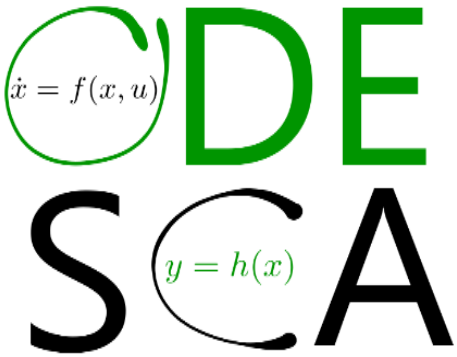

ODESCA User-Guide

CONTENTS

1 Introduction

ODESCA is a MATLAB tool for the creation and analysis of dynamic systems described by ordinary differential equations. The name ODESCA is an acronym for “Ordinary Differential Equation Systems: Creation and Analysis”.

For MBD (model-based-design) and parameterization, an algebraic model of the plant has a lot of advantages over a simulation model. With algebraic models, a multitude of mathematical approaches can be used for optimization and analysis.

To ensure uniform modeling and a simple way of working with differential equations, the tool ODESCA was created to fill the gap of modeling and analyzing ordinary differential equation systems in MATLAB.

With ODESCA, dynamical systems of the form

$$\dot{x} = f(x, u)$$
$$y = g(x, u)$$

can be set up in custom components. Components can be connected to each other in a system which provides methods for the analysis of the dynamical behavior. The analysis can be done by linearization at steady states and the implemented MATLAB features for linear systems. Moreover, some features for nonlinear analysis and synthesis are implemented, as well.

2 License

3 Installation

3.1 Requirements

The tool ODESCA was developed under MATLAB version R2016a. Therefore, the correct functionality of the tool is not guaranteed for older MATLAB releases.

The differential equations are described with symbolic variables which are part of the Symbolic Math Toolbox which is required for the tool.

For the linear analysis of nonlinear systems, ODESCA uses the control system toolbox.

The base tool will work with these three licenses. However, there are some functions which require additional licenses, e.g.: the function for the creation of nonlinear Simulink models for which the Simulink License is required or some functions of the class Util.

3.2 Required licenses

- MATLAB
- Symbolic Math Toolbox
- Control System Toolbox
- Other toolboxes for utility functions or special functions (see chapter 3.3)

3.3 Installation

The installation is accomplished as follows.

1. Download the latest release (https://github.com/odesca/ODESCA/archive/v1.0.zip).

2. Unzip the package to a location your choice.

3. Start MATLAB and run the file “addODESCAPaths.m” located in the just unzipped folder. On a clean run the command window should look like this:

```
>> addODESCAPaths
Symbolic math toolbox license has been checked out successfully.
Control system toolbox license has been checked out successfully.
Pathes added.
Version of ODESCA: v1.0
>>
```

At this point ODESCA is ready to use.

4a. The required ODESCA pathes are now added to the MATLAB search path for the current session. If you wish to install ODESCA permanently, you should save the MATLAB search path now by using the command ``savepath``.

4b. Another option might be the creation of a Command Shortcut (_Favorite Commands_ since R2018a) to add ODESCA smartly in future sessions. The following callback will do:

```
run('your_chosen_location\addODESCAPaths.m');
```

Examples and more information how to use ODESCA can be found in the Getting Started section

4 Getting Started

ODESCA is based on object-oriented programming, therefore it is relevant to understand the basic concept of the classes. After creating the classes, the data inside can be manipulated with the methods and the instances. These can be passed to functions.

For detail information about the classes move to the Class Documentation section.

The workflow in a new project typically follows these few steps:

- Creating the components (if not already created in other projects or as standard components)
- Creating instances of the components and parameterize them
- Creating system and connecting all components
- Analyzing the system mathematically (and creating a model of it)

In the following sections the four steps are described in more detail with examples. For a better understanding a system with two components will be created. Resistor and capacitor combined together in series form, create one component.

4.1 Creation of a custom component

For the creation and parameterization of the differential equations, components are needed. These components (E.g.: Pipes, Sensors, etc.) all have unique equations which have to be modeled. For this purpose, a sub class of ODESCA_Component has to be created. To generate a custom component, use the utility function

```
ODESCA_Util.createNewComponentFile()
```

This method will open a dialog for the creation of a new component file.

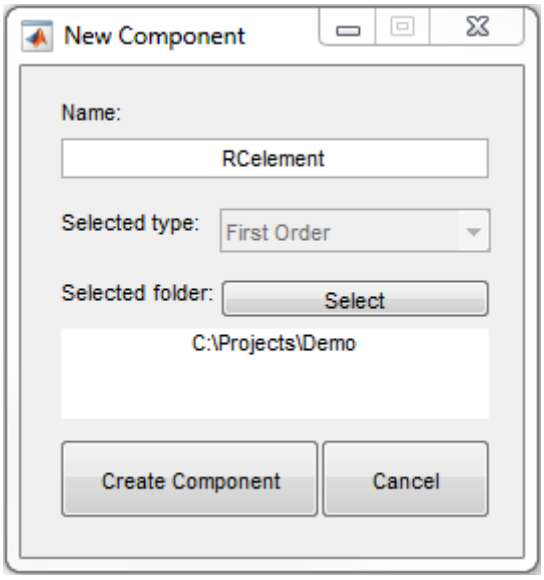

Figure 1: Utility window of creating a new component

After setting a name in the section Name, for example "RCelement", a location for the save location need to specified. By default, the location of example components is shown. The file should open automatically by clicking on the "Create Component" button.

Inside the file there are three sections which are marked as “User editable” where the differential equations system has to be described. Note that every change outside these sections (aside of the renaming of the file described above) may lead to an invalid component.

4.1.1 Definition of construction parameters

In this section every numeric variable necessary to create the equations can be defined in the array constructionParamNames. The names of the construction parameters have to be in this array as strings. For the example system the RC-element needs no construction parameters. Array and parameter area would look like below.

```
%================================================================
%% DEFINITION OF CONSTRUCTION PARAMETERS (User editable)
%================================================================

constructionParamNames = {};

%================================================================
```

In general, if no construction parameters are needed, the array has to be empty.

The construction parameters can be accessed in the two sections below. They are stored in a structure which is part of the superior class ODESCA_Component. To access these parameters, use the command

```
Obj.constructionParam.PARAMNAME
```

PARAMNAME is the name given to the construction parameters in the first section.

4.1.2 Definition of equations components

In the second section the states, inputs, outputs and parameters of the system have to be defined in the arrays stateNames, stateUnits, inputNames, inputUnits, outputNames, outputUnits, paramNames and paramUnits.
For each of these groups the names and the units have to be written to the arrays as strings. The number of units has to match the number of names. Each component need to have at least one output. Arrays can be empty if no states, no inputs or no parameters are required.

In the case of a RC-element, it could look like this:

```
%================================================================
%% DEFINITION OF EQUATION COMPONENTS (User editable)
%================================================================

stateNames  = {'Voltage_C'};
stateUnits  = {'V'};

inputNames  = {'Voltage_in'};
inputUnits  = {'V'};

outputNames = {'Voltage_out'};
outputUnits = {'V'};

paramNames  = {'Resistance', 'Capacity'};
paramUnits  = {'Ohm', 'Farad'};

% ================================================================
```

4.1.3 **Definition of equations**

In section three equations for state changes (f) and equations for outputs (g) of the component are created. To define these equations, the arrays f and g of the superior class ODESCA_Component have to be accessed.

For the state equations use the command

```
obj.f(NUM) =
```

and for the output equations use the command

```
obj.g(NUM) =
```

NUM is the number of the input in the arrays stateNames and inputNames.
Avoiding an error, the number of equations should match the number of states and the number of outputs defined in the second section.

