LUND UNIVERSITY

PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface

Andersson, Christian; Åkesson, Johan; Führer, Claus

2016

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Andersson, C., Åkesson, J., & Führer, C. (2016). *PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface*. (Technical Report in Mathematical Sciences; Vol. 2016, No. 2). Centre for Mathematical Sciences, Lund University.

Total number of authors:
3

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface

Christian Andersson, Johan Åkesson, Claus Führer

Preprints in Mathematical Sciences
2016:2

LUND UNIVERSITY

Faculty of Engineering
Centre for Mathematical Sciences
Numerical Analysis

CENTRUM SCIENTIARUM MATHEMATICARUM

PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface

Christian Andersson[a,b], Johan Åkesson[b], Claus Führer[a]

[a] *Centre for Mathematical Sciences, Lund University, Lund, Sweden*
[b] *Modelon AB, Lund, Sweden*

Abstract

With the advent of the Functional Mock-up Interface (FMI) standard, exchanging dynamic models between modeling and simulation tools has been greatly simplified. At the core of it, FMI is a standardized and unified model execution interface for dynamic models. FMI has gained widespread adoption among users and numerous commercial and open source tools implement support for the standard. In this article, the Python package PyFMI is introduced. PyFMI supports loading and execution of models compliant with the FMI standard, called Functional Mock-up Units (FMUs). It includes a master algorithm for simulation of coupled FMUs together with connections to both Assimulo, for simulation of single FMUs, and to SciPy, for performing parameter estimation. Accessing models compliant with FMI in Python, which is an open and accessible scripting language, is intended to further spread the standard and also promote and facilitate future development of the standard. This is due to Python being a convenient language for experimentation and prototyping of numerical algorithms. PyFMI is also demonstrated on a number of problems that highlights its viability for solving industrial grade simulation problems with FMUs.

Keywords: Functional Mock-up Interface; FMI; Python; Simulation; Co-Simulation; Ordinary differential equations; Parameter Estimation;

1. Introduction

Different simulation and modeling tools often use their own definition of how a model is represented and how model data is stored. Complications arise when trying to model parts in one tool and importing the resulting model in another tool, or when trying to verify a result by using a different simulation tool. The Functional Mock-up Interface (FMI) [1] is a standard to provide a unified model execution interface for exchanging dynamic system models between modeling

Email addresses: `chria@maths.lth.se` (Christian Andersson), `johan.akesson@modelon.com` (Johan Åkesson), `claus@maths.lth.se` (Claus Führer)

tools and simulation tools. The standard has been a great success as numerous tools[1], both open source and commercial have adopted it as well as having gained widespread adoption among users.

In this article, the Python package PyFMI is introduced. PyFMI supports loading and execution of models compatible with the FMI standard. Such models are called Functional Mock-up Units (FMUs). PyFMI is based on the open source package FMI Library [2]. It is designed to provide a high-level, easy to use, interface for working with FMUs. It connects the full set of methods in the FMI specification in an object-oriented approach. The package is not only a mapping of the FMI interface to Python, it provides much of the functionality needed to perform various experiments for both evaluating the complex dynamical system model by itself but also for evaluating the physics that the model represents. The evaluation of the model could be to verify the model dynamics by efficient simulation while evaluations of the physics could be to performing parameter estimations. These experimentations requires an extensive tool beyond the low-level FMI interface which motivates the package. Furthermore, with the FMI, simulation of coupled models in a co-simulation setting is possible. In this setting, the dynamics of each system is hidden and exchanging information between systems is done through inputs and outputs. This is important as in many cases, with complex systems, this is the only viable option due to that parts of the model is modeled in different tools. An algorithm for performing a co-simulation is called a *Master Algorithm* and within PyFMI a master algorithm has been implemented and made available.

PyFMI has been used successfully in a number of different applications such as in [3] and [4] as well as in [5]. It is additionally an integral part of the open source JModelica.org platform [6].

2. Motivation

The FMI standard fills a gap where before there has been costly custom solutions for coupling specific simulation environments and exchanging models between tools. In order to promote widespread use of the FMI standard and make it easily accessible, there is a need for an open package in an open platform for experimenting and working with FMUs. Furthermore, the FMI standard specifies a low-level interface in *C* that while efficient, not very user friendly, which means that there is a need for an high-level package for conveniently working with FMUs.

PyFMI grew from [7] where there was a need for working with the standard in the open source tool JModelica.org. It is written in the Python programming language which is a powerful dynamic programming language with a clear and readable syntax. The choice of using Python is motivated by the momentum the language has in scientific computing due to the many freely available packages, notably NumPy and SciPy [8], but also due to the fact that the language is easy

[1] https://www.fmi-standard.org/tools [accessed: 2016-03-18]

to learn, especially if the user has a background in Matlab. Moreover, together with Cython [9], efficiency and high-performance can be achieved. By providing an interface for working with FMUs from Python, the model is exposed to the full ecosystem that Python has to offer. Visualization and animations of simulation results can be done through matplotlib [10] and the flexibility that Python offers make it suitable for prototyping. The choice of Python is further motivated by the ease of connecting software written in different programming languages, such as C.

PyFMI is commonly used with Assimulo [11]. Assimulo is a package that provides solvers for solving dynamical systems, such as those represented by FMUs. The packages complement each other as Assimulo provides the solvers and PyFMI provides the problems.

PyFMI offers an open platform for working with FMUs and the algorithms that are included are open and accessible for modifications and further experimentations. PyFMI includes an open and available master algorithm for simulation of coupled FMUs. In a related work, the PySimulator [12], there is also support for working with FMUs from Python. In their case they use a different approach for coupling the FMUs to Python and are more focused on post processing of simulation results. Furthermore, there is no included master algorithm.

3. The Functional Mock-up Interface

The FMI standard is designed to provide a unified model execution interface for dynamic system models between modeling tools and simulation tools, Figure 1. The generated models, FMUs, are distributed and shared as compressed

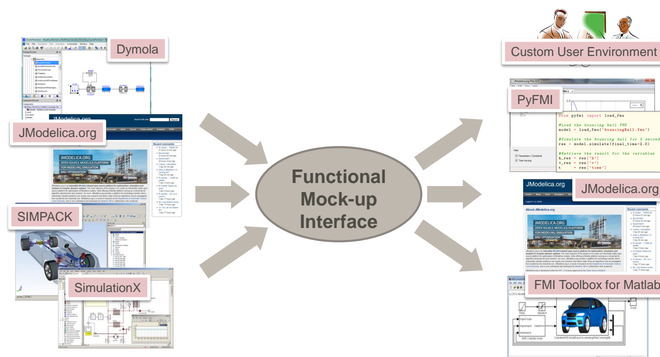

Figure 1: Exchange of dynamical models following the Functional Mock-up Interface.

archives. They include a shared object file containing the model information which is accessed through the FMI interface. The archive additionally contains an XML file containing metadata of the model, such as the sizes of the dynamic system and the names of the variables, parameters, constants and inputs. FMI

specifies two types of models, one named model exchange FMU, exposing an, possibly discontinuous, ordinary differential equation (Figure 2) while the other

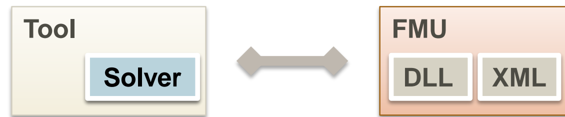

Figure 2: Overview of a model exchange FMU, note that an external solver is required by the tool in order to solve the model.

type is co-simulation FMU, exposing only the ability to perform a step in time (Figure 3). PyFMI supports both model types as-well as versions 1.0 [13, 14] and 2.0 [1].

Figure 3: Overview of a co-simulation FMU, note that the solver in this case is within the FMU.

The FMI standard specifies methods for interacting with a model, providing variable values and retrieving values. Methods for computing the derivatives, setting states and time are available for model exchange and for co-simulation, a method for performing an internal step are available. Furthermore, there are specific methods for initialization of a model. The interface is light-weight and much of the information about the internals are contained in the metadata, i.e. the XML file inside the FMU, which needs to be made available.

The standard describes a model exchange FMU with the following underlying mathematical representation,

$$\dot{x} = f(t, x, u; p, d) \tag{1a}$$

$$y = g(t, x, u; p, d) \tag{1b}$$

where t is the time, x are the continuous states, u are the inputs, p are the parameters and d are the discrete variables that are kept constant between events. Furthermore, y are the outputs. Additionally, the standard supports different types of events which can impact the model behavior. The three events are:

- *State Events*
  These events are dependent on the state solution profiles and thus not known a prior. The model provides a set of event indicators, z, that the

integrator monitors during the integration process,

$$z = h_{\text{state}}(t, x, u; d, p). \tag{2}$$

If one of the event indicators, z_i, switches domain, there is a state event. The integrator is then responsible for finding the time when the event occurred.

- *Time Events*
  These events on the other hand are known a prior, meaning that for each simulation segment it is known when the time event occur and thus this time is set as the simulation end time for that segment. Given a previous time event, T_{pre} (or the initial time, T_0), the next time event is computed using,

  $$T_{\text{next}} = h_{\text{time}}(t_{T_{\text{pre}}}, x_{T_{\text{pre}}}, u_{T_{\text{pre}}}; d_{T_{\text{pre}}}, p). \tag{3}$$

  An example is that after a certain elapsed time in the integration, a force is applied on the model.

- *Step Events*
  These last type of events are events that typically do not influence the model behavior, instead they are events to ease the numerical integration. For instance it can be a change of the continuous states in the model as the current states are no longer appropriate numerically. After each successful integrator step, T_{accepted}, the equation,

  $$E_{\text{step}} = h_{\text{step}}(t_{T_{\text{accepted}}}, x_{T_{\text{accepted}}}, u_{T_{\text{accepted}}}; d_{T_{\text{accepted}}}, p) \tag{4}$$

  is evaluated and if E_{step} is True, a step event is triggered.

Further, Equation 1 is valid during continuous simulation and prior to this, the FMI specifies that the FMU need to be initialized. The simulation and initialization is separated as to allow a flexible definition of initial conditions. An example could be that the initial values for the states are computed using an initial equation, which is only active during the initialization. The initial equations are described by,

$$\hat{x}_0, \hat{d}_0, \hat{p} = f_{\text{init}}(t_0, \bar{x}_0, u_0; \bar{d}_0, \bar{p}), \tag{5}$$

where $\bar{x}_0$ are states with known initial values, $\bar{d}_0$ are the known discrete variables and $\bar{p}$ are known parameters. The complete initial states vector is $x_0 = [\bar{x}_0, \hat{x}_0]$, and for discrete variables, $d_0 = [\bar{d}_0, \hat{d}_0]$, while the full parameter vector is $p = [\bar{p}, \hat{p}]$.

For full details about the mathematical representation, cf. [1] for version 2.0 and [13] for version 1.0.

