A simulation environment for coupled systems of discontinuous ODE:s

Teo Nilsson

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

Bachelor's thesis
ISSN: 1654-6229

September 6, 2013

LUND
UNIVERSITY

Modelon

Abstract

This thesis covers the implementation and usage of PyFMI 2.0, an enhancement of the already existing PyFMI, a Python based simulation environment for importing and solving discontinuous systems of ordinary differential equations with in- and outputs, so-called simulations of Functional Mock-up Units. In particular, PyFMI 2.0 uses the Functional Mock-up Interface, FMI, 2.0 for interacting with Functional Mock-up Units, FMU:s, for Model Exchange and Co-Simulation. A mathematical and intuitive approach to the interface is treated together with a comparison to the previous interface of version 1.0. By experiments, the thesis aims to evaluate the possible efficiency gain in the simulation-run due to directional derivatives used as Jacobians provided by the FMU version 2.0 compared to version 1.0, where Jacobians are computed by the numerical integrator. Finally, it is concluded that PyFMI 2.0 needs a more efficient algorithm to retrieve the Jacobians, otherwise the time-loss in the elapsed simulation time becomes significantly large.

Acknowledgements

I sincerely thank my supervisors, Christian Andersson for his continuous support combined with endless patience and professor Claus Führer for his wise suggestions and constructive criticism giving broader perspective to my thesis, both whom without this thesis would not be possible. My deepest appreciation extends to Johan Åkesson with colleagues and students at Modelon AB for sharing their expertise and helpfully endured my questions. I express my special gratitude to Anna-Maria Persson for invaluable inspiration and persistent encouragement during my studies in the mathematical subjects. Finally, to all wonderful and supporting people in my everyday life, you know who you are; Thank you!

Contents

Introduction

In mathematics, an equation with a variable t, an unknown function $y(t)$ and at least one derivative of this function is known as a *differential equation*. In contrast to an algebraic equation where the solution is a vector or scalar which satisfies the equation, a solution to a differential equation is a function $y(t)$ that satisfies the equation for all $t \in D_y$. From a theoretical point of view, two questions raise immediately, namely existence and uniqueness of the solutions. Given the existence of a solution one might consider the numerical point of view whether there exists a solution, or possible solutions, on closed form or one has to approximate a solution by numerical integration.

Differential equations often appear in models that try to describe time dependent processes in science. If the differential equations describe a relationship between physical objects one is tempted to believe that there exists at least one solution, but this argument is totally dependent of whether the equations are a correct description of reality or just an approximation [1, p.31].

Let us for example study a pendulum (Figure 1), that is a point mass m fastened in a point P with a mass-free line of length l. The angle from the equilibrium point is a continuous function of time, $y(t)$. It is possible to simplify by assuming that the sum of the potential energy and the kinetic energy stays fixed, that is ignoring all friction, making this an approximation of the real world as mentioned above.

At a fixed time-point t, the angle of deflection is $y(t)$ so the speed of the pendulum is $l \cdot y'(t)$ and the acceleration is $l \cdot y''(t)$. The force F making the pendulum swing is orthogonal to the line and given by $\sin(y(t)) = \frac{-F}{m \cdot g}$. By Newton's second law, force = mass $\cdot$ acceleration and letting $a = \frac{g}{l}$, $a > 0$, the differential equation can be rewritten as:

$$y''(t) + a \sin(y(t)) = 0 \tag{1}$$

New variables are introduced and the differential equation rewritten as an

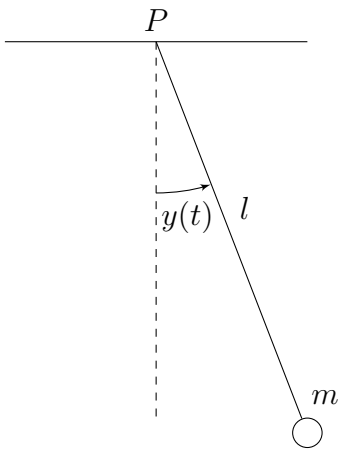

Figure 1: Pendulum

equivalent system of differential equations:

$$\begin{aligned} y_1'(t) &= y_2(t) \\ y_2'(t) &= -a\sin(y_1(t)) \end{aligned} \tag{2}$$

It is possible to guarantee a unique solution by adding a restriction mathematically known as initial conditions, which is interpreted as the initial position and speed of the pendulum [1, p.5].

The theory of differential equations together with the pendulum is treated more in Chapter 1 but before considering computing a solution to the problem, one would like to have a way to collect and represent all information about this, or a more general system of differential equations. In the system above, for example, it is preferable to distinguish between the function $y_1(t)$ that is the result or output, which another system might depend on, and the function $y_2(t)$ which is for internal use only. Among the constants, changing the parameter a is equal to changing the length of the line an hence the whole equation-system. While changing the initial conditions gives another set of solutions to an otherwise fixed system of differential equations. The Functional Mock-up Interface (FMI) defines a standard for representation of such systems of equations.

This thesis intends to describe and give an intuitive approach to the FMI standard, in particular it contains a comparison between FMI 1.0 and the new FMI 2.0 together with a mathematical description of the Functional Mock-up Unit (FMU), that is the package with differential equations. Furthermore, the thesis covers the implementation of PyFMI 2.0, a Python based software able to import and call the FMU:s allowing the user to connect multiple systems together to form a larger system of differential equations, connect the

system to a numerical integrator for computations and finally visualization of the solution. In the end, numerical experiments with PyFMI 2.0 are performed to evaluate possible improvements in the new FMI standard, primary due to the optionally provided Jacobians.

Reading guidance

Readers familiar with differential equations, in particular systems of ordinary differential equations are suggested to begin their reading at Chapter 2. If the reader understands the concept of FMI and posses knowledge of the differences and similarities of FMU:s for Model Exchange and Co-Simulation, then Chapter 2 is not necessary for the understanding of Chapter 3 and 4. Chapter 3 is recommended to look at for a more extensive understanding of the ideas behind the implementation of PyFMI 2.0, which is covered in Chapter 4.

Author's word

My intention behind the thesis was to apply knowledge of mathematics and numerical analysis to solve a problem raising from an industrial perspective but still maintaining a theoretical view of the task. With this in mind, the thesis has been carried out as a collaboration between Lund University and Modelon AB. The final product, PyFMI 2.0, is based on Python, an easy to read script language suitable for scientific programming and a favourite of my own. Curiousness about mathematical modelling as a mixture of mathematics and numerical analysis introduced me to the world of simulation, an inspiring application of theoretical mathematics. My efforts have been to give an intuitive yet mathematical approach to the topic, regardless of the reader's previous knowledge of the subject.

1 Ordinary differential equations

More general, an equation including variables $x_1, \ldots, x_m$, an unknown function $y(x_1, \ldots, x_m)$ of those variables and one or several derivatives of this function is known as a *differential equation*. The equation is an ordinary differential equation (ODE) if the unknown function y is a real or imaginary-valued function of one variable, that is: $y(t)\colon \mathbb{R} \mapsto \mathbb{R}$ or $y(t)\colon \mathbb{R} \mapsto \mathbb{C}$. In this case, the functions $y^{(k)}(t)$, $k \in \{1, \ldots, n\}$ are all derivatives of y with respect to the same variable t and the highest order derivative defines the order of the differential equation. For example the equation

$$y''(t) + a\sin(y(t)) = 0 \tag{1.1}$$

defines an ordinary differential equation of order two. A solution to a ordinary differential equation is a function $y(t)$ that satisfies the equation on its domain. In this example, a solution does not exist on closed form but is represented by a curve in the plane. The equation has infinitely many solutions but by introducing initial conditions $y(0) = 0, y'(0) = 1$ only a unique solution exists [1, p.5-7][2, p.443-447].

After this first example, a general ODE of order two with initial conditions is defined:

$$F(t, y(t), y'(t), y''(t)) = 0,\ (y(t_0), y'(t_0)) \in \mathbb{R}^2 \tag{1.2}$$

and also a general ODE of order $n \in \mathbb{Z}$ with initial conditions

$$F(t, y(t), \ldots, y^{(n)}(t)) = 0,\ \mathbf{y}(t_0) = \mathbf{t}_0 \tag{1.3}$$

where $\mathbf{y}(t) = (y(t), \ldots, y^{(n-1)}(t))$ and $\mathbf{t}_0 \in \mathbb{R}^n$ [1, p.1-2].

