JModelica.org User Guide

Version 1.17

JModelica.org User Guide: Version 1.17

Publication date 2015-12-08

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

List of Figures

List of Tables

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. 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 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 extension Optimica. Optimica enables users to conveniently formulate optimization problems based on Modelica models using simple but powerful constructs for encoding of optimization interval, cost function and constraints.

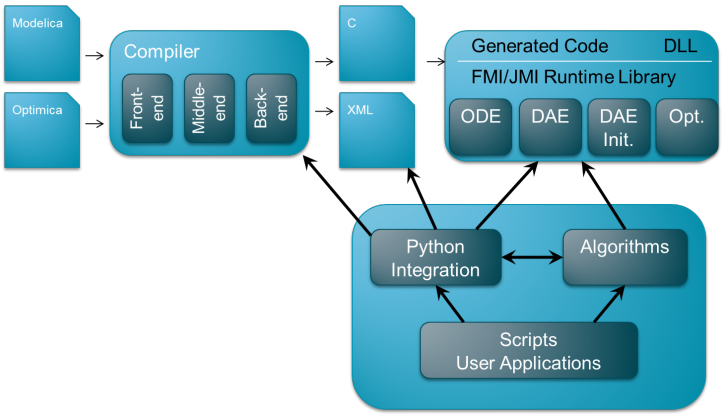
The JModelica.org compilers are developed in the compiler construction framework JastAdd. JastAdd is based on a number of different concepts, including object-orientation, aspect-orientation and reference attributed grammars. Compilers developed in JastAdd are specified in terms of declarative attributes 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 seamlessly with Python and Numpy.

JModelica.org offers strong support for the Functional Mock-up Interface (FMI) standard. FMI specifies a format for exchange of compiled dynamic models and it is supported by a large number of modeling and simulation tools, including established Modelica tools such as Dymola, OpenModelica, and SimulationX. 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', these zip-files containing the models are called FMUs (Functional Mock-up Units). FMI version 1.0 specifies two types of FMUs, either Model Exchange or Co-Simulation. The difference between them is that in a Co-Simulation FMU, the integrator for solving the system is contained in the model while in an Model Exchange FMU, an external integrator is needed to solve the system. The JModelica.org compiler supports export of FMUs and FMUs can be imported into Python using the Python packages included in the platform.

4. Architecture

Figure 1.1. JModelica.org 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. Export of Functional Mock-up Units (FMUs) is supported. There is also the option to export flattened Modelica models, including equations, in XML format.

- 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 to provide derivatives for optimization algorithm. The runtime system also contains the functions provided in the FMI API.

- Currently, JModelica.org features four different algorithms for solving dynamic optimization problems. There are three different algorithms based on direct collocation, which rely on the solver IPOPT for obtaining a solution to the resulting NLP. The default algorithm is encoded in C and relies on CppAD for computing the NLP derivatives. The other two algorithms are developed in Python and rely on CasADi for computing derivatives. There is also a derivative free optimization algorithm for model calibration based on measurement data that is applicable to FMUs.

- JModelica.org uses Python for scripting. For this purpose, JModelica.org provides a number of different Python packages. The Assimulo package provides integration with state of the art DAE and ODE solvers (including the SUNDIALS suite), PyFMI provides FMU import, whereas PyModelica interacts with the JModelica.org compilers. Finally, PyJMI contains drivers for the optimization algorithms. All packages are available as part of JModelica.org, and Assimulo and PyFMI are also available as stand alone Python packages from www.assimulo.org and www.pyfmi.org.

5. Extensibility

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

- The JModelica.org platform supports export and import of FMUs, which are compliant with the FMI standard. In addition, 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. FMI is the default format for simulation, whereas JMI is the default interface for optimization.

- In addition to the FMI and JMI interfaces, JModelica.org supports export of flat Modelica models in XML format. This format is based on FMI, and is suitable for integration with symbolic algorithms that can exploit access to the equations in symbolic form.
- 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 is supported on Linux and Windows (Vista, 7) with 32-bit or 64-bit architectures.

2. Installation on Windows

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

The Windows installer contains a binary distribution of JModelica.org, bundled with all required third-party software components. A list of the third-party dependencies can be found in Section 2.1, “Dependencies”. The installer sets up a pre-configured complete environment with convenient start menu shortcuts. Installation instructions are found in Section 2.2, “Installation”.

2.1. Dependencies

As of JModelica.org version 1.9, all dependencies are bundled in the installer. They are listed below, each with version number (where applicable) and link to corresponding web site.

- **Applications**
  - Java 1.7 (JRE)
  - MinGW (gcc 4.7.2)
  - Python 2.7
- **Libraries**
  - Ipopt 3.10.3
  - SuperLU 4.1
  - Beaver 0.9.6.1
  - CppAD
  - eXpat 2.1.0
  - Minizip
  - MSL (Modelica Standard Library)
  - SUNDIALS 2.4.0
  - Zlib 1.2.6
  - CasADi
- **Python packages**
  - Cython 0.18
  - Distribute 0.6.35
  - IPython 0.13.1
  - JCC 1.18

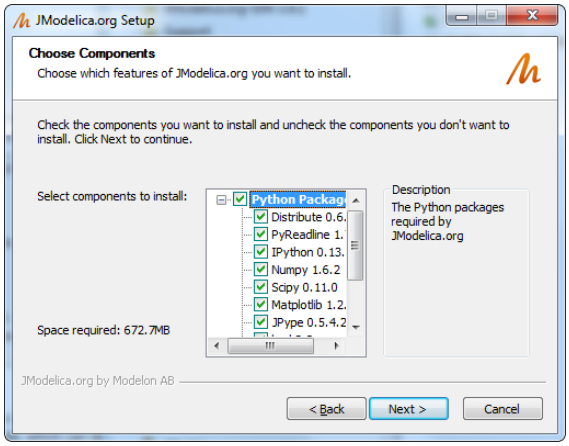
- JPype 0.5.4.2
- lxml 3.1.0
- matplotlib 1.2.0
- nose 1.2.1
- NumPy 1.6.2
- Pyreadline 1.7.1
- SciPy 0.11.0
- wxPython 2.8

2.2. Installation

Follow these step-by-step instructions to install JModelica.org using the Windows binary distribution.

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.



2.3. Verifying the installation

Test the installation by starting a IPython or pylab shell from the JModelica.org start menu and run a few examples. Starting the Python session from the Windows start menu will set all the environment variables required to run the JModelica.org Python interface.

```
# Import and run the VDP_sim example and plot results
from pyjmi.examples import VDP_sim
VDP_sim.run_demo()

# Import and run the CSTR example and plot results
from pyjmi.examples import cstr
cstr.run_demo()

# Import and run the CSTR example using CasADi and plot results
```

```
from pyjmi.examples import cstr_casadi
cstr_casadi.run_demo()
```

2.4. Compilation from sources

For compiling JModelica.org from sources on Windows there is a Software Development Kit (SDK) available for download. The SDK is a bundle of tools used to build JModelica.org from source code on Windows, please see the SDK User's guide, which can be reached from the download site, for more information.

3. Installation on Linux systems

This section describes a procedure for compiling JModelica.org from sources on Linux. The instructions have been verified to work on Ubuntu Linux release 12.04, 64bit.

3.1. Prerequisites

3.1.1. Installing pre-compiled packages

It is convenient to use a package management system, if available, of the Linux distribution to install the prerequisites. On Ubuntu systems, the apt-get command line program may be used:

```
sudo apt-get -y install g++
sudo apt-get -y install subversion
sudo apt-get -y install gfortran
sudo apt-get -y install ipython
sudo apt-get -y install cmake
sudo apt-get -y install swig
sudo apt-get -y install ant
sudo apt-get -y install openjdk-6-jdk
sudo apt-get -y install python-dev
sudo apt-get -y install python-numpy
sudo apt-get -y install python-scipy
sudo apt-get -y install python-matplotlib
sudo apt-get -y install cython
sudo apt-get -y install python-lxml
sudo apt-get -y install python-nose
sudo apt-get -y install python-jpype
sudo apt-get -y install zlib1g-dev
sudo apt-get -y install libboost-dev
```

On Ubuntu 12.04, the bundled jcc version is too old. A new enough version can be installed using pip:

```
sudo apt-get -y install python-pip
sudo pip install jcc
```

The following versions of each package have been tested and verified to work. Please note that in some cases, a minimum version is required.

Table 2.1. Package versions for Ubuntu

Package	Version	Note
`g++`	`4.6.3`	Tested version
`subversion`	`1.6.17`	Tested version
`gfortran`	`4.6.3`	Tested version
`ipython`	`0.12.1`	Tested version
`cmake`	`2.8.6`	Minimum version
`swig`	`2.0.4`	Tested version
`ant`	`1.8.2`	Tested version
`python-dev`	`2.7.3`	Tested version

Package	Version	Note
python-numpy	1.6.1	Tested version
python-scipy	0.9.0	Tested version
python-matplotlib	1.1.1	Tested version
cython	0.15	Minimum version
python-lxml	2.3.2	Tested version
python-nose	1.1.2	Tested version
python-jpype	0.5.4.2	Tested version
zlib1g-dev	1:1.2.3.4	Tested version
libboost-dev	1.48.0.2	Tested version
jcc	2.16	Minimum version

3.1.2. Compiling Ipopt

While Ipopt is available as a pre-compiled package for Ubuntu, it is recommended to build Ipopt from sources. The Ipopt packages provided for Ubuntu have had flaws (including the version provided for Ubuntu 12.04) that prevented usage with JModelica.org. Also, compiling Ipopt from sources is required when using the linear solvers MA27 or MA57 from the HSL library, since these are not available as open source software.

First, download the Ipopt sources from https://projects.coin-or.org/Ipopt and unpack the content:

```
tar xvf Ipopt-3.10.2.tgz
```

Then, retrieve the third party dependencies:

```
cd Ipopt-3.10.2/ThirdParty/Blas
./get.Blas
cd ../Lapack
./get.Lapack
cd ../Mumps
./get.Mumps
cd ../Metis
./get.Metis
cd ../../
```

If you have access to the HSL codes MA57 or MA27, copy their sources into the directory `ThirdParty/HSL`. In the next step, configure and compile Ipopt:

```
mkdir build
cd build
../configure --prefix=/home/<user_name>/<ipopt_installation_location>
make install
```

where `<user_name>` and `<ipopt_installation_location>` are replaced by the user directory and the installation directory of choice for Ipopt.

3.1.3. Installing JModelica.org with WORHP (optional)

As an alternative to IPOPT for optimization, the CasADi framework in JModelica.org also has support for the solver WORHP. Note that WORHP is closed source, but offers free personal academic licenses. To compile JModelica.org with support for WORHP, first obtain the WORHP binaries and a license file from http://www.worhp.de. Set the environment variables `$WORHP` to your directory containing the binaries and `$WORHP_LICENSE_FILE` to your license file.

Normally, this would be sufficient, but for now the following additional measures are needed. Find the following six lines in `$JMODELICA_SRC/ThirdParty/CasADi/CasADi/interface/worhp/worhp_internal.cpp` and remove them:

```
addOption("CutLength",OT_REAL,worhp_p_.CutLength,"Scaling factor for Cut recovery strategy");
```

```
addOption("Ma57PivotThresh",OT_REAL,worhp_p_.Ma57PivotThresh,"Pivoting tolerance for MA57 = CNTL(1)");
if (hasSetOption("CutLength")) worhp_p_.CutLength = getOption("CutLength");
if (hasSetOption("Ma57PivotThresh")) worhp_p_.Ma57PivotThresh = getOption("Ma57PivotThresh");
setOption("CutLength",worhp_p_.CutLength);
setOption("Ma57PivotThresh",worhp_p_.Ma57PivotThresh);
```

Find the line

```
option(WITH_WORHP "Compile the WORHP interface" OFF)
```

in `$JMODELICA_SRC/ThirdParty/CasADi/CasADi/CMakeLists.txt` and change OFF to ON.

3.2. Compiling

Make sure that all prerequisites are installed before compiling the JModelica.org platform. First, check out the JModelica.org sources:

```
svn co https://svn.jmodelica.org/trunk JModelica.org
```

Then configure and build JModelica.org:

```
cd JModelica.org
mkdir build
cd build
../configure --prefix=/home/<user_name>/<jmodelica_install_location> \
             --with-ipopt=/home/<user_name>/<ipopt_install_location>
make install
make casadi_interface
```

where `<user_name>` and `<jmodelica_installation_location>` are replaced by the user directory and the installation directory of choice for JModelica.org.

3.3. Testing JModelica.org

In order to verify that JModelica.org has been installed correctly, start an IPython shell using the command `/home/<user_name>/<jmodelica_install_location>/bin/jm_ipython` and run a few examples:

```
# Import and run the VDP_sim example and plot results
from pyjmi.examples import VDP_sim
VDP_sim.run_demo()

# Import and run the CSTR example and plot results
from pyjmi.examples import cstr
cstr.run_demo()

# Import and run the CSTR example using CasADi and plot results
from pyjmi.examples import cstr_casadi
cstr_casadi.run_demo()
```

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** Interface to the compilers. Compile Modelica and Optimica code into model units, FMUs. See Chapter 4, *Working with Models* for more information.
- **PyFMI** Work with models that have been compiled into FMUs (Functional Mock-up Units), perform simulations, parameter manipulation, plot results etc. See Chapter 5, *Simulation of FMUs* for more information.
- **PyJMI** Work with models that are represented in symbolic form based no the automatic differentiation tool CasADi. This package is mainly used for solving optimization problems. See Chapter 6, *Optimization* for more information.

2. Starting a Python session

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

2.1. Windows

If you are on Windows, there are three different Python shells 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.

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

2.2. Linux

To start the IPython shell with pylab on Linux open a terminal and enter the command:

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

3. Running an example

The Python packages `pyfmi` and `pyjmi` each contain a folder called `examples` in which there are several Python example scripts. The scripts demonstrate compilation, loading and simulation or optimization of models. The corresponding model files are located in the subdirectory `files`. The following code demonstrates how to run such an example. First a Python session must be started, see Section 2, “Starting a Python session” 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. Checking your installation

The JModelica.org Python packages require some third-party Python packages in order to run. 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 pyjmi
pyjmi.check_packages()

Performing pymodelica/pyjmi package check
=========================================

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

Python version:.............. 2.7.3

pymodelica/pyjmi version:..... 1.12


Dependencies:

Package                      Version
-------                      -------
assimulo..................... --
casadi....................... 1.7.0
Cython....................... 0.18
jpype........................ --
lxml......................... 3.1.0
matplotlib................... 1.2.0
nose......................... 1.2.1
numpy........................ 1.6.2
scipy........................ 0.11.0
wxPython..................... 2.8.12.1
pyreadline................... 1.7.1
setuptools................... 0.6
```

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 show how to redefine the JModelica.org environment variable `IPOPT_HOME`:

1. Go to the folder `$USERPROFILE` (Windows) or `$HOME` (Linux). To find out where `$USERPROFILE` or `$HOME` points to, open a Python shell and type:

   ```
   import os
   os.environ['USERPROFILE']   // Windows
   ```

```
os.environ['HOME']          // Linux
```

2. Create a folder and name it `.jmodelica.org` (or open it if it already exists)
3. In this folder, create a text file and name it `user_startup.py`.
4. Open the file and type

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

5. Save and close.
6. 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 or the Modelica language. You can search in old threads to see if someone has asked your question before or start a new thread if you are a registered user.

Chapter 4. Working with Models

1. Introduction to models

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

1.1. The different model objects in JModelica.org

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

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

`OptimizationProblem` objects for CasADi optimization do not currently have a corresponding file format, but are transferred directly from the JModelica.org compiler, based on Modelica and Optimica models. They contain a symbolic representation of the optimization problem, which is used with the automatic differentiation tool CasADi for optimization purposes. Read more about CasADi and how a `OptimizationProblem` object can be used for optimization in Section 5, “Dynamic optimization of DAEs using direct collocation with CasADi” in Chapter 6, *Optimization*.

2. Compilation

This section brings up how to compile a model to an FMU-ME / FMU-CS. Compiling a model to an FMU-ME / FMU-CS will be demonstrated in Section 2.1, “Simple FMU-ME compilation example” and Section 2.2, “Simple FMU-CS compilation example” respectively.

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

Section 2.6, “Compilation in more detail”, will go through some parts of the compilation process and how to perform these steps one by one.

2.1. Simple FMU-ME compilation example

The following steps compile a model to an FMU-ME version 1.0:

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

This is demonstrated in the following code example:

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

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

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

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

Once compilation has completed successfully, an FMU-ME 1.0 will have been created on the file system. The FMU is essentially a compressed file archive containing the files created during compilation that are needed when instantiating a model object. Return argument for `compile_fmu` is the full file path of the FMU that has just been created, this will be useful later when we want to create model objects. More about the FMU and loading models can be found in Section 3, “Loading models”.

In the above example, the model is compiled using default arguments and compiler options - the only arguments set are the model class and file name. However, `compile_fmu` has several other named arguments which can be modified. The different arguments, their default values and interpretation will be explained in Section 2.4, “Compiler settings”.

2.2. Simple FMU-CS compilation example

The following steps compiles a model to an FMU-CS version 1.0:

1. Import the JModelica.org compiler function `compile_fmu` from the package `pymodelica`.
2. Specify the model and model file.
3. Set the argument `target = 'cs'`
4. Perform the compilation.

This is demonstrated in the following code example:

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

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

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

In a Co-Simulation FMU, the integrator for solving the system is contained in the model. With an FMU-CS exported with JModelica.org, two different solvers are supported: CVode and Explicit Euler.

2.3. Compiling from libraries

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

- Adding the directory containing the library directory to the environment variable `MODELICAPATH`. The compiler will search for classes in all libraries found in any on the directories in `MODELICAPATH`. In this case the `file_name` argument of the compilation function can be omitted, assuming no additional Modelica files are needed.
- Setting the `'extra_lib_dirs'` compiler option to the path to the directory containing the library directory. This is equivalent to adding it to the `MODELICAPATH`, but only for that compilation.
- Giving the path to the library directory in the `file_name` argument of the compilation function. This allows adding a specific library to the search list (as opposed to adding all libraries in a specific directory).

By default, the script starting a JModelica.org Python shell sets the `MODELICAPATH` to the directory containing the version of the Modelica Standard Library (MSL) that is included in the installation. Thus, all classes in the MSL are available without any need to specify its location.

The Python code example below demonstrates these methods:

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

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

# Compile a model from the library MyLibrary, located in C:\MyLibs
fmu2 = compile_fmu('MyLibrary.MyModel', compiler_options = {'extra_lib_dirs':'C:/MyLibs'})

# The same as the last command, if no other libraries in C:\MyLibs are needed
fmu3 = compile_fmu('MyLibrary.MyModel', 'C:/MyLibs/MyLibrary')
```

2.4. Compiler settings

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

2.4.1. compile_fmu arguments

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

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

compile_fmu(class_name, file_name=[], compiler='auto', target='me', version='1.0',
            compiler_options={}, compile_to='.', compiler_log_level='warning',
            separate_process=True, jvm_args='')
Compile a Modelica model to an FMU.

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

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

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

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:\MyLibs1','c:\MyLibs2']}

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

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

Parameters::

    class_name --
```

```
        The name of the model class.

    file_name --
        A path (string) or paths (list of strings) to model files and/or
        libraries.
        Default: Empty list.

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

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

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

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

    compile_to --
        Specify target file or directory. If file, any intermediate directories
        will be created if they don't exist. If directory, the path given must
        exist.
        Default: Current directory.

    compiler_log_level --
        Set the logging for the compiler. Takes a comma separated list with
        log outputs. Log outputs start with a flag :'warning'/'w',
        'error'/'e', 'info'/'i' or 'debug'/'d'. The log can be written to file
        by appended flag with a colon and file name.
        Default: 'warning'

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

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


Returns::

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

2.4.2. Compiler options

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

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

# Import the compiler function
```

```
from pymodelica import compile_fmu

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

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

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

2.5. Compiling in a separate process

In JModelica.org, the compilers (`ModelicaCompiler` and `OptimicaCompiler`) are written in Java. When compiling a model from the Python interface, with e.g. `compile_fmu`, the default behavior is to compile the model in a separate process. This means that a specific JRE (Java Runtime Environment) is used for the compilation. For those on a 64 bit Windows this is can be very useful as the default JRE used with JPype is 32 bit. Also, in most cases, the JVM (Java Virtual Machine) can be given a larger heap space (especially when using a 64 bit JRE instead of a 32 bit) which enables compilation of larger models.

