JModelica.org User Guide

Version 1.3.0

JModelica.org User Guide: Version 1.3.0

Publication date 2010-08-29
Copyright © 2010 Modelon AB

Acknowledgements

Chapter 1. Introduction

1. About JModelica.org

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

2. Mission Statement

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

3. Technology

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

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

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

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

4. Architecture

Figure 1.1. JModelica platform architecture.
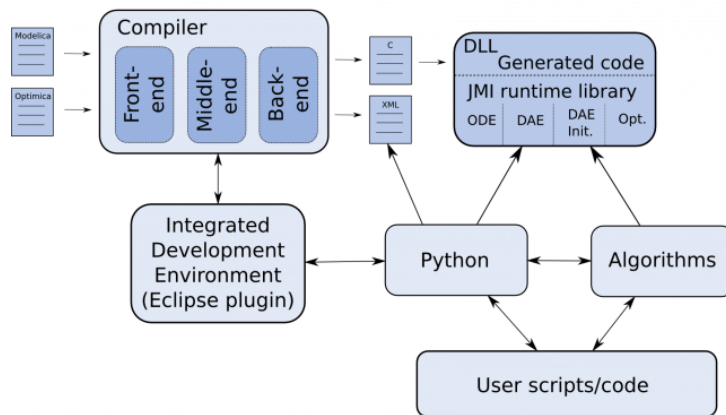
The JModelica.org platform consists of a number of different parts:

- The compiler front-ends (one for Modelica and one for Modelica/Optimica) transforms Modelica and Optimica code into a flat model representation. The compilers also check the correctness of model descriptions and reports errors.

- The compiler back-ends generates C code and XML code for Modelica and Optimica. The C code contains the model equations, cost functions and constraints whereas the XML code contains model meta data such as variable names and parameter values.

- The JModelica.org runtime library is written in C and contains supporting functions needed to compile the generated model C code. Also, the runtime library contains an integration with CppAD, a tool for computation of high accuracy derivatives by means of automatic differentiation.

- Currently, JModelica.org features one particular algorithm for solving dynamic optimization problems. The algorithm is based on collocation on finite elements and relies on the solver IPOPT for obtaining a solution of the resulting NLP.

- JModelica.org uses Python for scripting and prototyping. For this purpose, a Python package is under development with the objective of offering functions for driving the compilers and for accessing the (compiled) functions in the runtime library/generated C code.

5. Extensibility

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

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

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

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

- The JModelica.org compilers are developed using the compiler construction framework JastAdd. JastAdd features extensible compiler construction, both at the language level and at the implementation level. This feature

is explored in JModelica.org where the Optimica compiler is implemented as a fully modular extension of the core Modelica compiler. The JModelica.org platform is a suitable choice for experimental language design and research.

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

Chapter 2. Installation

1. Supported platforms

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

2. Binary distribution

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

2.1. Linux

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

2.2. Mac OS X

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

2.3. Windows

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

2.3.1. Prerequisites

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

2.3.1.1. Java

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

To install JRE

1. Get a JRE installer suitable for your platform here.

2. Run the installer.

2.3.1.2. Python

Python 2.6 with the following additional packages are required:

NumPy   The fundamental package needed for scientific computing with Python.

SciPy   A library of algorithms and mathematical tools for Python.

matplotlib   A plotting library for Python, with a MATLAB like interface.

PyReadline   A readline for Windows required by IPython.

IPython   An interactive shell for Python with additional shell syntax, code highlighting, tab completion, string completion, and rich history.

There are two options to install the necessary Python components, as described below.

Separate installation of Python prerequisites for Windows binary installer

1. Download and install Python 2.6.

2. Download and install the additional packages (be sure to select a version for Python 2.6 when applicable):

    Table 2.1. Python prerequisities

Package	Recommended version
NumPy	1.3.0
SciPy	0.7.1
matplotlib	0.99.1.1
PyReadline	1.5
IPython	0.10

Python(x,y) installation of Python prerequisites for Windows binary installer

Python(x,y) is a scientific-oriented Python distribution that includes all necessary dependencies (and a lot more!).

1. Download a Python(x,y) installer for Python 2.6.

2. Run the installer and select a full installation (or select the required packages manually for a smaller footprint).

2.3.2. Windows installer

To install JModelica.org from Windows installer

Make sure that all prerequisite components are installed before continuing.

1. Download a JModelica.org Windows binary installer.

2. Run the installer and follow the wizard.

    • Choose to install the additional Python packages when prompted, unless the are already installed.

Chapter 3. Getting started

The examples in JModelica.org uses the Python scientific library SciPy, which is dependent on NumPy and the mathematical plotting library matplotlib for plotting. These packages will thus also be used in some examples in this chapter. Most Python functions have built-in documentation, which can be accessed from the Python shell by invoking the help function, for example `help(numpy.size)`. Use this feature frequently to learn more about the packages used in this tutorial, including the jmodelica package. It is possible do the tutorials without any Python knowledge (although it helps to know the basics).

The tutorials in this chapter are preferably run in a Python shell using the Pylab mode. If you are running Windows, select the menu option 'pylab' provided with the JModelica.org installation. If you are running Linux or Mac OS X, open a terminal and enter the command:

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

As your first action, go to the working directory you have created:

```
In [1]: cd '/path/to/working/directory'
```

In order to run the Python script, use the 'run' command:

```
in [2]: run cstr.py
```

In most cases it is convenient to store the Python commands in a script file and run the script from the file by invoking the Python command run:

```
>>> run my_script.py
```

or

```
>>> run -i my_script.py
```

where the latter will render all variables to be accessible in the Python interpreter after termination of the script.

1. Compilation of models

This tutorial covers how to compile Modelica and Optimica models into C and XML and how to load the resulting DLLs into Python.

1.1. Compilation

There are two compilers available, `ModelicaCompiler` and `OptimicaCompiler`. Any model containing Optimica code has to be compiled with the `OptimicaCompiler`. This is the case with the CSTR model which will be used in the following examples.

1.1.1. Simple compilation example

Compiling a model file can be done with just a few lines of code.

1.1.1.1. Instantiating the compiler

First the `OptimicaCompiler` must be imported and instantiated. The compiler instance can then be used multiple times on different model files.

```
# Import the compiler
from jmodelica.compiler import OptimicaCompiler

# Get an instance of OptimicaCompiler
oc = OptimicaCompiler()
```

1.1.1.2. Options

Compiler options are read from an XML file 'options.xml' which can be found in the JModelica.org installation folder. The options are loaded from the file as the compiler is instantiated. Options for a compiler instance can

be modified and new options can be added interactively. There are four type categories: string, real, integer and boolean. The following example demonstrates how to get and set a string option.

```
# Get the string option default_msl_version
oc.get_string_option('default_msl_version')
>> '3.0.1'

# Set the default_msl_version to 3.0.1
oc.set_string_option('default_msl_version','3.0.1')
```

1.1.1.3. Compiling

Compile a model, using the previously created compiler instance, with the method `compile_model` which takes a model name and a model file name as arguments. Once compilation has completed successfully a DLL file along with a few other files will have been created on the file system. The DLL file can then be loaded using the class jmi.Model. The result model object is used to interact with the JMI Model interface. `compile_model` does the DLL loading for you and will return with the jmi.Model model object as return argument.

```
# Compile the model and get a jmi.Model as return argument
cstr_model = oc.compile_model('CSTR.CSTR_Opt', 'CSTR.mo')
```

If the compilation has failed an exception will be raised.

1.1.2. Targets

The `compile_model` method takes an optional argument target which is 'model' by default. There are three other options for this argument, 'model_noad', 'algorithm' and 'ipopt'. It is necessary to compile with target 'ipopt' to use the Ipopt algorithm interface for optimization.

```
# Compile the model with support for Ipopt and get a jmi.Model as return argument
cstr_model = oc.compile_model('CSTR.CSTR_Opt', 'CSTR.mo', target='ipopt')
```

1.1.3. Compilation in more detail

Compiling with `compile_model` actually bundles a few steps required for the compilation which can be run one by one. These steps will be described briefly here, for more information on these steps, see the Architecture section in the Introduction.

1.1.3.1. Flattening

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

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

# Generate an instance tree representation and get a reference to the model instance
model_instance = oc.instantiate_model(source_root, 'CSTR.CSTR_Opt')

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

1.1.3.2. Code generation

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

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

Several files are generated in this step.

1.1.3.3. Generate Shared Object file (DLL)

Finally, the DLL file is built where the C code is linked with the JModelica.org Model Interface (JMI) runtime library. The target argument must be set here if something other than the default 'model' is wanted.

```
# Compile DLL
oc.compile_dll('CSTR_CSTR_Opt', target='ipopt')
```

1.1.3.4. Loading the Shared Object file (DLL)

The DLL file is loaded using the class Model from which the JMI Model interface can be reached.

```
# Import Model
from jmodelica.jmi import Model
# Load dll file and create Model object
model = Model('CSTR_CSTR_Opt')
```

The model object can now be used to manipulate parameters and for optimization and simulation.

2. Simulation of models

2.1. Simulation of an electrical circuit

This example focus on how to use the high-level simulation functionality on a model of an electric circuit. The model is depicted in Figure (RLC.png) 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. This model is written in Modelica code and saved in the file RLC_Circuit.mo and is depicted in below.

Figure 3.1. Electric Circuit
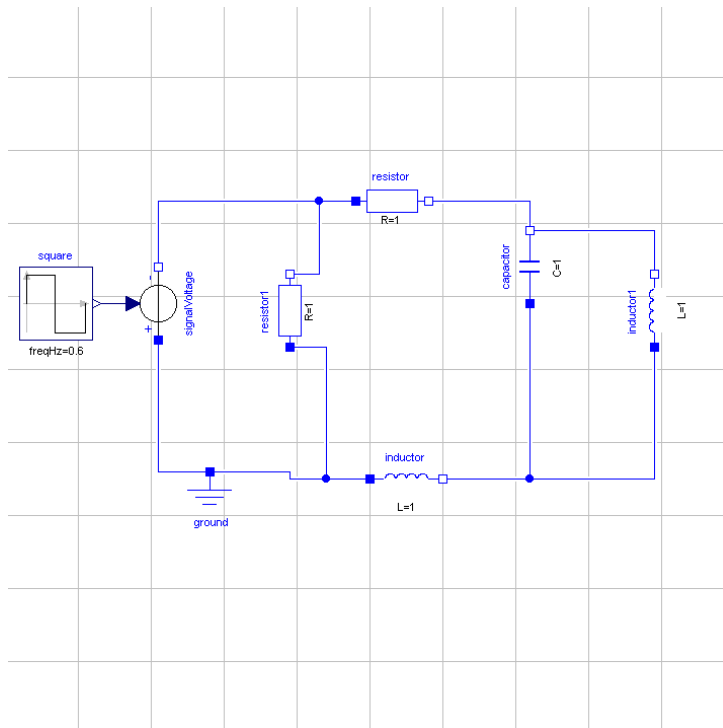
To use the functionality provided by the JModelica.org platform they first have to be imported into the Python script. So we start by importing the following:

```
from jmodelica import simulate
import pylab as P
```

The method 'simulate' is the high-level simulation method and the 'pylab' package is used here for plotting.