1.1 Different kinds of ODE:s

An ordinary differential equation of order n is said to be linear if it can be written as a linear combination

$$y^{(n)}(t) + \sum_{k=0}^{n-1} f_k(t) y^{(k)}(t) = g(t) \tag{1.4}$$

where $g(t)$ and $f_k(t)$ are continuous functions for $k \in \{0, \ldots, n-1\}$ and said to be non-linear otherwise. In particular, an ordinary linear differential equation does not include expressions of the form $(y(t))^2$ or $\sin(y(t))$. For an ordinary linear differential equation, any linear combination of solutions is a solution as well. For example is our differential equation modelling the pendulum non-linear since equation (1.1) contains the term $\sin(y(t))$. If one instead considers a pendulum where $y(t)$ is close to zero, making the approximation $\sin(y(t)) \approx y(t)$ since $\sin(y(t)) = y(t) + O(y^3(t))$ which instead gives

$$y''(t) + ay(t) = 0 \tag{1.5}$$

which is a linear ODE [3, p.18].

Depending on the relation between $y^{(n)}(t)$ and the other $n-1$ derivatives of $y(t)$ another categorisation of ordinary differential equations can be made. If the equation can be expressed as

$$F(t, y(t), y'(t), y''(t), \ldots, y^{(n-1)}(t)) = y^{(n)}(t) \tag{1.6}$$

for a function F, the equation is an explicit ordinary differential equation, while the more general relation

$$\hat{F}(t, y(t), y'(t), y''(t), \ldots, y^{(n)}(t)) = 0 \tag{1.7}$$

for a function $\hat{F}$ defines an implicit ordinary differential equation. Note that an implicit ODE can locally be written as an explicit ODE if the implicit function theorem is applicable [1, p.2].

The last property discussed before proceeding is discontinuity of ordinary differential equations. First note that the solution to an ODE is a continuous function $y(t)$. However, an ODE can be represented by different equations on different intervals, for example consider the ODE defined by

$$F = \begin{cases} F_{t<0}(t, y(t), \ldots, y^{(n)}(t)), & t < 0 \\ F_{t\geq 0}(t, y(t), \ldots, y^{(n)}(t)), & t \geq 0 \end{cases} \tag{1.8}$$

with solution $y_{t<0}(t)$ and $y_{t\geq 0}(t)$ where $\lim_{\epsilon \to 0^-} y_{t<0}(\epsilon) \neq y_{t\geq 0}(0)$ but still continuous on their domains. An ODE of this type, where F is not continuous

in each component over the interval where the equation is defined, but has to be represented by different ODE:s on different time-intervals, is called a *discontinuous ordinary differential equation* [4, p.196-198].

1.2 System of differential equations

A *system of ordinary differential equations* is a set of two or more coupled ordinary differential equations. The order of the system corresponds to the highest present derivative and sometimes an ODE or system of ODE:s of higher order can, for example by introduction of new variables, be reduced to a system of order one. The second order differential equation describing the pendulum

$$y''(t) + a\sin(y(t)) = 0 \tag{1.9}$$

can by introducing $y_1(t) = y(t)$, $y_2(t) = y_1'(t)$ be reduced to an equivalent non-linear system of order one.

$$\begin{aligned} y_1'(t) &= y_2(t) \\ y_2'(t) &= -a\sin(y_1(t)) \end{aligned} \tag{1.10}$$

With equivalent, it is referred to the identical set of solutions to the single equation and the system of equations. Similar to single equations, a system of differential equations is said to be linear if each one of the single equations is linear. For example, the approximation $\sin(y) \approx y$ given earlier, makes the system linear [2, p.409-415]:

$$\begin{aligned} y_1'(t) &= y_2(t) \\ y_2'(t) &= -ay_1(t) \end{aligned} \tag{1.11}$$

A linear system can be written in matrix-vector form and in particular a system of order one can written as

$$\mathbf{y}' = \mathbf{A}(t)\mathbf{y} + \mathbf{b}(t) \tag{1.12}$$

where $\mathbf{y}' = (y_1', \ldots, y_n')$, $\mathbf{y} = (y_1, \ldots, y_n)$, $\mathbf{b}(t) = (b_1(t), \ldots, b_n(t))$ and

$$\mathbf{A}(t) = \begin{pmatrix} a_{11}(t) & a_{12}(t) & \ldots & a_{1n}(t) \\ a_{21}(t) & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ a_{n1}(t) & \ldots & \ldots & a_{nn}(t) \end{pmatrix}$$

is an $n \times n$ matrix. Note that the elements in $\mathbf{A}$ do not have to be functions of the independent variable t but they may be constants [4, p.64]. This occurs in the linear version of the pendulum, namely:

$$\begin{pmatrix} y_1'(t) \\ y_2'(t) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -a & 0 \end{pmatrix} \begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{1.13}$$

The theory of existence and uniqueness of solutions to a linear system of differential equations of first order is rather evolved.[1] On the other hand, for non-linear systems the solutions might not be written on closed form but rather approximated as a curve in the phase-plane. To compute the approximation, one might consider the linear system that in some sense is closest to the non-linear one. A method known as linearisation gives an approximation in an open set close to a given point. For clearness, let our system only contain two equations and let the given point be (y_1^*, y_2^*). Then our approximation is given by:

$$\begin{aligned} \begin{pmatrix} y_1' \\ y_2' \end{pmatrix} &= \begin{pmatrix} f_1(y_1, y_2) \\ f_2(y_1, y_2) \end{pmatrix} \approx \\ \begin{pmatrix} f_1(y_1^*, y_2^*) \\ f_2(y_1^*, y_2^*) \end{pmatrix} &+ \begin{pmatrix} \frac{\partial f_1}{\partial y_1}(y_1^*, y_2^*)(y_1 - y_1^*) + \frac{\partial f_1}{\partial y_2}(y_1^*, y_2^*)(y_2 - y_2^*) \\ \frac{\partial f_2}{\partial y_1}(y_1^*, y_2^*)(y_1 - y_1^*) + \frac{\partial f_2}{\partial y_2}(y_1^*, y_2^*)(y_2 - y_2^*) \end{pmatrix} \end{aligned} \tag{1.14}$$

where the Jacobian of the system evaluated in the fixed point appears in the last term as a matrix-vector multiplication [4, p.69].

Now recall the linearised system describing the pendulum and recognize the matrix as the Jacobian evaluated in $(0, y_2^*)$ namely:

$$\begin{pmatrix} 0 & 1 \\ -a & 0 \end{pmatrix} = \mathbf{J}(0, y_2^*) \tag{1.15}$$

Even if this technique is not chosen to be used in the numerical computations, the Jacobian is sometimes needed in the numerical algorithms to find a solution. The evaluation of Jacobians is an essential part of this thesis since it is the main subject of the experiments in Chapter 4.

1.3 System of ODE:s with in- and outputs

The last section of this chapter considers a, in some sense, special case of a system of differential equations. Even if the form looks general, it is special in the sense that it is an explicit system of first order differential equations

[1] [4, p.35-40, p.51-54]

with the function $u(t)$ as a component on the right hand side of the equations, requires initial conditions and introduces output functions. Later in this thesis, when considering a standardized system of differential equations, this is the structure to bear in mind. Vector notation is used, where the underlying dimension is $n \in \mathbb{N}_{\geq 1}$, stating the system as [5]:

$$\begin{aligned} \mathbf{y}' &= \mathbf{f}(\mathbf{y}, \mathbf{u}, t) \\ \mathbf{v} &= \mathbf{g}(\mathbf{y}, \mathbf{u}, t) \\ \mathbf{y}(t_0) &= \mathbf{y}_0 \end{aligned} \tag{1.16}$$

Here the function $u(t)$, the input function, is known for every $t \in \mathbb{R}$ and can be the solution to another system or more generally, a function of the solution to another system. That is, $\mathbf{u}(t) = \hat{\mathbf{v}}(t)$ where $\hat{\mathbf{v}}(t)$ is a output function of another system which can be calculated as soon as that system is solved for any $t \in \mathbb{R}$.

