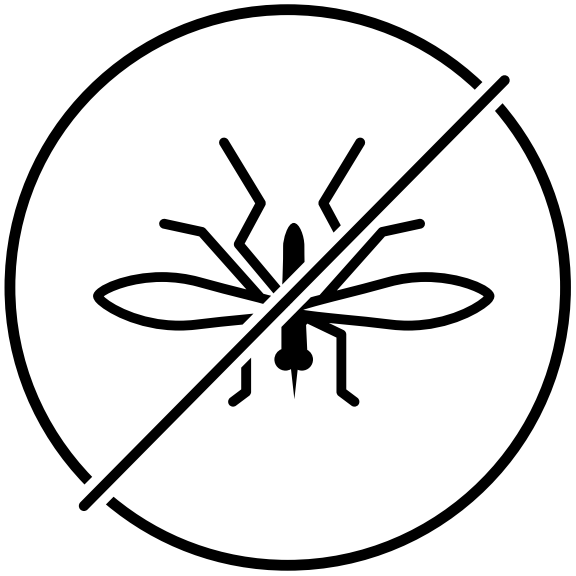

PyMoskito Documentation

Release 0.3.0

Stefan Ecklebe

Oct 04, 2018

Contents

Contents:

CHAPTER 1

Introduction

1.1 What is PyMoskito ?

PyMoskito aims to be a useful tool for students and researchers in the field of control theory that performs repetitive task occurring in modelling as well as controller and observer design.

The toolbox consists of two parts: Part one -the core- is a modular simulation circuit whose parts (Model, Controller and many more) can be easily fitted to one's personal needs by using one of the "ready to go" variants or deriving from the more powerful base classes.

To configure this simulation loop and to analyse its results, part two -the frontend- comes into play. The graphical user interfaces not only allows one to accurately tune the parameters of each part of the simulation but also allows to automate simulation runs e.g. to simulate different combinations of modules or parameters. This batch-like interface is feed by human readable yaml files which make it easy to store and reproduce simulated setups. Furthermore PyMoskito offers possibilities to run postprocessing on the generated results and easily lets you create plots for step responses.

1.2 What is PyMoskito not ?

Although the simulation loop is quite flexible it is **not** a complete block oriented simulation environment for model based-design but ideas for the development in this direction are always appreciated.

CHAPTER 2

Installation

2.1 General Options

At the command line:

```
$ pip install pymoskito
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv pymoskito
$ pip install pymoskito
```

From the repository:

```
$ git clone https://github.com/cklb/pymoskito
$ python setup.py install
```

2.2 For Windows

PyMoskito depends on Qt5 and VTK .

Qt5 is already included in the most python distributions, to have an easy start we recommend to use Winpython .

The wheel for the *VTK* package (Version >= 7) can be obtained from http://www.lfd.uci.edu/~gohlke/pythonlibs/#vtk . It can be installed using the Winpython Control Panel or directly via:

```
$ pip install VTK-VERSION_NAME_HERE.whl
```

from your winpython shell.

2.3 Troubleshooting

Missing dependencies (windows)

If the provided packages on your system are to old, pip may feel obligated to update them. Since the majority of packages now provide ready to use wheels for windows on pypi this should work automatically. If for some

reason this process fails, you will most certainly find an appropriate wheel here . After downloading just navigate your shell into the directory and call:

```
$ pip install PACKAGE_NAME.whl
```

to install it.

Missing vtk libraries (linux)

If importing `vtk` fails with something similar to:

```
>>> Import Error: vtkIOAMRPython module not found
```

then look at the output of:

```
$ ldd PATH/TO/SITE-PKGS/vtk/vtkIOAMRPython.so
```

to see which libs are missing.

CHAPTER 3

Tutorials

Below you will find some lessons on different aspects of the toolbox, sorted from basic to expert level.

Beginner:

3.1 Beginners Tutorial

Welcome to the PyMoskito Tutorial! It is intended to introduce new users to the toolbox. For more detailed descriptions, please see the *Users Guide* or the *Modules Reference*.

Within this tutorial, an inverse pendulum on cart will be simulated and stabilized. With the help of PyMoskito, the model as well as the controller will be tested by simulating the open and closed control loop.

All code is written in Python. If you want to refresh or expand your knowledge about this language, see e.g. the Python Tutorial.

3.1.1 PyMoskito's Signal Diagram

PyMoskito simulates the control loop as shown in Fig. 3.1. This tutorial will focus on the part highlighted in blue, since these modules are essential to run the toolbox.

Every block in this diagram represents a configurable part of the control loop that is implemented as a generic base class. By deriving from these base classes, it is easy to make sure that implemented classes work well within the context of the toolbox.

From the highlighted classes, the trajectory generator and the model mixer are considered reusable, therefore PyMoskito provides these classes ready to go. On the other hand, the model and the controller are determined by the specific system and have to be implemented to suit your problem. If you would like to implement one of the nonhighlighted classes, see the *Users Guide* or other *Tutorials* for help.

Next up the system, used for implementation is introduced.

3.1.2 A Pendulum on Cart

A pendulum is fixed on a cart, which can move in the horizontal direction. The cart has a mass M. The friction between the cart and the surface causes a frictional force $F_R = D \cdot \dot{s}$, in opposite direction to the movement of the cart. The pendulum has a mass m, a moment of intertia J, a length l and an angle of deflection φ. The friction in

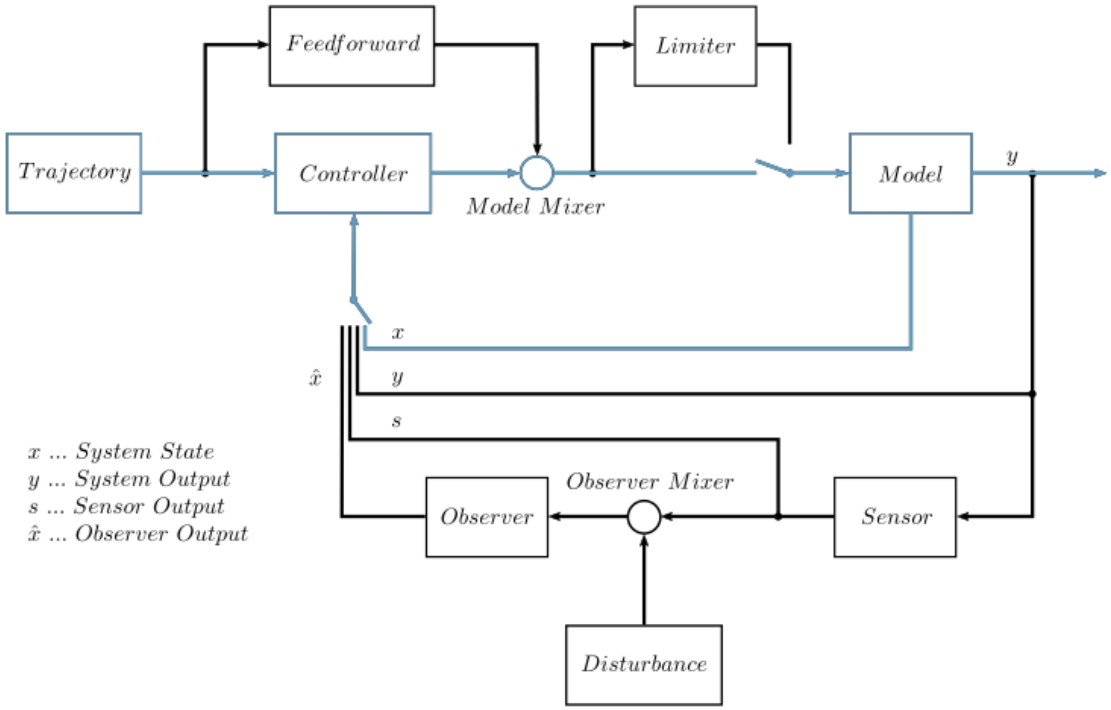

Fig. 3.1: The control loop implemented by PyMoskito

the joint where the pendulum is mounted on the cart causes a frictional torque $M_R = d \cdot \dot{\varphi}$, in opposite direction to the rotational speed of the pendululm. The system is illustrated in Fig. 3.2.

The task is to control the position s of the cart and to stabilize the pendulum in its downside position. The possibility of stabilizing the pendulum in its upside position is not implemented in this tutorial. Actuating variable is the force F.

With the state vector

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} s \\ \varphi \\ \dot{s} \\ \dot{\varphi} \end{pmatrix},$$

the model equations are given by

$$\dot{\boldsymbol{x}} = \begin{pmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \\ \dot{x_4} \end{pmatrix} = \begin{pmatrix} x_3 \\ x_4 \\ \frac{JF - JDx_3 - mlJx_4{}^2 \sin(x_2) + m^2 l^2 g \sin(x_2)\cos(x_2) - mldx_4 \cos(x_2)}{(M+m)J - (ml\cos(x_2))^2} \\ \frac{ml\cos(x_2)F - mlDx_3\cos(x_2) - (mlx_4)^2 \sin(x_2)\cos(x_2) + (M+m)mlg\sin(x_2) - (M+m)dx_4}{(M+m)J - (ml\cos(x_2))^2} \end{pmatrix}.$$

The cart's position

$$y = x_1 = s$$

is chosen as output of the system. With this model given, the next step is to implement a class containing these equations.

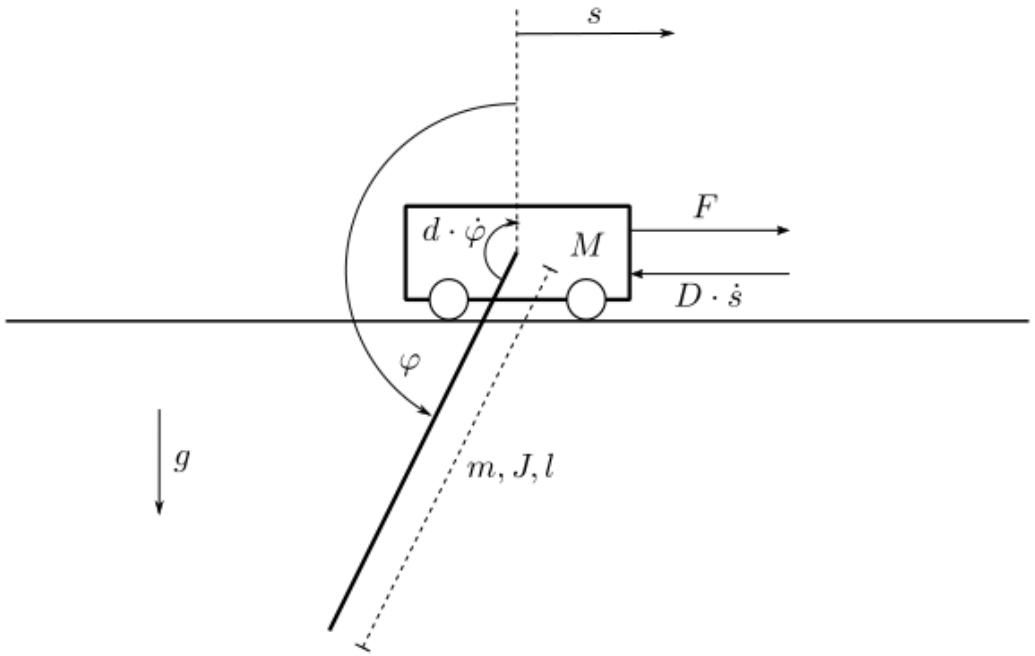

Fig. 3.2: The pendulum on an cart system

3.1.3 Implementing a Model

At first, a new class derived from the abstract class *Model* is necessary. Its basic functions will be calculating the state derivatives and the output from the model parameters, the current state and the input values.