Next, we need to provide the 'simulate' method with information about which model we would like to simulate and where it is stored. We also need information about the simulation interval. The information is then passed down in the following way,

```
res_object = simulate(model='RLC_Circuit_Square', file_name='RLC_Circuit.mo',
```

```
alg_args={'start_time':0.0,'final_time':20.0,'num_communication_points':0})
```

The return argument from 'simulate' is a result object where the simulation result can be retrieved using the method `get_result_data`. Here 'alg_args' are the arguments for the algorithm stored in a dictionary. The 'num_communication_points' represents the number of communication points stored by the algorithm. The default is 500 points and when set to zero (0), the internal steps calculated by the algorithm are stored. If the problem requires that the default options in the specific solver needs to be changed, they should be passed down in a dictionary called 'solver_args'. Typically these options can be the tolerances. Using the default simulation package, Assimulo, information regarding which algorithms are supported and the solver arguments can be found here, http://www.jmodelica.org/assimulo . The default solver is IDA and for a selection of the solver arguments to IDA, see the table below.

Table 3.1. Selection of solver arguments for IDA

Argument	Option
suppress_alg (Suppress the algebraic variables on the error test)	Boolean flag
initstep (The initial step-size)	Positive float
maxorder (The maximum order used)	Integer of max 5
maxh (Maximum step-size)	Positive float
atol (Absolute Tolerance)	Array of floats or Float
rtol (Relative Tolerance)	Float

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

```
res = res_object.get_result_data()
square_y = res.get_variable_data('square.y')
resistor_v = res.get_variable_data('resistor.v')
inductor1_i = res.get_variable_data('inductor1.i')
```

And then plotted with the help from pylab,

```
P.plot(square_y.t, square_y.x, resistor_v.t, resistor_v.x, inductor1_i.t, inductor1_i.x)
P.legend(('square.y','resistor.v','inductor1.i'))
P.show()
```

The simulation result is shown in the figure below:

Figure 3.2. Simulation result
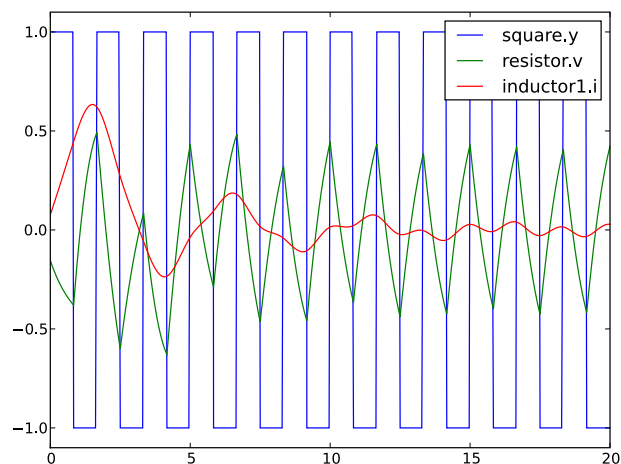
2.2. Simulation and parameter sweeps

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

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

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

  // The control signal
  input Real u;

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

Notice that the initial values of the states are parametrized by the parameters $x1\_0$ and $x2\_0$. Next, copy the Modelica code above into a file VDP.mo and save it in your working directory. Also, create a Python script file and name it vdp_pp.py. Start by copying the commands:

```
import numpy as N
import pylab as p
from jmodelica.compiler import ModelicaCompiler
from jmodelica import simulate
```

into the Python file. Compile and load the model:

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

# Create a Modelica compiler
mc = ModelicaCompiler()

# Compile and load model
model = mc.compile_model(model_name,mofile,target='ipopt')
```

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

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

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

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

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

```
for i in range(N_points):
    # Set initial conditions in model
    model.set_value('x1_0',x1_0[i])
    model.set_value('x2_0',x2_0[i])
    # Simulate
    sim_res = simulate(model,alg_args={'final_time':20})
    # Get simulation result
```

```
    res = sim_res.result_data
    x1=res.get_variable_data('x1')
    x2=res.get_variable_data('x2')
    # Plot simulation result in phase plane plot
    p.plot(x1.x, x2.x,'b')
p.grid()
p.show()
```

You should now see the following plot:

Figure 3.3. Simulation result-phase plane
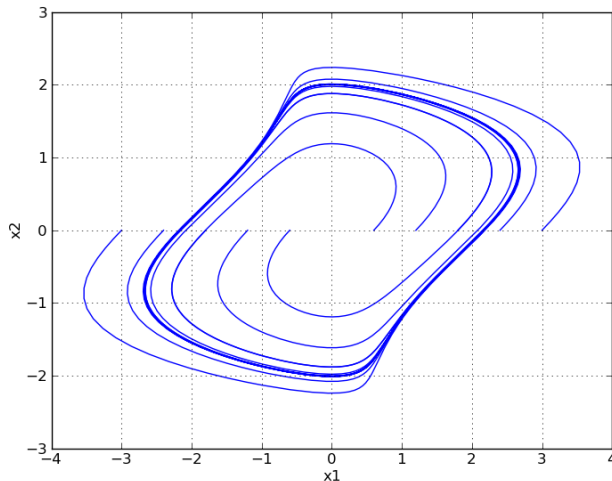
3. Solving optimal control problems

3.1. The van der Pol Oscillator

We consider the following Optimica model:

```
optimization VDP_Opt (objective = cost(finalTime),
                      startTime = 0,
                      finalTime = 20)

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

  // The control signal
  input Real u;

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

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

Create a new file named VDP_Opt.mo and save it in you working directory. Next, create a Python script file and a write (or copy paste) the following commands:

```
# Import the optimize function
from jmodelica import optimize

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

Next, we call the 'optimize' function which encapsulates operations for compiling, loading, executing the optimization algorithm, and loading the result from file:

```
res_object = optimize("VDP_Opt", "VDP_Opt.mo")
```

In this case, we use the default settings for the optimization algorithm and provide only the name of the Optimica class (the first argument) and the name of the file (VDP.mo). The return object is a result object which contains a reference to a jmodelica.Model object representing the compiled model and also the optimization result. To access the optimization result and the optimal trajectories:

```
res = res_object.get_result_data() #Get the result data
x1=res.get_variable_data('x1')
x2=res.get_variable_data('x2')
u=res.get_variable_data('u')
```

The return arguments are objects of the Python class jmodelica.io.Trajectory, which has two fields: 't' which represents the time vector and 'x' which represents the trajectory vector. t and x are both numpy arrays of the same length. Using the matplotlib package, we can visualize the optimization result:

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

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

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

You should now see the optimization result as shown below.

Figure 3.4. Van der Pol optimization result.
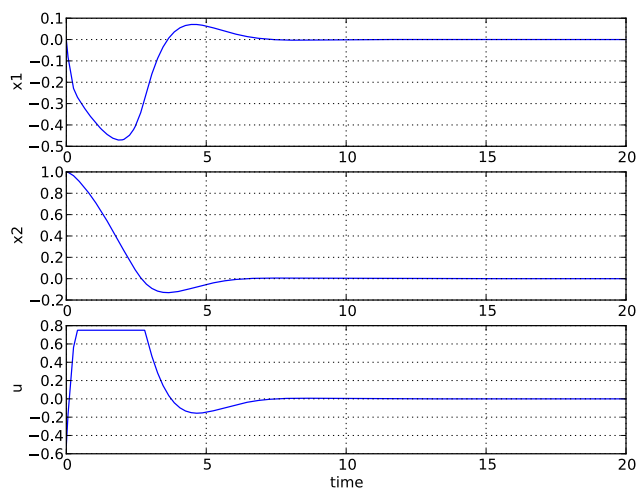
3.2. The Hicks Ray Continuously Stirred Reactor (CSTR)

This example 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:

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

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

This tutorial will cover the following topics:

- How to solve a DAE initialization problem. The initialization model have equations specifying that all derivatives should be identically zero, which implies that a stationary solution is obtained. Two stationary points, corresponding to different inputs, are computed. We call the stationary points A and B respectively. Point A corresponds to operating conditions where the reactor is cold and the reaction rate is low, whereas point B corresponds to a higher temperature where the reaction rate is high. For more information about the DAE initialization algorithm, see the JMI API documentation.

- An optimal control problem is solved where the objective is to transfer the state of the system from stationary point A to point B. The challenge is to ignite the reactor while avoiding uncontrolled temperature increase. It is also demonstrated how to set parameter and variable values in a model. More information about the simultaneous optimization algorithm can be found at JModelica.org API documentation.

- The optimization result is saved to file and then the important variables are plotted.

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

Start the tutorial by creating a working directory and copy the file $JMODELICA_HOME/Python/jmodelica/examples/files/CSTR.mo to your working directory. An on-line version of CSTR.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.

3.2.1. 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 past them either directly into you Python shell or, preferably, into your Python script file.

```
import os.path
from jmodelica import initialize
from jmodelica import simulate
from jmodelica import optimize

import jmodelica.jmi as jmi
from jmodelica.compiler import OptimicaCompiler

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

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 in the tutorial on Compilation of models.

```
# Create a Modelica compiler instance
oc = OptimicaCompiler()

# Compile the stationary initialization model into a DLL and load it
init_model = oc.compile_model("CSTR.CSTR_Init", "CSTR.mo", target='ipopt')
```

At this point, you may open the file CSTR.mo, 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:

```
In [8]: help(oc.compile_model)
```

3.2.2. 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_value('Tc',Tc_0_A)

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

# Store stationary point A
c_0_A = init_result.get_variable_data('c').x[0]
T_0_A = init_result.get_variable_data('T').x[0]

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

Notice how the function set_value is used to set the value of the control input. The initialization algorithm is invoked by calling the function 'initialize', which returns a result object where the initialization result is accessed with 'get_result_data' and stored in 'init_result'. The 'initialize' function relies on the algorithm Ipopt for computing the solution of the initialization problem. The values of the states corresponding to grade 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' function 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_value('Tc',Tc_0_B)

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

# Store stationary point B
c_0_B = init_result.get_variable_data('c').x[0]
T_0_B = init_result.get_variable_data('T').x[0]

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

We have now computed two stationary points for the system based on constant control inputs.

3.2.3. Solving an optimal control problem

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

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

subject to

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

$$T(t) \le 350$$

and is expressed in Optimica format in the class CSTR.CSTR_Opt in the CSTR.mo 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.mo file, a step input is filtered through a first order filter in order to generate a smooth input for the CSTR system. The filtering is done in order not to excite unstable modes of the system, and in particular to avoid sudden ignition. Notice also 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.

