2018 European Control Conference (ECC)
June 12-15, 2018. Limassol, Cyprus

ODESCA: A tool for control oriented modeling and analysis in MATLAB

Tim Grunert[1], Christian Schade[1], Claudia Michalik[2], Sven Fielsch[2], Lars Brandes[1] and Anton Kummert[2]

***Abstract*— The tool ODESCA and its idea of component based modeling and analysis of nonlinear systems in MATLAB is introduced. An overview over its main functionalities is given and within examples it is shown how the tool can be used for modeling and analysis tasks that often occur when applying advanced systems and control theory.**

I. INTRODUCTION

Increasing digitalization and an ongoing demand for efficiency and comfort pose a challenge to many developers. The fact that heating and cooling account for 50% of the EU's final energy consumption and even 79% of the households' consumption [1] poses an enormous challenge to the HVAC industry in times of energy transformation. To save energy, different appliances are connected and put into interaction. Meanwhile the same appliances are made smart which means that data from and to the cloud needs to be processed or offline algorithms for condition monitoring or virtual senors must be developed. In this context a deep understanding of advanced control theory and the appliances involved plays a big role for future developments.

A common approach is to develop complex functions likewise shown in the guideline VDI 2206 "Design methodology for mechatronic systems" [2]. An essential part in this methodology is to build up models for answering important questions early in the development process. These models are often implemented in simulation tools, e.g. Simulink, Simscape or Modelica. Regarding the methods of control theory however, symbolic models are more appropriate than simulation models. They enable a lot of additional mathematical methods and fulfill tasks, e.g. system identification, controller synthesis, as well as robustness and stability analysis.

While a lot of tools exist helping the user to work efficiently with simulation models, creating symbolic models is still a lot of manual and fault-prone work in computer algebra systems. To tackle this problem this paper introduces the tool ODESCA (acronym for "**O**rdinary **D**ifferential **E**quation **S**ystems: **C**reation and **A**nalysis"), developed to extend the existing simulation environment MATLAB/Simulink to allow modeling and the analysis of nonlinear systems in a comfortable, structured and therefore reproducible way.

[1]Lars Brandes, Tim Grunert and Christian Schade, are with Vaillant GmbH, D-42859 Remscheid, Germany {lars.brandes, tim.grunert, christian.schade}@vaillant-group.com

[2]Sven Fielsch, Anton Kummert and Claudia Michalik are with Faculty of Electrical, Information and Media Engineering, University of Wuppertal, D-42119 Wuppertal, Germany {s.fielsch, kummert, c.michalik}@uni-wuppertal.de

First functions of this tool were initially created in [3] to optimize the procedure of system identification and linearization of a gas boiler model. By applying nonlinear control algorithms, e.g in [4], additional functionalities were added. After reworking and composing these functions to a first version of ODESCA in [5] the tool is used sucessfully for a wide range of control applications occuring in the HVAC industry, e.g. [6].

In the second section of this paper a short overview over the structure of the tool is given and the typical workflow is shown. In the third section a minimal example is introduced which is used throughout the fourth section to show the advantages of working with ODESCA. In the fifth section a short conclusion and future goals for the tool are given.

ODESCA is published as free software under the GNU Lesser General Public License (LGPL) version 3 and can be downloaded here: http://github.com/odesca .

II. INTRODUCTION TO ODESCA

ODESCA is build to set up dynamical systems in the nonlinear state-space representation of the form:

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t), p) \\ y(t) &= g(x(t), u(t), p) \end{aligned} \tag{1}$$

where $x \in \mathbb{R}^n$ denotes the states, $u \in \mathbb{R}^m$ the inputs, $y \in \mathbb{R}^r$ the outputs, and $p \in \mathbb{R}^k$ the parameters of the system.

To enable this ODESCA makes use of the MATLAB object oriented (OO) programming techniques. This way objects, similar to the MATLAB state space (ss) object, can be created and the linear system repesentation is extended to represent dynamical systems of the above mentioned form (1).

ODESCA serves two main classes: components and systems. A component describes a part of an entire system, e.g. a pipe describing the temperature dynamics at its output. The component class supports the user defining the structure of a set of ordinary differential equations (ODEs). Additionally it contains information of the states, inputs, outputs and parameters. This way a library of basic components can be build up leading to reusability and testability in an encapsulated fashion. The system class supports the user to define the interaction between components e.g. setting the output temperature of a pipe as an input value of a temperature sensor. This procedure allows to set up and edit large symbolic models easily.