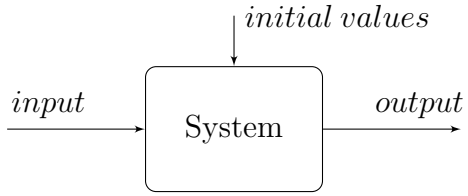

Figure 1.1: System with in- and outputs

This means that the solution to this system is dependent on the solution of another system, just like a third system's input function is our system's output function. In this way, it is possible to connect multiple systems of differential equations together where the different solutions require knowledge of each other. In applications, $u(t)$ is sometimes known as an input signal and in our pendulum-example this could correspond to an external force interfering with the motion of the pendulum, and this force might very well depend on the behaviour of other physical objects or something controlled by humans[5].

This thesis covers the implementation of the simulation environment PyFMI 2.0 which enables the user to import packages of those systems. It is then of importance that the structure of this system is well-defined so that no misunderstandings occurs in the communication between PyFMI and the systems. For this purpose, the newly released FMI 2.0 standard from MODELISAR is used.

2 Functional Mock-up Interface

The following chapter gives an introduction to the Functional Mock-up Interface (FMI) and emphasizes the main arguments and guidelines behind the foundation of the standard. It includes the structure of the Functional Mock-up Unit (FMU) and a presentation of the two types of FMU:s, namely Model Exchange and Co-Simulation. Especially, models for Model Exchange are explained from a mathematical view, and the main concept of simulation of the both types is treated in the end.

2.1 FMI

The behaviour of a physical object may sometimes be described by equations, as the pendulum in previous chapters or a bouncing ball, which makes it possible to create a model of the object. In particular, the motions of the objects above are, using the laws of Newton, suitably described by a second-order ODE which can be reduced to a system of two first order ODE:s. A modelling environment can export the equations, that describe the object, as callable code. Then the, for the modeller possibly unknown, end-user can evaluate the equations and perform a simulation over time of the object. This raises difficulties for the end user who can not use third-party software for the simulation since all modelling environments do not export the executable code in a mutual standardized way, making each model unnecessary dependent on the modelling environment. FMI defines, among other things, a standardized structure of the final callable code that could be exported, making the simulation environment independent of the source. FMI also defines a set of C-functions and data-types used for communication with the model and its variables. The callable code is exported as a FMU, that is a zip-file containing all necessary information about the model.

The FMI-standard and hence the exported FMU:s have been developed to support a wide range of platforms and simulation environments by avoiding unnecessary restrictions. The following list includes some of the guidelines

behind the development used to keep FMI robust[6, p.7-10].

- As mentioned above, the process of creating and exporting an FMU is independent of the simulation environment.
- Support for common modelling languages, that is Modelica, Simulink and SIMPACK models can be converted to FMU:s.
- The communication between a simulation environment and the imported FMU will not be slowed down by the FMI-functions.
- An FMU will be small to avoid unnecessary restriction of the simulation environments due to low memory. A part of this is the xml-file that helps storing static information so only information needed to perform a simulation needs to be kept in the memory.
- The interface will support numerical solvers used by the simulation environment, that is via FMI calls provide vectors with states, derivatives and other information needed by the solver.

2.2 Functional Mock-up Unit

The zip-file called FMU contains several files used by a simulation environment to create one or more instances of the model. The core is the standardized library (.dll, .so or equivalent) containing the code generated from the model-equations which can be accessed by the functions defined in FMI. Note that the whole structure of the FMU is a part of the FMI-standard. The xml-file can be read by any xml-parser but the library needs a defined interface for communication. A FMU is within the frames of the FMI-standard, either a FMU for Model Exchange (ME) or a FMU for Co-simulation (CS). The former is characterized by the need of a numerical integrator since it is a system of ODE:s while the latter type, the CS, is pre-packed with its own solver so the underlying differential equations may be hidden for the simulation environment. Both cases are treated more rigorously in the following sections of this chapter. Moreover, all or a subset of the model-variables and their properties are stored in a standardized way in the xml-file as a part of the FMU and accessed by any xml-parser and interpreted by the simulation environment. This makes the simulation more memory efficient since all variable attributes and static information are not stored in the memory but retrieved only if needed. Other model information and capabilities of the FMU are also specified in the xml-file. The FMI standard requires at

least one library in the FMU for creating an instance of the model on different types of platforms. The FMU also allows storage of tables and other libraries used by the model, documentation and a bitmap as model-icon[6, p.3].

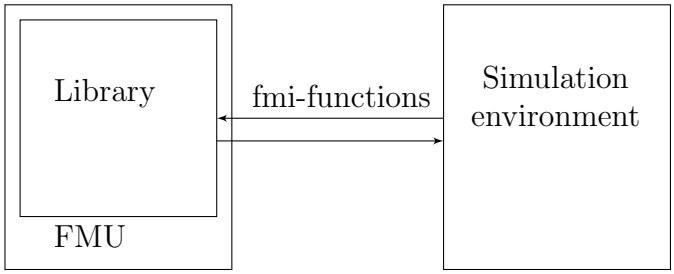

Figure 2.1: Interacting with FMU:s

By importing and unpacking FMU:s a simulation environment can create one or more instances of a model and simulate it over an arbitrary time interval. Note that performing a simulation of a model is equivalent to, in the case of ME, solving the system of ODE:s and for CS calling the built-in solver. The FMI-standard allows a way of connecting models together, making it possible to simulate large scale time-dependent systems using small independent components/systems of ODE:s with inputs and outputs that can be simulated/solved individually. This feature is a key-stone of the interface since it does not only allow the parts to simulated individually but it also allows models to be created independent of each other and then put together for simulation. An extended example would be to fasten another pendulum to the first pendulum (Figure 2.2) and optionally let the second pendulum has an input signal that affects the motion of that pendulum. Another component could instead be chosen to couple to the pendulum to make it more convenient that the different components can be created by different independent vendors. The systems of equations could be coupled together as in Figure 2.3.

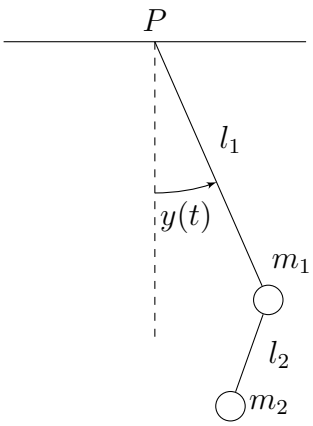

Figure 2.2: Double pendulum

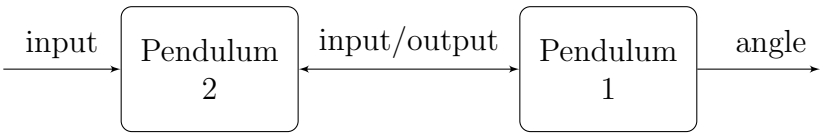

Figure 2.3: Connections between the pendulums

2.3 Common representation

The simulation of a ME-model differs a lot from a CS-model since the former needs an external numerical integrator for simulation, but from the point where the FMU is imported by a simulation environment, for example PyFMI, to the point where the simulation begins, the two kinds of models have large parts in common. Again recall the pendulum from the introduction, where the constant $a = l/g$ is the quotient between the length of the string and the gravity. If the user has a need to simulate the pendulum with certain values on l and g, those must be specified at some point, otherwise the system will use a set of default values and if other variables depend on a, this value has to be calculated before it is possible to calculate the value of those variables. Also, the user might want to change the length of the string or the gravity during the simulation and the FMU needs defined rules of whenever this is allowed.

In a FMU, each model-variable is a data-struct called fmiScalarVariable defined by the FMI 2.0. In the pendulum, the model-variables could for example be t, y_1, y_2, l, g. FmiScalarVariables mainly contain attributes with information and definitions of the variable and a type-definition. In FMI 2.0, the struct fmiSimpleType has five type-definitions, real, integer, Boolean,

string and enumeration and each model-variable has to be defined as one of these[6, p.35-37]. Other attributes are, among others:

- Name: a name of the model-variable.
- Value reference: an integer used as a handle to refer to the variable.
- Causality: defines the relation to other variables.
  - Parameter: the value is independent of other variables and constant during simulation.
  - Input: the value can be provided (for example as an output) from another model.
  - Output: the variable value can be used by another model.
  - Local: calculated from other variables, not used by another model.
- Variability: defines when the variable is allowed to change its value.
  - Constant: the value never changes.
  - Fixed: fixed after the model has been initialized.
  - Tunable: can only change its value at events, in particular tunable parameters are allowed.
  - Discrete: can only change its value at events.
  - Continuous: can change its value at any time.
- Initial: Defines how the initial value is calculated before the simulation starts.
  - Exact: initialized with a predefined start value.
  - Approx: initial value is the result of an iteration starting with the predefined start value.
  - Calculated: calculated from other variables.