Equations are defined symbolically using the symbolic variables [x1, x2, …] for the states, the symbolic variables [u1, u2, …] for the inputs and symbolic variables with the names of the parameters for the parameters. Symbolic variables that are not states, inputs or parameters, should not be part of the equations.

There are two ways to access these symbolic variables:
First is to access them on the component itself by calling the arrays

```
obj.x(NUM)
```

for the states,

```
obj.u(NUM)
```

for the inputs and

```
obj.u(NUM)
```

for the parameters where the position NUM of the symbolic variable corresponds with the position of the parts in the name arrays of the second section.

For the second way of use the template generates variables with the names of the states, inputs and parameters which contain the symbolic representation.

For the example of a RC-element, it could look like this:

```
%=============================================================
%% DEFINITION OF EQUATIONS (User Editable)
%=============================================================

obj.f(1) = (1/(Resistance * Capacity)) * (obj.u(1) - obj.x(1));
obj.g(1) = obj.x(1);

%=============================================================
```

After all the sections are filled, the component is ready for use. If the equations are not created correctly (wrong number of equations, wrong symbolic variables in the equations) the component will throw error on using it.

For a detailed example on how a component may look like, see the example “ExamplePipe” in the folder “Examples/Components/Examples”.

4.2 Creating instances of the components and parameterize them

After the creation of the custom component class files, instances of these classes can be created. In a new script called "RCSystem" these instances can be modified and parameterized before they are added to a system.

```
RC = RCelement('RC1')
```

If the custom component has construction parameters (like the nodes of a pipe) these construction parameters have to be set to numeric values before any other action can be made to the instance of the component. To set the construction parameters use the method

```
setConstructionParam(paramName, value)
```

After all construction parameters have been set, component will be filled with the states, inputs, outputs, parameters and equations. For this purpose, a method called

```
tryCalculateEquations()
```

is called internally, which checks, if the equations are created correctly in the class. If a component does not have construction parameters, the component will be filled when an instance of it is created.

After the equations have been calculated the component can be modified in different ways. The parameters can be set with values, the position of the inputs and outputs can be changed, the name of the component can be changed and parameters can be set as inputs.

```
RC.setParam('Resistance', 5)
RC.setParam('Capacity', 50)
```

An easy way to store different configurations of a component is to create a function which creates and parameterizes the component and returns it afterwards. If this function is stored in a folder which is called “+COMPONENTNAME” the function can be called by COMPONENTNAME.Function() name. The behavior is similar to a static method which creates a new parameterized version of the component. The “+”-Operator of the folder creates a new namespace. For an example see the folder “Examples/Components/Examples/+ExamplePipe”. Inside the file the concept is explained in detail.

The components can be added to a system described in the following chapter. If one component should be added multiple times to a system with slightly different parameters (E.g.: a system could have multiple pipes with different length or radius), the component can either be copied and the copy can be modified or one component can be used and the parameters are changed between the times the component is added to a system.

Note that a change made to a component after it was added to a system is not made to the equations of the system because the content of the component is copied to the system.

4.3 Create a system and connect all components

To connect components, create models and analyze the equations, a system is needed. So first of all, a new instance of the class ODESCA_System has to be created. To do so call the constructor in this way by using the command

```
ODESCA_System(name, comp)
```

The arguments “name” and “comp” specify the name of the system and the first component can be added.

```
sys = ODESCA_System('RCsystem', RC)
```

After creating a new system, components can be added by using the

```
addComponent(comp)
```

method. The component which should be added to the system has to be the argument of this method. It is not possible to add a component with the same name as a component already added to the system. Note that all construction parameters the component might have must be set before adding the component. Otherwise the component cannot be added to the system because of the equations cannot be created.

```
RC.setName('RC2')

sys.addComponent(RC)

sys.setParam('RC2_Resistance', 2)
sys.setParam('RC2_Capacity', 20)
```

While adding a component its name is added to the names of the states, inputs, outputs and parameters.
This is necessary to prevent name conflict and to determine to which component the data belongs.
E.g.: if a component named “Sensor” with a state called “Temperature” is added to the system, the name of the state changes to “Sensor_Temperature”.
Note that the properties of the component are copied to the system so a change to the instance of a component is not made to the corresponding equations in the system afterwards!

If the system only contains one component the next step can be skipped.

Now that the system is filled with components, it is time to connect them by replacing the inputs with outputs or equations. To do so use the method

```
connectInput(input, connection)
```

The argument “input” determines the input that should be replaced. The argument “connection” determines the variable that replaces the input. It can either be the name of an output as a string or a symbolic expression containing numeric values and states, inputs and parameters used in the system. NOTE that it must not contain the input which should be replaced, obviously.
E.g.: To connect the input of the second RC element (“RC2_Voltage_in”) to the output of the first RC element (“RC1_Voltage_out”) use the following command:

```
sys.connectInput('RC2_Voltage_in', 'RC1_Voltage_out')
```

If there are outputs that should not appear in the system or the model, use the command

```
removeOutput(toRemove)
```

The argument “toRemove” is the name of the output which should be removed or its position in the list of outputs.

```
sys.removeOutput('RC1_Voltage_out')
```

Now that the system is created, it can be analyzed. Furthermore, a nonlinear Simulink model can be created from it. To create a nonlinear Simulink model, use the function

```
createNonlinearSimulinkModel(system, options)
```

of the system class:

```
sys.createNonlinearSimulinkModel()
```

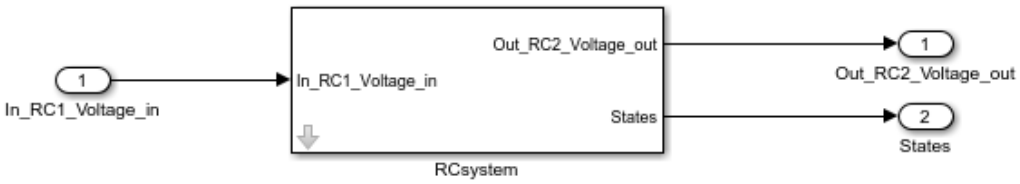

Figure 2: Nonlinear Simulink model

For more clarification, scripts with detailed comments that create systems can be found in the folder “Library/Systems/Example”.

4.4 Analyzing the system

After a system with all components is created, system can be analyzed mathematically. One way to do this is to create steady states. In a steady state the system is in an equilibrium which means the inputs and states are constant: f(x0, u0) = 0

To create a steady state, the constant inputs u0 and the constant states x0 of the steady state have to be used in the method

```
createSteadyState(x0, u0, name)
```

The method creates a new steady state and adds it to the system. The argument is name optional but helpful to give the steady state a meaningful identifier.

```
SteadyState = sys.createSteadyState([5, 5], 5, 'SteadyState')
```

After a steady state is created a linear approximation can be created. To create a linear approximation, use the method

```
linearize()
```

The method creates a linear approximation and adds it to the steady state.

Note: To create a linear approximation, the control system toolbox has to be available. If it is not, the method will throw an error.

