Will cause compilation to fail if no installation is found.

Option	Option type / Default value	Description
msvs_version	enumeration /	Microsoft Visual Studio Compiler version or year edition to compile C code with. If this option is not set, then the first installation found when searching the default install locations will be used. The default install locations are searched in the following order: 2012 (11.0), 2010 (10.0) (DEPRECATED), 2015 (14.0), 2017 (15.0), 2019 (16.0), 2022 (17.0), 2013 (12.0). If the msvs_path option is not set, then this option will be used to find an installation among the default install locations. If the msvs_path option is set then only that location will be searched and this option will be used to verify the compiler version of the found version. In both cases the compilation will fail if the installation found mismatches the version specified with this option.
nonlinear_solver	enumeration / kinsol	Decides which nonlinear equation solver to use. Alternatives are 'kinsol', 'realtime' or 'minpack' (DEPRECATED).
package_html_documentation	boolean / false	If enabled, then all diagnostics generated via use of options 'generate_html_diagnostics' and 'html_diagnostics_contents' will also be packaged as documentation in the FMU. If you are working with unencrypted libraries with protection annotations it is recommended that option 'treat_libraries_as_encrypted' is also enabled to limit the packaged documentation what would be packaged if the libraries were encrypted.
smoothness_check_as_ warnings	boolean / false	If enabled, then smoothness check problems are given as warning instead of errors. Otherwise problems found are given as errors.
state_initial_equations	boolean / false	If enabled, the compiler ignores initial equations in the model and adds parameters for controlling initial values of states. Default is false.
state_start_values_fixed	boolean / false	If enabled, then initial equations are generated automatically for differentiated variables even though the fixed attribute is equal to fixed. Setting this option to true is, however, often practical in optimization problems.
system_continuity_order	integer / -1	Require the model to be at least n times continuous differentiable where n is an integer given by this option. De-

Option	Option type / Default value	Description
		fault value is -1 which means that discontinuations are allowed.
tearing_division_tolerance	real / 1.0E-10	The minimum allowed size for a divisor constant when performing tearing.
time_events	boolean / true	If enabled, operators are allowed to generate time events.
allow_multiple_residuals_in_hgt_blocks	string / 'level1'	Controls if HGT blocks are allowed to have more than one residual. 'false' - All HGT blocks can only have one residual. An error is given if there are HGT blocks with more than one residual. 'level1' - Only blocks with HGT level=1 are allowed to have more than one residual. An error is given if there are HGT blocks at level=2 (or higher) with more than one residual. As a consequence combining this option with merge_blt_blocks=true guarantees a single unsolved block in the FMU. 'true' - No error check is done, all HGT blocks may have multiple residuals.
allow_non_scalar_nested_blocks	boolean / true	If disabled, an error is given if there are nested blocks which are non-scalar.
automatic_add_initial_equations	boolean / true	If enabled, then additional initial equations are added to the model based equation matching. Initial equations are added for states that are not matched to an equation.
c_compiler	string / 'gcc'	The C compiler to use to compile generated C code.
cc_extra_flags	string / ':O1'	Optimization level for c-code compilation
cc_extra_flags_applies_to	enumeration / functions	Parts of c-code to compile with extra compiler flags specified by ccompiler_extra_flags.
cc_split_element_limit	integer / 1000	When generating code for large systems, the code is split into multiple functions and files for performance reasons. This option controls how many scalar elements can be evaluated by a function. Value less than 1 indicates no split.
cc_split_function_limit	integer / 20	When generating code for large systems, the code is split into multiple functions and files for performance reasons. This option controls how many functions can be generated in a file. Value less than 1 indicates no split.
cc_split_function_limit_globals	integer / 200	When generating code for large systems, the code is split into multiple functions and files for performance reasons.

Option	Option type / Default value	Description
		This option controls how many functions can be generated in a file for initialization of global variables. Value less than 1 indicates no split.
`cc_string_literal_limit`	`integer` / `10000`	Limit the length of any string literal in the generated C code to at most this length, in characters. 0 means no limit. The C specification states that any compliant C compiler must support string literals at least 509 characters long. Setting this option to 509 will generate C code that is compliant with any standards-compliant C compiler. Most major compilers have higher or no practical limit.
`check_local_balancing`	`boolean` / `false`	If enabled, the compiler will check local model balance. If a model is not balanced an error will be reported.
`common_subexp_elim`	`boolean` / `true`	If enabled, the compiler performs a global analysis on the equation system and extract identical function calls into common equations.
`constant_parameters`	`boolean` / `false`	Fixed parameters except external objects will be treated as constants. A dependent parameter is only changed to constant if the binding equation or start value can be treated as constant. The start value is used when there is no binding equation. See the options `constant_parameters_include`, `constant_parameters_include_from`, `constant_parameters_skip` and `constant_parameters_skip_from` for finer control of which parameters are changed to constants. The default is that all parameters are changed.
`constant_parameters_ include`	`string` / `''`	Name pattern(s) of parameter variables that should be made constants with `constant_parameters`. These parameters will be changed to constants. A dependent parameter is only changed to constant if the binding equation or start value can be treated as constant. This option overrides the selection in `constant_parameters_skip` to provide finer control over which parameters are made constants. Multiple name patterns are separated by spaces.
`constant_parameters_ include_from`	`string` / `''`	Path to file (absolute, relative, file URI, or modelica URI) which contains name patterns (globs) of parameter variables that are made constants with

Option	Option type / Default value	Description
		constant_parameters. The contents of the file are treated as input to the constant_parameters_include option. Multiple name patterns are separated by spaces.
constant_parameters_skip	string / ''	Name patterns (globs) of parameters to be excluded/skipped from the effect of constant_parameters, and therefore will remain parameters. The pattern operates on the qualified name of the component. This behavior can be overridden by the selection from constant_parameters_include and constant_parameters_include_from. A dependent parameter is only changed to constant if the binding equation or start value can be treated as constant. Multiple name patterns are separated by spaces. Individual elements in primitive arrays cannot be excluded.
constant_parameters_skip_from	string / ''	Path to file (absolute, relative, file URI, or modelica URI) which contains name patterns (globs) of variables to be excluded/skipped from the effect of constant_parameters and not made constants. The contents of the file are treated as the input to the constant_parameters_skip option.
convert_to_input	string / ''	Name pattern(s) of parameters to be converted to input variables. Multiple name patterns are separated by spaces. Only independent fixed parameters, that are not used in a context where only parameters are allowed, can be converted. Patterns are GLOB patterns, for more information see description of the 'exclude_internal_variables' option in the users guide, in 'Obfuscating Variables in a Functional Mock-up Unit'.
convert_to_input_from	string / ''	Path to file (absolute, relative, file URI, or modelica URI) that contains name pattern(s) of parameters to be converted to input variables. Multiple name patterns are separated by spaces. Only independent fixed parameters, that are not used in a context where only parameters are allowed, can be converted. Patterns are GLOB patterns, for more information see description of the 'exclude_internal_variables' option in the users

Option	Option type / Default value	Description
		guide, in 'Obfuscating Variables in a Functional Mock-up Unit'.
`diagnostics_limit`	`integer` / `500`	This option specifies the equation system size at which the compiler will start to reduce model diagnostics. This option only affects diagnostic output that grows faster than linear with the number of equations.
`dynamic_states`	`boolean` / `true`	If enabled, dynamic states will be calculated and generated.
`dynamic_states_limit`	`integer` / `10`	Limit for size of dynamic state sets. Value < 0 indicates infinite. Value == 0 indicates disabled.
`eliminate_alias_constants`	`boolean` / `true`	If enabled, then alias constants are eliminated from the model.
`eliminate_alias_parameters`	`boolean` / `true`	If enabled, then alias parameters are eliminated from the model.
`eliminate_alias_variables`	`boolean` / `true`	If enabled, then alias variables are eliminated from the model.
`eliminate_flow_equations`	`boolean` / `true`	Breaks up algebraic loops in flow-like equations. Flow-like equations are linear equations with coefficients that are +1, -1, or 0. This includes the flow equations generated from connection sets.
`eliminate_linear_equations`	`boolean` / `true`	If enabled, then equations with linear sub expressions are substituted and eliminated.
`enable_modelon_istoplevel_annotation`	`boolean` / `false`	If enabled, annotating a connector class with (__Modelon(isTopLevel=`true`)) causes instances of it to always be treated as top-level connectors, even if declared in a model that is not top-level itself.
`enable_structural_ diagnosis`	`boolean` / `true`	If enabled, structural error diagnosis based on matching of equations to variables is used.
`enable_variable_scaling`	`boolean` / `false`	If enabled, then the `'nominal'` attribute will be used to scale variables in the model.
`equation_sorting`	`boolean` / `true`	If enabled, then the equation system is separated into minimal blocks that can be solved sequentially.
`exclude_internal_variables`	`string` / `''`	Name pattern(s) of internal variables to be excluded from the compilation target. Overridden by selection from `include_internal_variables` and

Option	Option type / Default value	Description
		include_internal_variables_from. Multiple name patterns are separated by spaces. More documentation can be found in the users guide in 'Obfuscating Variables in a Functional Mock-up Unit'.
exclude_internal_variables_from	string / ''	Path to file (absolute, relative, file URI, or modelica URI) which contains name pattern(s) of internal variables to be excluded from the compilation target. Overridden by selection from include_internal_variables and include_internal_variables_from. Multiple name patterns are separated by spaces and new lines. More documentation can be found in the users guide in 'Obfuscating Variables in a Functional Mock-up Unit'.
expose_temp_vars_in_fmu	boolean / false	If enabled, then all temporary variables are exposed in the FMU XML and accessible as ordinary variables
external_constant_evaluation_dynamic	boolean / true	If enabled, calls to external functions will be evaluated during compilation using a pre-compiled program (instead of generating and compiling one), if possible.
external_constant_evaluation_max_proc	integer / 10	The maximum number of processes kept alive for evaluation of external functions during compilation. This speeds up evaluation of functions using external objects during compilation.If less than 1, no processes will be kept alive, i.e. this feature is turned off.
extra_resources	string / ''	List of filenames (absolute or relative to working directory, separated by the platform path separator) of files to be included in the resources/extra directory in the generated FMU.
extra_resources_from	string / ''	Path to a file (absolute or relative to working directory) which contains a list of filenames (absolute or relative to working directory, separated by newline) of files to be included in the resources/extra directory in the generated FMU.
function_incidence_ computation	string / 'none'	Controls how matching algorithm computes incidences for function call equations. Possible values: 'none', 'all'. With 'none' all outputs are assumed to depend on all inputs. With 'all' the compiler analyses the function to determine dependencies.

Option	Option type / Default value	Description
function_inverses	boolean / true	If enabled, the compiler will use the inverse annotation to find function inverses when solving equations.
generate_runtime_option_parameters	boolean / true	If enabled, generate parameters for runtime options. Should always be true for normal compilation.
homotopy_type	enumeration / actual	Decides how homotopy expressions are interpreted during compilation. Can be set to either 'simplified' or 'actual' which will compile the model using only the simplified or actual expressions of the homotopy() operator.
ignore_within	boolean / false	If enabled, ignore within clauses both when reading input files and when error-checking.
include_internal_variables	string / ''	Name pattern(s) of internal variables to be included in the compilation target. Overrides selection from exclude_internal_variables and exclude_internal_variables_from. Multiple name patterns are separated by spaces. More documentation can be found in the users guide in 'Obfuscating Variables in a Functional Mock-up Unit'.
include_internal_variables_from	string / ''	Path to file (absolute, relative, file URI, or modelica URI) which contains name pattern(s) of internal variables to be included in the compilation target. Overrides selection from exclude_internal_variables and exclude_internal_variables_from. Multiple name patterns are separated by spaces and new lines. More documentation can be found in the users guide in 'Obfuscating Variables in a Functional Mock-up Unit'.
include_protected_ variables	boolean / false	Includes protected variables in the compilation target interface if the protection annotation on the class allows viewing variables.
inline_functions	enumeration / trivial	Controls what function calls are inlined. 'none' - no function calls are inlined. 'trivial' - inline function calls that will not increase the number of variables in the system. 'all' - inline all function calls that are possible.
inline_integration_method	string / 'ImplicitEuler'	This option controls the method to use together with inline integration.

Option	Option type / Default value	Description
keep_variables	string / ''	Name pattern(s) of variables that should not be obfuscated by renaming. Overrides selection from obfuscate_variables and obfuscate_variables_from. Multiple name patterns are separated by spaces. More documentation can be found in the users guide in 'Obfuscating Variables in a Functional Mock-up Unit'.
keep_variables_from	string / ''	Path to file (absolute, relative, file URI, or modelica URI) which contains name pattern(s) of variables that should not be obfuscated by renaming. Overrides selection from obfuscate_variables and obfuscate_variables_from. Multiple name patterns are separated by spaces and new lines. More documentation can be found in the users guide in 'Obfuscating Variables in a Functional Mock-up Unit'.
language	enumeration / Modelica	The language to compile for.
load_resources_at_runtime	boolean / false	If enabled, then 'Modelica.Utilities.Files.loadResource' loads resources at runtime, else at compilation. Resources loaded at compilation get packaged and can't be changed later. Resources loaded at runtime can be edited and their URI can be changed via their corresponding parameter. During runtime, resource loading with the 'modelica' scheme requires that all loaded libraries exist on the exact same absolute file path as they did during compilation.
local_iteration_in_tearing	enumeration / off	This option controls whether equations can be solved local in tearing. Possible options are: 'off', local iterations are not used (default). 'annotation', only equations that are annotated are candidates. 'all', all equations are candidates.
mathematical_domain_checks	boolean / true	If enabled, all mathematical operators will be checked for their correct domains and provide log messages when an operator is evaluated outside its correct domain.
max_n_proc	integer / 4	The maximum number of processes used during c-code compilation.
normalize_minimum_time_problems	boolean / true	If enabled, then minimum time optimal control problems encoded in Optimica are converted to fixed interval prob-

Option	Option type / Default value	Description
		lems by scaling of the derivative variables. Has no effect for Modelica models.
obfuscate_variables	string / ''	Name pattern(s) of variables to be obfuscated by renaming. Overridden by selection from keep_variables and keep_variables_from. Multiple name patterns are separated by spaces. More documentation can be found in the users guide in 'Obfuscating Variables in a Functional Mock-up Unit'.
obfuscate_variables_from	string / ''	Path to file (absolute, relative, file URI, or modelica URI) which contains name pattern(s) of variables to be obfuscated by renaming. Overridden by selection from keep_variables and keep_variables_from. Multiple name patterns are separated by spaces and new lines. More documentation can be found in the users guide in 'Obfuscating Variables in a Functional Mock-up Unit'.
propagate_derivatives	boolean / true	If enabled, the compiler will try to replace ordinary variable references with derivative references. This is done by first finding equations on the form x = der(y). If possible, uses of x will then be replaced with der(x).
show_error_location	enumeration / class	What location information to show for errors and warnings. Relative to file, class or both.
time_state_variable	boolean / false	If enabled, time is treated as a regular state variable. This will add a variable for time and an equation der(time) = 1.
variability_propagation	boolean / true	If enabled, the compiler performs a global analysis on the equation system and reduces variables to constants and parameters where applicable.
variability_propagation_algorithms	boolean / false	If enabled, the compiler includes modelica algorithms in variability propagation.
variability_propagation_external	boolean / true	If enabled, the compiler allows external constant evaluation during variability propagation.
variability_propagation_initial	boolean / true	If enabled, the compiler performs a global analysis on the initial equation system and reduces initial parameters to constants and parameters where applicable.

