API for Accessing OpenModelica Models From Python

B. Lie
University College of Southeast Norway
Porsgrunn, Norway
Email: Bernt.Lie@hit.no

S. Bajracharya, A. Mengist, L. Buffoni, A. Kumar
M. Sjölund, A. Asghar, A. Pop, P. Fritzson
Linköping University
Sweden
Email: peter.fritzson@liu.se

***Abstract*—This paper describes a new API for operating on Modelica models in Python, through OpenModelica. Modelica is an object oriented, acausal language for describing dynamic models in the form of Differential Algebraic Equations. Modelica and various implementations such as OpenModelica have limited support for model analysis, and it is of interest to integrate Modelica code with scripting languages such as Python, which facilitate the needed analysis possibilities. The API is based on a new class *ModelicaSystem* within package OMPython of OpenModelica, with methods that operate on instantiated models. Emphasis has been put on specification of a systematic structure for the various methods of the class. A simple case study involving a water tank is used to illustrate the basic ideas.**

***Keywords*-OpenModelica, Modelica, Python, Python API**

I. Introduction

Modelica is a modern, equation based, acausal language for encoding models of dynamic systems in the form of differential algebraic equations (DAEs), see e.g. [3] on Modelica and e.g. [2] on DAEs. *OpenModelica*[1] [4] is a mature, freely available toolset that includes *OpenModelica Connection Editor* (flow sheeting, textual editor with debugging facilities, and simulation environment) and the *OMShell* (command line execution, script based execution). OpenModelica Shell supports commands for simulation of Modelica models, for use of the Modelica extension *Optimica*, for carrying out analytic linearization via the Modelica package *Modelica_LinearSystem2*, and for converting Modelica models into Functional Mock-Up Units (FMUs) as well as for converting FMUs back to Modelica models. A tool *OMPython* has been developed and communicates with OpenModelica via CORBA, [5], [6]. Essentially, OMPython is a Python package which makes it possible to pass OpenModelica Shell commands as strings to a Python function, and then receive the results back into Python. This possibility does, however, require good knowledge of OpenModelica Shell commands and syntax. A tool, *PySimulator*,[2] has been developed to ease the use of Modelica from Python, [9], [6]. Essentially, PySimulator provides a GUI based on Python, where Modelica models can be run and results can presented. It is also possible to analyze the results using various packages in Python, e.g. FFT analysis. However, PySimulator currently does not give the user full freedom to integrate Modelica models with Python and use the full available set of packages in Python, or freely develop one's own analysis routines in Python.

Modelica and OpenModelica Shell in themselves have relatively little support for advanced analysis of models. Examples of such desirable analysis capabilities could be (i) study of model sensitivity, (ii) random number generation and statistical analysis, (iii) Monte Carlo simulation, (iv) advanced plotting capabilities, (v) general optimization capabilities, (vi) linear analysis and control synthesis, etc. Scripting languages such as MATLAB and Python hold most of these desirable analysis capabilities, and it is of interest to integrate Modelica models with such script languages. The free *JModelica.org* tool includes a Python package for converting Modelica models to FMUs, and then for importing the FMU as a Python object. This way, Modelica models can essentially be simulated from Python — Optimica is also supported. It is possible to do more advanced analysis with JModelica.org[3] via *CasADi*, see e.g. [7] and [8]. However, the possibilities in the work of Perera et al. use an old version of JModelica.org. It would be more ideal if these possibilities were supported by the tool developer.

It is thus of interest to develop an extension of OMPython which enables simulation and analysis of Modelica models with a better integration with the Python language, and in particular that such an extension is provided by the OpenModelica developers. A Python API[4] for controlling Modelica simulation and analysis from Python was proposed in February 2015[5]. Based on this proposal, a first version of a Python API has been implemented [1], and has then been further revised. This paper discusses the API, and illustrates how it can be used for automatic analysis of Modelica models from Python, exemplified by a simple water tank model. The paper is organized as follows. In Section 2, an overview of the API is given. In Section 3, use of the API is illustrated through simple analysis of a nonlinear reactor model. In Section 4, the API is discussed, some conclusions are drawn, and future work is discussed. Appendices hold details of the nonlinear reactor model.