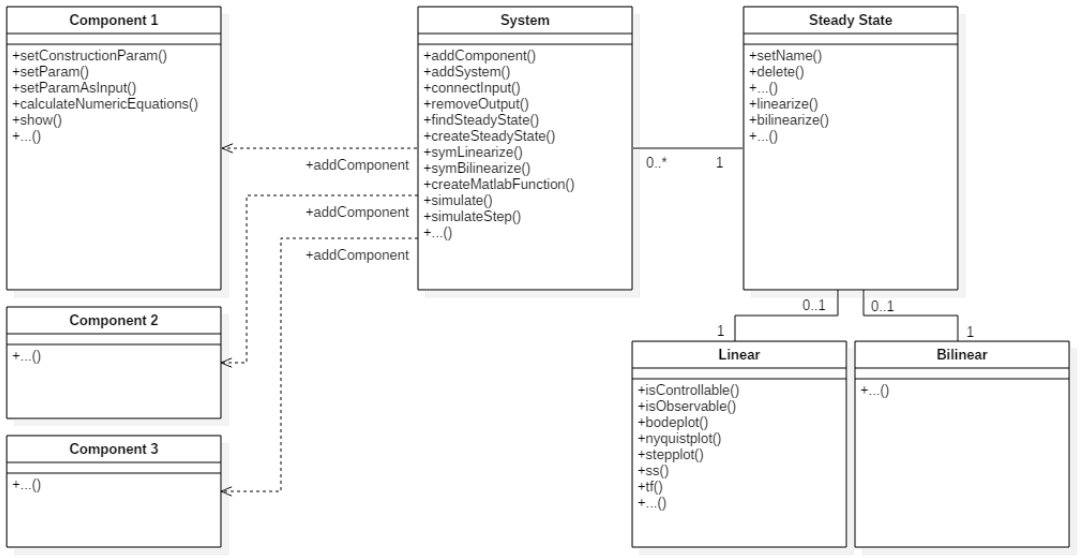

Fig. 1. Simplified class structure of ODESCA showing up the most important MATLAB classes and their methods to model and work with dynamic systems.

In contrast to other OO modeling languages, e.g. Modelica, ODESCA makes directly use of a computer algebra system. The MATLAB internal Symbolic Math Toolbox is used to represent the states, inputs and equations of components and systems. This way a lot of functionalites are available to affect the equations, calculating approximations or building interfaces. In the process of symbolic substitution, that brings the components equations into interaction, some effects like algebraic loops or zero dynamics can occur. Other modeling languages can handle these effects in simulation, but do not change the system equations. In ODESCA these effects needs be tackled manually by changeing the equations but results in well defined models that can be used for more than calculating simulation results.

Beside connecting components and the general access to the symbolic equations, the system class supports the user to perform analysis tasks. One fundamental function is the handling of multiple steady states of a system. Around a steady state system approximations can be deduced from the nonlinear equations. Currently the linear and bilinear state-space representations are supported. Further approximations, e.g. polynomial systems, are planned in future releases. In case of the linear state-space representation, an interface to the MATLAB state-space object is used as foundation to access the broad variety of functions based on it. Another important function creates a Simulink model of the system enabling the analysis in the time domain by defining custom simulation scenarios.

A simplified overview over the class structure of the tool including public methods is given in Fig. 1.

III. Example Application

For a better explanation of the tool an example application is introduced that consists only of a simple pipe and a temperature sensor model (see Fig. 2).

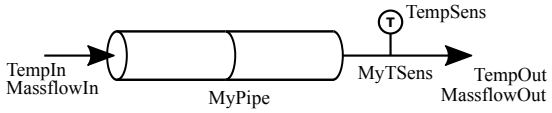

Fig. 2. Scheme of the example application consisting of a pipe modeled in two volumes and a temperature sensor at the outlet of the pipe.

To model the transport process through a simple pipe, it can be divided into an arbitrary number of volumes described as nodes. For reasons of simplicity effects like the power loss to the ambiance are neglected. Thus, the differential equation for the temperature ϑ in each node i results in

$$\frac{\partial \vartheta_i}{\partial t} = \frac{\dot{m}_{in} \cdot c_f \cdot (\vartheta_{i-1} - \vartheta_i)}{c_p \cdot \frac{m_p}{n} + \rho_f \cdot c_f \cdot \frac{V_p}{n}} \tag{2}$$

with heat capacity c, mass m and density ρ. The indices f and p assign the parameters to the fluid or the pipe. For the first node ϑ_{i-1} is replaced by the pipe inlet temperature. The outlet mass flow corresponds to the inlet mass flow.
The temperature sensor as the second component of the example consist of only one input and output for the temperature ϑ. The dynamic behavior is described as first order element

$$\frac{\partial \vartheta}{\partial t} = \frac{1}{T} \left(k \cdot \vartheta_{in} - \vartheta \right) \tag{3}$$

with time constant T and gain k.