For co-simulation, the standard rather describes a discrete interface to the underlying dynamic model, i.e. given the current internal state, input u_n and step size H of the model, return the outputs, y_{n+1}, at a time $T_n + H = T_{n+1}$,

$$y_{n+1} = \Phi(H, u_n; p), \tag{6}$$

where p are the parameters. The advancement of the states and time are completely hidden outside of the model and is also not specified by the standard, cf. Figure 4. A consequence of this is that, if there are events, these are also handled internally and are not visible from the outside. There is additionally the restriction that the outputs, y, cannot be evaluated, during simulation, for different inputs, u, without advancing the solution, i.e. performing a step with $H > 0$. However, as the advancement is hidden, this allows for specialized solvers to be

Figure 4: A co-simulation FMU and the connection to a tool for simulation. Note that the solver is inside the FMU.

used for the particular subsystem at hand, which may give an increased performance and a more stable simulation. As in the model exchange case, the initialization is done separately for co-simulation FMUs. The initialization is defined by,

$$\hat{p} = f_{\text{init}}(t_0, u_0; \bar{p}). \tag{7}$$

where $\bar{p}$ are known parameters. The full parameter vector is $p = [\bar{p}, \hat{p}]$.

For full details about the mathematical representation, cf. [1] for version 2.0 and [14] for version 1.0.

Between version 1.0 and version 2.0 of the standard, changes have been made. The most significant changes are:

- **Save/Get state**: Methods for retrieving and restoring the internal state of an FMU have been added.
- **Dependency information**: Information about which states and inputs impact the derivatives and the outputs has been added.
- **Directional derivatives**: Methods for computing the directional derivatives of Equation 1 and Equation 6 has been added.
- **Separate initialization**: The initialization of an FMU has been separated into a state of the FMU instead of a single call to an initialization method.
- **Tunable parameters**: Tunable parameters has been added which allow these types of parameters to be changed at events.

4. Overview and analyses

The FMI standard describes a light-weight interface for interacting with a model which by itself does not include any analyses of the dynamic model. In this section, the available capabilities of PyFMI is described and shown how they can be used. The major features of the package is linearization of an FMU, Section 4.1, simulation of an FMU, Section 4.2, simulation of coupled FMUs, Section 4.3, and estimation of parameters within an FMU, Section 4.4.

These analyses are necessary in order to support model-based design workflows. Linearization of a model is useful when, for instance, designing control systems using classical approaches and when analyzing stability of the model. Simulation and simulation of coupled systems are vital to understanding the dynamics and how the dynamics behave over time. With efficient simulation and access to solvers that are appropriate for a given problem, the return time is reduced resulting in more time for experimentations. Furthermore, a common situation in a model of a system is that not all parameters are given, only an approximation is known due to that the parameter can be hard to measure on a physical system that the model represents. In these cases, parameter estimation is key to make the model more representative of the physical system.

For illustration purposes, we consider a model of the van der Pol oscillator given by,

$$\dot{x}_1 = \mu[(1 - x_2^2)x_1 - x_2] + u, \quad x_1(t_0) = -0.6 \tag{8a}$$

$$\dot{x}_2 = x_1, \quad x_2(t_0) = 2 \tag{8b}$$

where $\mu = 20$ and u is the input signal. The Modelica code of the example is shown in Appendix A and the model is compiled into an FMU named `VDP.fmu`.

As a first step for using PyFMI, the FMU needs to loading into Python. In Example 4.1 this is explained. In Example 4.2, it is shown how to interact with the FMU.

Example 4.1 (Loading an FMU)**.** *The first step for working with FMUs is to load the model into Python, i.e. couple the binary from the FMU and read the model description containing information about the variables etc.*

```
#Convenience function for loading a general FMU
from pyfmi import load_fmu

#Loads the FMU and return a model object
model = load_fmu("VDP.fmu")
```

The FMU is automatically extracted and the metadata is read together with coupling of memory handling. If the model is discarded, memory is automatically handled and deallocated if necessary. No manual handling of memory is necessary.

Example 4.2 (Interacting with a model)**.** *Once the model is loaded into Python, values can be retrieved from the model using the high-level get/set methods.*

```
#Get the value of the variable 'mu'
mu = model.get("mu") #.set for setting values
print mu
>>> 20
```

For variable attributes, these can be obtained similarly,

```
#Get the start value of variable 'x1'
start_x1 = model.get_variable_start("x1")
print start_x1
>>> -0.6
```

All attributes such as min, max, nominal and start can just as easily be retrieved.

4.1. Linearization

For analyzing the dynamics of the system, linearizing the model (Equation 1) is usually the first step. The linearized state space form for a model exchange FMU is,

$$\dot{x} = Ax + Bu \tag{9a}$$

$$y = Cx + Du. \tag{9b}$$

In FMI there is no direct way of computing the matrices in the linearized state space form (Equation 9). There are however, methods for computing the directional derivatives, in FMI 2.0, with respect to a set of variables (here either x, u or a subset of them) and a set of functions (f or g or a subset) together with a seed vector. The definition of the directional derivatives are,

$$g_z = \frac{\mathrm{d}g(z)}{\mathrm{d}z} v \tag{10}$$

where $g(z)$ is a vector-valued function, z is the vector of variables and v is the seed vector. From the directional derivatives, the partial derivatives, the matrices A, B, C, D in Equation 9, can directly be computed by a sequence of calls with v replaced by unit vectors.

If structural information is available, e.g. if the structural dependency between x_i, u_i and $\dot{x}_j$ is known and between x_i,u_i and y_j, compression can be employed such that the number of evaluations of either the directional derivatives or evaluations of f (in case a finite difference approximations is used) is reduced.

Consider the ODE in Equation 11,

$$\dot{x}_1 = x_1 \tag{11a}$$

$$\dot{x}_2 = x_2 + x_3 \tag{11b}$$

$$\dot{x}_3 = x_1 + x_3. \tag{11c}$$

Now consider that the Jacobian, $\frac{\mathrm{d}\dot{x}}{\mathrm{d}x}$, is computed using a first order finite difference scheme requiring $3 + 1$ evaluations of the derivatives. However, due to

the structure of the ODE, the partial derivatives $\frac{\partial \dot{x}}{\partial x_1}$ and $\frac{\partial \dot{x}}{\partial x_2}$ can be computed simultaneously as $\dot{x}_1$ and $\dot{x}_3$ are independent of x_2 and $\dot{x}_2$ are independent of x_1. This leads to that the construction of the Jacobian only need $2+1$ evaluations of the derivatives. In general, an adjacency matrix, determining the structural relation between the variables (states or inputs) and the functions (derivatives or outputs), can be constructed, Equation 12 for the given ODE.

$$A_{\text{adj}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}. \tag{12}$$

Given the adjacency matrix, a compression can be computed aimed at reducing the number of evaluations of the derivatives or outputs. In PyFMI, the algorithm proposed in [15] is used.

With the growing size of models and due to that in general the state space matrices are sparse, utilizing this information is essential in order for efficient handling of the system. Using SciPy, the ability to represent these matrices is available and supported by PyFMI.

For co-simulation FMUs, the derivatives are not exposed, cf. Equation 6. However, the above applies to the outputs which are available.

4.2. Simulation of single models

A key feature of the package is the connection to Assimulo, which provides capabilities for performing simulations of model exchange FMUs using ODE solvers interfaced with Assimulo. The coupling is made possible by extending

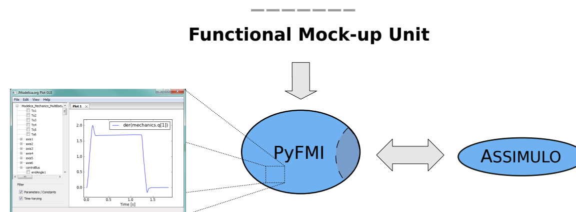

Figure 5: Coupling between PyFMI and Assimulo.

the definition of the problem classes accepted by Assimulo, Figure 5. With the extension, customizations related to FMI is made possible, such as exposing the different events in FMI to the solver.

Assimulo separates between a problem, which contains the problem equations, and the actual solver used for the integration. The problem object is not only limited to the derivatives equation, but it may also contain event functions which is necessary in the FMI case. Furthermore, the problem object can be used to define specific event handling and user defined result handling. All of these features are necessary in order to couple a model exchange FMU to a simulation environment. In Example 4.3, a simulation of an FMU is shown.

Example 4.3 (Simulating an FMU)**.** *A simulation of an FMU, either a model exchange or a co-simulation FMU, follows the same steps. First, the model is loaded.*

```
#Convenience function for loading a general FMU
from pyfmi import load_fmu

#Loads the FMU and return a model object
model = load_fmu("VDP.fmu")
```

Then, a simulation is performed by invoking the simulate method on the model object.

```
#Simulate the model
res = model.simulate(final_time=2)
```

The simulation results are returned in the `res` *object. In Figure 6, the simulation result for x_2 is shown.*

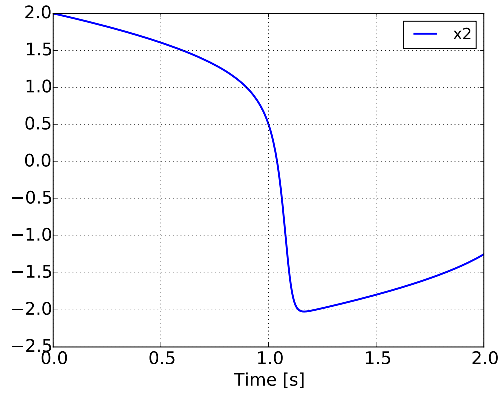

Figure 6: Simulation result of the van der Pol oscillator from Example 4.3.

A dynamical model typically has control signals or external forces acting on the model during a simulation. An example is the road profile for a vehicle. In general, this data is a list of points connected to a point in time,

$$(t_i, u_i), \quad i = 0, \ldots, N. \tag{13}$$

Within PyFMI, this data can be provided to the simulation setup and will be evaluated during the simulation using linear interpolation between the data points,

$$u(t) = u_i + (t - t_i)\frac{u_{i+1} - u_i}{t_{i+1} - t_i}, \quad t \geq t_i \wedge t < t_{i+1}, \quad i \in [0, N]. \tag{14}$$

Another option is that the expression for the control signals are known and for these cases providing a function, instead of data points,

$$u(t) = h(t) \tag{15}$$

is beneficial. The reason is that in the first case, discontinuities in higher derivatives are introduced which may degrade the performance of the simulation. In Example 4.4 an example using an input function is shown.

Example 4.4 (Inputs)**.** *In this example, an input function, $f(t) = 100 \sin(30t)$, is defined which provides the input to the variable u in the van der Pol oscillator model, Equation 8.*

```
import numpy as np #Import numpy

#Define the function, need to be dependent on time
def f(time):
    return 100*np.sin(30*time)

#Specify the input variable together with the function
input = ('u', f)

#Provide the input object to the simulate method
res = model.simulate(final_time=2, input=input)
```

During the integration, the input will be invoked during each evaluation of the model equations, for model exchange. For co-simulation FMUs, the input function will be evaluated at every step. In Figure 7, the simulation result for x_2

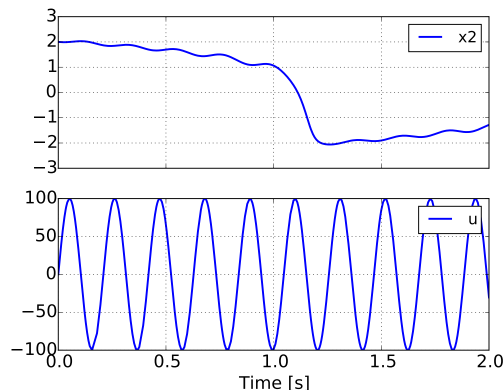

Figure 7: Simulation result of the van der Pol oscillator from Example 4.4 together with the input function, u.

and the input, u, is shown.