The environment variable `SEPARATE_PROCESS_JVM` can be set to point at a specific Java installation (JRE or JDK) for the compilation. For Windows users, the environment variable can be found (and set) in the file `setenv.bat` which is located in the JModelica.org installation folder. It can also be set locally in the Python shell. If `SEPARATE_PROCESS_JVM` is not set, `JAVA_HOME` will be used instead. It is also possible to pass arguments to the JVM with the `compile_fmu` argument '`jvm_args`'.

The following example demonstrates how to set the maximum heap space for the JVM to one gigabyte by setting the argument `jvm_args`:

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

# Compile in separate process
compile_fmu('myPackage.myModel', 'myModelFile.mo', jvm_args='-Xmx1g')
```

Another option is to access the compilers through the Python package *JPype* (this used to be the default behavior). This option is still available and can be ebabled by setting the argument `separate_process` to `False` when calling e.g. `compile_fmu`.

2.6. Compilation in more detail

Compiling with `compile_fmu` bundles quite a few steps required for the compilation from model file to FMU. Some of these steps will be briefly described in this section with code examples. For a more detailed review of the compile procedure, see Section 4, “Architecture” in Chapter 1, *Introduction*.

2.6.1. Creating a compiler

A compiler (which can be either a Modelica or Optimica compiler) is created by importing the Python classes from the compiler module. This example code will create a Modelica compiler and a target object.

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

# Create a compiler and compiler target object for FMU-ME version 1.0
mc = ModelicaCompiler()
target = mc.create_target_object("me", "1.0")
```

2.6.2. 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.

Note that the default setting for the compiler is to compile an FMU.

```
# Parse the model and get a reference to the root of the source AST
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', target)

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

2.6.3. Code generation

The next step is code generation, which produces C code containing the model equations, and XML files containing model meta data such as variable names and types.

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

3. Loading models

Compiled models, FMUs, are loaded in the JModelica.org Python interface with the `FMUModel(ME/CS)` class from the `pyfmi` module, while optimization problems for the CasADi-based optimization are transferred directly into the `OptimizationProblem` class from the `pyjmi` module . This will be demonstrated in Section 3.2, “Loading an FMU”, ??? and Section 3.3, “Transferring an OptimizationProblem”.

The model classes contain many methods with which models can be manipulated after instantiation. Amongst the most important methods are `initialize` and `simulate`, which are used when simulating. These are explained in Chapter 5, *Simulation of FMUs* and Chapter 6, *Optimization*. For more information on how to use the OptimizationProblem for optimization purposes, see Chapter 6, *Optimization*. The more basic methods for variable and parameter manipulation are explained in Section 4, “Changing model parameters”.

3.1. 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`.

There are two types of FMUs, Model Exchange and Co-Simulation. In a Co-Simulation FMU, the integrator for solving the system is contained in the model while in an Model Exchange FMU, an external integrator is needed to solve the system. JModelica.org supports export and import of FMU-ME and FMU-CS version 1.0 and 2.0. The solvers supported for FMU-CS export are CVode and Explicit Euler.

3.2. Loading an FMU

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

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

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

3.3. Transferring an OptimizationProblem

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

```
# Import transfer_optimization_problem
from pyjmi import transfer_optimization_problem

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

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

4. Changing model parameters

Model parameters can be altered with methods in the model classes once the model has been loaded. Some short examples in Section 4.1, “Setting and getting parameters” will demonstrate this.

4.1. Setting and getting parameters

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

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

```
# Compile and load the model
from pymodelica import compile_fmu
from pyfmi import load_fmu
my_fmu = compile_fmu('RLC_Circuit','RLC_Circuit.mo')
rlc_circuit = load_fmu(my_fmu)

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

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

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

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

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

5. Debugging models

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

5.1. Compiler logging

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

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

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

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

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

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

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

5.2. Runtime logging

5.2.1. Setting log level

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

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

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

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

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

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

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

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

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

5.2.2. Interpreting logs from FMUs produced by JModelica.org

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

```
...
FMIL: module = FMICAPI, log level = 5: Calling fmiInitialize
FMIL: module = Model, log level = 4: [INFO][FMU status:OK] <EquationSolve>Model equations evaluation invo
FMIL: module = Model, log level = 4: [INFO][FMU status:OK]   <BlockEventIterations>Starting block (local)
FMIL: module = Model, log level = 4: [INFO][FMU status:OK]     <vector name="ivs">        0.0000000000000
FMIL: module = Model, log level = 4: [INFO][FMU status:OK]     <vector name="switches">        0.00000000
FMIL: module = Model, log level = 4: [INFO][FMU status:OK]     <vector name="booleans"></vector>
FMIL: module = Model, log level = 4: [INFO][FMU status:OK]     <BlockIteration>Local iteration<value name
FMIL: module = Model, log level = 4: [INFO][FMU status:OK]       <JacobianUpdated><value name="block">0</
FMIL: module = Model, log level = 4: [INFO][FMU status:OK]         <matrix name="jacobian">
FMIL: module = Model, log level = 4: [INFO][FMU status:OK]                  -1.0000000000000000E+00,
FMIL: module = Model, log level = 4: [INFO][FMU status:OK]                  -1.0000000000000000E+00,
FMIL: module = Model, log level = 4: [INFO][FMU status:OK]                  -1.0000000000000000E+00,
FMIL: module = Model, log level = 4: [INFO][FMU status:OK]         </matrix>
FMIL: module = Model, log level = 4: [INFO][FMU status:OK]       </JacobianUpdated>
...
```

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

```
# Extract the log file XML contents into a pure XML file
```

```
pyjmi.log.extract_jmi_log(dest_xml_file_name, log_file_name)
```

The XML contents in the log file can also be parsed directly:

```
# Parse the entire XML log
log = pyjmi.log.parse_jmi_log(log_file_name)
```

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

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

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

```
# Parse the entire XML log
log = pyjmi.log.parse_jmi_log(log_file_name)
# Gather information pertaining to equation solves
solves = pyjmi.log.gather_solves(log)

print
print 'Number of solver invocations:',                      len(solves)
print 'Time of first solve:',                               solves[0].t
print 'Number of block solves in first solver invocation:', len(solves[0].block_solves)
print 'Names of iteration variables in first block solve:', solves[0].block_solves[0].variables
print 'Min bounds in first block solve:',                   solves[0].block_solves[0].min
print 'Max bounds in first block solve:',                   solves[0].block_solves[0].max
print 'Initial residual scaling in first block solve:',     solves[0].block_solves[0].initial_residual_sc
print 'Number of iterations in first block solve:',         len(solves[0].block_solves[0].iterations)
print
print 'First iteration in first block solve: '
print '  Iteration variables:',              solves[0].block_solves[0].iterations[0].ivs
print '  Scaled residuals:',                 solves[0].block_solves[0].iterations[0].residuals
print '  Jacobian:\n',                       solves[0].block_solves[0].iterations[0].jacobian
print '  Jacobian updated in iteration:',    solves[0].block_solves[0].iterations[0].jacobian_updated
print '  Residual scaling factors:',         solves[0].block_solves[0].iterations[0].residual_scaling
print '  Residual scaling factors_updated:', solves[0].block_solves[0].iterations[0].residual_scaling_upd
print '  Scaled residual norm:',             solves[0].block_solves[0].iterations[0].scaled_residual_norm
```

5.3. Getting HTML diagnostics

By setting the compiler option `generate_html_diagnostics` to true, a number of HTML pages containing diagnostics are generated. The HTML files are generated in the directory `Model_Name_diagnostics`, where `Model_Name` is the name of the compiled model. As compared to the diagnostics generated by the `compiler_log_level` argument, the HTML diagnostics contains only the most important information, but it also provides a better overview. Opening the file `Model_Name_diagnostics/index.html` in a web browser, results in a page similar to the one shown below.

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

Problems:
0 errors, 0 compliance errors, 1 warnings

Model before transformation

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 :                         53
  Number of Real variables:                   53
  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:      8
Number of Real derivative variables:          0
Number of Real algebraic variables:           45
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:                          51
Number of variables with binding expression:  4
  Number of Real variables with binding exp:  4
  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:                    55
Number of initial equations:                  0
Number of relational exps in equations:       1
Number of relational exps in init equations:  0
Flattened model

Connection sets

Model after transformation

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:             34
  Number of Real independent parameters:      19
  Number of Integer independent parameters:   2
  Number of Enum independent parameters:      4
  Number of Boolean independent parameters:   9
  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 :                         28
  Number of Real variables:                   28
  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:           20
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:                          24
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:                    24
Number of initial equations:                  4
Number of relational exps in equations:       1
Number of relational exps in init equations:  0
Transformed model

Alias sets (13 sets, 40 eliminated variables)

BLT diagnostics
BLT diagnostics table

Number of unsolved equation blocks in DAE initialization system: 1: {4}
Number of unsolved equation blocks in DAE system: 1: {4}

Number of unsolved equation blocks in DAE initialization system after tearing: 1: {1}
Number of unsolved equation blocks in DAE system after tearing: 1: {1}
```

Note that some of the entries, including `Problems`, `Flattened model`, `Connection sets`, `Transformed model`, `Alias sets`, `BLT diagnostics table` 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 7 ---
Solved block of 1 variables:
Computed variable:
  inertia2.flange_b.tau
Solution:
   - (  - ( damper.flange_b.tau ) ) - (  - ( spring.flange_b.tau ) ) + 0

--- Block 8 (Unsolvable block 0) ---
Non-solved linear block of 4 variables:
Coefficient variability: Parameter
Unknown variables:
  inertia2.a
```

```
  idealGear.flange_b.tau
  idealGear.flange_a.tau
  inertia1.a
Equations:
  inertia1.a = ( idealGear.ratio ) * ( inertia2.a )
  ( 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 )

...
```

Additionally there is a table view of the `BLT`. It can be found on the `BLT diagnostics table` page. It provides a graphical representation of the `BLT`. The view is limited to 300 equations due to the complexity of the graph.

Chapter 5. Simulation of FMUs

1. Introduction

JModelica.org supports simulation of models described in the Modelica language and models following the FMI standard, see Section 3, “Technology” in Chapter 1, *Introduction*. The simulation environment uses Assimulo as standard which is a standalone Python package for solving ordinary differential and differential algebraic equations. Loading and simulation of FMUs has additionally been made available as a separate Python package, PyFMI.

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

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 load_fmu method
from pymodelica import compile_fmu
from pyfmi import load_fmu

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

Next, we compile and load the model:

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

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

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

```
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']
```

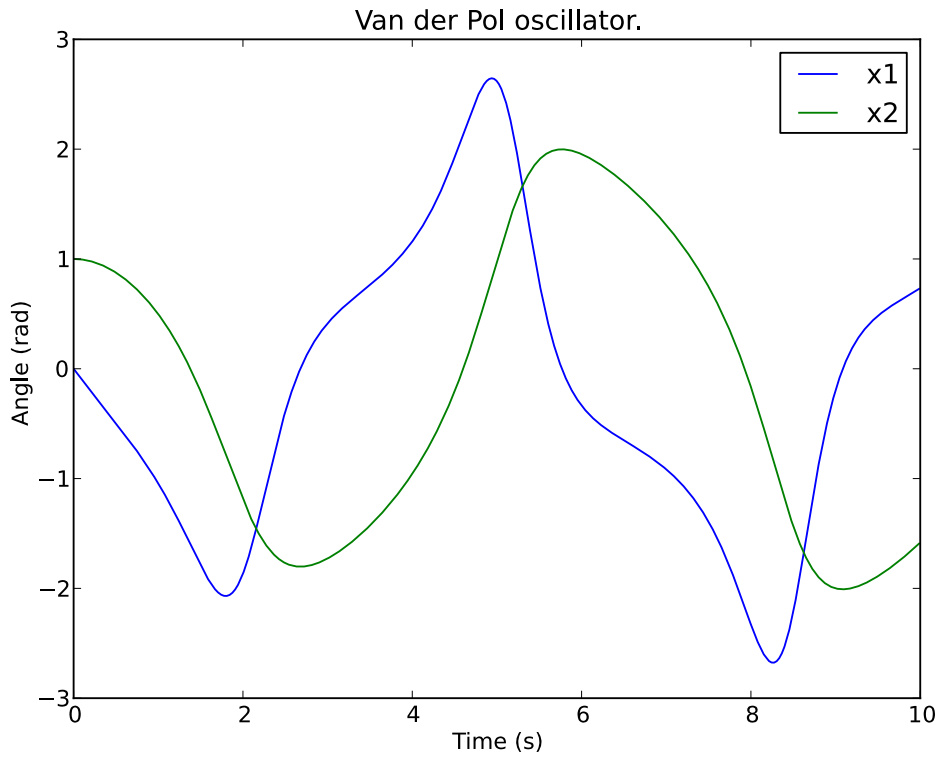
```
t  = res['time']
```

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 5.1, “Simulation result of the Van der Pol oscillator.” the simulation result is shown.

Figure 5.1. Simulation result of the Van der Pol oscillator.



3. Simulation of Models

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

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

`FMUModelME* / FMUModelCS*` also supports compiled models from other simulation/modelling tools that follow the FMI standard (extension .fmu) (either Model exchange FMUs or Co-Simulation FMUs). The support is both for FMI version 1.0 and FMI version 2.0. For more information about compiling a model in JModelica.org see Chapter 4, *Working with Models*.

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

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

And used in the following way:

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

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

3.1. Convenience method, load_fmu

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

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

3.2. Arguments

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

3.2.1. 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.

```
import numpy as N
t = N.linspace(0.,10.,100)           # Create one hundred evenly spaced points
u = N.sin(t)                         # Create the input vector
u_traj = N.transpose(N.vstack((t,u))) # Create the data matrix and transpose
                                     # it to the correct form
```

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

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

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

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

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

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

Note that the variables are now a List of variables.

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

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

and for the double input case:

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

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

3.2.2. Options for FMUModelME1 and FMUModelME2

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

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

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

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

   ...
```

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

Table 5.1. General options for AssimuloFMIAlg.

Option	Default	Description
solver	'CVode'	Specifies the simulation method that is to be used. Currently supported solvers are, CVode, Radau5ODE, RungeKutta34, Dopri5, RodasODE, LSODAR, ExplicitEuler. Although the recommended solver is "CVode".
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.
filter	None	A filter for choosing which variables to actually store result for. The syntax can be found here. An example is filter = "*der" , store all variables ending with 'der' and filter = ["*der*", "summary*"], store all variables with "der" in the name and all variables starting with "summary".
result_handling	"file"	Specifies how the result should be handled. Either stored to file or stored in memory. One can also use a custom handler. Available options: "file", "memory", "custom"

Lets look at an example, consider that you want to simulate 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               # Don't initialize the model
model.simulate(options=opts)             # Pass in the options to simulate and simulate
```

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

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

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

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

Table 5.2. Selection of solver arguments for CVode

Option	Default	Description
discr	'BDF'	The discretization method. Can be either 'BDF' or 'Adams'
iter	'Newton'	The iteration method. Can be either 'Newton' or 'FixedPoint'.
maxord	5	The maximum order used. Maximum for 'BDF' is 5 while for the 'Adams' method the maximum is 12
maxh	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.
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

3.2.3. Options for FMUModelCS1 and FMUModelCS2

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

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

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

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

   ...
```

In Table 5.3, “General options for FMICSAlg.” the general options for the FMICSAlg algorithm are described.

Table 5.3. General options for FMICSAlg.

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

3.3. Return argument

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

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

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

Table 5.4. Result Object

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

4. Examples

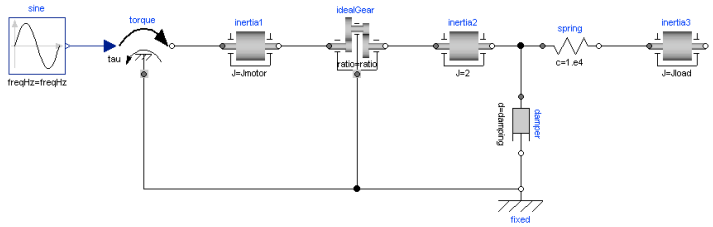
In the next sections, it will be shown how to use the JModelica.org platform for simulation of various FMUs.

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

4.1. Simulation of a high-index model

Mechanical component-based models often result in high-index DAEs. In order to efficiently integrate such models, Modelica tools typically employs an index reduction scheme, where some equations are differentiated, and dummy derivatives are selected. In order to demonstrate this feature, we consider the model `Modelica.Mechanics.Rotational.Examples.First` from the Modelica Standard library, see Figure 5.2, “Modelica.Mechanics.Rotational.First connection diagram”. The model is of high index since there are two rotating inertias connected with a rigid gear.

Figure 5.2. Modelica.Mechanics.Rotational.First connection diagram



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

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

```
from pyfmi import load_fmu
```

Next, the model is compiled and loaded:

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

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

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

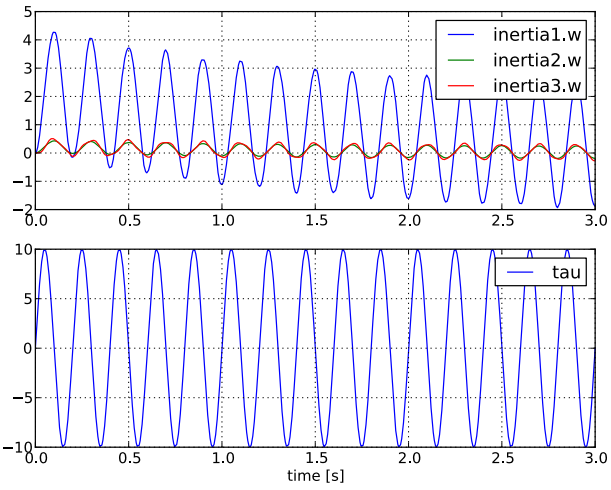
```
# Load result file
res = model.simulate(final_time=3.)
```

Finally, the simulation results are retrieved and plotted:

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

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

You should now see a plot as shown below.

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



4.2. Simulation and parameter sweeps

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

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

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

    // The control signal
    input Real u;

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

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

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

into the Python file. Compile and load the model:

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

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

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

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

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

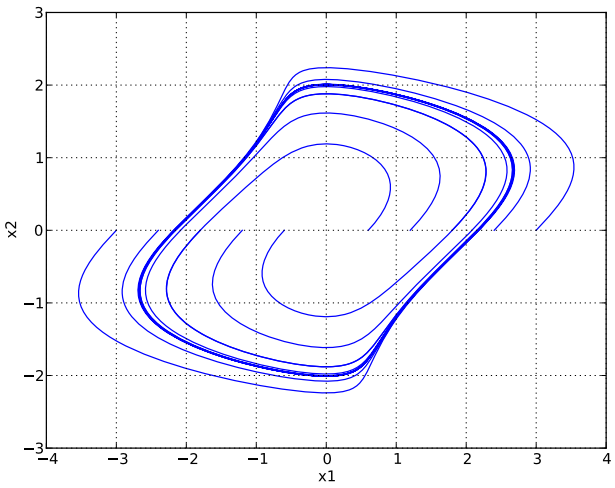
```
fig = P.figure()
P.clf()
P.hold(True)
P.xlabel('x1')
P.ylabel('x2')
```

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

You should now see a plot similar to that in Figure 5.4, “Simulation result-phase plane”.