Note that the attributes causality, variability and initial are not independent of each other. For example a variable can not have variability constant and causality input, for an exact table of allowed combinations, see the FMI documentation. Also the five type-definitions have their own attributes which the model-variables inherits, for example their values [6, p.39-43]. The gravity variable g in the pendulum model could for example be defined as following:

- Data-type: real

  - Start: -9.81
  - Min: -10
  - Max: 0

- Name: "Gravity"
- Value reference: 2
- Causality: parameter
- Variability: fixed
- Initial: exact

Each FMU also needs a defined model-structure, basically consisting of a set of inputs, outputs and derivatives. The variables listed under inputs/outputs are simply those whose causality is equal to input/output. The derivatives of the continuous states are an ordered set, in the case of the pendulum this is simply (y_1', y_2'). Note that the ordered set is of importance so the simulation environment, PyFMI, knows the relation between the right and left hand side of the system. Also note that derivatives and model-variables do not have to be exposed to the user, only accessible via FMI-functions, to solve the system. This can be used by model-manufacturers who can export a FMU without revealing knowledge of the model-equations [6, p.47-50].

In summary, before beginning the simulation, a simulation environment like PyFMI has to perform three main steps. The first is unpacking the imported FMU, which is a zip-file, read the xml, other included resources and connect to the library (.dll on Windows). Then the environment creates a unique instance of the model. This is necessary since some FMU have the capability to be instantiated several times allowing the environment to simulate multiple objects connected together using the same FMU. When instantiated, model-variables with initial conditions equal to "exact" or "approx" can be set. Finally, the model is initialized and the initial values of the model-variables are calculated, this is the beginning of the simulation.

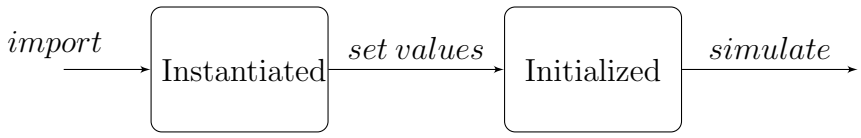

Figure 2.4: Simulator stages

2.4 FMI for Model Exchange

Simulating an ME-model is equivalent to solving a system of differential equations on state space form with events. Recall the definition from Chapter 1 and consider the equations again, in a slightly different way to highlight the different types of model-variables.

$$\begin{aligned} \mathbf{x}' &= \mathbf{f}(\mathbf{x}, \mathbf{u}, \mathbf{m}, t) \\ \mathbf{v} &= \mathbf{g}(\mathbf{x}, \mathbf{u}, \mathbf{m}, t) \\ \mathbf{x}(t_0) &= \mathbf{x}_0 \\ \mathbf{z} &= \mathbf{z}(\mathbf{x}, \mathbf{u}, \mathbf{m}, t) \end{aligned} \tag{2.1}$$

$\mathbf{x}(t)$ is a vector of the continuous state and is simply the vector $\mathbf{y}(t)$ from Chapter 1 and $\mathbf{m}(t)$ is a vector of time-discrete variables which are constant between events. Events are time-points $t_0, t_1, \ldots, t_m$, such that $t_{i+1} > t_i$, where discontinuities in the solutions occur, making this system piecewise continuous. On each interval $t_i \leq t < t_{i+1}$ the vector $\mathbf{x}(t)$ is continuous in t and continuous from the right in the endpoint, $\mathbf{x}(t_i) = \lim_{\epsilon \to 0^+} \mathbf{x}(t_i + \epsilon)$ and $\mathbf{m}(t_i) = \mathbf{m}(t)$. $\mathbf{m}(t)$ is constant between events and $\mathbf{x}(t)$ is continuous between events. The vector $\mathbf{v}$ is the output of the model that can be used in another model. The vector $\mathbf{z}(t)$ is a vector of so called event indicators, functions of time and the model-variables, that are continuous between events. If one of the event indicators $z_i(t)$ changes value from $z_i(t) > 0$ to $0 \geq z_i(t)$ or the other way around, a state event is triggered. The numerical integrator needs to check this vector after each integration step, and if needed, step back in time to find the exact time of the event. Note that with exact it is referred to within a given tolerance. In a FMU there are three different kind of events [6, p.56-58]:

- Time-event: events that occur at fixed time instants defined before the simulation begins. For example this could be an instant increase in the speed of the pendulum at a given time during the simulation. No indicator is needed for those since the environment knows when they occur, at for example $t = 3.0$.

- State-event: events that occur at time instants that are unknown when the simulation begins. The vector $\mathbf{z}(t)$ indicates when a state-event occurs. For example, consider the pendulum that bumps against a block fixed at an angle of 45 degrees from the equilibrium position (Figure 2.5). An event indicator could be $z_1(t) = 45 + y_1(t)$ that indicates that if the integrator computes that the angle $y_1(t) \leq -45$, a bounce have occurred and a state event is triggered. Note that the

variable describing the speed $y_2(t)$ is discontinuous at this point since it changes to $-y_2(t)$ at the bounce.

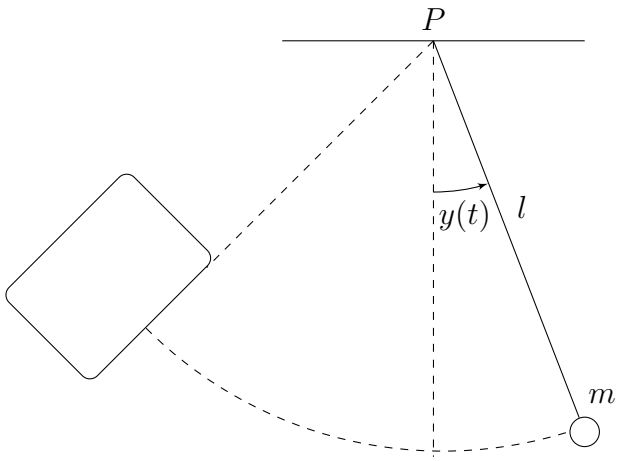

Figure 2.5: Pendulum with bounce

- Step-event: a numerical integrator has to inform the FMU about each completed integrator step. If the continuous states are not numerically suitable, the FMU triggers a step-event to stop the integration and change the states.

In summary, the events are internally handled by the FMU but the integrator has to be observant on indications from the FMU that an event has occurred. A simulation environment like PyFMI has to retrieve this information from the FMU by calling the FMI-functions and pass it to the integrator whenever demanded by the integrator. To perform a simulation from start-time t_{start} to stop-time t_{stop}, the following scheme (Figure 2.6) is followed [6, p.61-66]:

- Instantiated: a model instance is created.
- Continuous Evaluation: the FMU enters this state as soon as initialized and the solution at time $t = t_{start}$ is calculated. This is also the state of the FMU when the integrator is performing an integrator step. Here the integrator can set and get continuous states, retrieve derivatives, directional derivatives, event indicators and other information needed to perform an integrator step. If no event was detected, the simulation environment calls the function fmiCompletedIntegratorStep which marks the end of the current integrator step. Then values can

be stored and another integrator step can be taken. Except the state, only real continuous variables with causality input can be set in this state. Recall that discrete variables are constant between events and in particular, Continuous Evaluation is the state between events. If a state event was detected during the integrator step, an iteration over time is performed in this state until the time for the event has been determined, up to a given tolerance. Note that if an event occurred, the FMU changes state to Set Inputs before proceeding to Event Pending, otherwise it stays in the current state.

- Set Inputs: when an event occurs, the FMU changes state from Continuous Evaluation to set inputs. At this state, values of discrete variables with causality input and values of tunable variables are retrieved and set before proceeding to the next state, Event Pending. Those values have not been renewed since the FMU was in this state at the last event and are therefore not accurate any more. Recall that the variable values at discontinuities are continuous from the right but not generally from the left, so they need to be updated before proceeding. Those newly set values will be stored as soon Event Pending has finished, in other words, the integration step is complete and the FMU is back at Continuous Evaluation.

- Event Pending: The events are internally handled by the FMU at this stage. The simulation environment keeps calling the function fmiEventUpdate until the system is updated. Then the function fmiCompletedEventIteration is called and the state changes back to Continuous Evaluation and the integrator step is complete, which means that values will be stored and the simulation can continue in the state Continuous Evaluation.