[1] `www.openmodelica.org`
[2] `https://pypi.python.org/pypi/PySimulator`
[3] `www.JModelica.org`
[4] API = Application Programming Interface
[5] *Python API for Accessing OpenModelica Models*, by B. Lie, February 20, 2015, communicated to P. Fritzson at Linköping University.

II. Overview of Python API

A. Goal

Modeling and the use of Modelica with Python is of interest to a wide range of engineering disciplines. The computer science threshold of using Modelica with Python should be low. Ideally, the OMPython extension should work with simple one-click Python installations such as Anaconda[6] and Canopy[7]. Furthermore, the extension should support both 32 bit and 64 bit OpenModelica, work with both 32 bit and 64 bit Python, with Python 2.7 and Python 3.X, and on platforms Windows, OSX and Linux. These requirements e.g. imply that results should be returned as standard Python structures. However, it is reasonable that the OMPython extension depends on the NumPy package. Because Python has excellent plotting capabilities e.g. via Matplotlib, the OpenModelica Shell facility for plotting results should not be implemented — this is more naturally handled directly in Python.

B. Installing the OMPython extension

Under Windows, the new OMPython extension will be automatically installed in a file `__init__.py` in directory `share\omc\scripts\PythonInterface\OMPython` in the OpenModelica directory when OpenModelica is downloaded and installed. In order to activate the extension, the user must next run the command `python setup.py install` from the command line in the directory of the `setup.py` file, which is in the `PythonInterface` subdirectory. It follows that in order to activate the extension, the user must first install Python on the relevant computer. Under Linux/OSX, OMPython is part of `pip` (`pypi`) and is not shipped with the OpenModelica installer.

C. Status

Currently, the Python API is in a development status and has been tested with 32 bit Python 2.7 from the Anaconda installation in tandem with 32 bit OpenModelica v. 1.9.4 under Windows 8.1 and OpenModelica v. 1.9.6 under Windows 10, and a modified `__init__.py` file. OpenModelica uses CORBA for communication, and CORBA compatibility needs some refinement. The code is somewhat unstable when run from the Spyder IDE used with the Anaconda installation, but runs fine from Jupyter notebooks.

D. Description of the API

The API is described in the subsections below.

1) Python Class and Constructor: The name of the Python *class* which is used for operation on Modelica models, is *ModelicaSystem*. This class is equipped with an object constructor of the same name as the class. In addition, the class is equipped with a number of methods for manipulating the instantiated objects.

In this subsection, we discuss how to import the class, and how to use the constructor to instantiate an object.

The object is imported from package *OMPython*, i.e. with Python commands[8]:

```
>>> from OMPython import ModelicaSystem
```

Other Python packages to be used such as `numpy`, `matplotlib`, `pandas`, etc. must be imported in a similar manner.

The object constructor requires a minimum of 2 input arguments which are strings, and may need a third string input argument.

- The *first input argument* must be a string with the file name of the Modelica code, with Modelica file extension `.mo`. If the Modelica file is not in the current directory of Python, then the file path must also be included.
- The *second input argument* must be a string with the name of the Modelica model, including the namespace if the model is wrapped within a Modelica package.
- *A third input argument* is used if the Modelica model builds on other Modelica code, e.g. the Modelica Standard Library.

The result of using the object constructor is a Python object.

Example 1: Use of constructor.

Suppose we have a Modelica model with name *CSTR* wrapped in a Modelica package *Reactors* — stored in file `Reactor.mo`:

```
package Reactors
    // ...
    model CSTR
        /// ...
    end CSTR;
    //
end Reactors;
```

If this model does not use any external Modelica code and the file is located in the current Python directory, the following Python code instantiates a Python object `mod`:

```
>>> mod = ModelicaSystem('Reactors.mo',
 'Reactors.CSTR')
```

The user is free to choose any valid Python label name for the Python object.