Create a folder within a path of your choice. All files created during this tutorial need to be stored here. Create a file called:

```
model.py
```

With the first lines of code, import the library NumPy, the OrderedDictionary class and PyMoskito itself:

```
# -*- coding: utf-8 -*-
from collections import OrderedDict
import numpy as np

import pymoskito as pm


```

Derive your class from *Model*. Next, create an `OrderedDict` called `public_settings`. All entries in this dictionary will be accessible in the graphical interface of PyMoskito during runtime. While you have the freedom to name these entries as you like, the entry `initial state` is obligatory and must contain the initial state vector. All values entered will be the initial values for the model parameters:

```
class PendulumModel(pm.Model):
    public_settings = OrderedDict([("initial state", [0, 180.0, 0, 0]),
                                   ("cart mass", 4.3),  # [kg]
                                   ("cart friction", 10),  # [Ns/m]
                                   ("pendulum mass", 0.32),  # [kg]
                                   ("pendulum inertia", 0.07),  # [kg*m^2]
                                   ("pendulum friction", 0.03),  # [Nms]
                                   ("pendulum length", 0.35),  # [m]
```

(continues on next page)

(continued from previous page)

```
17                                      ("gravity", 9.81)])   # [m/s^2]
18
```

Within the constructor, you must define the number of inputs and states. Do so by storing these values in settings as seen in lines `24` and `25`. Adding output information as seen in line `26` is optional, this will make it easier to distinguish between several outputs of bigger systems. It is obligatory to call the constructor of the base class at the end. The constructor's argument `settings` is a copy of `public_settings` with all changes the user made in the interface:

```
20     def __init__(self, settings):
21         # conversion from degree to radiant
22         settings["initial state"][1] = np.deg2rad(settings["initial state"][1])
23         settings["initial state"][3] = np.deg2rad(settings["initial state"][3])
24
25         # add specific "private" settings
26         settings.update(state_count=4)
27         settings.update(input_count=1)
28         settings.update({"output_info": {0: {"Name": "cart position",
29                                              "Unit": "m"}}})
30         pm.Model.__init__(self, settings)
31
```

The calculation of the state derivatives takes place in a method that returns the results as an array. The method's parameters are the current time `t`, the current state vector `x`, and the parameter `args`. The later is free to be defined as you need it, in this case it will be the force `F` as the model input. To keep the model equations compact and readable, it is recommended to store the model values in variables with short names:

```
33     def state_function(self, t, x, args):
34         # definitional
35         s = x[0]
36         phi = x[1]
37         ds = x[2]
38         dphi = x[3]
39         F = args[0]
40
41         # shortcuts for readability
42         M = self._settings["cart mass"]
43         D = self._settings["cart friction"]
44         m = self._settings["pendulum mass"]
45         J = self._settings["pendulum inertia"]
46         d = self._settings["pendulum friction"]
47         l = self._settings["pendulum length"]
48         g = self._settings["gravity"]
49
50         dx1 = ds
51         dx2 = dphi
52         dx3 = ((J * F
53                 - J * D * ds
54                 - m * l * J * dphi ** 2 * np.sin(phi)
55                 + (m * l) ** 2 * g * np.sin(phi) * np.cos(phi)
56                 - m * l * d * dphi * np.cos(phi))
57                / ((M + m) * J - (m * l * np.cos(phi)) ** 2))
58         dx4 = ((m * l * np.cos(phi) * F
59                 - m * l * D * ds * np.cos(phi)
60                 - (m * l * dphi) ** 2 * np.cos(phi) * np.sin(phi)
61                 + (M + m) * m * l * g * np.sin(phi) - (M + m) * d * dphi)
62                / ((M + m) * J - (m * l * np.cos(phi)) ** 2))
63
64         dx = np.array([dx1, dx2, dx3, dx4])
65         return dx
66
```

The output of the system is calculated in a method with the current state vector as parameter. Returning the results as an array as previously would be possible. But in this case, the output is simply the position `s` of the cart, so extracting it from the state vector and returning it as a scalar is sufficient :

```
68      def calc_output(self, input_vector):
69          return input_vector[0]
```

This now fully implemented model class has a yet unknown behavior. To test it, the next step is to start PyMoskito for simulation purposes.

3.1.4 Starting the Program

For PyMoskito to start, an application needs to launch the toolbox and execute it. To do so, create a file in the same directory as the model and name it:

```
main.py
```

Copy the following code into your main file:

```
1  # -*- coding: utf-8 -*-
2  import pymoskito as pm
3
4  # import custom modules
5  import model
6
7
8  if __name__ == '__main__':
9      # register model
10     pm.register_simulation_module(pm.Model, model.PendulumModel)
11
12     # start the program
13     pm.run()
```

Note the `import` command in line `5`, which includes the earlier implemented model file in the application. The command in line `10` registers the model to the toolbox. This lets PyMoskito know that this module is available and adds it to the eligible options in the interface. Line `13` finally starts our application.

Use the command line to navigate to the directory of the main file and the model file and start the toolbox with the command:

```
$ python main.py
```

The upstarting interface of PyMoskito gives you the possibility to test the implemented model in the next step.

3.1.5 Testing the Model

Choose initial states that make the prediction of the system's reaction easy and compare them with the simulation results. After successfully starting the program, you will see the interface of the toolbox as shown in Fig. 3.3.

Within the Properties Window (1), double clicking on a value (all `None` by default) activates a drop down menu. Clicking again presents all eligible options. One of these options now is `PendulumModel`, since it was registered to PyMoskito earlier. Choose it now and press enter to confirm your choice.

By clicking on the arrow that appeared on the left of `Model`, all model parameters and the initial state are displayed. These are taken from the `public_settings` which have been defined earlier in the model. Double click on a value to change it manually. Press enter to confirm the input.

Choose the `PendulumModel`, the `ODEInt` as `Solver` and the `AdditiveMixer` as `ModelMixer`. Change the initial state of `Model` to [0, 100, 0, 0] and the end time of `Solver` to 20 as shown in Fig. 3.4.

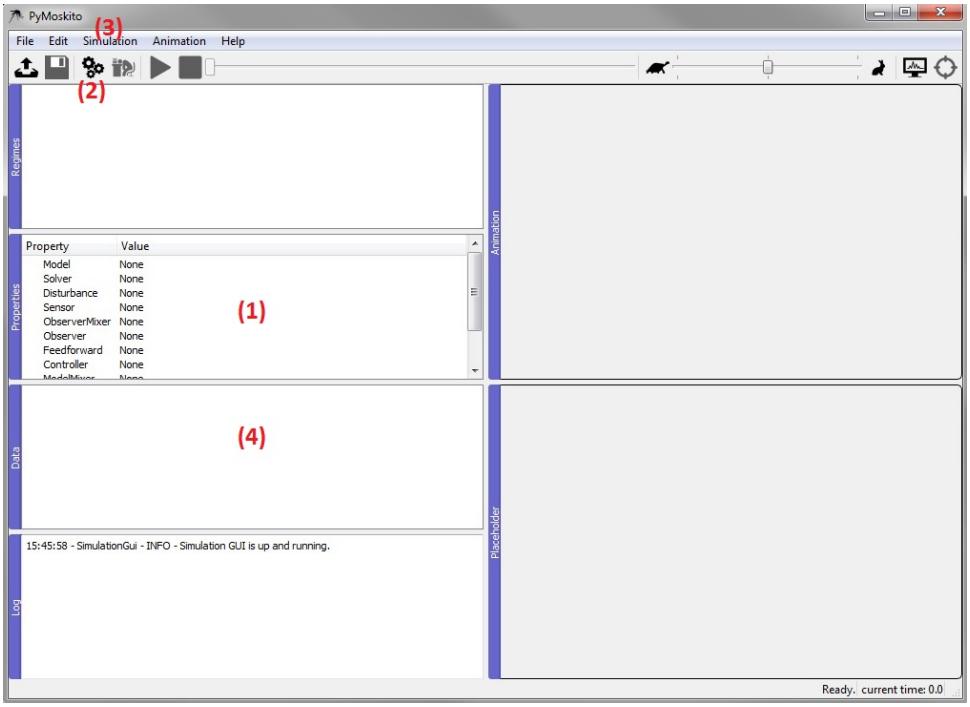

Fig. 3.3: The Interface of PyMoskito after start up

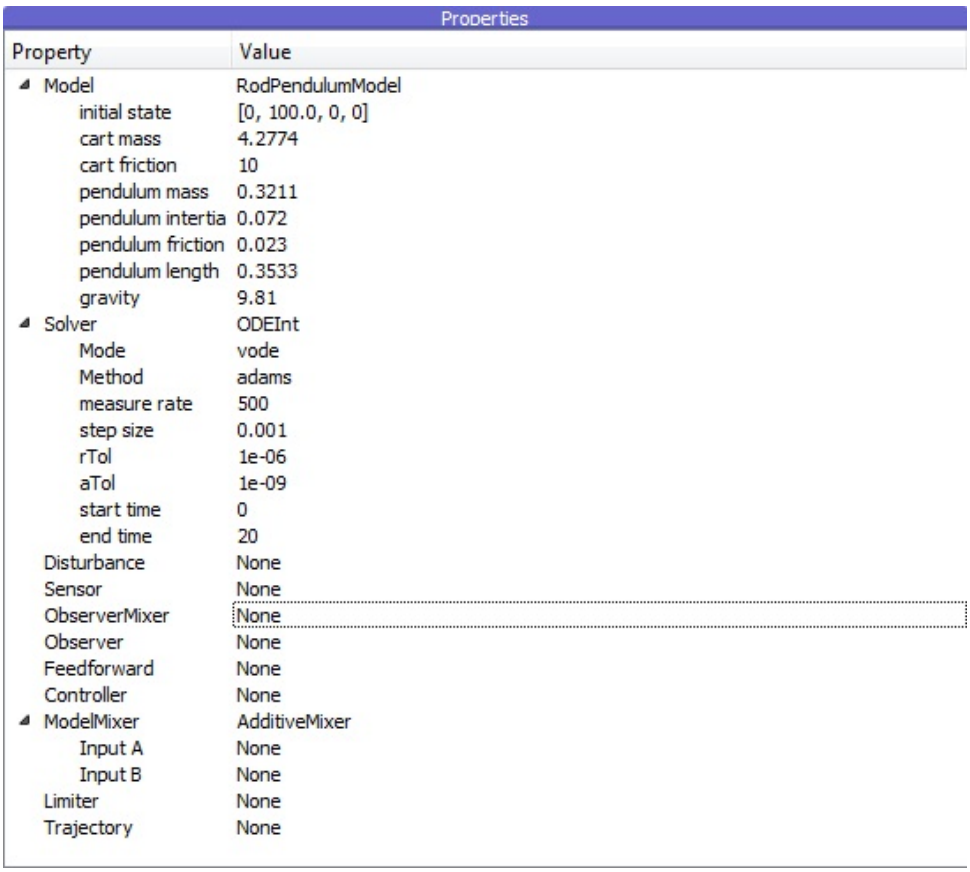

Fig. 3.4: The settings for testing the model class

Click the gearwheel button (2), use the drop-down menu (3) or press F5 to start the simulation. After a succesful simulation, all created diagrams will be listed in the Data Window (4). Double click on one to display it as shown in Fig. 3.5.

