MAKING MODELICA MODELS AVAILABLE FOR ANALYSIS IN PYTHON CONTROL SYSTEMS LIBRARY

Anushka Perera, Carlos Pfeiffer and Bernt Lie*
Telemark University College
Kjølnes ring 56, P.O. Box 203
N-3901 Porsgrunn
Norway

Tor Anders Hauge
Glencore Nikkelverk
Kristiansand
Norway

ABSTRACT

Modelica-based simulation environments are primarily targeted on model simulation, therefore they generally lack support for advanced analysis and synthesis needed for general control systems design and particularly for optimal control problems (OCPs), although a Modelica language extension (Optimica) exists to support general optimization problems. On the other hand, MATLAB has a rich set of control analysis and synthesis tools based on linear models. Similarly, Python has increasing support for such tools e.g. the "Python Control System Library" developed in Caltech. In this paper, we consider the possibility of automating the process of extracting linear approximations of Modelica models, and exporting these models to a tool with good support for linear analysis and design. The cost of software is an important aspect in our development. Two widly used free Modelica tools are OpenModelica and JModelica.org. Python is also freely available, and is thus a suitable tool for analysis and design in combination with the "Python Control System Library" package. In this work we choose to use JModelica.org as the Modelica tool because of its better integration with Python and CasADi, a CAS (Computer Algebra System) tool that can be used to linearize Modelica models. The methods that we discuss can in principle also be adapted for other Modelica tools. In this paper we present methods for automatically extracting a linear approximation of a dynamic model encoded in Modelica, evaluated at a given operating point, and making this linear approximation available in Python. The developed methods are illustrated by linearizing the dynamic model of a four tank level system, and showing examples of analysis and design based on the linear model. The industrial application of these methods to the Copper production plant at Glencore Nikkelverk AS, Kristiansand, Norway, is also discussed as current work.

Keywords: Modelica, JModelica.org, Python, CasADi, Symbolic/Numeric Linearization, Linear Analysis, python-control

1 INTRODUCTION

Modelica is becoming a de facto standard for modeling of large-scale complex physical systems. Since Modelica is object-oriented, declarative (acausal), and equation-based, it allows to create reusable components and to built efficient reconfigurable component-models. A Modelica-based simulation environment (a Modelica tool) is needed to simulate Modelica models.[1] Modelica tools mainly focus on model simulation. However, the model sim-

*Corresponding author: Phone: +47 41807744 E-mail: bernt.lie@hit.no

[1] A complete list of Modelica tools is available at https://www.modelica.org/tools.

ulation is not the only objective of mathematical modeling. Among others optimal control problems (OCPs), control analysis and synthesis, and state estimations are several aspects that require dynamic systems modeling. Optimica [1] extends Modelica language specifications to handle OCPs and JModelica.org provides Optimica compilers. OpenModelica partially support Optimica extension at the moment.

In order to exploit Modelica, either a simulation environment should be equipped with necessary tools for model analysis (e.g. good enough scripting facilities and/or GUI options) or Modelica should be interfaced with other existing tools. Some comercial Modelica tools provide interfaces to integrate toexternal software, such as Dymola integrating with MATLAB/Simulink. However, these tools are very expensive. A free tool, the JModelica.org platform, integrates completely with Python through two core Python packages: `pymodelica` for compilers and `pyfmi`/`pyjmi` for model import.[2]

Since we are mainly interested in free software tools that easily interface with other tools, we selected JModelica.org. The JModelica.org platform has also an interface with CasADi [2] and hence, it is possible to make Modelica/Optimica models available as symbolic model objects in Python. The `casadi` Python package is used to linearize Modelica models symbolically/numerically (see [3] for a detailed description.) and then the system matrices may be used in linear system analysis and in algorithms, in particular using the `python-control` package.[3]

This paper demonstrates usefulness of interfacing Modelica with Python via CasADi. The method is explained with a simple example (a four tanks system). As a case study of a real process the Copper production plant [4] at Glencore Nikkelverk AS, Kristiansand, Norway is considered by showing how to design a LQR (Linear Quadratic Regulator) optimal state feedback controller using the `python-control` package.

2 STRUCTURE OF LINEARIZATION

2.1 MODELICA AND DAES

The execution of a Modelica model is started with a model flattening process that removes the hierarchical structure (i.e. expansion of inherited base classes, adding connector equations, etc.) of the Modelica model into a flat model [5]. A flattened model provides a set of acausal differential-algebraic-discrete equations, or so called hybrid DAEs form, which is given by

$$F\left(t,\dot{x},x,u,z,m,p\right)=\left[\begin{array}{c} F_1\left(t,\dot{x},x,u,z,m,p\right) \\ F_2\left(t,\dot{x},x,u,z,m,p\right) \\ \cdots \\ F_m\left(t,\dot{x},x,u,z,m,p\right) \end{array}\right]=0, \tag{1}$$

where x, u, z, m, p, and t are respectively, the dynamic state vector, the input vector, the algebraic state vector, the piece-wise constant vector, the parameter vector, and time. The keyword `input` is used to define input variables and `output`[4] for defining output variables. Output variables are also algebraic variables, hence they are included in z. An output vector, y, may be expressed as:

$$y=H\left(t,x,u,z,m,p\right). \tag{2}$$

For simplicity and notational convenience, m and p are neglected and thereby we have:

$$F\left(t,\dot{x},x,u,z\right)=\left[\begin{array}{c} F_1\left(t,\dot{x},x,u,z\right) \\ F_2\left(t,\dot{x},x,u,z\right) \\ \cdots \\ F_m\left(t,\dot{x},x,u,z\right) \end{array}\right]=0. \tag{3}$$

Where, $m=dim(x)+dim(y)$. A flattened Modelica model is not yet ready to be solved for $\dot{x}$ and z. A complicated set of manipulations are done on flattened models: sorting equations $(F_1,F_2,\ldots,F_m)$, index reduction, common subexpression elimination, etc. prior to solving the equation 3 [5][7].

2.2 CONVERSION TO EXPLICIT STATE SPACE FORM

Consider the DAEs in the equation 3. Converting DAEs into explicit ODEs may be required in many

[2] See the JModelica.org user guide available at `http://www.jmodelica.org`.