Option	Option type / Default value	Description
`variability_propagation_initial_partial`	`boolean` / `false`	If enabled, the compiler allows partial constant evaluation of function calls in initial equations during variability propagation.
`write_iteration_variables_to_file`	`boolean` / `false`	If enabled, two text files containing one iteration variable name per row is written to disk. The files contains the iteration variables for the DAE and the DAE initialization system respectively. The files are output to the resource directory of the FMU.
`write_tearing_pairs_to_file`	`boolean` / `false`	If enabled, two text files containing tearing pairs is written to disk. The files contains the tearing pairs for the DAE and the DAE initialization system respectively. The files are output to the working directory.
`algorithms_as_functions`	`boolean` / `false`	If enabled, convert algorithm sections to function calls.
`causal_ports`	`boolean` / `false`	Treat top level connector variables as causal variables. Flow and instream variables as inputs, potential and stream variables as outputs.
`delayed_scalarization`	`boolean` / `true`	If enabled, delays scalarization of parameter arrays until C-code generation. This reduces the memory used during transformation of the flattened tree.
`disable_smooth_events`	`boolean` / `false`	If enabled, no events will be generated for smooth operator if order equals to zero.
`event_indicator_scaling`	`boolean` / `false`	If enabled, event indicators will be scaled with nominal heuristics.
`experimental_enable_removing_modifications`	`boolean` / `false`	Enable prototype implementation of MCP-0009 Removing modifications. This implementation has known limitations and likely additional bugs, use at your own discretion.
`experimental_enable_selective_model_extension`	`boolean` / `false`	Enable prototype implementation of MCP-0032 Selective model extension. This implementation has known limitations and likely additional bugs, use at your own discretion.
`flatten_only`	`boolean` / `false`	Exit the compilation process after flattening but before optimization of the flattened modelica code. This means no FMU is generated. This option can be used together with the `log_dependencies` option to exit the compilation process after the logging of dependencies.

Option	Option type / Default value	Description
fmi_ports	boolean / false	If enabled, XML code for FMI ports is generated.
generate_event_switches	boolean / true	If enabled, event generating expressions generates switches in the c-code. Setting this option to false can give unexpected results.
generate_sparse_block_ jacobian_threshold	integer / 100	Threshold for when a sparse Jacobian should be generated. If the number of torn variables is less than the threshold a dense Jacobian is generated.
log_dependencies	string / ''	Log dynamic dependencies in JSON format to the given file. Prints information on parsed Modelica files and instantiated models in JSON format to the file with the given (absolute or relative) filename. This option can be used together with the flatten_only option because the compilation process is exited after the logging of dependencies.
msl_version	enumeration / auto	Controls what version of the Modelica Standard Library will be automatically added to MODELICAPATH. Allowed values: "auto" - All included versions will be added. This is the default. "none" - No MSL version will be added. "3.2.3" or "4.0.0" - Only that specific version will be added.
remove_unused_temporaries	boolean / true	If enabled then unused temporary variables are eliminated from the model.
source_code_fmu	boolean / false	If enabled, external source code is packaged with the FMU.
target_platform_packages_ directory	string / ''	If set, then specifices the path to the directory which contains additional packages necessary for compiling to the target platform.
treat_libraries_as_ encrypted	boolean / false	Treat all loaded libraries as if they were encrypted. Intended only for testing.
cs_profiling	boolean / false	Option for turning on profiling for Co-Simulation FMUs.
cs_rel_tol	real / 1.0E-6	Tolerance for the adaptive solvers in the Co-Simulation case.
cs_solver	integer / 0	Specifies the internal solver used in Co-Simulation. 0 - CVode, 1 - Euler, 2 - RungeKutta2, 3 - Radau5.

Option	Option type / Default value	Description
cs_step_size	real / 0.001	Step-size for the fixed-step solvers in the Co-Simulation case.
enforce_bounds	boolean / true	If enabled, min / max bounds are enforced for iteration variables in nonlinear equation blocks.
iteration_variable_scaling	integer / 1	Scaling mode for the iteration variables in the equation block solvers: 0 - no scaling, 1 - scaling based on nominals, 2 - utilize heuristic to guess nominal based on min, max, start, etc.
log_level	integer / 3	Log level for the runtime: 0 - none, 1 - fatal error, 2 - error, 3 - warning, 4 - info, 5 - verbose, 6 - debug.
log_start_time	real / -1.0E20	Time for when the user log level starts to take affect.
log_stop_time	real / 1.0E20	Time for when the user log level stops to take affect.
mathematical_domain_warnings_limit	integer / -1	If enabled, this option determines how many warnings from the mathematical domain checks will be printed to the log. A value of -1 means that there is no limit. The warnings count is reset after the initialization phase.
nle_active_bounds_mode	integer / 0	Mode for how to handle active bounds: 0 - project Newton step at active bounds, 1 - use projected steepest descent direction.
nle_jacobian_calculation_mode	integer / 0	Mode for how to calculate the Jacobian: 0 - onesided differences, 1 - central differences, 2 - central differences at bound, 3 - central differences at bound and 0, 4 - central differences in second Newton solve, 5 - central differences at bound in second Newton solve, 6 - central differences at bound and 0 in second Newton solve, 7 - central differences at small residual, 8 - calculate Jacobian externally, 9 - Jacobian compresssion.
nle_jacobian_finite_difference_delta	real / 1.49E-08	Delta to use when calculating finite difference Jacobians.
nle_jacobian_update_mode	integer / 2	Mode for how to update the Jacobian: 0 - full Jacobian, 1 - Broyden update, 2 - Reuse Jacobian.
nle_solver_default_tol	real / 1.0E-10	Default tolerance for the equation block solver.
nle_solver_exit_criterion	integer / 3	Exit criterion mode: 0 - step length and residual based, 1 - only step length based, 2 - only residual based, 3 - hybrid.

Option	Option type / Default value	Description
nle_solver_max_residual_ scaling_factor	real / 1.0E10	Maximal scaling factor used by automatic and hybrid residual scaling algorithm.
nle_solver_min_residual_ scaling_factor	real / 1.0E-10	Minimal scaling factor used by automatic and hybrid residual scaling algorithm.
rescale_after_singular_jac	boolean / true	If enabled, scaling will be updated after a singular jacobian was detected (only active if automatic scaling is used).
rescale_each_step	boolean / false	If enabled, scaling will be updated at every step (only active if automatic scaling is used).
residual_equation_scaling	integer / 1	Equations scaling mode in equation block solvers: 0 - no scaling, 1 - automatic scaling, 2 - manual scaling, 3 - hybrid, 4 - aggressive automatic scaling, 5 - automatic rescaling at full Jacobian update
runtime_log_to_file	boolean / false	If enabled, log messages from the runtime are written directly to a file, besides passing it through the FMU interface. The log file name is generated based on the FMU name.
use_Brent_in_1d	boolean / true	If enabled, Brent search will be used to improve accuracy in solution of 1D non-linear equations.
block_solver_profiling	boolean / false	If enabled, methods involved in solving an equation block will be timed.
events_default_tol	real / 1.0E-10	Default tolerance for the event iterations.
events_tol_factor	real / 1.0E-4	Tolerance safety factor for the event indicators. Used when external solver specifies relative tolerance.
nle_brent_ignore_error	boolean / false	If enabled, the Brent solver will ignore convergence failures.
nle_solver_check_jac_cond	boolean / false	If enabled, the equation block solver computes and log the jacobian condition number.
nle_solver_max_iter	integer / 100	Maximum number of iterations for the equation block solver.
nle_solver_max_iter_no_ jacobian	integer / 10	Maximum number of iterations without jacobian update. Value 1 means an update in every iteration.

Option	Option type / Default value	Description
nle_solver_min_tol	real / 1.0E-12	Minimum tolerance for the equation block solver. Note that, e.g. default Kinsol tolerance is machine precision pwr 1/3, i.e. 1e-6.
nle_solver_regularization_ tolerance	real / -1.0	Tolerance for deciding when regularization should be activated (i.e. when condition number > reg tol).
nle_solver_step_limit_ factor	real / 10.0	Factor limiting the step-size taken by the nonlinear block solver.
nle_solver_tol_factor	real / 1.0E-4	Tolerance safety factor for the equation block solver. Used when external solver specifies relative tolerance.
nle_solver_use_last_ integrator_step	boolean / true	If enabled, the initial guess for the iteration variables will be set to the iteration variables from the last integrator step.
nle_solver_use_nominals_as_ fallback	boolean / true	If enabled, the nominal values will be used as initial guess to the solver if initialization failed.
time_events_default_tol	real / 2.22E-14	Default tolerance for the time event iterations.
use_jacobian_equilibration	boolean / false	If enabled, jacobian equilibration will be utilized in the equation block solvers to improve linear solver accuracy.
use_newton_for_brent	boolean / true	If enabled, a few Newton steps are computed to get a better initial guess for Brent.
block_solver_experimental_ mode	integer / 0	Activates experimental features of equation block solvers
cs_experimental_mode	integer / 0	Activates experimental features of CS ode solvers.

Appendix C. Thirdparty Dependencies

C.1. Introduction

All dependencies needed to run OCT are bundled in the installer and listed in Section C.2.

C.2. Applications, Libraries and Python Packages in OCT

The following applications, libraries and Python packages are part of the OCT installation.

Applications

- JDK 17.0.2
- MinGW (tdm-gcc 5.1.0)
- Python 3.7.13
- Swig 3.0.8

Libraries

- Ipopt 3.12.4
- boost 1.72.0
- Beaver 0.9.6.1
- eXpat 2.1.0
- Minizip
- MSL (Modelica Standard Library), see Section C.3.
- SUNDIALS 2.7.0
- Zlib 1.2.6
- CasADi

Python packages

zipp 0.6.0
xlwings 0.15.10
wheel 0.33.6
wxPython 4.0.7.post1
widgetsnbextension 3.5.1
webencodings 0.5.1
wcwidth 0.1.7
urllib3 1.25.3
traitlets 4.3.2
tornado 6.0.3
testpath 0.4.2
terminado 0.8.2
six 1.12.0
serpent 1.28
Send2Trash 1.5.0
scipy 1.3.1
SALib 1.3.8
requests 2.22.0
qtconsole 4.5.5
pyzmq 18.1.0
pywinpty 0.5.5
pywin32 225
pywin32-ctypes 0.2.0
pytz 2019.2
pythonnet 2.4.0
python-jenkins 1.5.0
python-dateutil 2.8.0
pyserial 3.4
pyrsistent 0.15.4
Pyro4 4.76
pyparsing 2.4.2
Pygments 2.4.2
pyDOE2 1.2.1
psutil 5.6.3
prompt-toolkit 2.0.9
prometheus-client 0.7.1
pip 19.0.3
Pillow 6.2.1
pickleshare 0.7.5
pbr 5.4.3
parso 0.5.1
nose 1.3.7
nose-cov 1.6
nbformat 4.4.0
nbconvert 5.6.0
natsort 6.0.0
multi-key-dict 2.0.3
more_itertools 3.0.5
mock 3.0.5
mistune 0.8.4
matplotlib 3.1.1
MarkupSafe 1.1.1
lxml 4.4.1
kiwisolver 1.1.0
keyring 19.1.0
jupyter 1.0.0
jupyter-core 4.5.0
jupyter-console 6.0.0
jupyter-client 5.3.2
jsonschema 3.0.2
JPype1 0.7.0
Jinja2 2.10.1
jedi 0.15.1
jdcal 1.4.1
jcc 3.5
ipywidgets 7.5.1
ipython 7.8.0
ipython-genutils 0.2.0
ipykernel 5.1.2
importlib-metadata 0.23
idna 2.8
html5lib 1.0.1
et-xmlfile 1.0.1
entrypoints 0.3
defusedxml 0.6.0
decorator 4.4.0
Cython 0.29.13
cycler 0.10.0
coverage 4.5.4
cov-core 1.15.0
comtypes 1.1.7
colorama 0.4.1

pandocfilters 1.4.2
pandas 0.25.1
openpyxl 2.6.3
numpy 1.17.1
notebook 6.0.1
chardet 3.0.4
certifi 2019.9.11
bleach 3.1.0
backcall 0.1.0
attrs 19.1.0

C.3. Modelica Standard Library

C.3.1. Modelica Standard Library 3.2.3

For this release, Modelica Standard Library (MSL) version 3.2.3 build 3 with a number of patches applied is used. The patches are listed below, but can also be found on the Modelon fork of MSL:

- To the model *Modelica.Blocks.Examples.NoiseExamples.ActuatorWithNoise* defined in Modelica/Blocks/package.mo a StateSelect.always is added for Controller.y1. With this patch dynamic state selection is avoided. See also the reported issue 2189 on the GitHub repository for Modelica Association.

- The model *Modelica.Electrical.Analog.Examples.SimpleTriacCircuit* is patched to with a looser tolerance to improve numerical stability near certain event points.

- The state selection in *Modelica.Magnetic.FluxTubes.Examples.Hysteresis.HysteresisModelComparison* is patched to improve the numerical robustness, see issue 2248 on the GitHub repository for Modelica Association.

- In *Modelica.Fluid.Examples.TraceSubstances.RoomCO2WithControls* the experiment tolerance is tightened to 1e-008 instead of 1e-006 to avoid chattering.

- In *Modelica.Magnetic.FluxTubes.Examples.Hysteresis.SinglePhaseTransformerWithHysteresis1* an initial equation has been added in order to fully specify the initial system, see issue 3409 on the GitHub repository for Modelica Association.

- In *Modelica.Electrical.Machines.Examples.AsynchronousInductionMachines.AIMC_InverterDrive* the connection to ground has been relocated from the rectifier to the inductor in order to simplify the set of linear equations for proper state selection.

- In *Modelica.Electrical.PowerConverters.Examples.ACDC.RectifierBridge2mPulse.DiodeBridge2mPulse* the experiment tolerance is tightened to 3e-007 instead of 1e-006 to avoid infinite loops due to numerical noise.

- In Modelica/Media/package.mo array sizes are specified since OCT does not support this type of unspecified array sizes. This affects the models Modelica.Media.Examples.ReferenceAir.MoistAir, Modelica.Media.Examples.ReferenceAir.MoistAir1 and Modelica.Media.Examples.ReferenceAir.MoistAir2.

- In Modelica/Resources/C-Sources/ModelicaStandardTablesUsertab.c it is removed weak symbol linking for MacOS since MacOS doesn't support it. This affects the models using ModelicaStandardTables.

- The component *m* has been removed from models *Modelica.Electrical.QuasiStationary.MultiPhase.Sources.FrequencySweepCurrentSource* and *Modelica.Electrical.QuasiStationary.MultiPhase.Sources.FrequencySweepVoltageSource* to avoid non-identical duplicate components. See also the reported issue 4021 on the GitHub repository for Modelica Association.

With the patches listed above applied, all example models in version 3.2.3 build 3 of MSL simulate correctly with OCT.

The following changes have also been applied in order to support 3D animations in Modelon Impact:

- In file MSL323/ModelicaServices/package.mo, a number of parameters / variables have been added.
- In file MSL323/ModelicaServices/package.mo, the equation *(x, y, z, C) = surfaceCharacteristic(nu,nv,multiColoredSurface)* has been added.
- In file MSL323/ModelicaServices/package.mo, the annotation *Protection(access=Access.hide)* has been added.
- In directory MSL323/Modelica/Resources/Data/Shapes/Engine, the three files *crank.glb*, *piston.glb* and *rod.glb* have been added. These are files converted from their respective dxf-format in that same directory.
- In directory MSL323/Modelica/Resources/Data/Shapes/RobotR3, the six files *b0.glb*, *b1.glb*, *b2.glb*, *b3.glb*, *b4.glb*, *b5.glb* and *b6.glb* have been added. These are files converted from their respective dxf-format in that same directory.