Start by creating an input trajectory to be passed to the simulator:

```
# Create the time vector
t = N.linspace(1,150.,100)
# Create the input vector from the target input value. The
# target input value is here increased in order to get a
# better initial guess.
u = (Tc_0_B+35)*N.ones(N.size(t,0))
# Create a matrix where the first column is time and the second column represents
# the input trajectory.
u_traj = N.transpose(N.vstack((t,u)))
```

Next, compile the model and set model parameters:

```
# Compile the optimization initialization model and load the DLL
init_sim_model = oc.compile_model("CSTR.CSTR_Init_Optimization", "CSTR.mo", target='ipopt')

# Set model parameters
init_sim_model.set_value('cstr.c_init',c_0_A)
init_sim_model.set_value('cstr.T_init',T_0_A)
init_sim_model.set_value('Tc_0',Tc_0_A)
init_sim_model.set_value('c_ref',c_0_B)
init_sim_model.set_value('T_ref',T_0_B)
init_sim_model.set_value('Tc_ref',u[0])
```

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

```
sim_result_object = simulate(init_sim_model,alg_args={'start_time':0.,'final_time':150.,
                                              'input_trajectory':u_traj})
res = sim_result_object.get_result_data()
```

The function 'simulation' first computes consistent initial conditions and then simulates the model in the interval 0 to 150 seconds with the input trajectory specified by 'u_traj'. Notice that the arguments to the simulation function is specified in a Python dictionary. Take a moment to read the interactive help for the 'simulate' function.

The simulation result object is returned and to retrieve the simulation data use the method 'get_result_data', from which you may now retrieve trajectories for plotting:

```
# Extract variable profiles
c_init_sim=res.get_variable_data('cstr.c')
T_init_sim=res.get_variable_data('cstr.T')
Tc_init_sim=res.get_variable_data('cstr.Tc')

# Plot the results
plt.figure(1)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(c_init_sim.t,c_init_sim.x)
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(T_init_sim.t,T_init_sim.x)
plt.grid()
```

```
plt.ylabel('Temperature')

plt.subplot(313)
plt.plot(Tc_init_sim.t,Tc_init_sim.x)
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

Look at the plots and make sure you understand the effect of the filter. Think about alternative, better ways to chose the input profile. Also, try to increase the value 35 that was added to the target input: how much can you increase this value without experiencing sudden ignition of the reactor?

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

```
cstr = oc.compile_model("CSTR.CSTR_Opt", "CSTR.mo", target='ipopt')
```

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:

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

cstr.set_value('cstr.c_init',c_0_A)
cstr.set_value('cstr.T_init',T_0_A)
```

In order to solve the optimization problem, we need to specify the mesh on which the optimization is performed. The simultaneous optimization algorithm is based on a collocation method that 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].

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.mo 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:

```
# Initialize the mesh
n_e = 100 # Number of elements
hs = N.ones(n_e)*1./n_e # Equidistant points
n_cp = 3; # Number of collocation points in each element

opt_result_object = optimize(cstr,alg_args={'n_e':n_e,'hs':hs,'n_cp':n_cp,'init_traj':res})
res = opt_result_object.get_result_data()
```

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. Again, the arguments to the algorithm (number of elements, number of collocation points, element length vector and initial guess object) are given in a Python dictionary. 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.get_variable_data('cstr.c')
```

```
T_res=res.get_variable_data('cstr.T')
Tc_res=res.get_variable_data('cstr.Tc')

c_ref=res.get_variable_data('c_ref')
T_ref=res.get_variable_data('T_ref')
Tc_ref=res.get_variable_data('Tc_ref')
```

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

```
plt.figure(1)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(c_res.t,c_res.x)
plt.plot(c_ref.t,c_ref.x,'--')
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(T_res.t,T_res.x)
plt.plot(T_ref.t,T_ref.x,'--')
plt.grid()
plt.ylabel('Temperature')

plt.subplot(313)
plt.plot(Tc_res.t,Tc_res.x)
plt.plot(Tc_ref.t,Tc_ref.x,'--')
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

Your should now see a plot as the one below:

Figure 3.5. Optimization result
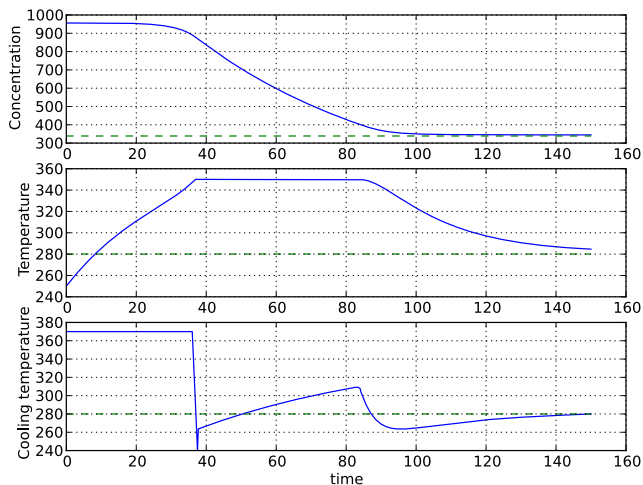
Take a minute to analyze the optimal profiles and to answer the following questions:

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?

3.2.4. 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 input trajectory
t = Tc_res.t
u = Tc_res.x
u_traj = N.transpose(N.vstack((t,u)))

# Comile the Modelica model first to C code and
# then to a dynamic library
sim_model = oc.compile_model("CSTR.CSTR","CSTR.mo",target='ipopt')

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

sim_result_object = simulate(sim_model,compiler='optimica',
                             alg_args={'start_time':0.,'final_time':150.,
                                       'input_trajectory':u_traj})
res = sim_result_object.get_result_data()
```

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

```
# Extract variable profiles
c_sim=res.get_variable_data('c')
T_sim=res.get_variable_data('T')
Tc_sim=res.get_variable_data('Tc')

# Plot the results
plt.figure(3)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(c_res.t,c_res.x,'--')
plt.plot(c_sim.t,c_sim.x)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(T_res.t,T_res.x,'--')
plt.plot(T_sim.t,T_sim.x)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Temperature')

plt.subplot(313)
plt.plot(Tc_res.t,Tc_res.x,'--')
plt.plot(Tc_sim.t,Tc_sim.x)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

You should now see a plot similar to:

Figure 3.6. Simulated system response
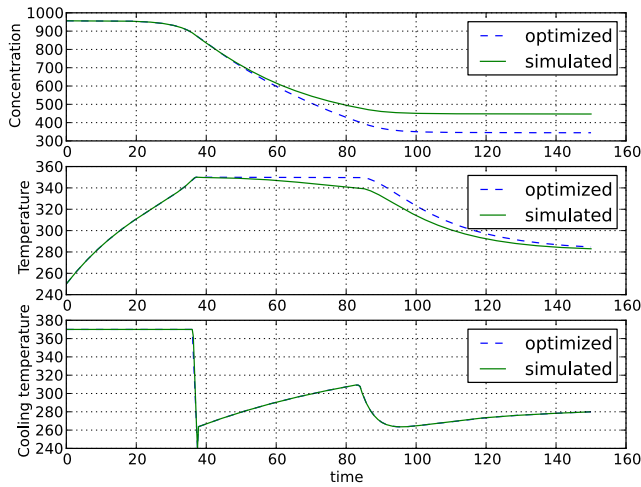
Discuss why the simulated trajectories differs from the optimized counterparts.

3.2.5. 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.

3.2.6. 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.

4. Solving parameter estimation problems

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

Figure 3.7. A schematic figure of the quadruple tank process.
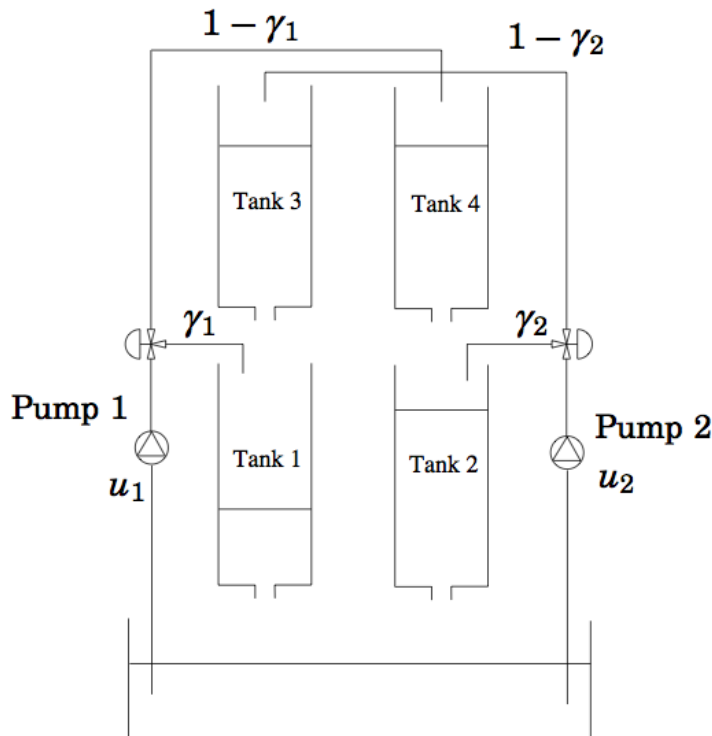
The dynamics of the system is given by the differential equations:

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

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

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

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

Where the parameter values are given in the table below:

Table 3.2. Quadruple tank parameter values

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

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

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

```
from scipy.io.matlab.mio import loadmat
import matplotlib.pyplot as plt
import numpy as N
from jmodelica import simulate
from jmodelica import optimize
from jmodelica.compiler import OptimicaCompiler
```

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:

Figure 3.8. Measured state profiles
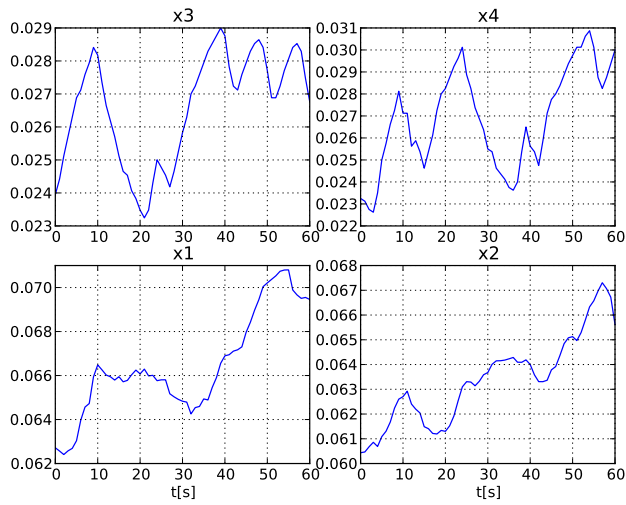
Figure 3.9. Input profiles used in estimation experiment
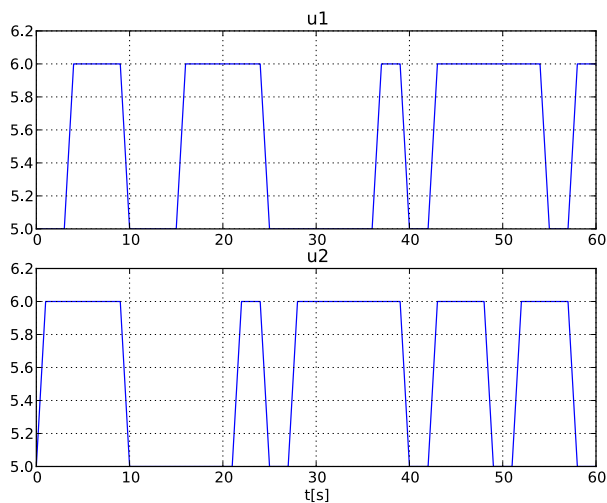
In order to evaluate the accuracy of nominal model parameter values, start by simulating the model, assuming that the start values of the states are given by the state measurement at the start of the experiment. This assumption can be expressed in the model:

```
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:

```
# Simulate model response with nominal parameters
res_sim = simulate('QuadTankPack.Sim_QuadTank',
                   'QuadTankPack.mo',
                   compiler='optimica',
                   alg_args={'input_trajectory':u,
                             'start_time':0.,
                             'final_time':60})
```

The simulation result can now be extracted:

```
# Load simulation result
x1_sim = res_sim.result_data.get_variable_data('qt.x1')
x2_sim = res_sim.result_data.get_variable_data('qt.x2')
x3_sim = res_sim.result_data.get_variable_data('qt.x3')
x4_sim = res_sim.result_data.get_variable_data('qt.x4')
u1_sim = res_sim.result_data.get_variable_data('u1')
u2_sim = res_sim.result_data.get_variable_data('u2')
```

and then plotted:

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

plt.figure(2)
plt.subplot(2,1,1)
plt.plot(u1_sim.t,u1_sim.x,'r')
plt.subplot(2,1,2)
plt.plot(u2_sim.t,u2_sim.x,'r')
plt.show()
```

The plot below shows the result of the simulation.

Figure 3.10. Simulation result for the nominal model
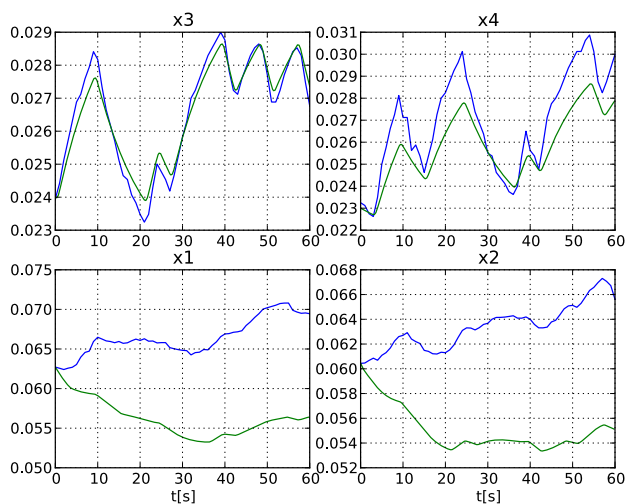
Here, the simulated profiles are given by the green curves. Clearly, there is a mismatch in the response, especially for the two lower tanks. Think about why the model does not match the data, i.e., which parameters may have wrong values.

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

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

   // Initial tank levels
  parameter Modelica.SIunits.Length x1_0 = 0.06255;
  parameter Modelica.SIunits.Length x2_0 = 0.06045;
  parameter Modelica.SIunits.Length x3_0 = 0.02395;
  parameter Modelica.SIunits.Length x4_0 = 0.02325;

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

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

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

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

```
# Create Optimica compiler
oc = OptimicaCompiler()

# Compile model
qt_par_est = oc.compile_model("QuadTankPack.QuadTank_ParEst",
                              "QuadTankPack.mo",target='ipopt')
```

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_value("t_meas["+`i+1`+"]",t_meas[i])
    qt_par_est.set_value("y1_meas["+`i+1`+"]",y1_meas[i])
```

```
    qt_par_est.set_value("y2_meas["+`i+1`+"]",y2_meas[i])
```

We are now ready to solve the optimization problem:

```
# Number of element in collocation algorithm
n_e = 100
# Normalized element lengths
hs = N.ones(n_e)/n_e
# Number of collocation points
n_cp = 3

# Solve parameter optimization problem
res_opt = optimize(qt_par_est,alg_args={"n_e":n_e,"n_cp":3, \
                                         "result_mesh":"element_interpolation",
                                         "hs":hs})
```

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_opt.result_data.get_variable_data("qt.a1")
a2_opt = res_opt.result_data.get_variable_data("qt.a2")

# Print optimal parameter values
print('a1: ' + str(a1_opt.x[-1]*1e4) + 'cm^2')
print('a2: ' + str(a2_opt.x[-1]*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_opt.result_data.get_variable_data("qt.x1")
x2_opt = res_opt.result_data.get_variable_data("qt.x2")
x3_opt = res_opt.result_data.get_variable_data("qt.x3")
x4_opt = res_opt.result_data.get_variable_data("qt.x4")
u1_opt = res_opt.result_data.get_variable_data("qt.u1")
u2_opt = res_opt.result_data.get_variable_data("qt.u2")

plt.figure(1)
plt.subplot(2,2,1)
plt.plot(x3_opt.t,x3_opt.x,'k')
plt.subplot(2,2,2)
plt.plot(x4_opt.t,x4_opt.x,'k')
plt.subplot(2,2,3)
plt.plot(x1_opt.t,x1_opt.x,'k')
plt.subplot(2,2,4)
plt.plot(x2_opt.t,x2_opt.x,'k')
plt.show()
```

You will now see a plot looking like:

Figure 3.11. State profiles corresponding to estimated values of a1 and a2.
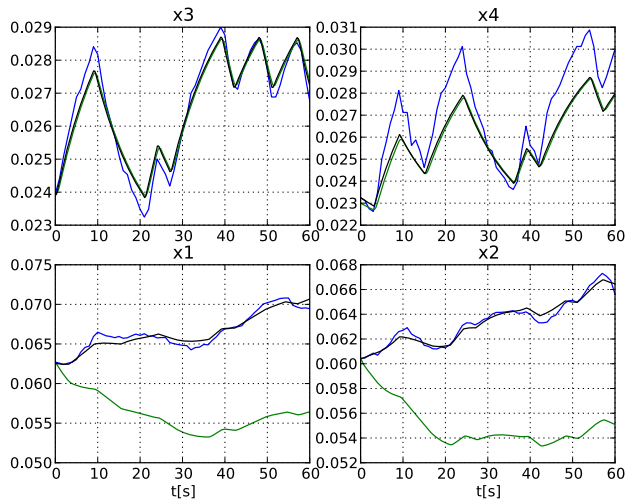
The profiles corresponding to the estimated values of a1 and a2 are shown in black curves. As can be seen, the match between the model response and the measurement data has been significantly increased. Is the behavior of the model consistent with the estimated parameter values?

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
qt_par_est2 = oc.compile_model("QuadTankPack.QuadTank_ParEst2",
                               "QuadTankPack.mo",target='ipopt')

# 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_value("t_meas["+`i+1`+"]",t_meas[i])
    qt_par_est2.set_value("y1_meas["+`i+1`+"]",y1_meas[i])
    qt_par_est2.set_value("y2_meas["+`i+1`+"]",y2_meas[i])
    qt_par_est2.set_value("y3_meas["+`i+1`+"]",y3_meas[i])
    qt_par_est2.set_value("y4_meas["+`i+1`+"]",y4_meas[i])

# Solve parameter estimation problem
res_opt2 = optimize(qt_par_est2,alg_args={"n_e":n_e,"n_cp":3, \
                                          "result_mesh":"element_interpolation","hs":hs})
```

Next, we print the optimal parameter values:

```
# Get optimal parameter values
a1_opt2 = res_opt2.result_data.get_variable_data("qt.a1")
a2_opt2 = res_opt2.result_data.get_variable_data("qt.a2")
a3_opt2 = res_opt2.result_data.get_variable_data("qt.a3")
a4_opt2 = res_opt2.result_data.get_variable_data("qt.a4")

# Print optimal parameter values
print('a1:' + str(a1_opt2.x[-1]*1e4) + 'cm^2')
print('a2:' + str(a2_opt2.x[-1]*1e4) + 'cm^2')
print('a3:' + str(a3_opt2.x[-1]*1e4) + 'cm^2')
print('a4:' + str(a4_opt2.x[-1]*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.result_data.get_variable_data("qt.x1")
x2_opt2 = res_opt2.result_data.get_variable_data("qt.x2")
x3_opt2 = res_opt2.result_data.get_variable_data("qt.x3")
x4_opt2 = res_opt2.result_data.get_variable_data("qt.x4")
u1_opt2 = res_opt2.result_data.get_variable_data("qt.u1")
u2_opt2 = res_opt2.result_data.get_variable_data("qt.u2")

# Plot
plt.figure(1)
plt.subplot(2,2,1)
plt.plot(x3_opt2.t,x3_opt2.x,'r')
plt.subplot(2,2,2)
plt.plot(x4_opt2.t,x4_opt2.x,'r')
plt.subplot(2,2,3)
plt.plot(x1_opt2.t,x1_opt2.x,'r')
plt.subplot(2,2,4)
plt.plot(x2_opt2.t,x2_opt2.x,'r')
plt.show()
```

The resulting plot should look like:

Figure 3.12. State profiles corresponding to estimated values of a1, a2, a3 and a4
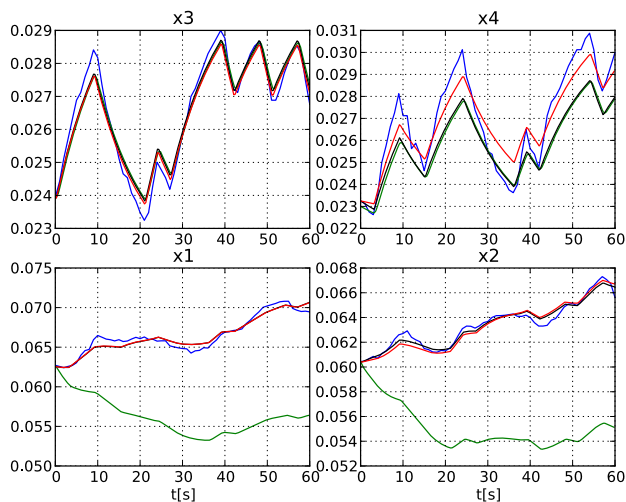
The red curves represent the case where a1, a2, a3 and a4 has been estimated.

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

5. Working with file I/O

In this tutorial you will learn how to load simulation/optimization results.

5.1. I/O functionality

The module `jmodelica.io` provides useful functions for exporting and loading simulation or optimization results from Dymola. The result files can be in Dymola textural or Dymola binary format. The variable data is saved

together with the variable names which makes it possible to load the result files and match result data with a specific variable.

5.2. Loading result data

To load a result data file saved using the export functionality jmodelica.io.export_result_dymola the class ResultDymolaTextual in the same module, jmodelica.io, is used. The result object can then be used to retrieve data for a specific variable.