Furthermore, general options for controlling the simulation are available. In Example 4.5, changing options are shown and in Table 1, the options for controlling a simulation of a model exchange FMU is shown.

A simulation of a co-simulation FMU follow the syntax as shown above. However, as the simulation do not require an external solver, the options available are limited, cf. Table 2.

Example 4.5 (Providing options to the simulation)**.** *Setting options for controlling the simulation can be done in two steps. First the available options are retrieved from the model object.*

```
#Retrieves the options dictionary
opts = model.simulate_options()
```

The `opts` *object is a Python dictionary which contains the available options. Setting an option is done using the normal Python dictionary syntax.*

```
opts["ncp"] = 500 #Change the number of output points
```

For the simulation to use the options, these need to be provided when invoking the simulation.

```
#Provide the options object to the simulate method
res = model.simulate(options=opts)
```

Table 1: Description of the options available in the default algorithm in order to control the simulation of an model exchange FMU.

Option	**Description**
solver	*The ODE solver to use. By default, all solvers from Assimulo can be used.*
ncp	*The number of communication points, i.e. the number of requested (equally spaced) points to store the result.*
initialize	*If the model should be initialized or not.*
result_handling	*Determines how the result should be stored, either on file or directly in memory.*
result_handler	*Ability to specify a custom result handler.*
filter	*A filter for choosing which variables to actually store result for.*
extra_equations	*Determines if additional equations should be solved together with the model.*
{solver}_options	*Specifies additional solver specific options.*

Table 2: Description of the options available in the default algorithm for simulation of a co-simulation FMU.

Option	**Description**
ncp	*The number of communication points, i.e. the number of requested (equally spaced) points to store the result.*
initialize	*If the model should be initialized or not.*
result_handling	*Determines how the result should be stored, either on file or directly in memory.*
result_handler	*Ability to specify a custom result handler.*
filter	*A filter for choosing which variables to actually store result for.*

4.3. Simulation of coupled models

A second key feature of the package is the ability to simulate coupled systems, i.e. coupled FMUs. PyFMI implements a master algorithm which includes the approaches used for co-simulation discussed and analyzed in [16]. The implementation supports simulation of a coupled system via a parallel approach,

i.e. Jacobi-like, defined as (for M models),

$$y_{n+1}^{[i]} = \Phi^{[i]}(H, u_n^{[i]}; p), \quad i = 1, \dots M \tag{16}$$

$$u_{n+1} = c(y_{n+1}). \tag{17}$$

The algorithm proceeds by first providing inputs to a model and then performing a global time step, for the ith model: $y_{n+1}^{[i]} = \Phi^{[i]}(H, u_n^{[i]}; p)$. This can be done for all models simultaneously and once all models have performed the step, information is exchanged between the models and inputs for the next step are computed using the coupling equations, $c(\cdot)$. This is the commonly used approach for simulation of coupled systems. In Example 4.6, a simulation of a coupled system using PyFMI is explained.

Example 4.6 (Coupled system simulation)**.** *In order to simulate a coupled system, first of all the models needs to be loaded and collected together,*

```
sub_system1 = load_fmu("Subsystem1.fmu") #First model
sub_system2 = load_fmu("Subsystem2.fmu") #Second model
models = [sub_system1, sub_system2] #List the models
```

This list may contain an arbitrary number of models and the ordering in the list is irrelevant.

Secondly, the coupling needs to be specified. Here the following convention is used. First, from which model is the variable data coming from? It should be an reference to a model. Second, the name of the variable in the model where data is coming from. Thirdly, the reference to the receiving model and finally the name of the receiving variable.

```
#Connecting inputs / outputs from two models
connections = [(sub_system1,"x_chassi",sub_system2,"x_chassi"),
               (sub_system2,"v_chassi",sub_system1,"v_chassi")]
```

The connection list can contain an arbitrary number of connections.

The main implementation and the user entry-point is the `Master` *class for a simulation of a coupled systems. This class needs to be imported from the package.*

```
#Import of the Master object
from pyfmi import Master
```

The models together with their connections can then be loaded into the `Master` *class.*

```
#Create the simulator object
master_simulator = Master(models, connections)
```

Once the simulator object is created, a simulation is performed using the simulate method.

```
master_simulator.simulate(start_time=0.0, final_time=1.0)
```

The simulation statistics are printed and the simulation result are returned in the `res` object, just as in the case for a simulation of a single system.

Included in the master algorithm are variants of the above algorithm. Higher order extrapolation is possible for the inputs, from using constant polynomials, as is shown, up-to using quadratic polynomials for the inputs. Additionally, the update of inputs between time steps introduces discontinuities in the input signals due to the coupling equations. Using a smoothing approach on the inputs, continuity is preserved [16]. Another issue is the stability of the algorithm, depending on the couplings between the models the algorithm may become unstable. Using the directional derivatives, a stabilization can be performed. For details, cf. [16]. Both the smoothing and the stabilization is implemented in the master algorithm.

Furthermore, the master algorithm may used together with an error estimation based on Richardson extrapolation [17]. The estimate is based on performing a global integration step twice using different input. A first step is performed using a step size H. This step is compared with two steps of step size $H/2$ where inputs and outputs between the subsystems are updated before taking the second step of step size $H/2$.

For initialization, the master algorithm supports initialization based on graph cycle detection [16]. The idea is that the dependency information between inputs and outputs are used to detect cycles and in so doing, computing an evaluation order of the input / output variables of the separate models. This is done in order to simplify the initialization problem.

Simulation of coupled systems are restricted to models following the co-simulation interface for FMI 2 and as in the case of simulation of a single system, options are available to control the mater algorithm and are shown in Table 3.

Table 3: Description of the options available in the master algorithm in order to control the simulation of the coupled system.

Option	**Description**
step_size	*The global step size to be used when using the fixed step approach.*
extrapolation_order	*The order of the extrapolation for the coupling variables.*
linear_correction	*Defines if linear correction for the coupling variables should be used during the simulation.*
execution	*Defines if the models are to be evaluated in parallel or in serial.*
smooth_coupling	*Defines if the extrapolation should be smoothen, i.e. the coupling variables are adapted so that they are* $\mathcal{C}^0$ *instead of* $\mathcal{C}^{-1}$ *in case the extrapolation order is* > 0.
num_threads	*Specifies the number of threads to be used when the execution is set to parallel.*
error_controlled	*Defines if the algorithm should adapt the step size during the simulation.*
atol	*The absolute tolerance in case an error controlled simulation is performed.*
rtol	*The relative tolerance in case an error controlled simulation is performed.*
result_handling	*Specifies how the result should be handled. Either stored to file or stored in memory.*
filter	*A filter for choosing which variables to actually store result for.*

4.4. Parameter estimation

Verifying the dynamics of a model usually requires that parameters are validated against experimental data due to that not all parameters are known. The parameters can either be tuned manually or by an optimization aimed at minimizing the difference between experimental data and the model response. In [18], PyFMI were extended with parameter estimation using derivative free methods and which has since been further extended by coupling to SciPy's minimization algorithms and with an improved user interface. In Example 4.7, an example on how to perform parameter estimation is shown.

Example 4.7 (Parameter estimation)**.** *This example illustrates how parameter estimation can be performed. As before, the model is loaded into Python using the* `load_fmu` *method.*

```
from pyfmi import load_fmu

# Load model
model = load_fmu("MyModel.fmu")
```

Second, the measurment data need to be stacked into a matrix. Here it assumed that the data is stored in the arrays, `t_meas`*,* `x1_meas` *and in* `x2_meas`*.*

```
#Stack the measurement data into a matrix
#The measurments, x1_meas,x2_meas, are 1-dim arrays
meas_data = np.vstack((t_meas,x1_meas,x2_meas)).transpose()
```

Following the same approach as in the simulation case, the estimation is performed by invoking the estimation method on the model object.

```
#Invoke the estimation for the parameters k1 and k2
res_est = model.estimate(parameters=['k1','k2'], measurements=(['
    x1','x2'],meas_data))
```

Here, the parameters of interests, $k1$ *and* $k2$*, are specified together with the measurement data. The estimation is performed, by default, with SciPy's Nelder-Mead routine. The resulting parameters are returned in the* `res` *object.*

The parameter estimation is coupled to SciPy's optimization routine and the default algorithm used is the Nelder-Mead method [19]. The method is a derivative free method. The parameter estimation is available for all FMU types using the same syntax as shown in the above example.

5. Implementation overview

The core of PyFMI is implemented in Cython [9] which is a static compiler for Python. It allows to mix the programming languages C and Python interchangeably. The added benefit of mixing the languages is that the main part of the package, where readability and scripting functionality matter, is based on Python and performance critical parts are kept in C. In this way computational performance is preserved as opposed to relying solely on Python. Not only does

Cython allow to mix the languages, it also allow to connect to external C code which is imperative due to the dependency on the FMI Library [2].

As shown in Example 4.2, the high-level methods commonly uses the names of the variables instead of the value references which is an identifier for a variable, used by the FMI interface. Using the names results in a convenient way for working with variables although it introduces an overhead, cf. Figure 8.

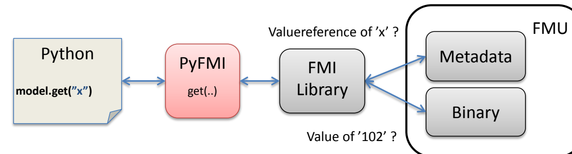

Figure 8: Overview of the functionality of the high-level method *get*. The variable name for which the value is requested is sent to PyFMI. The variable name is translated into a value reference by help of the metadata using FMI Library. Using the value reference, the value is retrieved from the binary, also through FMI Library, and passed to the user.

The methods in the specification are connected via a high-level interface as well as access to the metadata. For specific use cases, direct access to the low-level methods are necessary and they have additionally been made available.

The algorithms implemented in the master algorithm are all based on a Jacobi-like scheme where the individual models perform a global time-step and then exchange information. A global time-step can be performed simultaneously for all models and thus also be straightforwardly parallelized. In PyFMI this is implemented, cf. Figure 9.

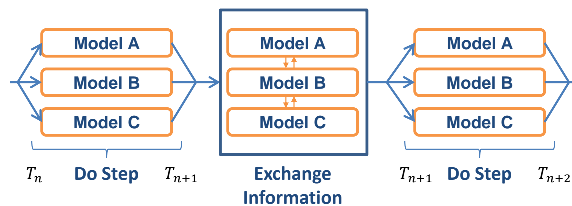

Figure 9: Performing the global time steps in parallel when simulating a coupled system using the implemented master algorithm. The exchange of information is done in serial.

5.1. Architecture

In PyFMI, each version and type of model defined in the standard is represented by its own Cython class, cf. Figure 10. The versions and model types all contain their specific functionality and extensions. However, much of the functionality between them are common motivating the structure.