[3] `http://www.cds.caltech.edu/~murray/wiki/Control_Systems_Library_for_Python`.

[4] The variables which are prefixed with `input`/`output` keywords within the Modelica components at the highest hierarchy of a component-model are appeared as input/output variables after flattening.

applications or to use most standard ODE solvers. If $\frac{\partial F}{\partial [\dot{x},z]^T}$ is not singular (a necessary condition for implicit to explicit transformation), then $[\dot{x},z]^T$ can be written as continuous functions of t, x, and u.[5] On the other hand, if $\frac{\partial F}{\partial [\dot{x},z]^T}$ is singular, then implicit to explicit transformation may not be possible. Algebraic constraint among t, x, z and u can make $\frac{\partial F}{\partial [\dot{x},z]^T}$ singular and in such situations, the constraint equations are differentiated with respect to time, t.

Theorem 1 *The index of a DAE, $F(t,\dot{x},x,u,z)=0$, is the minimum number of times that all or part of the DAE must be differentiated with respect to t in order to determine $[\dot{x},\dot{z}]^T$ as a continuous function of x, z, u, and t [8].*

The definition to the index of a system of DAEs is given in theorem 1. Higher index (i.e. index > 1) problems may be reduced into at most index 1 problems systematically using the Pantelides algorithm [9]. For simplicity, the DAEs

$$f(t,\dot{x},x,u,z)=0 \tag{4}$$

and

$$g(t,x,u,z)=0 \tag{5}$$

are considered in the following discussion (a special case of the equation 3). Sometimes, it may be possible to express algebraic state variables ($\in z$), explicitly in terms of t, x and u and in such cases the index of the problem is said to be 0. By differentiating the equation 5, we get:

$$\frac{\partial g}{\partial t}+\frac{\partial g}{\partial x}\cdot\dot{x}+\frac{\partial g}{\partial u}\cdot\dot{u}+\frac{\partial g}{\partial z}\cdot\dot{z}=0. \tag{6}$$

If $\frac{\partial g}{\partial z}$ is not singular, then the equation 6 is used to find $\dot{z}$ and hence, the initial problem (equations 4 and 5) is said to be an index 1 problem and equations 4 and 6 gives an index 0 problem. The equation 4, in the general case, gives implicit ODEs, however often they appear as explicit ODEs (i.e. $\dot{x}=f(t,x,u,z)$). If $\frac{\partial g}{\partial z}$ is singular, it means there are algebraic dependencies among t, x, and u. In this case the algebraic constraints in equation 6 are differentiated once more and if this gives a possibility to find $\dot{z}$, then the initial system of DAEs is an index 2 problem. Constraint equations are differentiated, as many times as the index of the initial problem, until an index 0 problem is obtained. Note that a reduced index 0 (or 1) problem may not necessarily give the solution to the initial high index problem, unless consistent initial conditions are given [10].
After reducing the index and BLT sorting[6], we have a causal system of DAEs,

$$\tilde{f}\left(t,\dot{\tilde{x}},\tilde{x},\tilde{u},\tilde{z}\right)=0, \tag{7}$$

and

$$\tilde{g}(t,\tilde{x},\tilde{u},\tilde{z})=0 \tag{8}$$

with index 1. $\dot{\tilde{x}}$ is the new dynamic state vector of the reduced problem and $\tilde{u}=\left[u,\frac{du}{dt},\frac{d^2u}{dt^2},\ldots\right]$. The index reduction process may result in adding additional variables and those variables are stacked in $\tilde{z}$. For examples, the dummy derivatives, the state variables which has become algebraic, etc [11]. As $\frac{\partial\tilde{g}}{\partial\tilde{z}}$ is not singular and thereby, it is thus possible to explicitly find $\dot{\tilde{z}}$ (if needed) by:

$$\dot{\tilde{z}}=-\left(\frac{\partial\tilde{g}}{\partial\tilde{z}}\right)^{-1}\cdot\left[\frac{\partial\tilde{g}}{\partial\tilde{t}}+\frac{\partial\tilde{g}}{\partial\tilde{x}}\cdot\dot{\tilde{x}}+\frac{\partial\tilde{g}}{\partial\tilde{u}}\cdot\dot{\tilde{u}}\right]. \tag{9}$$

Consistent initialization gives the solution to the equations 7 and 9 identical to the initial higher index problem in the equations 4 and 5.

2.3 LINEARIZATION

Suppose that $\left(t_0,\dot{\tilde{x}}_0,\tilde{x}_0,\tilde{u}_0,\tilde{z}_0\right)$ exists such that $\tilde{f}\left(t_0,\dot{\tilde{x}}_0,\tilde{x}_0,\tilde{u}_0,\tilde{z}_0\right)=0$ and $\tilde{g}(t_0,\tilde{x}_0,\tilde{u}_0,\tilde{z}_0)=0$, then $\left(t_0,\dot{\tilde{x}}_0,\tilde{x}_0,\tilde{u}_0,\tilde{z}_0\right)$ is an operating point. In many cases, it is required to find the linear approximation for given nonlinear model with respect to an operating point. The linear approximation to equations 7 and 8 are given by:

$$\frac{\partial\tilde{f}}{\partial\tilde{t}}+\frac{\partial\tilde{f}}{\partial\dot{\tilde{x}}}\cdot\delta\dot{\tilde{X}}+\frac{\partial\tilde{f}}{\partial\tilde{x}}\cdot\delta\tilde{X}+\frac{\partial\tilde{f}}{\partial\tilde{u}}\cdot\delta\dot{\tilde{U}}+\frac{\partial\tilde{f}}{\partial\tilde{z}}\cdot\delta\dot{\tilde{Z}}=0 \tag{10}$$

and

$$\frac{\partial\tilde{g}}{\partial\tilde{t}}+\frac{\partial\tilde{g}}{\partial\tilde{x}}\cdot\delta\tilde{X}+\frac{\partial\tilde{g}}{\partial\tilde{u}}\cdot\delta\dot{\tilde{U}}+\frac{\partial\tilde{g}}{\partial\tilde{z}}\cdot\delta\dot{\tilde{Z}}=0. \tag{11}$$

Jacobian matrices are evaluated at $\left(t_0,\dot{\tilde{x}}_0,\tilde{x}_0,\tilde{u}_0,\tilde{z}_0\right)$. $\frac{\partial\tilde{f}}{\partial\dot{\tilde{x}}}$ and $\frac{\partial\tilde{g}}{\partial\tilde{z}}$ are not singular and as a result equations 10 and 11 can be transformed into a state space form.
The usual procedure to obtain numerical Jacobian

[5] The implicit function theorem.