Several icons in the MSL bundle of version 3.2.3 are also updated visually such that they are using a gradient fill, this change is in line with the pull request at issue 3952.

C.3.2. Modelica Standard Library 4.0.0

For this release, Modelica Standard Library (MSL) version 4.0.0 with a number of patches applied is used. The patches are listed below, but can also be found on the Modelon fork of MSL:

- The method *ModelicaInternal_fullPathName* in *ModelicaInternal.c* is updated according to the reported issue issue 3660 on the GitHub repository for Modelica Association.
- To the model *Modelica.Blocks.Examples.NoiseExamples.ActuatorWithNoise* defined in Modelica/Blocks/package.mo a StateSelect.always is added for Controller.y1. With this patch dynamic state selection is avoided. See also the reported issue 2189 on the GitHub repository for Modelica Association.
- The model *Modelica.Electrical.Analog.Examples.SimpleTriacCircuit* is patched to with a looser tolerance to improve numerical stability near certain event points.
- The model *Modelica.Electrical.Analog.Examples.OpAmps.DifferentialAmplifier* is patched with an updated nominal to improve the scaling of the problem.

- The model *Modelica.Electrical.Machines.Examples.InductionMachines.IMC_DCBraking* is patched to improve simulation performance. The variables *imc.is[1]* and *imc.is[2]* now have *StateSelect.always*, *imc.is[3]* has *StateSelect.never*.

- The following functions were made impure: realFFTwriteToFile, initializeImpureRandom and readRealParameter in files Modelica/Math/FastFourierTransform.mo, Modelica/Math/Random.mo and Modelica/Utilities/Examples.mo respectively. The reason is that they contain calls to several impure functions. See also the reported issue 3857 on the GitHub repository for Modelica Association.

- The function *getMemory* in Modelica/Electrical/Digital.mo has been made impure since it contains a call to another impure function *Modelica.Utilities.Streams.readLine*. To account for this, *if initial()* has been replaced with *when initial* in both *Modelica.Electrical.Digital.Memories.DLATRAM* and *Modelica.Electrical.Digital.Memories.DLATROM*. See also the reported issue 3855 on the GitHub repository for Modelica Association.

- In *Modelica.Media.Examples.SolveOneNonlinearEquation.Inverse_sine* the call to function *print* has been put within an *initial equation* since *print* is an impure function. See also the reported issue 3856 on the GitHub repository for Modelica Association.

- In *Modelica.Electrical.PowerConverters.Examples.ACDC.RectifierBridge2mPulse.DiodeBridge2mPulse* the experiment tolerance is tightened to 3e-007 instead of 1e-006 to avoid infinite loops due to numerical noise.

- The parameters *Basic.LeakageWithCoefficient leakage* and *Modelica.Magnetic.QuasiStatic.FluxTubes.Basic.LeakageWithCoefficient leakage* in files *Magnetic/FluxTubes/Examples/BasicExamples/QuadraticCoreAirgap.mo* and *Magnetic/QuasiStatic/FluxTubes/Examples/BasicExamples/QuadraticCoreAirgap.mo* had their start values explicitly set to 1e-8 for numerical stability.

- In *Modelica/Utilities/Examples.mo* several calls to function *Modelica.Utilities.Streams.readMatrixSize* have been wrapped in *pure* since the function is impure.

- In *Modelica.Electrical.Machines.Examples.AsynchronousInductionMachines.AIMC_InverterDrive* the connection to ground has been relocated from the rectifier to the inductor in order to simplify the set of linear equations for proper state selection.

- The subpackage Modelica.Clocked is currently not supported and thus has been removed.

- In Modelica/Resources/C-Sources/ModelicaStandardTablesUsertab.c it is removed weak symbol linking for MacOS since MacOS doesn't support it. This affects the models using ModelicaStandardTables.

- In *Modelica.Fluid.Examples.TraceSubstances.RoomCO2WithControls* the experiment tolerance is tightened to 1e-008 instead of 1e-006 to avoid chattering.

- In Modelica/Media/package.mo array sizes are specified since OCT does not support this type of unspecified array sizes. This affects the models Modelica.Media.Examples.ReferenceAir.MoistAir, Modelica.Media.Examples.ReferenceAir.MoistAir1 and Modelica.Media.Examples.ReferenceAir.MoistAir2.

- The component *m* has been removed from models *Modelica.Electrical.QuasiStatic.Polyphase.Sources.FrequencySweepCurrentSource* and *Modelica.Electrical.QuasiStatic.Polyphase.Sources.FrequencySweepVoltageSource* to avoid non-identical duplicate components. See also the reported issue 4021 on the GitHub repository for Modelica Association.

The following changes have also been applied in order to support 3D animations in Modelon Impact:

- In file MSL400/ModelicaServices/package.mo, a number of parameters / variables have been added.
- In file MSL400/ModelicaServices/package.mo, the equation *(x, y, z, C) = surfaceCharacteristic(nu,nv,multiColoredSurface)* has been added.
- In file MSL400/ModelicaServices/package.mo, the annotation *Protection(access=Access.hide)* has been added.
- In directory MSL400/Modelica/Resources/Data/Shapes/Engine, the three files *crank.glb*, *piston.glb* and *rod.glb* have been added. These are files converted from their respective dxf-format in that same directory.
- In directory MSL400/Modelica/Resources/Data/Shapes/RobotR3, the six files *b0.glb*, *b1.glb*, *b2.glb*, *b3.glb*, *b4.glb*, *b5.glb* and *b6.glb* have been added. These are files converted from their respective dxf-format in that same directory.

Several icons in the MSL bundle of version 4.0.0 are also updated visually such that they are using a gradient fill, this change is in line with the pull request at issue 3952.

C.4. Additional Libraries in OCT

The following libraries are part of some builds of the OCT installation on Windows.

Libraries

- gson 2.8.9

C.5. Math Kernel Library (MKL)

Note that the generated FMUs and the solvers connected via the MATLAB® interface as well as the Python interface uses linear algebra functionality from Intel's Math Kernel Library (MKL). This means that there may be cases where the results of a solve/simulation can vary slightly between runs. To understand why this may occur and how to mitigate it, please see Obtaining Run-to-Run Numerical Reproducible Results.

Appendix D. Using External Functions in Modelica

D.1. Introduction

External functions to a Modelica model is described by the language specification (3.2r2 Section 12.9 External Function Interface). This appendix is intended to describe tool specific behaviour and common problems in relation to the external functions. OCT supports interfacing with both C and FORTRAN 77 functions.

D.2. External objects

The variability of external objects will be interpreted as parameter, even if the component is not declared with a variability prefix. This is due to the fact that the constructors should only be called once and a discrete or continuous external object would have to call its constructor many times. An external object may be dependent on fixed false parameters. In that case the constructor will be called during the solving of the initial system. External objects are not allowed in algebraic loops. If that is the case an error will be given during compilation.

D.3. LibraryDirectory

In addition to the base directory specified by the LibraryDirectory annotation the compiler looks for libraries in <base>/<arch> and <base>/<arch>/<comp> with higher priority for the more specific directories. If the function is intended to be used on multiple platforms or with multiple c-compilers there needs to be a specific version of the library file for each intended use. For example, if a library is to be used on both 32 and 64 bit windows, using both gcc and other c compilers, one would have to add several versions of the library. Each compiled specifically for each platform and compiler combination. Note that the version of the compiler is also specified, since different versions of the same compiler could be incompatible.

```
<base>/win32/gcc472/
<base>/win64/gcc472/
```

D.4. GCC

When compiling with GCC we use the -std=c89 flag. This means any header file included must conform to the c89 standard. A common issue is comments. c89 does not allow "//" comments, Only "/* */". When the header file include "//" comments the compilation will fail. The error message usually looks something like this:

```
sources/Test_Test1_funcs.c:4:1: error: expected identifier or '(' before '/' token
```

D.5. Microsoft Visual Studio

For a library built with Microsoft Visual Studio (VS) to be compatible with OCT it needs to be linked with the /MT option (as opposed to /MD). This will enable the static runtime linking which makes sure the corresponding VS runtime code is included in the FMU, rather than depending on a specific dll from a VS redistributable package to be installed. When the library was linked using /MD the compilation will fail. The error message usually looks something like this:

```
LINK : warning LNK4098: defaultlib 'MSVCRT' conflicts with use of
    other libs; use /NODEFAULTLIB:library
MSVCRT.lib(MSVCR120.dll) : error LNK2005: _printf already defined
    in LIBCMT.lib(printf.obj)
```

When using VS the corresponding directories are searched when looking for included libraries.

```
<base>/win32/vs2012/
<base>/win64/vs2012/
```

OCT also allow for tool specific libraries to be provided in <base>/<arch>/<comp>/oct. This is useful when the library is intended to be used in multiple tools which require specific tweaks which are not compatible, for example, linking options.

```
<base>/win32/vs2012/oct/
<base>/win64/vs2012/oct/
```

Appendix E. Release Notes

E.1. Release notes for the OPTIMICA Compiler Toolkit version 1.48.1

This release contains a minor internal fix compared to version 1.48.

E.2. Release notes for the OPTIMICA Compiler Toolkit version 1.48

E.2.1. Compiler Changes

Important improvements to the compiler:

- Added an error check to ensure a compiled simulation model is either a block or a model.
- Added a new check for input/output connection restrictions. A warning is raised if multiple signal sources are connected to the same target, producing an invalid connection set. This will be changed to an error in a future release.
- Resolved an issue where expandable connectors in if-equations would result in an *unknown program error* crash.
- Removed the obsolete FMUX compilation target.
- Removed warning about missing pure and impure keywords for external constructors and destructors.
- Equations of the form $x = -x$ now simplify to $x = 0$.
- The file *index.html* in the HTML diagnostics now contains additional *Expanded block information* references, linking to *blockInfo.html*.
- Blocks in *blockInfo.html* in the HTML diagnostics now contain links to the respective blocks in *blt.html* or *initBlt.html*.
- Warnings about C-strings longer than 509 characters are now suppressed during compilation.
- Changed so the local balancing check no longer checks components resulting in a less pedantic check in some cases.
- Fixed an issue with computing array sizes for arguments to functions.

- Deprecated target platform *win32*, use *win64* instead.
- Removed the deprecated "MSL400" and "MSL323" as values for *msl_version*. Use "4.0.0" and "3.2.3" instead.
- Deprecated the *OptimicaCompiler.jar* file and the *OptimicaCompiler* entry point class. Variants of OCT with optimization support should instead use the new compiler option *language* set to *Optimica*.
- Removed the deprecated compiler option *generate_mof_files*. The option *generate_html_diagnostics* should be used instead.
- The support for compiling with MSVS 2010 has been deprecated and will be removed in a future release. Consider switching to 2012 or newer to avoid future breaking changes.
- A list of called functions is included in the log when asserts events trigger.

The release contains a number of bug fixes to the compiler, among them:

- Resolved an issue with undefined references in source code FMUs.
- Stopped excluding *win32_dirent.c* from the sources list in source code FMUs.
- Fixed an issue where an inlined function call with constant arguments would lose its type when resulted in an empty array.
- Fixed an issue with the evaluation of if-expressions that could cause a crash.
- Fixed an issue where compiler would fail after scalarizing functions returning empty arrays.
- Fixed an issue when computing the variability for array constructors with iterators.
- Strings passed to external functions are now marked as *const*, this resolves incorrect warnings from the C-compiler (incompatible pointer types).
- Fixed an issue with compile-time evaluation of external functions that print to *stdout*.
- Fixed null pointer exception that could occur for function call with missing argument.

E.2.2. API Improvements

Important improvements to the compiler API:

- Fixed an issue where a library conversion could fail if it had changes for multiple top-level classes.
- Changed so that *Expression.getDeclaredText()* for the constructor of an operator record shows the expected expression.

E.2.3. Runtime Improvements

Runtime bug fixes:

- Fixed an issue where the simulation log could contain invalid XML.

E.2.4. Python Packages Improvements

The following fixes and additions have been made to the *DynamicDiagnostics* package:

- Fixed an issue that could cause *get_events_details* and *print_event_details* to fail with large log files.
- Fixed an issue causing *get_events_details* and *print_event_details* to not show *boolean* or *integer* based switches.
- Added new parameters to *get_events_details* and *print_event_details* to make various post-processing steps optional. E.g., removal of switches or events without effective discrete changes.
- Added a new function named *get_times_state_limits_step_length*. Returns points in time when a state error limited the step-size.
- Added an option to *get_number_of_times_state_limits_step_length* to count all states that would limit step size, instead of only the largest errors.

The PyFMI version has been updated from 2.10.3 to 2.11.0, this entails the following changes:

- Added *result_handling = None* as option, deprecated *none*.
- Calls to get continuous state derivatives when there are none will no longer result in FMU calls.
- It is now possible use *dynamic_diagnostics* with a custom result handler.
- The variable *pyfmi.common.diagnostics_prefix* has been moved to *pyfmi.common.diagnostics* as *DIAGNOSTICS_PREFIX*.

SteadyState has been updated with the following changes:

- The class *LogViewer* skips parsing of empty *SolverInvocation* XML elements generated from e.g., pre-propagation blocks.

Miscellaneous bug fixes:

- Corrected an issue with negated alias variables having the wrong sign for *pyjmi.symbolic_elimination.BLTOptimizationProblem*.

E.3. Release notes for the OPTIMICA Compiler Toolkit version 1.46

E.3.1. Compiler Changes

Important improvements to the compiler:

- Fixed so that variable dependencies from reinit equations are included under "OCT_StateEvents" in the model description.

E.3.2. API Improvements

Important improvements to the compiler API:

- Improved the message provided in AmbiguousNodeExceptions to be clearer and contain location information.
- Added platform constant for Linux 64.

E.4. Release notes for the OPTIMICA Compiler Toolkit version 1.44.4

E.4.1. The release contains a bug fix to the compiler:

Added support for deducing causality for primitive arrays assuming all elements should have the same causality.

E.5. Release notes for the OPTIMICA Compiler Toolkit version 1.44.3

E.5.1. The release contains a bug fix to the compiler:

Fixed an issue where a declared component in an expandable connector base class was removed despite being present.

E.6. Release notes for the OPTIMICA Compiler Toolkit version 1.44.2

E.6.1. Compiler Changes

The compiler can now deduce causality for expandable connector variables.

E.7. Release notes for the OPTIMICA Compiler Toolkit version 1.44

E.7.1. Compiler Changes

Important improvements to the compiler:

- Added an error for use of function, package or operator as a type for components.
- Added support for the *unbounded* attribute for Real variables.
- Changed such that nominals of continuous states now can depend on the initial system.

The release contains a number of bug fixes to the compiler, among them:

- Resolved a bug with optimization of records which caused a crash if the components of type equivalent records were of different orders.
- A warning for missing binding expression should no longer be incorrectly reported for *fixed=false* array parameters of size 0.
- Resolved an issue with the lookup of redeclared classes that could result in a *LOOKUP_NOT_FOUND* error.

Deprecations

- The solver '*minpack*' for compiler options '*init_nonlinear_solver*' and '*nonlinear_solver*' is now deprecated. The deprecation will be changed to a removal in version 1.48, use solver '*kinsol*' instead.
- API methods *experimental_getClassInConstrainedbyClause* and *experimental_hasConstrainedbyClause* are now deprecated, use *getConstrainingClass* instead.