- Terminated: The simulation has either been aborted or reached the final time $t = t_{stop}$ for which the solution is retrieved before the model instance is disposed of. Note that a time event occurs at $t = t_{stop}$, which means that the stop time is detected and handled as an event as above.

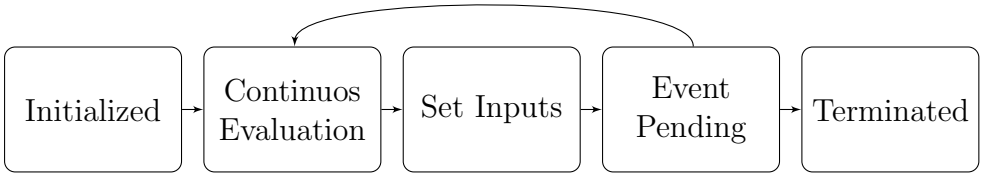

Figure 2.6: States

2.5 FMI for Co-Simulation

To simulate a FMU for Co-simulation, no external numerical integrator is needed, but instead the FMU uses its individual built-in integrator so the interface suits well for coupling two or more FMU:s together for simulation of larger dynamic systems. For this purpose, a master algorithm is needed which handles the communication between the involved models, in other words retrieving outputs and setting inputs at certain time instants. Since FMI do not define this algorithm, no such algorithm is implemented in PyFMI, and therefore not given much focus in this thesis. Instead focus will be given to the structure of a CS-model and the simulation process of a single model [6, p.75-76].

Since each Co-Simulation FMU uses its individual solver that is hidden from the user in the library in the FMU, those models do not need such a strict predefined structure of the involved equations. As long as the FMU returns correctly on the calls from the FMI-functions, the manufacturer can hide knowledge of the solver inside the library in the FMU. Before starting the simulation of a model, as with a ME-model, the simulation environment needs to unpack the zip-file, read the xml-file and further data and also connect to the library. Then an instance of the model is created and variables with initial equal to "exact" or "approx" can be set. Finally the slave (the built-in solver) is instantiated and initial values are computed, which marks the start of the simulation. To perform a simulation, the solver computes the solution on subintervals $[t_i, t_{i+1}]$ of non-zero length of the whole simulation interval $[t_{start}, t_{stop}]$, where $t_0 = t_{start}$ and $t_N = t_{stop}$. The step size is then by definition $h_i = t_{i+1} - t_i$ where $i = 0, \ldots, N-1$. If the solver can handle variable step size, this is announced by the capability flags in the FMU. Similar to a ME-model, a simulation environment follows this scheme (Figure 2.7) to perform a simulation of a CS-model [6, p.77-83]:

1. Instantiate Slave: a model instance (including the solver) is created.
2. Initialize: the model is initialized and the solution at time $t = t_{start}$ can be retrieved. When initializing, t_{stop} is optionally defined. Defining the

simulation interval when initializing can be used to check if the model is valid on that interval. Also output is retrieved to be used by other models if any.

3. Do step: the built-in solver performs integration over the interval $[t_i, t_{i+1}]$. At time $t = t_{i+1}$ the input/output variable values of the models are set/get and the solution at this time instant is retrieved. If no error occurs, this process is iterated until $t_{i+1} = t_{stop}$.

4. Terminate Slave: the simulation has reached its end. The solution at this point is retrieved and the model is disposed of with this call.

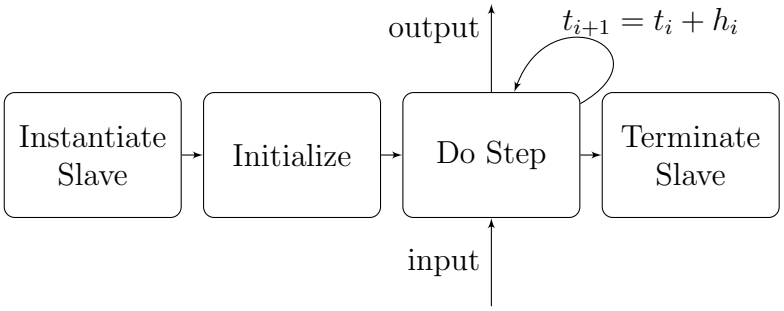

Figure 2.7: FMI-function calls

3 FMI 2.0 compared to 1.0

The beta FMI 2.0 version was released in august 2012 and includes both differences and new features compared to the existing version, FMI 1.0. The CS-interface and ME-interface have been merged, which implies that a FMU can contain both a CS-model and a ME-model and due to this, the fmi-functions for FMI 2.0 are not backward compatible with FMI 1.0. Many of the new features are optional and each FMU is equipped with capability flags, providing the simulation environment with the limits of the model.

3.1 Differences

The following are the main improvements in FMI 2.0 compared to FMI 1.0 [6, p.96-100] [7][8].

- The variability "tunable" is added for parameters. A parameter with variability tunable can, in opposite to a parameter with variability fixed, change its value during a simulation. A tunable input variable is a tunable parameter from another model. A tunable parameter can change its value only at events. If a tunable parameter or tunable input changes its value at an event, tunable output and tunable local variables must be recomputed. This is treated as a start of a new simulation where the initial values are the current values, including the newly computed parameters.

- For better event handling, FMI 2.0 for ME has been equipped with a new function fmiEventIterationConverged, that has to be called after the FMU has completed the update after a triggered event. This function-call marks the transition from the state Event Update to Continuous Evaluation.

- A variable can optionally have a defined unit, in which the variable value is set and retrieved in with the fmi-functions. In FMI 2.0 seven SI units, "kg", "m", "s", "A", "K", "mol" and "cd" are included together

with the unit "rad" and the unit of a variable value has to be a function of those. For example the gravity in the pendulum would be m/s^2. The standardized unit definition makes it easier to convert an output variable value to an input variable value between different models if the units mismatch. Those predefined attributes differs from FMI 1.0 where the unit was displayed as a string, but had a loss of naming convention, making it difficult for a simulation environment to connect FMU:s together.

- In FMI 2.0 the input, output and state variables are ordered sets. This information can be used by the simulation environment for linearizing (recall Chapter 1). Optionally the derivatives dependency on the states can be defined which can be used to efficiently provide the Jacobian as a sparse matrix.

- The set of capability flags has been extended to provide information about the new features. The new flags for ME are:

  - completedIntegratorStepNotNeeded
  - canBeInstantiatedOnlyOncePerProcess
  - canNotUseMemoryManagementFunctions
  - canGetAndSetFMUstate
  - canSerializeFMUstate
  - providesDirectionalDerivatives
  - completedEventIterationIsProvided

  The new flags for CS are:

  - needsExecutionTool
  - canGetAndSetFMUstate
  - canSerializeFMUstate

  while

  - canRejectSteps

  has been removed.

- In FMI 1.0, variables of the same type with equal value reference and equal or negated value was defined as aliases and anti-aliases. In FMI 2.0, variables are not explicitly defined to be aliases, but if they are

of the same type and have identical value references, they must have equal values. Note that they might still have different attributes.

- In FMI 1.0, the interface was equipped with logging that logged the function calls from the simulation environment to the FMU. In FMI 2.0 this logging utility has defined logging categories, giving the simulation environment the possibility to log all or only a subset of the logging categories. The creator of the FMU can define own categories but the standardized are:
  - logEvents
  - logSingularLinearSystems
  - logNonlinearSystems
  - logDynamicStateSelection

3.2 New features

The following are the main new features in FMI 2.0 compared to FMI 1.0. FMU:s use capability flags to signal the ability to handle the new features [6, p.96-100].

- **Jacobians:** The ability to provide directional derivatives implies the ability to provide Jacobians by using proper seed vectors to retrieve the partial derivatives from the directional derivative. The Jacobian can be used by the numerical integrator or when coupling FMU:s together. FMI 2.0 provides the option to define the outputs and derivatives dependencies on the states together with respectively type of dependency, where the dependencies "non-linear", "parameter" and "discrete" are available. This information can be used by the FMU for faster computation of the Jacobian. In the same way, variables can define their dependency on the input variables.

- **FMU-states:** FMU:s can optionally save the complete state including all information needed to restarting the simulation at the current state later on. This could for example be done after each completed integrator step over all simulated FMU:s in case of failures in the next step. If indicated by the capability flags, the FMU can serialize and deserialize the FMU-state for storage on file.