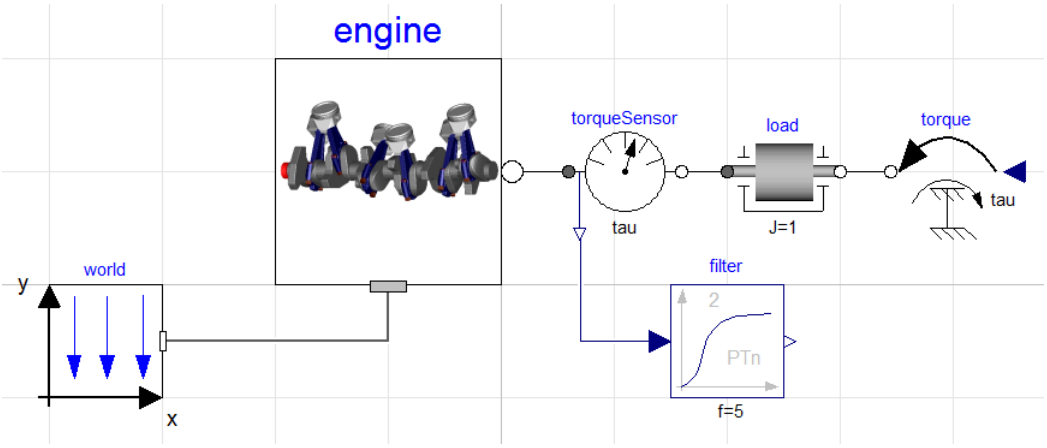
Figure 5.4. Simulation result-phase plane



4.3. Simulation of an Engine model with inputs

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

Figure 5.5. Overview of the Engine model



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

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

  input Real u;

  import Modelica.Mechanics.*;

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

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

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

equation
  torque.tau = u;

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

end EngineV6_analytic_with_input;
```

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

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

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

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

model = load_fmu(name)
```

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

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

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

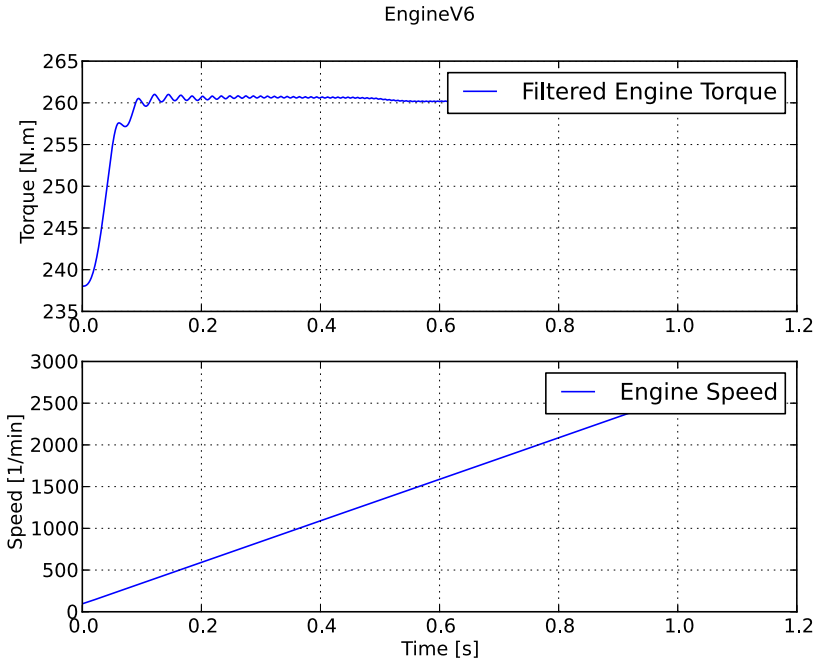
```
res = model.simulate(options=opts)
```

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

```
P.subplot(211)
P.suptitle("EngineV6")
P.plot(res["time"],res["filteredEngineTorque"], label="Filtered Engine Torque")
P.grid()
P.legend()
P.ylabel("Torque [N.m]")
P.subplot(212)
P.plot(res["time"],res["engineSpeed_rpm"], label="Engine Speed")
P.grid()
P.legend()
P.xlabel("Time [s]")
P.ylabel("Speed [1/min]")
P.show()
```

In Figure 5.6, “Resulting trajectories for the engine model.” the trajectories are shown.

Figure 5.6. Resulting trajectories for the engine model.



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

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

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

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

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

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

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

4.4. Simulation using the native FMI interface

This example shows how to use the native JModelica.org FMI interface for simulation of an FMU (FMI 1.0 for Model Exchange). 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 (https://www.fmi-standard.org/). The bouncing ball model is to be simulated using the explicit Euler method with event detection.

The example can also be found in the Python examples catalog in the 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.

4.4.1. Implementation

Start by importing the necessary modules,

```
import numpy as N
import pylab as P                  # Used for plotting
from pyfmi.fmi import FMUModelME1  # The FMI Interface for Model Exchange
```

Next, the FMU is to be loaded and initialized

```
# Load the FMU by specifying the fmu together with the path.
bouncing_fmu = FMUModelME1('/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, `FMUModelME1` 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_variable_valueref('h')] + \
        [bouncing_fmu.get_variable_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 depends on inputs which can be set using the `set_(real/...)` methods.

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

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

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

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

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

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

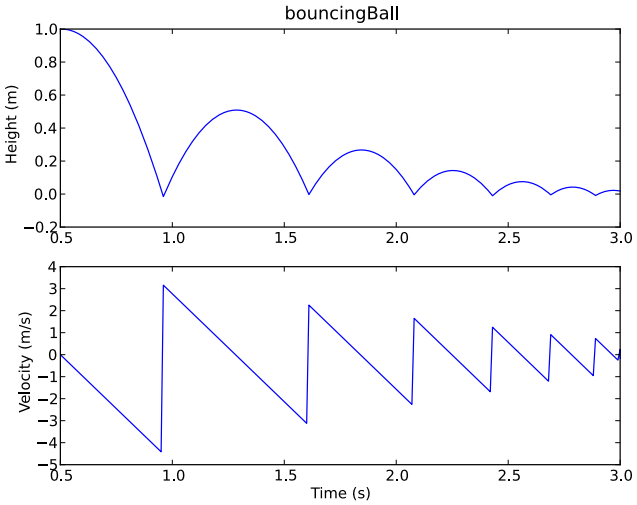
In the end of the loop, the solution is stored and the old event indicators are stored for use in the next loop.

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 5.7. Simulation result



4.5. Simulation of Co-Simulation FMUs

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

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

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

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

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

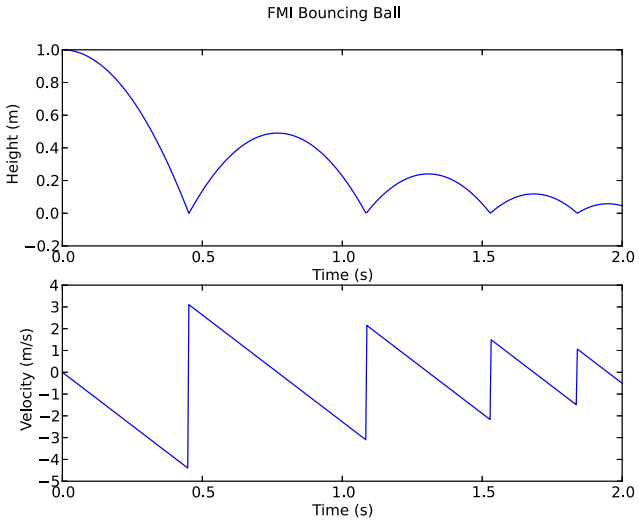
```
res = model.simulate(final_time=2.)
```

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

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

and the figure below shows the results.

Figure 5.8. Simulation result



Chapter 6. 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. 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, “A first example” and in Section 3, “Solving optimization problems”, the details of how the optimization algorithms are invoked are explained. The following sections contain tutorial exercises that illustrates how to set up and solve different kinds of optimization problems.

When formulating optimization problems, models are expressed in the Modelica language, whereas optimization specifications are given in the Optimica extension which is described in Chapter 8, *Optimica*. 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 (objectiveIntegrand = x1^2 + x2^2 + u^2,
                      startTime = 0,
                      finalTime = 20)

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

  // The control signal
  input Real u;

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

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

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

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

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

Next, we transfer the model:

```
# Transfer the optimization problem to casadi
 op = transfer_optimization_problem("VDP_pack.VDP_Opt2", "VDP_Opt.mop")
```

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

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

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

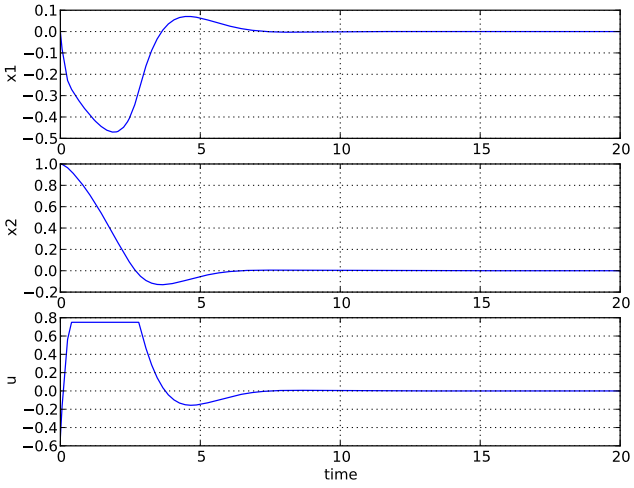
```
# Extract variable profiles
x1=res['x1']
x2=res['x2']
u=res['u']
t=res['time']
```

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

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

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

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

You should now see the optimization result as shown in Figure 6.1, “Optimal profiles for the VDP oscillator”.

Figure 6.1. Optimal profiles for the VDP oscillator



Optimal control and state profiles for the Van Der Pol optimal control problem.

3. Solving optimization problems

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

- **Dynamic optimization of DAEs using direct collocation with CasADi.** This algorithm is the default algorithm for solving optimal control and parameter estimation problems. It is implemented in Python, uses CasADi for

computing function derivatives and the nonlinear programming solvers IPOPT or WORHP for solving the resulting NLP. In terms of functionality, this algorithm is very similar to the deprecated JMU-based algorithm, but offers significant performance improvements in several regards. Use this method if your model is a DAE and does not contain discontinuities.

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

- **Dynamic optimization of DAEs using direct collocation with JMUs (Deprecated in JModelica.org 1.15)**. This algorithm is implemented in C, uses CppAD for computing function derivatives and IPOPT for solving the resulting nonlinear program (NLP).

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

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

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

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

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

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

    Parameters::

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

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

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

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

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

```
              LocalDAECollocationAlgOptions for LocalDAECollocationAlg.
            Default: Empty dict

    Returns::

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

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

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

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

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

    Parameters::

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

    Returns::

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

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

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

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

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

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

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

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

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

4. Scaling

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

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

Here, the `nominal` attribute is used to specify that the variable pressure takes on values which are on the order of 1e5. In order to use `nominal` attributes for scaling with CasADi-based algorithms, scaling is enabled by setting the algorithm option `variable_scaling` to `True`, and is enabled by default . When scaling is enabled, all variables

with a set nominal attribute are then scaled by dividing the variable value with its nominal value, i.e., from an algorithm point of view, all variables should take on values close to one. Notice that variables typically vary during a simulation or optimization and that it is therefore not possible to obtain perfect scaling. In order to ensure that model equations are fulfilled, each occurrence of a variable is multiplied with its nominal value in equations. For example, the equation:

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

is replaced by the equation

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

when `variable scaling` is enabled.

The algorithm in Section 5, “Dynamic optimization of DAEs using direct collocation with CasADi” also has support for providing trajectories (obtained by for example simulation) that are used for scaling. This means that it usually is not necessary to provide nominal values for all variables, and that it is possible to use time-varying scaling factors.

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

5. Dynamic optimization of DAEs using direct collocation with CasADi

5.1. Algorithm overview

The direct collocation method 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 nonlinear programming solvers IPOPT and WORHP can be used to solve the nonlinear program resulting from collocation. The needed first- and second-order derivatives are obtained using CasADi by algorithmic differentiation.

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

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

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

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

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

The standard options for the algorithm are shown in Table 6.1, “Standard options for the CasADi- and collocation-based optimization algorithm”. Additional documentation is available in the Python class documentation.

The algorithm also has a lot of experimental options, which are not as well tested and some are intended for debugging purposes. These are shown in Table 6.2, “Experimental and debugging options for the CasADi- and collocation-based optimization algorithm”, and caution is advised when changing their default values.

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

Option	Default	Description
`n_e`	50	Number of finite elements.
`hs`	None	Element lengths. Possible values: None, iterable of floats and "free" None: The element lengths are uniformly distributed. iterable of floats: Component i of the iterable specifies the length of element i. The lengths must be normalized in the sense that the sum of all lengths must be equal to 1. "free": The element lengths become optimization variables and are optimized according to the algorithm option free_element_lengths_data. WARNING: The "free" option is very experimental and will not always give desirable results.
`n_cp`	3	Number of collocation points in each element.
`expand_to_sx`	"NLP"	Whether to expand the CasADi MX graphs to SX graphs. Possible values: "NLP", "DAE", "no". "NLP": The entire NLP graph is expanded into SX. This will lead to high evaluation speed and high memory consumption. "DAE": The DAE, objective and constraint graphs for the dynamic optimization problem expressions are expanded into SX, but the full NLP graph is an MX graph. This will lead to moderate evaluation speed and moderate memory consumption. "no": All constructed graphs are MX graphs. This will lead to low evaluation speed and low memory consumption.
`init_traj`	None	Variable trajectory data used for initialization of the NLP variables.
`nominal_traj`	None	Variable trajectory data used for scaling of the NLP variables. This option is only applicable if variable scaling is enabled.
`blocking_factors`	None (not used)	Blocking factors are used to enforce piecewise constant inputs. The inputs may only change values at some of the element boundaries. The option is either None (disabled), given as an instance of pyjmi.optimization.casadi_collocation.BlockingFactors or as a list of blocking factors. If the options is a list of blocking factors, then each element in the list specifies the number of collocation elements for which all of the inputs must be constant. For example, if blocking_factors == [2, 2, 1], then the inputs will attain 3 different values (number of elements in the list), and it will change values between collocation element number 2 and 3 as well as number 4 and 5. The sum of all elements in the list must be the same as the number of collocation elements and the length of the list determines the number of separate values that the inputs may attain. See the documentation of the BlockingFactors class for how to use it. If blocking_factors

Option	Default	Description
		is None, then the usual collocation polynomials are instead used to represent the controls.
`external_data`	None	Data used to penalize, constrain or eliminate certain variables.
`delayed_feedback`	None	If not `None`, should be a `dict` with mappings `'delayed_var': ('undelayed_var', delay_ne)`. For each key-value pair, adds the the constraint that the variable `'delayed_var'` equals the value of the variable `'undelayed_var'` delayed by `delay_ne` elements. The initial part of the trajectory for `'delayed_var'` is fixed to its initial guess given by the `init_traj` option or the `initialGuess` attribute. `'delayed_var'` will typically be an input. This is an experimental feature and is subject to change.
`solver`	'IPOPT'	Specifies the nonlinear programming solver to be used. Possible choices are 'IPOPT' and 'WORHP'.
`IPOPT_options`	IPOPT defaults	IPOPT options for solution of NLP. See IPOPT's documentation for available options.
`WORHP_options`	WORHP defaults	WORHP options for solution of NLP. See WORHP's documentation for available options.

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

Option	Default	Description
`free_element_lengths_data`	None	Data used for optimizing the element lengths if they are free. Should be None when hs != "free".
`discr`	'LGR'	Determines the collocation scheme used to discretize the problem. Possible values: "LG" and "LGR". "LG": Gauss collocation (Legendre-Gauss) "LGR": Radau collocation (Legendre-Gauss-Radau).
`named_vars`	False	If enabled, the solver will create a duplicated set of NLP variables which have names corresponding to the Modelica/Optimica variable names. Symbolic expressions of the NLP consisting of the named variables can then be obtained using the get_named_var_expr method of the collocator class. This option is only intended for investigative purposes.
`init_dual`	None	Dictionary containing vectors of initial guess for NLP dual variables. Intended to be obtained as the solution of an optimization problem which has an identical structure, which is stored in the dual_opt attribute of the result object. The dictionary has two keys, 'g' and 'x', containing vectors of the corresponding dual variable intial guesses. Note that when using IPOPT, the option warm_start_init_point has to be activated for this option to have an effect.
`variable_scaling`	True	Whether to scale the variables according to their nominal values or the trajectories provided with the nominal_traj option.
`equation_scaling`	False	Whether to scale the equations in collocated NLP. Many NLP solvers default to scaling the equations, but

Option	Default	Description
		if it is done through this option the resulting scaling can be inspected.
nominal_traj_mode	{"_default_mode": "linear"}	Mode for computing scaling factors based on nominal trajectories. Four possible modes: "attribute": Time-invariant, linear scaling based on Nominal attribute "linear": Time-invariant, linear scaling "affine": Time-invariant, affine scaling "time-variant": Time-variant, linear scaling Option is a dictionary with variable names as keys and corresponding scaling modes as values. For all variables not occuring in the keys of the dictionary, the mode specified by the "_default_mode" entry will be used, which by default is "linear".
result_file_name	""	Specifies the name of the file where the result is written. Setting this option to an empty string results in a default file name that is based on the name of the model class.
write_scaled_result	False	Return the scaled optimization result if set to True, otherwise return the unscaled optimization result. This option is only applicable when variable_scaling is enabled and is only intended for debugging.
print_condition_numbers	False	Prints the condition numbers of the Jacobian of the constraints and of the simplified KKT matrix at the initial and optimal points. Note that this is only feasible for very small problems.
result_mode	'collocation_points'	Specifies the output format of the optimization result. Possible values: "collocation_points", "element_interpolation" and "mesh_points" "collocation_points": The optimization result is given at the collocation points as well as the start and final time point. "element_interpolation": The values of the variable trajectories are calculated by evaluating the collocation polynomials. The algorithm option n_eval_points is used to specify the evaluation points within each finite element. "mesh_points": The optimization result is given at the mesh points.
n_eval_points	20	The number of evaluation points used in each element when the algorithm option result_mode is set to "element_interpolation". One evaluation point is placed at each element end-point (hence the option value must be at least 2) and the rest are distributed uniformly.
checkpoint	False	If checkpoint is set to True, transcribed NLP is built with packed MX functions. Instead of calling the DAE residual function, the collocation equation function, and the lagrange term function n_e * n_cp times, the check point scheme builds an MXFunction evaluating n_cp collocation points at the same time, so that the packed MXFunction is called only n_e times. This approach improves the code generation and it is expected to reduce the memory usage for constructing and solving the NLP.
quadrature_constraint	True	Whether to use quadrature continuity constraints. This option is only applicable when using Gauss colloca-

Option	Default	Description
		tion. It is incompatible with eliminate_der_var set to True. True: Quadrature is used to get the values of the states at the mesh points. False: The Lagrange basis polynomials for the state collocation polynomials are evaluated to get the values of the states at the mesh points.
`mutable_external_data`	True	If true and the `external_data` option is used, the external data can be changed after discretization, e.g. during warm starting.

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

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

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

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

```
res_opt.get_solver_statistics()
```

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

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

Returns::

    return_status --
        Return status from nonlinear programming solver.

    nbr_iter --
        Number of iterations.

    objective --
       Final value of objective function.

    total_exec_time --
        Execution time.
```

5.1.1. Reusing the same discretization for several optimization solutions

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

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

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

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

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

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

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

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

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

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

and current values retrieved with

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

New initial trajectories can be set with

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

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

External data can be updated with

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

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

Settings to the nonlinear solver can be changed with

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

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

5.1.2. Warm starting

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

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

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

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

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

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

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

```
set_warm_start_options(solver)
```

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

5.2. Examples

5.2.1. Optimal control

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

$$\dot{c}(t) = \frac{F_0(c_0 - c(t))}{V} - k_0 c(t) e^{-\mathrm{EdivR}/T(t)}$$

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

This tutorial will cover the following topics:

- How to solve a DAE initialization problem. The initialization model has equations specifying that all derivatives should be identically zero, which implies that a stationary solution is obtained. Two stationary points, corresponding to different inputs, are computed. We call the stationary points A and B respectively. Point A corresponds to operating conditions where the reactor is cold and the reaction rate is low, whereas point B corresponds to a higher temperature where the reaction rate is high.
- An optimal control problem is solved where the objective is to transfer the state of the system from stationary point A to point B. The challenge is to ignite the reactor while avoiding uncontrolled temperature increases. It is also demonstrated how to set parameter and variable values in a model. More information about the simultaneous optimization algorithm can be found at 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 directly into a Python shell, in some cases with minor modifications. Alternatively, you may copy the commands into a text file, e.g., `cstr_casadi.py`.

Start the tutorial by creating a working directory and copy the file `$JMODELICA_HOME/Python/pyjmi/examples/files/CSTR.mop` to your working directory. An online 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 a Python script file, save it to the working directory.

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

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

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

# Load the FMU
init_model = load_fmu(init_fmu)
```

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

```
help(compile_fmu)
```

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

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

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

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

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

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

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

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

# Print some data for stationary point B
print(' *** Stationary point B ***')
print('Tc = %f' % Tc_0_B)
print('c = %f' % c_0_B)
print('T = %f' % T_0_B)
```

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

5.2.1.2.1. Solving an optimal control problem

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

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

subject to

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

$$T(t) \le 350$$

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

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

First, compile the model and set model parameters:

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

# Load the model
init_sim_model = load_fmu(init_sim_fmu)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

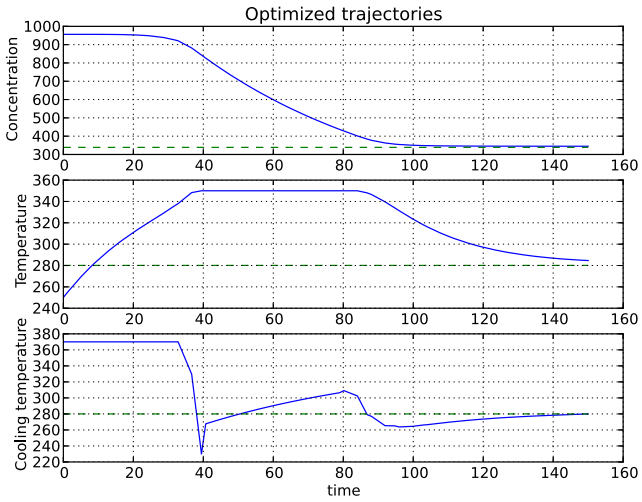
Finally, we plot the result using the functions available in matplotlib:

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

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

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

```
plt.xlabel('time')
plt.show()
```

You should now see the plot shown in Figure 6.2, “Optimal profiles for the CSTR problem.”.

Figure 6.2. Optimal profiles for the CSTR problem.



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

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

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

5.2.1.3. Verify optimal control solution

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

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

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

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

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

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

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

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

# Simulate using optimized input
sim_opts = sim_model.simulate_options()
```

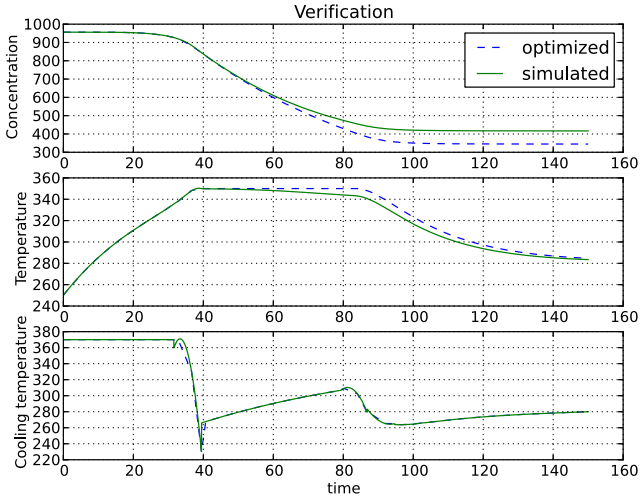
```
sim_opts['CVode_options']['rtol'] = 1e-6
sim_opts['CVode_options']['atol'] = 1e-8
res = sim_model.simulate(start_time=0., final_time=150.,
                         input=('Tc', opt_input), options=sim_opts)
```

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

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

# Plot the results
plt.figure(3)
plt.clf()
plt.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 6.3, “Optimal control profiles and simulated trajectories corresponding to the optimal control input.”.