All methods of class *ModelicaSystem* refers to the instantiated object, in standard Python fashion. Thus, method `simulate()` is invoked with the Python command:

```
>>> mod.simulate()
```

In the subsequent overview of methods, the object name is not included. In practice, of course, it must be included in order to operate on the object in question.

[6] `www.continuum.io/downloads`

[7] `www.enthought.com/products/canopy`

[8] The Python prompt >>> is not typed, and does not appear in script files, in iPython or in Jupyter notebooks.

Methods may have no input arguments, one, or several input arguments. Methods may or may not return results — if the methods do not return results, the results are stored within the object.

2) Utility routines, converting Modelica ↔ FMU: Two utility methods convert files between Modelica files with file extension `.mo` and Functional Mock-up Unit (FMU) files with file extension `.fmu`.

1) `convertMo2Fmu()` — method for converting the Modelica model of the object, say `ModelName`, into FMU file.
   - Required input arguments: none, operates on the Modelica file associated with the object.
   - Optional input arguments:
     - `className`: string with the class name that should be translated,
     - `version`: string with FMU version, "1.0" or "2.0"; the default is "1.0".
     - `fmuType`: string with FMU type, "me" (model exchange) or "cs" (co-simulation); the default is "me".
     - `fileNamePrefix`: string; the default is \'className\'.
     - `generatedFileName`: string, returns the full path of the generated FMU.
   - Result: file `ModelName.fmu` in the current directory
2) `convertFmu2Mo(s)` — method for converting an FMU file into a Modelica file.
   - Required input arguments: string `s`, where `s` is name of FMU file, including extension `.fmu`.
   - Optional input arguments: a number of optional input arguments, e.g. the possibility to change working directory for the imported FMU files.
   - Result: Assume the name of the file is `fmuName.fmu`. Then file `fmuName_me_FMU.mo` is generated in the current Python directory.

3) Getting and setting information: Quite a few methods are dedicated to getting and setting information about objects. With two exceptions — `getQuantities()` and `getSolutions()` — the *get* methods have identical use of input arguments and results, while all the set methods have identical use of input arguments, with results stored in the object.

Getting quantity information: Method `getQuantities()` does not accept input arguments, and returns a *list of dictionaries*, one dictionary for each quantity. Each dictionary has the following keys — with values being strings, too.

- `Changeable` — value `'true'` or `'false'`,
- `Description` — the string used in Modelica to describe the quantity, e.g. `'Mass in tank, kg'`,
- `Name` — the name of the quantity, e.g. `'T'`, `'der(T)'`, `'n[1]'`, `'mod1.T'`, etc.,
- `Value` — the value of the quantity, e.g. `'None'`, `'5.0'`, etc.,
- `Variability` — `'continuous'`, `'parameter'`.

When applying the *Pandas* method *DataFrame* to the returned list of dictionaries, the result is a conveniently typeset table in Jupyter notebooks. Modelica `constants` are not included in the returned quantities.

Standard get methods: We consider methods `getXXXs()`, where `XXXs` is in {`Continuous`, `Parameters`, `Inputs`, `Outputs`, `SimulationOptions`, `OptimizationOptions`, `LinearizationOptions`}. Thus, methods `getContinuous()`, `getParameters()`, etc.

Two calling possibilities are accepted.

- `getXXXs()`, i.e. without input argument, returns a *dictionary* with names as keys and values as ... values.
- `getXXXs(S)`, where `S` is a *sequence* of strings of names, returns a *tuple* of values for the specified names.

Getting solutions: We consider method `getSolutions()`. Two calling possibilities are accepted.