4 PyFMI 2.0

One of the main subjects of this thesis is the implementation of PyFMI 2.0 which is an extension to the already existing PyFMI software. PyFMI is a Python based simulation environment, with support for both ME FMU:s and CS FMU:s, where the user can load FMU:s for evaluation and integration, in other words simulation. Through a Python-shell, PyFMI aims to provide a user-friendly approach to the FMI-functionality via PyFMI provided functions. The subject of PyFMI 2.0 is to extend this functionality to include support for ME and CS FMU:s of version 2.0 according to the new FMI 2.0. PyFMI is a free to use open-source software which can be downloaded together with all requirements [9] or as a part of JModelica.org [10]. PyFMI is using the following packages to connect and communicate with the FMU and for numerical integration:

- FMI Library, FMIL: is a C-language application from Modelon AB that provides a complete interface to the FMI-standard making the interaction with FMU:s via fmi-function-calls easier. It also handles unzipping of FMU:s, access to the model-equations by connecting the DLL-file and parsing of the xml-file to retrieve model information.
- Assimulo: the numerical integrator used by PyFMI to compute the solution to FMU:s for ME. Assimulo is a package for solving explicit and implicit ODE:s. PyFMI contains an adapted problem class for the simulation of a FMU for ME [11].

4.1 Usage

The following section covers the basic usage of PyFMI 2.0, from loading a FMU to simulation and settings of available options. It is supposed that PyFMI 2.0 has been imported to the current Python session before the commands are run. Note that this is not a documentation nor a complete description of the functionality of PyFMI but rather an overview of the intended

way to make use of PyFMI. Default arguments in function-calls might not be displayed and explained unless required for the understanding of what is returned.

4.1.1 Loading a FMU

The following line of code can be used to load a FMU in PyFMI 2.0. The load-function finds out the kind and version of the FMU and creates an instance accordingly, in this case, model.

```
>> model = load_fmu(fmu = 'MyModel.fmu', path = 'c:/
   FMUdirectory')
```

The load-function takes a total of five input arguments where only the first lacks a default value:

- fmu: File name of the FMU to be loaded.
- path: Full path to the directory of the FMU.
- enable_logging: Boolean that enables/disables logging of errors.
- kind: Since a FMU of version 2.0 can contain both a ME and CS model, one has to be chosen in those cases.
- log_file_name: Defines a customized file-name for the log-file.

Note that after this function-call, the model has been instantiated but not yet initialized.

4.1.2 Simulation

Once a model has been created, a simulation can be performed. In the most simple case, only the following command is needed to perform a default simulation between $t_{start} = 0.0$ and $t_{stop} = 1.0$, where object "res" becomes an instance of the result.

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

This mutual command that works for both CS and ME performs a simulation with default values. In total, the simulate-function takes five input arguments.

- start_time: start time of the simulation.
- final_time: stop time of the simulation.

- input: input signal for the simulation.
- algorithm: your own simulate/solver-algorithm compatible with PyFMI and Assimulo can be used instead.
- options: options for the solver (see next subsection).

While using PyFMI 2.0, adding a question mark after the method provides a full list of available input arguments and default values. That is:

```
>> model.simulate?
```

Furthermore, the result is retrieved as a vector including each simulation-step for the variable called "variable_name" by this command:

```
>> result_variable = res['variable_name']
```

The whole list of model-variables is retrieved by:

```
>> res.keys()
```

while the solution at start time and end time is retrieved by:

```
>> res.initial('variable_name')
>> res.final('variable_name')
```

Moreover, the plot GUI can visualize the solution to selected variables by reading this data from the result file.

4.1.3 Setting options

This chapter presents how options for the model and simulation can be set, in other words overwrite the default values, before the actual simulation is carried out. Note that simulation over an arbitrary interval is possible by only two commands as shown above. Also note that previous subsections made no difference between ME and CS models but when interacting with models, the options available is dependent on the FMU-kind. Differences in this subsection will be evident from the context.

For both ME and CS models, the first of the following commands returns all model-variables in a dictionary using variable names as keys and the values are objects containing all necessary variable attributes. For example, the middle and the last command returns the value reference and the variability of variable called 'variable_name'.

```
>> all_variables = model.get_model_variables()
>> value_reference = all_variables['variable_name'].
   value_reference
>> variability = all_variables['variable_name'].variability
```

Before initialization, the variables with causality equal 'parameter' can be set. In the following commands, the current value is evaluated and then the value reference used to set a new value for the variable.

```
>> model.get_real(value_reference)
>> model.set_real(value_reference, new_value)
```

Usually, initialization is done by the simulation function but this can be done manually to be able to set and get variables after the initialization and before simulation. The initialization of a ME-model is done by:

```
>> model.initialize(tolControlled = True, relativeTolerance =
    None)
```

Where the former argument enables the use of relative tolerances which is specified by the latter, in this case the default relative tolerance. The initialization of a CS-model is done by:

```
>> model.initialize(tStart = 0.0, tStop = 2.0,
   StopTimeDefined = True, relTol = None)
```

Since the CS-model provides a built-in integrator, the simulation interval must be defined. Use:

```
>> opts = model.simulate_options()
```

to retrieve an object with options for the simulator. The options available obviously differs between the two FMU-kinds due to different solvers but among others, the following options are mutual, and can be set:

```
>> opts['initialize'] = False
>> opts['ncp'] = 200
>> opts['result_file_name'] = 'a_desired_file_name.txt'
```

Where 'ncp' defines the number of communication points between start and stop time. Since initialization was done manually, the options are used to prevent the simulator from reinitialization and simulates as previous with.

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

4.1.4 New methods

This section presents the main set of new methods that do not have an analogue in PyFMI 1.0 but is available in PyFMI 2.0 for interacting with FMU:s of version 2.0. The following method returns the number of available logging categories together with a list of the categories.

```
>> n_categories, categories_list = model.get_categories()
```

The next four commands all return ordered dictionaries, using variable names as keys and an object with corresponding variable information as values. The first returns the derivatives, the left hand side of the system, as variables. The second returns the continuous states and the third and fourth return all variables with causality equal to input and output respectively.

```
>> derivatives_list = model.get_derivatives_list()
>> states_list = model.get_states_list()
>> input_list = model.get_input_list()
>> output_list = model.get_output_list()
```

The two latter are intended to ease the coupling of FMU:s while the two former are needed when evaluating the directional derivatives as following:

```
>> dir_der = model.get_directional_derivatives(var_ref =
   states_ref, func_ref = derivatives_ref, v = lin)
```

Where states_ref is a list with value references available in states_list above and analogue for derivatives_ref. Argument lin is a list specifying the linear combination of partial derivatives which sum up to the directional derivative.

The following methods are **not** currently working, see the Chapter 5 for a discussion, but are **intended** to be used as following when implemented. To log selected categories, use:

```
>> n_categories, categories_list = model.get_categories()
>> log_cat = categories_list[0:2] + categories_list[3]
>> model.set_debug_logging(logging_on = True, categories =
   log_cat)
```

Finally, it will be possible to save the complete state of a FMU, in other words the variable values and additional data needed to restart the simulation later from the current state. The state can also be serialized for storage as following:

```
>> current_state = model.get_fmu_state()
>> serialized_state = model.serialize_fmu_state(current_state
   )
>> model.free_fmu_state(current_state)
```

and later:

```
>> current_state = model.deserialize_fmu_state(
   serialized_state)
>> model.set_fmu_state(current_state)
>> model.free_fmu_state(current_state)
```

4.2 Example, a bouncing ball

This section covers an actual simulation of a bouncing ball by primary using the commands explained in previous subsections. The example aims to give an enlarged understanding of the usage and the possibilities together with a concrete visualized result. To clarify, finding the solution to the system of equations gives the position and speed of the bouncing ball as a function of time. The model contains five real continuous variables listed below:

- HEIGHT
- HEIGHT_SPEED
- HEIGHT_ACC
- GRAVITY
- BOUNCE_COF

Loading of the bouncing ball of type ME:

```
>> fmu = 'bouncingBall2_me'
>> path = 'C:/myFMUs'
>> model = load_fmu(fmu, path)
```

Perform a simulation during two seconds:

```
>> opts = model.simulate_options()
>> opts['ncp'] = 200
>> result = model.simulate(final_time = 2.0, options=opts)
```