A simulation, as previously shown, is performed by invoking the simulate method on the model object.

```
#Definition of the simulate method
```

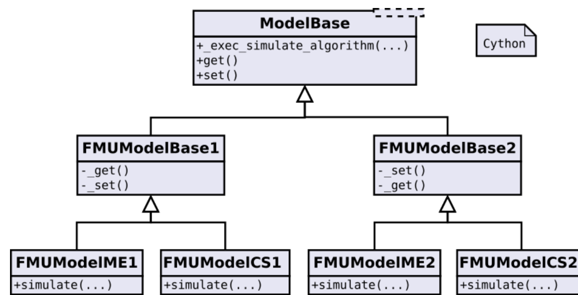

Figure 10: Overview of the class diagram for the classes holding the FMUs.

```
model.simulate( start_time='Default', final_time='Default',
                input=(), algorithm='AssimuloFMIAlg', options={}
                    )
```

The method allows a number of arguments such as defining the start and final time of the simulation, inputs and specifying an algorithm. There are two algorithms available in PyFMI, one with the coupling to Assimulo in order to gain access to solvers and one for simulation of co-simulation FMUs. Additionally, user defined algorithms may be used. In Figure 11, the relation between the model objects and the algorithms together with algorithmic options are shown.

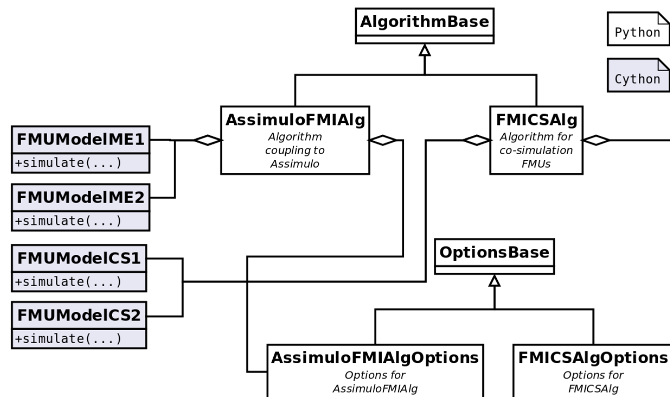

Figure 11: Overview of the class diagram and the coupling between model objects and algorithms for simulation together with the algorithms options.

A simulation of a coupled system is different from simulating a single system. This is due to that the coupled system needs to be defined. Specifically the coupling between the models needs to be specified. An algorithm for simulating a coupled system is usually called a master algorithm and here the implementation is contained in a `Master` class. In Figure 12, the relations between the

classes are shown.

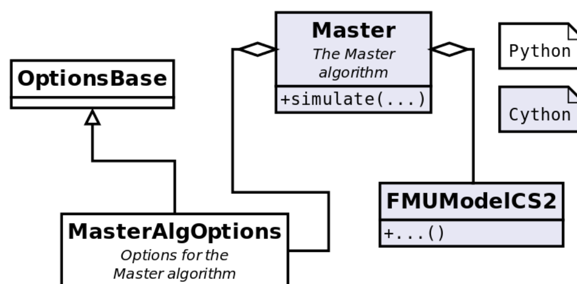

Figure 12: Coupling between the FMU model class, options and the master algorithm.

The interface for the parameter estimation follow that of the simulate method where the method is invoked on a model object.

```
#Definition of the estimate method
model.estimate( parameters, measurements,
    input=(), algorithm='SciEstAlg', options={} )
```

The `parameters` are the parameters of interest to tune while the `measurements` is the experimental data. Additionally, inputs can be set. In Figure 13, the relation between the model objects and the algorithm for parameter estimation is shown together with algorithmic options.

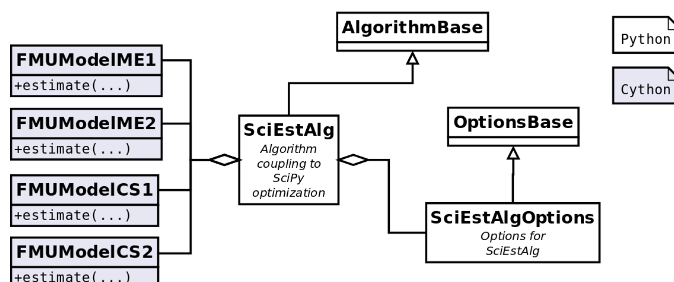

Figure 13: Overview of the class diagram and the coupling between model objects and algorithms for parameter estimations together with the algorithms options.

5.2. Result handling

Within the package, simulation results are handled through a base class, `ResultHandler`, that determines the interface for the underlying specific storage types, cf. Figure 14. The possible options are to store the result to a specific file format supported by for instance the Modelica tool, Dymola [20], store the result in a CSV file or store the result directly in memory. Additionally, a custom result handler can be provided to the simulation so that the result is handled in a user defined way.

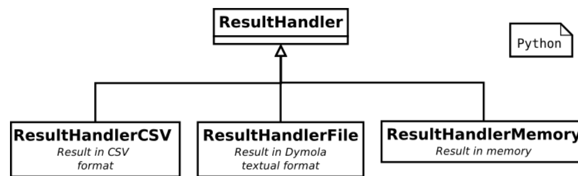

Figure 14: Overview of the class diagram for the classes storing the result.

The simulation result is returned to the user after a successful simulation in a general format, independent on how the result was actually stored. In Example 5.1, accessing the simulation results are shown.

Example 5.1 (Result handling)**.** *Invoking the simulate method on a model object result in that the computed simulation result is returned.*

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

Trajectories for specific variables are easily retrieved by operations on the result object.

```
res["x"] #Result trajectory the variable x
res["time"] #Result trajectory for the time
```

In case of a simulation of a coupled system, the result is returned as above.

```
res = master_simulator.simulate()
```

However, accessing the individual variable trajectories, both the model from which the variable is defined and the variable itself is needed.

```
res[sub_system1]["x"] #Result trajectory the variable x
                      #from the model object "sub_system1"
res[sub_system1]["time"] #Result trajectory for the time
                         #from the model object "sub_system1"
```

Visualization of the trajectories can easily by done using the matplotlib [10] package.

For large industrial models, the stored result can easily be gigabytes of data and the data handling can have a significant impact on the simulation performance. Coupling the result handling with the filter option in Table 1,Table 2 and Table 3, i.e. storing only the variables of interest, reduces both.

6. Case studies

6.1. Simulation of a woodpecker

This example is intended to show how PyFMI can handle hybrid systems, as model exchange FMUs from different sources, illustrated by a toy woodpecker, [21]. The model consists of a vertical bar attached to the ground, a sleeve able

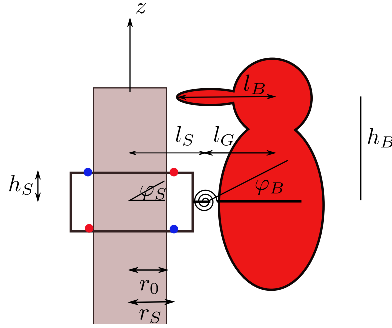

Figure 15: Schematic figure of the woodpecker.

to slide along the bar and the woodpecker which is attached to the sleeve via a spring, cf. Figure 15. Impact is modeled without friction for simplicity. In [11], the woodpecker was defined in Python and simulated using Assimulo. Here, the model is modeled in Modelica and exported as model exchange FMUs from Dymola [20] and JModelica [6]. The Modelica code is shown in Appendix B.

The woodpecker is loaded into PyFMI and simulated with the solver CVode [22] connected through Assimulo with absolute and relative tolerance set to 10^{-6}.

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

#Get the options
opts = model.simulate_options()

#Specify tolerances
opts["CVode_options"]["atol"] = 1e-6
opts["CVode_options"]["rtol"] = 1e-6

#Simulate
res = model.simulate(final_time=tf, options=opts)
```

This was performed for the FMUs from the different tools. In Figure 16 the simulation results are shown for the Dymola FMU. In Figure 17, a comparison is made between an FMU generated from JModelica and Dymola, simulated using CVode and tolerances set to 10^{-6}. The reference used was computed using the JModelica generated FMU with the Radau5 [23] solver connected through Assimulo together with absolute and relative tolerance set to 10^{-10}.

6.2. Co-simulation of a quarter car

In this example, a quarter car, cf. Figure 18, is simulated with step size control. In a co-simulation setup, this example was discussed in [17] and the intention with the example is to show that PyFMI are able to replicate the results shown in that article. The quarter car is governed by the equations,

$$m_c\ddot{x}_c = k_c(x_w - x_c) + d_c(\dot{x}_w - \dot{x}_c) \tag{18a}$$

$$m_w\ddot{x}_w = k_w(0.1 - x_w) + k_c(x_w - x_c) + d_c(\dot{x}_w - \dot{x}_c) \tag{18b}$$

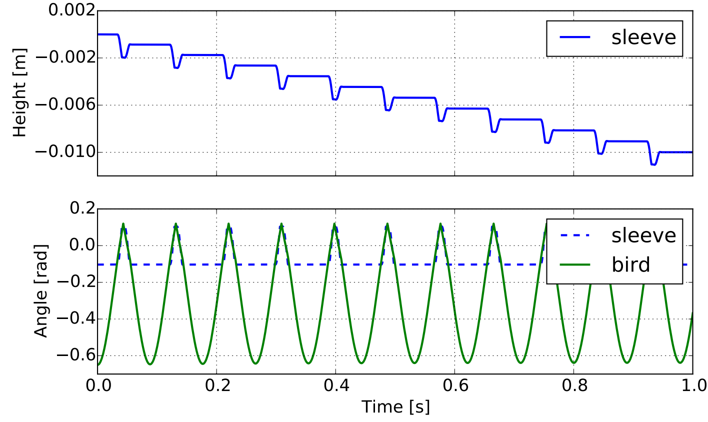

Figure 16: The height of the sleeve and the angle of both the sleeve and the bird of the woodpecker from Section 6.1.

with the constants, $m_w = 40kg$, $m_c = 400kg$, $k_w = 150000N/m$, $k_c = 15000N/m$ and $d_c = 1000Ns/m$.

The system is decoupled with the chassis being one sub-system and the wheel another. The coupling is given by

$$y = I \begin{bmatrix} x_c \\ \dot{x}_c \\ x_w \\ \dot{x}_w \end{bmatrix} \tag{19a}$$

$$u = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} y \tag{19b}$$

where there is no direct feed-through.

FMUs of the subsystems were generated using Dymola [20] as co-simulation FMUs with support for saving the internal state and setting the internal state which allows for re-computation of a global step.

Using the implemented master algorithm, cf. Section 4.3, to simulate the coupled system the algorithm itself needs to be imported together with methods for loading the FMU into Python.

```
from pyfmi import load_fmu
from pyfmi.master import Master
```

The FMUs are then loaded into Python.