IV. Main functions of ODESCA

A. Setting up Components

To create a new custom component a subclass of `ODESCA_Component` needs to be created. A small GUI helps copying a template and can be called with the command:

```
>> ODESCA_Util.createNewComponentFile
```

In the indicated editable sections of the template, names and units for the component states, inputs, outputs and parameters are set as arrays of strings. As an essential requirement the component must contain at least one output. By accessing the arrays `obj.f(i)` and `obj.g(i)` the differential equations are defined using the symbolic variables for the states `obj.x(i)`, inputs `obj.u(i)` and parameters `obj.p(i)` with the appropriate indices i or, alternatively, the defined symbolic names.

By the definition of construction parameters the structure of the differential equations can be modified which allows, for example, the creation of finite element method (FEM) modeling with a variable number of equations.

As an example the script below shows the modeling of the dynamic behavior of a temperature sensor as first order element:

```
classdef OCLib_TSensor < ODESCA_Component
...
constructionParamNames = {};
...
stateNames  = {'Temp'};
stateUnits  = {'°C'};

inputNames  = {'TempIn'};
inputUnits  = {'°C'};

outputNames = {'TempOut'};
outputUnits = {'°C'};

paramNames  = {'TimeConst', 'Gain'};
paramUnits  = {'s', '1'};
...
obj.f(1) = (Gain*TempIn-Temp)/TimeConst;
obj.g(1) = Temp;
...
end
```

Using this class definition a various number of component objects can be created and parameterized in MATLAB by calling:

```
>> TSens = OCLib_TSensor('MyTSens');
>> TSens.setParam('Gain', 1);
>> TSens.setParam('TimeConst', 2);
```

In case of a component using construction parameters these parameters needs to be set first with numerical values.

```
>> Pipe = OCLib_Pipe('MyPipe');
>> Pipe.setConstructionParam('Nodes',2);
>> Pipe.setParam('cPipe', 500);
>> Pipe.setParam('mPipe', 0.5);
>> Pipe.setParam('VPipe', 0.001);
>> Pipe.setParam('RhoFluid', 998);
>> Pipe.setParam('cFluid', 4182);
```

B. Setting up Systems

System objects can be easily created by calling:

```
>> PipeSys = ODESCA_System('MySystem',TSens);
```

As systems must consist of at least one component, the existing component object from section IV-A is given as argument. Multiple components can later be added by using:

```
>> PipeSys.addComponent(Pipe);
```

With this command the systems set of equations of the nonlinear statespace representation (1) is extended by the components equations like shown in (4).

$$\begin{aligned}\hat{f}_s(\hat{x}_s,\hat{u}_s,\hat{p}_s) &= \begin{pmatrix} f_s(x_s,u_s,p_s) \\ f_c(x_c,u_c,p_c) \end{pmatrix} \\ \hat{g}_s(\hat{x}_s,\hat{u}_s,\hat{p}_s) &= \begin{pmatrix} g_s(x_s,u_s,p_s) \\ g_c(x_c,u_c,p_c) \end{pmatrix}\end{aligned} \tag{4}$$

where $\hat{f}_s$ and $\hat{g}_s$ describes the system functions after adding the component to the system and f_c and g_c describes the functions of the component.

With this kind of system an interaction between components is not possible. To achieve this the following command can be used to create a coupeled system:

```
>> PipeSys.connectInput('MyTSens_TempIn', ...
                        'MyPipe_TempOut');
```

This command substitutes a given input $\hat{u}_{s,i}$ with an arbitrary symbolical expression, usually an output equation of the system $\hat{g}_{s,j}(...)$. Therefore the tool ensures that $\hat{g}_{s,j}(...)$ is not dependent of $\hat{u}_{s,i}$.

To reduce the system complexity remaining system outputs can be removed using the command:

```
>> PipeSys.removeOutput('MyPipe_mDotOut');
>> PipeSys.removeOutput('MyPipe_TempOut');
```

In the system functions this effects that the selected row in the output function of the system $\hat{g}_{s,l}$ is removed.