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



Discuss why the simulated trajectories differ from their optimized counterparts.

5.2.1.4. 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.

5.2.1.5. 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.

5.2.2. Minimum time problems

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

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

subject to the Van der Pol dynamics:

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

and the constraints:

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

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

This problem is encoded in the following Optimica specification:

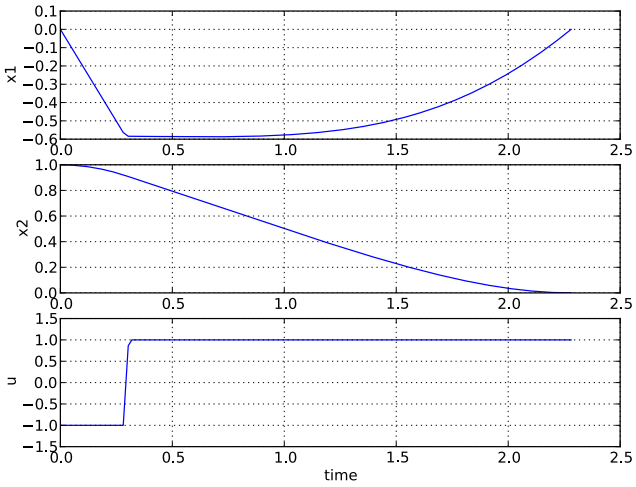
```
optimization VDP_Opt_Min_Time (objective = finalTime,
                               startTime = 0,
                               finalTime(free=true,min=0.2,initialGuess=1))

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

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

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

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

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

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

# Import the JModelica.org Python packages
from pymodelica import compile_fmu
from pyfmi import load_fmu
from pyjmi import transfer_optimization_problem

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

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

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

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

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

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

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



5.2.3. Optimization under delay constraints

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

We consider the optimization problem

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

subject to the dynamics

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

and the boundary conditions

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

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

We model everything except the delay constraint in the Optimica specification

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

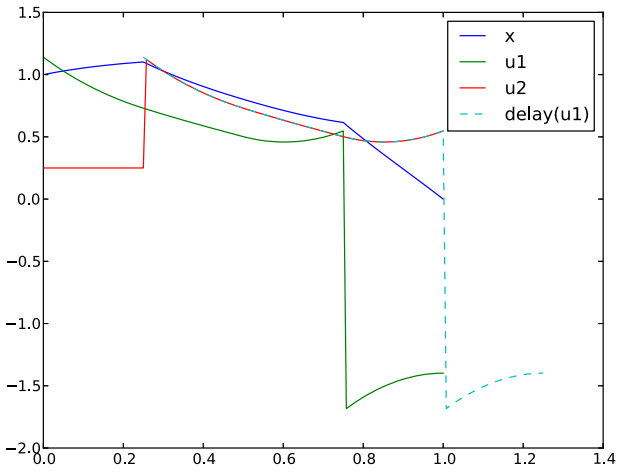
The problem is then solved in the following Python script. Notice how the delay constraint is added using the `delayed_feedback` option, and the initial part of u_2 is set using the `initialGuess` attribute:

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

# Import JModelica.org Python packages
from pyjmi import transfer_optimization_problem

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

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

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

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

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

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

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

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

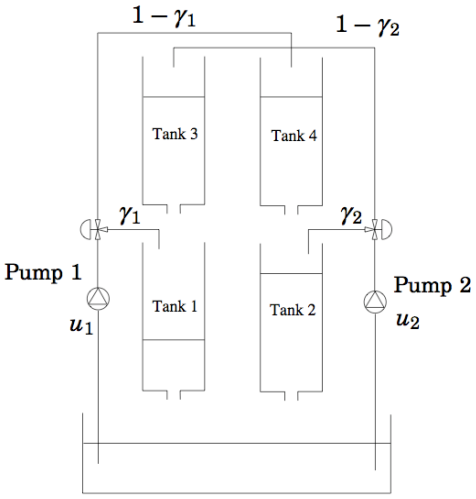
Figure 6.5. Optimization result for delayed feedback example.



5.2.4. Parameter estimation

In this tutorial it will be demonstrated how to solve parameter estimation problems. We consider a quadruple tank system depicted in Figure 6.6, “A schematic picture of the quadruple tank process.”.

Figure 6.6. A schematic picture of the quadruple tank process.



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

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

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

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

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

Where the nominal parameter values are given in Table 6.3, “Parameters for the quadruple tank process.”.

Table 6.3. Parameters for the quadruple tank process.

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

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

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

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

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

```
import os
from collections import OrderedDict

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

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

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

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

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

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

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

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

# Extract data series
t_meas = data['t'][6000::100, 0] - 60
y1_meas = data['y1_f'][6000::100, 0] / 100
y2_meas = data['y2_f'][6000::100, 0] / 100
y3_meas = data['y3_d'][6000::100, 0] / 100
y4_meas = data['y4_d'][6000::100, 0] / 100
u1 = data['u1_d'][6000::100, 0]
u2 = data['u2_d'][6000::100, 0]

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

plt.close(2)
plt.figure(2)
plt.subplot(2, 1, 1)
plt.plot(t_meas, u1)
plt.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 6.7, “Measured state profiles.” and Figure 6.8, “Control inputs used in the identification experiment.”.

Figure 6.7. Measured state profiles.

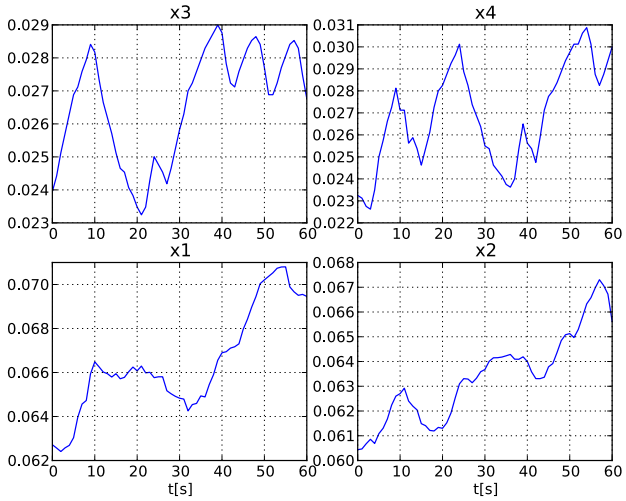
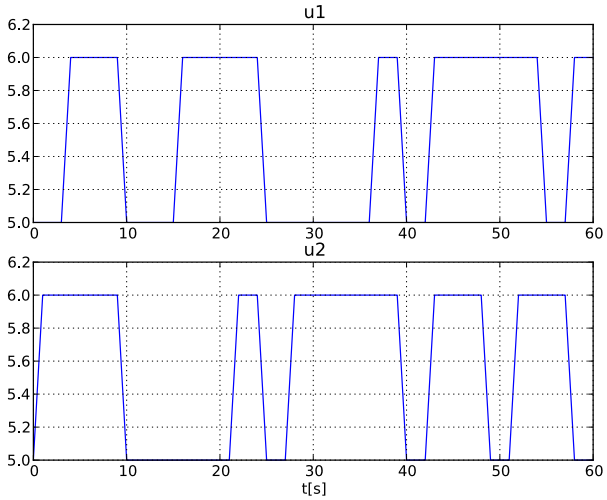


Figure 6.8. Control inputs used in the identification experiment.



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

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

Now, the model can be simulated:

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

The simulation result can now be extracted:

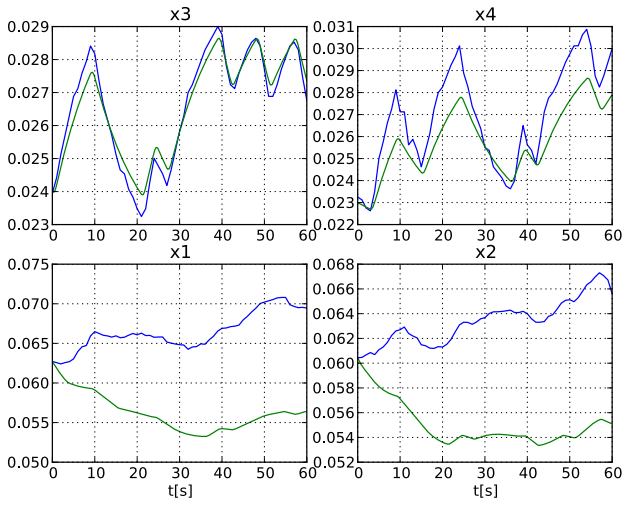
```
# Load simulation result
x1_sim = res_sim['x1']
x2_sim = res_sim['x2']
x3_sim = res_sim['x3']
x4_sim = res_sim['x4']
t_sim  = res_sim['time']
u1_sim = res_sim['u1']
u2_sim = res_sim['u2']
```

and then plotted:

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

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

Figure 6.9, “Simulation result for the nominal model.” shows the result of the simulation.

Figure 6.9. Simulation result for the nominal model.



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

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

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

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

end QuadTank_ParEstCasADi;
```

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

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

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

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

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

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

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

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

You should get an output similar to:

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

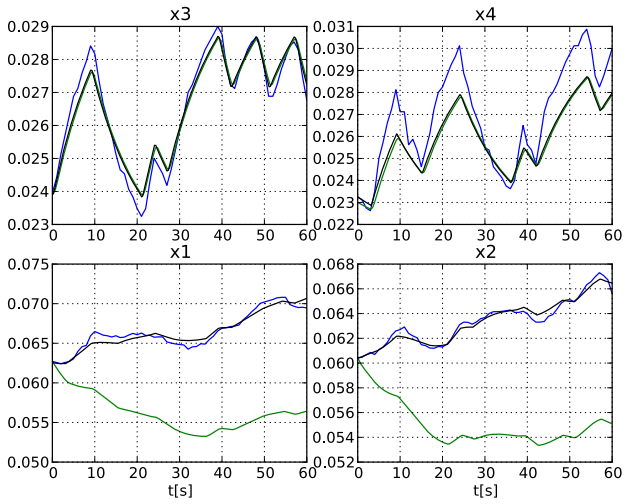
The estimated values are slightly smaller than the nominal values - think about why this may be the case. Also note that the estimated values do not necessarily correspond to the physically true values. Rather, the parameter values are adjusted to compensate for all kinds of modeling errors in order to minimize the mismatch between model response and measurement data.

Next we plot the optimized profiles:

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

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

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

You will see the plot shown in Figure 6.10, “State profiles corresponding to estimated values of a1 and a2.”.

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



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

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

5.3. Investigating optimization progress

This section describes some tools that can be used to investigate the progress of the nonlinear programming solver on an optimization problem. This information can be useful when debugging convergence problems; some of it

(e.g. dual variables) may also be useful to gain a better understanding of the properties of an optimization problem. To make sense of the information that can be retrieved, we first give an overview of the collocation procedure that transcribes the optimization problem into a Nonlinear Program (NLP).

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

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

```
res = model.optimize(options=opts)
solver = res.get_solver()
```

5.3.1. Collocation

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

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

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

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

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

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

5.3.2. Inspecting residuals

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

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

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

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

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

The corresponding time points can be retrieved with

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

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

To get an overview of which residuals are the largest,

```
solver.get_residual_norms()
```

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

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

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

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

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

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

```
solver.get_residual_scale_norms()
```

5.3.3. Inspecting the constraint Jacobian

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

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

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

```
solver.print_nonfinite_jacobian_entries()
```

or returned with

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

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

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

# Find the indices of all entries with absolute value > 10
```

```
J.data = abs(J.data) > 10
c_inds, xx_inds = N.nonzero(J)

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

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

5.3.4. Inspecting dual variables

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

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

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

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

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

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

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

The corresponding time points can be retrieved with

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

5.3.5. Inspecting low level information about NLP solver progress

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

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

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

See the docstring for the respective method for more information.

6. Derivative-Free Model Calibration of FMUs

Figure 6.11. The Furuta pendulum.

This tutorial demonstrates how to solve a model calibration problem using an algorithm that can be applied to Functional Mock-up Units. The model to be calibrated is the Furuta pendulum shown in Figure 6.11, “The Furuta pendulum.”. The Furuta pendulum consists of an arm rotating in the horizontal plane and a pendulum which is free to rotate in the vertical plane. The construction has two degrees of freedom, the angle of the arm, φ, and the angle of the pendulum, θ. Copy the file `$JMODELICA_HOME/Python/pyjmi/examples/files/FMUs/Furuta.fmu` to your working directory. **Note that the Furuta.fmu file is currently only supported on Windows.** Measurement data for φ and θ is available in the file `$JMODELICA_HOME/Python/pyjmi/examples/files/FurutaData.mat`. Copy this file to your working directory as well. These measurements will be used for the calibration. Open a text file, name it `furuta_par_est.py` and enter the following imports:

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

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

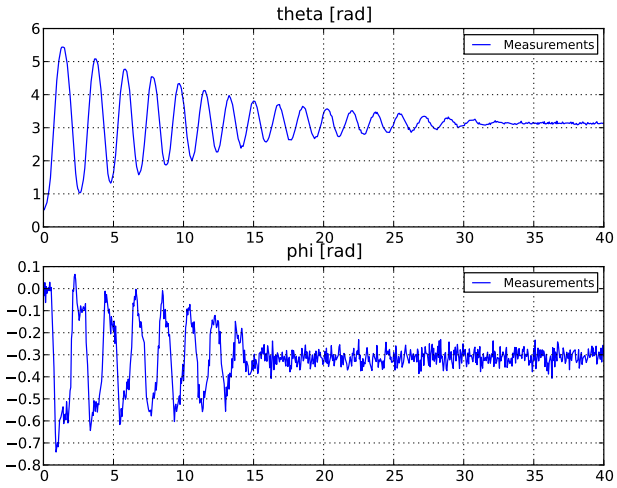
```
# Load measurement data from file
data = loadmat('FurutaData.mat',appendmat=False)
# Extract data series
t_meas = data['time'][:,0]
phi_meas = data['phi'][:,0]
theta_meas = data['theta'][:,0]
```

Now, plot the measurements:

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

This code should generate Figure 6.12, “Measurements of θ and φ for the Furuta pendulum.” showing the measurements of θ and φ.

Figure 6.12. Measurements of θ and φ for the Furuta pendulum.



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

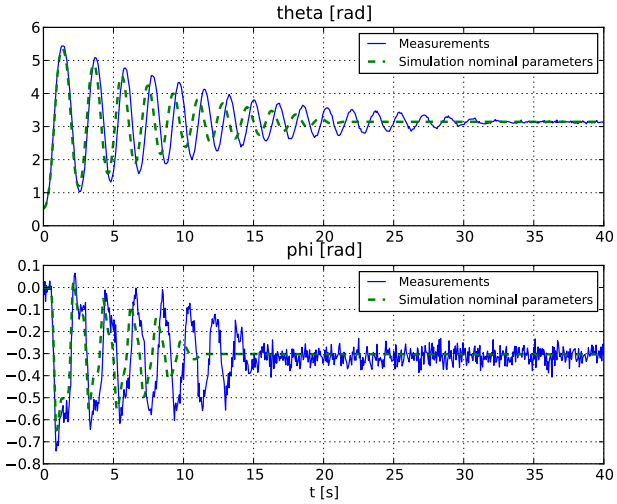
```
# Load model
model = load_fmu("Furuta.fmu")
# Simulate model response with nominal parameters
res = model.simulate(start_time=0.,final_time=40)
# Load simulation result
phi_sim = res['armJoint.phi']
theta_sim = res['pendulumJoint.phi']
t_sim = res['time']
```

Then, we plot the simulation result:

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

Figure 6.13, “Measurements and model simulation result for φ and θ when using nominal parameter values in the Furuta pendulum model.” shows the simulation result together with the measurements.

Figure 6.13. 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 $\left[c_{\text{arm}} \quad c_{\text{pend}}\right]^T$ is the parameter vector. φ^{meas} and θ^{meas} are the measurements of φ and θ, respectively, and φ^{sim} and θ^{sim} are the corresponding simulation results. Now, add code defining a starting point for the algorithm (use the nominal parameter values) as well as lower and upper bounds for the parameters:

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

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

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

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

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

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

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

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

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

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

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

This should give something like the following printout:

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

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

