JModelica.org User Guide

Version 1.4.0

JModelica.org User Guide: Version 1.4.0

Publication date 2010-10-15

Acknowledgements

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

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).

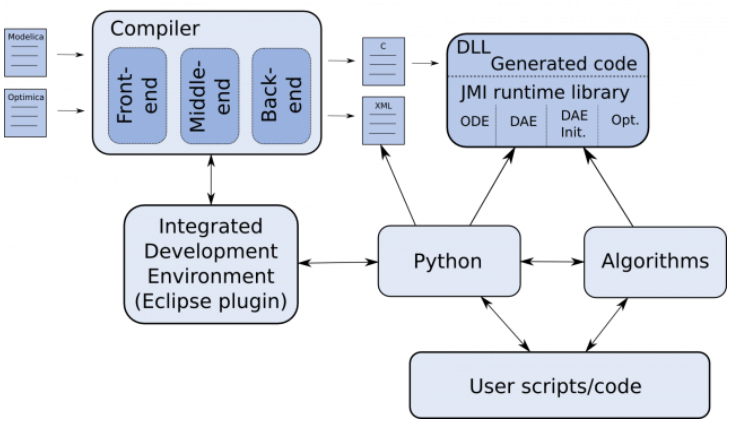
A unique feature of JModelica.org is the support for the innovative extension Optimica. Optimica enables you to conveniently formulate optimization problems based on Modelica models using simple but powerful constructs for encoding of optimization interval, cost function and constraints. Optimica also 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. Prerequisites

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

2.1. Java

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

Install a JRE

1. Get a JRE installer suitable for your platform here.
2. Run the installer.

2.2. Python

Python 2.6 with the following additional packages are required:

Table 2.1. Python prerequisites

Package	Recommended version	Description
NumPy	1.3.0	The fundamental package needed for scientific computing with Python.
SciPy	0.7.1	A library of algorithms and mathematical tools for Python.
matplotlib	0.99.1.1	A plotting library for Python, with a MATLAB like interface.
IPython	0.10	An interactive shell for Python with additional shell syntax, code highlighting, tab completion, string completion, and rich history.
PyReadline	1.5	Note: Windows only. A readline for Windows required by IPython.

3. Binary distribution

3.1. Windows

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

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

3.1.1. Installing JModelica.org with Windows installer

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

1. Download a JModelica.org Windows binary installer.

2. Run the installer and follow the wizard. Choose to install the additional Python packages when prompted, unless the are already installed.

3.2. Linux

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

3.3. Mac OS X

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

4. Software Development Kit

The JModelica.org Software Development Kit (SDK) is a bundle of tools needed to build the JModelica.org sources on Windows. The JModelica.org SDK windows installer sets up a complete pre-configured environment with convenient start menu shortcuts.

4.1. Prerequisites

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

Install a JDK

1. Get a JDK here.

2. Run the installer.

4.2. Installing the JModelica.org SDK

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

4.2.1. Bundled tools

The following tools are bundled in the JModelica.org SDK.

- MinGW, MSYS
- Ipopt
- JPype
- lxml
- nose
- CollabNet Subversion Client
- Apache Ant
- SUNDIALS
- Cython
- DocBook

4.2.2. Step-by-step instructions

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

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

3. Build the sources:

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

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

   ```
   cd /JModelica.org-SDK
   ./configure.sh
   ```

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

   ```
   cd build
   make
   make install
   ```

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

```
# import the vdp example
from jmodelica.examples import vdp

# run the vdp
vdp.run_demo()
```

Chapter 3. Getting started

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

1. Starting a Python session

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

1.1. Windows

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

- Python - Normal command shell started with Python.
- IPython - Interactive shell for Python with, for example, code highlighting and tab completion.
- pylab - IPython shell which also loads the package pylab.

1.2. Linux or Mac OS

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

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

2. Run an example

There are several example scripts which compile, load and simulate/optimize models in the `examples` folder under the Python package in the JModelica.org installation. The models themselves are located in the subdirectory `files`. The following code demonstrates how to run such an example. First a Python session must be started, see Section 1. The JModelica.org Python interface is preferably run in a IPython shell with Pylab mode.

```
# import the VDP example
from jmodelica.examples import vdp

# run the example
vdp.run_demo()
```

This code will run the example and plot some results.

3. Check your installation

The JModelica.org Python interface requires a few Python packages in order to run correctly. Use the function `jmodelica.check_packages()` to list the dependencies and version located.

```
# import the function from jmodelica
from jmodelica import check_packages

# run check_packages
check_packages()

Performing JModelica package check
==================================

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

Python version:.............. 2.6.5

JModelica version:........... r2074
```

```
Dependencies:

Package                       Version
-------                       -------
numpy........................ 1.3.0                         Ok
scipy........................ 0.7.1                         Ok
matplotlib................... 0.99.1                        Ok
jpype........................ --                            Ok
lxml......................... 2.2.8                         Ok
nose......................... 0.11.3                        Ok
assimulo..................... --                            Ok
pyreadline................... 1.5                           Ok
```

Missing or having the wrong version of a package can be a source of errors. Therefore it can be useful to run `jmodelica.check_packages()` after installation or when trouble-shooting.

4. The JModelica.org user forum

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

Chapter 4. Working with Models

1. Introduction to models

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

2. Compilation

In its simplest form, compilation only requires a few lines of code; importing a compiler function from JModelica.org and specifying a Modelica or Optimica model and file location. This will be demonstrated in Section 2.1. For more advanced usage there are compiler options and parameters which can be modified, this will be explained in Section 2.2. The last section in this part, Section 2.3, will go through the compilation process and how to compile a model step-by-step.

2.1. Simple compilation example

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

1. Import the JModelica.org compiler function `compile_jmu` from `jmodelica.jmi`.
2. Specify the model and model file.
3. Perform the compilation.

This is demonstrated in the following code example.

```
# import the compiler function
from jmodelica.jmi import compile_jmu

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

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

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

In the above example, compilation has been performed with default parameters and options. The only parameters specified are model name and file. `compile_jmu` has several other parameters which can be modified. The different parameters, their default values and interpretation will be explained in Section 2.2.

2.2. Compiler settings

2.2.1. compile_jmu parameters

The `compile_jmu` parameters can be listed with the interactive help. The compiler target, which is set with the parameter `target`, is further explained in Section 2.2.3.

```
# display the docstring for compile_jmu with the Python command 'help'
from jmodelica.jmi import compile_jmu
help(compile_jmu)
```

```
Help on function compile_jmu in module jmodelica.jmi:

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

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

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

    * class_name and file_name is passed:
        - file_name can be a single file as a string or a list of file_names
          (strings).
        - Default compiler is ModelicaCompiler but will switch to
          OptimicaCompiler if a .mop file is found in file_name.

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

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

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

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

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

    Parameters::

        class_name --
            The name of the model class.

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

        compiler --
            'modelica' or 'optimica' depending on whether a ModelicaCompiler or
            OptimicaCompiler should be used. Set this argument if default
            behaviour should be overridden.
            Default: Depends on argument file_name.

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

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

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

    Returns::
```

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

2.2.2. Compiler options

Compiler options are read from an XML file, `options.xml`, which can be found in the JModelica.org installation folder under the folder `Options`. The options are loaded from the file when a compiler is created, that is when `compile_jmu` is run. Options for a compiler instance can be modified by editing the file before compiling. There are four type categories: string, real, integer and boolean. The available options, default values and description are listed in Table 4.1.

Table 4.1. Compiler options

Option	Default	Description
`normalize_minimum_time_problems`	`true`	When this option is set to `true` the minimum time optimal control problems encoded in Optimica are converted to fixed interval problems by scaling of the derivative variables. (Boolean option.)
`enable_variable_scaling`	`false`	If this option is `true`, then the "nominal" attribute will be used to scale variables in the model. (Boolean option.)
`halt_on_warning`	`false`	If this option is set to `false` one or more compiler warnings will not stop compilation of the model. (Boolean option.)
`automatic_add_initial_equations`	`true`	When this option is set to `true`, then additional initial equations are added to the model based on a the result of a matching algorithm. Initial equations are added for states that are not matched to an equation. (Boolean option.)
`generate_fmi_xml`	`false`	If this option is `true` the model description part of the XML variables file will be FMI compliant. To generate an XML which will validate with FMI schema the option `generate_xml_equations` must also be set to `false`. (Boolean option.)
`eliminate_alias_variables`	`true`	If this option is set to `true`, then alias variables are eliminated from the model. (Boolean option.)
`extra_lib_dirs`	`""` (Empty string)	The value of this option is appended to the value of the `MODELICAPATH` environment variable for determining in what directories to search for libraries. (String option.)
`generate_xml_equations`	`false`	If this option is `true`, then model equations are generated in XML format. (Boolean option.)
`state_start_values_fixed`	`false`	This option enables the user to specify if initial equations should be generated automatically for differentiated variables even though the fixed attribute is equal to `fixed`. Setting this option to `true` is, however, often practical in optimization problems. (Boolean option.)
`equation_sorting`	`false`	If this option is `true`, equations are sorted using the BLT algorithm. (Boolean option.)
`index_reduction`	`false`	If this option is `true`, index reduction is performed. (Boolean option.)

Option	Default	Description
enable_symbolic_diagnosis	true	Enable this option to invoke the structural error diagnosis based on the matching algorithm. (Boolean option.)

2.2.3. Compiler targets

There are four compiler targets available:

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

The compile_model parameter target is 'ipopt' by default which will work for most cases. However, if JModelica.org has been built without Ipopt libraries the target parameter would have to be changed to any other suitable target that does not include the Ipopt libraries. The target model_noad must be used if the model contains external equations since external equations do not work with the AD interface at the moment.

2.3. Compilation in more detail

Compiling with compile_jmu bundles quite a few steps required for the compilation. These steps will be described briefly here, for a more detailed review on the compilation steps see Section 4 in Chapter 1.

2.3.1. Create a compiler

A compiler, can be either a Modelica or Optimica compiler, is created by importing the Python classes from the compiler module. The compiler constructors do not take any arguments. This example code will create a Modelica compiler.

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

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

2.3.2. Source tree generation and flattening

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

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

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

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

2.3.3. Code generation

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

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

2.3.4. Generate a shared object file

Finally, the shared object file is built where the C code is linked with the JModelica.org Model Interface (JMI) runtime library. The `compile_binary` method takes one obligatory parameter, the name of the C file that was generated in the previous step. The parameter `target` must also be set here if something other than the default `'model'` is wanted. In this example, `target` is changed to `'ipopt'`. For more information on targets, see Section 2.2.3.