```
#Load the FMUs
model_wheel  = load_fmu(fmu_wheel)
model_chassi = load_fmu(fmu_chassi)
```

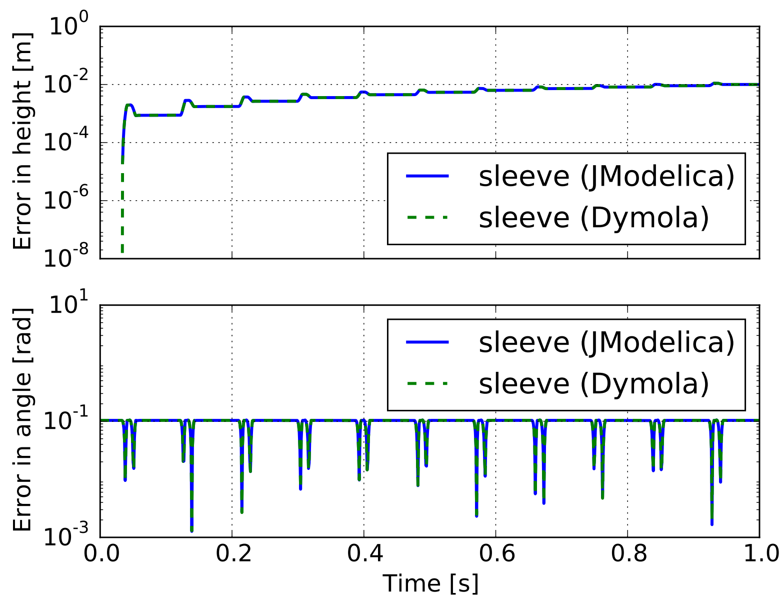

Figure 17: Comparison between a JModelica.org and a Dymola generated FMU of the toy woodpecker in Section 6.1.

The coupling is specified by a connection matrix where the first object specifies the model from where the output should be retrieved from. The second part specifies to what sub-system the values should be provided and to which variable.

```
#Specify the coupling
connections = [(model_chassi,"x_chassi",model_wheel,"x_chassi"),
               (model_chassi,"v_chassi",model_wheel,"v_chassi"),
               (model_wheel,"x_wheel",model_chassi,"x_wheel"),
               (model_wheel,"v_wheel",model_chassi,"v_wheel")]
```

The next step is to load the master algorithm with the models and the couplings.

```
models = [model_chassi, model_wheel]

#Load the models into the master algorithm
master_simulator = Master(models, connections)
```

Specifying the options is done through the options dictionary.

```
opts = master_simulator.simulate_options()

#(0 = Constant, 1 = Linear)
master_opts["extrapolation_order"] = 0
master_opts["error_controlled"] = True
master_opts["rtol"] = 1e-4
master_opts["atol"] = 1e-4
```

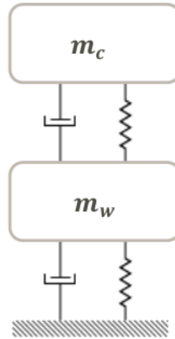

Figure 18: The quarter car model from Section 6.2.

The use of Richardson for the error estimation is specified as well as both the absolute and the relative tolerance. The tolerances was set to 10^{-4}. Finally the coupled system can be simulated using the `simulate` method.

```
#Simulate the coupled system
res = master_simulator.simulate(final_time=1)
```

In Figure 19 the result is shown for both the position and the velocity. The figures also show the reference trajectory which was calculated by simulating the monolithic system using the solver CVode with a tolerance of 10^{-12}. The monolithic system was exported as an model exchange FMU using JModelica.org [6] and simulated using PyFMI together with Assimulo. In Figure 20 the estimated error is shown together with the global step size and the time points where a step rejection occurred.

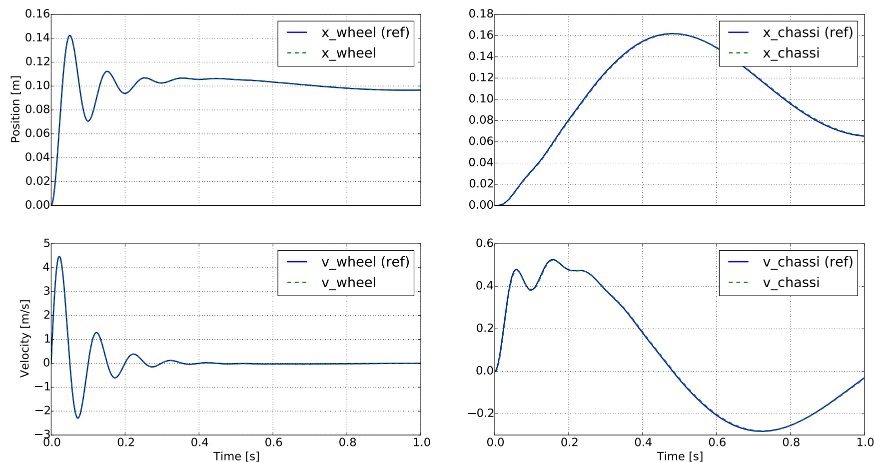

Figure 19: The velocity and the position of the quarter car from Section 6.2 simulated using constant extrapolation and a step size of 0.001 together with the reference solution.

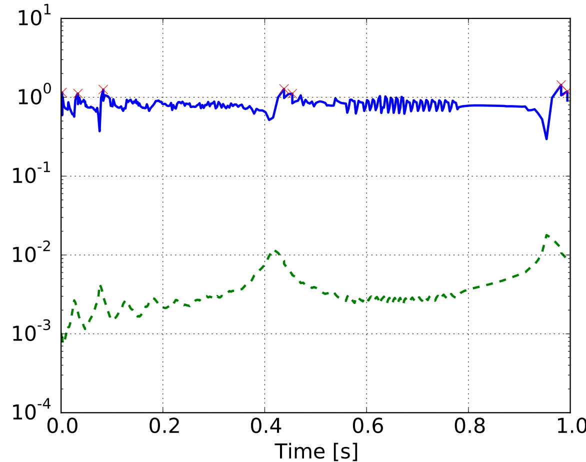

Figure 20: The normalized estimated error (solid) together with the global step size (dashed) and the time points for where a step was rejected (cross) when simulating the quarter car from Section 6.2. The simulation was carried out using constant extrapolation together with a relative and an absolute tolerance of 10^{-4}.

Simulations using higher order extrapolation was additionally carried out to investigate the influence of the extrapolation order on the number of steps. In Table 4, simulation statistics is shown for when using various order on the extrapolation. As can be seen from the table, using a higher order extrapolation polynomial results in a decrease of the number of steps.

Table 4: Simulation statistics for when simulating the Quarter Car in Section 6.2 using various order on the extrapolation. The simulation was performed using the parallel approach together with variable step size and an absolute and a relative tolerance of 10^{-4}.

Extrapolation order	0	1	2
Number of global steps	300	92	71
Number of error test failures	4	1	5

The example show that we are able to reproduce the results in [17] using the developed tools.

6.3. Parameter estimation in a quadruple tank

In this example, the parameter estimation capabilities within PyFMI is demonstrated on a quadruple tank model [24]. The example is inspired by the tank example in JModelica.org. The model consists of four coupled tanks, stacked two by two, and coupled so that the third tank deposits water into the first and the fourth tank deposits water into the second. The amount of water deposited is dependent on the size of the tube, connecting the tanks. Furthermore, tank one and two also have runoff dependent on the size of a tube, although they are not connected any other tank. The input to the model are voltages, controlling two pumps which pumps water into the system. Pump one

pumps water into the first and fourth tank, while the second pump pumps water into the second and third, cf Figure 21. The goal of the parameter estimation

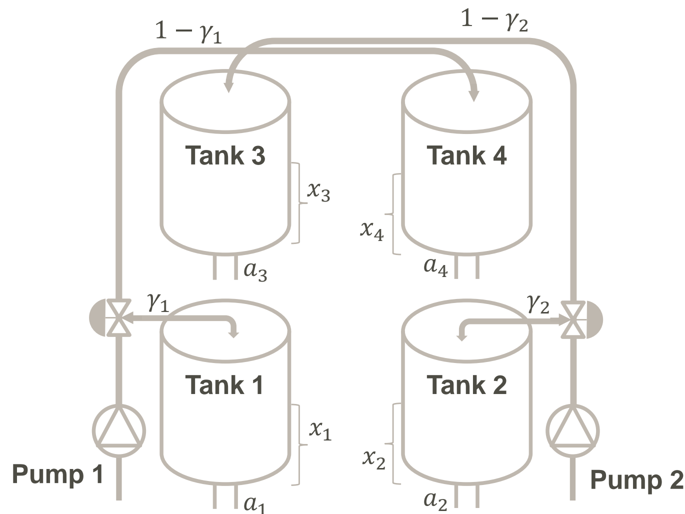

Figure 21: Visualization of the quadruple tank in Section 6.3. The tanks all have runoff determined by a_{1-4} and the top tank deposits water into the bottom tanks. Water enters the tanks and is controlled via pumps one and two. The water level in the tanks is the variables, x_{1-4}.

is to estimate the size of the tubs for the water runoff. The quadruple tank was modeled in Modelica and given in Appendix C, and compiled into an FMU using JModelica.org.

The input trajectories and measurement used in this example was recorded on a experimental setup of the tank system[2]. In Figure 22, the input voltages are shown. Furthermore, an initial estimate for the parameters controlling the runoff ($a_{[1-4]}$) are shown in Table 5.

Table 5: Initial parameters in Section 6.3.

$a_1 = 0.03$ cm^2	$a_2 = 0.03$ cm^2
$a_3 = 0.03$ cm^2	$a_4 = 0.03$ cm^2

Now, as a first step, the data needs to be imported into Python. The data is stored in a Matlab format and using SciPy, this can be read.

```
from scipy.io.matlab.mio import loadmat
data = loadmat('quadtank_measurements')
```

The measurement and input signals are extracted from the loaded data.

[2]The data was recorded at the Department of Automatic Control, Lund, Sweden by Kristian Soltesz

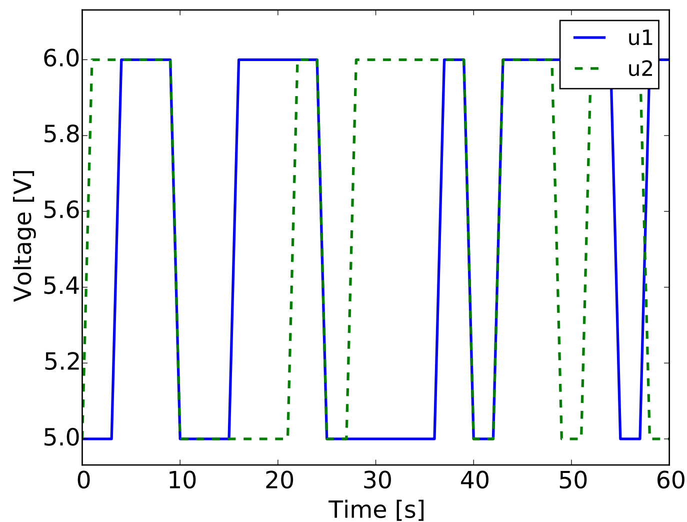

Figure 22: Input signals in Section 6.3

```
#Time vector
t_meas = data['t'][6000::100,0]-60
#Tank levels
x1_meas = data['y1_f'][6000::100,0]/100
x2_meas = data['y2_f'][6000::100,0]/100
x3_meas = data['y3_d'][6000::100,0]/100
x4_meas = data['y4_d'][6000::100,0]/100
#Input signals
u1 = data['u1_d'][6000::100,0]
u2 = data['u2_d'][6000::100,0]
```

With the loaded input signals and the initial parameter values, a simulation is performed as below.