To get all linearization of the steady states, use the method linear() which returns the linearization's in an array.

```
lin = linear(SteadyState)
```

Now that one or multiple linearization where obtained, a number of linear analysis method can be used. E.g.: if lin is an array of multiple linearization, the call

```
lin.bodeplot('from', 1, 'to', n)
```

plots the bode plot for all linearization. The options 'from' and 'to' specify that only the plot from input 1 to output number n should be displayed.
In this example there is only one bode plot to be created, so the following command can be used.

```
lin.bodeplot()
```

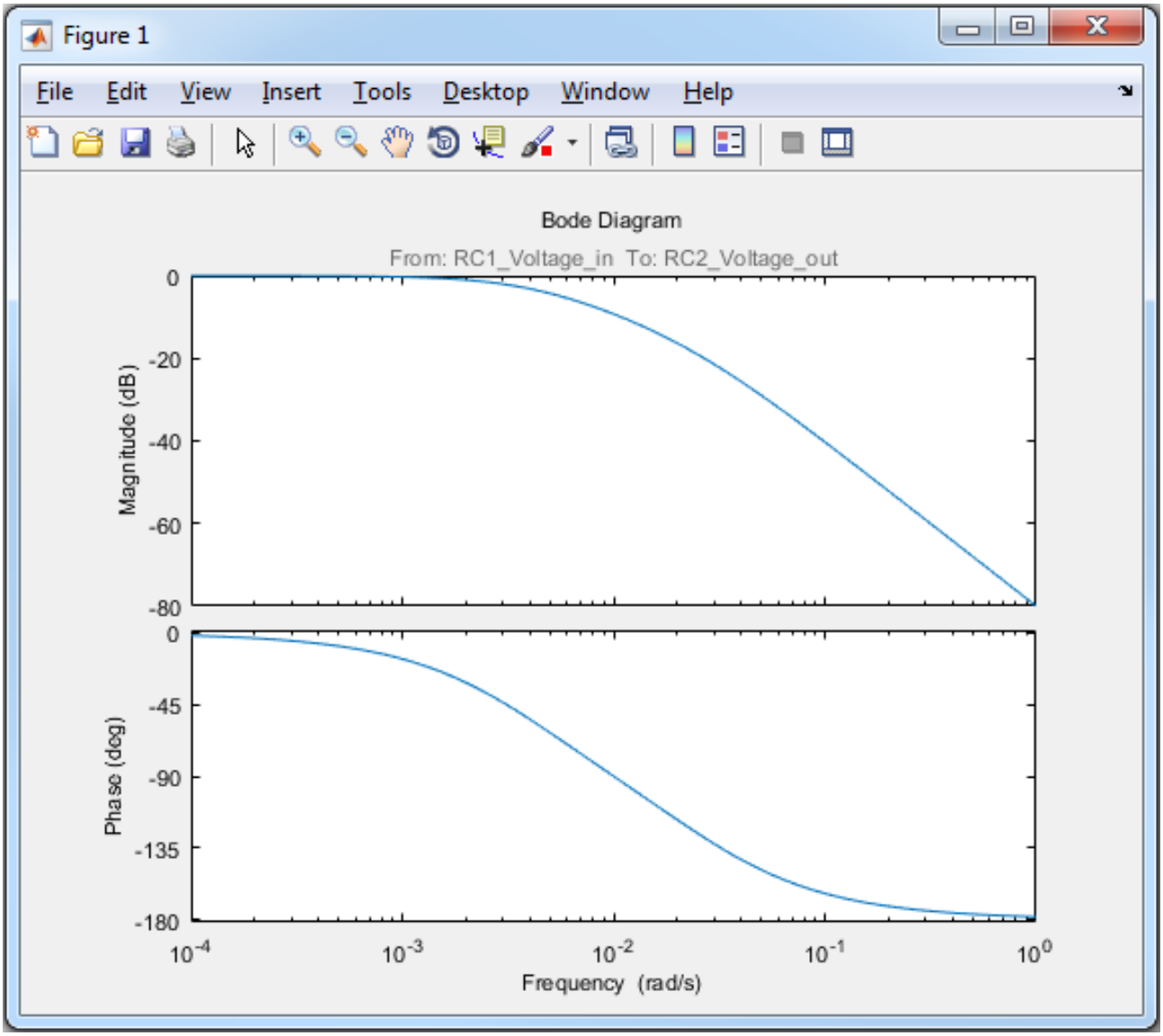

Figure 3: Bode plot of the steady state

Another example is the analysis of the stability which can be done easily for all linearization with the command

```
lin.isAsymptoticStable()
```

The method returns an array where each entry corresponds to the linearization in the array lin. 1 means asymptotic stable. In the same way the ability to control and to observe can be checked with the following command:

```
stable = lin.isAsymptoticStable()
ctrl = lin.isControllable('hautus')
obsv = lin.isObservable('hautus')
```

For a detailed example on how a system analyze may look like, see the example “PipeSystem” in the folder “Examples/Systems”.

5 Class Documentation – Properties and Methods

In this section the classes are documented with their methods and properties. Only the methods which have public access are listed. All properties are READ ONLY and can only be modified within the methods of the classes.

5.1 Class structure

ODESCA is based on object-oriented classes. The class diagram is shown in figure 1.
Meaning of colors:

- **Yellow:** Classes provided by MATLAB
- **Gray:** Abstract Classes, provide functionalities but cannot be instanced
- **White:** Classes which can be instanced and be used by the User
- **Blue:** Representation of classes created by the User

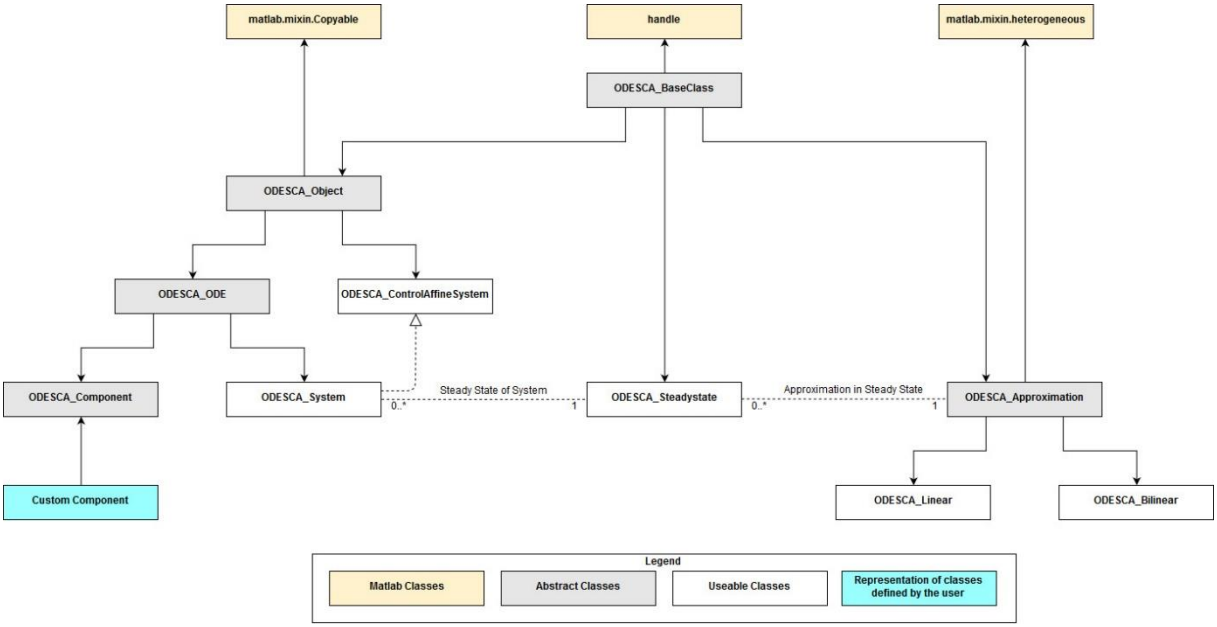

Figure 4: Class Diagram

- **BaseClass:**
  This class is used to keep track of the ODESCA version an instance of a class was created in if it was saved and loaded again.