E.7.2. API Improvements

Important improvements to the compiler API:

- Changed how conversion messages from libraries are presented for elements that cannot be automatically converted.
- Added *getConstrainingClass* to *InstanceClass* and *InstanceComponent* which returns the constraining class of that element.
- A library conversion should no longer crash when converting a modifier without a value, or when applying an empty conversion script.

- Fixed an issue with how a library conversion updates modifiers for class redeclares.

E.7.3. Runtime Improvements

Important improvements to the simulation framework:

- Improved the logging of events so that the expression of the event is added to the log.
- Improved the logging of assert events to also include the time of the event and the conditional expression that triggered the event.

Runtime bug fixes:

- Fixed a bug where mathematical domain checks would not raise warnings for (non square-root) powers of negative numbers.

E.7.4. Optimization Improvements

- Improved support for external objects by allowing strings and arrays in the constructor function.

E.7.5. Python Packages Improvements

The following fixes and additions have been made to the *DynamicDiagnostics* package:

- Added the function *plot_order_vs_time* to plot the order of solver(*CVode* only) over time.
- Made the *time_interval* parameter available to all plotting functions.
- Fixed a bug where the upper limit of *time_interval* was not used correctly.
- Added *marker* option to *plot_step_size_vs_time* and *plot_order_vs_time*.
- Added the function *get_events_details* to retrieve detailed event information and also the function *print_event_details* for printing of these.

The Assimulo version has been updated from 3.3 to 3.4.1, this entails the following changes:

- The *Radau5ODE* Fortran implementation has been removed.
- Fixed an issue where *Radau5ODE* simulation would not stop when a set time limit has been exceeded.
- Fixed an issue where *opts['Radau5ODE_options']['atol']* would receive (machine precision) sized changes with repeated solver calls and events.

- Added support for *rtol = 0* and rtol vectors for CVode, without sensitivites. If used via *pyfmi*, the values in an rtol vector need to be equal except for zeros.

The PyFMI version has been updated from 2.9.8 to 2.10.3, this entails the following changes:

- Calls to *get_(real|integer|boolean|string)* with empty lists as arguments no longer trigger a call to the FMU.
- Removed deprecated functionality for the following classes *pyfmi.fmi.FMUModelCS1, FMUModelME1, FMUModelCS2, FMUModelME2, FMUModelME1Extended*. Note that this affects objects created from *load_fmu*, in particular for removed arguments:
  - Function *get_log_file_name*, use *get_log_filename* instead.
  - Function *set_fmil_log_level*, use *set_log_level* instead.
  - Argument *path*, use *fmu* instead.
  - Argument *enable_logging*, use *log_level* instead.
  - Argument *tStart*, use *start_time* instead.
  - Argument *tStop*, use *stop_time* instead.
  - Argument *StopTimeDefined*, use *stop_time_defined* instead.
  - Argument *tolControlled*, use *tolerance_defined* instead.
  - Argument *relativeTolerance*, use *tolerance* instead.
  - Attribute *version*, use *get_version* function instead.
- Fixed a crash when using *ExplicitEuler* with *dynamic_diagnostics* on models with events.
- Changed Jacobian to use nominals retrieved via *fmu.nominals_continuous_states* instead of *fmu.get_variable_nominal(valueref)*.
- Malformed log messages no longer trigger exceptions. Troublesome characters are replaced with a standard replacement character.
- Absolute tolerances calculated with state nominals retrieved before initialization will be recalculated with state nominals from after initialization when possible.
- Added method to retrieve the unbounded attribute for real variables: *get_variable_unbounded* (FMI2 only).
- For unbounded states, the simulate method attempts to create a vector of relative tolerances and entries that correspond to unbounded states are set to zero. (FMI2 ME only)

E.8. Release notes for the OPTIMICA Compiler Toolkit version 1.42.1

E.8.1. Compiler Changes

This release contains a license integration update.

E.9. Release notes for the OPTIMICA Compiler Toolkit version 1.42

E.9.1. Compiler Changes

Important improvements to the compiler:

- Added support for compiling with Visual Studio 2022 versions.

E.9.2. API Improvements

Important improvements to the compiler API:

- Added method *getClassModification* for getting the *class modification* of a modifier.
- Fixed an issue where getting sizes that depend on function calls to missing functions would throw exceptions.
- Fixed an issue with an error message not reporting the correct class name for classes that could not found.

E.10. Release notes for the OPTIMICA Compiler Toolkit version 1.40

E.10.1. Compiler Changes

Important improvements to the compiler:

- The compiler option *generate_mof_files* is now deprecated, use *generate_html_diagnostics* instead.
- Added check that binding equations for constants and parameters are of an appropriate variability.
- Updated diagnostics for missing inner components to use *missingInnerMessage* annotation.
- Added compliance checks for use of operators *initial*, *sample* and *delay* in functions.

- Added error checks for use of operators *initial*, *terminal*, *sample* and *delay* in functions.
- The *Java* (*OpenJDK*) version bundled with Windows binary installers has been updated to *17.0.2.* from *11.0.2*.
- Added support for compilation of Windows FMUs from CentOS. Usage and limitations are described in the new UsersGuide chapter *Cross-platform generation of FMUs*.
- Updated third-party dependency *gson* from version *2.8.5* to *2.8.9*to resolve a detected vulnerability issue.

The release contains a number of bug fixes to the compiler, among them:

- Fixed NoSuchElement exception (and stack trace) in when-construct.
- Fixed a crash when evaluating global constant records inside functions.
- Temporary variables generated by the compiler, that were erroneously visible in the model description XML, are now hidden.
- Warnings informing about filtered warnings can be filtered with the identifier *FILTERED_WARNINGS*.
- Warnings that could not be filtered, such as *UNUSED_GLOB_PATTERN*, can now be filtered with the *filter_warnings* option.
- Added missing support for passing arrays to the loadResource function and built-in operator change.
- Fixed incorrect default value for option *html_diagnostics_contents*, it was *full* instead of *statistics*.
- Fixed a NullPointerException that occurred when transferring to CasADi with option *generate_html_diagnostics* enabled.
- Fixed a code generation issue related to accessing sub-components of global record constants.
- Compiling with option *generate_ode_jacobian* now properly applies corresponding derivative function when *zeroDerivative* modifiers exist within the annotation.
- Fixed an issue that caused the FMU model description to have variables with *variability=constant* and *initial=calculated*, which is not a valid combination.
- Dependencies listed with option *event_indicator_structure* now include solved variables used in equation blocks.
- Fixed bug where a missing short class base class could lead to a crash rather than an error message.
- Fixed crash in function inlining that could happen when an argument was an empty array.
- Fixed issues with compiler state being incorrectly restored after a failed compilation.

- Fixed and improved the error check for expandable connectors being incorrectly connected to non-expandable connectors.

E.10.2. API Improvements

Bug fixes to the compiler API:

- Fixed incorrect source text representation of for-expression.
- Fixed a libary conversion error check that could cause errors when a component has been renamed to a name that already exists in one of the base classes.
- Fixed a bug with *getLatestModifier* causing modifications from some base classes to be included for components.
- Fixed a bug where *getVisibilityPrefix* could give the wrong result for redeclared elements.

E.10.3. Runtime Improvements

Important improvements to the simulation framework:

- Improved the chattering warning in the simulation logfile. It now displays all Modelica expressions that cause chattering at a specific point in time, this requires the log level to be at least *warning*.
- Added *warning*to simulation logfile that lists all iteration variables with missing start values in a nonlinear block.
- Added *error*to simulation logfile in case of block initialization failure, that list all iteration variables with missing start values in a nonlinear block.
- Added *Radau5* as Co-Simulation solver, select by compiling with compilter option *cs_solver* set to *3*, or by setting the FMU variable *_cs_solver* to *3*.

Runtime bug fixes:

- Fixed issues where event detection and order selection in CVode Co-simulation FMUs *could* slightly differ after setting an FMU state.
- Fixed an issue where setting an FMU state did not correctly restore all internal nonlinear solver tolerances.
- Fixed a crash that could occur when setting an FMU state created using a different instance of the same FMU.

E.10.4. Python Packages Improvements

The Python version bundled with the Windows installer has been upgraded from Python *3.7.4* to *3.7.13*.

The Assimulo version has been updated from 3.2.9 to 3.3, this entails the following changes:

- Changed the name of opts['Radau5ODE_options']['solver'] to opts['Radau5ODE_options']['implementation'], where "opts=fmu.simulate_options()".
- Added support for sparse linear solver in Radau5ODE. Select via: opts['Radau5ODE_options']['linear_solver']='SPARSE' (default = 'DENSE'), additionally requires setting opts['with_jacobian']=True, where "opts=fmu.simulate_options()". Only available for opts['Radau5ODE_options']['implementation']='c' (default)

The PyFMI version has been updated from 2.9.7 to 2.9.8, this entails the following changes:

- Removed some DeprecationWarnings.

E.10.5. Compliance

The following Modelon libraries are compatible with OPTIMICA Compiler Toolkit version 1.40:

- Airconditioning Library 1.25
- Aircraft Dynamics Library 1.8
- Electrification Library 1.8
- Engine Dynamics Library 2.10
- Environmental Control Library 3.14
- Fuel Cell Library 1.16
- Fuel System Library 5.3
- Heat Exchanger Library 2.10
- Hydraulics Library 4.18
- Hydro Power Library 2.15
- Jet Propulsion Library 2.5
- Liquid Cooling Library 2.10
- Pneumatics Library 2.14
- Thermal Power Library 1.25
- Vapor Cycle Library 2.10
- Vehicle Dynamics Library 4.2

E.11. Release notes for the OPTIMICA Compiler Toolkit version 1.38.2

E.11.1. API Improvements

Important improvements to the compiler API:

- Fixed a crash with *getLatestModifier* when working with some redeclared full classes.

E.12. Release notes for the OPTIMICA Compiler Toolkit version 1.38.1

E.12.1. API Improvements

Important improvements to the compiler API:

- Fixed a bug with *getLatestModifier* causing modifications from some base classes to be included for components.

E.13. Release notes for the OPTIMICA Compiler Toolkit version 1.38

E.13.1. Compiler Changes

Important improvements to the compiler:

- Added description to event indicator variables. The description shows the event generating expression.
- Improved the duplicated components check so that it now points to the extends clause where the duplicated component is inherited in the checked model.
- Improved error message when linker fails to find a required library.

The release contains a number of bug fixes to the compiler, among them:

- Fixed erroneous balancing related error messages emitted from unbalanced components in partial instances.
- Fixed so local balancing check considers the Modelon connector vendor annotation "isTopLevel".
- Fixed exception that could occur when inlining functions in encrypted models.

- Fixed so that the extent of a class can be inherited if a local coordinate system without an extent is defined.
- Fixed so forward slashes are used as path separator when creating FMUs.
- Patched the provided MSL with a fix to avoid non-identical duplicate components, according to PR 4022 on ModelicaStandardLibrary GitHub.

E.13.2. API Improvements

Important improvements to the compiler API:

- API method checkClass now checks for incorrect choices annotations.
- Connector SVG metadata now includes port sizes and is generated for all connectors with a placement when *generate_svg_metadata_tag* is used.
- Added a new method getLatestModifier to InstanceClass and InstanceComponent that finds the latest modifier that modifies the class or component.
- Added a new method canGetInstanceClass() to check if a element can be obtained correctly.
- Added API option *api_allow_missing_elements*. When disabled the API will throw an error when trying to traverse a class extending a missing class.

E.13.3. Python Packages Improvements

The file *startup.py* has been removed together with several minor improvements to both *pymodelica* and *pyjmi*. For *pymodelica*, the exception *JError* has been removed and a new exception *PyModelicaException* has been added which all other inherit from. Some standard Python exceptions that were raised by *pymodelica* have now been replaced with module specific ones. The object *pymodelica.environ* has been moved to *pymodelica.compiler_environment.environ* but is still usable as *pymodelica.environ*. We recommend users of this to update their code to import it instead from *pymodelica.compiler_environ*.

E.13.4. Runtime Improvements

The release contains a number of bug fixes to runtime, among them:

E.14. Release notes for the OPTIMICA Compiler Toolkit version 1.36.1

E.14.1. Compiler Changes

The release contains a bug fix to the compiler:

- Resolved an issue with undefined references in source code FMUs that invoked *Modelica.Utilities.Files.loadResource*.
- Added option *package_html_documentation*. When enabled all generated HTML diagnostics will be packaged in the FMU.
- Added option *html_diagnostics_contents*. Its value specifies what diagnostics content will be generated. In addition to model statistics, new content choices are build information and model documentation.

E.15. Release notes for the OPTIMICA Compiler Toolkit version 1.36

E.15.1. Compiler Changes

Important improvements to the compiler:

- Added compiler options for skipping parameters from the *constant_parameters* option.
- Added support for ExternalObject components without binding equations when the constructor has a binding equation for every input.
- Added support for runtime resolving with *Modelica.Utilities.Files.loadResource*. This is controlled via the new option *load_resources_at_runtime*.
- Stricter checking of derivative annotations, controlled by the new option *check_derivative_annotations*. Currently this options defaults to false, unless *generate_block_jacobian* is enabled. Use of the *check_derivative_annotations* option is recommended as it may reveal errors in models that were previously undetected. We plan to change the default of this option in a future release.
- Changed so dependent parameters only turn constant if the entire binding equation will be constant when using *constant_parameters*.
- The start value is now used to turn parameters in to constants when there is no binding equation when using *constant_parameters*.
- Improved symbolic elimination of variables and equations in conjuction with a factor of zero.
- Allowed the MSL distributed with OCT to be separated from the distribution (and e.g. loaded from another location on the filesystem). However, doing so will make compiler option *msl_version* behave as if it was set to value *none*.

The release contains a number of bug fixes to the compiler, among them:

- The compiler would crash for some models involving overconstrained connectors.

- The local balancing check did not count the components inside of multi-dimensional array components correctly.
- Relaxed error check to allow non-parameter expressions in external object constructor arguments.
- Fixed bugs with how the the balance check counts equations for variables given values with modifications.
- Fixed a problem with how folders are created during compilation.
- Fixed a bug with how the balance check counts components and equations in zero or unknown sized record arrays.
- Fixed the bug that short classes extending partial models were not also considered partial.
- The compiler used to always link with ModelicaExternalC from Modelica 4.0.0, even when not specified by annotations in external functions. For backwards compatibility, the compiler will still try to link with ModelicaExternalC, but only when Modelica is specified in the uses-annotation, and then with the loaded Modelica library. This is non-standard behavior that may be changed in the future.
- The compiler used to always link with *ModelicaStandardTables*, and to include *ModelicaStandardTables.h*. This is no longer the case, and now needs to be explicitly specified in the same way as for other external code.
- Resolved an issue where an impure function was not properly inlined in some cases.
- Patched the provided MSL4 with a fix for *ModelicaInternal_fullPathName* according to PR 3663 on ModelicaStandardLibrary GitHub.
- Fixed bug which caused exception when printing dynamic state blocks in the debug log for encrypted libraries.
- Fixed a bug where certain system locales would prevent the license check from passing.
- Fixed issue with Modelica function getInstanceName when compiling models with experiment modifiers.
- Fixed compiler crash when using functions inheriting from built-in functions.

