JModelica.org User Guide

Version 1.6.1

JModelica.org User Guide: Version 1.6.1

Publication date 2012-01-15

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 is 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

1. Introduction 1
    1. About JModelica.org 1
    2. Mission Statement 1
    3. Technology 1
    4. Architecture 2
    5. Extensibility 2
2. Installation 4
    1. Supported platforms 4
    2. Prerequisites 4
        2.1. Java 4
        2.2. Python 4
    3. Binary distribution 5
        3.1. Windows 5
            3.1.1. Bundled tools, libraries and Python packages 5
            3.1.2. Installing JModelica.org with Windows installer 6
        3.2. Linux 6
        3.3. Mac OS X 7
    4. Software Development Kit 7
        4.1. Prerequisites 7
        4.2. Installing the JModelica.org SDK 7
            4.2.1. Bundled tools, libraries and Python packages 7
            4.2.2. Step-by-step instructions 7
        4.3. Changing Ipopt solver 8
3. Getting started 9
    1. The JModelica.org Python packages 9
    2. Starting a Python session 9
        2.1. Windows 9
        2.2. Linux or Mac OS 9
    3. Run an example 10
    4. Check your installation 10
    5. Redefining the JModelica.org environment 11
        5.1. Example redefining IPOPT_HOME 11
    6. The JModelica.org user forum 12
4. Working with Models 13
    1. Introduction to models 13
        1.1. The different model objects in JModelica.org 13
    2. Compilation 13
        2.1. Simple JMU compilation example 14
        2.2. Simple FMU compilation example 14
        2.3. Simple FMUX compilation example 15

2.4. Compiler settings 15
2.4.1. compile_jmu parameters 15
2.4.2. compile_fmu parameters 17
2.4.3. compile_fmux parameters 19
2.4.4. Compiler options 20
2.4.5. Compiler targets for `compile_jmu` 23
2.4.6. Compiler targets for `compile_fmu` 23
2.4.7. Compiler targets for `compile_fmux` 23
2.5. Compilation in more detail 23
2.5.1. Create a compiler 23
2.5.2. Source tree generation and flattening 24
2.5.3. Code generation 24
3. Loading models 24
3.1. The JMU 25
3.2. The FMU 25
3.3. The FMUX 25
3.4. Loading a JMU 25
3.5. Loading an FMU 26
3.6. Loading an FMUX 26
4. Variable and parameter manipulation 26
4.1. Model variable XML files 26
4.2. Setting and getting variables 26
4.3. Loading and saving parameters 27
4.3.1. Loading XML values file 27
4.3.2. Writing to XML values file 28
5. Debugging models 29
5. FMI Interface 32
1. Overview of JModelica.org FMI Python package 32
2. Example 33
2.1. Simulation using the native FMI interface 34
2.1.1. Implementation 34
6. Initialization 39
1. Solving DAE initialization problems 39
2. How JModelica.org creates the initialization system of equations 40
3. Initialization algorithms 41
3.1. Initialization using IPOPT 41
3.2. Initilization using KInitSolveAlg 42
3.2.1. The use of constraints 43
3.2.2. Verbosity of KINSOL 44
7. Simulation 45
1. Introduction 45
2. A first example 45

3. Simulation of Models 47
3.1. Arguments 48
3.1.1. Input 48
3.1.2. Options for JMUModel 49
3.1.3. Options for FMUModel 52
3.2. Return argument 54
4. Examples 54
4.1. Simulation with inputs 55
4.2. Simulation of a discontinuous system 59
4.3. Simulation of a high-index model 61
4.4. Simulation and parameter sweeps 63
4.5. Simulation with sensitivities 65
4.6. Simulation of an FMU 67
8. Optimization 71
1. Introduction 71
2. A first example 71
3. Solving optimization problems 73
3.1. Algorithms 75
3.1.1. Direct collocation 75
4. Optimal control 78
4.1. Compile and instantiate a model object 79
4.2. Solve the DAE initialization problem 79
4.3. Solving an optimal control problem 80
4.4. Verify optimal control solution 84
4.5. Exercises 86
4.6. References 86
5. Minimum time problems 87
6. Parameter optimization 89
7. Scaling 102
8. Optimization using Pseduo-Spectral methods. 103
8.1. CasADi 103
8.2. Options 103
8.3. Examples 105
8.3.1. Van der Pol oscillator 106
8.3.2. Hohmann Transfer 107
8.4. Limitations 111
9. Derivative-Free Model Calibration of FMUs 112
9. Graphical User Interface for Visualization of Results 118
1. Plot GUI 118
1.1. Introduction 118
1.2. Edit Options 120
1.3. View Options 124

1.4. Example 124
10. Optimica 126
1. A new specialized class: `optimization` 127
2. Attributes for the built in class Real 127
3. A Function for accessing instant values of a variable 128
4. Class attributes 128
5. Constraints 129
11. Abstract syntax tree access 131
1. Tutorial on Abstract Syntax Trees (ASTs) 131
1.1. About Abstract Syntax Trees 131
1.2. Load the Modelica standard library 132
1.3. Count the number of classes in the Modelica standard library 133
1.4. Dump the instance AST 134
1.5. Flattening of the filter model 137
12. Limitations 139
13. Release notes 141
1. Release notes for JModelica.org version 1.6 141
1.1. Highlights 141
1.2. Compilers 141
1.2.1. Index reduction 141
1.2.2. Modelica compliance 141
1.3. Python 141
1.3.1. Graphical user interface for visualization of simulation and optimization results 141
1.3.2. Simulation with function inputs 141
1.3.3. Compilation of XML models 142
1.3.4. Python version upgrade 142
1.4. Optimization 142
1.4.1. Derivative- free optimization of FMUs 142
1.4.2. Pseudo spectral methods for dynamic optimization 142
1.5. Eclipse Modelica plugin 142
1.6. Contributors 142
1.6.1. Previous contributors 143
2. Release notes for JModelica.org version 1.5 143
2.1. Highlights 143
2.2. Compilers 144
2.2.1. When clauses 144
2.2.2. Equation sorting 144
2.2.3. Connections 144
2.2.4. Eclipse IDE 144
2.2.5. Miscellaneous 144
2.3. Simulation 144
2.3.1. FMU export 144

2.3.2. Simulation of ODEs 144
2.3.3. Simulation of hybrid and sampled systems 145
2.4. Initialization of DAEs 145
2.5. Optimization 145
2.6. Contributors 145
2.6.1. Previous contributors 145
3. Release notes for JModelica.org version 1.4 146
3.1. Highlights 146
3.2. Compilers 146
3.2.1. Enumerations 146
3.2.2. Miscellaneous 146
3.2.3. Improved reporting of structural singularities 146
3.2.4. Automatic addition of initial equations 146
3.3. Python interface 146
3.3.1. Models 146
3.3.2. Compiling 147
3.3.3. initialize, simulate and optimize 147
3.3.4. Result object 147
3.4. Simulation 147
3.4.1. Input trajectories 147
3.4.2. Sensitivity calculations 147
3.4.3. Write scaled simulation result to file 148
3.5. Contributors 148
3.5.1. Previous contributors 148
4. Release notes for JModelica.org version 1.3 148
4.1. Highlights 148
4.2. Compilers 149
4.2.1. The Modelica compiler 149
4.2.2. The Optimica compiler 150
4.3. JModelica.org Model Interface (JMI) 150
4.3.1. The collocation optimization algorithm 150
4.4. Assimulo 151
4.5. FMI compliance 151
4.6. XML model export 151
4.6.1. `noEvent` operator 151
4.6.2. `static` attribute 151
4.7. Python integration 151
4.7.1. High-level functions 151
4.7.2. File I/O 152
4.8. Contributors 152
4.8.1. Previous contributors 152
5. Release notes for JModelica.org version 1.2 152

5.1. Highlights 152
5.2. Compilers 153
5.2.1. The Modelica compiler 153
5.2.2. The Optimica Compiler 155
5.3. The JModelica.org Model Interface (JMI) 155
5.3.1. General 155
5.4. The collocation optimization algorithm 155
5.4.1. Piecewise constant control signals 155
5.4.2. Free initial conditions allowed 156
5.4.3. Dens output of optimization result 156
5.5. New simulation package: Assimulo 156
5.6. FMI compliance 156
5.7. XML model export 156
5.8. Python integration 156
5.8.1. New high-level functions for optimization and simulation 157
5.9. Contributors 157
5.9.1. Previous contributors 157
Bibliography 158
Index 159

List of Figures

1.1. JModelica platform architecture. 2
2.1. Selecting Python packages in the *Choose components* window. 6
5.1. Simulation result 38
7.1. Simulation result of the Van der Pol oscillator. 47
7.2. A schematic picture of the quadruple tank process. 55
7.3. Tank levels 58
7.4. Input trajectories 59
7.5. Electric Circuit 60
7.6. Simulation result 61
7.7. Modelica.Mechanics.Rotational.First connection diagram 62
7.8. Simulation result for Mecahnics.Rotational.Examples.First 63
7.9. Simulation result-phase plane 65
7.10. Sensitivity results. 67
7.11. Full Robot 68
7.12. Robot Results 69
7.13. Comparison with Dymola 70
8.1. Optimal profiles for the VDP oscillator 73
8.2. Optimal profiles for the CSTR problem. 84
8.3. Optimal control profiles and simulated trajectories corresponding to the optimal control input. 86
8.4. Minimum time profiles for the Van der Pol Oscillator. 89
8.5. A schematic picture of the quadruple tank process. 89
8.6. Measured state profiles. 92
8.7. Control inputs used in the identification experiment. 92
8.8. Simulation result for the nominal model. 94
8.9. State profiles corresponding to estimated values of a1 and a2. 97
8.10. State profiles corresponding to estimated values of a1, a2, a3 and a4. 99
8.11. Handling of discontinuities. 105
8.12. The Van der Pol oscillator optimized using Gauss PseudoSpectral method. 107
8.13. Hohmann Transfer 111
8.14. The Furuta pendulum. 112
8.15. Measurements of θ and φ for the Furuta pendulum. 113
8.16. Measurements and model simulation result for φ and θ when using nominal parameter values in the Furuta pendulum model. 114
8.17. Measurements and model simulation results for φ and θ with nominal and optimal parameters in the model of the Furuta pendulum. 117
9.1. Overview of JModelica.org Plot GUI 118
9.2. A result file has been loaded. 119
9.3. Plotting a trajectory. 119
9.4. Figure Options. 120
9.5. Figure Axis and Labels Options. 121

9.6. Figure Lines and Legends options. 121
9.7. An additional plot has been added. 122
9.8. Moving Plot Figure. 122
9.9. GUI after moving the plot window. 123
9.10. Complex Figure Layout. 123
9.11. Figure View Options. 124
9.12. Multiple figure example. 125
10.1. Optimization result 130

List of Tables

2.1. Required Python packages 4
2.2. IPython dependencies on Windows 5
4.1. Compiler options 20
5.1. Conversion table. 32
6.1. Options for the collocation-based optimization algorithm 41
6.2. Options for KInitSolveAlg 42
6.3. Options for KINSOL contained in the `KINSOL_options` dictionary 43
6.4. Values allowed in the `constraints` array 43
6.5. Verbosity levels in KINSOL 44
7.1. General options for AssimuloAlg. 50
7.2. Selection of solver arguments for CVode 51
7.3. Selection of solver arguments for IDA 51
7.4. General options for AssimuloFMIAlg. 52
7.5. Selection of solver arguments for CVode 53
7.6. Result Object 54
8.1. Options for the collocation-based optimization algorithm 76
8.2. Parameters for the quadruple tank process. 90
8.3. Options for the Pseudospectral optimization algorithms. 104

Chapter 1. Introduction

1. About JModelica.org

JModelica.org is an extensible Modelica-based open source platform for optimization, simulation and analysis of complex dynamic systems. The main objective of the project is to create an industrially viable open source platform for optimization of Modelica models, while offering a flexible platform serving as a virtual lab for algorithm development and research. As such, JModelica.org is intended to provide a platform for technology transfer where industrially relevant problems can inspire new research and where state of the art algorithms can be propagated form academia into industrial use. JModelica.org is a result of research at the Department of Automatic Control, Lund University, [Jak2007]and is now maintained and developed by Modelon AB in collaboration with academia.

2. Mission Statement

To offer a community-based, free, open source, accessible, user and application oriented Modelica environment for optimization and simulation of complex dynamic systems, built on well-recognized technology and supporting major platforms.

3. Technology

JModelica.org relies on the established modeling language Modelica. Modelica targets modeling of complex heterogeneous physical systems, and is becoming a de facto standard for dynamic model development and exchange. There are numerous model libraries for Modelica, both free and commercial, including the freely available Modelica Standard Library (MSL).

A unique feature of JModelica.org is the support for the innovative extension Optimica. Optimica enables you to conveniently formulate optimization problems based on Modelica models using simple but powerful constructs for encoding of optimization interval, cost function and constraints. Optimica also features annotations for choosing and tailoring the underlying numerical optimization algorithm a particular optimization problem.

The JModelica.org compilers are developed in the compiler construction framework JastAdd. JastAdd is based on established concepts, including object orientation, aspect orientation and reference attributed grammars. Compilers developed in JastAdd are specified in terms of declarative attributes and equations which together forms an executable specification of the language semantics. In addition, JastAdd targets extensible compiler development which makes it easy to experiment with language extensions.

For user interaction JModelica.org relies on the Python language. Python offers an interactive environment suitable for scripting, development of custom applications and prototype algorithm integration. The Python packages Numpy and Scipy provide support for numerical computation, including matrix and vector operations, basic linear algebra and plotting. The JModelica.org compilers as well as the model executables/dlls integrate seemlessly with Python and Numpy.

4. Architecture

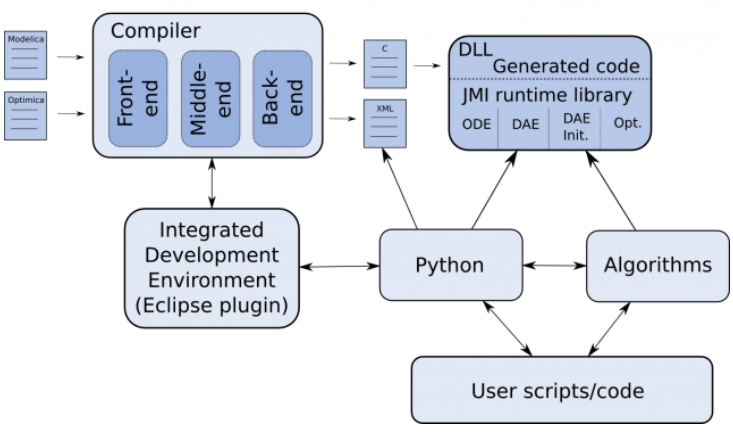


Figure 1.1 JModelica platform architecture.

The JModelica.org platform consists of a number of different parts:

- The compiler front-ends (one for Modelica and one for Modelica/Optimica) transforms Modelica and Optimica code into a flat model representation. The compilers also check the correctness of model descriptions and reports errors.
- The compiler back-ends generates C code and XML code for Modelica and Optimica. The C code contains the model equations, cost functions and constraints whereas the XML code contains model meta data such as variable names and parameter values.
- The JModelica.org runtime library is written in C and contains supporting functions needed to compile the generated model C code. Also, the runtime library contains an integration with CppAD, a tool for computation of high accuracy derivatives by means of automatic differentiation.
- Currently, JModelica.org features one particular algorithm for solving dynamic optimization problems. The algorithm is based on collocation on finite elements and relies on the solver IPOPT for obtaining a solution of the resulting NLP.
- JModelica.org uses Python for scripting and prototyping. For this purpose, a Python package is under development with the objective of offering functions for driving the compilers and for accessing the (compiled) functions in the runtime library/generated C code.

5. Extensibility

The JModelica.org platform is extensible in a number of different ways:

- JModelica.org features a C interface for efficient evaluation of model equations, the cost function and the constraints: the JModelica Model Interface (JMI). JMI also contains functions for evaluation of derivatives and sparsity and is intended to offer a convenient interface for integration of numerical algorithms.

- In addition to the the C interface, model meta data can be exported in XML. In the future this feature is intended to be extended to include full model export in XML, which in turn enables use of XML techniques such as XPATH and XSLT.

- JastAdd produces compilers encoded in pure Java. As a result, the JModelica.org compilers are easily embedded in other applications aspiring to support Modelica and Optimica. In particular, a Java API for accessing the flat model representation and an extensible template-based code generation framework is offered.

- The JModelica.org compilers are developed using the compiler construction framework JastAdd. JastAdd features extensible compiler construction, both at the language level and at the implementation level. This feature is explored in JModelica.org where the Optimica compiler is implemented as a fully modular extension of the core Modelica compiler. The JModelica.org platform is a suitable choice for experimental language design and research.

An overview of the JModelica.org platform is given [Jak2010]

Chapter 2. Installation

1. Supported platforms

JModelica.org can be installed on Linux, Mac OS X, and Windows (XP, Vista, 7) with 32-bit or 64-bit architectures. Most development work is carried out on 32-bit Mac OS X , 32 and 64-bit Linux and 32-bit Windows 7, so these platforms tend to be best tested.

2. Prerequisites

Make sure to install the required software components listed in this section before installing JModelica.org.

2.1. Java

It is required to have a Java Runtime Environment (JRE) version 6 installed on your computer.

Install a JRE

1. Get a 32-bit / 64-bit JRE installer suitable for your platform. For Windows users the 32-bit JRE must be installed, even for 64-bit computers.

2. Run the installer and follow the instructions to install the JRE on your computer.

2.2. Python

Python 2.7 with the following additional packages are required.

Install Python 2.7

1. Get a Python 2.7 installer suitable for your platform. Note that for Windows users the 32-bit version must be chosen, even for 64-bit computers.

2. Install the additional Python packages listed in the tables below.

Table 2.1 Required Python packages

Package	Recommended version	Description
NumPy	1.6.1	The fundamental package needed for scientific computing with Python.
SciPy	0.9.0	A library of algorithms and mathematical tools for Python.

Package	Recommended version	Description
matplotlib	1.0.1	A plotting library for Python, with a MATLAB like interface.
IPython	0.11	An interactive shell for Python with additional shell syntax, code highlighting, tab completion, string completion, and rich history.

IPython has extra dependencies on Windows, in this case these extra Python packages must be installed:

Table 2.2 IPython dependencies on Windows

Package	Recommended version	Description
Pyreadline	1.7	A Python implmementation of GNU readline.
setuptools	0.6c11	A package for downloading, building, installing, upgrading, and uninstalling Python packages.

Please note that the recommended version of a package is not always the same as the latest version available. It is the package version that JModelica.org has been tested and verified to be working with.

3. Binary distribution

3.1. Windows

Pre-built binary distributions for Windows are available in the Download section of www.jmodelica.org.

The JModelica.org Windows installer contains a binary distribution of JModelica.org built using the JModelica.org-SDK, bundled with required third-party software components. The JModelica.org Windows installer sets up a pre-configured complete environment with convenient start menu shortcuts.

3.1.1. Bundled tools, libraries and Python packages

The following tools, libraries and Python packages are bundled with the binary distribution of JModelica.org.

- MingGW
- Ipopt
- JPype
- lxml

- nose
- wxPython
- SUNDIALS
- Cython
- SuperLU
- Casadi

3.1.2. Installing JModelica.org with Windows installer

Make sure that all prerequisite components described in Section 2 are installed before continuing.

1. Download a JModelica.org Windows binary installer and save the executable file somewhere on your computer.
2. Run the file by double-clicking and selecting "Run" if prompted with a security warning. This will launch an installer which should be self-explanatory.
    - In the *Choose Components* window, select which of the bundled Python packages that should be installed. Make sure that any package not already installed on your computer is checked.

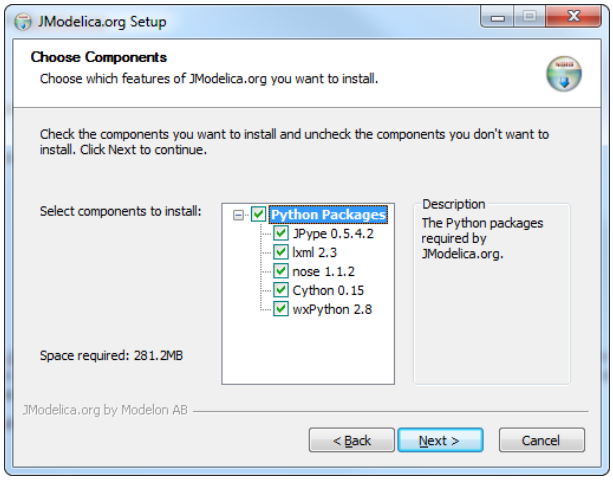


Figure 2.1 Selecting Python packages in the *Choose components* window.

3.2. Linux

Currently, no pre-built binary distributions are provided for Linux.

3.3. Mac OS X

Currently, no pre-built binary distributions are provided for Mac OS X.

4. Software Development Kit

The JModelica.org Software Development Kit (SDK) is a bundle of tools needed to build the JModelica.org sources on Windows. The SDK is mainly intended for those of you who want to do some developement work on the platform. Another reason to work with the SDK is if you want to be able to build new versions with the latest code changes. For a well documented, tested and stable version of JModelica.org, we refer to our binary distribution which is built (for Windows) with every release.

The JModelica.org SDK Windows installer sets up a complete pre-configured environment with convenient start menu shortcuts.

4.1. Prerequisites

The Python prerequisites for installing the JModelica.org SDK are the same as listed in Section 2.2. For Java however it is required to have a 32-bit JDK installed, not only a 32-bit JRE as for the binary installation.

Install a JDK

1. Get a 32-bit JDK.

2. Run the installer and follow the instructions to install the JDK on your computer.

4.2. Installing the JModelica.org SDK

Make sure that the prerequisites listed in Section 4.1 are installed before starting the JModelica.org SDK installation.

4.2.1. Bundled tools, libraries and Python packages

All tools, libraries and Python packages included in the binary distribution, see Section 3.1.1, are also included in the SDK. The SDK also includes a few extra tools:

- MinGW, MSYS
- CollabNet Subversion Client
- Apache Ant

4.2.2. Step-by-step instructions

1. Download the JModelica.org SDK and save the executable file somewhere on your computer.

2. Run the file by double-clicking and selecting "Run" if prompted with a security warning. This will launch an installer which should be self-explanatory. Select "Yes" at the end when asked if the source code should be checked out from JModelica.org as this is needed in the next step.

3. Build the sources:

   a. Start the msys shell from the JModelica.org start menu.

   b. Configure the sources for build and installation (this will create the subdirectories `build` and `install`) with:

   ```
   ./configure.sh
   ```

   c. Compile the sources with `make` from the `build` folder:

   ```
   cd build
   make && make install
   ```

4. Test the installation by starting a Python shell from the JModelica.org start menu and running an example. Starting the Python session from this start menu will set all the environment variables required to run the JModelica.org Python interface.

```
# Import the vdp example
from pyjmi.examples import vdp

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

4.3. Changing Ipopt solver

The solver bundled in JModelica.org is Ipopt-MUMPS. However, it is possible to use other Ipopt solvers. This requires a few changes in the setup and rebuilding JModelica. The procedure is:

1. Open the script file `configure.sh`, located in the SDK root folder, in a text editor.

2. Search for the line *--with-ipopt* and exchange the default Ipopt solver path with the path to the solver that you want to use. Save and close.

3. Do the whole installation procedure again, that is: run configure, make and make install.

4. Open the file `setenv.bat`, which is also found in the SDK root folder, in a text editor.

5. Look for the line which sets the `IPOPT_HOME` environment variable and exchange the default path with the path to your solver.

Chapter 3. Getting started

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

1. The JModelica.org Python packages

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

- `pymodelica` - Compile Modelica and Optimica code into model units (FMUs and JMUs, 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.
- `pyjmi` - Work with models that have been compiled into JMUs. Perform simulations, solve optimization problems, parameter manipulation, plot results etc.

The Python packages will be further presented in Chapter 4, Chapter 5, Chapter 6, Chapter 7, Chapter 8 and Chapter 9.

2. Starting a Python session

Starting a Python session differs somewhat depending on your operating system.

2.1. Windows

If you are running Windows, three different Python shells are available under the JModelica.org 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.

2.2. Linux or Mac OS

To start the IPython shell with PyLab on Linux or Mac OS X, open a terminal and enter the command:

```
> $JMODELICA_HOME/Python/jm_ipython.sh -pylab
```

3. Run 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 2 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.

4. Check your installation

The JModelica.org Python packages requires some additional Python packages in order to run correctly. Use the function `<package>.check_packages()`, where `<package>` is either `pymodelica`, `pyfmi` or `pyjmi`, to list which Python packages that are found on your computer. Missing or having the wrong version of a package can be a source of errors. Therefore it can be useful to run `<package>.check_packages()` after installation or when trouble-shooting.

```
# Import the function from pyfmi
from pyfmi import check_packages

# Run check_packages
check_packages()

Performing pyfmi package check
==============================

Platform..................... win32

Python version:.............. 2.7.2


Dependencies:

Package                      Version
-------                      -------
numpy........................ 1.6.1                        Ok
scipy........................ 0.9.0                        Ok
matplotlib................... 1.0.1                        Ok
```

```
lxml.......................... 2.3.0                                Ok
assimulo...................... n/a                                  Ok
wxPython...................... 2.8.12.1                             Ok
pyreadline.................... 1.7                                  Ok
setuptools.................... 0.6c11                               Ok
```

5. Redefining the JModelica.org environment

When importing `pyjmi` or `pymodelica` in Python the script `startup.py` is run which sets the environment used by JModelica.org for the current Python session. For example, the environment variable `JMODELICA_HOME` points at the JModelica.org installation directory and `IPOPT_HOME` points at the Ipopt installation directory. One or more of these environment variables set in `startup.py` can be overridden by a user defined script: `user_startup.py`.

The script `startup.py` looks for `user_startup.py` in the folder

- `$USERPROFILE/.jmodelica.org/` (Windows)
- `$HOME/.jmodelica.org/` (unix)

If the script `user_startup.py` is not found, the default environment variables will be used.

5.1. Example redefining IPOPT_HOME

The following step-by-step procedure will redefine the JModelica.org environment variable `IPOPT_HOME`:

1. Create a text file and name it `user_startup.py`.
2. Place the file in `$USERPROFILE/.jmodelica.org/` (Windows) or `$HOME/.jmodelica.org/` (unix). To find out what `$USERPROFILE` or `$HOME` points to, open a Python shell and type:

   ```
   import os
   os.environ['USERPROFILE']   // windows
   os.environ['HOME']          // unix
   ```

3. Open the file and type

   ```
   environ['IPOPT_HOME']='<new path to Ipopt home>'
   ```

4. Save and close.
5. Check your changes by opening a Python shell, import `pyjmi` and check the `IPOPT_HOME` environment variable:

   ```
   import pyjmi
   pyjmi.environ['IPOPT_HOME']
   ```

6. The JModelica.org user forum

Please use the JModelica.org forum for any questions related to JModelica.org. You can search in old threads to see if someone has asked your question before or start a new thread if you are a member.

Chapter 4. Working with Models

1. Introduction to models

Modelica and Optimica models can be compiled and loaded in the JModelica.org Python interface as model objects. These model objects can then be used for simulation and optimization purposes. This chapter will cover how to compile Modelica and Optimica models, set compiler parameters and 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.

1.1. The different model objects in JModelica.org

There are three different kinds of model objects that can be created with JModelica.org: `JMUModel`, `FMUModel` and `CasadiModel`. The `JMUModel` is created by loading a *JMU*, which is a compressed file with a JModelica.org specific structure. The `FMUModel` is created by loading an *FMU* (Functional Mock-up Unit), which is a compressed file following the FMI (Functional Mock-up Interface) standard. The `CasadiModel` is created by loading an *FMUX* which is a compressed file containing the XML file needed to work with CasADI.

JMUs are created by compiling Modelica or Optimica models with JModelica.org. The JMU can then be loaded in a `JMUModel` object and used for simulation and optimization purposes.

FMUs are created by compiling Modelica models with JModelica.org (or any other tool supporting FMU export). Optimica models can not be compiled into FMUs. The FMU can then be loaded in an `FMUModel` object and used for simulation purposes.

FMUXes are created by compiling Modelica or Optimica models with JModelica.org. The FMUX main content is an XML file which follows the FMI standard with some additional JModelica.org specific elements and an equation section in which all equations used in the model are represented. The FMUX can be loaded in a `CasadiModel` and thereby use the automatic differentiation tool CasADi for optimization purposes. Read more about CasADi and how a `CasadiModel` object can be used for optimization in Section 8 in Chapter 8.

2. Compilation

Compiling a model is done with just a few steps; importing a compiler function from the JModelica.org Python package `pymodelica`, specifying a model class and file location and performing the actual compilation. This will be demonstrated in Section 2.1 for the JMU, in Section 2.2 for the FMU and in Section 2.3 for the FMUX.

For more advanced usage of the compiler functions, there are compiler options and parameters which can be modified. These will be explained in Section 2.4.

Section 2.5, will go through some parts of the compilation process and how to perform these steps one by one.

2.1. Simple JMU compilation example

The following steps compiles a model to a JMU in the JModelica.org Python interface:

1. Import the JModelica.org compiler function `compile_jmu` 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_jmu

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

# Compile the model, return argument is the file name of the JMU
compile_jmu(model_name, mo_file)
>> '.\\myPackage_myModel.jmu'
```

Once compilation has completed successfully a JMU file will have been created on the file system. The JMU file is essentially a compressed file archive containing files created during compilation that are needed when instantiating a model object. Return argument for `compile_jmu` is the full file path of the JMU that has just been created, this will be useful later when we want to create model objects. More about the JMU file and loading models can be found in Section 3.