Before performing another simulation resetting is required.

```
>> model.reset()
```

The model is now ready for initialization and another simulation-run. One can check the variables 'HEIGHT' and 'HEIGHT_SPEED' and replace the values with new ones. This can be done since the values are exact and not calculated from other variables. The comments are the outputs returned by the functions. In the end, an instance of the options class is retrieved and set to prevent re-initialization.

```
>> model.initialize()
>> variables = model.get_model_variables()
>> variables['HEIGHT'].value_reference          #0
>> variables['HEIGHT_SPEED'].value_reference    #1
>> model.get_real(0)                           #1.0
>> model.get_real(1)                           #4.0
```

```
>> model.set_real(0, 3.0)
>> model.set_real(1, -1.0)
>> opts = model.simulation_options()
>> opts['initialize'] = False
```

Note that this implies that the ball begins at a height of two meters and is thrown downwards to the ground. The simulation is performed, the connection terminated and the result visualized analogue to the previous simulation.

```
>> model.simulate(final_time = 2.0, options = opts)
>> model.terminate()
```

The following results (Figure 4.1) are retrieved from the plot-GUI, showing the solutions to the variables 'HEIGHT' and 'HEIGHT_SPEED' in the previous simulations:

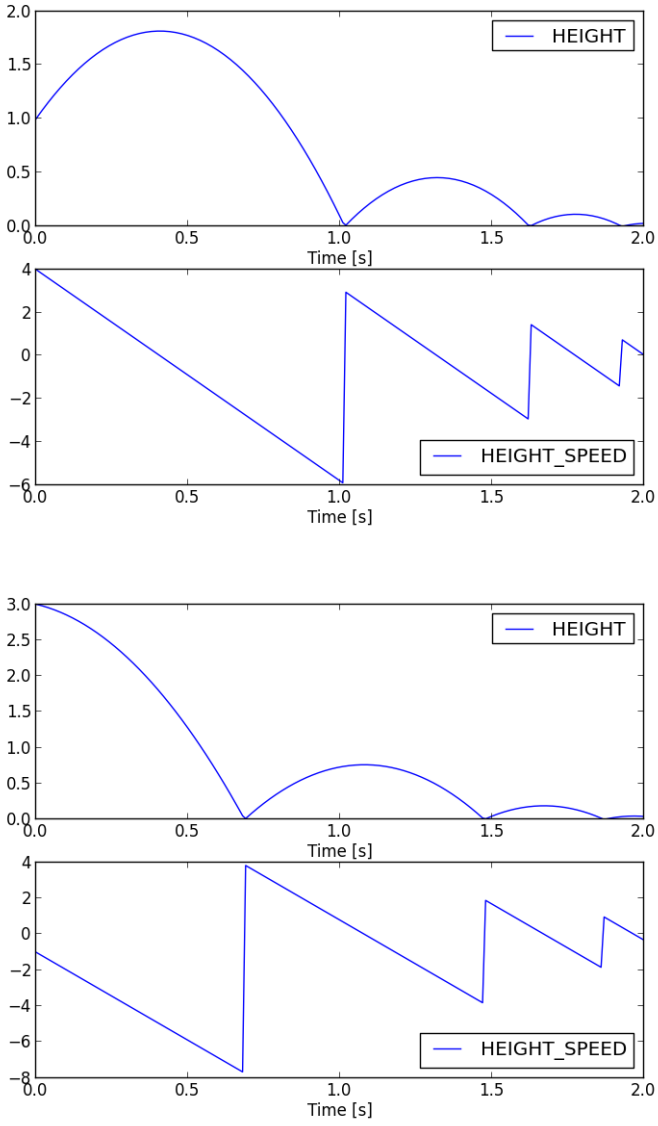

Figure 4.1: Bouncing balls

4.3 Numerical improvements

The following section evaluates possible improvements in the simulation-run due to the new FMI 2.0 standard. PyFMI 2.0 is used to import equivalent models of version 1.0 and 2.0 and running separate simulations. A rigorous description is given in the next subsection, thereafter the script used for the experiment is presented together with the result and a conclusion.

4.3.1 Set up

The aim of the experiment is to evaluate the difference in simulation time and function-calls done by Assimulo. Especially the ability to provide Jacobian matrices might result in fewer function-calls since Assimulo does not need to numerically approximate the partial derivatives in the Jacobian. In all simulations ME-models are used and those of version 2.0 have the ability to provide directional derivatives and hence Jacobians. Furthermore, the experiment consists of two pairs of FMU:s, where each pair consists of two FMU:s containing the same model, but differ in the version since one is of version 1.0 and the other of version 2.0. All FMU:s are generated by Dymola 2014.

The first model will be denoted as Coupled Clutches and describes according to the picture (Figure 4.2), three coupled clutches with masses in between. In the end-point, a torque rotates the first inertia (mass), J1, with a torque according to an input signal. The torque is transferred to the next inertia, J2, via the first clutch whose disks rotate according to the friction between them. The second and third clutch is not invoked until time-events for this purpose are triggered at time 0.4 sec and 0.9 sec respectively. Each clutch can either be forward sliding, backward sliding or locked (both disks rotate as a single one).

- Simulation time: 1.5 sec
- Number of continuous states: 8
- Number of event indicators: 54

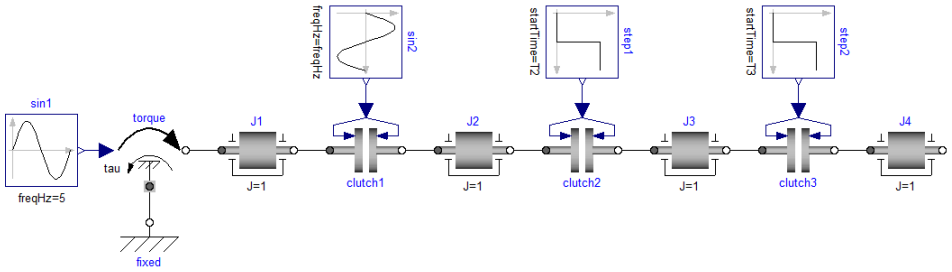

Figure 4.2: Coupled Clutches

The second model will be denoted as Robot and is a bigger model in the sense of more continuous states, thus a greater number of equations in the system. The robot uses motors, gears and breaks to move from the user defined start position to a given end position as fast as possible.

- Simulation time: 1.5 sec
- Number of continuous states: 36
- Number of event indicators: 98