```
model = load_fmu("Quadtank.fmu") #Load the FMU

# Create the input matrix
u = N.transpose(N.vstack((t_meas,u1,u2)))

# Simulate the model response, given the initial parameters
res = model.simulate(final_time=60, input=(['u1','u2'],u))
```

The model response, for the simulation, is shown in Figure 23. As seen in the figure, there is a discrepancy between the simulated response and the measurement. By performing the parameter estimation, the hope is that this discrepancy will be decreased.

Performing the parameter estimation requires that the interested parameters are specified, here $a_{[1-4]}$. Furthermore, which variables that have measurements need to be specified together with the measurement data. As in the simulation case, the inputs need also be provided.

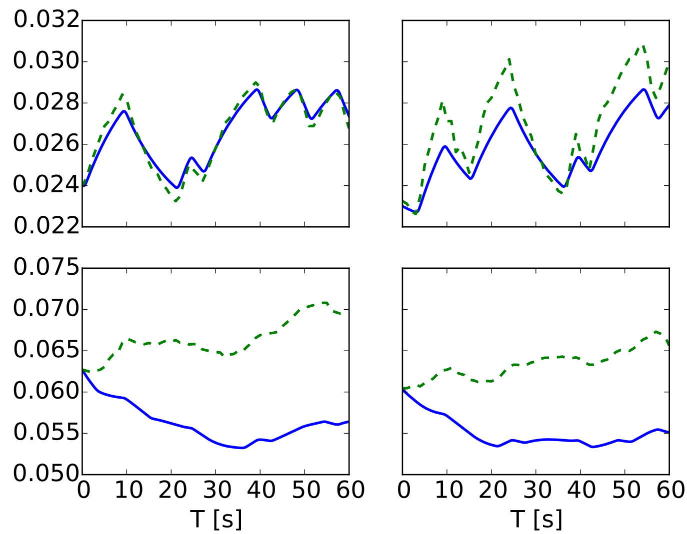

Figure 23: A comparison of the measured data (dashed) together with the simulated response (solid) given the initial values of the parameters. The trajectories are the tank levels. The left top figure represents the third tank and the top right, the fourth tank. The left bottom figure is the first tank while the right figure is the second.

```
meas_data = N.vstack((t_meas,N.vstack((y1_meas,y2_meas,y3_meas,
    y4_meas)))).transpose()
res_est = model.estimate(parameters=['a1','a2','a3','a4'],
                         measurements=(['x1','x2','x3','x4'],
                             meas_data),
                         input=(['u1','u2'],u))
```

Using the default algorithm, the call to the estimate method will invoke the Nelder-Mead algorithm [19], which is a derivative free optimization method, included in SciPy. The returned object contains the estimated parameters which are shown in Table 6. In order to verify the model response, the estimated parameter values are set to the model and the model is simulated once more.

```
model = load_fmu("Quadtank.fmu") #Load the FMU

# Setting the estimated parameter values into the model
model.set(['a1','a2','a3','a4'],
          [res_est["a1"], res_est["a2"],res_est["a3"],res_est["a4
              "]])

# Simulate the model response, given the estimated parameters
res = model.simulate(final_time=60, input=(['u1','u2'],u))
```

The simulated response, given the estimated parameter values, are shown in Figure 24. As seen in the figure, using the estimated parameters, the model response has substantially been improved when compared to the simulation with the initial parameter values, Figure 23.

Table 6: Estimated parameters in Section 6.3.

$a_1 = 0.02660115$ cm^2	$a_2 = 0.0270179$ cm^2
$a_3 = 0.03008687$ cm^2	$a_4 = 0.02929907$ cm^2

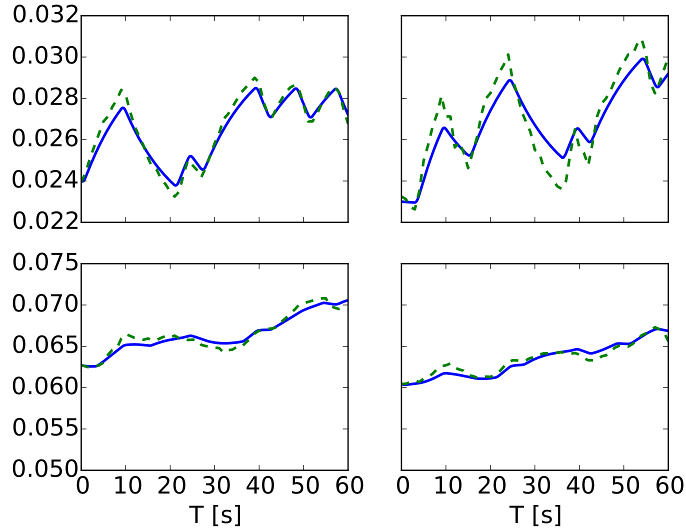

Figure 24: A comparison of the measured data (dashed) together with the simulated response (solid) given the estimated values of the parameters. The trajectories are the tank levels. The left top figure represents the third tank and the top right, the fourth tank. The left bottom figure is the first tank while the right figure is the second.

6.4. Parallel co-simulation of a race car

In this example, a race car is modeled in Modelica using the commercial Vehicle Dynamics Library [25]. In the example, the car is driven by a virtual driver that tries to stay onto an eight shaped course with increasing velocity in order to investigate the dynamic response of the car, especially when changing the turning direction. The model is simulated as a coupled system in a co-simulation setup where the model has been separated into wheels and chassis, Figure 25. The intention of the example is to highlight the parallelization feature in the implemented master algorithm. The model of the chassis was compiled into a co-simulation FMU using Dymola [20] while the model of a wheel was exported using JModelica.org [6]. The models contain about 90k parameters, constants and variables in total.

An increase in performance using the parallelization can only be expected if the majority of the simulation time is not spent in a single model. In this example, more time is spent in the simulation of the chassis than for a wheel, cf. Table 7. However, when considering the total simulation time and the chassis part of it, a speedup is expected when using the parallelization.

In order to specify that global steps, in the master algorithm, should be performed in parallel, the options need to be set.

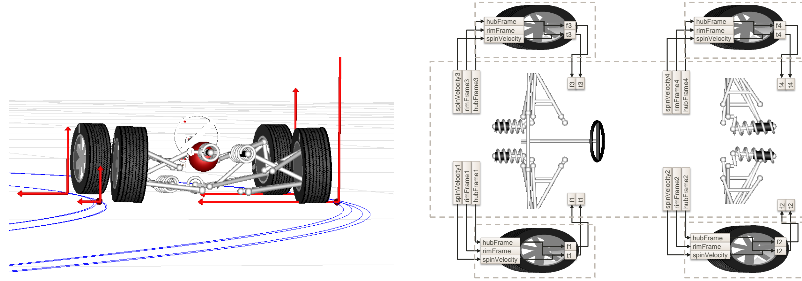

Figure 25: Visualization of the race car (left) and visualization of the couplings in the race car from Section 6.4 in a co-simulation setup where the wheel and chassis has been divided into separate models (right). © Modelon.

Table 7: Normalized elapsed time, for each model, for a simulation of the race car from Section 6.4. The overhead is time not spent in the separate models, storing the results for example.

Total	Chassis	Wheels (each)	Overhead
1.0	0.31	0.16	0.05

```
# Retrieve the simulation options
master_options = master_simulator.simulate_options()

master_options["execution"] = "parallel"
master_options["num_threads"] = 1
```

Furthermore, due to the amount of variables and parameters in the models, the filter is set so that only the interesting variables are stored.