```
# Load the CSTR results
res = jmodelica.io.ResultDymolaTextual('CSTR_CSTR_Opt_result.txt')

# Get variable data for T_ref
res.get_variable_data('T_ref').x
>> array([ 280.099198,  280.099198])
```

There is a similar function for retrieving results from a file in Dymola binary format.

6. Setting and saving model parameters

This tutorial shows how to set model parameters and how to load and save parameter data from/to XML files.

6.1. Model parameter XML files

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

- <model class name>.xml

- <model class name>_values.xml

The parameter meta data is saved in <model class name>.xml and the parameter values in <model class name>_values.xml. The name of the parameter is used to map a parameter value in the values file to a parameter specification.

6.2. Get and set value

The model parameters can be accessed with via the jmi.Model interface. It is possible to look at the whole vector of, for example, all real parameters in the model or one specific parameter. Accessing one specific parameter requires that the parameter name is known.

The following code example assumes the CSTR model has been compiled and the DLL file loaded in jmi.Model.

```
# Get independent real parameter vector
cstr_model.get_real_pi()
>> array([  1.66666667e-03,   1.00000000e+03,   1.66666667e-03,
           3.50000000e+02,   2.19000000e-01,   1.20000000e+09,
           8.75000000e+03,   9.15600000e+02,   1.00000000e+03,
           2.39000000e+02,  -5.00000000e+04,   1.00000000e+02,
           1.00000000e+03,   3.50000000e+02,   5.00000000e+02,
           3.20000000e+02,   3.00000000e+02,   1.00000000e+00,
           1.00000000e+00,   1.00000000e+00,   0.00000000e+00,
           1.50000000e+02])

# Get independent parameter c_ref
cstr_model.get_value('c_ref')
>> 500.0

# Set independent parameter
cstr_model.set_value('c_ref', 450)
# c_ref has now changed
cstr_model.get_value('c_ref')
>> 450.0
```

6.3. Loading from and saving to XML

6.3.1. Loading XML values file

It is possible to load the values from an XML file as is done automatically when the `jmi.Model` object was first created. If, for example, there were many local changes to parameters it could be desirable to reset everything as it was from the beginning.

```
# Set parameter
cstr_model.set_value('c_ref', 450)
cstr_model.get_value('c_ref')
>> 450.0
# Load values XML file
cstr_model.load_parameters_from_XML()
# Parameter has now been reset
cstr_model.get_value('c_ref')
>> 500.0
```

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

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

6.3.2. Writing to XML values file

Setting a parameter value with `Model.set_value` only changes the value in the vector loaded when `jmi.Model` was created, which means that they will not be saved. To save all changes made to parameters in a model, the values have to be written to the XML values file.

```
# Set parameter
cstr_model.set_value('c_ref', 450)
# Save parameters to values XML
cstr_model.write_parameters_to_XML()
# Changed parameter has now been saved in XML file
cstr_model.get_value('c_ref')
>> 450.0
```

If `write_parameters_to_XML()` is called without arguments the values will be written to the XML file which was created when the model was compiled (following the name conventions mentioned above). It is also possible to save the changes in a new XML file. This is quite convenient since different parameter value settings can easily be saved and loaded in the model.

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

Chapter 4. FMI Interface

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

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

1. Overview of JModelica.org FMI Python package

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

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

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

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

Table 4.1. Conversion table.

FMI C-Interface	JModelica.org FMI Python Interface
const char* fmiGetModelTypesPlatform()	string FMIModel.model_types_platform
const char* fmiGetVersion()	string FMIModel.version
fmiComponent fmiInstantiateModel(...)	FMIModel.__init__()
void fmiFreeModelInstance(fmiComponent c)	FMIModel.__del__()
fmiStatus fmiSetDebugLogging(...)	none FMIModel.set_debug_logging(flag)
fmiStatus fmiSetTime(...)	FMIModel.time
fmiStatus fmi(Get/Set)ContinuousStates(...)	FMIModel.continuous_states
fmiStatus fmiCompletedIntegratorStep(...)	boolean FMIModel.completed_integrator_step()
fmiStatus fmiSetReal/Integer/Boolean/String(...)	none FMIModel.set_real/integer/boolean/string(valueref,values)
fmiStatus fmiInitialize(...)	none FMIModel.initialize() (also sets the start attributes)
struct fmiEventInfo	FMIModel.get_event_info()
fmiStatus fmiGetDerivatives(...)	numpy.array FMIModel.get_derivatives()

FMI C-Interface	JModelica.org FMI Python Interface
fmiStatus fmiGetEventIndicators(...)	numpy.array FMIModel.get_event_indicators()
fmiStatus fmiGetReal/Integer/Boolean/String(...)	numpy.array FMIModel.get_real/integer/boolean/ string(valueref)
fmiStatus fmiEventUpdate(...)	none FMIModel.event_update()
fmiStatus fmiGetNominalContinuousStates(...)	FMIModel.nominal_continuous_states
fmiStatus fmiGetStateValueReferences(...)	numpy.array FMIModel.get_state_value_references()
fmiStatus fmiTerminate(...)	FMIModel.__del__()

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

```
FMIModel.get_log()
```

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

```
FMIModel.get_real?
```

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

2. Examples

In the next two sections, it will be shown how to use the JModelica.org platform both for simulation of an FMU using the native Python interface and how to simulate an FMU using JModelica.org's simulation package, Assimulo.

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.

2.1. Simulation using the native FMI interface

This example shows how to use the native JModelica.org FMI interface for simulation of an FMU. The FMU that is to be simulated is the bouncing ball example from Qtronics FMU SDK (http://www.qtronic.de/en/fmusdk.html). This example is written similar to the example in the documentation of the 'Functional Mock-up Interface for Model Exchange' version 1.0 (http://www.functional-mockup-interface.org/). The bouncing ball model is to be simulated using the explicit Euler method with event detection.

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

The bouncing ball consists of two equations,

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

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

$$h > 0$$

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

$$v_a = -ev_b$$

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

2.1.1. Implementation

Start by importing the necessary modules,

```
        import numpy as N
        import pylab as P #Used for plotting
        from jmodelica.fmi import FMIModel #The FMI Interface
```

Next, the FMU is to be loaded and initialized,

```
        #Load the FMU by specifying the fmu together with the path.
        bouncing_fmu = FMIModel('/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, `FMIModel` tries to find the starting time in the XML-file structure 'default experiment' and if successful starts the simulation from that time. Also if the XML-file does not contain any information about the default experiment the simulation is started from time zero.

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

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

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

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

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

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

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

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

            #Set the time
            bouncing_fmu.time = time

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

If an event occurred, we enter the iteration loop where we loop until the solution of the new states have converged. During this iteration we can also retrieve the intermediate values with the normal `get` methods. At this point `eInfo` contains information about the changes made in the iteration. If the state values have changed, they are retrieved. If the state references have changed, meaning that the state variables no longer have the same meaning

as before by pointing to another set of continuous variables in the model, for example in the case with dynamic state selection, new absolute tolerances are calculated with the new nominal values. Finally the model is checked for a new time event.

```
event_ind = event_ind_new

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

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

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

After the loop have finished, by reaching the final time, we plot the simulation results
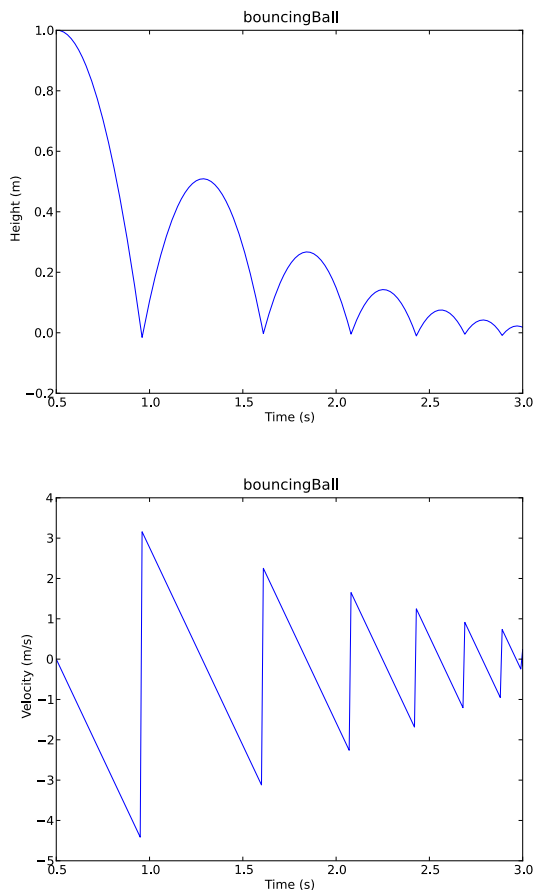
```
#Plot the height
P.figure(1)
P.plot(t_sol,N.array(sol)[:,0])
P.title(bouncing_fmu.get_name())
P.ylabel('Height (m)')
P.xlabel('Time (s)')
#Plot the velocity
P.figure(2)
P.plot(t_sol,N.array(sol)[:,1])
P.title(bouncing_fmu.get_name())
P.ylabel('Velocity (m/s)')
P.xlabel('Time (s)')
P.show()
```

and the figure below shows the results.

Figure 4.1. Simulation result

2.2. Example using a compiled FMU

This example will show how to use the JModelica.org's FMI-interface together with its simulation package, Assimulo. The FMU to be simulated is the full Robot from the Modelica standard library (3.1) where it is located in `Mechanics.MultiBody.Examples.Systems.RobotR3`. It consists of brakes, motors, gears and path planning. The model consists of 36 continuous states and around 700 algebraic variables together with 98 event functions and also a few thousand constants/parameters. The FMU was generated using Dymola 7.4.

Figure 4.2. Full Robot
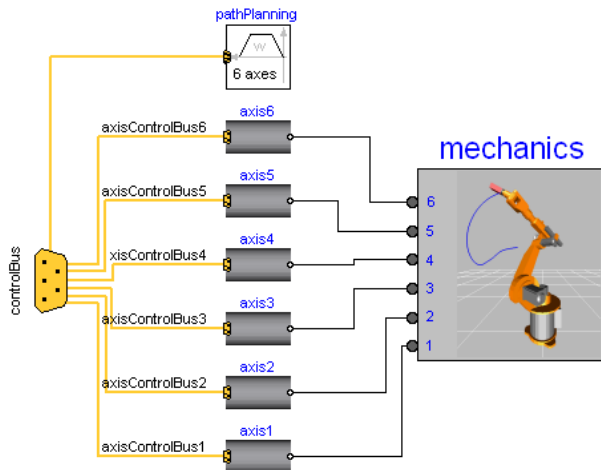
2.2.1. Implementation

We start by importing the necessary method and module,