E.15.2. API Improvements

Important improvements to the compiler API:

- Added a warning for when extends in a package cannot be placed in a given order due to limitations.
- Fixed a bug where *convertLibrary* failed if multiple versions of the same library were on MODELICAPATH.

E.15.3. Python Packages Improvements

The PyFMI version has been updated to 2.9.7, this entails the following changes:

- Added an argument to *ResultDymolaBinary* to allow for reading updated data from the loaded file.

- Added option *synchronize_simulation* to allow for synchronizing simulation with (scaled) real-time.
- Added setup.cfg that lists all Python package dependencies in order to run PyFMI.
- Resolved an issue that would occurr when reading large result files or streams causing the data to be corrupt due to an integer overflow.

E.15.4. Runtime Improvements

The release contains a number of bug fixes to runtime, among them:

- Added fallback for block-initialization in dynamic simulation: If initialization of a block in the dynamic system fails with the solution of the initial system, we re-attempt initializing that block with the start values of the corresponding iteration variables.
- Fixed problem with computing the directional derivatives in blocks of equations that only contains discrete variables.
- Fixed problem where the MSL function fullPathName could lead to segmentation fault.
- Fixed segmentation fault with ModelicaMessage when called outside of its validity range.
- Removed an unnecessary model evaluation when inputs where being retrieved from the model.

E.16. Release notes for the OPTIMICA Compiler Toolkit version 1.34.2

E.16.1. Compiler Changes

The release contains a bug fix to the compiler:

- Fixed bug where a class that has an Icon or Diagram annotation, but no coordinate system defined in it, would not use one from an inherited class.

E.17. Release notes for the OPTIMICA Compiler Toolkit version 1.34.1

E.17.1. Compiler Changes

The release contains a number of bug fixes to the compiler, among them:

- Fixed a bug where using *msl_version="none"* could cause C compilation error.

E.17.2. API Improvements

Important improvements to the compiler API:

- Added the method API.clearModelicaPath for removing all paths added to the MODELICAPATH with API.addModelicaPath.

E.18. Release notes for the OPTIMICA Compiler Toolkit version 1.34

E.18.1. Compiler Changes

Important improvements to the compiler:

- Improved handling of duplicate variables.
- Added compiler option *allow_library_version_mismatch* that converts errors due to library version mismatch to warnings.
- Add support for compiling with Visual Studio 2019 versions.
- Improved error check for connect clauses to give more detailed error messages and made it slightly less strict in partial classes.
- Made handling of *MODELICAPATH* consistent between different ways of using the compiler.
  - The environment variable *MODELICAPATH* is no longer read, instead the value must be passed as an argument.
  - The option msl_version can be used to control what versions of MSL among those included are added to *MODELICAPATH*; all, none or a specific version.
- Some icons in the Modelica Standard Library bundled with OCT have received updates, where they now are using gradient fills. See Section C.3 for more information.

E.18.2. API Improvements

Important improvements to the compiler API:

- Changed so getComponentClass() for InstanceComponent and getInstanceClass() for InstanceExtends always throws an exception if the the class cannot be found.
- Added support for evaluating Enum literals in both SourceTree and InstanceTree.

- Added support for evaluating non-literal expression in Modifiers and Annotations from the InstanceTree.
- Added method for getting the diagram of a *connector* or *expandable connector* as an SVG image.
- Fixed issue with generation of SVG icons for component arrays.
- Fixed bug with external functions resulting in an invalid algorithm being returned by the API.
- Fixed a bug with getDeclaredSubscripts() for redeclared multi-dimensional array components resulting in an IndexOutOfBoundsException.

E.18.3. Python Packages Improvements

The Assimulo version has been updated from 3.2.7 to 3.2.9, this entails the following changes:

- Added a C implementation of the Radau5ODE solver (previously: Fortran). One can select the solver via "opts['Radau5ODE_options']['solver']='c'" (or 'f', default='c'), where "opts=fmu.simulate_options()".

E.18.4. Runtime Improvements

Important improvements to Runtime:

- If NonLinearConvergence error in block, we added logging for ivs, residuals, condition number and number of iterations.
- Added logging if a simulation fails due to non-convergence in a block with zero rows/columns in the jacobian.

E.19. Release notes for the OPTIMICA Compiler Toolkit version 1.32.1

E.19.1. Compiler Changes

This release contains a bug fix to the compiler:

- Fixed an issue where changing some options would not take effect, if done in between two compilations using the same API object when compiling in single process mode.

E.20. Release notes for the OPTIMICA Compiler Toolkit version 1.32

E.20.1. Compiler Changes

Important improvements to the compiler:

- Added support for MSL 4.0.0. Note that the sub package *Modelica.Clocked* is not yet supported and is therefore stripped out. Compiling with MSL 3.2.3 is still possible and can be done by changing the compiler option *msl_version* and/or utilizing a "uses"-annotation.
- Versions of libraries can be selected with "uses"-annotation if multiple versions exist on *MODELICAPATH*.
- Updated the HTML diagnostics generated when compiler option *generate_html_diagnostics* is enabled. The pages contain more data, and the presentation has been improved. There is a new page called *blockInfo.html* that contains information on each block within the model.
- Improved the error message displayed when a redeclaration is missing elements from a constraining type.
- Changed such that *smooth(0, expr)* generates event indicators but no event switches within internal blocks. As a consequence it behaves as *noEvent* except for the generated event indicators. The behaviour of *actualStream* has not been affected by this change.

Simulation of FMUs will now attempt new fallbacks when convergence fails:

- a fallback that perturbs the current point and improves the worst residual when the non-linear solver fails and the ODE solver step is small.
- a fallback ignoring min/max bounds for iteration variables in non-linear systems, only checking bounds compliance for final solution.
- a fallback for infinite loops during simulation if the iteration variables are very close to each other.
- a fallback when a residual is not improved enough and we are not the taking full Newton step in the non-linear solver.

Important improvements to the support of CasADi:

- Models with generated stream variables for encrypted libraries can now be transferred to CasADi.
- Resolved an issue when the annotation *HideResult* was set to *true* which prevented CasADi transfer when an encrypted library was loaded.
- Equations and variables introduced by inlining now have correct visibility and can be transferred to CasADi.
- Equations added by the compiler option *state_initial_equations* now have correct visibility and can be transferred to CasADi.

The release contains a number of bug fixes to the compiler, among them:

- Source code FMUs no longer include the filenames of temporary files used for constant evaluation.
- Start and nominal attributes set via type modifications now have lower priority than those set via component modifications. This priority is used when selecting attributes for alias variables.

- Resolved a crash with Linux FMUs using *Modelica.Utilities.System.getTime*.
- Resolved an issue where differentiation of an external function could cause a crash.
- Resolved an issue with an internal annotation used for imported FMUs where there would be a false error regarding unlocated resource files during simulation.
- Removed incorrect warning about missing binding equation for outer parameters and constants.

E.20.2. API Improvements

Important improvements to the compiler API:

- Added an editing operation that can re-order the elements in a *SourceClass*.
- Added support for renaming components and classes to the same name as an enclosing class.
- Added a method *isTopLevelClassInFile* for SourceClass.
- Added an overload of *checkSyntax* that can check the input as a class element (previously only as file).

The release contains a number of bug fixes to the compiler API, among them:

- Fixed a crash during library conversion due to a bug with value lookup from inside array subscripts.
- Resolved a bug with how the local balance check counts unknowns and equations in models with arrays with unknown or zero size.
- The local balance check now computes the correct number of unknowns and equations for connector inputs and flow variables from outer components.
- Using *renameClass* should now correctly reflect the name change corresponding package.order file.
- Fixed an issue with positional information for syntax errors.
- Fixed an issue where *API.getTopLevelSourceClasses* would throw an exception.
- Fixed a issue where adding an element followed by renaming a library without a order file could cause the created order file to not be moved.
- Fixed an issue with loading the same library from both a symbolic link and the real path.
- Fixed bug with CopyClass crashing when trying to copy a file consisting of only a short class.

E.20.3. Python Packages Improvements

PyFMI has been updated from version 2.8.10 to 2.9.5, this includes the following changes:

- Diagnostic data is now saved in the binary result file instead of the log file. Additionally more data can be accessed if the simulation option *logging* is enabled.

- Attempts to get continuous states when there are no such states will now return *fmi2_status_ok* instead of an error.

Assimulo has been updated from version 3.2.5 to 3.2.7, this resolves minor deprecation warnings visible with newer versions of Python.

SteadyState has been updated with the following changes:

- Intermediate points for both cancelled and failed simulations can now be saved after each iteration by enabling the steadystate solver option *save_last_integration_point*.

E.20.4. Compliance

The following Modelon libraries are compatible with OPTIMICA Compiler Toolkit version 1.32:

- Airconditioning Library 1.23
- Aircraft Dynamics Library 1.6
- Electrification Library 1.7
- Engine Dynamics Library 2.8
- Environmental Control Library 3.12
- Fuel Cell Library 1.14
- Fuel System Library 5.1
- Heat Exchanger Library 2.8
- Hydraulics Library 4.16
- Hydro Power Library 2.14
- Jet Propulsion Library 2.3
- Liquid Cooling Library 2.8
- Pneumatics Library 2.12
- Thermal Power Library 1.23
- Vapor Cycle Library 2.8

- Vehicle Dynamics Library 4.0

E.21. Release notes for the OPTIMICA Compiler Toolkit version 1.30

E.21.1. Compiler Changes

Important improvements to the compiler:

- Updated error message displayed when line-search fails for nonlinear blocks.
- Event indicators are no longer generated for expressions in when clauses.
- Errors and warnings now reported with location in enclosing class rather than file when possible. Option show_error_location to restore old behavior added.

The release contains a number of bug fixes to the compiler, among them:

- Models with fixed parameters that have start values but no binding equations now use the start value as a binding equation.
- Accesses to arrays that leave some indices unspecified are now correctly handled.
- Fixed bug where source code FMUs would not list split .c files.
- Fixed name lookup from within size specifiers in component redeclares.
- Fixed issue with Lapack that could result in segmentation faults on Linux.
- Fixed bug where partial indices of arrays were not correctly type-checked.
- Fixed a bug where values from the record constructor did not propagate into the size of a array.

E.21.2. API Improvements

Important improvements to the compiler API:

- Fixed copyClass so that subpackage structure is retained. Copying a structured package will no longer convert the copy to an unstructured package.
- Added editing methods for removing, setting and adding both normal and initial-equations in the source tree.
- Perform more local error checks in InstanceClass.checkClassLocal.
- Added a method for checking if an equation is an initial equation.

- Added methods for obtaining all algorithm from classes and components in both Instance and Source tree.
- Added a method for checking if an algorithm is an initial algorithm.
- Added editing methods for removing, setting and adding both normal and initial-algorithm in the source tree.
- Fixed an issue where accessing libraries with syntax errors could cause an exception if the library name had a version number.

The release contains a number of bug fixes to the compiler API, among them:

- Improved handling of inherited graphics annotations in connectors.
- Fixed bug a where copyClass would fail for models containing numbers prefixed with a '+' sign.
- Fixed bug that caused redeclare extends clauses to lose their formatting when saving.
- Fixed an issue where API.checkClass would report errors unrelated to the model being checked.
- Fixed several bugs for editing operations requiring updates to symbolic links.
- Fixed bug where InstanceClass.getAllMatchingRedeclareChoices would cause an exception if a loaded library had syntax errors.

E.22. Release notes for the OPTIMICA Compiler Toolkit version 1.28.4

This release contains a bug fix to the compiler API:

- Saving changes to a file that is a symbolic link will no longer replace the link with a new file.

E.23. Release notes for the OPTIMICA Compiler Toolkit version 1.28.3

This release contains a bug fix to the compiler API:

- Fixed bug where moving a class could cause syntax errors in accesses to the moved class.

E.24. Release notes for the OPTIMICA Compiler Toolkit version 1.28.2

This release contains bug fixes to the compiler API:

- Fixed bugs related to setting source text for classes with syntax errors.

E.25. Release notes for the OPTIMICA Compiler Toolkit version 1.28.1

This release contains bug fixes to the compiler API:

- Fixed bug where copying a class containing named function call arguments would cause exceptions.
- Fixed bug where editing a class after fixing a syntax error would cause exceptions.

E.26. Release notes for the OPTIMICA Compiler Toolkit version 1.28

E.26.1. Compiler Changes

Important improvements to the compiler:

- Added support for the *inverse* annotation.
- Changed the default target to ME2.0 for both command-line and API calls.
- Improved the C code-generation to improve simulation performance.
- Added support for the *HideResult* annotation.
- Added a new ODE solver to CS FMUs, a second order Runge-Kutta method.
- Improved error message when index reduction fails due to the need of differentiating an equation that uses an input.
- Enabled BLT table diagnostics for encrypted models.
- Added support for checking if a model is locally balanced. This is limited to models without expandable or overdetermined connectors.

Important improvements to the support of CasADi:

- The names of inputs and outputs in functions are now transferred to Casadi.
- Functions in if-statements are now using CasADis conditional operator for MXFunctions instead of the conditional operator for MX.

- Several minor performance improvements.

The release contains a number of bug fixes to the compiler, among them:

- Fixed a stack overflow error that would occur with conversion scripts on large models.
- Fixed a conversion script limitation which caused some components in redeclared classes to not be updated.
- Fixed a bug that would result in *UnsupportedOperationException*.
- Fixed a stack overflow error when too many scalars were assigned in the same algorithm or function call.
- Fixed bug where using multi-byte characters in a description string could cause an invalid FMU if the Java default character encoding was not UTF-8.
- Fixed so that derivative variable inherits its visibility from where it is derived from.
- Fixed so that event indicator variables (added with the *event_output_vars* option) are shown in encrypted models.
- Fixed so that the compiler finds the correct folder for external libraries on linux based on gcc version.
- Fixed a scalability bug. This fix results in a large compilation time decrease for specific models.
- Fixed an issue with OCT on MATLAB® unable to find the gcc compiler.
- Fixed an issue where *bltTable.html* was generated even though *diagnostics_limit* was exceeded.
- Fixed a bug that could cause an unbalanced system after optimization.
- Fixed a bug where the options *convert_to_input* and *delayed_scalarization* could cause a crash if the glob pattern matched an array variable.
- Fixed a bug so that equations and variables are exposed when they are allowed for encrypted models.
- Fixed a bug where the compilation result could differ with and without encryption for the same model.

E.26.2. Python Packages Improvements

Bug fixes to Python packages:

- Fixed an exception that would occur when getting an event summary in *DynamicDiagnostics* in the case where only time events are present.
- Added support getting the Hessian of the outputs in *ModelProblem*.
- Fixed a bug where *LogViewer* in the steadystate package would fail to retrieve jacobians for very large models.

The Assimulo version has been updated from 3.2.4 to 3.2.5, this entails the following changes:

- Additional Fortran compile flags can now be provided during setup.
- Fixed an issue that caused exceptions during setup when Sundials was not found.