```
# Compile a shared object file
c_file = 'myPackage_myModel.c'
mc.compile_binary(c_file, target='ipopt')
```

3. Loading models

Compiled models, JMUs, are loaded in the JModelica.org Python interface with the `JMUModel` class from the `jmodelica.jmi` module. This will be demonstrated in Section 3.2. The `JMUModel` class contains many methods with which the model can be manipulated once it has been instantiated. Amongst the most important methods are the `initialize`, `simulate` and `optimize` methods. These are explained in Chapter 7 and Chapter 8. The more basic `JMUModel` methods for variable and parameter manipulation are explained in Section 4.

3.1. The JMU file

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

3.2. Loading a JMU

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

```
# import JMUModel from jmodelica.jmi
from jmodelica.jmi import JMUModel
myModel = JMUModel('myPackage_myModel.jmu')
```

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

```
# import compile_jmu and JMUModel
from jmodelica.jmi import compile_jmu
from jmodelica.jmi import JMUModel

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

3.3. Loading an FMU

An FMU (Functional Mock-up Unit) is a compressed file which follows the FMI (Functional Mock-up Interface) standard. The FMU file can be loaded in JModelica.org with the class `FMUModel` in the `jmodelica.fmi` module. The following short example demonstrates how to do this in a Python shell or script.

```
# import FMUModel from jmodelica.fmi
from jmodelica.fmi import FMUModel
myModel = FMUModel('myFMU.fmu')
```

The FMUModel instance can then be used to set parameters and used for simulations. Read more about the FMI and FMU in Chapter 5. How to simulate an FMU is described in Chapter 7.

4. Variable and parameter manipulation

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

4.1. Model variable XML files

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

- `<model class name>.xml`
- `<model class name>_values.xml`

The variable 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 XML values file to a parameter specification in the XML variables file.

4.2. Setting and getting variables

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

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

```
# compile and load the model
from jmodelica.jmi import compile_jmu
from jmodelica.jmi import JMUModel
jmu_name = compile_jmu('RLC_Circuit','RLC_Circuit.mo')
rlc_circuit = JMUModel(jmu_name)

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

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

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

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

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

4.3. Loading and saving parameters

4.3.1. Loading XML values file

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

```
# look at parameters 'resistor.R' and 'sine.offset'
rlc_circuit.get('resistor.R)
>> 1.0
rlc_circuit.get('sine.offset')
>> 0.0

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

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

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

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

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

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

4.3.2. Writing to XML values file

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

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

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

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

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

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

Chapter 5. FMI Interface

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

A model is distributed in a zip-file with the extension '.fmu', containing several files. These zip-files containing the models are called FMUs (Functional Mock-up Units). The important files in an FMU are mainly the XML-file, which contains the definitions of all variables and then files containing the C-functions which can be provided in source and/or binary form. FMI standard also supports providing documentation and resources together with the FMU. For more information regarding the FMI standard, please visit http://www.functional-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 `FMUModel` together with methods for unzipping the FMU and for writing the simulation results. Connected to this interface is a wrapper for JModelica.org's simulation package to enable an easy simulation of the FMUs. The simulation wrapper is located in `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 5.1. Conversion table.

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

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

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

```
FMUModel.get_log()
```

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

```
FMUModel.get_real?
```

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

2. Example

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

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)]
```

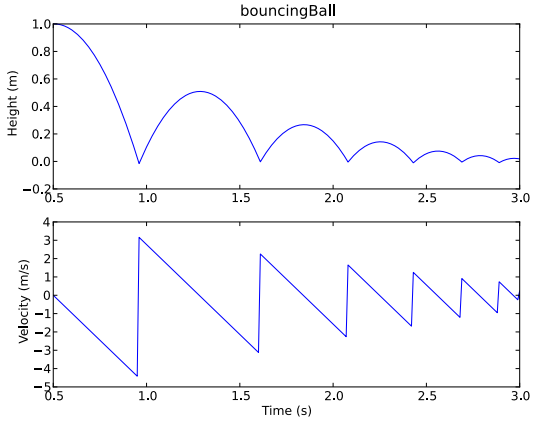
In the end of the loop, the solution is stored and the old event indicators are stored for use in the next loop.

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 5.1. Simulation result



Chapter 6. Initialization

1. Solving DAE initialization problems

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

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

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

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

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

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

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

    The default algorithm for this function is IpoptInitializationAlg.

    Parameters::

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

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

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

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

    Returns::

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

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

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

```
    default initialization algorithm will be returned.

    Parameters::

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

    Returns::

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

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

```
x = init_model['x']
y = init_model['y']
```

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

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

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

- If the number of equations is greater than the number of variables the system is overdetermined. Such a system may not have a solution, and will be treated as ill-defined. An exception is thrown in this case.
- If the number of equations is less than the number of variables the system is underdetermined and such a system has infinitely many solutions. In this case, the compiler tries to balance the system by setting some fixed attributes to `true`. So if the user supplies too few initial conditions, some variables with the attribute `fixed` set to `false` may be changed to `true` during initialization.

3. Initialization algorithms

JModelica.org has the ability to support different initialization algorithms. The choice of algorithm is done when calling the `initialize`:

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

In the example above the default algorithm using IPOPT is chosen. This is currently the only fully implemented initialization algorithm.

Chapter 7. Simulation

1. Introduction

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

2. A first example

This example focus on how to use JModelica.org's simulation functionality in the most basic way. The model which is to be simulated is the Van der Pol problem described in ???. The model is also available from the examples in the file VDP.mop.

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

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

    // The control signal
    input Real u;

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

Create a new file in your working directory called `VDP.mo` and save the model. Note that this model is described as an ODE and can thus be simulated using ODE solvers from Assimulo.

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

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

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

Next, we compile and load the model:

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

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

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

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

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

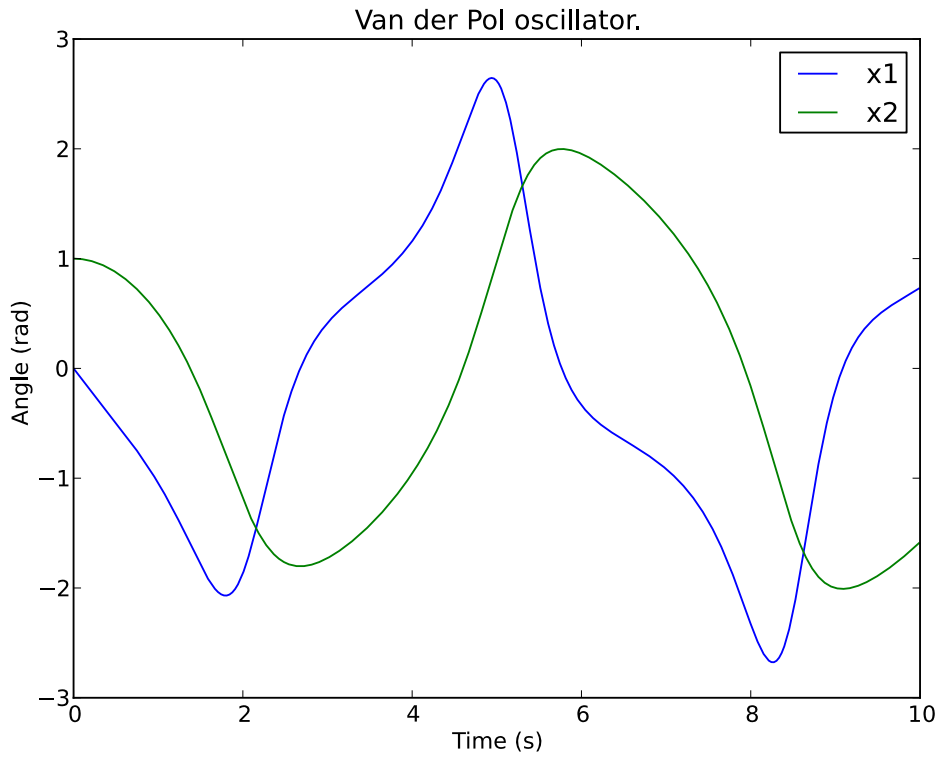
```
x1 = res['x1']
x2 = res['x2']
t  = res['time']
```

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

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

In Figure 7.1 the simulation result is shown.