```
from jmodelica import simulate
import pylab as P
```

The `simulate` method is a high-level method for simulation where the input are the model of interest together with options for the algorithm and options for the solver. We are interested in simulating the Robot from time 0.0 to 1.8 using 1000 communication points. This information is specified in a dictionary called `alg_args` where all the algorithms options are specified. For the solver options there is another dictionary called `solver_args`, where for example the tolerances can be specified. Information regarding the solver arguments can be found here, http://www.jmodelica.org/assimulo and a selection of solver arguments can be found in the table below. Currently only the solver CVode is supported.

Table 4.2. Selection of solver arguments for CVode

Argument	Option
discr (Discretization method)	Adams / BDF (string), (default BDF)
iter (Iteration method)	Newton / FixedPoint (string), (default Newton)
maxord (The maximum order)	Positive float (max 5 for BDF and 12 for Adams)
write_cont (Turn on continuous writing of result)	Boolean (default True)
rtol (Relative Tolerance)	Float (default value in XML-file or 1.0e-4)
atol (Absolute Tolerance(s))	Float or Array of Floats (default 0.01*rtol*nominal_continuous_states)

In the code below, the simulation options are passed on to `simulate`

```
simRobot = simulate('Robot.fmu', alg_args={'final_time':1.8,
                        'num_communication_points':1000})
```

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

The simulation result can then be retrieved from the result object `simRobot` by calling the method `get_result_data()`.

```
res = sim_res.get_result_data()
```

To retrieve data about a variable from the result data (res), use the method `get_variable_data` together with the name of the variable.

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

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

```
P.plot(dq1.t,dq1.x,dq6.t,dq6.x)
P.legend(['der(mechanics.q[1])','der(mechanics.q[6])'])
P.xlabel('Time (s)')
P.ylabel('Joint Velocity (rad/s)')
P.title('Full Robot')
P.show()
```

Below is the resulting figure together with a comparison from the simulation result generated by Dymola.
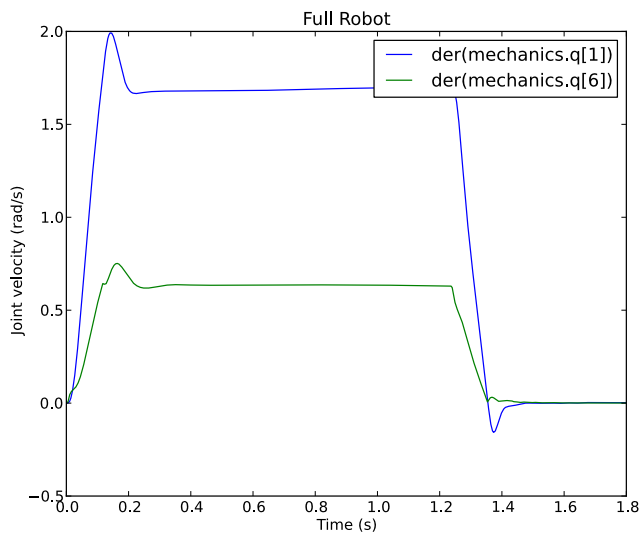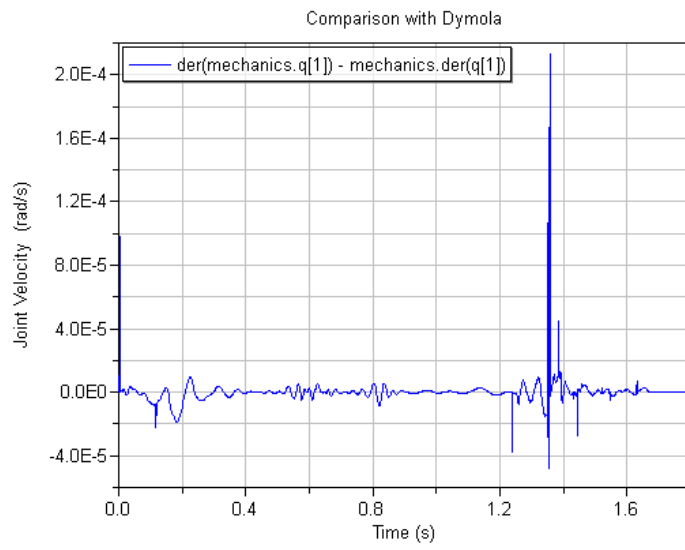
Figure 4.3. Full Robot Results

Figure 4.4. Comparison with Dymola

Chapter 5. High-level functions: initialize, simulate and optimize

The purpose of the high-level functions is to provide a flexible user interface to initializing, simulating and optimizing a model. In its simplest form, the functions can be used with all arguments set to default values requiring very few user inputs. For more complex usage there are a number of options that can be set which enables more user control over the course of events.

1. Overview

This section will give a brief overview to the high-level functions, which algorithms are available at the moment, how to change settings and how the return argument is composed. The section ends with a short simulation example, but the general principle of the example also applies for the `initialize` and `optimize` functions.

1.1. Importing the high-level functions

The `initialize`, `simulate` and `optimize` functions are all located in the package `jmodelica`. There are two main ways of making them available in the Python shell or in a script

- **Import the package jmodelica.** This will create a new namespace with all attributes from the `jmodelica` package. To use any of the functions, the package name must then be used as a prefix.

```
# import the optimize function by importing jmodelica
import jmodelica

# optimize is now available
jmodelica.optimize
<function optimize at 0x05FE8970>

# so is simulate and initialize
jmodelica.simulate
<function simulate at 0x05FF4FB0>
jmodelica.initialize
<function initialize at 0x05FF6030>
```

- **Import a specific function.** Using the `from` statement a specific function will be imported in the current namespace.

```
# import the simulate function from jmodelica
from jmodelica import simulate

# type simulate and hit enter
simulate
<function simulate at 0x05FF8170>
```

Note that neither `optimize` nor `initialize` is available now, they must also be imported explicitly.

```
# type optimize and hit enter
NameError: name 'optimize' is not defined

# import optimize
from jmodelica import optimize
optimize
<function optimize at 0x05FEDAF0>
```

1.2. Default arguments

1.2.1. Function arguments

The only required input to any of the high-level functions is the model object, the rest of the arguments all have default values. The default values are listed in the docstring of the function. Using the interactive help in the Python

shell or looking at the API documentation will display the docstring for a certain function. It is then possible to
see what arguments are available and their default values.

```
# docstring for simulate
Docstring:
    Compact function for model simulation.

The intention with this function is to wrap model compilation, creation of
a model object and simulation in one function call. The simulation
method depends on which algorithm is used, this can be set with the
function argument 'algorithm'. Arguments for the algorithm and solver are
passed as dicts. Which arguments that are valid depends on which algorithm
is used, see the algorithm implementation in algorithm_drivers.py for details.

The default algorithm for this function is AssimuloAlg.

The simplest way of using the function is to pass the model name and path
to the model file (a jmi.Model is enough if model is already compiled) and
use the default values for all other arguments.

Parameters::

    model --
        Model object or model name (supply model name if model should be
        (re)compiled, then mo-file must also be provided)
    file_name --
        Path to model file or list of paths to model files.
        Default: empty string (no compilation)
    compiler --
        Set compiler that model should be compiled with, 'modelica' or
        'optimica'.
        Default: 'modelica'
    compiler_target --
        Target argument to compiler.
        Default: 'ipopt'
    compiler_options --
        Dict with options for the compiler (see options.xml for possible
        values).
        Default: empty dict
    algorithm --
        The algorithm which will be used for the simulation is
        specified by passing the algorithm class in this argument. The
        algorithm class 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: AssimuloAlg
    alg_args --
        All arguments for the chosen algorithm should be listed in this dict.
        Valid arguments depend on the algorithm chosen, see algorithm
        implementation in algorithm_drivers.py for details.
        Default: empty dict
    solver_args --
        All arguments for the chosen solver should be listed in this dict.
        Valid arguments depend on the chosen algorithm and possibly which
        solver has been selected for the algorithm. See algorithm
        implementation in algorithm_drivers.py for details.
        Default: empty dict

Returns::

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

The `alg_args` and `solver_args` are arguments for the algorithm and solver chosen, they will be passed on to the
algorithm in the high-level function call. The next section will list the `alg_args` options for all algorithms and
their default values. The `solver_args` argument will be explained in the section after that.

1.2.2. Algorithm argument `alg_args`

The content of the `alg_args` argument is different depending on which algorithm is used. The argument is a dict with default values for all options. The following tables will list the options available in the `alg_args` argument for each algorithm.

Table 5.1. `alg_args` **options for** `AssimuloAlg`

Option	Description	Default value
start_time	Simulation start time.	0.0
final_time	Simulation stop time.	1.0
num_communication_points	Number of points where the solution is re-turned. If set to 0 the integrator will return at it's internal steps.	500
solver	Set which solver to use with class name as string. This determines whether a DAE or ODE problem will be created.	'IDA'
input_trajectory	Trajectory data for model inputs. The argument should be a matrix where the first column represents time and the following columns represents input trajectory data.	An empty matrix, i.e., no input trajectories.
initialize	Do initialization if True, skip initialization if False.	True

Table 5.2. `alg_args` **options for** `AssimuloFMIAlg`

Option	Description	Default value
start_time	Simulation start time.	0.0
final_time	Simulation stop time.	1.0
num_communication_points	Number of points where the solution is re-turned. If set to 0 the integrator will return at it's internal steps.	500
solver	Set which solver to use with class name as string.	'CVode'
input_trajectory	Trajectory data for model inputs. The argument should be a matrix where the first column represents time and the following columns represents input trajectory data. If the `input_trajectory` is set the property `input_names` in `solver_args` must be set and reflect the variables for which the input is going to be adjusted.	An empty matrix, i.e., no input trajectories.