The PyFMI version has been updated from 2.8.3 to 2.8.10, this entails the following changes:

- Added support for writing result data to streams.
- Fixed segfault when storing data from models with a huge number of variables.
- Loading of FMUs can now be done from an unzipped folder if the argument *allow_unzipped_fmu* is set to *True*.
- The argument *path* to *load_fmu* and the different FMI-classes is now deprecated
- Added support to log to streams via the keyword argument *log_file_name*. This is supported for all the FMI-classes as well as the function *load_fmu*.
- Improved performance of the Master algorithm.
- Updated exception types when loading of an FMU fails.
- Added safety check for updated binary files which can cause issues.
- A matrix of all the results from a binary file can now be retrieved even if delayed loading is used.
- The written binary file is now always consistent, i.e. if a simulation aborts, it can still be read.
- Changed default loading strategy for binary files, now the trajectories are loaded on demand instead of all at the same time.
- Updated Master algorithm options documentation and fixed result file naming.
- Fixed *block_initialization* in Master algorithm when using Python 3.
- Fixed an issue where the attribute *initial* was not properly set on *ScalarVariable2*.
- Fixed an issue with *get_variable_nominal* that would occur when specifying the value reference of a variable.
- Added a utility function to determine if the maximum log file size has been reached.
- Added support for parsing boolean values in the XML log parser.
- Added support for option *logging* for different ODE solvers.

E.26.3. API Improvements

Important improvements to the compiler API:

- Changed *EditingManager.copyClass()* to preserve all provided formatting information including comments and indentation. Before default formatting was used and comments discarded.
- Added method *SourceClass.getSyntaxErrors()* to get a list of syntax errors.
- Implemented *getSourceText()* and *setSourceText()* on *SrcBadLibNodes* and *SrcBadClassDecl* to allow repair of syntax errors.
- Added an editing operation *setComponentTypeSpecifier* for setting the type of a source component.
- Libraries with a syntax error in a top level file can be loaded without getting an exception.
- Added an editing operation *setDescriptionString* for setting the description string for classes, components and imports.
- Added a method *SourceClass.save()* for saving an individual changed class. This is possible as long as no refactoring which requires saving everything have been made.
- Added a method *getSourceClassesInFile* for getting the top-level classes in a Modelica source.
- Added methods *checkClass* and *checkClassLocal* for performing error checks on an *InstanceClass*.

The release contains a number of bug fixes to the compiler API, among them:

- Fixed bugs when editing libraries loaded using symbolic links.
- Fixed bug where it was not possible to print redeclare choice annotations in encrypted libraries.
- Fixed a library conversion bug that could erroneously report an error for class redeclares when base class is redeclared.
- Fixed an issue with obtaining modifications from the *InstanceComponents* in an array.
- Fixed bug with variability prefixes which caused protected elements in a component to become visible.
- Fixed a bug with conversion scripts that prevented updating of named arguments in function calls.

E.26.4. General

Other notable changes:

- Updated ModelicaServices package to support 3D animation.

E.27. Release notes for the OPTIMICA Compiler Toolkit version 1.26

E.27.1. Compiler Changes

Important improvements to the compiler:

- When compiling with msvs as C-compiler, the default bitness for the generated FMU has been changed from 32-bit to the bitness of the JVM. This change affects when compiling via the API as well as the command-line interface.
- Changed the default compiler from msvs to gcc on Windows. The compiler can still be changed with the option *c_compiler*.
- Added a compiler option *flatten_only* that terminates compilation early without generating an FMU. This option can be used together with the option *log_dependencies*.
- Added a compiler option *generate_svg_metadata_tag* that controls if the metadata tag is added during SVG generation.
- Added the FMI 2.0 dependenciesKind attribute to generated FMUs.
- Added HTML diagnostics of the equation system and initial equation system in BLT form for encrypted models. See the links to "BLT for DAE System" and "BLT for Initialization System".
- Added support for connect equations in the diagnostics for encrypted models.
- The flat code representation in the diagnostics has been simplified by not listing protected variables in their own section.
- Failed size evaluations should display clearer and more descriptive error messages.
- Added options *convert_to_input* and *convert_to_input_from*, that allows converting independent parameters to inputs of the FMU or CasADi model.
- Improved symbolic simplification in sets of linear equations containing derivative uses.
- Improved symbolic simplification and alias detection in sets of linear equations with coefficients +/- 1. This structure is common when coupling electrical components and can have a significant impact on the simulation performance.

Important improvements to the support of CasADi:

- Added support for external functions when transferring models to CasADi.
- Added limited support for using functions with flexible input sizes with CasADi.
- Added support for if-statements in functions when transferring to CasADi.
- Upgraded third party CasADi library. The CasADi bundled with OCT is now version 3.5.5 instead of 3.2.1.

The release contains a number of bug fixes to the compiler, among them:

- Fixed incorrect handling of dot-operators (.+, .*, etc) for arrays of operator records.
- Updated the algorithm for symbolically solve equations in order to protect against structurally division by zero.
- Fixed type checking bug when redeclaring matrix components.
- Fixed bug where list of warnings was not cleared between consecutive compilations.

E.27.2. Python Packages Improvements

New packages:

- Added a Python package for ODE solver dynamic diagnostics.
- Added a Python package for importing CS FMUs into Modelica libraries.

Important changes to the steadystate Python package:

- The ability to solve equations for additional non-residual output variables has been added to ModelProblem. It is also now possible to evaluate the residuals and Jacobian of active and held iteration variable sets.
- The time and memory required for initialization of a ModelProblem instance, and Jacobian evaluation, has been reduced, allowing much larger models with O(10000) algebraic equations to be transferred and solved.
- Added a method for getting output variable names to ModelProblem. Also, an option to return a sparse/dense Jacobian has been added to ModelProblem.

E.27.3. Runtime Fixes and Improvements

- Fixed incorrect resetting of scaling in non-linear internal FMU blocks, proper update of runtime options and other bugs related to get/set FMU state.

E.27.4. API Improvements

Important improvements to the compiler API:

- Added a method *InstanceComponent.getDeclaredComponentClass* for retrieving the type specifier for a component as declared in the source code.
- Added a method *DeclarationSize.getDeclaredSubscript* for retrieving the size subscripts as strings for InstanceComponents.

The release contains a number of bug fixes to the compiler API, among them:

- Fixed a bug with *getTopLevelSourceClasses* that contained out of date information after unloading library.
- Fixed a bug that changed the result of *getDeclaredText* for *Modelica.Utilities.Files.loadResource* calls to *loadResource*.

E.28. Release notes for the OPTIMICA Compiler Toolkit version 1.24.6

This release contains bug fixes to the compiler:

- Parameters affecting connect statements are now properly marked as structural.

E.29. Release notes for the OPTIMICA Compiler Toolkit version 1.24.5

This release contains bug fixes to the compiler API:

- A bug with conversion scripts that prevented components from being correctly considered for encrypted libraries.

E.30. Release notes for the OPTIMICA Compiler Toolkit version 1.24.4

This release contains features added to the compiler:

- Added option (mathematical_domain_warnings_limit) to control the maximum number of log warnings from the mathematical domain checks that can be printed during a simulation.

E.31. Release notes for the OPTIMICA Compiler Toolkit version 1.24.3

In this release, the Modelica Standard Library (MSL) bundled with OCT have received some modifications in order to add support for 3D animations in Modelon Impact. For additional information of the applied changes, see Section C.3.

This release contains bug fixes to the compiler API:

- Prevented one situation where a access is updated incorrectly by two conversion rules.
- Conversion script script bugs fixed including not correctly considering components that are not mentioned in the script.

E.32. Release notes for the OPTIMICA Compiler Toolkit version 1.24.2

This release contains bug fixes to the compiler API:

- Fixed a bugs with convertModifiers that prevented class redeclare from being removed.

E.33. Release notes for the OPTIMICA Compiler Toolkit version 1.24.1

This release contains bug fixes to the compiler API:

- Fixed a bug with conversion script that lead to some files not being saved.
- Fixed a bug which could cause a exception for some combinations of conversion rules.

E.34. Release notes for the OPTIMICA Compiler Toolkit version 1.24

E.34.1. Compiler Changes

Important improvements to the compiler:

- Removed usage of thread-local storage for global constants in functions.
- Some equations and variables which are allowed to be seen based in the protection annotation are now printed in flattened code and some diagnostics.
- Added support for CasADi with Python 3. The CasADi bundled with OCT is now version 3.2.1 instead of 2.1.0.
- Improved support for solving (scalar) linear equations during compile time.
- OCT is now using FMI Library version 2.2.3 instead of 2.2.

The release contains a number of bug fixes to the compiler, among them:

- Fixed a C compilation error for the option generate_ode_jacobian that could occur when converting an integer array to a real array in a function call.
- Fixed a bug for records arrays with parameters that have both dependent and independent elements.
- Fixed a bug where array components in expandable connectors would sometimes get the wrong modifiers.
- Ensured that no exception is thrown when the output directory for html diagnostics already exists, and updated the log message. The log message is now shown at log level INFO.
- Fixed a bug when flattening certain records that would lead to NullPointerExceptions.
- Fixed a bug with deserialization of FMU states for CS FMUs that would sometimes result in a crash.
- Fixed a bug with saving the state of an FMU in between time events.
- Fixed a bug that caused assertion warnings to be absent from the simulation log for steady-state problems.
- Ensured that no exception is thrown when resolving certain malformed Modelica URIs.
- Setting a saved FMU state on a new FMU instance should no longer crash.

E.34.2. Python Packages Improvements

Important changes to Python packages:

- Steadystate solver supports "time_limit" solver option to stop execution if the elapsed time exceeds a specified limit.
- In the Steady State framework, there is now support for non-quadratic models. Steady-state problems can be constructed from models with an unequal number of iteration variables and residuals.

The Assimulo version has been updated from 3.2 to 3.2.4, this entails the following changes:

- Automatic detection of BLAS has been improved.
- Several minor performance improvements.

The PyFMI version has been updated from 2.7.4 to 2.8.3, this entails the following changes:

- Major performance improvements.
- Fixed issue with save / get FMU state.

- Fixed so that default options are not overriden when setting solver options.
- Exposed the dependencies kind attributes from FMI 2.0.
- Fixed result saving when saving only the "time" variable.

E.34.3. API Improvements

Important improvements to the compiler API:

- All InstanceTree methods for obtaining connect equations no longer throws access exception when some connects cannot be shown. A list of visible connects is now always returned.

The release contains a number of bug fixes to the compiler API, among them:

- Fixed a bug where setting modifiers for components in short classes would give incorrect modifiers.
- Fixed a problem with *moveClass* which caused incorrect code when globalizing qualified names.
- Fixed a bug with removing and recreating components which caused exception when saving.
- Fixed a bug with removing components where some annotations are not removed.
- Fixed a bug with setting a value annotation where the annotation is not updated correctly.
- Fixed a bug when unloading a library with base classes redeclared in another library caused exception during unload.
- Fixed a bug with moving and renaming a short class which caused exception.
- Fixed a bug causing the top class list to be not updated when it needs to.
- Fixed a bug with convertClass that caused incorrectly updated names when a class is moved into a subpackage.
- Fixed a bug with library conversion that caused incorrectly updated names when replaced by a shorter names.

E.35. Release notes for the OPTIMICA Compiler Toolkit version 1.22.1

This release contains a bug fix to the compiler:

- Fixed a bug with set and get state of FMUs that would sometimes result in a crash.
- When an encrypted library is on the path but not actually used, diagnostics are no longer restricted.

E.36. Release notes for the OPTIMICA Compiler Toolkit version 1.22

E.36.1. Compiler Changes

Important improvements to the compiler:

- The file flattened.html of the HTML diagnostics now includes some unencrypted parts of a model when encrypted libraries are loaded.
- The file index.html of the HTML diagnostics is now generated when encrypted libraries are loaded. Additionally, model diagnostics are now also printed in log-files for log-level *verbose* when encrypted libraries are loaded.
- Changed so BLT tables are not generated if the number of equations is greater than the limit set with the *diagnostics_limit* option.
- Changed so protected variables are excluded from the FMI modelDescription by default. Also added an option *include_protected_variables* for including protected variables in the modelDescription. Using the option, a protected variable will be included if its protection annotation allows it to be inspected in the class.
- Exported FMUs following FMI 2.0 now supports the feature to get and set an internal FMU state. Additionally serialization and de-serialization is supported.

The release contains a number of bug fixes to the compiler, among them:

- Fixed a bug for the event_indicator_structure option where there in some cases were differing numbers of event indicators in the EventIndicators and ModelVariables sections.
- Fixed a C-code compilation error that occurred in some cases when using the *linspace* operator in functions.
- Fixed a bug for the *event_output_vars* option where multiple event indicators were created for a single relation expression.
- Fixed a bug where equations were sorted incorrectly in the BLT when using the option *event_output_vars*.
- Fixed a bug that caused variables with *reinit()* to cause an error in certain circumstances.
- Prevented out of memory errors during C-code compilation for models with many event indicators.
- Fixed a bug where compilation would fail with message 'Unknown switch index for relational operator, not in list of switches.'
- Fixed a bug when some elements of an array need parameter equations, and some need binding equations. The bug led to either incorrect code or an *ArrayIndexOutOfBoundsException*.

- Fixed a *NullPointerException* when a record declared within another record has elements with differing variability.
- Fixed a *StackOverflowError* that could occur with certain types of array expressions in record constructors.
- Fixed a bug when *cat()* gets an empty array as one of its arguments.
- Fixed a bug which caused ArrayIndexOutOfBoundsException when multiple variables were assigned with a function inside an If-equation.
- It is now allowed to specify more than one path for Include, Library, LibraryDirectory and IncludeDirectory.
- Fixed a bug where array components in expandable connectors would sometimes get the wrong modifiers.

E.36.2. API Improvements

Important improvements to the compiler API:

- Added method *isVirtualArrayElement()* on InstanceComponent to identify virtual array elements. These elements are added to empty array components or array components with an unknown number of array elements.
- Added support for convertClassIf.
- Added support for convertMessage.
- It is now possible to get the connect clauses of a source tree class, using *SourceClass.getConnects()*.

The release contains a number of bug fixes to the compiler API, among them:

- Fixed an issue where structured libraries were not copied correctly.
- *SourceElement.getSourcePath()* no longer throws an exception for built-in types and functions.
- Fixed a bug with *convertModifiers()* when no *oldModifiers* were specified. The bug caused all existing matching modifiers to be replaced.
- Fixed a bug with conversion scripts where the lookup of classes could cause an exception.
- Fixed a limitation in the detection and fixing of name conflicts when updating libraries with conversion scripts.
- The methods for getting connect equations in the instance tree now properly throw exceptions when protection annotations do not allow viewing the diagram, instead of returning an empty list. This applies to:
  - InstanceElement.getConnects()
  - InstanceElement.getAllConnects()

  - InstanceElement.getAllActiveConnects()

- Fixed a bug that when moving a class, redeclared elements would sometimes be incorrectly globalized, yielding incorrect code.

- Fixed several sources of *UnbackedNodeExceptions* after editing operations.

E.37. Release notes for the OPTIMICA Compiler Toolkit version 1.20.1

E.37.1. Bug fixes

The following bugs affecting the 1.20 release only have been fixed:

- Fixed that in some cases, setSourceText() would not preserve formatting and could emit syntactically incorrect code.

- Fixed that after reading a file with very long lines, editing operations could not be performed in the structured type hierarchy the file belongs to.

E.38. Release notes for the OPTIMICA Compiler Toolkit version 1.20

E.38.1. Compiler Changes

Important improvements to the compiler:

- Added a new compiler option *time_state_variable* for treating time as a regular state variable. This will add an extra variable and equation *der(time) = 1*.

- Reinits are now included in dependency calculations between event indicators and states.

- Added functionality to turn on detailed runtime logging at a specific interval.

- Several minor improvements resulting in a reduced compilation time.

The release contains a number of bug fixes to the compiler, among them:

- Fixed an issue where it was not possible to use directional derivatives with large models.

- Fixed an issue with directional derivatives where record arrays would get an undefined size in generated code.