Figure 7.1. Simulation result of the Van der Pol oscillator.



3. Simulation of Models

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

- `JMUModel (located in jmodelica.jmi)`
- `FMUModel (located in jmodelica.fmi)`

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

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

```
class JMUModel(...)
    ...
    def simulate(self,
             start_time=0.0,
             final_time=1.0,
             input=(),
```

```
            algorithm='AssimuloAlg',
            options={}):
```

And used in the following way:

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

The only difference between the simulation method in `FMUModel` is that the algorithm is `AssimuloFMIAlg`. They are although both connected to the Assimulo package and able to use its solvers with the limitation that the FMI standard only support ODEs.

3.1. Arguments

The start and final time attributes are simply the time where the solver should start the integration and stop the integration. The input however is a bit more complex. The input defines the input trajectories to the model and should be a 2-tuple consisting of the name(s) of the input variables in the model and the second argument should be a data matrix with the time vector as the first column and second column should correspond to the first name in the in the first argument and so forth.

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

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

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

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

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

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

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

```
t = N.linspace(0.,10.,100) #Create one hundred evenly spaced points
u1 = N.sin(t) #Create the first input vector
u2 = N.cos(t) #Create the second input vector
u_traj = N.transpose(N.hstack((t,u1,u2))) #Create the data matrix and
                                          #transpose it to the correct form

input_model = (['u1','u2'], u_traj)

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

Note that the variables are now a List of variables.

The algorithm attribute is where the different simulation package can be specified, however currently only a connection to Assimulo is supported and connected through the algorithm `AssimuloAlg` for `JMUModel` and `AssimuloFMIAlg` for `FMUModel`.

3.1.1. Options for JMUModel

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

```
opts = JMUModel.simulate_options()
```

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

```
>>> help(opts)
   Options for simulation of a JMU model using the Assimulo simulation package.
   The Assimulo package contain both explicit solvers (CVode) for ODEs and
   implicit solvers (IDA) for DAEs. The ODE solvers require that the problem
```

```
  is written on the form, ydot = f(t,y).

  Assimulo options::

      solver     --
          Specifies the simulation algorithm that is to be used.
          Default 'IDA'

      ncp        --
          Number of communication points. If ncp is zero, the solver
          will return the internal steps taken.
          Default '0'

      initialize --
          If set to True, an algorithm for initializing the
          differential equation is invoked, otherwise the
          differential equation is assumed to have consistent
          initial conditions.
          Default is True.

      write_scaled_result --
          Set this parameter to True to write the result to file without
          taking scaling into account. If the value of scaled is False,
          then the variable scaling factors of the model are used to
          reproduced the unscaled variable values.
          Default: False

  The different solvers provided by the Assimulo simulation package provides
  different options. These options are given in dictionaries with names
  consisting of the solver name concatenated by the string '_option'. The most
  common solver options are documented below, for a complete list of options
  see, http://www.jmodelica.org/assimulo

  Options for IDA::

      rtol    --
          The relative tolerance.
          Default 1.0e-6

      atol    --
          The absolute tolerance.
          Default 1.0e-6

      maxord  --
          The maximum order of the solver. Can range between 1 to 5.
          Default 5

  Options for CVode::

      rtol    --
          The relative tolerance.
          Default 1.0e-6

      atol    --
          The absolute tolerance.
          Default 1.0e-6

      discr   --
          The discretization method. Can be either 'BDF' or 'Adams'
          Default 'BDF'

      iter    --
          The iteration method. Can be either 'Newton' or 'FixedPoint'
          Default 'Newton'
```

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

```
...
```

```
opts = model.simulate_options() #Retrieve the default options

opts['solver'] = 'CVode' #Change the solver from IDA to CVode

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

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

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

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

#or

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

More information regarding the solver options can be found here, http://www.jmodelica.org/assimulo and a selection of solver arguments for the ODE solver CVode can be found in Table 7.1 and for the DAE solver IDA in Table 7.2.

Table 7.1. Selection of solver arguments for CVode

Argument	Option
discr (The discretization method)	'BDF' or 'Adams'
iter (The iteration method)	'Newton' or 'FixedPoint'
maxord (The maximum order used)	Integer of max 5 (BDF) or 12 (Adams)
maxh (Maximum step-size)	Positive float
atol (Absolute Tolerance)	Array of floats or Float
rtol (Relative Tolerance)	Float

Table 7.2. Selection of solver arguments for IDA

Argument	Option
maxord (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
suppress_alg (Suppress the algebraic variables on the error test)	Boolean

3.1.2. Options for FMUModel

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

```
opts = FMUModel.simulate_options()
```

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

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

   Assimulo options::

       solver --
```

```
            Specifies the simulation algorithm that is to be used. Currently the
            only supported solver is 'CVode'.
            Default: 'CVode'

        ncp    --
            Number of communication points. If ncp is zero, the solver will
            return the internal steps taken.
            Default: '0'

        initialize --
            If set to True, the initializing algorithm defined in the FMU model
            is invoked, otherwise it is assumed the user have manually invoked
            model.initialize()
            Default is True.

        write_scaled_result --
            Set this parameter to True to write the result to file without
            taking scaling into account. If the value of scaled is False,
            then the variable scaling factors of the model are used to
            reproduced the unscaled variable values.
            Default: False

    The different solvers provided by the Assimulo simulation package provides
    different options. These options are given in dictionaries with names
    consisting of the solver name concatenated by the string '_option'. The most
    common solver options are documented below, for a complete list of options
    see, http://www.jmodelica.org/assimulo

    Options for CVode::

        rtol    --
            The relative tolerance. The relative tolerance are retrieved from
            the 'default experiment' section in the XML-file and if not
            found are set to 1.0e-4
            Default: 1.0e-4

        atol    --
            The absolute tolerance.
            Default: rtol*0.01*(nominal values of the continuous states)

        discr   --
            The discretization method. Can be either 'BDF' or 'Adams'
            Default: 'BDF'

        iter    --
            The iteration method. Can be either 'Newton' or 'FixedPoint'
            Default: 'Newton'
```

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

```
...
opts = model.simulate_options() #Retrieve the default options

#opts['solver'] = 'CVode' #Not necessary, default solver is CVode

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

opts['initialize'] = False #Dont initialize the model

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

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

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

More information regarding the solver options can be found here, http://www.jmodelica.org/assimulo and a selection of solver arguments for the ODE solver CVode can be found in Table 7.1.

3.2. Return argument

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

```
res = model.simulate()

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

This can be done for all the variables, parameters and constants defined in the model and is the preferred way of retrieving the result. There are however some more options available in the result object, see Table 7.3.

Table 7.3. Result Object

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

4. Examples

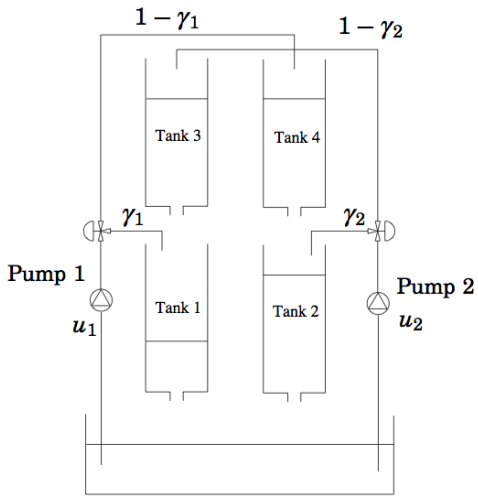
In the next sections, it will be shown how to use the JModelica.org platform for simulation of various JMUs and FMUs.

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

4.1. Simulation with inputs

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

Figure 7.2.



A schematic picture of the quadruple tank process.

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

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

   // Tank levels
  Modelica.SIunits.Length x1(start=x1_0,min=0.0001/*,max=0.20*/);
  Modelica.SIunits.Length x2(start=x2_0,min=0.0001/*,max=0.20*/);
  Modelica.SIunits.Length x3(start=x3_0,min=0.0001/*,max=0.20*/);
  Modelica.SIunits.Length x4(start=x4_0,min=0.0001/*,max=0.20*/);

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

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

end QuadTank;
```