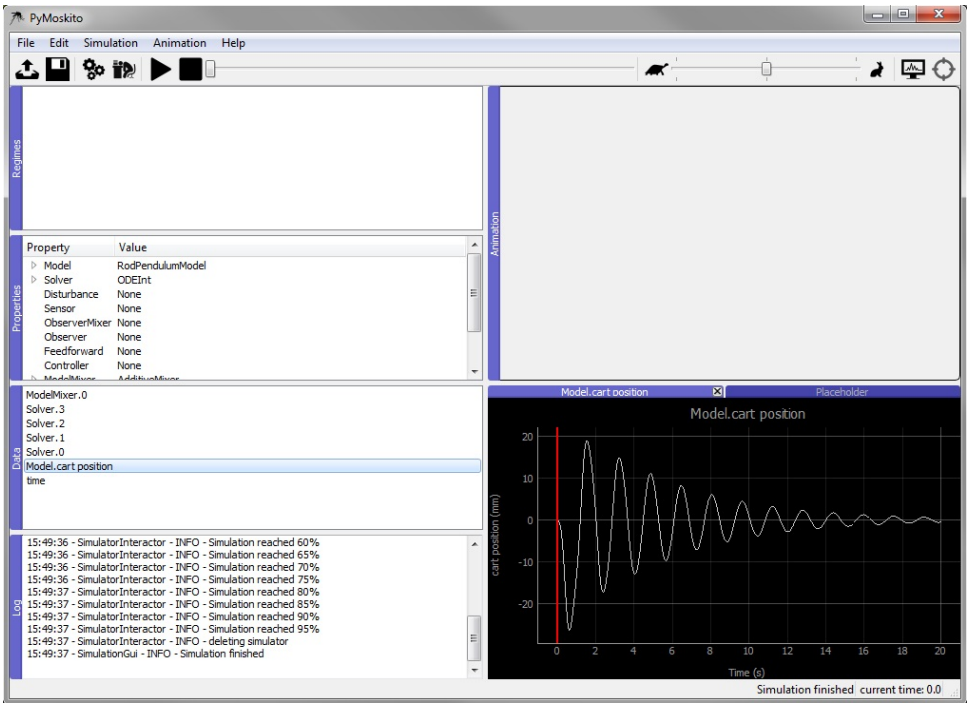

Fig. 3.5: The Interface of PyMoskito after a successful simulation

Feel free to experiment with the properties and see, if the model reacts the way you would have predicted. After testing the model class, a controller shall be implemented.

3.1.6 Implementing a Controller

To close the loop a controller has to be added. This can easily be done by deriving from the `Controller` class. Its task is to stabilize the pendulum by calculating a suitable input for the model. To keep things simple, the linear state-feedback controller in this scenario and it is based on the linearized system which is given by

$$\boldsymbol{A} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{m^2 l^2 g}{z} & -\frac{JD}{z} & \frac{mld}{z} \\ 0 & -\frac{(M+m)mlg}{z} & \frac{mlD}{z} & -\frac{(M+m)d}{z} \end{pmatrix} \quad \boldsymbol{B} = \begin{pmatrix} 0 \\ 0 \\ \frac{J}{z} \\ -\frac{ml}{z} \end{pmatrix} \quad \boldsymbol{C} = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix}$$

with

$$z = J(M+m) - m^2 l^2.$$

The linear control law is given by

$$u = -\boldsymbol{K}\boldsymbol{x} + \boldsymbol{V}\boldsymbol{y_d}$$

with the control gain $\boldsymbol{K}$ and the prefilter $\boldsymbol{V}$. One possibility to calculate the control gain is by using the Ackermann formula.

With all necessary equations, the implementation of the controller class can begin. Start by creating a file called:

```
controller.py
```

Import the same classes as in the model class:

```
# -*- coding: utf-8 -*-
import numpy as np
from collections import OrderedDict

import pymoskito as pm


```

Derive your controller from *Controller* Next, create `public_settings` as before in the model. Its entries will be accessable in the graphical interface of PyMoskito during runtime. This time, the only parameters will be the desired poles of the closed loop, which the controller shall establish:

```
class BasicController(pm.Controller):
    public_settings = OrderedDict([("poles", [-2, -2, -2, -2])
                                   ])

```

Within the constructor, it is obligatory to set the `input order` and an `input type`. The `input order` determines how many derivatives of the trajectory will be required, sinc eour controller is very simple a `0` will do here. Valid entries for `input type` are *system_state*, *system_output*, *Observer* and *Sensor*. In our case we will go for `system_state`. After all necessary updates, call the constructor of the base class as seen in line `20`. Store the linearized system matrices and the equilibrium state. To make matrix operations possible, use the array type provided by NumPy. PyMoskito's Controltools provide functions to calculate the values of a linear state feedback and a prefilter, which can be used as seen in lines `49-50`. The method `place_siso()` is an implementation of the Ackermann formula:

```
    def __init__(self, settings):
        settings.update(input_order=0)
        settings.update(input_type="system_state")

        pm.Controller.__init__(self, settings)

        # model parameters
        g = 9.81  # gravity [m/s^2]
        M = 4.2774  # cart mass [kg]
        D = 10  # cart friction constant [Ns/m]
        m = 0.3211  # pendulum mass [kg]
        d = 0.023  # pendulum friction constant [Nms]
        l = 0.3533  # pendulum length [m]
        J = 0.072  # pendulum moment of intertia [kg*m^2]

        # the system matrices after linearization in phi=PI
        z = (M + m) * J - (m * l) ** 2
        A = np.array([[0, 0, 1, 0],
                      [0, 0, 0, 1],
                      [0, (m * l) ** 2 * g / z, -J * D / z, m * l * d / z],
                      [0, -(M + m) * m * l * g / z, m * l * D / z,
                       -(M + m) * d / z]
                      ])
        B = np.array([[0],
                      [0],
                      [J / z],
                      [-l * m / z]
                      ])
        C = np.array([[1, 0, 0, 0]])

        # the equilibrium state as a vector
        self._eq_state = np.array([0, np.pi, 0, 0])

        # pole placement of linearized state feedback
        self._K = pm.controltools.place_siso(A, B, self._settings["poles"])
```

(continues on next page)

(continued from previous page)

```
        self._V = pm.controltools.calc_prefilter(A, B, C, self._K)

```

That would be all for the constructor. The only other method left to implement contains the actual control law and will be called by the solver during runtime. Its parameters are the current time, the current values of trajectory, feedforward and controller input. The parameter `**kwargs` holds further information, which is explained in *pymoskito.simulation_modules.Controller*. For our example, we will just ignore it. Since this controller will be stabilizing the system in the steady state [0,0,0,0], it has to be subtracted to work on the small signal scale.

```
    def _control(self, time, trajectory_values=None, feedforward_values=None,
                 input_values=None, **kwargs):
        x = input_values - self._eq_state
        yd = trajectory_values - self._eq_state[0]
        output = - np.dot(self._K, x) + np.dot(self._V, yd[0])

        return output
```

Finally, import the controller file and register the controller class to PyMoskito by adding two lines to the main.py file as done before with the model class. Your *main.py* should now look like this, with the changed lines highlighted:

```
# -*- coding: utf-8 -*-
import pymoskito as pm

# import custom modules
import model
import controller


if __name__ == '__main__':
    # register model
    pm.register_simulation_module(pm.Model, model.PendulumModel)

    # register controller
    pm.register_simulation_module(pm.Controller, controller.BasicController)

    # start the program
    pm.run()
```

Having put all pieces together, we are now ready to run our scenario.

3.1.7 Closing the Control Loop

Firstly, start PyMoskito from the commandline and reapply the previous steps:

- select `PendulumModel` as `Model`
- change the initial state of `PendulumModel` to `[0, 100, 0, 0]`
- select `ODEInt` as `Solver`
- change the end time of `Solver` to 10
- select `AdditiveMixer` as `ModelMixer`

Now, it gets interesting, select:

- the new `BasicController` as `Controller`
- change `Input A` of `ModelMixer` to `Controller` and
- select `Setpoint` as `Trajectory`

to generate desired values for out new setup. The setpoint `0` demands that the cart position (the output defined by our model) should be kept at zero.

To enter string values, type `'Controller'` or `"Controller"` and remember to press enter to confirm the input.

The Properties window should now look like Fig. 3.6

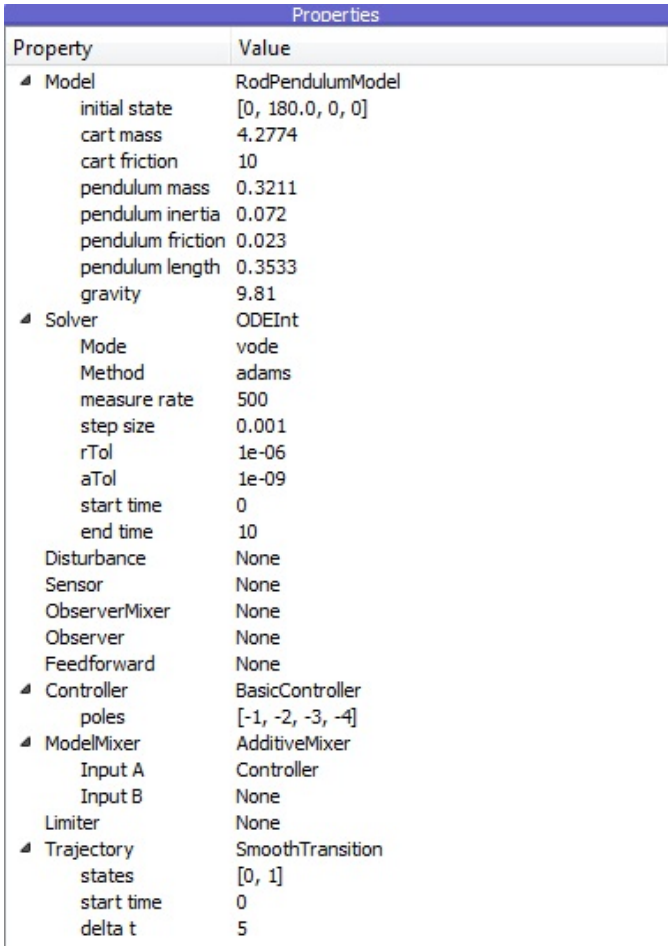

Fig. 3.6: The properties window with changes for testing applied

Now, hit `F5` to run the simulation. After simulating, you find a few more diagrams in the data section. Fig. 3.7 shows the example of the control error.

Feel free to experiment with the settings and see, if the control loop reacts the way you would have predicted. Keep in mind that the implemented controller is static. The control law does not adapt to changes of the model parameters, since the controller gain is calculated from values stored in the controller class. You can use this effect to simulate the situation, where the controller design was based on model parameters that differ from the real parameters of the process.

These were all the fundamental functions of PyMoskito considered necessary to work with it. One more important, but also more advanced feature is the system's visualization in 2D or 3D. This animation appears in the window at the top right, which remained grey during this tutorial (see Fig. 3.3, Fig. 3.5, Fig. 3.7). For more information on this topic, see the *lesson on visualization*.

3.1.8 Further Reading

After completing the Beginners Tutorial, there are several ways to go on if you did not find an answer to your problem or simply would like to know more.