- Fixed a bug that would result in an integer overflow error in the linear equation elimination.
- Fixed exception when unit attribute for array parameter has array value.
- Fixed an issue where equation assert statements gave multiple assertion warnings for the same problem.
- Fixed an issue where using the option *event_indicator_structure* for models with *elsewhen* clauses caused an exception.

E.38.2. API Improvements

Important improvements to the compiler API:

- Added support for conversion scripts using the method *convertLibrary()* on the class API. In this release there is no support for *convertMessage* and *convertClassIf*.
- Added a method *getDeclaredSize()* to *SourceComponent* and *SourceShortClass* that returns the array size as declared in the Modelica code.
- Added methods for retrieving constraining clauses to the source api.

The release contains a number of bug fixes to the compiler API, among them:

- Fixed a bug where a value for enumeration literals would be queried when evaluating expressions with a *VariableValueProvider*.
- Unloading encrypted libraries without license features will now release the licensing executables.
- Fixed a bug when deleting, replacing or changing a modifier: Caches in the Source-API would not be flushed, yielding *UnbackedNodeExceptions* in some cases.
- Fixed a bug that deleting a class or unloading a library could cause an UnbackedNodeException in rare cases.
- Fixed a bug that when changing a component name by using *renameComponent()* or *changeComponentName()* to the name of a built-in Modelica function, a *NullPointerException* would be thrown.
- Fixed a bug that lines longer than 4095 character in a file caused syntactical problem due to formatting problems. The syntax problems are now avoided but user provided formatting and comments are removed in such files.
- Added missing checks for name conflicts further down the class hierarchy for *renameComponent()* and *renameClass()*.

E.38.3. Python Packages Improvements

Important changes to Python packages:

- Made the steadystate package independent of PyJMI
- Updated PyFMI to version 2.7.3
- Updated Assimulo to version 3.2

E.38.4. General

Other notable changes:

- Added the the protection annotation *Access.hide* to the ModelicaServices package.
- CasADi is temporarily unavailable for this release.

E.38.5. Compliance

E.38.5.1. Modelon libraries

For OPTIMICA Compiler Toolkit version 1.20 the following libraries provided by Modelon are compliant:

- Aircraft Dynamics Library 1.3
- Electrification Library 1.4.1
- Engine Dynamics Library 2.5build3
- Environmental Control Library 3.9build3
- Fuel Cell Library 1.11build3
- Fuel System Library 4.8build3
- Heat Exchanger Library 2.5build2
- Hydraulics Library 4.13build3
- Hydro Power Library 2.11build3
- Jet Propulsion Library 2.0
- Liquid Cooling Library 2.5build2
- Pneumatics Library 2.9build3
- Thermal Power Library 1.20build3

- Vapor Cycle Library 2.5build3
- Vehicle Dynamics Library 3.5

E.39. Release notes for the OPTIMICA Compiler Toolkit version 1.18

E.39.1. CasADi

CasADi is temporarily unavailable for this release.

E.39.2. Compiler Changes

Important improvements to the compiler:

- Improved performance of handling encrypted libraries.

The release contains a number of bug fixes to the compiler, among them:

- A rare bug in constant evaluation of attributes of parameters that are arrays of primitive types was resolved.
- The variability of parameters used to set the *stateSelect* attribute of a variable is now structural.
- Fixed a bug where linear equations were eliminated incorrectly.

E.39.3. API Improvements

Important improvements to the compiler API:

- Grant access to more annotations in encrypted libraries. Details are found in the API documentation of the Annotation interface.
- Deprecated the method *SourceClass.iconSVG()* because icons rendered by it are of poor quality. Use *InstanceClass.iconSVG()* instead.
- The following functionality is now exposed through the EditingManager, and the methods themselves are now deprecated:
  - *SourceShortClassImpl.setSuperClass(String)*. This method now also preserves string comments and annotations.
  - *Annotation.experimental_setValue(String)*
  - *Modifier.experimental_replaceWith(String)*

- Added API method *API.checkSyntaxLibrary(Path)* for running a syntax check on all Modelica files in a library.
- Generated icons no longer include sub-connectors of connectors.

The release contains a number of bug fixes to the compiler API, among them:

- *Expression.getDeclaredText()*: parenthesized expressions would not print parentheses for expressions retrieved from the Source-API.
- Components are now final when modified final without any value modification. Previously, the final modifier did not have any effect in this case.
- The API can now correctly determine the size of row components. Previously some cases were erroneously reported as scalars.
- Fixed a problem where *getQualifiedName()*, in one case, did not contain indices, and where *getElementName()* contained parent names.
- Many bug fixes where made in the functionality of the *EditingManager*. It can now be considered non-experimental while in previous releases, it should have been marked as such.
- CompilationException did not contain any warnings that occurred during compilation if there were errors as well.

E.39.4. Python Packages Improvements

Important changes to Python packages:

- It is now possible to add additional Java libraries to classpath via PyModelica before starting the Java Virtual Machine (JVM).
- In the Steady State framework, there is now support for using dynamic models, i.e. using states as iteration variables and derivatives as residuals.

E.39.5. Compliance

E.39.5.1. Modelon libraries

In release 1.18 we have updated the Modelica Standard Library (MSL) version to 3.2.3, build 3. It has been modified with additional patches, see Section C.3.

For OPTIMICA Compiler Toolkit version 1.18 the following libraries provided by Modelon are compliant:

- Aircraft Dynamics Library 1.3
- Electrification Library 1.4

- Engine Dynamics Library 2.5
- Environmental Control Library 3.9
- Fuel Cell Library 1.11
- Fuel System Library 4.8
- Heat Exchanger Library 2.5
- Hydraulics Library 4.13
- Hydro Power Library 2.11
- Jet Propulsion Library 1.6
- Liquid Cooling Library 2.5
- Pneumatics Library 2.9
- Thermal Power Library 1.20
- Vapor Cycle Library 2.5
- Vehicle Dynamics Library 3.5

E.40. Release notes for the OPTIMICA Compiler Toolkit version 1.16

E.40.1. Compiler Changes

With this release OPTIMICA Compiler Toolkit is shipped with Python 3.7.4.

Important changes to the compiler:

- Added a compiler option *mathematical_domain_checks* which determines if domain checks are to be performed for all mathematical functions.
- Fixed a bug with partial variability propagation resulting in unbalanced equation systems.
- Added a method for evaluation at compile time of calls to external functions that are supplied in DLLs, without compiling an executable on the fly. This decreases compilation time for models with many such function calls. Activated by the option *external_constant_evaluation_dynamic*. This option is on by default.
- Nominals are now always positive in the FMI model description.

- Improved attribute evaluation robustness for some cases. Parametric attributes are now only generated in model descriptions for inputs and states.
- Format string argument to the String function is now respected during constant evaluation.
- A graphical Text-primitive with a line-break in its textString annotation value no longer throws a NullPointerException".

E.40.2. API Improvements

Important improvements to the compiler API:

- Added Annotation.getDescriptionString() to access an annotation's description string.
- Added EditingManager.copyComponent() for copying a component to another class.
- Removed InstanceComponent.experimental_copyInto() and experimental_duplicate() as that functionality is now available through the EditingManager. See JavaDoc for updated semantics.
- Added FilterableComponent.getConditionExpression() to access the optional component's optional condition expression.
- Fixed an issue causing incorrect behavior of editing operations and several API functions after EditingManager.addTopLevelClass() or addTopLevelPackage() has been used.
- EditingManager.removeClass() no longer has restrictions regarding which classes can be removed.
- Fixed issue with EditingManager.copyClass() which did not globalize all required accesses.

E.40.3. Python Packages Improvements

Important changes to Python packages:

The PyFMI version has been updated from 2.5.3 to 2.5.7, this entails the following changes:

- Fixed several bugs for Python 3 compliance.
- Fixed an issue with computing the Jacobian using the structural information of the model when the model has no state.
- The default value of the simulation option *maxh* is now based on *ncp* (number of communicaton points), start and stop time, according to the following equation: *maxh=(stop-start)/ncp*.
- The default value of *ncp* (number of communicaton points) is now 500 instead of 0.

- The default value for Jacobian compression has been changed. It is now applied when using CVode for systems of size larger than 10.
- An option has been added to specify if variable descriptions should be stored in the result file.

E.41. Release notes for the OPTIMICA Compiler Toolkit version 1.14

E.41.1. Important information

Note, with the next release the Python distribution bundled with OCT is planned to be updated from version 2.7 to version 3.x.

E.41.2. Compiler Changes

Important changes to the compiler:

- Added dependency to gson thirdparty library.
- Option event_indicator_structure is no longer experimental. Updated the functionality of the option to only include reverse dependencies.
- Added options for including extra resource files in FMU: extra_resources and extra_resources_from.

E.41.3. API Improvements

Important improvements to the compiler API:

- Switch to "java.nio.file.Path" instead of "File" or "String"s representing paths in the API. The API is backwards-compatible, but methods referring to File and String are deprecated. In corner-cases, behavior may differ as the old File-API handles malformed paths such as "/C:/path" and the empty path differently.
- Added support for loading libraries as read-only and for checking if a library or a source element is read-only. Encrypted libraries and libraries in MODELICAPATH will be forced into a read-only state regardless of the client's request.
- Can now generate mirrored SVG icons in the API by using negative width and/or height.
- Fixed a bug that "flow" connector components were incorrectly reported as having causality prefix "input" for InstanceComponent only. They are now correctly reported as having causality "none" for both instance and source components.
- Annotation.getBindingExpression() now only returns non-null for actual binding expressions, i.e., it now confirms to the documentation. New function asExpression() added for interpreting an Annotation as

an expression. This now also works for function calls. Traversal of annotations has also changed: Annotation.valueAsAnnotation() now returns the value directly, e.g., for "x = f()", an annotation referencing "f()" is returned. Before, 'forPath("f")' or "subAnnotations().iterator().next()" was needed.

E.41.4. Python Packages Improvements

Important changes to Python packages:

The PyFMI version has been updated from 2.5 to 2.5.3, this entails the following changes:

- Fixed wrong default value for FMUModelME1Extended.
- Improved relative imports of Assimulo dependent classes.
- Fixed issue with unicode symbols in result files.
- Fixed a number of encode/decoding issues for Python3.
- Forced no copy if the provided array is already correct, minor performance improvement.
- Fixed issue with corrupt result files after failed simulations.

For the steady state framework the following has been changed:

- Added an experimental feature ResultHandling to enable writing results to different file formats.
- Initialization failures in FMUProblem should now properly close the log file.

E.41.5. MATLAB interface

Important updates to the MATLAB® interface:

- Fixed such that an empty command window is no longer displayed in MATLAB® during compilation.

E.41.6. Compliance

E.41.6.1. Modelon libraries

For OPTIMICA Compiler Toolkit version 1.14 the following libraries provided by Modelon are compliant:

- Aircraft Dynamics Library 1.2
- Electrification Library 1.3
- Engine Dynamics Library 2.4

- Environmental Control Library 3.7
- Fuel Cell Library 1.10
- Fuel System Library 4.6
- Heat Exchanger Library 2.4
- Hydraulics Library 4.11
- Hydro Power Library 2.10
- Liquid Cooling Library 2.4
- Pneumatics Library 2.7
- Thermal Power Library 1.19
- Vapor Cycle Library 2.4
- Vehicle Dynamics Library 3.4

E.42. Release notes for the OPTIMICA Compiler Toolkit version 1.12

A major change in this release is the update of Java version used into OpenJDK 11.

E.42.1. Compiler Changes

Important changes to the compiler:

- Added support for unspecified enumerations.
- Protection annotations are now used to determine if a variable is included in modelDescription.xml.
- Turned off loop unrolling in functions to increase performance.
- Improved array initialization for primitive arrays in global constants.
- Added option for controlling maximum allowed size of dynamic state sets. Changed default from 8 to 10.

E.42.2. API Improvements

Important improvements to the compiler API:

- Added method for retrieving access protection from classes.
- Allow evaluation of binding expressions in source and instance tree given user-provided variable values.
- Added experimental method for copying a class into another class.
- Added experimental methods for copying components.
- Improving and finalizing the modifiable and qualified name methods.

E.42.3. MATLAB interface

Important updates to the MATLAB® interface:

- Changed the backend of OCT (MATLAB) to do compiler specific calls through the MATLAB® Python interface instead of Java.
- Made it possible to release specific residual variables without having to release related indices for iteration variables.

E.42.4. Compliance

E.42.4.1. Modelon libraries

For OPTIMICA Compiler Toolkit version 1.12 the following libraries provided by Modelon are compliant:

- Engine Dynamics Library 2.3
- Environmental Control Library 3.6
- Fuel Cell Library 1.9
- Fuel System Library 4.5
- Heat Exchanger Library 2.3
- Hydraulics Library 4.10
- Hydro Power Library 2.9
- Liquid Cooling Library 2.3
- Modelon Base Library 3.3
- Pneumatics Library 2.6

- Thermal Power Library 1.18
- Vapor Cycle Library 2.3
- Vehicle Dynamics Library 3.3

E.43. Release notes for the OPTIMICA Compiler Toolkit version 1.10

E.43.1. Compiler Changes

Important changes to the compiler:

- Fixed a bug that resulted in variables not being re-initialized in certain cases by adding writeback of reinitialization in block residual functions.

E.43.2. API Improvements

Important improvements to the compiler API:

- A short class declaration can now be changed to point to another class via setSuperClass(String).

E.43.3. Compliance

E.43.3.1. Modelon libraries

For OPTIMICA Compiler Toolkit version 1.10 the following libraries provided by Modelon are compliant:

- Engine Dynamics Library 2.3
- Environmental Control Library 3.6
- Fuel Cell Library 1.9
- Fuel System Library 4.5
- Heat Exchanger Library 2.3
- Hydraulics Library 4.10
- Hydro Power Library 2.9
- Liquid Cooling Library 2.3

- Modelon Base Library 3.3
- Pneumatics Library 2.6
- Thermal Power Library 1.18
- Vapor Cycle Library 2.3
- Vehicle Dynamics Library 3.3

E.44. Release notes for the OPTIMICA Compiler Toolkit version 1.8

E.44.1. API Improvements

Important improvements to the compiler API:

- Added experimental feature to instance components that returns the annotation node from the component's class in the context of the component.

In addition there are some minor performance and debug improvements.

E.44.2. Compliance

E.44.2.1. Modelon libraries

For OPTIMICA Compiler Toolkit version 1.8 the following libraries provided by Modelon are compliant:

- Engine Dynamics Library 2.2
- Environmental Control Library 3.5
- Fuel Cell Library 1.8
- Fuel System Library 4.4
- Heat Exchanger Library 2.2
- Hydraulics Library 4.9
- Hydro Power Library 2.8
- Liquid Cooling Library 2.2

- Pneumatics Library 2.5
- Thermal Power Library 1.17
- Vapor Cycle Library 2.2
- Vehicle Dynamics Library 3.2

E.45. Release notes for the OPTIMICA Compiler Toolkit version 1.6.1

E.45.1. API Improvements

Important improvements to the compiler API:

- Fixed bug where API method getAllMatchingRedeclareChoices() failed for a redeclared class declaration, where the constraining type was on the original declaration and not a fully qualified name.
- Fixed a bug in experimental_rename that corrupted whitespace formatting in getSourceText().

E.46. Release notes for the OPTIMICA Compiler Toolkit version 1.6

E.46.1. Compiler Changes

Important changes to the compiler:

- Fix bug where prettyprint for modifier failed.
- Fixed type checking of array sizes for elements in arrays of records.
- Made sure to release the package.order file after reading.

