****

ODESCA User-Guide



**Contents**

[1 Introduction 3](#_Toc498013867)

[2 Requirements 4](#_Toc498013868)

[2.1 Required licenses 4](#_Toc498013869)

[3 License 5](#_Toc498013870)

[4 Basic Tool structure 6](#_Toc498013871)

[4.1 Class structure 6](#_Toc498013872)

[4.2 Short description of the classes 6](#_Toc498013873)

[4.3 Utilities 8](#_Toc498013874)

[5 Typical Workflow 9](#_Toc498013875)

[5.1 Creation of a custom component 9](#_Toc498013876)

[5.2 Create an instance of a custom component and parameterize it 11](#_Toc498013877)

[5.3 Create a system and connect all components 12](#_Toc498013878)

[5.4 Analyzing the system 13](#_Toc498013879)

[6 Class Documentation – Properties and Methods 15](#_Toc498013880)

[6.1 BaseClass 15](#_Toc498013881)

[6.2 Object 15](#_Toc498013882)

[6.3 Component 17](#_Toc498013883)

[6.4 System 17](#_Toc498013884)

[6.5 Steady State 18](#_Toc498013885)

[6.6 Approximation 19](#_Toc498013886)

[6.7 Linear 19](#_Toc498013887)

[6.8 Bilinear 20](#_Toc498013888)

# Introduction

For Model-based control algorithm development and algorithm parameterization, algebraic models of the plants have a lot of advantages over simulation models. With algebraic models, a multitude of mathematical approaches can be used for optimization and analytical purposes.

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 analysis of ordinary differential equation systems in MATLAB.

With ODESCA, dynamical systems of the form

xdot = f(x,u)

y = g(x,u)

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

The name ODESCA is an acronym for “**O**rdinary **D**ifferential **E**quation **S**ystems: **C**reation and **A**nalysis”.

# 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.

## Required licenses

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

# License

ODESCA is free software: you can redistribute it and/or modify

it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or

(at your option) any later version.

ODESCA is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License

along with ODESCA. If not, see <http://www.gnu.org/licenses/>.

Copyright 2017 Tim Grunert, Christian Schade, Lars Brandes, Sven Fielsch, Claudia Michalik, Matthias Stursberg

# Basic Tool structure

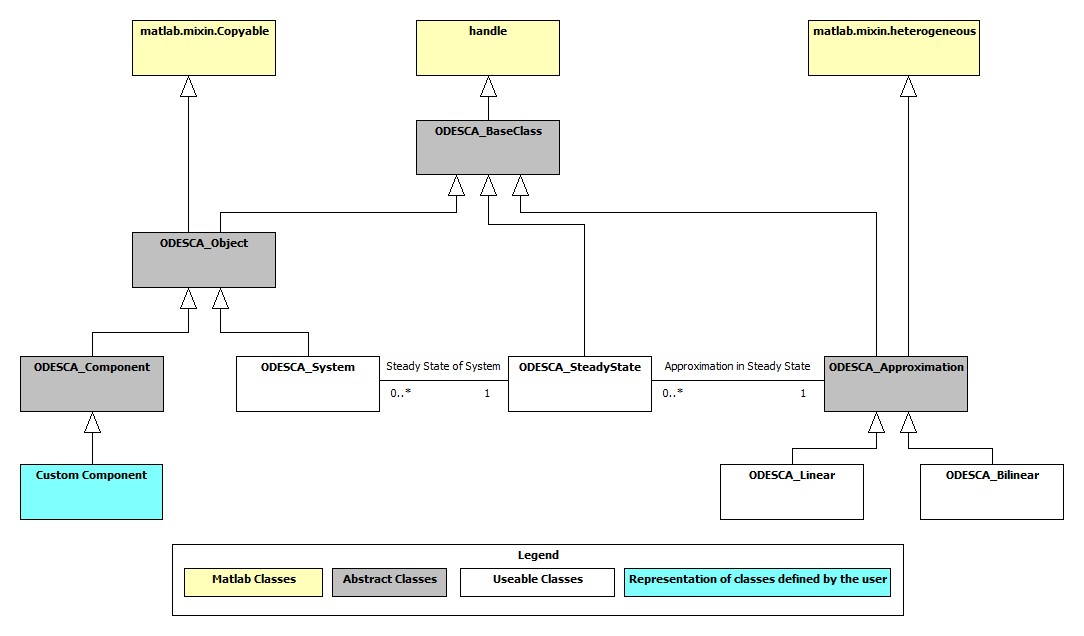
## Class structure

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 1: Class Diagram



## Short description of the classes

* **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 this class provides a system to handle the parameters in the equations and methods to organize the equation system. The classes Component and System are subclasses of Object. Therefore access to all public methods of the object class is granted.

* **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 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 on the utilities, see section **3.3**.

* **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 by calling the class constructor.

* **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*cannot exist without its *System*, so if the system is deleted, the SteadyState is deleted too.