```
# Load model
model = load_fmu("Furuta.fmu")
# Set optimal parameter values into the model
model.set('armFriction',armFrictionCoefficient_opt)
model.set('pendulumFriction',pendulumFrictionCoefficient_opt)
# Simulate model response with optimal parameter values
```

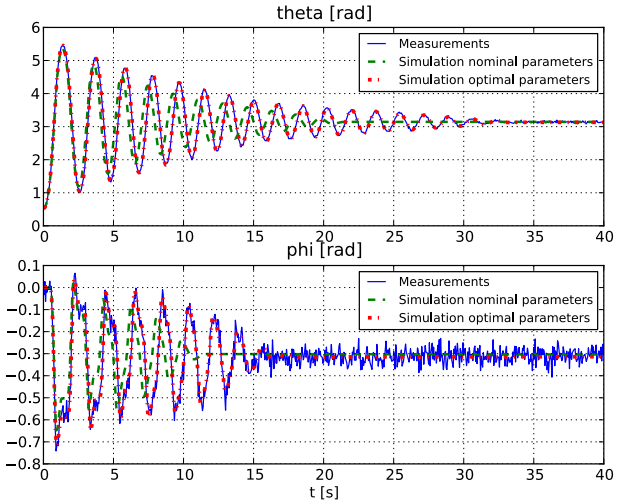
```
res = model.simulate(start_time=0.,final_time=40)
# Load simulation result
phi_opt = res['armJoint.phi']
theta_opt = res['pendulumJoint.phi']
t_opt = res['time']
```

Finally, we plot the simulation result:

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

This should generate the Figure 6.14, “Measurements and model simulation results for φ and θ with nominal and optimal parameters in the model of the Furuta pendulum.”. As can be seen, the agreement between the measurements and the simulation result has improved considerably. The model has been successfully calibrated.

Figure 6.14. Measurements and model simulation results for φ and θ with nominal and optimal parameters in the model of the Furuta pendulum.



Chapter 7. Graphical User Interface for Visualization of Results

1. Plot GUI

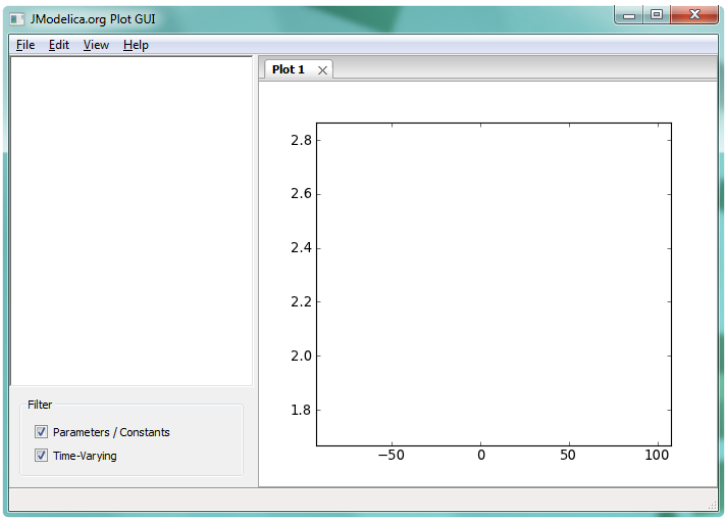
JModelica.org comes with a graphical user interface (GUI) for displaying simulation and / or optimization results. The GUI supports result files generated by JModelica.org and Dymola (both binary and textual formats).

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 7.1. Overview of JModelica.org Plot GUI

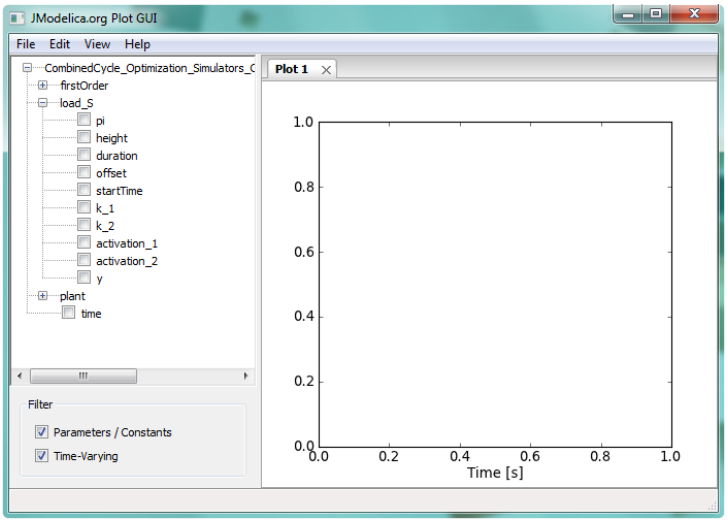


1.1. Introduction

An overview of the GUI is shown in Figure 7.1, “Overview of JModelica.org Plot GUI”. As can be seen, the plot figures are located to the right and can contain multiple figures in various configurations. The left is dedicated to show the loaded result file(s) and corresponding variables together with options for filtering time-varying variables and parameters/constants.

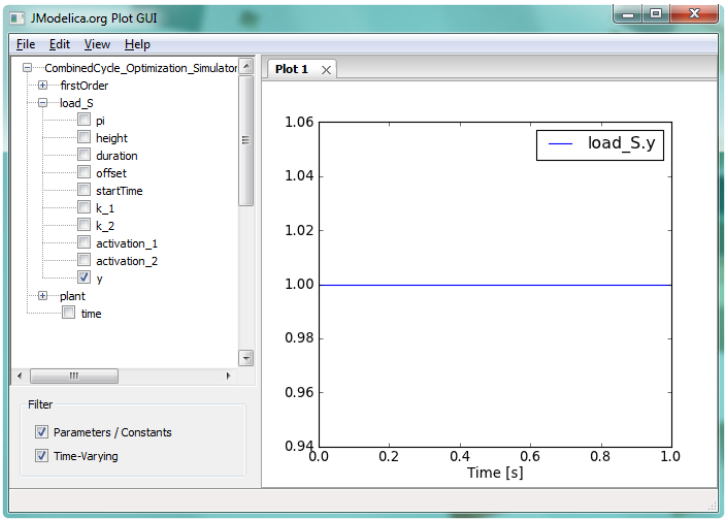
Loading a result file is done using the `File` menu selection `Open` which opens a file dialog where either textual (.txt) results or binary (.mat) results can be chosen. The result is then loaded into a tree structure which enables the user to easily browse between components in a model, see Figure 7.2, “A result file has been loaded.” . Multiple results can be loaded either simultaneously or separately by using the `File` menu option `Open` repeatedly.

Figure 7.2. A result file has been loaded.



Displaying trajectories is done by simply checking the box associated with the variable of interest, see Figure 7.3, “Plotting a trajectory.”. Removing a trajectory follows the same principle.

Figure 7.3. Plotting a trajectory.

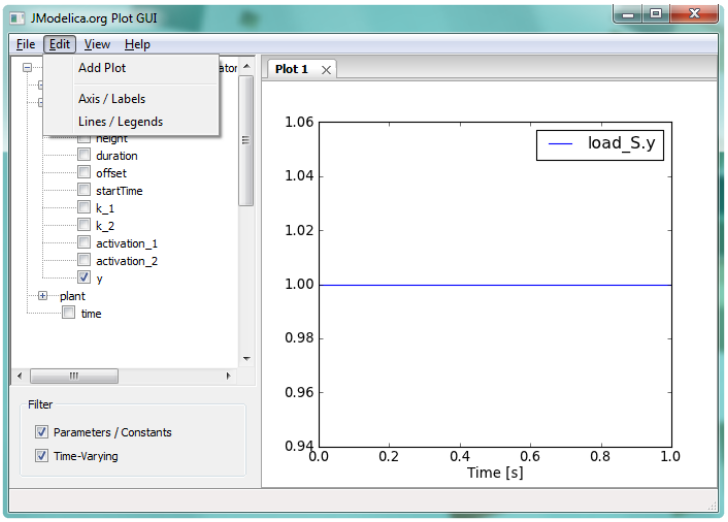


A result can also be removed from the tree view by selecting an item in the tree and by pressing the delete key.

1.2. Edit Options

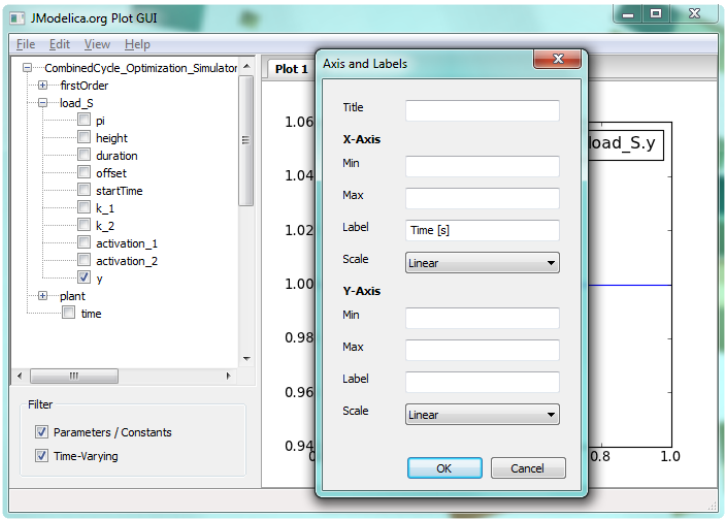
The GUI allows a range of options, see Figure 7.4, “Figure Options.”, related to how the trajectories are displayed such as line width, color and draw style. Information about a plot can in addition be defined by setting titles and labels. Options related to the figure can be found under the `Edit` menu as well as adding more plot figures.

Figure 7.4. Figure Options.



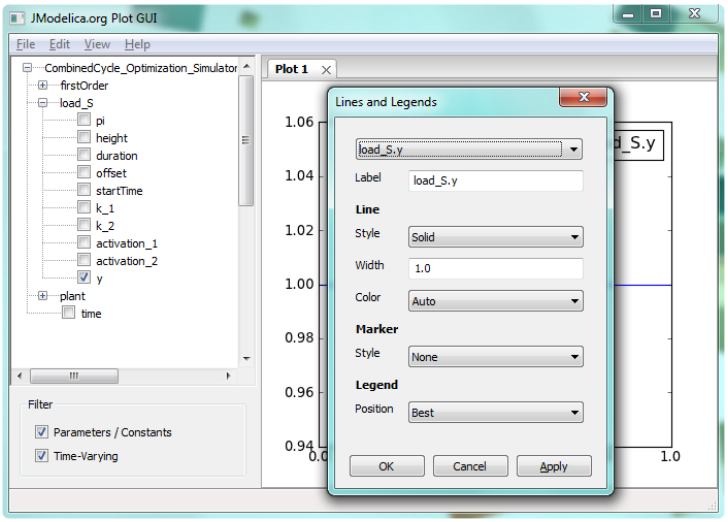
Under `Axis/Labels`, see Figure 7.5, “Figure Axis and Labels Options.”, options such as defining titles and labels in both X and Y direction can be found together with axis options.

Figure 7.5. Figure Axis and Labels Options.



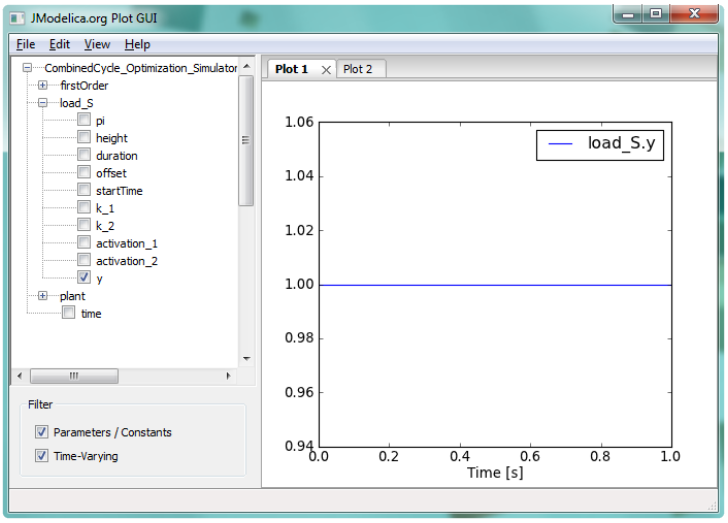
Under `Lines/Legends`, options for specifying specific line labels and line styles can be found, see Figure 7.6, “Figure Lines and Legends options.”. The top drop-down list contains all variables related to the highlighted figure and the following input fields down to `Legend` are related to the chosen variable. The changes take effect after the button `OK` has been pressed. For changing multiple lines in the same session, the `Apply` button should be used.

Figure 7.6. Figure Lines and Legends options.



Additional figures can be added from the `Add Plot` command in the `Edit` menu. In Figure 7.7, “An additional plot has been added.” an additional figure have been added.

Figure 7.7. 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 7.8, “Moving Plot Figure.” "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 7.9, “GUI after moving the plot window.” the result is shown.

Figure 7.8. Moving Plot Figure.

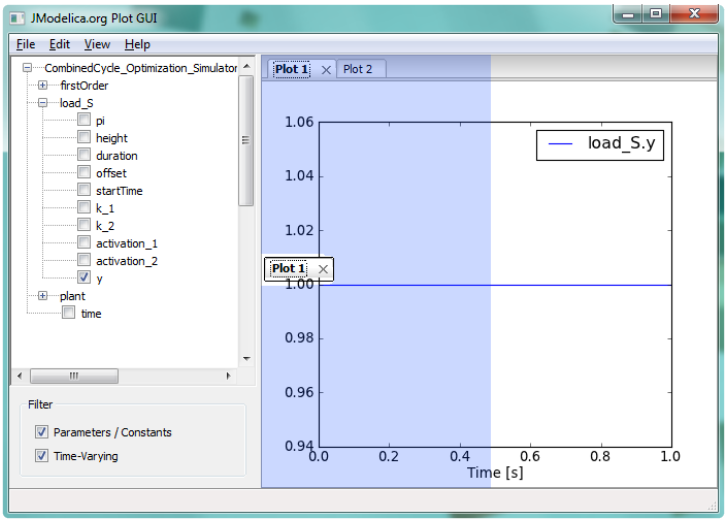
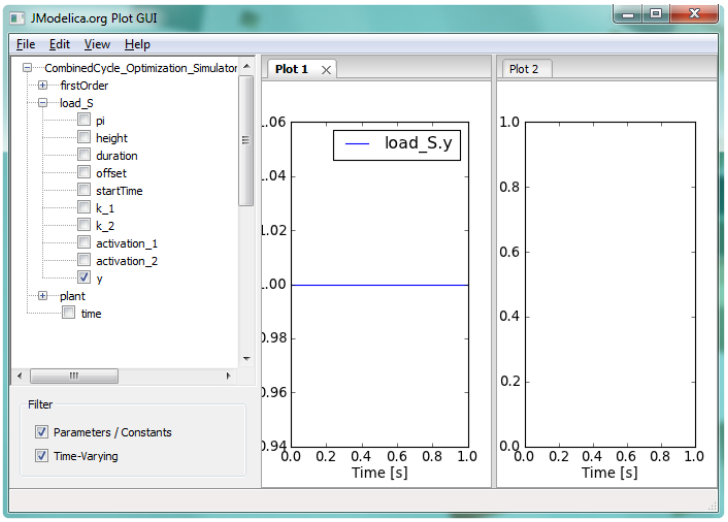
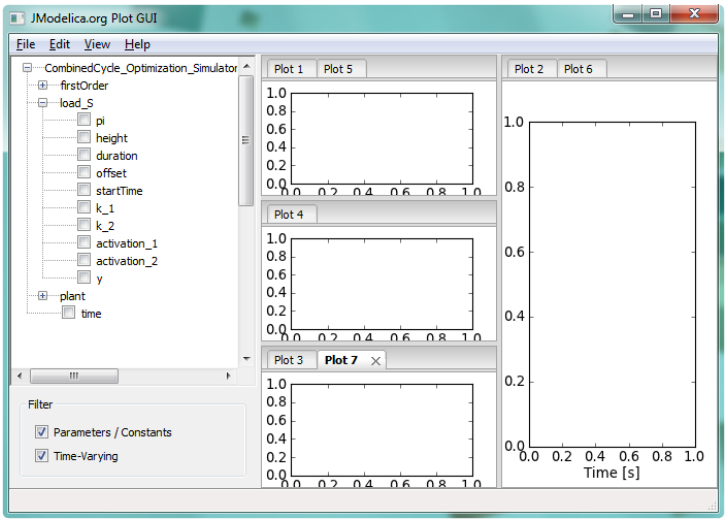


Figure 7.9. GUI after moving the plot window.



If we are to add more figures, an increasingly complex figure layout can be created as is shown in Figure 7.10, “Complex Figure Layout.” where figures also have been dragged to other figure headers.

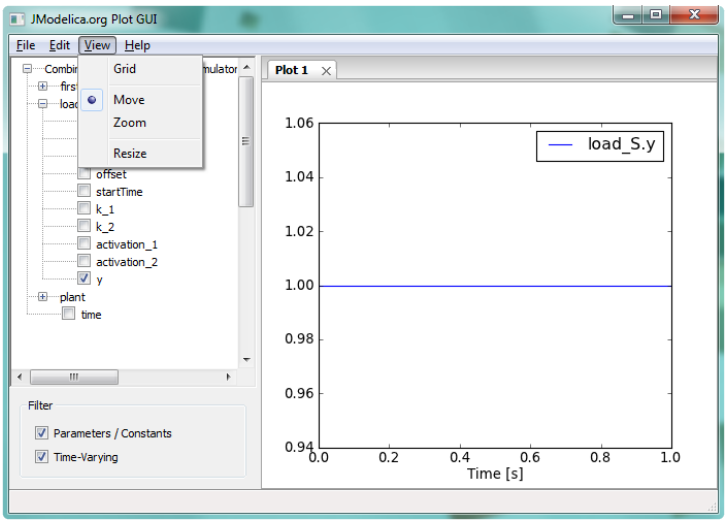
Figure 7.10. Complex Figure Layout.



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 7.11. Figure View Options.

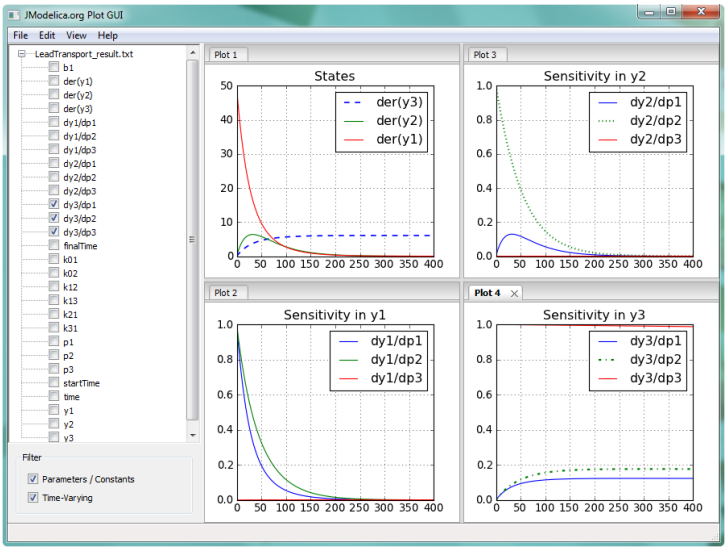


Moving a figure with the `move` option is performed by simply pressing the left mouse button and while still holding down the button, dragging the plot to the area of interest. A zoom operation is performed in a similar fashion.

1.4. Example

Figure 7.12, “Multiple figure example.” shows an example of how the GUI can be used to plot four different plots with different labels. Some of the lines have also been modified in width and in line style. A grid is also shown.

Figure 7.12. Multiple figure example.



Chapter 8. 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 `optimization` classes in the current version of Optimica. Rather, the underlying assumption is that an `optimization` class defines an optimization problem, that is solved offline. 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. Note that such initial guesses may be needed both for control and state variables. For such variables, however, the proposed strategy for providing initial guesses may sometimes be inadequate. In some cases, a better solution is to use simulation data to initialize the optimization problem. This approach is also supported by the Optimica compiler. In the double integrator example, the control variable *u* is a free optimization variable, and accordingly, the `free` attribute is set to `true`. Also, the `initialGuess` attribute is set to 0.0.

```
optimization DIMinTime
  DoubleIntegrator di(u(free=true,
                        initialGuess=0.0));
  input Real u = di.u;
end DIMinTime;
```

3. A Function for accessing instant values of a variable

An important component of some dynamic optimization problems, in particular parameter estimation problems where measurement data is available, is variable access at discrete time instants. For example, if a measurement data value, y_i, has been obtained at time t_i, it may be desirable to penalize the deviation between y_i and a corresponding variable in the model, evaluated at the time instant t_i. In Modelica, it is not possible to access the value of a variable at a particular time instant in a natural way, and a new construct therefore has to be introduced.

All variables in Modelica are functions of time. The variability of variables may be different-some are continuously changing, whereas others can change value only at discrete time instants, and yet others are constant. Nevertheless, the value of a Modelica variable is defined for all time instants within the simulation, or optimization, interval. The time argument of variables are not written explicitly in Modelica, however. One option for enabling access to variable values at specified time instants is therefore to associate an implicitly defined function with a variable declaration. This function can then be invoked by the standard Modelica syntax for function calls, `y(t_i)`. The name of the function is identical to the name of the variable, and it has one argument; the time instant at which the variable is evaluated. This syntax is also very natural since it corresponds precisely to the mathematical notation of a function. Note that the proposed syntax `y(t_i)` makes the interpretation of such an expression context dependent.

In order for this construct to be valid in standard Modelica, `y` must refer to a function declaration. With the proposed extension, `y` may refer either to a function declaration or a variable declaration. A compiler therefore needs to classify an expression `y(t_i)` based on the context, i.e., what function and variable declarations are visible. This feature of Optimica is used in the constraint section of the double integrator example, and is described below.

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

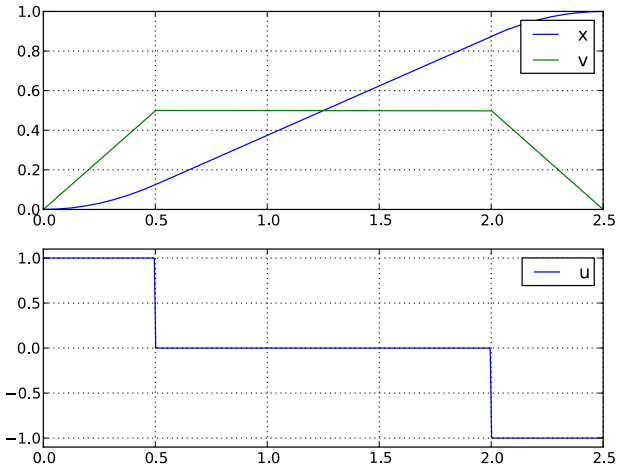
Constraints are similar to equations, and in fact, a path equality constraint is equivalent to a Modelica equation. But in addition, inequality constraints, as well as point equality and inequality constraints should be supported. It is therefore natural to have a separation between equations and constraints. 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 seen in Figure 8.1, “Optimization result”.

Figure 8.1. Optimization result



Chapter 9. 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. Note 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/pyjmi/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 pymodelica
from pymodelica.compiler_wrappers 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 and compiler target object
mc = ModelicaCompiler()

# Build trees as if for an FMU for ME v 1.0
target = mc.create_target_object("me", "1.0")
```

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 has 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(0). \
               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 an iterator over 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 an iterator over of local classes using the method ClassDecl.classes()
    # which returns a Java Iterable object over ClassDecl objects.
    local_classes = class_decl.classes().iterator()

    num_classes = 0
    # Loop over all local classes
    while local_classes.hasNext():
        # Call count_classes recursively for all local classes
        # (including the contained class itself)
        num_classes += 1 + count_classes(local_classes.next(), depth + 1)

    # If the class declaration corresponds to a package, print
    # the number of hierarchically contained classes
    if class_decl.isPackage() 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 see 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 39 hierachically contained classes
The package Modelica.Blocks has 343 hierachically contained classes
The package Modelica.ComplexBlocks has 44 hierachically contained classes
The package Modelica.StateGraph has 66 hierachically contained classes
The package Modelica.Electrical has 992 hierachically contained classes
The package Modelica.Magnetic has 174 hierachically contained classes
The package Modelica.Mechanics has 558 hierachically contained classes
The package Modelica.Fluid has 687 hierachically contained classes
The package Modelica.Media has 1791 hierachically contained classes
The package Modelica.Thermal has 95 hierachically contained classes
The package Modelica.Math has 166 hierachically contained classes
The package Modelica.ComplexMath has 31 hierachically contained classes
The package Modelica.Utilities has 97 hierachically contained classes
The package Modelica.Constants has 0 hierachically contained classes
The package Modelica.Icons has 32 hierachically contained classes
The package Modelica.SIunits has 584 hierachically contained classes
The package Modelica has 5715 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", target)
```

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, target)
```

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.numAlgebraicContinousRealVariables())
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 10. Limitations

This page lists the current limitations of the JModelica.org platform. The development of the platform can be followed at the Trac site, where future releases and associated features are planned. The JModelica.org platform download page has links to compliance reports detailing the current MSL compliance.