- **Object:**
  The class *Object* owns the properties and methods to store differential equation systems of the form

  xdot = f(x,u)
  y = g(x,u)

  in symbolic variables. Furthermore, it provides a system to handle parameters and methods to organize the equation system.

- **ODE:**
  This class stores differential equation systems in the form

  xdot = f(x,u)
  y = g(x,u)

in symbolic variables. Furthermore, it provides a system to handle parameters and methods to organize the equation system.

- **Component:**
  The class provides functionality for the creation of differential equations and their parameters. It is used as a super class for custom components which can be added to a ODESCA_System.

- **CustomComponents:**
  The custom components are created by the user. They are representations of the different parts of system and are used to define the differential equations which describe the dynamic behavior. After a custom component class was created it can be instanced, parameterized and added to a *System* instance. The class file for a new custom component can be create with the method "createNewComponentFile()" of the class ODESCA_Util. For more information, check the utilities section.

- **System:**
  The class *System* is used to combine components to a system and analyze it with mathematical approaches. Furthermore, a nonlinear Simulink model can be created of the system. It is the only class in ODESCA which can be created directly by the user.

- **ControlAffineSystem:**
  This class contains the control affine representation of a nonlinear dynamic system. It stores differential equation systems in the form

  ```
  xdot = f0(x) + f1(x) * u
  y    = g(x,u)
  ```

  and can only be created out of an existing ODESCA_System.

- **SteadyState:**
  For the handling and analysis of a systems steady states (f(x,u) = 0!) the class called *SteadyState* is used. It is the representation of a system in a chosen steady state and provides the functionality to approximate the system around the steady state. This is useful to analyze the behavior of the *System* the steady state belongs to. A SteadyState is not existing without its linked System. Hence SteadyState vanishes when System is deleted.

- **Approximation:**
  All approximations attached to a SteadyState (e.g. a linearized or a bilinearized system) are subclasses of the class Approximation. Approximation mainly serves as an interface. An Approximation cannot exist without a SteadyState.

- **Linear:**
  This class is a subclass of *Approximation.* It represents the linear behavior of a system at the steady state the instance of this class belongs to. This makes use of the state space object (ss) of the control system toolbox and adds and improves functionalities. The main use of this class is to perform a linear analysis of a system in a steady state.

- **Bilinear:**
  This class is a subclass of Approximation. It stores the matrices of a bilinear system. The bilinear system represents the bilinearized System in its steady state.

Note that all class names of the tool ODESCA start with the name "ODESCA_". E.g.: the system class is called "ODESCA_System"

5.2 BaseClass

The BaseClass is the base of all classes in ODESCA and cannot be instantiated. Every other class in ODESCA is derived from this class. The main reason for the base class to exist is to keep track of the version number of ODESCA an instance of a class was create under. For this reason, the class provides two hidden properties.

5.2.1 Properties

Name	Description
version	Version of ODESCA the instance of the class was first created in
classDefinitionVersion	Current version of ODESCA (version of the class definition)

The properties have a public get access.

The two properties are hidden which means they do NOT appear in the list of properties, but trust me, they are there! The properties have a public get access.
The difference between version and classDefinitionVersion shows up after an instance which was saved to a .mat file is loaded. The property version is set at the first initialization of a class and never changes afterwards. The property classDefinitionVersion always matches the current ODESCA version. So if an instance of a class is created, the two properties have the same value. If an instance was saved and loaded in a later ODESCA version, the version is still the same as when it was saved but the classDefinitionVersion has changed. This properties can help with problems occurring after an instance is loaded from a .mat file.

Note, that the listed properties of the BaseClass have no own help side.

5.3 Object

The class ODESCA_Object is the absolute base for all other classes in the tool. It provides the possibility to store everything needed to describe nonlinear differential equations, like the states, inputs, outputs and parameters. It provides methods to modify the parameters of the object and methods to get information about the object.

This class is abstract so no instance can be created. It is meant to be the super class for the classes ODESCA_Component and ODESCA_System

5.3.1 Properties

Name	Description
name	Stores the name of the instance of this class
param	Structure that stores the parameters used in the equations with their current value
p	Array that stores the symbolic counterparts for the parameters
paramUnits	Cell array that stores the units of the parameters
x	Array with the symbolic states of the system
u	Array with the symbolic inputs of the system
stateNames	Cell array that stores the names of the states
inputNames	Cell array that stores or the names of the inputs
outputNames	Cell array that stores the names of the outputs
stateUnits	Cell array that stores the units of the states

inputUnits	Cell array that stores the units of the inputs
outputUnits	Cell array that stores the units of the outputs

To store the nonlinear differential equations in the form of

$$\dot{x} = f(x, u)$$
$$y = g(x, u)$$

all components of the equations (states, inputs, parameters) have to be stored as symbolic variables.

The names of the states are stored in the array **stateNames** and the symbolic variables representing the states are stored in the array x (e.g. x = [x1, x2, x3 …]) at the same position as the corresponding name.
The inputs are stored in the same way: **inputNames** stores the names and u stores the corresponding symbolic variables (e.g. u = [u1, u2, u3 …]).
The names of the outputs are stored in the array **outputNames**. There are no symbolic variables for the outputs because they do not appear inside the equations.
The parameters are stored in the structure **param** where the name of each field is the name of the parameter and the given value is stored in the field. The symbolic representation of each parameter is stored in the array **p** where the position corresponds to the order of the fields (first symbolic parameter corresponds to the first field of the structure). The symbolic parameters have the same name as the parameters in the structure.
The physical units of the components are stored as strings in the arrays **stateUnits**, **inputUnits**, **outputUnits** and **paramUnits** where the position corresponds to the arrays with the names. The units are mandatory even though the correctness of the strings as physical units cannot be checked.

Note that ALL properties are READ ONLY. For modification use the methods of the object.

For detailed information on each property use the MATLAB help function in the way "**help ODESCA_Object.PROPERTYNAME**" (E.g.: use "**help ODESCA_Object.param**" to get more information about how the parameters are stored.)

5.3.2 Methods

Name	Description
checkParam	Checks if all parameters are set to a value
getInfo	Creates structure with information about states, inputs and outputs
getParam	Returns the values and names of the parameters as cell arrays
isValidSymoblic	Checks if all symbolic variables are part of the object
setName	Sets the name of the instance
setParam	Sets a parameter to a scalar numeric value or to empty

For detailed information on each method use the MATLAB help function for the class or for the single method. E.g.: "**help ODESCA_Object**" or "**help ODESCA_Object.METHODNAME**"

5.4 ODE

ODE is short for ordinary differential equations. This class is the basic class for an ODESCA_Object.
The Class provides a system to handle parameters and methods to organize the equation system.
This class is used as a superclass.

5.4.1 Properties

Name	Description
f	Array with the symbolic equations for the state changes
g	Array with the symbolic equations for the outputs

The equations of the outputs are stored in the array g. The equations describing the change of the states (xdot) are stored in the array f. The protected properties are meant to changed inside a sub class in a way depending on the function of the subclass.

5.4.2 Methods

Name	Description
calculateNumericEquations	Calculates the numeric equations if all parameters are set
copyElement	Checks if all symbolic variables are part of the object
show	Shows the equations, states, inputs, outputs and parameters of the object
setAllParamAsInput	Sets the specified parameter as an input of the object
setParamAsInput	Sets the name of the instance
switchInputs	Switches two inputs
switchOutputs	Switches two outputs
switchStates	Switches two states
getSymbolicStructure	Gets the symbolic structure

5.5 Component

The class ODESCA_Component is a child of the class ODESCA_Object. It is used to create the nonlinear differential equations, inputs, outputs, states and parameters. The creation of these parts may depend on so called construction parameters defined in the component.