[6] BLT stands for Block-Lower-Triangular.

matrices is to use finite difference methods. For example, a finite difference approximation to $\frac{\partial \tilde{f}}{\partial \dot{\tilde{x}}}$ using the central difference method is

$$\frac{\tilde{f}\left(t,\dot{\tilde{x}}+I_{dim(x)}\cdot h,\tilde{x},\tilde{u},\tilde{z}\right)-\tilde{f}\left(t,\dot{\tilde{x}}-I_{dim(x)}\cdot h,\tilde{x},\tilde{u},\tilde{z}\right)}{2\cdot h^2}. \tag{12}$$

Where h is a small-enough positive number and $I_{dim(x)}$ is a $dim(x)$-by-$dim(x)$ unit matrix. There are several drawbacks in finite difference methods: truncation errors, choosing h is harder, and the results depends on h. In order to avoid such problems, automatic/algorithmic differentiation (AD) techniques can be used, where derivatives are calculated as accurate as up to the working precision of a given computer. AD techniques are used to evaluate derivatives of functions defined by means of a high-level programming language such as Python/C++/etc.[7] The AD is implemented with the help of a computer algebra system (CAS) tool, which provides symbolic manipulations over mathematical expressions. A CAS tool is used to create symbolic variables, matrices, expressions, functions and do symbolic mathematical manipulations on them such as symbolic differentiation[8], integration, etc. There are many CAS tools available such as Maple, Mathematica, SymPy, CasADi, Maxima, etc.[9] A CAS tool may or many not support AD. For example the `sympy` python package doesn't support AD while Maple does. If $\tilde{f}$ and $\tilde{g}$ in equations 7 and 8 can be symbolically expressed using a CAS tool which support AD, then the Jacobian matrices in equations 10 and 11 can be evaluated efficiently using AD techniques.

[7]Consider a function $\bar{f}$ such that $\bar{y}=\bar{f}(\bar{x})$, where $\bar{x}=[\bar{x}_1,\bar{x}_2,\ldots,\bar{x}_n]$ and $\bar{y}=[\bar{y}_1,\bar{y}_2,\ldots,\bar{y}_m]$. $\bar{x}$ and $\bar{y}$ are the independent and the dependent variable vectors respectively. Often, it is possible to represent y_i-x_j relationships using elementary unary/binary operations (+, -, etc.) and elementary functions (sin, cos, etc.). Let $\bar{y}_1=\bar{x}_1\cdot e^{\bar{x}_1\cdot\bar{x}_2}$. $\bar{y}_1$ can be expressed in terms of basic functions and unary/binary operators using a set of intermediate variables ($\bar{z}_k$'s): $\bar{z}_0=\bar{x}_1$, $\bar{z}_{-1}=\bar{x}_2$, $\bar{z}_1=\bar{z}_0\cdot\bar{z}_{-1}$, $\bar{z}_2=e^{\bar{z}_1}$, $\bar{z}_3=\bar{z}_0\cdot\bar{z}_2$, $\bar{z}_4=\bar{z}_3$, and $\bar{y}_1=\bar{z}_4$. $\bar{z}_k$ can be written as

$$\bar{z}_k=\bar{f}_e^k(\bar{z}_i), \tag{13}$$

where $i<k$ and $\bar{f}_e^k$ contains elementary functions and operations. Now, for example $\frac{\partial\bar{y}_1}{\partial\bar{x}_1}=\frac{\partial\bar{z}_4}{\partial\bar{z}_0}=\frac{\partial\bar{f}_e^4}{\partial\bar{z}_0}$ is given, by applying chain-rule, by

$$\frac{\partial\bar{z}_4}{\partial\bar{z}_0}=\Sigma_{i=1}^{4-1}\frac{\bar{f}_e^4(\bar{z}_i)}{\partial\bar{z}_i}\cdot\frac{\partial\bar{z}_i}{\partial\bar{z}_0}. \tag{14}$$

$\frac{\bar{f}_e^4(\bar{z}_i)}{\partial\bar{z}_i}$ is known as $\bar{f}_e^4$ contains known elementary functions. In order to find $\frac{\partial\bar{z}_i}{\partial\bar{z}_0}$, the equation 14 is applied again and so on. The derivative evaluation may be done in one of two modes: forward and reverse. The method just mentioned above is the forward mode. For further details, refer [12].

[8]Note that symbolic differentiation is not AD.

[9]`http://www.autodiff.org/` gives a list of available AD tools.

2.4 JMODELICA.ORG OPTIONS

There are several ways of creating Modelica/Optimica model objects, so called model export, in JModelica.org: FMU, JMU, and FMUX.[10] FMUs are based on FMI (Functional Mock-up Interface) standards[11] and all others are JModelica.org platform specific. The `pymodelica` package contains compilers for compiling Modelica/Optimica models into FMUs, JMUs, and FMUXes. But FMU-export doesn't support Optimica. FMUXes are crucial here because in order to work with symbolic DAEs, FMUX model units should be used and the relevant compiler is `compile_fmux`. A compiled model is stuffed in a zip file (with the file extension '.fmux') and the modelDescription.xml file is contained in it. modelDescription.xml file gives a flat model description of Modelica/Optimica models. JMUs closely follow FMI standards. zip files of both FMUs/JMUs provide a compiled C-codes and binaries besides modelDescription.xml files while in FMUXes only the modelDescription.xml file is given. The model import (loading FMU/JMU/FMUX model objects into Python) may be done via two Python packages: PyFMI and PyJMI. PyFMI is for FMUs while PyJMI for JMUs/FMUXes.