Table 5.3. `alg_args` **options for** `CollocationLagrangePolynomialsAlg`

Option	Description	Default value
n_e	Number of finite elements.	50
n_cp	Number of collocation points.	3
hs	Vector containing the normalized element lengths.	Equidistant points using default n_e.
blocking_factors	Blocking factor vector.	None (not used)
init_traj	A reference to an object of type Result-DymolaTextual or ResultDymolaBinary	None (i.e. not used, set this argument to

Option	Description	Default value
	containing variable trajectories used to initialize the optimization problem.	activate initialization)
result_mesh	Determines which function will be used to get the solution trajectories. Possible values are, 'element_interpolation', 'mesh_interpolation' or 'default'. See optimization.ipopt for more info.	'default'
result_file_name	Name of result file.	Empty string (default generated file name will be used)
result_format	Format of result file.	'txt'
n_interpolation_points	The number of points in each finite element at which the result is returned. Only available for result_mesh = 'element_interpolation'.	None

1.2.3. Algorithm argument `solver_args`

The solver_args argument is a dict of options for the solver selected for the algorithm. The options depend on which solver has been chosen and the best way to find what options are available is to check the documentation for the specific solver. One limitation is that the options in the solver must be implemented as Python properties.

The solvers and corresponding options available in the Assimulo package can be found on the Assimulo web page.

1.3. Result object

Every algorithm returns its own result object and all result objects have a common base class jmodelica.algorithm_drivers.ResultBase. This means that no matter which algorithm is used in the high-level function, the function will always return an object which can be manipulated with the methods and properties of the ResultBase class.

Table 5.4. The `jmodelica.algorithm_drivers.ResultBase` class

Method	Property	Description
get_model()	model	The jmodelica.jmi.Model object that was used in the algorithm.
get_result_file_name()	result_file_name	The name of the result file created on the file system.
get_solver()	solver	The solver used in the algorithm.
get_result_data()	result_data	The result data object containing the whole initialization, simulation or optimization result matrix.

1.4. Algorithms

The algorithms that are used in the high-level functions are implemented as classes in the module jmodelica.algorithm_drivers. They are all subclasses of the base algorithm class jmodelica.algorithm_drivers.AlgorithmBase which contains methods that all algorithm classes must implement. The currently available algorithms are displayed in the table below.

Table 5.5. Algorithms accessible from high-level functions

Algorithm	Use in	Default	Returns
AssimuloAlg	simulate	yes	AssimuloSimResult
AssimuloFMIAlg	simulate	no	AssimuloSimResult

Algorithm	Use in	Default	Returns
`CollocationLagrangePolynomial-` `sAlg`	`optimize`	yes	`CollocationLagrangePolynomial-` `sResult`
`IpoptInitializationAlg`	`initialize`	yes	`IpoptInitResult`
`JFSInitAlg`	`initialize`	no	`JFSInitResult`

1.5. Short simulation example

Here is a short example which will demonstrate how to use the high-level function `simulate`. The RLC circuit model will be used in the example. This model and a Python script which runs the example can be found in the `jmodelica.examples` package.

Start by creating the model object:

```
# The model name and mo-file
model_name = 'RLC_Circuit'
mo_file = 'RLC_Circuit.mo'

# Create jmi.Model object
from jmodelica import jmi
rlc_model = jmi.Model(model_name, mo_file)
```

Then import the `simulate` function and simulate using all default arguments, this means that the `AssimuloAlg` algorithm will be used. Save the result object in a variable.

```
# Import simulate
from jmodelica import simulate

# Simulate with default arguments and save the result object in a variable
sim_res = simulate(model_name, mofile)
```

The result data can then be extracted from the result object and plotted.

```
# Get the result data and plot some signals
res = sim_res.result_data
sine_y = res.get_variable_data('sine.y')
resistor_v = res.get_variable_data('resistor.v')
inductor1_i = res.get_variable_data('inductor1.i')
```

Figure 5.1. Result data from a simulation of s of the RLC Circuit
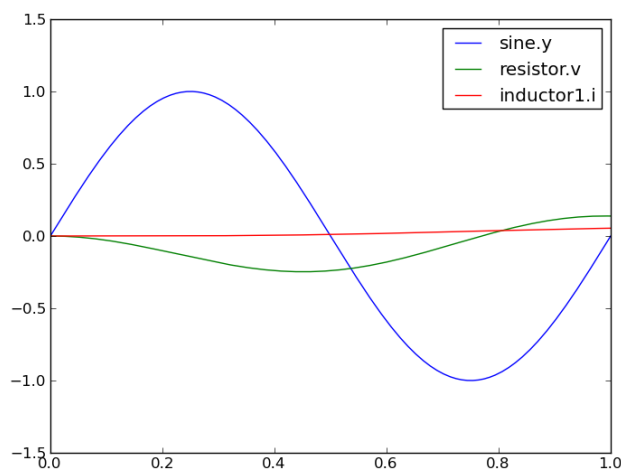
The default simulation time for the `AssimuloAlg` algorithm is 1s. This can be changed by altering the algorithm argument '`final_time`'

```
# Simulate again, this time with 'final_time' set to 30s
sim_res = simulate(model_name, mofile), alg_args={'final_time':30})
```

Plotting with the same commands gives the result which can be seen in the figure below.

Figure 5.2. Result data from a simulation of 30s of the RLC Circuit
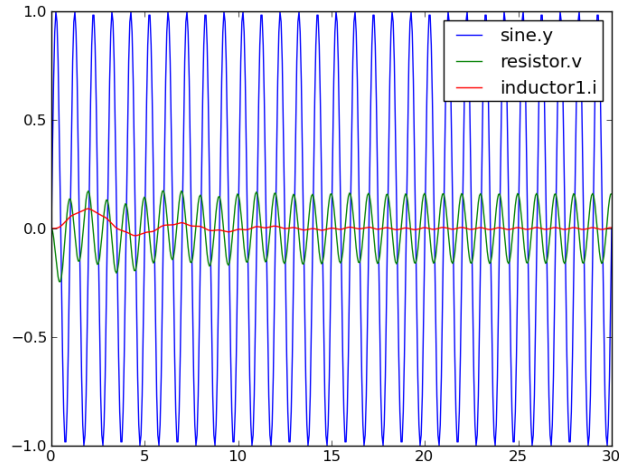
Chapter 6. Advanced topics

1. Tutorial on Abstract Syntax Trees (ASTs)

1.1. About Abstract Syntax Trees

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

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

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

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

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

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

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

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

Three different usages of ASTs are shown:

- Count the number of classes in the Modelica standard library. In this example, a Python function is defined to traverse the source AST which results from parsing of the Modelica standard library.

- Instantiate the CauerLowPassAnalog model. The instance AST for this model is dumped and it is demonstrated how the merged modification environments can be accessed. Also, it is shown how a component redeclaration affects the instance tree.

- Flatten the CauerLowPassAnalog model instance and print some statistics of the flattened Model.

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

Start the tutorial by creating a working directory and copy the file $JMODELICA_HOME/Python/ jmodelica/examples/files/CauerLowPassAnalog.mo to your working directory. An on-line version of

CauerLowPassAnalog.mo is also available (depending on which browser you use, you may have to accept the site certificate by clicking through a few steps). If you choose to create Python script file, save it to the working directory. The tutorial is based on a model from the Modelica Standard Library: Modelica.Electrical.Analog.Basic.Examples.CauerLowPassAnalog.

1.2. Start the Python shell

Next, open a Python shell, preferably using the Pylab mode. If you are running Windows, select the menu option provided with the JModelica.org installation. If you are running Linux or Mac OS X, open a terminal and enter the command:

```
> /path/to/jmodelica_installation/Python/jm_ipython.sh -pylab
```

As your first action, go to the working directory you have created:

```
In [1]: cd '/path/to/working/directory'
```

In order to run the Python script, use the 'run' command:

```
In [2]: run -i ast_example.py
```

Notice the '-i' switch which is used in this tutorial in order to avoid loading the Modelica standard library multiple times and thereby preventing the Python shell from running out of memory.

1.3. Load the Modelica standard library

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

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

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

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

# Import JPype
import jpype

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    # Get the number of local classes.
    num_classes = local_classes.size()

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

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

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

We then call the function:

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

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

```
The package Modelica.UsersGuide has 16 hierachically contained classes
The package Modelica.Constants has 0 hierachically contained classes
The package Modelica.Icons has 16 hierachically contained classes
The package Modelica.SIunits has 532 hierachically contained classes
```

```
The package Modelica.StateGraph has 64 hierachically contained classes
The package Modelica.Blocks has 258 hierachically contained classes
The package Modelica.Electrical has 361 hierachically contained classes
The package Modelica.Math has 74 hierachically contained classes
The package Modelica.Mechanics has 474 hierachically contained classes
The package Modelica.Media has 1064 hierachically contained classes
The package Modelica.Thermal has 88 hierachically contained classes
The package Modelica.Utilities has 86 hierachically contained classes
The package Modelica has 3045 hierachically contained classes
```

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

1.5. Dump the instance AST

We shall now turn our attention to the CauerLowPassAnalog model. Specifically, we would like to analyze the instance hierarchy of the model by dumping the tree structure to the Python shell. In addition, we will look at the merged modification environment of each instance AST node. Again, we will use methods defined for the Java objects representing the AST.

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

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

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

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

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

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

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

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

    # Print
    print(str)

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

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

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

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

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

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

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

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

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

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

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

```
                                            final startTime=startTime}
```

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

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

1.6. Flattening of the filter model

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

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

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

Let us have a look at the flattened model:

```
print(filter_flat_model)
```

We may also retrieve some model statistics:

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

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

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

Chapter 7. 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

$$\dot{x}(t) = v(t) \quad , \quad x(t) = 0$$
$$\dot{v}(t) = u(t) \quad , \quad v(t) = 0$$

and

$$v(t_f) = 0 \quad x(t_f) = 1$$
$$1 < u(t) < -1 \quad v(t) < 0.5$$

In this problem, the final time, tf, is free, and the objective is thus to minimize the time it takes to transfer the state of the double integrator from the point (0,0) to (1,0), while respecting bounds on the velocity *v(t)* and the input *u(t)*. A Modelica model for the double integrator system is given by:

```
model DoubleIntegrator
  Real x(start=0);
  Real v(start=0);
  input Real u;
equation
  der(x)=v;
  der(v)=u;
end DoubleIntegrator;
```

In summary, the Optimica extension consists of the following elements:

• A new specialized class: `optimization`

• New attributes for the built-in type Real: `free` and `initialGuess`

• A new function for accessing the value of a variable at a specified time instant

• Class attributes for the specialized class `optimization`: objective}, `startTime`, `finalTime` and `static`

• A new section: `constraint`

• Inequality constraints

1. A new specialized class: `optimization`

A new specialized class, called `optimization`, in which the proposed Optimica-specific constructs are valid is supported by Optimica. This approach is consistent with the Modelica language, since there are already several other specialized classes, e.g., `record`, `function` and `model`. By introducing a new specialized class, it also becomes straightforward to check the validity of a program, since the Optimica-specific constructs are only valid inside an `optimization` class. The `optimization` class corresponds to an optimization problem, static or dynamic, as specified above. Apart from the Optimica-specific constructs, an `optimization` class can also contain component and variable declarations, local classes, and equations.