In the above example, compilation has been performed with default parameters and options. The only parameters specified are the model class name and file. `compile_jmu` has several other parameters which can be modified. The different parameters, their default values and interpretation will be explained in Section 2.4.

2.2. Simple FMU compilation example

The steps required to compile a model to an FMU is very similar to compiling a model to a JMU:

1. Import the JModelica.org compiler function `compile_fmu` from the package `pymodelica`.
2. Specify the model and model file.
3. Perform the compilation.

The only difference is really the requirement on the model, it must be a pure Modelica model. The following code example demonstrates how to compile an FMU.

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

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

# Compile the model, return argument is the file name of the FMU
compile_fmu(model_name, mo_file)
>> '.\\myPackage_myModel.fmu'
```

As in the JMU case, when the compilation has completed successfully an FMU file will have been created on the file system. The return argument is also the full file path of the FMU that has just been created, which will be useful later when we want to create model objects. More about the FMU file and loading models can be found in Section 3.

2.3. Simple FMUX compilation example

Compiling an FMUX follows the same principle as compiling a JMU and an FMU.

1. Import the JModelica.org compiler function `compile_fmux` from the package `pymodelica`.
2. Specify the model and model file.
3. Perform the compilation.

The Python code example below will perform these steps.

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

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

# Compile the model, return argument is the file name of the FMUX
compile_fmux(model_name, mo_file)
>> '.\\myPackage_myModel.fmux'
```

2.4. Compiler settings

The compiler function parameters can be listed with the interactive help in Python. The parameters are explained in the corresponding Python *docstring* which is visualized with the interactive help. This is demonstrated in the code examples below.

The parameter `target`, is further explained in Section 2.4.5, Section 2.4.6 and Section 2.4.7.

2.4.1. compile_jmu parameters

The `compile_jmu` parameters can be listed with the interactive help.

```
# Display the docstring for compile_jmu with the Python command 'help'
from pymodelica import compile_jmu
help(compile_jmu)
```

```
Help on function compile_jmu in module pymodelica.compiler:

compile_jmu(class_name, file_name=[], compiler='auto', target='ipopt', compiler_options={},
            compile_to='.', compiler_log_level='warning')
    Compile a Modelica or Optimica model to a JMU.

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

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

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

    Library directories can be added to MODELICAPATH by listing them in a
    special compiler option 'extra_lib_dirs', for example:

        compiler_options =
            {'extra_lib_dirs':['c:\MyLibs\MyLib1','c:\MyLibs\MyLib2']}

    Other options for the compiler should also be listed in the compiler_options
    dict.

    The compiler target is 'ipopt' by default which means that libraries for AD
    and optimization/initialization algortihms will be available as well as the
    JMI. The other targets are:

        'model' --
            AD and JMI is included.
        'algorithm' --
            AD and algorithm but no Ipopt linking.
        'model_noad' --
            Only JMI, that is no AD interface. (Must currently be used when
            model includes external functions.)

    Parameters::

        class_name --
            The name of the model class.

        file_name --
            Model file (string) or files (list of strings), can be both .mo or
            .mop files.
            Default: Empty list.
```

```
        compiler --
            'auto' if a compiler should be selected automatically depending on
            file ending, 'modelica' if a ModelicaCompiler should be used or
            'optimica' if a OptimicaCompiler should be used.
            Default: 'auto' (i.e. depends on argument file_name)

        target --
            Compiler target. 'model', 'algorithm', 'ipopt' or 'model_noad'.
            Default: 'ipopt'

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

        compile_to --
            Specify location of the compiled JMU. Directory will be created if
            it does not exist.
            Default: Current directory.

        compiler_log_level --
            Set the log level for the compiler. Valid options are 'warning'/'w',
            'error'/'e' or 'info'/'i'.
            Default: 'warning'

    Returns::

        Name of the JMU which has been created.
```

2.4.2. compile_fmu parameters

The `compile_fmu` parameters 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='modelica', target='model_fmume',
            compiler_options={}, compile_to='.', compiler_log_level='warning')
    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 file as a string or a list of file_names
```

```
      (strings).

Library directories can be added to MODELICAPATH by listing them in a
special compiler option 'extra_lib_dirs', for example:

    compiler_options =
        {'extra_lib_dirs':['c:\MyLibs\MyLib1','c:\MyLibs\MyLib2']}

Other options for the compiler should also be listed in the compiler_options
dict.

The compiler target is 'model_fmume' by default which means that the shared
file contains the FMI for Model Exchange API. This is currently the only
target that is possible to use.

Parameters::

    class_name --
        The name of the model class.

    file_name --
        Model file (string) or files (list of strings), can be both .mo or
        .mop files.
        Default: Empty list.

    compiler --
        The compiler used to compile the model. The only possible compiler
        that can be used currently is ModelicaCompiler.
        Default: 'modelica'

    target --
        Compiler target.
        Default: 'model_fmume'

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

    compile_to --
        Specify location of the compiled FMU. Directory will be created if
        it does not exist.
        Default: Current directory.

    compiler_log_level --
        Set the log level for the compiler. Valid options are 'warning'/'w',
        'error'/'e' or 'info'/'i'.
        Default: 'warning'

Returns::

    Name of the FMU which has been created.
```

2.4.3. compile_fmux parameters

The `compile_fmux` parameters can be listed with the interactive help.

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

compile_fmux(class_name, file_name=[], compiler='auto', compiler_options={}, compile_to='.',
             compiler_log_level='warning')
    Compile a Modelica model to an FMUX.

    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 file as a string or a list of file_names
          (strings).

    Library directories can be added to MODELICAPATH by listing them in a
    special compiler option 'extra_lib_dirs', for example:

        compiler_options =
            {'extra_lib_dirs':['c:\MyLibs\MyLib1','c:\MyLibs\MyLib2']}

    Other options for the compiler should also be listed in the compiler_options
    dict.


    Parameters::

        class_name --
            The name of the model class.

        file_name --
            Model file (string) or files (list of strings), can be both .mo or
            .mop files.
            Default: Empty list.

        compiler --
            The compiler used to compile the model.
            Default: 'auto'

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

```
        compile_to --
            Specify location of the compiled FMUX. Directory will be created if
            it does not exist.
            Default: Current directory.

        compiler_log_level --
            Set the log level for the compiler. Valid options are 'warning'/'w',
            'error'/'e' or 'info'/'i'.
            Default: 'warning'

    Returns::

        Name of the FMUX which has been created.
```

2.4.4. Compiler options

Compiler options are read from an XML file, `options.xml`, which can be found in the JModelica.org installation folder under the folder `Options`. The options are loaded from the file when a compiler is created, that is when `compile_jmu`, `compile_fmu` or `compile_fmux` is run. Options for a compiler instance can be modified interactively when compiling using the parameter `compiler_options`. This is shown in an example compiling an Optimica model below.

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

# Specify Optimica model and model file
model_name = 'myPackage.myModel'
mo_file = 'myPackage.mop'

compile_jmu(model_name, mo_file, compiler_options={"enable_variable_scaling":True})
```

There are four type categories: string, real, integer and boolean. The available options, default values and description are listed in Table 4.1.

Table 4.1 Compiler options

Option	Default	Description
`normalize_minimum_time_problems`	`true`	When this option is set to `true` then minimum time optimal control problems encoded in Optimica are converted to fixed interval problems by scaling of the derivative variables. (Boolean option.)
`enable_variable_scaling`	`false`	If this option is `true`, then the "nominal" attribute will be used to scale variables in the model. (Boolean option.)
`halt_on_warning`	`false`	If this option is set to `false` one or more compiler warnings will not stop compilation of the model. (Boolean option.)

Option	Default	Description
automatic_add_initial_equations	true	When this option is set to true, then additional initial equations are added to the model based on a the result of a matching algorithm. Initial equations are added for states that are not matched to an equation. (Boolean option.)
generate_ode_jacobian	false	If this option is set to true, code for computing ODE Jacobians are generated. (Boolean option.)
eliminate_alias_variables	true	If this option is true, then alias variables are eliminated from the model. (Boolean option.)
extra_lib_dirs	""	The value of this option is appended to the value of the MODELICAPATH environment variable for determining in what directories to search for libraries. (String option.)
generate_xml_equations	false	If this option is true, then model equations are generated in XML format. (Boolean option.)
state_start_values_fixed	false	This option enables the user to specify if initial equations should be 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. (Boolean option.)
equation_sorting	false	If this option is true, equations are sorted using the BLT algorithm. (Boolean option.)
index_reduction	true	If this option is true, index reduction is performed. (Boolean option.)
enable_structural_diagnosis	true	Enable this option to invoke the structural error diagnosis based on the matching algorithm. (Boolean option.)
compliance_as_warning	false	When this option is set to true, then compliance errors are treated as warnings instead. This can lead to the compiler or solver crashing. Use with caution! (Boolean option.)
generate_html_diagnostics	false	When this option is set to true model diagnostics is generated in HTML format. This includes the flattened model, connection sets, alias sets and BLT form. (Boolean option.)

Option	Default	Description
generate_ode	false	If this option is set to true, code for solving ODEs is generated. (Boolean option.)
convert_free_dependent_ parameters_to_algebraics	true	If this option is true, free dependent parameters are converted to algebraic variables. (Boolean option.)
generate_dae	true	If this option is true, code for solving DAEs is generated. (Boolean option.)
generate_dae_jacobian	false	If this option is true, code for computing DAE Jacobians is generated. (Boolean option.)
export_functions	false	Export used Modelica functions to generated C code in a manner that is compatible with the external C interface in the Modelica Language Specification. (Boolean option.)
export_functions_vba	false	If this option is true, then VBA-compatible wrappers for exported functions are created. Requires export_functions = true. (Boolean option.)
generate_fmi_cs_xml	false	If this option is true the model description part of the XML variables file will be FMI for co simulation compliant. To generate an XML which will validate with FMI schema the option generate_xml_equations must also be false.
advanced_tearing	false	If this option is set to true, a more advanced tearing algorithm is used. This algorithm may give smaller torn systems, but may also take longer time to execute. This option only has an effect if the option enable_tearing is set to true.
divide_by_vars_in_tearing	false	If this option is set to true, 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.
enable_tearing	false	If this option is set to true, tearing of equation systems is enabled.
generate_fmi_me_xml	false	If this option is true the model description part of the XML variables file will be FMI for model exchange compliant. To generate an XML which will validate with FMI schema the option generate_xml_equations must also be false.

Option	Default	Description
`inline_functions`	`false`	Perform function inlining on model after flattening.
`fmi_version`	`1.0`	FMI version to use in compilation.

2.4.5. Compiler targets for `compile_jmu`

There are four compiler targets available for `compile_jmu`:

- `ipopt`: Compiled model will include JMI interface, AD and linking to Ipopt libraries. There is support for optimization and initialization algorithm.
- `model`: Compiled model will include support for JMI interface and AD.
- `algorithm`: Compiled model will include support for JMI interface. AD and algorithm but not link with the Ipopt libraries.
- `model_noad`: Compiled model will only include the JMI interface.

The compiler target is '`ipopt`' by default which will work for most cases. However, if JModelica.org has been built without Ipopt libraries the target would have to be changed to any other suitable target that does not include the Ipopt libraries.

The target `model_noad` must be used if the model contains external equations since external equations do not work with the AD interface at the moment.

2.4.6. Compiler targets for `compile_fmu`

Currently, the only possible target available for `compile_fmu` is '`model_fmume`'. With this target the compiled model will include the FMI for Model Exchange API.

2.4.7. Compiler targets for `compile_fmux`

Compiler targets are not relevant for `compile_fmux` since no binary file is compiled.

2.5. Compilation in more detail

Compiling with `compile_jmu`, `compile_fmu` or `compile_fmux` bundles quite a few steps required for the compilation from model file to JMU, FMU or FMUX. Some of these steps will be described briefly here, for a more detailed review on the compilation steps see Section 4 in Chapter 1.

2.5.1. Create a compiler

A compiler, can be either a Modelica or Optimica compiler, is created by importing the Python classes from the compiler module. The compiler constructors only has optional arguments. These are the paths to the template XML

and C files which in most cases can be left to the default values. (Note that in the `compile_fmux` case the only template needed is to the Model description XML template file since no C code is produced.) This example code will create a Modelica compiler.

```
# import the class ModelicaCompiler from the compiler module
from pymodelica.compiler import ModelicaCompiler

# create a compiler instance
mc = ModelicaCompiler()
```

2.5.2. Source tree generation and flattening

In the first step of the compilation, the model is parsed and instantiated. Then the model is transformed into a flat representation which can be used to generate C and XML code. If there are errors in the model, for example syntax or type errors, Python exceptions will be thrown during these steps.

```
# Parse the model and get a reference to the source root
source_root = mc.parse_model('myPackage.mo')

# Generate an instance tree representation and get a reference to the model instance
model_instance = mc.instantiate_model(source_root, 'myPackage.myModel')

# Perform flattening and get a flat representation
flat_rep = mc.flatten_model(model_instance)
```

2.5.3. Code generation

The next step is the code generation which produces C code containing the model equations and a couple of XML files containing model meta data such as variable names and types and parameter values. Note that in the `compile_fmux` case, only XML code is generated in this step.

```
# Generate code
mc.generate_code(flat_rep)
```

3. Loading models

Compiled models, JMUs, FMUs and FMUXes, are loaded in the JModelica.org Python interface with the `JMUModel` class from the `pyjmi` module, the `FMUModel` class from the `pyfmi` module and the `CasadiModel` class from the `pyjmi` module respectively. This will be demonstrated in Section 3.4, Section 3.5 and Section 3.6.

The model classes contain many methods with which models can be manipulated after instantiation. Amongst the most important methods are `initialize`, `simulate` and `optimize`. (`optimize` is only relevant for JMU model instances.) These are explained in Chapter 7 and Chapter 8. For more information on how to use the CasadiModel for optimization purposes, see Section 8 in Chapter 8. The more basic methods for variable and parameter manipulation are explained in Section 4.

3.1. The JMU

The JMU is a compressed file which contains all files needed to load and work with the compiled model in JModelica.org. The JMU contains the shared object file, the XML files with model variable and parameter data and some other files created during compilation of the model. The JMU file format is a JModelica.org specific format but is designed to follow the FMU file format from the FMI standard as much as possible. A JMU is created when compiling a Modelica or Optimica model with `pymodelica.compile_jmu`, see Section 2.

3.2. The FMU

The FMU (Functional Mock-up Unit) is a compressed file which follows the FMI (Functional Mock-up Interface) standard. An FMU is created when compiling a Modelica model with `pymodelica.compile_fmu`.

Read more about the FMI/FMU standard in Chapter 5.

3.3. The FMUX

The FMUX is a compressed file which only contains an XML file and a couple of resource files created during the flattening of the model. The XML file follows the FMI standard with some JModelica.org specific extra elements and a section of equations. The equations part contains all equations used in the model. The FMUX is created when compiling a Modelica or Optimica model with `pymodelica.compile_fmux`.

3.4. Loading a JMU

A JMU file is loaded in JModelica.org with the class `JMUModel` in the `pyjmi` module. The following simple example demonstrates how to do this in a Python shell or script.

```
# Import JMUModel from pyjmi
from pyjmi import JMUModel
myModel = JMUModel('myPackage_myModel.jmu')
```

The only parameter in the `JMUModel` constructor is the name of the JMU file, including any file path. When compiling and loading it is therefore practical to use the return argument from `compile_jmu`, which is the path to the JMU created. The following example demonstrates this.

```
# Import compile_jmu and JMUModel
from pymodelica import compile_jmu
from pyjmi import JMUModel

# Compile and load model
jmu_name = compile_jmu('myPackage.myModel','myPackage.mo')
myModel = JMUModel(jmu_name)
```

3.5. Loading an FMU

The FMU file can be loaded in JModelica.org with the class `FMUModel` in the `pyfmi` module. The following short example demonstrates how to do this in a Python shell or script.

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

The `FMUModel` instance can then be used to set parameters and used for simulations.

3.6. Loading an FMUX

The FMUX file can be loaded in JModelica.org with the class `CasadiModel` in the `pyjmi` module. The following short example demonstrates how to do this in a Python shell or script.

```
# Import CasadiModel from pyjmi
from pyjmi import CasadiModel
myModel = CasadiModel('myFMUX.fmux')
```

4. Variable and parameter manipulation

Model variables and parameters can be manipulated with methods in the model classes once the model has been loaded. Some short examples in Section 4.2 will demonstrate this. Model variable meta data and parameter values are saved in XML files which are generated during compilation, these are briefly explained in Section 4.1. The XML file containing the parameters can be used to save different sets of parameters for one model, see Section 4.3.

4.1. Model variable XML files

The model variable meta data and parameter values are saved in XML files which are generated during the compilation. They follow the name convention:

- `modelDescription.xml`
- `<model_class_name>_values.xml`

The variable meta data is saved in `modelDescription.xml` and the parameter values in `<model_class_name>_values.xml`. Note that the parameter values file is JModelica.org specific and is not a part of the FMI standard and it is not part of the FMUX either. The name of the parameter is used to map a parameter value in the XML values file to a parameter specification in the XML variables file.

4.2. Setting and getting variables

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

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

```
# Compile and load the model
from pymodelica import compile_jmu
from pyjmi import JMUModel
jmu_name = compile_jmu('RLC_Circuit','RLC_Circuit.mo')
rlc_circuit = JMUModel(jmu_name)

# Get the value of the variable 'resistor.R'
resistor_r = rlc_circuit.get('resistor.R')
resistor_r
>> 1.0

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

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

```
# Create a list of variables and values
vars = ['resistor.R', 'resistor.v', 'capacitor.C', 'capacitor.v']
values = rlc_circuit.get(vars)
values
>> [2.0, 0.0, 1.0, 0.0]

# Change some of the values
values[0] = 3.0
values[3] = 1.0
rlc_circuit.set(vars, values)
rlc_circuit.get(vars)
>> [3.0, 0.0, 1.0, 1.0]
```

4.3. Loading and saving parameters

This section is only relevant for the `JMUModel`.

4.3.1. Loading XML values file

It is possible to (re)load the parameter values from an XML file as is done automatically when the `pyjmi.JMUModel` object was first created. If, for example, there were many local changes to parameters it could be desirable to reset everything as it was from the beginning. The following example shows how reloading the parameter values from the XML file resets the parameters in the model. The model is taken from the `pyjmi.examples` package and is assumed to be compiled and loaded.

```
# Look at parameters 'resistor.R' and 'sine.offset'
```

```
rlc_circuit.get('resistor.R)
>> 1.0
rlc_circuit.get('sine.offset')
>> 0.0

# Change them
rlc_circuit.set('resistor.R', 2.0)
rlc_circuit.set('sine.offset', 0.5)

# Look at them again
rlc_circuit.get('resistor.R)
>> 2.0
rlc_circuit.get('sine.offset')
>> 0.5

# Reset them by loading the original XML values file
rlc_circuit.load_parameters_from_XML()

# 'resistor.R' and 'sine.offset' have now been reset
rlc_circuit.get('resistor.R)
>> 1.0
rlc_circuit.get('sine.offset')
>> 0.0
```

The default behaviour is to load the same file as was created during compilation. If another file should be used this must be passed as an argument to the method.

```
# Load other XML file
rlc_circuit.load_parameters_from_XML('new_values.xml')
```

4.3.2. Writing to XML values file

Setting a parameter value with `JMUModel.set` only changes the value in the vector loaded when `pyjmi.JMUModel` was created, which means that it will not be saved when the model is discarded. To save all local changes made to the model parameters, the values have to be written to the XML values file.

```
# Set a parameter
rlc_circuit.set('inductor.L', 1.5)

# Save parameters to the XML values file
rlc_circuit.write_parameters_to_XML()

# Load the XML values file once again and see that the changed parameter was saved in
# the XML file
rlc_circuit.load_parameters_from_XML()
rlc_circuit.get('inductor.L')
>> 1.5
```

If `write_parameters_to_XML()` is called without arguments the values will be written to the XML values file in the JMU which was created when the model was compiled (following the name conventions mentioned above). It

is also possible to save the changes in a new XML file. This is quite convenient since different parameter value settings can then easily be saved and reloaded in the model.

```
# Save to specific XML file
rlc_circuit.write_parameters_to_XML('test_values.xml')
```

5. Debugging models

The JModelica.org compilers can generate debugging information in order to facilitate localization of errors. There are two mechanisms for generating such diagnostics: dumping of debug information to the system output or generation of HTML code that can be viewed with a standard web browser.

By setting the argument `compiler_log_level` to `'i'`, diagnostics is printed to the standard output, normally the terminal window from which the compiler is invoked. The resulting diagnostics provides detailed information from each of the transformation steps in the symbolic manipulation algorithms.

By setting the compiler option `generate_html_diagnostics` to true, a number of HTML pages containing similar diagnostics is 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 similar to the one shown below.

```
Modelica.Mechanics.Rotational.Examples.First

Number of independent constants:              1
  Number of Real independent constants:       1
  Number of Integer independent constants:    0
  Number of Enum independent constants:       0
  Number of Boolean independent constants:    0
  Number of String independent constants:     0
Number of dependent constants:                0
  Number of Real dependent constants:         0
  Number of Integer dependent constants:      0
  Number of Enum dependent constants:         0
  Number of Boolean dependent constants:      0
  Number of String dependent constants:       0
Number of independent parameters:             20
  Number of Real independent parameters:      14
  Number of Integer independent parameters:   0
  Number of Enum independent parameters:      4
  Number of Boolean independent parameters:   2
  Number of String independent parameters:    0
Number of dependent parameters:               6
  Number of Real dependent parameters:        6
  Number of Integer dependent parameters:     0
  Number of Enum dependent parameters:        0
  Number of Boolean dependent parameters:     0
```

```
  Number of String dependent parameters:      0
Number of variables :                         33
  Number of Real variables:                   33
  Number of Integer variables:                0
  Number of Enum variables:                   0
  Number of Boolean variables:                0
  Number of String variables:                 0
Number of Real differentiated variables:      4
Number of Real derivative variables:          4
Number of Real algebraic variables:           25
Number of inputs:                             0
  Number of Real inputs:                      0
  Number of Integer inputs:                   0
  Number of Enum inputs:                      0
  Number of Boolean inputs:                   0
  Number of String inputs:                    0
Number of discrete variables :                0
  Number of Real discrete variables:          0
  Number of Integer discrete variables:       0
  Number of Enum discrete variables:          0
  Number of Boolean discrete variables:       0
  Number of String discrete variables:        0
Number of equations:                          29
Number of variables with binding expression:  0
  Number of Real variables with binding exp:  0
  Number of Integer variables binding exp:    0
  Number of Enum variables binding exp:       0
  Number of Boolean variables binding exp:    0
  Number of String variables binding exp:     0
Total number of equations:                    29
Number of initial equations:                  4
Number of relational exps in equations:       1
Number of relational exps in init equations:  0
Flattened model

Transformed model

Problems:
0 errors, 0 compliance errors, 0 warnings

Alias sets

Connection sets

BLT diagnostics
Number of unsolved equation blocks in DAE initialization system: 1: {7}
Number of unsolved equation blocks in DAE system: 1: {7}
```

Note that some of the entries, including `Flattened model`, `Transformed model`, `Connection sets`, `Alias sets` and `BLT diagnostics` are links to sub pages containing additional information. For example, the BLT diagnostics page contains information about individual systems of equations:

```
...