- `getSolutions()`, i.e. without input arguments, returns a *list of strings of names* of quantities for which there is a *solution = time series*.[9]
- `getSolutions(S)`, where `S` is a *sequence* of strings of names, returns a *tuple* of values = 1D numpy arrays = time series for the specified names.

Setting methods: The information that can be set is a subset of the information that can be set. Thus, we consider methods `setXXXs()`, where `XXXs` is in {`Parameters`, `Inputs`, `SimulationOptions`, `OptimizationOptions`, `LinearizationOptions`}, thus methods `setParameters()`, `setInputs()`, etc. Two calling possibilities are accepted.

- `setXXXs(K)`, with `K` being a sequence of keyword assignments of type `quantity name = value`. Here, the quantity name could be a parameter name (i.e., not a string), an input name, etc.
  - For parameters and simulation/optimization/linearization options, the value should be a numerical value or a string (e.g. a string of ODE solver name such as `'dassl'`, etc.).
  - For inputs, the value could be a numerical value if the input is constant in the time range of the simulation,
  - For inputs, the value could alternatively be a *list* of tuples (t_j, u_j), i.e., $[(t_1, u_1), (t_2, u_2), \ldots, (t_N, u_N)]$ where the input varies linearly between (t_j, u_j) and (t_{j+1}, u_{j+1}), where $t_j \leq t_{j+1}$, and where at most two subsequent time indices t_j, t_{j+1} can have the same value. As an example, `[..., (1,10), (1,20), ...]` describes

[9]The reason why a dictionary with every name as key and time series as values is not returned, is that the amount of data would be exhaustive.

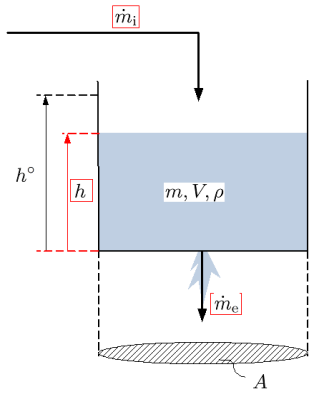

Figure 1. Driven water tank, with externally available quantities framed in red: initial mass is emptied through bottom at rate $\dot{m}_\mathrm{e}$, while at the same time water enters the tank at rate $\dot{m}_\mathrm{i}$.

a perfect jump in input value from value 10 to value 20 at time instance 1.
  - This type of sequence of input arguments does not work for certain quantity names, e.g. `'der(T)'`, `'n[1]'`, `'mod1.T'`, because Python does not allow for label names `der(T)`, `n[1]`, `mod1.T`, etc.
- `setXXXs(**D)`, with `D` being a *dictionary* with quantity names as keywords and values as described with the alternative input argument `K`.

4) Operating on Python object: simulation, optimization: The following methods operate on the object, and have no *input arguments*. The methods have no return values, instead the results are stored within the object.

- `simulate()` — simulates the system with the given simulation options
- `optimize()` — optimizes the Optimica problem with the given optimization options

To retrieve the results, method `getSolutions()` is used as described previously.

5) Operating on Python object: linearization[10]*:* The following methods are proposed for linearization:

- `linearize()` — with no input argument, returns a tuple of 2D numpy arrays (matrices) A, B, C and D.
- `getLinearInputs()` — with no input argument, returns a list of strings of names of inputs used when forming matrices B and D.
- `getLinearOutputs()` — with no input argument, returns a list of strings of names of outputs used when forming matrices C and D.
- `getLinearStates()` — with no input argument, returns a list of strings of names of states used when forming matrices A, B, C and D.

III. Use of API for model analysis

A. Case study: simple tank filled with liquid

We consider the tank in Fig. 1 filled with water.

[10]This part of the API is not completed at the moment, and may change.

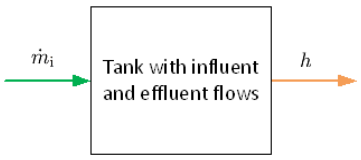

Figure 2. Functional diagram of tank with influent and effluent flow.