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

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

from jmodelica.jmi import compile_jmu
from jmodelica.jmi import JMUModel
```

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

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

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

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

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

Next, we compile and load the model:

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

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

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

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

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

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

And then plotted with the help from `matplotlib`:

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

plt.figure(2)
plt.subplot(2,1,1)
plt.plot(t_sim,u1_sim,'r')
plt.title('u1')
plt.subplot(2,1,2)
```

```
plt.plot(t_sim,u2_sim,'r')
plt.title('u2')
plt.xlabel('t[s]')
plt.show()
```

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

Figure 7.3. Tank levels

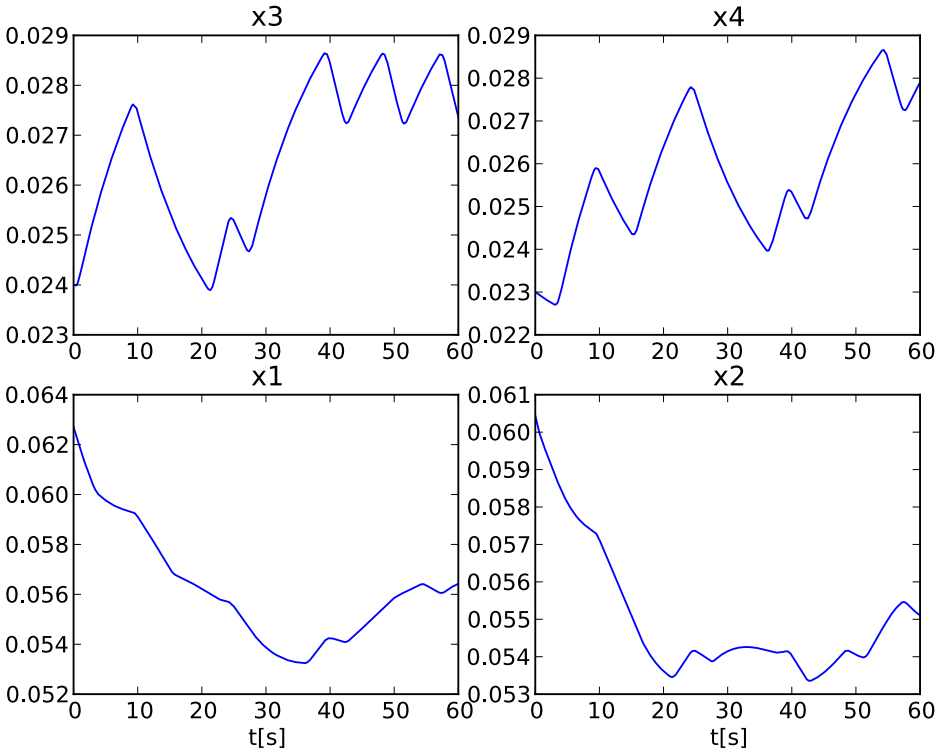
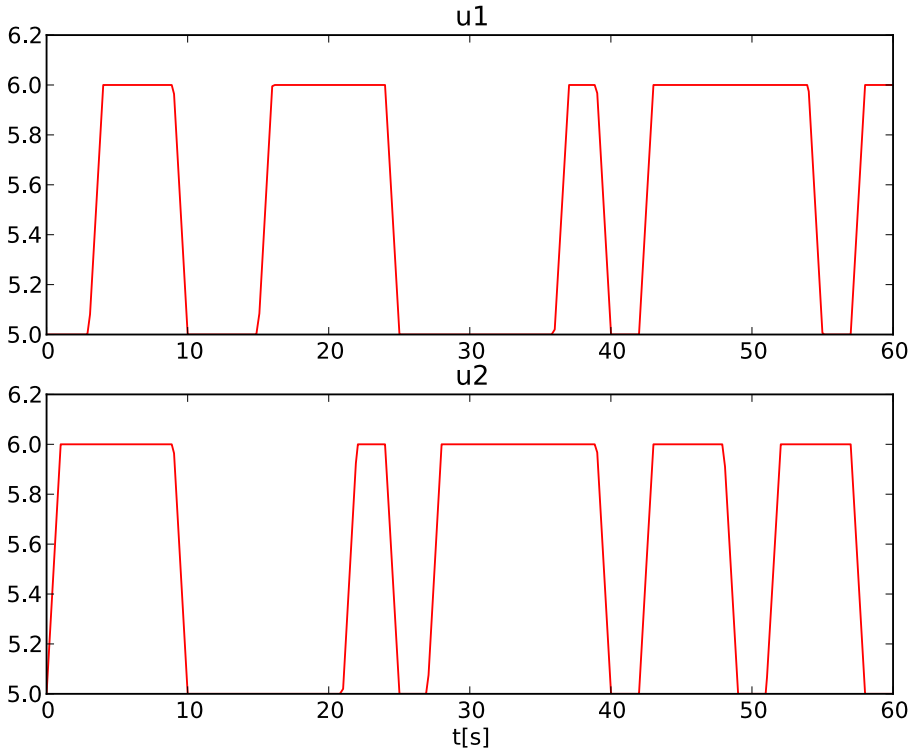


Figure 7.4. Input trajectories



4.2. Simulation of an ODE

Simulation of ODEs in JModelica.org is currently limited to models written explicitly on the form:

$\dot{y} = f(t,y)$

An example is the Van der Pol oscillator described in Section 2. In ??? it is shown again for convenience.

```
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;
```

We see here that the state derivatives are written on the left hand side while the equations are written on the right hand side. There is also a limitation that events are currently not handled.

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

```
# Import the function for compilation of models and the JMUModel class
```

```
from jmodelica.jmi import compile_jmu
from jmodelica.jmi import JMUModel

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

Next, we compile and load the model:

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

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

Now we would like to change the default solver IDA to the ODE solver CVode and also change the relative tolerance to 0.0001. First we retrieve the default options and then change the option:

```
opts = vdp.simulate_options() #Retrieve the default options

opts['solver'] = 'CVode' #Change the solver to CVode
opts['CVode_options']['rtol'] = 0.0001 #Change the relative tolerance for CVode
```

Now lets simulate the problem ten seconds:

```
res = vdp.simulate(final_time=10, options=opts)
```

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

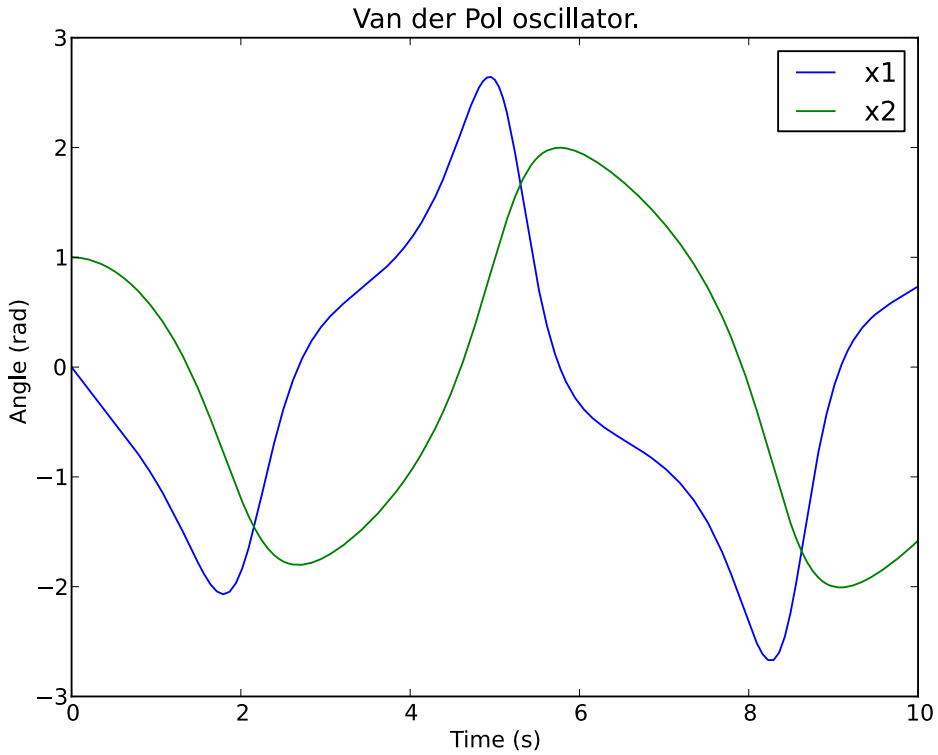
```
x1 = res['x1']
x2 = res['x2']
t  = res['time']
```

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

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

In Figure 7.5 the simulation result is shown.

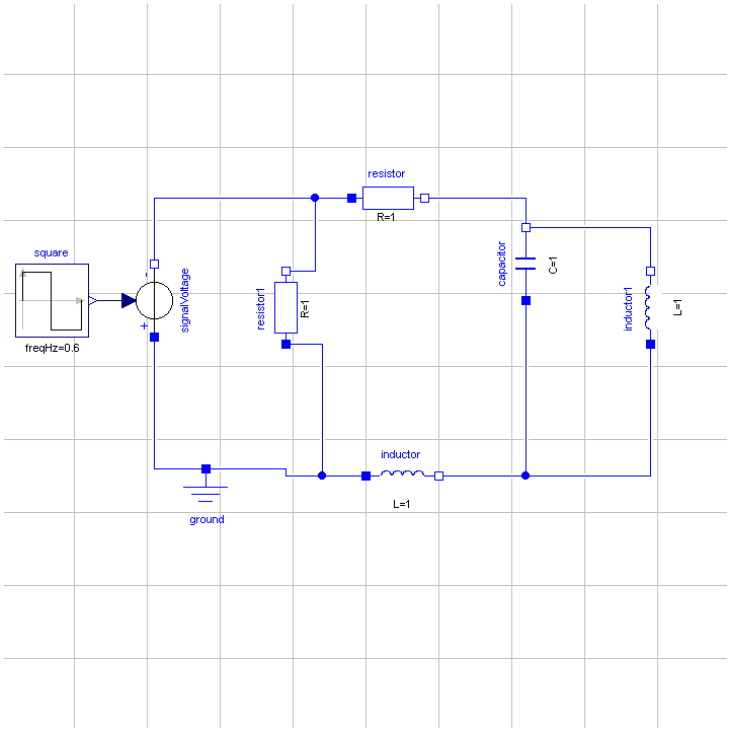
Figure 7.5. Simulation result of the Van der Pol oscillator.



4.3. Simulation of a discontinuous system

The model which is to be simulated is an electric circuit. The model is depicted in Figure 7.6 and consists of resistances, inductors and a capacitor. The circuit is connected to a voltage source which generates a square-wave with an amplitude of 1.0 and a frequency of 0.6 Hz. The model is also available from the examples in the file RLC_Circuit.mo.

Figure 7.6. Electric Circuit



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

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

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

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

Next, we compile and load the model:

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

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

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

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

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

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

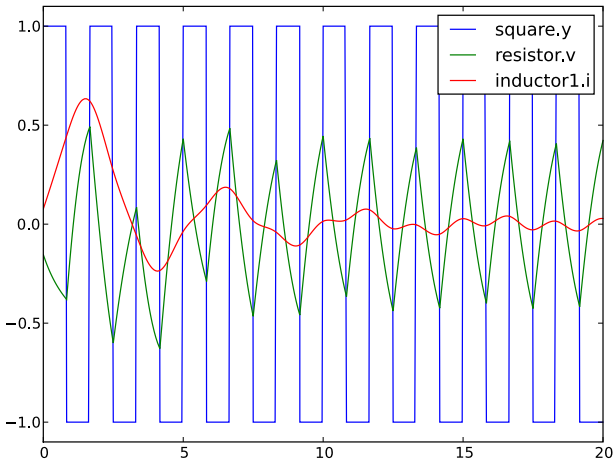
And then plotted with the help from `matplotlib`,