* **Approximation:**

All approximations which are attached to a SteadyState has to be subclasses of the class *Approximation*which mainly serves as an interface to ensure that all approximations provide certain methods and properties. An *Approximations* cannot exist without a *SteadyState*, so if the SteadyState is deleted, the approximations is deleted too.

* **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.

Note that all class names of the tool ODESCA start with the name “ODESCA\_”. E.g.: the system class is called “ODESCA\_System”

## Utilities

In addition to the classes mentioned in the section above, the tool provides 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.

List of all utility functions:

|  |  |  |
| --- | --- | --- |
| Name | Description | Requirement |
| createNewComponentFile | Starts a dialog to create a new custom component file from a template | Keine |
| toPDF | Creates a .pdf which documents a ODESCA\_Object class in latex style | MiKTeX (v2.9) |

# Typical Workflow

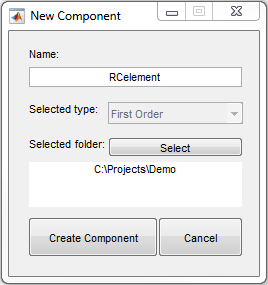
To use ODESCA, it’s important to understand that object-oriented base. Because everything is made out of classes, you can use them in the console after creation. The data can be manipulated with the methods and the instances can be passed to functions.

The workflow in a new project typically follows these 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 three steps are described in more detail with examples.

## 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. A blueprint for these custom components can be found in the folder Library/Components in the file ODESCA\_Component\_Template.

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 and open the file after the creation.

|  |
| --- |
| ODESCA\_Util.createNewComponentFile() |

Figure 2: Dialog New Component

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.

The first section is the “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 (E.g.: for the RC-element no construction parameters are needed. So the array and the parameter area would look like below.).

|  |
| --- |
| %==============================================================  %% DEFINITION OF CONSTRUCTION PARAMETERS (User editable)  %==============================================================    constructionParamNames = {};    %============================================================== |

If no construction parameters are needed, the array have 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 super class ODESCA\_Component. To access these parameters use the command

“obj.constructionParam.PARAMNAME” where PARAMNAME is the name given to the construction parameters in the first section.

In the second section called “Definition of equations components” 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 MUST have at least one output! The other arrays can be empty if no states, no inputs or no parameters are required.

For the example 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'};    % ============================================================= |

The third section which is called “Definition of equations” is the part where the equations for the state changes (f) and the equations for the outputs (g) of the component are created.

To define these equations, the arrays f and g of the super 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) = where NUM is the number of the input in the arrays stateNames and inputNames.

Make sure that the number of equations matches the number of states and the number of outputs defined in the second section.

The 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. The equations must not contain other symbolic variables which are not states, inputs or parameters! 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.p(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”.

## Create an instance of a custom component and parameterize it

After the creation of the custom component class files, instances of these classes can be created. 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.

## 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). You can now use the arguments “name” and “comp” to specify the name of the system and the first component added to it or you can just create an empty system.

|  |
| --- |
| sys = ODESCA\_System('RCsystem', RC) |

After creating a new system, components can be added by using the addComponent(comp) method. You have to pass the component which should be added to the system as an 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 which input should be replaced. The argument “connection” is the thing the input is replaced with. 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.: if you want to connect the input of the second RC element called “RC2\_Voltage\_in” with the output of the first RC element called “RC1\_Voltage\_out” you can use the following command:

|  |
| --- |
| sys.connectInput('RC2\_Voltage\_in', 'RC1\_Voltage\_out') |

If there are outputs you don’t want to appear in the system or the model, you can use the removeOutput(toRemove)where 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() |

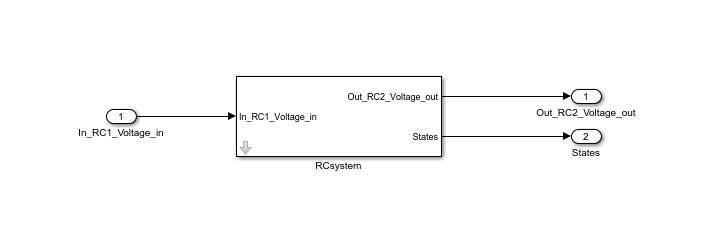


Figure 3: Nonlinear Simulink model

For more clarification, scripts with detailed comments that create systems can be found in the folder

“Library/Systems/Example”.

## 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 was 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.

|  |
| --- |
| linearize(SteadyState) |

If there is more than one steady state to a system, the function linearize()can be called on the whole array of steady states.