- The Modelica compliance of the front-end is limited; the following features are currently not supported:
  - The support for String variables and parameters is limited.
  - Partial support for external functions; records are not supported as arguments or return values.
  - The following built-in functions are not supported:

`terminal()`

  - The following built-in functions are only supported in FMUs:

`ceil(x)`	`integer(x)`	`reinit(x, expr)`
`div(x,y)`	`mod(x,y)`	`sample(start,interval)`
`edge(b)`	`pre(y)`	`semiLinear(...)`
`floor(x)`	`rem(x,y)`	`sign(v)`
`initial()`	`delay(...)`	`spatialDistribution(...)`

- In the Optimica front-end the following constructs are not supported:
  - Annotations for transcription information.
- The following limitations apply to FMUs compiled with JModelica.org:
  - Source code FMUs can not be generated, only binary FMUs.
  - Functions for setting and getting string variables do not work.
  - The dependenciesKind attribute in the XML file for FMU 2.0 is not generated.
  - Directional derivatives are known to have limitations in some cases.
  - Asynchronous simulation is not supported.
  - FMU states (set, get and serialize) are not supported.
- The following limitations apply to optimization using CasADi-based collocation with JModelica.org:
  - Incomplete support for the `Integer` and `Boolean` types: To the extent that they are supported, they are treated more or less like reals.
  - No support for `String` and `enumeration` types.
  - Attributes with any name can be set on any type of variable.
  - The property of whether an optimization problem has free or fixed time horizon cannot be changed after compilation.
- The following limitations apply to JMUs compiled with JModelica.org (note that JMUs are deprecated in JModelica.org 1.15):
  - 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 for interacting with JMUs does not comply with FMI specification.
- Discrete variables are not supported in JMUs.

Appendix A. Compiler options

1. List of options that can be set in compiler

Appendix B. Release notes

1. Release notes for JModelica.org version 1.17

1.1. Highlights

For this release, all focus has been on Modelica/MSL compliance. All example models in the bundled MSL version both check and simulate correctly with this release.

1.2. Compiler

1.2.1. Compliance

For this release, the Modelica Standard Library (MSL) version 3.2.1 build 4 is used with some additional hand-picked revisions from trunk. All example models in this version of MSL simulate correctly with the JModelica.org 1.17 release. The results can be seen in the compliance reports for simulation and check on the JModelica.org download site.

The trunk version of MSL has some additional example models compared to version 3.2.1. build 4. Compliance reports for trunk MSL can be found on the JModelica.org public Jenkins, using trunk version of JModelica.org.

2. Release notes for JModelica.org version 1.16

2.1. Highlights

- Strong focus on Modelica/MSL compliance
- A number of improvements to the CasADi tool chain for optimization

2.2. Compiler

2.2.1. Compliance

For this release, there has been a strong emphasis on improving Modelica/MSL compliance. In several MSL subpackages almost all example models now simulate with a correct result. Complete compliance reports can be found on JModelica.org public Jenkins.

Especially, compliance improvements have been made in the following subpackages:

- Modelica.Mechanics.MultiBody
- Modelica.Blocks
- Modelica.Electrical.Analog
- Modelica.Electrical.Digital
- Modelica.Electrical.QuasiStationary
- Modelica.Electrical.Spice3
- Modelica.Magnetic
- Modelica.Mechanics.Rotational
- Modelica.Media

- Modelica.Thermal
- Modelica.Math

Further, the following operators are now supported:

- delay
- spatialDistribution

2.2.2. Support for dynamic state select

JModelica.org now does dynamic state selection, when necessary.

2.3. Optimization

Several additions and improvements in the CasADi tool chain for optimization have been made. Among the most important are:

- Warm starting - descretize an optimization problem once, solve it multiple times with different parameters, inputs, and initial guesses
- Classes for Model Predictive Control and Moving Horizon Estimation
- Back tracking from discretized problem to original. Trace back residuals, dual variables, and troublesome Jacobian entries to the original model's equations and variables
- Possible to inspect equation scaling
- Checkpointing option to reduce discretization work

3. Release notes for JModelica.org version 1.15

3.1. Highlights

- FMI export supporting FMI 2.0
- FMI import supporting FMI 2.0 with PyFMI
- Improved MSL compliance
- Support for over-constrained initialization systems
- Dynamic optimization framework based on CasADi 2.0
- Improved numerical algorithms in FMU runtime

3.2. Compiler

3.2.1. Compliance

Many bug fixes in the compiler has resulted in greatly increased MSL support. Most or all of the tests and examples for the following MSL sub-libraries now compile and simulate successfully (complete compliance information for MSL can be found on the JModelica.org website, www.jmodelica.org):

- Blocks
- ComplexBlocks
- Electrical.Analog

- Electrical.Machines
- Electrical.MultiPhase
- Electrical.QuasiStationary
- Electrical.Spice3
- Magnetic
- Mechanics.Rotational
- Mechanics.Translational
- Media
- Thermal
- Utilities

3.2.2. Support for over-constrained initialization systems

Automatic balancing of over-constrained initial systems is now implemented. This means that the compiler automatically checks the consistency of the initial system and automatically removes redundant initial equations.

3.2.3. FMU 2.0 export

Support for export of FMUs according to the the recently released FMI 2.0 specification, both for Model Exchange and Co-Simulation, has been added.

3.2.4. Improved numerical algorithms in FMU runtime

Numerous improvements has been made to the FMU runtime code. Specific improvements include solving one-dimensional non-linear systems more robustly.

3.2.5. CasADi 2.0 support in Optimization

The CasADi based optimization tool chain has been updated to work with CasADi 1.9 and later (which is not backwards compatible with CasADi 1.8 and earlier). This allows exploiting new CasADi improvements such as bug fixes, pluggable solvers, and improved documentation. The version of CasADi that is included in JModelica.org is now 2.0.

3.3. Simulation

Support for the recently released FMI 2.0 specification has been included in PyFMI. FMUs following FMI 2.0 can now be loaded and simulated just as easily as FMUs following FMI 1.0.

4. Release notes for JModelica.org version 1.14

4.1. Highlights

- All models in the Modelica Standard Library, except Modelica.Fluid and those using operator delay() or function pointers, pass error check
- FMI export supporting FMI 2.0RC2
- FMI import supporting FMI 2.0RC2 with PyFMI
- Improved error messages from the compiler

- Various improvements and extensions to the CasADi-based optimization toolchain

4.2. Compiler

4.2.1. Compliance

A lot of work with compliance has resulted in that almost all models in the Modelica Standard Library now pass error check. Exceptions are models in Modelica.Fluid and those using the operator delay() or function pointers. In particular, the following improvements have been made:

- Support for arrays indexed with enumerations or Booleans
- Support for overloaded operators and the Complex type
- Improved error messages
- Support for structural parameters depending on external C/Fortran code
- Support for index reduction of optimization classes
- Improved modularization and extension points in the compiler
- Support for index reduction of optimization classes
- Many bug fixes to improve Modelica compliance

4.2.2. New compiler API

A new Java API for calling the compiler through a separate process has been added.

4.2.3. FMI 2.0 RC2 export

Support for export of FMUs that are compliant with FMI 2.0 RC2 has been added.

4.3. Simulation

Support for import and simulation of 2.0 RC2 FMUs with the Python package PyFMI.

4.4. Optimization

The following improvements have been made to the CasADi-based collocation algorithm:

- More efficient memory usage and code generation for function evaluations
- Interface added to WORHP, which serves as an alternative to IPOPT
- More general treatment of blocking factors. In particular it is now possible to penalize and constrain the discontinuity jumps.

5. Release notes for JModelica.org version 1.13

5.1. Highlights

- FMI 2.0 Export, according to RC1
- New CasADi tool chain for optimization
- In-lined switches
- Improved compliance

5.2. Compilers

5.2.1. FMI 2.0 RC1 export

FMI 2.0 export according to RC1 is supported. There are some limitations, summarized in the list below.

- Support for dependencies but not for dependenciesKind in the XML tag ModelStructure
- Support for directional derivative but known to have limitations in some cases
- No support for strings and running asynchronuously
- No support for FMU states (set, get and serialize)

5.2.2. Compliance

- Improved support for expandable connectors
- Improved support for unknown array sizes in functions
- Improved handling of the state select attribute
- Many bug fixes

5.3. Simulation

5.3.1. In-lined switches

In-lined switches have been introduced, which gives a more robust initialization and simulation of systems with discrete parts.

5.4. Optimization

5.4.1. New CasADi tool chain

- Support for more Modelica features than previous CasADi-based tool chain
  - User defined functions in models
    - No support for control flow
- Flat model is exposed in Python in symbolic form using CasADi, and can be inspected and manipulated
- Support for a variety of collocation options

6. Release notes for JModelica.org version 1.12

6.1. Highlights

- Greatly improved support for Modelica.Mechanics.MultiBody
- Support for expandable connectors
- Support for when statements
- Support for event generating built-in functions
- Support for overconstrained connection graphs
- Support for reinit() operator

6.2. Compilers

The following compliance improvements have been made:

- Improved support for algorithms, including when statements.
- Improved support for if equations.
- Improved handling of discrete constructs.
- Improved handling of attributes in alias sets.
- Improved index reduction algorithm.
- Added support for expandable connectors and for overconstrained connection systems.
- Added support for automatic differentiation of functions with smoothOrder annotation.
- Added support for String operations.
- Many bug fixes.

Added check mode, where a class is checked for errors to see if it can be used in a simulation class.

Class annotations are now only allowed as the last element of the class, as per the Modelica 3.2 specification.

6.3. Simulation

The following simulation improvements have been made:

- Improved the simulation run-time with support for the improvements made in the compiler
- Improved the robustness when solving linear and nonlinear blocks.

JModelica.org now simulates the example models from the MultiBody package in MSL with the exception of the few models that require dynamic state selection.

6.4. Contributors

Bengt-Arne Andersson

Christian Andersson

Tove Bergdahl

Emil Fredriksson

Magnus Gäfvert

Toivo Henningsson

Jonathan Kämpe

Björn Lennernäs

Fredrik Magnusson

Jesper Mattsson

Iakov Nakhimovski

Jon Sten

Johan Ylikiiskilä

Johan Åkesson

6.4.1. Previous contributors

Sofia Gedda

Petter Lindgren

John Lindskog

Tobias Mattsson

Lennart Moraeus

Philip Nilsson

Teo Nilsson

Patrik Meijer

Kristina Olsson

Roberto Parrotto

Jens Rantil

Philip Reuterswärd

Jonas Rosenqvist

7. Release notes for JModelica.org version 1.11

7.1. Highlights

- Runtime logging
- Support for ModelicaError and assert
- Additional method in block solver
- Support for ModelicaStandardTables in MSL
- Improved compliance

7.2. Compilers

The following compliance improvements have been made:

- Most of the previously unsupported operators are now supported for FMUs
- Support for assert clauses
- String operations are now supported (this is useful for asserts, even though String variables are not supported)
- Support for vectorization for built-in functions
- Inlining of simple functions is now activated by default
- Several bug fixes

7.3. Simulation

7.3.1. Runtime logging

The runtime logging has been much improved with a new debugging and analysis framework. This enables debugging of convergence issues in non-linear systems of equations.

7.3.2. Support for ModelicaError and assert

The compiler and runtime has support for ModelicaError and assert clauses. If an assert clause fails or a ModelicaError is called, the integrator will reject the current step.

7.4. Contributors

Bengt-Arne Andersson

Christian Andersson

Tove Bergdahl

Emil Fredriksson

Magnus Gäfvert

Toivo Henningsson

Jonathan Kämpe

Fredrik Magnusson

Jesper Mattsson

Iakov Nakhimovski

Jon Sten

Johan Ylikiiskilä

Johan Åkesson

7.4.1. Previous contributors

Sofia Gedda

Björn Lennernäs

Petter Lindgren

John Lindskog

Tobias Mattsson

Lennart Moraeus

Philip Nilsson

Teo Nilsson

Patrik Meijer

Kristina Olsson

Roberto Parrotto

Jens Rantil

Philip Reuterswärd

Jonas Rosenqvist

8. Release notes for JModelica.org version 1.10

8.1. Highlights

- Export of FMUs for Co-Simulation
- Import of FMU 2.0b4 in PyFMI
- Improved log format for FMUs
- Improved variable scaling in the CasADi collocation
- Improved handling of measurement data in the CasADi collocation
- Improved logging from compilers
- Improved Modelica compliance

8.2. Compilers

The following compliance improvements have been made:

- The following operators are now supported:
  - smooth()
  - skew(x)
  - scalar(A)
  - vector(A)
  - matrix(A)
  - diagonal(v)
- Improved handling of unmatched HGT. All unmatched iteration variables and residual equations are now paired and treated the same way as regular HGT pairs.
- Improvements have been made to analytical jacobians. Notably full support for functions and bug fixes.

Also many bug fixes and performance improvements have been made.

8.2.1. Export of FMUs for Co-Simulation

Export of FMUs for Co-Simulation version 1.0 is now supported. Specifying a co-simulation FMU instead of a model exchange FMU is done via an option to the `compile_fmu` method. The internal solver in the co-simulation FMU is CVode from the Sundials suite and there is also an explicit Euler method. The choice of the solver can be changed via a parameter in the FMU.

8.3. Python

8.3.1. Improved result data access

Modified handling of simulation and optimization results to facilitate post processing of results such as plotting. Accessing variables and parameters from results will always return a vector of size equal to the time vector. Also,

the base result class (`JMResultBase`) has two new functions, `initial` and `final`, which will always return initial and final value of the simulation/optimization as scalar values. See both Chapter 5, *Simulation of FMUs* and Chapter 6, *Optimization* for plotting code examples.

8.3.2. Improved error handling

Improved error handling of compiler problems (exceptions, errors and warnings). Problems are now given in the same way as regardless if JPype or separate process is used when compiling. Additionally errors and warning are now returned as python objects to facilitate easier post processing of compiler problems. It is also possible to retrieve warnings from the return result of `compile_fmu`, `compile_jmu` and `compile_fmux`, e.g.:

```
r = compile_fmu('Test', 'test.mo')
print r.warnings
```

8.3.3. Parsing of FMU log files

FMUs and JMUs created with JModelica.org now produce logs in a structured XML format, which can be either parsed using tools in the Python module `pyjmi.log` or using general purpose XML tools. See Section 5.2, “Run-time logging” for code examples.

8.4. Simulation

8.4.1. Support for FMU version 2.0b4

Added support for simulation of models following the FMI version 2.0 beta 4, both model exchange FMUs and co-simulation FMUs.

8.4.2. Result filter

Added an option to the simulation method for filtering which variables are stored. This is especially useful in case of large models with many variables as just selecting a subset of variables to store can speed up the simulation. Additionally there is now the option to store the result directly in the memory instead of writing the result to file.

8.4.3. Improved solver support

Improvements on the solvers has been made resulting in that simulation of Model Exchange FMUs can now be performed by a number of solvers. See the simulation options for the supported solvers. For example there is now an Radau5 solver.

8.5. Optimization

8.5.1. Improved variable scaling

The variable scaling performed based on nominal trajectories for the CasADi collocation has been improved and can now be set individually for each variable. It also has a more robust default behavior.

8.5.2. Improved handling of measurement data

The old class `ParameterEstimationData` for the CasADi collocation has been replaced by `MeasurementData`. The new class can also be used for optimal control, and not only parameter estimation, and also offers additional strategies in the handling of the data.

8.6. Contributors

Bengt-Arne Andersson

Christian Andersson

Tove Bergdahl

Emil Fredriksson

Magnus Gäfvert

Toivo Henningsson

Jonathan Kämpe

John Lindskog

Fredrik Magnusson

Jesper Mattsson

Iakov Nakhimovski

Teo Nilsson

Jon Sten

Johan Ylikiiskilä

Johan Åkesson

8.6.1. Previous contributors

Sofia Gedda

Björn Lennernäs

Petter Lindgren

Tobias Mattsson

Lennart Moraeus

Philip Nilsson

Patrik Meijer

Kristina Olsson

Roberto Parrotto

Jens Rantil

Philip Reuterswärd

Jonas Rosenqvist

9. Release notes for JModelica.org version 1.9.1

This release contains a bug fix which eliminates a dependency on external libraries in FMUs. Apart from this bug fix, the release is identical to JModelica.org version 1.9.

10. Release notes for JModelica.org version 1.9

10.1. Highlights

- Improved function inlining

- Manual selection of iteration variables in tearing algorithm - Hand Guided Tearing (HGT)
- Support for external objects
- Simulation of Co-simulation FMUs in Python
- Improved compiler execution speed
- Improved compiler memory efficiency
- Support for MSL CombiTables
- Improvements to the CasADi-based collocation optimization algorithm, including support for non-fixed time horizons and supplying nominal trajectories for scaling purposes
- Updated to MSL 3.2

10.2. Compilers

10.2.1. Improved Modelica compliance

The following compliance improvements have been made:

- Support for external objects (classes extending the predefined partial class ExternalObject)
- Support for the same component being added from more than one extends clause.
- Many bug fixes, notably concerning inheritance and redeclares.

10.2.2. Support for MSL CombiTables

There is now support for MSL CombiTables, both 1D and 2D. The table can be either read from file or explicitly supplied as a parameter matrix.

10.2.3. Support for hand guided tearing

The tearing algorithm in the compiler can now be influenced by user selected residuals and iteration variables, in order to make such selections explicit, e.g., to exploit physical insight in the choice of iteration variables. The selections are made by means of vendor specific annotations and can be done at the component level and at the system level.

10.2.4. Improved function inlining

Improved support for inlining of functions. Notably a new in-lining mode has been added, where functions that can be inlined without introducing additional variables to the model. The inlining algorithm has also been expanded to handle more situations.

10.2.5. Memory and execution time improvements in the compiler

The compilation times for large simulation models has been reduced by more than two orders of magnitudes. Also, the memory required to compile large models has been decreased by two orders of magnitude. As a consequence, larger models up to 100.000 equations can be comfortably compiled on a standard computer.

10.3. Python

10.3.1. Compile in separate process

The possibility to compile in a separate process from the Python interface has been added. This is enabled with an argument to `compile_fmu`, `compile_jmu` or `compile_fmux` which is `False` by default. It is also possible to

pass arguments to the JVM. This enables, among other things, users on 64 bit Windows to use a 64 bit JRE (Java Runtime Environment) for compiling a model.