--- Block 12---
Solved block of 1 variables:
Computed variable:
  torque.flange.tau
Solution:
  ( sine.offset + (if time < sine.startTime then 0 else ( sine.amplitude ) *
  ( sin(( ( ( 2 ) * ( 3.141592653589793 ) ) * ( sine.freqHz ) ) *
  ( time - ( sine.startTime ) ) + sine.phase) )) ) / (  - ( 1 ) )

--- Block 13---
Non-solved block of 7 variables:
Unknown variables:
  idealGear.der(phi_a,2)
  idealGear.der(phi_b,2)
  inertia2.der(_der_phi)
  inertia2.a
  idealGear.flange_b.tau
  idealGear.flange_a.tau
  inertia1.a
Equations:
  der_2_idealGear_phi_a = inertia1.a - ( 0 )
  der_2_idealGear_phi_a = ( idealGear.ratio ) * ( der_2_idealGear_phi_b ) + ( 0 ) *
                          ( der_idealGear_phi_b ) + ( 0 ) * ( der_idealGear_phi_b ) +
                          ( 0 ) * ( idealGear.phi_b )
  der_2_idealGear_phi_b = inertia2.der(_der_phi) - ( 0 )
  inertia2.a = inertia2.der(_der_phi)
  ( inertia2.J ) * ( inertia2.a ) =  - ( idealGear.flange_b.tau ) + inertia2.flange_b.tau
  0 = ( idealGear.ratio ) * ( idealGear.flange_a.tau ) + idealGear.flange_b.tau
  ( inertia1.J ) * ( inertia1.a ) =  - ( torque.flange.tau ) - ( idealGear.flange_a.tau )

...
```

Chapter 5. FMI Interface

FMI (Functional Mock-up Interface) is a standard for exchanging models between different modeling and simulation environments. FMI defines a model execution interface consisting of a set of C-function signatures for handling the communication between the model and a simulation environment. Models are presented as ODEs with time, state and step events. FMI also specifies that all information related to a model, except the equations, should be stored in an XML formated text-file. The format is specified in the standard and specifically contains information about the variables, names, identifiers, types and start attributes.

A model is distributed in a zip-file with the extension '.fmu', containing several files. These zip-files containing the models are called FMUs (Functional Mock-up Units). The important files in an FMU are mainly the XML-file, which contains the definitions of all variables and then files containing the C-functions which can be provided in source and/or binary form. FMI standard also supports providing documentation and resources together with the FMU. For more information regarding the FMI standard, please visit http://www.functional-mockup-interface.org/.

1. Overview of JModelica.org FMI Python package

The JModelica.org interface to FMI is written in Python and is intended to be a close copy of the defined C-interface for an FMU and provides classes and functions for interacting with FMUs.

The JModelica.org platform offers a Pythonic and convenient interface for FMUs which can be used to connect other simulation software. JModelica.org also offers a connection to Assimulo, the default simulation package included in JModelica.org so that FMUs can easily be simulated.

The interface is located in `pyfmi.fmi` and consist of the class `FMUModel` together with methods for unzipping the FMU and for writing the simulation results. Connected to this interface is a wrapper for JModelica.org's simulation package to enable an easy simulation of the FMUs. The simulation wrapper is located in `pyfmi.simulation.assimulo`, `FMIODE`.

In the table below is a list of the FMI C-interface and its counterpart in the JModelica.org Python package. We have adapted the name convention of lowercase letters and underscores separating words. For methods with no calculations, as for example `fmi(Get/Set)ContinuousStates` they are instead of different methods, connected with a property. In the table, a lack of parenthesis indicates that the method is instead a property.

Table 5.1 Conversion table.

FMI C-Interface	JModelica.org FMI Python Interface
const char* fmiGetModelTypesPlatform()	string FMUModel.model_types_platform
const char* fmiGetVersion()	string FMUModel.version
fmiComponent fmiInstantiateModel(...)	FMUModel.__init__()
void fmiFreeModelInstance(fmiComponent c)	FMUModel.__del__()

FMI C-Interface	JModelica.org FMI Python Interface
fmiStatus fmiSetDebugLogging(...)	none FMUModel.set_debug_logging(flag)
fmiStatus fmiSetTime(...)	FMUModel.time
fmiStatus fmi(Get/Set)ContinuousStates(...)	FMUModel.continuous_states
fmiStatus fmiCompletedIntegratorStep(...)	boolean FMUModel.completed_integrator_step()
fmiStatus fmiSetReal/Integer/Boolean/String(...)	none FMUModel.set_real/integer/boolean/ string(valueref,values)
fmiStatus fmiInitialize(...)	none FMUModel.initialize() (also sets the start attributes)
struct fmiEventInfo	FMUModel.get_event_info()
fmiStatus fmiGetDerivatives(...)	numpy.array FMUModel.get_derivatives()
fmiStatus fmiGetEventIndicators(...)	numpy.array FMUModel.get_event_indicators()
fmiStatus fmiGetReal/Integer/Boolean/String(...)	numpy.array FMUModel.get_real/integer/boolean/ string(valueref)
fmiStatus fmiEventUpdate(...)	none FMUModel.event_update()
fmiStatus fmiGetNominalContinuousStates(...)	FMUModel.nominal_continuous_states
fmiStatus fmiGetStateValueReferences(...)	numpy.array FMUModel.get_state_value_references()
fmiStatus fmiTerminate(...)	FMUModel.__del__()

If logging is set to `True` the log can be retrieved with the method,

```
FMUModel.get_log()
```

Documentation of the functions can also be accessed interactively from IPython by using for instance,

```
FMUModel.get_real?
```

There is also a one-to-one map to the C-functions, meaning that there is an option to use the low-level C-functions as they are specified in the standard instead of using our wrapping of the functions. These functions are also located in `FMUModel` and is named with a leading underscore together with the same name as specified in the standard.

2. Example

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

For more examples on how to simulate an FMU using JModelica.org's high-level features, see Chapter 7.

2.1. Simulation using the native FMI interface

This example shows how to use the native JModelica.org FMI interface for simulation of an FMU. 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 1.0 (http://www.functional-mockup-interface.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 JModelica.org platform.

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.

2.1.1. Implementation

Start by importing the necessary modules,

```
        import numpy as N
        import pylab as P #Used for plotting
        from pyfmi import FMUModel #The FMI Interface
```

Next, the FMU is to be loaded and initialized,

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

        Tstart = 0.5 #The start time.
        Tend   = 3.0 #The final simulation time.

        bouncing_fmu.time = Tstart #Set the start time before the initialization.
                                   #(Defaults to 0.0)

        bouncing_fmu.initialize() #Initialize the model. Also sets all the start
```

```
                                    #attributes defined in the XML file.
```

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 setting the property `time`. The model is also initialized, which must be done before the simulation is started.

Note that if the start time is not specified, `FMUModel` 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.

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
        vref  = [bouncing_fmu.get_valueref('h')] + \
                [bouncing_fmu.get_valueref('v')] #Retrieve the valureferences for the
                                                 #values 'h' and '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 have 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 is depended 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 monitor 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 have occurred, it needs to be handled, see below.

```
            #Event handling
            if step_event or time_event or state_event:

                eInfo = bouncing_fmu.get_event_info()
                eInfo.iterationConverged = False

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

                    #Retrieve solutions (if needed)
                    if eInfo.iterationConverged == False:
                        #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.stateValuesChanged:
                x = bouncing_fmu.continuous_states

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

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

If an event occurred, we enter the iteration loop 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.

```
        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 have 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.

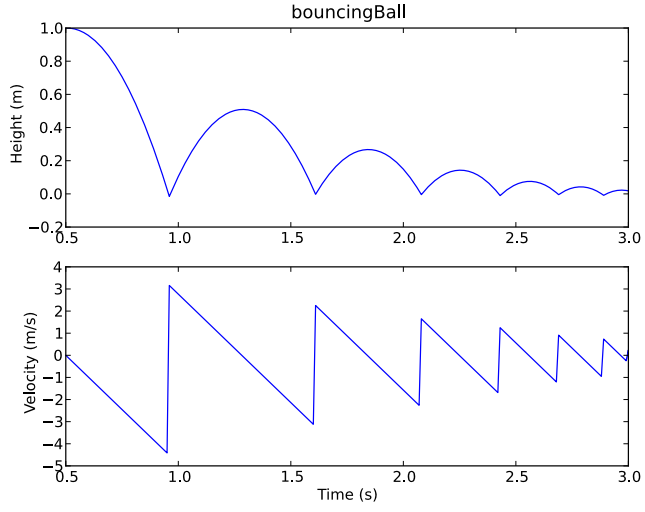


Figure 5.1 Simulation result

Chapter 6. Initialization

1. Solving DAE initialization problems

Before a model can be simulated it must be initialized, i.e. consistent initial values must be computed. To do this, JModelica.org supplies the JMUmodel member function `initialize`, which initializes the JMUmodel. The function is called after compiling and creating a JMUModel:

```
# Compile the stationary initialization model into a DLL
model_name = compile_jmu("My.Model", "/path/to/MyModel.mo")

# Load a model instance into Python
init_model = JMUModel(model_name)

# Solve the DAE initialization system
init_result = init_model.initialize()
```

The JMUModel instance `init_model` is now initialized and is ready to be simulated.

The interactive help for the initialize method is shown by the command:

```
>>> help(init_model.initialize)
    The initialization method depends on which algorithm is used, this can
    be set with the function argument 'algorithm'. Options for the algorithm
    are passed as option classes or as pure dicts. See
    JMUModel.initialize_options for more details.

    The default algorithm for this function is IpoptInitializationAlg.

    Parameters::

        algorithm --
            The algorithm which will be used for the initialization is
            specified by passing the algorithm class 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 own
            algorithms and use them with this function.
            Default: 'IpoptInitializationAlg'

        options --
            The options that should be used in the algorithm. For details on
            the options do:

                >> myModel = JMUModel(...)
                >> opts = myModel.initialize_options()
                >> opts?
```

```
            Valid values are:
                - A dict which gives IpoptInitializationAlgOptions with
                  default values on all options except the ones listed in
                  the dict. Empty dict will thus give all options with
                  default values.
                - An options object.
            Default: Empty dict

    Returns::

        Result object, subclass of algorithm_drivers.ResultBase.
```

Options for the available initialization algorithms can be set by first retrieving an options object using the `JMUModel` method `initialize_options`:

```
>>> help(init_model.initialize_options)
    Get an instance of the initialize options class, prefilled with default
    values. If called without argument then the options class for the
    default initialization algorithm will be returned.

    Parameters::

        algorithm --
            The algorithm for which the options class should be fetched.
            Possible values are: 'IpoptInitializationAlg', 'KInitSolveAlg'.
            Default: 'IpoptInitializationAlg'

    Returns::

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

Having solved the initialization problem, the result of the initialization can be retrieved from the return result object:

```
x = init_result['x']
y = init_result['y']
```

2. How JModelica.org creates the initialization system of equations

To find a set of consistent initial values a system of non-linear equations, called the system of initialization equations, is solved. This system is composed from the DAE equations, the initial equations, some resulting from start attributes with the fixed attribute set to true. Start attributes with the fixed attribute set to false are treated as initial guesses for the numerical algorithm used to solve the initialization problem

Some initialization algorithms require the system of initial equations to be well defined in the sense that the number of variables must be equal to the number of equations. If this is not the case, the

- If the number of equations is greater than the number of variables the system is overdetermined. Such a system may not have a solution, and will be treated as ill-defined. An exception is thrown in this case.

- If the number of equations is less than the number of variables the system is underdetermined and such a system has infinitely many solutions. In this case, the compiler tries to balance the system by setting some fixed attributes to `true`. So if the user supplies too few initial conditions, some variables with the attribute `fixed` set to `false` may be changed to `true` during initialization.

3. Initialization algorithms

3.1. Initialization using IPOPT

JModelica.org provides a method for DAE initialization that is based on IPOPT, the mathematical formulation of the algorithm can be found in the JMI API documentation. Notice that this algorithm does not rely on the causalization procedure (in particular the BLT transformation) which is common. Instead, the DAE residual is introduced as an equality constraint when solving an optimization problem where the squared difference between the non-fixed start values and their corresponding variables are minimized. As a consequence, the algorithm relies on decent start values for *all* variables. This approach is generally more sensitive to lacking initial guesses for start values than are algorithms based on causalization.

The algorithm provides the options summarized in Table 6.1.

Table 6.1 Options for the collocation-based optimization algorithm

Option	Default	Description
`result_file_name`	Empty string (default generated file name will be used)	Specifies the name of the file where the optimization result is written. Setting this option to an empty string results in a default file name that is based on the name of the optimization class.
`result_format`	'txt'	Specifies in which format to write the result. Currently only textual mode is supported.
`write_scaled_result`	False	Write the scaled optimization result if set to true. This option is only applicable when automatic variable scaling is enabled. Only for debugging use.

In addition to the options for the collocation algorithm, IPOPT options can also be set by modifying the dictionary `IPOPT_options` contained in the collocation algorithm options object. Here, all valid IPOPT options can be specified, see the IPOPT documentation for further information. For example, setting the option `max_iter`:

```
opts['IPOPT_options']['max_iter'] = 300
```

makes IPOPT terminate after 300 iterations even if no optimal solution has been found.

Some statistics from IPOPT can be obtained by issuing the command:

```
>>> res_init.solver.init_opt_ipopt_get_statistics()
```

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

```
>>> help(res_init.solver.init_opt_ipopt_get_statistics)
    Get statistics from the last optimization run.

    Returns::

        return_status --
            The return status from IPOPT.

        nbr_iter --
            The number of iterations.

        objective --
            The final value of the objective function.

        total_exec_time --
            The execution time.
```

3.2. Initilization using KInitSolveAlg

JModelica.org also provides a method for DAE initialization based on the non-linear equaion solver KINSOL from the SUNDIALS suite. KINSOL is currently comprised in the Assimulo package, included when installing JModelica.org. KINSOL is based on Newtons method for solving non-linear equations and is thus locally convergent. Attempts are made to make KInitSolveAlg as robust as possible but the possibility of finding a local minimum instead of the solution still remains. If the solution found by KInitSolveAlg is a local minimum a warning will be printed. The initial guesses passed to KINSOL are the ones supplied as start attributes in the current Modelica model.

KInitSolveAlg also implements an improved linear solver connected to KINSOL. This linear solver implements Tikhonov regularization to handle the problems of singular Jacobiansas well as support for SuperLU, an efficient sparse linear solver.

The options providable are summarized in Table 6.2.

Table 6.2 Options for KInitSolveAlg

Option	Default	Description
`use_constraints`	False	A flag indicating whether constraints are to be used *during* initialization. Further explained in Section 3.2.1.
`constraints`	None	A `numpy.array` containing floats that, when supplied, defines the constraints on the variables. Further explained in Section 3.2.1.
`result_format`	'txt'	Specifies in which format to write the result. Currently only textual mode is supported.

Option	Default	Description
result_file_name	Empty string (default generated file name will be used)	Specifies the name of the file where the optimization result is written. Setting this option to an empty string results in a default file name that is based on the name of the optimization class.
KINSOL_options	A dictionary with the defalt KINSOL options	These are the options sent to the KINSOL solver. These are reviewed in detail in Table 6.3.

Table 6.3 Options for KINSOL contained in the KINSOL_options dictionary

Options	Default	Descriptions
use_jac	True	Flag indicating whether or not KINSOL uses the jacobian supplied by JModelica.org (True) or if KINSOL evaluates the Jacobian through finite differences (False). Finite differences is currently not available in sparse mode.
sparse	False	Flag indicating whether the problem should be treated as sparse (True) or dense (False).
verbosity	0	An integer regulating the level of output from KINSOL. Further explained in Section 3.2.2.

3.2.1. The use of constraints

KINSOL, and hence also KInitSolvAlg, only support simple unilateral constraints, that is constraining a variable to being positive or negative. If the option use_constraints is set to True, constraints are use. What constraints that are used depends on whether or not the user has supplied constraints with the constraints option. If set, these will be used otherwise constraints will be computed by reading the min and max attributes from the Modelica file. How the constraint array is written is summarized in Table 6.4.

Table 6.4 Values allowed in the constraints array

Value	Constraint
0.0	Unconstrained.
1.0	Greater than, or equal to, zero.

Value	Constraint
2.0	Greater than zero.
-1.0	Less than, or equal to, zero.
-2.0	Less than zero.

When the constraints are read from the Modelica file the value from Table 6.4 most fitting to the min and max values is chosen. For example a variable with min set to 3.2 and max set to 5.6 is constrained to be greater than zero. When the algorithm is finished however the result will be compared with the min and max values from the model testing if the solution fulfills the constraints set by the Modelica file.

3.2.2. Verbosity of KINSOL

There are four different levels of verbosity in KINSOL with 0 being "silent" and 3 being the "loudest". What is output is reviewed in Table 6.5.

Table 6.5 Verbosity levels in KINSOL

Verbosity level	Output
0	No information displayed.
1	In each nonlinear iteration the following information is displayed: the scaled Euclidean norm of the residual at the current iterate, the scaled Euclidian norm of the Newton step as well as the number of function evaluations performed so far.
2	Level 1 output as well as the Euclidian and in finity norm of the *scaled* residual at the current iterate
3	Level 2 output plus additional values used by the global strategy as well as statistical information from the linear solver.

Chapter 7. Simulation

1. Introduction

JModelica.org supports simulation of models described in the Modelica language and models following the FMI standard see Chapter 5. The simulation environment uses Assimulo as standard which is a standalone Python package for solving ordinary differential and differential algebraic equations.

2. A first example

This example focuses on how to use JModelica.org'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 JModelica.org in the file VDP.mop.

```
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 FMUModel class
from jmodelica import compile_fmu
from pyfmi import FMUModel

# 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 = FMUModel(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 creations 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 in a dictionary-like way the simulation result:

```
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 7.1 the simulation result is shown.

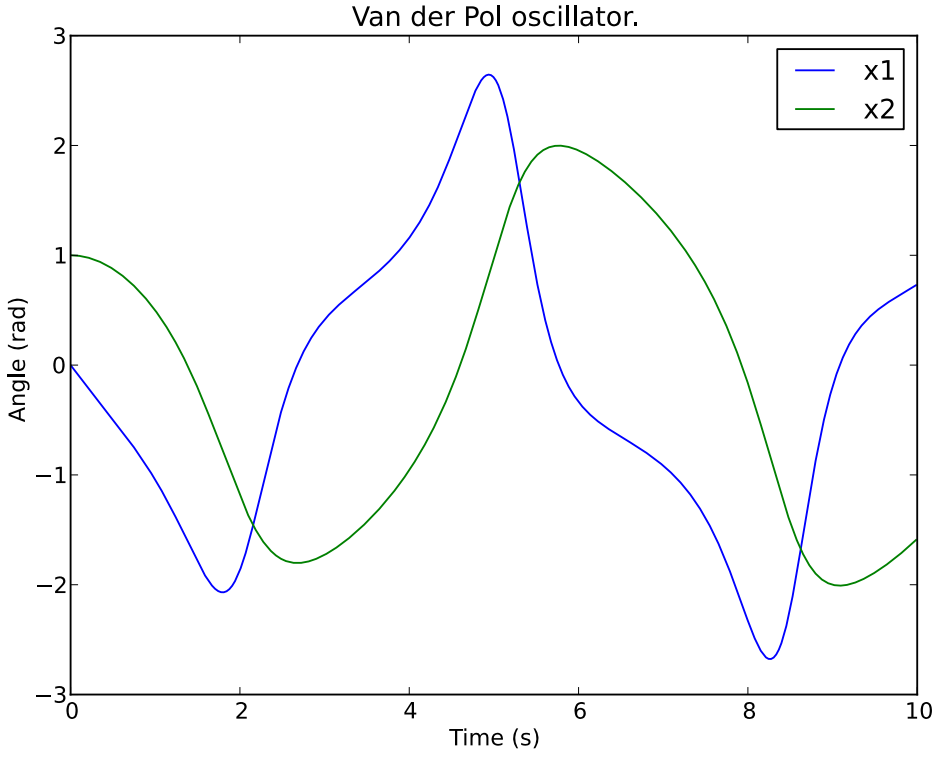


Figure 7.1 Simulation result of the Van der Pol oscillator.

3. Simulation of Models

Simulation of models in JModelica.org is preformed via the simulate method of a model object. The model objects in JModelica.org are:

- `JMUModel (located in pyjmi)`
- `FMUModel (located in pyfmi)`

The difference between the two are that `JMUModel` supports compiled models from JModelica.org (extension .jmu) while the `FMUModel` supports compiled models from other simulation/modelling tools that follow the FMI standard (extension .fmu). For more information about compiling a model in JModelica.org 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 `FMUModel` is defined as:

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

And used in the following way:

```
res = FMUModel.simulate() #Using default values
```

The only difference between the simulation method in `JMUModel` compared to `FMUModel` is that the algorithm is `AssimuloAlg`. They are although both connected to the Assimulo package and able to use its solvers.

3.1. 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 `AssimuloAlg` for `JMUModel` and `AssimuloFMIAlg` for `FMUModel`.

3.1.1. Input

The input defines the input trajectories to the model and should be a 2-tuple consisting of the name(s) of the input variables and the second argument should be either a data matrix or a function. If the argument is a data matrix it should contain a time vector as the first column and the second column should correspond to the first name in the first argument and so forth. If instead the second argument is a function it should be defined to take the time as input and return the number of inputs in the order defined by the first argument.

For example, consider that we have a model with an input variable `u1` and that the model should be driven by a sinus wave as input. Also we are interested in the interval 0 to 10.

```
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:

```
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 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)
```

3.1.2. Options for JMUModel

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

```
opts = JMUModel.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 simulation of a JMU model using the Assimulo simulation package.
   The Assimulo package contain both explicit solvers (CVode) for ODEs and
   implicit solvers (IDA) for DAEs. The ODE solvers require that the problem
   is written on the form, ydot = f(t,y).

   ...
```

In Table 7.1 the general options for the AssimuloAlg algorithm are described while in Table 7.3 a selection of the different solver arguments for the DAE solver IDA is shown. In Table 7.2 a selection of 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 7.1 General options for AssimuloAlg.

Option	Default	Description
solver	'IDA'	Specifies the simulation method that is to be used.
ncp	0	Number of communication points. If ncp is zero, the solver will return the internal steps taken.
initialize	True	If set to True, an algorithm for initializing the differential equation is invoked, otherwise the differential equation is assumed to have consistent initial conditions.
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.

Lets look at an example, consider that you want to simulate a JMU 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' #Change the solver from IDA to CVode

opts['CVode_options']['discr'] = 'Adams' #Change from using BDF to Adams

model.simulate(options=opts) #Pass in the options to simulate and simulate
```

It should also be noted from the above example the options regarding a specific solver, say the tolerances for `CVode` or `IDA`, 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

#or

opts['IDA_options']['atol']  = 1.0e-6 #Options specific for IDA
```

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 7.2 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	Inf	Maximum step-size. Positive float.
atol	1.0e-6	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-6	Relative Tolerance. Positive float.

Table 7.3 Selection of solver arguments for IDA

Option	Default	Description
maxord	5	The maximum order used. Positive integer.
maxh	Inf	Maximum step-size. Positive float.
atol	1.0e-6	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-6	Relative Tolerance. Positive float.
suppress_alg	False	Suppress the algebraic variables on the error test. Can be either False or True.
sensitivity	False	If set to True, sensitivities for the states with respect to parameters set to free in the model will be calculated.

3.1.3. Options for FMUModel

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

```
opts = FMUModel.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 7.4 the general options for the AssimuloFMIAlg algorithm are described while in Table 7.5 a selection of the different solver arguments for the ODE solver CVode is shown. Note that there are minor differences in the tolerances compared to the options described in Table 7.2. More information regarding the solver options can be found here, http://www.jmodelica.org/assimulo.

Table 7.4 General options for AssimuloFMIAlg.

Option	Default	Description
solver	'CVode'	Specifies the simulation method that is to be used.
ncp	0	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

Option	Default	Description
		results in a default file name that is based on the name of the model class.

Lets look at an example, consider that you want to simulate a 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 #Dont initialize the model

model.simulate(options=opts) #Pass in the options to simulate and simulate
```

It should also be noted from the above example 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 7.5 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	Inf	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.

Option	Default	Description
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

3.2. 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 7.6.

Table 7.6 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.

4. Examples

In the next sections, it will be shown how to use the JModelica.org platform for simulation of various JMUs and 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 text file, which also is the recommended way.

4.1. Simulation with inputs

This example will demonstrate how a model with two inputs with data from a matlab-file can be simulated. The model to be simulated is a quadruple tank connected to two pumps, which is also the inputs to the model. The model is depicted in Figure 7.2 and in the code below the corresponding Modelica code is listed.

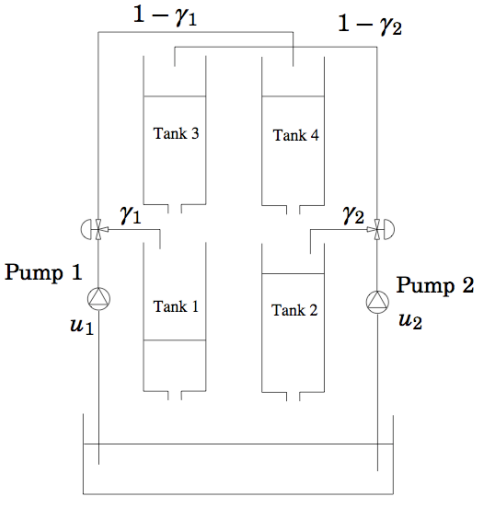


Figure 7.2 A schematic picture of the quadruple tank process.

```
model QuadTank
    // Process parameters
  parameter Modelica.SIunits.Area A1=4.9e-4, A2=4.9e-4, A3=4.9e-4, A4=4.9e-4;
  parameter Modelica.SIunits.Area a1=0.03e-4, a2=0.03e-4, a3=0.03e-4, a4=0.03e-4;
  parameter Modelica.SIunits.Acceleration g=9.81;
  parameter Real k1_nmp(unit="m3^/s/V") = 0.56e-6, k2_nmp(unit="m^3/s/V") = 0.56e-6;
  parameter Real g1_nmp=0.30, g2_nmp=0.30;

   // Initial tank levels
  parameter Modelica.SIunits.Length x1_0 = 0.06270;
  parameter Modelica.SIunits.Length x2_0 = 0.06044;
  parameter Modelica.SIunits.Length x3_0 = 0.02400;
  parameter Modelica.SIunits.Length x4_0 = 0.02300;

   // Tank levels
  Modelica.SIunits.Length x1(start=x1_0,min=0.0001/*,max=0.20*/);
  Modelica.SIunits.Length x2(start=x2_0,min=0.0001/*,max=0.20*/);
```

```
  Modelica.SIunits.Length x3(start=x3_0,min=0.0001/*,max=0.20*/);
  Modelica.SIunits.Length x4(start=x4_0,min=0.0001/*,max=0.20*/);

  // Inputs
  input Modelica.SIunits.Voltage u1;
  input Modelica.SIunits.Voltage u2;

 equation
   der(x1) = -a1/A1*sqrt(2*g*x1) + a3/A1*sqrt(2*g*x3) +
                                   g1_nmp*k1_nmp/A1*u1;
   der(x2) = -a2/A2*sqrt(2*g*x2) + a4/A2*sqrt(2*g*x4) +
                                   g2_nmp*k2_nmp/A2*u2;
   der(x3) = -a3/A3*sqrt(2*g*x3) + (1-g2_nmp)*k2_nmp/A3*u2;
   der(x4) = -a4/A4*sqrt(2*g*x4) + (1-g1_nmp)*k1_nmp/A4*u1;

end QuadTank;
```

Lets begin with the the example, copy and paste the Modelica code and save it into `QuadTank.mo` and open a python script file. We start by importing the necessary objects:

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

from jmodelica import compile_jmu
from pyjmi import JMUModel
```

The input data is stored in `qt_par_est_data.mat` which can be found in the `examples/files` catalogue in JModelica.org. Copy it into your working directory and paste the following commands to load the data-file and extract the data trajectories:

```
data = loadmat('qt_par_est_data.mat',appendmat=False)

# Extract data series
t_meas = data['t'][6000::100,0]-60
u1 = data['u1_d'][6000::100,0]
u2 = data['u2_d'][6000::100,0]
```

The trajectories have now been extracted and needs to be stacked into a data matrix with the first column as the time vector and the following columns the input of `u1` and `u2`. The names of the variables needs also be connected in the input object:

```
# Build input trajectory matrix for use in simulation
u_data = N.transpose(N.vstack((t_meas,u1,u2)))
input_object = (['u1','u2'], u_data)
```

Next, we compile and load the model:

```
# compile JMU
jmu_name = compile_jmu('QuadTank', 'QuadTank.mo')
```

```
# Load model
model = JMUModel(jmu_name)
```

Now, that the model is compiled and the input have been adapted, lets give the information to the simulate method and simulate:

```
# Simulate model with input trajectories
res = model.simulate(final_time=60, input=input_object)
```

The result is retrieved by accessing the `res` variable as a dictionary with the variable name as key:

```
x1_sim = res['x1']
x2_sim = res['x2']
x3_sim = res['x3']
x4_sim = res['x4']
u1_sim = res['u1']
u2_sim = res['u2']
t_sim  = res['time']
```

And then plotted with the help from `matplotlib`:

```
plt.figure(1)
plt.subplot(2,2,1)
plt.plot(t_sim,x3_sim)
plt.title('x3')
plt.subplot(2,2,2)
plt.plot(t_sim,x4_sim)
plt.title('x4')
plt.subplot(2,2,3)
plt.plot(t_sim,x1_sim)
plt.title('x1')
plt.xlabel('t[s]')
plt.subplot(2,2,4)
plt.plot(t_sim,x2_sim)
plt.title('x2')
plt.xlabel('t[s]')
plt.show()

plt.figure(2)
plt.subplot(2,1,1)
plt.plot(t_sim,u1_sim,'r')
plt.title('u1')
plt.subplot(2,1,2)
plt.plot(t_sim,u2_sim,'r')
plt.title('u2')
plt.xlabel('t[s]')
plt.show()
```

In Figure 7.3 the result of the tank levels are shown and in Figure 7.4 the input signals are shown.

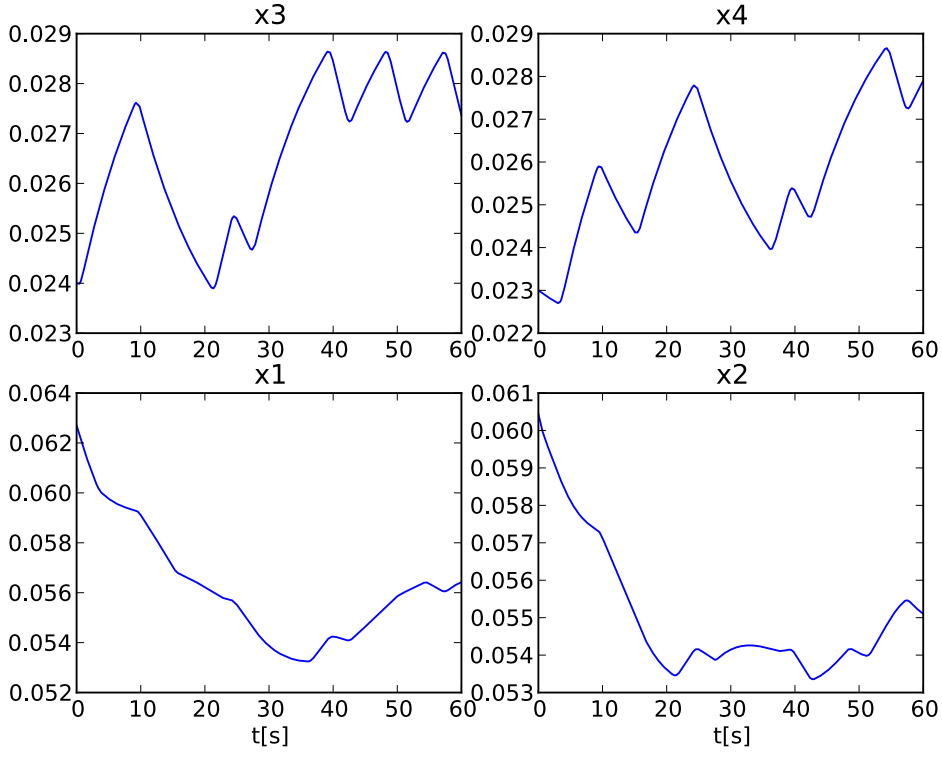


Figure 7.3 Tank levels

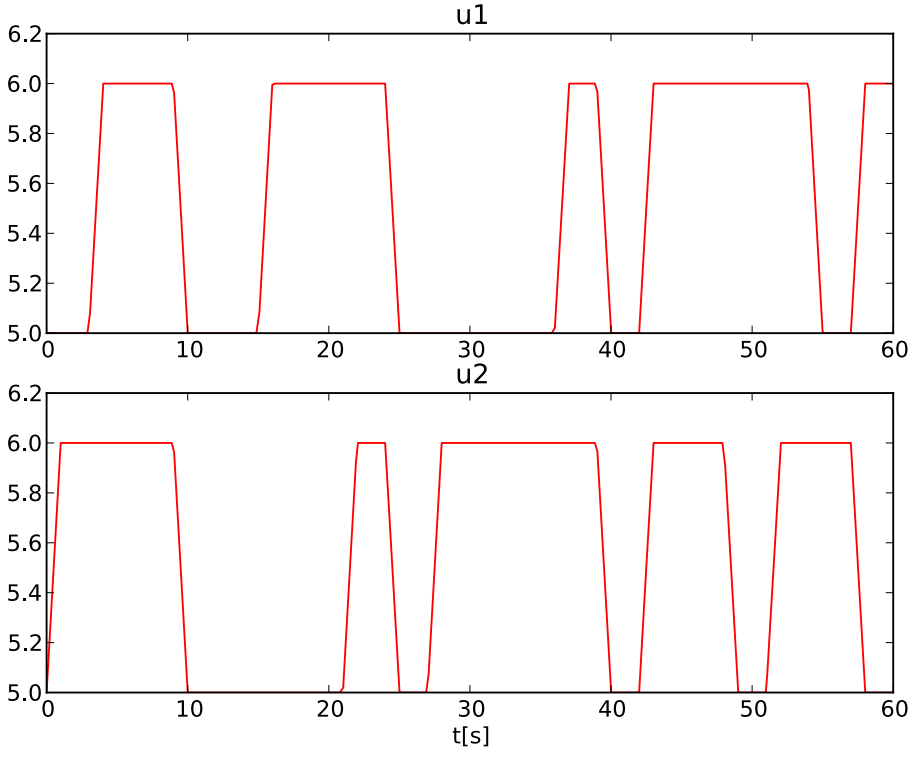


Figure 7.4 Input trajectories

4.2. Simulation of a discontinuous system

The model which is to be simulated is an electric circuit. The model is depicted in Figure 7.5 and consists of resistances, inductors and a capacitor. The circuit is connected to a voltage source which generates a square-wave with an amplitude of 1.0 and a frequency of 0.6 Hz. The model is also available from the examples in the file RLC_Circuit.mo.

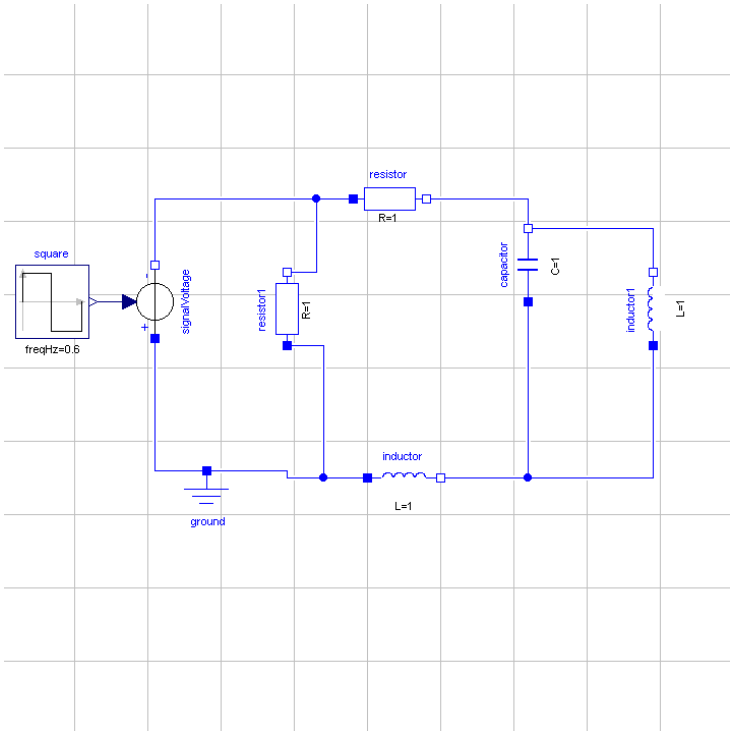


Figure 7.5 Electric Circuit

This examples assumes that the file `RLC_Circuit.mo` is located in the working directory.

Start by creating a Python script file and write or (copy paste) the command for importing the model object and for compiling a model together with the library used for plotting:

```
# Import the function for compilation of models and the JMUModel class
from jmodelica import compile_jmu
from pyjmi import JMUModel

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

Next, we compile and load the model:

```
# Compile model
jmu_name = compile_jmu("RLC_Circuit_Square","RLC_Circuit.mo")

# Load model
rlc = JMUModel(jmu_name)
```

Now we are ready to simulate our model. We are interested in simulating the model from 0.0 to 20.0 seconds. The start time is default to 0.0 so no need to change that, but the final time needs to be changed:

```
res = rlc.simulate(final_time=20.0) #Simulate the model from 0.0 to 20.0 seconds
```

After a successful simulation the statistics are printed in the prompt and the results are stored in the variable 'res'. To view the result, we have to retrieve information about the variables we are interested of which is easily done in the following way:

```
square_y    = res['square.y']
resistor_v  = res['resistor.v']
inductor1_i = res['inductor1.i']
time        = res['time']
```

And then plotted with the help from `matplotlib`,

```
plt.figure(1)
plt.plot(time, square_y, time, resistor_v, time, inductor1_i)
plt.legend(('square.y','resistor.v','inductor1.i'))
plt.show()
```

The simulation result is shown in Figure 7.6.

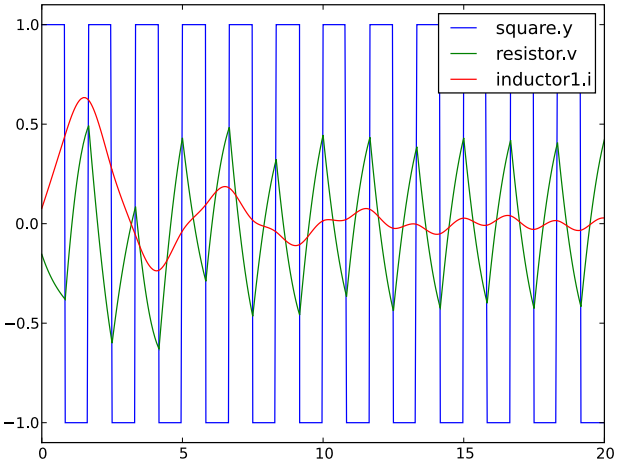


Figure 7.6 Simulation result

4.3. 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 differen-

tiated, 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 7.7. The model is of high index since there are two rotating inertias connected with a rigid gear.

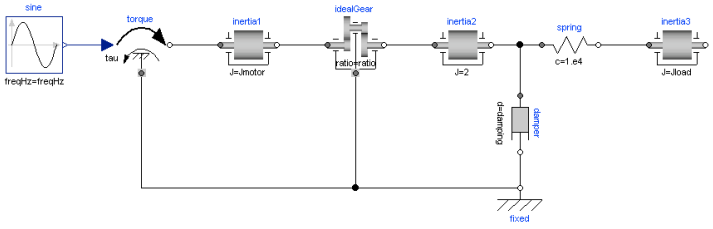


Figure 7.7 Modelica.Mechanics.Rotational.First connection diagram

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

```
import matplotlib.pyplot as plt
from jmodelica import compile_fmu
from pyfmi import FMUModel
```

Next, the model is compiled and loaded:

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

# Load model
model = FMUModel(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.

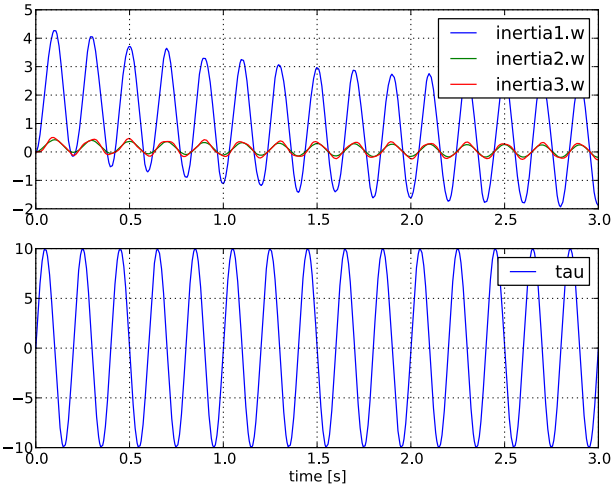


Figure 7.8 Simulation result for Mecahnics.Rotational.Examples.First

4.4. 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 jmodelica import compile_fmu
from pyfmi import FMUModel
```

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)

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

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.hold(True)
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):
    # 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 7.9.

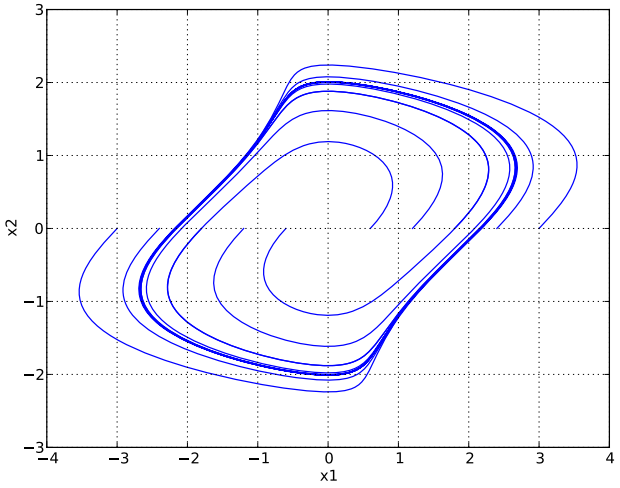


Figure 7.9 Simulation result-phase plane

4.5. Simulation with sensitivities

This example will show how to use JModelica.org to simulate an Optimica model and calculate sensitivities of the state variables with respect to a number of free parameters.

The model equations is taken from the Robertson example in the Sundials suite (https://computation.llnl.gov/casc/sundials/main.html) and the model is shown in the code below.

```
optimization Robertson
    parameter Real p1(free=true)=0.040;
    parameter Real p2(free=true)=1.0e4;
    parameter Real p3(free=true)=3.0e7;
```

```
    Real y1(start=1.0, fixed=true);
    Real y2(start=0.0, fixed=true);
    Real y3(start=0.0);
  equation
    der(y1) = -p1*y1 + p2*y2*y3;
    der(y2) =  p1*y1 - p2*y2*y3 - p3*(y2*y2);
    0.0 = y1 + y2 + y3 - 1;
end Robertson;
```

In the model we have set the parameters to free which means that we want to calculate sensitivities of the states with respect to the free parameters.

Lets begin with the the example, copy and paste the Optimica code and save it into `Robertson.mop` and open a python script file. We start by importing the necessary objects:

```
# Import the function for compilation of models and the JMUModel class
from jmodelica import compile_jmu
from pyjmi import JMUModel

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

Next, we compile and load the model:

```
# Compile model
jmu_name = compile_jmu("Robertson","Robertson.mop")

# Load model
model = JMUModel(jmu_name)
```

Notice that sensitivity computations are only supported for JMUModels currently. Now that the model is loaded, we have to change the option to activate the sensitivity calculations and also the absolute tolerances:

```
# Get and set the options
opts = model.simulate_options() #Get the options
opts['IDA_options']['atol'] = [1.0e-8, 1.0e-14, 1.0e-6] #Change the tolerance
opts['IDA_options']['sensitivity'] = True #Activate the sensitivity calculations
opts['ncp'] = 400 #Change the number of communication points
```

Now lets simulate the model:

```
res = model.simulate(final_time=4, options=opts)
```

The sensitivity results are stored as `d{variable name}/d{parameter name}` in the result object. We are interested in the following sensitivities:

```
dy1dp1 = res['dy1/dp1']
dy2dp1 = res['dy2/dp1']
```

```
dy3dp1 = res['dy3/dp1']
time = res['time']
```

To plot the trajectories using `matplotlib`, use the following commands:

```
plt.plot(time, dy1dp1, time, dy2dp1, time, dy3dp1)
plt.legend(('dy1/dp1', 'dy2/dp1', 'dy3/dp1'))
plt.show()
```

In Figure 7.10 the sensitivities are plotted.

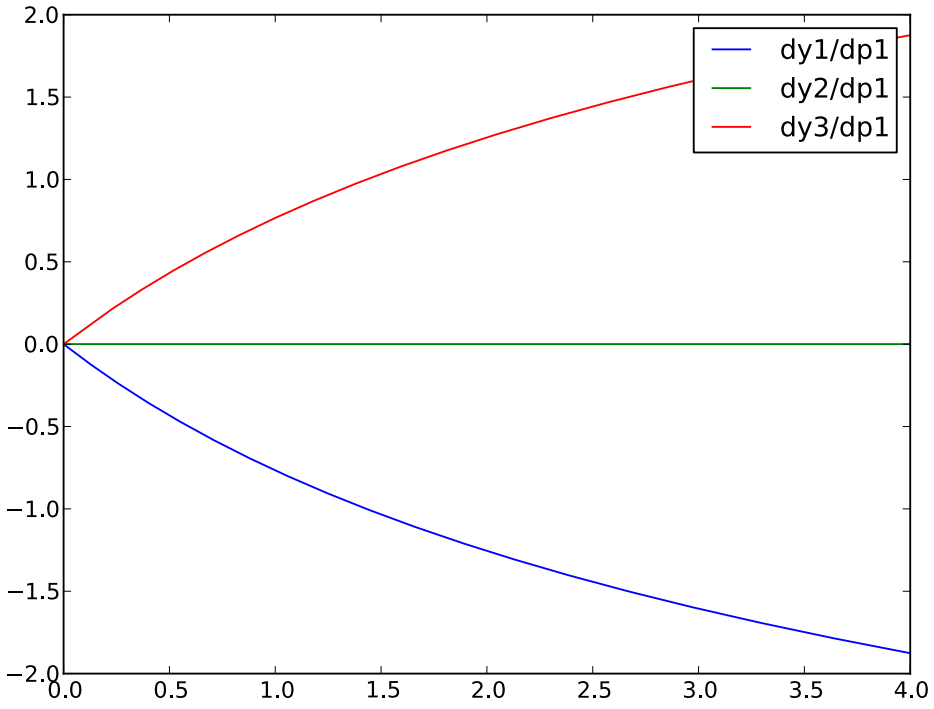


Figure 7.10 Sensitivity results.

4.6. Simulation of an FMU

This example will show how to use the JModelica.org's FMI-interface together with its simulation package, Assimulo. The FMU to be simulated is the full Robot from the Modelica standard library (3.1) where it is located

in `Mechanics.MultiBody.Examples.Systems.RobotR3`. It consists of brakes, motors, gears and path planning. The model consists of 36 continuous states and around 700 algebraic variables together with 98 event functions and also a few thousand constants/parameters. The FMU was generated using Dymola 7.4.

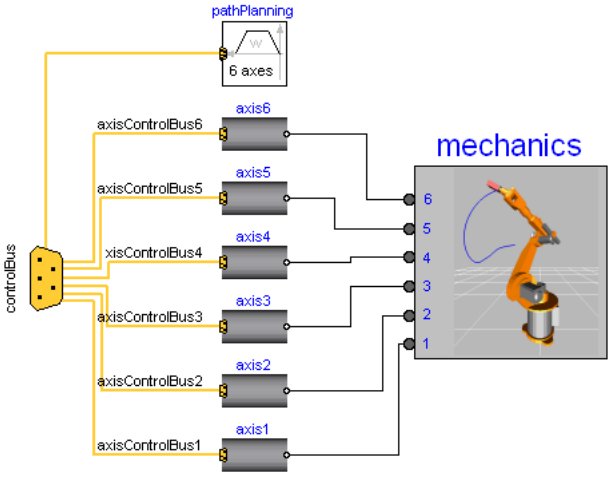


Figure 7.11 Full Robot

This examples assumes that an FMU of the robot named:

`Modelica_Mechanics_MultiBody_Examples_Systems_RobotR3_fullRobot.fmu`

exists in the working folder.

Start by creating a Python script file and write or (copy paste) the command for importing the model object and the library used for plotting:

```
# Import the FMUModel class
from pyfmi import FMUModel

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

Next, we load the FMU into the model object:

```
robot = FMUModel('Modelica_Mechanics_MultiBody_Examples_Systems_RobotR3_fullRobot.fmu')
```

We are interested in simulating the Robot from time 0.0 to 1.8 using 1000 communication points and using tolerances specified in the FMU. This information is specified to the simulate method:

```
res = robot.simulate(start_time=0.0, final_time=1.8, options={'ncp':1000})
```

This preforms the simulation and the statistics will be printed in the prompt.

To retrieve data about a variable from the result data, access it as a dictionary with the name of the variable as key:

```
dq1  = res['der(mechanics.q[1])']
dq6  = res['der(mechanics.q[6])']
time = res['time']
```

Now we have loaded and retrieved the variables of interest. So lets plot them.

```
plt.plot(time,dq1,time,dq6)
plt.legend(['der(mechanics.q[1])','der(mechanics.q[6])'])
plt.xlabel('Time (s)')
plt.ylabel('Joint Velocity (rad/s)')
plt.title('Full Robot')
plt.show()
```

In Figure 7.12 the result is shown and in Figure 7.13 a comparison between Dymola and JModelica.org is plotted.

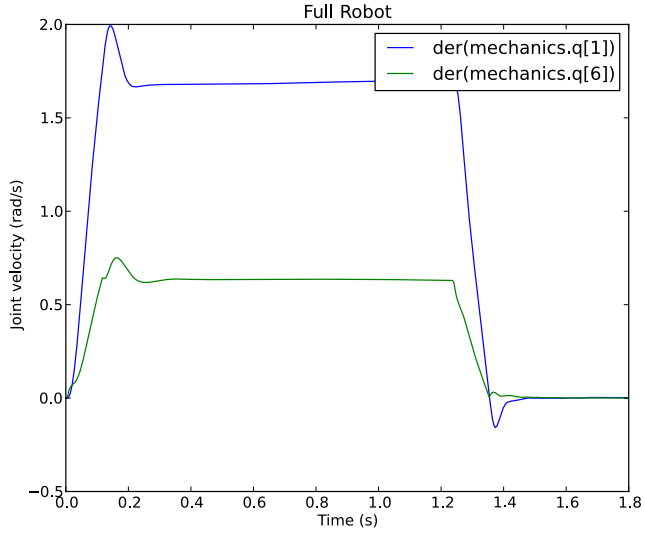


Figure 7.12 Robot Results

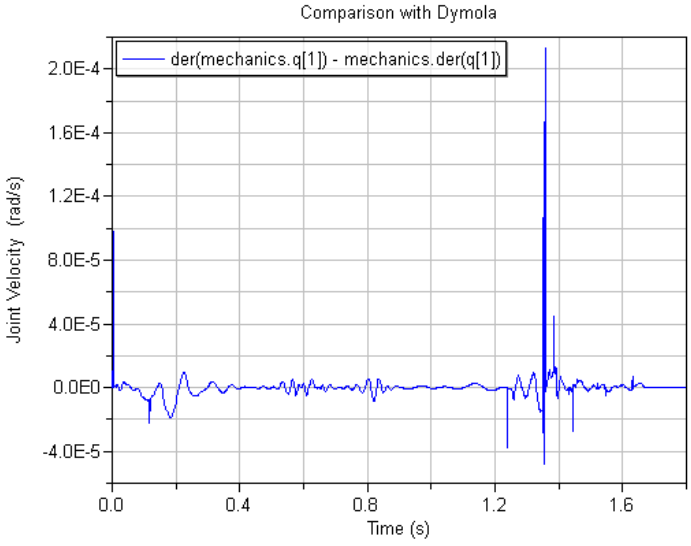


Figure 7.13 Comparison with Dymola

Chapter 8. Optimization

1. Introduction

JModelica.org 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. In this 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 2 and in Section 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 Section 9. The tutorial exercises in this chapter assumes that the reader is familiar with the basics of Modelica and Optimica.

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 (objective = cost(finalTime),
                      startTime = 0,
                      finalTime = 20)

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

  // The control signal
  input Real u;

  Real cost(start=0,fixed=true);

equation
  der(x1) = (1 - x2^2) * x1 - x2 + u;
  der(x2) = x1;
  der(cost) = x1^2 + x2^2 + u^2;
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 compilation of models and the JMUModel class
from jmodelica import compile_jmu
from pyjmi import JMUModel

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

Next, we compile and load the model:

```
# Compile model
jmu_name = compile_jmu("VDP_Opt","VDP_Opt.mop")
# Load model
vdp = JMUModel(jmu_name)
```

The function `compile_jmu` invokes the Optimica compiler and compiles the model into a DLL, which is then loaded when the `vdp` object is created. This object represents the compiled model and is used to invoke the optimization algorithm:

```
res = vdp.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:

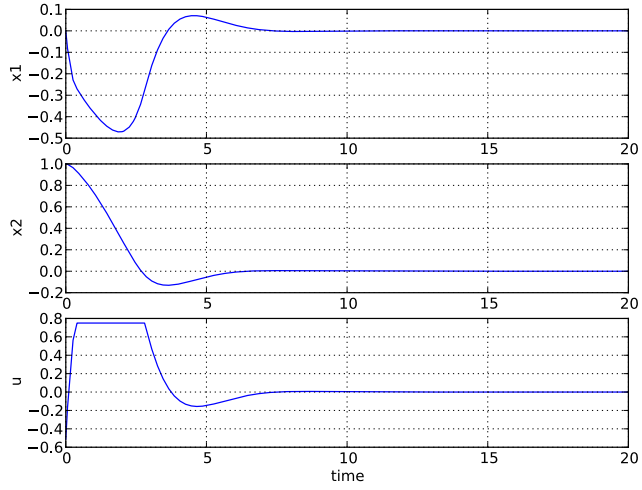
```
plt.figure(1)
plt.clf()
plt.subplot(311)
plt.plot(t,x1)
plt.grid()
plt.ylabel('x1')

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 8.1.



Optimal control and state profiles for the Van Der Pol optimal control problem.
Figure 8.1 Optimal profiles for the VDP oscillator

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 XX. To illustrate how to solve optimization problems the Van der Pol problem presented above is used. First, the model is compiled and loaded:

```
model_name = compile_jmu("VDP_Opt","VDP.mo")
model = JMUModel(model_name)
```

All operations that can be performed on the model are available as methods of the `model` object and can be accessed by tab completion. Invoking an optimization algorithm is done by calling the method `JMUModel.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 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(model.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:
            - 'CollocationLagrangePolynomialsAlg'. 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: 'CollocationLagrangePolynomialsAlg'

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

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

            Valid values are:
            - A dict that overrides some or all of the default values
              provided by CollocationLagrangePolynomialsAlgOptions. An empty
              dict will thus give all options with default values.
            - A CollocationLagrangePolynomialsAlgOptions object.
            Default: Empty dict

    Returns::

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

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

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

In the next section the available algorithms are described. 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(vpd.optimize_options)
    Get an instance of the simulate options class, prefilled with
    default values. If called without argument then the options
    class for the default simulation algorithm will be returned.

    Parameters::

        algorithm --
            The algorithm for which the options class should be
            fetched. Possible values are: 'AssimuloAlg',
            'AssimuloFMIAlg'.
            Default: 'AssimuloAlg'

    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 = vpd.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 supported algorithms are documented in Section 3.1.

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 in case of variables and as numeric values in the case of parameters. 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 as used when solving the optimization problem.

3.1. Algorithms

3.1.1. Direct collocation

The direct collocation method supported by JModelica.org 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. For an overview of simultaneous optimization algorithms, see [2]. The algorithm IPOPT is used to solve the non-linear program resulting from collocation.

The collocation method implemented in JModelica.org requires that the model to be optimized does not contain discontinuities such as if equations, when clauses or integer variables.

The mathematical formulation of the algorithm can be found in the JMI API documentation.

The collocation algorithm provides a number of options, summarized in Table 8.1.

Table 8.1 Options for the collocation-based optimization algorithm

Option	Default	Description
`n_e`	50	Number of elements of the finite element mesh.
`n_cp`	3	Number of collocation points in each element. Values between 1 and 10 are supported.
`hs`	Equidistant points using default n_e	A vector containing n_e elements representing the finite element lengths. The sum of all element should equal to 1.
`blocking_factors`	None (not used)	A vector of blocking factors. Blocking factors are specified by a vector of integers, where each entry in the vector corresponds to the number of elements for which the control profile should be kept constant. For example, the blocking factor specification [2,1,5] means that u_0=u_1 and u_3=u_4=u_5=u_6=u_7 assuming that the number of elements is 8. Notice that specification of blocking factors implies that controls are present in only one collocation point (the first) in each element. The number of constant control levels in the optimization interval is equal to the length of the blocking factor vector. In the example above, this implies that there are three constant control levels. If the sum of the entries in the blocking factor vector is not equal to the number of elements, the vector is normalized, either by truncation (if the sum of the entries is larger than the number of element) or by increasing the last entry of the vector. For example, if the number of elements is 4, the normalized blocking factor vector in the example is [2,1,1]. If the number of elements is 10, then the normalized vector is [2,1,7].

Option	Default	Description
`init_traj`	None (i.e. not used, set this argument to activate initialization)	Variable trajectory data used for initialization of the optimization problem. The data is represented by an object of the type `jmodelica.io.ResultDymolaTextual`.
`result_mode`	'default'	Specifies the output format of the optimization result. 'default' gives the the optimization result at the collocation points. 'element_interpolation' computes the values of the variable trajectories using the collocation interpolation polynomials. The option 'n_interpolation_points' is used to specify the number of evaluation points within each finite element. 'mesh_interpolation' computes the values of the variable trajectories at points defined by the option 'result_mesh'.
`n_interpolation_points`	20	Number of interpolation points in each finite element if the result reporting option result_mode is set to 'element_interpolation'.
`result_mesh`	None	A vector of time points at which the the optimization result is computed. This option is used if result_mode is set to 'mesh_interpolation'.
`result_file_name`	Empty string (default generated file name will be used)	Specifies the name of the file where the optimization result is written. Setting this option to an empty string results in a default file name that is based on the name of the optimization class.
`result_format`	'txt'	Specifies in which format to write the result. Currently only textual mode is supported.
`write_scaled_result`	False	Write the scaled optimization result if set to true. This option is only applicable when automatic variable scaling is enabled. Only for debugging use.

In addition to the options for the collocation algorithm, IPOPT options can also be set by modifying the dictionary `IPOPT_options` contained in the collocation algorithm options object. Here, all valid IPOPT options can be specified, see the IPOPT documentation for further information. For example, setting the option `max_iter`:

```
opts['IPOPT_options']['max_iter'] = 300
```

makes IPOPT terminate after 300 iterations even if no optimal solution has been found.

Some statistics from IPOPT can be obtained by issuing the command:

```
>>> res_opt.solver.opt_coll_ipopt_get_statistics()
```

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

```
>>> help(res.solver.opt_coll_ipopt_get_statistics)
    Get statistics from the last optimization run.