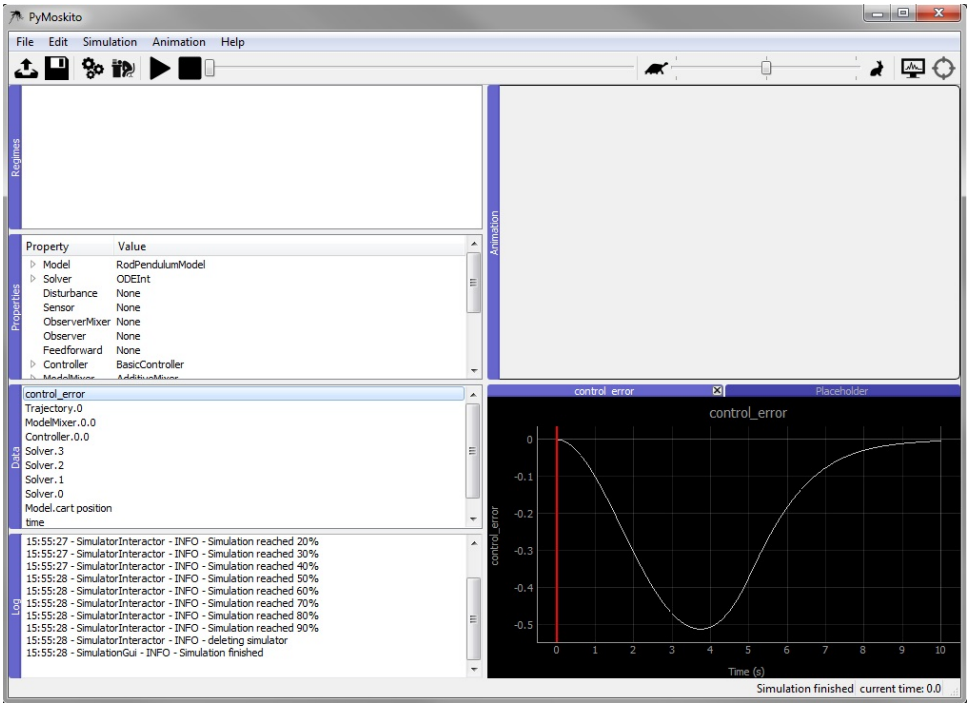

Fig. 3.7: PyMoskito's interface with the control error diagram displayed

PyMoskito contains a few *examples* of fully implemented systems. Take a peak into what is possible by running their simulations or reading their source code.

The *Users Guide* offers instructions to access the full potential of PyMoskito. Read about animated visualization, storing simulation settings for future reuse, comparing simulation results and more.

The *Modules Reference* contains documentations of all classes, that are part of PyMoskito. Read about all abstract classes being part of the control loop, the modules realizing the simulation or the interface and more.

3.2 Visualization

This tutorial covers the subject of how to visualize your system using pymoskito.

To do this, you can provide a `pm.Visualizer` to the toolbox which will then be used to show the system. To accomplish this, pymoskito uses the VisualizationToolkit (vtk) for natty 3d plots. However, if vtk is not available a fallback method using the matplotlib is also supported.

Before we start visualizing, we need to choose a system. For sake of simplicity, the simple_example system from the *introduction* will be used. Visualizers for both toolkits will be explained in the following sections.

3.2.1 Visualization using the Matplotlib

Building the visualizer

The overall plan here is to derive a class that we will call *MplPendulumVisualizer* from `MplVisualizer`. In its constructor, we will lay down all the elements we want to use to visualize the system. In our case these will be the beam on which the cart is moving, the cart and of course the pendulum. Later on, the method `pymoskito.MplVisualizer.update_scene()` will be called repeatedly from the GUI to update the visualization.

We will start off with the following code:

```
# -*- coding: utf-8 -*-
import numpy as np
import pymoskito as pm

import matplotlib as mpl
import matplotlib.patches
import matplotlib.transforms


class MplPendulumVisualizer(pm.MplVisualizer):

    # parameters
    x_min_plot = -.85
    x_max_plot = .85
    y_min_plot = -.6
    y_max_plot = .6

    cart_height = .1
    cart_length = .2

    beam_height = .01
    beam_length = 1

    pendulum_shaft_height = 0.027
    pendulum_shaft_radius = 0.020

    pendulum_height = 0.5
    pendulum_radius = 0.005
```

On top, we import some modules we'll need later on. Once this is done we derive our *MplPendulumVisualizer* from `MplVisualizer`. What follows below are some parameters for the matplotlib canvas and the objects we want to draw, fell free to adapt them as you like!

In the first part of the constructor, we set up the canvas:

```

    def __init__(self, q_widget, q_layout):
        # setup canvas
        pm.MplVisualizer.__init__(self, q_widget, q_layout)
        self.axes.set_xlim(self.x_min_plot, self.x_max_plot)
        self.axes.set_ylim(self.y_min_plot, self.y_max_plot)
        self.axes.set_aspect("equal")

```

Afterwards, our "actors" are created:

```
        self.beam = mpl.patches.Rectangle(xy=[-self.beam_length/2,
                                              -(self.beam_height
                                                + self.cart_height/2)],
                                          width=self.beam_length,
                                          height=self.beam_height,
                                          color="lightgrey")

        self.cart = mpl.patches.Rectangle(xy=[-self.cart_length/2,
                                              -self.cart_height/2],
                                          width=self.cart_length,
                                          height=self.cart_height,
                                          color="dimgrey")

        self.pendulum_shaft = mpl.patches.Circle(
            xy=[0, 0],
            radius=self.pendulum_shaft_radius,
```

(continues on next page)

(continued from previous page)

```
            color="lightgrey",
            zorder=3)

        t = mpl.transforms.Affine2D().rotate_deg(180) + self.axes.transData
        self.pendulum = mpl.patches.Rectangle(
            xy=[-self.pendulum_radius, 0],
            width=2*self.pendulum_radius,
            height=self.pendulum_height,
            color=pm.colors.HKS07K100,
            zorder=2,
            transform=t)

```

Note that a transformation object is used to get the patch in the correct place and orientation. We'll make more use of transformations later. For now, all that is left to do for the constructor is to add our actors t the canvas:

```
        self.axes.add_patch(self.beam)
        self.axes.add_patch(self.cart)
        self.axes.add_patch(self.pendulum_shaft)
        self.axes.add_patch(self.pendulum)

```

After this step, the GUI knows how our system looks like. Now comes the interesting part: We use the systems state vector (the first Equation in *introduction*) which we obtained from the simulation to update our drawing:

```
    def update_scene(self, x):
        cart_pos = x[0]
        phi = np.rad2deg(x[1])

        # cart and shaft
        self.cart.set_x(cart_pos - self.cart_length/2)
        self.pendulum_shaft.center = (cart_pos, 0)

        # pendulum
        ped_trafo = (mpl.transforms.Affine2D().rotate_deg(phi)
                     + mpl.transforms.Affine2D().translate(cart_pos, 0)
                     + self.axes.transData)
        self.pendulum.set_transform(ped_trafo)

        # update canvas
        self.canvas.draw()


```

As defined by our model, the first element of the state vector x yields the cart position, while the pendulum deflection (in rad) is given by `x[1]` . Firstly, cart and the pendulum shaft are moved. This can either be done via `set_x()` or by directly overwriting the value of the `center` attribute. For the pendulum however, a transformation chain is build. It consists of a rotation by the pendulum angle `phi` followed by a translation to the current cart position. The last component is used to compensate offsets from the rendered window.

Lastly but important: The canvas is updated vie a call to `self.canvas.draw()`

The complete class can be found under:

```
pymoskito/examples/simple_pednulum/visualizer_mpl.py
```

Registering the visualizer

To get our visualizer actually working, we need to register it. For the simple *main.py* of our example this would mean adding the following lines:

```python
# -*- coding: utf-8 -*-
import pymoskito as pm

# import custom modules
import model
import controller
import visualizer_mpl


if __name__ == '__main__':
    # register model
    pm.register_simulation_module(pm.Model, model.PendulumModel)

    # register controller
    pm.register_simulation_module(pm.Controller, controller.BasicController)

    # register visualizer
    pm.register_visualizer(visualizer_mpl.MplPendulumVisualizer)

    # start the program
    pm.run()
```

After starting the program, this is what you should see in the top right corner:

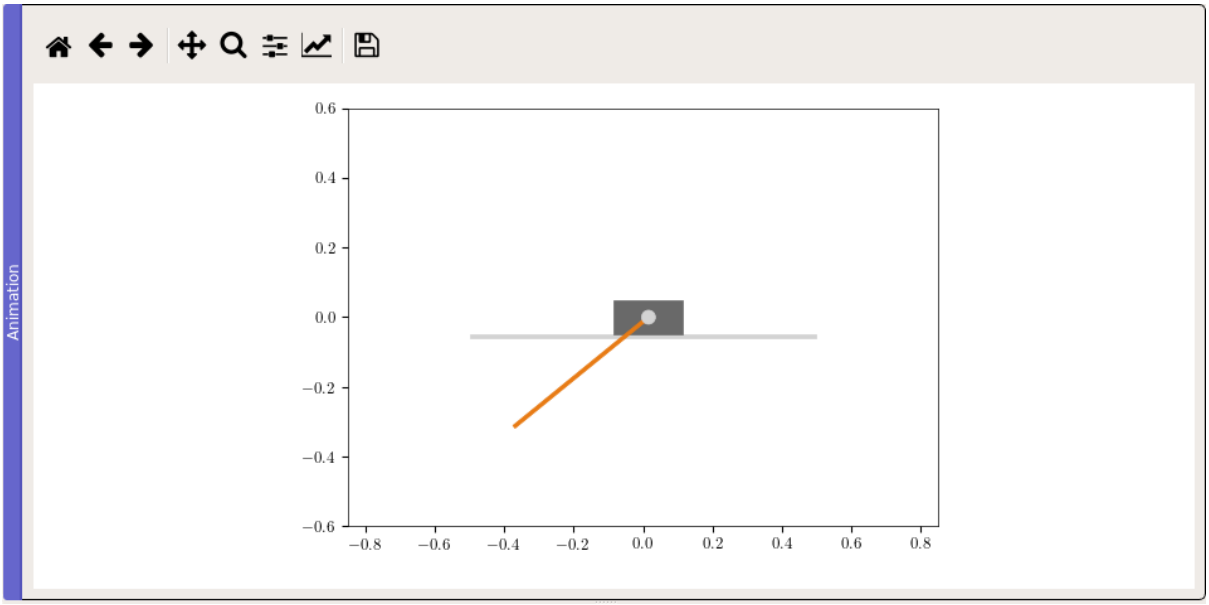

Fig. 3.8: Matplotlib visualization of the simple pendulum system

If you are looking for a fancier animation, check out the *VTK Tutorial*.

3.2.2 Visualization using VTK

Building the visualizer

TODO

Intermediate:

3.3 Switching Systems

TODO

3.4 Postprocessing

TODO

Expert:

3.5 Metaprocessing

TODO

CHAPTER 4

Examples

PyMoskito comes with quite a set of interesting examples from the field of control theory. To run an example just enter:

```
$ python -m pymoskito.examples.NAME
```

where `NAME` is the name, given in parenthesis behind the example titles.

List of Examples:

4.1 Ball and Beam (ballbeam)

A beam is pivoted on a bearing in its middle. The position of a ball on the beam is controlable by applying a torque into the bearing.

The ball has a mass M, a radius R and a moment of inertia J_b. Its distance r to the beam center is counted positively to the right. For the purpose of simplification, the ball can only move in the horizontal direction.

The beam has a length L, a moment of inertia J and its deflection from the horizontal line is the angle θ.

The task is to control the position r of the ball with the actuation variable being the torque τ. The interesting part in this particular system is that while being nonlinear and intrinsically unstable, it's relative degree is not well-defined. This makes it an exciting but yet still clear lab example. The description used here is taken from the publication *[Hauser92]* .