With the resulting system the world is open for a lot of analytical or optimization based analysis and synthesis. Some examples are given in the next sections.

C. Creating Approximations

With the completed system several analysis tasks can be performed. Due to the complexity of nonlinear systems, analysis methods exist for certain classes only. The next step is therefore creating approximations around defined steady states and the application of existing methods for a deep analysis of the systems dynamics.
For the example of the pipe and temperature sensor system the steady state $u_0 = (40\ \ 0.1)^T$ for the pipe inlet temperature and pipe inlet mass flow is regarded. An appropriate state vector x_0 can be calculated using the system's equations:

```
>> steadyState = vpasolve(subs(...
        PipeSys.calculateNumericEquations,...
        PipeSys.u,u0),PipeSys.x);
>> x0(1,1) = steadyState.x1;
>> x0(2,1) = steadyState.x2;
>> x0(3,1) = steadyState.x3;
>> x0 = double(x0);
```

Using the method

```
>> ss1 = PipeSys.createSteadyState(...
                x0,u0,'ss1');
```

the steady state is created around u_0, x_0 and added to the system. As the system can be linked to multiple steady states like shown in Fig. 1 the name is a helpful argument to identify the desired point. A steady state instead is linked to exactly one system. After the steady state was created an approximation can be created which is linked to it. A common approach is the linear approximation which is realized in the class `ODESCA_Linear`. The class stores the linear state space matrices A, B, C, D. It should be noted that the control system toolbox is necessary for executing the approximation.
By calling the command

```
>> sys_lin = ss1.linearize();
```

provided methods allow e.g. the proof of stability, controllability and observability as well as the calculation of transfer functions and bode plots for the linear approximation of the nonlinear system. In MATLAB the properties check can be performed calculating the rank of the observability and stability matrices according to Kalman. For large systems this can lead to numerical problems and is therefore not recommended for testing. Instead Hautus test can be applied leading to more reliable results [7]. The tool ODESCA provides the possibility to choose between those two methods. The code below shows some exemplary approach for testing the systems properties:

```
>> stable = sys_lin.isAsymptoticStable();
>> ctrl = sys_lin.isControllable('hautus');
>> obsv = sys_lin.isObservable('hautus');
```

Bode plots of the dynamic system can be drawn by defining the input and output explicitly or for all combinations.

```
>> sys_lin.bodeplot();
```

The analysis can also be performed for multiple steady states. For the system example this is shown for three steady states $u_{0,1} = (40\ \ 0.1)^T$, $u_{0,2} = (40\ \ 0.2)^T$ and $u_{0,3} = (40\ \ 0.25)^T$ with constant pipe inlet temperature and varying mass flows. The method `linearize()` returns all linearizations of all defined steady states in an array. Executing

```
>> lin = PipeSys.steadyStates.linearize();
>> lin.bodeplot('from', 1, 'to', 1);
```

leads to the bode plot shown in Fig. 3. Further methods for the linear approximation are explained in the user guide document.
Besides the linear approximation ODESCA also provides the

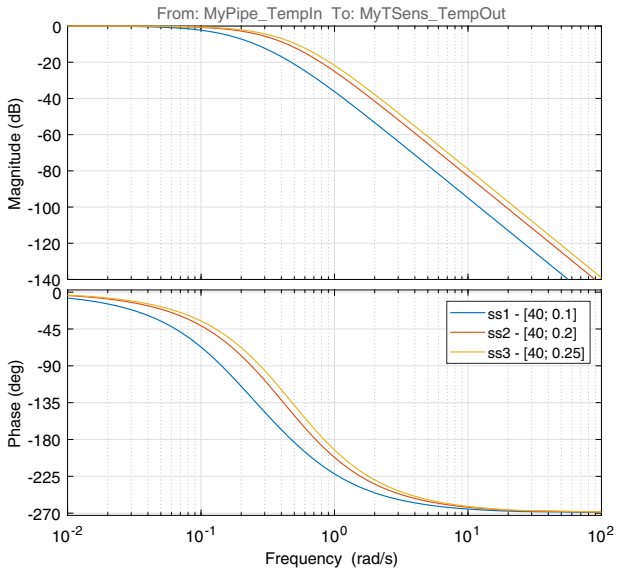

Fig. 3. Amplitude and phase diagram for three different steady states with varying mass flows from pipe temperature inlet to temperature sensor outlet.

bilinear approximation. The bilinear state space representation regarded in this work is described by