CasADi is a symbolic framework for AD and non-linear optimization as well as it is a CAS tool. CasADi can import Modelica/Optimica models, where those models have been transformed into compatible XML-files (modelDescription.xml) [13][14] and generates symbolic DAEs/OCPs. the `parseFMI()` method which is defined within the CasADi class `SymbolicOCP` is used to import XML-based Modelica/Optimica models. See [15] for more details. CasADi integra-

[10]The latest JModelica.org version 1.14 has introduced a new model class using the compiler `transfer_optimization_problem`, which is available in `pyjmi` package. See the user guide for further details. In this paper JModelica.org version 1.12 is considered.

[11]`https://www.fmi-standard.org/`.

tion with JModelica.org [16] opens up a provision to use Modelica/Optimica models with complete flexibility within Python, and making it possible to exploit modeling power in Modelica as well as scripting power in Python.

3 A PYTHON IMPLEMENTATION WITH AN EXAMPLE

3.1 STRUCTURE OF PYTHON SCRIPT

A simple four-tank system is considered (taken from [17]). See Figure 1 for the schematic model of the system. The mathematical model is given by equations 15 - 20. The table 1 contains parameters. The Optimica model is stored in a text file named `TankSystems` with the file extension `.mop` (in this case, the file extension may have been used to be `.mo`). `TankSystems.mop` contains `TankSystems` package and this package contained two Modelica models: `FourTanks` for the dynamic model and `FourTanks_init` for the steady state model in order to find steady state. See appendix A for the Optimica code.

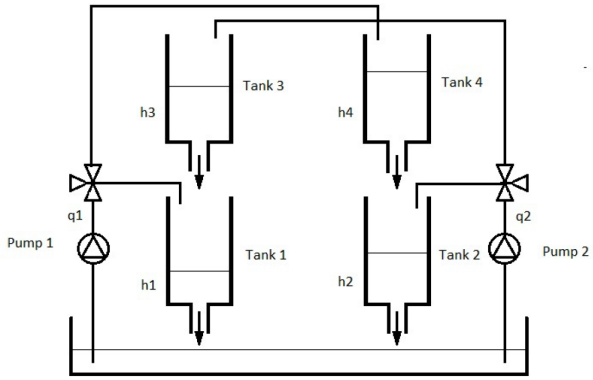

Figure 1: A schematic diagram for the four tank systems.

$$\frac{dh_1(t)}{dt} = -\frac{c_1 \cdot \sqrt{h_1(t)}}{A_1} + \frac{c_3 \cdot \sqrt{h_3(t)}}{A_1} + \frac{\gamma_1 \cdot q_1(t)}{A_1} \quad (15)$$

$$\frac{dh_2(t)}{dt} = -\frac{c_2 \cdot \sqrt{h_2(t)}}{A_2} + \frac{c_4 \cdot \sqrt{h_4(t)}}{A_2} + \frac{\gamma_2 \cdot q_2(t)}{A_2} \quad (16)$$

$$\frac{dh_3(t)}{dt} = -\frac{c_3 \cdot \sqrt{h_3(t)}}{A_3} + \frac{(1-\gamma_2) \cdot q_2(t)}{A_3} \quad (17)$$

$$\frac{dh_4(t)}{dt} = -\frac{c_4 \cdot \sqrt{h_4(t)}}{A_4} + \frac{(1-\gamma_1) \cdot q_1(t)}{A_4} \quad (18)$$

$$\frac{dq_2(t)}{dt} = -\frac{1}{\tau_2} \cdot q_2(t) + \frac{k_2}{\tau_2} \cdot v_2(t) \quad (19)$$

$$\frac{dq_1(t)}{dt} = -\frac{1}{\tau_1} \cdot q_1(t) + \frac{k_1}{\tau_1} \cdot v_1(t) \quad (20)$$

Variable	Value	Units
A_1	12.57	cm^2
A_2	12.57	cm^2
A_3	12.57	cm^2
A_4	12.57	cm^2
c_1	9.82	$c \cdot m^{5/2}/s$
c_2	5.76	$c \cdot m^{5/2}/s$
c_3	9.02	$c \cdot m^{5/2}/s$
c_4	8.71	$c \cdot m^{5/2}/s$
K_1	6.94	$c \cdot m^3/V$
K_2	8.72	$c \cdot m^3/V$
τ_1	6.15	s
τ_2	13.2	s

Table 1: Four-Tanks System Model Parameters.

The Python code used to compile the Modelica/Optimica Four Tanks is given below.

```
# Import compiler compile_fmux
from pymodelica import compile_fmux
# Compile Modelica/Optimica models
file_name = 'TankSystems.mop'
model_name = 'FourTanks'
compile_fmux(model_name,file_name)
# Note: name of the '.zip' file created is
\'FourTanks.fmux'
```

Now, the FMUX model object is imported as a `CasadiModel` object. See below:

```
from pyjmi import CasadiModel
casadiModelObject = CasadiModel('FourTanks.fmux')
# Get flat ocp representation
ocp = casadiModelObject.ocp
```

`ocp` gives a flat representation of Modelica/Optimica models based on the modelDescription.xml. `ocp.ode` and `ocp.alg` represent symbolic expressions for ordinary differential equations (ODEs) and algebraic equations respectively. Use Python commands `print ocp` and `help(ocp)` to get help. Now, the symbolic DAEs are available for general use in Python,[12] hence Modelica/Optimica models can be used in various

[12]For example, it is possible to implement Pantelides algorithm with symbolic DAEs.

algorithms and in analysis using CasADi functionalities, `numpy`[13], `matplotlib`[14], `scipy`[15] and `python-control` like Python packages. Use the following Python code to import CasADi and CasADi tools.

```
from casadi import *
from casadi.tools import *
```