```
plt.figure(1)
plt.plot(time, square_y, time, resistor_v, time, inductor1_i)
```

```
plt.legend(('square.y','resistor.v','inductor1.i'))
plt.show()
```

The simulation result is shown in Figure 7.7.

Figure 7.7. Simulation result



4.4. 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.jmi import compile_jmu, JMUModel
```

into the Python file. Compile and load the model:

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

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

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

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

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

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

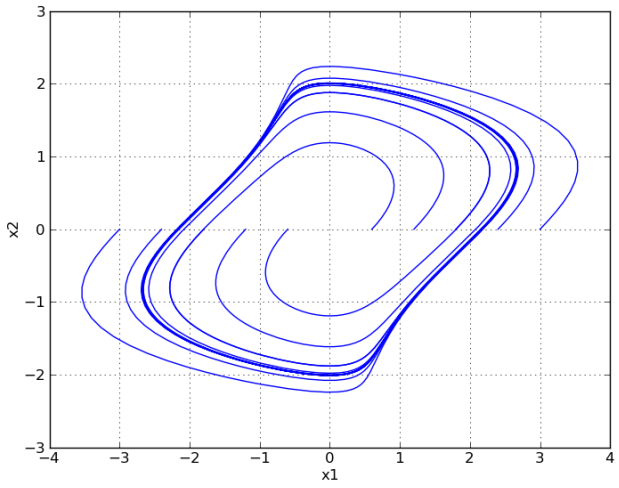
```
fig = p.figure()
p.clf()
p.hold(True)
p.xlabel('x1')
p.ylabel('x2')
```

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

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

You should now see a plot like in Figure 7.8.

Figure 7.8. Simulation result-phase plane

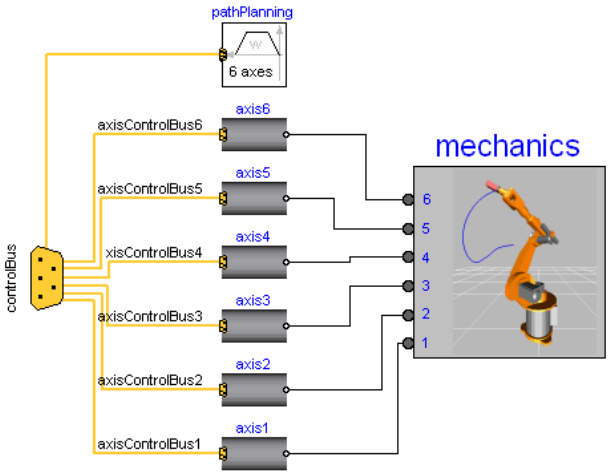


4.5. Simulation of an FMU

This example will show how to use the JModelica.org's FMI-interface together with its simulation package, Assimulo. The FMU to be simulated is the full Robot from the Modelica standard library (3.1) where it is located in `Mechanics.MultiBody.Examples.Systems.RobotR3`. It consists of brakes, motors, gears and path planning.

The model consists of 36 continuous states and around 700 algebraic variables together with 98 event functions and also a few thousand constants/parameters. The FMU was generated using Dymola 7.4.

Figure 7.9. Full Robot



This examples assumes that an FMU of the robot named `Modelica_Mechanics_MultiBody_Examples_Systems_RobotR3_fullRobot.fmu` exists in the working folder.

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

```
# Import the FMUModel class
from jmodelica.fmi import FMUModel

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

Next, we load the FMU into the model object:

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

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

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

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

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

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

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

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

```
plt.show()
```

In Figure 7.10 the result is shown and in Figure 7.11 a comparison between Dymola and JModelica.org is plotted.

Figure 7.10. Robot Results

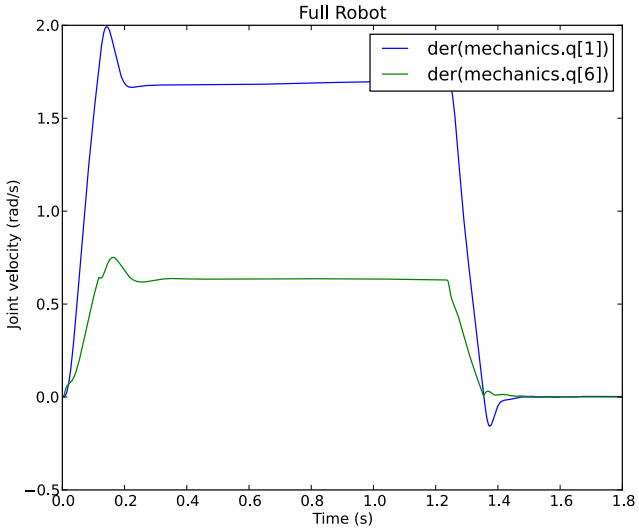
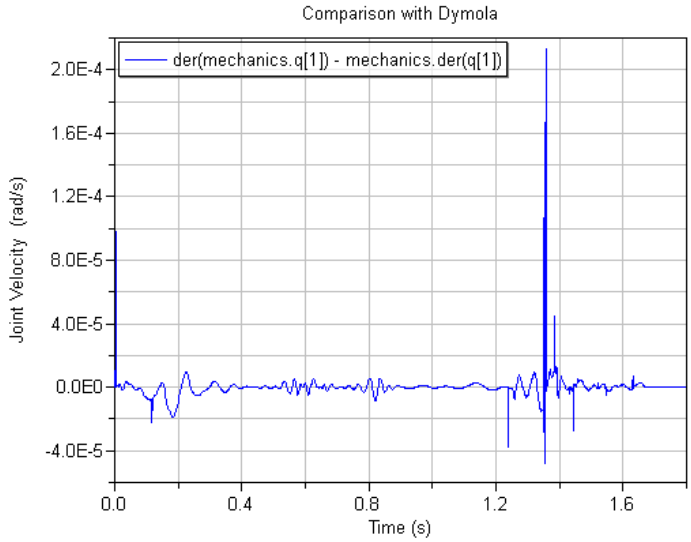


Figure 7.11. Comparison with Dymola



Chapter 8. Optimization

1. Introduction

JModelica.org supports optimization of dynamic and steady state models. Many engineering problems can be cast as optimization problems, including optimal control, minimum time problems, optimal design, and model calibration. In this these different types of problems will be illustrated and it will be shown how they can be formulated and solved. The chapter starts with an introductory example in Section 2 and in Section 3, the details of how the optimization algorithms are invoked are explained. The following sections contain tutorial exercises that illustrates how to set up and solve different kinds of optimization problems.

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

2. A first example

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

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

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

  // The control signal
  input Real u;

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

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

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

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

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

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

Next, we compile and load the model:

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

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

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

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

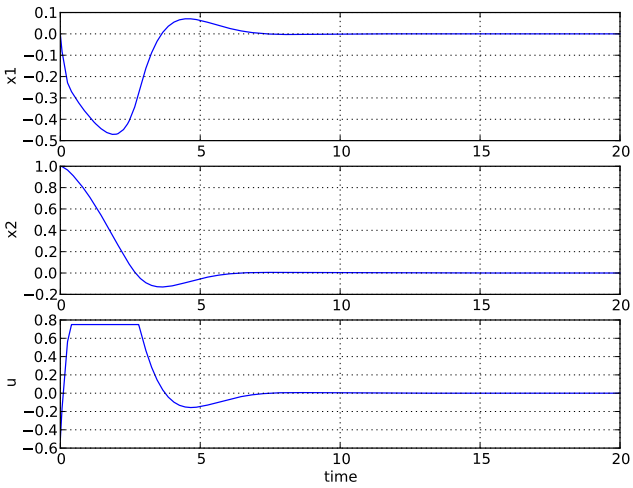
```
# Extract variable profiles
x1=res['x1']
x2=res['x2']
u=res['u']
t=res['time']
```

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

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

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

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

You should now see the optimization result as shown in Figure 8.1.

Figure 8.1.



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

3. Solving optimization problems

The first step when solving an optimization problem is to formulate a model and an optimization specification and then compile the model as described in XX. To illustrate how to solve optimization problems the Van der Pol problem presented above is used. First, the model is compiled and loaded:

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

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

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

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

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

    Parameters::

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

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

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

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

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

    Returns::

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

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

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

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