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t) + \sum_{i=1}^{m} u_i N_i x + \sum_{i=1}^{n} x_i M_i x \\ &\quad + \sum_{i=1}^{n} u_i G_i u \\ y(t) &= Cx(t) + Du(t)\end{aligned} \tag{5}$$

with matrices of appropriate dimensions. Note that the tool ensures that the corresponding colums of the matrices M_i and G_i are zero vectors to avoid quadratic terms.
Regarding HVAC systems or the smaller example of the pipe and temperature sensor system bilinearities appear naturally in the nonlinear equations [8]. Therefore, in some cases the bilinear approximation is a promising alternative to linear approximations for nonlinear systems. The bilinear state space representation for a steady state u_0, x_0 can be calculated calling:

```
>> sys_bilin = ss1.bilinearize();
```

The above mentioned matrices are numerically available in the resulting object. With them advanced synthesis and analysis approaches using semidefinite programming e.g. YALMIP [9] and SeDuMi [10] can be applied.

A comparison of the linear approximation vs. the bilinear approximation around the steadystate $u_{0,1}$ is shown in Fig. 4. There at $t = 5s$ an inlet temperature step from $40°C$ to $60°C$ is given to the approximations while the massflow is constantly hold at 0.2 kg/s. It is noted here that for this example the bilinear approximation completely represents the nonlinear system. In the mentioned figure it is clear to

see, that the step response of the linear approximation is slower than the bilinear approximation. The reason for this behavior is the difference between the operation point where the linearization was made to the simulated one. For HVAC appliances it is not uncommon to have massflows that can change by factor 10. Since in this example the simulated massflow was only twice as high as the one used for the linearization, one can imagine that bilinear approximations are of interest.

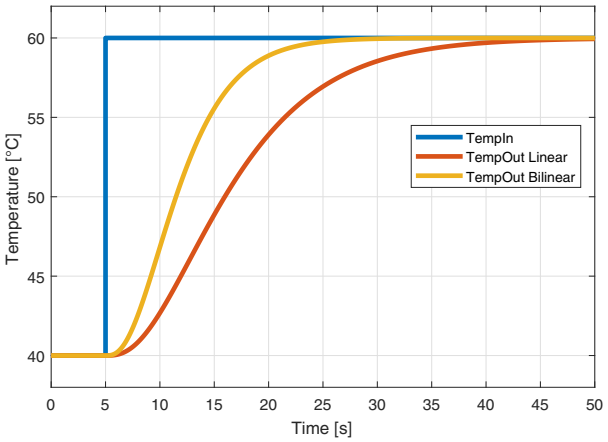

Fig. 4. Comparison of the step response of the linear and the bilinear approximation. At $t = 5s$ an inlet temperature step from $40°C$ to $60°C$ is given while the massflow is constantly hold at $0.2kg/s$.

D. Creating MATLAB functions

ODESCA also provides a method to create MATLAB function handles of the equations describing the system behavior according to (1). By calling

```
>> [f,g] = PipeSys.createMatlabFunction();
```

the function handles can easily be used for optimization tasks using either the MATLAB internal optimizers or specialized external tools for dynamic optimization e.g. CasADi [11]. Feeding the functions f and g with the CasADi specific symbolics leads directly to useable variables for optimization. The following code listing example makes use of the CasADi specific symbolics class `SX`. To use this example CasADi needs to be installed in MATLAB. Moreover the following example is arranged to be similar to the CasADi example `direct_single_shooting.m` from the example package.

```
>> % Declare model variables
>> x1 = SX.sym('x1');
>> x2 = SX.sym('x2');
>> x3 = SX.sym('x3');
>> x = [x1; x2; x3];
>> u1 = SX.sym('u1');
>> u2 = SX.sym('u2');
>> u = [u1;u2];
>> % Model equations
>> [f,g] = PipeSys.createMatlabFunction(...
                'useNumericParam',true);
>> xdot = f(x,u);
>> ...
```

E. Creating simulation models

To create a simulation model of the described pipe and temperature sensor system in Simulink the function

```
>> PipeSys.createNonlinearSimulinkModel();
```

is executed. The system's equations are written automatically into a Matlab function block with the inputs and outputs as defined before. Fig. 5 shows the resulting Simulink model.