Table I
Parameters for driven tank with constant cross sectional area.

Parameter	Value	Unit	Comment
ρ	1	kg / L	Density of liquid
A	5	dm^2	Constant cross sectional area
K	5	kg / s	Valve constant
h^∇	3	dm	Level scaling

Water with initial mass $m(0)$ is emptied by gravity through a hole in the bottom at effluent mass flow rate $\dot{m}_\mathrm{e}$, while at the same time water is filled into the tank at influent mass flow rate $\dot{m}_\mathrm{i}$.

Our *modeling objective* is to find the liquid level h. This objective is illustrated by the *functional diagram* in Fig. 2. The functional diagram depicts the causality of the *system* ("Tank with influent and effluent mass flow"), where *inputs* (green arrow) cause a change in the system and is observed at *outputs* (orange arrow)[11]. Here, the input variable is the influent mass flow rate $\dot{m}_\mathrm{i}$, while the output variable is the quantity we are interested in, h.

B. Model summary

The model can be summarized in a form suitable for implementation in Modelica as

$$\begin{aligned} \frac{dm}{dt} &= \dot{m}_\mathrm{i} - \dot{m}_\mathrm{e} \\ m &= \rho V \\ V &= Ah \\ \dot{m}_\mathrm{e} &= K\sqrt{\frac{h}{h^\nabla}}. \end{aligned}$$

To complete the model description, we need to specify model parameters and operating conditions. Model parameters (constants) are given in Table I.

The operating conditions are given in Table II.

C. Modelica encoding of model

The Modelica code describes the core model of the tank, `ModWaterTank`, and consists of a *first section* where

[11]Although Modelica is an *acausal* modeling language, it is useful to think in terms of causality during model development.

Table II
Operating condition for driven tank with constant cross sectional area.

Quantity	Value	Unit	Comment
$h(0)$	1.5	dm	Initial level
$m(0)$	$\rho h(0) A$	kg	Initial mass
$\dot{m}_\mathrm{i}(t)$	2	kg / s	Nominal influent mass flow rate; may be varied

constants and variables are specified, and a *second section* where the model equations are specified.

```
model ModWaterTank
    // Main driven water tank model
    // author:    Bernt Lie
    //            University College of
    //            Southeast Norway
    //            April 18, 2016
    //
    // Parameters
    constant Real rho = 1 "Density";
    parameter Real A = 5 "Tank area";
    parameter Real K = 5 "Valve const";
    parameter Real h_max = 3 "Scaling";
    // Initial state parameters
    parameter Real h_0 = 1.5
    "Init.level";
    parameter Real m_0 = rho*h_0*A
    "Init.mass";
    // Declaring variables
    // -- states
    Real m(start = m_0, fixed = true)
        "Mass in tank, kg";
    // -- auxiliary variables
    Real V "Tank liquid volume, L";
    Real md_e "Effluent mass flow";
    // -- input variables
    input Real md_i "Influent mass
    flow";
    // -- output variables
    output Real h "Tank liquid level,
    dm";
// Equations constituting the model
equation
    // Differential equation
    der(m) = md_i - md_e;
    // Algebraic equations
    m = rho*V;
    V = A*h;
    md_e = K*sqrt(h/h_max);
end ModWaterTank;
```

As seen from the *first section* of model `ModWaterTank`, the model has 4 essential parameters (`rho-h_max`) of which one is a Modelica *constant* (`rho`) while other 3 are design parameters, compare this to Table I. Furthermore, the model contains 2 "initial state" parameters, where 1 of them can be chosen at liberty, `h_0`, while the other one, `m_0`, is computed automatically from `h_0`, see Table II. The purpose of the "free parameter" `h_0` is that it is easier for the user to specify level than mass. Also, free "initial state" parameters makes it possible for the user to change the initial states from outside of model `ModWaterTank`, e.g., from Python.