```
>>> help(vpd.optimize_options)
    Get an instance of the simulate options class, prefilled with
    default values. If called without argument then the options
    class for the default simulation algorithm will be returned.

    Parameters::

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

    Returns::

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

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

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

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

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

Available options for supported algorithms are documented in Section 3.1.

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

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

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

3.1. Algorithms

3.1.1. Direct collocation

The direct collocation method supported by JModelica.org can be used to solve dynamic optimization problems, including optimal control problems and parameter optimization problems. In the collocation method, the dynamic model variable profiles are approximated by piecewise polynomials. This method of approximating a differential equation corresponds to a fixed step implicit Runge-Kutta scheme, where the mesh defines the length of each step. Also, the number of collocation points in each element, or step, needs to be provided. This number corresponds to the stage order of the Runge-Kutta scheme. The selection of mesh is analogous to the choice of step length in a one-step algorithm for solving differential equations. Accordingly, the mesh needs to be fine-grained enough to ensure sufficiently accurate approximation of the differential constraint. For an overview of simultaneous optimization algorithms, see [2]. The algorithm IPOPT is used to solve the non-linear program resulting from collocation.

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

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

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

Table 8.1. Options for the collocation-based optimization algorithm

Option	Default	Description
`n_e`	50	Number of elements of the finite element mesh.
`n_cp`	3	Number of collocation points in each element. Values between 1 and 10 are supported.
`hs`	Equidistant points using default n_e	A vector containing n_e elements representing the finite element lengths. The sum of all element should equal to 1.
`blocking_factors`	None (not used)	A vector of blocking factors. Blocking factors are specified by a vector of integers, where each entry in the vector corresponds to the number of elements for which the control profile should be kept constant. For example, the blocking factor specification [2,1,5] means that u_0=u_1 and u_3=u_4=u_5=u_6=u_7 assuming that the number of elements is 8. Notice that specification of blocking factors implies that controls are present in only one collocation point (the first) in each element. The number of constant control levels in the optimization interval is equal to the length of the blocking factor vector. In the example above, this implies that there are three constant control levels. If the sum of the entries in the blocking factor vector is not equal to the number of elements, the vector is normalized, either by truncation (if the sum of the entries is larger than the number of element) or by increasing the last entry of the vector. For example, if the number of elements is 4, the normalized blocking factor vector in the example is [2,2]. If the number of elements is 10, then the normalized vector is [2,1,7].
`init_traj`	None (i.e. not used, set this argument to activate initialization)	Variable trajectory data used for initialization of the optimization problem. The data is represented by an object of the type `jmodelica.io.DymolaResultTextual`.
`result_mode`	'default'	Specifies the output format of the optimization result. 'default' gives the the optimization result at the collocation points. 'element_interpolation' computes the values of the variable trajectories using the collocation interpolation polynomials. The option 'n_interpolation_points' is used to specify the number of evaluation points within each finite element. 'mesh_interpolation' computes the values of the variable trajectories at points defined by the option 'result_mesh'.
`n_interpolation_points`	20	Number of interpolation points in each finite element if the result reporting option result_mode is set to 'element_interpolation'.
`result_mesh`	None	A vector of time points at which the the optimization result is computed. This option is used if result_mode is set to 'mesh_interpolation'.
`result_file_name`	Empty string (default generated file name will be used)	Specifies the name of the file where the optimization result is written. Setting this option to an empty string

Option	Default	Description
		results in a default file name that is based on the name of the optimization class.
`result_format`	'txt'	Specifies in which format to write the result. Currently only textual mode is supported.
`write_scaled_result`	False	Write the scaled optimization result if set to true. This option is only applicable when automatic variable scaling is enabled. Only for debugging use.

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

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

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

4. Optimal control

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

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

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

This tutorial will cover the following topics:

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

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

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

4.1. Compile and instantiate a model object

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

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

from jmodelica.jmi import compile_jmu
from jmodelica.jmi import JMUModel
```

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

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

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

Notice that automatic scaling of the model is enabled by setting the compiler option `enable_variable_scaling` to true. At this point, you may open the file CSTR.mop, containing the CSTR model and the static initialization model used in this section. Study the classes CSTR.CSTR and CSTR.CSTR_Init and make sure you understand the models. Before proceeding, have a look at the interactive help for one of the functions you used:

```
In [8]: help(compile_jmu)
```

4.2. Solve the DAE initialization problem

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

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

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

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

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

Notice how the method `set` is used to set the value of the control input. The initialization algorithm is invoked by calling the `JMUModel` method `initialize`, which returns a result object from which the initialization result can be accessed. The `initialize` method relies on the algorithm IPOPT for computing the solution of the initialization problem. The values of the states corresponding to 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` method and take a moment to look through it. The procedure is now repeated for operating point B:

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

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

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

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

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

4.3. Solving an optimal control problem

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

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

subject to

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

$$T(t) \le 350$$

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

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

First, compile the model and set model parameters:

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

# Load the model
init_sim_model = JMUModel(jmu_name)

# Set model parameters
init_sim_model.set('cstr.c_init',c_0_A)
init_sim_model.set('cstr.T_init',T_0_A)
init_sim_model.set('c_ref',c_0_B)
init_sim_model.set('T_ref',T_0_B)
init_sim_model.set('Tc_ref',Tc_0_B)
```

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

```
res = init_sim_model.simulate(start_time=0.,final_time=150.)
```

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

The simulation result object is returned and to retrieve the simulation data use Python dictionary access to retrieve the variable trajectories.

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

# Plot the results
plt.figure(1)
plt.clf()
```

```
plt.hold(True)
plt.subplot(311)
plt.plot(t_init_sim,c_init_sim)
plt.grid()
plt.ylabel('Concentration')

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

plt.subplot(313)
plt.plot(t_init_sim,Tc_init_sim)
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

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

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

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

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

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

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

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

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

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

```
n_e = 100 # Number of elements

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

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

In this case, we would like to increase the number of finite elements in the mesh from 50 to 100. This is done by setting the corresponding option and provide it as an argument to the `optimize` method. You should see the output of Ipopt in the Python shell as the algorithm iterates to find the optimal solution. Ipopt should terminate with a message like 'Optimal solution found' or 'Solved to an acceptable level' in order for an optimum to be found. The optimization result object is returned and the optimization data are stored in `res`.

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

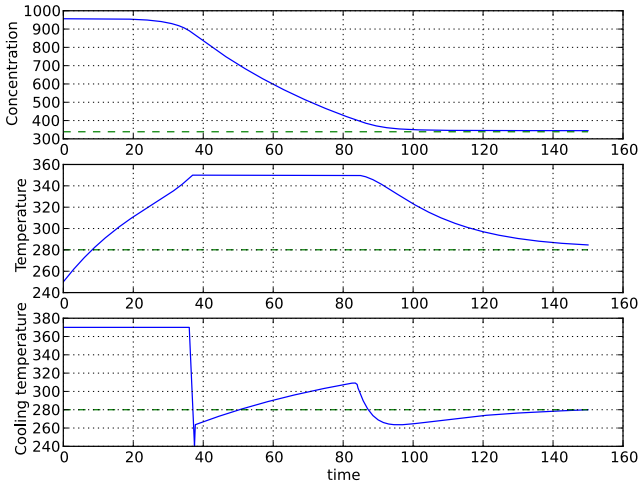
```
# Extract variable profiles
c_res=res['cstr.c']
T_res=res['cstr.T']
Tc_res=res['cstr.Tc']
time_res = res['time']

c_ref=res['c_ref']
T_ref=res['T_ref']
Tc_ref=res['Tc_ref']
```

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

```
# Plot the result
plt.figure(2)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(time_res,c_res)
plt.plot([time_res[0],time_res[-1]],[c_ref,c_ref],'--')
plt.grid()
plt.ylabel('Concentration')

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

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

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

Figure 8.2.