If necessary, `ocp.makeExplicit()` method can be used to transform ODEs from implicit to explicit form.[16] Derivatives ($\in \dot{x}$), dynamic states ($\in x$), algebraic states ($\in z$), independent parameters ($\in p_i$), dependent parameters ($\in p_d$), free parameters ($\in p_f$), time (t), and control signals ($\in u$) are given by respectively `casadiModelObject.dx`, `ocp.x`, `ocp.z`, `ocp.pi`, `ocp.pd`, `ocp.pf`, `ocp.t`, and `ocp.u`. For example, `ocp.x[i]` gives x_{i+1} ($0 \leq i \leq dim(x)-1$). `ocp.x[i]` is in `variable` data type, and it has to be converted into `SX` data type before creating `SXFunction` instances. This is done by `ocp.x[i].var()` [3]. Then all the states variables (in SX type) are stuffed in a Python `list`. The same procedure is applied to other variables as well. Using `ocp.eliminateDependent()`, dependent parameters are eliminated. `ocp.ode` can be taken as a function of t, $\dot{x}$, x, z, and u and let it be $0 = f(t,\dot{x},x,u,z)$. Now, f is defined as an `SXFunction` class instance. Say, `ffun`. See below for the Python code to create it (see appendix B for the complete Python script):

```
# Define DAEs
f = ocp.ode
g = ocp.alg
# Create an SXFunction for f and g
ffun = SXFunction([t,vertcat(xDot),vertcat(x),\
vertcat(u),vertcat(z)],[f])
gfun= SXFunction([vertcat(x),vertcat(u),
vertcat(z)],[g])
ffun.init()
gfun.init()
```

Note that as explained in subsection 2.2, index reduction should be done on f and g, if the problem is higher index, to obtain lower index problems before creating `ffun` and `gfun`. Anyway, the four-tank system model has the index equal to 0. For an example, $\frac{\partial f}{\partial u}$ is given by `ffun.jac(2)`. See the result (by entering `ffun.jac(2)` in the command line) given below.

```
Matrix<SX>(
[[00,  00 ]
 [00,  00 ]
 [00,  00 ]
 [00,  00 ]
 [-1.12846,  00]
 [00,  -0.660606]]
)
```

Numerical Jacobian matrices are then found for given $(t_0,\dot{x}_0,x_0,u_0,z_0)$. See the code given below.

```
f_u_fun.setInput(t0,0)
f_u_fun.setInput(dx0,1)
f_u_fun.setInput(x0,2)
f_u_fun.setInput(u0,3)
f_u_fun.setInput(z0,4)
f_u_fun.evaluate()

f_u_num = f_u_fun.getOutput()
```

Operating points are usually choosen at steady states. A steady state, x_0 is calculated by: (1) compiling the static Modelica model TankSystems.FourTanks_Init into a JMU, (2) loading the JMU model, (3) setting the input vector u_0, and (4) finally, initializing the JMU model using `initialize()` method.[17] Hence, it is possible to find system matrices A, B, C, and D based on the Jacobian matrices just evaluated.
As the system matrices are available, the `python-control` package can be used in control analysis and synthesis. In order to import the `python-control` use:

```
import control
```

or

```
from control import *
```

As a summery to this section, the following points are made: (1) an Optimica package is created with two Modelica models (dynamic and static) in it, (2) use the static model to find the steady state using a JMU model object, (3) import the dynamic model as a CasadiModel object model and use `casadi` to linearization of symbolic DAEs (after reducing the index if needed), and (4) use the linearized model with the `python-control` package. See TankSystem.mop and TankSystem.py in the appendices A and B.

[13] `http://www.numpy.org/`.
[14] `http://matplotlib.org/`.
[15] `http://www.scipy.org/`.
[16] This is possible only if ODEs are linear w.r.t. $\dot{x}$.
[17] Initialization is done by formulating DAEs and equations given within `initial equation` clause in residual form and minimizing sum of square error using the `Ipopt` solver. See JModelica.org user guide.

4 INDUSTRIAL CASE STUDY

We consider the chlorine leaching and electro-winning process which is a part of the nickel refinery of Glencore Nikkelverk in Kristiansand, Norway. A mechanistic models is presented in [4] and it is a MIMO system with 3 inputs (u_1, u_2, u_3), 11 disturbances (w_1, w_2, ..., w_{11}), 3 outputs (y_1, y_2, y_3) and 39 states (x_1, x_2, ..., x_{39}). The process is in large-scale and it is complex (multi variable nature, non-linearities, etc.). Hence, Modelica and Optimica are ideal for the modeling and optimization. Also the process is a good candidate for model based control. In this section, what is explained in subsection 3.1 will be applied to the copper plant model.

The following demonstrations shows how to use the `python-control` tool to design a (infinite-horizon, continuous-time) LQR state feedback controller[18] for the linearized Copper plant model. The linearized model is given by

$$\dot{\delta x} = A \cdot \delta x + B \cdot \delta u, \tag{21}$$

and

$$\delta y = C \cdot \delta x + D \cdot \delta u, \tag{22}$$

where $\dot{\delta x}$, δx, δu, and δy are deviation variables with respect to a steady state point, x_0. Thus, $\dot{\delta x}_0 = 0$, $\delta x_0 = 0$, $\delta u_0 = 0$ and $\delta y_0 = 0$. The procedure to find A, B, C and D as well as x_0 is already given (see subsection 3.1). Use the following script to create a state space model (`sys`) object and optionally, the state space model may be transformed into transfer function form (`sys2`).

```
#Import python-control package
import control as ctrl
#Create state space model
sys = ctrl.ss(A,B,C,D)
#If needed, state space==>transfer function
sys2 = ctrl.ss2tf(sys)
```

The `ctrl.lqr()` method calculates the optimal feedback controller, $\delta u = -K \cdot \delta x$, such that minimizing the cost function J:

$$J = \int_0^\infty \left(\delta x^T \cdot Q \cdot \delta x + \delta u^T \cdot R \cdot \delta u + 2 \cdot \delta x^T \cdot N \cdot \delta u\right) \cdot dt. \tag{23}$$

[18]This paper mainly focus on demonstrating the idea of making Modelica models available in Python in general and in particular using the `python-control` package. Therefore, a detailed theoretical discussion about LQR state feedback controllers is not given here. For more details about LQR state feedback controllers refer [6].