```
master_options["filter"] = {model_chassi:"*summary*" ,
                    model_wheel_lf: "forces.f_*",
                    model_wheel_lb: "forces.f_*",
                    model_wheel_rf: "forces.f_*",
                    model_wheel_rb: "forces.f_*"}
```

The test was run on laptop with two cores. Using the two cores, the simulation time was reduced by 34%. While this is not optimal, this is still a substantial decrease of the simulation time.

The full Python script can be found in Appendix D.

6.5. Sparsity exploitation in a chromatography separation process

In [3], the robustness of a high-pressure liquid chromatographic process (Figure 26) was investigated. Given a nominal input trajectory, the aim was to quantify the robustness of the process with regards to disturbances in the input. The process is described by an ODE with a scalar input,

$$\dot{x} = f(x, u), \quad x \in \mathbb{R}^{142}, u \in \mathbb{R}. \tag{20}$$

An FMU for the process was generated from Dymola 2016.

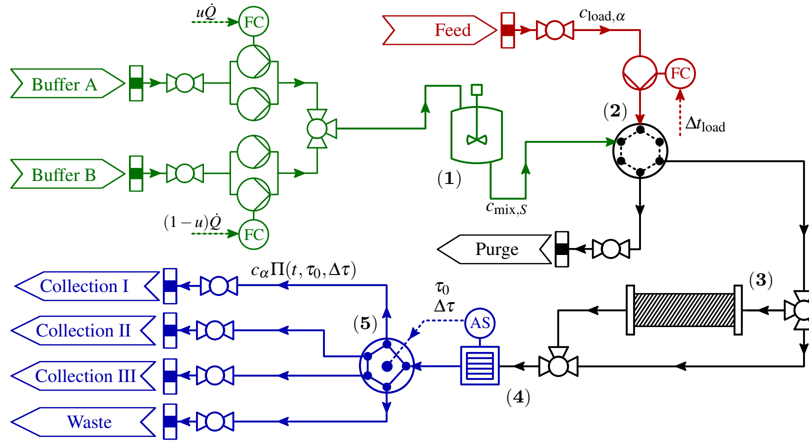

Figure 26: Visualization of the chromatography separation process used in Section 6.5. The input, u, controls how much of buffer A and B enters the process through the mixing tank (1). The feed enters the system through (2), where also the buffers passes. The separation takes place in the separation column (3). The output from the process is collected in either I, II, III or dumped as waste. The collection is determined using the detector (4) and the valve (5).

In order to quantify robustness towards disturbances, a Lyapunov equation needs to be solved,

$$\dot{P} = AP + PA^T + BB^T, \quad P(t_0) = B(t_0)B^T(t_0) \tag{21}$$

where, $A = \frac{\partial f}{\partial x}$ and $B = \frac{\partial f}{\partial u}$. The primary focus of this section is to demonstrate how sparsity information in FMUs can be used to significantly decrease simulation times. For a full problem statement and results, cf. [3]. In Figure 27, the structure of A is shown. As Equation 21 is matrix valued, we first vectorize the equation which results in,

$$\text{vec}(\dot{P}) = (I \otimes A)\text{vec}(P) + (A \otimes I)\text{vec}(P) + \text{vec}(B^T B), \tag{22}$$

where $\otimes$ is the Kronecker product. The full system can then be formed as,

$$\begin{bmatrix} \dot{x} \\ \text{vec}(\dot{P}) \end{bmatrix} = \begin{bmatrix} f(t,x,u) \\ (I \otimes A)\text{vec}(P) + (A \otimes I)\text{vec}(P) + \text{vec}(B^T B) \end{bmatrix}. \tag{23}$$

Furthermore, the process model results in a stiff problem which requires an implicit solver. Thus, we need the Jacobian of Equation 23 which is defined as,

$$J = \begin{bmatrix} A & 0 \\ 0 & I \otimes A + A \otimes I \end{bmatrix}. \tag{24}$$

In Figure 28, the structure of the Jacobian is shown.

In order to solve the augmented system, Equation 23, PyFMI was extended to be able to add equations that are solved together with the FMU. Furthermore,

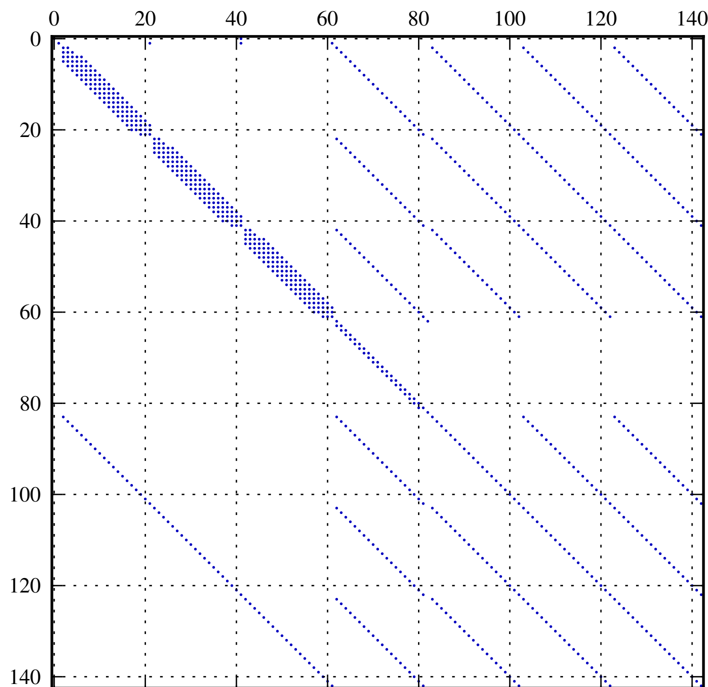

Figure 27: The structure of the matrix A from Section 6.5.

in order to efficiently solve the above problem, the structure of A needs to be taking into account. By using the compression discussed in Section 4.1, the number of calls to the directional derivatives was reduced by 90%. Furthermore, with the connection to SuperLU [26] from CVode, the Jacobian can be provided as a sparse matrix.

By using both the compression for computing A and by providing the Jacobian (Equation 24) as a sparse matrix, the simulation time was reduced to only 4% of the original time where a dense representation of the Jacobian was used and no compression.

In Appendix E, the full Python script for adding the Lyapunov equations to a simulation of an FMU is shown.

7. Conclusions

In this article, we presented PyFMI, a software for working with models following the Functional Mock-up Interface. The package support models following version 1.0 and 2.0 of the standard as well as the different model types, model exchange and co-simulation. Interactions with the models are conveniently per-

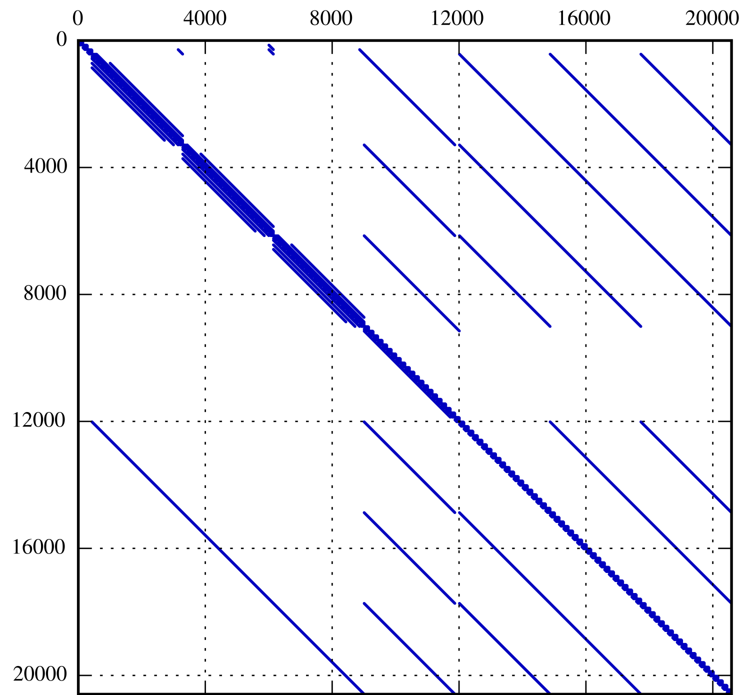

Figure 28: The structure of the Jacobian for the augmented system, Equation 24, used in Section 6.5.

formed using high-level methods and if needed, access to the low-level methods is additionally available.

With a connection to the simulation package Assimulo, simulation of model exchange FMUs can be performed using state of the art integrators. For coupled systems, PyFMI implements a master algorithm for simulation of coupled co-simulation FMUs. Furthermore, the simulation analyses are complemented with support for parameter estimation. Having these analyses easily available in an open tool, we hope that the standard will continue to grow and spread even further.

The package is demonstrated on a number of problems and show promising results. PyFMI[3] is freely available under the LGPL [27] license.

[3] http://www.pyfmi.org [accessed 2016-03-24]

Acknowledgements

This work was supported in part by the Lund Center for Control of Complex Engineering Systems (LCCC), funded by the Swedish Research Council, and is gratefully acknowledged.

Appendix A. The van der Pol oscillator - Modelica

Modelica code representing the van der Pol oscillator used in the examples in Section 4.

```
model VDP
    // Parameters
    parameter Real mu = 2e1;

    // The states
    Real x1(start=-0.6);
    Real x2(start=2);

    // The control signal
    input Real u;

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

Appendix B. Woodpecker - Modelica

Modelica code representing the toy woodpecker from Section 6.1 is shown below.

```
model Woody
  //Constants
  constant Real g = 9.81;
  //Parameters
  parameter Real mS = 3.0e-4, mB = 4.5e-3;
  parameter Real r0 = 2.5e-3, rS = 3.1e-3;
  parameter Real JS = 5.0e-9, JB = 7.0e-7;
  parameter Real hS = 5.8e-3, hB = 2.0e-2;
  parameter Real lS = 1.0e-2, lB = 2.01e-2;

  parameter Real masstotal = mS + mB;
  parameter Real rM = rS - r0;
  parameter Real cp = 5.6e-3, lG = 1.5e-2;

  //Continuous variables
  Real z(start = 0.0),         zp(start = 0.0);
  Real phiS(start = -0.10344), phiSp(start = 0.0);
  Real phiB(start = -0.65),    phiBp(start = 0.0);
  Real phiBpp(start = 1.40059e2);
  Real lam1(start = -0.6911), lam2(start = -0.1416);
  Integer state(start = 2, fixed = true);
  discrete Real last_update(start=0);
```

```
equation
  der(z) = zp;
  der(phiS) = phiSp;
  der(phiB) = phiBp;
  phiBpp = der(phiBp);
  if state == 1 then
    masstotal * der(zp) + mB * lS * der(phiSp) + mB * lG * der(
        phiBp) + masstotal * g = 0;
    mB * lS * der(zp) + (JS + mB * lS * lS) * der(phiSp) + mB *
        lS * lG * der(phiBp) - cp * (phiB - phiS) + mB * lS * g =
         -lam1;
    mB * lG * der(zp) + mB * lS * lG * der(phiSp) + (JB + mB * lG
         * lG) * der(phiBp) - cp * (phiS - phiB) + mB * lG * g =
        -lam2;
    lam1 = 0;
    lam2 = 0;
  elseif state == 2 then
    masstotal * der(zp) + mB * lS * der(phiSp) + mB * lG * der(
        phiBp) + masstotal * g = -lam2;
    mB * lS * der(zp) + (JS + mB * lS * lS) * der(phiSp) + mB *
        lS * lG * der(phiBp) - cp * (phiB - phiS) + mB * lS * g =
         (-hS * lam1) - rS * lam2;
    mB * lG * der(zp) + mB * lS * lG * der(phiSp) + (JB + mB * lG
         * lG) * der(phiBp) - cp * (phiS - phiB) + mB * lG * g =
        0;
    //Index 3
    //0 = (rS-r0)+hS*phiS;
    //0 = der(z)+rS*phiSp;
    //Index 1
    0 = hS * der(phiSp);
    0 = der(zp) + rS * der(phiSp);
  else
    masstotal * der(zp) + mB * lS * der(phiSp) + mB * lG * der(
        phiBp) + masstotal * g = -lam2;
    mB * lS * der(zp) + (JS + mB * lS * lS) * der(phiSp) + mB *
        lS * lG * der(phiBp) - cp * (phiB - phiS) + mB * lS * g =
         hS * lam1 - rS * lam2;
    mB * lG * der(zp) + mB * lS * lG * der(phiSp) + (JB + mB * lG
         * lG) * der(phiBp) - cp * (phiS - phiB) + mB * lG * g =
        0;
    //Index 3
    //0 = (rS-r0)-hS*phiS;
    //0 = der(z)+rS*phiSp;
    //Index 1
    0 = -hS * der(phiSp);
    0 = der(zp) + rS * der(phiSp);
  end if;
algorithm
  when {rM + hS * phiS < 0.0, rM - hS * phiS < 0.0} then
    if state == 1 and phiBp < 0 then
      state := 2;
      last_update := time;
    end if;
    if state == 1 and phiBp > 0 then
      state := 3;
      last_update := time;
```

```
    end if;
  elsewhen {lam1 > 1e-8, lam1 < -1e-8} then
    if state == 2 and time - last_update > 0 then
        last_update := time;
      state := 1;
    end if;
    if state == 3 and time - last_update > 0 then
      state := 1;
      last_update := time;
    end if;
  end when;
equation
  when {hB * phiB - (lS + lG - lB - r0) > 0 and phiBp > 0} then
    reinit(phiBp, -pre(phiBp));
  elsewhen state == 2 then
    reinit(phiBp, (mB * lG * pre(zp) + mB * lS * lG * pre(phiSp)
        + (JB + mB * lG * lG) * pre(phiBp)) / (JB + mB * lG * lG)
        );
    reinit(phiSp, 0.0);
    reinit(zp, 0.0);
  elsewhen state == 3 then
    reinit(phiBp, (mB * lG * pre(zp) + mB * lS * lG * pre(phiSp)
        + (JB + mB * lG * lG) * pre(phiBp)) / (JB + mB * lG * lG)
        );
    reinit(phiSp, 0.0);
    reinit(zp, 0.0);
  end when;
end Woody;
```

Appendix C. Quadruple tank - Modelica

Modelica code representing the quadruple tank model from Section 6.3 is shown below. Courtesy of JModelica.org.

```
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(min=1e-6,nominal=1e-6)=0.03
    e-4, a2(nominal=1e-6)=0.03e-4;
parameter Modelica.SIunits.Area a3(nominal=1e-6)=0.03e-4, a4(
    nominal=1e-6)=0.03e-4;
parameter Modelica.SIunits.Acceleration g=9.81;
parameter Real k1_nmp(unit="m^3/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;

// Tank levels
Modelica.SIunits.Length x1(start=0.0627);
Modelica.SIunits.Length x2(start=0.06044);
Modelica.SIunits.Length x3(start=0.024);
Modelica.SIunits.Length x4(start=0.023);

// 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;
```