Next, one variable is given with initial value — the state `m` — is initialized with the "initial state" parameter `m_0`. Then, 2 variables are defined as auxiliary variables (algebraic variables), `V` and `md_e`.[12]

One input variable is defined — `md_i` — this is the influent mass flow rate $\dot{m}_\mathrm{i}$, see Table II. Inputs are characterized by that their values are not specified in model the core model — here `ModWaterTank`. Instead, their values must be given in an external model/code — we will specify this input in Python. Finally, 1 output is given — `h`.

In the *second section* of model `ModWaterTank`, the Model equations exactly map the mathematical model given in Section III-B.

For illustrative purposes, the core model `ModWaterTank` is wrapped within a package named `WaterTank` and stored in file `WaterTank.mo`,

```
package WaterTank
    // Package for simulating
    //    driven water tank
    // author:    Bernt Lie
    //            University College of
    //            Southeast Norway
    //            April 18, 2016
    //
    model ModWaterTank
        // Main driven water tank model
        // ....
        ....
    end ModWaterTank;
    // End package
end WaterTank;
```

D. Use of Python API

First, the following Python statements are executed — we did this in Jupyter notebook.

```
from OMPython import ModelicaSystem
import numpy as np
import numpy.random as nr
%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
LW = 2
```

Here, we use *NumPy* to handle simulation results, etc. The random number package will be used in a sensitivity/Monte Carlo study. The *magic* function `%matplotlib inline` is used to embed *Matplotlib* plots within the Jupyter notebook; to save these plots into files, simply right-click the plots. However, more options for saving files are available if the magic function is *excluded*, and instead command `plt.show()` is added after the plot commands have been completed. *Pandas* are used to illustrate presenting data in tables in Jupyter notebook. Finally, label `LW` is used to give a conform line width in plots.

E. Basic simulation of model

We instantiate object `tank` with the following command:

[12] `md` is notation for m with a dot, $\dot{m}$, i.e., a mass flow rate.

In [9]: `pd.DataFrame(q)`

Out[9]:

	Changeable	Description	Name	Value	Variability
0	true	Mass in tank, kg	wt.m	None	continuous
1	false	Mass in tank, kg	der(wt.m)	None	continuous
2	false	External input, passed on to instantiated mode...	_md_i	None	continuous
3	false	Tank liquid level, dm	wt.h	None	continuous
4	false	Effluent mass flow rate from tank, kg/s	wt.md_e	None	continuous
5	true	Cross sectional area of tank, dm2	wt.A	5.0	parameter
6	true	Valve constant, kg/s	wt.K	5.0	parameter
7	true	Initial tank level, dm	wt.h_0	1.5	parameter
8	true	Scaling level, dm	wt.h_s	3.0	parameter
9	false	Initial tank mass, kg	wt.m_0	None	parameter
10	false	Tank liquid volume, L	wt.V	None	continuous
11	false	Influent mass flow rate to tank, kg/s	wt.md_i	None	continuous

Figure 3. Typesetting of Data Frame of quantity list in Jupyter notebook.

```
tank = ModelicaSystem('WaterTank.mo',
 'WaterTank.ModWaterTank')
```

whereupon Python/Jupyter notebook responds that the OMC Server is up and running the file. Next, we are interested in which *quantities* are available in the model. In the sequel, Python prompt `>>>` is used when Jupyter notebook actually uses `In[*]` — where `*` is some number, while the response in Jupyter notebook is prepended with `Out[*]`.

```
>>> q = tank.getQuantities()
>>> type(q)
list
>>> len(q)
11
>>> q[0]
{'Changeable': 'true',
'Description': 'Mass in tank, kg',
'Name': 'm',
'Value': None,
'Variability': 'continuous'}
>>> pd.DataFrame(q)
```

The last command leads Jupyter notebook to typeset a tabular presentation of the quantities, Fig. 3. The results in Fig. 3 should be compared to the Modelica model in Section III-C. Observe that Modelica *constants* are not included in the quantity list.