Let, N is a zero matrix. K is the state feedback gain matrix and it is given by $K = R^{-1} \cdot B^T \cdot S$. S is found by solving the algebraic Riccati equation (ARE)

$$A^T \cdot S + S \cdot A^T - S \cdot B \cdot R^{-1} \cdot B^T \cdot S + Q = 0. \tag{24}$$

Use `K,S,E=ctrl.lqr(sys,Q,R,N)` finds K, S, and E. E gives Eigenvalues of the closed loop system. Now, the closed loop system is given by

$$\dot{\delta x} = (A - B \cdot K) \cdot \delta x, \tag{25}$$

and

$$\delta y = (C - D \cdot K) \cdot \delta x. \tag{26}$$

The closed loop system is simulated for a small perturbation in δx, say 0.01. Q and R are positive definite matrices and are used as the tunning parameters. A possibility is to set Q to be a unit matrix while R is a diagonal matrix and its elements are used in tunning. See the code given below and the results are given in figure 2.

```
Q = 1.0*np.eye(n_x,n_x)
R = 0.001*np.eye(n_u,n_u)
N = np.eye(n_x,n_u)
K,S,E=ctrl.lqr(sys,Q,R,N)

A1 = A - np.dot(B,K)
B1 = np.zeros((n_x,n_u))
C1 = C - np.dot(D,K)
D1 = np.zeros((n_y,n_u))
sys3 = ctrl.ss(A1,B1,C1,D1)

t0 = 0.
tf = 20.
X0 = 0.1*np.ones((n_x,1))
N = 500
T = np.linspace(t0,tf,N)

plt.figure(0)
plt.hold(False)
for i in range(n_y):
    Y,T=ctrl.step(sys3,T,x0,0,i)
    plt.plot(T,Y,label = 'y_{0}'.format(i+1))
    plt.hold(True)
    plt.xlabel('Time')
    plt.title('Outputs')
    plt.legend(loc='upper right', numpoints = 1)
    plt.grid(True)
    plt.show()
```

Note that both disturbances and control inputs are stacked in δu such that 12th, 6th and 5th elements are δu_1, δu_2, and δu_3 respectively.[19] When designing the LQR state feedback controller above, δu is considered to contain only control variables. However, this is not realistic. δu should have been decomposed as $\delta u := [\delta u, \delta w]^T$ and handled disturbances

[19]Check `print ocp.u`.

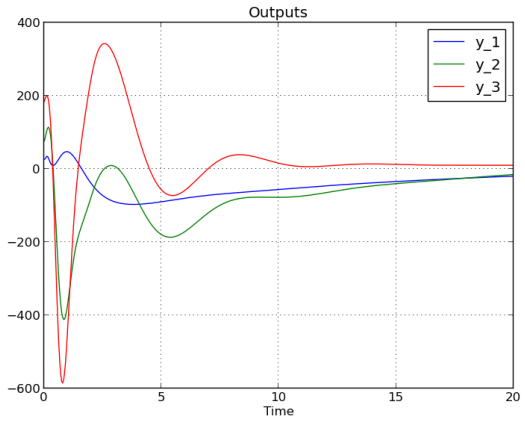

Figure 2: The $\delta y_1 - \delta y_2 - \delta y_3$ vs. t plot.

accordingly. Since, this paper is mainly concentrated on demonstrating the possibility of analyzing Modelica models in Python, in detail discussions on controller syntheis is not given here.

5 CONCLUSIONS

The features of Modelica language, in particular the notion of acausal modeling, have made it a powerful tool for modeling physical systems. However, the Modelica standards target primarily on model simulation, which is just one of the aspects of modeling. It is important that Modelica models are available for general use, but not just for the simulation. CasADi has an interface to Modelica/Optimica and JModelica.org is linked with CasADi. Therefore, Modelica-CasADi-JModelica.org combination provides a useful way to access Modelica/Optimica models in Python. Although, CasADi and JModelica.org has some limitations, they have provided a starting point. In this paper, it was explained the usefulness of interfacing Modelica models with Python. Special emphasis was given on the Python control system library as an up coming Python control tool, which could be an alternative to MATLAB control system toolbox.

Finally, couple of suggestions are made. CasADi-Modelica interface (via XML representation of Modelica models) may be further developed to support Modelica specification as much as possible. At the moment CasADi-Modelica interface is underdeveloped. The idea pointed out in this paper, in principal, for example may also be implemented within MATLAB environment. MathWorks provides the Simscape language and the Symbolic Math Toolbox (a CAS tool). The Simscape language is similar to Modelica. Therefore, Simscape-Symbolic Math Toolbox-MATLAB core may be designed to do the same as what Modelica-CasADi-Python does.

REFERENCES

[1] Åkesson J. *Optimica — An Extension of Modelica Supporting Dynamic Optimization*. 6th International Modelica Conference 2008.

[2] Andersson J. *A General-Purpose Software Framework for Dynamic Optimization*. PhD thesis. Arenberg Doctoral School, KU Leuven: Department of Electrical Engineering (ESAT/SCD) and Optimization in Engineering Center, Kasteelpark Arenberg 10, 3001-Heverlee, Belgium, 2013.

[3] Perera A. *Using CasADi for Optimization and Symbolic Linearization/Extraction of Causality Graphs of Modelica Models via JModelica.Org*. HiT Report No. 5. Porsgrunn: Telemark University College. `https://teora.hit.no/handle/2282/2175`. ISBN 978-82-7206-380-0. 2014.

[4] Lie B, Hauge TA. *Modeling of an industrial copper leaching and electrowinning process, with validation against experimental data*. Proceedings SIMS 2008, 49th Scandinavian Conference on Simulation and Modeling. Oslo University college. Oct 7-8, 2008.

[5] Fritzson P. *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica*. Wiley, 2011.

[6] Åström KJ, Murray RM. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2008.

[7] Cellier FE, Kofman E. *Continuous System Simulation*. Springer, 2006.

[8] Brenan KE, Campbell SL, Petzold LR. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM, 1996.

[9] Pantelides CC. *The Consistent Initialization of Differential-Algebraic Systems*. SIAM Journal Scientific Statistical Computation, 1988.

[10] Bendtsen C., Thomsen PG. *Numerical Solution of Differential Algebraic Equations*. TECHNICAL Report No. 5. Porsgrunn: Department of Mathematical Modelling, Technical University of Denmark. http://www2.imm.dtu.dk/documents/ftp/tr99/tr08_99.pdf. 1999.

[11] Mattsson SE, Söderlind G. *Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives*. SIAM Journal Scientific Statistical Computation, 1993.

[12] Griewank A, Walther A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.

[13] Pop A, Fritzson P. *ModelicaXML: A Modelica XML Representation with Applications*. Proceedings 3rd International Modelica Conference 2003.