To get all linearization of the steady states, use the method linear() which returns the linearizations 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() |

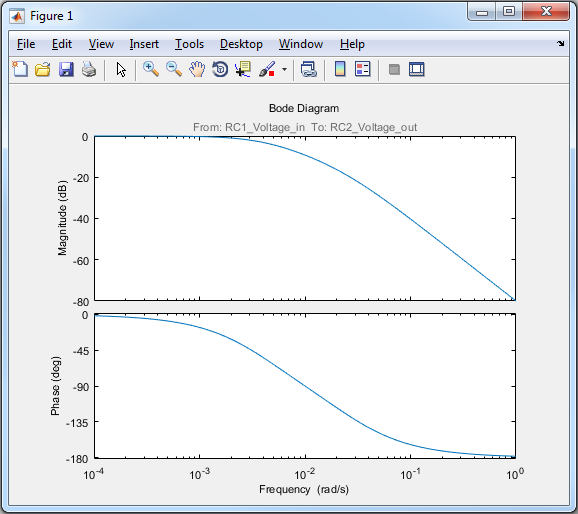


Figure 4: Bode plot of the steady state

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.

|  |
| --- |
| lin.isAsymptoticStable() |

# 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 because all protected and private methods can and must not be used by the user. All properties are READ ONLY and can only be modified within the methods of the classes.

A drawing on the classes can be found in chapter 3.

For detailed information on each property and each method use the MATLAB help function in the way   
“**help** **ODESCA\_CLASSNAME.PROPERTYNAME**” or “**help** **ODESCA\_CLASSNAME.METHODNAME**”.

E.g.: use “help ODESCA\_Object.param” to find out more about how the parameters are stored.

For detailed informations on the classes of ODESCA see “ODESCA-ClassDocumentation”.

## BaseClass

The BaseClass, who could guess, is the base for all classes in ODESCA. It is abstract so it 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.

* **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 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.

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

## 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, parameters and the equations themselves. It provides methods to modify the parameters and equations of the object (e.g. switch the order of inputs) 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.

* **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 |
| f | Array with the symbolic equations for the state changes |
| g | Array with the symbolic equations for the outputs |
| 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 |

* **Methods**

|  |  |
| --- | --- |
| Name | Description |
| calculateNumericEquations | Calculates the numeric equations if all parameters are set |
| 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 |
| getSymbolicStructure | Creates a structure with the symbolic variables of the system |
| isValidSymoblic | Checks if all symbolic variables are part of the object |
| show | Shows the equations, states, inputs, outputs and parameters of the object |
| setAllParamAsInput | Sets all parameters as inputs of the object |
| setName | Sets the name of the instance |
| setParam | Sets a parameter to a scalar numeric value or to empty |
| setParamAsInput | Sets the specified parameter as an input of the object |
| switchInputs | Switches two inputs |
| switchOutputs | Switches two outputs |
| switchStates | Switches two states |

## 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.

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 4.1 “Creation of a custom component” in this documentation.

If an instance of a subclass of the ODESCA\_Component class is created and has construction parameters, none of the fields except of the construction parameters are filled. This is because the creation of the content depends on the construction parameters (e.g.: in a pipe the number of nodes used to simulate the behavior of a pipe). So to create the equations, states, inputs, outputs and parameters the construction parameters have to be set! 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 that it is not necessary to call this method before a component is added to a system but the process of adding the component will fail if there are unset construction parameters.

* **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 |

* **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 |

Note, that the Methods of the ODESCA\_Object are also available for the ODESCA\_Component.

## 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.

By adding a component or system, all the equations, states, inputs, outputs and parameters are added to the existing arrays. This means, if there has been a change inside an instance of a component after it was added to a system, the change 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.

* **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 |

* **Methods**

|  |  |
| --- | --- |
| Name | Description |
| addComponent | Adds a component to the system |
| addSystem | Adds the given system to the existing |
| connectInput | Substitutes the chosen input with a symbolic expression |
| 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 |
| 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 |

Note, that the Methods of the ODESCA\_Object are also available for the ODESCA\_System.

## Steady State

A steady state of a system is a state, where all the inputs and states (and therefore outputs) are constant over time: xdot = f(x0,u0) = 0 !

In this state a nonlinear system can be analyzed in many ways. With this class, approximation 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.

To create a steady state, the method createSteadyState(x0, u0) of the class ODESCA\_System has to be called. It returns a new instance of the class steady state.

A steady state cannot exist without an instance of the class ODESCA\_System. The steady state is always attached to the system and the system is always attached to the steady state.

If the system is deleted, the steady state will be deleted too.

Note that the steady state is completely numeric, which means 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().

* **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 |

* **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 |

## 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 instantiated. All subclasses which derive from approximation can be stored in the same array even so they may have different properties and methods.

* **Properties**

|  |  |
| --- | --- |
| Name | Description |
| steadyState | Steady State the approximation belongs to |

* **Methods**

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

## 

## Linear

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

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

The linear 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 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.

* **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 |
| 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 |

* **Methods**

|  |  |
| --- | --- |
| Name | Description |
| bodeplot | Plots the bode diagram for the linearization |
| nyquistplot | Plots the nyquist diagram for the linearization |
| stepplot | Plots the step response for the linearization |
| isAsymptoticStable | Checks if the linearizations are asymptotically stable |
| isObservable | Checks if the linearizations are controllable |
| isControllable | Checks if the linearizations are observable |
| discretize | Calculates the discrete linear matrices |

## Bilinear

The class Bilinear represents the bilinear approximation of a system in a steady state. It has the form

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.

* **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 |

* **Methods**

No public methods for the class are implemented yet.