Appendix D. Race car - Python

The full Python script used in Section 6.4 is shown below.

```python
from pyfmi import load_fmu
from pyfmi.master import Master

#Load the corresponding FMUs
model_chassi   = load_fmu("Chassis.fmu")
model_wheel_lf = load_fmu("TyreForcesSlick.fmu")
model_wheel_lb = load_fmu("TyreForcesSlick.fmu")
model_wheel_rf = load_fmu("TyreForcesSlick.fmu")
model_wheel_rb = load_fmu("TyreForcesSlick.fmu")

for model_wheel in [model_wheel_lf,model_wheel_lb,model_wheel_rf,
     model_wheel_rb]:
    model_wheel.set("_cs_solver", 1) #Set the Explicit Euler solver
    model_wheel.set("_cs_step_size", 2e-6) #Set the step size

#Define a list of loaded FMUs
models = [model_chassi, model_wheel_lf, model_wheel_lb,
                        model_wheel_rf, model_wheel_rb]

#Specify the connections
connections = []
for i,wheel_number in enumerate([1,2,3,4]):
    connections.extend(
        [(models[i+1], out, model_chassi,
         out.replace("1","%d"%wheel_number,1))
         for out in models[i+1].get_output_list().keys()])
    for out in model_wheel_lf.get_input_list().keys():
        if out != "spinVelocity":
            connections.append(
    (model_chassi, out.replace(".", "%d."%wheel_number, 1),
     models[i+1], out))
        else:
            connections.append(
    (model_chassi, "spinVelocity%d"%wheel_number, models[i+1], out))

#Specify steering
model_chassi.set("steeringInEight.left_turn", 1)

#Create the Master simulator
master_simulator = Master(models, connections)

#Specify the simulation options
master_options = master_simulator.simulate_options()
master_options["step_size"] = step_size
master_options["execution"] = "parallel"
master_options["num_threads"] = 2 #Set the number of threads
master_options["filter"] = {model_chassi:"*summary*" ,
                model_wheel_lf: "forces.f_*",
                model_wheel_lb: "forces.f_*",
```

```
                model_wheel_rf: "forces.f_*",
                model_wheel_rb: "forces.f_*"}
master_options["initialize"] = True
master_options["block_initialization"] = True

#Simulate the coupled system
res = master_simulator.simulate(final_time=1.0, options = master_options)

#Retrieve the results
t = res[model_chassi]["time"]
stearing_wheel = res[model_chassi]["chassis.summary_p_sw"]
p_x = res[model_chassi]["chassis.summary_r_0[1]"]
p_y = res[model_chassi]["chassis.summary_r_0[2]"]
```

Appendix E. Chromatography separation process - Python

The Python script used for adding the Lyapunov equations in Section 6.5 to a simulation using PyFMI.

```
import numpy as np
import scipy.sparse as sp
from assimulo.problem import Explicit_Problem

class AppendedODEs(Explicit_Problem):

    def __init__(self, model):

        assert model.get_version() == "2.0" #Assert the FMI version is 2.0
        assert model.get_capability_flags()["providesDirectionalDerivatives
            "] == True #Assert directional derivatives are provided

        self._model = model
        self.setup()

        self._res = []
        self._res_C = []
        self._res_CPCT = []
        self._order = "F"
        self._sparse_representation = True

        self.f_nbr = self._nbr_states*self._nbr_states
        self.y0 = np.zeros(self.f_nbr)

        [derv_state_dep, derv_input_dep] = model.
            get_derivatives_dependencies()
        self.jac_nnz = 2*self._nbr_states*np.sum([len(derv_state_dep[key])
            for key in derv_state_dep.keys()])+self._nbr_states*self.
            _nbr_states

    def get_size(self):
        return self.f_nbr

    def setup(self):
        self._nbr_states = len(self._model.get_states_list())
        self._nbr_inputs = len(self._model.get_input_list())

    #User defined extra right-hand-side
    def rhs(self, P):
        #A = df/dx, B = df/du
        A,B,C,D = self._model.get_state_space_representation(C=False, D=
            False)
        A = A.toarray(order=self._order)
        B = B.toarray(order=self._order)
```

```
        P = P.reshape(self._nbr_states,self._nbr_states, order=self._order)

        #dP = A P + P A^T + B B^T
        dP = A.dot(P)+P.dot(A.transpose())+B.dot(B.transpose())

        return dP.flatten(order=self._order)

    def jac(self, P):

        [A,B,C,D] = self._model.get_state_space_representation(A=True, B=
             False, C=False, D=False)

        data = []
        row_ind = []
        col_ind = []

        Aco = A.tocoo()
        AjFull = [A[:,j] for j in range(self._nbr_states)]

        for i in range(self._nbr_states):
            data.extend(Aco.data)
            row_ind.extend(i*self._nbr_states+Aco.row)
            col_ind.extend(i*self._nbr_states+Aco.col)

            col_ind_i = range(i*self._nbr_states,(i+1)*self._nbr_states)

            for j,val in enumerate(AjFull[i].data):
                data.extend([val]*self._nbr_states)
                row_ind.extend(range(AjFull[i].indices[j]*self._nbr_states
                     ,(AjFull[i].indices[j]+1)*self._nbr_states))
                col_ind.extend(col_ind_i)

        PJac = sp.coo_matrix((data, (row_ind, col_ind)))

        return PJac

    #User defined handle result for the extra equations
    def handle_result(self, export, P):
        [A,B,C,D] = self._model.get_state_space_representation(A=False, B=
             False, C=True, D=False)
        C = C.toarray(order=self._order)

        P = P.reshape(self._nbr_states,self._nbr_states, order=self._order)
        self._res_CPCT.append(np.dot(np.dot(C,P),np.transpose(C)).flatten(
             order=self._order))
```

References

[1] MODELISAR, Functional Mock-up Interface, Version 2.0, Interface specification, MODELISAR (July 2014).

[2] Modelon AB, FMI Library, `http://www.jmodelica.org/FMILibrary`, [accessed 2016-02-03] (2014).

[3] A. Holmqvist, C. Andersson, F. Magnusson, J. Åkesson, Methods and tools for robust optimal control of batch chromatographic separation processes, Processes 3 (3) (2015) 568. `doi:10.3390/pr3030568`.

[4] C. Andersson, J. Andreasson, C. Führer, J. Åkesson, A workbench for multibody systems ode and dae solvers, in: 2nd Jnt. Int. Conf. Multibody Syst. Dyn., 2012.

[5] Xogeny, FMQ, http://www.xogeny.com/products/, [accessed: 2016-03-18] (2016).

[6] J. Åkesson, K.-E. Årzén, M. Gäfvert, T. Bergdahl, H. Tummescheit, Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem, Comput. Chem. Eng. 34 (11) (2010) 1737–1749.

[7] C. Andersson, J. Åkesson, C. Führer, M. Gäfvert, Import and export of functional mock-up units in jmodelica.org, in: Proc. 8th Int. Modelica Conf., LiU E-Press, 2011, pp. 329–338. doi:10.3384/ecp11063329.

[8] T. E. Oliphant, Python for scientific computing, IEEE Comput. Sci. Eng. 9 (3) (2007) 10–20. doi:10.1109/MCSE.2007.58.

[9] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, K. Smith, Cython: The best of both worlds, IEEE Comput. Sci. Eng. 13 (2) (2011) 31 –39. doi:10.1109/MCSE.2010.118.

[10] J. D. Hunter, Matplotlib: A 2d graphics environment, Comput. Sci. Eng. 9 (3) (2007) 90–95.

[11] C. Andersson, C. Führer, J. Åkesson, Assimulo: A unified framework for ODE solvers, Math. Comput. Simul. 116 (2015) 26 – 43. doi:10.1016/j.matcom.2015.04.007.

[12] A. Pfeiffer, M. Hellerer, S. Hartweg, M. Otter, M. Reiner, Pysimulator - a simulation and analysis environment in python with plugin infrastructure, in: Proc. 9th Int. Modelica Conf, LiU E-Press, 2012, pp. 523–536. doi:10.3384/ecp12076523.

[13] MODELISAR, Functional mock-up interface for model exchange, Interface specification, MODELISAR (January 2010).

[14] MODELISAR, Functional mock-up interface for co-simulation, Interface specification, MODELISAR (October 2010).

[15] A. R. Curtis, M. J. D. Powell, J. K. Reid, On the estimation of sparse jacobian matrices, J. Inst. Math. Appl 13 (1) (1974) 117–120.

[16] C. Andersson, Methods and tools for co-simulation of dynamic systems with the functional mock-up interface, Ph.D. thesis, Lund Uni., Lund, Sweden, manuscript in preparation (2016).

[17] T. Schierz, M. Arnold, C. Clauss, Co-simulation with communication step size control in an fmi compatible master algorithm, in: Proc. 9th Int. Modelica Conf, LiU E-Press, 2012, pp. 205–214. doi:10.3384/ecp12076205.

[18] C. Andersson, S. Gedda, J. Åkesson, S. Diehl, Derivative-free parameter optimization of functional mock-up units, in: Proc. 9th Int. Modelica Conf, LiU E-Press, 2012, pp. 819–828. doi:10.3384/ecp12076819.

[19] J. A. Nelder, R. Mead, A simplex method for function minimization, Comput. J. 7 (4) (1965) 308–313. doi:10.1093/comjnl/7.4.308.

[20] Dassault Systèmes, Dymola - multi-engineering modeling and simulation - version 2016, http://www.dymola.com/, [accessed: 2016-02-03] (2015).

[21] R. I. Leine, D. H. Van Campen, C. H. Glocker, Nonlinear Dynamics and Modeling of Various Wooden Toys with Impact and Friction, J. Vib. Control 9 (1-2) (2003) 25. doi:10.1177/107754603030741.

[22] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, C. S. Woodward, Sundials: Suite of nonlinear and differential/algebraic equation solvers, ACM Trans. Math. Softw. 31 (3) (2005) 363–396. doi:10.1145/1089014.1089020.

[23] E. Hairer, G. Wanner, Solving Ordinary Differential Equations. II. Stiff and Differential-Algebraic Problems, 2nd Edition, Springer Ser. Comput. Math., Springer-Verlag, Berlin, 1996.

[24] K. H. Johansson, The quadruple-tank process: a multivariable laboratory process with an adjustable zero, IEEE Trans. Contr. Syst. Technol. 8 (3) (2000) 456–465. doi:10.1109/87.845876.

[25] J. Andreasson, M. Gäfvert, The vehicledynamics library - overview and application, in: Proc. 5th Int. Modelica Conf., Modelica Association, 2006.

[26] J. W. Demmel, J. R. Gilbert, X. S. Li, An asynchronous parallel supernodal algorithm for sparse gaussian elimination, SIAM J. Matrix Analysis and Applications 20 (4) (1999) 915–952.

[27] Free Software Foundation, GNU Lesser General Public License, version 3, http://www.gnu.org/licenses/lgpl.html, [accessed: 2014-05-07] (2007).

Preprints in Mathematical Sciences 2016:2
ISSN 1403-9338

LUTFNA-5008-2016

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/