Instances of subclasses of the ODESCA_Component class may be added to systems (see ODESCA_System.addComponent()).
Note that the class ODESCA_Component itself is abstract so it cannot be initialized directly. Use the utility function ODESCA_Util.createNewComponentFile to create new custom components.

For detailed information on the creation of custom components see chapter Creation of a custom component in the Getting Started section.

If an instance of a subclass of the ODESCA_Component class (e.g.: the class “Pipe” is a subclass of “ODESCA_Component”) is created and has construction parameters, none of the fields except of the construction parameters are filled. The reason for this is the creation of the content depends on the construction parameters (e.g.: the number of nodes used to simulate the behavior of a pipe). To create the equations, states, inputs, outputs and parameters the construction parameters need to be set first.
After all construction parameters are set, the equations can be calculated. A component without construction parameters will be fully initialized on the creation of the instance.

Note the process of adding the component will fail if there are unset construction parameters.

This class inherits from ODESCA_Object. Therefore, it is passed by reference and can be copied.

5.5.1 Properties

Name	Description
constructionParam	Structure of parameters needed for the construction of the equations
FLAG_EquationsCalculated	Flag that determines if the equations have been calculated

The structure **constructionParam** stores all parameters which are needed in the process of creating the equations, inputs, outputs and states. E.g.: A pipe always contains the same equations but the number of states depends on the number of nodes the model uses. Therefore, if a pipe is being modeled, 'nodes' would be set as a construction parameter. These construction parameters have to be set before the equations are created.

The property **FLAG_EquationsCalculated** determines if the equations have been created already. It is set false before the calculation and changed to true afterwards.

Note that ALL properties are READ ONLY. For modification use the methods of the object.

For detailed information on each property use the MATLAB help function in the way "**help ODESCA_Component.PROPERTYNAME**" (E.g.: use "**help ODESCA_Component.constructionParam**" to find out more about how the construction parameters are stored.)

5.5.2 Methods

Name	Description
checkConstructionParam	Checks if all construction parameters are set
checkEquationsCorrect	Checks if all equations were set, true if they are set
setConstructionParam	Set a construction parameter to a numeric value
tryCalculateEquations	Calculates the Equations if all construction parameters are set

For detailed information on each method use the MATLAB help function for the class or for the single method. E.g.: "**help ODESCA_Component**" or "**help ODESCA_Component.METHODNAME**"

5.6 System

The class ODESCA_System is the most important class for the user. It is used to combine components into a system, to connect the equations of the components and most importantly to analyze the system with mathematical methods.

The system class is used directly by creating an instance of it. After initializing a new instance, one or more components can be added and connected. Furthermore, it is possible to add other systems.

By adding a component or system, all the equations, states, inputs, outputs and parameters are added to the existing arrays. By this means, changes inside an instance of a component, after adding to a system, will not affect the system.

Note that the name of a component is added to all states, inputs, outputs and parameters when the component is added to the system. This is necessary for the tool to work correctly and the prevention of name conflicts.

It is possible to rename a component after adding to a system, to avoid name conflicts. This is could be the case by combining two systems owning some components with the same name. Otherwise these components will be renamed.

The utility functions provide a possibility to create a nonlinear Simulink model from the system. For more information on utility functions see the chapter utilities.

A subpart of the system is the steady state. In this state the system is in an equilibrium which means with constant inputs the states are also constant:

$$\dot{x} = f(x_0, u_0) = 0\,!$$

The steady states are used for linear analysis of the nonlinear system. Each steady state has a reference to the system it belongs to and a system has a list of all its steady states. The steady states are represented in the class ODESCA_SteadyState.

Use the methods createSteadyState() to create a new steady state for a system. They can be deleted with the method removeSteadyState() or by calling the delete() method on the instance of a steady state.

The class ODESCA_System inherits from ODESCA_Object. Therefore, it is passed by reference and can be copied. Note that on creating a copy of the system, all steady states are copied and do not depend on the original instance of ODESCA_System.

5.6.1 **Properties**

Name	**Description**
components	Names of components which have been added to the system
defaultSampleTime	Default size of a time step for discrete systems
steadyStates	List of steady states linked to the system

The property **defaultSampleTime** is used as the sample time if a discrete system is created without providing a sample time. The array **components** stores the names of all components which have been added to the system. In the property **steadyStates** all steady states linked to the system are stored.

Note that ALL properties are READ ONLY. For modification use the methods of the object.

For detailed information on each property use the MATLAB help function in the way "**help ODESCA_System.PROPERTYNAME**" (E.g.: use "**help ODESCA_System.steadyStates**" to get more information about how the steady states are stored.)

5.6.2 **Methods**

Name	**Description**
addComponent	Adds a component to the system
addSystem	Adds the given system to the existing
calculateValidSteadyStates	Calculates all valid steady states and links them to the system
connectInput	Substitutes the chosen input with a symbolic expression
createControlaffineSystem	Creates a control-affine system out of the equations of the system
createMatlabFunction	Creates a Matlab functions out of the equations of the system
createNonlinearSimulinkModel	Creates a Simulink model of the ODESCA_System instance

createSteadyState	Creates a new ODESCA_SteadyState and links it to the system
createPIDController	Creates a PID controller for MIMO systems and links it to the nonlinear simulink model
equalizeParam	equalizeParam
findSteadyState	Method to find steady states, can retrun one, multiple or no results
removeOutput	Removes an output from the system
removeSteadyState	Removes a steady state and its link to the system
renameComponent	Renames a component within a system
setDefaultSampleTime	Sets the default sample time of the system
simulateStep	Simulates a step in the nonlinear system
symLinearize	Linearize the equations symbolically

For detailed information on each method use the MATLAB help function for the class or for the single method. E.g.: "**help ODESCA_System**" or "**help ODESCA_System.METHODNAME**"

5.7 Control Affine System

This class contains the control-affine representation of a nonlinear dynamic system. Using a coordinate neighborhood, a control-affine system has the form:

$$\dot{x} = f_0(x) + \sum_{i=1}^{m} f_i(x) u_i$$

If the system equations are already in the control-affine form, the property approxflag is set to false. Otherwise, first order low passes with small time constant are added in order to get a control affine approximation. In this case, the property approxflag is set to true.

This class can only be created with the method createControlAffineSystem() of the class ODESCA_System. An instance of this class belongs to an ODESCA_System instance at every time. If the instance is deleted, the instance of this class will be deleted too.

5.7.1 Properties

Name	Description
f0	Array with the part f0(x) of the symbolic equations for the state changes
f1	Array with the part f1(x) of the symbolic equations for the state changes
approxflag	Array with the Boolean variable that indicate if approximation was used
system	System that belongs to the created control-affine system

The array f0 stores the equation parts form the initialized system which contains no inputs. Hence f1 stores the parts which contains inputs. As mentioned earlier, if the inputs are nonlinear, an approximation with a first order low pass is signaled through the approxflag.

5.7.2 Methods

Methods for control-affine system are not implemented jet.

5.8 Steady State

A steady state of a system is a state, where all the inputs and states (and thereby outputs) are constant over time:

$$\dot{x} = f(x_0, u_0) = 0\,!$$

In this state a nonlinear system can be analyzed in many ways. Approximations of a system in a steady state (like linearization, bilinearization, etc) can be calculated. These approximations can be used to perform analysis of a system in the steady state. A list of all approximations in the steady state is stored in the property approximations. The steady state and its approximations have references of each other.

To create a steady state, the method createSteadyState(x0, u0) of the class ODESCA_System need to be used. It returns a new instance of the class steady state.