[14] Casella F, Donida F., Åkesson J. *An XML Representation of DAE Systems Obtained from Modelica Models*. Proceedings 7th International Modelica Conference 2009.

[15] Andersson J, Gillis J, Diehl M. *User Documentation for CasADi*, 2014.

[16] Andersson J, Åkesson J, Casella F, Diehl M. *Integration of CasADi and JModelica.org*. 8th International Modelica Conference 2011.

[17] Pfeiffer CF. *Modeling, Simulation and Control for an Experimental Four Tanks Systems using ScicosLab*. 52nd Scandinavian Simulation and Modeling Society Conference 2011.

APPENDIX

A TANKSYSETEMS.MOP

```
package TankSystems
  //=====Dynamic model=====
  model FourTanks
  //Parameters
  parameter Real h1_init = 7.0;
  parameter Real h2_init = 7.0;
  parameter Real h3_init = 8.3;
  parameter Real h4_init = 3.1;
  parameter Real q1_init = 1;
  parameter Real q2_init = 1;
  parameter Real c1 = 9.82;
  parameter Real c2 = 5.76;
  parameter Real c3 = 9.02;
  parameter Real c4 = 8.71;
  parameter Real A1 = 12.57;
  parameter Real A2 = 12.57;
  parameter Real A3 = 12.57;
  parameter Real A4 = 12.57;
  parameter Real gama1 = 0;
  parameter Real gama2 = 0;
  parameter Real tau1 = 6.15;
  parameter Real tau2 = 13.2;
  parameter Real k1 = 6.94;
  parameter Real k2 = 8.72;
  //Dynamic variables
  Real h1(start=h1_init,fixed=true);
  Real h2(start=h2_init,fixed=true);
  Real h3(start=h3_init,fixed=true);
  Real h4(start=h4_init,fixed=true);
  Real q1(start=q1_init,fixed=true);
  Real q2(start=q2_init,fixed=true);
  //Output variables
  Real z1 = sqrt(h1)^2;
  Real z2 = sqrt(h2)^2;
  //Note: if we use z1 = h1, & z2 = h2 instead, then
  //z1 & z2 would not be+9 considered as algebraic
  //variables when the model is imported to CasADi.
  //The reason is that in the modelDescription.xml
  //file, both z1 & h1  would have  the same
  //valueReference. The same applied for z2. However,
  //there could be a better way of handling this!
  //Input variables
  input Real v1;
  input Real v2;
  equation
  der(h1) = ((-c1 * sqrt(h1)) +
  c3 * sqrt(h3) + gama1 * q1) / A1;
  der(h2) = ((-c2 * sqrt(h2)) +
  c4 * sqrt(h4) + gama2 * q2) / A2;
  der(h3) = ((-c3 * sqrt(h3)) +
  (1 - gama2) * q2) / A3;
  der(h4) = ((-c4 * sqrt(h4)) +
  (1 - gama1) * q1) / A4;
  der(q1) = ((-q1) + k1 * v1) / tau1;
  der(q2) = ((-q2) + k2 * v2) / tau2;
  end FourTanks;
  //=====Dynamic model=====
  //=====Static model=====
  model FourTanks_Init
  extends FourTanks(h1(fixed=false),
  h2(fixed=false),  h3(fixed=false),
  h4(fixed=false),q1(fixed=false),
  q2(fixed=false));
  initial equation
  der(h1) = 0;
  der(h2) = 0;
  der(h3) = 0;
  der(h4) = 0;
```

```
  der(q1) = 0;
  der(q2) = 0;
  end FourTanks_Init;
  //=====Static model=====
end TankSystems;
```