    Returns::

        return_status --
            Return status from IPOPT.

        nbr_iter --
            Number of iterations.

        objective --
            Final value of objective function.

        total_exec_time --
            Execution time.
```

4. 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 is 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 have 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. For more information about the DAE initialization algorithm, see the JMI API documentation.
- 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 increase. 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 JModelica.org 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 directely into a Python shell, in some cases with minor modifications. Alternatively, you may copy the commands into a text file, e.g., cstr.py.

Start the tutorial by creating a working directory and copy the file $JMODELICA_HOME/Python/jmodelica/examples/files/CSTR.mop to your working directory. An on-line version of CSTR.mop is also available (depending on which browser you use, you may have to accept the site certificate by clicking through a few steps). If you choose to create Python script file, save it to the working directory.

4.1. 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 jmodelica import compile_jmu
from pyjmi import JMUModel
```

Before we can do operations on the model, such as optimizing it, the model file must be compiled and the resulting DLL file loaded in Python. These steps are described in more detail Section 4.

```
# Compile the stationary initialization model into a JMU
jmu_name = compile_jmu("CSTR.CSTR_Init","CSTR.mop",
    compiler_options={"enable_variable_scaling":True})

# load the JMU
init_model = JMUModel(jmu_name)
```

Notice that automatic scaling of the model is enabled by setting the compiler option `enable_variable_scaling` to true. 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_jmu)
```

4.2. 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 inputs for Stationary point A
Tc_0_A = 250
init_model.set('Tc',Tc_0_A)

# Solve the DAE initialization system with Ipopt
init_result = init_model.initialize()
```

```
# Store stationary point A
c_0_A = init_result['c'][0]
T_0_A = init_result['T'][0]

# 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 `JMUModel` method `initialize`, which returns a result object from which the initialization result can be accessed. The `initialize` method relies on the algorithm IPOPT for computing the solution of the initialization problem. 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 a printout of the stationary values. Display the help text for the `initialize` method and take a moment to look through it. The procedure is now repeated for operating point B:

```
# Set inputs for Stationary point B
Tc_0_B = 280
init_model.set('Tc',Tc_0_B)

# Solve the DAE initialization system with Ipopt
init_result = init_model.initialize()
# Store stationary point B
c_0_B = init_result['c'][0]
T_0_B = init_result['T'][0]

# 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.

4.3. 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

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

$$T(t) \le 350$$

and is expressed in Optimica format in the class CSTR.CSTR_Opt 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. Since initial guesses are needed for all discretized variables along the optimization interval, simulation provides a convenient mean 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. Therefore, also the cost function is included in the initialization model.

First, compile the model and set model parameters:

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

# Load the model
init_sim_model = JMUModel(jmu_name)

# Set model parameters
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.

```
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 and to retrieve the simulation data use Python dictionary access to retrieve the variable trajectories.

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

# Plot the results
plt.figure(1)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(t_init_sim,c_init_sim)
plt.grid()
```

```
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(t_init_sim,T_init_sim)
plt.grid()
plt.ylabel('Temperature')

plt.subplot(313)
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 model containing the optimal control problem:

```
# Compile model
jmu_name = compile_jmu("CSTR.CSTR_Opt", "CSTR.mop")

# Load model
cstr = JMUModel(jmu_name)
```

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
cstr.set('Tc_ref',Tc_0_B)
cstr.set('c_ref',c_0_B)
cstr.set('T_ref',T_0_B)

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

Collocation-based optimization algorithms often require a good initial guess in order to achieve fast convergence. Also, if the problem is non-convex, initialization is even more critical. Initial guesses can be provided in Optimica by the `initialGuess` attribute, see the CSTR.mop file for an example for this. Notice that initialization in the case of collocation-based optimization methods means initialization of all the control and state profiles as a function of time. In some cases, it is sufficient to use constant profiles. For this purpose, the `initialGuess` attribute works well. In more difficult cases, however, it may be necessary to initialize the profiles using simulation data, where an initial guess for the input(s) has been used to generate the profiles for the dependent variables. This approach for initializing the optimization problem is used in this tutorial.

We are now ready to solve the actual optimization problem. This is done by invoking the method optimize:

```
n_e = 100 # Number of elements
```

```
# Set options
opt_opts = cstr.optimize_options()
opt_opts['n_e'] = n_e
opt_opts['init_traj'] = res.result_data

res = cstr.optimize(options=opt_opts)
```

In this case, we would like to increase the number of finite elements in the mesh from 50 to 100. This is done by setting the corresponding option and provide it as an argument to the `optimize` method. 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 an acceptable level' in order for an optimum to be 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 result
plt.figure(2)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(time_res,c_res)
plt.plot([time_res[0],time_res[-1]],[c_ref,c_ref],'--')
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(time_res,T_res)
plt.plot([time_res[0],time_res[-1]],[T_ref,T_ref],'--')
plt.grid()
plt.ylabel('Temperature')

plt.subplot(313)
plt.plot(time_res,Tc_res)
plt.plot([time_res[0],time_res[-1]],[Tc_ref,Tc_ref],'--')
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
```

```
plt.show()
```

Notice that parameters are returned as scalar values whereas variables are returned as vectors and that this must be taken into account when plotting. Your should now the plot shown in Figure 8.2.

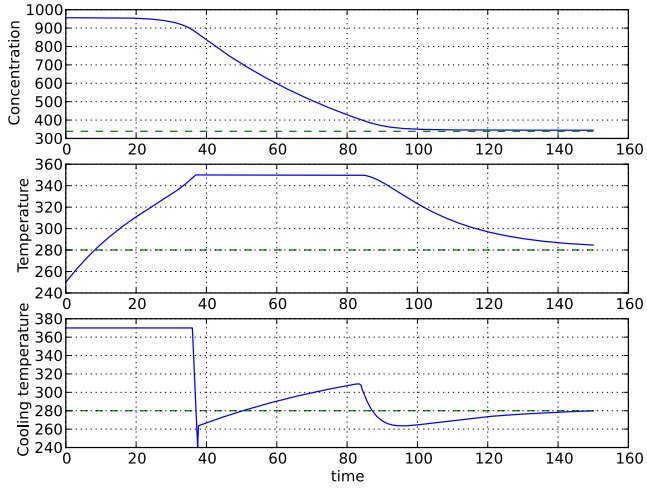


Figure 8.2 Optimal profiles for the CSTR problem.

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?

4.4. Verify optimal control solution

Solving optimal control problems by means of direct collocation implies that the differential equation is approximated by a discrete time 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 a DAE solver 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.

The procedure for setting up and executing this simulation is similar to above:

```
# Simulate to verify the optimal solution
```

```
# Set up the input trajectory
t = time_res
u = Tc_res
u_traj = N.transpose(N.vstack((t,u)))

# Compile the Modelica model to a JMU
jmu_name = compile_jmu("CSTR.CSTR", "CSTR.mop")

# Load model
sim_model = JMUModel(jmu_name)

sim_model.set('c_init',c_0_A)
sim_model.set('T_init',T_0_A)
sim_model.set('Tc',u[0])

res = sim_model.simulate(start_time=0.,final_time=150.,
    input=('Tc',u_traj))
```

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.hold(True)
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 8.3.

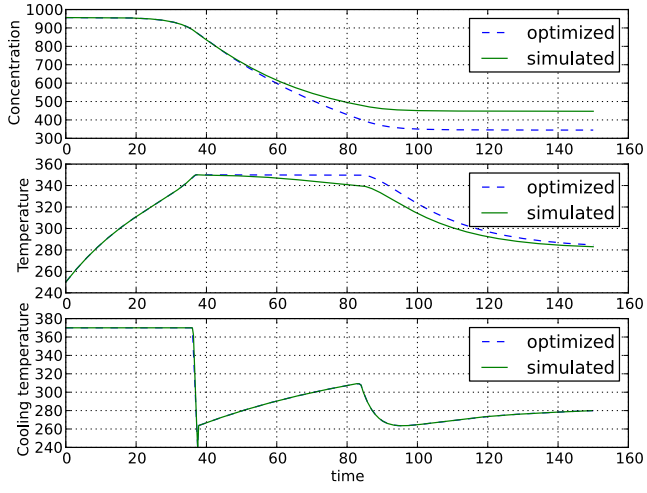


Figure 8.3 Optimal control profiles and simulated trajectories corresponding to the optimal control input.

Discuss why the simulated trajectories differs from the optimized counterparts.

4.5. 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. 2-10 collocation points are supported.

4.6. 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.

5. 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:

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

and the constraints:

$$\begin{aligned} x(t_f) &= 1, \quad v(t_f) = 0 \\ v(t) &\leq 0.5, \quad -1 \leq u(t) \leq 1 \end{aligned}$$

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 JModelica.org Python packages
from jmodelica import compile_jmu
from pyjmi import JMUModel

model_name = 'VDP_pack.VDP_Opt_Min_Time'

jmu_name = compile_jmu('VDP_Opt_Min_Time', 'VDP_Opt_Min_Time.mop')
vdp = JMUModel(jmu_name)
res = vdp.optimize()

# Extract variable profiles
x1=res['x1']
x2=res['x2']
u=res['u']
tf=res['finalTime']
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)
plt.grid()
plt.ylabel('u')
plt.xlabel('time')
plt.show()
```

The resulting control and state profiles are shown in Figure 8.4. Notice the difference as compared to Figure Figure 8.1, where the Van der Pol oscillator system is optimized using a quadratic objective function.

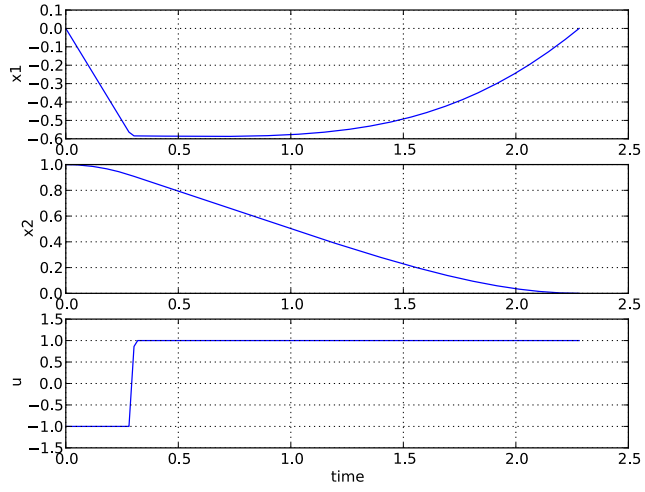


Figure 8.4 Minimum time profiles for the Van der Pol Oscillator.

6. Parameter optimization

In this tutorial it will be demonstrated how to solve parameter estimation problems. We consider a quadruple tank system depicted in Figure 8.5.

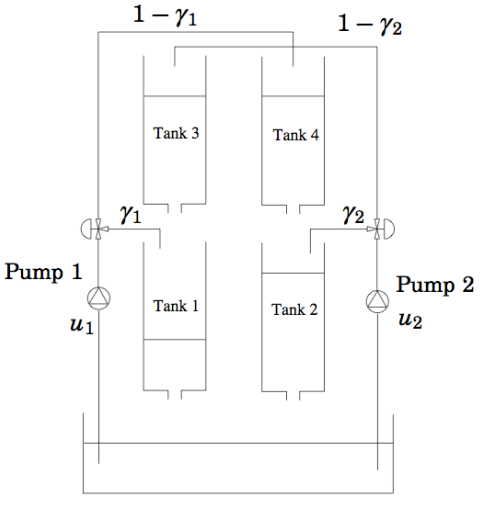


Figure 8.5 A schematic picture of the quadruple tank process.

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 parameter values are given in Table 8.2.

Table 8.2 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.py. Then enter the imports:

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

from jmodelica import compile_jmu
from pyjmi import JMUModel
```

into the file. 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.figure(1)
plt.clf()
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.show()

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

You should now see two plots showing the measurement state profiles and the control input profiles similar to Figure 8.6 and Figure 8.7.

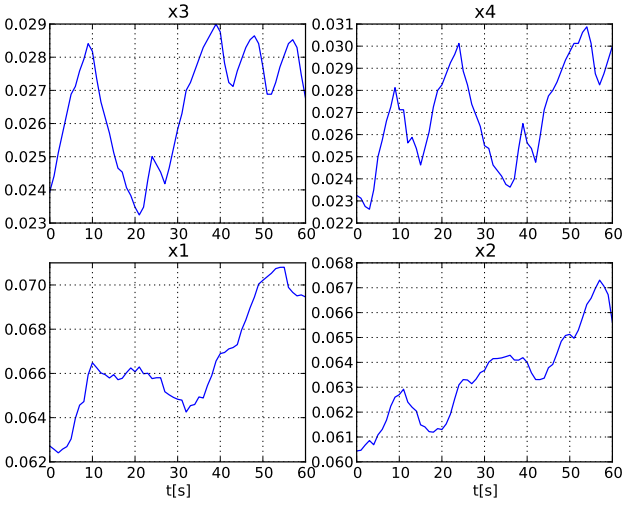


Figure 8.6 Measured state profiles.

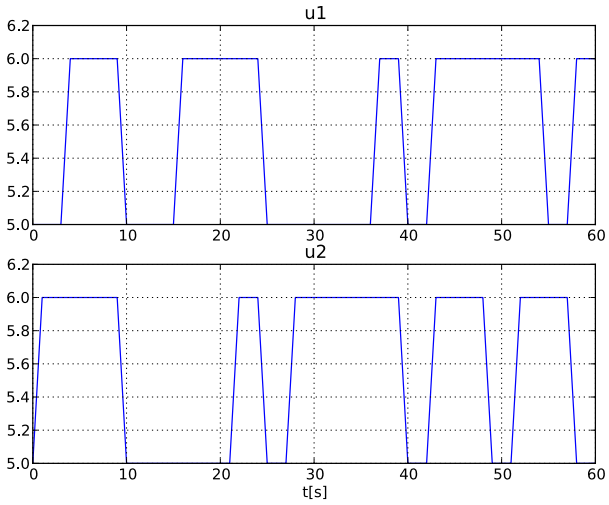


Figure 8.7 Control inputs used in the identification experiment.

In order to evaluate the accuracy of nominal model parameter values, start by simulating the model, assuming that the start values of the states are given by the state measurement at the start of the experiment. This assumption can be expressed in the model:

```
model Sim_QuadTank
```

```
  QuadTank qt;
  input Real u1 = qt.u1;
  input Real u2 = qt.u2;
initial equation
  qt.x1 = 0.0627;
  qt.x2 = 0.06044;
  qt.x3 = 0.024;
  qt.x4 = 0.023;
end Sim_QuadTank;
```

Notice that initial equations have been added to the model. Before the model is simulated, 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:

```
# compile JMU
jmu_name = compile_jmu('QuadTankPack.Sim_QuadTank','QuadTankPack.mop')

# Load model
model = JMUModel(jmu_name)

# Simulate model response with nominal parameters
res = 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['qt.x1']
x2_sim = res['qt.x2']
x3_sim = res['qt.x3']
x4_sim = res['qt.x4']
t_sim  = res['time']
u1_sim = res['u1']
u2_sim = res['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.show()
```

```
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 8.8 shows the result of the simulation.

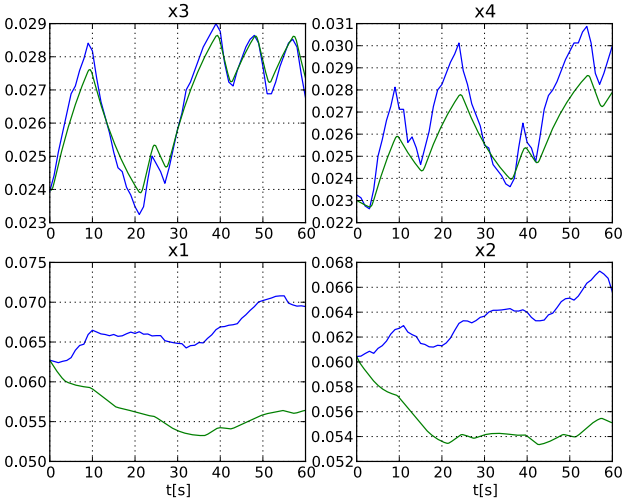


Figure 8.8 Simulation result for the nominal model.

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 contained in the class `QuadTankPack.QuadTank_ParEst`:

```
optimization QuadTank_ParEst (objective=sum((y1_meas[i] - qt.x1(t_meas[i]))^2 +
                                            (y2_meas[i] - qt.x2(t_meas[i]))^2 for i in 1:N_meas),
                                             startTime=0,finalTime=60)

    // Initial tank levels
  parameter Modelica.SIunits.Length x1_0 = 0.06255;
  parameter Modelica.SIunits.Length x2_0 = 0.06045;
```

```
  parameter Modelica.SIunits.Length x3_0 = 0.02395;
  parameter Modelica.SIunits.Length x4_0 = 0.02325;

  QuadTank qt(x1(fixed=true),x1_0=x1_0,
              x2(fixed=true),x2_0=x2_0,
              x3(fixed=true),x3_0=x3_0,
              x4(fixed=true),x4_0=x4_0,
              a1(free=true,initialGuess = 0.03e-4,min=0,max=0.1e-4),
              a2(free=true,initialGuess = 0.03e-4,min=0,max=0.1e-4));

  // Number of measurement points
  parameter Integer N_meas = 61;
  // Vector of measurement times
  parameter Real t_meas[N_meas] = 0:60.0/(N_meas-1):60;
  // Measurement values for x1
  // Notice that dummy values are entered here:
  // the real measurement values will be set from Python
  parameter Real y1_meas[N_meas] = ones(N_meas);
  // Measurement values for x2
  parameter Real y2_meas[N_meas] = ones(N_meas);
  // Input trajectory for u1
  PRBS1 prbs1;
  // Input trajectory for u2
  PRBS2 prbs2;
equation
  connect(prbs1.y,qt.u1);
  connect(prbs2.y,qt.u2);
end QuadTank_ParEst;
```

The cost function is here given as a squared sum of the difference between the measured profiles for x1 and x2 and the corresponding model profiles. Also the, parameters a1 and a2 are set to be free, and are given initial guesses as well as bounds. As for the measurement data, parameter vectors are declared, but only dummy data is provided in the model - the actual data values will be set from the Python script. Also, the input profiles are connected to signal generators that outputs the same input profiles as those used in the experiment. Take some time to look at QuadTankPack.mo and locate the classes used above.

Before the optimization problem can be solved, the Optimica specification needs to be compiled:

```
# Compile parameter optimization model
jmu_name = compile_jmu("QuadTankPack.QuadTank_ParEst","QuadTankPack.mop")

# Load the model
qt_par_est = JMUModel(jmu_name)
```

Next, we load the measurement data into the model:

```
# Number of measurement points
N_meas = N.size(u1,0)

# Set measurement data into model
```

```
for i in range(0,N_meas):
    qt_par_est.set("t_meas["+`i+1`+"]",t_meas[i])
    qt_par_est.set("y1_meas["+`i+1`+"]",y1_meas[i])
    qt_par_est.set("y2_meas["+`i+1`+"]",y2_meas[i])
```

We are now ready to solve the optimization problem:

```
n_e = 100 # Numer of element in collocation algorithm

# Get an options object for the optimization algorithm
opt_opts = qt_par_est.optimize_options()
# Set the number of collocation points
opt_opts['n_e'] = n_e

# Solve parameter optimization problem
res = qt_par_est.optimize(options=opt_opts)
```

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

```
# Extract optimal values of parameters
a1_opt = res["qt.a1"]
a2_opt = res["qt.a2"]

# Print optimal 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: 0.0272cm^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["qt.x1"]
x2_opt = res["qt.x2"]
x3_opt = res["qt.x3"]
x4_opt = res["qt.x4"]
u1_opt = res["qt.u1"]
u2_opt = res["qt.u2"]
t_opt  = res["time"]

# Plot
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.show()
```

You will see the plot shown in Figure 8.9.

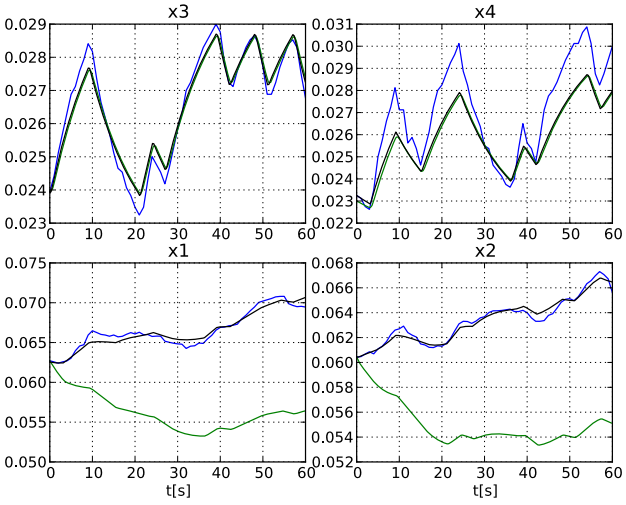


Figure 8.9 State profiles corresponding to estimated values of a1 and a2.

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 increased. Is the behavior of the model consistent with the estimated parameter values?

Never the less, 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. Take a minute to locate the class `QuadTankPack.QuadTank_ParEst2` and make sure that you understand the model. Solve the optimization problem by typing the Python code:

```
# Compile second parameter estimation model
jmu_name = compile_jmu("QuadTankPack.QuadTank_ParEst2", "QuadTankPack.mop")

# Load model
qt_par_est2 = JMUModel(jmu_name)
```

```
# Number of measurement points
N_meas = N.size(u1,0)

# Set measurement data into model
for i in range(0,N_meas):
    qt_par_est2.set("t_meas["+`i+1`+"]",t_meas[i])
    qt_par_est2.set("y1_meas["+`i+1`+"]",y1_meas[i])
    qt_par_est2.set("y2_meas["+`i+1`+"]",y2_meas[i])
    qt_par_est2.set("y3_meas["+`i+1`+"]",y3_meas[i])
    qt_par_est2.set("y4_meas["+`i+1`+"]",y4_meas[i])

# Solve parameter estimation problem
res_opt2 = qt_par_est2.optimize(options=opt_opts)
```

Next, we print the optimal parameter values:

```
# Get optimal parameter values
a1_opt2 = res_opt2["qt.a1"]
a2_opt2 = res_opt2["qt.a2"]
a3_opt2 = res_opt2["qt.a3"]
a4_opt2 = res_opt2["qt.a4"]

# Print optimal parameter values
print('a1:' + str(a1_opt2*1e4) + 'cm^2')
print('a2:' + str(a2_opt2*1e4) + 'cm^2')
print('a3:' + str(a3_opt2*1e4) + 'cm^2')
print('a4:' + str(a4_opt2*1e4) + 'cm^2')
```

The output in the console should be similar to:

```
a1:0.0266cm^2
a2:0.0271cm^2
a3:0.0301cm^2
a4:0.0293cm^2
```

Think about the result - can you explain why the estimated value of a4 is slightly smaller than the nominal value? Finally, plot the state profiles corresponding to the estimated parameters:

```
# Extract state and input profiles
x1_opt2 = res_opt2["qt.x1"]
x2_opt2 = res_opt2["qt.x2"]
x3_opt2 = res_opt2["qt.x3"]
x4_opt2 = res_opt2["qt.x4"]
u1_opt2 = res_opt2["qt.u1"]
u2_opt2 = res_opt2["qt.u2"]
t_opt2  = res_opt2["time"]

# Plot
plt.figure(1)
plt.subplot(2,2,1)
```

```
plt.plot(t_opt2,x3_opt2,'r')
plt.subplot(2,2,2)
plt.plot(t_opt2,x4_opt2,'r')
plt.subplot(2,2,3)
plt.plot(t_opt2,x1_opt2,'r')
plt.subplot(2,2,4)
plt.plot(t_opt2,x2_opt2,'r')
plt.show()
```

The resulting plot is shown in Figure 8.10.

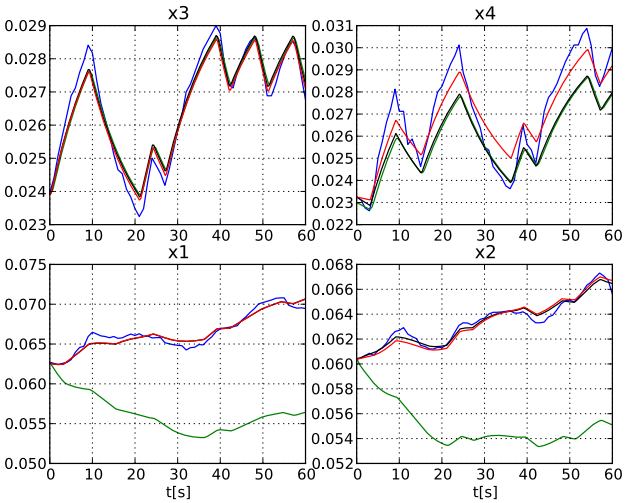


Figure 8.10 State profiles corresponding to estimated values of a1, a2, a3 and a4.

The red curves represent the case where a1, a2, a3 and a4 has been estimated.

Take a moment to think about the results. Are there other parameters that could have been selected for estimation?

Having computed the parameter values that fits the data, we proceed to compute the standard deviations for the parameter estimates. This information is valuable when judging how accurate the estimates are. For an introduction to statistical inference in parameter estimation problems, see [Eng2001].

The covariance matrix of the estimated parameter vector is given by the expression:

$$COV(p^*) = \hat{\sigma}_\varepsilon^2 \left(J^T J \right)^{-1}$$

where *J* is the Jacobian of the error residual and σ is the estimated measurement noise variance. In order to compute the residual Jacobian, the sensitivity equations needs to be computed.

The model `QuadTankPack.QuadTank_Sens2` is used for the sensitivity simulation. Notice that the `free` attribute is used to mark the parameters for which sensitivities should be computed:

```
optimization QuadTank_Sens2

   extends QuadTank (x1(fixed=true),x1_0 = 0.0627,
                     x2(fixed=true),x2_0 = 0.06044,
                     x3(fixed=true),x3_0 = 0.024,
                     x4(fixed=true),x4_0 = 0.023,
                     a1(free=true),
                     a2(free=true),
                     a3(free=true),
                     a4(free=true));

end QuadTank_Sens2;
```

In a first step to simulating the sensitivity equations for the model, we compile the model and set the optimal parameter values:

```
# compile JMU
jmu_name = compile_jmu('QuadTankPack.QuadTank_Sens2',
                       'QuadTankPack.mop')

# Load model
model = JMUModel(jmu_name)

model.set('a1',a1_opt2)
model.set('a2',a2_opt2)
model.set('a3',a3_opt2)
model.set('a4',a4_opt2)
```

Next, we set the `IDA_option` `sensitivity` to `true`, and simulate the model:

```
# Get an options object
sens_opts = model.simulate_options()

# Enable sensitivity computations
sens_opts['IDA_options']['sensitivity'] = True

# Simulate sensitivity equations
sens_res = model.simulate(input=(['u1','u2'],u),start_time=0.,
                          final_time=60, options = sens_opts)
```

Using the results of sensitivity simulation, the Jacobian and the residual error vector can be created:

```
# Get result trajectories
x1_sens = sens_res['x1']
x2_sens = sens_res['x2']
x3_sens = sens_res['x3']
x4_sens = sens_res['x4']

dx1da1 = sens_res['dx1/da1']
```

```
dx1da2 = sens_res['dx1/da2']
dx1da3 = sens_res['dx1/da3']
dx1da4 = sens_res['dx1/da4']

dx2da1 = sens_res['dx2/da1']
dx2da2 = sens_res['dx2/da2']
dx2da3 = sens_res['dx2/da3']
dx2da4 = sens_res['dx2/da4']

dx3da1 = sens_res['dx3/da1']
dx3da2 = sens_res['dx3/da2']
dx3da3 = sens_res['dx3/da3']
dx3da4 = sens_res['dx3/da4']

dx4da1 = sens_res['dx4/da1']
dx4da2 = sens_res['dx4/da2']
dx4da3 = sens_res['dx4/da3']
dx4da4 = sens_res['dx4/da4']
t_sens = sens_res['time']


# Create a trajectory object for interpolation
traj=TrajectoryLinearInterpolation(t_sens,
    N.transpose(N.vstack((x1_sens,x2_sens,x3_sens,x4_sens,
                          dx1da1,dx1da2,dx1da3,dx1da4,
                          dx2da1,dx2da2,dx2da3,dx2da4,
                          dx3da1,dx3da2,dx3da3,dx3da4,
                          dx4da1,dx4da2,dx4da3,dx4da4))))

# Create Jacobian
jac = N.zeros((61*4,4))

# Error vector
err = N.zeros(61*4)

# Extract Jacobian and residual error information
i = 0
for t_p in t_meas:
    vals = traj.eval(t_p)
    for j in range(4):
        for k in range(4):
            jac[i+j,k] = vals[0,4*j+k+4]
        err[i] = vals[0,0] - y1_meas[i/4]
        err[i+1] = vals[0,1] - y2_meas[i/4]
        err[i+2] = vals[0,2] - y3_meas[i/4]
        err[i+3] = vals[0,3] - y4_meas[i/4]
    i = i + 4
```

Notice the convention for how the sensitivity variables are named.

Finally, we compute and print the standard deviations for the estimated parameters:

```
# Compute estimated variance of measurement noice
v_err = N.sum(err**2)/(61*4-2)

# Compute J^T*J
A = N.dot(N.transpose(jac),jac)

# Compute parameter covariance matrix
P = v_err*N.linalg.inv(A)

# Compute standard deviations for parameters
sigma_a1 = N.sqrt(P[0,0])
sigma_a2 = N.sqrt(P[1,1])
sigma_a3 = N.sqrt(P[2,2])
sigma_a4 = N.sqrt(P[3,3])

print "a1: " + str(sens_res['a1']) + ", standard deviation: " + str(sigma_a1)
print "a2: " + str(sens_res['a2']) + ", standard deviation: " + str(sigma_a2)
print "a3: " + str(sens_res['a3']) + ", standard deviation: " + str(sigma_a3)
print "a4: " + str(sens_res['a4']) + ", standard deviation: " + str(sigma_a4)
```

You should now see the standard deviations for the estimated parameters printed.

7. Scaling

Many physical models contains variables with values that differs several orders of magnitude. A typical example is thermodynamic models containing pressures, temperatures and mass flows. Such large differences in values 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 about 1e5. In order to use `nominal` attributes for scaling, the compiler option `enable_variable_scaling` is set to `True`, see Section 2.2.2. All variables with a nominal attribute set to true, is then scaled by dividing the variable value with its nominal value, i.e., from an algorithm point of view, all variables will 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*T_nom)
```

when `enable_variable_scaling` is set to `true`.

For debugging purposes, is 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 requires additional scaling. The option write_scaled_result has been introduced as an option to the `initialize`, `simulate` and `optimize` methods of `JMUModel` for this purpose.

8. Optimization using Pseduo-Spectral methods.

New collocation algorithms have been implemented using the automatic differentiation tool CasADi connected to IPOPT. The collocation algorithms are the Gauss Pseudospectral-, the Legendre Pseudospectral- and the Radau Pseudospectral-method. These methods can be used either as a local method or a global method where in the local case the time horizon is discretized into elements and each element are approximated with a polynomial. In the global case, a polynomial is used over the complete time horizon. For a mathematic overview of the methods, see the Python API documentation, for the class `PseudoSpectral`.

The implementation resides in the module `casadi_interface` and contains a model class `CasadiModel`. A compiler method for creating models compliant with `CasadiModel` resides in `jmodelica`, `compile_fmux`.. Usage of the methods is consistent with the work flow for optimization of `JMUModels`. Shown below is a Python code example describing how to invoke the optimization method.

```
from jmodelica import compile_fmux
from pyjmi import CasadiModel

casadi_name = compile_fmux("MyModel", "MyModels.mop")
model = CasadiModel(casadi_name)

res = model.optimize(algorithm="CasadiPseudoSpectral")

#Plot...
```

As can be seen from the above code example the real difference from the native optimization method is that here we use a different model class and a different compilation class, except from that, it can be used in similar fashion.

8.1. CasADi

CasADi is a "minimalistic computer algebra system implementing automatic differentiation". It is used here to discretize the problem and to provide the necessary derivations, while also provide a direct connection to IPOPT. CasADi can be found here, http://www.casadi.org and is required in order to use these optimization methods.

8.2. Options

Options can be viewed and set via the options dictionary which can then be passed to the optimize call. The option dictionary can be retrieved with the below code example where `model` is a `CasadiModel` instance. It is also shown how to use the interactive help.

```
opts = model.optimize_options(algorithm="CasadiPseudoSpectral")
```

```
opts? #View the help text
```

To provide the options to the optimize call, use the options attribute.

```
res = model.optimize(algorithm="CasadiPseudoSpectral", options=opts)
```

In Table 8.3 the options for the PseudoSpectral methods are shown.

Table 8.3 Options for the Pseudospectral optimization algorithms.

Option	Default	Description
`n_e`	1	Number of elements of the finite element mesh.
`n_cp`	20	Number of collocation points in each element. Values between 1 and 80 are supported.
`discr`	"LG"	Determines the discretization of the problem. Can be either "LG" (Legendre-Gauss), LGR (Legendre-Gauss-Radau), LGL (Legendre-Gauss-Lobatto).
`link_options`	[] (not used)	This option allows users to specify states that are allowed to be discontinuous between phases (elements) and to connect the transition with a model parameter. Example, [(1,"x1","dx1")], this allowes the variable x1 to be discontinuous between phase 1 and 2 with the parameter dx1. The generating constraint becomes, x1_N^1 - x1_0^2 - dx1 = 0. There is no limit that the same parameter can be used in multiple transition such as, [(1,"x1","dx1"), (1,"x2","dx1"), (2,"x1","dx1")]
phase_options	None (not used)	This options allows users to connect parameters to phase boundaries (time). Example, in a three phase problem the parameters t1 and t2 can be specified to be the boundaries of the phases such as, ["t1", "t2"], the option free_phases have also be set to true. Default: None
free_phases	False	Specifies if the location of the phases should be allowed to be changed by the optimizer. Default: False
`init_traj`	None (i.e. not used, set this argument to activate initialization)	Variable trajectory data used for initialization of the optimization problem. The data is represented by an object of the type `jmodelica.io.ResultDymolaTextual`.
`result_file_name`	Empty string (default generated file name will be used)	Specifies the name of the file where the optimization result is written. Setting this option to an empty string

Option	Default	Description
		results in a default file name that is based on the name of the optimization class.
`result_format`	'txt'	Specifies in which format to write the result. Currently only textual mode is supported.

Options that need a little more attention are the `free_phases`, `link_options` and `phase_options`. These options allows a user to setup a problem involving discontinuities. As an example, say that in a model a state is allowed to be changed discontinuous at an arbitrary time point, `tp`, with the parameter `dv`. See Figure 8.11.

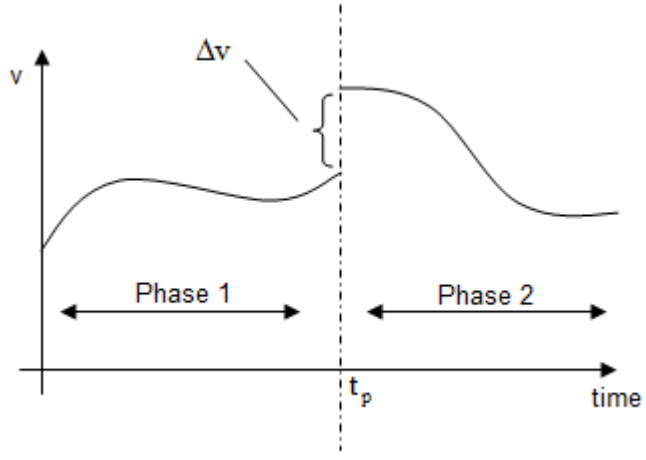


Figure 8.11 Handling of discontinuities.

In this case we set up the solver to use two elements (phases) and at the phase boundary we impose an extra constraint that our state in phase one plus the discontinuity should be equal to our state in phase two.

```
opts = model.optimize_options(algorithm="CasadiPseudoSpectral")

opts["n_e"]=2 #Two elements
opts["link_options"] = [(1,"v","dv")] #The parameter dv should be connected
                                      #between phase 1 and 2 to variable v
opts["phase_options"] = ["t1"] #The parameter t1 connects phase 1 and 2
opts["free_phases"] = True #The phase boundary is allowed to be changed.
```

Note that the parameters `dv` and `t1` should be defined in the model and should be set to free.

8.3. Examples

In the following subsections it will be shown how to use the PseudoSpectral methods on `Optimica` models.

8.3.1. Van der Pol oscillator

This example shows the well known Van der Pol (VDP) oscillator being optimized as an optimal control problem. The formulation is somewhat different than in previous example of the VDP problem, as here there is no extra `cost` state, instead the cost function have been directly defined in the Lagrange term (objectiveIntegrand). There is also no bounds on `u`. Below, the `Optimica` for the example is shown.

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

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

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

   // The control signal
   input Real u;

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

To compile and load the VDP model into JModelica we first import the necessary Python modules. Then we call the compiler method and we load it into our model class. Finally the optimization is performed by a call to the optimize method. An extra note in this example is that we change the number of collocation points from the default 20 to 40 via the options dictionary.

```
from jmodelica import compile_fmux
from pyjmi import CasadiModel

casadi_name = compile_fmux("VDP_Opt", "VDP.mop")
model = CasadiModel(casadi_name)

#Get the options dictionary
opts = model.optimize_options(algorithm="CasadiPseudoSpectral")
opts["n_cp"] = 40 #Number of collocation points

res = model.optimize(algorithm="CasadiPseudoSpectral", options=opts)

#Plot...
```

The result can then be used to visualize the result and a resulting figure is shown in Figure 8.12.

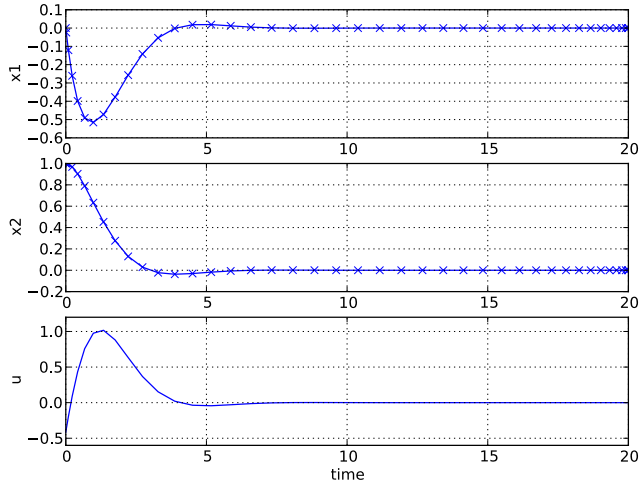


Figure 8.12 The Van der Pol oscillator optimized using Gauss PseudoSpectral method.

8.3.2. Hohmann Transfer

This example will show how to use the pseudo-spectral method for solving a problem of altitude change of a satellite orbiting earth. The problem have a well known solution called a Hohmann Transfer from the German scientist Walter Hohmann. The solution consists of two rocket burns, one for leaving the current orbit and one when the final orbit have been reached, both perpendicular to the radial component in a rotation coordinating system. The magnitude of the rocket burns can also be calculated theoretically.

We will solve this problem using the gauss pseudo-spectral method with two phases. The phases will be connected in such a way that the velocities are allowed to be discontinuous, see Figure 8.11 . The magnitude of the discontinuity will correspond to the first rocket burn. The second rocket burn are connected as constraints at the final time.

We start by setting up our model of a planar two-body system where the bodies are the earth and the satellite. We also scale the problem with the orbital time and the radius of the earth.

```
model Orbital
  //"State variables"
  Real x(fixed=true,start=x0);
  Real y(fixed=true,start=y0);
  Real vx(fixed=true,start=vx0);
  Real vy(fixed=true,start=vy0);

  //"Initial values"
  parameter Real x0=rStart;
  parameter Real y0=0;
  parameter Real vx0=0;
  parameter Real vy0=vStart;
```

```
  //"Constants"
  constant Real G=6.67384e-11 "Gravitational Constant";
  constant Real M=5.9742e24 "Earth Mass";
  constant Real R=6.371e6 "Earth Radii";

  //"Parameters"
  parameter Real my=G*M/R^3*T^2;
  parameter Real T=5400; //Orbital time of rStart

  parameter Real rFinal = rStart*2 "Radii of final orbit";
  parameter Real rStart = (R+300e3)/R "Radii of start orbit";
  parameter Real vStart = 7730/R*T "Velocity of start orbit";

equation //Equation of motion of a planar two-body problem
  der(x)=vx;
  der(y)=vy;
  der(vx)=-x*my/(x^2+y^2)^(3/2);
  der(vy)=-y*my/(x^2+y^2)^(3/2);
end Orbital;
```

Our starting orbit is 300 km above the earths surface and our target orbit is 6971 km.

Next, we set up our optimization problem.

```
optimization HohmannTransfer(objective=(dx1^2+dy1^2)^0.5+(dx2^2+dy2^2)^0.5, startTime=0, finalTime(free=tr
  extends Orbital;

  //"Rocket burns"
  parameter Real dx1(free=true,min=-2,max=2,start=0.05);
  parameter Real dy1(free=true,min=-2,max=2,start=0.05);
  parameter Real dx2(free=true,min=-2,max=2,start=0.05);
  parameter Real dy2(free=true,min=-2,max=2,start=0.05);

  //"Time for firing the first rockets"
  parameter Real t1(free=true,min=0.001,max=0.01, start=0.005);

constraint
  (x(finalTime)^2+y(finalTime)^2)^0.5=rFinal; //"Final position constraint"
  (vx(finalTime)+dx2)^2+(vy(finalTime)+dy2)^2=my/rFinal; //"Final velocity"
  x(finalTime)*(vx(finalTime)+dx2)+y(finalTime)*(vy(finalTime)+dy2)=0; //"Final Radial velocity component
end HohmannTransfer;
```

As can be seen from the above the code, our objective is to minimize the rocket burns (d1 and d2) with the final time unknown. The constraints are set on the final position (circular) and on the final velocities so that we are actually staying in the target orbit. There is also a parameter t1 set to be free which will be connected to the problem via Python and will be used to specify the time of the first rocket burn. From the problem, one can also see that dx1 and dy1 are not included in the problem, except for the objective. They will be connected via Python and determine the velocity changes between phase 1 and phase 2. Note that in the following scripts the models have been stored as `Hohmann.mop` .

We start our script by importing the necessary modules and methods for compiling the two different models and also for loading the compiled models into JModelica.orgs framework.

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

from jmodelica import compile_jmu
from pyjmi import JMUModel

from jmodelica import compile_fmux
from pyjmi import CasadiModel

jmu_name  = compile_jmu("Orbital","Hohmann.mop")
fmux_name = compile_fmux("HohmannTransfer", "Hohmann.mop")
```

Next, the optimization model needs to be loaded and options specified.

```
#Optimization
model = CasadiModel(fmux_name)

opts = model.optimize_options(algorithm="CasadiPseudoSpectral")

opts["n_cp"] = 40 #Number of collocation points
opts["n_e"] = 2   #Number of phases
opts["free_phases"] = True #The phase boundary is allowed to be changed in time
opts["phase_options"] = ["t1"] #The phase boundary is connected to variable t1
opts["link_options"] = [(1,"vy","dy1"),(1,"vx","dx1")] #Allow for discontinuities between phase 1 and 2 fo
                                                       #The discontinuities are connected by dy1 and dx1
```

In the above code we have loaded our optimization model `fmux_name` into a class, `CasadiModel`. Our options dictionary is retrieved by calling the optimization options method with our method of choice as input, `CasadiPseudoSpectral`. We then specify the number of collocation points in each phase and also the number of phases. The next options are a bit more interesting. The option `free_phases` specifies that the boundary between phase 1 and phase 2 are allowed to be changed by the optimizer and the `phase_options` specifies that the boundary should be connected to the parameter `t1` in the model where we also have specified constraints on the parameter. Finally we connect the first rocket burn (dx1 and dy1) to the corresponding variables, dx and dy, allowing them to be changed on the phase boundary.

Now we are ready to optimize our problem.

```
#Optimize
res_opt = model.optimize(algorithm="CasadiPseudoSpectral",options=opts)

#Get results
dx1,dy1,dx2,dy2 = res_opt["dx1"],      res_opt["dy1"],    res_opt["dx2"],res_opt["dy2"]
r1,r2,my        = res_opt["rStart"],   res_opt["rFinal"], res_opt["my"]
tf,t1           = res_opt["time"][-1], res_opt["t1"]
```

The above code performs the optimization and also retrieves the values used to verify the solution.

To verify the solution we perform simulations using the resulting rocket burns and even simulate further in time to verify that we are staying in the targeted orbit.

```
#Verify solution by simulation
model = JMUModel(jmu_name)

#Simulation of Phase 1
res = model.simulate(final_time=t1,options={'ncp':100,'solver':"CVode"})

x_phase_1,y_phase_1 = res["x"], res["y"]

#Simulation of Phase 2
model.set("vx",dx1+res["vx"][-1])
model.set("vy",dy1+res["vy"][-1])

res = model.simulate(start_time=t1,final_time=tf,options={'ncp':100,'solver':"CVode"})

x_phase_2,y_phase_2 = res["x"], res["y"]

#Simulation of Phase 3 (verify that we are still in orbit)
model.set("vx",dx2+res["vx"][-1])
model.set("vy",dy2+res["vy"][-1])

res = model.simulate(start_time=tf,final_time=tf*2,options={'ncp':100,'solver':"CVode"})

x_phase_3,y_phase_3 = res["x"], res["y"]
```

In the above simulation code we used the model compiled in the beginning of the script ,`"Orbital"` and then performed three simulations. One for the first phase, one for the second phase and finally an extra simulation for verification. In between the simulations we added the calculated rocket burns to the velocities and also retrieved the result.

Next, we set up the figure where we plot the the initial and target orbit together with the earth and the satellite trajectory. We also plot the rocket burns.

```
#Plot Earth
r = 1.0
xE = r*N.cos(N.linspace(0,2*N.pi,200))
yE = r*N.sin(N.linspace(0,2*N.pi,200))
plt.plot(xE,yE,label="Earth")

#Plot Orbits
r = res["rStart"]
xS = r*N.cos(N.linspace(0,2*N.pi,200))
yS = r*N.sin(N.linspace(0,2*N.pi,200))
plt.plot(xS,yS,label="Low Orbit")

r = res["rFinal"]
xSE = r*N.cos(N.linspace(0,2*N.pi,200))
ySE = r*N.sin(N.linspace(0,2*N.pi,200))
```

```
plt.plot(xSE,ySE,label="High Orbit")

#Plot Satellite trajectory
x_sim=N.hstack((N.hstack((x_phase_1,x_phase_2)),x_phase_3))
y_sim=N.hstack((N.hstack((y_phase_1,y_phase_2)),y_phase_3))

plt.plot(x_sim,y_sim,"-",label="Satellite")

#Plot Rocket Burns
plt.arrow(x_phase_1[-1],y_phase_1[-1],0.5*dx1,0.5*dy1, width=0.01,label="dv1")
plt.arrow(x_phase_2[-1],y_phase_2[-1],0.5*dx2,0.5*dy2, width=0.01,label="dv2")

plt.legend()
plt.show()
```

In Figure 8.13 the result is visualized. The arrows represents the rocket burns.

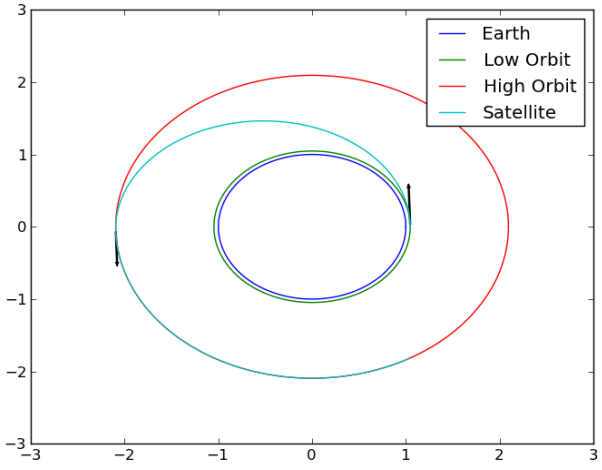


Figure 8.13 Hohmann Transfer

8.4. Limitations

Current limitations for the PseudoSpectral methods include,

- Final time or start time cannot be explicitly in the cost function. Although states of final time /start time can be used.
- No path constraints. However variable bounds can be used.
- Only explicit ordinary differential equations.

- Still under development and not extensively tested, consider using `JMUModel.optimize` as a first choice.
- No support for scaling.

9. Derivative-Free Model Calibration of FMUs

Figure 8.14 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 8.14. 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/pyfmi/examples/files/FMUs/Furuta.fmu to your working directory. **Notice that this the Furuta.fmu file currently only supports 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 FMUModel
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 8.15 showing the measurements of θ and φ.

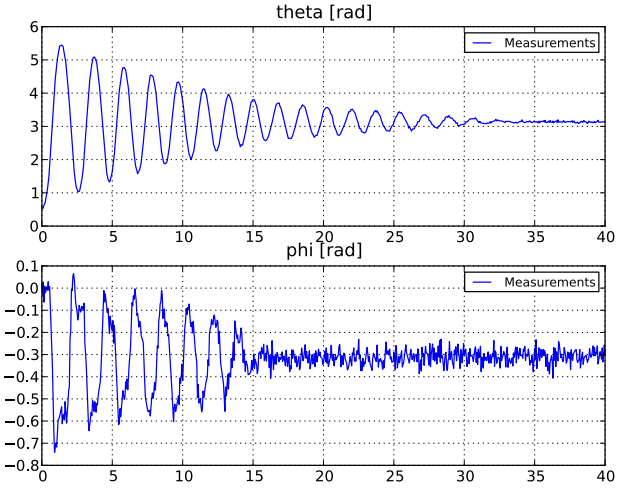


Figure 8.15 Measurements of θ and φ for the Furuta pendulum.

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

```
# Load model
model = FMUModel("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 8.16 shows the simulation result together with the measurements.

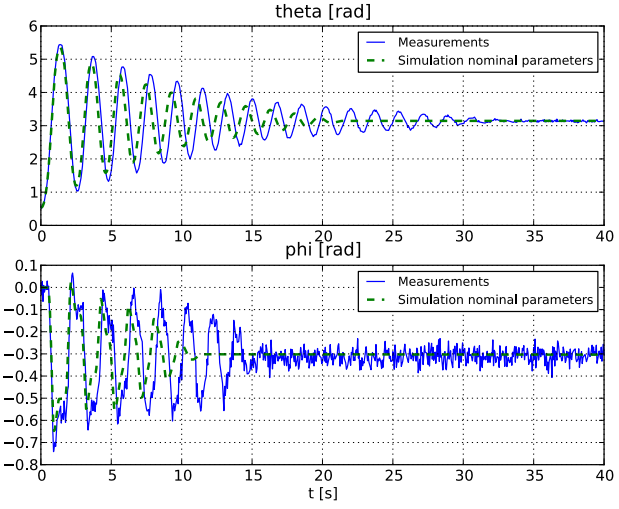


Figure 8.16 Measurements and model simulation result for φ and θ when using nominal parameter values in the Furuta pendulum model.

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:

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

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 FMUModel
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 = FMUModel('../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 shall set the optimized parameter values into the model and simulate it:

```
# Load model
```

```
model = FMUModel("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']
```

And 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 8.17. As can be seen, the agreement between the measurements and the simulation result has improved considerably. The model has been successfully calibrated.

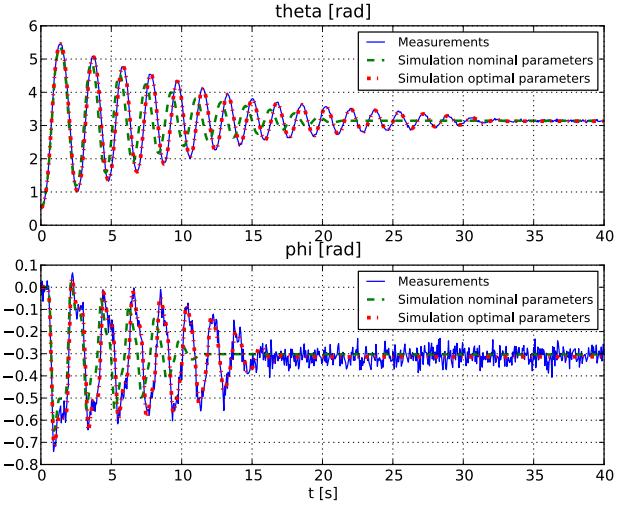


Figure 8.17 Measurements and model simulation results for φ and θ with nominal and optimal parameters in the model of the Furuta pendulum.