With the state vector

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} r \\ \dot{r} \\ \theta \\ \dot{\theta} \end{pmatrix},$$

the nonlinear model equations are given by

$$\dot{\boldsymbol{x}} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{pmatrix} = \begin{pmatrix} x_2 \\ \frac{MR^2}{J_b + MR^2}(x_1 x_4^2 - g\sin(x_3)) \\ x_4 \\ \frac{\tau - M\cdot(2x_1x_2x_4 + gx_1\cos(x_3))}{Mx_1^2 + J + J_b} \end{pmatrix}.$$

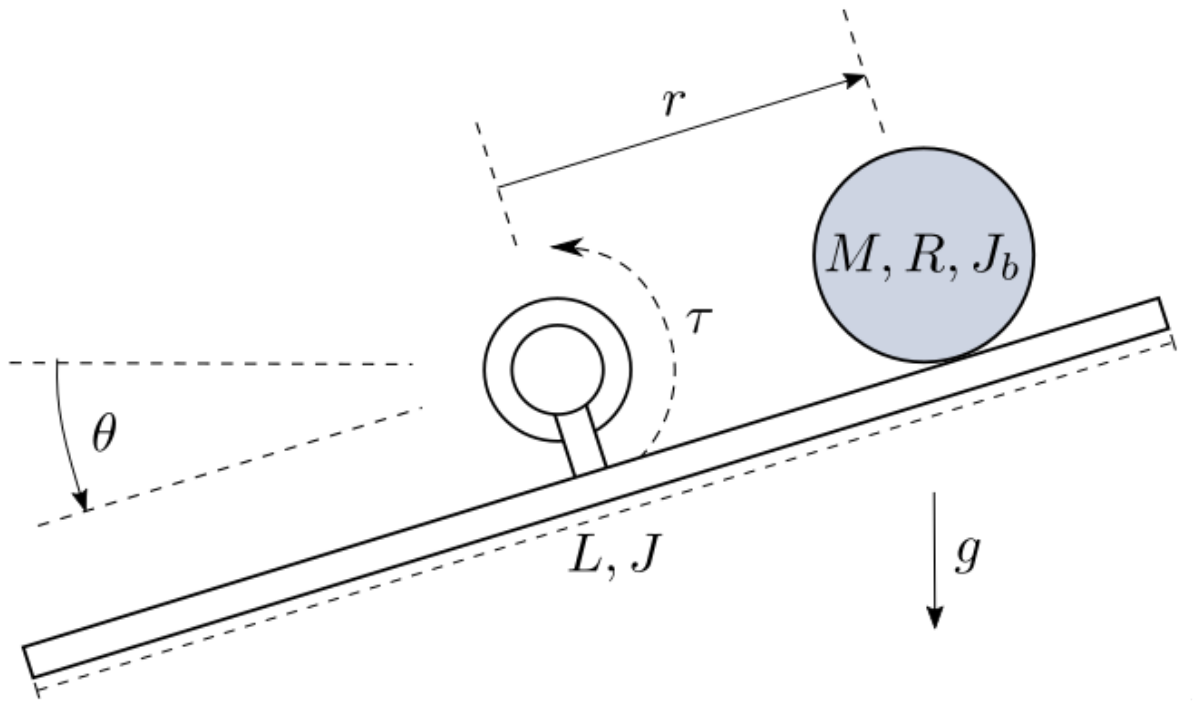

Fig. 4.1: The ball and beam system

Violations of the model's boundary conditions are the ball leaving the beam

$$|x_1| > \frac{L}{2}$$

or the beam's deflection reaching the vertical line

$$|x_3| > \frac{\pi}{2}.$$

The ball's position

$$y = x_1 = r$$

is chosen as output.

The example comes with five controllers. The `FController` and `GController` both implemenent a input-output-linearization of the system and manipulate the second output derivative by ignoring certain parts of the term. The `JController` ignores the nonlinear parts of the linearized model equations, also called standard jacobian approximation. The `LSSController` linearizes the nonlinear model in a chosen steady state and applies static state feedback. The `PIXController` also linearizes the model and additionally integrates the control error.

`LinearFeedforward` implements a compensation of the linear system equation parts, with the aim of reducing the controllers task to the nonlinear terms of the equations and disturbances.

The example comes with four observers. `LuenbergerObserver`, `LuenbergerObserverReduced` and `LuenbergerObserverInt` are different implementations of the Luenberger observer. The second of these improves its performance by using a different method of integration and the third uses the solver for integration. The `HighGainObserver` tries to implement an observer for nonlinear systems, However, the examination for observability leads to the presumption that this attempt should fail.

A 3D visualizer is implemented. In case of missing VTK, a 2D visualization can be used instead.

An external `settings` file contains all parameters. All implemented classes import their initial values from here.

At program start, the main loads two regimes from the file `default.sreg`. `test-nonlinear` is a setting of the nonlinear controller moving the ball from the left to the right side of the beam. `test-linear` shows the step response of a linear controller, resulting in the ball moving from the middle to the right side of the beam.

The example also provides ten different modules for postprocessing. They plot different combinations of results in two formats, one of them being `pdf`. The second format of files can be given to a metaprocessor.

The structure of `__main__.py` allows starting the example without navigating to the directory and using an `__init__.py` file to outsource the import commands for additional files.

4.2 Ball in Tube (balltube)

A fan at the bottom of a tube produces an air stream moving upwards. A ball levitates in the air stream.

The task is to control the ball's position z. Actuating variable is the motor's control signal u_{pwm}.

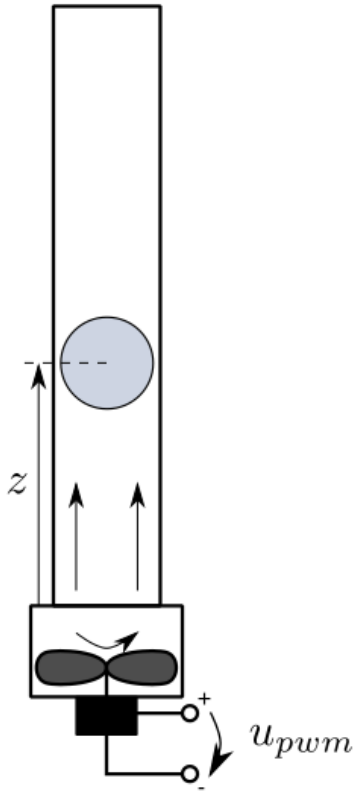

Fig. 4.2: The ball in tube system

The example comes with two models, which differ in the reaction to the ball falling down. The `BallInTubeModel` makes the ball stick to the ground once it falls down. The `BallInTubeSpringModel` lets the ball to jump back up again:

4.2.1 Ball in Tube Model

A fan at the bottom of a tube produces an air stream moving upwards. A ball levitates in the air stream.

The fan rotates with the rotational speed η. It produces an air stream with the velocity v. The factor k_L describes the proportionality between the air's volume flow rate and the fan's rotational speed. The motor driving the fan is modeled as a PT2-element with the amplification k_s, the damping d and the time constant T. An Arduino Uno controls the motor, its discrete control signal u_{pwm} has a range of $0 - 255$ and amplifies the supply voltage V_{cc}.

The ball covers an area A_B and has a mass m. Its distance to the tube's bottom is the position z. The gap between the ball and the tube covers an area A_{Sp}. The factor k_V describes the proportionality between the force of flow resistance and the velocity of the air streaming through the gap.

The tube has a height h.

The task is to control the ball's position z. Actuating variable is the motor's control signal u_{pwm}.

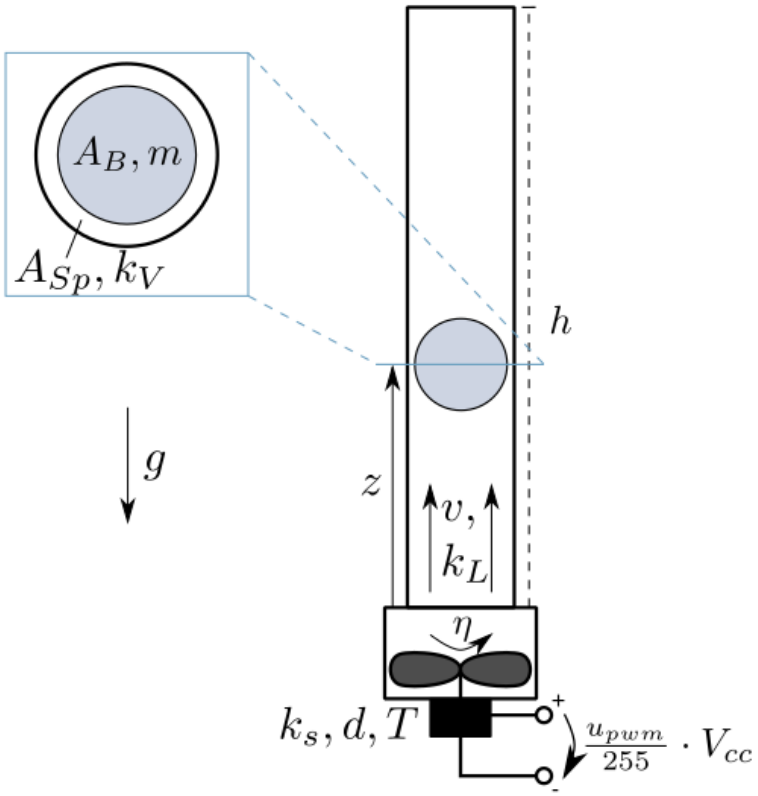

Fig. 4.3: The ball in tube system in detail

TWith the state vector

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \eta \\ \dot{\eta} \\ z \\ \dot{z} \end{pmatrix},$$

the model equations are given by

$$\dot{\boldsymbol{x}} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{pmatrix} = \begin{pmatrix} x_2 \\ -\frac{1}{T^2}x_1 - \frac{2d}{T}x_2 + \frac{k_s}{T^2}\frac{u_{pwm}}{255}V_{cc} \\ x_4 \\ \frac{k_L}{m}\left(\frac{k_V x_1 - A_B x_4}{A_{Sp}}\right)^2 - g \end{pmatrix}.$$

In case of the ball falling down and reaching a position $x_3 < 0$ below the fan, the root function of the model overrides the ball's position $x_3 = 0$ and velocity $x_4 = 0$.

The model's boundary condition is violated if the ball leaves the tube on the upper end:

$$x_3 > h$$

The ball's position

$$y = x_3 = z$$

is chosen as output.

4.2.2 Ball in Tube Spring Model

This model contains all equations of the *Ball in Tube Model* except for one single change: The dynamics of the ball bouncing back up once it falls to the ground.

Instead of overriding the ball's position and speed once the ball falls below the fan, the fourth system equation is overwritten by an extended version

$$\dot{x}_4 = \frac{K x_3}{m} - \frac{D x_4}{m} + \frac{k_L}{m}\left(\frac{k_V x_1 - A_B x_4}{A_{Sp}}\right)^2 - g.$$

This inserts a spring with the stiffness K and the damping D on the ground of the tube.

The `OpenLoop` controller ignores the current state and output of the model, as well as trajectory values. Instead it gives the opportunity to set the actuating variable u_{pwm} manually.

The ball's position is used as a flat output in this flatness based feedforward module:

4.2.3 Ball in Tube Feedforward

Analyzing the system for flatness leads to finding the ball's position as the flat output of the system, meaning that all other system variables can be calculated from it. This can be retraced easily with the following chain of equations:

$$\begin{aligned}
x_3 &= y \\
&= f_3(y) \\
x_4 &= \dot{y} \\
&= f_4(\dot{y}) \\
x_1 &= \frac{A_{Sp}}{k_V}\sqrt{\frac{m}{k_L}(\ddot{y}+g)} + \frac{A_B}{k_V}\dot{y} \\
&= f_1(y, \dot{y}, \ddot{y}) \\
x_2 &= \frac{m A_{Sp}^2 y^{(3)}}{2 k_V k_L (k_V x_1 - A_B \dot{y})} + \frac{A_B}{k_V}\ddot{y} \\
&= f_2(y, \dot{y}, \ddot{y}, y^{(3)}) \\
u &= \frac{m T^2 A_{Sp}^2 y^{(4)} - 2 k_L T^2 (k_V x_2 - A_B \ddot{y})^2}{2 k_s k_V k_L (k_V x_1 - A_B \dot{y})} + \frac{A_B T^2}{k_s k_V} y^{(3)} + \frac{2dT}{k_s} x_2 + \frac{1}{k_s} x_1 \\
&= f_u(y, \dot{y}, \ddot{y}, y^{(3)}, y^{(4)})
\end{aligned}$$

The last equation $u = f_u(y, \dot{y}, \ddot{y}, y^{(3)}, y^{(4)})$ is implented in this feedforward module. The highest order of derivatives is $y^{(4)}$, so the trajectory generator needs to provide a trajectory that is differentiable at least four times.