B TANKSYSTEMS.PY

Let, $0 = f(\dot{x},x,u,z)$ and $0 = g(x,u,z)$. The linearized model is given by, $0 = \alpha \cdot \delta\dot{x} + \beta \cdot \delta x + \gamma \cdot \delta u + \delta \cdot \delta z$ and $0 = \zeta \cdot \delta x + \eta \cdot \delta u + \sigma \cdot \delta z$, where $\alpha = \frac{\partial f}{\partial \dot{x}}$, $\beta = \frac{\partial f}{\partial x}$,..., and $\sigma = \frac{\partial g}{\partial z}$. δy is taken as $\delta y = [\kappa_x\ \kappa_u\ \kappa_z] \cdot [\delta x\ \delta u\ \delta u]^T$, where κ_x, κ_u, and κ_z should be given.

```
#Importing necessary packages.
import numpy as np
import matplotlib.pyplot as plt
import control as ctrl
from casadi import *
from casadi.tools import *
from pymodelica import compile_jmu
from pymodelica import compile_fmux
from pyjmi import JMUModel
from pyjmi import CasadiModel

#Compiling (to a JMU)/loading
#steady state model.
jmu_init = compile_jmu \
("TankSystems.FourTanks_Init", \
"TankSystems.mop")
init_model = JMUModel(jmu_init)

#Set inputs
v1_0 = 1.
v2_0 = 2.
u_0 = [v1_0, v2_0]
u = ['v1','v2']
init_model.set(u,u_0)

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

#Store steady state
h1_0 = init_result['h1'][0]
h2_0 = init_result['h2'][0]
h3_0 = init_result['h3'][0]
h4_0 = init_result['h4'][0]
q1_0 = init_result['q1'][0]
q2_0 = init_result['q2'][0]

#Compiling (to a FMUX)/loading dynamic model
fmux_name = compile_fmux \
("TankSystems.FourTanks", \
"TankSystems.mop")
model = CasadiModel(fmux_name)

#Get access to OCP
ocp = model.ocp

# Get differential state
n_x = len(ocp.x)
x = list()
for i in range(n_x):
    x.append(ocp.x[i].var())

#Get derivatives
xDot = list()
for i in range(n_x):
    xDot.append(model.dx[i])

# Get input
n_u = len(ocp.u)
u = list()
for i in range(n_u):
    u.append(ocp.u[i].var())

#Get algebraic states
n_z = len(ocp.z)
z = list()
for i in range(n_z):
    z.append(ocp.z[i].var())

#Eliminating dependent parameters
ocp.eliminateDependent()

#Define DAEs
f = ocp.ode
g = ocp.alg

#Create SXFunction instances for f and g
ffun = SXFunction([vertcat(xDot),vertcat(x), \
vertcat(u),vertcat(z)],[f])
gfun= SXFunction([vertcat(x),vertcat(u), \
vertcat(z)],[g])
ffun.init()
gfun.init()

#Define x0, u0, and z0
x0 = [h1_0,h2_0,h3_0,h4_0,q1_0,q2_0]
xDoto0 = [0.,0,0,0,0,0]
u0 = [v1_0,v2_0]
z0 = [h1_0,h2_0]

#Find symbolic/numeric Jacobian matrices
f_xDot = ffun.jac(0)
f_xDot_fun = SXFunction([vertcat(xDot),vertcat(x), \
vertcat(u),vertcat(z)],[f_xDot])
f_xDot_fun.init()

f_xDot_fun.setInput(xDoto0,0)
f_xDot_fun.setInput(x0,1)
f_xDot_fun.setInput(u0,2)
f_xDot_fun.setInput(z0,3)
f_xDot_fun.evaluate()

f_xDot_num = f_xDot_fun.getOutput()

alpha = np.array(f_xDot_num)
#
f_x = ffun.jac(1)
f_x_fun = SXFunction([vertcat(x),vertcat(u),\
vertcat(z)],[f_x])
f_x_fun.init()

f_x_fun.setInput(x0,0)
f_x_fun.setInput(u0,1)
f_x_fun.setInput(z0,2)
f_x_fun.evaluate()

f_x_num = f_x_fun.getOutput()

beta = np.array(f_x_num)
#
f_u = ffun.jac(2)
f_u_fun = SXFunction([vertcat(x),vertcat(u),\
vertcat(z)],[f_u])
```

```
f_u_fun.init()

f_u_fun.setInput(x0,0)
f_u_fun.setInput(u0,1)
f_u_fun.setInput(z0,2)
f_u_fun.evaluate()

f_u_num = f_u_fun.getOutput()

gamma = np.array(f_u_num)
#
f_z = ffun.jac(3)
f_z_fun = SXFunction([vertcat(x),vertcat(u), \
vertcat(z)],[f_z])
f_z_fun.init()

f_z_fun.setInput(x0,0)
f_z_fun.setInput(u0,1)
f_z_fun.setInput(z0,2)
f_z_fun.evaluate()

f_z_num = f_z_fun.getOutput()

delta = np.array(f_z_num)
#
g_x = gfun.jac(0)
g_x_fun = SXFunction([vertcat(x),vertcat(u), \
vertcat(z)],[g_x])
g_x_fun.init()

g_x_fun.setInput(x0,0)
g_x_fun.setInput(u0,1)
g_x_fun.setInput(z0,2)
g_x_fun.evaluate()

g_z_num = g_x_fun.getOutput()

zeta = np.array(g_z_num)
#
g_u = gfun.jac(1)
g_u_fun = SXFunction([vertcat(x),vertcat(u),\
vertcat(z)],[g_u])
g_u_fun.init()

g_u_fun.setInput(x0,0)
g_u_fun.setInput(u0,1)
g_u_fun.setInput(z0,2)
g_u_fun.evaluate()

g_u_num = g_u_fun.getOutput()

eta = np.array(g_u_num)
#
g_z = gfun.jac(2)
g_z_fun = SXFunction([vertcat(x),vertcat(u), \
vertcat(z)],[g_z])
g_z_fun.init()

g_z_fun.setInput(x0,0)
g_z_fun.setInput(u0,1)
g_z_fun.setInput(z0,2)
g_z_fun.evaluate()

g_z_num = g_z_fun.getOutput()

sigma = np.array(g_z_num)

# Define A, B, C, and D matrices
n_y = 2
kappa_x = np.eye(n_y,n_x)
kappa_u = np.zeros((n_y,n_u))
kappa_z = np.zeros((n_y,n_z))
if np.allclose(np.linalg.det(alpha),0.) != True:
    if np.allclose(np.linalg.det(sigma),0.) \
    != True:
        A = np.dot(np.linalg.inv(alpha),(-beta+\
        np.dot(delta,np.dot(np.linalg.inv(sigma),\
        zeta))))
        B = np.dot(np.linalg.inv(alpha),(-gamma+\
        np.dot(delta,np.dot(np.linalg.inv(sigma),\
        eta))))
        C = kappa_x - np.dot(kappa_z,\
        np.dot(np.linalg.inv(sigma),zeta))
        D = kappa_u - np.dot(kappa_z,\
        np.dot(np.linalg.inv(sigma),eta))
    #Use python-control
    #Create state space model object
        sys = ctrl.ss(A,B,C,D)
        print sys
    #State space to transfer function model object
        sys2 = ctrl.ss2tf(sys)
        print sys2
    # Simulate the system given input
        t0 = 0.
        tf = 120.
        N = 1000
        T = np.linspace(t0,tf,N)
        U = np.dot(np.diag([v1_0,v2_0]),np.ones((n_u,N)))
        t, yout, xout = ctrl.forced_response(sys,T,U,x0)

        plt.figure(0)
        plt.hold(False)
        for i in range(n_y):
            plt.plot(t,yout[i],'.',label = \
            'y_{0}'.format(i+1))
            plt.hold(True)
        plt.xlabel('Time')
        plt.title('Outputs')
        plt.legend(loc='upper right', numpoints = 1)
        plt.show()
        plt.figure(1)
        plt.hold(False)
        for i in range(n_x):
            plt.plot(t,xout[i],'.',label = \
            'x_{0}'.format(i+1))
            plt.hold(True)
        plt.xlabel('Time')
        plt.title('States')
        plt.legend(loc='upper right', numpoints = 1)
        plt.show()
        plt.figure(2)
        plt.hold(False)
        for i in range(n_u):
            plt.plot(t,U[i,:],'.',label = \
            'u_{0}'.format(i+1))
            plt.hold(True)
        plt.xlabel('Time')
        plt.title('Inputs')
        plt.legend(loc='upper right', numpoints = 1)
        plt.show()
    else:
        print 'sigma is singular. This case is \
        not considered.==>check Pantelides algorithm.'
else:
    print 'Alpha is singular. This case is \
    not considered.==>check Pantelides algorithm.'
```