Chapter 9. Graphical User Interface for Visualization of Results

1. Plot GUI

JModelica.org comes with a graphical user interface (GUI) for displaying simulation and / or optimization results. The GUI supports result files generated by JModelica.org 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 JModelica.org. The GUI can also be started by typing the following commands in a Python shell:

```
from pyjmi.common.plotting import plot_gui #or pyfmi.common.plotting import plot_gui
plot_gui.startGUI()
```

Note that the GUI requires the Python package wxPython.

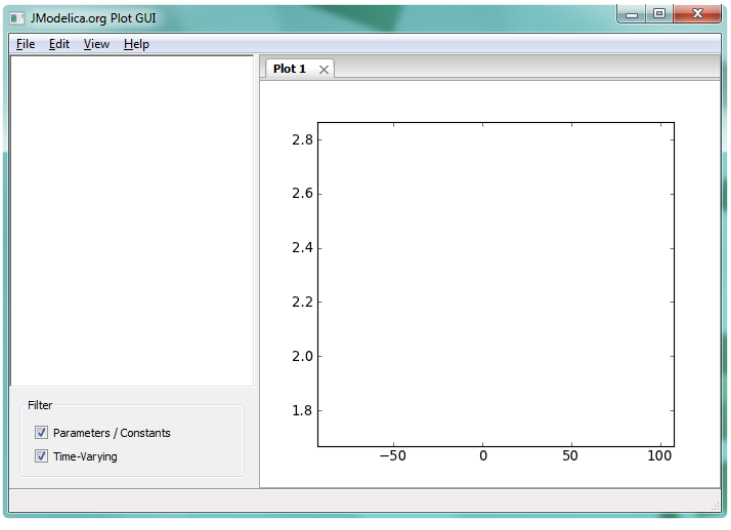


Figure 9.1 Overview of JModelica.org Plot GUI

1.1. Introduction

An overview of the GUI is shown in Figure 9.1. 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 its(theirs) 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 a user to easily browse between components in a model, see Figure 9.2 . Multiple results can be loaded either simultaneously or separately by using the `File` menu option `Open` repeatedly.

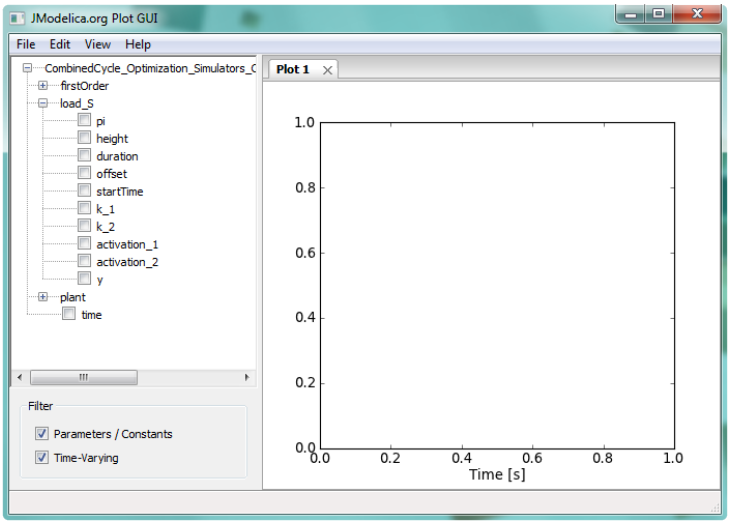


Figure 9.2 A result file has been loaded.

Displaying trajectories is done by simply checking the box associated with the variable of interest, see Figure 9.2. Removing a trajectory follows the same principle.

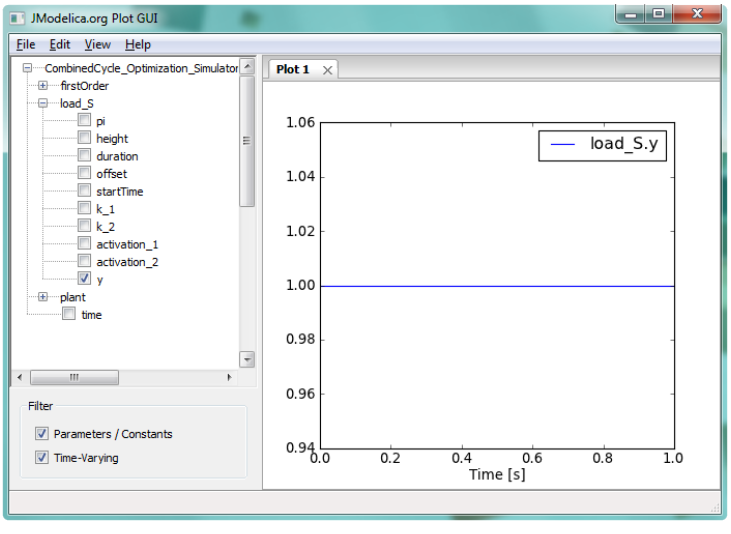


Figure 9.3 Plotting a trajectory.

A result can also be removed from the tree view by selecting an item in the tree and by pressing the delete key.

1.2. Edit Options

The GUI allows a range of options 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.

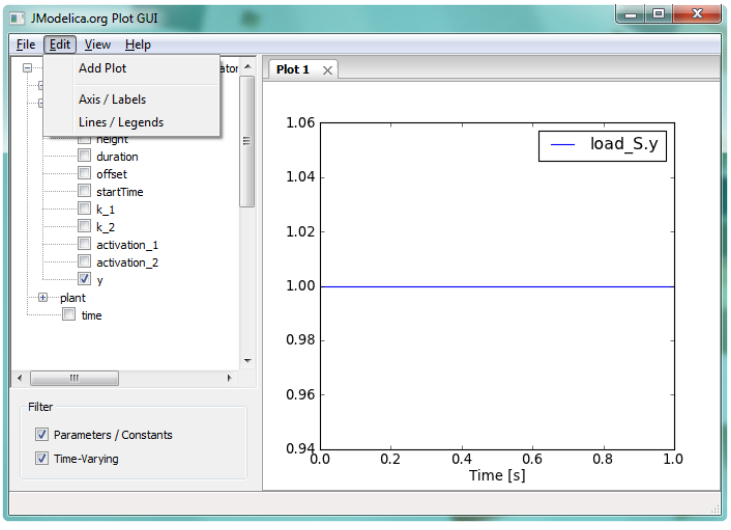


Figure 9.4 Figure Options.

Under `Axis/Labels`, options such as defining titles and labels in both X and Y direction can be found together with axis options.

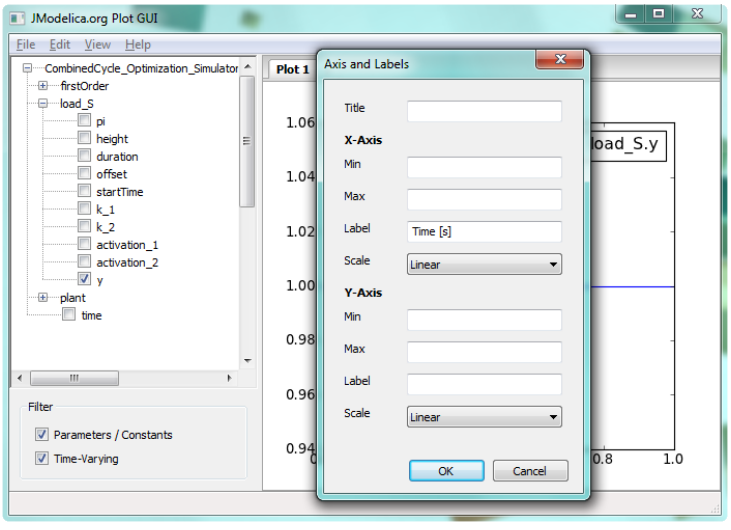


Figure 9.5 Figure Axis and Labels Options.

Under `Lines/Legends`, options for specifying specific line labels and line styles can be found, see Figure 9.6. 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` have been pressed. For changing multiple lines in the same session, the `Apply` button should be used.

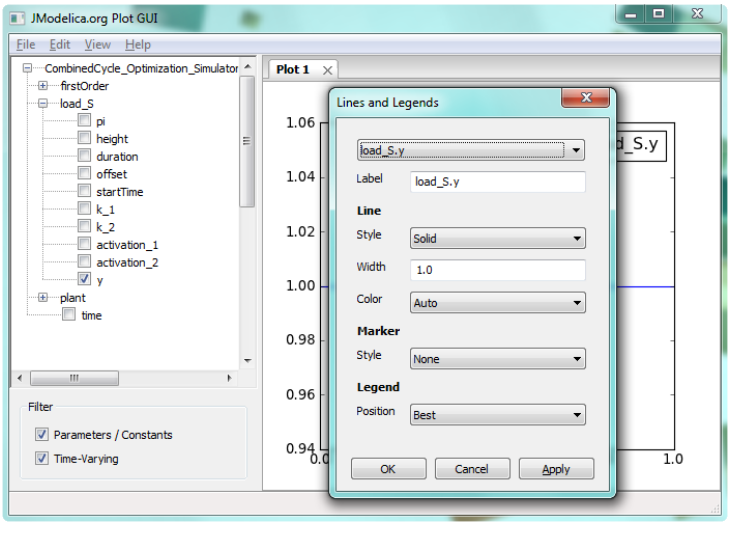


Figure 9.6 Figure Lines and Legends options.

Additional figures can be added from the `Add Plot` command in the `Edit` menu. In Figure 9.7 an additional figure have been added.

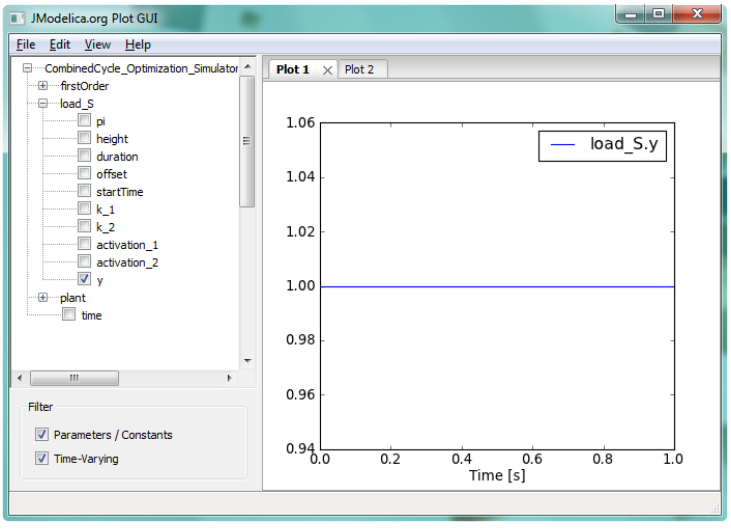


Figure 9.7 An additional plot has been added.

The figures can be positioned by choosing a figure tab and moving it to one of the borders of the GUI. In Figure 9.8 "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 9.9 the result is shown.

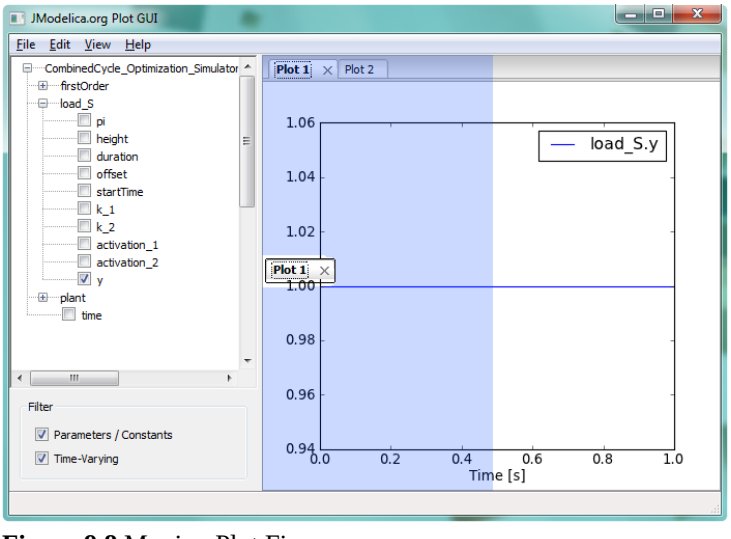


Figure 9.8 Moving Plot Figure.

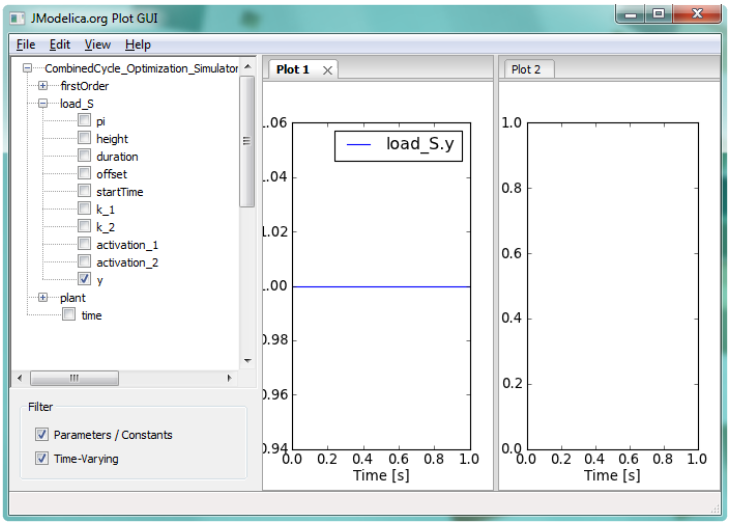


Figure 9.9 GUI after moving the plot window.

If we are to add more figures, an increasingly complex figure layout can be created as is shown in Figure 9.10 where figures also have been dragged to other figure headers.

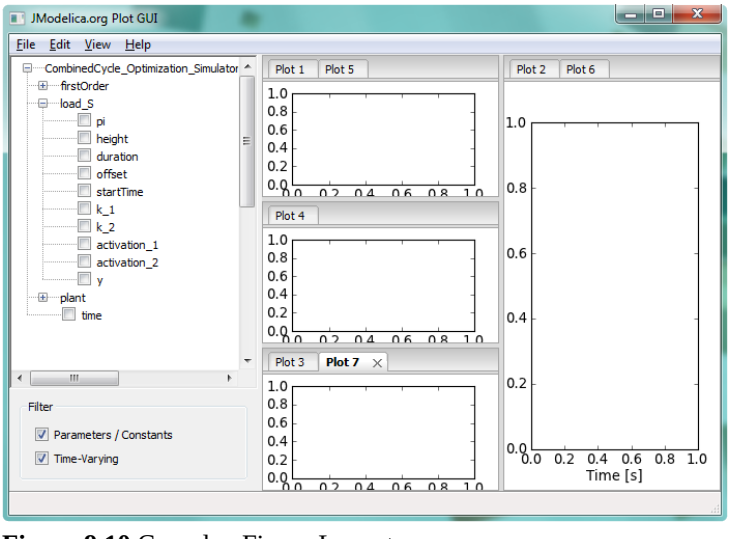


Figure 9.10 Complex Figure Layout.

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.

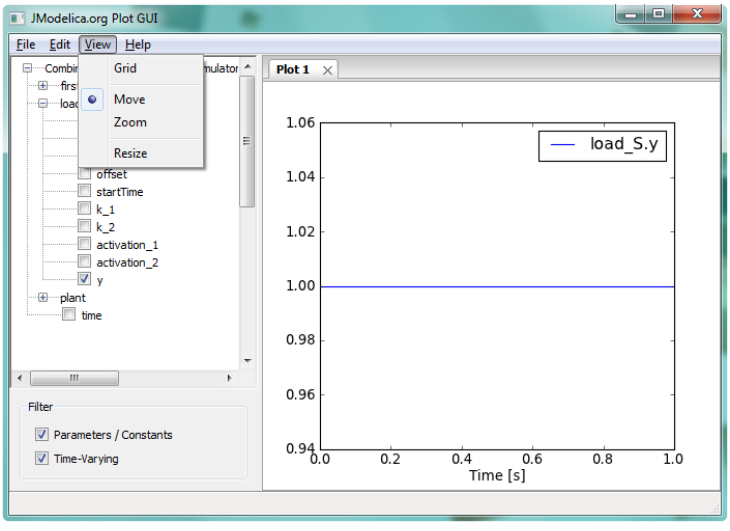


Figure 9.11 Figure View Options.

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.

1.4. Example

In Figure 9.12 an example showing 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.

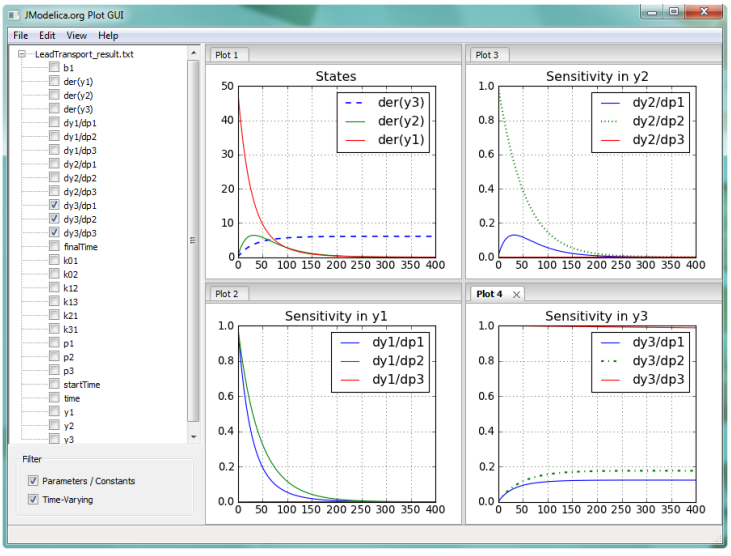


Figure 9.12 Multiple figure example.

Chapter 10. Optimica

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

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 \texttt{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.

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. Notice 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;
```

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. Notice 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.

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.

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:

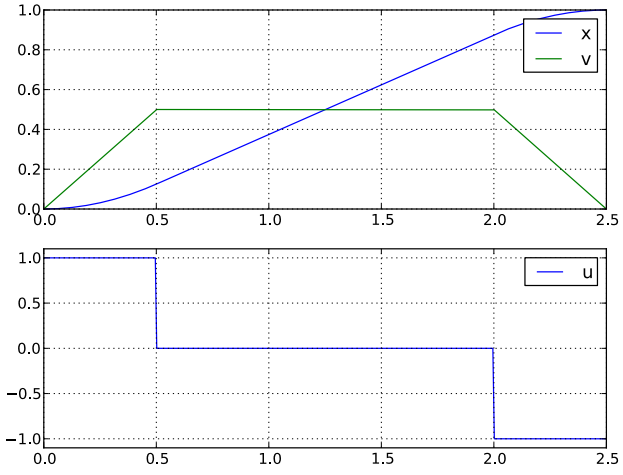


Figure 10.1 Optimization result

Chapter 11. Abstract syntax tree access

1. Tutorial on Abstract Syntax Trees (ASTs)

1.1. About Abstract Syntax Trees

A fundamental data structure in most compilers is the Abstract Syntax Tree (AST). An AST serves as an abstract representation of a computer program and is often used in a compiler to perform analyses (e.g., binding names to declarations and checking type correctness of a program) and as a basis for code generation.

Three different ASTs are used in the JModelica.org front-ends.

- The source AST results from parsing of the Modelica or Optimica source code. This AST shares the structure of the source code, and consists of a hierarchy consisting of Java objects corresponding to class and component declarations, equations and algorithms. The source AST can also be used for unparsing, i.e., pretty printing of the source code.

- The instance AST represents a particular model instance. Typically, the user selects a class to instantiate, and the compiler then computes the corresponding instance AST. The instance AST differs from the source AST in that in the former case, all components are expanded down to variables of primitive type. An important feature of the instance AST is that it is used to represent modification environments; merging of modifications takes place in the instance AST. As a consequence, all analysis, such as name and type analysis takes is done based on the instance AST.

- The flat AST represents the flat Modelica model. Once the instance AST has been computed, the flat AST is computed simply by traversing the instance AST and collecting all variables of primitive type, all equations and all algorithms. The flat AST is then used, after some transformations, as a basis for code generation.

For more information on how the JModelica.org compiler transforms these ASTs, see the paper "Implementation of a Modelica compiler using JastAdd attribute grammars" by J.Åkesson et. al.

This tutorial demonstrates how the Python interface to the three different ASTs in the compiler can be used. The JPype package is used to create Java objects in a Java Virtual Machine which is seamlessly integrated with the Python shell. The Java objects can be accessed interactively and methods of the object can be invoked.

For more information about the Java classes and their methods used in this example, please consult the API documentation for the Modelica compiler. Notice however that the documentation for the compiler front-ends is still very rudimentary. Also, the interfaces to the source and instance AST will be made more user friendly in upcoming versions.

Three different usages of ASTs are shown:

- Count the number of classes in the Modelica standard library. In this example, a Python function is defined to traverse the source AST which results from parsing of the Modelica standard library.
- Instantiate the CauerLowPassAnalog model. The instance AST for this model is dumped and it is demonstrated how the merged modification environments can be accessed. Also, it is shown how a component redeclaration affects the instance tree.
- Flatten the CauerLowPassAnalog model instance and print some statistics of the flattened Model.

The Python commands in this tutorial may be copied and pasted directly into a Python shell, in some cases with minor modifications. You are, however, strongly encouraged to copy the commands into a text file, e.g., ast_example.py.

Start the tutorial by creating a working directory and copy the file $JMODELICA_HOME/Python/jmodelica/examples/files/CauerLowPassAnalog.mo to your working directory. An on-line version of CauerLowPassAnalog.mo is also available (depending on which browser you use, you may have to accept the site certificate by clicking through a few steps). If you choose to create Python script file, save it to the working directory. The tutorial is based on a model from the Modelica Standard Library: Modelica.Electrical.Analog.Basic.Examples.CauerLowPassAnalog.

1.2. Load the Modelica standard library

Before we can start working with the ASTs, we need to import the Python packages that will be used

```
# Import library for path manipulations
import os.path

# Import the JModelica.org Python packages
import jmodelica
from jmodelica.compiler import ModelicaCompiler

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

# Import JPype
import jpype

# Create a reference to the java package 'org'
org = jpype.JPackage('org')
```

Also, we need to create an instance of a Modelica compiler in order to compile models:

```
# Create a compiler
mc = ModelicaCompiler()
```

In order to avoid parsing the same file multiple times (we will not change the Modelica file in this tutorial), we will check the variable 'source_root' exists in the shell before we parse the file CauerLowPassAnalog.mo:

```
# Don't parse the file if it har already been parsed.
try:
    source_root.getProgramRoot()
except:
    # Parse the file CauerLowPassAnalog.mo and get the root node
    # of the source AST
    source_root = mc.parse_model("CauerLowPassAnalog.mo")
```

At this point, try the built-in help feature of Python by typing the following command in the shell to see the help text for the function you just used.

```
In [2]: help(mc.parse_model)
```

In the first part of the tutorial, we will not work with the filter model, but rather load the Modelica standard library. Again, we check if the library has already been loaded:

```
# Don't load the standard library if it is already loaded
try:
    modelica.getName().getID()
except NameError, e:
    # Load the Modelica standard library and get the class
    # declaration AST node corresponding to the Modelica
    # package.
    modelica = source_root.getProgram().getLibNode(1). \
               getStoredDefinition().getElement(0)
```

The means to access the node in the source AST corresponding to the class (package) declaration of the Modelica library is somewhat cumbersome; the source AST interface will be improved in later versions.

1.3. Count the number of classes in the Modelica standard library

Having accessed a node in the source AST, we may now perform analysis by traversing the tree. Say that we are interested in counting the number of classes (packages, models, blocks, functions etc.) in the Modelica standard library. As the basis for traversing the AST, we may use the method ClassDecl.classes() that returns a list of local classes contained in a class. Based on this method, a Python function for traversing the class hierarchy of the source AST can be defined:

```
def count_classes(class_decl,depth):
    """ Count the number of classes hierarchically contained
    in a class declaration."""

    # Get a list of local classes using the method ClassDecl.classes()
    # which returns a Java ArrayList object containing ClassDecl objects.
    local_classes = class_decl.classes()

    # Get the number of local classes.
```

```
    num_classes = local_classes.size()

    # Loop over all local classes
    for i in range(local_classes.size()):
        # Call count_classes recursively for all local classes
        num_classes = num_classes + \
                      count_classes(local_classes.get(i),depth + 1)

    # If the class declaration corresponds to a package, print
    # the number of hierarchically contained classes
    if class_decl.getRestriction().getNodeName() == 'MPackage' \
           and depth <= 1:
        print("The package %s has %d hierachically contained classes" \
              %(class_decl.qualifiedName(),num_classes))

    # Return the number of hierachically contained classes
    return num_classes
```

We then call the function:

```
# Call count_classes for 'Modelica'
num_classes = count_classes(modelica,0)
```

Now run the script and study the printouts in the Python shell. The first time the script is run, you will se printouts corresponding also to the compiler accessing individual files of the Modelica standard library; the loading of the library is done on demand as the library classes are actually accessed. Run the script once again (using the '-i' switch), to get a cleaner output, which should now look similar to:

```
The package Modelica.UsersGuide has 16 hierachically contained classes
The package Modelica.Constants has 0 hierachically contained classes
The package Modelica.Icons has 16 hierachically contained classes
The package Modelica.SIunits has 532 hierachically contained classes
The package Modelica.StateGraph has 64 hierachically contained classes
The package Modelica.Blocks has 258 hierachically contained classes
The package Modelica.Electrical has 361 hierachically contained classes
The package Modelica.Math has 74 hierachically contained classes
The package Modelica.Mechanics has 474 hierachically contained classes
The package Modelica.Media has 1064 hierachically contained classes
The package Modelica.Thermal has 88 hierachically contained classes
The package Modelica.Utilities has 86 hierachically contained classes
The package Modelica has 3045 hierachically contained classes
```

Take some time to ponder the results and make sure that you understand how the Python function 'count_classes' works and which Python variables corresponds to references into the source AST.

1.4. Dump the instance AST

We shall now turn our attention to the CauerLowPassAnalog model. Specifically, we would like to analyze the instance hierarchy of the model by dumping the tree structure to the Python shell. In addition, we will look at the

merged modification environment of each instance AST node. Again, we will use methods defined for the Java objects representing the AST.

First we create an instance of the CauerLowPassAnalog filter. Again we only create the instance if it has not already been created:

```
# Don't instantiate if instance has been computed already
try:
    filter_instance.components()
except:
    # Retrieve the node in the instance tree corresponding to the class
    # Modelica.Electrical.Analog.Examples.CauerLowPassAnalog
    filter_instance = mc.instantiate_model(source_root,"CauerLowPassAnalog")
```

Next we define a Python function for traversing the instance AST and printing each node in the shell. We also print the merged modification environment for each instance node. In order to traverse the AST, we use the methods InstNode.instComponentDeclList() and InstNode.instExtendsList(), which both return an object of the class List, which in turn contain instantiated component declarations and instantiated extends clauses. By invoking the 'dump_inst_ast' function recursively for each element in these lists, the instance AST is in effect traversed. Due to the internal representation of the instance AST, nodes of type InstPrimitive, corresponding to primitive variables, are not leaves in the AST as would be expected. To overcome this complication, we simply check if a node is of type InstPrimitive, and if this is the case, the recursion stops.

The environment of an instance node is accessed by calling the method InstNode.getMergedEnvrionment(), which returns a list of modifications. According to the Modelica specification, outer modifications overrides inner modifications, and accordingly, modifications in the beginning of the list has precedence over later modifications.