Optimal profiles for the CSTR problem.

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

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

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

4.4. Verify optimal control solution

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

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

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

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

# Load model
sim_model = JMUModel(jmu_name)

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

res = sim_model.simulate(start_time=0.,final_time=150.,
    input_trajectory=u_traj)
```

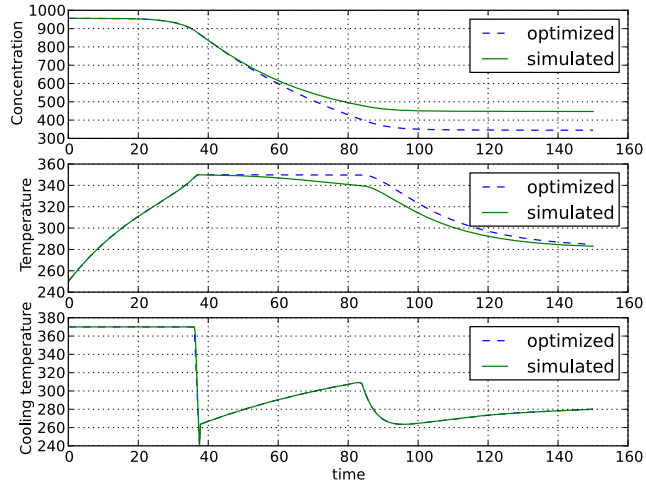
Finally, we load the simulated data and plot it to compare with the optimized trajectories:

```
# Extract variable profiles
```

```
c_sim=res['c']
T_sim=res['T']
Tc_sim=res['Tc']
time_sim = res['time']

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

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

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

You should now see the plot shown in Figure 8.3.

Figure 8.3.



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

Discuss why the simulated trajectories differs from the optimized counterparts.

4.5. Exercises

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

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

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

3. Add terminal constraints ('cstr.T(finalTime)=someParameter') for the states so that they are equal to point B at the end of the optimization interval. Now reduce the length of the optimization interval. How short can you make the interval?

4. Try varying the number of elements in the mesh and the number of collocation points in each interval. 2-10 collocation points are supported.

4.6. References

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

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

5. Minimum time problems

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

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

subject to the Van der Pol dynamics:

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

and the constraints:

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

This problem is encoded in the following Optimica specification:

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

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

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

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

constraint
  // terminal constraints
```

```
  x1(finalTime)=0;
  x2(finalTime)=0;
end VDP_Opt_Min_Time;
```

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

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

# Import the JModelica.org Python packages
from jmodelica.jmi import compile_jmu
from jmodelica.jmi import JMUModel

model_name = 'VDP_pack.VDP_Opt_Min_Time'
mo_file = curr_dir+'/files/VDP.mop'

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

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

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

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

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

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

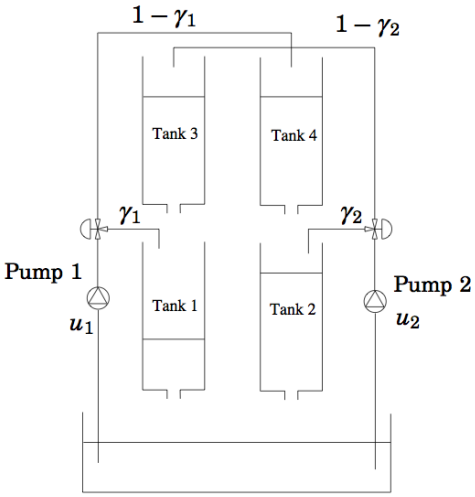
Figure 8.4.

Minimum time profiles for the Van der Pol Oscillator.

6. Parameter optimization

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

Figure 8.5.



A schematic picture of the quadruple tank process.

The dynamics of the system 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 Table 8.2.

Table 8.2. Parameters for the quadruple tank process.

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

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

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

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

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

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

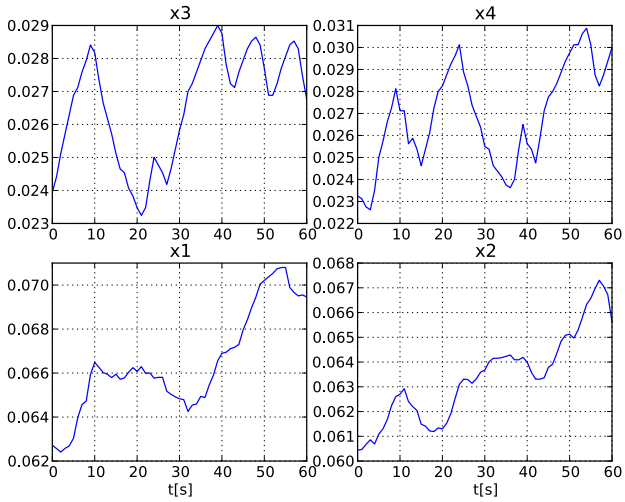
```
import numpy as N

from jmodelica.jmi import compile_jmu
from jmodelica.jmi import JMUModel
```

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

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

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

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

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

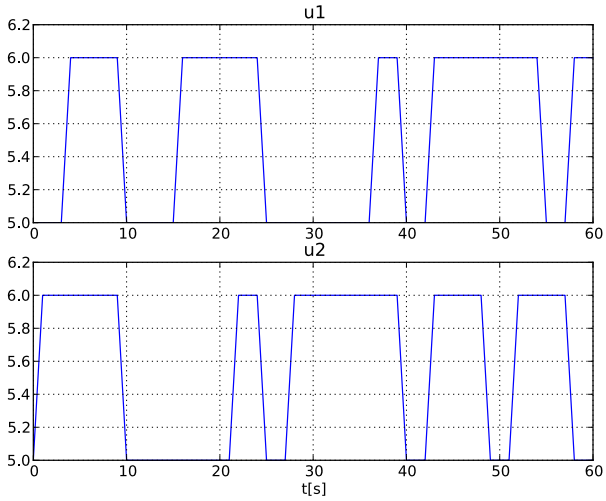
You should now see two plots showing the measurement state profiles and the control input profiles similar to Figure 8.6 and Figure 8.7.

Figure 8.6.



Measured state profiles.

Figure 8.7.



Control inputs used in the identification experiment.

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

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

Notice that initial equations have been added to the model. Before the model is simulated, a matrix containing the input trajectories is created:

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

Now, the model can be simulated:

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

# Load model
model = JMUModel(jmu_name)

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

The simulation result can now be extracted:

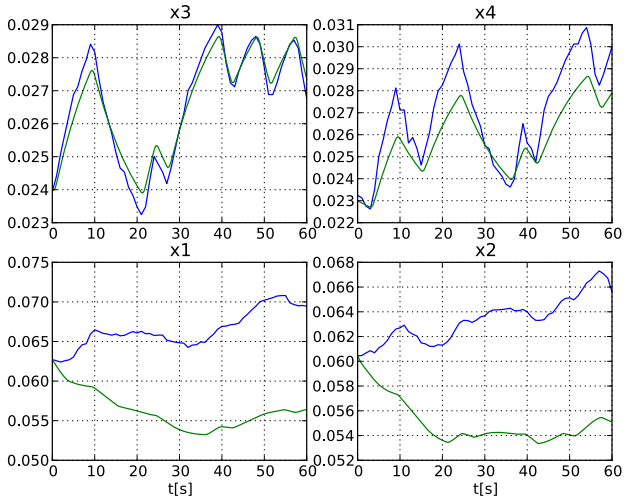
```
# Load simulation result
x1_sim = res['qt.x1']
x2_sim = res['qt.x2']
x3_sim = res['qt.x3']
x4_sim = res['qt.x4']
t_sim  = res['time']
u1_sim = res['u1']
u2_sim = res['u2']
```

and then plotted:

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

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

Figure Figure 8.8 shows the result of the simulation.

Figure 8.8.



Simulation result for the nominal model.

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

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

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

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

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

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

```
  connect(prbs1.y,qt.u1);
  connect(prbs2.y,qt.u2);
end QuadTank_ParEst;
```

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

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

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

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

Next, we load the measurement data into the model:

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

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

We are now ready to solve the optimization problem:

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

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

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

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

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

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

You should get an output similar to:

```
a1: 0.0266cm^2
a2: 0.0272cm^2
```

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

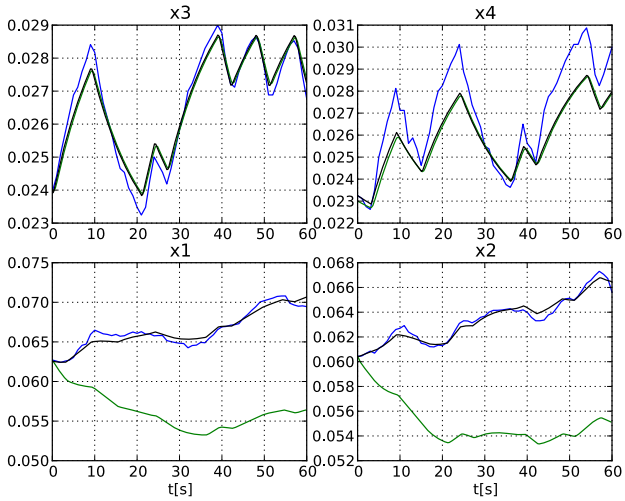
Next we plot the optimized profiles:

```
# Load state profiles
x1_opt = res["qt.x1"]
```

```
x2_opt = res["qt.x2"]
x3_opt = res["qt.x3"]
x4_opt = res["qt.x4"]
u1_opt = res["qt.u1"]
u2_opt = res["qt.u2"]
t_opt  = res["time"]

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

You will see the plot shown in Figure Figure 8.9.