It is not possible to declare components from \texttt{optimization} classes in the current version of Optimica. Rather, the underlying assumption is that an `optimization` class defines an optimization problem, that is solved off-line. An interesting extension would, however, be to allow for `optimization` classes to be instantiated. With

this extension, it would be possible to solve optimization problems, on-line, during simulation. A particularly interesting application of this feature is model predictive control, which is a control strategy that involves on-line solution of optimization problems during execution.

As a starting-point for the formulation of the optimization problem consider the `optimization` class:

```
optimization DIMinTime
  DoubleIntegrator di;
  input Real u = di.u;
end DIMinTime;
```

This class contains only one component representing the dynamic system model, but will be extended in the following to incorporate also the other elements of the optimization problem.

2. Attributes for the built in class Real

In order to superimpose information on variable declarations, two new attributes are introduced for the built-in type Real. Firstly, it should be possible to specify that a variable, or parameter, is free in the optimization. Modelica parameters are normally considered to be fixed after the initialization step, but in the case of optimization, some parameters may rather be considered to be free. In optimal control formulations, the control inputs should be marked as free, to indicate that they are indeed optimization variables. For these reasons, a new attribute for the built-in type Real, `free`, of boolean type is introduced. By default, this attribute is set to `false`.

Secondly, an attribute, `initialGuess`, is introduced to enable the user to provide an initial guess for variables and parameters. In the case of free optimization parameters, the `initialGuess` attribute provides an initial guess to the optimization algorithm for the corresponding parameter. In the case of variables, the `initialGuess` attribute is used to provide the numerical solver with an initial guess for the entire optimization interval. This is particularly important if a simultaneous or multiple-shooting algorithm is used, since these algorithms introduce optimization variables corresponding to the values of variables at discrete points over the interval. Notice that such initial guesses may be needed both for control and state variables. For such variables, however, the proposed strategy for providing initial guesses may sometimes be inadequate. In some cases, a better solution is to use simulation data to initialize the optimization problem. This approach is also supported by the Optimica compiler. In the double integrator example, the control variable u is a free optimization variable, and accordingly, the `free` attribute is set to `true`. Also, the `initialGuess` attribute is set to 0.0.

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

3. A Function for accessing instant values of a variable

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

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

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

4. Class attributes

In the optimization formulation above, there are elements that occur only once, i.e., the cost function and the optimization interval. These elements are intrinsic properties of the respective optimization formulations, and should be specified, once, by the user. In this respect the cost function and optimization interval differ from, for example, constraints, since the user may specify zero, one or more of the latter.

In order to encode these elements, class attributes are introduced. A class attribute is an intrinsic element of a specialized class, and may be modified in a class declaration without the need to explicitly extend from a built-in class. In the Optimica extension, four class attributes are introduced for the specialized class `optimization`. These are `objective`, which defines the cost function, `startTime`, which defines the start of the optimization interval, `finalTime`, which defines the end of the optimization interval, and `static`, which indicates whether the class defines a static or dynamic optimization problem. The proposed syntax for class attributes is shown in the following `optimization` class:

```
optimization DIMinTime (
        objective=finalTime,
        startTime=0,
        finalTime(free=true,initialGuess=1))
 DoubleIntegrator di(u(free=true,
                        initialGuess=0.0));
 input Real u = di.u;
end DIMinTime;
```

The default value of the class attribute `static` is `false`, and accordingly, it does not have to be set in this case. In essence, the keyword `extends` and the reference to the built-in class have been eliminated, and the modification construct is instead given directly after the name of the class itself. The class attributes may be accessed and modified in the same way as if they were inherited.

5. Constraints

Constraints are similar to equations, and in fact, a path equality constraint is equivalent to a Modelica equation. But in addition, inequality constraints, as well as point equality and inequality constraints should be supported. It is therefore natural to have a separation between equations and constraints. In Modelica, initial equations, equations, and algorithms are specified in separate sections, within a class body. A reasonable alternative for specifying constraints is therefore to introduce a new kind of section, `constraint`. Constraint sections are only allowed inside an `optimization` class, and may contain equality, inequality as well as point constraints. In the double integrator example, there are several constraints. Apart from the constraints specifying bounds on the control input *u* and the velocity *v*, there are also terminal constraints. The latter are conveniently expressed using the mechanism for accessing the value of a variable at a particular time instant; `di.x(finalTime)=1` and `di.v(finalTime)=0`. In addition, bounds may have to be specified for the `finalTime` class attribute. The resulting optimization formulation may now be written:

```
optimization DIMinTime (
    objective=finalTime,
    startTime=0,
    finalTime(free=true,initialGuess=1))
 DoubleIntegrator di(u(free=true,
                        initialGuess=0.0));
 input Real u = di.u;
constraint
 finalTime>=0.5;
 finalTime<=10;
 di.x(finalTime)=1;
 di.v(finalTime)=0;
 di.v<=0.5;
```

```
 di.u>=-1; di.u<=1;
end DIMinTime;
```

The Optimica specification can be translated into executable format and solved by a numerical solver, yielding the result:

Figure 7.1. Optimization result
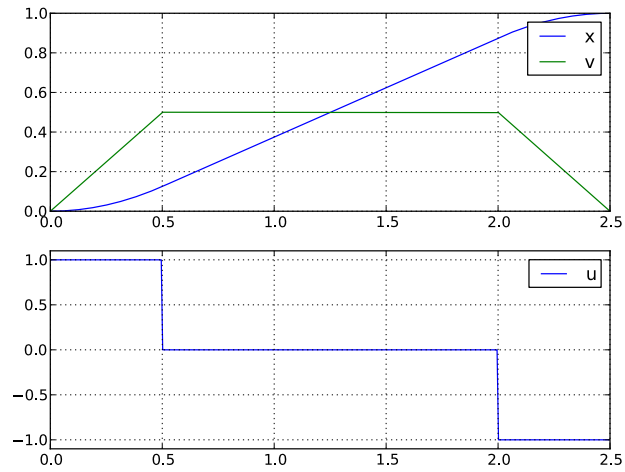
Chapter 8. Limitations

This page lists the current limitations of the JModelica.org platform, as of version 1.3.0. The development of the platform can be followed at the Trac site, where future releases and associated features are planned. In order to get an idea of the current Modelica compliance of the compiler front-end, you may look at the associated test suite. All models with a test annotation can be flattened.

- The Modelica compliance of the front-end is limited; the following features are currently not supported:

  - If expressions are supported, but not:

    - When clauses

    - If equations

  - Parsing of full Modelica 3.2 (Modelica 3.0 is supported)

  - Integer and boolean variables (integer and boolean parameters and constants are supported)

  - Strings

  - Enumerations

  - Generics (redeclare constructs) is only partially supported

  - Partial support for external functions, only external C functions with scalar inputs and outputs are supported.

  - The following built-in functions are not supported:

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

  - Overloaded operators (Modelica Language Specification, chapter 14)

  - Stream connections with more than two connectors are not supported.

  - Mapping of models to execution environments (Modelica Language Specification, chapter 16)

- In the Optimica front-end the following constructs are not supported:

- Annotations for transcription information

- The JModelica.org Model Interface (JMI) has the following Limitations:

  - The ODE interface requires the Modelica model to be written on explicit ODE form in order to work.

  - Second order derivatives (Hessians) are not provided

  - The interface does not yet comply with FMI specification

- The JModelica.org FMI Model Interface (FMI) has the following Limitations:

  - The FMI interface only supports FMUs distributed with binaries, not only source code.

  - Options for setting and getting string variables does not work

Chapter 9. Release notes

1. Release notes for JModelica.org version 1.2

1.1. 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

1.2. Compilers

1.2.1. The Modelica compiler

1.2.1.1. 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.

1.2.1.2. 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)

1.2.1.3. 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.

1.2.1.4. 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.

1.2.2. The Optimica Compiler

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

1.3. The JModelica.org Model Interface (JMI)

1.3.1. General

1.3.1.1. 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.

1.3.1.2. 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.

1.3.1.3. 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.

1.3.1.4. 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.

1.4. The collocation optimization algorithm

1.4.1. 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.

1.4.2. 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.

1.4.3. 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.

1.5. 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.

1.6. 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.

1.7. 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.

1.8. 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.

1.8.1. 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.

1.9. Contributors

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Jesper Mattsson

Philip Nilsson

Roberto Parrotto

Philip Reuterswärd

Johan Åkesson

1.9.1. Previous contributors

Jens Rantil

2. Release notes for JModelica.org version 1.3

2.1. 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)

2.2. Compilers

2.2.1. The Modelica compiler

2.2.1.1. 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.

2.2.1.2. Redecare

Redeclares as class elements are now supported.

2.2.1.3. Conditional components

Conditional components are now supported.

2.2.1.4. 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.

2.2.1.5. 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.

2.2.1.6. Stream connectors

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

2.2.1.7. 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.

2.2.2. The Optimica compiler

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

2.2.2.1. New class attribute objectiveIntegrand

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

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

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.

2.2.2.2. 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.

2.3. JModelica.org Model Interface (JMI)

2.3.1. The collocation optimization algorithm

2.3.1.1. 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.

2.3.1.2. 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.

2.4. 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.

2.5. 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.

2.6. XML model export

2.6.1. `noEvent` operator

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

2.6.2. `static` attribute

Support for the Optimica attribute static has been implemented.

2.7. Python integration

2.7.1. High-level functions

2.7.1.1. Model files

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

2.7.1.2. 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.

2.7.2. File I/O

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

2.8. Contributors

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Jesper Mattsson

Roberto Parrotto

Johan Åkesson

Philip Reuterswärd

2.8.1. Previous contributors

Philip Nilsson

Jens Rantil

Bibliography

[Jak2007] Johan Åkesson. *Tools and Languages for Optimization of Large-Scale Systems*. LUTFD2/TFRT--1081--SE. Lund University. Sweden. 2007.

[Jak2008b] Johan Åkesson, Görel Hedin, and Torbjörn Ekman. *Tools and Languages for Optimization of Large-Scale Systems*. 117-131. *Electronic Notes in Theoretical Computer Science*. 203:2. April 2008.

[Jak2008a] Johan Åkesson. *Optimica—An Extension of Modelica Supporting Dynamic Optimization. Proc. 6th International Modelica Conference 2008*. Modelica Association. March 2008.

[Jak2010] Johan Åkesson, Karl-Erik Årén, Magnus Gäfvert , , and . *Modeling and Optimization with Optimica and JModelica.org—Languages and Tools for Solving Large-Scale Dynamic Optimization Problem. Computers and Chemical Engineering*. 203:2. 2010.

Index

C
CppAD, 5

I
IPOPT, 5

J
JastAdd, 4
JMI, 5
   (see also JModelica Model Interface)
JModelica Model Interface (see JMI)

M
Modelica, 4

O
Optimica, 4

X
XML, 5
XPATH, 5