Next, we check the simulation options:

```
>>> tank.getSimulationOptions()
{'solver': 'dassl',
'startTime': 0.0,
'stepSize': 0.002,
'stopTime': 1.0,
'tolerance': 1e-06}
```

It should be observed that the *stepSize* is the frequency at which solutions are *stored*, and is not the step size of the solver. The number of data points stored, is thus `(stopTime-startTime)/stepSize` with due rounding. This means that if we increase the *stopTime* to a large number, we should also increase the *stepSize* to avoid storing a large number of information.

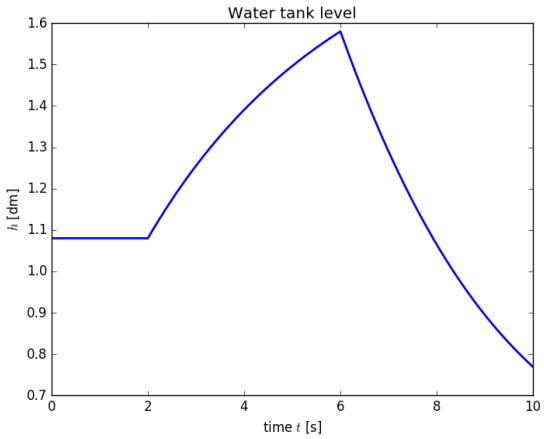

Figure 4. Tank level when starting from steady state, and $\dot{m}_{\mathrm{i}}(t)$ varies in a straight line between the points $(t_j, \dot{m}_{\mathrm{i}}(t_j))$ given by the list $[(0,3),(2,3),(2,4),(6,4),(6,2),(10,2)]$.

To this end, we want to simulate the system for a long time, until the level reaches steady state. Possible inputs are:

```
>>> tank.getInputs()
{'md_i': None}
```

where value `None` implies that the available input, `md_i`, has yet not been set. We could use `None` as input, which will be interpreted as zero. But let us instead set $\dot{m}_{\mathrm{i}} = 3$, simulate for a long time, and change "initial state" parameter $h(0)$ to the steady state value of h:

```
>>> tank.setInputs(md_i=3)
>>> tank.setSimulationOptions\
    (stopTime=1e4, stepSize=10)
>>> tank.simulate()
>>> h = tank.getSolutions('h')
>>> tank.setParameters(h_0 = h[-1])
```

Next, we set back to stop time to 10, and specify an input sequence with a couple of jumps:

```
>>> tank.setSimulationOptions\
    (stopTime=10, stepSize=0.02)
>>> tank.setInputs(md_i = [(0,3),(2,3),
(2,4),(6,4),(6,2),(10,2)])
```

Finally, we simulate the model with the time varying input, and plot the result:

```
>>> tank.simulate()
>>> tm, h = tank.getSolutions('time',\
    'h')
>>> plt.plot(tm,h,linewidth=LW,
color='blue', label=r'$h$')
>>> plt.title('Water tank level')
>>> plt.xlabel(r'time $t$ [s]')
>>> plt.ylabel(r'$h$ [dm]')
```

The result is displayed in Fig. 4.

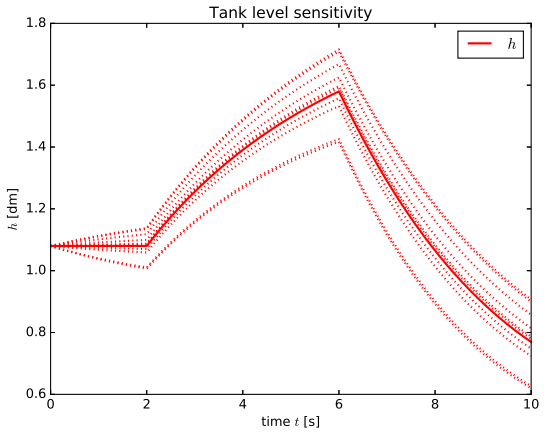

Figure 5. Uncertainty in tank level with a 5% uncertainty in valve constant K. The input is like in Fig. 4.