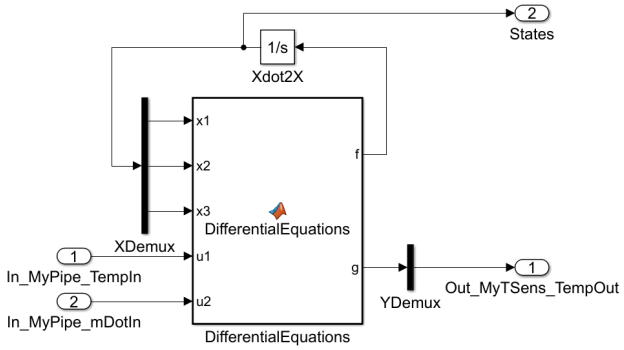

Fig. 5. Simulink model for example system with pipe and temperature sensor created by the method `createNonlinearSimulinkModel()`

The model's mask consists of different tabs for the initial conditions and each component including all parameters for the appropriate component. Therefore, the parameter values can easily be changed in the model without creating a new system (see Fig. 6).

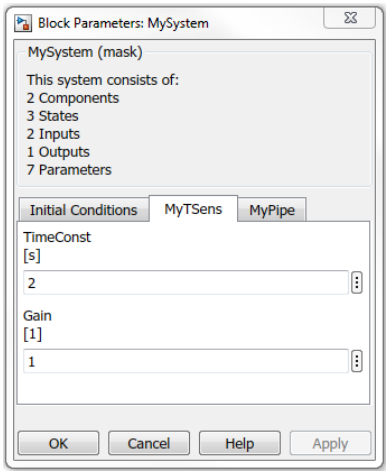

Fig. 6. Mask of the Simulink block created with the function `createNonlinearSimulinkModel()` that enables to vary initial conditions and components parameters for enhanced simulation studies.

V. CONCLUSION AND OUTLOOK

As the previous sections showed, ODESCA aims to fill the gap of structured, component-based and reusable control-oriented modeling and analysis. At the same time, it is open for interfaces to other tools fulfilling tasks like simulation and optimization. In addition to the easy interfacing to CasADi (see section IV-D), the translation to models usable in the ACADO toolkit [12] has already been implemented. This enables an automated workflow to use models build in ODESCA in optimal control and other tasks fulfilled by ACADO. These and other interfaces could be further enhanced in the future to make the same models usable in a variety of different tools. Moreover additional system approximations and representations should be implemented for rapid prototyping of various approaches. Of course, users are invited to contribute new functions and make proposals to improve existing ones.

REFERENCES

[1] European Comission, "An eu strategy on heating and cooling com(2016) 51," 02 2016.

[2] Verein Deutscher Ingenieure, "Design methodology for mechatronic systems," 06 2004.

[3] T. Grunert, "Modellbasierte Entwicklung eines optimierten Regelungskonzepts der Brauchwasserdurchlauferhitzung von wandhängenden Gasthermen," Master's thesis, Bergische Universität Wuppertal, 11 2013.

[4] T. Grunert, S. Fielsch, and M. Stursberg, "Estimating the outlet temperature of a plate heat exchanger: Application of bilinear observers," in *2015 IEEE Conference on Control Applications (CCA)*, Sept 2015, pp. 466–471.

[5] C. Schade, "Entwicklung eines Frameworks zur Erstellung und Analyse von nichtlinearen regelungstechnischen Systemen," Master's thesis, Bergische Universität Wuppertal, 10 2016.

[6] S. Fielsch, T. Grunert, M. Stursberg, and A. Kummert, "Model predictive control for hydronic heating systems in residential buildings," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 4216 – 4221, 2017, 20th IFAC World Congress.

[7] J. Lunze, *Regelungstechnik 2: Mehrgrößensysteme, Digitale Regelung*, ser. Springer-Lehrbuch. Springer Berlin Heidelberg, 2014.

[8] R. Mohler, *Nonlinear Systems: Applications to bilinear control*, ser. Nonlinear Systems. Prentice Hall, 1991.

[9] J. Löfberg, "Yalmip : A toolbox for modeling and optimization in MATLAB," in *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004.

[10] J. F. Sturm, "Using sedumi 1.02, a matlab toolbox for optimization over symmetric cones," *Optimization Methods and Software*, vol. 11, no. 1-4, pp. 625–653, 1999.

[11] J. Andersson, J. kesson, and M. Diehl, "Dynamic optimization with CasADi," in *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, Dec 2012, pp. 681–686.

[12] B. Houska, H. Ferreau, and M. Diehl, "ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization," *Optimal Control Applications and Methods*, vol. 32, no. 3, pp. 298–312, 2011.