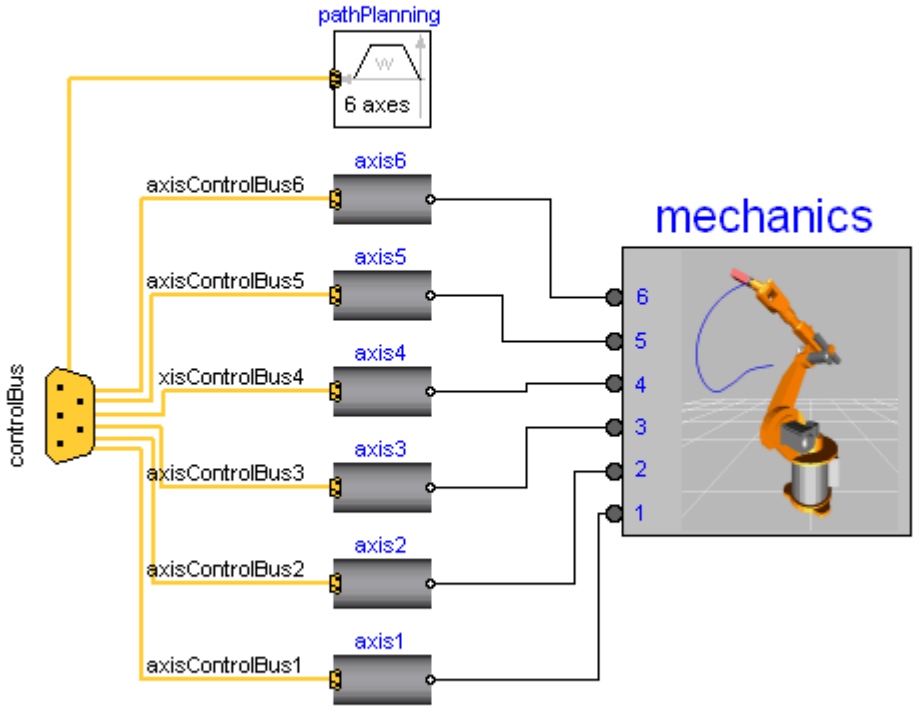

Figure 4.3: Robot

4.3.2 Experiment

The following Python scripts are used to import the FMU:s in PyFMI 2.0 and perform the simulation as described above. Note that CoupledClutches1 and Robot1 are both instances of models of version 1.0 while CoupledCluches2 and Robot2 are instances of models of version 2.0.

```
"""
Script for evaluating the difference between a FMU of version
    1.0 and 2.0.
"""
from pyfmi.fmi import load_fmu

path_to_me_1 = 'C:/myFMUs/ME1.0'
path_to_me_2 = 'C:/myFMUs/ME2.0'

fmu1='
   Modelica_Mechanics_Rotational_Examples_CoupledClutches_ME1
   .fmu'
fmu2='
   Modelica_Mechanics_Rotational_Examples_CoupledClutches_ME2
   .fmu'

CoupledClutches1 = load_fmu(fmu1, path_to_me_1)
CoupledClutches2 = load_fmu(fmu2, path_to_me_2)

CoupledClutches1.simulate(final_time=1.5)
CoupledClutches2.simulate(final_time=1.5)
```

```
"""
Script for evaluating the difference between a FMU of version
    1.0 and 2.0.
"""
from pyfmi.fmi import load_fmu

path_to_me_1 = 'C:/myFMUs/ME1.0'
path_to_me_2 = 'C:/myFMUs/ME2.0'

fmu1='Modelica_Mechanics_MultiBody_Examples_Systems_'+'
   RobotR3_fullRobot_ME1.fmu'
fmu2='Modelica_Mechanics_MultiBody_Examples_Systems_'+'
   RobotR3_fullRobot_ME2.fmu'

Robot1 = load_fmu(fmu1, path_to_me_1)
Robot2 = load_fmu(fmu2, path_to_me_2)

Robot1.simulate(final_time=1.5)
Robot2.simulate(final_time=1.5)
```

The following are the simulation results given as outputs from Assimulo. Note that the Jacobians are provided by the ME FMU of version 2.0 which reduces the number of function-evaluations within the Jacobian-evaluation to zero.

Model	**Coupled Clutches**	**Coupled Clutches**
Version	1.0	2.0
Simulation interval	0.0 - 1.5 sec	0.0 - 1.5 sec
E. simulation time (sec)	0.19824363493	0.258893650795
Number of:		
Steps	278	276
Function Evaluations	436	436
Jacobian Evaluations	12	12
F-Eval During J-Eval	96	0
Root Evaluations	365	363
Error Test Failures	21	22
Newton Iterations	388	388
Newton Conv. Failures	0	0
State-Events	9	9
Solver options:		
Solver	CVode	CVode
Linear Multistep method	BDF	BDF
Nonlinear Solver	Newton	Newton
Maxord	5	5
Tolerances (absolute)	0.000001 (all)	0.000001 (all)
Tolerances (relative)	0.0001	0.0001

Figure 4.4: Result from Coupled Clutches

Model	**Robot**	**Robot**
Version	1.0	2.0
Simulation interval	0.0 - 1.5 sec	0.0 - 1.5 sec
E. simulation time (sec)	4.11637415297	5.8523514988
Number of:		
Steps	1876	1558
Function Evaluations	2284	1981
Jacobian Evaluations	58	54
F-Eval During J-Eval	2088	0
Root Evaluations	2178	1886
Error Test Failures	24	30
Newton Iterations	2136	1833
Newton Conv. Failures	0	0
State-Events	33	33
Solver options:		
Solver	CVode	CVode
Linear Multistep method	BDF	BDF
Nonlinear Solver	Newton	Newton
Maxord	5	5
Tolerances (absolute)	0.000001 (all)	0.000001 (all)
Tolerances (relative)	0.0001	0.0001

Figure 4.5: Result from Robot

4.3.3 Conclusion

The both models of version 2.0 have a tendency to fewer function evaluations, root evaluations and Newton iterations which most likely is due to the fewer steps taken. Obviously they have no function evaluations during Jacobian evaluations since the Jacobian is provided by the FMU. Nevertheless, the models of version 2.0 have significantly longer elapsed simulation time, approximately 30 percent longer for Coupled Clutches and approximately 42 percent longer for the Robot. In PyFMI 2.0, the Jacobian is currently retrieved from the FMU as a loop over the colons in the Jacobian with calculated directional derivatives as elements, which require significantly more time than the algorithm used in Assimulo. The final conclusion is that PyFMI 2.0 needs, if possible, a more efficient way to evaluate the Jacobians, otherwise the time-loss becomes indefensible large.

5 Discussion

In summary, the concept of simulation has been studied using the standardized FMI 2.0 interface. In particular a comparison between FMI 2.0 and FMI 1.0 was done and the new features kept in mind when implemented PyFMI 2.0. Especially the thesis puts focus on the difference in simulation efficiency using the provided directional derivatives as a Jacobian instead of letting the numerical integrator itself carry out the computations. Less focus was put on the difference in elapsed simulation time between FMU:s for CS of version 1.0 versus 2.0. The main reason is the inability to change the settings of the built-in integrator, in particular the user is not able to enable or disable the use of directional derivatives. Despite the lack of obvious possible improvements in the interface it would never the less be interesting to compare simulation time of those two. No comparison is made between a ME-model of version 1.0 and a ME-model of version 2.0 without provided Jacobians since, as in the case of CS, no other improvements are assumed to reduce the simulation time significantly enough to be noticed. Bearing in mind the significantly slower simulation run for the new version it is recommended to make a further study of the iteration where the Jacobian is retrieved colon by colon as vectors with partial derivatives as well as the need of a complete Jacobian. The last argument is motivated by the few function evaluations made during the Jacobian evaluations in Assimulo. Another suggestion is to use the dependency information to create sparse matrices to get an advantage compared to the current calculations, which is not satisfactory fast. Until the simulation speed has been reduced, it is desirable to let the use of provided Jacobians be optionally.

More enhancements could be done by solving the problem with the currently yet not implemented methods explained in the subsection "New methods" in Chapter 4. Those methods, together with the methods for setting and getting the value of string-variables are not implemented correctly due to the same problem, namely the passing of pointers as input arguments to functions in Cython, in which PyFMI is written. This is required by the FMI-wrapping FMIL-functions for returning and passing strings. In par-

ticular there is a problem with passing lists of pointers, as in the case of setting categories to log. Solving this problem brings light to a set of wanted methods.

This last section presents a brief review of my own propositions on future enhancements of PyFMI 2.0. They are not dependent on a certain version of FMI but only features that could possible increase the number of users by being more user-friendly.

- **Base units:** FMI 2.0 defines base units for simplified coupling of FMU:s which is a huge founding-stone of the definition of the interface. Currently PyFMI 2.0 do not provide an easy to use functionality to retrieve and convert base units, which also requires factors and offsets.

- **Master Algorithm:** As mentioned above, coupling FMU:s is an essential part of the FMI concept and currently an user of PyFMI has to do this coupling by their own scripts. In other words, PyFMI 2.0 do not include a master algorithm for coupling FMU:s together.

- **Graphical User Interface, GUI:** A GUI would make the interaction with FMU:s more convenient. If a master algorithm was implemented, the GUI could also visualize the coupling between FMU:s, in other words make the handling of input and output signals visible for the user.

Bibliography

[1] K. G. Andersson and L.-C. Böiers, *Ordinära differentialekvationer*. Studentlitteratur, 1992.

[2] G. F. Simmons and S. G. Krantz, *Differentialekvationer med historik*. Liber, 2011.

[3] M. M. Tiller, *Modelica*. Kluwer Academic Publishers, 2001.

[4] E. Hairer, S. P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I*. Springer, 2000.

[5] MIT-OpenCourseWare, "Terminology: Systems and signals," Fall 2011.

[6] "Functional mock-up interface for model exchange and co-simulation," August 2012.

[7] "Functional mock-up interface for model exchange," January 2010.

[8] "Functional mock-up interface for co-simulation," October 2010.

[9] "https://pypi.python.org/pypi/pyfmi."

[10] "http://www.jmodelica.org/page/12."

[11] "http://www.jmodelica.org/assimulo."