E.46.2. API Improvements

Important improvements to the compiler API:

- Changed modifier exists method to work correctly for unmodified components.
- Fixed experimental_delete for classes in structured class retrieved by anything else than traversal.
- Fixed bug in generation of SVG icons in the instance tree.

E.46.3. Python Packages Improvements

Important improvements to our Python packages:

- Changed exception type in the constructor of FMUProblem for unbalanced problems.

E.46.4. Compliance

E.46.4.1. Modelon libraries

For OPTIMICA Compiler Toolkit version 1.6 the following libraries provided by Modelon are compliant:

- Engine Dynamics Library 2.2
- Environmental Control Library 3.5
- Fuel Cell Library 1.8
- Fuel System Library 4.4
- Heat Exchanger Library 2.2
- Hydraulics Library 4.9
- Hydro Power Library 2.8
- Liquid Cooling Library 2.2
- Pneumatics Library 2.5
- Thermal Power Library 1.17
- Vapor Cycle Library 2.2
- Vehicle Dynamics Library 3.2

E.47. Release notes for the OPTIMICA Compiler Toolkit version 1.5

E.47.1. Compiler Changes

Important changes to the compiler:

- Updated the GUID calculation in FMUs to only be dependent on the generated model description XML.

- Fixed crash bug when combinding exlude_internal_variables in combination with hand guided tearing.
- Updated calculation order of start values and dependent parameters to reduce number of evaluations.
- Fixed that binding expression splitting doesn't check bounds.
- Don't allow any inlining of equations in when loops.
- Fixed exception from specific combination of record with array of unknown size in if expression in binding expression.
- Enabled evaluation of non-literal expressions in annotations.

E.47.2. API Improvements

The focus areas of this release have mainly been performance issues and bug fixes. The most important improvements to the compiler API:

- Allowed some limited evaluation of binding expressions in source and instance tree.
- API doesn't any longer throw exception when iterating beyond the array components with binding expression.
- Fixed the unability to add a modification to a component with a value (binding expression).
- Fixed bug in getSizeParameter() which caused error when declared in class inside a component.

E.47.3. Python Packages Improvements

Important improvements to our Python packages:

- Fixed issue with reusing the PyFMI computed FD Jacobian.
- Fixed such that log viewer get jacobians returns matrices even for 1-dimensional problems.
- Added check against empty lists to the constructor of FMUProblem.
- Improved performance of the constructor in FMUProblem.

E.47.4. Compliance

E.47.4.1. Modelon libraries

For OPTIMICA Compiler Toolkit version 1.5 the following libraries provided by Modelon are compliant:

- Engine Dynamics Library 2.2

- Environmental Control Library 3.5
- Fuel Cell Library 1.8
- Fuel System Library 4.4
- Heat Exchanger Library 2.2
- Hydraulics Library 4.9
- Hydro Power Library 2.8
- Liquid Cooling Library 2.2
- Pneumatics Library 2.5
- Thermal Power Library 1.17
- Vapor Cycle Library 2.2
- Vehicle Dynamics Library 3.2

E.48. Release notes for the OPTIMICA Compiler Toolkit version 1.4

E.48.1. Runtime Changes

Important changes to Runtime:

- Improved the runtime logging framework due to performance issues with specific models.

E.48.2. Compiler Changes

Important changes to the compiler:

- Minor improvements of compilation and simulation performance.
- loadResource on directories now always leads to compile time evaluation of path.

E.48.3. API Improvements

The focus areas of this release have mainly been performance issues and bug fixes. The most important improvements to the compiler API:

- Added support for obtaining scalar parameters used as array sizes.

- Improved behaviour of getAllMatchingRedeclareChoices() on redeclarations.
- Connectors without placement annotation are no longer rendered in icons for the instance tree.
- Improved functionality of getSourceText().
- The modification tree constructed by setBindingExpression should now be correct.

E.48.4. Python Packages Improvements

Important improvements to our Python packages:

- Made several improvements to the diagnostic scripts in the steady state framework.
- The steady state solver can now be used without generating a logfile.
- Added the possibility to use zero-dimensional problems in the steady state framework.

E.48.5. Compliance

E.48.5.1. Modelon libraries

For OPTIMICA Compiler Toolkit version 1.4 the following libraries provided by Modelon are compliant:

- Engine Dynamics Library 2.2
- Environmental Control Library 3.5
- Fuel Cell Library 1.8
- Fuel System Library 4.4
- Heat Exchanger Library 2.2
- Hydraulics Library 4.9
- Hydro Power Library 2.8
- Liquid Cooling Library 2.2
- Pneumatics Library 2.5
- Thermal Power Library 1.17
- Vapor Cycle Library 2.2
- Vehicle Dynamics Library 3.2

E.49. Release notes for the OPTIMICA Compiler Toolkit version 1.3.1

E.49.1. API Improvements

Important improvements to the compiler API:

- Fixed a defect which caused an exception to be thrown when calling the getAllMatchingRedeclareChoices() method in InstanceClass and InstanceComponent if there was a non-library file loaded.

E.49.2. Compliance

E.49.2.1. Modelon libraries

For OPTIMICA Compiler Toolkit version 1.3.1 the following libraries provided by Modelon are compliant:

- Engine Dynamics Library 2.1
- Environmental Control Library 3.4
- Fuel Cell Library 1.7
- Fuel System Library 4.3
- Heat Exchanger Library 2.1
- Hydraulics Library 4.8
- Hydro Power Library 2.7
- Liquid Cooling Library 2.1
- Pneumatics Library 2.4
- Thermal Power Library 1.16
- Vapor Cycle Library 2.1
- Vehicle Dynamics Library 3.1

E.50. Release notes for the OPTIMICA Compiler Toolkit version 1.3

- Added partial and experimental functionality for managing model editing.

E.50.1. Compiler Changes

Important changes to the compiler:

- Fixed a bug where linear equation elimination broke HGT equations.
- Support for initial parameter external objects.

E.50.2. API Improvements

Important improvements to the compiler API:

- Added partial and experimental functionality for managing model editing.
- Added possibility to get size information of instance components.
- Added the ability to remove modifiers.
- Added support for obtaining size dimension type information.
- Added support in API for getting source text of source classes.
- Added support in API for getting all matching redeclare choices for instance classes and instance components.

E.50.3. Python Packages Improvements

Important improvements to our Python packages:

- Added diagnostics package to Python steady states module.
- Added license check for using the solver in the Python steady state module.
- Fixed issue with getting time varying variables (sometimes wrong variables were returned)
- Added functionality to set enumerations with strings.
- Improved input handling for FMI2.

E.50.4. Compliance

E.50.4.1. Modelon libraries

For OPTIMICA Compiler Toolkit version 1.3 the following libraries provided by Modelon are compliant:

- Engine Dynamics Library 2.1

- Environmental Control Library 3.4
- Fuel Cell Library 1.7
- Fuel System Library 4.3
- Heat Exchanger Library 2.1
- Hydraulics Library 4.8
- Hydro Power Library 2.7
- Liquid Cooling Library 2.1
- Pneumatics Library 2.4
- Thermal Power Library 1.16
- Vapor Cycle Library 2.1
- Vehicle Dynamics Library 3.1

E.51. Release notes for the OPTIMICA Compiler Toolkit version 1.2

- Support for 64-bit Python has been added together with new and updated Python packages, see Section E.51.3. This enables simulation of 64-bit FMUs with PyFMI.
- The bundled gcc C-compiler is now updated to gcc-tdm 5.1.0 which enables compilation of 64-bit FMUs.
- Full string support
- Changed the default FMI version when compiling FMUs from Python to FMI 2.0 instead of FMI 1.0.
- Changed default file storing method to binary format when simulating FMUs with PyFMI in Python.
- Achieved significant speedup in some models by adding a new memory allocation algorithm.
- Added support for source code FMUs
- Added support for steady-state computations from Python.
- Added functionality for obfuscating variable names in generated FMU
- Added functionality for hiding internal variables in the FMU.

- There is now support for OPC communication through the bundled Python package OpenOPC together with *Graybox OPC Automation Wrapper*.

E.51.1. Compiler Changes

Important changes to the compiler:

- Added option for disabling external evaluation during variability propagation
- Improved Hand-Guided Tearing: Allow initial parameter variability in hold expression

E.51.2. API Improvements

Important improvements to the compiler API:

- Improved access control for encrypted libraries.
- Improved library license checks.
- Added a representation of modifiers which enables the retrieval of modification points for classes and components.
- Improved handling of primitive types and enums.
- Improved SVG rendering.

E.51.3. Python distributions

With this release there are now two bundled Python interpreters, one for 32-bit mode and one for 64-bit mode. In addition, there are now many more Python packages included and previously bundled packages are updated to newer versions. Among the new packages we have: *XLwings, openpyxl, freeopcua, jupyter, SALib, natsort, pyserial, pyro* and *coverage.*

E.51.4. Compliance

E.51.4.1. Modelon libraries

For OPTIMICA Compiler Toolkit version 1.2 the following libraries provided by Modelon are compliant:

- Engine Dynamics Library 2.1
- Environmental Control Library 3.4
- Fuel Cell Library 1.7

- Fuel System Library 4.3
- Heat Exchanger Library 2.1
- Hydraulics Library 4.8
- Hydro Power Library 2.7
- Liquid Cooling Library 2.1
- Pneumatics Library 2.4
- Thermal Power Library 1.16
- Vapor Cycle Library 2.1
- Vehicle Dynamics Library 3.1

E.52. Release notes for the OPTIMICA Compiler Toolkit version 1.1

One of the focus areas for this release has been performance. The following are the main improvements:

- With this release compiler performance has been improved with up to 30% in compilation time and 20% in memory reduction for some benchmark models.
- Support for the sparse solver SuperLU is added for simulation of FMI2 ME FMUs in PyFMI.
- A sparse solver has been added to solve large linear blocks.

The OPTIMICA Compiler Toolkit API has been extended with some new features.

E.52.1. API Improvements

The main improvement to the compiler API is the added decoupling to the compiler API and underlying model. This enables the API to flush the underlying model without affecting the API nodes used by the user. In addition, there are several minor new features and a couple of minor bug fixes.

E.52.2. Compliance

E.52.2.1. Modelon libraries

For OPTIMICA Compiler Toolkit version 1.1 the following libraries provided by Modelon are compliant:

- Engine Dynamics Library 2.0

- Environmental Control Library 3.3.1
- Fuel Cell Library 1.6
- Fuel System Library 4.2
- Heat Exchanger Library 2.0
- Hydraulics Library 4.7
- Hydro Power Library 2.6.4
- Liquid Cooling Library 2.0
- Pneumatics Library 2.3
- Thermal Power Library 1.15
- Vapor Cycle Library 2.0
- Vehicle Dynamics Library 3.0

E.52.2.2. Modelica Standard Library (MSL)

No changes since previous release.

E.52.2.3. IBPSA

No changes since previous release.

E.53. Release notes for the OPTIMICA Compiler Toolkit version 1.0

Initial release.

E.53.1. Compliance

E.53.1.1. Modelon libraries

For OPTIMICA Compiler Toolkit version 1.0 the following libraries provided by Modelon are compliant with some limitations:

Table E.1

Library	Limitations

Engine Dynamics Library 1.2.8	Fully compliant
Environmental Control Library 3.3	Fully compliant
Fuel Cell Library 1.5	Fully compliant
Fuel System Library 4.1	Fully compliant
Heat Exchanger Library 1.6	Numerical robustness issues for *HeatExchangerTests.HeatExchangers.FlatTubes.Experiments.CondenserReceiver*
Hydraulics Library 4.6	Fully compliant
Hydro Power Library 2.6.3	Fully compliant
Liquid Cooling Library 1.5.3	Fully compliant
Pneumatics Library 2.2	Fully compliant
Thermal Power Library 1.14	Numerical robustness issues for: *ThermalPowerTests.Examples.Coal.SuperCriticalRankine400MWe*, *ThermalPowerTests.Medium.FastExhaustWithAsh.OverCriticalRankine400MWe*, *ThermalPowerTests.Medium.WaterPolynominal.HRSG* and *ThermalPowerTests.Medium.WaterPolynominal.OverCriticalRankine400MWe*.
Vapor Cycle Library 1.5	Fully compliant
Vehicle Dynamics Library 2.5	Fully compliant

E.53.1.2. Modelica Standard Library (MSL)

For this release, the Modelica Standard Library (MSL) version 3.2.2 build 3 with the following patches applied is used:

- To the model *Modelica.Blocks.Examples.NoiseExamples.ActuatorWithNoise* defined in Modelica/Blocks/package.mo a StateSelect.always is added for Controller.y1. With this patch dynamic state selection is avoided. See also the reported issue 2189 on the GitHub repository for Modelica Association.

- The model *Modelica.Fluid.Examples.Tanks.TanksWithOverflow* does not initialize with the original parametrization in MSL 3.2.2 build 3 due to variable bounds not being respected, see issue 2060 on the GitHub repository for Modelica Association. In the patch additional fluid flows through an overflow pipe if the level of the upper tank exceeds 6 meters instead of 10 meters.

- The state selection in *Modelica.Magnetic.FluxTubes.Examples.Hysteresis.HysteresisModelComparison* is patched to improve the numerical robustness, see issue 2248 on the GitHub repository for Modelica Association.

- In *Modelica.Fluid.Examples.TraceSubstances.RoomCO2WithControls* the experiment tolerance is tightened to 1e-007 instead of 1e-006 to avoid chattering.

- In *Modelica.Fluid.Examples.InverseParameterization* pump.m_flow_start is set to 0.5 instead of 0.0. With this change the correct branch is chosen in the actualStream operator, see issue 2063 on the GitHub repository for Modelica Association.

With the patches listed above applied, all example models in version 3.2.2 build 3 of MSL simulate correctly with OPTIMICA Compiler Toolkit.

E.53.1.3. IBPSA

There is support for IBPSA, a Modelica library for building and disctrict energy systems. More information can be found on the Modelica IBPSA library GitHub website.

Bibliography

[Jak2007] Johan Åkesson. *Tools and Languages for Optimization of Large-Scale Systems*. LUTFD2/TFRT--1081--SE. Lund University. Sweden. 2007.

[Jak2008b] Johan Åkesson, Görel Hedin, and Torbjörn Ekman. *Tools and Languages for Optimization of Large-Scale Systems*. 117-131. *Electronic Notes in Theoretical Computer Science*. 203:2. April 2008.

[Jak2008a] Johan Åkesson. *Optimica - An Extension of Modelica Supporting Dynamic Optimization*. *Proc. 6th International Modelica Conference 2008*. Modelica Association. March 2008.

[Jak2010] Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. *Modeling and Optimization with Optimica and JModelica.org - Languages and Tools for Solving Large-Scale Dynamic Optimization Problem. Computers and Chemical Engineering*. 203:2. 2010.

[Eng2001] Peter Englezos and Nicolas Kalogerakis. *Applied Parameter Estimation for Chemical Engineers*. Marcel Dekker Inc. 2001.

[Mag2015] Fredrik Magnusson and Johan Åkesson. *Dynamic Optimization in JModelica.org*. 471-496. *Processes*. 3:2. 2015.

[Mag2016] Fredrik Magnusson. *Numerical and Symbolic Methods for Dynamic Optimization*. Lund University. Sweden. 2016.

[Kinsol2011] *Kinsol user's guide*. http://computation.llnl.gov/casc/sundials/documentation/kin_guide.pdf

[FMI2017] *Functional Mock-up Interface standard*. http://www.fmi-standard.org