10.4. Simulation

10.4.1. Simulation of co-simulation FMUs

Support for simulation of co-simulation FMUs following the FMI version 1.0 has been implemented and follows the same work-flow as for loading and simulating an model exchange FMU, i.e:

```
from pyfmi import load_fmu
model = load_fmu("CS_Model.fmu")
res = model.simulate(final_time=1.0)
...
```

10.5. Optimization

10.5.1. Improvements to CasADi-based collocation algorithm

The following features have been added to the CasADi-based collocation algorithm

- Support for non-fixed time horizons, allowing the formulation of, for example, minimum-time problems
- Possibility to supply nominal trajectories based on simulation results, which are used to compute (possibly time-variant) scaling factors. This makes it possible to conveniently obtain good scaling for all variables in a model.
- Possibility to use more advanced interpolation of optimized inputs based on collocation polynomials, instead of linear interpolation, providing higher accuracy when simulating a system using optimized inputs
- Setting of nominal attributes from Python in loaded models

10.6. Contributors

Bengt-Arne Andersson

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Fredrik Magnusson

Jesper Mattsson

Iakov Nakhimovski

Jonas Rosenqvist

Jon Sten

Johan Ylikiiskilä

Johan Åkesson

10.6.1. Previous contributors

Sofia Gedda

Petter Lindgren

Tobias Mattsson

Lennart Moraeus

Philip Nilsson

Patrik Meijer

Kristina Olsson

Roberto Parrotto

Jens Rantil

Philip Reuterswärd

11. Release notes for JModelica.org version 1.8.1

This release is identical to JModelica.org version 1.8 apart from one important bug fix. The issue that has been fixed concerns the scaling of start attributes in JMUs.

12. Release notes for JModelica.org version 1.8

12.1. Highlights

- Improved Modelica compliance of the compiler front-end, including support for if equations and inner/outer declarations
- Optimized performance and memory utilization of the compiler front-end
- A new state selection algorithm with support for user defined state selections
- A new function inlining algorithm for conversion of algorithmic functions into equations
- Improvements to the CasADi-based collocation optimization algorithm, including support for terminal constraints

12.2. Compilers

12.2.1. Improved Modelica compliance

The following compliance improvements have been made:

- Support for if equations
- Support for inner/outer declarations
- Expressions in der() operator
- Function call equations in when equations
- Limited support for String parameters. String parameters are now supported in the compiler front-end, although they are discarded in the code generation.

Also, many bug fixes and performance improvements in the compiler are included in this release.

12.2.2. Function inlining

There is a new function inlining algorithm for conversion of algorithmic functions into equations.

12.2.3. New state selection algorithm

The new state selection algorithm takes user input (stateSelect attribute) into account and implements heuristics to select states that avoids, if possible, iteration of non-linear systems of equations.

12.3. Python

12.3.1. Simplified compiling with libraries

The compiler now support adding extra libraries as files, which makes it easier to compile a model using a structured library not in the MODELICAPATH. Both Python functions `compile_jmu` and `compile_fmu` support this. For example, compiling `A.B.Example` from a library `A` in directory `LibDir` with `compile_fmu`, this can now be written as:

```
compile_fmu('A.B.Example', 'LibDir/A')
```

12.4. Optimization

12.4.1. Improvements to CasADi-based collocation algorithm

The CasADi-based collocation algorithm has been improved with new features

- Support for point constraints
- Setting of parameter values from Python in loaded models
- Setting of min/max attributes from Python in loaded models

12.5. Contributors

Bengt-Arne Andersson

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Fredrik Magnusson

Jesper Mattsson

Tobias Mattsson

Iakov Nakhimovski

Jon Sten

Johan Ylikiiskilä

Johan Åkesson

12.5.1. Previous contributors

Sofia Gedda

Petter Lindgren

Lennart Moraeus

Philip Nilsson

Patrik Meijer

Kristina Olsson

Roberto Parrotto

Jens Rantil

Philip Reuterswärd

13. Release notes for JModelica.org version 1.7

13.1. Highlights

- Improved support for hybrid systems, including friction models and ideal diodes
- Support for tearing of equation systems
- Support for external Fortran functions
- Support for function inlining
- Reorganization of the Python code: a new stand-alone package, PyFMI, provided
- A novel dynamic optimization algorithm implemented in Python based on collocation and CasADi is provided

13.2. Compilers

13.2.1. Support for mixed systems of equations

Mixed systems of equations, i.e., equation systems containing both real and integer/boolean variables are supported. Such systems commonly occurs in, e.g., friction models and diode models.

13.2.2. Support for tearing

Tearing is a technique to improve simulation efficiency by reducing the number of iteration variables when solving systems of equations. A tearing algorithm relying on graph-theoretical methods has been implemented, which is used to generate more efficient simulation code.

13.2.3. Improved Modelica compliance

With added support for external Fortran function and many bug fixes, the compiler now handles many models that previously would not compile.

13.2.4. Function inlining

Calls to Modelica functions (i.e. not external functions) in equations can now be inlined, by adding the equivalent equations and temporary variables. This allows some transformations that are specific to equations to be performed on the function calls as well. It also allows compilation targets that does not handle functions, such as CasADi, to be used with models containing functions. Currently, only functions that only contains assignment statements are supported. Such function are common in e.g. media libraries.

13.3. Python

13.3.1. New package structure

The Python code has been refactored into three packages:

- **PyFMI** A package for working with FMUs, perform simulations, interact with the model, plotting of result data and more. This package can be used stand-alone, see www.pyfmi.org.

- **PyJMI** A package for working with JMUs, solve optimization problems, perform simulations, model interaction and more.
- **PyModelica** A package containing Modelica and Optimica compilers.

13.3.2. Support for shared libraries in FMUs

The FMU import and export now supports dependencies on extra shared libraries. For the export, the shared libraries are placed in the same folder as the model binary. Similarly, any shared libraries packed with the model binary will be found when importing the FMU.

13.4. Simulation

13.4.1. Simulation of hybrid systems

The improved compiler support for mixed systems of equations is matched by extensions to the JModelica.org simulation runtime system, enabling simulation of more sophisticated hybrid models. Amongst others, the classic Modelica.Mechanics.Rotational.Examples.CoupledClutches benchmark model can be now simulated.

13.5. Optimization

13.5.1. A novel CasADi-based collocation algorithm

A novel CasADi-based collocation algorithm is provided. The new algorithm is implemented in Python and relies on the CasADi package for computation of derivatives and interaction with IPOPT. The new algorithm is an order of magnitude faster than the existing collocation algorithm on many problems, and provides significantly improved flexibility.

13.6. Contributors

Bengt-Arne Andersson

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Petter Lindgren

Fredrik Magnusson

Jesper Mattsson

Patrik Meijer

Iakov Nakhimovski

Johan Ylikiiskilä

Johan Åkesson

13.6.1. Previous contributors

Sofia Gedda

Lennart Moraeus

Philip Nilsson

Kristina Olsson

Roberto Parrotto

Jens Rantil

Philip Reuterswärd

14. Release notes for JModelica.org version 1.6

14.1. 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

14.2. Compilers

14.2.1. 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.

14.2.2. 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).

14.3. Python

14.3.1. 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.

14.3.2. 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.

14.3.3. Compilation of XML models

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

14.3.4. Python version upgrade

The Python package has been updated to Python 2.7.

14.4. Optimization

14.4.1. 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.

14.4.2. 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.

14.5. 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.

14.6. 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

14.6.1. Previous contributors

Philip Nilsson

Roberto Parrotto

Jens Rantil

Philip Reuterswärd

15. Release notes for JModelica.org version 1.5

15.1. 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.

15.2. Compilers

15.2.1. When clauses

When clauses are supported in the Modelica compiler.

15.2.2. 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.

15.2.3. Connections

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

15.2.4. Eclipse IDE

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

15.2.5. Miscellaneous

Fixed several compiler bugs.

15.3. Simulation

15.3.1. 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.

15.3.2. 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.

15.3.3. 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.

15.4. 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.

15.5. 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.

15.6. Contributors

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Jesper Mattsson

Johan Ylikiiskilä

Johan Åkesson

15.6.1. Previous contributors

Philip Nilsson

Roberto Parrotto

Jens Rantil

Philip Reuterswärd

16. Release notes for JModelica.org version 1.4

16.1. 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

16.2. Compilers

16.2.1. Enumerations

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

16.2.2. Miscellaneous

Fixed many compiler bugs, especially concerning complex class structures.

16.2.3. 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.

16.2.4. 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.

16.3. Python interface

16.3.1. 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`

16.3.2. 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.

16.3.3. 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.

16.3.4. 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']
```

16.4. Simulation

16.4.1. 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.

16.4.2. 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.

16.4.3. 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.

16.5. Contributors

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Jesper Mattsson

Johan Ylikiiskilä

Johan Åkesson

16.5.1. Previous contributors

Philip Nilsson

Roberto Parrotto

Jens Rantil

Philip Reuterswärd

17. Release notes for JModelica.org version 1.3

17.1. 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)

17.2. Compilers

17.2.1. The Modelica compiler

17.2.1.1. 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.

17.2.1.2. Redecare

Redeclares as class elements are now supported.

17.2.1.3. Conditional components

Conditional components are now supported.

17.2.1.4. 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.

17.2.1.5. 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.

17.2.1.6. Stream connectors

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

17.2.1.7. 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.

17.2.2. The Optimica compiler

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

17.2.2.1. 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.

17.2.2.2. 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.

17.3. JModelica.org Model Interface (JMI)

17.3.1. The collocation optimization algorithm

17.3.1.1. 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.

17.3.1.2. 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.

17.4. 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.

17.5. 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.

17.6. XML model export

17.6.1. `noEvent` operator

Support for the built-in operator `noEvent` has been implemented.

17.6.2. `static` attribute

Support for the Optimica attribute static has been implemented.

17.7. Python integration

17.7.1. High-level functions

17.7.1.1. Model files

Passing more than one model file to high-level functions supported.

17.7.1.2. 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.

17.7.2. File I/O

Rewriting xmlparser.py has improved performance when writing simulation result data to file considerably.

17.8. Contributors

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Jesper Mattsson

Roberto Parrotto

Johan Åkesson

Philip Reuterswärd

17.8.1. Previous contributors

Philip Nilsson

Jens Rantil

18. Release notes for JModelica.org version 1.2

18.1. 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

18.2. Compilers

18.2.1. The Modelica compiler

18.2.1.1. 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.

18.2.1.2. 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)

18.2.1.3. 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.

18.2.1.4. 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.

18.2.2. The Optimica Compiler

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

18.3. The JModelica.org Model Interface (JMI)

18.3.1. General

18.3.1.1. 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.

18.3.1.2. 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.

18.3.1.3. 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.

18.3.1.4. 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.

18.4. The collocation optimization algorithm

18.4.1. 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.

18.4.2. 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.

18.4.3. 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.

18.5. 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.

18.6. 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.

18.7. 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.

18.8. 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.

18.8.1. 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.

18.9. Contributors

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Jesper Mattsson

Philip Nilsson

Roberto Parrotto

Philip Reuterswärd

Johan Åkesson

18.9.1. Previous contributors

Jens Rantil

Appendix C. Initialization and simulation of JMUs (Deprecated in JModelica.org 1.15)

1. Introduction

There are two different means to simulate a Modelica model in JModelica.org: either as a Functional Mock-Up Unit (FMU) or as a JModelica.org Model Unit (JMU). In the former case, the model is converted into an Ordinary Differential Equation (ODE), whereas in the latter case, the model is simulated as a Differential Algebraic Equation (DAE). The default, and recommended, means to simulate models is to use FMUs, since this approach provides better performance, a more robust initialization mechanism, and significantly better support for hybrid systems. Simulation of JMUs may still be useful, however, in some optimization applications where simulation of the model as a DAE/JMU, is an important prerequisite for optimization by means of the collocation algorithms.

This chapter demonstrates how to initialize and simulate a model which has been compiled to a JMU. To read about simulation of FMUs, see Chapter 5, *Simulation of FMUs*.

2. Initialization of JMUs

2.1. 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 JMU
from pymodelica import compile_jmu
model_name = compile_jmu("My.Model", "/path/to/MyModel.mo")

# Load the model instance into Python
from pyjmi import JMUModel
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']
```

2.2. 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.

2.3. Initialization algorithms

2.3.1. 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. Note 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 C.1, “Options for the collocation-based optimization algorithm”.

Table C.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.
```

2.3.2. Initialization using KInitSolveAlg

JModelica.org also provides a method for DAE initialization based on the non-linear equation 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 Jacobians as well as support for SuperLU, an efficient sparse linear solver.

The options providable are summarized in Table C.2, “Options for KInitSolveAlg”.

Table C.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 2.3.2.1, “The use of constraints”.
`constraints`	`None`	A `numpy.array` containing floats that, when supplied, defines the constraints on the variables. Further explained in Section 2.3.2.1, “The use of constraints”.
`result_format`	`'txt'`	Specifies in which format to write the result. Currently only textual mode is supported.
`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 C.3, “Options for KINSOL contained in the `KINSOL_options` dictionary”.

Table C.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`).

2.3.2.1. The use of constraints

KINSOL, and hence also `KInitSolvAlg`, only supports simple unilateral constraints, that is constraining a variable to being positive or negative. If the option `use_constraints` is set to `True`, constraints are used. Which 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 C.4, “Values allowed in the `constraints` array”.

Table C.4. Values allowed in the `constraints` array

Value	Constraint
0.0	Unconstrained.
1.0	Greater than, or equal to, zero.
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 C.4, “Values allowed in the `constraints` array” 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.

2.3.2.2. Verbosity of KINSOL

There are four different levels of verbosity in KINSOL with 0 being silent and 3 being the most verbose. The verbosity level is controlled by the FMU log level. Table C.5, “Verbosity levels in KINSOL” describes what is output.

Table C.5. Verbosity levels in KINSOL

FMU log level	Verbosity level	Output
<= 2	0	No information displayed.
3	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.
4	2	Level 1 output as well as the Euclidian and in finity norm of the *scaled* residual at the current iterate
>= 5	3	Level 2 output plus additional values used by the global strategy as well as statistical information from the linear solver.

3. Simulation of JMUs

Simulation of JMUs in JModelica.org is performed via the simulate method of the JMU model object. The model object is called `JMUModel` and is located in the JModelica.org Python package `pyjmi`. `JMUModel` supports compiled models from JModelica.org which have the extension `.jmu`.

```
# Import JMUModel from pyjmi and load the JMU
from pyjmi import JMUModel
my_model = JMUModel('myJMU.jmu')
```

The simulation method in `JMUModel` is by default connected to the Assimulo simulation package and thus able to use its solvers. Continuing the short example from above, the following code will simulate the loaded JMU using default values and options:

```
res = my_model.simulate()
```

3.1. The simulate function

There are several parameters that can be set in the `JMUModel.simulate` function.

`start_time`	The time when the solver should start the integration.
`final_time`	The time when the solver should finish the integration.
`input`	Input signal for the simulation. (Further explained in below.)
`algorithm`	The algorithm that will be used for the simulation. Currently only a connection to Assimulo is supported and connected through the algorithm `AssimuloAlg`.
`options`	The options to be used in the algorithm. (Further explained in below.)

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.

```
import numpy as N
t = N.linspace(0.,10.,100)            # Create one hundred evenly spaced points
u = N.sin(t)                          # Create the input vector
u_traj = N.transpose(N.vstack((t,u))) # Create the data matrix and transpose
                                      # it to the correct form
```

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

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

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

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

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

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

Note that the variables are now a List of variables.

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

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

and for the double input case:

```
def input_function(t):
```

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

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

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 C.6, “General options for AssimuloAlg.” the general options for the `AssimuloAlg` algorithm are described while in Table C.8, “Selection of solver arguments for IDA” a selection of the different solver arguments for the DAE solver IDA is shown. In Table C.7, “Selection of solver arguments for CVode” 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 C.6. 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
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 C.7. 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 C.8. 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. 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 C.9, “Result Object”.

Table C.9. 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.

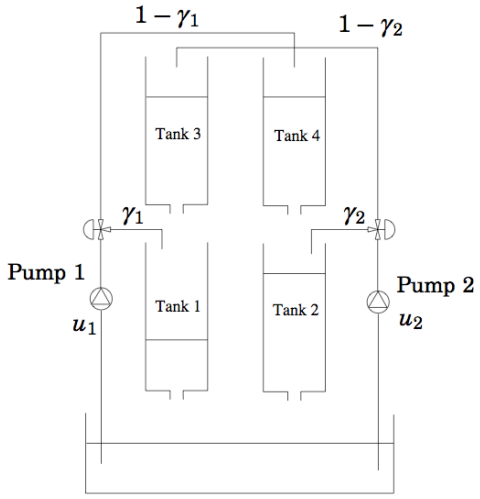
3.2. Examples

In the next sections, it will be shown how to use the JModelica.org platform for simulation of various JMUs.

3.2.1. 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 also are the inputs to the model. The model is depicted in Figure C.1, “A schematic picture of the quadruple tank process.” and in the code below the corresponding Modelica code is listed.

Figure C.1. 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;
```

Let's 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 pymodelica import compile_jmu
from pyjmi import JMUModel
```

The input data is stored in `qt_par_est_data.mat` which can be found in the `Python/pyjmi/examples/files` catalogue in the JModelica.org install folder. 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 has been adapted, let's 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 C.2, “Tank levels” the result of the tank levels are shown and in Figure C.3, “Input trajectories” the input signals are shown.

Figure C.2. Tank levels

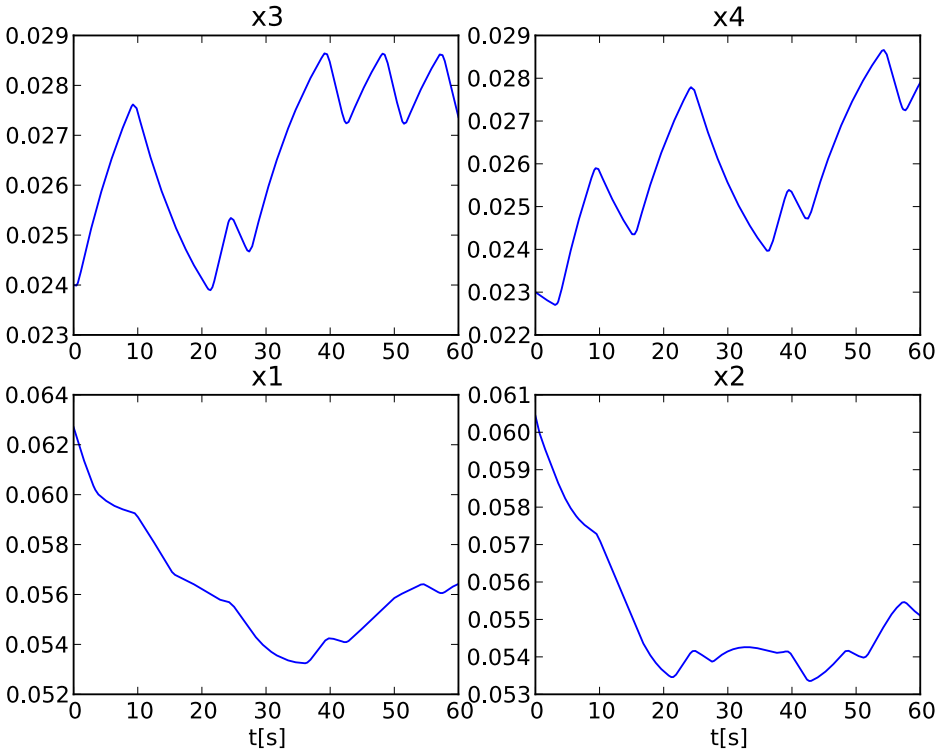
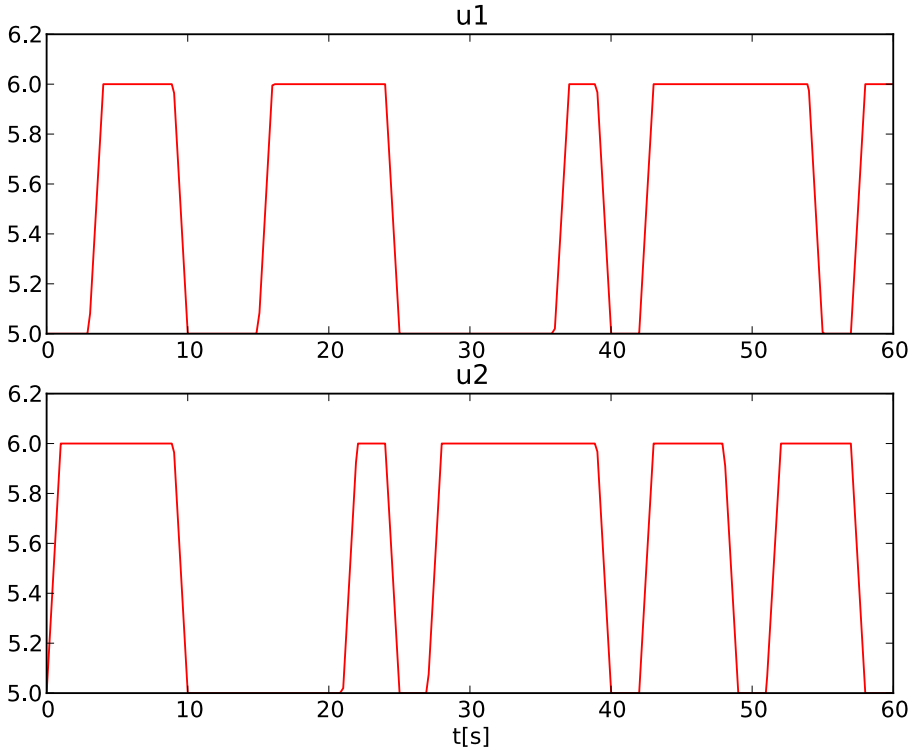


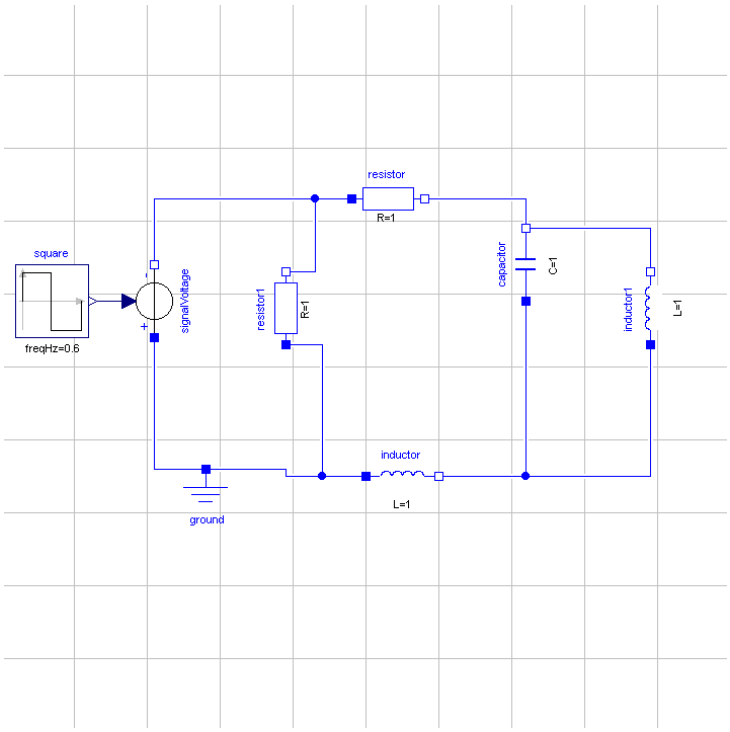
Figure C.3. Input trajectories



3.2.2. Simulation of a discontinuous system

The model to be simulated in this example is an electric circuit. The model is depicted in Figure C.4, “Electric Circuit” 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 C.4. Electric Circuit



This example 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 pymodelica 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 in which is easily done in the following way:

```
square_y    = res['square.y']
resistor_v  = res['resistor.v']
```

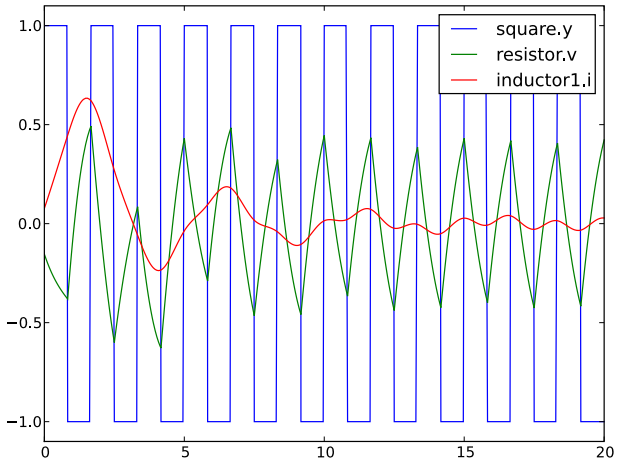
```
inductor1_i = res['inductor1.i']
time        = res['time']
```

And then plotted with `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 C.5, “Simulation result”.

Figure C.5. Simulation result



3.2.3. 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.

Let's begin with the the example. Copy and paste the Optimica code and save it into `Robertson.mop`, then open a Python script file. We start by importing the necessary objects:

```
# Import the function for compilation of models and the JMUModel class
from pymodelica 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)
```

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

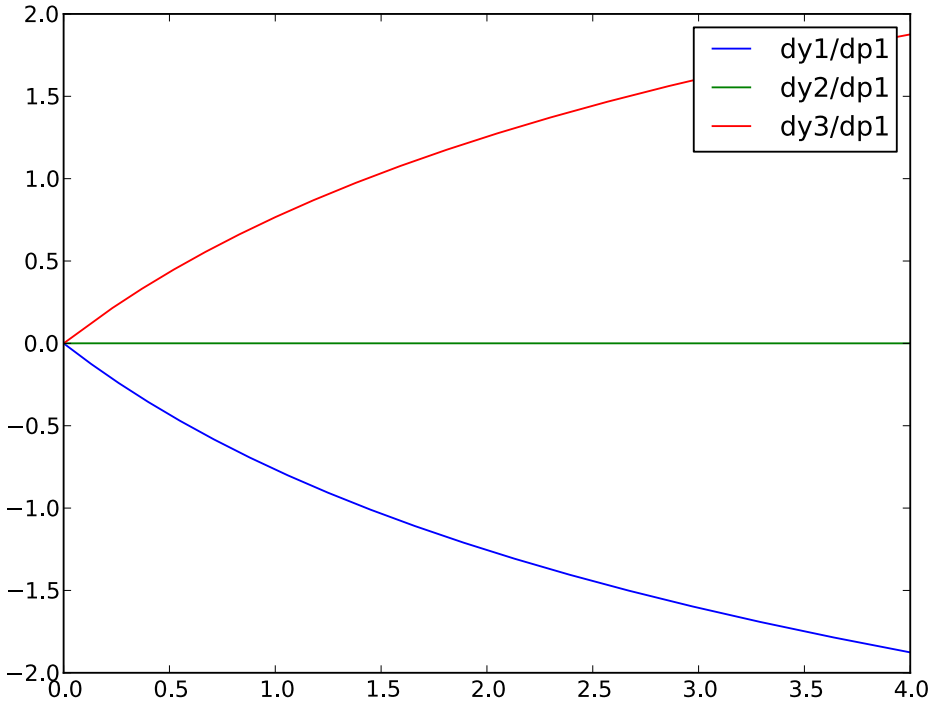
```
dy1dp1 = res['dy1/dp1']
dy2dp1 = res['dy2/dp1']
dy3dp1 = res['dy3/dp1']
time = res['time']
```

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 C.6, “Sensitivity results.” the sensitivities are plotted.

Figure C.6. Sensitivity results.



Appendix D. Dynamic optimization of DAEs using direct collocation with JMUs (Deprecated in JModelica.org 1.15)

1. Dynamic optimization of DAEs using direct collocation with JMUs

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 D.1, “Options for the JMU and collocation-based optimization algorithm”.

Table D.1. Options for the JMU and 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

Option	Default	Description
		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].
`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 `pyjmi.common.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.
```

1.1. Examples

1.1.1. Optimal control

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

1.1.1.1. 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 pymodelica 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)
```

1.1.1.2. 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.

1.1.1.3. Solving an optimal control problem

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

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

$$\text{subject to}$$

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

$$T(t) \le 350$$

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']
```

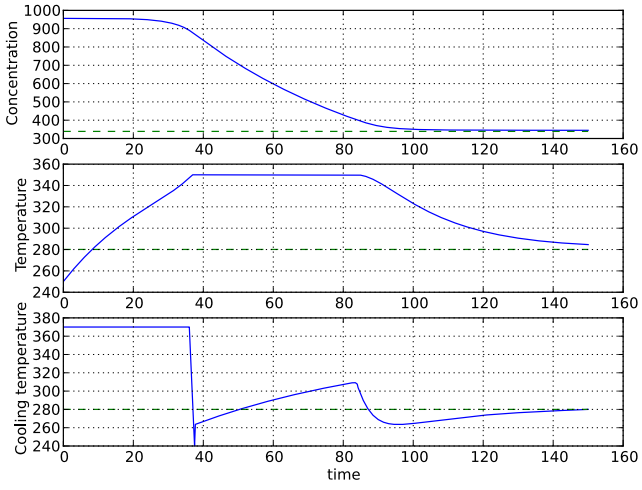
```
T_res=res['cstr.T']
Tc_res=res['cstr.Tc']
time_res = res['time']

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. You should now see the plot shown in Figure D.1, “Optimal profiles for the CSTR problem.”.

Figure D.1. 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?

1.1.1.4. 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:

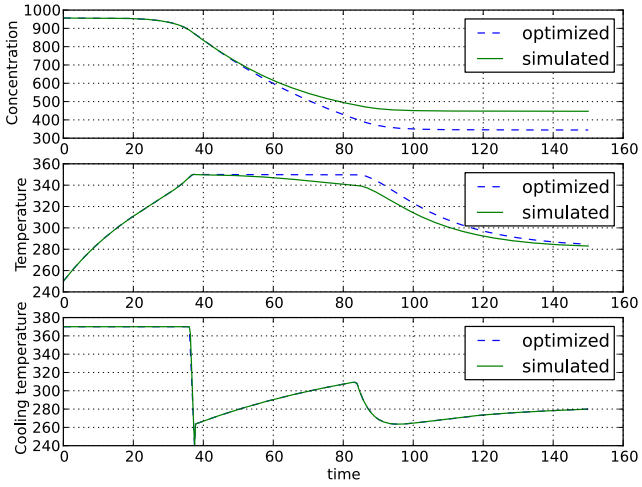
```
# Simulate to verify the optimal solution
# Set up the input trajectory
t = time_res
u = Tc_res
u_traj = N.transpose(N.vstack((t,u)))

# 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 D.2, “Optimal control profiles and simulated trajectories corresponding to the optimal control input.”.

Figure D.2. Optimal control profiles and simulated trajectories corresponding to the optimal control input.



Discuss why the simulated trajectories differs from the optimized counterparts.

1.1.1.5. 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.

1.1.1.6. 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.

1.1.2. Minimum time problems

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

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

subject to the Van der Pol dynamics:

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

and the constraints:

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

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

This problem is encoded in the following Optimica specification:

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

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

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

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

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

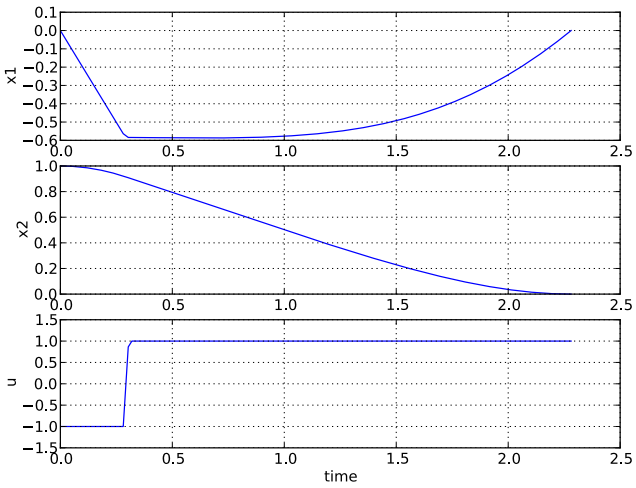
Notice how the class attribute `finalTime` is set to be free in the optimization. The problem is solved by the following Python script:

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

# Import the JModelica.org Python packages
from pymodelica 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 D.3, “Minimum time profiles for the Van der Pol Oscillator.”. Notice the difference as compared to Figure Figure 6.1, “Optimal profiles for the VDP oscillator”, where the Van der Pol oscillator system is optimized using a quadratic objective function.

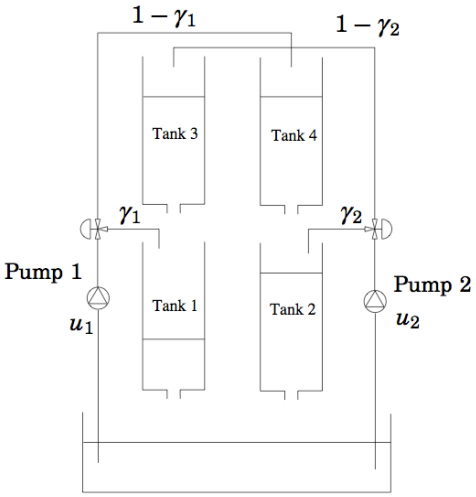
Figure D.3. Minimum time profiles for the Van der Pol Oscillator.



1.1.3. Parameter optimization

In this tutorial it will be demonstrated how to solve parameter estimation problems. We consider a quadruple tank system depicted in Figure 6.6, “A schematic picture of the quadruple tank process.”.

Figure D.4. 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 6.3, “Parameters for the quadruple tank process.”.

Table D.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 pymodelica 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 6.7, “Measured state profiles.” and Figure 6.8, “Control inputs used in the identification experiment.”.

Figure D.5. Measured state profiles.

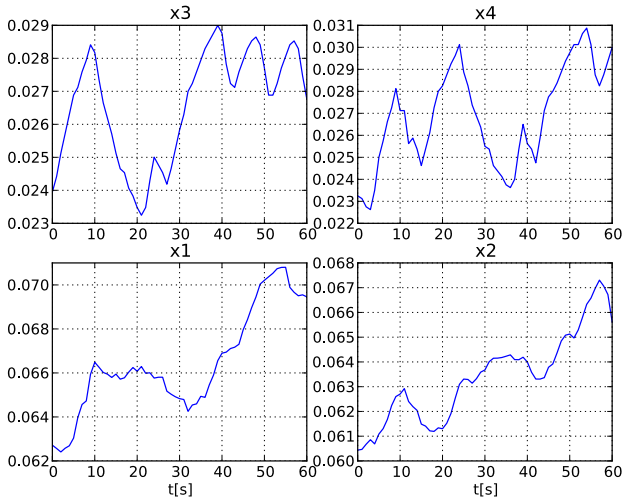
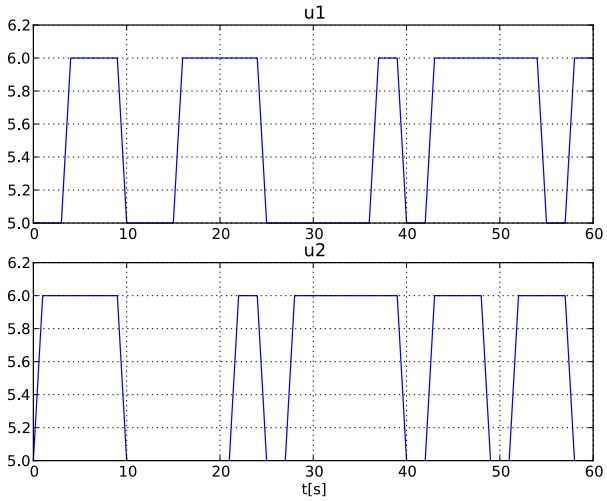


Figure D.6. Control inputs used in the identification experiment.



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:

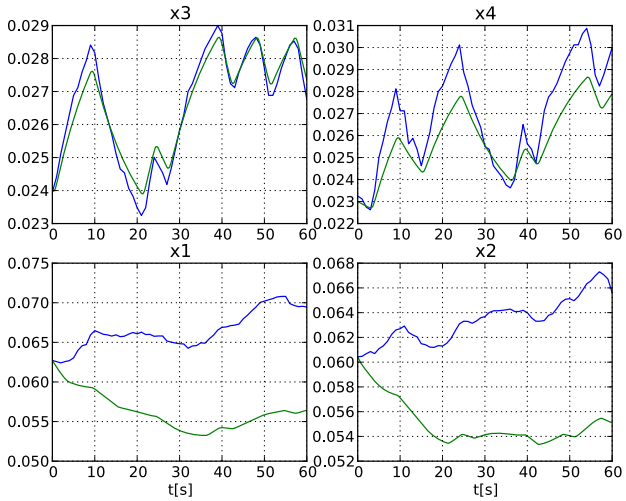
```
# Load simulation result
x1_sim = res['qt.x1']
x2_sim = res['qt.x2']
x3_sim = res['qt.x3']
x4_sim = res['qt.x4']
t_sim  = res['time']
u1_sim = res['u1']
u2_sim = res['u2']
```

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 6.9, “Simulation result for the nominal model.” shows the result of the simulation.

Figure D.7. 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.mop` 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.final("qt.a1")
a2_opt = res.final("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.

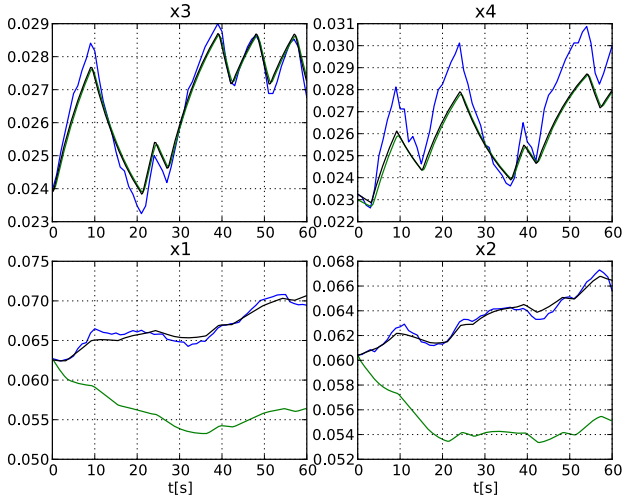
Next we plot the optimized profiles:

```
# 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 6.10, “State profiles corresponding to estimated values of a1 and a2.”.

Figure D.8. 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.final("qt.a1")
a2_opt2 = res_opt2.final("qt.a2")
a3_opt2 = res_opt2.final("qt.a3")
a4_opt2 = res_opt2.final("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:

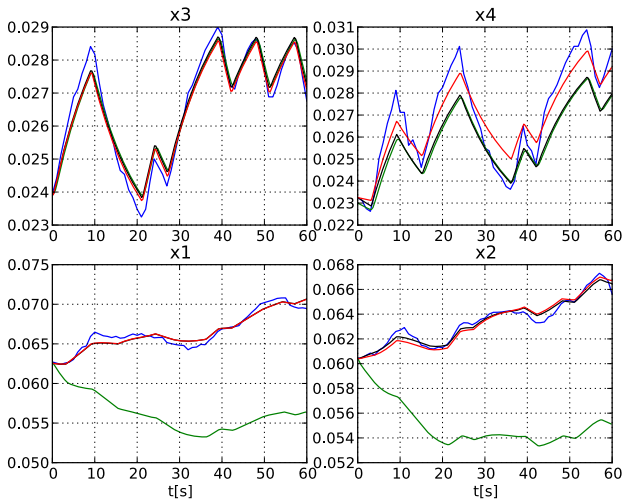
```
a1:0.0266cm^2
a2:0.0271cm^2
a3:0.0301cm^2
a4:0.0293cm^2
```

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 D.9, “State profiles corresponding to estimated values of a1, a2, a3 and a4.”.

Figure D.9. 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.final('a1')) + ", standard deviation: " + str(sigma_a1)
print "a2: " + str(sens_res.final('a2')) + ", standard deviation: " + str(sigma_a2)
print "a3: " + str(sens_res.final('a3')) + ", standard deviation: " + str(sigma_a3)
```

```
print "a4: " + str(sens_res.final('a4')) + ", standard deviation: " + str(sigma_a4)
```

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

Bibliography

[Jak2007] Johan Åkesson. *Tools and Languages for Optimization of Large-Scale Systems*. LUTFD2/TFRT--1081--SE. Lund University. Sweden. 2007.

[Jak2008b] Johan Åkesson, Görel Hedin, and Torbjörn Ekman. *Tools and Languages for Optimization of Large-Scale Systems*. 117-131. *Electronic Notes in Theoretical Computer Science*. 203:2. April 2008.

[Jak2008a] Johan Åkesson. *Optimica—An Extension of Modelica Supporting Dynamic Optimization*. *Proc. 6th International Modelica Conference 2008*. Modelica Association. March 2008.

[Jak2010] Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. *Modeling and Optimization with Optimica and JModelica.org—Languages and Tools for Solving Large-Scale Dynamic Optimization Problem*. *Computers and Chemical Engineering*. 203:2. 2010.

[Eng2001] Peter Englezos and Nicolas Kalogerakis. *Applied Parameter Estimation for Chemical Engineers*. Marcel Dekker Inc. 2001.