A 3D visualizer is implemented. In case of missing VTK, a 2D visualization can be used instead.

An external `settings` file contains all parameters. All implemented classes import their initial values from here.

Regimes are stored in two files. At program start, the main function loads six regimes from the file `default.sreg`. In addition, nine regimes can be loaded manually from the file `experiments.sreg`.

The structure of `__main__.py` allows starting the example without navigating to the directory and using an `__init__.py` file to outsource the import commands for additional files.

The example also provides a package for symbolic calculation.

4.3 Tandem Pendulum (pendulum)

Two pendulums are fixed on a cart, which can move in the horizontal direction.

The cart has a mass m_0. The friction between the cart and the surface causes a frictional force $F_r = d_0 \cdot \dot{s}$, in opposite direction as the velocity $\dot{s}$ of the cart.

Each pendulum has a mass m_i, a moment of intertia J_i, a length l_i and an angle of deflection φ_i. The friction in the joint where the pendulums are mounted on the cart causes a frictional torque $M_{ir} = d_i \cdot \dot{\varphi}_i$, in opposite direction as the speed of rotation $\dot{\varphi}_i$. The system is shown in Fig. 4.4 .

The task is to control the position s of the cart and to stabilize the pendulums in either the upward or downward position. Actuating variable is the force F.

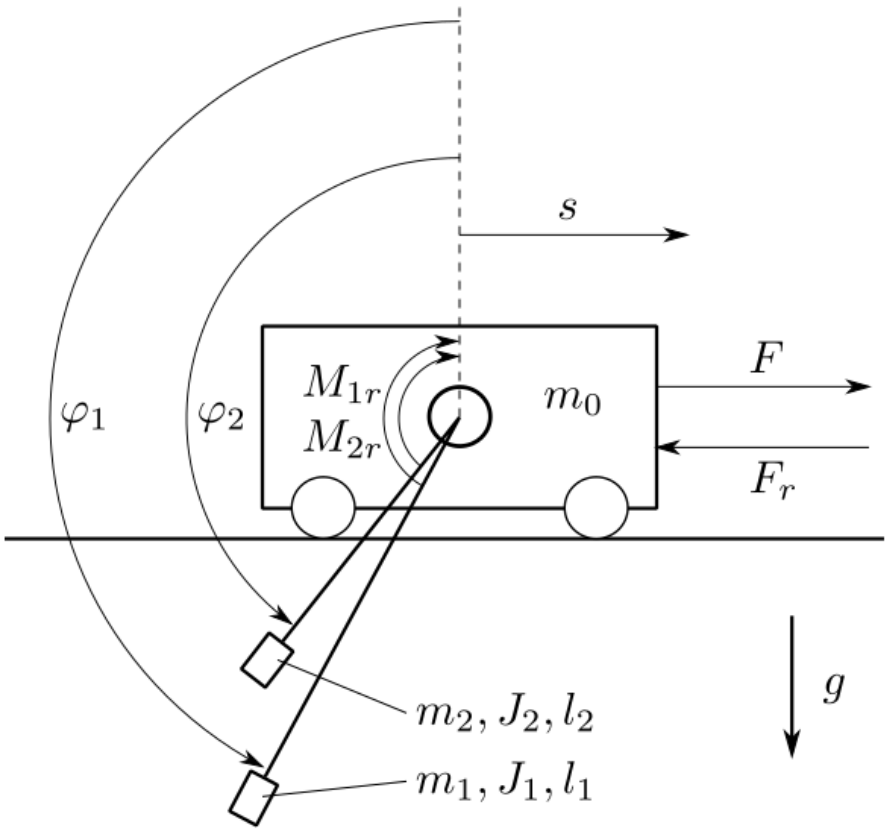

Fig. 4.4: The pendulum system

The example comes with three models. A point mass model, a rigid body model and a partially linearized model.

The state vector $\boldsymbol{x}$ is chosen in all three models as:

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} s \\ \dot{s} \\ \varphi_1 \\ \dot{\varphi}_1 \\ \varphi_2 \\ \dot{\varphi}_2 \end{pmatrix}$$

The class `TwoPendulumModel` is the implementation of a point mass model. The mass of each pendulum is considered concentrated at the end of its rod. The model resulting model equations are relatively simple and moments of inertia do not appear:

$$\dot{x} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \\ \dot{x}_5 \\ \dot{x}_6 \end{pmatrix} = \begin{pmatrix} x_2 \\ \frac{1}{M}\left(F_1 + F_2 + F - d_0 x_2 - \frac{d_1 x_4}{l_1}\cos(x_3) - \frac{d_2 x_6}{l_2}\cos(x_5)\right) \\ x_4 \\ \frac{g}{l_1}\sin(x_3) - \frac{d_1 x_4}{m_1 l_1^2} + \frac{\cos(x_3)}{l_1 M}\left(F_1 + F_2 + F - d_0 x_2 - \frac{d_1 x_4}{l_1}\cos(x_3) - \frac{d_2 x_6}{l_2}\cos(x_5)\right) \\ x_6 \\ \frac{g}{l_2}\sin(x_5) - \frac{d_2 x_6}{m_2 l_2^2} + \frac{\cos(x_5)}{l_2 M}\left(F_1 + F_2 + F - d_0 x_2 - \frac{d_1 x_4}{l_1}\cos(x_3) - \frac{d_2 x_6}{l_2}\cos(x_5)\right) \end{pmatrix}$$

$$
\begin{aligned}
M &= m_0 + m_1 \sin^2(x_3) + m_2 \sin^2(x_5) \\
F_1 &= m_1 \sin(x_3)(g\cos(x_3) - l_1 x_4^2) \\
F_2 &= m_2 \sin(x_5)(g\cos(x_5) - l_2 x_6^2)
\end{aligned}
$$

The class `TwoPendulumRigidBodyModel` is the implementation of a rigid body model. The rods are considered to have a mass and can not be ignored, each pendulum has a moment of inertia J_{DPi}:

$$
\dot{x} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \\ \dot{x}_5 \\ \dot{x}_6 \end{pmatrix} = \begin{pmatrix} x_2 \\ \frac{term2 + term3 + term4}{term1} \\ x_4 \\ \frac{1}{J_{DP1}} \left(m_1^* l_1^* \cos(x_3)\dot{x}_2 + M_1 - d_1 x_4 + m_1^* l_1^* g \sin(x_3) \right) \\ x_6 \\ \frac{1}{J_{DP2}} \left(m_2^* l_2^* \cos(x_5)\dot{x}_2 + M_2 - d_2 x_6 + m_2^* l_2^* g \sin(x_5) \right) \end{pmatrix}
$$

$$
\begin{aligned}
term1 &= m_0^* + m_1^* + m_2^* - \frac{m_1^{*2} l_1^{*2} \cos^2(x_3)}{J_{DP1}} - \frac{m_2^{*2} l_2^{*2} \cos^2(x_5)}{J_{DP2}} \\
term2 &= \frac{m_1^* l_1^* \cos(x_3)}{J_{DP1}} (M_1 - d_1 x_4 + m_1^* l_1^* g \sin(x_3)) \\
term3 &= \frac{m_2^* l_2^* \cos(x_5)}{J_{DP2}} (M_2 - d_2 x_6 + m_2^* l_2^* g \sin(x_5)) \\
term4 &= F - d_0 x_2 - m_1^* l_1^* x_4^2 \sin(x_3) - m_2^* l_2^* x_6^2 \sin(x_5)
\end{aligned}
$$

The class `TwoPendulumModelParLin` is the implementation of a the partially linearized point mass model. The input is chosen as

$$
u_{tr} = \frac{1}{M} \left(F_1 + F_2 + F - d_0 x_2 - \frac{d_1 x_4}{l_1} \cos(x_3) - \frac{d_2 x_6}{l_2} \cos(x_5) \right),
$$

with M, F_1 and F_2 as before in `TwoPendulumModel`. This transforms the model equations into the input afine form

$$
\dot{x} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \\ \dot{x}_5 \\ \dot{x}_6 \end{pmatrix} = \begin{pmatrix} x_2 \\ 0 \\ x_4 \\ \frac{g}{l_1} \sin(x_3) - \frac{d_1 x_4}{m_1 l_1^2} \\ x_6 \\ \frac{g}{l_2} \sin(x_5) - \frac{d_2 x_6}{m_2 l_2^2} \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ \frac{\cos(x_3)}{l_1} \\ 0 \\ \frac{\cos(x_5)}{l_2} \end{pmatrix} u_{tr}
$$

All three models define the cart's position

$$
y = x_1 = s
$$

as the output of the system.

The example comes with five controllers. Two of them, `LinearStateFeedback` and `LinearStateFeedbackParLin`, implement linear state feedback, both using the package `symbolic_calculation` to calculate their gain and prefilter. The `LinearQuadraticRegulator` calculates its gain and prefilter by solving the continuous algebraic Riccati equation. The `LjapunovController` is designed with the method of Ljapunov to stabilize the pendulums in the upward position. And finally the `SwingUpController`, especially designed to swing up the pendulums using linear state feedback and to stabilize the system by switching to a Ljapunov controller once the pendulums point upwards.

A 3D visualizer is implemented. In case of missing VTK, a 2D visualization can be used instead.

An external `settings` file contains all parameters. All implemented classes import their initial values from here.

At program start, the main loads eleven regimes from the file `default.sreg`. The provided regimes not only show the stabilization of the system in different steady-states (e.g. both pendulums pointing downwards or both pointing upwards) but also ways to transition them between those states (e.g. swinging them up).

The example also provides two modules for postprocessing. They plot different combinations of results in two formats, one of them being `pdf`. The second format of files can be passed to a metaprocessor.

The structure of `__main__.py` allows starting the example without navigating to the directory and using an `__init__.py` file to outsource the import commands for additional files.

4.4 Car with Trailers (car)

A car pulls multiple trailers. All parts of the vehicle have one axis for simplification.

The car moves forward with a velocity v and turns with a ratotional speed ω. The coordinates x and y describe the car's distance to the origin of a stationary coordinate system.

The car's and the trailer's deflections regarding the horizontal line are φ_1, φ_2 and φ_3.