Logically a steady state cannot exist without an instance of the class ODESCA_System. Therefor if the system is deleted, the steady state will be deleted too.

Note: The steady state is completely numeric; hence all parameters of a system have to be set in order to create a steady state! To create a symbolic linearization of a system, see the method ODESCA_System.symLinearize().

The steady state has a flag called isStructuralValid. It determines the steady state matches the structure of the system. If the system has no changes after the steady state was created, it is set true. Otherwise the flag is set false. For an invalid steady state, no approximations can be made. Some of the methods of the approximations might not work with an invalid steady state.

The ODESCA_SteadyState is a subclass of the MATLAB class handle so the instances is passed by reference and can be stored in variables.

5.8.1 **Properties**

Name	**Description**
name	Name of the steady state
x0	Values of the states in the steady state
u0	Values of the inputs in the steady state
y0	Values of the outputs in the steady state
approximations	Array of the system approximations at the steady state
system	System instance the steady state refers to
param	Parameter set of the system the steady state refers to
structuralValid	Flag to determine if the steady state is structural valid
numericValid	Flag to determine if the steady state is numerical valid

The property **name** stores the name of the steady state. If it is not given while creation, the steady state designated a default name. The properties **x0**, **u0** and **y0** store the values of the states, inputs and outputs for which the system is in a steady state.

The approximations of a system in a steady state are stored in the property **approximations**. The approximations can be created with the corresponding methods. If the steady state is deleted, all approximations are deleted as well and therefore aren't useable anymore.

The property **system** is a reference to the ODESCA_System instance the steady state belongs to. The property param holds the parameter structure of the system the steady state refers to at the point the steady state was created. The property **structuralValid** determines if the system the steady belongs to has changed structural (e.g.: remove of an output, switch of equations). In this case some functions like the approximation are not possible anymore. The property **numericalValid** determines if the given values for x0, u0 and y0 lead to a steady state of the system. If the values are not correct

in a certain rang of precision, numerical problems can occur on the analysis of the steady state. A warning is thrown in this case.

Note that ALL properties are READ ONLY. For modification use the methods of the object.

For detailed information on each property use the MATLAB help function in the way "**help ODESCA_SteadyState.PROPERTYNAME**" (E.g.: use "**help ODESCA_SteadyState.H**" to find out more about how the transfer functions are stored.)

5.8.2 Methods

Name	Description
isNumericValid	Checks if the steady state is numerically valid for the system
setName	Sets the name of the steady state
delete	Custom delete() method which overwrites the default delete()

The methods for the approximations are listed below:

Name	Description
linearize	Calculate the linear approximation of the system in the steady state
linear	Returns the instances of the ODESCA_Linear class

For detailed information on each method use the MATLAB help function for the class or for the single method. E.g.: "**help ODESCA_SteadyState**" or "**help ODESCA_SteadyState.METHODNAME**"

5.9 Approximation

The approximation class serves as an interface for all approximations which can be created of a system in a steady state. It specifies the behavior for the deletion and the reference to the steady state it belongs to. An approximation cannot exist without a steady state.

The class is abstract and cannot be instanziated. It is a subclass of matlab.mixin.heterogeneous which means all subclasses which derive from approximation can be stored in the same array even so they may have different properties and methods.

5.9.1 Properties

Name	Description
steadyState	Steady State the approximation belongs to

The property **steadyState** is a reference to the steady state the approximation belongs to. Since an approximation cannot exist without a steady state, this property is always filled. If the steady state it refers to is deleted, this approximation is deleted too.

5.9.2 Methods

Name	Description
delete	Custom delete() method which overwrites the default delete()

The class approximation overwrites the default delete() behavior to make sure the delete method is called correctly. If the method is called at an approximation, the reference to the approximation is automatically removed from the steady state it belongs to.

5.10 **Linear**

The class Linear represents the linear approximation of a system in a steady state. It has the form

$$\vec{x} = \underline{A} \times \vec{x} + \underline{B} \times \vec{u}$$
$$\vec{y} = \underline{C} \times \vec{x} + \underline{D} \times \vec{u}$$

where the Matrices A, B, C and D are numeric.

The class provides a number of method for the linear analysis of the system like the plotting of bode- and nyquist plots or the analysis of stability, controllability and observability.

The linear class contains a state space object (ss) and a transfer function object (tf) of the control system toolbox which represent the same linearization. The methods of the class linear make use of the functionalities of the control system toolbox and extend them.

The methods of this class are designed to work with arrays of the class to make analysis and comparison as easy as possible.

The instances of the class have to belong to a steady state. If the steady state the instance belongs to is deleted, the instance is deleted too.

5.10.1 **Properties**

Name	**Description**
A	Time continuous system matrix
B	Time continuous input matrix
C	Output matrix
D	Feedthrough matrix
Ad	Time discrete system matrix
Bd	Time discrete input matrix
K	State feedback gain
L	Observer feedback gain
V	Precompensation matrix
form	Form of the system matrices (canonical forms or normal forms)
discreteSampleTime	Default sample time of the system the steady state refers to
ss	State space model of the linearization
tf	Transfer functions of the linearization

The properties **A**, **B**, **C** and **D** store the system matrices of a LTI system. **Ad** and **Bd** store the time discrete counterparts of the A and B. The property discreteSampleTime stores the sample time with which the discrete matrices Ad and Bd where calculated.

The property ss stores a state space object (ss) of the control system toolbox. It contains all information available in the system, the linearization's steady state belongs to and is set up for the time continuous linear system. The property **tf** stores a transfer function object (tf) of the control system toolbox.

The properties **K**, **L** and **V** store state feedback gain, observer feedback gain and the precompensation matrix. This are needed for create an observer or a full state feedback controller.

5.10.2 Methods

Name	Description
bodeplot	Plots the bode diagram for the linearization
createFSF	Creates a full state feedback controller
createKalmanFilter	Creates a kalman filter
createLQR	Creates a linear quadradic controller
createLuenbergerObserver	Creates a Luenberger observer
discretize	Calculates the discrete linear matrices
isAsymptoticStable	Checks if the linearizations are asymptotically stable
isControllable	Checks if the linearizations are observable
isObservable	Checks if the linearizations are controllable
nyquistplot	Plots the nyquist diagram for the linearization
stepplot	Plots the step response for the linearization
toCCF	Creates the controllable canonical form of the linear system
toOCF	Creates the observable canonical form of the linear system

NOTE: The methods of the class steady state are designed to work for arrays of the class! This is useful to compare linear approximations with little effort.

For detailed information on each method use the MATLAB help function for the class or for the single method. E.g.: "**help ODESCA_Linear**" or "**help ODESCA_Linear.METHODNAME**"

The method **bodeblot()** creates a bodeplot for the linearization's. It can take multiple name-value-pair arguments to specify the plotting behavior. Refer to Getting Started for an example on how to use bodeplot.

For the creation of a full state feedback controller refer to Example: State Controller for an Inverted Pendulum.

5.11 Bilinear

The linearized model mentioned is sometimes inadequate, and there is a growing need for a better approximation of the nonlinear system when robust behavior of nonlinear systems is required over the complete operating range. One method to deal with systems of this type

$$\dot{x} = f_0(x) + \sum_{i=1}^{m} g_i(x)u_i(t)$$
$$y(t) = g(x)$$

This class represents the bilinearization of a nonlinear dynamic system in a steady state. The bilinearisation is described with the matrices A, B, C, D, G, M and N. It has the form

$$\dot{\vec{x}} = \underline{A}\vec{x} + \sum_{i=0}^{m} u_i G_i \vec{u} + \sum_{i=0}^{m} u_i N_i \vec{x} + \sum_{i=0}^{n} x_i M_i \vec{x} + \underline{B}\vec{u}$$
$$\vec{y} = \underline{C}\vec{x} + \underline{D}\vec{u}$$

where m is the number of inputs and n is the number of states. The matrices A, B, C and D are equal to the matrices of a linear approximation.