```
def dump_inst_ast(inst_node, indent):
    """Pretty print an instance node, including its merged enviroment."""

    # Get the merged environment of an instance node
    env = inst_node.getMergedEnvironment()

    # Create a string containing the type and name of the instance node
    str = indent + inst_node.prettyPrint("")
    str = str + " {"

    # Loop over all elements in the merged modification environment
    for i in range(env.size()):
        str = str + env.get(i).toString()
        if i<env.size()-1:
            str = str + ", "
        str = str + "}"

    # Print
    print(str)

    # Get all components and dump them recursively
    components = inst_node.instComponentDeclList
```

```
    for i in range(components.getNumChild()):
        # Assume that primitive variables are leafs in the instance AST
        if (inst_node.getClass() is \
            org.jmodelica.modelica.compiler.InstPrimitive) is False:
            dump_inst_ast(components.getChild(i),indent + "  ")

    # Get all extends clauses and dump them recursively
    extends= inst_node.instExtendsList
    for i in range(extends.getNumChild()):
        # Assume that primitive variables are leafs in the instance AST
        if (inst_node.getClass() is \
            org.jmodelica.modelica.compiler.InstPrimitive) is False:
            dump_inst_ast(extends.getChild(i),indent + "  ")
```

Take a minute and make sure that you understand the essential parts of the function.

Having defined the function 'dump_inst_ast', we call it with the CauerLowPassAnalog instance as an argument.

```
# Dump the filter instance
dump_inst_ast(filter_instance,"")
```

You should now see a rather lengthy printout in your shell window. Let us have a closer look at a few of the instances in the model. First look at the printouts for a resistor in the model:

```
    InstComposite: Modelica.Electrical.Analog.Basic.Resistor R1 {R=1}
      InstPrimitive: SI.Resistance R {=1, start=1, final quantity="Resistance", \
                                      final unit="Ohm"}
      InstExtends: Interfaces.OnePort {R=1}
        InstPrimitive: SI.Voltage v {final quantity="ElectricPotential", final unit="V"}
        InstPrimitive: SI.Current i {final quantity="ElectricCurrent", final unit="A"}
        InstComposite: PositivePin p {}
          InstPrimitive: SI.Voltage v {final quantity="ElectricPotential", final unit="V"}
          InstPrimitive: SI.Current i {final quantity="ElectricCurrent", final unit="A"}
        InstComposite: NegativePin n {}
          InstPrimitive: SI.Voltage v {final quantity="ElectricPotential", final unit="V"}
          InstPrimitive: SI.Current i {final quantity="ElectricCurrent", final unit="A"}
```

The model instance if of type InstComposite, and contains two elements, one primitive variable, R, and one extends clause. The modification environment for the resistor contains a value modification '=1' and some modifications of the built in attributes for the type Real. The InstExtends node contains a number of child nodes, which corresponds to the content of the class Interfaces.OnePort. Notice the difference between the source AST, where an extends node is essentially a leaf in the tree, whereas in the instance tree, the extends clause is expanded.

Let us have a look at the effects of redeclarations in the instance AST. In the CauerLowPassAnalog model, a step voltage signal source is used, which in turn relies on redeclaration of a generic signal source to a step. The instance node for the step voltage source 'V' is given below:

```
    InstComposite: Modelica.Electrical.Analog.Sources.StepVoltage V {V=0, startTime=1, \
                                                                     offset=0}
```

```
      InstPrimitive: SI.Voltage V {=0, start=1, final quantity="ElectricPotential", \
                                   final unit="V"}
      InstExtends: Interfaces.VoltageSource {V=0, startTime=1, offset=0,
           redeclare Modelica.Blocks.Sources.Step signalSource(height=V)}
        InstPrimitive: SI.Voltage offset {=0, =0, final quantity="ElectricPotential", \
                                          final unit="V"}
        InstPrimitive: SI.Time startTime {=1, =0, final quantity="Time", final unit="s"}
        InstReplacingComposite: Modelica.Blocks.Sources.Step signalSource {height=V, \
                                          final offset=offset, final startTime=startTime}
          InstPrimitive: Real height {=V, =1}
          InstExtends: Interfaces.SignalSource {height=V, final offset=offset, \
                                                final startTime=startTime}
            InstPrimitive: Real offset {=offset, =0}
            InstPrimitive: SIunits.Time startTime {=startTime, =0, final quantity="Time", \
                                                   final unit="s"}
            InstExtends: SO {height=V, final offset=offset, final startTime=startTime}
              InstPrimitive: RealOutput y {}
              InstExtends: BlockIcon {height=V, final offset=offset,
                                      final startTime=startTime}
```

Here we see how the modification "redeclare Modelica.Blocks.Sources.Step signalSource(height=V)" affects the instance AST. The node InstReplacingComposite represents the component instance, instantiated from the class Modelica.Blocks.Sources.Step, resulting from the redeclaration. As a consequence, this branch of the instance AST is significantly altered by the redeclare modification.

Now look at the modification environment for the component instance startTime. The environment contains two value modifications: '=1' and '=0'. As noted above, the first modification in the list corresponds to the outermost modification and have precedence over the following modifications. Take a minute to figure out the origin of the modifications by looking upwards in the instance AST.

1.5. Flattening of the filter model

Having computed the instance, we can now flatten the model:

```
# Don't flatten model if it already exists
try:
    filter_flat_model.name()
except:
    # Flatten the model instance filter_instance
    filter_flat_model = mc.flatten_model(filter_instance)
```

During flattening, the instance tree is traversed and all primitive declarations and equations are collected. In addition, such as scalarization and elimination of alias variables are performed.

Let us have a look at the flattened model:

```
print(filter_flat_model)
```

We may also retrieve some model statistics:

```
print("*** Model statistics for CauerLowPassAnalog *** ")
print("Number of differentiated variables: %d" \
      % filter_flat_model.numDifferentiatedRealVariables())
print("Number of algebraic variables:      %d" \
      % filter_flat_model.numAlgebraicRealVariables())
print("Number of equations:                %d" \
      % filter_flat_model.numEquations())
print("Number of initial equations:        %d" \
      % filter_flat_model.numInitialEquations())
```

How many variables and equations is the model composed of? Does the model seem to be well posed?

At this point, take some time to explore the 'filter_flat_model' object by typing 'filter_flat_model.<tab>' in the Python shell to see what methods are available. You may also have a look in the Modelica compiler API.

Chapter 12. Limitations

This page lists the current limitations of the JModelica.org platform, as of version 1.5.0. The development of the platform can be followed at the Trac site, where future releases and associated features are planned. In order to get an idea of the current Modelica compliance of the compiler front-end, you may look at the associated test suite. All models with a test annotation can be flattened.

- The Modelica compliance of the front-end is limited; the following features are currently not supported:
  - Parsing of full Modelica 3.2 (Modelica 3.0 is supported)
  - Integer, Boolean and enumeration variables for optimizations (parameters and constants are supported, as are discrete variables in simulations)
  - Only partial support for enumeration variables in FMUs (parameters and constants are supported)
  - Strings
  - Arrays indexed with enumerations or Booleans
  - If equations are only supported with parameter or constant test expressions, and recompilation is required for a change of a parameter used in the test expression to take effect on the if equation.
  - Functions with array inputs with sizes declared as ':' are only partially supported.
  - Connect clauses does not handle arrays of connectors properly.
  - Partial support for external functions, only external C functions with scalar inputs and outputs are supported.
  - The following built-in functions are not supported:

sign(v)	cardinality()	reinit(x, expr)
semiLinear(...)	scalar(A)	String(...)
Subtask.decouple(v)	vector(A)	div(x,y)
delay(...)	matrix(A)	mod(x,y)
terminal()	diagonal(v)	rem(x,y)
sample(start, interval)	product(...)	ceil(x)
edge(b)	outerProduct(v1, v2)	floor(x)
skew(x)	symmetric(A)	integer(x)

  - Overloaded operators (Modelica Language Specification, chapter 14)

  - Stream connections with more than two connectors are not supported.
  - Mapping of models to execution environments (Modelica Language Specification, chapter 16)
- In the Optimica front-end the following constructs are not supported:
  - Annotations for transcription information
- The JModelica.org Model Interface (JMI) has the following Limitations:
  - The ODE interface requires the Modelica model to be written on explicit ODE form in order to work.
  - Second order derivatives (Hessians) are not provided
  - The interface does not yet comply with FMI specification
- The JModelica.org FMI Model Interface (FMI) has the following Limitations:
  - The FMI interface only supports FMUs distributed with binaries, not source code.
  - Options for setting and getting string variables does not work

Chapter 13. Release notes

1. Release notes for JModelica.org version 1.6

1.1. Highlights

- A new derivative free parameter optimization algorithm for FMUs
- A new pseudo spectral optimization algorithm
- Index reduction to handle high-index DAEs
- A new graphical user interface for plotting of simulation and optimization results
- Icon rendering and many improvements in the Eclipse Modelica plug-in

1.2. Compilers

1.2.1. Index reduction

High-index systems, commonly occurring in mechanical systems, are supported in JModelica.org 1.6. The implementation relies on Pantelides' algorithm and the dummy derivative selection algorithm.

1.2.2. Modelica compliance

The following improvements to the Modelica compliance of the editors has been made:

- Partial support for the `smooth()` operator (not used in event handling, otherwise supported).
- Support for global name lookup (i.e. names starting with a dot are looked up from the top scope).

1.3. Python

1.3.1. Graphical user interface for visualization of simulation and optimization results

A new graphical interface for displaying simulation and / or optimization results have been implemented. The interface also supports results generated from Dymola, both binary and textual.

1.3.2. Simulation with function inputs

The Python simulation interface has been improved so that top level inputs in FMUs can be driven by Python functions in addition to tables.

1.3.3. Compilation of XML models

A new convenience function for compilation of Modelica and Optimica models into XML, including equations, has been added.

1.3.4. Python version upgrade

The Python package has been updated to Python 2.7.

1.4. Optimization

1.4.1. Derivative- free optimization of FMUs

The derivative-free optimization algorithm in JModelica.org enables users to calibrate dynamic models compliant with the Functional Mock-up Interface standard (FMUs) using measurement data. The new functionality offers flexible and easy to use Python functions for model calibration and relies on the FMU simulation capabilities of JModelica.org. FMU models generated by JModelica.org or other FMI-compliant tools such as AMESim, Dymola, or SimulationX can be calibrated.

1.4.2. Pseudo spectral methods for dynamic optimization

Pseudo spectral optimization methods, based on collocation, are now available. The algorithms relies on CasADi for evaluation of derivatives, first and second order, and IPOPT is used to solve the resulting non-linear program. Optimization of ordinary differential equations and multi-phase problems are supported. The algorithm has been developed in collaboration with Mitsubishi Electric Research Lab, Boston, USA, where it has been used to solve satellite navigation problems.

1.5. Eclipse Modelica plugin

The JModelica.org Eclipse plugin has improved to the point where we are ready to do a release. Version 0.4.0 is now available from the JModelica.org website.

Changes from the versions that has been available from the SVN repository are mainly stability and performance improvements. To this end, some features have been disabled (auto-complete and format file/region). There are also a few new features, most notably support for rendering of class icons.

1.6. Contributors

Christian Andersson

Tove Bergdahl

Sofia Gedda

Magnus Gäfvert

Petter Lindgren

Fredrik Magnusson

Jesper Mattsson

Patrik Meijer

Lennart Moraeus

Kristina Olsson

Johan Ylikiiskilä

Johan Åkesson

1.6.1. Previous contributors

Philip Nilsson

Roberto Parrotto

Jens Rantil

Philip Reuterswärd

2. Release notes for JModelica.org version 1.5

2.1. Highlights

- FMU export
- Improvements in compiler front-end
- Equation sorting and BLT
- Symbolic solution of simple equations
- Improved simulation support for hybrid and sampled systems
- Improved initialization with Kinsol and SuperLU

- Improved support for external functions.

2.2. Compilers

2.2.1. When clauses

When clauses are supported in the Modelica compiler.

2.2.2. Equation sorting

Equations are sorted using Tarjan's algorithm and the resulting BLT representation is used in the C code generation. Also, trivial equations are solved and converted into assignment statements.

2.2.3. Connections

Added support for connecting arrays of components and for connect equations in for clauses.

2.2.4. Eclipse IDE

The JModelica plugin for Eclipse has been updated to be more stable and to syntax highlight Modelica 3.2 code properly.

2.2.5. Miscellaneous

Fixed several compiler bugs.

2.3. Simulation

2.3.1. FMU export

JModelica.org 1.5 supports export of Functional Mock-up Interface (FMI) compliant models (FMUs). The exported models follows the FMI standard and may be imported in other FMI compilant simulation tools, or they may be simulated using JModelica.org using the FMU import feature introduced in version 1.4. The exported FMUs contain an XML file, containing model meta data such as variable names, a DLL, containing the compiled C functions specified by FMI, and additional files containing the flattened Modelica model useful for debugging purposes.

2.3.2. Simulation of ODEs

A causalization approach to simulation of Modelica models has been implemented. This means that the DAE resulting from flattening is transformed into an ODE, and ODE solvers can be used to simulate the model. This feature is a requirement for export of FMUs. This strategy has required the symbolic algorithms and the C code generation module to be adapted as described above. In addition, the simulation runtime system has been extended

to allow for trivial equations converted into assignments and for implicit systems of equations. The latter are solved using the Newton solver KINSOL, modified to support regularization to handle singular Jacobian matrices.

2.3.3. Simulation of hybrid and sampled systems

When clauses are now supported, as well as the sample operator. Accordingly, some classes of hybrid systems may be simulated as well as sampled control systems. In addition, variables of type Integer and Boolean are also supported.

2.4. Initialization of DAEs

A novel initialization algorithm based on the Newton solver KINSOL from the SUNDIALS suite is introduced. The KINSOL solver has been improved by adding support for Jacobian regularization in order to handle singular Jacobians and by interfacing the sparse linear solver SuperLU in order to more efficiently handle large scale systems.

2.5. Optimization

Curtis Powell Reid seeding has been implemented to speed up computation of sparse Jacobians. When solving large optimization problems, this can give a speed-up factor of up to 10-15.

2.6. Contributors

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Jesper Mattsson

Johan Ylikiiskilä

Johan Åkesson

2.6.1. Previous contributors

Philip Nilsson

Roberto Parrotto

Jens Rantil

Philip Reuterswärd

3. Release notes for JModelica.org version 1.4

3.1. Highlights

- Improved Python user interaction functions
- Improvements in compiler front-end
- Support for sensitivity analysis of DAEs using Sundials
- Introduced new model concept, jmu-models.
- Support for enumerations

3.2. Compilers

3.2.1. Enumerations

Added support for enumerations to the same extent as Integers, except that arrays indexed with enumerations are not supported.

3.2.2. Miscellaneous

Fixed many compiler bugs, especially concerning complex class structures.

3.2.3. Improved reporting of structural singularities

Systems which are structurally singular now generates an error message. Also, high-index systems, which are not yet supported, are reported as structurally singular systems.

3.2.4. Automatic addition of initial equations

A matching algorithm is used to automatically add initial equations to obtain a balanced DAE initialization system. If too few initial equations are given, the algorithm will set the `fixed` attribute to true for some of the differentiated variables in the model.

3.3. Python interface

3.3.1. Models

- Introduced new model class `jmodelica.jmi.JMUModel` which replaced `jmodelica.jmi.JMIModel`.
- `jmodelica.fmi.FMIModel` changed name to `jmodelica.fmi.FMUModel`.

- `jmodelica.jmi.JMIModel.get_value` and `set_value` have changed to `jmodelica.jmi.JMUModel.get` and `set`, which have also been introduced for `jmodelica.fmi.FMUModel`

3.3.2. Compiling

- Introduced JMU files which are compressed files containing files created during compilation.
- Introduced new method `jmodelica.jmi.compile_jmu` which compiles Modelica or Optimica models to JMUs. These JMUs are then used when creating a `JMUModel` which loads the model in a Python object.
- Removed possibility to compile models directly in high-level functions, initialize, simulate and optimize. Instead `compile_jmu` should be used.

3.3.3. initialize, simulate and optimize

- `initialize`, `simulate` and `optimize` are no longer functions under `jmodelica` but methods of `jmodelica.jmi.JMUModel` and `jmodelica.fmi.FMUModel` (`initialize` and `simulate` only).
- New objects for options to `initialize`, `simulate` and `optimize` have been introduced. The `alg_args` and `solver_args` parameters have therefore been removed. The options from `alg_args` and `solver_args` can now be found in the options object. Each algorithm for `initialize`, `simulate` and `optimize` have their own options object.

3.3.4. Result object

Added convenience methods for getting variable trajectories from the result. The result trajectories are now accessed as objects in a dictionary:

```
res = model.simulate()
yres = res['y']
```

3.4. Simulation

3.4.1. Input trajectories

Changed how the input trajectories are handled. The trajectories now have to be connected to an input variable as a 2-tuple. The first argument should be a list of variables or a single variable. The second argument should be a data matrix with the first column as the time vector and the following columns corresponding to the variables in the first argument.

3.4.2. Sensitivity calculations

Sensitivity calculations have been implemented when using the solver IDA from the Assimulo package. The sensitivity calculations are activated with the the option:

opts['IDA_options']['sensitivity'] = True

which calculates sensitivities of the states with respect to the free parameters.

3.4.3. Write scaled simulation result to file

In some cases, it is useful to be able to write the scaled simulation result when the option `enable_variable_scaling` is set to true. Specifically, this supports debugging to detect if additional variables should have a nominal value. This feature is available also for initialization and optimization.

3.5. Contributors

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Jesper Mattsson

Johan Ylikiiskilä

Johan Åkesson

3.5.1. Previous contributors

Philip Nilsson

Roberto Parrotto

Jens Rantil

Philip Reuterswärd

4. Release notes for JModelica.org version 1.3

4.1. Highlights

- Functional Mockup Interface (FMI) simulation support
- Support for minimum time problems
- Improved support for redeclare/replaceable in the compiler frontend

- Limited support for external functions
- Support for stream connections (with up to two connectors in a connection)

4.2. Compilers

4.2.1. The Modelica compiler

Arrays

Slice operations are now supported.

Array support is now nearly complete. The exceptions are:

- Functions with array inputs with sizes declared as ':' - only basic support.
- A few array-related function-like operators are not supported.
- Connect clauses does not handle arrays of connectors properly.

Redecare

Redeclares as class elements are now supported.

Conditional components

Conditional components are now supported.

Constants and parameters

Function calls can now be used as binding expressions for parameters and constants. The handling of Integer, Boolean and record type parameters is also improved.

External functions

- Basic support for external functions written in C.
- Annotations for libraries, includes, library directories and include directories supported.
- Platform directories supported.
- Can not be used together with CppAD.
- Arrays as arguments are not yet supported. Functions in Modelica_utilies are also not supported.

Stream connectors

Stream connectors, including the operators inStream and actualStream and connections with up to two stream connectors are supported.

Miscellaneous

The error checking has been improved, eliminating many erroneous error messages for correct Modelica code.

The memory and time usage for the compiler has been greatly reduced for medium and large models, especially for complex class structures.

4.2.2. The Optimica compiler

All support mentioned for the Modelica compiler applies to the Optimica compiler as well.

New class attribute objectiveIntegrand

Support for the objectiveIntegrand class attribute. In order to encode Lagrange cost functions of the type

$$\int_{t_0}^{t_f} L(.)\ \mathrm{dt}$$

the Optimica class attribute `objectiveIntegrand` is supported by the Optimica compiler. The expression L may be utilized by optimization algorithms providing dedicated support for Lagrange cost functions.

Support for minimum time problems

Optimization problems with free initial and terminal times can now be solved by setting the free attribute of the class attributes startTime and finalTime to true. The Optimica compiler automatically translates the problem into a fixed horizon problems with free parameters for the start en terminal times, which in turn are used to rescale the time of the problem.

Using this method, no changes are required to the optimization algorithm, since a fixed horizon problem is solved.

4.3. JModelica.org Model Interface (JMI)

4.3.1. The collocation optimization algorithm

Dependent parameters

Support for free dependent parameters in the collocation optimization algorithm is now implemented. In models containing parameter declarations such as:

```
parameter Real p1(free=true);
```

```
parameter Real p2 = p1;
```

where the parameter p2 needs to be considered as being free in the optimization problem, with the additional equality constraint:

```
p1 = p2
```

included in the problem.

Support for Lagrange cost functions

The new Optimica class attribute objectiveIntegrand, see above, is supported by the collocation optimization algorithm. The integral cost is approximated by a Radau quadrature formula.

4.4. Assimulo

Support for simulation of an FMU (see below) using Assimulo. Simulation of an FMU can either be done by using the high-level method *simulate* or creating a model from the FMIModel class together with a problem class, FMIODE which is then passed to CVode.

4.5. FMI compliance

Improved support for the Functional Mockup Interface (FMI) standard. Support for importing an FMI model, FMU (Functional Mockup Unit). The import consist of loading the FMU into Python and connecting the models C execution interface to Python. Note, strings are not currently supported.

Imported FMUs can be simulated using the Assimulo package.

4.6. XML model export

4.6.1. `noEvent` operator

Support for the built-in operator `noEvent` has been implemented.

4.6.2. `static` attribute

Support for the Optimica attribute static has been implemented.

4.7. Python integration

4.7.1. High-level functions

Model files

Passing more than one model file to high-level functions supported.

New result object

A result object is used as return argument for all algorithms. The result object for each algorithm extends the base class `ResultBase` and will therefore (at least) contain: the model object, the result file name, the solver used and the result data object.

4.7.2. File I/O

Rewriting xmlparser.py has improved performance when writing simulation result data to file considerably.

4.8. Contributors

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Jesper Mattsson

Roberto Parrotto

Johan Åkesson

Philip Reuterswärd

4.8.1. Previous contributors

Philip Nilsson

Jens Rantil

5. Release notes for JModelica.org version 1.2

5.1. Highlights

- Vectors and user defined functions are supported by the Modelica and Optimica compilers
- New Python functions for easy initialization, simulation and optimization
- A new Python simulation package, Assimulo, has been integrated to provide increased flexibility and performance

5.2. Compilers

5.2.1. The Modelica compiler

Arrays

Arrays are now almost fully supported. This includes all arithmetic operations and use of arrays in all places allowed in the language specification. The only exception is slice operations, that are only supported for the last component in an access.

Function-like operators

Most function-like operators are now supported. The following list contains the function-like operators that are *not* supported:

- sign(v)
- Integer(e)
- String(...)
- div(x,y)
- mod(x,y)
- rem(x,y)
- ceil(x)
- floor(x)
- integer(x)
- delay(...)
- cardinality()
- semiLinear()
- Subtask.decouple(v)
- initial()
- terminal()
- smooth(p, expr)

- sample(start, interval)
- pre(y)
- edge(b)
- reinit(x, expr)
- scalar(A)
- vector(A)
- matrix(A)
- diagonal(v)
- product(...)
- outerProduct(v1, v2)
- symmetric(A)
- skew(x)

Functions and algorithms

Both algorithms and pure Modelica functions are supported, with a few exceptions:

- Use of control structures (if, for, etc.) with test or loop expressions with variability that is higher than parameter is not supported when compiling for CppAD.
- Indexes to arrays of records with variability that is higher than parameter is not supported when compiling for CppAD.
- Support for inputs to functions with one or more dimensions declared with ":" is only partial.

External functions are not supported.

Miscellaneous

- Record constructors are now supported.
- Limited support for constructs generating events. If expressions are supported.
- The noEvent operator is supported.

- The error checking has been expanded to cover more errors.
- Modelica compliance errors are reported for legal but unsupported language constructs.

5.2.2. The Optimica Compiler

All support mentioned for the Modelica compiler applies to the Optimica compiler as well.

5.3. The JModelica.org Model Interface (JMI)

5.3.1. General

Automatic scaling based on the `nominal` attribute

The Modelica attribute `nominal` can be used to scale variables. This is particularly important when solving optimization problems where poorly scaled systems may result in lack of convergence. Automatic scaling is turned off by default since it introduces a slight computational overhead: setting the compiler option `enable_variable_scaling` to `true` enables this feature.

Support for event indicator functions

Support for event indicator functions and switching functions are now provided. These features are used by the new simulation package Assimulo to simulate systems with events. Notice that limitations in the compiler front-end applies, see above.

Integer and boolean parameters

Support for event indicator functions and switching functions are now provided. These features are used by the new simulation package Assimulo to simulate systems with events. Notice that limitations in the compiler front-end applies, see above.

Linearization

A function for linearization of DAE models is provided. The linearized models are computed using automatic differentiation which gives results at machine precision. Also, for index-1 systems, linearized DAEs can be converted into linear ODE form suitable for e.g., control design.

5.4. The collocation optimization algorithm

5.4.1. Piecewise constant control signals

In control applications, in particular model predictive control, it is common to assume piecewise constant control variables, sometimes referred to as blocking factors. Blocking factors are now supported by the collocation-based optimization algorithm, see `jmodelica.examples.cstr_mpc` for an example.

5.4.2. Free initial conditions allowed

The restriction that all state initial conditions should be fixed has been relaxed in the optimization algorithm. This enables more flexible formulation of optimization problems.

5.4.3. Dens output of optimization result

Functions for retrieving the optimization result from the collocation-based algorithm in a dense format are now provided. Two options are available: either a user defined mesh is provided or the result is given for a user defined number of points inside each finite element. Interpolation of the collocation polynomials are used to obtain the dense output.

5.5. New simulation package: Assimulo

The simulation based on pySundials have been removed and replaced by the Assimulo package which is also using the Sundials solvers. The main difference between the two is that Assimulo is using Cython to connect to Sundials. This has substantially improved the simulation speed. For more info regarding Assimulo and its features, see: http://www.jmodelica.org/assimulo.

5.6. FMI compliance

The Functional Mockup Interface (FMI) standard is partially supported. FMI compliant model meta data XML document can be exported, support for the FMI C model execution interface is not yet supported.

5.7. XML model export

Models are now exported in XML format. The XML documents contain information on the set of variables, the equations, the user defined functions and for the Optimica´s optimization problems definition of the flattened model. Documents can be validated by a schema designed as an extension of the FMI XML schema.

5.8. Python integration

- The order of the non-named arguments for the ModelicaCompiler and OptimicaCompiler function `compile_model` has changed. In previous versions the arguments came in the order `(model_file_name, model_class_name, target = "model")` and is now `(model_class_name, model_file_name, target = "model")`.

- The functions `setparameter` and `getparameter` in `jmi.Model` have been removed. Instead the functions `set_value` and get_value (also in `jmi.Model`) should be used.

- Caching has been implemented in the xmlparser module to improve execution time for working with jmi.Model objects, which should be noticeable for large models.

5.8.1. New high-level functions for optimization and simulation

New high-level functions for problem initialization, optimization and simulation have been added which wrap the compilation of a model, creation of a model object, setup and running of an initialization/optimization/simulation and returning of a result in one function call. For each function there is an algorithm implemented which will be used by default but there is also the possibility to add custom algorithms. All examples in the example package have been updated to use the high-level functions.

5.9. Contributors

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Jesper Mattsson

Philip Nilsson

Roberto Parrotto

Philip Reuterswärd

Johan Åkesson

5.9.1. Previous contributors

Jens Rantil

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 Årén, Magnus Gäfvert , , and . *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.

Index

C
CppAD, 2

I
IPOPT, 2

J
JastAdd, 1
JMI, 3
    (see also JModelica Model Interface)
JModelica Model Interface (see JMI)

M
Modelica, 1

O
Optimica, 1

X
XML, 3
XPATH, 3