The distances between the axles and the couplings from front to back are d_1, l_2, d_2 and l_3

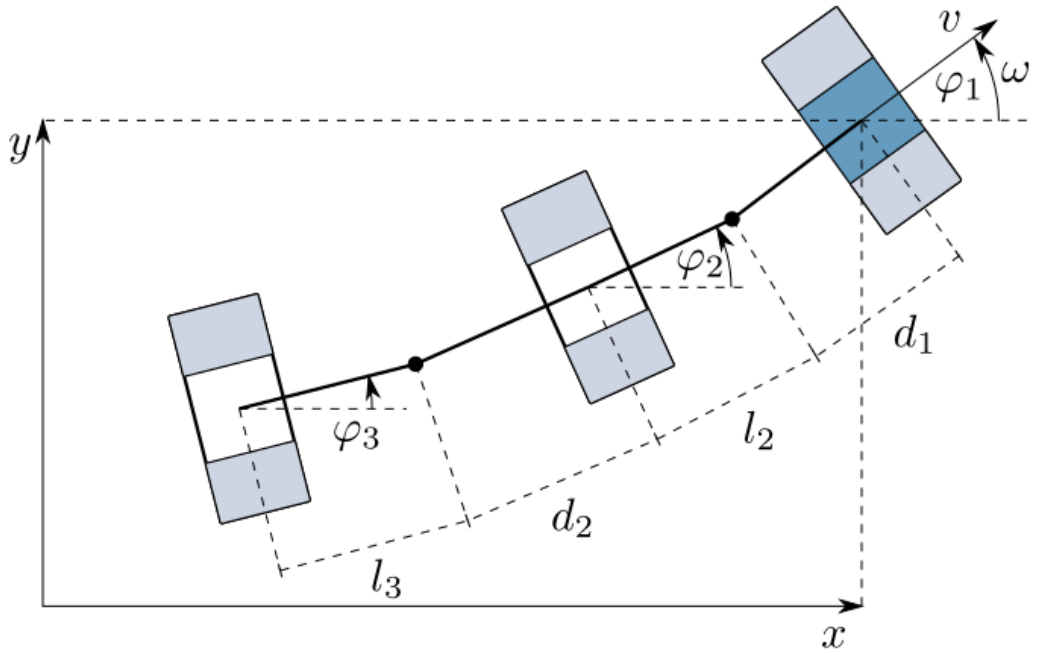

Fig. 4.5: The car system

With the state vector

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} x \\ y \\ \varphi_1 \\ \varphi_2 \\ \varphi_3 \end{pmatrix},$$

the model equations are given by

$$\dot{\boldsymbol{x}} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \\ \dot{x}_5 \end{pmatrix} = \begin{pmatrix} v\cos(x_3) \\ v\sin(x_3) \\ \omega \\ \frac{1}{l_2}\sin(x_3 - x_4)v - \frac{d_1}{l_2}\cos(x_3 - x_4)\omega \\ \dot{x}_5 \end{pmatrix},$$

with

$$\dot{x}_5 = \left(\frac{1}{l_3}\sin(x_3 - x_5) - \frac{l_2 + d_2}{l_2 l_3}\sin(x_3 - x_4)\cos(x_4 - x_5)\right) v + \dots$$
$$\cdots + \left(-\frac{d_1}{l_3}\cos(x_3 - x_5) + \frac{d_1(l_2 + d_2)}{l_2 l_3}\cos(x_3 - x_4)\cos(x_4 - x_5)\right)\omega.$$

The driving speed v and the turning speed ω are set to constant values in the state function of the model. They are potential actuating variables to control the system.

There is no output defined.

The example comes with no controller, with a 2D visualization, an external `settings` file containing all initial values for the parameters and one regime loaded from the file `default.sreg` by the main at program start.

The structure of `__main__.py` allows starting the example without navigating to the directory and using an `__init__.py` file to outsource the import commands for additional files.

CHAPTER 5

Users Guide

tbd

CHAPTER 6

PyMoskito Modules Reference

Because every feature of PyMoskito must have a test case, when you are not sure how to use something, just look into the `tests/` directories, find that feature and read the tests for it, that will tell you everything you need to know.

Most of the things are already documented though in this document, that is automatically generated using PyMoskito's docstrings.

Click the "modules" (modindex) link in the top right corner to easily access any PyInduct module, or use this contents:

6.1 Simulation GUI

class pymoskito.simulation_gui.**SimulationGui**
class for the graphical user interface

apply_regime_by_name(*regime_name*)
Apply the regime given by *regime_name* und update the regime index.

Returns *True* if successful, *False* if errors occurred.

Return type bool

closeEvent(*self*, *QCloseEvent*)

export_simulation_data(*ok*)
Query the user for a custom name and export the current simulation results.

Parameters **ok** – unused parameter from QAction.triggered() Signal

increment_playback_time()
go one time step forward in playback

load_regimes_from_file(*file_name*)
load simulation regime from file :param file_name:

new_simulation_data(*status*, *data*)
Slot to be called when the simulation interface has completed the current job and new data is available.

Parameters

- **status** (*str*) – Status of the simulation, either - *finished* : Simulation has been finished successfully or - *failed* : Simulation has failed.
- **data** (*dict*) – Dictionary, holding the simulation data.

pause_animation()
pause the animation

play_animation()
play the animation

plot_data_vector(*item*)
Creates a plot widget based on the given item.

If a plot for this item is already open no new plot is created but the existing one is raised up again.

Parameters **item** (*Qt.ListItem*) – Item to plot.

postprocessing_clicked()
starts the post- and metaprocessing application

regime_dclicked(*item*)
Apply the selected regime to the current target.

reset_camera_clicked()
reset camera in vtk window

run_next_regime()
Execute the next regime in the regime batch.

start_regime_execution()
Simulate all regimes in the regime list.

start_simulation()
start the simulation and disable start button

stop_animation()
Stop the animation if it is running and reset the playback time.

stop_regime_execution()
Stop the batch process.

update_gui()
Updates the graphical user interface to reflect changes of the current display time.

This includes:

- timestamp
- visualisation window
- time cursors in diagrams

update_playback_speed(*val*)
adjust playback time to slider value

Parameters **val** –

update_playback_time()
adjust playback time to slider value

pymoskito.simulation_gui.**run**(*regimes=None*)
Helper function to launch the PyMoskito GUI

6.2 Simulation Modules

class pymoskito.simulation_modules.**SimulationModule**(*settings*)
Smallest unit pof the simulation framework.

This class provides necessary functions like output calculation and holds all settings that can be accessed by the user. The `public_settings` are read by the `SimulationInterface` and the rendered by the GUI. All entries stated in this dictionary will be available as changeable settings for the module. On initialization, a possibly modified (in terms of its values) version of this dict will be passed back to this class and is thenceforward available via the `settings` property.

The most important method is `calc_output()` which is called by the `Simulator` to retrieve this modules output.

Parameters **settings** (*OrderedDict*) – Settings for this simulation module. These entries will be shown in the properties view and can be changed by the user. The important entries for this base class are:

***output info*:** Dict holding an information dictionaries with keys *Name* and *Unit* for each element in the output data. If available, these information are used to display reasonable names in the result view and to display the corresponding units for the result plots.

Warn: Do NOT use '.' in the *output_info* name field.

exception pymoskito.simulation_modules.**SimulationException**

class pymoskito.simulation_modules.**Trajectory**(*settings*)
Base class for all trajectory generators

_desired_values(*t*)
Placeholder for calculations of desired values.

Parameters **t** (*float*) – Time.

Returns Trajectory output. This should always be a two-dimensional array holding the components in to 0th and their derivatives in the 1th axis.

Return type Array

class pymoskito.simulation_modules.**Feedforward**(*settings*)
Base class for all feedforward implementations

_feedforward(*time*, *trajectory_values*)
Placeholder for feedforward calculations.

Parameters

- **time** (*float*) – Current time.
- **trajectory_values** (*array-like*) – Desired values from the trajectory generator.

Returns Feedforward output. This signal can be added to the controllers output via the *ModelMixer* and is also directly passed to the controller.

Return type Array

class pymoskito.simulation_modules.**Controller**(*settings*)
Base class for controllers.

Parameters **settings** (*dict*) – Dictionary holding the config options for this module. It must contain the following keys:

input_order The order of required derivatives from the trajectory generator.

input_type Source for the feedback calculation and one of the following: *system_state* , *system_output* , *Observer* or *Sensor* .

_control(*time*, *trajectory_values=None*, *feedforward_values=None*, *input_values=None*, ***kwargs*)
Placeholder for control law calculations.

For more sophisticated implementations overload `calc_output()` .

Parameters

- **time** (*float*) – Current time.
- **trajectory_values** (*array-like*) – Desired values from the trajectory generator.
- **feedforward_values** (*array-like*) – Output of feedforward block.
- **input_values** (*array-like*) – The input values selected by `input_type` .
- ****kwargs** – Placeholder for custom parameters.

Returns Control output.

Return type Array

class pymoskito.simulation_modules.**Limiter**(*settings*)
Base class for all limiter variants

_limit(*values*)
Placeholder for actual limit calculations.

Parameters **values** (*array-like*) – Values to limit.

Returns Limited output.

Return type Array

class pymoskito.simulation_modules.**ModelMixer**(*settings*)

class pymoskito.simulation_modules.**Model**(*settings*)
Base class for all user defined system models in state-space form.

Parameters **settings** (*dict*) – Dictionary holding the config options for this module. It must contain the following keys:

input_count The length of the input vector for this model.

state_count The length of the state vector for this model.

initial state The initial state vector for this model.

check_consistency(*x*)
Check whether the assumptions, made in the modelling process are violated.

Parameters **x** – Current system state

Raises *ModelException* – If a violation is detected. This will stop the simulation process.

initial_state
Return the initial state of the system.

root_function(*x*)
Check whether a reinitialisation of the integrator should be performed.

This can be the case if there are discontinuities in the system dynamics such as switching.

Parameters **x** (*array-like*) – Current system state.

Returns

- bool: *True* if reset is advised.
- array-like: State to continue with.

Return type tuple

state_function(*t*, *x*, *args*)
Calculate the state derivatives of a system with state x at time t.

Parameters

- **x** (*Array-like*) – System state.
- **t** (*float*) – System time.

Returns Temporal derivative of the system state at time t.

exception pymoskito.simulation_modules.**ModelException**
Exception to be raised if the current system state violates modelling assumptions.

class pymoskito.simulation_modules.**Solver**(*settings*)
Base Class for solver implementations

class pymoskito.simulation_modules.**Disturbance**(*settings*)
Base class for all disturbance variants

_disturb(*value*)
Placeholder for disturbance calculations.

If the noise is to be dependent on the measured signal use its *value* to create the noise.

Parameters **value** (*array-like float*) – Values from the source selected by the `input_signal` property.

Returns Noise that will be mixed with a signal later on.

Return type array-like float

class pymoskito.simulation_modules.**Sensor**(*settings*)
Base class for all sensor variants

_measure(*value*)
Placeholder for measurement calculations.

One may reorder or remove state elements or introduce measurement delays here.

Parameters **value** (*array-like float*) – Values from the source selected by the `input_signal` property.

Returns 'Measured' values.

Return type array-like float

class pymoskito.simulation_modules.**ObserverMixer**(*settings*)

class pymoskito.simulation_modules.**Observer**(*settings*)
Base class for observers

_observe(*time*, *system_input*, *system_output*)
Placeholder for observer law.

Parameters

- **time** – Current time.
- **system_input** – Current system input.
- **system_output** – Current system output.

Returns Estimated system state

6.3 Generic Simulation Modules

class pymoskito.generic_simulation_modules.**LinearStateSpaceModel**(*settings*)
The state space model of a linear system.

The parameters of this model can be provided in form of a file whose path is given by the setting `config file` . This path should point to a pickled dict holding the following keys:

***system*:** An Instance of `scipy.signal.StateSpace` (from scipy) representing the system,

***op_inputs*:** An array-like object holding the operational point's inputs,

***op_outputs*:** An array-like object holding the operational point's outputs.

state_function(*t*, *x*, *args*)
Calculate the state derivatives of a system with state x at time t.

Parameters

- **x** (*Array-like*) – System state.
- **t** (*float*) – System time.

Returns Temporal derivative of the system state at time t.

class pymoskito.generic_simulation_modules.**ODEInt**(*settings*)
Wrapper for ode_int from Scipy project

integrate(*t*)
Integrate until target step is reached.

Parameters **t** – target time

Returns system state at target time

set_input(**args*)
propagate input changes to ode_int

class pymoskito.generic_simulation_modules.**ModelInputLimiter**(*settings*)
ModelInputLimiter that limits the model input values.

Settings: *Limits*: (List of) list(s) that hold (min, max) pairs for the corresponding input.

class pymoskito.generic_simulation_modules.**Setpoint**(*settings*)
Provides setpoints for every output component.

If the output is not scalar, just add more entries to the list. By querying the differential order from the controller (if available) the required derivatives are given.

Note: Keep in mind that while this class provides zeros for all derivatives of the desired value, they actually strive to infinity for $t = 0$.