The matrices G, N and M take into account if different states and/or inputs are multiplied with each other in the nonlinear system equations. Hence a multiplication of a state or an input with itself would represent a nonlinearity, the corresponding matrix entries are 0.

The instances of the class have to belong to a steady state. If the steady state the instance belongs to is deleted, the instance is deleted too.

5.11.1 Properties

Name	Description
A	Time continuous system matrix
B	Time continuous input matrix
C	Output matrix
D	Feedthrough matrix
G	Matrix for input-input bilinearity
N	Matrix for input-state bilinearity
M	Matrix for state-state bilinearity
discreteSampleTime	Default sample time of the system the steady state refers to

5.11.2 Methods

Methods for bilinear approximations are not implemented jet.

Refer to Example: Bilinear Estimator for a Plate Heat Exchanger for a detail view on how to use a bilinear estimator.

6 Utilities

In addition to the classes mentioned in the section above, the tool provides a number of utility functions which provide interfaces to other programs and some additional features. These features may have additional toolboxes or programs as requirement to be used. They are grouped in the class **ODESCA_Util** where the utility functions are static methods which can be called without creating an instance of the class.

For detailed help on all utility functions use the MATLAB help command (E.g.: for the function createNonlinearSimulinkModel type the command "helpDESAC_Util.createNonlinearSimulinkMode()")

List of all utility functions:

Name	Description	Requirement
createNonlinearSimulinkModel	Creates a Simulink model of the ODESCA_Systen instance	MATLAB Simulink
createNewComponentFile	Starts a dialog to create a new custom component file from a template	None
toPDF	Creates a .pdf which documents a ODESCA_Object class in latex style	MiKTeX (v2.9)

7 Example: Bilinear Estimator for a Plate Heat Exchanger

In this example, we will develop a bilinear estimator for a plate heat exchanger (PHEX) which is inspired by the work of Grunert et al. from 2015 (https://doi.org/10.1109/CCA.2015.7320673). You can see the scheme of the whole heating appliance in the following picture.

PHEX

The state we want to estimate is the outlet temperature of the plate heat exchanger, which is the Secondary Heat Exchanger in the picture. Designing an appropriate estimator would make the omission of the outlet temperature sensor possible. In order to achieve that, it is necessary to design an observer which can estimate all states accurately.

The estimator is based on a system model, which we need to set up first. In the interests of simplification, we can use a free body model which consists only of the PHEX, the sensors and the tubes which are located between the PHEX and the sensors.

7.1 Creation of the Components

The PHEX is already modeled as an ODESCA_Component. You can find it in the folder Examples->Components with the name OCLib_PlateHex. If you need help creating a custom component, please refer to our Getting Started Section above.

Having modeled the PHEX, we are able to create an instance of it.

```
PHEX_comp = OCLib_PlateHex('myPHEX')
```

As you should see in the Command Window, the properties of the created component are still quite empty. The reason is a so called construction parameter which needs to be predisposed here. This parameter is the number of nodes in which the PHEX is divided. In every node, the internal states such as temperature and massflow are considered locally constant. The number of nodes affects the number of states, therefore, this parameter needs to be set before MATLAB can create any equations. We will divide the PHEX in 3 nodes.

```
PHEX_comp.setConstructionParam('Nodes',3)
PHEX_comp % looking at the properties again
```

Now, looking at the properties again, you should see that the equations have been calculated for 6 states (3 temperatures on the primary and 3 on the secondary side). For more information about the properties, please refer to our Class Documentation Section above.

Take a look at all the properties of the component. The only missing fields are inside the property 'param'. In the next step we will complete the component by setting all the parameters to the values in the mentioned paper.

```
PHEX_comp.setParam('cHex', 500);
PHEX_comp.setParam('mHex', 1.154);
PHEX_comp.setParam('Volume1', 0.216*10^(-3));
PHEX_comp.setParam('Volume2', 0.24*10^(-3));
PHEX_comp.setParam('HexArea', 0.216);
PHEX_comp.setParam('RhoFluid', 998);
PHEX_comp.setParam('cFluid', 4182);
```

As you can read in the original paper, the thermal transmittance between primary and secondary side is given by a characteristic curve with three optimized parameters. This behavior is also already modeled as an ODESCA_Component in the Examples folder with the name OCLib_HeatTransferFit. We will create this component now and set the parameters. Later, we will take care of the connection between the components.

```
HTF_phex = OCLib_HeatTransferFit('myHeatTransferFit');
HTF_phex.setParam('c1', 6529);
HTF_phex.setParam('c2', 4029);
HTF_phex.setParam('c3', 3.29);
```

The model of the system also includes two temperature sensors (for outlet and return temperature) and a pipe between the PHEX and the return temperature sensor. You can find those components in the Examples folder, as well: OCLib_TSensor and OCLib_Pipe. The creation as well as the parameterization is done in the following.

```
OTSensor = OCLib_TSensor('myOTSensor');
OTSensor.setParam('TimeConst',1.8);
OTSensor.setParam('Gain',1);

RTSensor = OCLib_TSensor('myRTSensor');

PHEXtoRTSensor_Pipe = OCLib_Pipe('myPipe');
PHEXtoRTSensor_Pipe.setConstructionParam('Nodes',1);
PHEXtoRTSensor_Pipe.setParam('VPipe',0.55*1e-3);
% Note: There is no information given about the parameters cPipe and
mPipe but only about their product C_t.
%       Since these two parameters only appear as C_t in the equations,
we can simply set one of these parameters to 1 and the other to the value
of C_t.
PHEXtoRTSensor_Pipe.setParam('cPipe',1);
PHEXtoRTSensor_Pipe.setParam('mPipe',150);
```

As you might have noticed, we didn't set all the parameters of the new components yet. The reason is that those parameters already appeared in the some of the other components. We will use an ODESCA function to equalize those parameters later to avoid mistakes and reduce computation time.

7.2 Creation of the System

After we created and parameterized all the components, we can now create an ODESCA_System for our further work.
At first, we initialize the system with the PHEX and add all the components to it.

```
PHEX_sys = ODESCA_System('myPHEXSystem',PHEX_comp);
PHEX_sys.addComponent(HTF_phex);
PHEX_sys.addComponent(OTSensor);
PHEX_sys.addComponent(RTSensor);
PHEX_sys.addComponent(PHEXtoRTSensor_Pipe);
```

As already mentioned, we will equalize the parameters now.

```
PHEX_sys.param % take a look at the parameter list before equalizing
PHEX_sys.equalizeParam('myPHEX_cFluid',{'myPipe_cFluid'})
PHEX_sys.equalizeParam('myPHEX_RhoFluid',{'myPipe_RhoFluid'})
PHEX_sys.equalizeParam('myOTSensor_Gain',{'myRTSensor_Gain'})
PHEX_sys.equalizeParam('myOTSensor_TimeConst',{'myRTSensor_TimeConst'})
PHEX_sys.param % take a look at the parameter list after equalizing
```

If you take a look at the parameter list in the PHEX system, you should see that the parameters in the second arguments of the equalizing function have vanished. Those parameters have been replaced inside of the equations with the parameters in the first arguments of the equalizing function, respectively.