F. Parameter sensitivity/Monte Carlo simulation

It is of interest to study how the model behavior varies with varying uncertain parameter values, e.g. the effluent valve constant K. This can be done as follows:

```
>>> par = tank.getParameters()
>>> K = par['K']
>>> KK = K + (nr.randn(10)-0.5)*K/20
>>> tank.simulate()
>>> tm, h = tank.getSolutions('time',\
    'h')
>>> plt.plot(tm,h,linewidth = LW,
color = 'red', label=r'$h$')
>>> for k in KK:
        tank.setParameters(K=k);
        tank.simulate()
        tm, h = tank.getSolutions\
('time','h')
        plt.plot(tm,h,linewidth=LW,
           color='red',linestyle=\
'dotted',label='_nolabel_')
>>> plt.title('Tank level sensitivity')
>>> plt.xlabel(r'time $t$ [s]')
>>> plt.ylabel(r'$h$ [dm]')
>>> plt.legend()
```

The result is as shown in Fig. 5.

IV. Discussion and conclusions

This paper introduces some ongoing work on extending OpenModelica with a Python API, so that Modelica models can be run and analyzed from within Python. The new Python API is briefly described, and the use of this API is then illustrated by simulating a very simple model of a water tank.

Future work will include further testing, e.g., with optimization, extending the API so that it works on more platforms, and extending the API to include analytic model linearization.

References

[1] Bajracharya, S. (2016). *Enhanced OpenModelica Python Interface*. MSc thesis, Department of Computer and Information Science, Linköping University, Sweden.

[2] Brenan, K.E., Campbell, S.L., Petzold, L.R. (1987). *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations, 2nd edition*. SIAM, Philadelphia.

[3] Fritzson, P. (2014). *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach, second edition.* Wiley-IEEE Press, Piscataway, NJ. ISBN 978-1-118-85912-4.

[4] Fritzson, P., Aronsson, P., Pop, A., Lundvall, H., Nyström, K., Saldamli, L., Broman, D., Sandholm, A. (2006). "OpenModelica – A Free Open-Source Environment for System Modeling, Simulation, and Teaching". *Proceedings of the 2006 IEEE Conference on Computer Aided Control System Design*, Munich, Germany, October 4–6, 2006.

[5] Ganeson, A.K. (2012). *Design and Implementation of a User Friendly OpenModelica - Python interface*. Master's Thesis, Department of Computer and Information Science, Linköping University. Reg Nr: LIU-IDA/LITH-EX-A12/037SE .

[6] Ganeson, A.K., Fritzson, F., Rogovchenko, O., Asghar, A., Sjölund, M., Pfeiffer, A. (2012). "An OpenModelica Python Interface and its use in PySimulator ". *Proceedings of the 9th International Modelica Conference*, September 3-5, 2012, Munich, Germany. DOI 10.3384/ecp12076537.

[7] Perera, M. Anushka S., Lie, B., and Pfeiffer, C.F. (2015). "Structural Observability Analysis of Large Scale Systems Using Modelica and Python". *Modeling, Identification and Control*, **Vol 36**, No 1, pp. 53–65.

[8] Perera, M. Anushka S., Hauge, T.A., and Pfeiffer, C.F. (2015). "Parameter and State Estimation of Large-Scale Complex Systems Using Python Tools". *Modeling, Identification and Control*, **Vol 36**, No 3, pp. 189–198.

[9] Pfeiffer, A., Hellerer, M., Hartweg, S., Otter, M., and Reiner, M. (2012). "PySimulator – A Simulation and Analysis Environment in Python with Plugin Infrastructure". *Proceedings of the 9th International Modelica Conference*, September 3-5, 2012, Munich, Germany. DOI 10.3384/ecp12076523.