Figure 8.9.



State profiles corresponding to estimated values of a1 and a2.

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

Never the less, There is still a mismatch for the upper tanks, especially for tank 4. In order to improve the match, a second estimation problem may be formulated, where the parameters a1, a2, a3, a4 are free optimization variables, and where the squared errors of all four tank levels are penalized. Take a minute to locate the class `QuadTankPack.QuadTank_ParEst2` and make sure that you understand the model. Solve the optimization problem by typing the Python code:

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

# Load model
qt_par_est2 = JMUModel(jmu_name)

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

# Set measurement data into model
```

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

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

Next, we print the optimal parameter values:

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

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

The output in the console should be similar to:

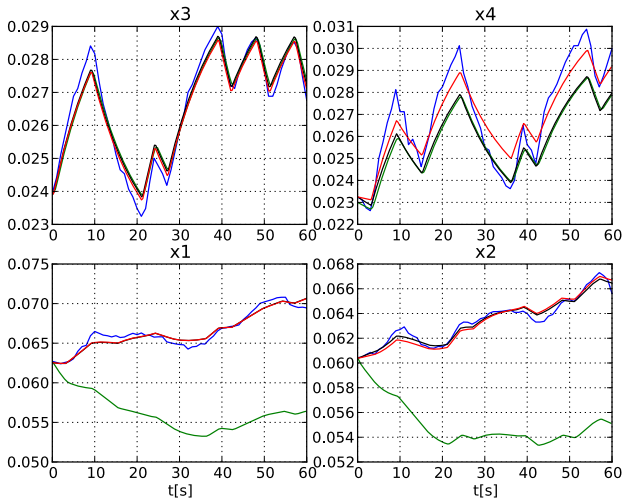
```
a1:0.0266cm^2
a2:0.0271cm^2
a3:0.0301cm^2
a4:0.0293cm^2
```

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

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

# Plot
plt.figure(1)
plt.subplot(2,2,1)
plt.plot(t_opt2,x3_opt2,'r')
plt.subplot(2,2,2)
plt.plot(t_opt2,x4_opt2,'r')
plt.subplot(2,2,3)
plt.plot(t_opt2,x1_opt2,'r')
plt.subplot(2,2,4)
plt.plot(t_opt2,x2_opt2,'r')
plt.show()
```

The resulting plot is shown in Figure Figure 8.10.

Figure 8.10.



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

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

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

7. Scaling

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

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

Here, the `nominal` attribute is used to specify that the variable pressure takes on values which are about 1e5. In order to use `nominal` attributes for scaling, the compiler option `enable_variable_scaling` is set to `True`, see Section 2.2.2. All variables with a nominal attribute set to true, is then scaled by dividing the variable value with its nominal value, i.e., from an algorithm point of view, all variables will take on values close to one. Notice that variables typically vary during a simulation or optimization and that it is therefore not possible to obtain perfect scaling. In order to ensure that model equations are fulfilled, each occurrence of a variable is multiplied with its nominal value in equations. For example, the equation:

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

is replaced by the equation

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

when `enable_variable_scaling` is set to `true`.

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

Chapter 9. Optimica

In this chapter, the Optimica extension will be presented and informally defined. The Optimica extension in described in detail in [Jak2008a], where additional motivations for introducing Optimica can be found.The presentation will be made using the following dynamic optimization problem, based on a double integrator system, as an example:

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

subject to the dynamic constraint

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

and

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

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

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

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

- A new specialized class: `optimization`
- New attributes for the built-in type Real: `free` and `initialGuess`
- A new function for accessing the value of a variable at a specified time instant
- Class attributes for the specialized class `optimization`: objective}, `startTime`, `finalTime` and `static`
- A new section: `constraint`
- Inequality constraints

1. A new specialized class: `optimization`

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

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

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

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

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

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

2. Attributes for the built in class Real

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

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

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

3. A Function for accessing instant values of a variable

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

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

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

4. Class attributes

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

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

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

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

5. Constraints

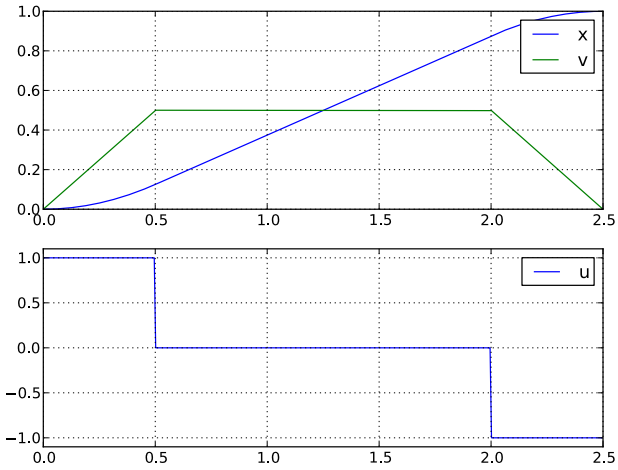
Constraints are similar to equations, and in fact, a path equality constraint is equivalent to a Modelica equation. But in addition, inequality constraints, as well as point equality and inequality constraints should be supported. It is therefore natural to have a separation between equations and constraints.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

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

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

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

Figure 9.1. Optimization result



Chapter 10. Abstract syntax tree access

1. Tutorial on Abstract Syntax Trees (ASTs)

1.1. About Abstract Syntax Trees

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

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

- The source AST results from parsing of the Modelica or Optimica source code. This AST shares the structure of the source code, and consists of a hierarchy consisting of Java objects corresponding to class and component declarations, equations and algorithms. The source AST can also be used for unparsing, i.e., pretty printing of the source code.
- The instance AST represents a particular model instance. Typically, the user selects a class to instantiate, and the compiler then computes the corresponding instance AST. The instance AST differs from the source AST in that in the former case, all components are expanded down to variables of primitive type. An important feature of the instance AST is that it is used to represent modification environments; merging of modifications takes place in the instance AST. As a consequence, all analysis, such as name and type analysis takes is done based on the instance AST.
- The flat AST represents the flat Modelica model. Once the instance AST has been computed, the flat AST is computed simply by traversing the instance AST and collecting all variables of primitive type, all equations and all algorithms. The flat AST is then used, after some transformations, as a basis for code generation.

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

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

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

Three different usages of ASTs are shown:

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

The Python commands in this tutorial may be copied and pasted 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 11. 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
  - 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 source code.
- Options for setting and getting string variables does not work

Chapter 12. Release notes

1. Release notes for JModelica.org version 1.4

1.1. Highlights

- Improved Python user interaction functions
- Improvements in compiler front-end
- Support for sensitivity analysis of DAEs using Sundials

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

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

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

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

3. Release notes for JModelica.org version 1.2

3.1. 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

3.2. Compilers

3.2.1. The Modelica compiler

3.2.1.1. 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.

3.2.1.2. 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)

3.2.1.3. 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.

3.2.1.4. 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.

3.2.2. The Optimica Compiler

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

3.3. The JModelica.org Model Interface (JMI)

3.3.1. General

3.3.1.1. 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 scal-

ing is turned off by default since it introduces a slight computational overhead: setting the compiler option `enable_variable_scaling` to `true` enables this feature.

3.3.1.2. 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.

3.3.1.3. 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.

3.3.1.4. 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.

3.4. The collocation optimization algorithm

3.4.1. 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.

3.4.2. 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.

3.4.3. 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.

3.5. 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.

3.6. 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.

3.7. 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.

3.8. 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.

3.8.1. 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.

3.9. Contributors

Christian Andersson

Tove Bergdahl

Magnus Gäfvert

Jesper Mattsson

Philip Nilsson

Roberto Parrotto

Philip Reuterswärd

Johan Åkesson

3.9.1. Previous contributors

Jens Rantil

Bibliography

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

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

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

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

Index