At this point, the system is completely parameterized, however, the components are not connected in any way but exist parallel and independently of one another inside the system. We will connect them by substituting in- and outputs of the respective components.

```
PHEX_sys.connectInput('myPipe_TempIn','myPHEX_Temperature1Out')
PHEX_sys.connectInput('myPipe_mDotIn','myPHEX_Massflow1Out')
PHEX_sys.connectInput('myOTSensor_TempIn','myPHEX_Temperature2Out')
PHEX_sys.connectInput('myRTSensor_TempIn','myPipe_TempOut')
PHEX_sys.connectInput('myPHEX_Massflow1In','myHeatTransferFit_Massflow1Ou
t')
PHEX_sys.connectInput('myPHEX_Massflow2In','myHeatTransferFit_Massflow2Ou
t')
PHEX_sys.connectInput('myPHEX_kHex','myHeatTransferFit_k_Hex')
```

The four remaining inputs of the system are the two massflows and the two temperatures of the primary and secondary side of the PHEX. However, we can still see all of the outputs of the components, though, in the real system, we are only able to measure the return and the outlet temperature using the two sensors. To get a model of the real system, we can remove all unwanted outputs as follows.

```
PHEX_sys.removeOutput('myPHEX_Temperature1Out')
PHEX_sys.removeOutput('myPHEX_Temperature2Out')
PHEX_sys.removeOutput('myPHEX_Massflow1Out')
PHEX_sys.removeOutput('myPHEX_Massflow2Out')
PHEX_sys.removeOutput('myHeatTransferFit_k_Hex')
PHEX_sys.removeOutput('myHeatTransferFit_Massflow1Out')
PHEX_sys.removeOutput('myHeatTransferFit_Massflow2Out')
PHEX_sys.removeOutput('myPipe_TempOut')
PHEX_sys.removeOutput('myPipe_mDotOut')
PHEX_sys % take a look at the system now
```

If you take a look at the system now, you should see 9 states, 4 inputs, 2 outputs and 15 parameters.

7.3 Bilinearization of the System

For the creation of a bilinear observer, we need a bilinear approximation of the nonlinear system. The approximation is only accurate close to the center of the series. We want to bilinearize the system at the most common steady state of the system, which we assume as the following input u0 along with the resulting state x0.

```
u0 = [60; 10; 0.25; 0.15];
```

The resulting state x0 will be calculated in the following using the system equations. We also want to calculate y0 for setting up the bilinear observer later.

```
[PHEX_sys_f,PHEX_sys_g] = PHEX_sys.calculateNumericEquations();
steadystate = vpasolve(subs(PHEX_sys_f,PHEX_sys.u,u0),PHEX_sys.x);
x0 = double([steadystate.x1; steadystate.x2; steadystate.x3;
steadystate.x4; steadystate.x5; steadystate.x6; steadystate.x7;
steadystate.x8; steadystate.x9]);
y0 = double(subs(PHEX_sys_g,PHEX_sys.x,x0));
```

Now, we can create an ODESCA_SteadyState which is the base for the creation of a system approximation such as the bilinearization. If (x0,u0) is no valid steady state, the creation of an ODESCA_SteadyState will display a warning. The function _bilinearize_ finally creates an instance of the class ODESCA_Bilinear.

```
ss = PHEX_sys.createSteadyState(x0,u0,'mySteadyState');
PHEX_sys_bilin = ss.bilinearize();
```

All the different classes (ODESCA_Component, ODESCA_System, ODESCA_SteadyState, ODESCA_Bilinear) are still connected to each other so that it is always distinct, which system belongs to which approximation et cetera. For example

```
PHEX_sys_bilin.steadyState.system.components
```

will give you the names of all components linked to the system.

7.4 Creation of the Estimator

For the common LQE (linear quadratic estimation) approach, only the linear parts of the system are considered to calculate the feedback matrix L. The weights Q and R are chosen as unity matrices with decreased values of R to achieve higher observer gains. The lqr function of matlab can be applied to the dual system for calculating the observer gain.

```
Q = eye(9,9);
R = eye(2,2)*0.01;
L = (lqr(PHEX_sys_bilin.A',PHEX_sys_bilin.C',Q,R))';
```

To finally see the results, you can create a Simulink model with the nonlinear system and the bilinear observer. The creation of the nonlinear system is done by the following command.

```
PHEX_sys.createNonlinearSimulinkModel
```

However, we already created the whole test system for you in Examples->Systems->Estimation_PHEX. Note, that you need to execute the previous commands in order to run the Simulink model.

As you can see, for different initial states the simple bilinear observer estimates all states accurately when the temperatures perform steps. Even when the sensors signals are disturbed, all temperatures inside the PHEX are nearly stationary accurate. The stationary accuracy of the outputs could be achieved by using a reduced observer. The only problem of this observer can be detected when the massflows perform steps. This issue is tackled in the mentioned paper.

8 Example: State Controller for an Inverted Pendulum

In this example we will develop a state controller for an inverted pendulum. The inverted pendulum is an example commonly found in textbooks and research literature. Its popularity derives in part from the fact that it is unstable without control. Furthermore, the dynamics of the system are nonlinear. The task of the state controller is to balance the pendulum, by applying a force to the cart that the pendulum is attached to.
In order to develop a state controller, a system is needed.

8.1 Creation of the Components

The pendulum is already modeled as an ODESCA_Component. It is placed in the folder Examples->Components with the name OCLib_Pendulum. More information's on how to create a custom component, please refer to out Getting Started section.

Having modeled the pendulum, we are able to create an instance of it.

```
MyPendulum = OCLib_Pendulum('Pendulum');
```

In the next step we will complete the component by setting all parameters. M0 and M1 describe the mass of the cart respectively the weight at the end of the pendulum. For these parameters we set 4 kg and 0.36 kg. The length to pendulum center of mass l_s is 0.451 m. Theta, mass moment of inertia of the pendulum, is set to 0.08433 kg*m^2. 10 kg/s we set for the speed proportional friction constant Fr. For the gravity g and the coefficient of friction for pendulum we set 9.81 m/s^2 and 0,00145 kgm^2/s.

It should look similar to this:

```
MyPendulum.setParam('M0',4);
MyPendulum.setParam('M1',.36);
MyPendulum.setParam('l_s',.451);
MyPendulum.setParam('theta',.08433);
MyPendulum.setParam('Fr',10);
MyPendulum.setParam('C',.00145);
MyPendulum.setParam('g',9.81);
```

8.2 Creation of the System

After we created and parametrized all the components, we can now create an ODESCA_System. To initialize the system with the pendulum, use the following expression:

```
PendulumSys = ODESCA_System('MyPendulumSys',MyPendulum);
```

For a simpler model of the pendulum, we remove the output 'Pendulum_Angle'. Therefor we use the following command:

```
PendulumSys.removeOutput('Pendulum_Angle');
```

8.3 Linearization of the System

For the creation of a state controller, we need to linearize the modeled system of the pendulum. At first we create a steadystate and afterwards the linearization. The approximation is only accurate close to the center of the series. To create a steadystate we assume the input u0 as [0,0,0,0]. The following command creates the steadystate around the input u0.

```
ss1 = PendulumSys.createSteadyState([0,0,0,0],0,'ss1');
```

As mention earlier, the next step is to linearize. In ODESCA it is quite simple to do so. We use the steadystate stored in ss1 on the command linearize as showing here.

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

8.4 Creation of the State Controller

For creating a state controller, we use the following function.

```
sys_lin.createFSF();
```

This function creates a full state feedback controller with precompastion using the state space model of the linearized system calculated before. The feedback matrix K and the prefilter matrix V are calculated inside the function. Finally, the nonlinear system including the controller is created in Simulink.

However, we already created the whole system for you in Examples->Systems->State_Controller_Pendulum. Note, that you need to run the created script in order to run the Simulink model.