---

class pymoskito.generic_simulation_modules.**HarmonicTrajectory**(*settings*)
This generator provides a scalar harmonic sinus signal with derivatives up to order n

class pymoskito.generic_simulation_modules.**SmoothTransition**(*settings*)
provides (differential) smooth transition between two scalar states

class pymoskito.generic_simulation_modules.**PIDController**(*settings*)
PID Controller

class pymoskito.generic_simulation_modules.**LinearStateSpaceController**(*settings*)
A controller that is based on a state space model of a linear system.

This controller needs a linear statespace model, just as the *LinearStateSpaceModel* . The file provided in `config file` should therefore contain a dict holding the entries: `model`, `op_inputs` and `op_outputs` .

If poles is given (differing from *None*) the state-feedback will be computed using `pymoskito.place_siso()` . Furthermore an appropriate prefilter is calculated, which establishes stationary attainment of the desired output values.

Note: If a SIMO or MIMO system is given, the control package as well as the slycot package are needed the perform the pole placement.

---

class pymoskito.generic_simulation_modules.**DeadTimeSensor**(*settings*)
Sensor that adds a measurement delay on chosen states

class pymoskito.generic_simulation_modules.**GaussianNoise**(*settings*)
Noise generator for gaussian noise

class pymoskito.generic_simulation_modules.**AdditiveMixer**(*settings*)
Signal Mixer that accumulates all input signals.

Processing is done according to rules of numpy broadcasting.

6.4 Simulation Core

class pymoskito.simulation_core.**SimulationStateChange**(***kwargs*)
Object that is emitted when Simulator changes its state.

Keyword Arguments

- **type** – Keyword describing the state change, can be one of the following
  - *init* Initialisation
  - *start* : Start of Simulation
  - *time* : Accomplishment of new progress step
  - *finish* : Finish of Simulation
  - *abort* : Abortion of Simulation
- **data** – Data that is emitted on state change.
- **info** – Further information.

class pymoskito.simulation_core.**Simulator**(*settings*, *modules*)
This Class executes the time-step integration.

It forms the Core of the physical simulation and interacts with the GUI via the :py:class:‘‘SimulationInterface‘

Calculated values will be stored every 1 / measure rate seconds.

run()
Start the simulation.

stop()
Stop the simulation.

6.5 Processing GUI

class pymoskito.processing_gui.**PostProcessor**(*parent=None*)

closeEvent(*self*, *QCloseEvent*)

6.6 Processing Core

class pymoskito.processing_core.**PostProcessingModule**
Base Class for Postprocessing Modules

static calc_l1_norm_abs(*meas_values*, *desired_values*, *step_width*)
Calculate the L1-Norm of the absolute error.

Parameters

- **step_width** (*float*) – Time difference between measurements.
- **desired_values** (*array-like*) – Desired values.
- **meas_values** (*array-like*) – Measured values.

static calc_l1_norm_itae(*meas_values*, *desired_values*, *step_width*)
Calculate the L1-Norm of the ITAE (Integral of Time-multiplied Absolute value of Error).

Parameters

- **step_width** (*float*) – Time difference between measurements.
- **desired_values** (*array-like*) – Desired values.
- **meas_values** (*array-like*) – Measured values.

process(*files*)
worker-wrapper function that processes an array of result files This is an convenience wrapper for simple processor implementation. Overload for more sophisticated implementations :param files:

run(*data*)
Run this postprocessor.

This function will be called from *process()* with the simulation results from one simulation result file.

Overwrite this function to implement your own PostProcessor.

Args; param data: simulation results from a pymoskito simulation result file.

Returns With a figure Canvas an a name.

Return type Dict

class pymoskito.processing_core.**MetaProcessingModule**
Base Class for Meta-Processing Modules

plot_family(*family*, *x_path*, *y_path*, *mode*, *x_index=-1*, *y_index=-1*)
plots y over x for all members that can be found in family sources :param family: :param x_path: :param y_path: :param mode: :param x_index: :param y_index: :return:

set_plot_labeling(*title=''*, *grid=True*, *x_label=''*, *y_label=''*, *line_type='line'*)
helper to quickly set axis labeling with the good font sizes :param title: :param grid: :param x_label: :param y_label: :param line_type: :return:

6.7 Controltools

This file contains some functions that are quite helpful when designing feedback laws. This collection is not complete and does not aim to be so. For a more sophisticated collection have a look at the *symbtools* (https://github.com/TUD-RST/symbtools) or *control* package which are not used in this package to keep a small footprint.

pymoskito.controltools.**char_coefficients**(*poles*)
Calculate the coefficients of a characteristic polynomial.

Parameters **poles** (list or numpy.ndarray) – pol configuration

Returns coefficients

Return type numpy.ndarray

pymoskito.controltools.**place_siso**(*a_mat*, *b_mat*, *poles*)
Place poles for single input single output (SISO) systems:

- pol placement for state feedback: A and b

- pol placement for observer: A^T and c

Parameters

- **a_mat** (numpy.ndarray) – System matrix.:math:*A*
- **b_mat** (numpy.ndarray) – Input vector b or Output matrix c .
- **poles** (list or numpy.ndarray) – Desired poles.

Returns Feedback vector or k or observer gain l^T .

Return type numpy.ndarray

pymoskito.controltools.**calc_prefilter**(*a_mat*, *b_mat*, *c_mat*, *k_mat=None*)

Calculate the prefilter matrix

$$\boldsymbol{V} = -\left[\boldsymbol{C}\left(\boldsymbol{A}-\boldsymbol{B}\boldsymbol{K}\right)^{-1}\boldsymbol{B}\right]^{-1}$$

Parameters

- **a_mat** (numpy.ndarray) – system matrix
- **b_mat** (numpy.ndarray) – input matrix
- **c_mat** (numpy.ndarray) – output matrix
- **k_mat** (numpy.ndarray) – control matrix

Returns Prefilter matrix

Return type numpy.ndarray

pymoskito.controltools.**controllability_matrix**(*a_mat*, *b_mat*)

Calculate controllability matrix and check controllability of the system.

$$\boldsymbol{Q_c} = \begin{pmatrix}\boldsymbol{B} & \boldsymbol{AB} & \boldsymbol{A}^2\boldsymbol{B} & \cdots & \boldsymbol{A}^{n-1}\boldsymbol{B}\end{pmatrix}$$

Parameters

- **a_mat** (numpy.ndarray) – system matrix
- **b_mat** (numpy.ndarray) – manipulating matrix

Returns controllability matrix $\boldsymbol{Q_c}$

Return type numpy.ndarray

pymoskito.controltools.**observability_matrix**(*a_mat*, *c_mat*)

Calculate observability matrix and check observability of the system.

$$\boldsymbol{Q_o} = \begin{pmatrix}\boldsymbol{C} \\ \boldsymbol{CA} \\ \boldsymbol{CA}^2 \\ \vdots \\ \boldsymbol{CA}^{n-1}\end{pmatrix}$$

Parameters

- **a_mat** (numpy.ndarray) – system matrix
- **c_mat** (numpy.ndarray) – output matrix

Returns observability matrix $\boldsymbol{Q_o}$

Return type numpy.ndarray

pymoskito.controltools.**lie_derivatives**(*h*, *f*, *x*, *order=1*)

Calculates the Lie-Derivative from a scalar field $h(x)$ along a vector field $f(x)$.

Parameters

- **h** (*sympy.matrix*) – scalar field
- **f** (*sympy.matrix*) – vector field
- **x** (*sympy.matrix*) – symbolic representation of the states
- **order** (*int*) – order

Returns lie derivatives in ascending order

Return type list of sympy.matrix

6.8 Tools

Tools, functions and other funny things

pymoskito.tools.**rotation_matrix_xyz**(*axis*, *angle*, *angle_dim*)

Calculate the rotation matrix for a rotation around a given axis with the angle φ.

Parameters

- **axis** (*str*) – choose rotation axis “x”, “y” or “z”
- **angle** (*int or float*) – rotation angle φ
- **angle_dim** (*str*) – choose “deg” for degree or “rad” for radiant

Returns rotation matrix

Return type numpy.ndarray

pymoskito.tools.**get_resource**(*res_name*, *res_type='icons'*)

Build absolute path to specified resource within the package

Parameters

- **res_name** (*str*) – name of the resource
- **res_type** (*str*) – subdir

Returns path to resource

Return type str

6.9 Contributions to docs

All contributions are welcome. If you’d like to improve something, look into the sources if they contain the information you need (if not, please fix them), otherwise the documentation generation needs to be improved (look in the doc/ directory).

CHAPTER 7

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

7.1 Types of Contributions

7.1.1 Report Bugs

Report bugs at https://github.com/cklb/pymoskito/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

7.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

7.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.

7.1.4 Write Documentation

PyMoskito could always use more documentation, whether as part of the official PyMoskito docs, in docstrings, or even on the web in blog posts, articles, and such.

7.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/cklb/pymoskito/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.2 Get Started!

Ready to contribute? Here's how to set up pymoskito for local development.

1. Fork the pymoskito repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pymoskito.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv pymoskito
$ cd pymoskito/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ python setup.py test
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

7.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.4+ and for PyPy. Check https://travis-ci.org/cklb/pymoskito/pull_requests and make sure that the tests pass for all supported Python versions.

7.4 Tips

To run a subset of tests:

```
$ python -m unittest pymoskito/tests/test_gui.py
```

CHAPTER 8

Credits

8.1 Development Lead

- Stefan Ecklebe <stefan.ecklebe@tu-dresden.de>

8.2 Contributors

- Christoph Burggraf
- Marcus Riesmeier
- Jonas Hoffmann
- Jens Wurm

CHAPTER 9

History

9.1 0.3.0 (2018-10-01)

- Added a new plot system
- Added a last simulation list
- Added more log messages
- Removed latex as an requirement for the main GUI, only required for the Postprocessor

9.2 0.2.3 (2018-05-14)

- Added sensible examples for Post- and Meta processors in the Ball and Beam example
- Fixed Issue regarding the Disturbance Block
- Removed error-prone pseudo post processing
- Fixed problems due to changes in trajectory generators

9.3 0.2.2 (2018-03-28)

- Added extensive beginners guide (thanks to Jonas) and tutorial section
- Added extended documentation for examples (again, thanks to Jonas)

9.4 0.2.1 (2017-09-07)

- Fixed issue when installing via pip
- Fixed issue with metaprocessors and added example metaprocessor for ballbeam
- Downgraded requirements

9.5 0.2.0 (2017-08-18)

- Second minor release with lots of new features.
- Completely overhauled graphical user interface with menus and shortcuts.
- PyMoskito now comes with three full-fledged examples from the world of control theory, featuring the Ball and Beam- and a Tandem-Pendulum system.
- The main application now has a logger window which makes it easier to see what is going on in the simulation circuit.
- Several bugs concerning encoding issues have been fixed
- Unittest have been added and the development now uses travis-ci
- Version change from PyQt4 to Pyt5
- Version change form Python 2.7 to 3.5+
- Changed version to GPLv3 and added appropriate references for the used images.
- Improved the export of simulation results
- Introduced persistent settings that make opening files less painful.
- Made vtk an optional dependency and added matplotlib based visualizers.
- Large improvements concerning the sphinx-build documentation
- Fixed issue concerning possible data types for simulation module properties
- Introduced new generic modules that directly work on scipy StateSpace objects.

9.6 0.1.0 (2015-01-11)

- First release on PyPI.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Bibliography

[Hauser92] Hauser, J.; Sastry, S.; Kokotovic, P. Nonlinear Control Via Approximate Input-Output-Linearization: The Ball and Beam Example. IEEE Trans. on Automatic Control, 1992, vol 37, no. 3, pp. 392-398

Python Module Index

p

Index

Symbols

A

C

D

E

F

G

H

I

L

T

U